

Kafka Streams API For Developers using Java/SpringBoot

Dilip Sundarraaj

About Me

- Dilip
- Building Software's since 2008
- Teaching in **UDEMY** Since 2016

What's Covered?

- Introduction to Kafka Streams
- Explore KStream API and KTable API
- Explore different operators using **High Level DSL**
- Build Stateless and Stateful Applications
- Explore Aggregations, Joins & Windowing
- Build a **Realtime Retail application** using Kafka Streams API
- Build Kafka Streams Application using **SpringBoot**
- **Interactive Queries** using SpringBoot
- Unit and Integration Tests using **JUnit5**

Targeted Audience

- Any developer who is curious to understand the Kafka Streams API.
- Hands-on oriented course
- Any developer who is interested in exploring different realtime use cases using Kafka Streams API.
- Building Kafka Streams Application using Spring Boot.

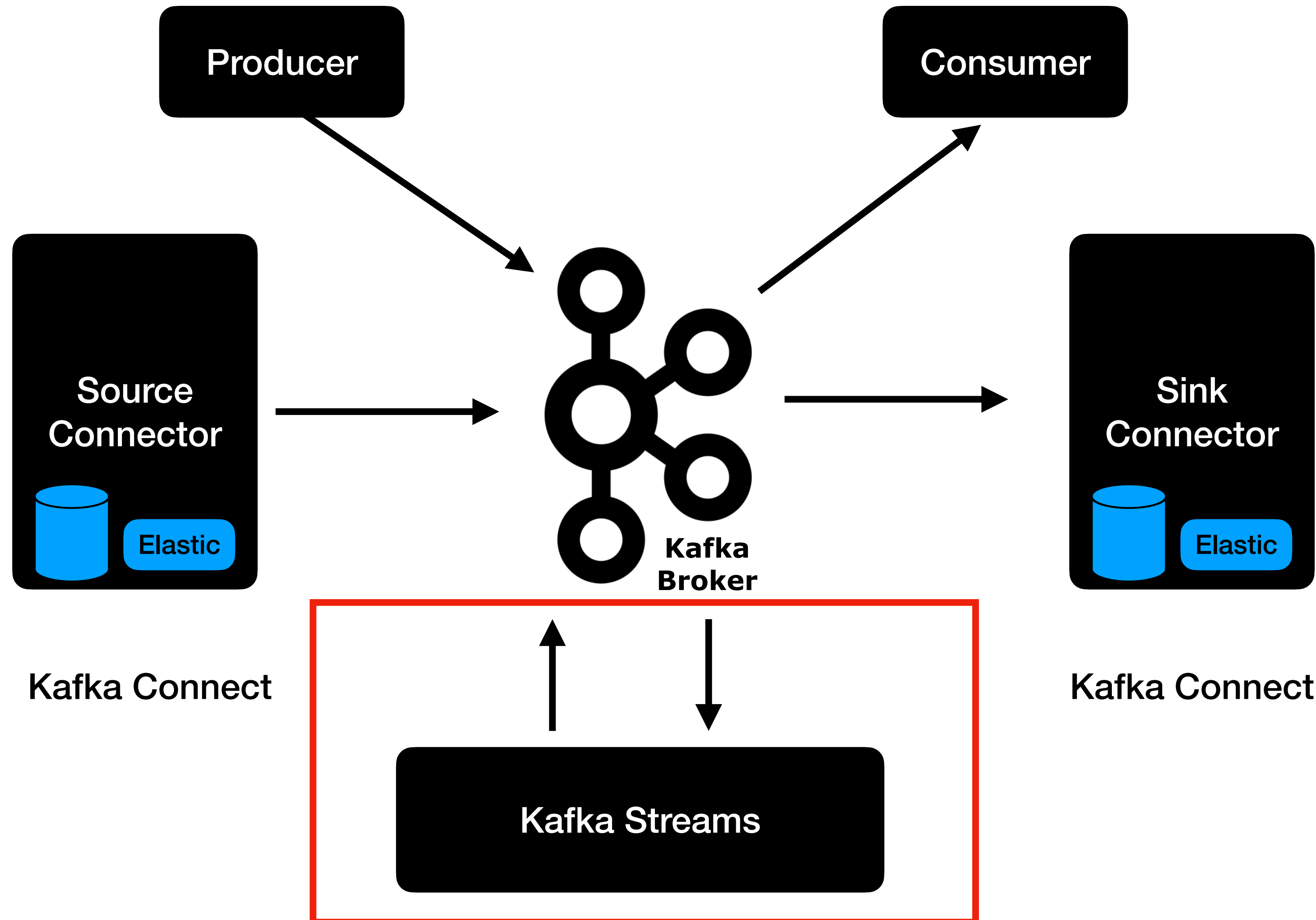
Source Code

Thank You!

Prerequisites

- Java 17
- Docker
- Prior Java Experience is a must
 - Functional programming concepts such as Lambdas, Streams API.
- Prior Kafka Experience is a must
- Prior Spring Framework/SpringBoot is a nice to have
- **IntelliJ** or any other IDE

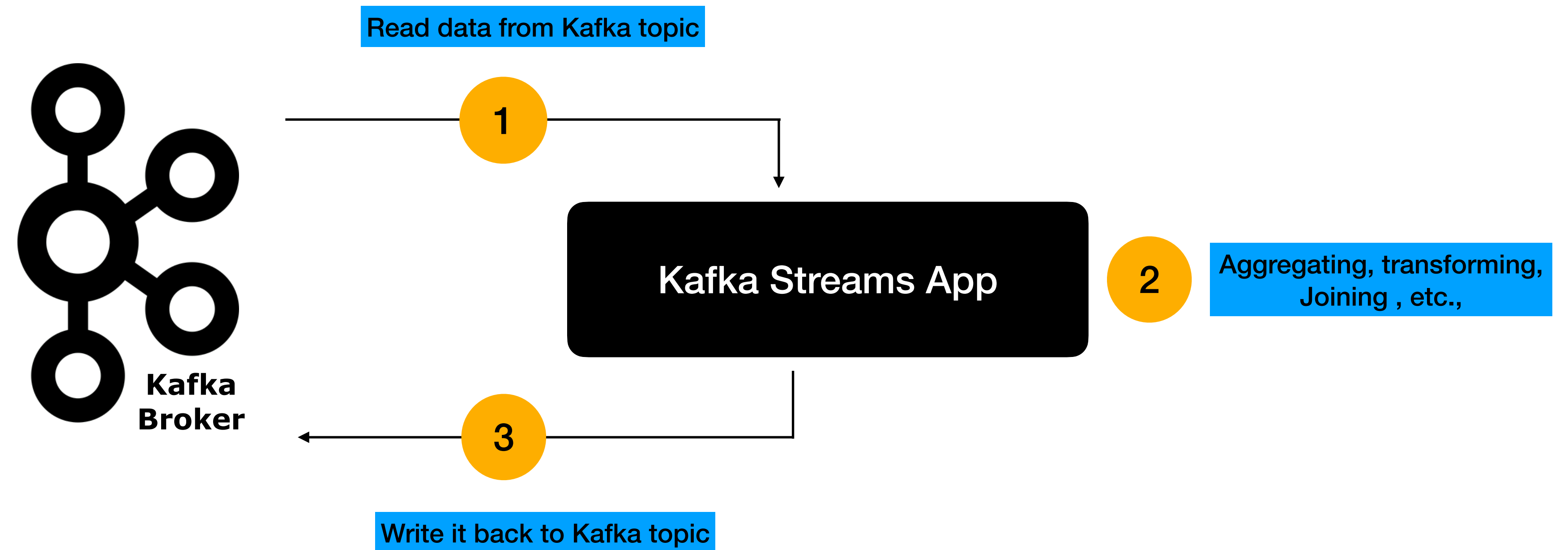
Overview of Kafka Architecture & Client APIs



Introduction to Kafka Streams API

- Kafka Streams is a Java Library which primary focuses on :
 - Data Enrichment
 - Transformation of Data
 - Aggregating the data
 - Joining data from multiple Kafka topics
- Kafka Streams API uses Functional programming Style
 - Lambdas
 - Map, filter , flatMap operators similar to Modern Java features
- Kafka Streams API is build on top of Java 8

Data Flow in a Kafka Streams App



Behind the scenes Streams API uses the producer and consumer API to read and write it back to Kafka topic

How is this different from Consumer API ?

- Consumer applications built using Consumer API are stateless.
- Kafka Consumer App Data Flow :
 - Read the event
 - Process the event
 - Move on to the next event
- Consumer apps are great for notifications or processing each event independent of each other.
- Consumer apps does not have an easy way to join or aggregate events.

Streams API use cases

- Stateful and Stateless applications
- Stateless applications are similar to what we build using the Consumer API
- Stateful Operations:
 - Retail:
 - Calculating the total number of orders in realtime
 - Calculating a total revenue made in realtime
 - Entertainment
 - Total number of tickets sold in realtime
 - Total revenue generated by a movie in realtime

Streams API Implementations

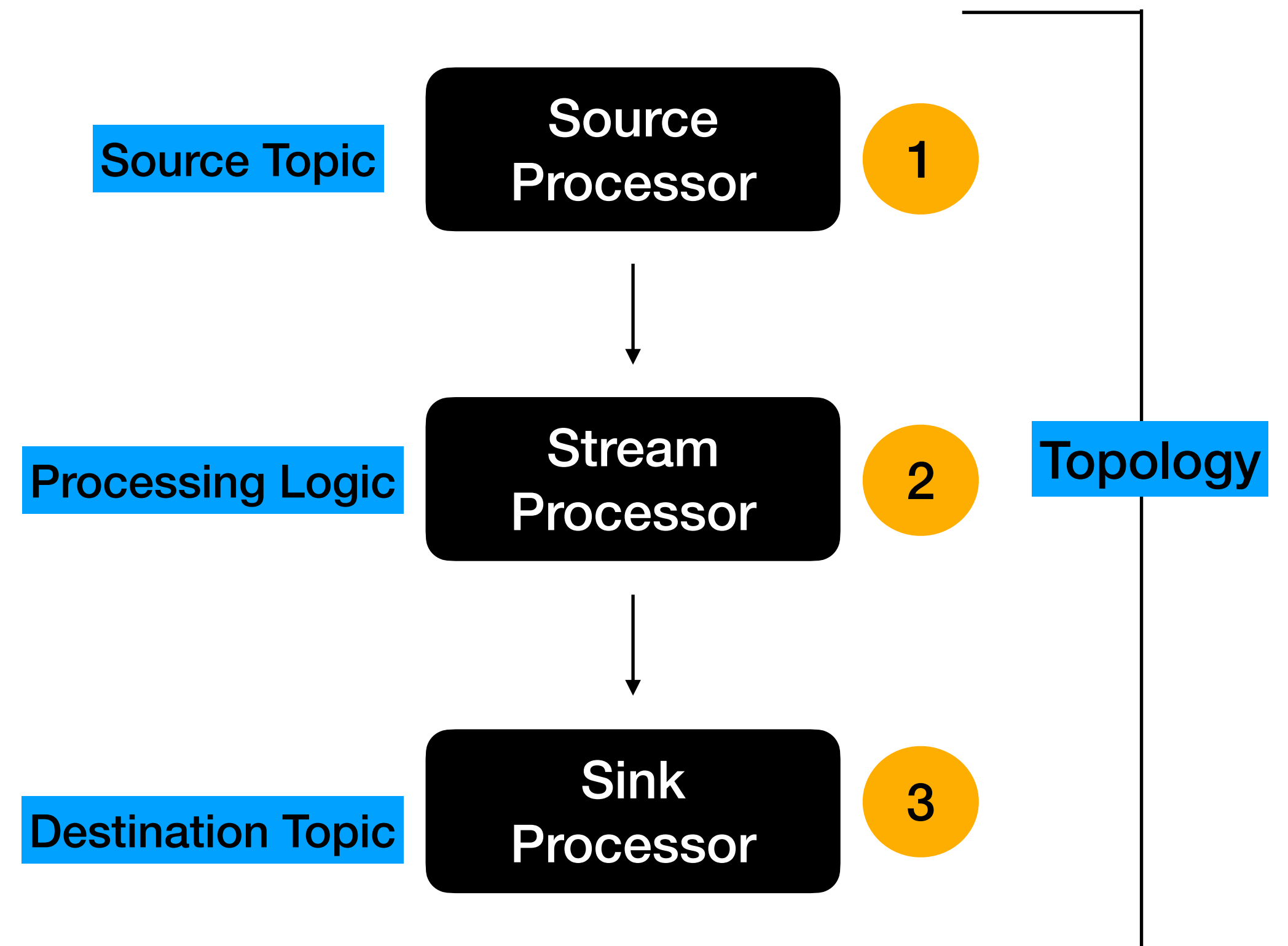
- Streams DSL
 - This is a high level API
 - Use operators to build your stream processing logic
- Processor API
 - This is a low level API
 - Complex compared to Streams DSL
 - Streams DSL is built on top of Processor API

