Stream Processing using Apache Kafka® Streams

Version 7.0.0-v1.0.3

CONFLUENT

Table of Contents

Introduction

CONFLUENT

Global Education

Class Logistics and Overview

Copyright & Trademarks

Prerequisite

This course requires a working knowledge of the Apache Kafka architecture.

New to Kafka? Need a refresher?

Sign up for free ***Confluent Fundamentals for Apache Kafka*** course at https://confluent.io/training

Agenda

1. **Starting with Stream Processing**
   a. Bridging from Fundamentals and core Apache Kafka
   b. Kafka Streams concepts
2. **Stateful Processing and Advanced Operations**
   a. Time-based processing
   b. Stateful processing
   c. Custom processing
3. **Safely Deploying and Operating Stream Processing**
   a. Testing, Troubleshooting, Monitoring
   b. Deployment
   c. Security

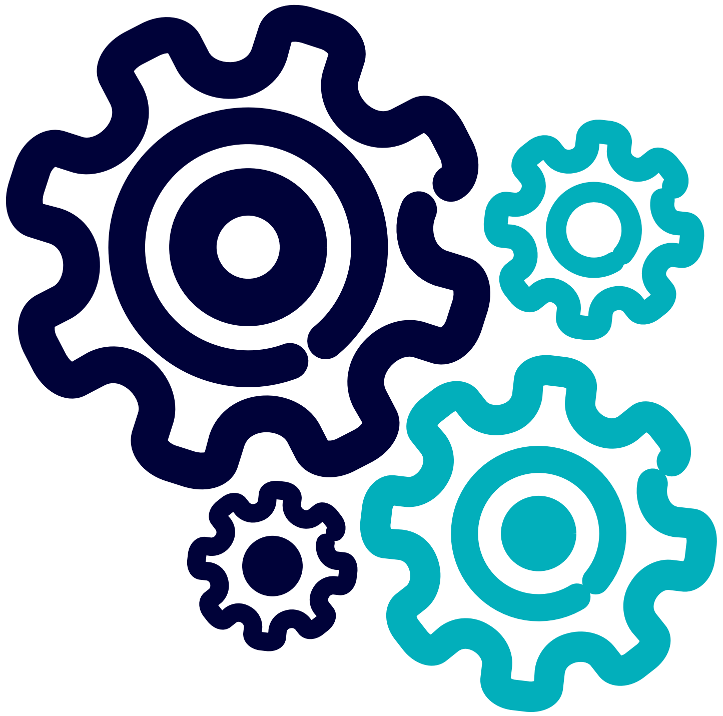
Course Objectives

Upon completion of this course, you should be able to:

- Identify common patterns and use cases for real-time stream processing
- Describe the high-level architecture of Apache Kafka Streams
- Write real-time applications with the Kafka Streams API to filter, transform, enrich, aggregate, and join data streams
- Describe how Kafka Streams provide the elastic, fault-tolerant, high-performance stream processing capabilities
- Test, secure, deploy, and monitor Kafka Streams applications

Throughout the course, hands-on exercises and discussion activities will reinforce the topics being discussed.

Class Logistics



- Timing
  - Start and end times
  - Can I come in early/stay late?
  - Breaks
  - Lunch
- Physical Class Concerns
  - Restrooms
  - Wi-Fi and other information
  - Emergency procedures
  - Don't leave belongings unattended

No recording, please!

How to get the courseware?

1. Register at **training.confluent.io**
2. Verify your email
3. Log in to **training.confluent.io** and enter your **license activation key**
4. Go to the **Classes** dashboard and select your class

Introductions

- About you:
  - What is your name, your company, and your role?
  - Where are you located (city, timezone)?
  - What is your experience with Kafka?
  - Which other Confluent courses have you attended, if any?
  - What is your language of choice?
  - Optional talking points:
    - What are some other distributed systems you like to work with?
    - What technology most excited you early in your life?
- About your instructor

Starting with Stream Processing Overview

CONFLUENT

Global Education

Agenda

This is a branch of our stream processing content on stateful processing and advanced operations. It is broken down into the following modules:

1. Recap of Kafka and Bridge to Streaming
2. Intro to Kafka Streams

01: Introduction to Kafka Streams

CONFLUENT

Global Education

Module Overview

This module contains two lessons:

- What Do You Need to Know about Group Management in Kafka Before Creating Streaming Applications?
- How Can You Leverage Streaming to Transform the Immutable Data in Your Kafka Cluster?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite: Core Branch of Developer Course
- Recommended Follow-Up: Working with Kafka Streams

01a: What Do You Need to Know about Group Management in Kafka Before Creating Streaming Applications?

Description

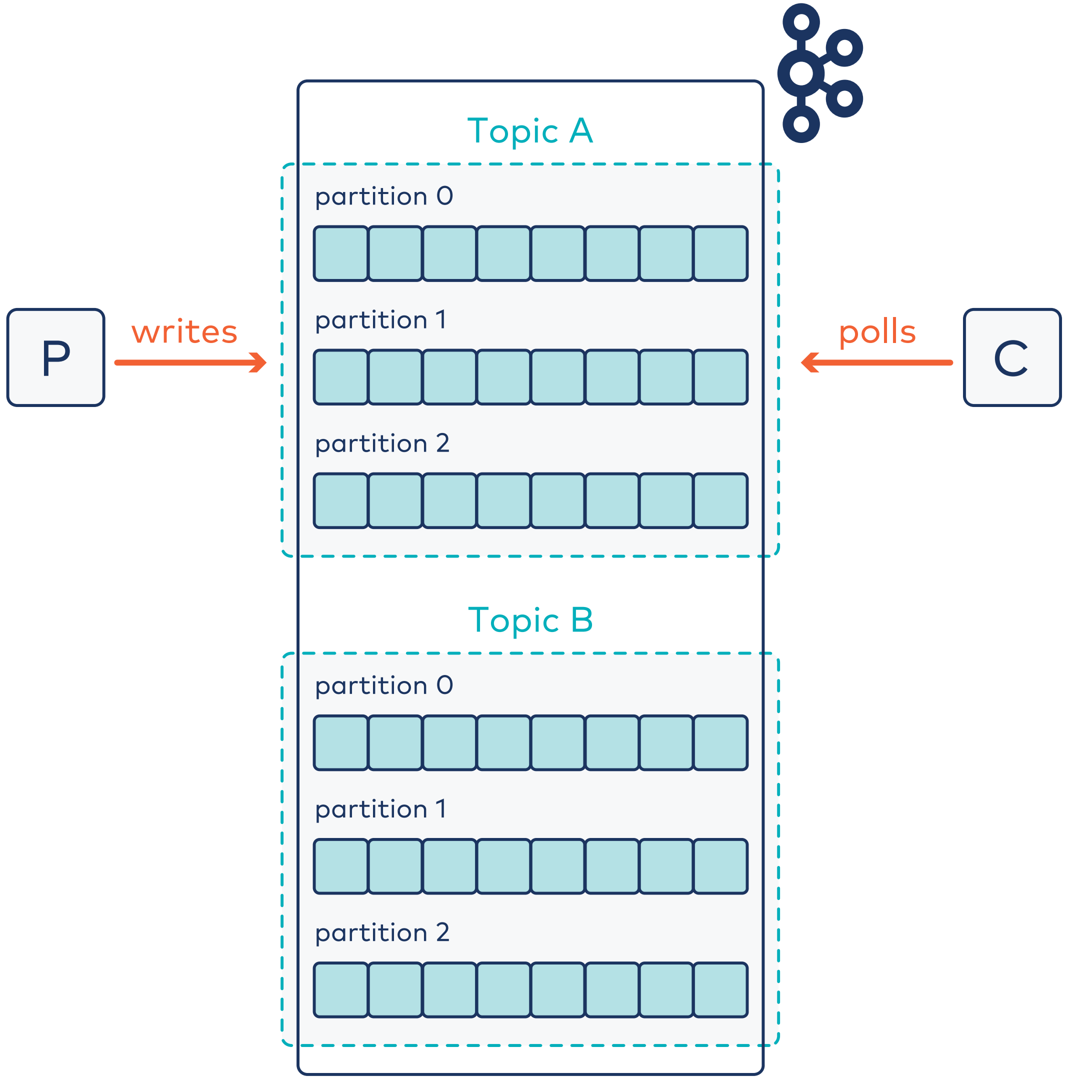
Apache Kafka is the De Facto Standard for Real-Time Event Streaming. The Kafka consumer group protocol allows for hands-off resource management and load balancing. Incremental cooperative rebalancing protocol allows Kafka Streams applications to scale and smoothly handle failures.

What is Apache Kafka?

Kafka is an event streaming platform. Three key benefits are scalability, fault-tolerance, and reliability:

- You can publish and subscribe to events
- Kafka can store events for as long as you want
- You can process and analyze events

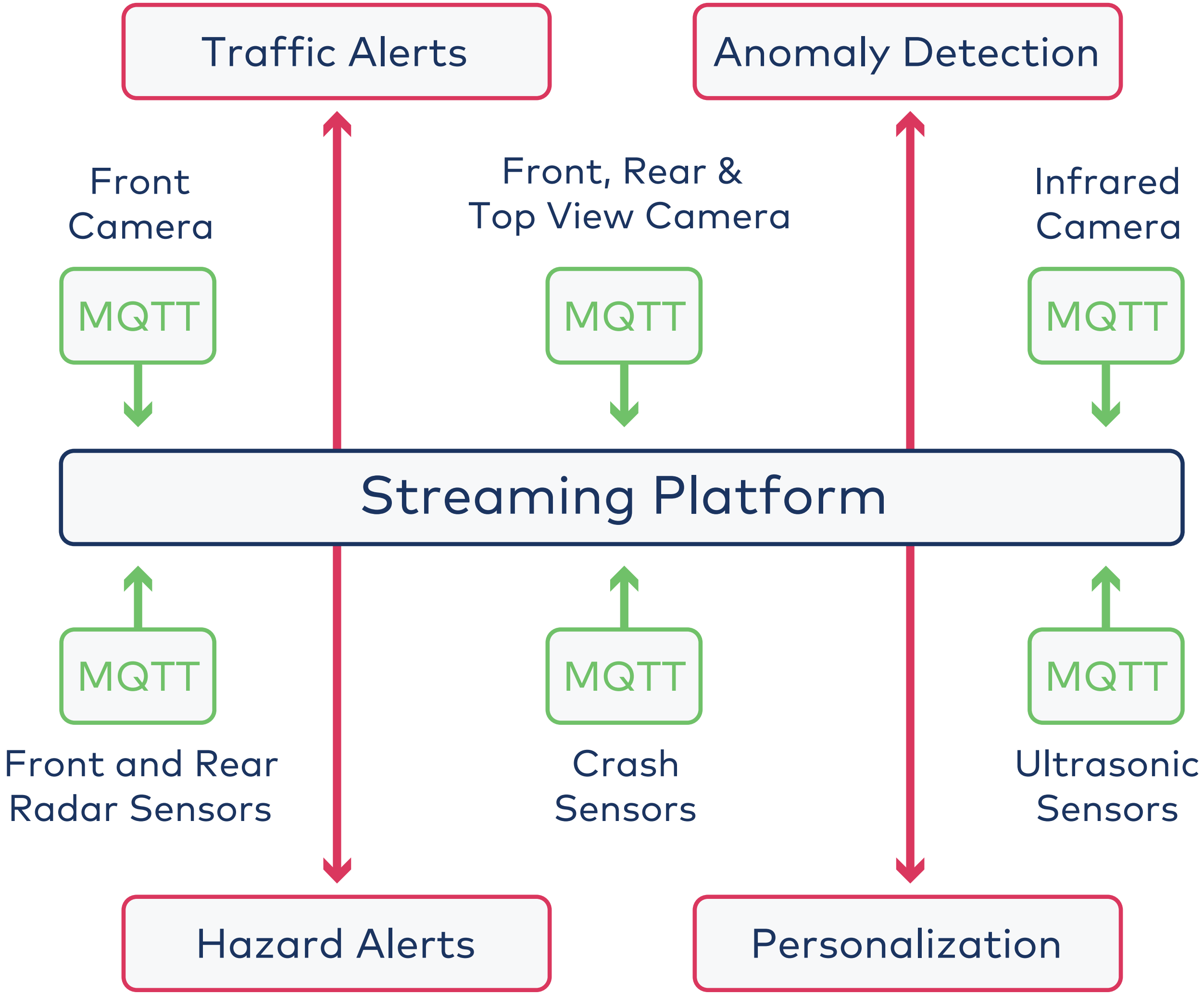
Apache Kafka as a Streaming Platform



Data in Kafka is immutable

Use Case: Automotive Internet of Things

The Future of the
Automotive Industry is a
Real-Time Data Cluster



Traffic Alerts

Anomaly Detection

Front
Camera

MQTT

Front, Rear &
Top View Camera

MQTT

Infrared
Camera

MQTT

Streaming Platform

MQTT

Front and Rear
Radar Sensors

MQTT

Crash
Sensors

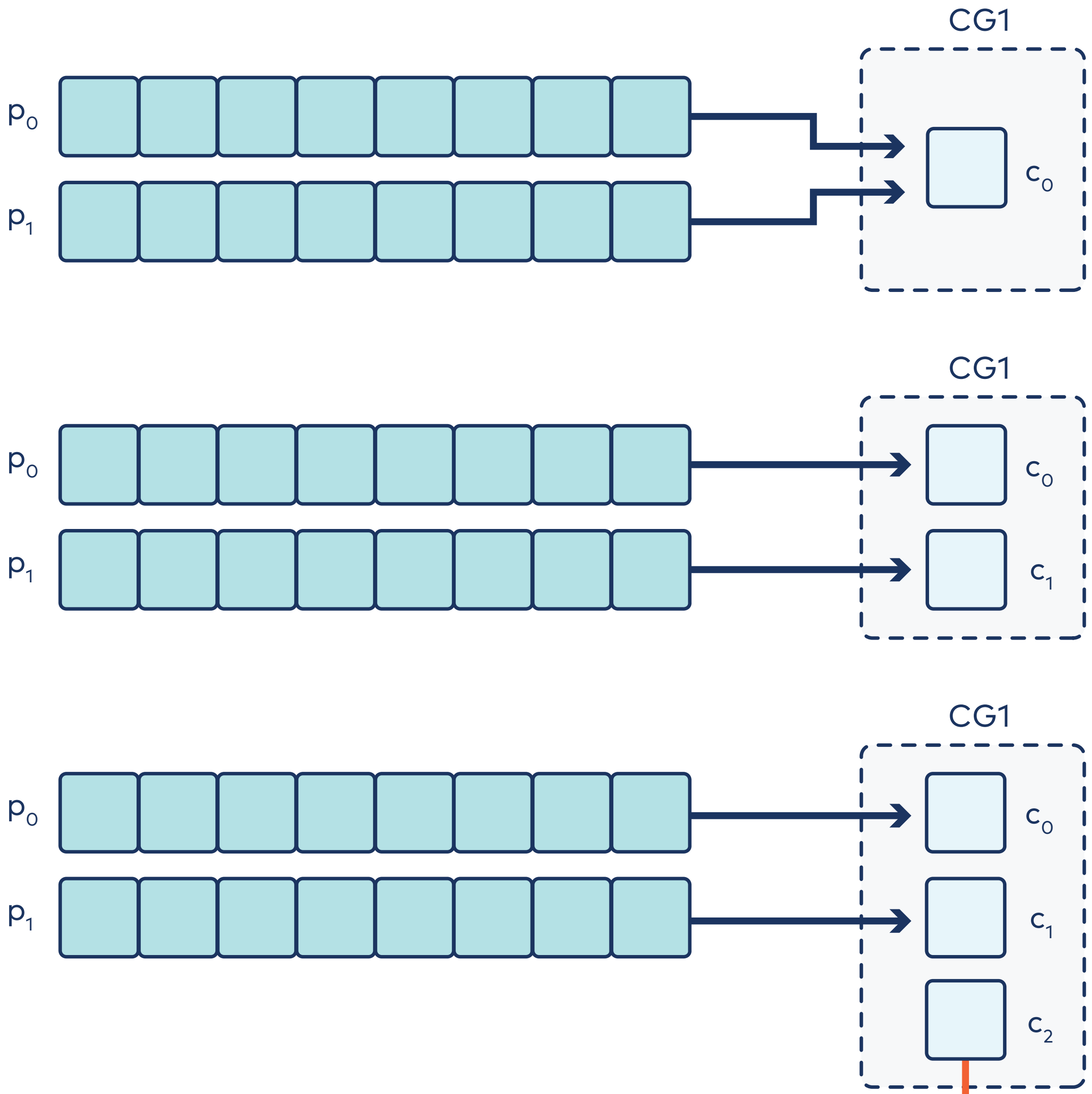
MQTT

Ultrasonic
Sensors

Hazard Alerts

Personalization

Consumer Groups



Stays idle and is available to provide the fault tolerance and load balance

Consumer Partition Assignment Strategy

RangeAssignor

- Use when joining data from multiple topics (default)

RoundRobin

- Use when performing stateless operations on records from many topics
  - **Sticky**
    - RoundRobin with a best effort to maintain assignments across rebalances
  - **CooperativeSticky**
    - Sticky but it uses consecutive rebalances rather than the single stop-the-world used by Sticky

01b: How Can You Leverage Streaming to Transform the Immutable Data in Your Kafka Cluster?

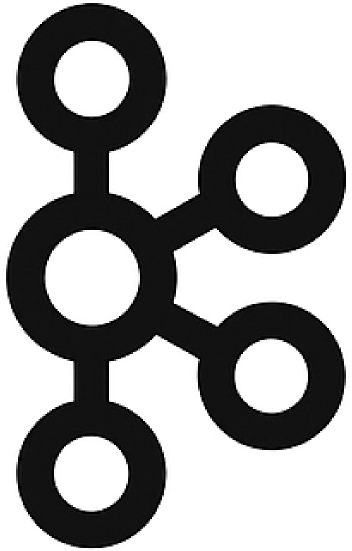
Description

Kafka Streams, Confluent ksqlDB and Flink are three of the options to build real-time streaming applications. Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in an Apache Kafka cluster. The Streams processor topology defines the stream processing computational logic for the application.

How Do You Process Data in Kafka?

- Data in Kafka is immutable
- But what if you need to transform, enrich the data in Kafka? For example...
  - Filter, merge, group, repartition, etc. (stateless)
  - Aggregate, join, etc. (stateful)

Options for Writing Streaming Applications

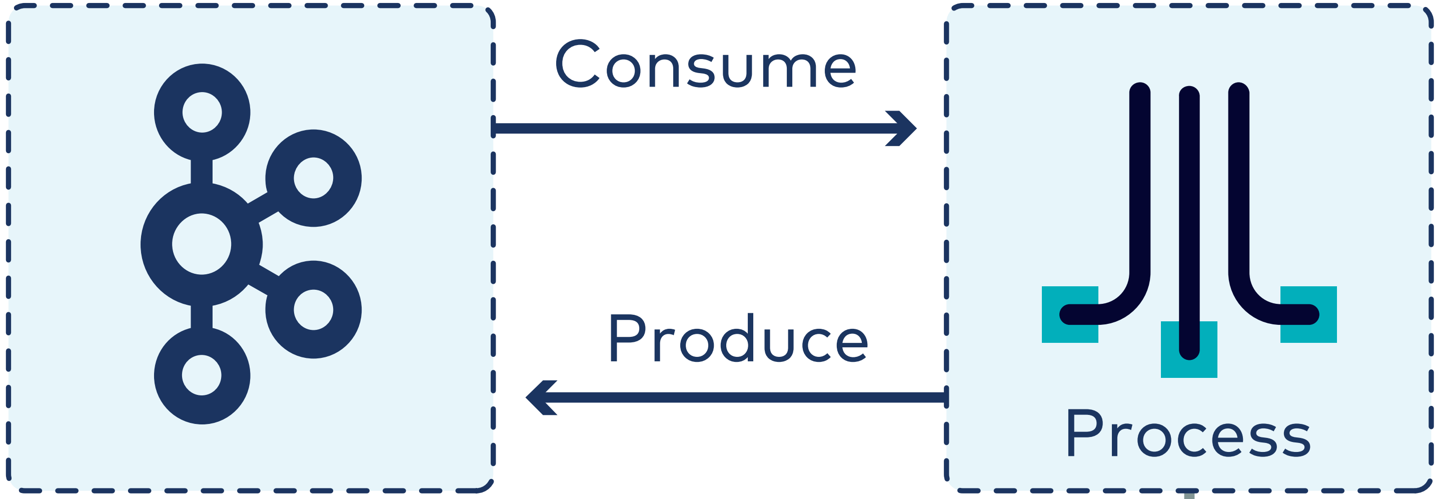


Kafka Streams

ksqlDB

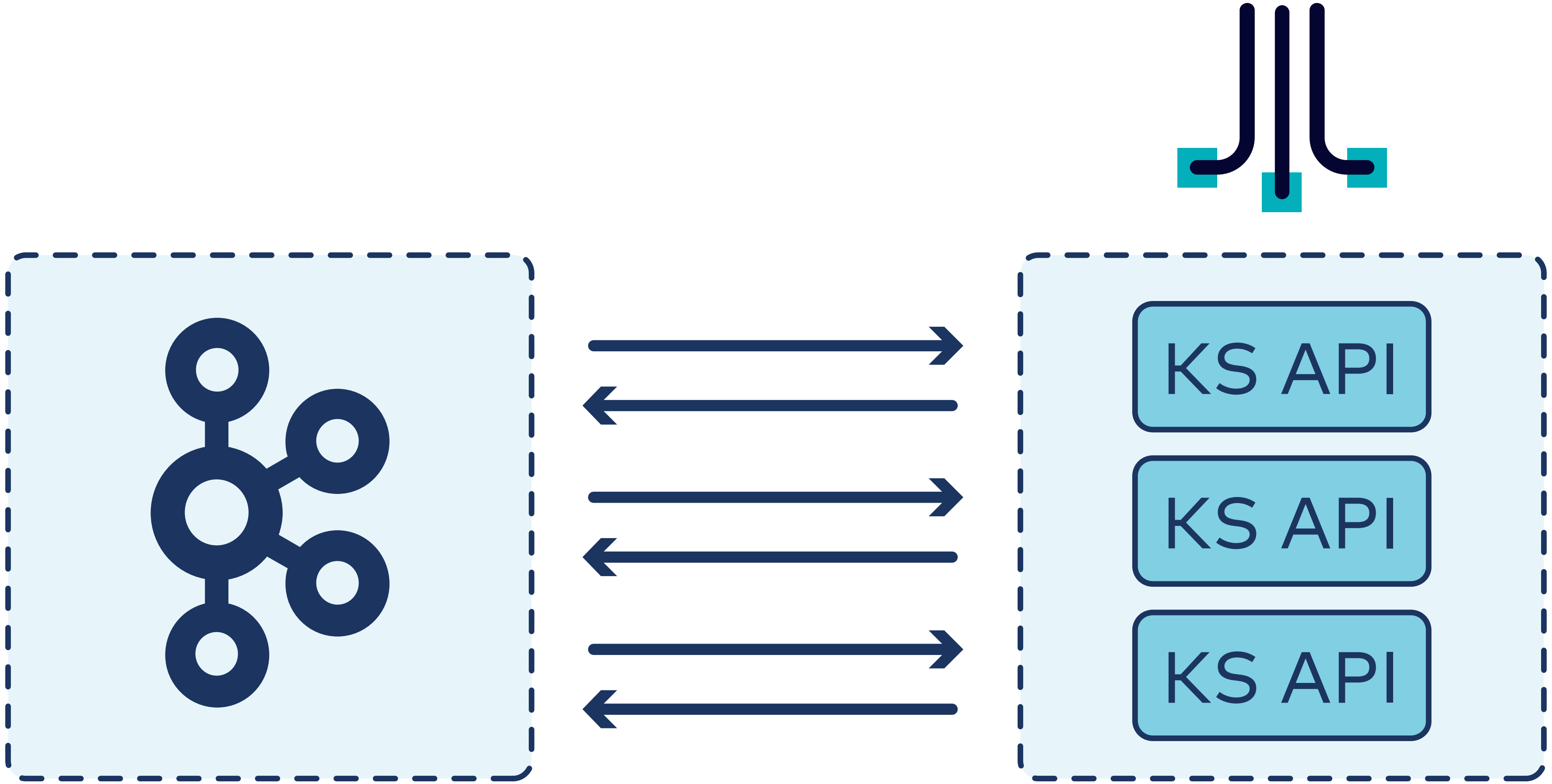
Flink

Kafka Streams is a Client of Kafka



Kafka Streams application doesn't run inside the brokers

Same Application, Many Instances



Parallelism Model

Kafka Streams uses the concepts of stream partitions and stream tasks as logical units of its parallelism model.

Links between Kafka Streams and Kafka:

- Each stream partition…
  - is an ordered sequence of data records.
  - maps to a Kafka topic partition.
- A data record in the stream maps to a Kafka message from that topic.
- The keys of data records determine the partitioning of data in both Kafka and Kafka Streams.

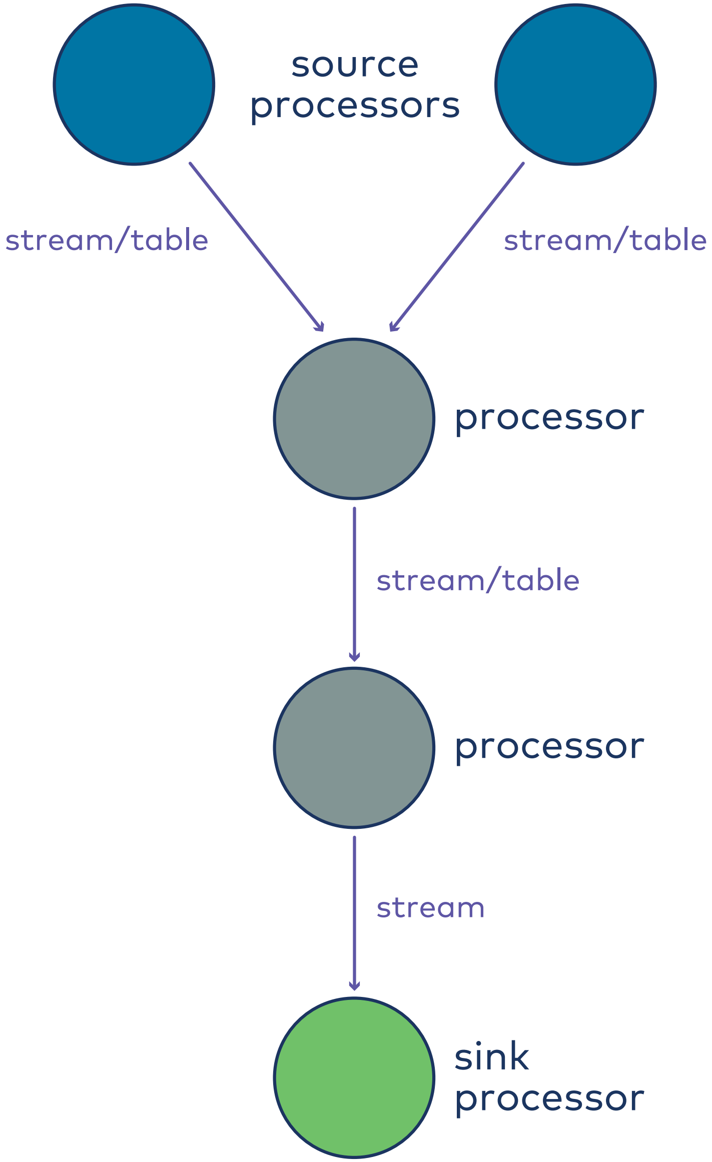
Processor Topology

Processor topology

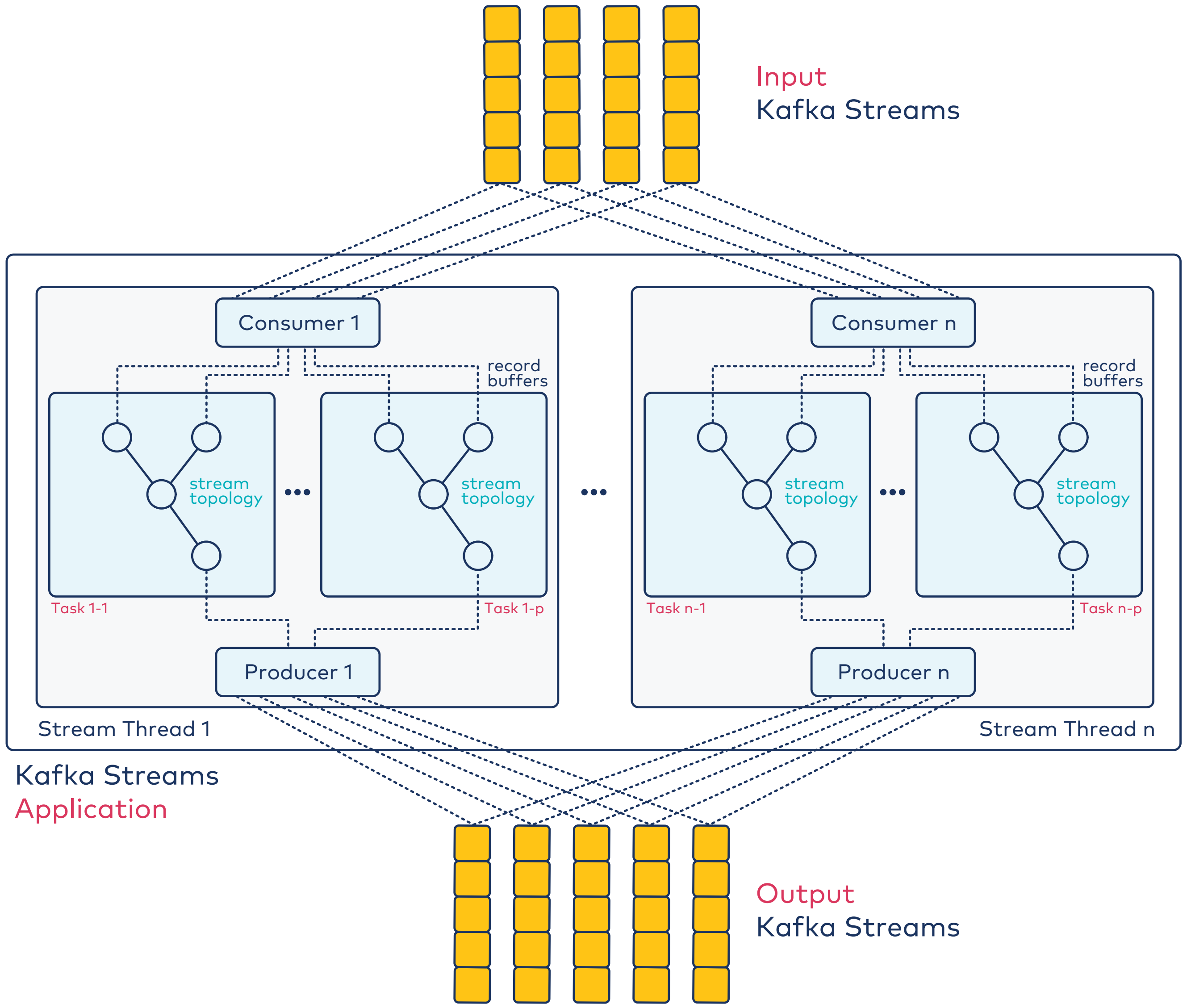
computational logic of the data processing performed by a stream processing application.

