TEKCRUX

We ReDefine IT

kafka

Module 7
Kafka Stream Processing

Objectives

After completing of this module, you should be able to:

- ✓ Describe what is Stream Processing
- ✓ Learn different types of Programming Paradigms
- ✓ Describe Stream Processing Design Patterns
- ✓ Explain Kafka Streams & Kafka Streams API

TEKCRUX
We ReDefine IT

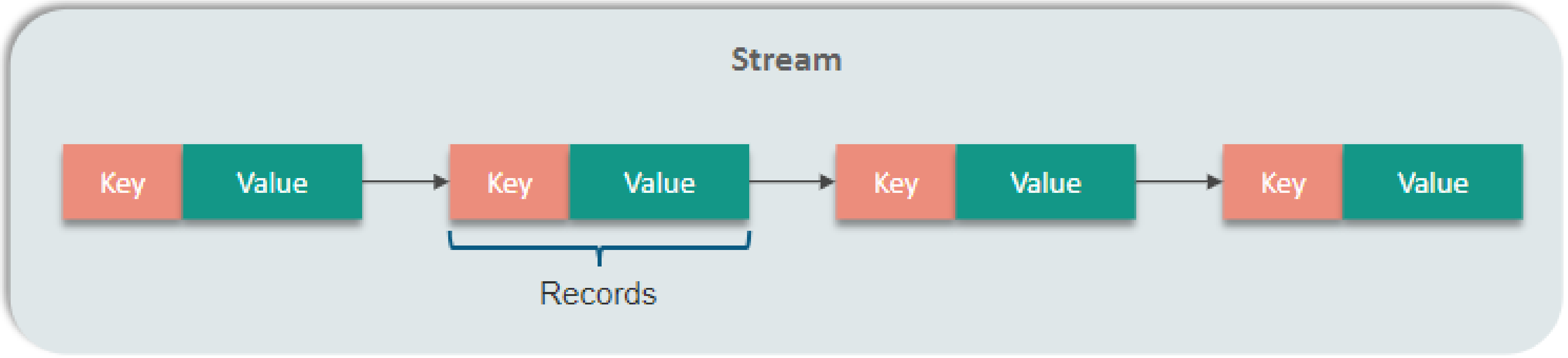
Let’s Understand Kafka Stream Processing

Data Stream / Event Stream

What is Data Stream / Event Stream

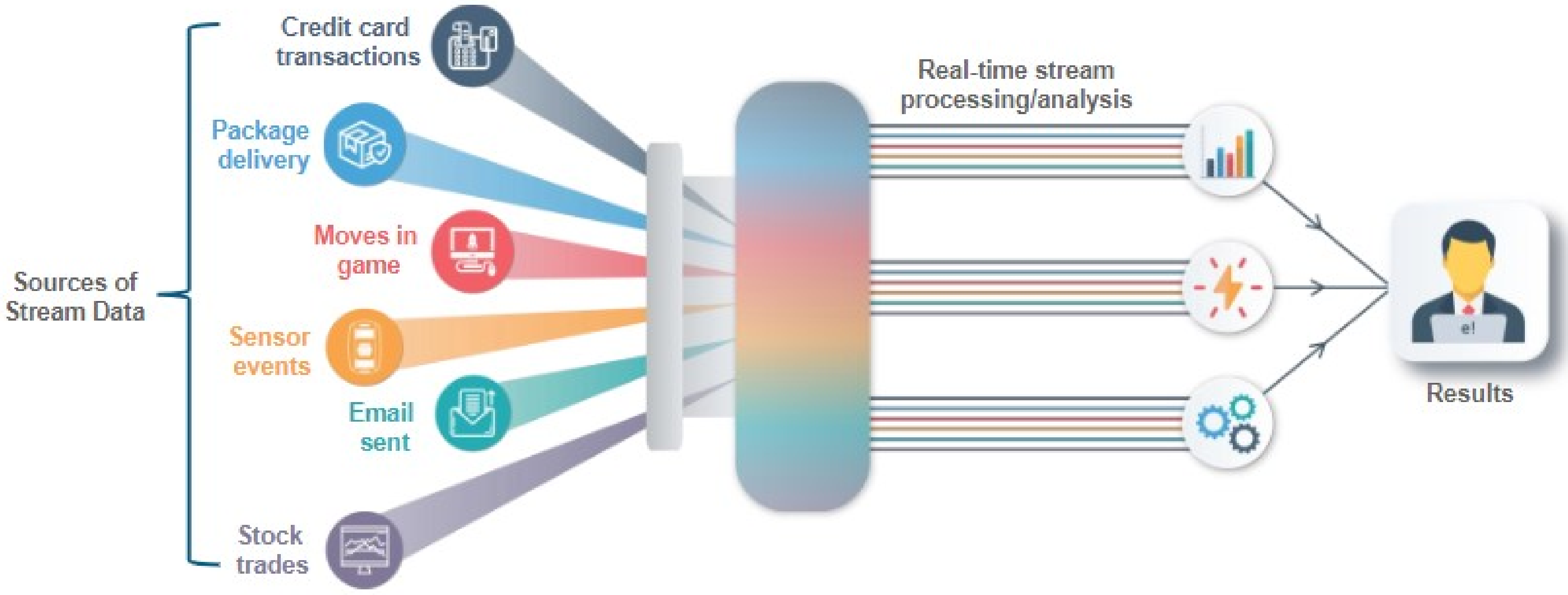
Data Stream is an unbounded(infinite and ever growing), continuous real-time flow of records(key-value pairs)

We don't need to explicitly request new records, we just receive them



What is Data Stream / Event Stream

Stream processing refers to the ongoing processing of one or more event streams as it is produced or received



Stream Processing

There are few other attributes of event streams model, in addition to their unbounded nature:

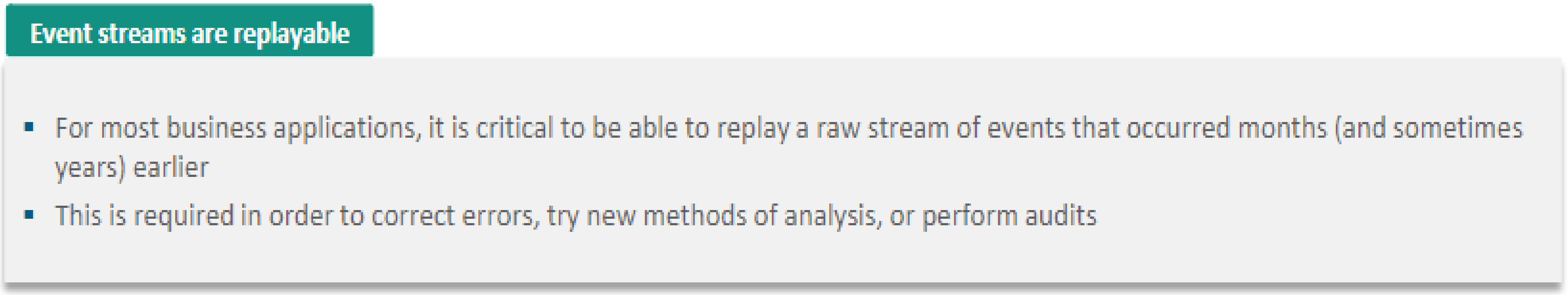
Event streams are ordered

- There is an inherent notion of which events occur before or after other events. This is clear when looking at financial events
- A sequence in which you first put money in my account and later spend the money is very different from a sequence at which you first spend the money and later cover my debt by depositing money back
- The latter will incur overdraft charges while the former will not

Immutable data records

- Events, once occurred, can never be modified. A financial transaction that is cancelled does not disappear.
- Instead, an additional event is written to the stream, recording a cancellation of previous transaction.

Stream Processing

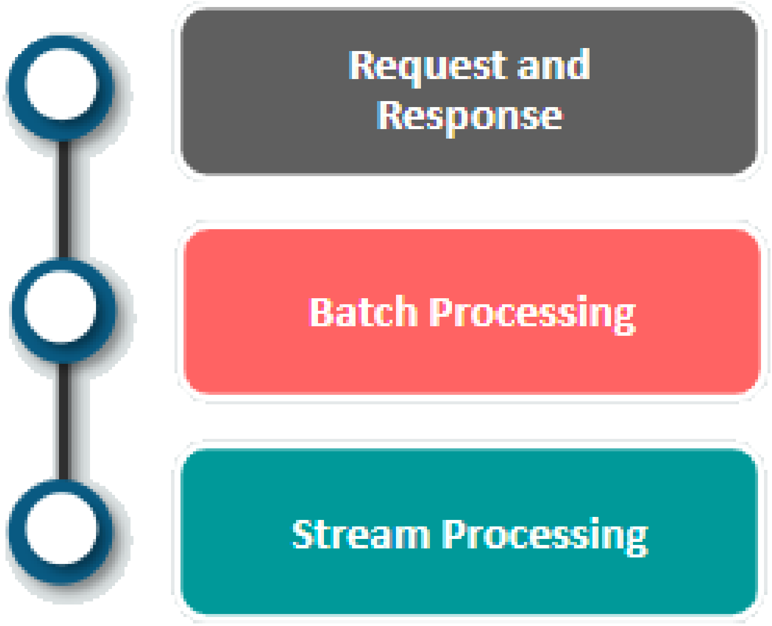


Event streams are replayable

- For most business applications, it is critical to be able to replay a raw stream of events that occurred months (and sometimes years) earlier
- This is required in order to correct errors, try new methods of analysis, or perform audits

Lets see how different Programming Paradigms compare for better understanding of how Stream Processing fits into Software Architectures