Kafka Streams Terminologies

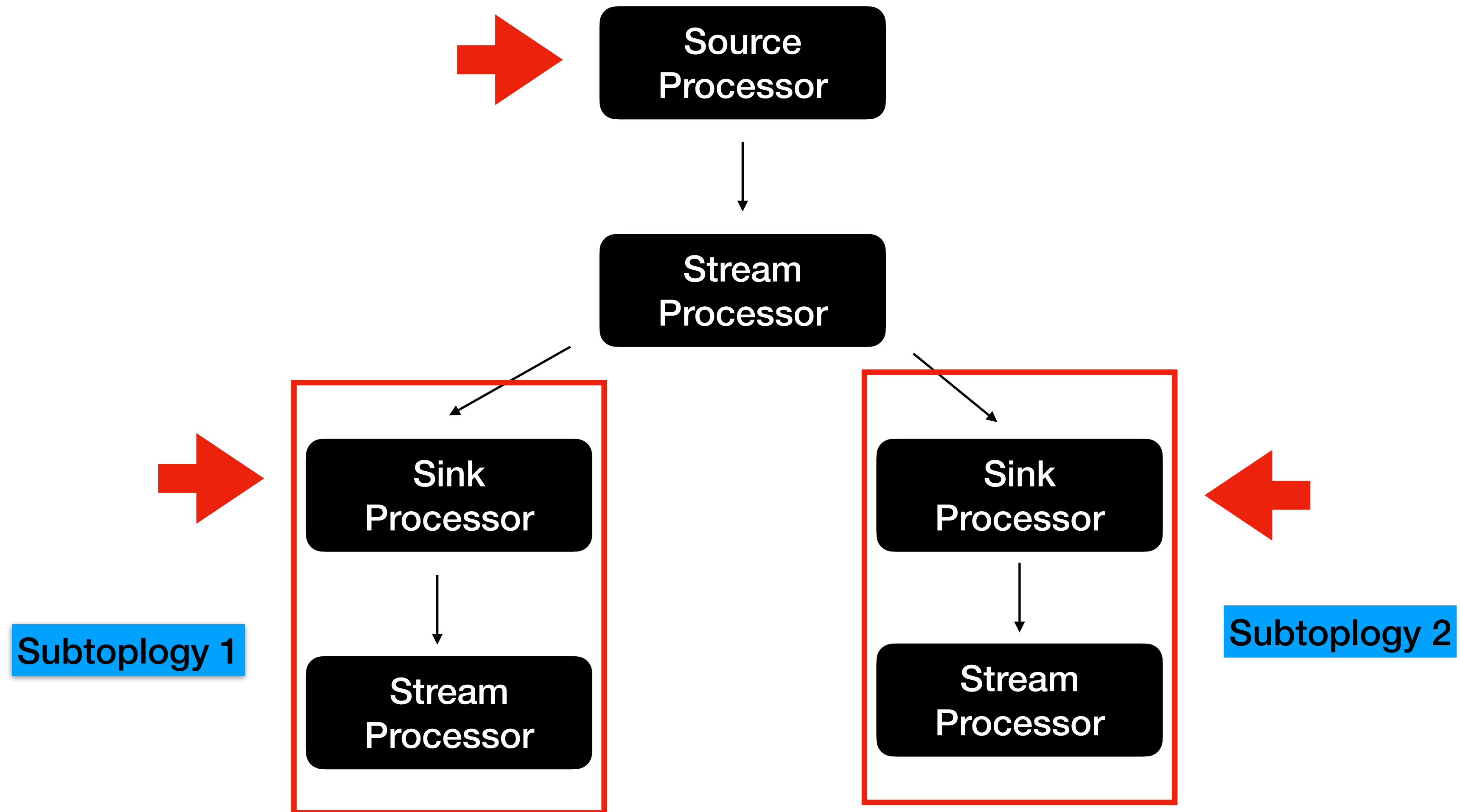
Topology & Processor

Topology & Processors

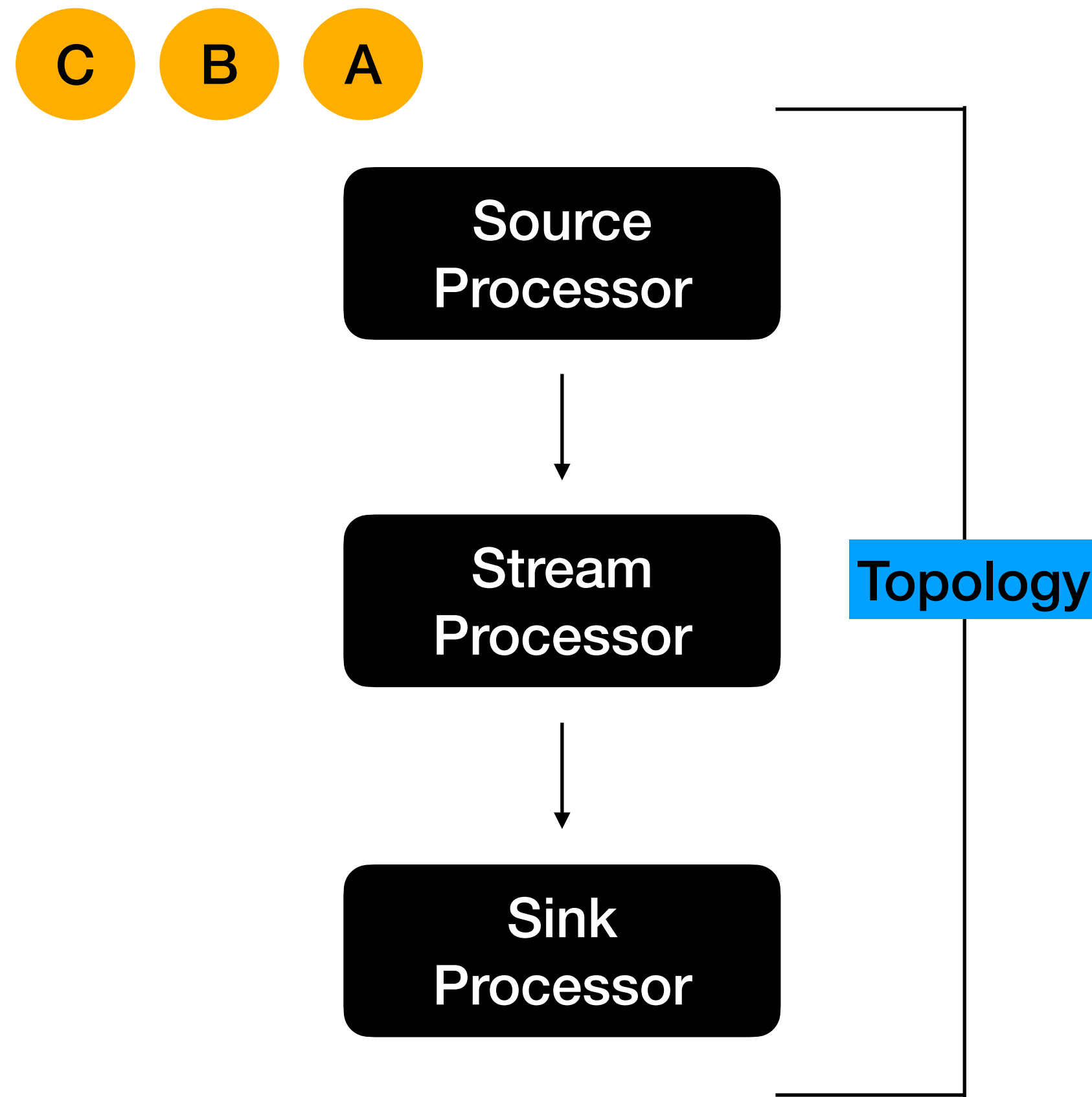
- A Kafka stream processing application has a series of processors.
- The concept of designing the Kafka Streams application in this approach is basically a **Directed Acyclic Graph (DAG)**.
- The collection of processors in a Kafka Streams is called a **Topology**.



Topology and SubTopology

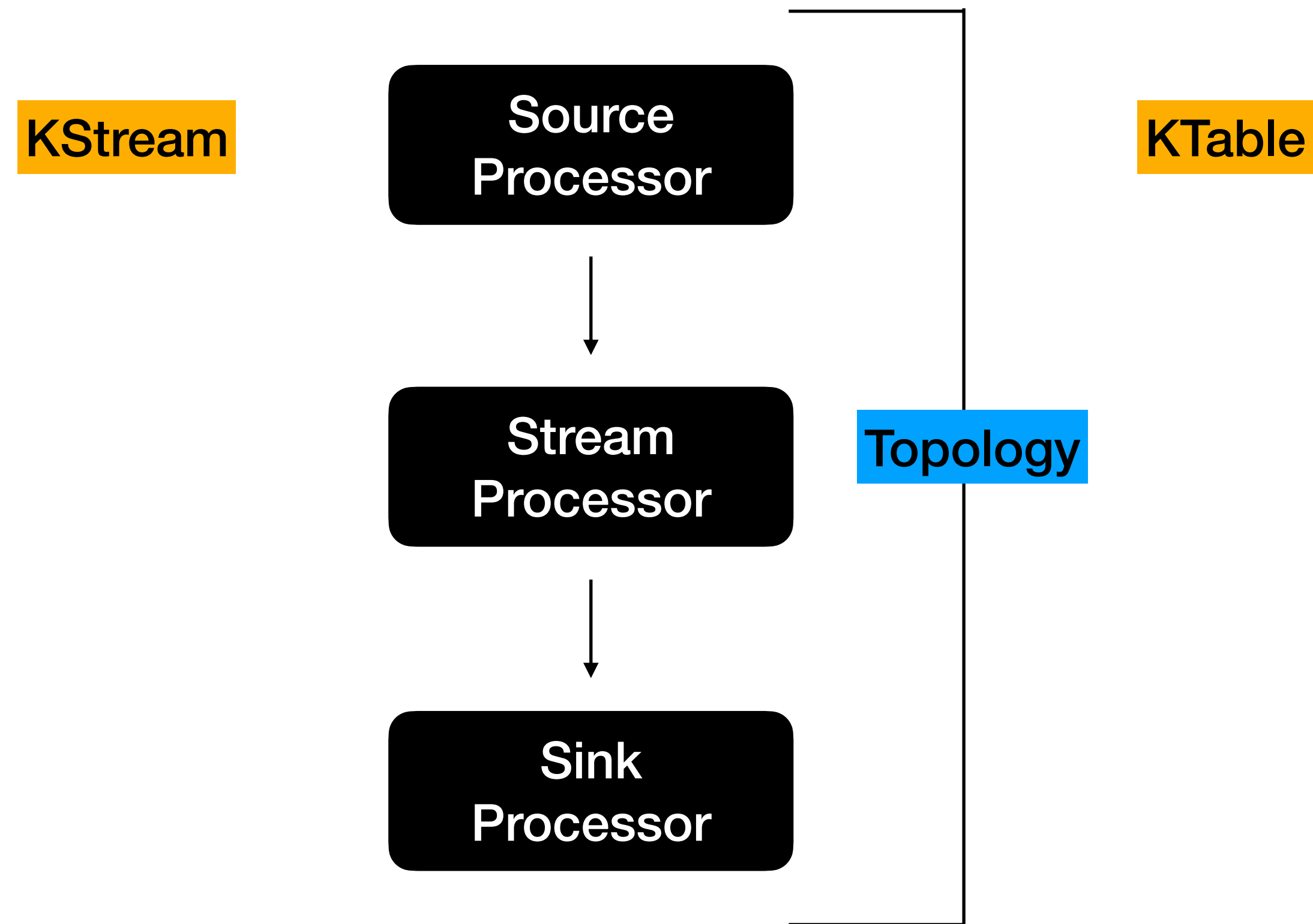


How the Data Flows in a Kafka Streams Processing ?