Details:

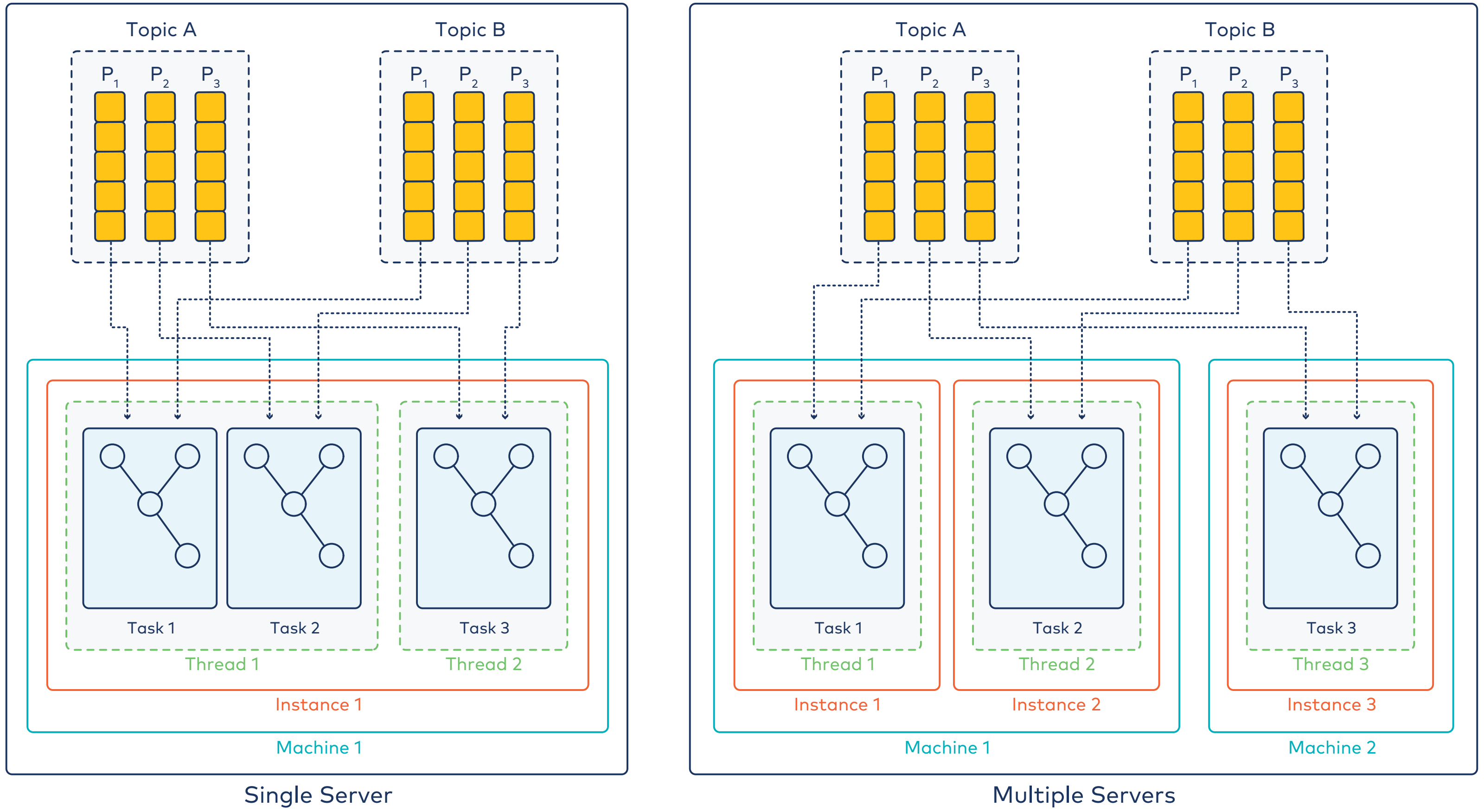
- A topology is a graph of stream processors (nodes) that are connected by streams (edges).
- You can define topologies via the low-level Processor API or via the Kafka Streams DSL.



Streams Architecture - Single Application Instance with Multiple Threads Configured



Streams Architecture - Multiple Application Instances on Multiple Machines



Lab: Scaling a Kafka Streams Application

Please work on **Lab 1a: Scaling a Kafka Streams Application**

Refer to the Exercise Guide

CONFLUENT
Global Education

Module Overview

This module contains four lessons:

- What Are the Big-Picture Kafka Streams Concepts?
- How Do You Put Together a Kafka Streams Application?
- What are Some Operations You Can Use To Transform Streams?
- What Changes When Your Stream Processing Needs to Track State?

Where this fits in:

- Hard Prerequisite: Introduction to Kafka Streams
- Recommended Prerequisite: Core Branch of Developer Course

02a: What Are the Big-Picture Kafka Streams Concepts?

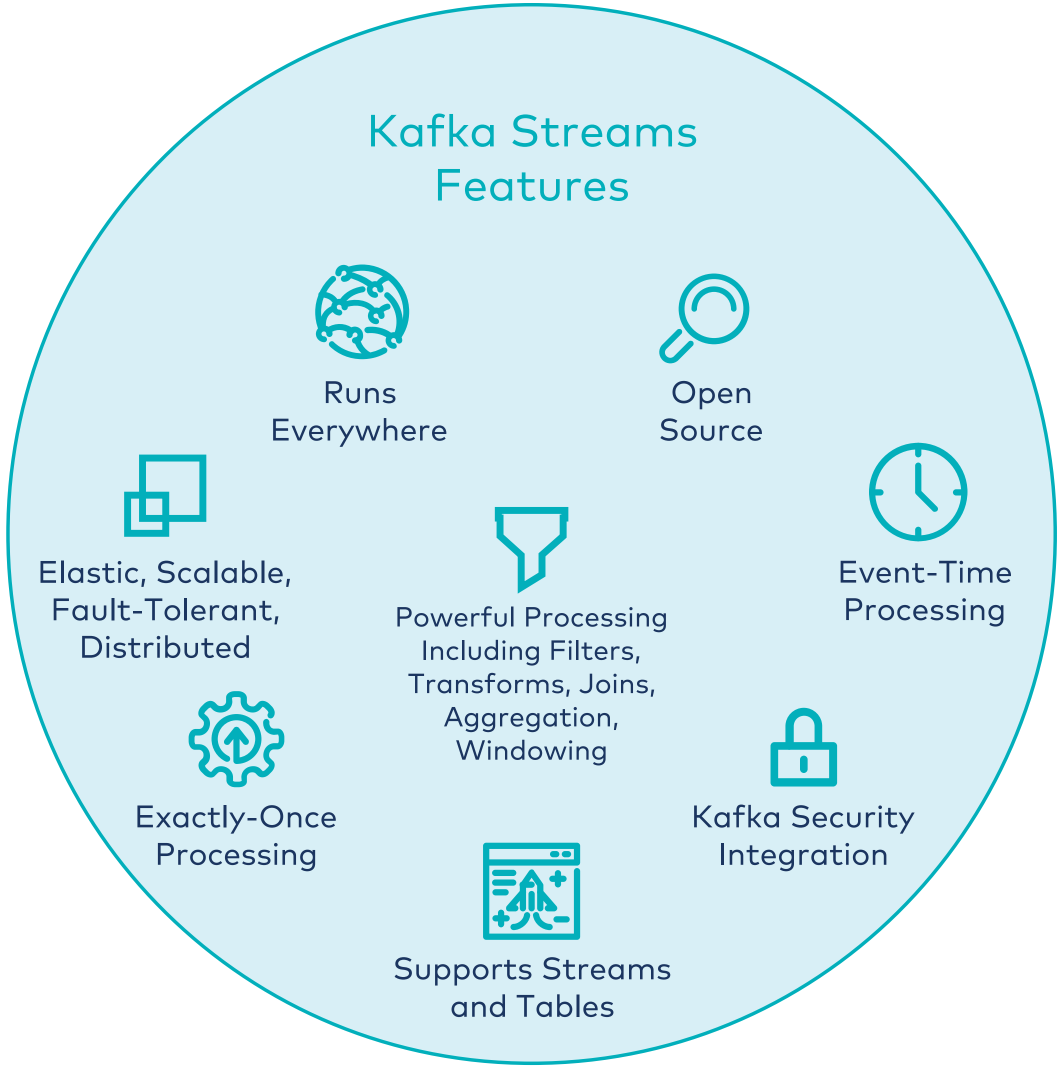
Description

Kafka Streams is a Java library, and it uses the Domain Specific Language to define processor topology. The Kafka Streams DSL is built on top of the Streams Processor API and it has built-in abstractions for streams and tables in form of `KStream`, `KTable`, and `GlobalKTable` objects.

Kafka Streams Applications

- The Streams API of Apache Kafka is available through a Java library which can be used to write a distributed streams application.
- Using the DSL (Domain Specific Language), you can define processor topologies in your application. The steps are:
    - *Consume*: Specify input streams that are read from Kafka topics.
    - *Process*: Compose transformations on these streams.
    - *Produce*: Write the resulting output streams back to Kafka topics.

Properties of Kafka Streams



Kafka Streams Features

Runs Everywhere

Open Source

Elastic, Scalable, Fault-Tolerant, Distributed

Powerful Processing Including Filters, Transforms, Joins, Aggregation, Windowing

Event-Time Processing

Exactly-Once Processing

Kafka Security Integration

Supports Streams and Tables

Kafka Streams Applications Topology

- The logic is defined as a processor topology, the graph of stream processors and streams.
- You can define the processor topology with the Kafka Streams APIs:
  - Kafka Streams DSL
  - Processor API

Kafka Streams DSL

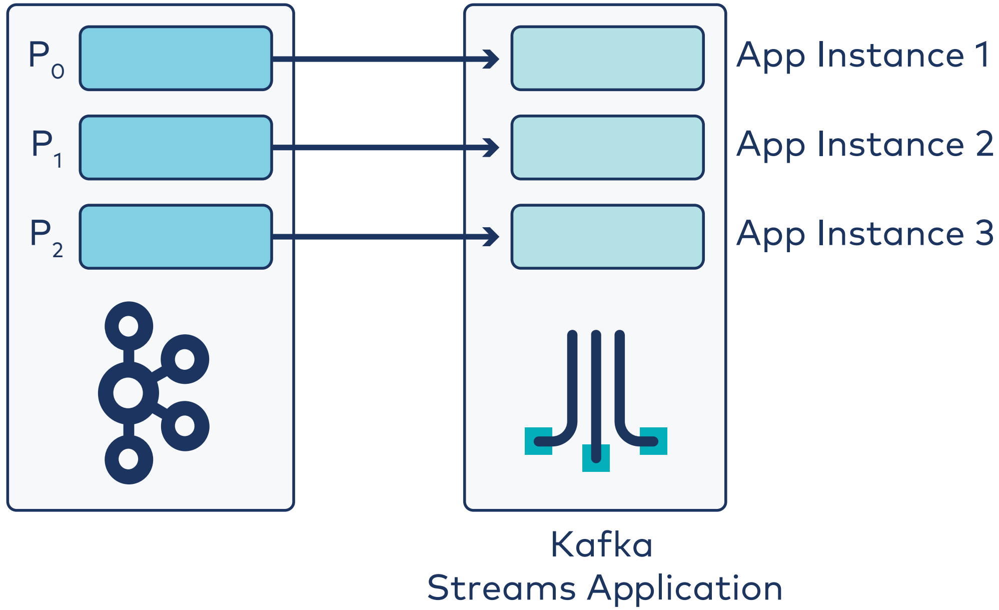
- The Kafka Streams DSL is built on top of the Streams Processor API.
- It has built-in abstractions for streams and tables in form of:
    - `KStream` (stream)
    - `KTable` (table)
        - `GlobalKTable`

Stream and Table Example

Event	Stream	Table	State
Bus XYZ (key) **departed** from NYC (value)	Insert	Insert	Traveling to Chicago
Bus XYZ (key) **arrived** at Chicago (value)	Insert	Update	Waiting for passengers
Bus ABC (key) **departed** from Boston (value)	Insert	Insert	Traveling to Florida
Bus XYZ (key) **departed** from Chicago (value)	Insert	Update	Traveling to Salt Lake City
Bus XYZ (key) **arrived** at Salt Lake City (value)	Insert	Update	Waiting for passengers
Bus XYZ (key) : null (value)	Insert	Delete	Bus is decommissioned
Key (null): arrived at San Francisco	Insert	Ignored	(blank)

KStream and **KTable** Objects

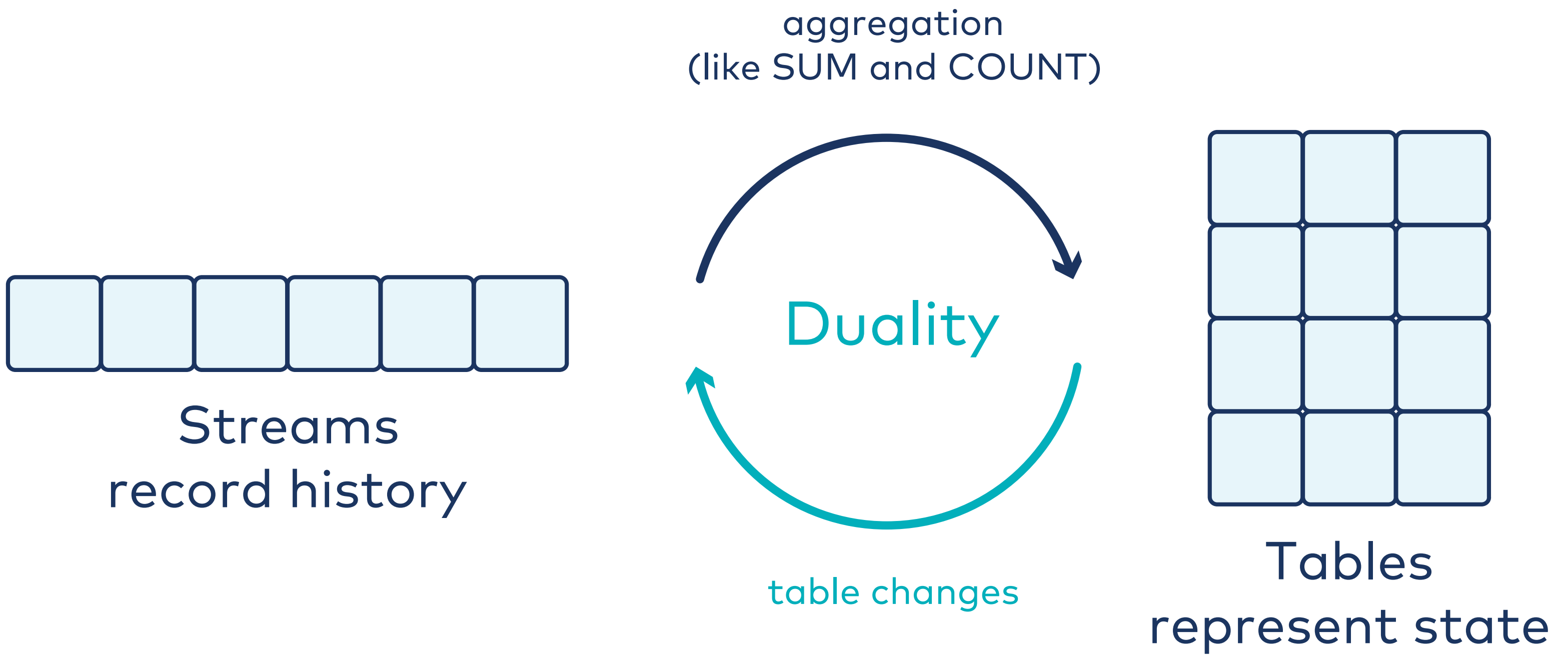
KStream	KTable
Immutable	Mutable
Unbounded	Bounded
Insert (append)	Insert/Update/Delete
Can have many events/key	One event/key
Partitioned	Partitioned
Ordering is guaranteed per partition	Ordering is not guaranteed per partition
Persistent, durable, and fault-tolerant	Persistent, durable, and fault-tolerant



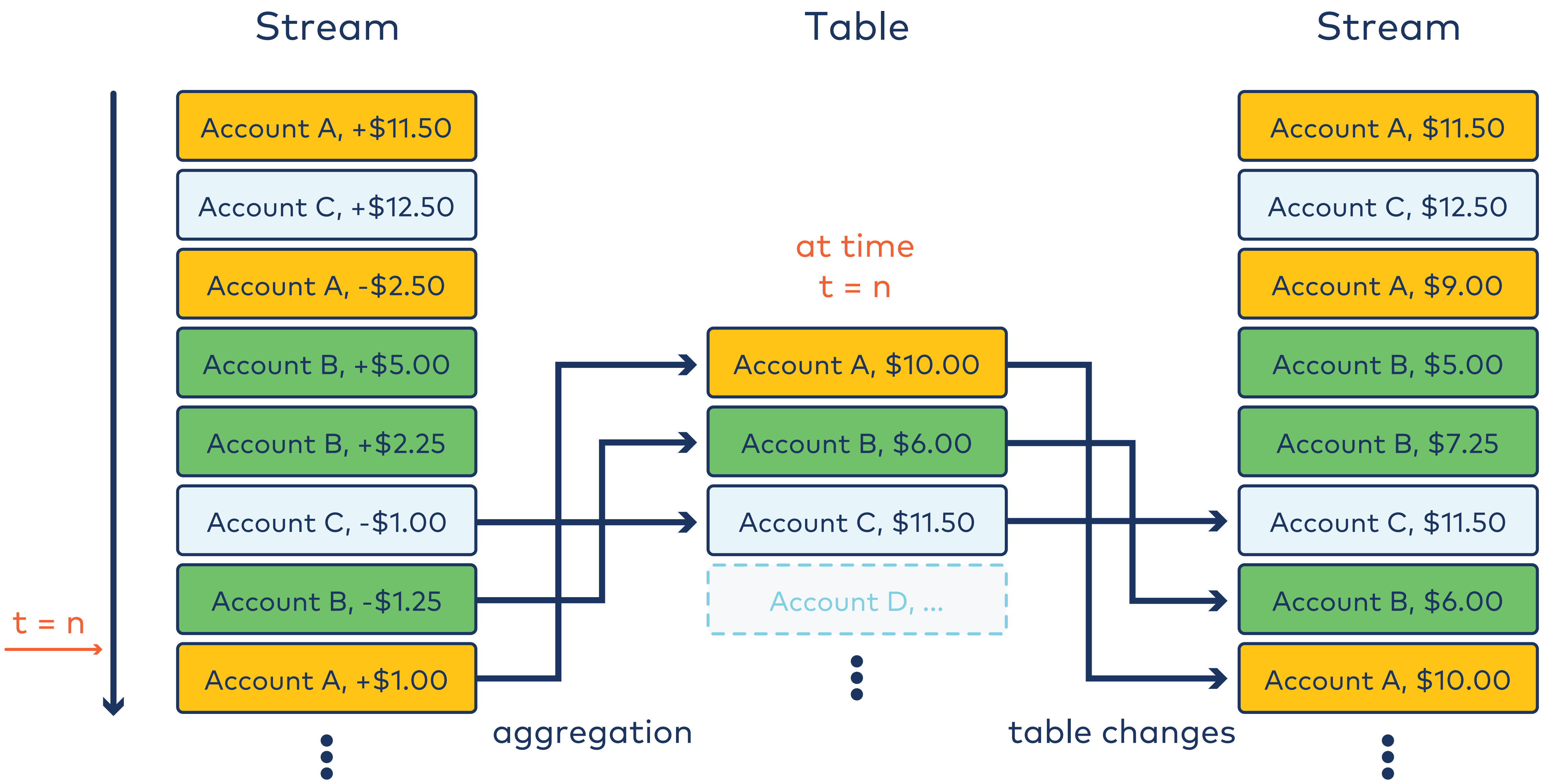
Stream-Table Duality

Relationship between streams and tables:

- You can turn a stream into a table by aggregating the stream with operations such as `COUNT()` or `SUM()`
- We can turn a table into a stream by capturing the changes made to the table—inserts, updates, and deletes—into a "change stream."



Stream-Table Duality With Aggregation Example



Stream

- Account A, +$11.50
- Account C, +$12.50
- Account A, -$2.50
- Account B, +$5.00
- Account B, +$2.25
- Account C, -$1.00
- Account B, -$1.25
- Account A, +$1.00

t = n

aggregation

Table

at time
t = n

- Account A, $10.00
- Account B, $6.00
- Account C, $11.50
- Account D, ...

table changes

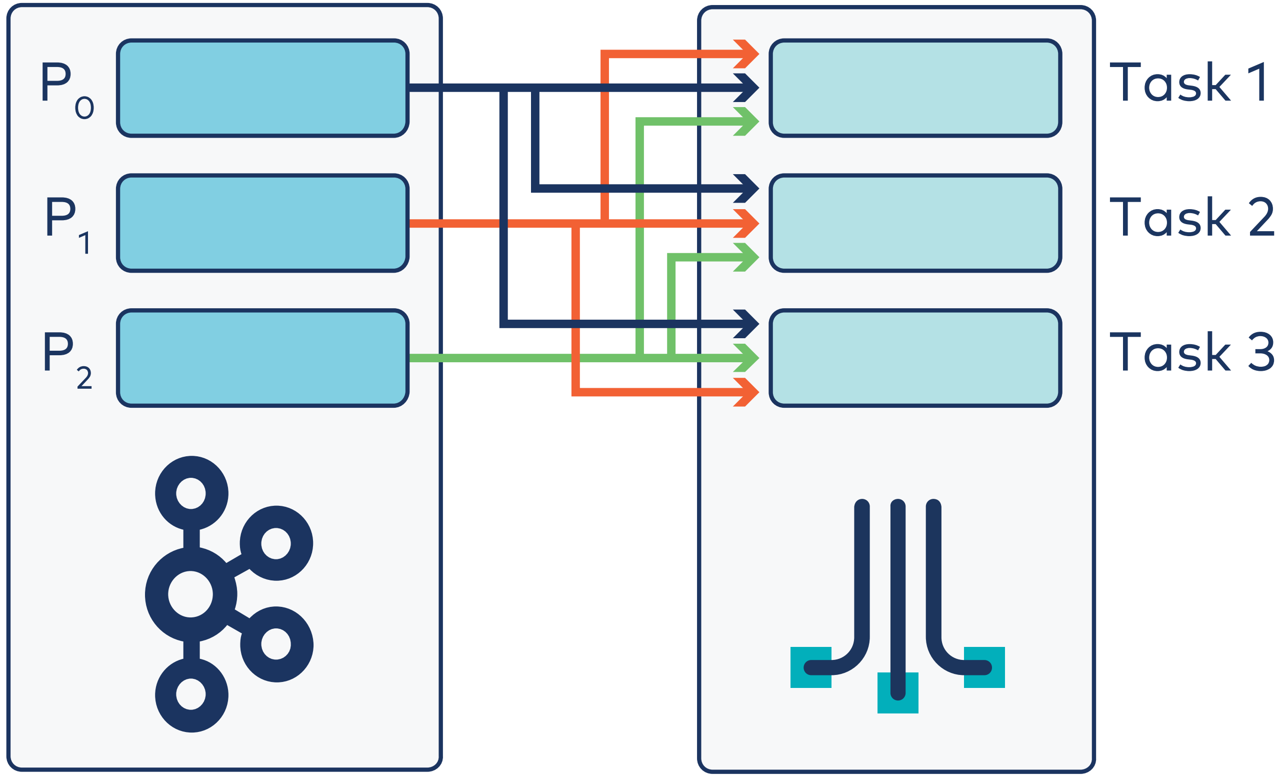
Stream

- Account A, $11.50
- Account C, $12.50
- Account A, $9.00
- Account B, $5.00
- Account B, $7.25
- Account C, $11.50
- Account B, $6.00
- Account A, $10.00

GlobalKTable

GlobalKTable: All keys from all partitions can be queried locally in each application instance since the whole topic will be consumed by each task

- **NOT** partitioned
- Mutable
- Bounded
- Insert/Update/Delete
- One event/key
- Ordering is not guaranteed per partition
- Persistent, durable, and fault-tolerant



Kafka
Streams Application

Activity: Streams vs. Tables

Review the scenario on the left and determine the type of object from the right column that is best suited to storing it.

Scenario	Table/Stream
Checking account balance	Table? / Stream?
The past five years of experience for your resume	Table? / Stream?
Sequence of moves in a chess game	Table? / Stream?
State of a chess board at a given time	Table? / Stream?
Count of countries to which you have traveled	Table? / Stream?
Your addresses over the last five years for a visa application	Table? / Stream?
Items you have shopped for online	Table? / Stream?
RSVP responses from guests for a party you are having	Table? / Stream?

02b: How Do You Put Together a Kafka Streams Application?

Description

Any Java application that makes use of the Kafka Streams library is considered a Kafka Streams application. The computational logic of a Kafka Streams application is defined as a processor topology. A Kafka Streams Application written in Java has five clearly identifiable sections.

Kafka Streams Application Anatomy

Imports

```
code goes here
```

```
public class StreamsApp
{
   public static void main(String[] args)
   {
```

Config

```
      //code goes here
```

Streaming topology

```
      //code goes here
```

Shutdown behavior

```
      //code goes here
```

Start app

```
      //code goes here
```

```
   }
}
```

Kafka Streams Application Anatomy - Imports

Libraries available for writing Kafka Streams applications:

Group ID	Artifact ID	Version	Description	Req?
`org.apache.kafka`	`kafka-streams`	7.0.1-ccs	Base library for Kafka Streams	Yes
`org.apache.kafka`	`kafka-streams-scala_2.11`, `kafka-streams-scala_2.12`	7.0.1-ccs	Scala API for Kafka Streams	No
`org.apache.kafka`	`kafka-clients`	7.0.1-ccs	Apache Kafka® client library, contains built-in serializers/deserializers	Yes
`org.apache.avro`	`avro`	1.8.2	Apache Avro library	Avro only
`io.confluent`	`kafka-streams-avro-serde`	7.0.1	Confluent's Avro Serializer/Deserializer	Avro only

Kafka Streams Application Anatomy - Configuration

Imports

Config

```
Properties settings = new Properties();
settings.put(StreamsConfig.APPLICATION_ID_CONFIG,
             "streams-app-1");
settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
             "kafka-1:9092, kafka-2:9092, kafka-3:9092");
settings.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
             Serdes.String().getClass());
settings.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
             Serdes.Double().getClass());
// ...
```

Topology

```
//code goes here
```

Shutdown

```
//code goes here
```

Start app

```
//code goes here
```

Kafka Streams Application Anatomy - Topology

Imports

Config

```
// code goes here
```

Streaming topology

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, Double> temperatures =
   builder.stream("temp-topic");
KStream<String, Double> highTemps =
   temperatures.filter((key, value) -> value > 25);
highTemps.to("high-temps-topic");

Topology topology = builder.build();
```

Shutdown

```
//code goes here
```

Start app

```
//code goes here
```

Kafka Streams Application - Shutdown

Imports

Config

```
// code goes here
```

Topology

```
// code goes here
```

```
KafkaStreams streams = new KafkaStreams(topology, settings);
```

Shutdown behavior

```
final CountDownLatch latch = new CountDownLatch(1);
Runtime.getRuntime().addShutdownHook(new(Thread(() -> {
   streams.close();
   latch.CountDown();
}));
```

Start app

```
//code goes here
```

Kafka Streams Application - Start Streaming

Imports

Config

```
// code goes here
```

Streaming topology

```
// code goes here
```

```
KafkaStreams streams = new KafkaStreams(topology, settings);
```

Shutdown

```
// code goes here
```

Start app

```
try
{
   streams.start();
   latch.await();
}
catch(final Throwable e) { /* ... */ }
System.exit(0);
```

Summary: Full Program (1)

```java
// imports

public class StreamsApp
{
   public static void main(String[] args)
   {
      Properties settings = new Properties();
      settings.put(StreamsConfig.APPLICATION_ID_CONFIG", streams-app-1");
      settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
                   "kafka-1:9092, kafka-2:9092, kafka-3:9092");
      settings.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
      settings.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Double().getClass());
      // ...

      StreamsBuilder builder = new StreamsBuilder();

      KStream<String, Double> temperatures = builder.stream("temp-topic");
      KStream<String, Double> highTemps = temperatures.filter((key, value) -> value > 25);
      highTemps.to("high-temps-topic");
```

Summary: Full Program (2)

```java
      Topology topology = builder.build();

      KafkaStreams streams = new KafkaStreams(topology, settings);

      final CountDownLatch latch = new CountDownLatch(1);
      Runtime.getRuntime().addShutdownHook(new(Thread(() -> {
         streams.close();
         latch.CountDown();
      }));

      try
      {
         streams.start();
         latch.await();
      }
      catch(final Throwable e) { /* ... */ }
      System.exit(0);
   }
}
```

Alternate Serde Configuration

In the running example, we specified default Serdes in the Config.

Alternative: specify Serdes upon each use. Here is what is different:

```
final Serde<String> stringSerde = Serdes.String();
final Serde<Double> doubleSerde = Serdes.Double();
```

Streaming topology

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, Double> temperatures =
   builder.stream("temp-topic",
                  Consumed.with(stringSerde, doubleSerde));
KStream<String, Double> highTemps =
   temperatures.filter((key, value) -> value > 25);
highTemps.to("high-temps-topic",
             Produced.with(stringSerde, doubleSerde));

Topology topology = builder.build();
```

Lab: Anatomy of a Kafka Streams App

Please work on **Lab 2a: Anatomy of a Kafka Streams App**

Refer to the Exercise Guide

02c: What are Some Operations You Can Use To Transform Streams?