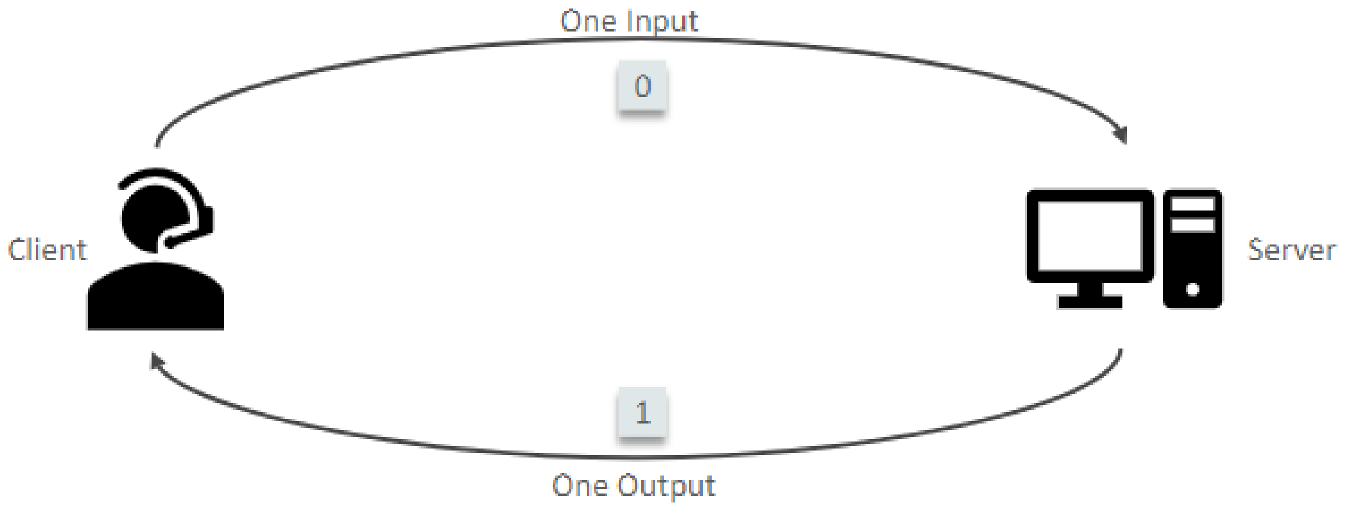
Different Programming Paradigms



Request and Response

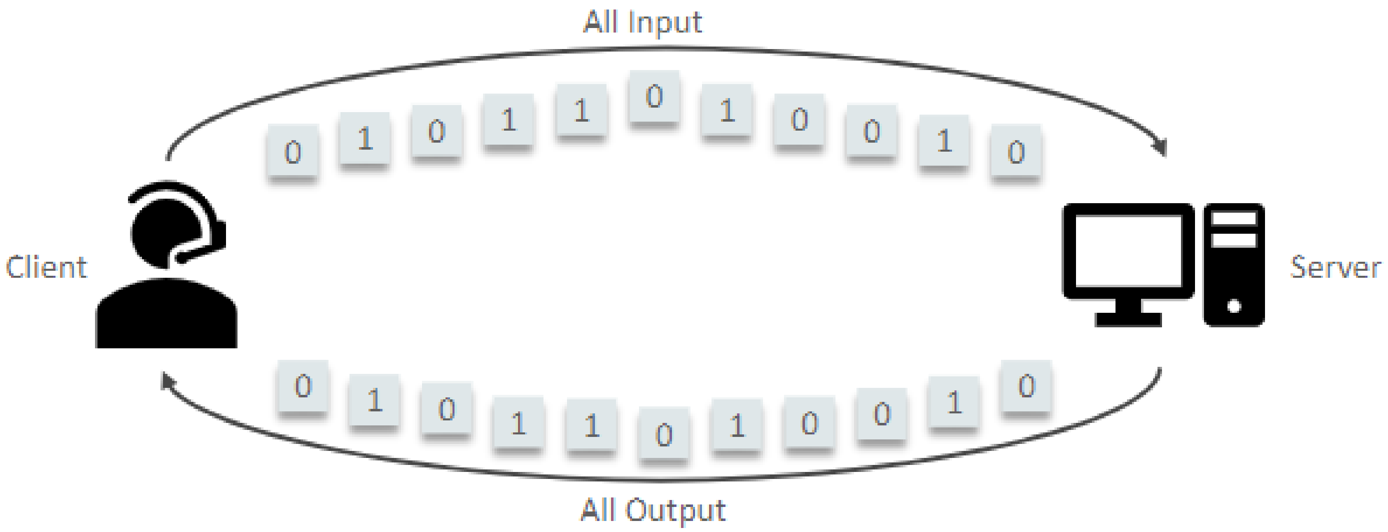
- This is the lowest latency paradigm, with response times ranging from submilliseconds to a few milliseconds
- This mode of processing is usually blocking—an app sends a request and waits for the processing system to respond

Examples:	Online transaction processing (OLTP), Point-of sale systems, credit card processing, and time-tracking systems



Batch Processing

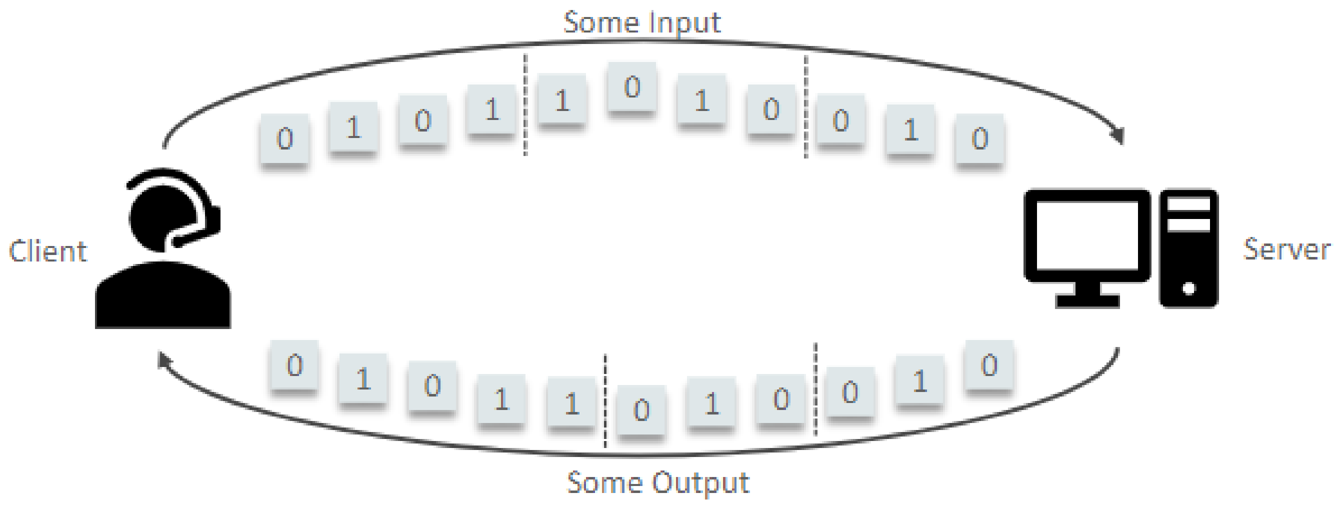
- This is the high-latency/high-throughput option and the processing system wakes up at set times
- It reads & writes all required input/output, & goes away until is scheduled to run
- In the database world, data is loaded in huge batches once a day, reports are generated, and users look at the same reports until the next data load occurs



Stream Processing

- This is the lowest latency paradigm, with response times ranging from submilliseconds to a few milliseconds
- Most business processes happen continuously, and as long as the business reports are updated continuously, the processing can proceed without anyone waiting for a specific response

Examples: Processes like alerting on suspicious credit transactions, adjusting prices in real-time based on supply and demand



Stream Processing Concepts

Stream processing is similar to any type of data processing—we write code that receives data, does something with the data—a few transformations, aggregates, enrichments, etc.—and then place the result somewhere

Some key concepts that are unique to stream processing are:

- Time
- State
- Stream-Table Duality
- Time Windows

Time

In the context of stream processing, having a common notion of time is critical because most stream applications perform operations on time windows

Stream-processing systems typically refer to the following notions of time:

Event time

This is the time the events we are tracking occurred and the record was created

01

Log append time

This is the time the event arrived to the Kafka broker and was stored there

02

Processing time

This is the time at which a stream-processing application received the event in order to perform some calculation

03

When working with time, it is important to be mindful of time zones. The entire data pipeline should standardize on a single time zones; otherwise, results of stream operations will be confusing and often meaningless.

State

Stream processing becomes really interesting when you have operations that involve multiple events: counting the number of events by type, moving averages, joining two streams to create an enriched stream of information, etc.

In such cases, we need to keep track of more information— how many events of each type we received this hour, all events that require joining, sums, averages, etc.

Information that is stored between events is called a state

It is often tempting to store the state in variables that are local to the stream processing app, such as a simple hash-table to store moving counts

☛ *However, this is not a reliable approach for managing state in stream processing because when the stream-processing application is stopped, the state is lost, which changes the results*

Local State

Local or internal state

- State that is accessible only by a specific instance of the stream-processing application
- This state is maintained and managed with an embedded, in-memory database running within the application

Advantage

The advantage of local state is that it is extremely fast

Disadvantage

Limited to the amount of memory available

☛ *As a result, many of the design patterns in stream processing focus on ways to partition the data into sub-streams that can be processed using a limited amount of local state*

External State

External state

- State that is maintained in an external datastore, often a NoSQL system like Cassandra

Advantage

Virtually unlimited size and it can be accessed from multiple instances of the application or even from different applications

Disadvantage

Extra latency and complexity introduced with an additional system

Most stream-processing apps try to avoid dealing with an external store, or at least limit the latency overhead by caching information in the local state
This usually introduces challenges with maintaining consistency between the internal and external state

Stream-Table Duality

This duality means that a stream can be viewed as a table, and vice versa

A simple form of a table is a collection of key-value pairs, also called a map or associative array