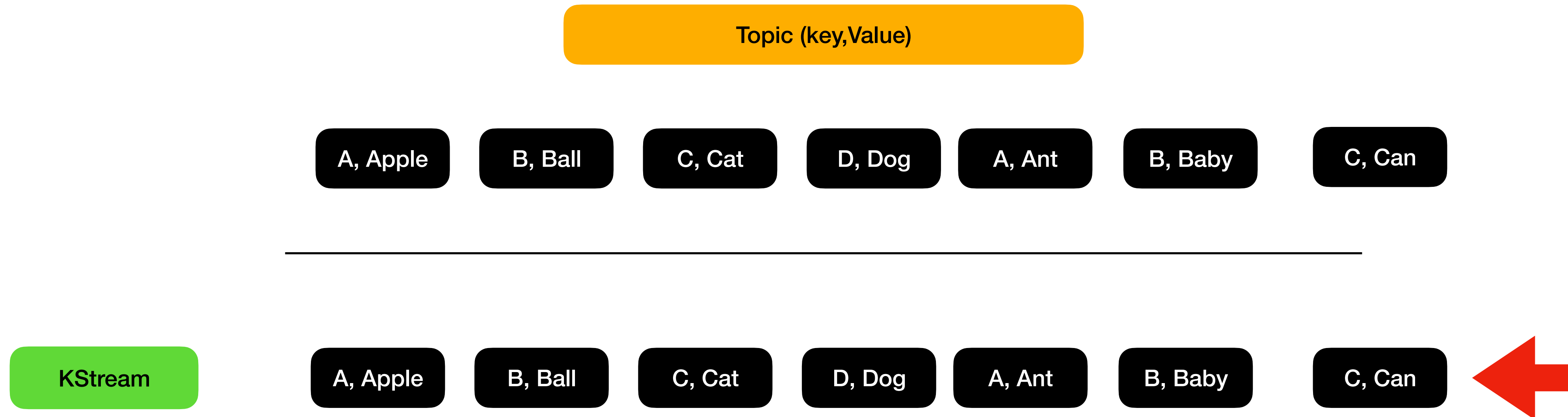
- At any given point of time, only one record gets processed in the Topology.
- With Subtopology, this rule is applicable to each subtopolgy

Introduction to KStreams API



KStream

- **KStream** is an abstraction in Kafka Streams which holds each event in the Kafka topic



- KStream gives you access to all the records in the Kafka topic.
- KStream treats each event independent of one another.
- Each event will be executed by the whole topology.
- Any new event is made available to the **KStream**
- KStream can be called as **record stream** or **log**
- **Record Stream** is infinite
- Analogy : Inserts into a DB table

Greeting Kafka Streams App




Prerequisites for Greetings App

- Kafka environment
- Topology

filter

- filter
 - This operator is used to drop elements in the Kafka Stream that does not meet a certain criteria.
 - This operator takes in a **Predicate** Functional Interface as an input and apply the predicate against the input.

```
var upperCaseStream = greetingsStream  
     .filter((key, value) -> value.length()>5)  
    .mapValues((readOnlyKey, value) -> value.toUpperCase());
```

Allow the stream events whose length is greater than 5

filterNot

- filterNot
 - Functionality of this operator is the the opposite of **filter** operator.

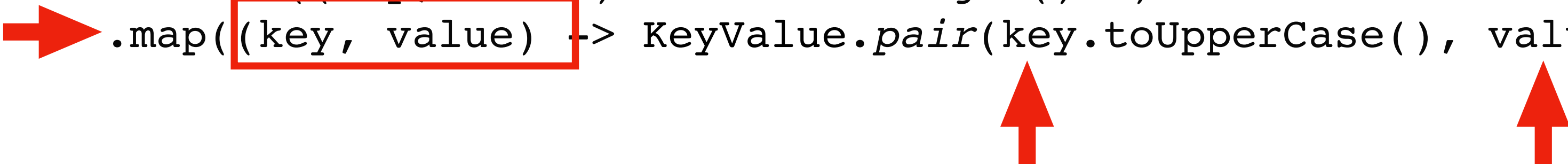
```
var upperCaseStream = greetingsStream
    → .filterNot((key, value) -> value.length()>5)
       .mapValues((readOnlyKey, value) -> value.toUpperCase());
```

Allow the stream events whose length is lesser than 5.

map

- This operator is used when we have to transform the key and value to another form.

```
var upperCaseStream = greetingsStream
    .filter((key, value) -> value.length()>5)
    ➔ .map((key, value) -> KeyValue.pair(key.toUpperCase(), value.toUpperCase()));
```



mapValues

- This is used when we have a usecase to transform just the values in the Kafka Stream.

```
var upperCaseStream = greetingsStream
    .filter((key, value) -> value.length()>5)
    ➡ .mapValues((readOnlyKey, value) -> value.toUpperCase())
```

flatMap

- This operator can be used when a single event is going to create multiple possible events downstream.

Apple → A P P L E

```
var upperCaseStream = greetingsStream
    .filter((key, value) -> value.length()>5)
    .flatMap((key, value) -> {
        1 var newValue = Arrays.asList(value.split(""));
        2 var keyValueList = newValue
            .stream().map(t -> KeyValue.pair(key.toUpperCase(), t))
            .collect(Collectors.toList());
        return keyValueList;
    })
```

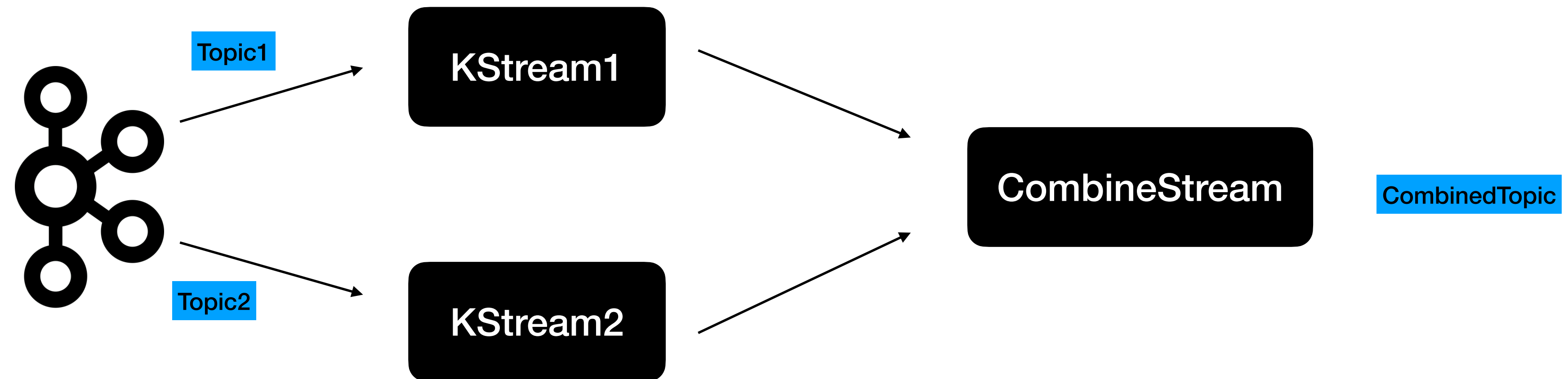
flatMapValues

- Very similar to flatMap but we just access and change the values.

```
var upperCaseStream = greetingsStream
    .filter((key, value) -> value.length()>5)
    ➔ .flatMapValues((readOnlyKey, value) -> {
        var newValue = Arrays.asList(value.split(""));
        return newValue;
    });
```

merge

- This operator is used to combine two independent Kafka Streams into a single Kafka Stream

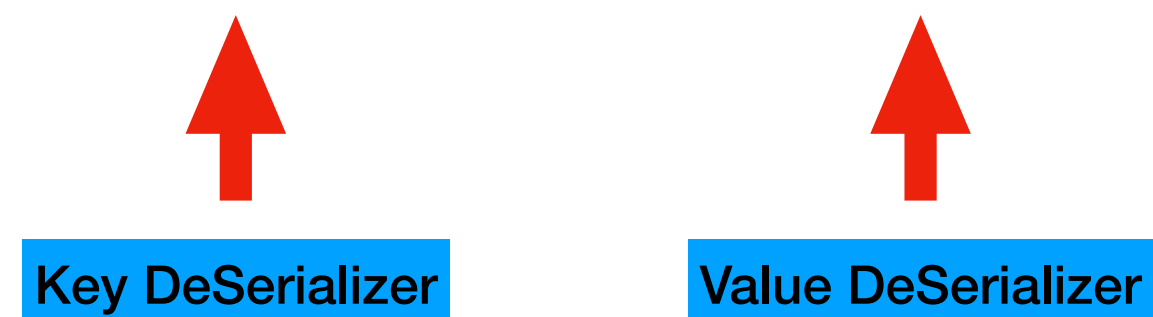


Serialization/Deserialization in Kafka Streams API

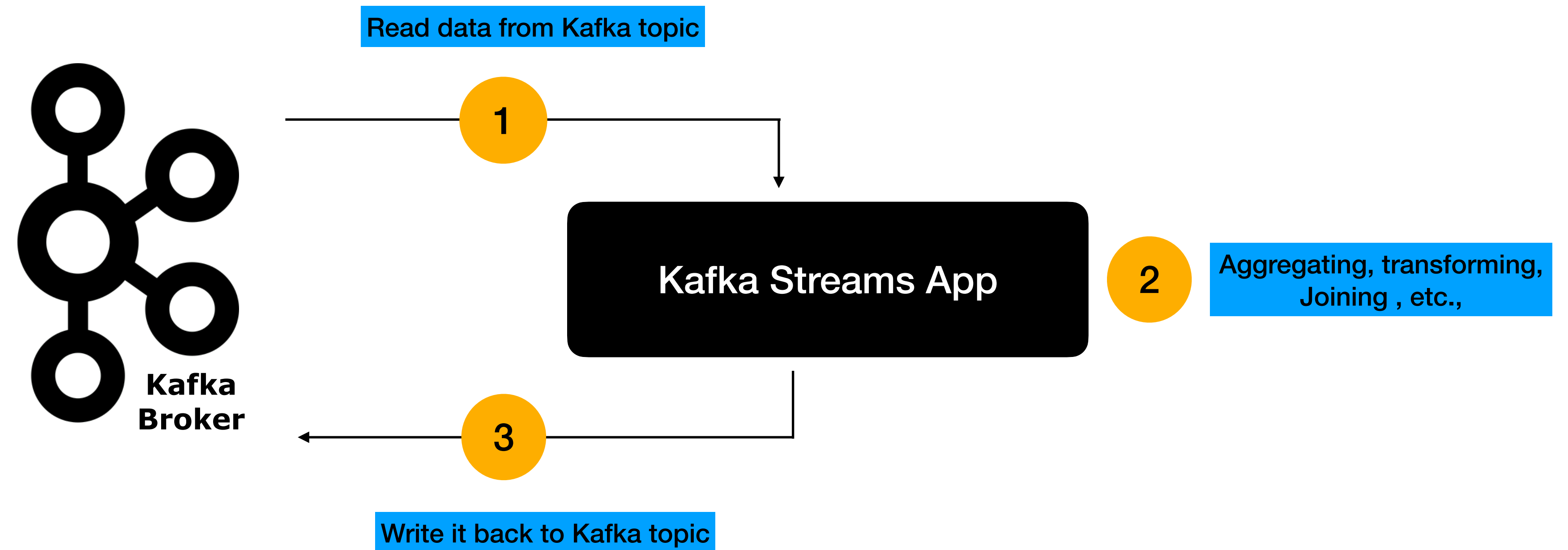
Serdes

- Serdes is the factory class in Kafka Streams that takes care of handling serialization and deserialization of key and value.

```
var greetingsStream = streamsBuilder.stream(GREETINGS,  
Consumed.with(Serdes.String(), Serdes.String()));
```