Description

The `KStream` and `KTable` interfaces support stateless and stateful transformations. Stateless transformations `mapValues`, `flatMapValues`, `filter`, etc. Some of the stateless transformations like `map` and `flatMap` mark the stream for repartitioning.

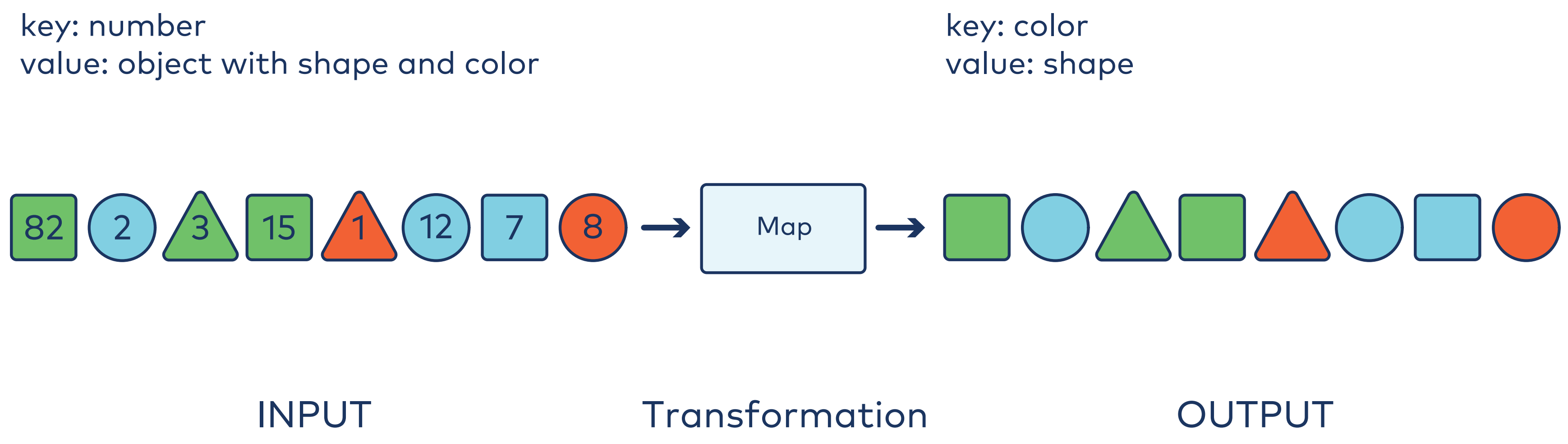
Transforming Data

- Kafka streams supports a number of transformation operation using the objects **`KStream`** and **`KTable`**.
- These operations can be translated into one or more connected processors into the underlying processor topology.
  - Some `KStream` transformations may generate one or more `KStream` objects or a `KTable` object.
  - All `KTable` transformation operations can only generate another `KTable`.
  - All of these transformation methods can be chained together to compose a complex processor topology.
  - The transformation operations fall into these categories:
    - Stateless
    - Stateful

Stateless Transforming

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

Stateless Operations - Mapping



map

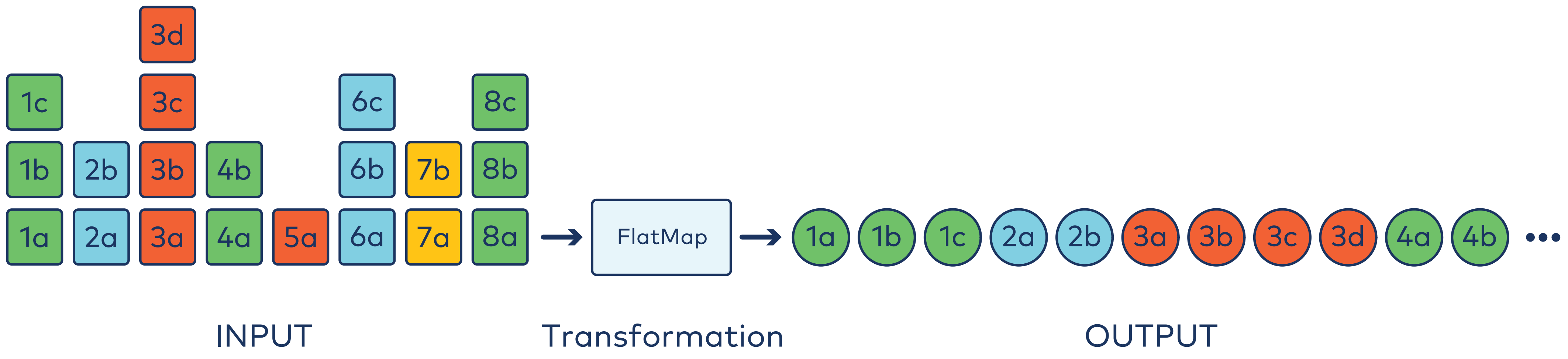
- new key and new value

```
KStream<Color, Shape> result =
    stream.map((key, value) ->
                KeyValue.pair(value.color, value.shape));
```

map values

```
KStream<Integer, String> result =
    stream.mapValues(value -> value.toUpperCase());
```

Stateless Operations - **flatMap**



flat map

- new key and new value

```
KStream<Char, String> words =
    sentences.flatMap((key, value) ->
                        KeyValue.pair(value.substring(0,1),
                                      Arrays.asList(value.split("\\W+"))));
```

flat map values

```
KStream<byte[], String> words =
    sentences.flatMapValues(value -> Arrays.asList(value.split("\\W+")));
```

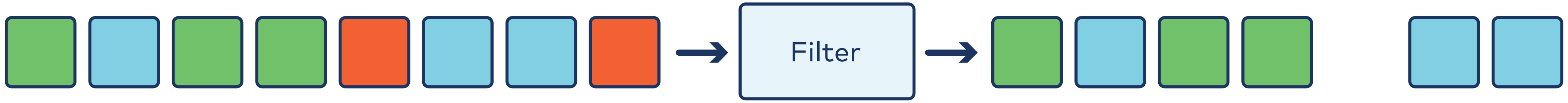
Stateless Operations - **selectKey**

select key

```java
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream(...);

KStream<String, String> rekeyed =
    stream.selectKey((key, value) -> value.split(" ")[0]);
```

Stateless Operations - Filtering

key: shape
value: color



INPUT Transformation OUTPUT

filter

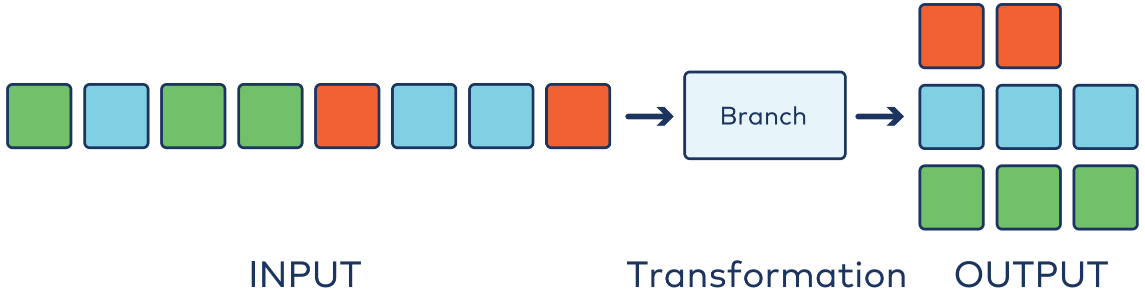
```
KStream<Shape, Color> nonOrangeItems =
    stream.filter((key, value) -> !value.equals("orange"));
```

inverse filter

```
KStream<Shape, Color> nonOrangeItems =
    stream.filterNot((key, value) -> value.equals("orange"));
```

Stateless Operations - Splitting

Need to change graphic. Best done leveraging Seth...



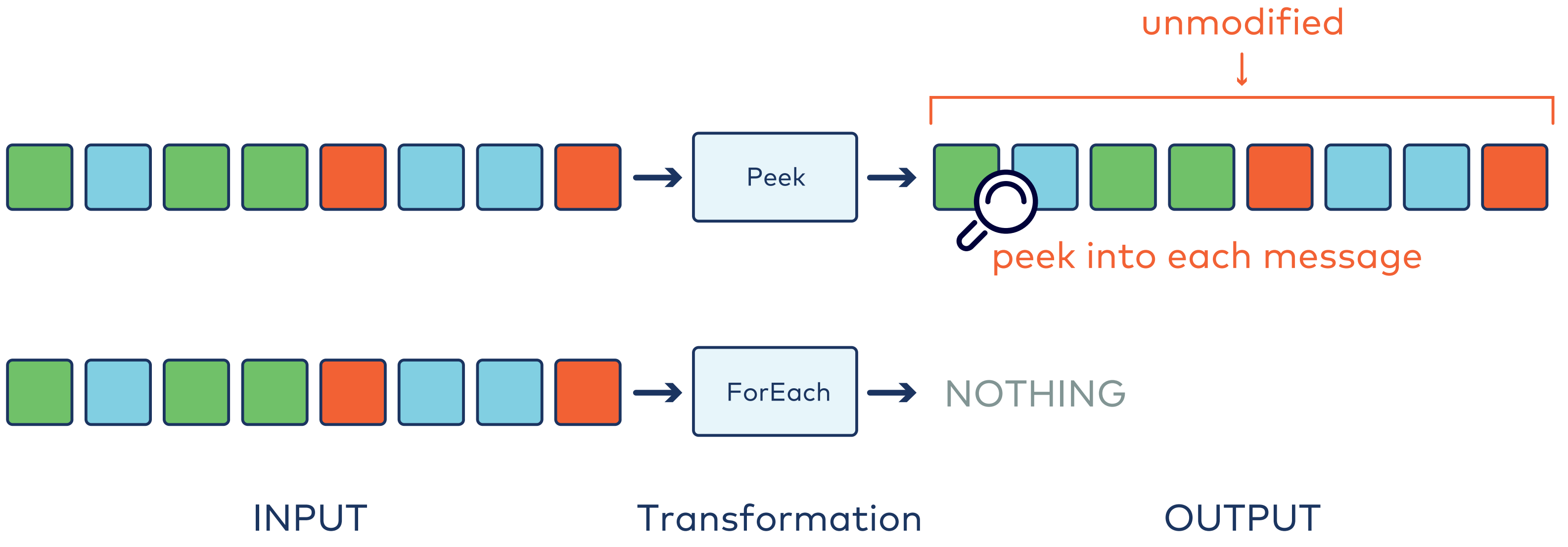
branch

```
Map<String, KStream<String, Long>> branches = stream.split()
  .branch((key, value) -> key.startsWith("A")) /* predicate 1 */
  .branch((key, value) -> key.startsWith("B")) /* predicate 2 */
  .defaultBranch()                             /* default branch */
```

Returns 3 new `KStream`s:

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

Stateless Operations - **peek** & **forEach**



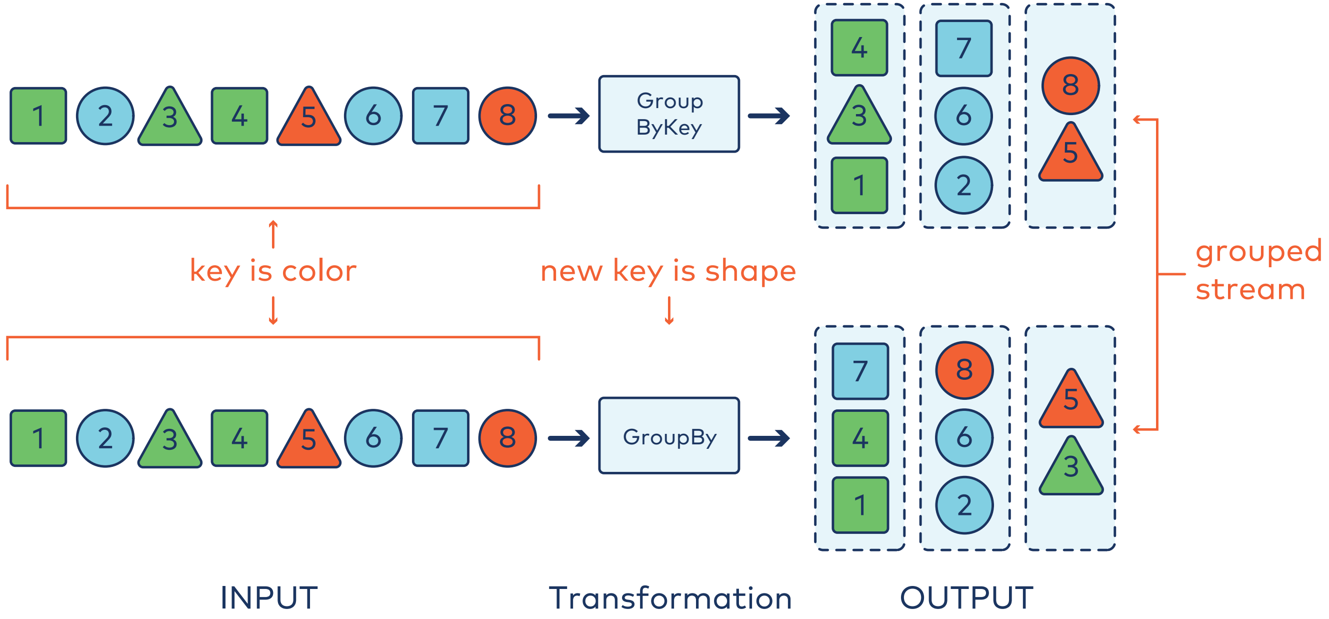
peek

```
KStream<String, Color> unmodifiedStream =
    stream.peek((key,value) -> someAction(key, value));
```

for each

```
stream.forEach((key,value) -> someAction(key, value));
```

Stateless Operations - **groupByKey** & **groupBy**



group by key

```
KGroupedStream<Color, Shape> groupedStream
                        = stream.groupByKey();
```

group by

```
KGroupedStream<Shape, Shape> groupedStream
                        = stream.groupBy((key, value) -> value);
```

Output object is a `KGroupedStream`. There is also a `KGroupedTable`.

Stateless Operations - **toStream**

Table to Stream

```
StreamsBuilder builder = new StreamsBuilder();
KTable<byte[], String> table = builder.table(...);

KStream<byte[], String> stream = table.toStream();
```

Repartitioning

- Internal repartitioning topics are internal intermediate topics that are created by the Streams API.
- Kafka Streams creates two types of internal topics with the following naming convention:
  - **Repartitioning:** `<applicationID>-<operatorName>-repartition`
  - **Changelog:** `<applicationID>-<operatorName>-changelog`

ℹ More to come on the changelog in a later lesson on Fault Tolerance.

Repartitioning (2)

- The following are some of the functions which cause repartitioning:
  - `groupBy`
  - `map`
  - `flatMap`
  - `selectKey`
- Repartitioning can also be triggered manually using the `repartition()` method:

```
KStream<byte[], String> stream = ... ;
KStream<byte[], String> repartitionedStream =
        stream.repartition(Repartitioned.numberOfPartitions(10));
```

Activity: Stateless True/False

True or false?

1. Aggregation is applied to records of the same key.
2. Grouping is a prerequisite for aggregation.
3. You cannot run `groupBy` on a `KStream` or a `KTable`.

Lab: Working With JSON

Please work on **Lab 2b: Working With JSON**

Refer to the Exercise Guide

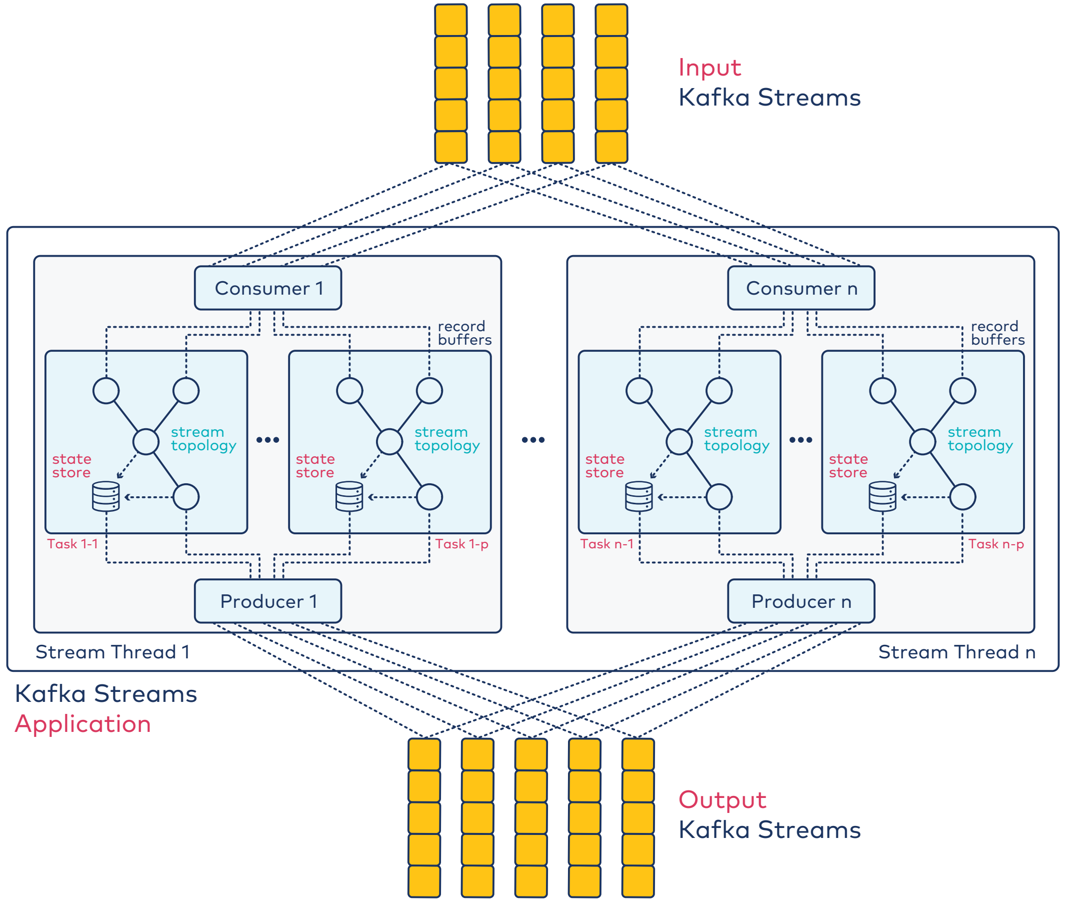
02d: What Changes When Your Stream Processing Needs to Track State?

Description

Stateful transformations depend on state for processing inputs and producing outputs and require a state store associated with the stream processor. The Kafka Streams API enables your applications to be queryable using Interactive Queries. The `KTable` abstraction leverages configured memory (RAM) size of cache for internal caching and compaction of records.

Stateful Transformations

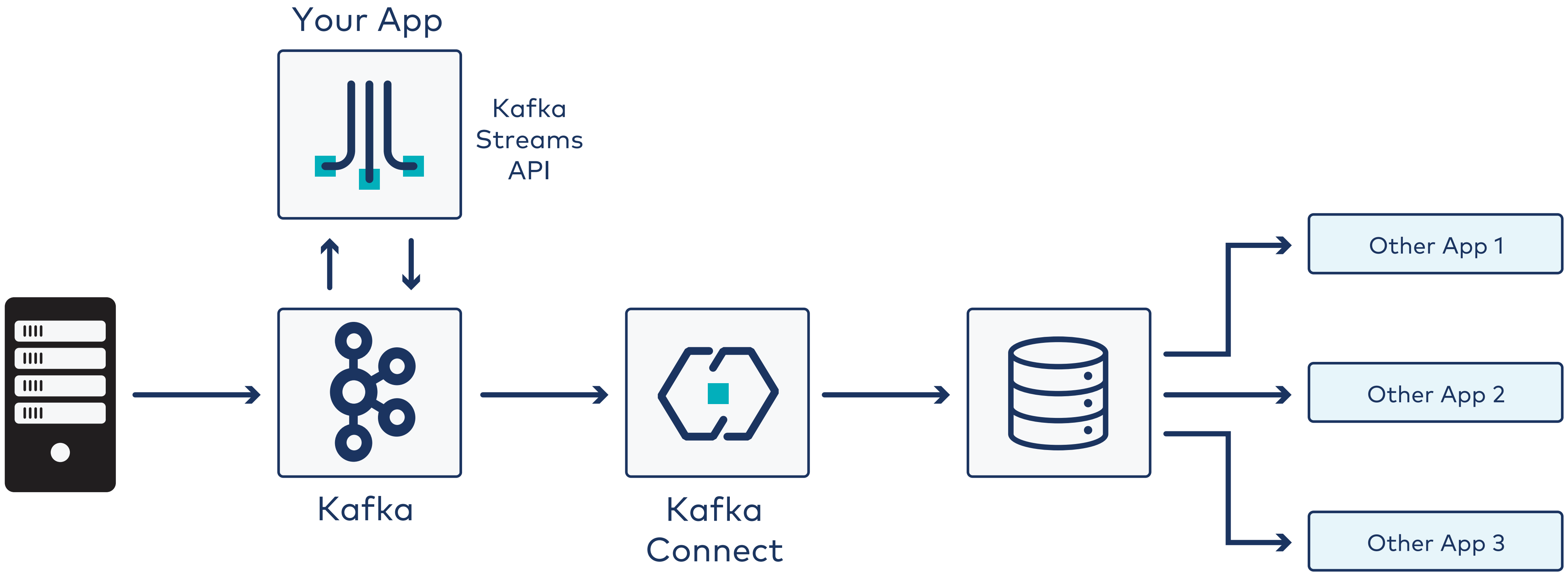
- Stateful transformations depend on state for processing inputs and producing outputs and **require a state store** associated with the stream processor.
- State stores are fault-tolerant.



KTable Memory Management

- You can specify the total memory (RAM) size that is used for an instance of a processing topology.
- Used for internal caching and compacting of records before they are written to state stores, or forwarded downstream to other nodes.
- Divided equally among the Kafka Stream threads of a topology.
- Each thread maintains a memory pool accessible by its tasks' processor nodes for caching.

Sharing Data



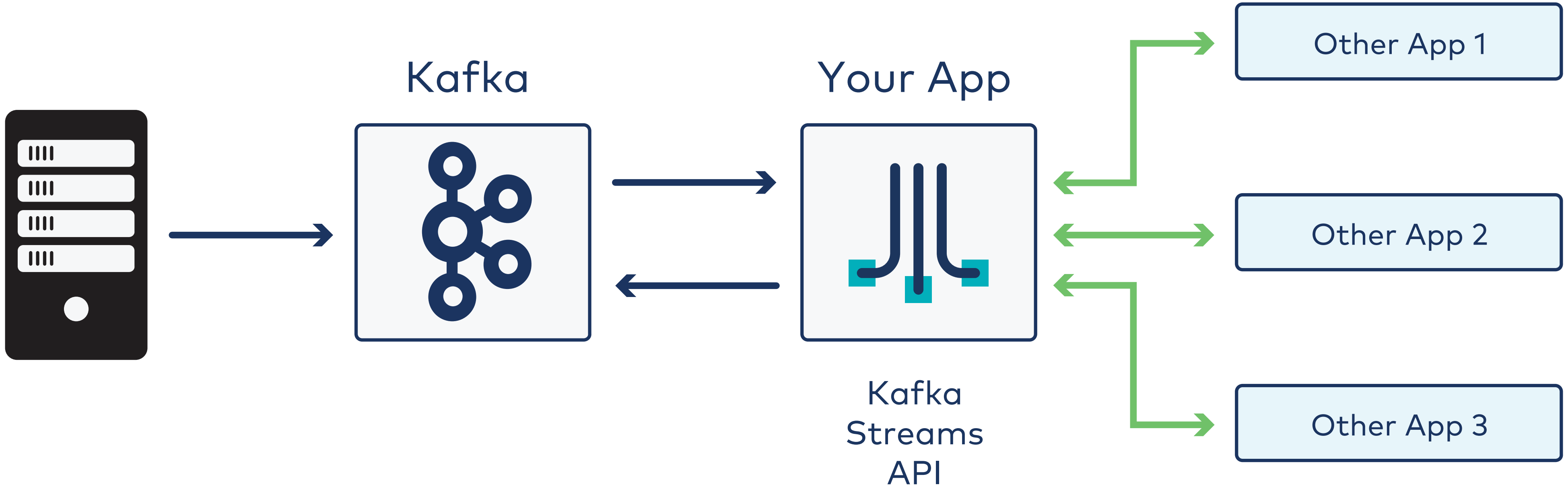
1 Capture business events in Kafka

2 Process events with Kafka Streams

3 Load processed data to external DBs and systems to share latest results

4 Other Apps query DB for latest results

Interactive Queries

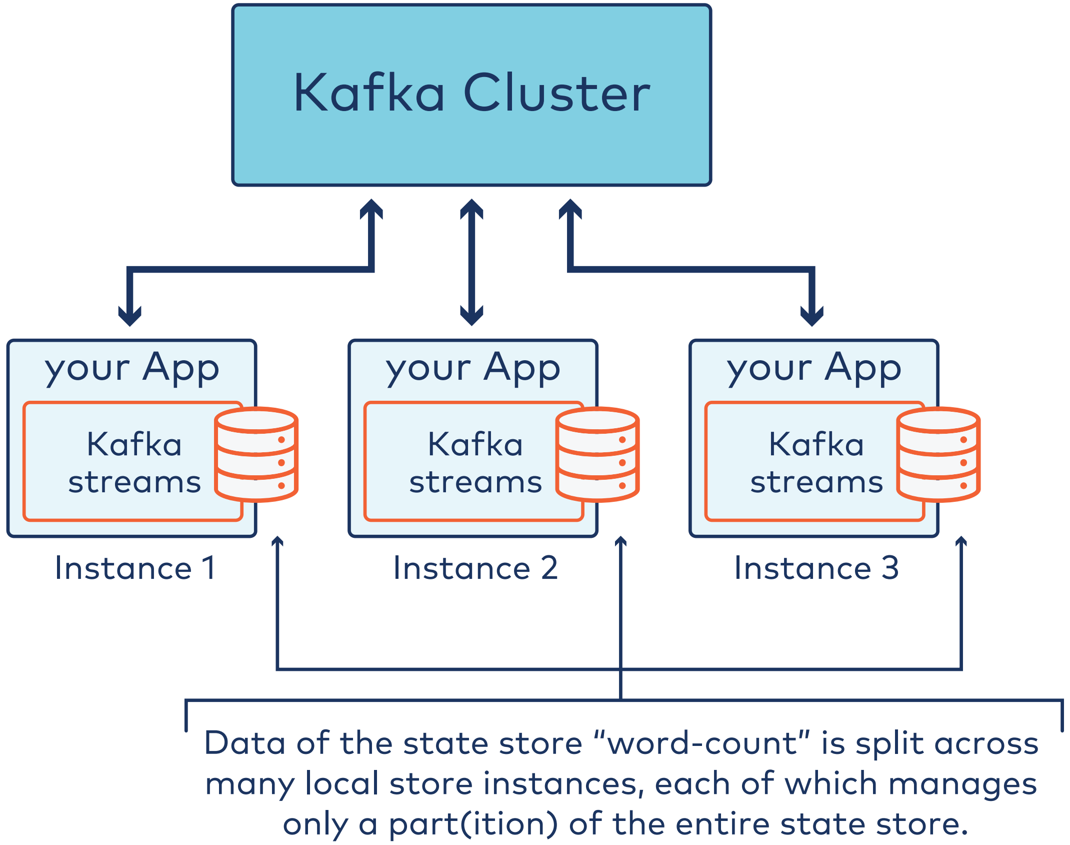


1 Capture business events in Kafka

2 Process events with Kafka Streams

3 With interactive queries, other Apps can directly query the latest results

Interactive Queries Explained



Data of the state store "word-count" is split across many local store instances, each of which manages only a part(ition) of the entire state store.

Querying state stores is always read-only

Stateful Processing and Advanced Operations Overview

CONFLUENT

Global Education

Agenda

This is a branch of our stream processing content on stateful processing and advanced operations. It is broken down into the following modules:

3. Time and Windowing
4. Aggregrations
5. Joins
6. Custom Processing

This branch assumes proficiency in concepts from the Starting with Stream Processing branch. Alternatively, students who have compeleted the Core branch and Additional Components of a Kafka Deployment branch of the Developer training will be mostly prepared for this content.