Table

key1	value1
key2	value2
key3	value3

Stream as Table

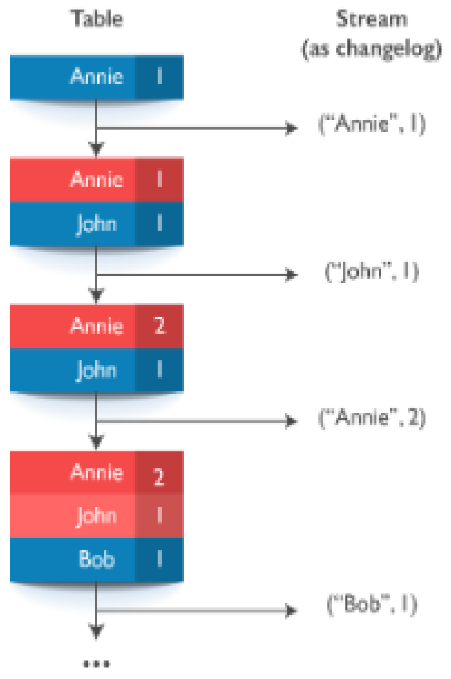
- A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table
- A stream is thus a table in disguise, and it can be easily turned into a table by replaying the changelog from beginning to end

Table as Stream

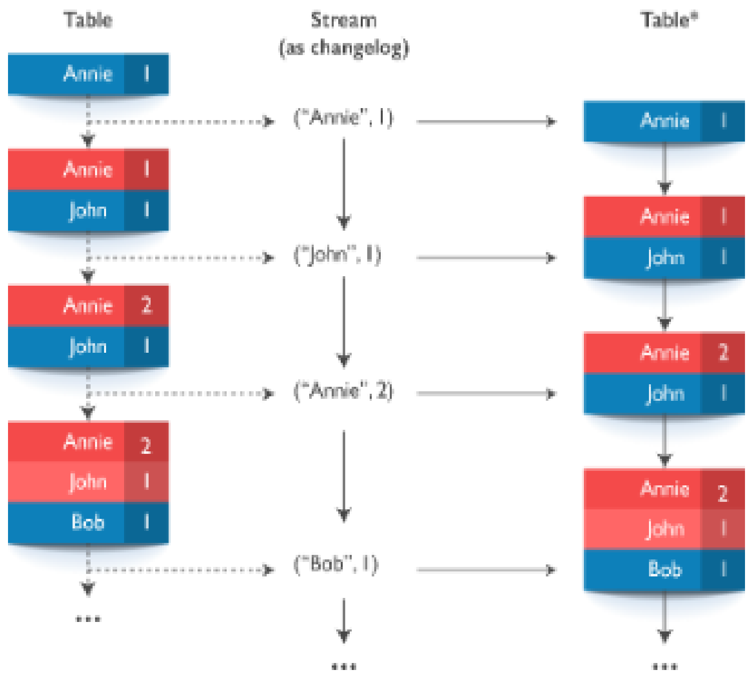
- A table is a snapshot, of the latest value for each key in a stream (a stream's data records are key-value pairs)
- A table is thus a stream in disguise, and it can be easily turned into a stream by iterating over each key-value entry in the table

Stream-Table Duality : Example

- Here a table tracks the total number of pageviews by user (first column of diagram below)
- The state changes between different points in time & different revisions of the table can be represented as a changelog stream



Interestingly, because of the stream-table duality, the same stream can be used to reconstruct the original table:



Time Windows

A stream processor may need to divide data records into time buckets, i.e. to **window** the stream by time. This is usually needed for join and aggregation operations, etc.

Kafka Streams currently defines the following types of windows:

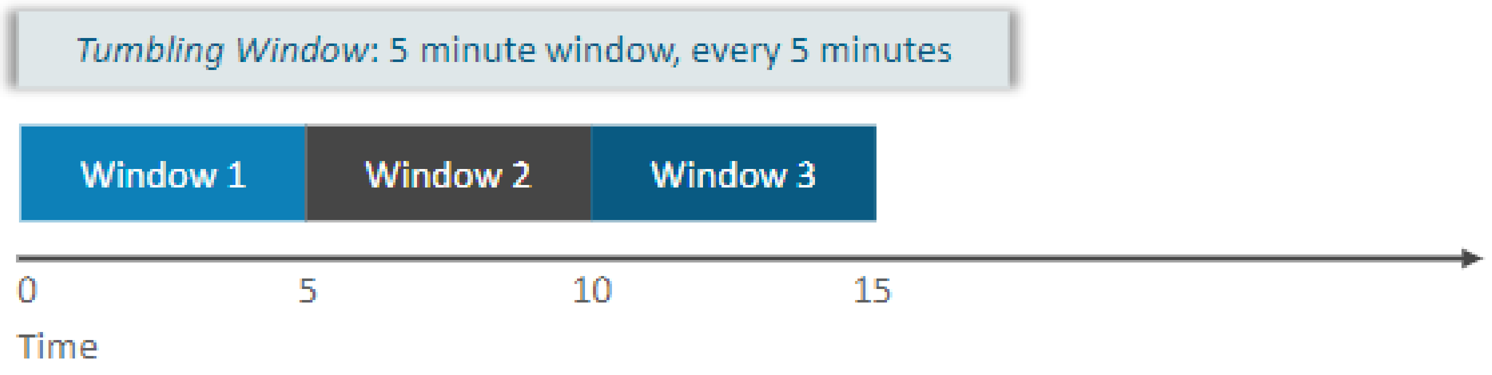
Hopping time windows

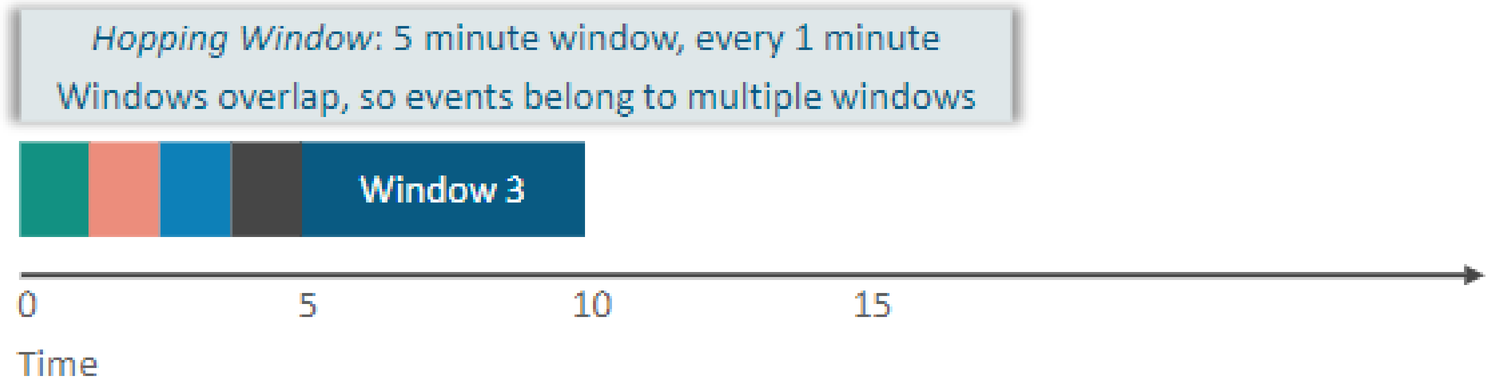
- Windows based on time intervals
- They model fixed-sized, (possibly) overlapping windows
- Defined by two properties: the window's size and its advance interval (aka "hop")
- Advance interval specifies by how much a window moves forward relative to the previous one
- Hopping windows can overlap a data record may belong to more than one such windows

Tumbling time windows

- A special case of hopping time windows
- They model fixed-size, non-overlapping, gap-less windows
- Defined by a single property: the window's size
- A tumbling window is a hopping window whose window size is equal to its advance interval
- Tumbling windows never overlap, a data record will belong to one and only one window

Tumbling Windows Vs Hopping Windows



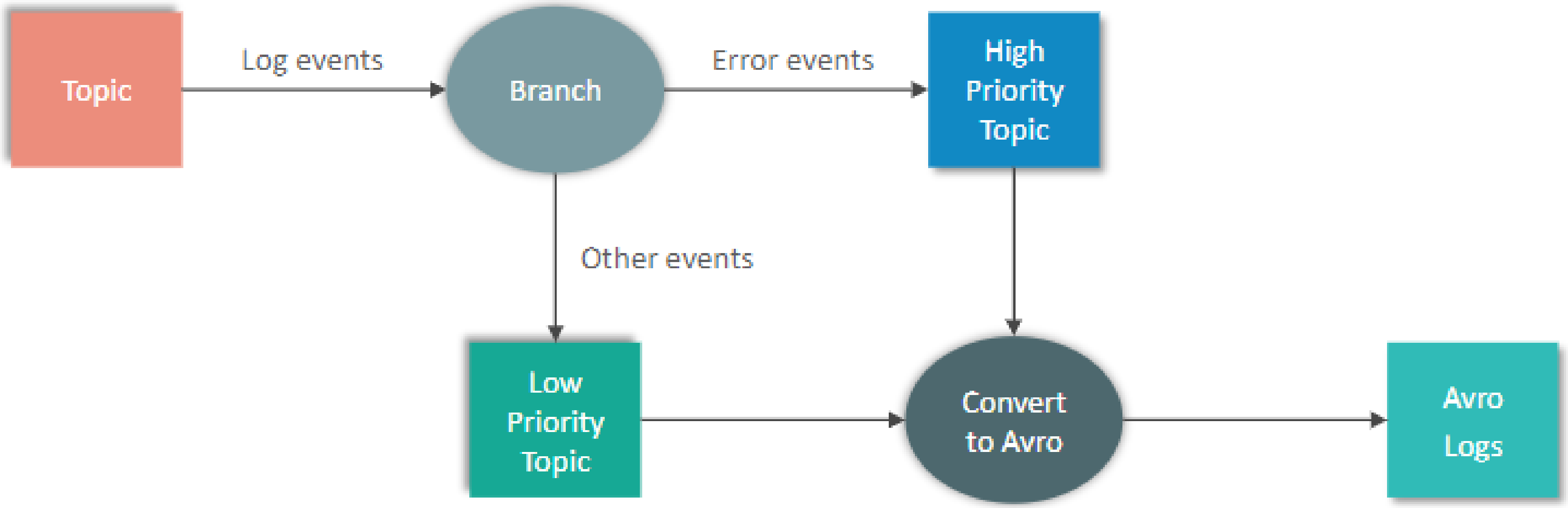

Hopping Window: 5 minute window, every 1 minute
Windows overlap, so events belong to multiple windows
Window 3
0
5
10
15
Time