Data Flow in a Kafka Streams App

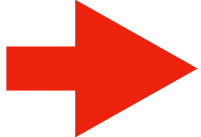


Behind the scenes Streams API uses the producer and consumer API to read and write it back to Kafka topic

Enhanced Greeting Message



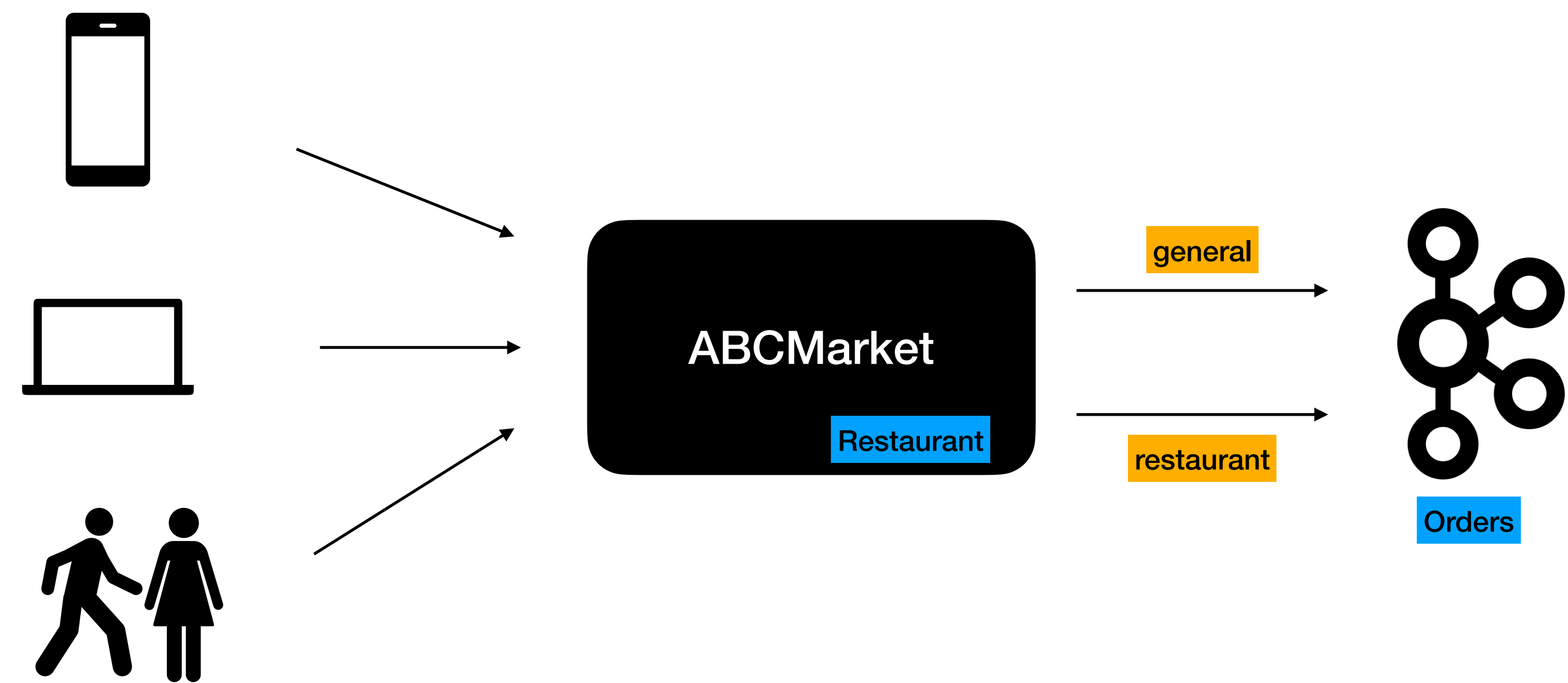
```
{  
  "message": "Good Morning",  
  "timeStamp": "2022-12-04T05:26:31.060293"  
}
```



What's needed to build a Custom Serde ?

- Serializer
- Deserializer
- **Serde** that holds the Serializer and Deserializer

Overview of the retail App



Data Model for the Order

General Order

```
{
  "orderId": 12345,
  "locationId": "store_1234",
  "finalAmount": 27.00,
  "orderType": "GENERAL",
  "orderLineItems": [
    {
      "item": "Bananas",
      "count": 2,
      "amount": 2.00
    },
    {
      "item": "Iphone Charger",
      "count": 1,
      "amount": 25.00
    }
  ],
  "orderedDateTime": "2022-12-05T08:55:27"
}
```

Restaurant Order

```
{
  "orderId": 12345,
  "locationId": "store_1234",
  "finalAmount": 15.00,
  "orderType": "RESTAURANT",
  "orderLineItems": [
    {
      "item": "Pizza",
      "count": 2,
      "amount": 12.00
    },
    {
      "item": "Coffee",
      "count": 1,
      "amount": 3.00
    }
  ],
  "orderedDateTime": "2022-12-05T08:55:27"
}
```

New Business requirements for our app

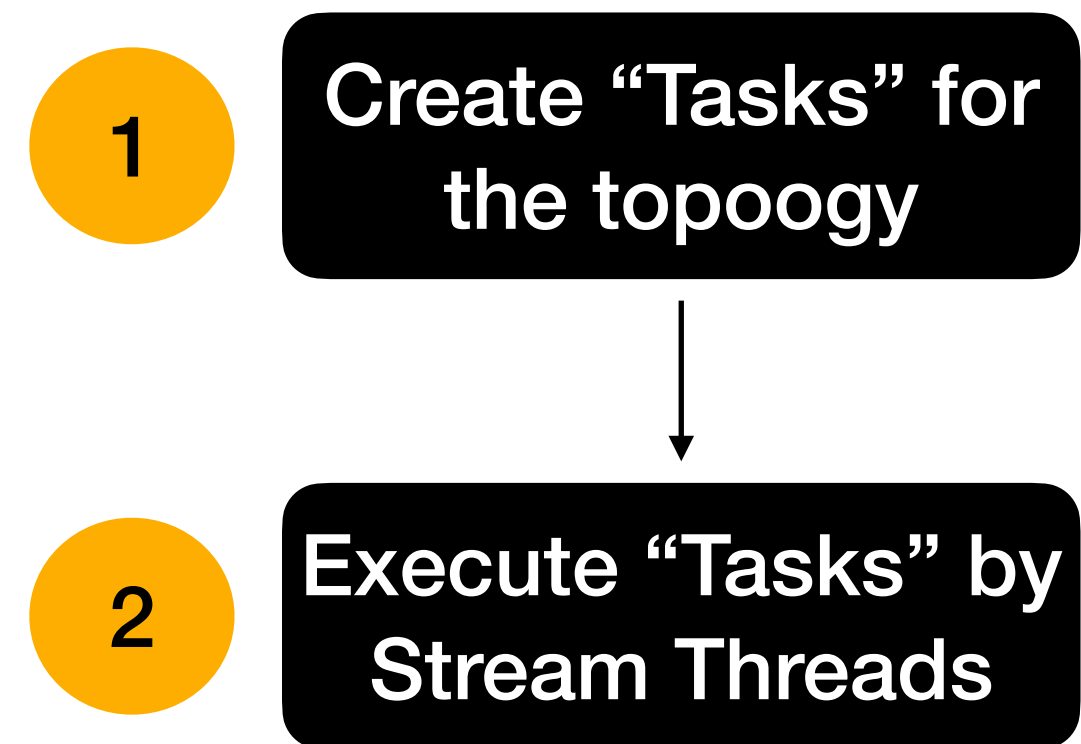
- Split the restaurant and general orders and publish them into different topics
- Just Publish the Transaction amount of these two types of Orders
 - Transform the Order type to Revenue Type

Kafka Streams - Internals

Kafka Streams - Topology, Task & Threads

- **Topology** in Kafka Streams means its a **data pipeline or template** on how the data is going to be flow or handled in our Kafka Streams application.
- A **Topology** in itself does not execute on its own.

How Kafka Streams executes the Topology ?



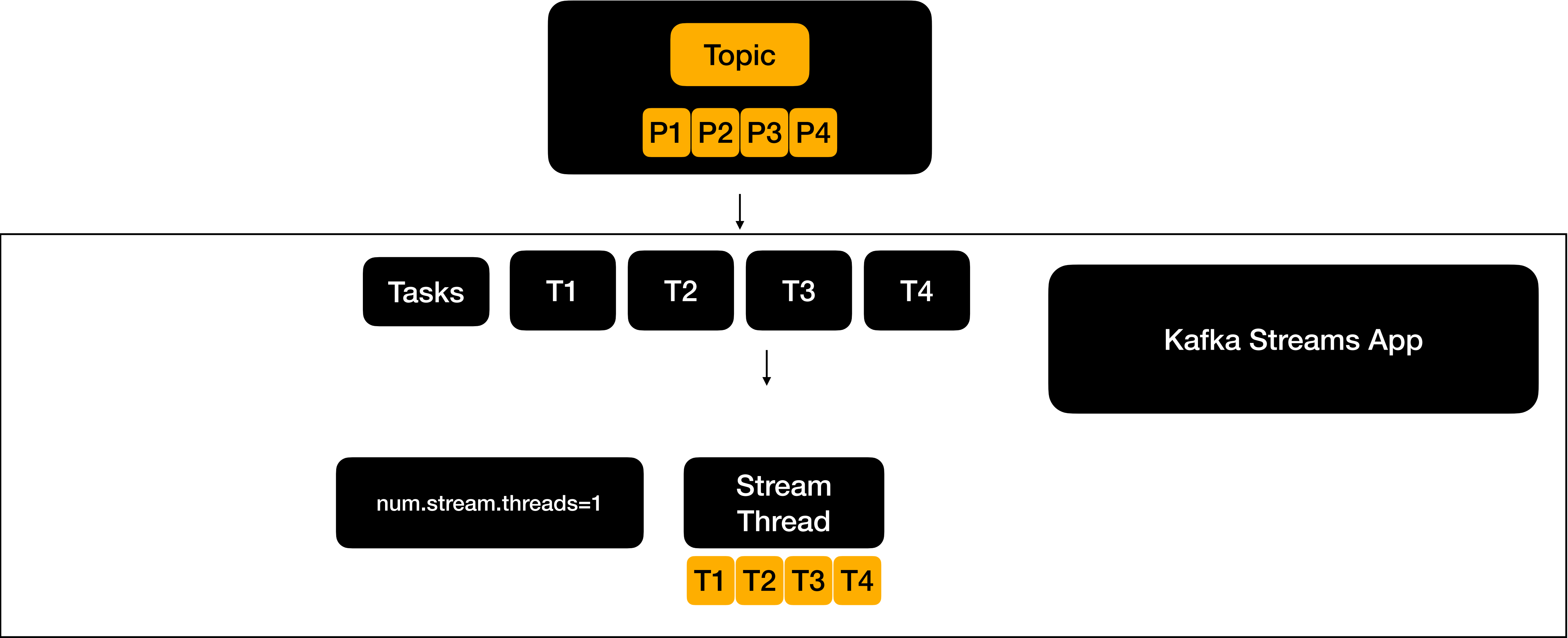
Tasks in Kafka Streams

- A **Task** is basically the unit of work in a Kafka Streams application
- How many Tasks can be present in a Kafka Streams Application?
 - The maximum number of tasks is determined by the number of partitions of the source topic.
 - **For Example**, If the source topic has four partitions then we will have four tasks created behind the scenes for the Kafka Streams application
- Each **Task** can be executed in parallel by the Kafka Streams application