03: Time and Windowing

CONFLUENT

Global Education

Module Overview

This module contains three lessons:

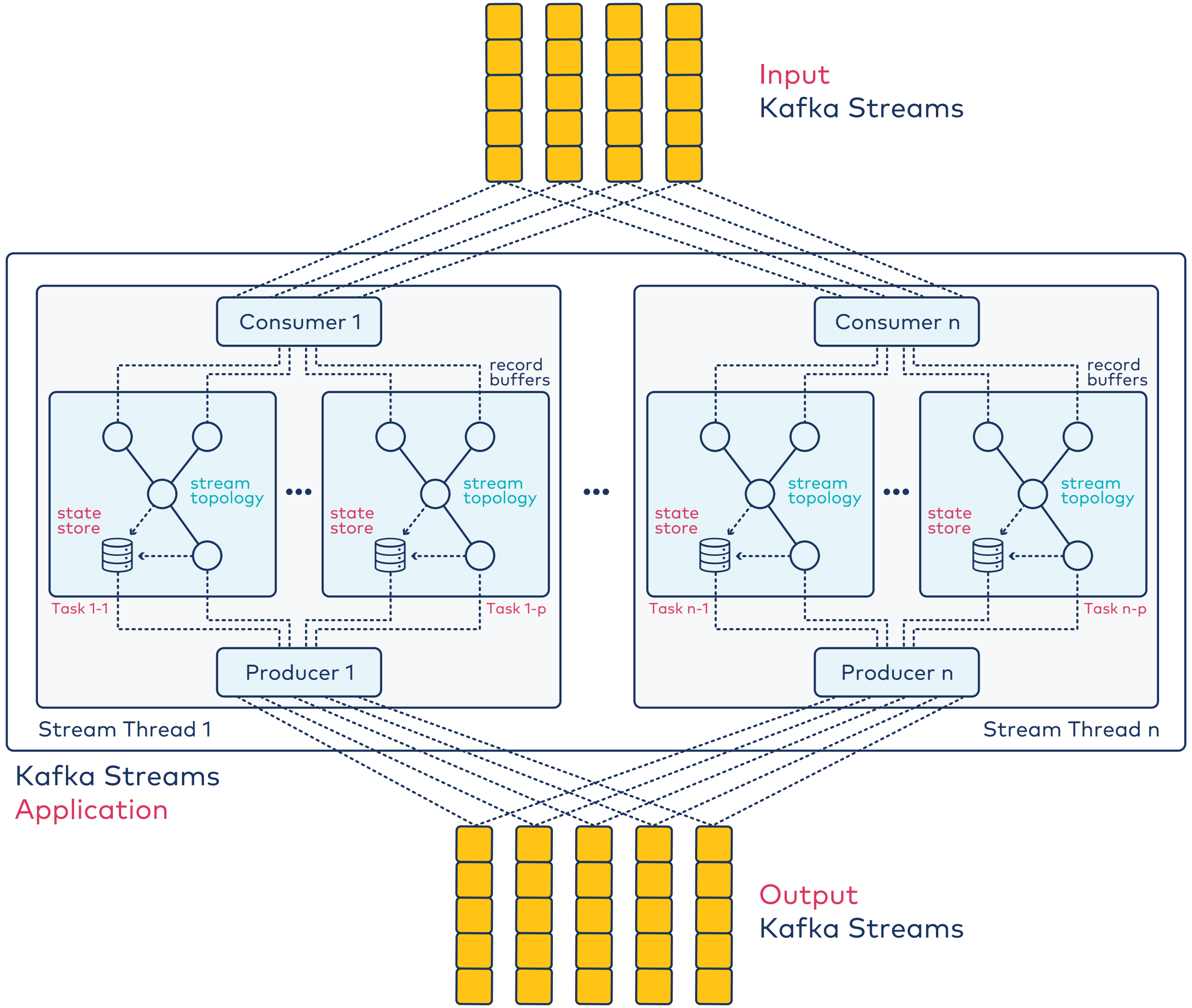
- How Does Time Work in Stream Processing?
- How Can You Divide up Streams into Time Windows?
- How Can You Make Windows Handle Late-Arriving Events and Limit Their Output?

Where this fits in:

- Hard Prerequisite: Working with Kafka Streams
- Recommended Follow-Up: Aggregations, Joins, and/or Custom Processing

Stateful Transformations Recap

- Stateful transformations depend on state for processing inputs and producing outputs.
- Each requires a state store associated with the stream processor.
- State stores are fault-tolerant.



03a: How Does Time Work in Stream Processing?

Description

The notion of time has a critical aspect in stream processing, and how it is modeled and integrated. When working with stream processing, it is important to understand the concept of time.

The Notion of Time

Event-time

The point in time when an event or data record occurred

Ingestion-time

The point in time when an event or data record is stored in a topic partition by a Kafka broker

Processing-time

The point in time when the event or data record happens to be processed by the stream processing application (that is, when the record is being consumed)

Timestamps

- Per-record timestamps describe the **progress of a stream** with regards to time (event time).
- Timestamp stores the the *event-time* of the application.
    - This differentiates with the *wall-clock-time*, which is when the application is actually executing.
- Event-time is also used to synchronize multiple input streams within the same application.
- Kafka Streams assigns a timestamp to every data record via timestamp extractors.

Timestamp Assignment

The way the timestamps are assigned depends on:

Action	Output record time
New output records are generated	Input record timestamps
Output records from aggregation	Latest input record

Timestamp Assignment for Specific Operations

For aggregations and joins, timestamps are computed using the following rules:

Operation	Output record time
Stateless	Same as input record
Aggregations	`max` timestamp across all records, per key
Joins (stream-table)	Same as input stream record
Joins (stream-stream, table-table)	`max(left.ts, right.ts)`

ℹ We'll discuss aggregations and joins in detail in the modules to come.

03b: How Can You Divide up Streams into Time Windows?

Description

Windowing lets you control how to group records that have the same key for stateful operations like aggregations or joins into windows. Windows are tracked per record key. Tumbling, hopping, and session are commonly used windows.

Types of Windows

Windowing allows you to group records by the same key for stateful operations.

Types of windows:

- Tumbling
- Hopping
- Sliding
- Session

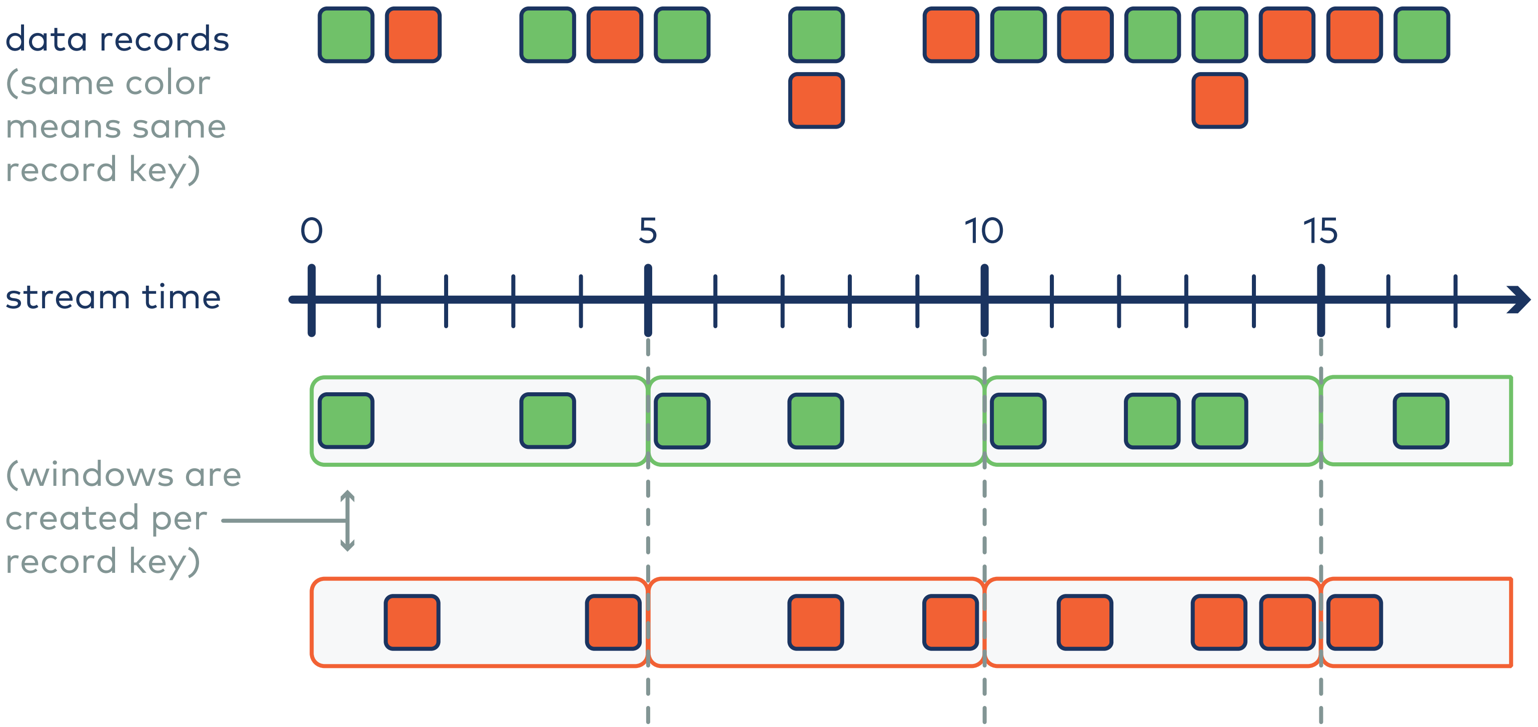
Differences Between the Window Types

Window name	Behavior	Short description
Tumbling	Time-based	Fixed-size, non-overlapping, gap-less windows
Hopping	Time-based	Fixed-size, overlapping windows
Sliding	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session	Session-based	Dynamically-sized, non-overlapping, data-driven windows

Tumbling Windows

- Fixed-size
- Non-overlapping, gap-less
- Defined by a single property: the window's size
- Aligned to the epoch
- Each window is inclusive of the lower bound and exclusive of the upper bound

A 5-min Tumbling Window


data records
(same color means same record key)
0
5
10
15
stream time
(windows are created per record key)

Tumbling Window Code Example

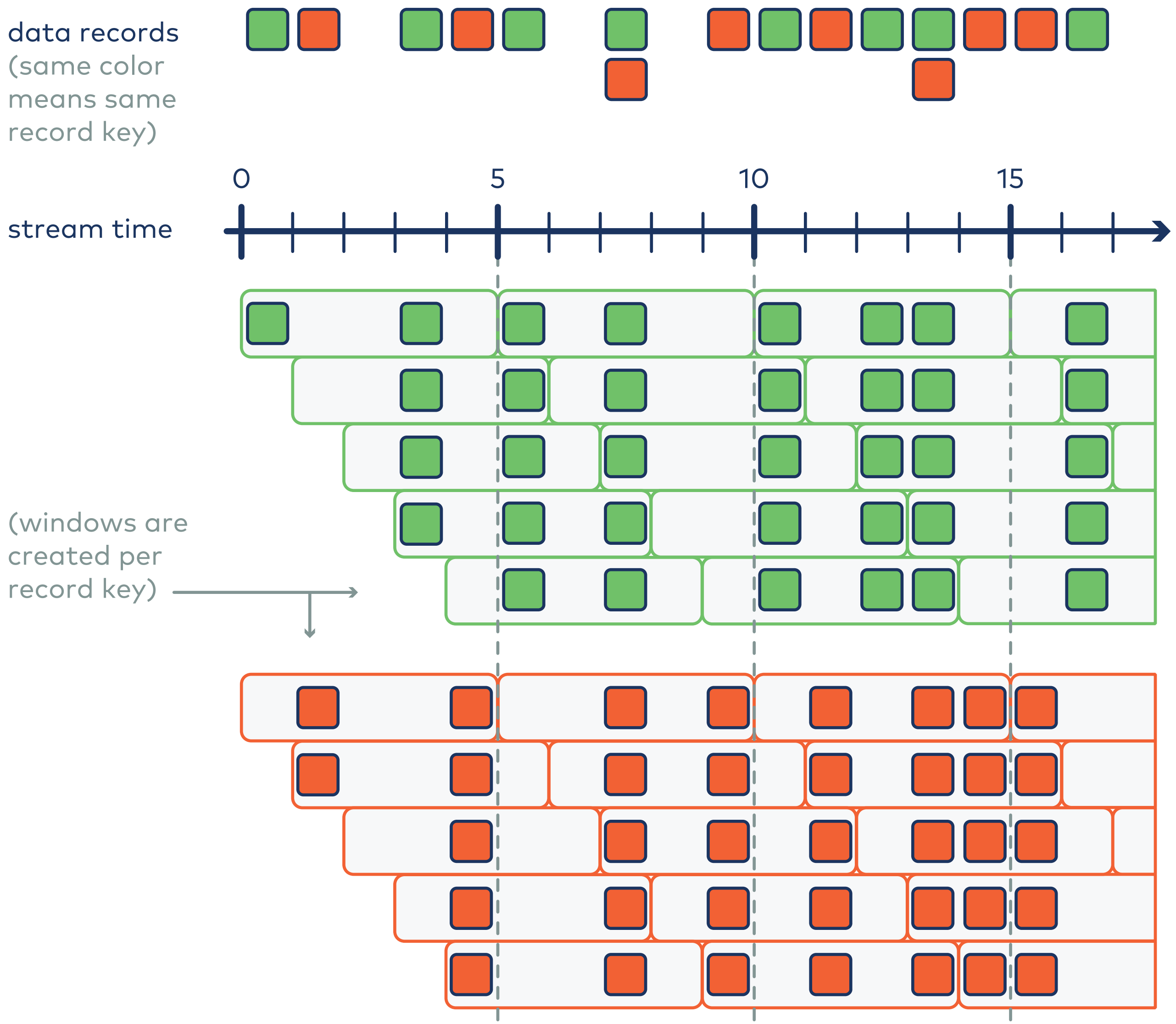
Kafka Streams

```
builder.<String, Rating>stream(ratingTopic)
            .map((key, rating) -> new KeyValue<>(rating.getTitle(), rating))
            .groupByKey()
            .windowedBy(TimeWindows.of(Duration.ofMinutes(10)))
            .count()
            .toStream()
            .map((Windowed<String> key, Long count) -> new KeyValue<>(key.key(), count.toString()))
            .to(ratingCountTopic, Produced.with(Serdes.String(), Serdes.String()));
```

Hopping Windows

- Fixed-sized
- Overlapping windows
- Defined by two properties: the window's size and its advance interval
- Aligned to the epoch
- Each window is inclusive of the lower bound and exclusive of the upper bound

A 5-min Hopping Window with a 1-min "hop"



data records (same color means same record key)
0
5
10
15
stream time
(windows are created per record key)

Hopping Window Code Example

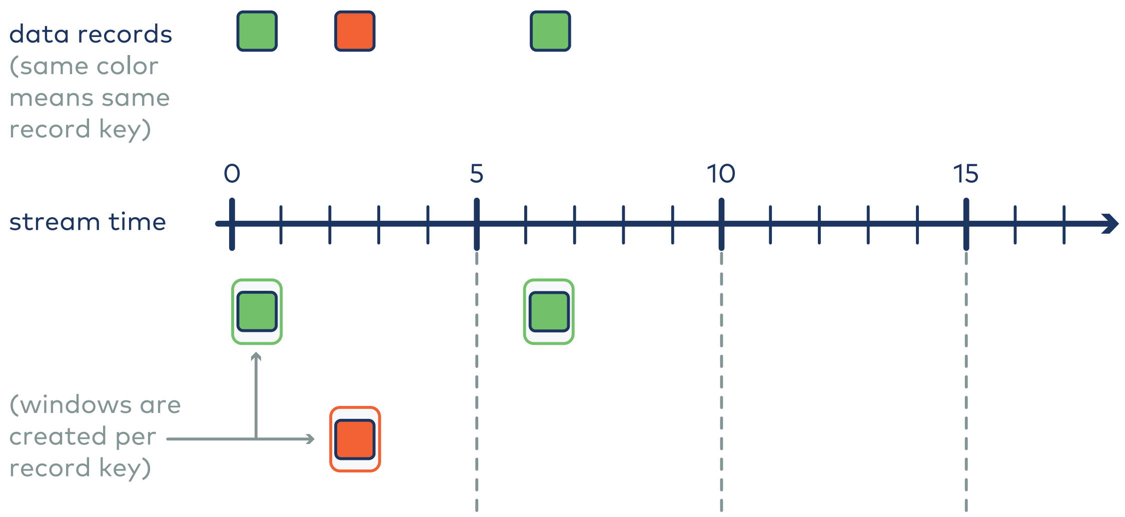
Kafka Streams

```
KStream<String, GenericRecord> pageViews = ...;

KTable<Windowed<String>, Long> windowedPageViewCounts;

windowedPageViewCounts = pageViews
                            .groupByKey(Grouped.with(Serdes.String(), genericAvroSerde))
                            .windowedBy(TimeWindows.of(Duration.ofMinutes(5)
                                                               .advanceBy(Duration.ofMinutes(1))))
                            .count()
```

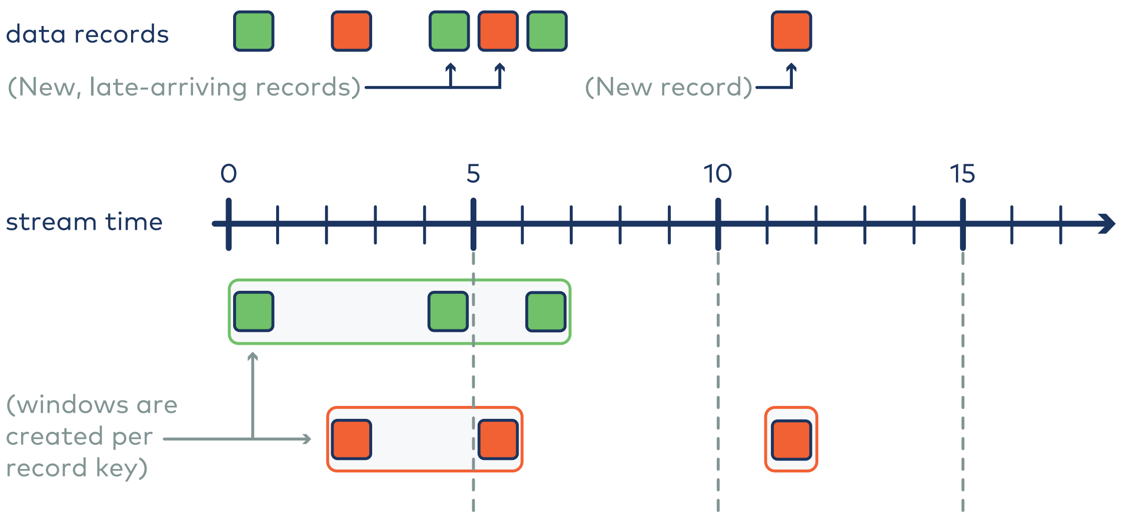
Session Windows

- Session windows are used to aggregate key-based events, sessions.
- All windows are tracked independently across keys, e.g., windows of different keys typically have different start and end times.
- Window sizes vary. Even windows for the same key typically have different sizes based on activity and idleness.

A Session Window with a 5-min inactivity gap


data records
(same color means same record key)
0
5
10
15
stream time
(windows are created per record key)

A Session Window with a 5-min inactivity gap


data records
(New, late-arriving records)
(New record)
0
5
10
15
stream time
(windows are created per record key)

Session Windows Code Example

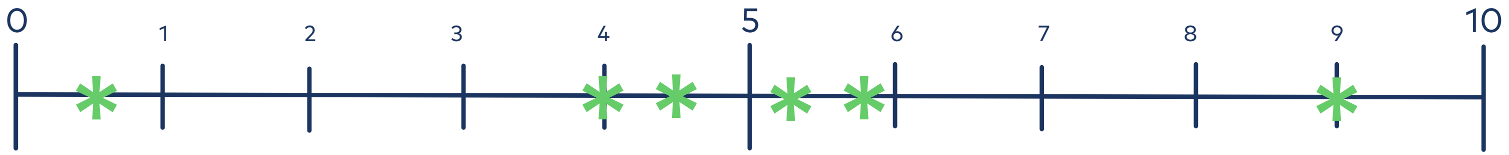
Kafka Streams

```
builder.stream(inputTopic,
               Consumed.with(Serdes.String(), clickSerde)
              .groupByKey()
              .windowedBy(SessionWindows.with(Duration.ofMinutes(5)))
              .count()
              .toStream()
              .map((windowedKey, count) -> {
                    String start = timeFormatter.format(windowedKey.window().startTime());
                    String end = timeFormatter.format(windowedKey.window().endTime());
                    String sessionInfo = String.format("Session info started: %s ended: " +
                        "%s with count %s", start, end, count);
                    return KeyValue.pair(windowedKey.key(),sessionInfo);
              })
```

Activity: Tumbling and Hopping Windows

Discuss with a partner or small group:

Consider this timeline, where each * represents an event:



Then:

1. Suppose we had tumbling windows of size 5. What events would be in each window?
2. Suppose instead we had hopping windows of size 5, advance by 3. What events would be in each window?

03c: How Can You Make Windows Handle Late-Arriving Events and Limit Their Output?

Description

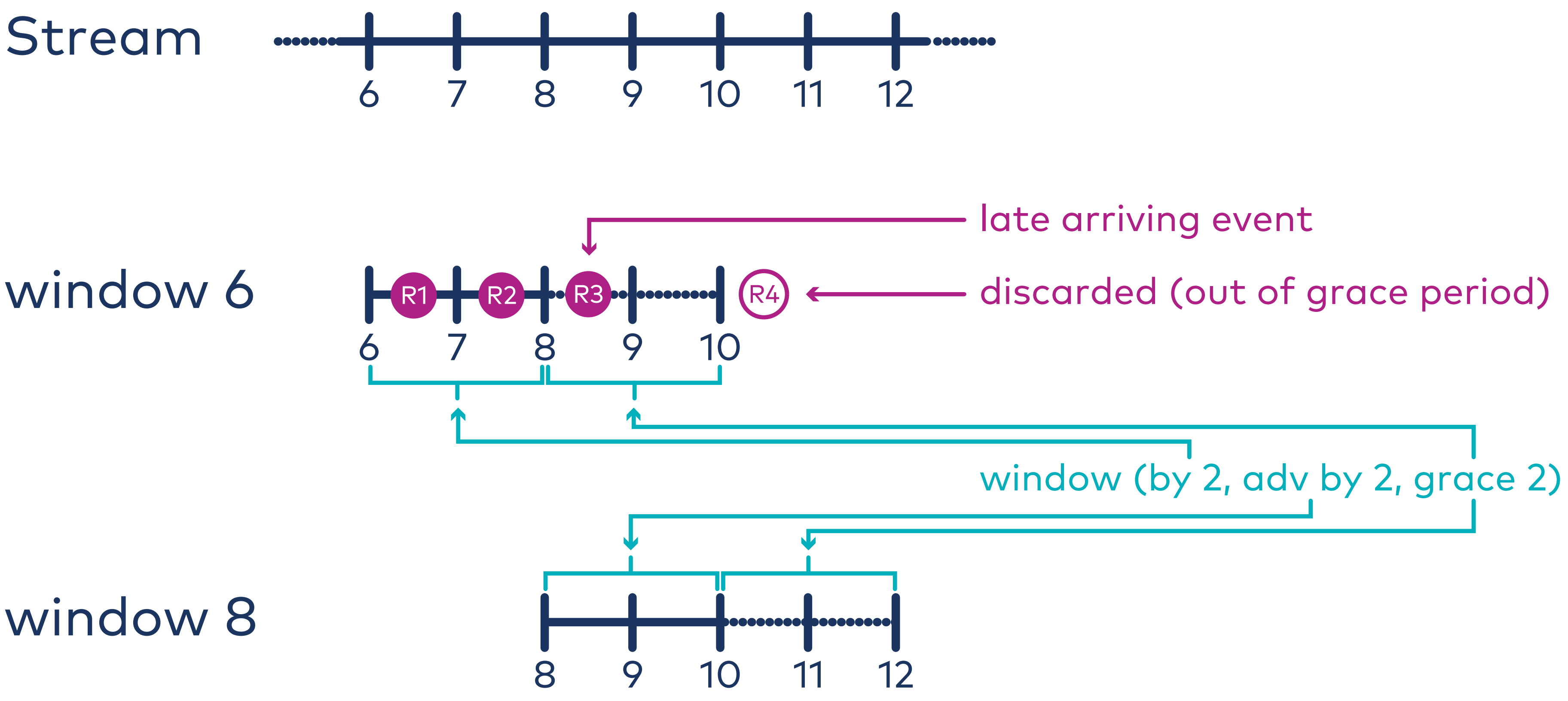
In Kafka Streams, late-arriving records can be handled by configuring a grace period. A grace period is an extension to the size of a window, and it allows events with timestamps greater than the window-end (but less than the window-end plus the grace period) to be included in the windowed calculation.

Grace Period

- An extension to the size of a window.
- Allows events with timestamps greater than the window-end (but less than the window-end plus the grace period) to be included in the windowed calculation.
- A record is discarded if it arrived after a grace period of a window is over, i.e., `record.ts` > `window-end-time + grace-period`.
- Tumbling, hopping, and sliding windows use the concept of grace period.
- The grace period supersedes retention time.

⚠ The default grace period is 24 hours.

Grace Period Explained



Suppress

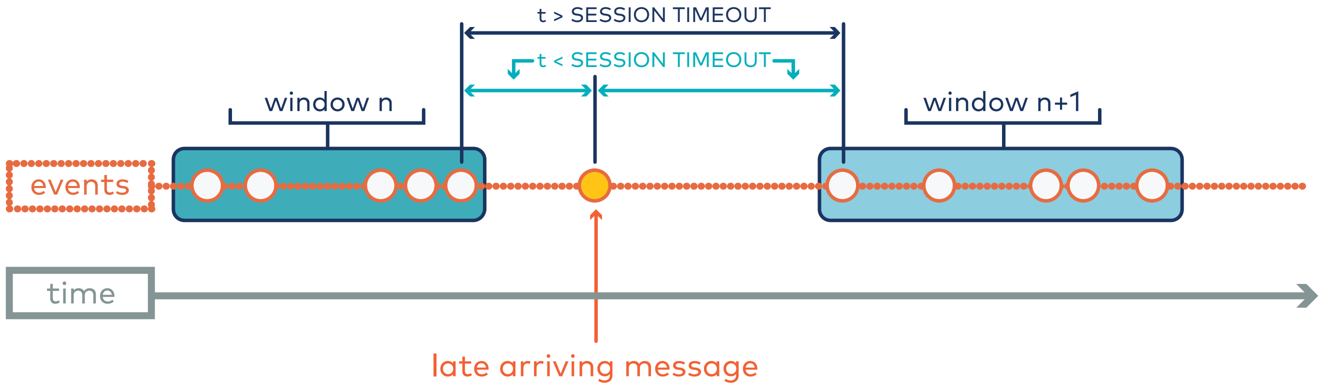
- `suppress` is an optional DSL operator.
- `suppress` offers strong guarantees about when exactly it forwards `KTable` updates downstream.
- The operator will suppress all the output results until window closes (`window size` + `grace period`).

Code Sample for **suppress**

```
KGroupedTable<String, String> groupedTable = ...;

groupedTable.count()
            .suppress(untilTimeLimit(Duration.ofMinutes(5),
                                     maxBytes(1_000_000L).emitEarlyWhenFull()))
            .toStream();
```

Late-Arriving Events with Session Windows

- We might have an event arrive later than its timestamp…
  - …and it belonged in an existing session window…
  - …but session windows were decided based on inactivity
- What to do? Possibilities:
  - Join that event to an existing session?
  - Merge existing sessions?



We will see this again with code in the next module!

CONFLUENT

Global Education

Module Overview

This module contains three lessons:

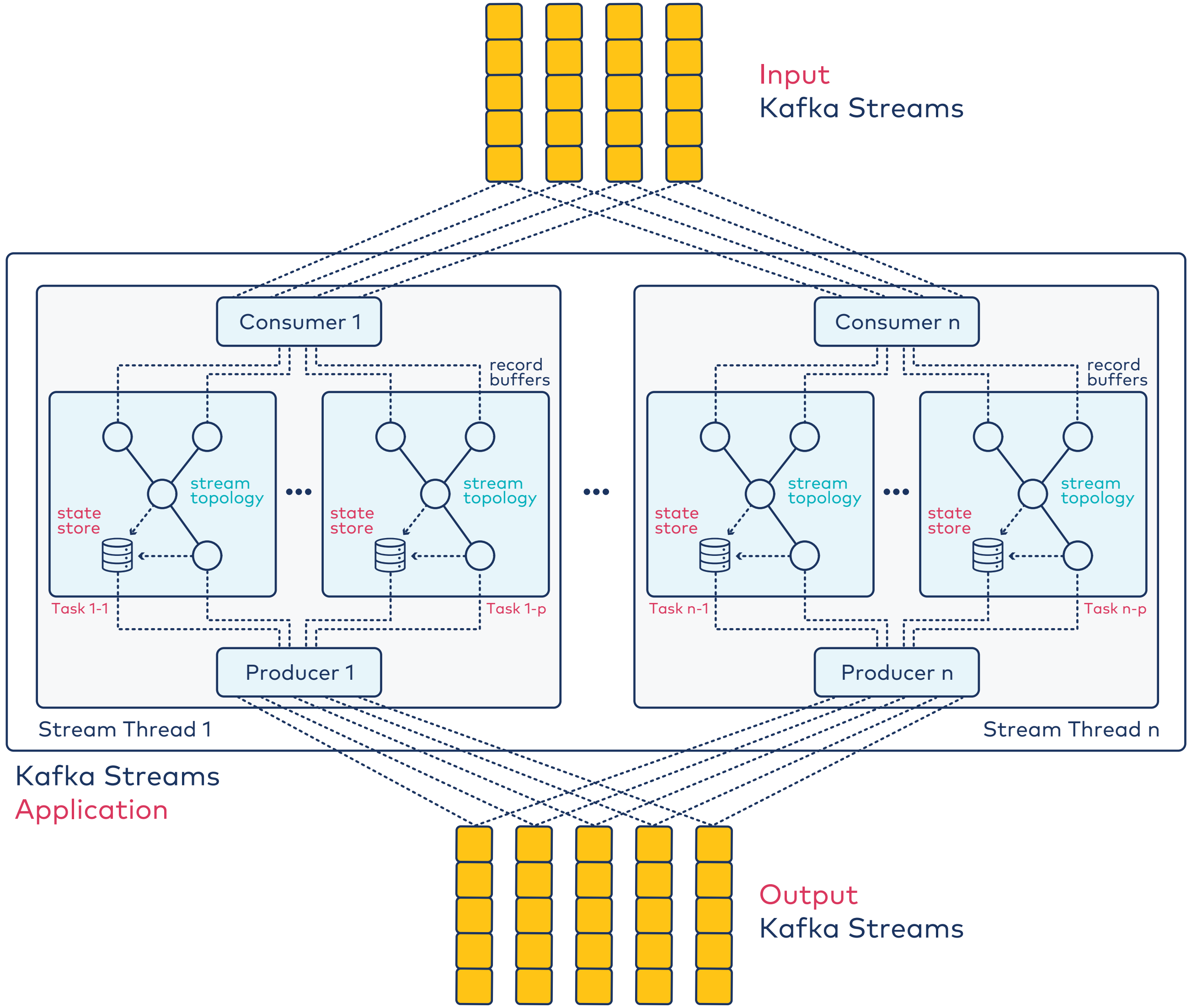
- How Do You Aggregate Data in Kafka Streams?
- What If You Want to Window Your Aggregations?

Where this fits in:

- Hard Prerequisite: Time and Windowing
- Recommended Follow-Up: Joins and/or Custom Processing

Stateful Transformations Recap

- Stateful transformations depend on state for processing inputs and producing outputs.
- Each requires a state store associated with the stream processor.
- State stores are fault-tolerant.



Aggregation Overview

Aggregations...