Stream Processing Design Patterns

- Single Event Processing
- Processing with Local State
- Multiphase Processing/Repartitioning
- Processing with External Lookup: Stream-Table Join
- Streaming Join
- Out-of-Sequence Events

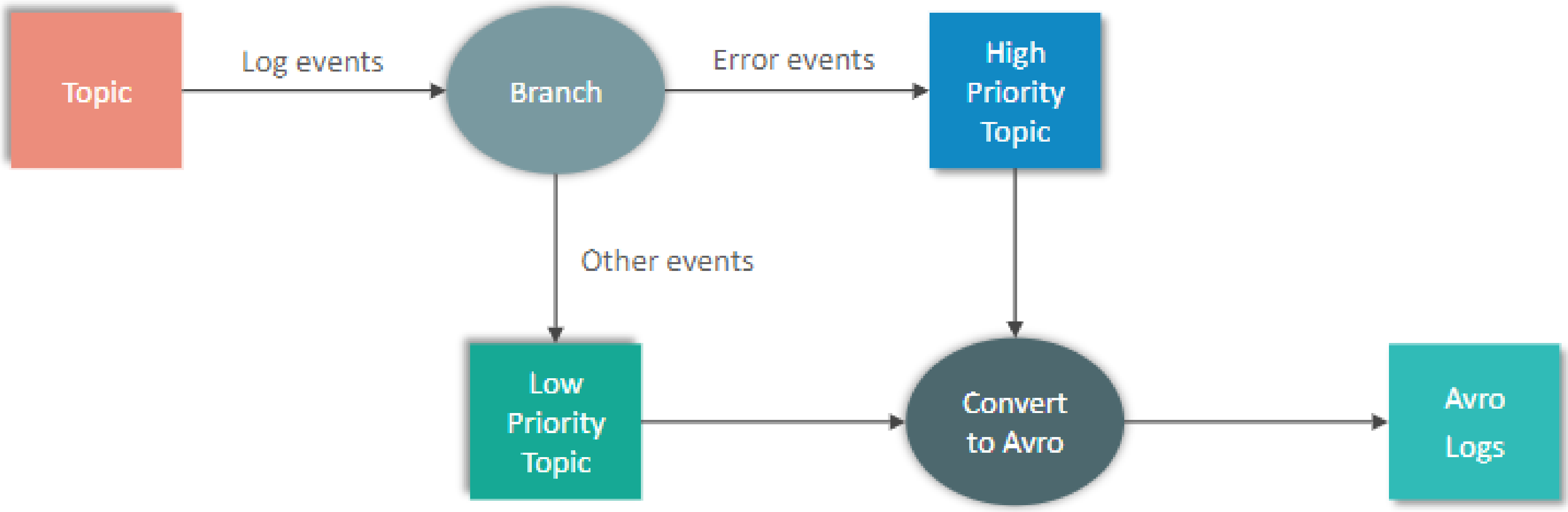
Single Event Processing

- The most basic pattern of stream processing is the processing of each event in isolation
- This is also known as a map/filter pattern because it is commonly used to filter unnecessary events from the stream
- Stream-processing app consumes events from the stream, modifies them, & then produces the events to another stream



Single Event Processing : Example

Example: An app that reads log messages from a stream and writes ERROR events into a high-priority stream, and the rest of the events into a low-priority stream



Processing with Local State

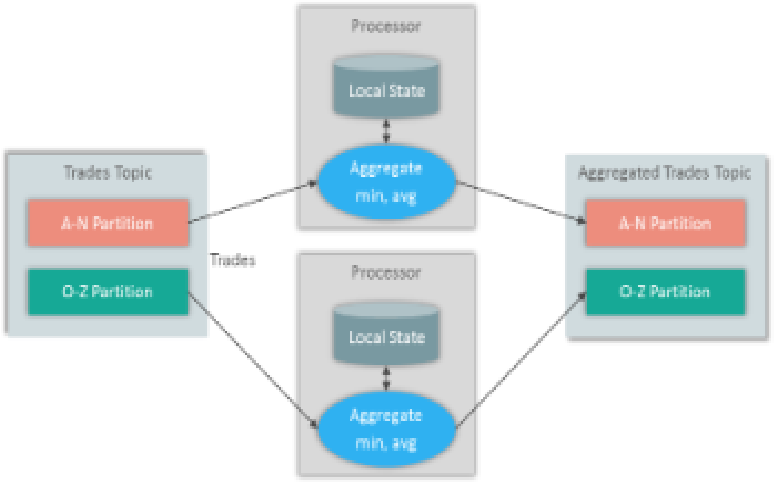
Most stream-processing applications are concerned with aggregating information, especially time-window aggregation

Example: Finding the minimum and maximum stock prices for each day of trading and calculating a moving average. These aggregations require maintaining a state for the stream

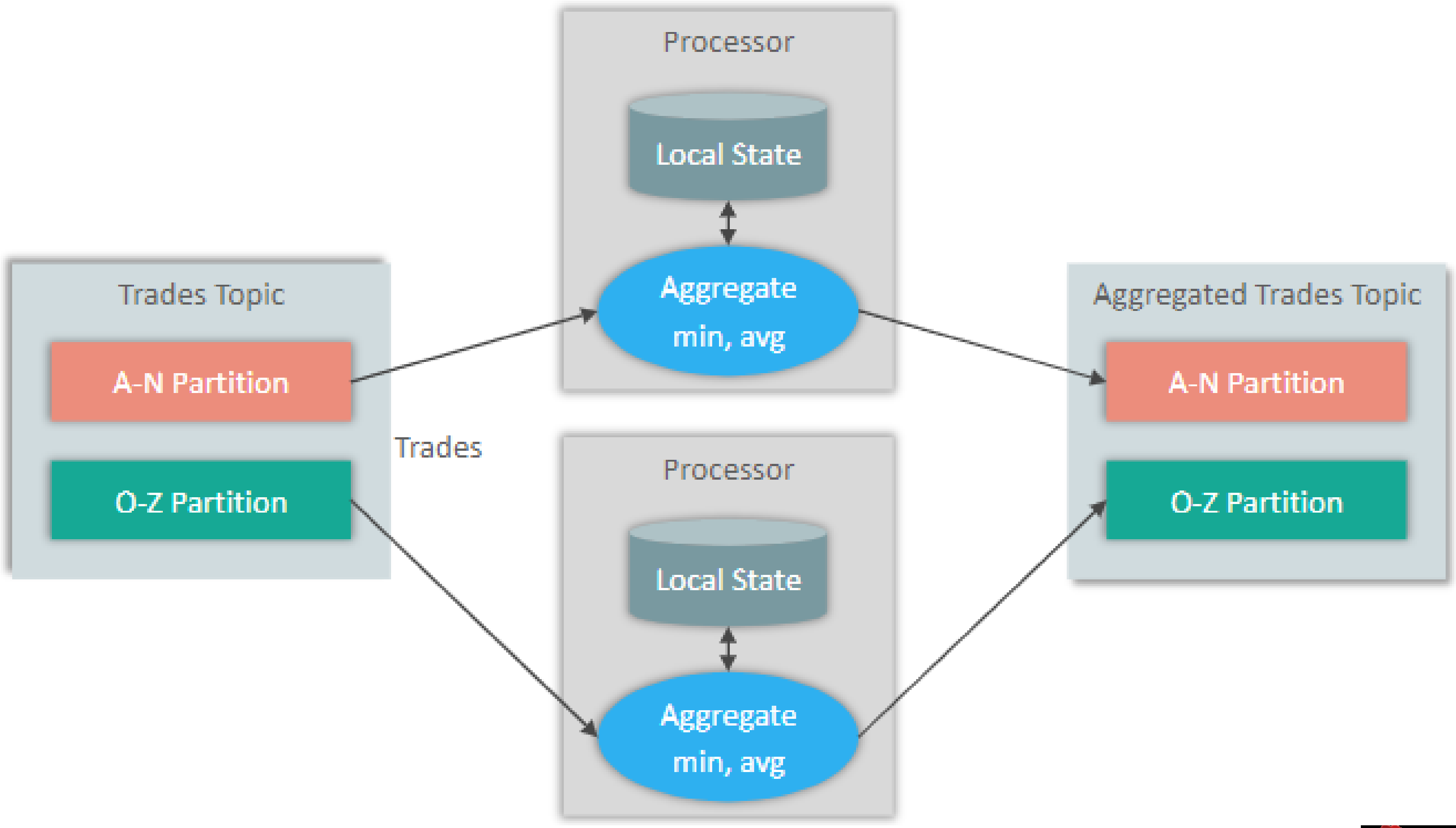
Example

In order to calculate the min & average price each day, we need to store the min & max values until the current time. Then compare each new value in the stream to the stored min & max

All these can be done using local state because each operation in our example is a group by aggregate i.e. we perform the aggregation per stock symbol, not on the entire stock market in general.



Stream Processing with Local State : Example



Multiphase Processing / Repartitioning

Local state is great if we need a group by type of aggregate

But what if we need a result that uses all available information?