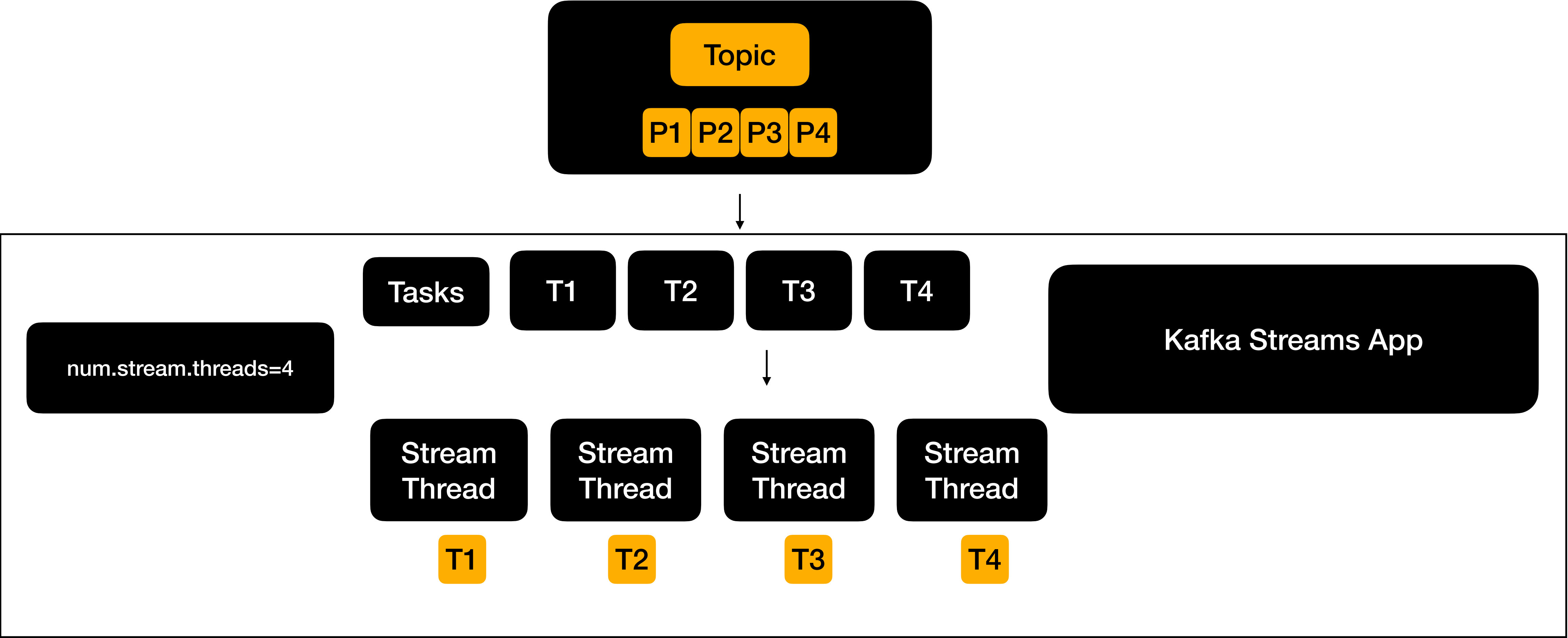
Threads in Kafka Streams

- **Threads** or **Stream Threads** are the ones that execute the Tasks in a Kaka Streams application.
- By default the number of stream threads is 1.
 - This value is controlled by the **num.stream.threads** property.
 - You can modify it based on how many tasks you want to execute in parallel.

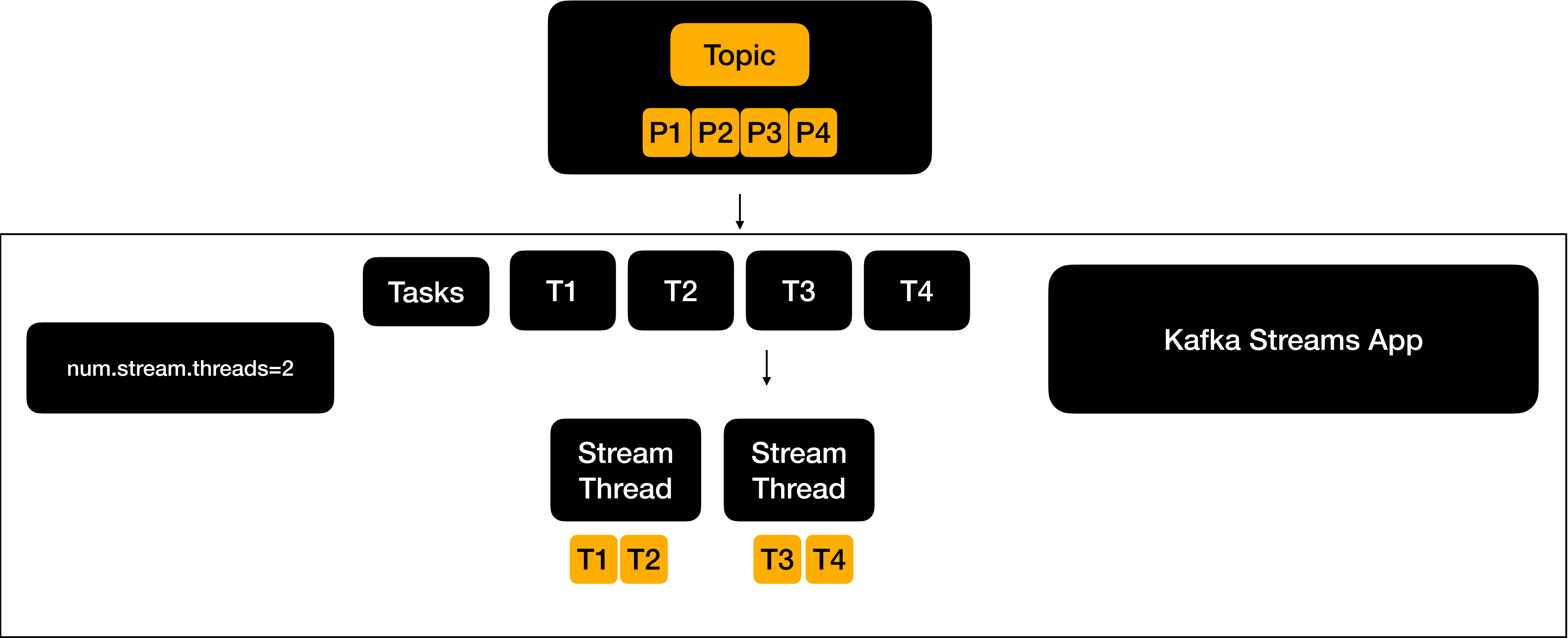
Kafka Streams - Default Stream Threads



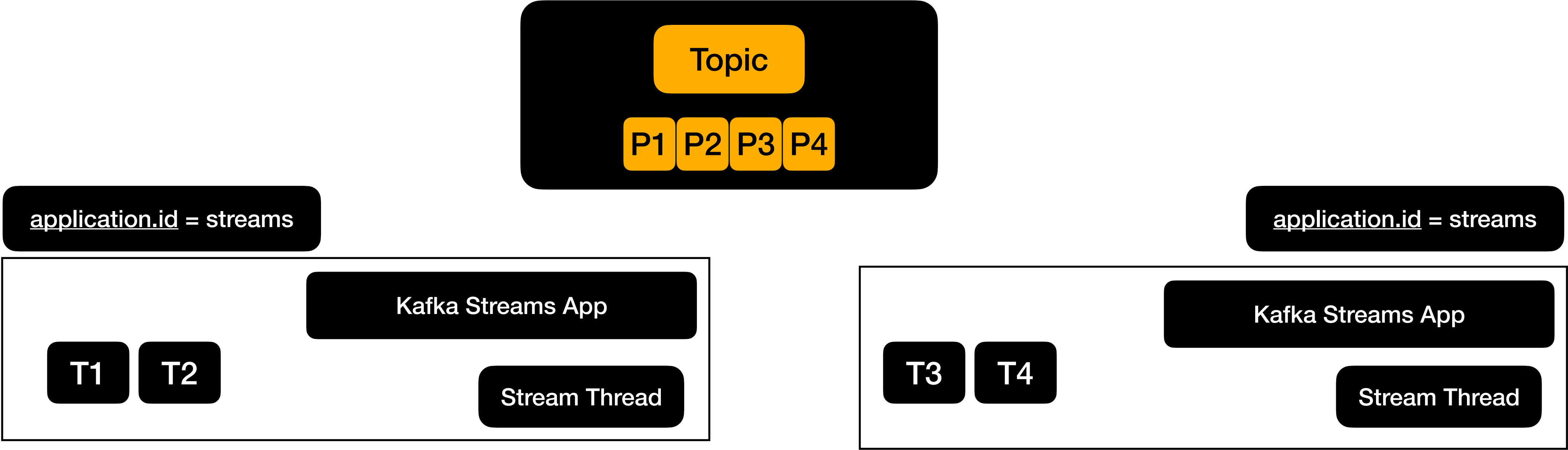
Kafka Streams - Parallelism - Approach 1



Kafka Streams - Parallelism - Approach 1



Kafka Streams - Parallelism - Approach 2



What's the ideal number of Stream Threads ?