- Are key-based operations
- Are performed on windowed or non-windowed data
- Require a state store
- Use a windowing state store to collect the latest aggregation results per window behind the scenes

04a: How Do You Aggregate Data in Kafka Streams?

Description

Aggregations are key-based operations, meaning they always operate over records values of the same key. When aggregating a `KTable`, updates require us to subtract an old value before adding a new value as compared to stream aggregation, which does not have the notion of a subtractor.

Stateful Operations - toTable

Create Stream

```
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream(topicName);
```

Stream to Table

```
KTable<byte[], String> table = stream.toTable();
```

or

```
KTable<byte[], String> table =
    stream.toTable(Materialized.as("table-store-name"));
```

Kafka Streams Operations During Aggregation: Streams

You specify a few operations to define aggregations. These apply to streams:

Operation	When it Runs & What it Does	Stream?
Initializer	Upon a new bucket being seen Says what the initial aggregate value for the bucket is	✓
Adder	When a record is added to a bucket Says how the aggregate value for the bucket changes to reflect member joining	✓

Aggregating a KStream

```java
final StreamsBuilder builder = new StreamsBuilder();

// Create stream with default serdes and then group by key
KGroupedStream<String, String> groupedStream =
        builder.stream("input-topic").groupByKey();

KTable<String, Long> aggregatedStream = groupedStream.aggregate(
    () -> 0L, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
    Materialized
        .as("aggregated-stream-store")                 /* state store name */
        .withValueSerde(Serdes.Long());       /* serde for aggregate value */
)
```

Aggregating a KTable - Code (1)

```java
final StreamsBuilder builder = new StreamsBuilder()

/*
Create sales table using default serdes defined elsewhere.
Key: salesID String, value: SalesInfo with region and amount.
*/

KTable<String, SalesInfo> sales = builder.table("sales-topic");

// Group the sales table by region
KGroupedTable<String, Integer> groupedTable = sales
    .groupBy(
        (saleID, saleInfo) -> KeyValue.pair(saleInfo.region, saleInfo.amount),
        Serdes.String(), Serdes.Integer()
    );

//...
```

Aggregating a KTable - Code (2)

```java
// Aggregate value of the groupedTable, which is sales amount.
KTable<String, Integer> aggregated = groupedTable.aggregate(
    () -> 0, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
    (aggKey, oldValue, aggValue) -> aggValue - oldValue, /* subtractor */
    Materialized.<String, Integer, KeyValueStore<Bytes, byte[]>>
        as("aggregated-table-store" /* state store name */)
        .withKeySerde(Serdes.String()) /* key serde */
        .withValueSerde(Serdes.Integer())); /* serde for aggregate value */
```

Stateful Operations: count

count

```java
KTable<String, Long> aggregatedStream = groupedStream.count();

KTable<String, Long> aggregatedTable = groupedTable.count();
```

Stateful Operations: reduce

reduce

```
KTable<byte[], Integer> aggregatedStream = groupedStream.reduce(
    (aggValue, newValue) -> aggValue + newValue);

KTable<byte[], Integer> aggregatedTable = groupedTable.reduce(
    (aggValue, newValue) -> aggValue + newValue, /* adder */
    (aggValue, oldValue) -> aggValue - oldValue /* subtractor */);
```

Timestamps of Aggregate Records

One more detail to note:

- The timestamp of records generated by aggregations is the `max` timestamp across all records, per key

Lab: Windowing & Aggregation

Please work on **Lab 4a: Windowing & Aggregation**

Refer to the Exercise Guide

CONFLUENT

Global Education

Module Overview

This module contains two lessons:

- How Can You Join Data Across Stream Processing Entities?
- How Can You Join Data With Foreign Keys?

Where this fits in:

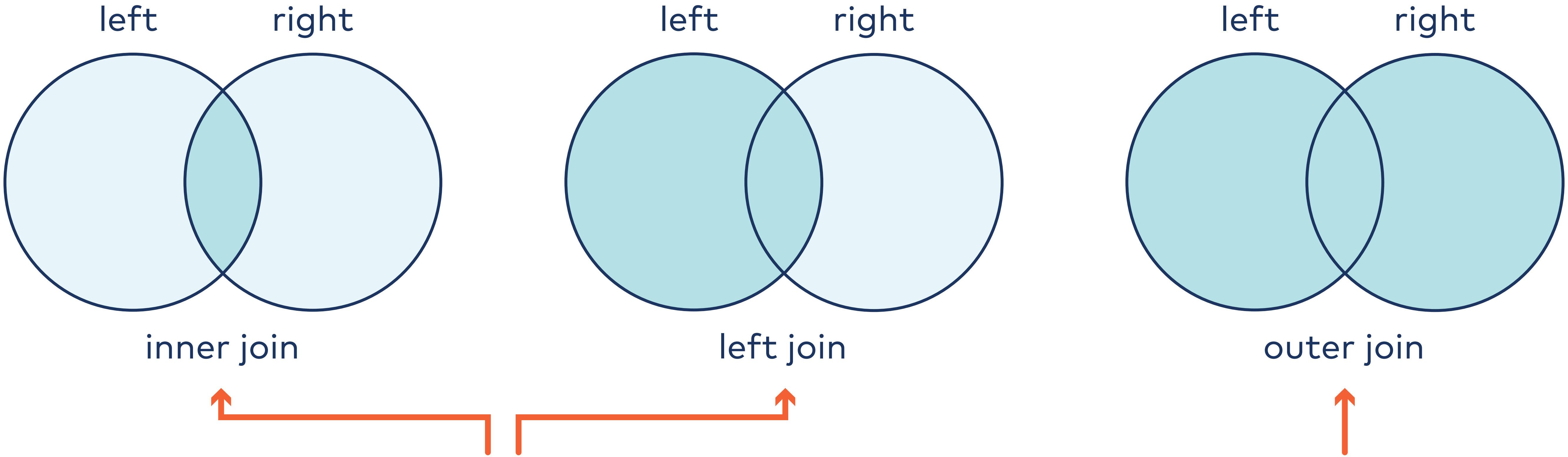
- Hard Prerequisite: Time and Windowing
- Recommended Follow-Up: Aggregations and/or Custom Processing

05a: How Can You Join Data Across Stream Processing Entities?

Description

Kafka Streams allow you to merge streams of events in real time. They support inner, left, and outer joins. For joining, input data must be co-partitioned. Join operations can be windowed or non-windowed.

Visualizing Joins in Kafka Streams



Stream — Stream
Table — Table
Stream — Table
KStream — GlobalKTable (supported in Kafka Streams only)

Stream — Stream
Table — Table

Join Requirements

Input data must be co-partitioned when joining.

The requirements for data co-partitioning are:

- The input topics of the join (left side and right side) must have the same number of partitions.
- All applications that write to the input topics must have the same partitioning strategy.
- The input topics use the same set of keys.

Joins using Kafka Streams

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
`KStream`, `KStream` → `KStream`	windowed	✓	✓	✓
`KTable`, `KTable` → `KTable`	*non*-windowed	✓	✓	✓
`KStream`, `KTable` → `KStream`	*non*-windowed	✓	✓	✗
`KStream`, `GlobalKTable` → `KStream`	*non*-windowed	✓	✓	✗
`KTable`, `GlobalKTable` → ?	✗	✗	✗	✗

⚠ Data needs to be co-partitioned

Join Operations in Code

Kafka Streams

```java
KStream<String, String> joined = left.join(
    right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(Duration.ofMinutes(5)),
    Joined.with(
        Serdes.String(),    /* key */
        Serdes.Long(),      /* left value */
        Serdes.Double())    /* right value */
);
```

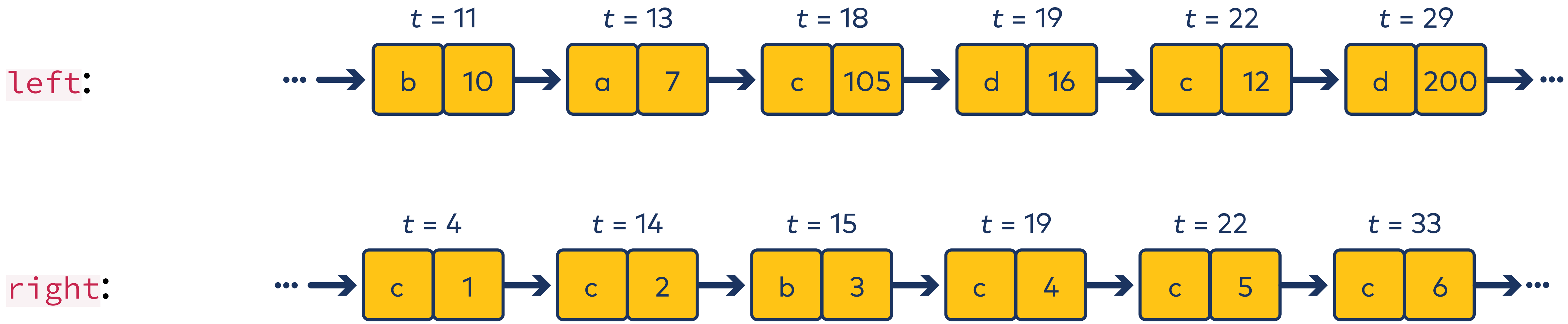
Timestamps of Joined Records

Here is how timestamps are decided for records created by joins:

Operation	Output record time
Joins (stream-table)	Same as input stream record
Joins (stream-stream, table-table)	`max(left.ts, right.ts)`

Activity: Joining Streams

Suppose you have two streams:



`left`: ··· → (t = 11) b 10 → (t = 13) a 7 → (t = 18) c 105 → (t = 19) d 16 → (t = 22) c 12 → (t = 29) d 200 → ···

`right`: ··· → (t = 4) c 1 → (t = 14) c 2 → (t = 15) b 3 → (t = 19) c 4 → (t = 22) c 5 → (t = 33) c 6 → ···

Suppose we are doing a join of `left` with `right` with a join window of time `5`.

Consider only the record at `t = 18` (key `c`, value `105`) in the left stream. Which records in the right stream will be joined with it?

Bonus Question (If you have time): What will the timestamps of the joined records be?

Lab: Joining Two Streams

Please work on **Lab 5a: Joining Two Streams**

Refer to the Exercise Guide

05b: How Can You Join Data With Foreign Keys?

Description

How do foreign-key joins work? Explore how a foreign key join relates a record from one table that contains a column that matches the primary key in another table.

What is a Foreign Key Join?

Primary-key joins

joins records from two tables where the key of both records are the same

Foreign-key join

joins a record from one table that contains a column that matches the primary key in another table

Foreign-Key Join in Kafka Streams

- `KTable`-`KTable` foreign-key joins are always *non-windowed* joins.
- There are two input tables: `left` and `right`:
    - A `foreign-key extractor` function is applied to the left record with a new intermediate record created and
    - It is used to lookup and `join` with the corresponding primary key on the right record.
- The output of the operation is a new `KTable`.
- `INNER` and `LEFT OUTER` joins are supported.
- No co-partitioning required.

Foreign-Key Join in Kafka Streams Example: Schemas

left input: Track Purchase

```
"fields": [
   {"name": "id", "type": "long"},
   {"name": "song_title", "type": "string"},
   {"name": "album_id", "type": "long"},
   {"name": "price", "type": "double"}
]
```

right input: Album

```
"fields": [
   {"name": "id", "type": "long"},
   {"name": "title", "type": "string"},
   {"name": "genre", "type": "string"},
   {"name": "artist", "type": "string"}
]
```

output: Music Interest

```
"fields": [
   {"name": "id", "type": "string"},
   {"name": "genre", "type": "string"},
   {"name": "artist", "type": "string"}
]
```

CONFLUENT

Global Education

Module Overview

This module contains two lessons:

- How Do You Leverage the Processor API for Low-Level Processing?

Where this fits in:

- Hard Prerequisite: Working with Kafka Streams
- Recommended Prerequisite: Time and Windowing
- Recommended Follow-Up: Aggregations and/or Joins

06a: How Do You Leverage the Processor API for Low-Level Processing?

Description

The Processor API is a low-level API which allows you to customize and implement special logic that is not available in the DSL. The Kafka Streams DSL is built on the Processor API.

What is the Processor API?

The Kafka Streams DSL is built on top of the Streams Processor API.

What is the Processor API (PAPI)?

- The PAPI allows you to:
  - Define a custom processor
  - Connect processors
  - Interact with the state stores

> ℹ The Processor API can be used to implement both stateless and stateful operations.

Where Can the PAPI be Useful?

- Customization
- Combining ease-of-use with full flexibility where needed

Defining Stream Processor and State Store

Stream Processor

define a stream processor by implementing the `Processor` interface which provides the `process()` API method; to implement stateful transformation, provide one or more state store to `Processor` or `Transformer`

State Stores

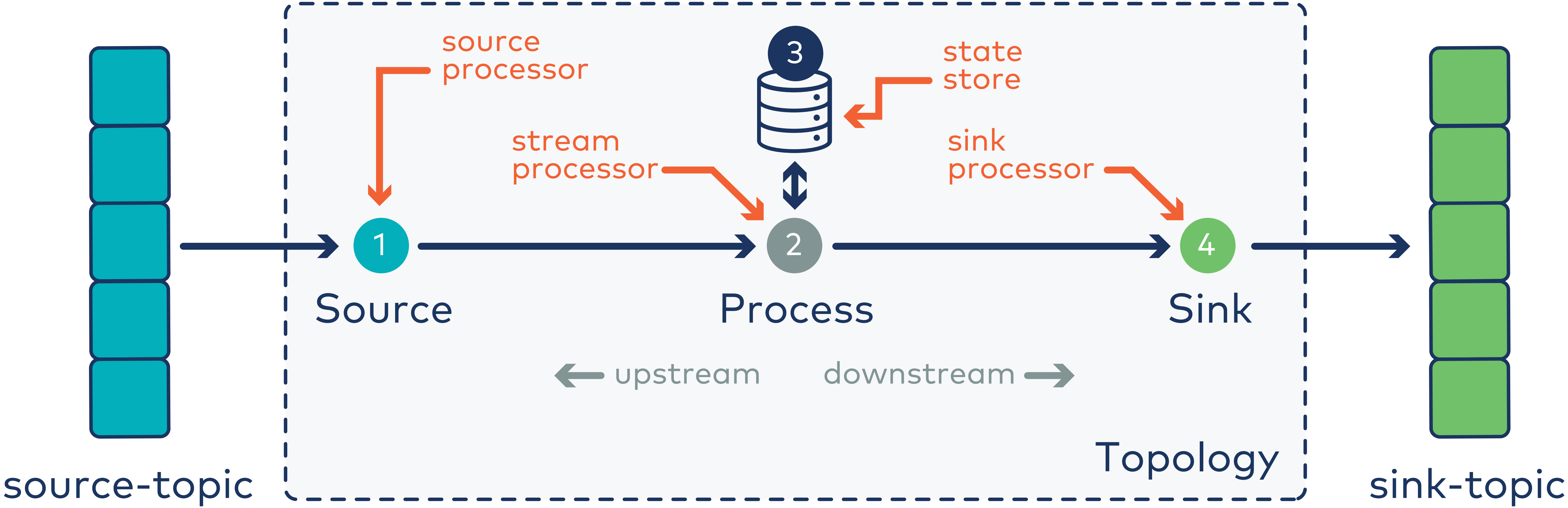
to implement a stateful transformation, provide one or more state store to `Processor` or `Transformer`

Processor API - Operations

Function	Returns
`process()`	Terminal operation - does not return `KStream`
`transform()`	Returns 0 or 1 output record
`transformValues()`	Returns 1 output record - cannot change the key
`flatTransform()`	Returns 0 or more output records
`flatTransformValues()`	Returns 0 or more output records - cannot change the key

Processor API

```
builder.addSource("Source", "source-topic")
       .addProcessor("Process", () -> WordCountProcessor(), "Source")
       .addStateStore(countStoreBuilder, "Process")
       .addSink("Sink", "sink-topic", "Process");
```



Example Integrating DSL with PAPI Transform Operation

```java
builder.stream("lines-topic", Consumed.with(Serdes.String(), Serdes.String()))
        .flatMapValues(line ->
            Arrays.asList(line.toLowerCase(Locale.getDefault()).split(" ")))
        .selectKey((k, word) -> word)
        .repartition(Repartitioned.with(Serdes.String(), Serdes.String()))
        .transform(WordCountTransformer::new, storeBuilder.name())
        .to("word-count-topic", Produced.with(Serdes.String(), Serdes.Long()));
```

Processor API - Implementing the Transformer

```
public class WordCountTransformer
    implements Transformer<String, String, KeyValue<String, Long>>
{
   // ...

   private KeyValueStore<String, Long> kvStore;

   // ...

   public void init(final ProcessorContext context) {...}
   public KeyValue<String, Long> transform(String word, String dummy) {...}
}
```

Lab: Using the Processor API

Please work on **Lab 6a: Using the Processor API**

Refer to the Exercise Guide

Stream Processing Operational Issues Overview

CONFLUENT

Global Education

Agenda

This is a branch of our stream processing content on operatonal issues related to streaming. It is broken down into the following modules:

7. Testing, Troubleshooting, and Monitoring
8. Deployment
9. Security

This branch assumes proficiency in concepts from the Starting with Stream Processing branch.

07: Testing, Monitoring, and Troubleshooting

CONFLUENT

Global Education

Module Overview

This module contains three lessons:

- How Should You Test Streaming Applications?
- How Can You Monitor Streaming Applications?
- How Should You Troubleshoot Streaming Applications?

Where this fits in:

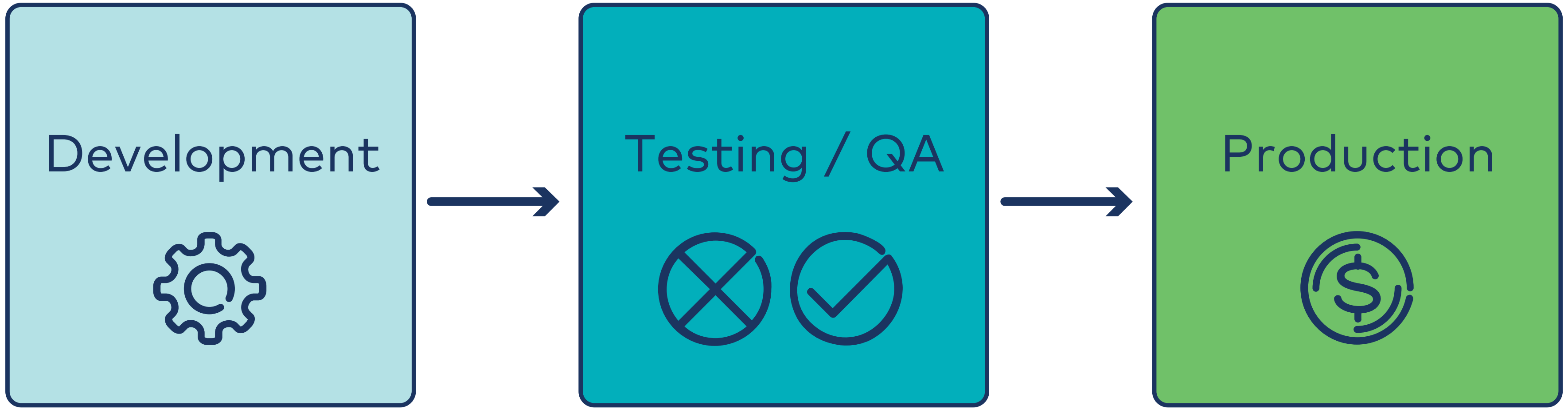
- Hard Prerequisite: Introduction to Kafka Streams
- Recommended Prerequisite: Working with Kafka Streams
- Recommended Follow-Up: Either of Deployment or Security

07a: How Should You Test Streaming Applications?

Description

How can you ensure your Kafka Streams application is working as expected? Learn different types of testing and how to test applications.

Why Test?



Development

Testing / QA

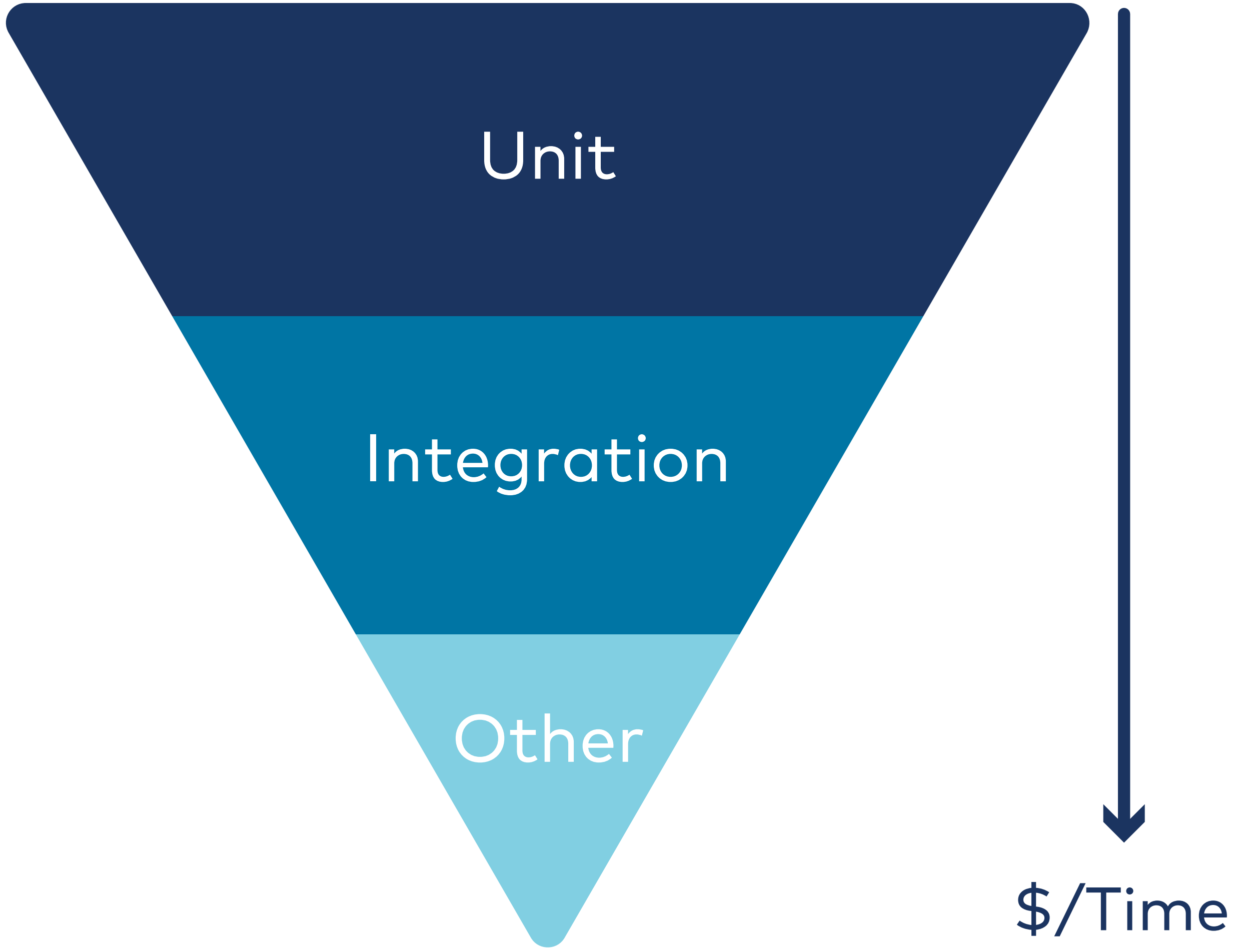
Production

Types of Testing

Unit testing: Testing actual behavior of one component against intended behavior of the API.

Integration testing: Several pieces are tested working in conjunction.

Other testing: Performance, soak, chaos testing: For optimizing your client applications, ensuring long-running code, and resilience against failures.



Generating the Test Data

Use data that is as close to realistic as possible for testing. You could:

- Write your own Kafka client application.
- Use the Datagen Connector.

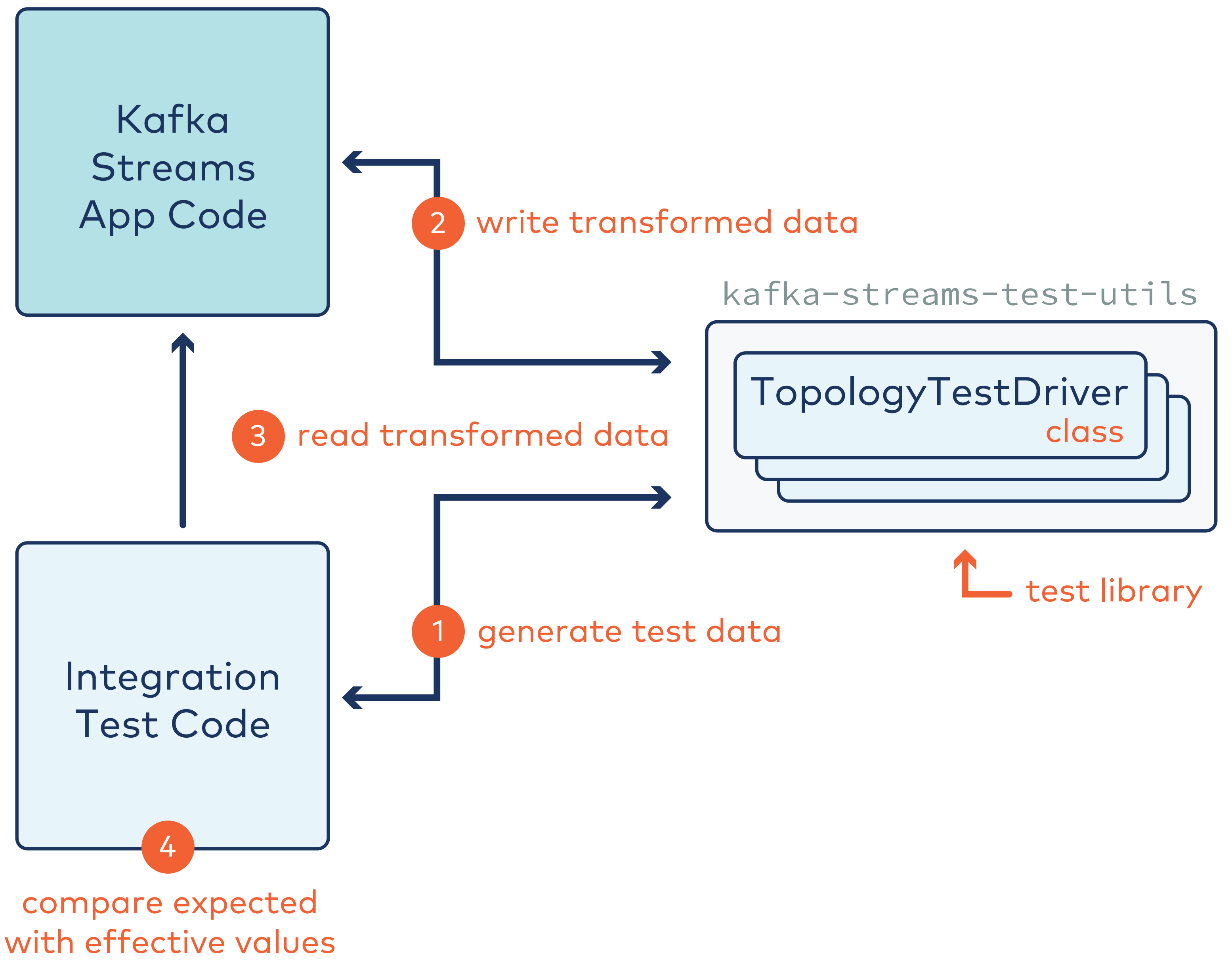
Test Utilities

	Kafka Streams	JVM Producer & Consumer	librdkafka Producer & Consumer
Unit Testing	TopologyTestDriver	MockProducer, MockConsumer	rdkafka_mock
Integration Testing	Testcontainers	Testcontainers	trivup
	Confluent Cloud	Confluent Cloud	Confluent Cloud

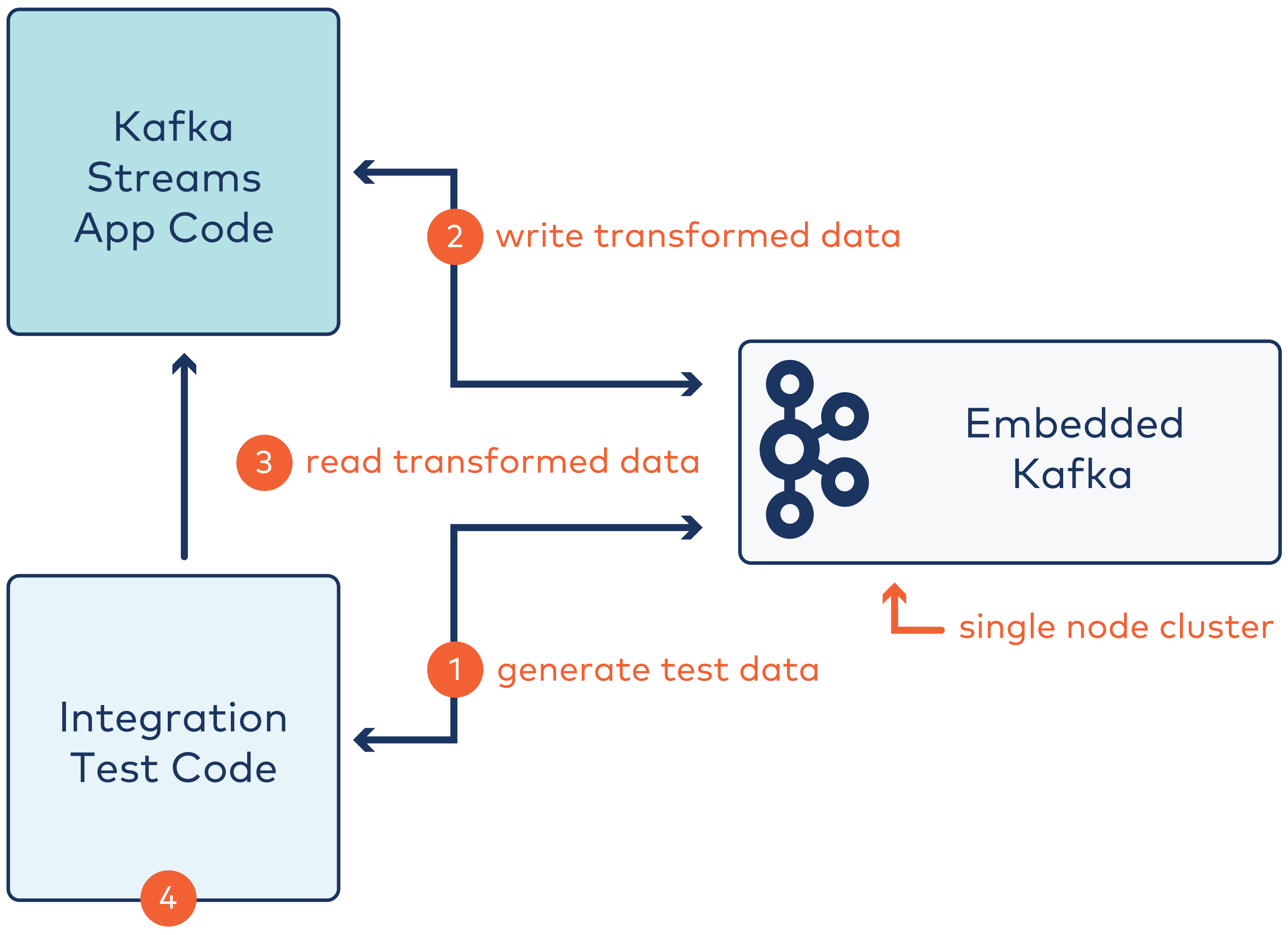
Unit Testing for the Processor API

- Test isolated class.
- Mock or stub all dependencies.
- Mock specific to Kafka Streams apps using the PAPI:
    - `MockProcessorContext` in `kafka-streams-test-utils`