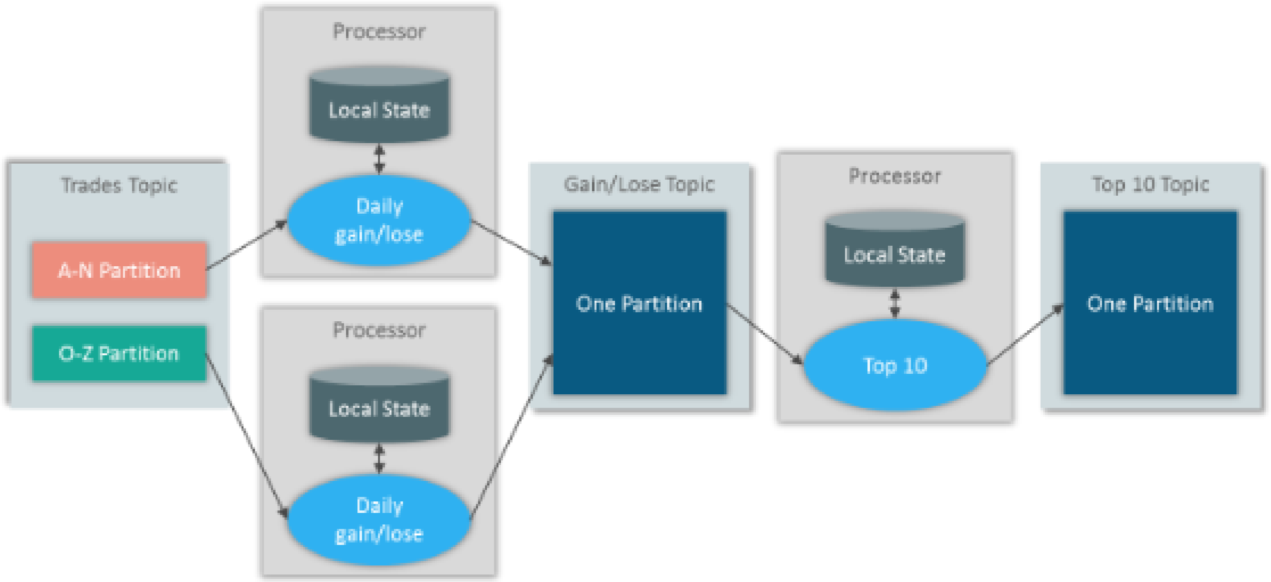
Example

Suppose we want to publish the top 10 stocks each day

We cannot do it locally on each application instance because all the top 10 stocks could be in partitions assigned to other instances

Multiphase Processing / Repartitioning

We need a two-phase approach:



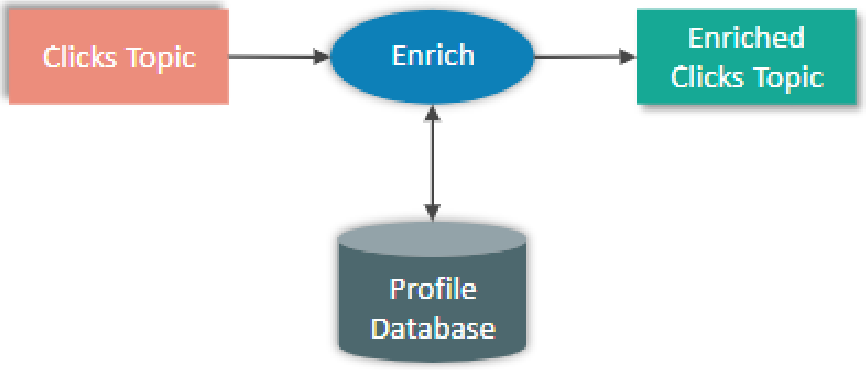
- First, we calculate the daily gain/loss for each stock symbol
- Then we write the results to a new topic with a single partition

This partition will be read by a single application instance that can then find the top 10 stocks for the day

Processing with External Lookup: Stream-Table Join

Sometimes stream processing requires integration with data external to the stream— validating transactions against a set of rules stored in a database, or enriching clickstream information with data about the users who clicked

The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click plus the user age and gender to another topic



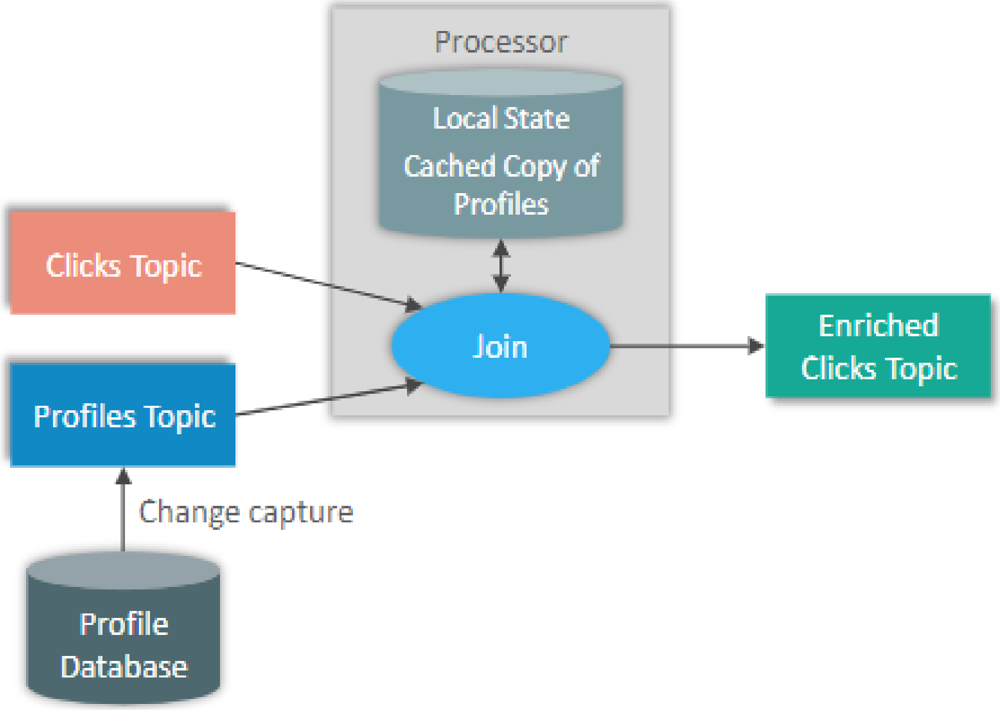
- Here an external lookup adds significant latency to the processing of every record—usually between 5-15 milliseconds.
- In many cases, this is not feasible
- We want a solution that scales better in order to get good performance

Stream Processing Design Patterns

When we get click events, we can look up the user_id at our local cache and enrich the event

And because we are using a local cache, this scales a lot better and will not affect the database and other apps using it

We refer to this as a stream-table join because one of the streams represents changes to a locally cached table



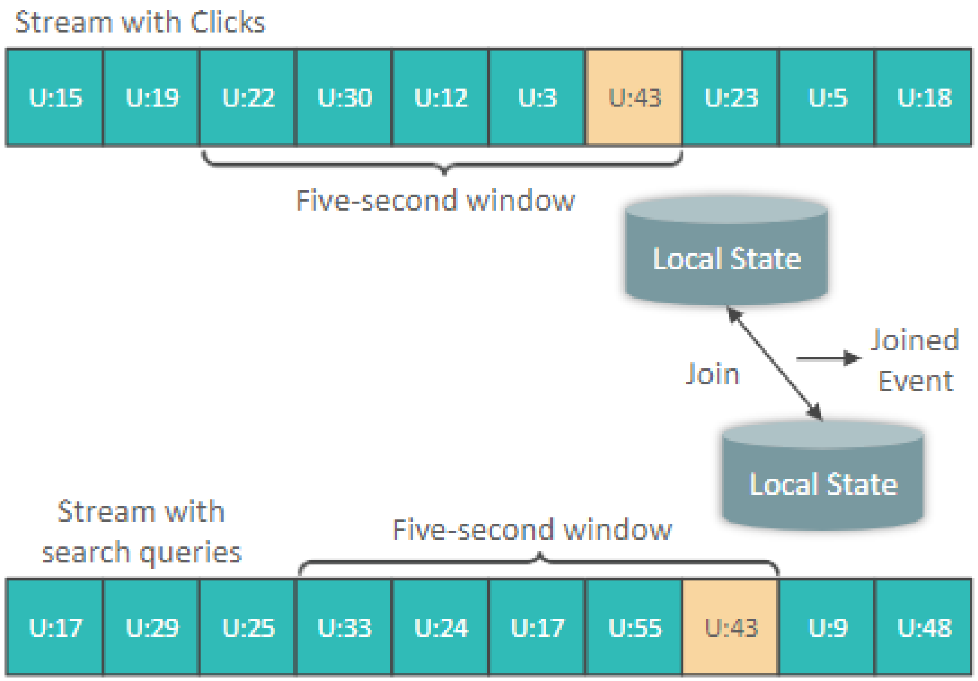
Streaming Join

When we join two streams, we are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time-windows. This is why a streaming-join is also called a windowed-join

Example

Here we are matching search queries with the results they clicked on so that we will know which result is most popular for which query

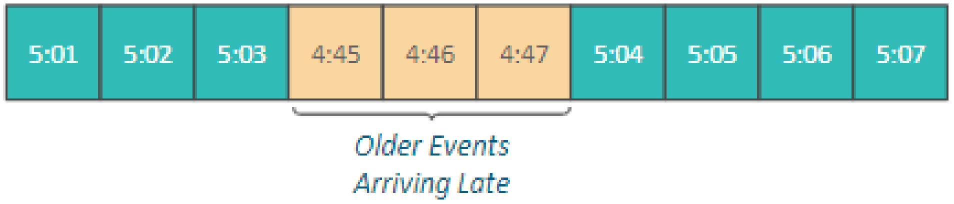
We assume the result is clicked seconds after the query was entered into our search engine. So we keep a few-seconds-long window on each stream and match the results from each window



Out-of-Sequence Events

Out-of-sequence events happen quite frequently and expectedly in IoT (Internet of Things) scenarios

Example: A mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects



This typically means the application has to do the following:

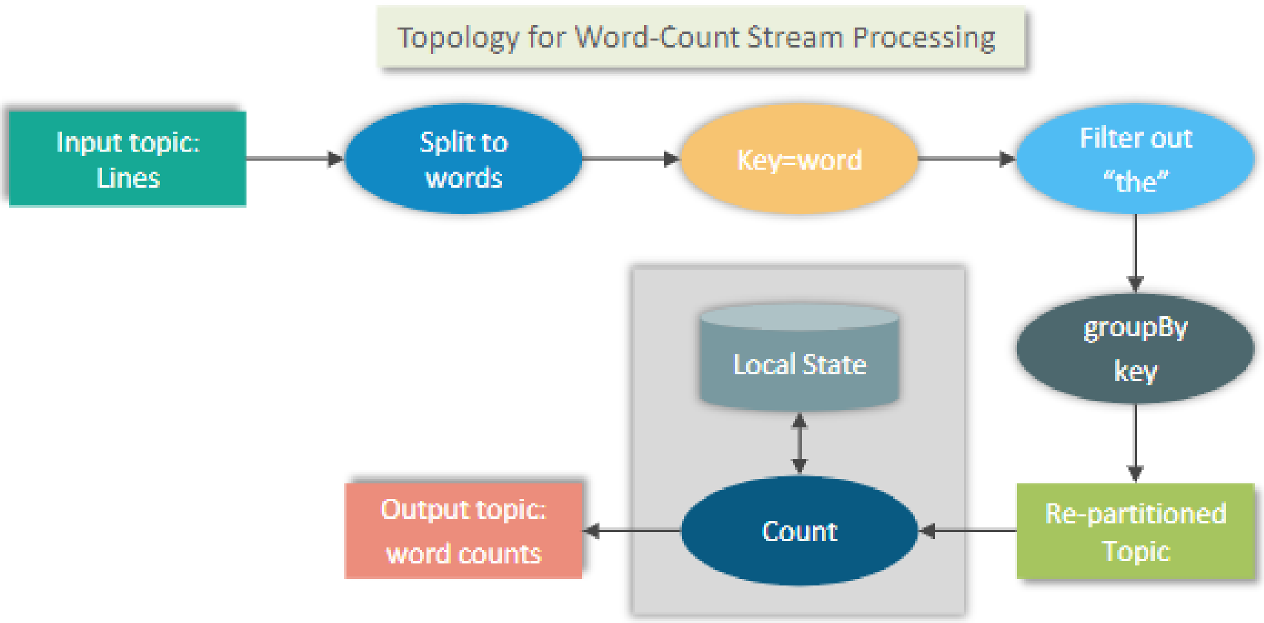
- Recognize that an event is out of sequence
- Define a time period during which it will attempt to reconcile out-of-sequence events
- Have an in-band capability to reconcile this event
- Be able to update results

Kafka Streams - Architecture

Building a Topology

Every streams application implements and executes at least one topology
Topology (DAG) is a set of operations and transitions that every event moves through from input to output

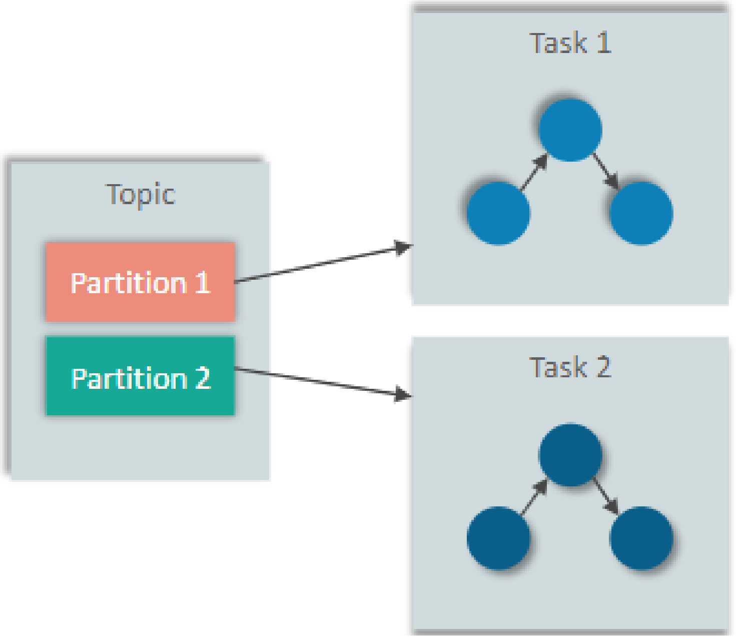
- The topology is made up of processors— those are the nodes in the topology graph (represented by circles in the diagram)
- Most processors implement an operation of the data—filter, map, aggregate, etc.
- A topology always starts with one or more source processors and finishes with one or more sink processors



Scaling the Topology

Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application

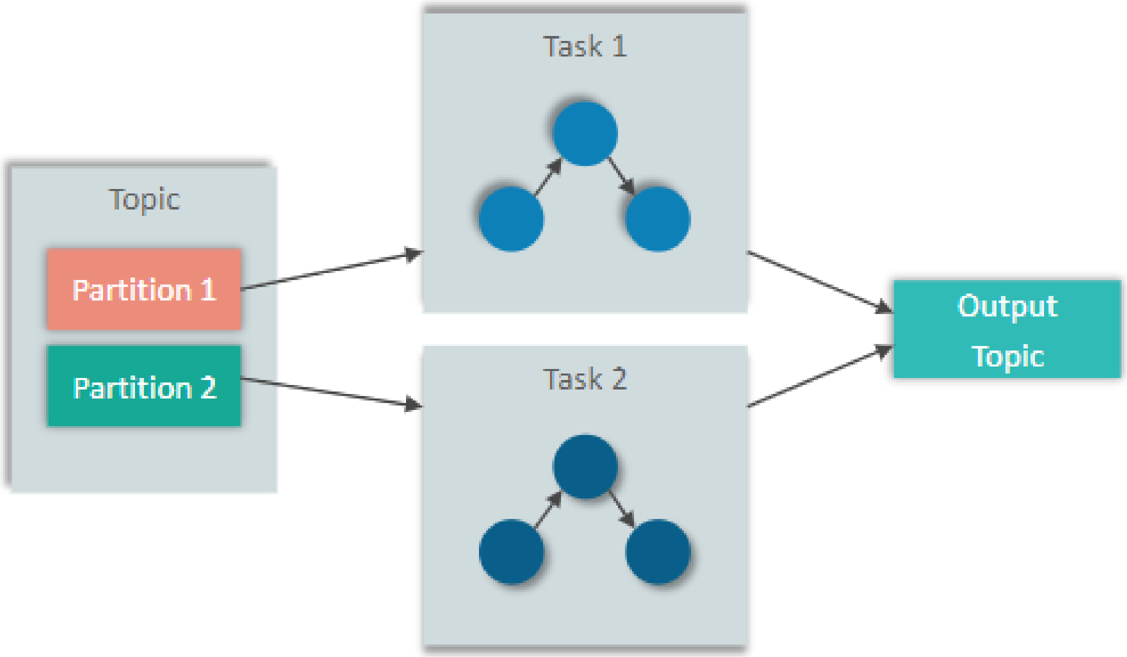
- We can run the Streams application on one machine with multiple threads/machines
- The Streams engine parallelizes execution of a topology by splitting it into tasks



Scaling the Topology

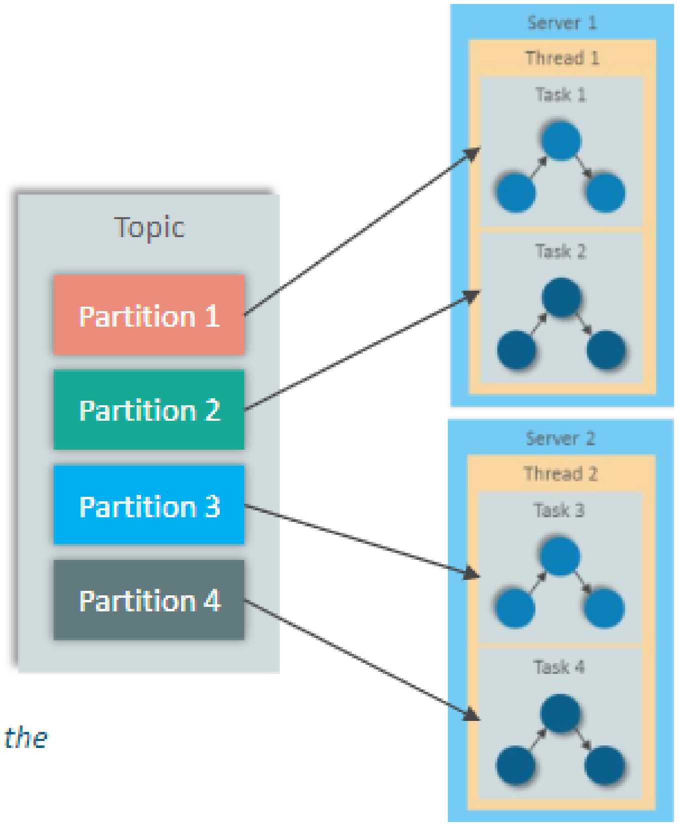
Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application

- The number of tasks is determined by the Streams engine & depends on the number of partitions in the topics
- Each task is responsible for a subset of the partitions
- Tasks are the basic unit of parallelism in Kafka Streams, as each task can execute independently



Scaling the Topology

- The developer of the application can choose the number of threads each application instance will execute
- If multiple threads are available, every thread will execute a subset of the tasks that the application creates
- If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server
- This is the way streaming applications scale: we will have as many tasks as you have partitions in the topics we are processing



If we want to process faster, add more threads. If we run out of resources on the server, start another instance of the application on another server

Scaling the Topology

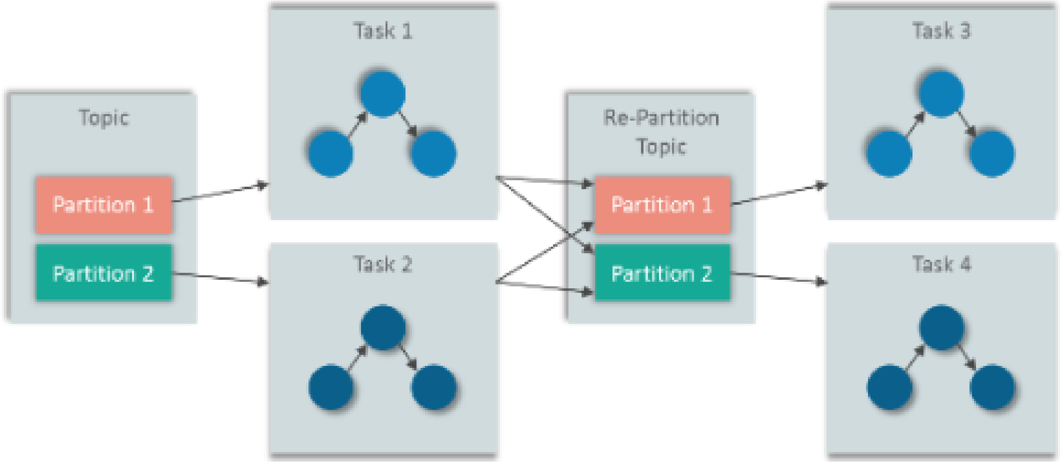
Sometimes a processing step may require results from multiple partitions, which could create dependencies between tasks

Kafka Streams repartitions by writing the events to a new topic with new keys & partitions

Then another set of tasks reads events from the new topic and continues processing.