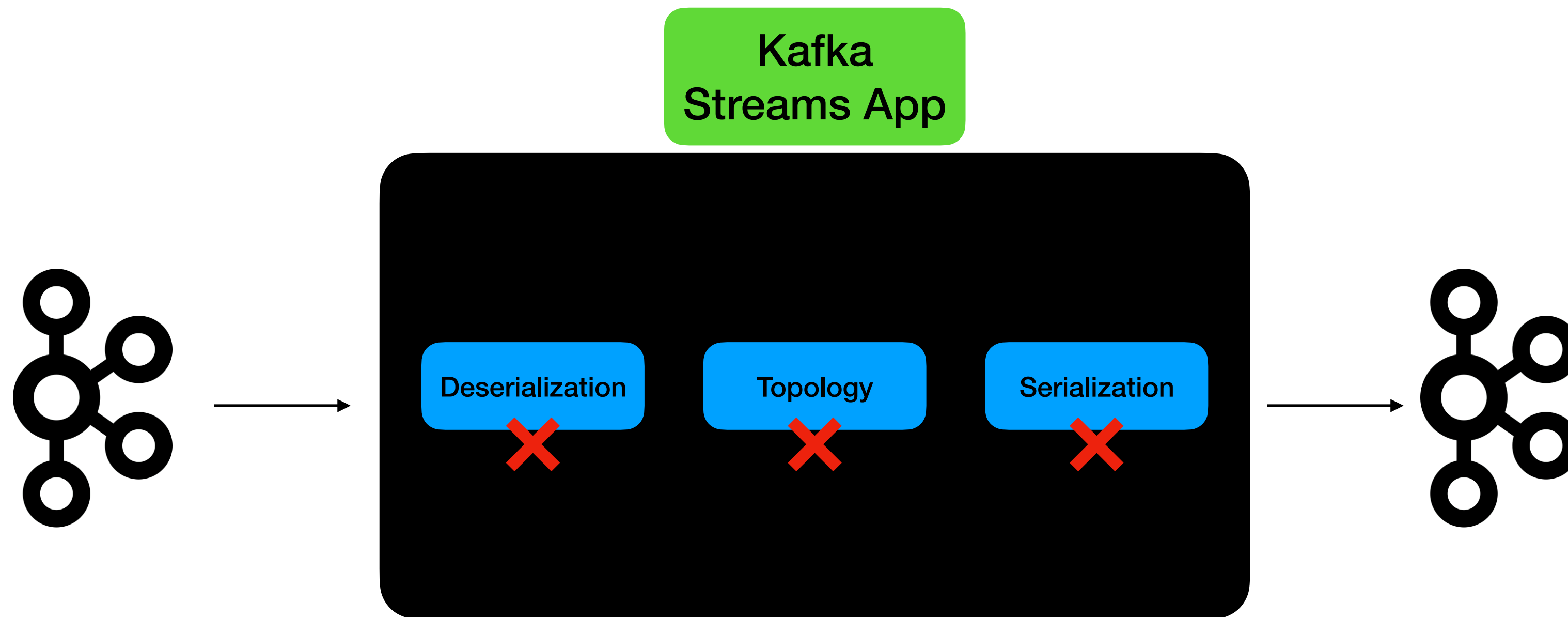
- Keep the Stream threads size equal to the number of cores in the machine.

```
Runtime.getRuntime().availableProcessors()
```

Kafka Streams & Consumer Groups

- Kafka Streams behind the scenes uses consumer api to stream from a topic.
- application.id is equivalent to group.id in Kafka Consumer.
- Kafka Streams gets all the benefits of consumer groups.
 - Tasks are split across multiple instances using the consumer group concept.
 - Fault Tolerance : If one instance goes down then rebalance will be triggered to redistribute the tasks.

Errors In Kafka Streams



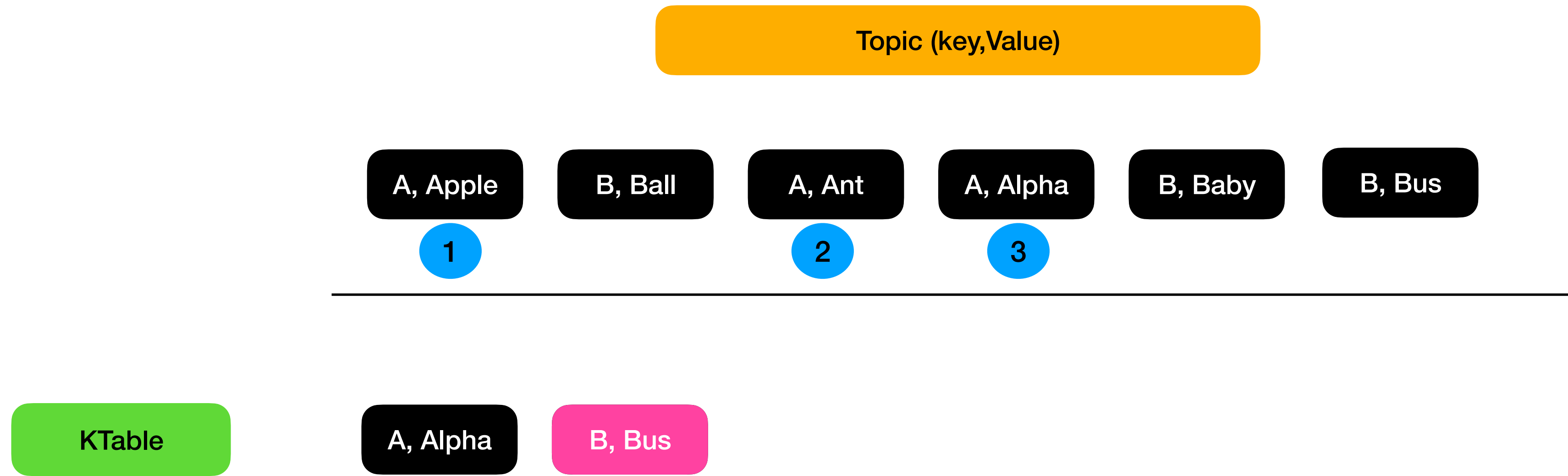
- Deserialization or Transient Errors at the entry level.
- RuntimeExceptions in the Application Code(Topology).
- Serialization or Transient Errors when producing the data.

ErrorHandlers in Kafka Streams

Error	Error Handler
Deserialization	DeserializationExceptionHandler ←
Application Error (Topology)	StreamsUncaughtExceptionHandler ←
Serialization	ProductionExceptionHandler ←

KTable

- **KTable** is an abstraction in Kafka Streams which holds latest value for a given key.



- KTable is also known as **update-stream** or a **change log**.
- KTable represents the latest value for given key in a Kafka Record.
- Record that comes with the same key updates the previous value.
- Any Record without the Key is ignored.
- Analogy (Relational DB):
 - You can think of this as an update operation in the table for a given **primary-key**

How to create a KTable ?

```
1 usage
public static Topology build(){
    StreamsBuilder streamsBuilder = new StreamsBuilder();

    var wordsTable : KTable<String, String> = streamsBuilder
        .table( topic: "words", Consumed.with(Serdes.String(), Serdes.String())
            , Materialized.as( storeName: "words-store" )
        );
```

- StateStore is necessary for storing the data in an additional entity.
 - **Materialized.as("words-store") creates a stateStore.**
 - This is needed to retaining the data in an app crash or restart or deployment. ←
- The default state store is **RocksDB**.
 - RocksDB is a high performance embedded database for key-value data.
 - This is an open source DB under the Apache 2.0.
 - Data in the embedded key-value store is also persisted in a file system.
- Data in the Rocks DB also maintained in a changelog topic for **Fault Tolerance**.

When to use a KTable?

- Any business usecase that requires the streaming app to maintain the latest value for a given key can benefit from KTable.
- Example
 - Stock Trading App that requires to maintain the latest value for a given Stock symbol.

KTable - Under the Hood

- When does KTable decides to emit the data to the downstream nodes?
- There are two configurations that controls this:
 - `cache.max.bytes.buffering`
 - `commit.interval.ms`

cache.max.bytes.buffering

- **KTable** uses a cache internally for deduplication and the cache serves as buffer.
 - This buffer holds the previous value, any new message with the same key updates the current value.
- Caching also helps with the amount of data written into the RocksDB
- `cache.max.bytes.buffering = 10485760(~10MB)`
- So, if the buffer size is greater than the **cache.max.bytes.buffering** value then the data will be emitted to the downstream nodes

commit.interval.ms

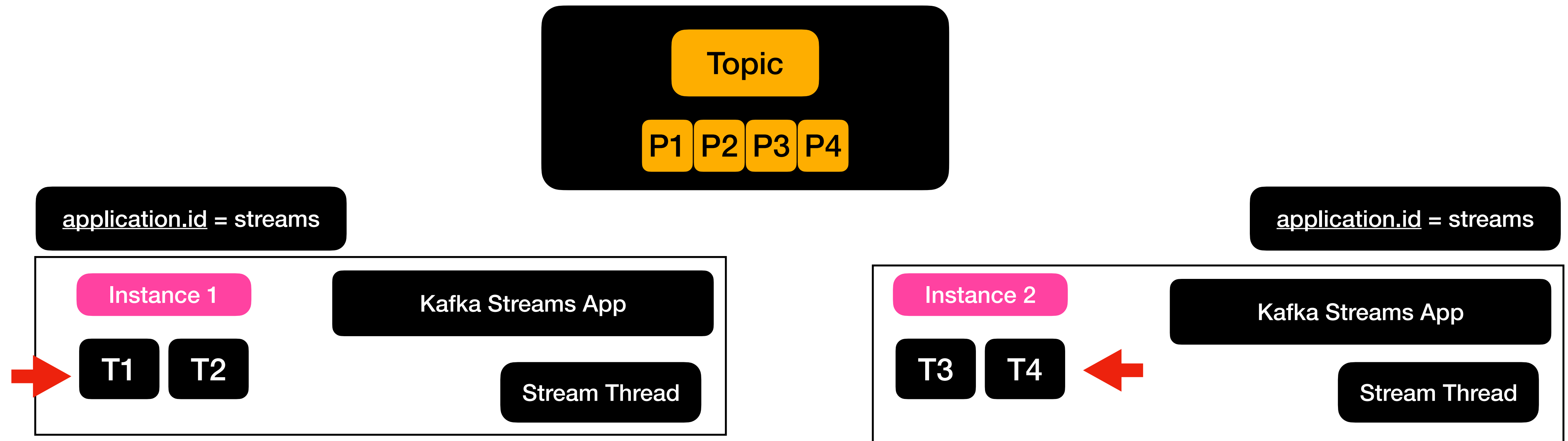
- commit.interval.ms = 30000 (30 seconds)
- **KTable** emits the data to the downstream processor nodes once the commit interval is exhausted.

GlobalKTable

```
var wordsGlobalTable = streamsBuilder
    ➔ .globalTable("words", Consumed.with(Serdes.String(), Serdes.String()),
        , Materialized.as("words-global-store")
    );
```

- GlobalKTable is very similar to KTable.
 - This also stores the latest value for a given key.
- The operations are very limited in a GlobalKTable.

KTable vs GlobalKTable



- KTable will have the keys distributed between these two instances.
- The reason being we have tasks split between them.
- Instance 1 has access to only keys that are tied to Task T1 and T2.
- Instance 2 has access to only keys that are tied to Task T3 and T4.

- An instance of GlobalKTable in each of these instance will have access to all the keys in each instance.
- Instance 1 and Instance 2 has access to all the keys in the topic.

When to use KTable vs GlobalKTable?

- KTable is an advanced API and it supports many operators to enrich the data.
- Use KTable if you have a larger **KeySpace**.
 - This means if you have a huge number of Keys(may be in **Millions**).
 - **KTable** is better for joins where time needs to be synchronized.
- Use **GlobalKTable** if the **KeySpace** is smaller.

Stateful operations in Kafka Streams

- Stateful Operations are one of the awesome feature of Kafka Streams.
- What are the stateful operations that can be performed in Kafka Streams ?
 1. Aggregation of Event Streams
 - Calculating total number of orders made in a retail company.
 - Calculating total revenue made for the company.
 2. Joining Event Streams
 - Combining data from two independent topics(event streams) based on a key.
 3. Windowing
 - This is the concept of Grouping Data in a certain time window.
 - **Example** : Calculate the number of orders made in an hour or a day or a week or a month.