Integration Tests - Test Driver



Integration Tests - Embedded Kafka



2 write transformed data

3 read transformed data

1 generate test data

single node cluster

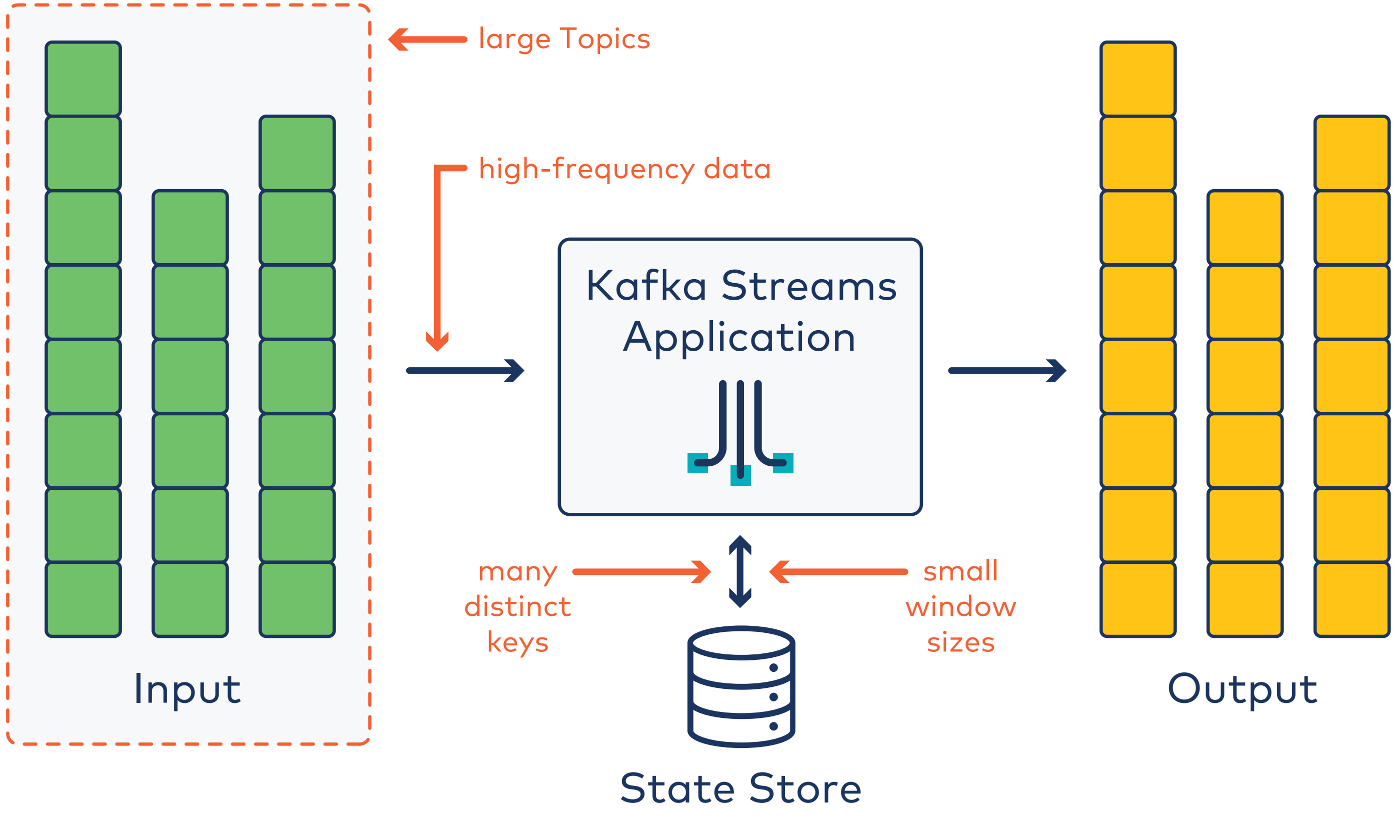
4 compare expected with effective values

Other Testing

- Performance testing
- Soak testing
- Chaos testing

Benchmark with Apache Kafka command-line tools like

- `kafka-producer-perf-test`
- `kafka-consumer-perf-test`



Lab: Integration Tests Using Embedded Kafka

Please work on **Lab 7a: Integration Tests Using Embedded Kafka**

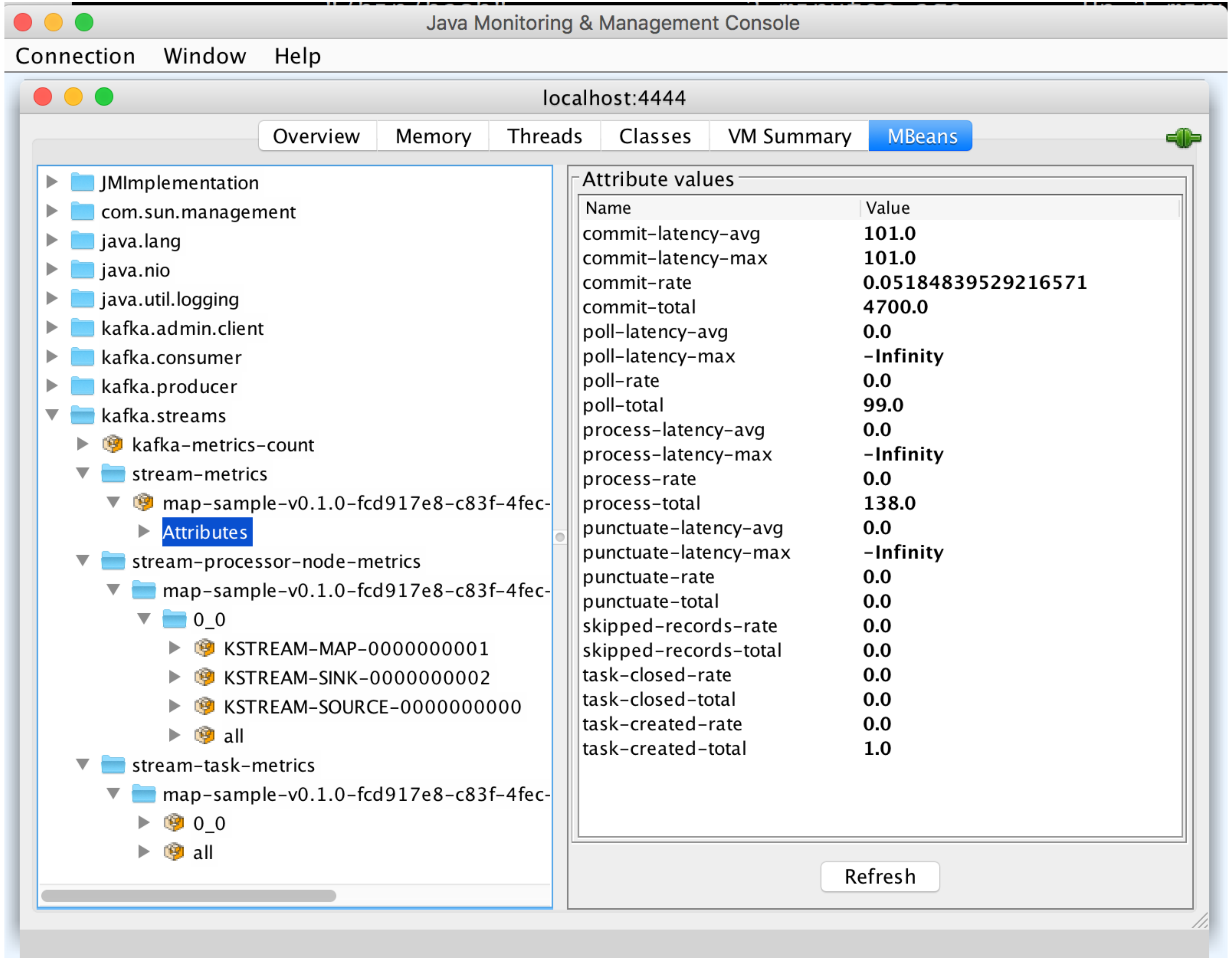
Refer to the Exercise Guide

07b: How Can You Monitor Streaming Applications?

Description

Once a Kafka Streams application runs in production, monitoring it is of the utmost importance. The Kafka Streams library reports a variety of metrics through JMX. Confluent Control Center is one of the ideal tools to use to monitor.

Using JMX-Based Monitoring

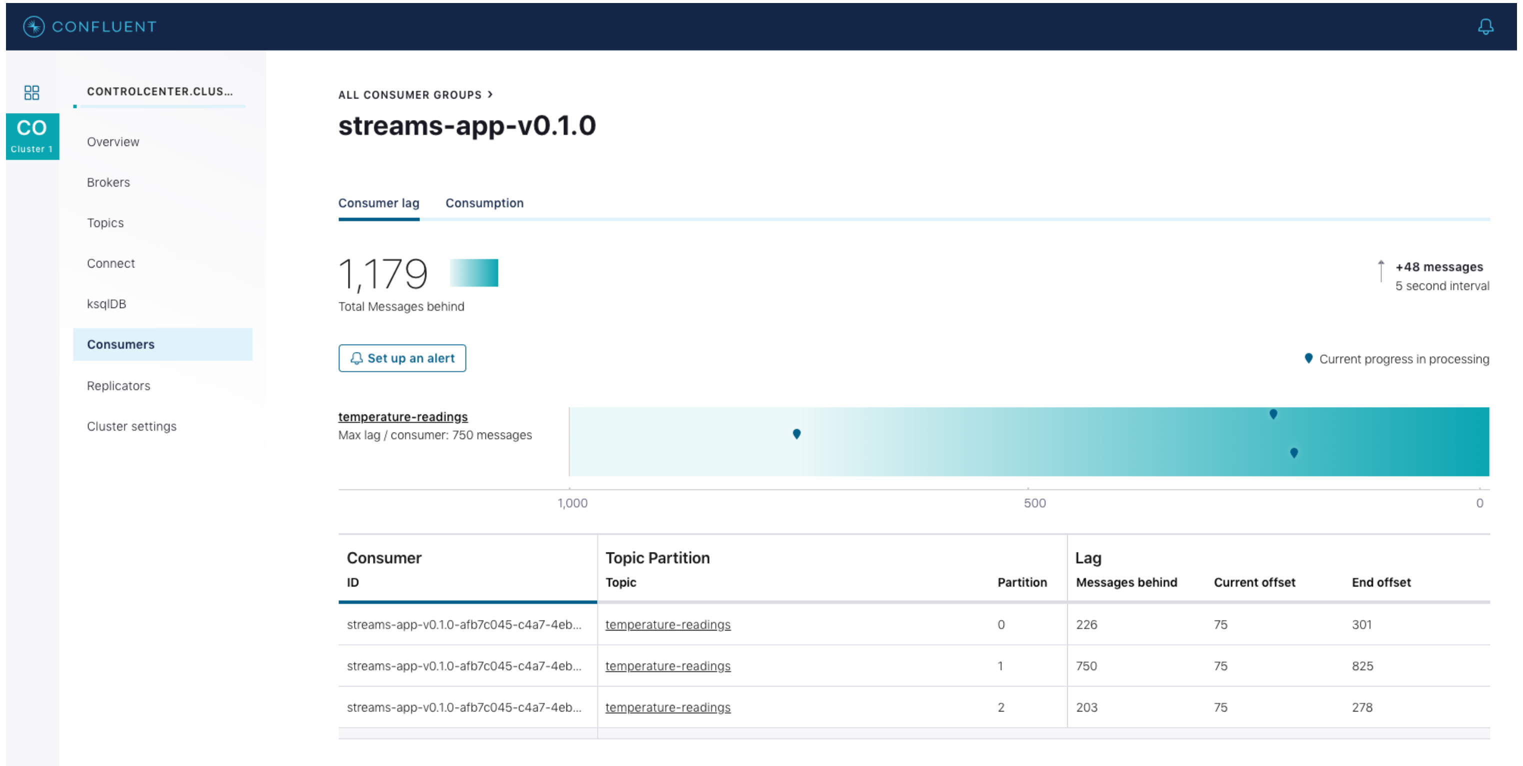


```
▶ JMImplementation
▶ com.sun.management
▶ java.lang
▶ java.nio
▶ java.util.logging
▶ kafka.admin.client
▶ kafka.consumer
▶ kafka.producer
▼ kafka.streams
    ▶ kafka-metrics-count
    ▼ stream-metrics
        ▼ map-sample-v0.1.0-fcd917e8-c83f-4fec-
            ▶ Attributes
    ▼ stream-processor-node-metrics
        ▼ map-sample-v0.1.0-fcd917e8-c83f-4fec-
            ▼ 0_0
                ▶ KSTREAM-MAP-0000000001
                ▶ KSTREAM-SINK-0000000002
                ▶ KSTREAM-SOURCE-0000000000
                ▶ all
    ▼ stream-task-metrics
        ▼ map-sample-v0.1.0-fcd917e8-c83f-4fec-
            ▶ 0_0
            ▶ all
```

Attribute values

Name	Value
commit-latency-avg	101.0
commit-latency-max	101.0
commit-rate	0.05184839529216571
commit-total	4700.0
poll-latency-avg	0.0
poll-latency-max	-Infinity
poll-rate	0.0
poll-total	99.0
process-latency-avg	0.0
process-latency-max	-Infinity
process-rate	0.0
process-total	138.0
punctuate-latency-avg	0.0
punctuate-latency-max	-Infinity
punctuate-rate	0.0
punctuate-total	0.0
skipped-records-rate	0.0
skipped-records-total	0.0
task-closed-rate	0.0
task-closed-total	0.0
task-created-rate	0.0
task-created-total	1.0

Confluent Control Center - Monitoring Interceptors



streams-app-v0.1.0

1,179
Total Messages behind

+48 messages
5 second interval

temperature-readings
Max lag / consumer: 750 messages

Consumer	Topic Partition		Lag		
ID	Topic	Partition	Messages behind	Current offset	End offset
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	0	226	75	301
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	1	750	75	825
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	2	203	75	278

- Set `producer.interceptor.classes` equal to:

  `io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor`

- Set `consumer.interceptor.classes` equal to:

  `io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor`

Kafka Streams Metrics Configurations

Parameter Name	Description	Default Value
`metric.reporters`	A list of classes to use as metrics reporters	the empty list
`metrics.num.samples`	The number of samples maintained to compute metrics	2
`metrics.recording.level`	The highest recording level for metrics	`INFO`
`metrics.sample.window.ms`	The window of time a metrics sample is computed over	30000 milliseconds

Kafka Streams Metrics

Category	Log level	MBean
Client Metrics	`info`	`kafka.streams:type=stream-metrics,client-id=[clientId]`
Thread Metrics	`info`	`kafka.streams:type=stream-thread-metrics,thread-id=[threadId]`
Task Metrics	`debug`	`kafka.streams:type=stream-task-metrics,thread-id=[threadId],task-id=[taskId]`
Processor Node Metrics	`debug`	`kafka.streams:type=stream-processor-node-metrics,thread-id=[threadId],task-id=[taskId],processor-node-id=[processorNodeId]`
State Store Metrics	`debug`	`kafka.streams:type=stream-state-metrics,thread-id=[threadId],task-id=[taskId],[storeType]-id=[storeName]`
RocksDB Metrics	`debug`	`kafka.streams:type=stream-state-metrics,thread-id=[threadId],task-id=[taskId],[storeType]-id=[storeName]`
Record Cache Metrics	`debug`	`kafka.streams:type=stream-record-cache-metrics,thread-id=[threadId],task-id=[taskId],record-cache-id=[storeName]`

Lab: Using JConsole to Monitor a Streams App

Please work on **Lab 7b: Using JConsole to Monitor a Streams App**

Refer to the Exercise Guide

07c: How Should You Troubleshoot Streaming Applications?

Description

Kafka streams errors are categorized in 3 broad categories: during data consumption from Kafka, while transforming or enriching data, and when producing the processed data back to Kafka. Kafka Streams applications can be reset and forced to reprocess it data by using the application reset tool.

Kafka Streams Application - Viewing a Topology

Streams topologies can become quite complex.

```
    TopologyDescription description = topology.describe();
    System.out.println(description);
```

```
  Sub-topology: 0
    Processor: KSTREAM-FILTER-0000000005(stores: []) --> KSTREAM-SINK-0000000004
                                                      <-- KSTREAM-KEY-SELECT-0000000002
    Processor: KSTREAM-KEY-SELECT-0000000002(stores: []) --> KSTREAM-FILTER-0000000005
                                                      <-- KSTREAM-FLATMAPVALUES-0000000001
...
  Sub-topology: 1
    Source: KSTREAM-SOURCE-0000000006(topics: Counts-repartition) --> KSTREAM-AGGREGATE-0000000003
    Processor: KTABLE-TOSTREAM-0000000007(stores: []) --> KSTREAM-SINK-0000000008
                                                      <-- KSTREAM-AGGREGATE-0000000003
    Sink: KSTREAM-SINK-0000000008(topic: outputTopic) <-- KTABLE-TOSTREAM-0000000007
...
```

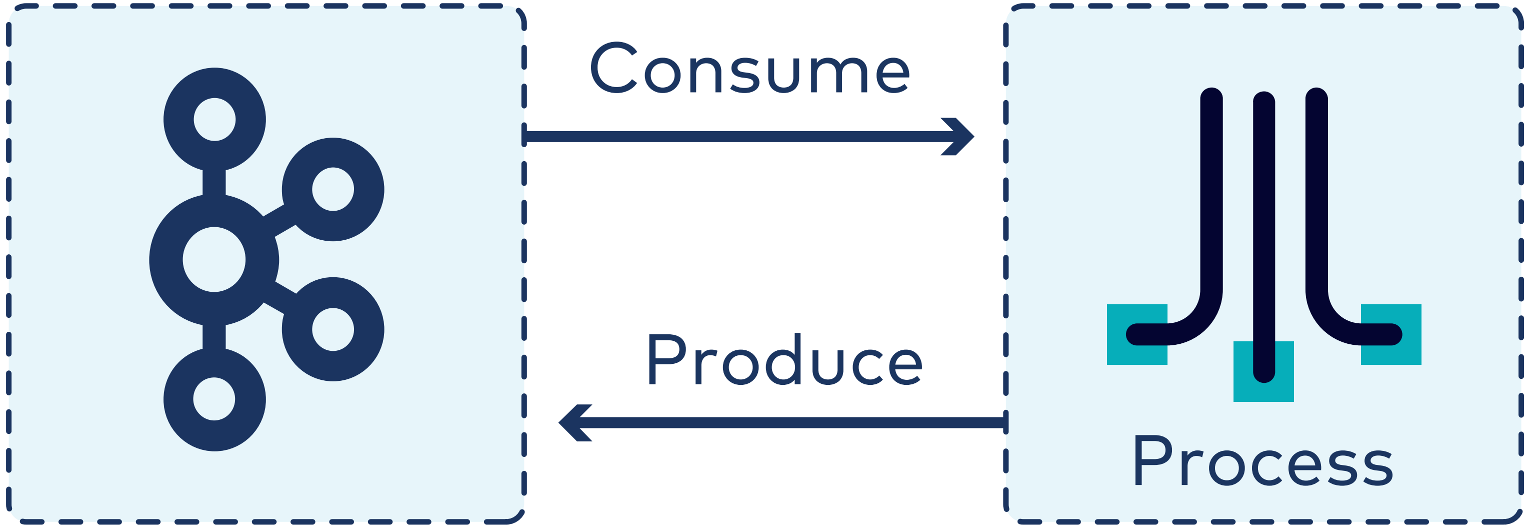
❗ Use the Kafka Streams Topology Visualizer to analyze the topology

Kafka Streams Application - Naming Processors

```
KStream<String,String> stream =
builder.stream("input", Consumed.as("Customer_transactions_input_topic"));
stream.filter((k,v) -> !v.equals("invalid_txn"), Named.as("filter_out_invalid_txns"))
      .mapValues((v) -> v.substring(0,5), Named.as("Map_values_to_first_6_characters"))
      .to("output", Produced.as("Mapped_transactions_output_topic"));
...
```

```
Sub-topology: 0
 Source: Customer_transactions_input_topic (topics: [input])
   --> filter_out_invalid_txns
 Processor: filter_out_invalid_txns (stores: [])
   --> Map_values_to_first_6_characters
   <-- Customer_transactions_input_topic
  ...
 Sink: Mapped_transactions_output_topic (topic: output)
   <-- Map_values_to_first_6_characters
```

Kafka Streams - Error Categories



Consumption Errors - Poison Pill Record

This gives a Serializer/Deserializer error.

Use `org.apache.kafka.streams.errors.DeserializationExceptionHandler` interface to customize how to handle those poison pills. The options are:

- Fail-fast - `LogAndFailExceptionHandler`
- Log and skip - `LogAndContinueExceptionHandler`
- Quarantine corrupted records (dead letter queue)
- Implement a custom serde

Processing Errors

- Exception related to the logic which will eventually shut down the application. For example, `ProducerFencedException`
- Use the `StreamsUncaughtExceptionHandler` interface.

Producing Errors

- Errors that occurred while producing the data back to Kafka topic. For example, `RecordTooLargeException`.
- Use `ProductionExceptionHandler` interface.

Interactive Queries Related Errors

- Handling `InvalidStateStoreException`:

```
org.apache.kafka.streams.errors.InvalidStateStoreException:
    the state store, my-key-value-store, may have migrated
      to another instance.
    at org.apache.kafka.streams.state.internals
       .StreamThreadStateStoreProvider
       .stores(StreamThreadStateStoreProvider.java:49)
    at org.apache.kafka.streams.state.internals
       .QueryableStoreProvider.getStore(QueryableStoreProvider.java:55)
    at org.apache.kafka.streams.KafkaStreams
       .store(KafkaStreams.java:699)
```

- Reasons could be
  - The local `KafkaStreams` instance is not yet ready.
  - The state store was just migrated to another instance.

(Output formatted to fit slide)

Interactive Queries Related Errors - Prevention

Guard against `InvalidStateStoreException` when calling `KafkaStreams#store()`

```java
public static <T> T waitUntilStoreIsQueryable(final String storeName,
                                              final QueryableStoreType<T> queryableStoreType,
                                              final KafkaStreams streams) throws InterruptedException
{
  while (true)
  {
    try
    {
      return streams.store(storeName, queryableStoreType);
    }
    catch (InvalidStateStoreException ignored)  // store not yet ready for querying
    {
      Thread.sleep(100);
    }
  }
}
```

Invalid Timestamp Exception

You could get an exception similar to this:

```
Exception in thread "StreamThread-1"
   org.apache.kafka.streams.errors.StreamsException:
      Input record {...} has invalid (negative) timestamp.
      Possibly because a pre-0.10 producer client was used to write
       this record to Kafka without embedding a timestamp,
        or because the input topic was created before upgrading
        the Kafka cluster to 0.10+.
        Use a different TimestampExtractor to process this data.
  at
  org.apache.kafka.streams.processor.
  FailOnInvalidTimestamp.onInvalidTimestamp(FailOnInvalidTimestamp.java:62)
```

(Output formatted to fit slide)

Kafka Streams Application Reset Tool

The application reset tool **does** the following based on the type of topic:

Topic Type	Action
Input topics	Reset offsets to specified position
Intermediate topics	Skip committed consumer offset to the end of the topic
Internal topics	Delete the internal topic

The application reset tool **does not**:

- Reset output topics of an application
- Reset the local environment of your application instances

Running the Application Reset Tool

1. Run the application reset tool:

   `<path-to-confluent>/bin/kafka-streams-application-reset`.

2. Reset the local environments of your application instances.

3. Delete the application's local state directory prior to restarting it on same machine. Use any of the following methods:

   a. The API method `KafkaStreams#cleanUp()`.

   b. Manually delete the corresponding local state directory.

⚠ All instances of your application must be stopped.

CONFLUENT

Global Education

Module Overview

This module contains five lessons:

- How Can You Leverage Parallelism in Stream Processing?
- What if You Need to Adjust Processing Power in Your Stream Processing Deployment?
- How Can I Make Your Stream Processing Deal with Failures?
- What Are Some Guidelines for Sizing Your Stream Processing Deployment?
- What Configurations Should You Set for Kafka Streams?

Where this fits in:

- Hard Prerequisite: Introduction to Kafka Streams
- Recommended Prerequisite: Working with Kafka Streams
- Recommended Follow-Up: Either other module in this branch

08a: How Can You Leverage Parallelism in Stream Processing?

Description

Kafka Streams uses the Apache Kafka producer and consumer APIs, and leverages the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity. In Kafka Streams, the basic unit of parallelism is a stream task. So, to improve the parallelism, increase the number of partitions for the input topics which will automatically lead to a proportional increase in the number of tasks.

Deployment Concepts

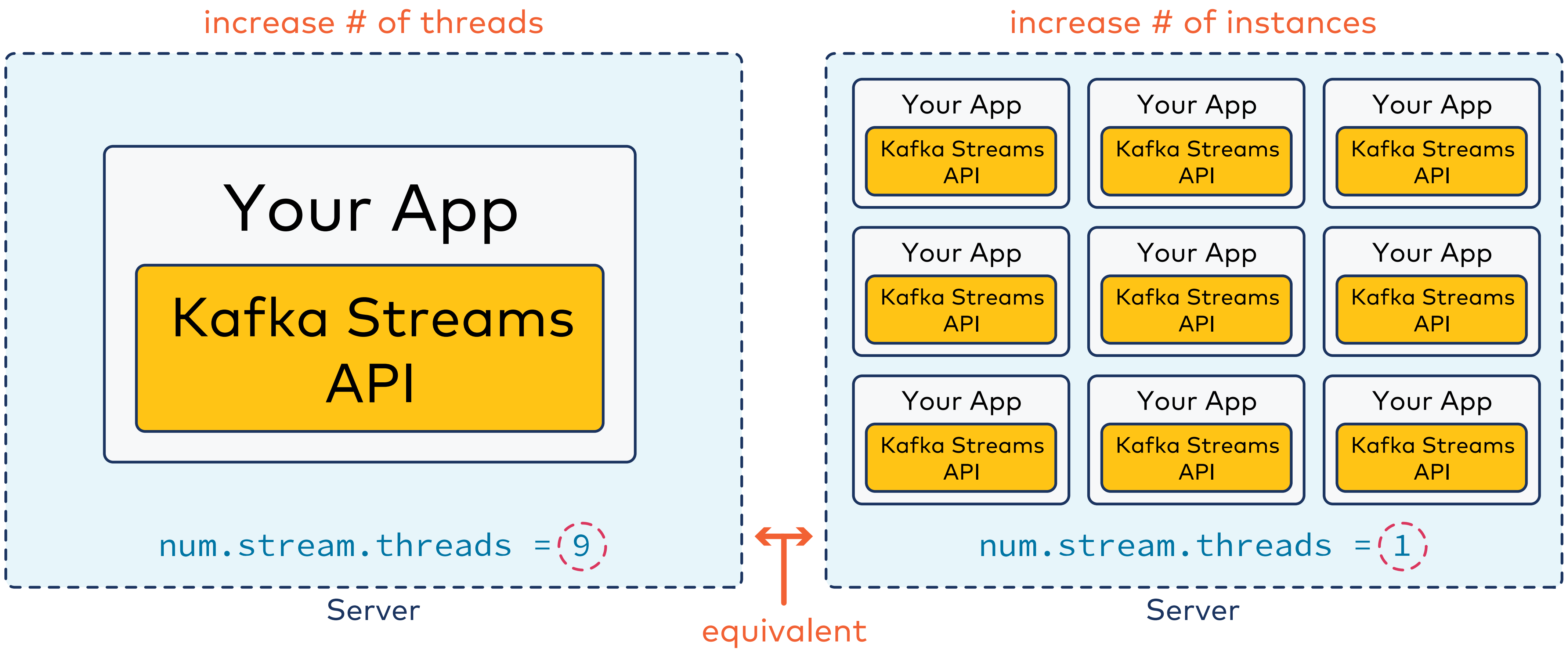
- Kafka Streams uses Kafka's **Producer and Consumer APIs**
- Unit of parallelism is a **Task**
- Task **Placement** matters
- Load Balancing is **automatic**

08b: What if You Need To Adjust Processing Power in Your Stream Processing Deployment?