The repartitioning steps break our topology into two subtopologies, each with its own tasks

The second set of tasks depends on the first, because it processes the results of the first subtopology



Let’s Learn about Kafka Streams API

Stream API Use Cases

Kafka Streams powers parts of our analytics pipeline and delivers endless options to explore and operate on the data sources we have at hand

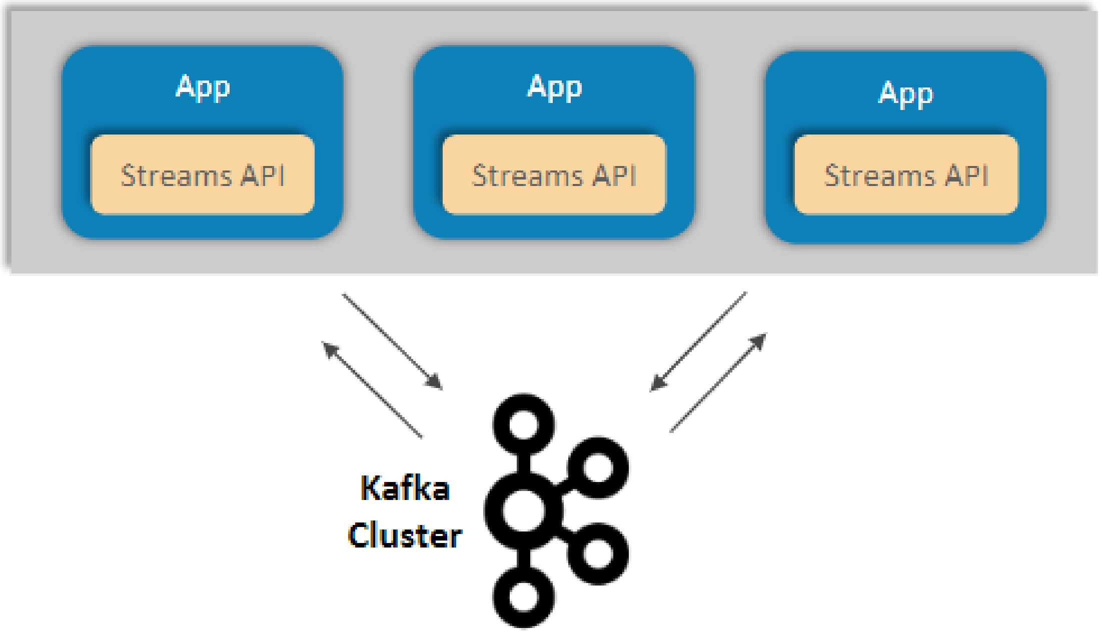
Rabobank is one of the 3 largest banks in the Netherlands. It is used by an increasing amount of financial processes and services, one of which is Rabo Alerts and is built using Kafka Streams

Pinterest uses Kafka Streams API at large scale to power the real-time, predictive budgeting system of their advertising infrastructure

The New York Times uses Kafka Streams API to store and distribute, in real-time, published content to the various applications and systems that make it available to the readers

Using the Kafka Streams API

- Call the Kafka Streams from Java or Scala applications
- The Kafka Streams API interacts with a Kafka Cluster
- The application does not run directly on Kafka brokers



Kafka Stream Features

Highly scalable, elastic, distributed, and fault-tolerant application.

Stateful and stateless processing.

Event-time processing with windowing, joins, and aggregations.

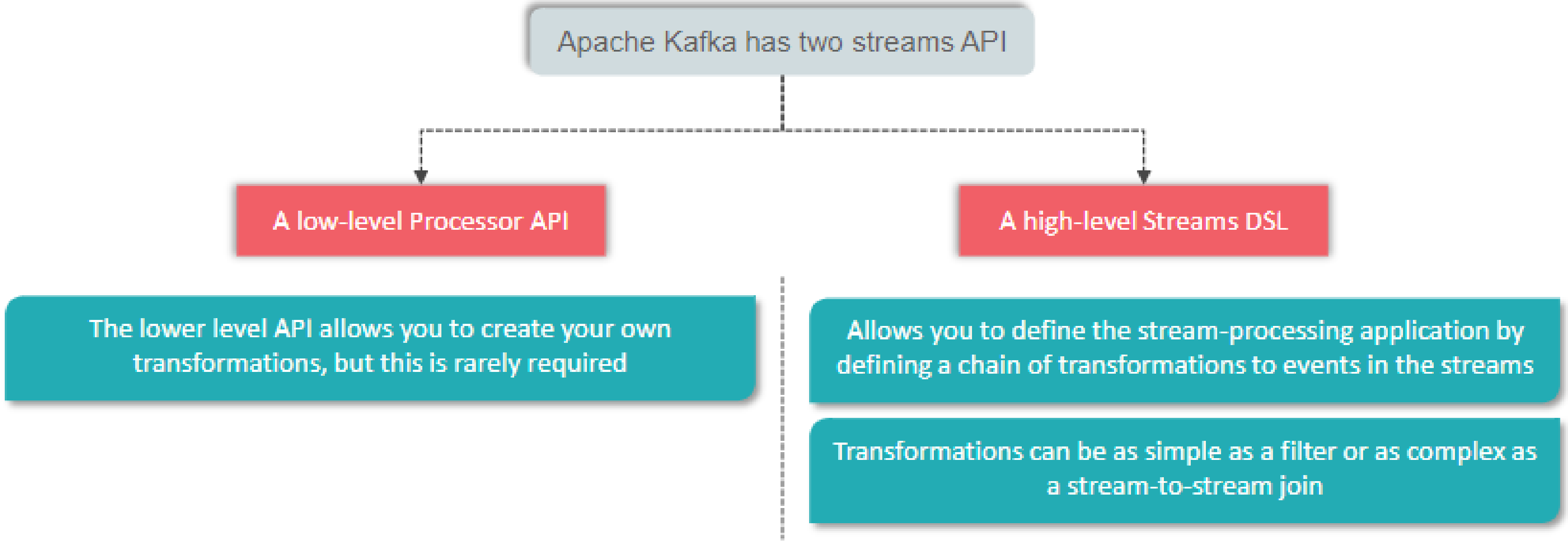
Low barrier to entry, which means it does not take much configuration and setup to run a small scale trial of stream processing; the rest depends on your use case.

No separate cluster requirements for processing (integrated with Kafka).

Employs **one-record-at-a-time processing** to achieve millisecond processing latency, and supports **event-time based windowing operations** with the late arrival of records

Supports Kafka Connect to connect to different applications and databases

Kafka Streams API



Apache Kafka has two streams API

A low-level Processor API

The lower level API allows you to create your own transformations, but this is rarely required

A high-level Streams DSL

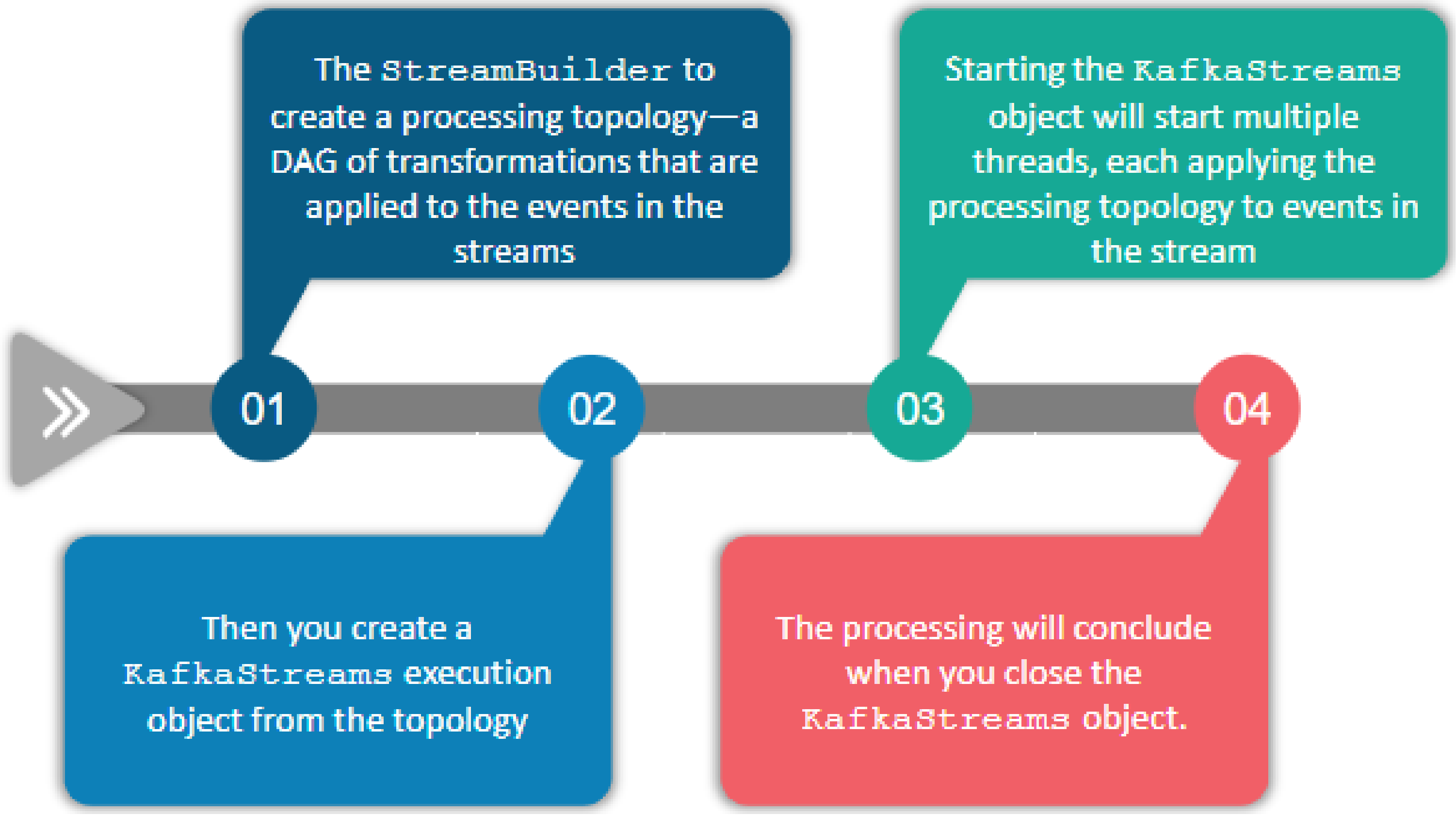
Allows you to define the stream-processing application by defining a chain of transformations to events in the streams