Stateful Operators in Kafka Streams

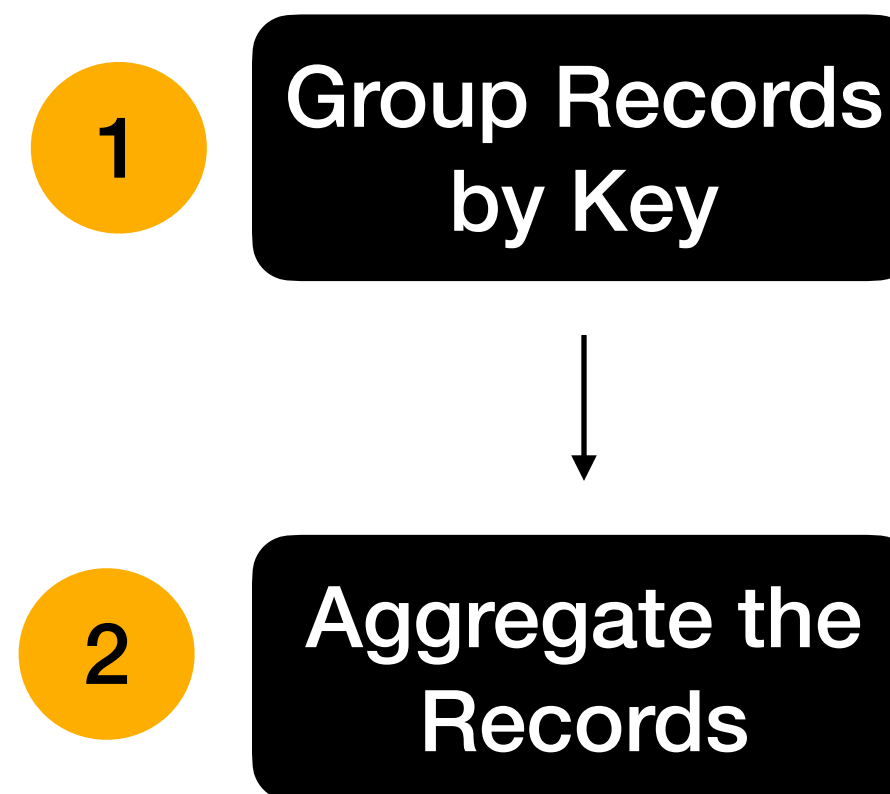
- Aggregation of Data
 - count , reduce and aggregate
- Joining Data
 - join, leftJoin, outerJoin
- Windowing Data
 - windowedBy

Aggregation of Data

- Aggregation is the concept of combining multiple input values to produce one output.
- Some of the example usecases:
 - Calculating total number of orders made by the company.
 - Calculating total revenue made by the company.

How Aggregation works ?

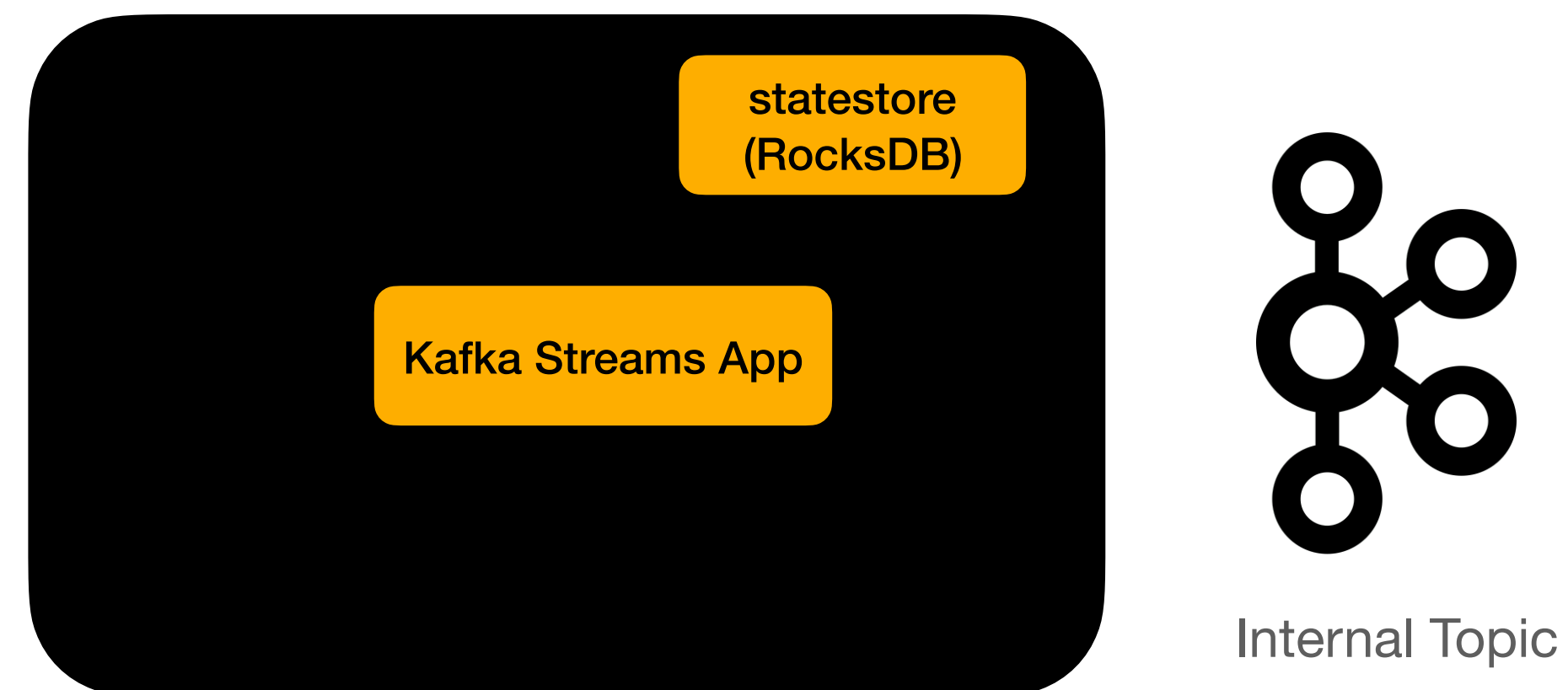
- Aggregations works only on Kafka Records that has **non-null** Keys



- There are two configurations that controls this:
 - `cache.max.bytes.buffering`
 - `commit.interval.ms`

Results of Aggregation

- Aggregation results are present in the StateStore and internal Kafka Topic.
- Aggregated data needs to be made available to the outside world or the partner teams looking for this data.



Aggregation operators in Kafka Streams

- We have three operators in **Kafka Streams** library to support aggregation:
 - count
 - reduce
 - aggregate

count Operator

- This is used to count the number of different events that share the same key.

```
var groupedString = wordsTable
```

```
➔ .groupByKey(Grouped.with(Serdes.String(), Serdes.String()))
```

```
➔ .count(Named.as("count-per-alphabet"));
```

- This will return a KTable of this type KTable<String, Long>.
 - String -> Key of the record
 - Long -> Value of the record
- The aggregated value gets stored in an internal topic.

count

Topic (key,Value)	A, Apple	A, Alpha	B, Bus
-------------------	----------	----------	--------



Count	[A : 1]	[A : 2] [B : 1]
-------	---------	-----------------

reduce operator

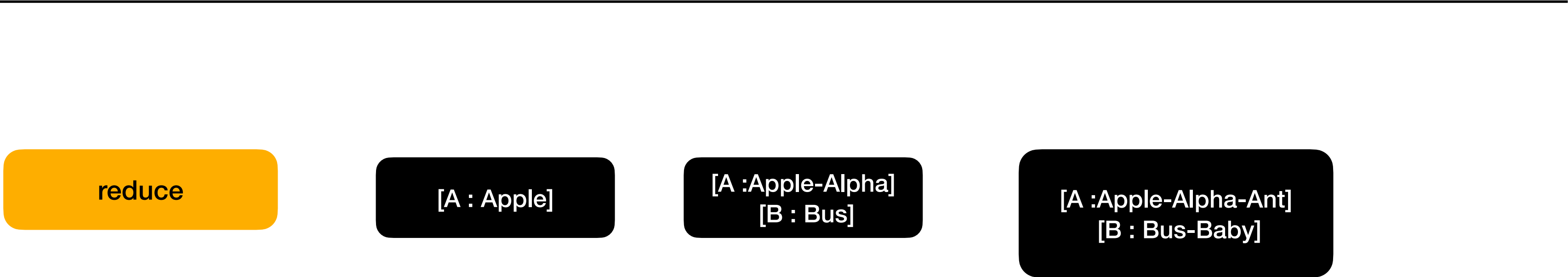
- This operator is used to reduce multiple values to a single value that shares the same key.

```
var groupedString = wordsTable
    →.groupByKey(Grouped.with(Serdes.String(), Serdes.String()))
    →.reduce((value1, value2) -> {
        log.info("value1 : {} , value2 : {} ", value1, value2);
        → return value1.toUpperCase()+"-"+value2.toUpperCase();
    });
```

Concatenate each event by a hyphen (-) .

- The return type of the reduce operator must be the same type as the value.
- The aggregated value gets stored in an internal topic.

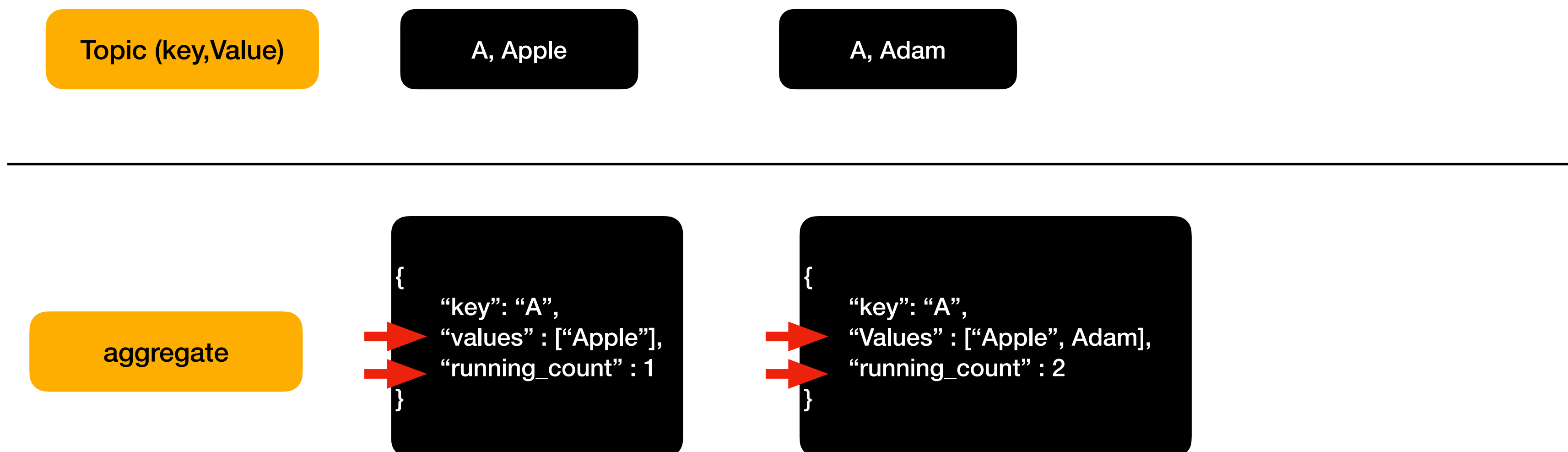
Reduce



Use this operator if you have a use case to continuously aggregate the events in this fashion.

aggregate

- An aggregate operator is similar to **reduce** operator.
- The aggregated type can be different from actual Kafka Records that we are acting on.



- The aggregated value gets stored in an internal topic.