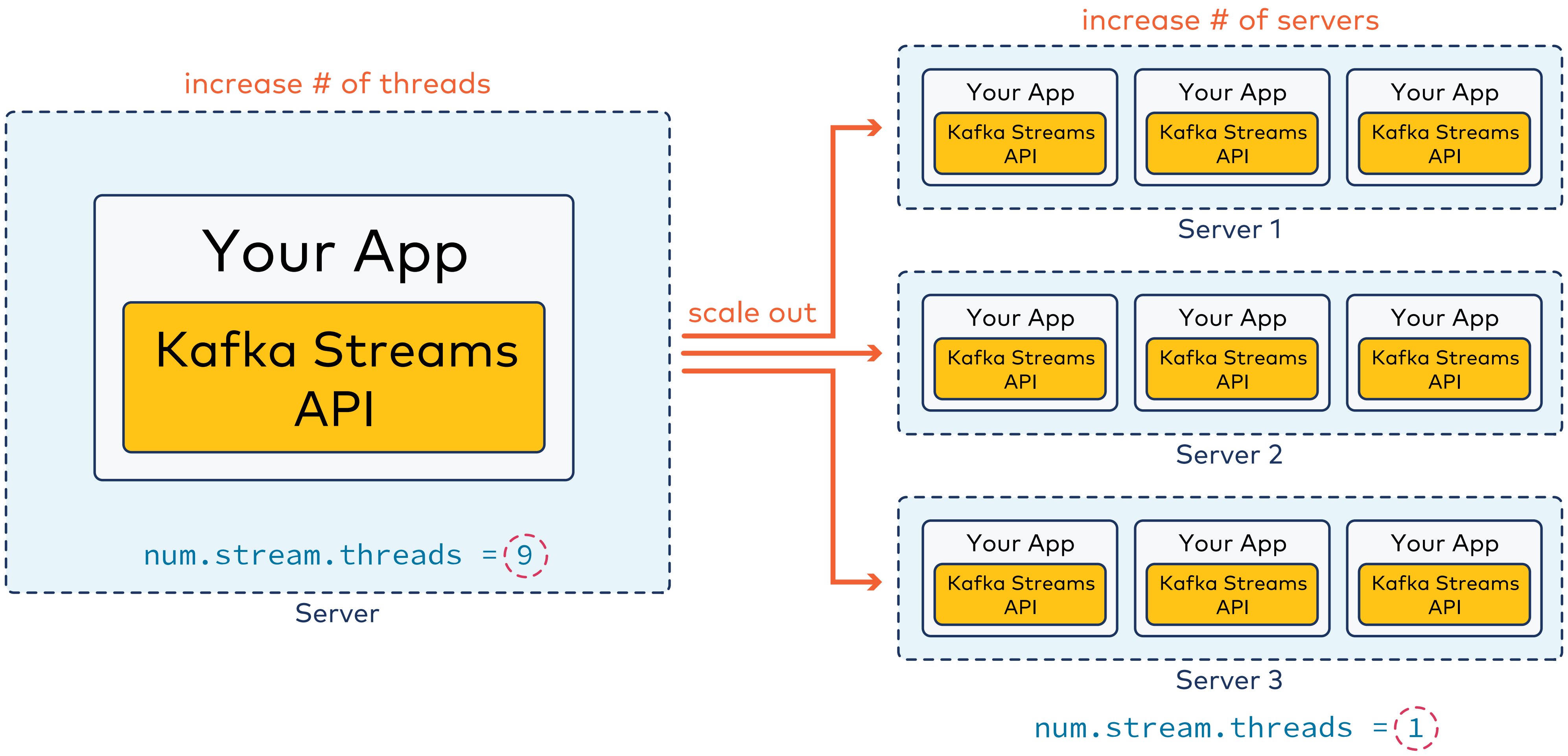
Description

Kafka Streams makes your stream processing applications elastic and scalable. You can add and remove processing capacity dynamically during application runtime without any downtime or data loss.

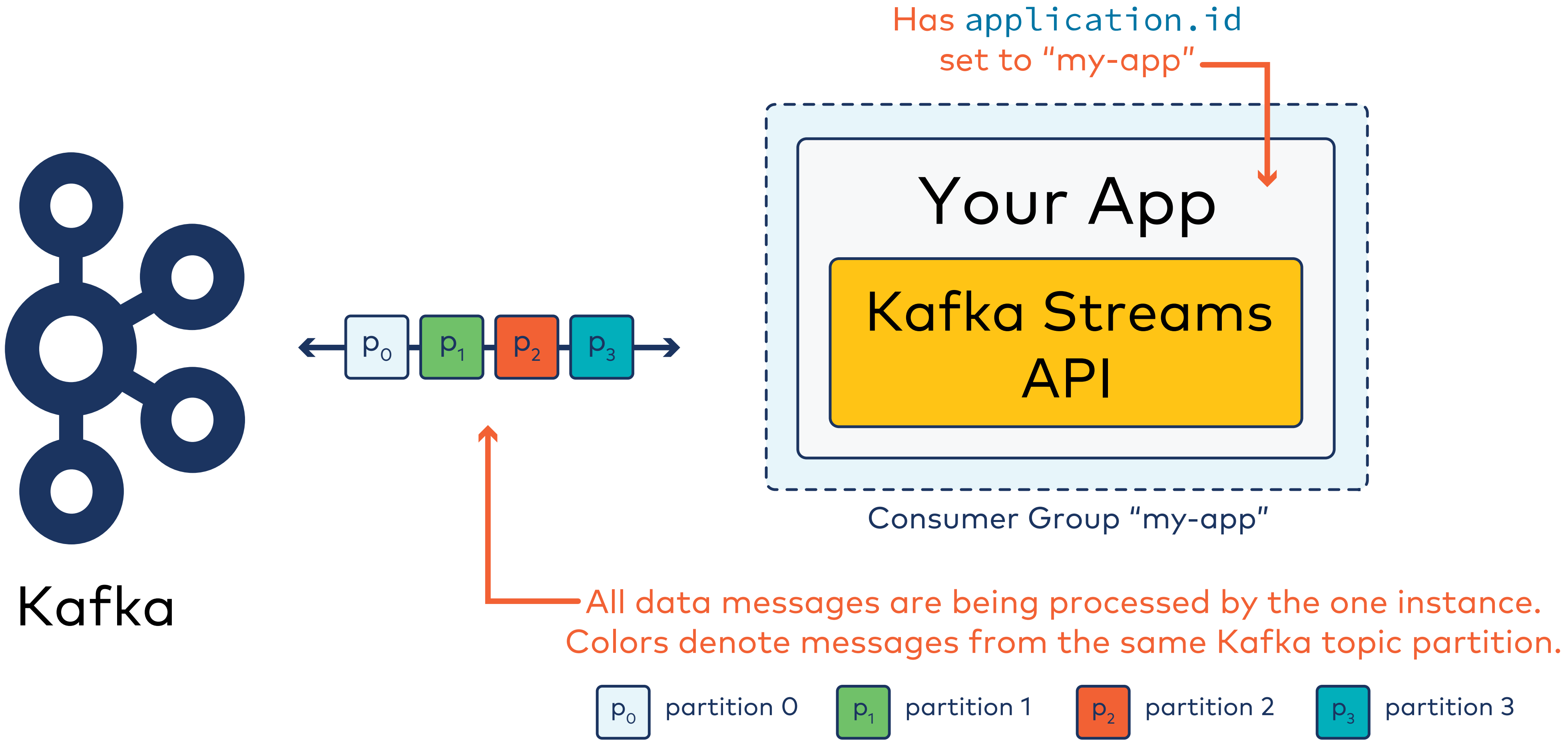
Task Placement - Scale Up



Task Placement - Scale Out



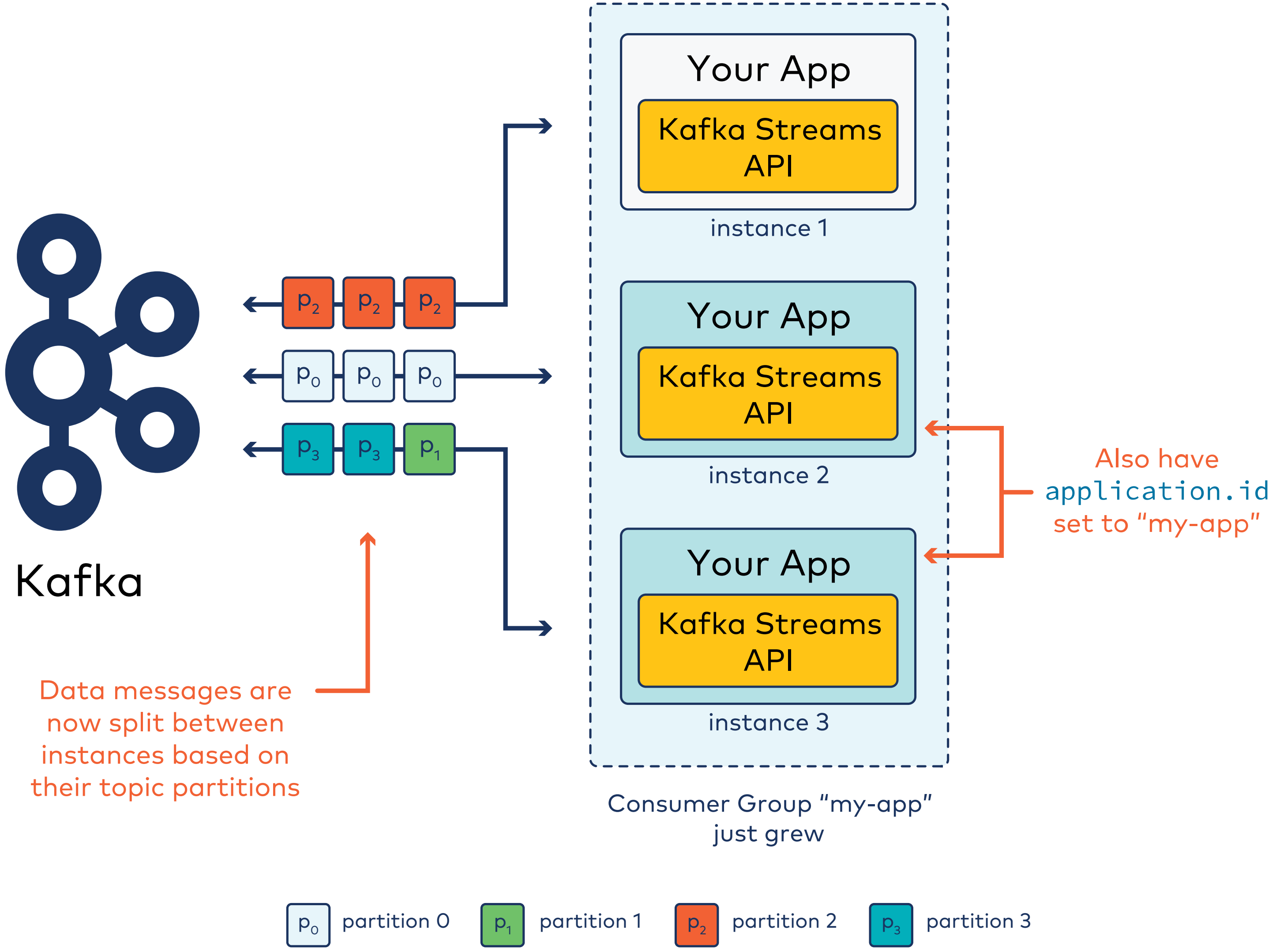
Elastic Scaling



Has `application.id` set to "my-app"

All data messages are being processed by the one instance.
Colors denote messages from the same Kafka topic partition.

Elastic Scaling - Scaling Up

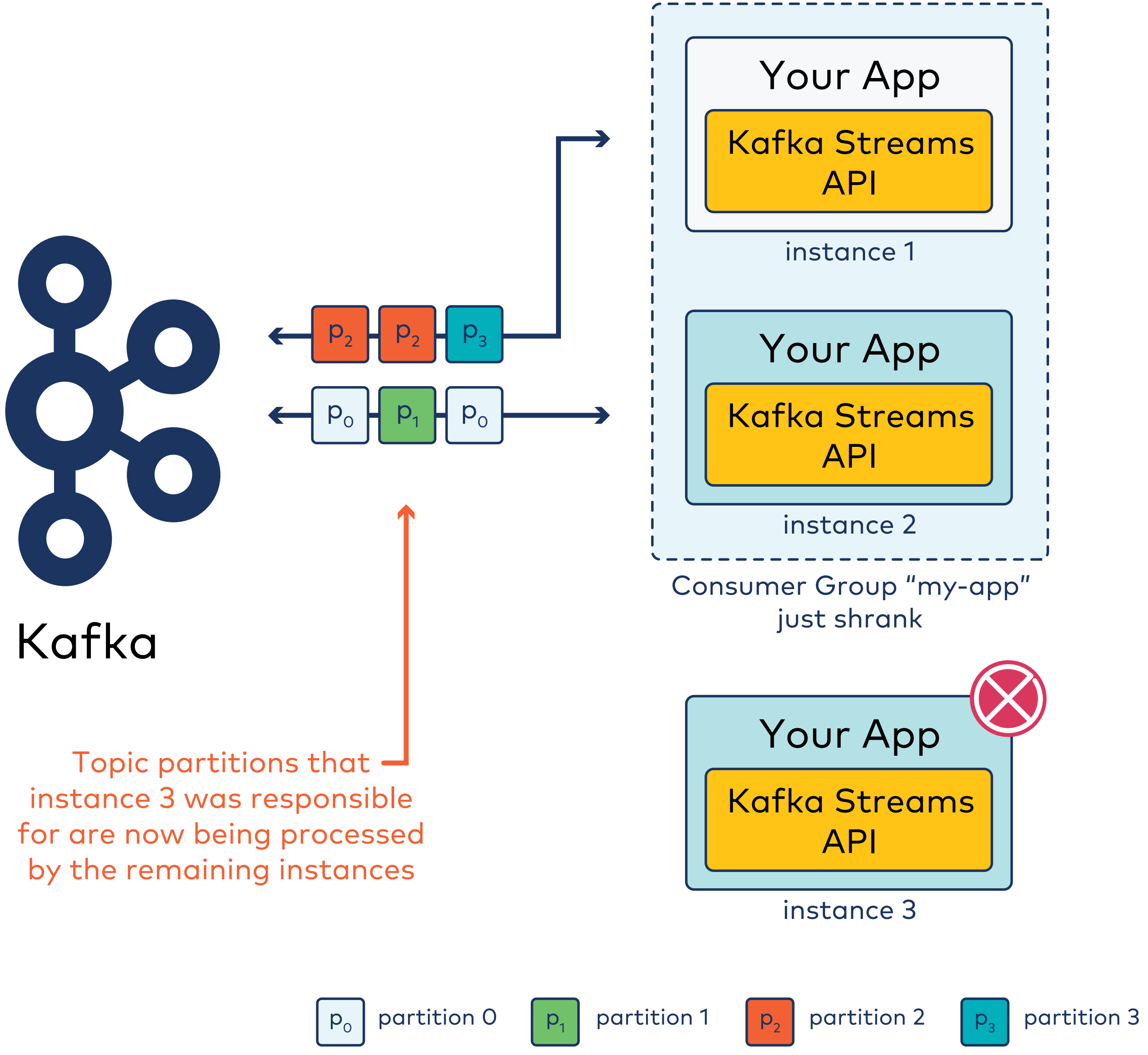


Data messages are now split between instances based on their topic partitions

Also have `application.id` set to "my-app"

Consumer Group "my-app" just grew

Elastic Scaling - Scaling Down



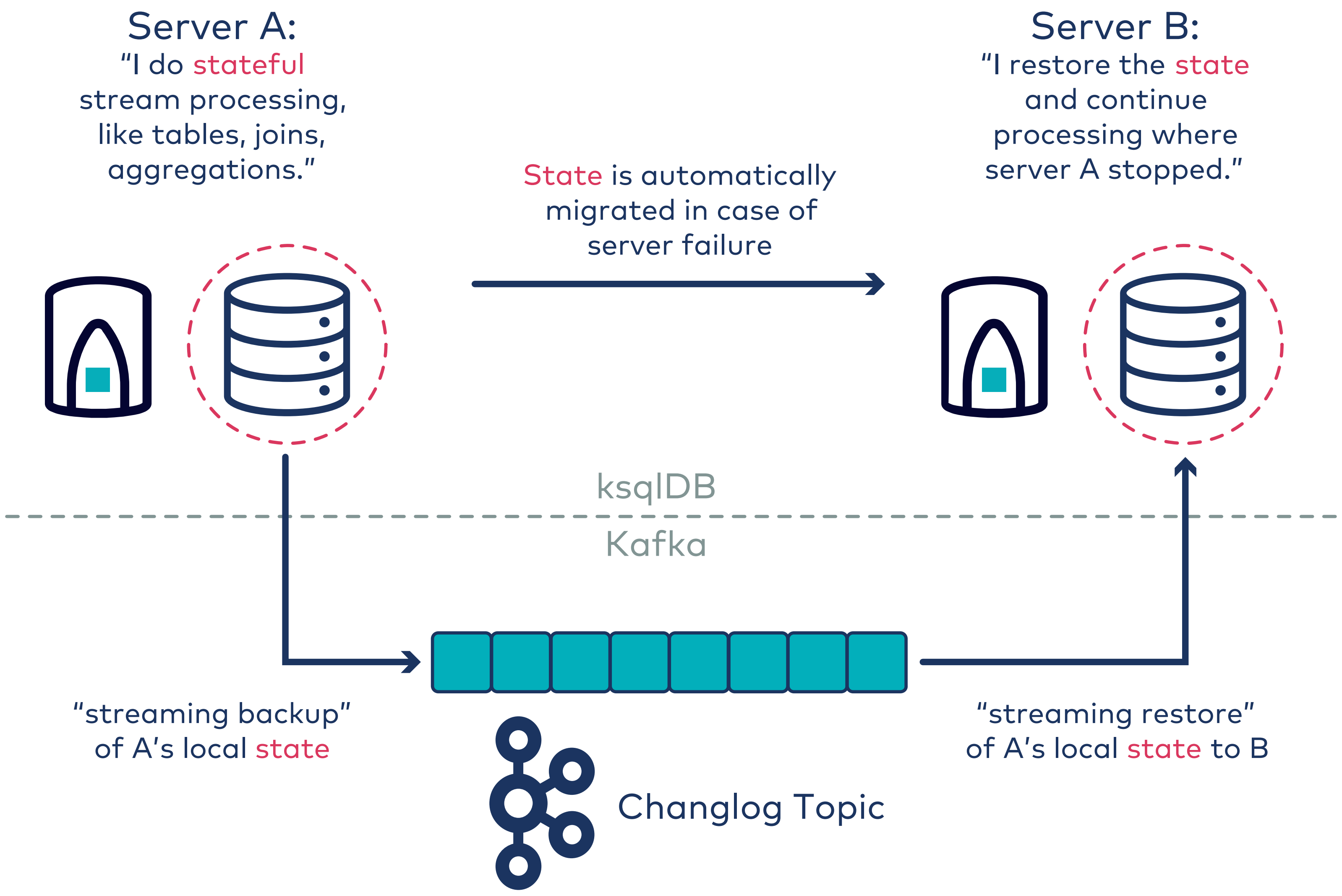
Topic partitions that instance 3 was responsible for are now being processed by the remaining instances

08c: How Can I Make Your Stream Processing Deal With Failures?

Description

Kafka Streams uses the Kafka Group Coordination Protocol which provides automatic fault tolerance and load sharing. Configure `num.standby.replicas` to be 1 or greater to reduce the recovery time during the fault.

Fault Tolerance Powered by Kafka



Server A:
"I do stateful stream processing, like tables, joins, aggregations."

State is automatically migrated in case of server failure

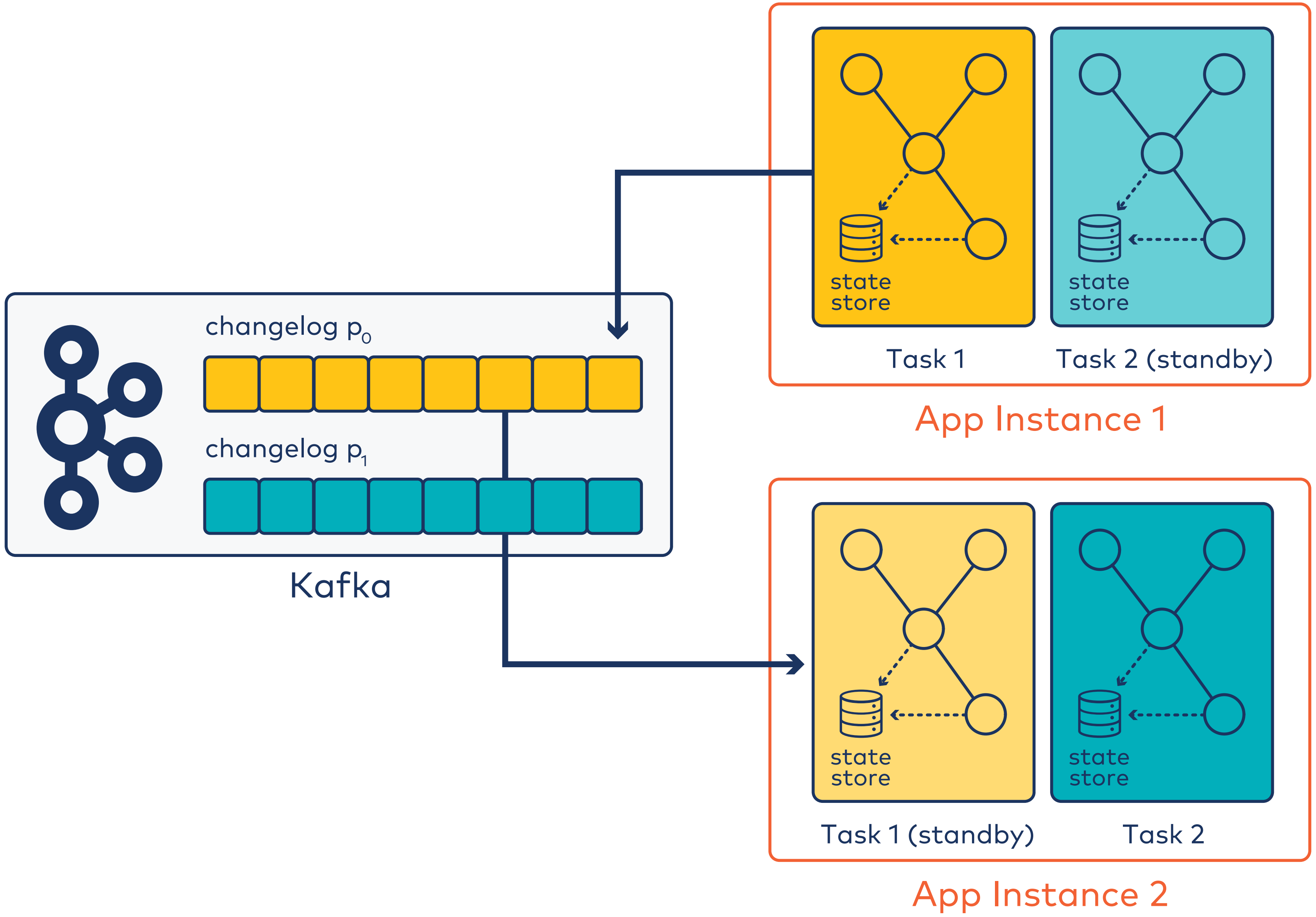
Server B:
"I restore the state and continue processing where server A stopped."

"streaming backup" of A's local state

Changlog Topic

"streaming restore" of A's local state to B

Standby Replicas



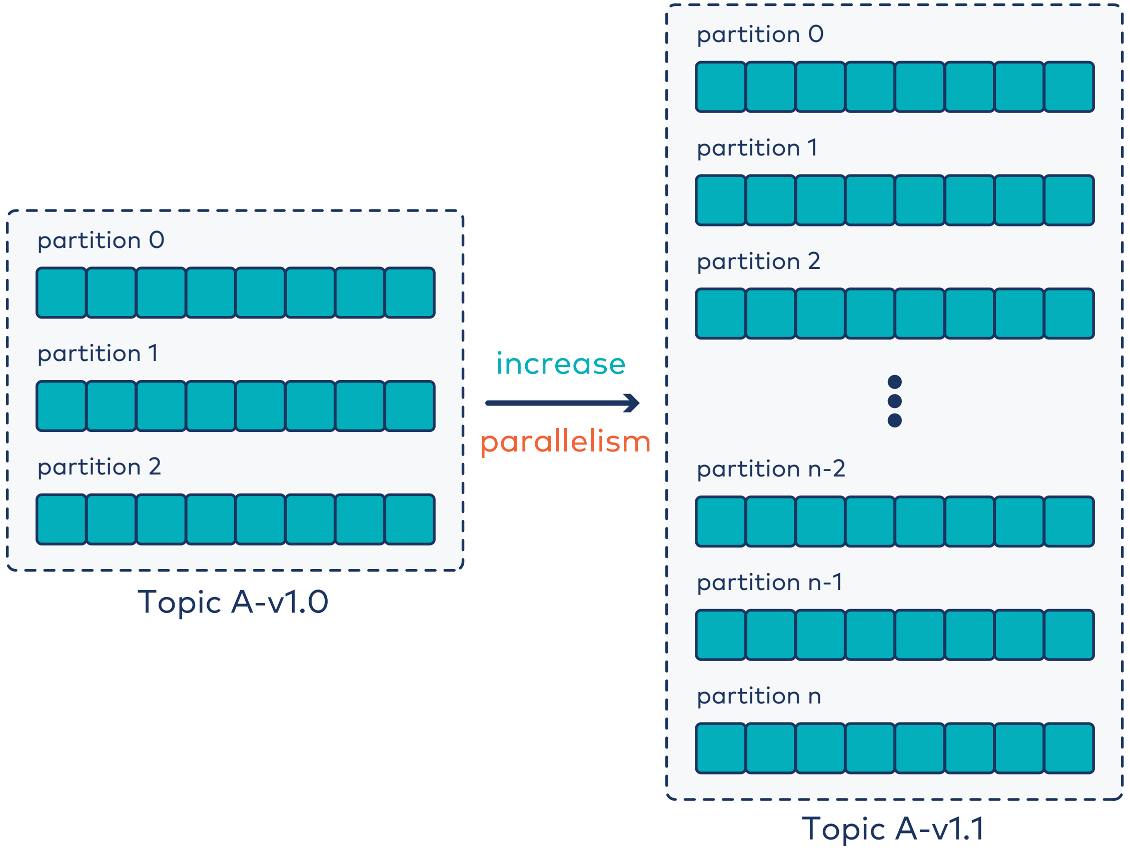
08d: What Are Some Guidelines for Sizing Your Stream Processing Deployment?

Description

Kafka Streams is a simple, powerful streaming library built on top of Apache Kafka. Under the hood, there are several key considerations to account for when provisioning your resources to run Kafka Streams applications.

Tuning Parallelism and Retention

Parallelism



partition 0

partition 1

partition 2

Topic A-v1.0

increase parallelism

partition 0

partition 1

partition 2

partition n-2

partition n-1

partition n

Topic A-v1.1

Data Retention Time

lower retention time

Number of Streams Instances

Important Sizing Factors:

- Throughput
- Operation Types (filters, joins, aggregations)
- Data Schema
- Number of Partitions
- Key Space

How Many Kafka Streams App Instances?

- Number of instances `<=` number of topic-partitions
- Distribute & balance data (topics)
- Distribute processing workload

Number of Kafka Brokers

Kafka Streams increases Broker Load:

- Topics from Streams and Tables
- State Store Changelog Topics
- Standby Replicas
- Repartitioning

08e: What Configurations Should You Set for Kafka Streams?

Description

We explore some of the important configuration properties for Kafka Streams.

Kafka Streams Configurations

Configuration property	Description	Default value
`application.id`	An identifier for the stream processing application	
`bootstrap.servers`	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster	
`state.dir`	Directory location for state store	`/tmp/kafka-streams`
`cache.max.bytes.buffering`	Maximum number of memory bytes to be used for buffering across all threads	`10485760`
`client.id`	An ID prefix string used for the client IDs of internal consumers and producers with pattern `'-StreamThread--'`	`""`

Stream Configurations

Configuration Property	Description	Default Value
`num.standby.replicas`	The number of standby replicas for each task	`0`
`num.stream.threads`	The number of threads to execute stream processing	`1`
`processing.guarantee`	The processing guarantee that should be used	`at_least_once`
`replication.factor`	The replication factor for changelog topics and repartition topics created by the stream processing application	`-1`
`commit.interval.ms`	The frequency in milliseconds with which to save the position of the processor	`30000` (30 seconds)

CONFLUENT

Global Education

Module Overview

This module contains one lesson:

- How Do You Secure Your Stream Processing?

Where this fits in:

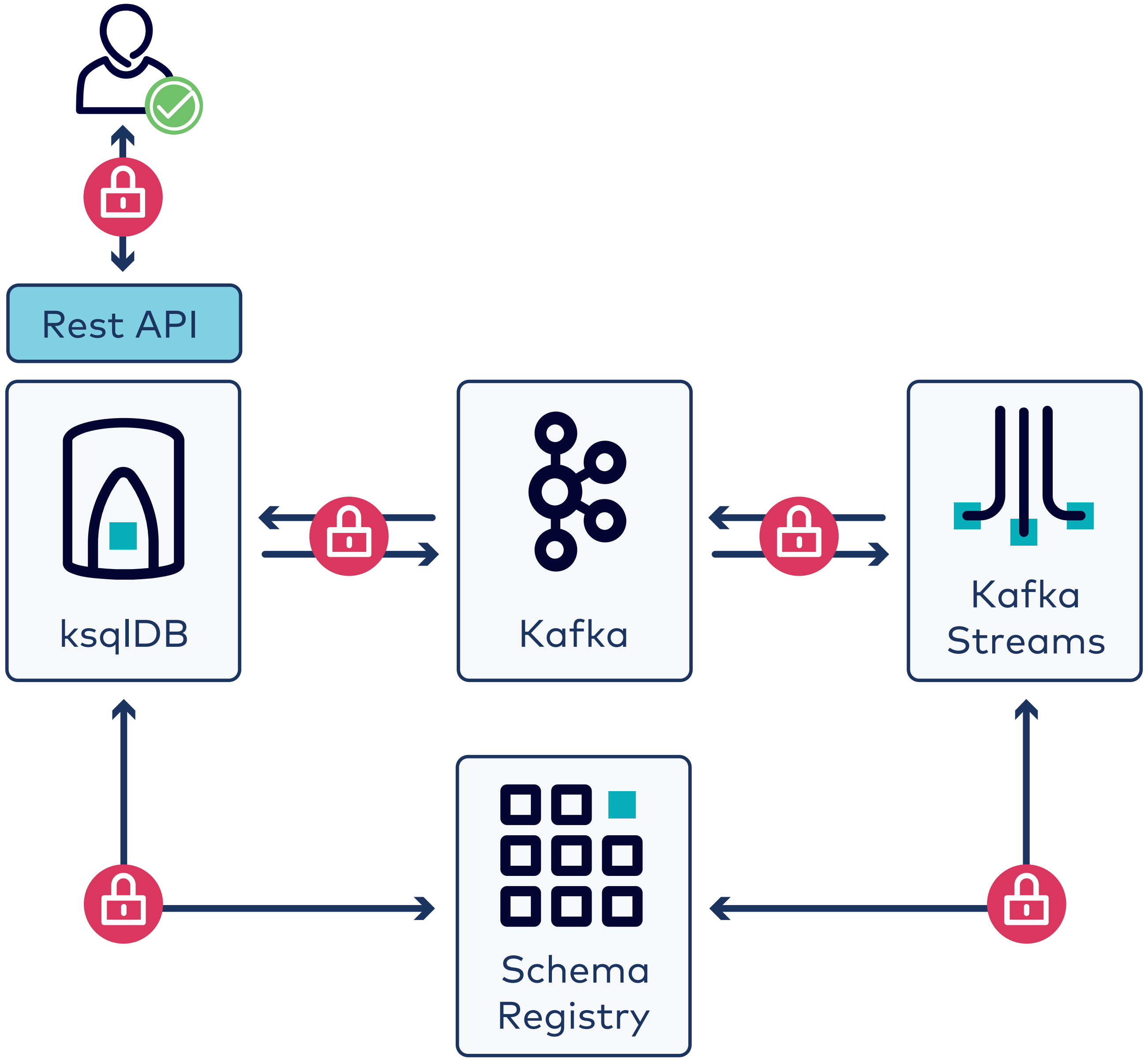
- Hard Prerequisite: Introduction to Kafka Streams
- Recommended Prerequisite: Working with Kafka Streams
- Recommended Follow-Up: Either of Deployment or Testing, Troubleshooting, and Monitoring

09a: How Do You Secure Your Stream Processing?