Transformations can be as simple as a filter or as complex as a stream-to-stream join

☛ *We will use Kafka Streams DSL in our examples*

Kafka Streams DSL API

An application that uses the DSL API always starts with:



01 The `StreamBuilder` to create a processing topology—a DAG of transformations that are applied to the events in the streams

02 Then you create a `KafkaStreams` execution object from the topology

03 Starting the `KafkaStreams` object will start multiple threads, each applying the processing topology to events in the stream

04 The processing will conclude when you close the `KafkaStreams` object.

Kafka Streams : Word Count Example

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

Properties props = new Properties();

props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
```

Application ID is used to coordinate the instances of the application
This name must be unique for each Kafka Streams application working with the same Kafka cluster

Kafka Streams : Word Count Example

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

Properties props = new Properties();

props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
```

The Kafka Streams application always reads data from Kafka topics and writes its output to Kafka topics
Here we are telling our app where to find Kafka

Kafka Streams : Word Count Example

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

Properties props = new Properties();

props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
```

When reading and writing data, our app will need to serialize and deserialize, so we provide default Serde classes
If needed, we can override these defaults

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

We create a KStreamBuilder object and start defining a stream by pointing at the topic, which we'll use as our input

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

Each event we read from the source topic is a line of words; we split it up using a regular expression into a series of individual words. Then we take each word and put it in the event record key so it can be used in a group-by operation

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

We filter out the word "the," just to show how easy filtering is

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

And we group by key, so we now have a collection of events for each unique word

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

We count how many events we have in each collection. The result of counting is a Long data type. We convert it to a String so it will be easier for humans to read the results

Kafka Streams : Word Count Example

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream counts = source.flatMapValues(value->
Arrays.asList(pattern.split(value.toLowerCase())))
.map((key, value) -> new KeyValue<Object,Object>(value, value))
.filter((key, value) -> (!value.equals("the")))
.groupByKey()
.count("CountStore").mapValues(value->
Long.toString(value)).toStream();

counts.to("wordcount-output");
```

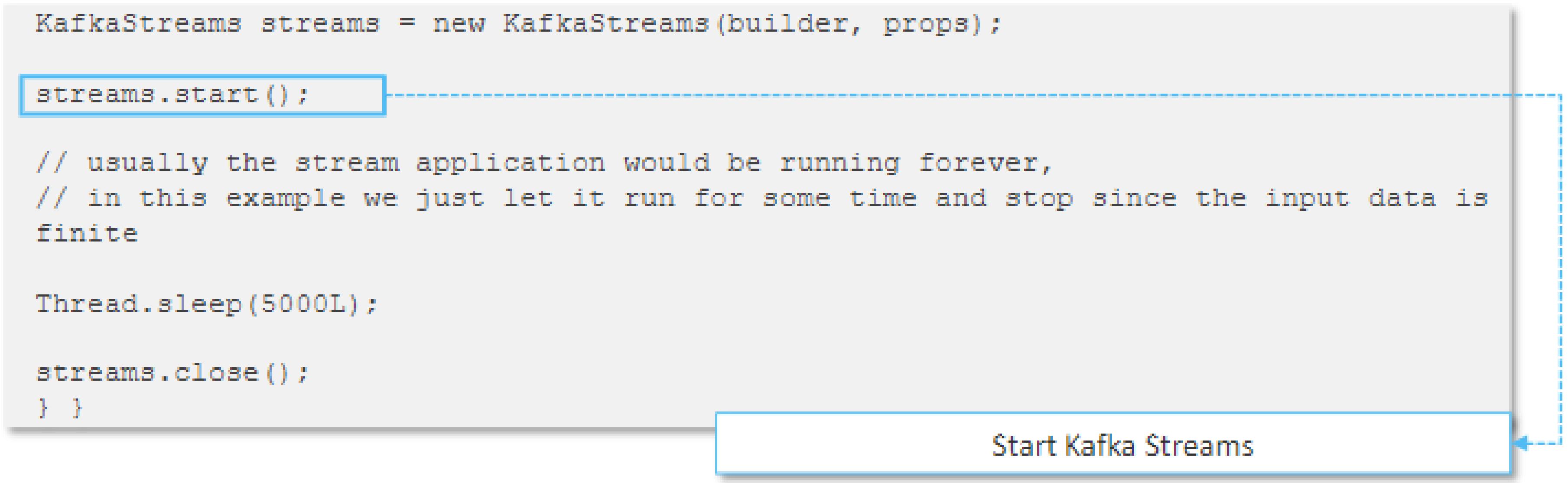
Only one thing left—write the results back to Kafka

Kafka Streams : Word Count Example

```
KafkaStreams streams = new KafkaStreams(builder, props);

streams.start();

// usually the stream application would be running forever,
// in this example we just let it run for some time and stop since the input data is
finite

Thread.sleep(5000L);

streams.close();
} }
```

Define a KafkaStreams object based on our topology and the properties we defined

Kafka Streams : Word Count Example



```
KafkaStreams streams = new KafkaStreams(builder, props);

streams.start();

// usually the stream application would be running forever,
// in this example we just let it run for some time and stop since the input data is
finite

Thread.sleep(5000L);

streams.close();
} }
```

Kafka Streams : Word Count Example

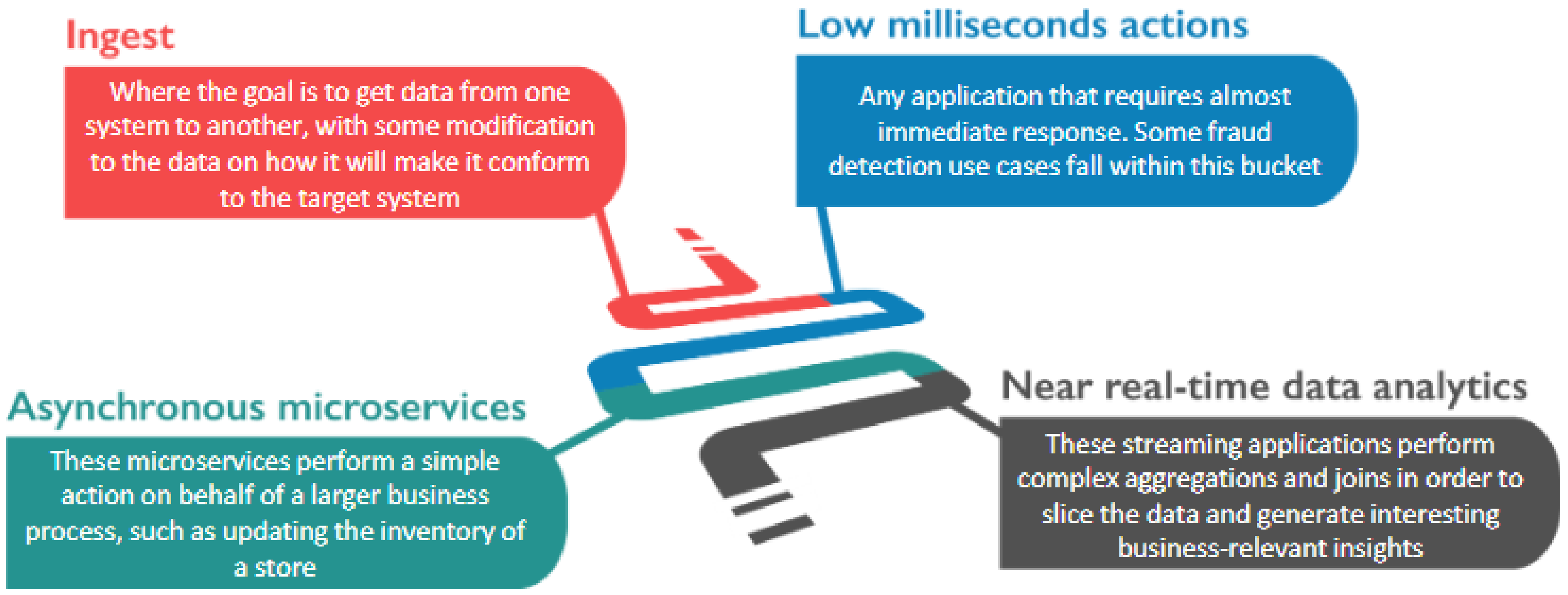
```
KafkaStreams streams = new KafkaStreams(builder, props);

streams.start();

// usually the stream application would be running forever,
// in this example we just let it run for some time and stop since the input data is
finite

Thread.sleep(5000L);

streams.close();
} }
```

After a while, stop it

How to choose a Stream Processing Framework?

When choosing a stream-processing framework, it is important to consider the type of application you are planning on writing

Different types of applications call for different stream-processing solutions:



Ingest

Where the goal is to get data from one system to another, with some modification to the data on how it will make it conform to the target system

Low milliseconds actions

Any application that requires almost immediate response. Some fraud detection use cases fall within this bucket

Asynchronous microservices

These microservices perform a simple action on behalf of a larger business process, such as updating the inventory of a store

Near real-time data analytics

These streaming applications perform complex aggregations and joins in order to slice the data and generate interesting business-relevant insights

TEKCRUX
We ReDefine IT