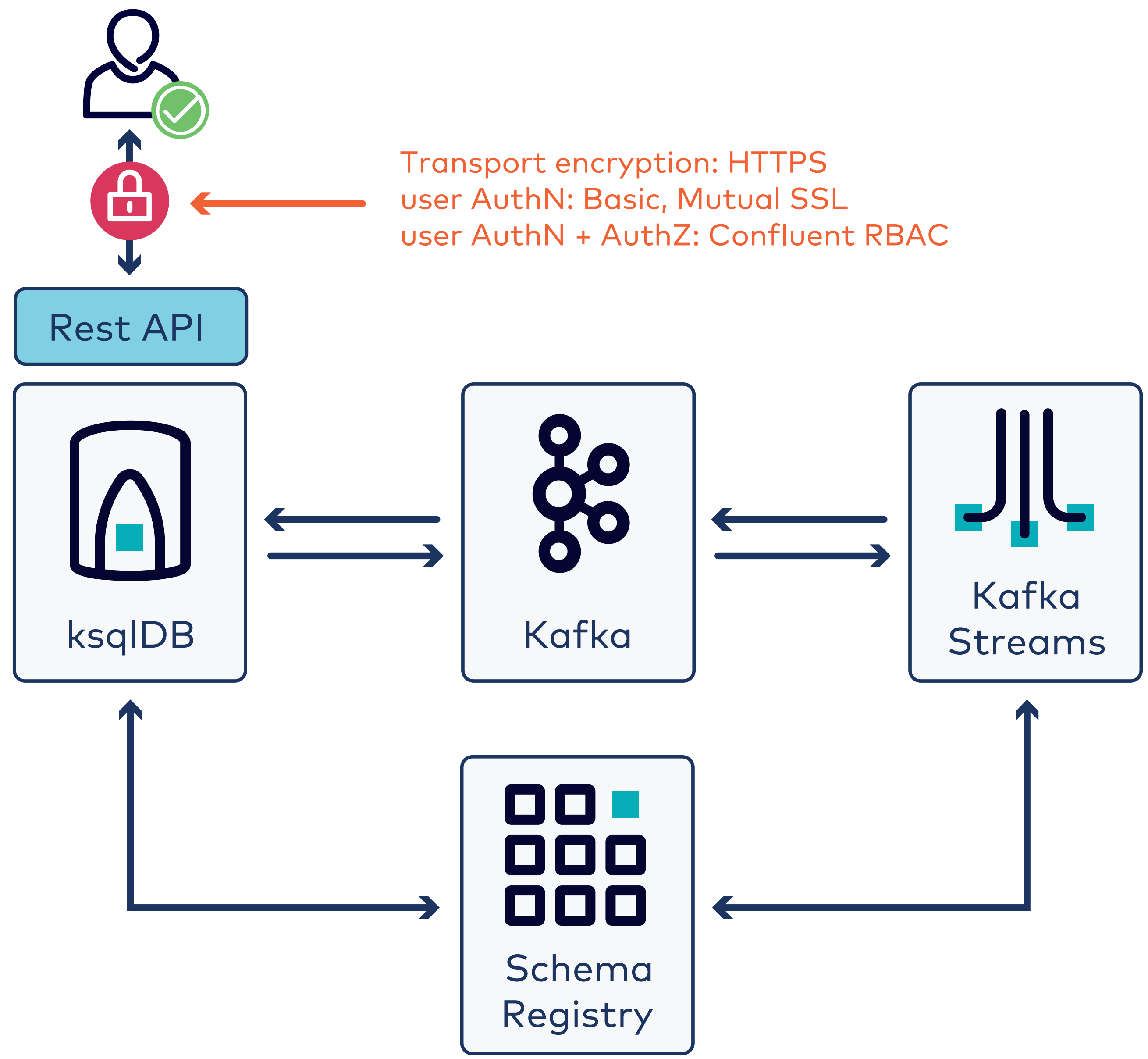
Description

How do you ensure only verified entities can access your Kafka Streams applications? This lesson explores what you need to know to secure access to your Kafka Streams applications.

Security Overview

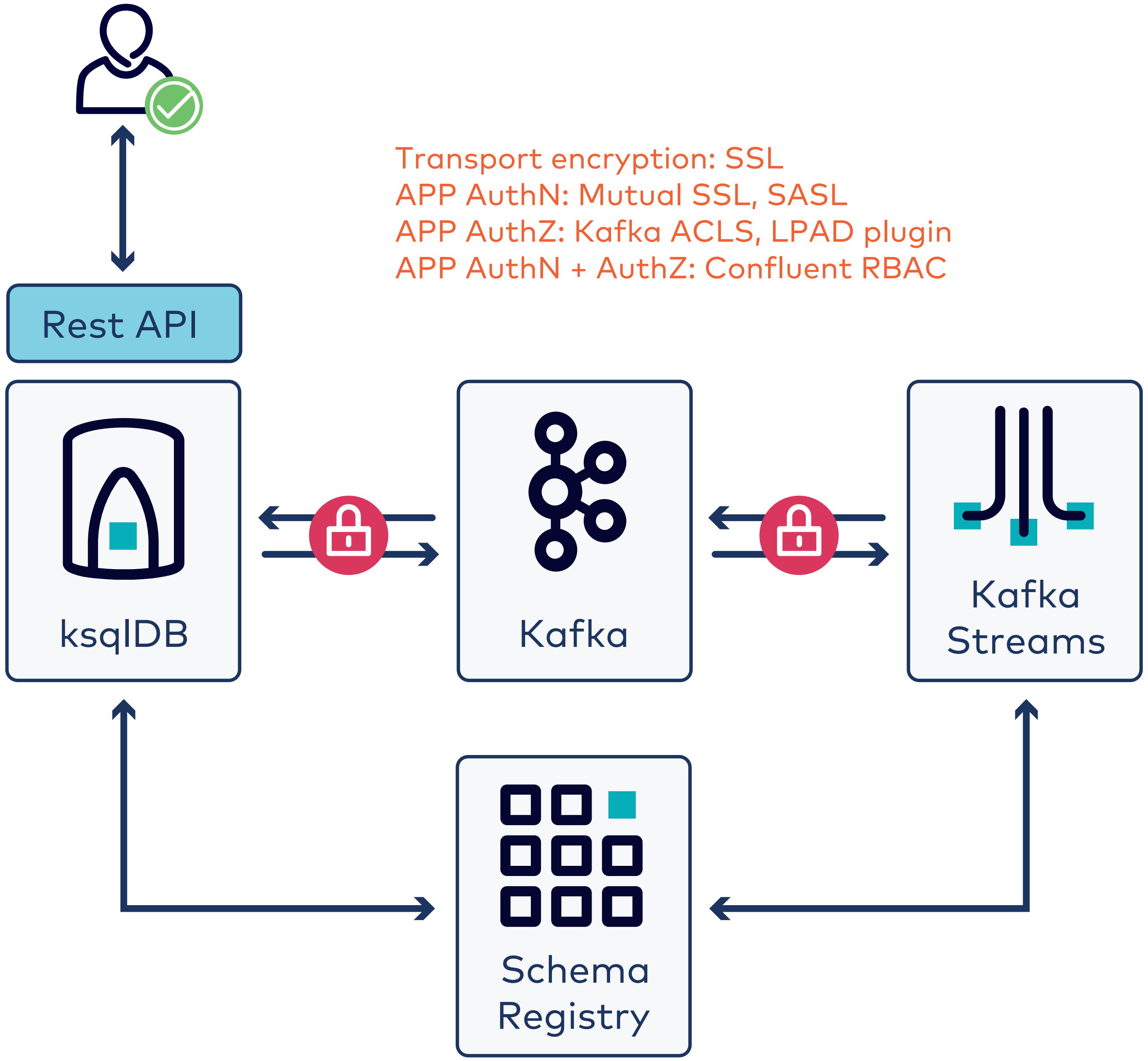


Security Overview — ksqlDB REST API Users



Transport encryption: HTTPS
user AuthN: Basic, Mutual SSL
user AuthN + AuthZ: Confluent RBAC

Security Overview — Connecting to Kafka



Transport encryption: SSL
APP AuthN: Mutual SSL, SASL
APP AuthZ: Kafka ACLS, LPAD plugin
APP AuthN + AuthZ: Confluent RBAC

Rest API

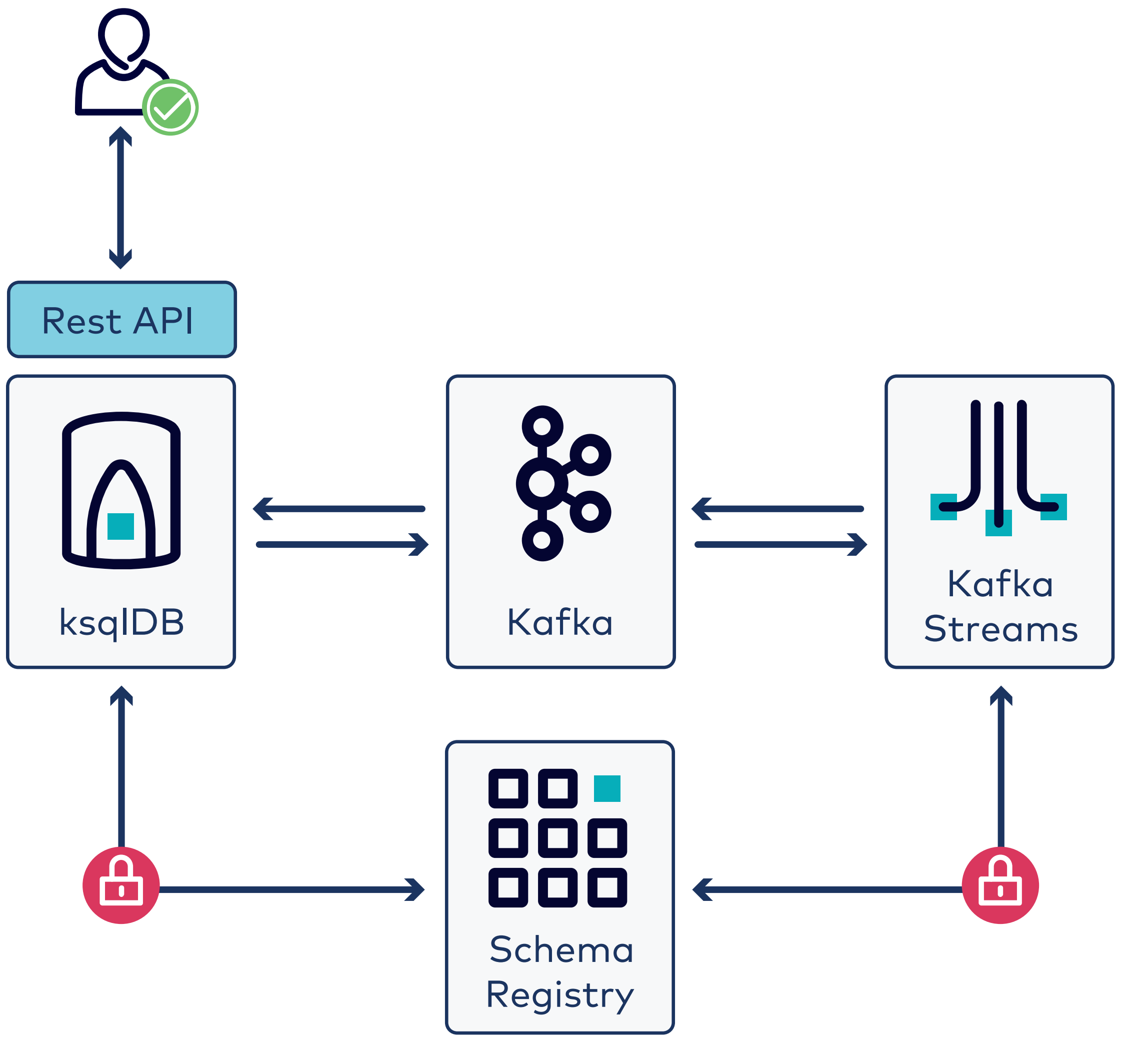
ksqlDB

Kafka

Kafka Streams

Schema Registry

Security Overview — Schema Registry



Transport encryption: HTTPS
AuthN: Basic, Mutual SSL
AuthZ: Schema Registry ACL Plugin
AuthN + AuthZ: Confluent RBAC

Access Control Lists (ACLs)

Authorizing Access to Resources:

- Blanket Access: indicated by `*`
- Individual ACLs
- Prefixed Resources

Access Control Lists (ACLs)

- The `DESCRIBE_CONFIGS` operation on the CLUSTER resource type.
- The `CREATE` operation on the CLUSTER resource type.
- The `DESCRIBE`, `READ`, `WRITE`, and `DELETE` operations on all TOPIC resource types.
- The `DESCRIBE` and `READ` operations on all GROUP resource types.

ACL Prefixes

- Allow Streams to manage its own internal topics and consumer groups:

```
$ kafka-acls --add \
    --allow-principal User:team1 \
    --operation All \
    --topic team1-streams-app1-topic1 \
    --group team1-streams-app1 \
    --resource-pattern-type prefixed
```

- Simpler ACL management: **Use prefixed resources** (see example a few slides later).

Creation of Internal and Output Topics

- An application may have permission to create its internal and output topics.
- Alternatively, one can manually create those topics:
  - Internal topic should have the same number of partitions as an input topic.
  - Changelog topics must be created with log compaction enabled.
  - For changelog topics for windowed `KTable`s, apply `delete,compact`.
  - For repartition set `cleanup.policy=delete` and allow `delete` operation.

Encryption In Transit Example

```java
/* ... non-security settings ... */
Properties settings = new Properties();
settings.put(StreamsConfig.APPLICATION_ID_CONFIG, "secure-kafka-streams-app");
settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.com:9093");
/* ... security settings ... */
settings.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
settings.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
            "/etc/security/tls/kafka.client.truststore.jks");
settings.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
            "truststore-password");
/* For mutual SSL, we also configure the keystore */
settings.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,
            "/etc/security/tls/kafka.client.keystore.jks");
settings.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG,
            "keystore-password");
settings.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG,
            "private-key-password");
```

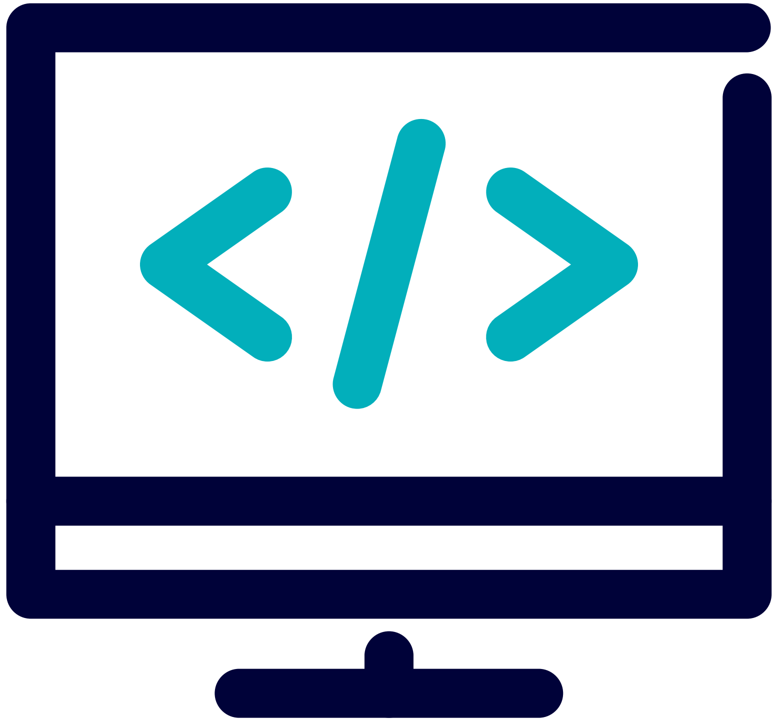
Securing Monitoring Interceptors

```
producer.confluent.monitoring.interceptor.bootstrap.servers =
    kafka:9091
producer.confluent.monitoring.interceptor.security.protocol =
    SASL_SSL
producer.confluent.monitoring.interceptor.ssl.truststore.location =
    /etc/kafka/secrets/client.truststore.jks
producer.confluent.monitoring.interceptor.ssl.truststore.password =
    confluent
producer.confluent.monitoring.interceptor.sasl.mechanism =
    PLAIN
producer.confluent.monitoring.interceptor.sasl.jaas.config =
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="client"
    password="client-secret"
```

Lab: Securing a Kafka Streams Application

Please work on **Lab 9a: Securing a Kafka Streams Application**

Refer to the Exercise Guide



Conclusion

CONFLUENT

Global Education

Course Contents

Now that you have completed this course, you should have the skills to:

- Identify common patterns and use cases for real-time stream processing
- Describe the high-level architecture of Apache Kafka Streams
- Write real-time applications with the Kafka Streams API to filter, transform, enrich, aggregate, and join data streams
- Describe how Kafka Streams provide elastic, fault-tolerant, high-performance stream processing capabilities
- Test, secure, deploy, and monitor Kafka Streams applications

Other Confluent Training Courses

- Confluent Developer Skills for Building Apache Kafka®
- Apache Kafka® Administration by Confluent
- Confluent Advanced Skills for Optimizing Apache Kafka®
- Managing Data in Motion with Confluent Cloud

For more details, see https://confluent.io/training

Confluent Certified Developer for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and Confluent products, and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours a day!

Cost: $150

Register online: `www.confluent.io/certification`

Confluent Certified Administrator for Apache Kafka

Duration: 90 minutes

Qualifications: Solid work foundation in Confluent products and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours per day!

Cost: $150

Register online: `www.confluent.io/certification`

We Appreciate Your Feedback!

Please complete the course survey now.

Thank You!

Additional Problems to Solve

CONFLUENT

Global Education

Overview

This section contains a few additional problems to be solved that will reinforce the concepts in this course.

Some of these problems were originally written as warm-up problems for instructor-led training for this course. Your instructor may or may not choose to incorporate some or all of these problems in class; you may find them to provide additional enrichment in any case. Some other problems originally created as warm-up problems have been adapted into activities in the content of this version of this course.

Some other problems were written as "food for thought" extra problems, not necessarily intended to be used in the flow of a class, but intended to give curious students additional problems to think about.

Problem A: Getting Started with Stream Concepts

Suppose we are working with a Kafka cluster that has topic `purchases_topic`, which has these key-value pairs: (*a*, 15), (*b*, 52), (*b*, 32), (c, 2), (*a*, 21), (c, 71). Keys can be interpreted as an ID of a user, and values can be interpreted as how much a purchase costs, rounded to the nearest dollar. Then...

a. If we create a stream from this topic with the current data, what is in the stream?
b. If we treat that stream as a table, what is in the table?
c. How could you interpret the meaning of what is the table?

Problem B: Getting Started with the DSL

Part 1

Look at this code, reformatted from the slides in the lesson "Anatomy of a Kafka Streams App":

```java
Properties settings;
Serde<String> stringSerde;
Serde<Double> doubleSerde;
StreamsBuilder builder;
KStream<String, Double> temps;
KStream<String, Double> highTemps;
Topology topology;
KafkaStreams streams;

// ...

stringSerde = serdes.String();
doubleSerde = serdes.Double();

builder = new StreamsBuilder();
```

```java

temps = builder.stream("temp-topic",
                       Consumed.with(stringSerde, doubleSerde));

highTemps = temps.filter((key, value) -> value > 25);

highTemps.to("high-temp-topic",
             Produced.with(stringSerde, doubleSerde));

topology = builder.build();

streams = new KafkaStreams(topology, settings);
streams.start();

//...
```

Consider these two lines of code:

a. `highTemps = temps.filter((key, value) -> value > 25);`

b. `streams.start()`

Which executes first? Explain.

Part 2

Suppose you have a stream of events whose keys are account numbers and whose values are delimited text listings of transactions for the corresponding account for a month at a time. Your goal is to create a stream where keys are account numbers and values are *individual* transactions parsed from the input stream. You plan to use the Kafka Streams DSL to do this.

a. Would you alter the existing stream or create a new stream? Why?
b. What DSL operation would be ideal to achieve this task? Explain.
c. What DSL operation would achieve this task, but be a poor choice? Explain.

Problem C: Aggregating a KTable - Demographic Data

Consider the Step by Step KTable aggregation example on the Slide "Aggregating a `KTable` - Step by Step." This problem is in the same vein, but a second example for you to work out. The end goal here would be to calculate the average age of users by postal code. Here is our problem setup to do this...

- Inputs will be tuples: (user ID, (postal code, age in years))
- State will be a key-value pair, where the keys are postal codes and values are, in turn, pairs of (total age of users in postal code, number of users in postal code).
- Beyond the problem at hand, one would simply do a final division step for each state element.

Fill in a copy of this table, step by step, as modeled in the slide, but for the scenario described above:

Time-stamp	Input Record	Inter-preted As	Grouping	Initializer	Adder	Subtractor	Changed State
1	(a, (16802, 20))						
2	(b, (16802, 18))						
3	(c, (16803, 70))						
4	(d, (16801, 35))						

Time-stamp	Input Record	Inter-preted As	Grouping	Initializer	Adder	Subtractor	Changed State
5	(a, (16801, 20))						
6	(e, (16802, 19))						
7	(b, null)						

Problem D: Basic Windowing

Part 1: Comparing Types of Windows

Let’s pretend that, for whatever reason, we are required to choose only some ksqlDB features to keep and cannot keep them all. We are only permitted to use one of tumbling or hopping windows, but not both. Which would you pick and why?

Part 2: Calculating Windows

Suppose we have a timeline of click events, all with the same key, that happen at the following times: 1, 2, 4, 7, 8, 9, 12, 14, 17, 19. List the times of events included in each window if the windowing mode is...

a. Tumbling with size 5

b. Hopping with size 5 and "advance by" 3

Problem E: Windowing with Keys

Changing Keys

Now, let's change the above prior problem statement.

Recall, we had a timeline of click events, that happened at the following times: 1, 2, 4, 7, 8, 9, 12, 14, 17, 19.

Before, we said all messages had the same key. Suppose, instead,

- all messages have key k_1, ...
- ... except for the messages at times 4, 12, and 14. All of these have key k_2.

With this change,

a. How does your answer to the prior problem on tumbling windows, size 5, change?
b. How does your answer to the prior problem on hopping windows, size 5, advance by 3, change?

Problem F: Sliding Windows

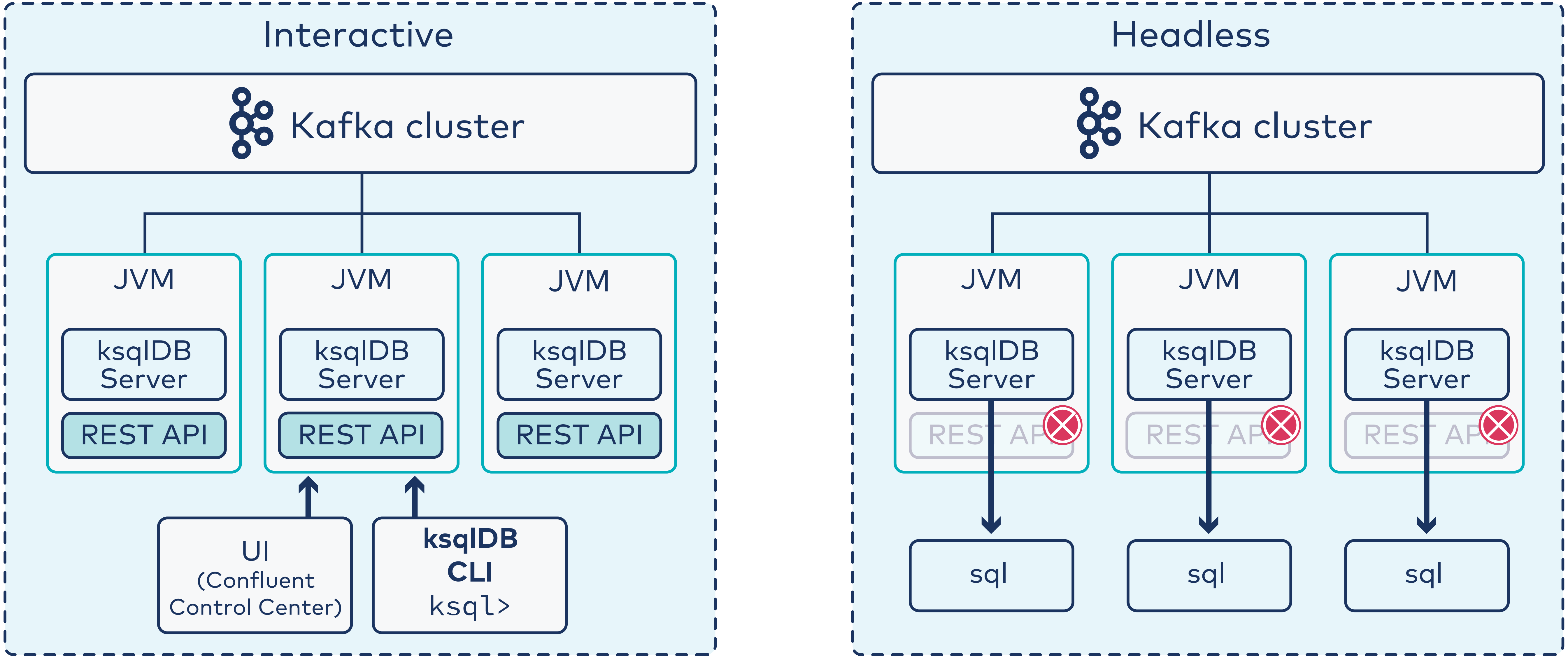
Sliding Windows

Another problem started like this: Suppose we have a timeline of click events, all with the same key, that happen at the following times: 1, 2, 4, 7, 8, 9, 12, 14, 17, 19. List the times of events included in each window if the windowing mode is...

Let's now consider **sliding** windows, again with **size 5**, and solve the same problem. **BUT** in order to keep the problem reasonable in length, consider *only* the events at time 1, 4, 7, and 8.

Problem G: Deployment Modes

There are two deployment modes for ksqlDB:



You want to run ksqlDB SQL queries at the command line. Should/could the ksqlDB server(s) be run in interactive mode or headless mode? Could both work? Neither? Does it matter? Why or why not?

Problem H: Aggregating Where the Adder Isn't Adding; Reduce

On the Slide titled "Aggregating a `KStream`," we talk about using `aggregate()` to add up the lengths of some strings. In that example, our adder is literally an adder, but it doesn't need to be.

If you come from a programming or CS background, you may recall in your first programming class learning how to solve some of the classic problems you can solve with loops - like sums and counts. Extreme values fit into the same group. Read the problems below and see if you can solve them:

a. Using `aggregate`, compute the maximum value in `inputStream` (assuming it is as on the referenced slide, a stream with `Long` values. (Hint: the ternary conditional operator is your friend.)
b. Look a few slides ahead and look up the documentation for `reduce()` and use it to solve the same problem as (a).
c. We were able to use both `aggregate()` and `reduce()` to solve this problem. Why? What is a characteristic of that, if changed, would make one or the other not suitable?

Problem I: Repartitioning Streams

Suppose we have a Kafka topic *t* with numeric keys and values that are tuples with a character and a numeric value. Suppose *t* has two partitions:

p_0:

- [2, (*a*, 10)]
- [2, (*b*, 6)]
- [4, (*a*, 7)]
- [4, (*a*, 25)]

p_1:

- [1, (*b*, 3)]
- [3, (*a*, 7)]

Under the hood, streams are partitioned. Consider the following:

a. Suppose we've initialized and built a stream `tStream` from *t*. What do you expect the partitioning for the resulting stream to be?

b. Suppose this code runs:

```
tStream.map((key, value) -> KeyValue.pair(value.getLetter()),
                                          new Pair(key,
                                                   value.getNumber()))
```

What do you expect the partitioning for the intermediate result to be?

c. Suppose we add the following on to the previous code:

```
        .filter((key, value) -> (value.number <= 10))
```

What do you expect the partitioning for the intermediate result to be?

d. Now suppose we further add on to the previous code:

```
       .groupByKey()
```

What do you expect the partitioning for the intermediate result to be?

e. At some point among the above operations, under the hood, information is produced to a new Kafka topic and a subtopology reads it back to the Streams app. When do you expect that to happen?

f. Finally, suppose the code in (b) instead used `mapValues` and made the output value the letter part of the input value. How would your answers to (b)-(e) change? Why?

Problem J: Using the Branch Operation

Consider the module 2 slide "Stateless Operations - `branch`." Think of a practical application, ideally in your company's context, where you could leverage this. Draw the processor topology.