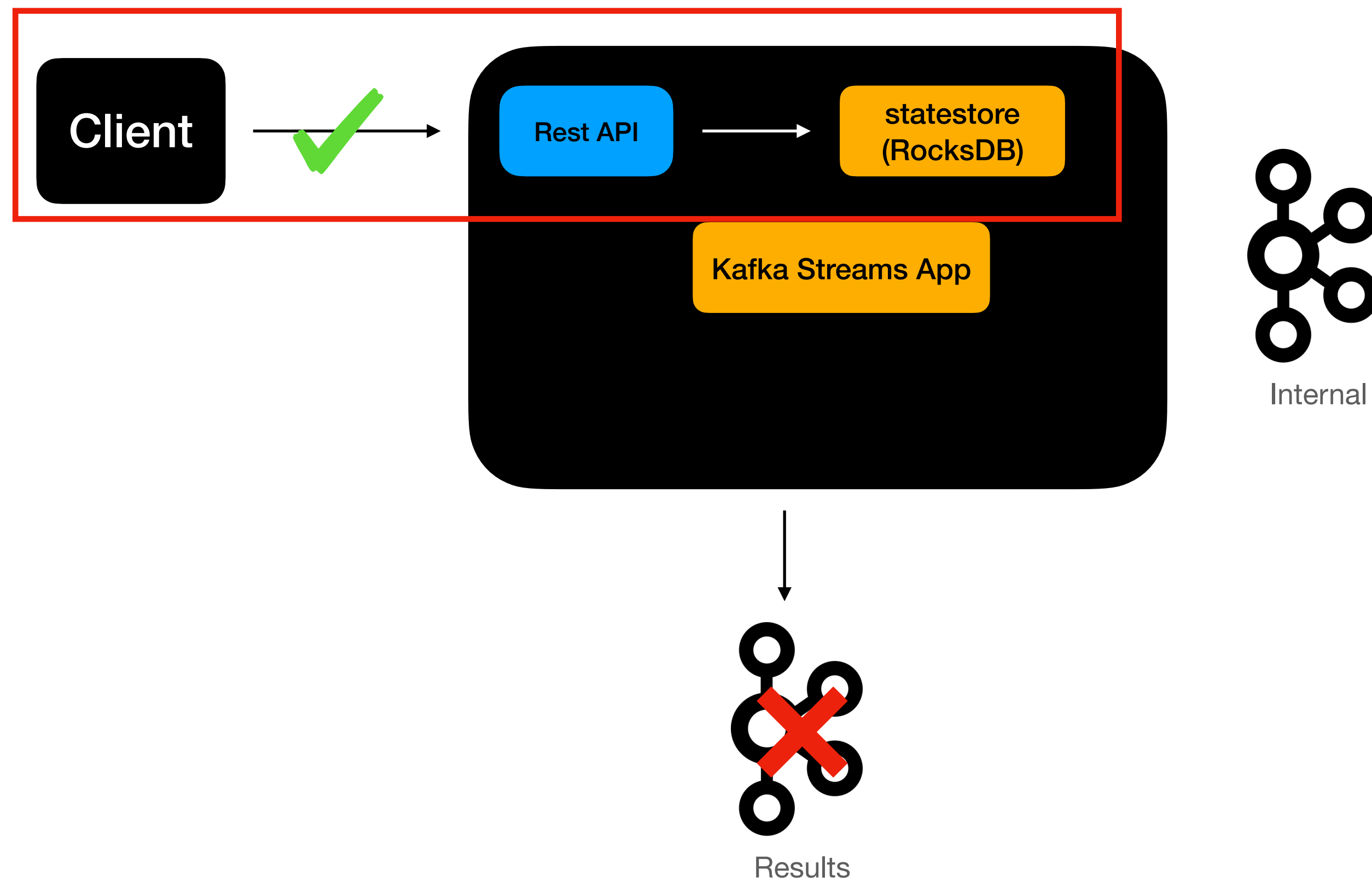
aggregate

```
Initializer<AlphabetWordAggregate> alphabetWordAggregateInitializer = AlphabetWordAggregate::new;
```

```
Aggregator<String, String, AlphabetWordAggregate> aggregator =  
    (key,value, alphabetWordAggregate )-> {  
        ➡ return alphabetWordAggregate.updateNewEvents(key, value);  
    };  
    ↑
```

```
var aggregatedStream = groupedString  
    .aggregate(  
        ➡ alphabetWordAggregateInitializer,  
        ➡ aggregator,  
        ➡ Materialized  
            .<String, AlphabetWordAggregate,  
              KeyValueStore< Bytes, byte[]>>as("aggregated-words")  
            .withKeySerde(Serdes.String())  
            .withValueSerde(SerdesFactory.alphabetWordAggregate())  
    );
```

Sharing results of Aggregated Data

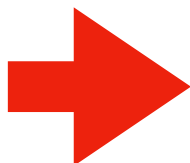


Aggregation in OrderManagement Service

- Count the Total number of orders.
- Total Revenue made from the orders

selectKey operator

- The primary role of this operator is to **rekey** the Kafka records.

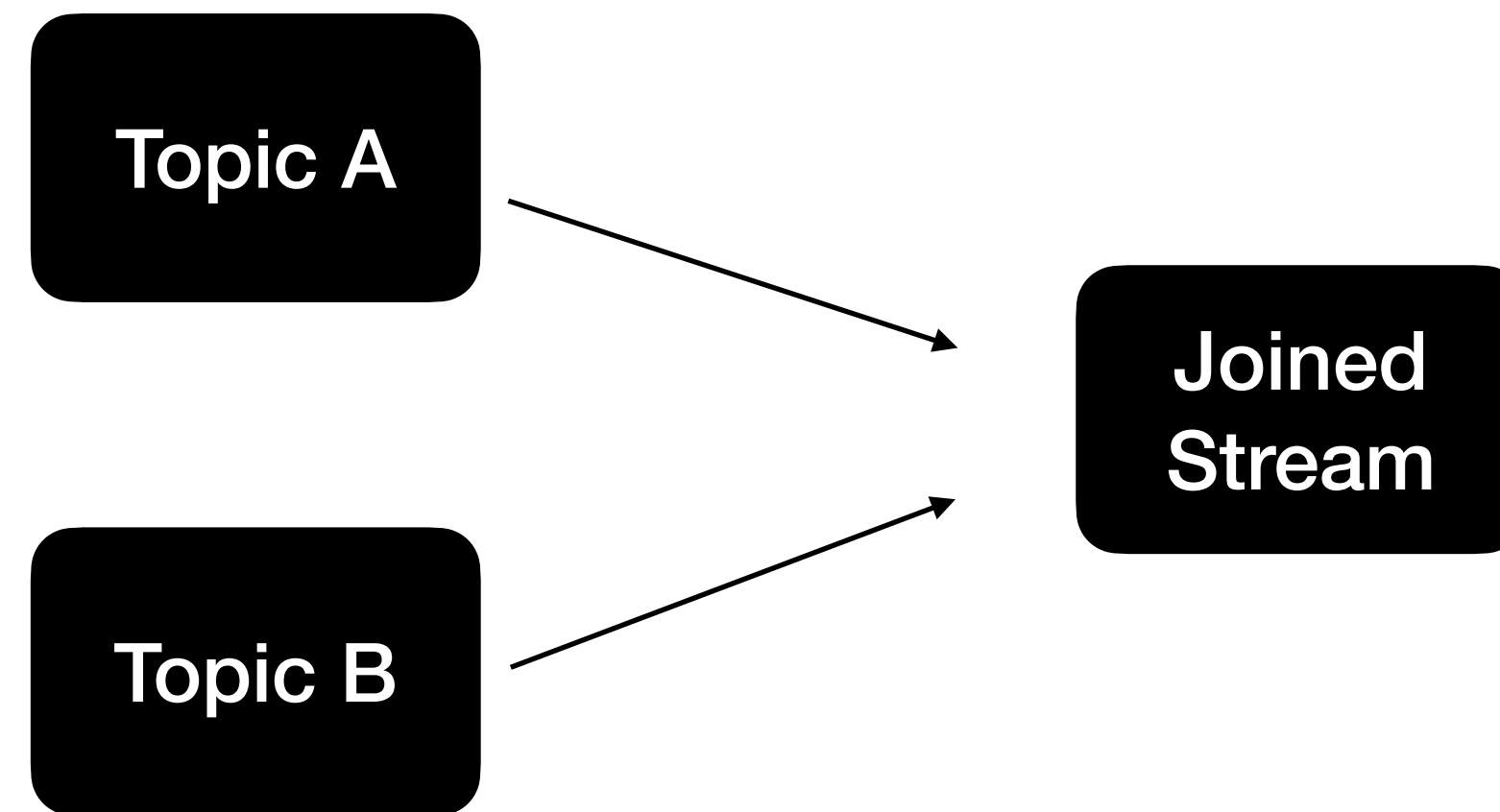
```
var ordersStream = streamsBuilder
    .stream(ORDERS,
           Consumed.with(Serdes.String(), SerdesFactory.orderSerdes()))
     .selectKey((key, value) -> value.locationId());
```

- selectKey call will trigger the repartition of the records.
 - Internal re-partition topic gets created and records will be persisted in the topic for repartition.
 - Streams task needs to read these messages again for further processing

The effect is very similar to using “map” operation with “groupBy” that I explained in the previous lecture.

Joining in Kafka Streams

- Joins in Kafka Streams are used to combine data from multiple topics.
- Analogy : Joining Data from multiple tables in a **Relational DB**.



- Joins will only be performed if there is a matching key in both the topics.
- How is this different from **merge** operator?
 - Merge operator has no condition or check to combine events.

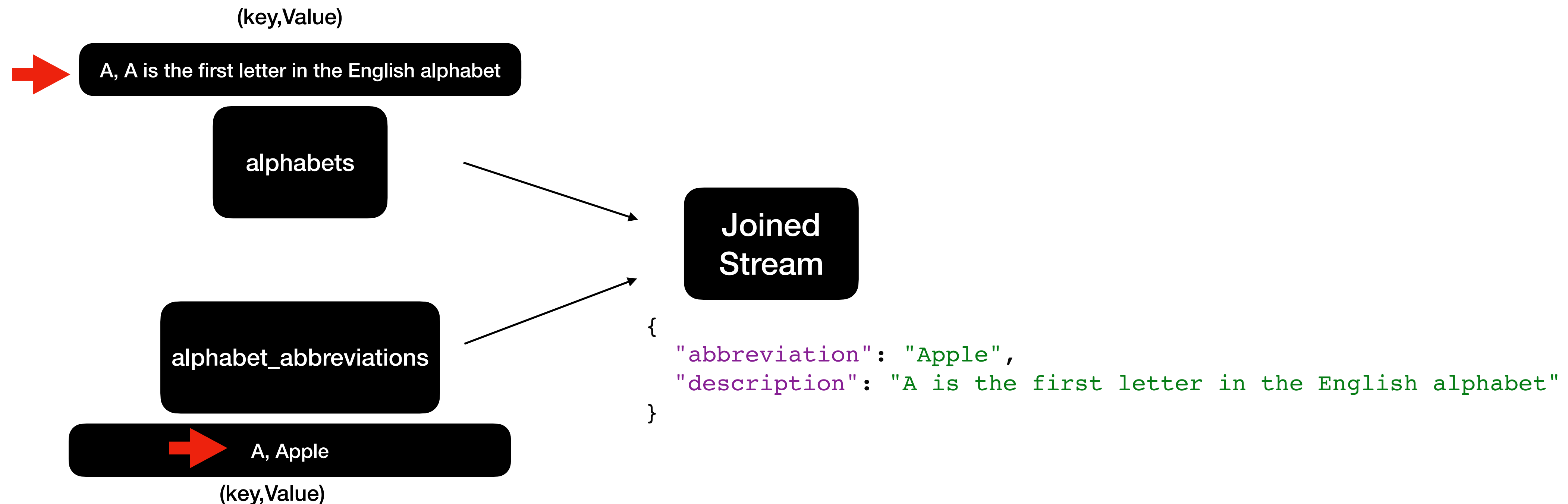
Join operators in Kafka Stream

- join
- leftJoin
- outerJoin

Types of Joins in Kafka Streams

Join Types		Operators
1	KStream-KTable	join,leftJoin
2	KStream-GlobalKTable	join,leftJoin
3	KTable-KTable	join,leftJoin,outerJoin
4	KStream-KStream	join,leftJoin,outerJoin

join



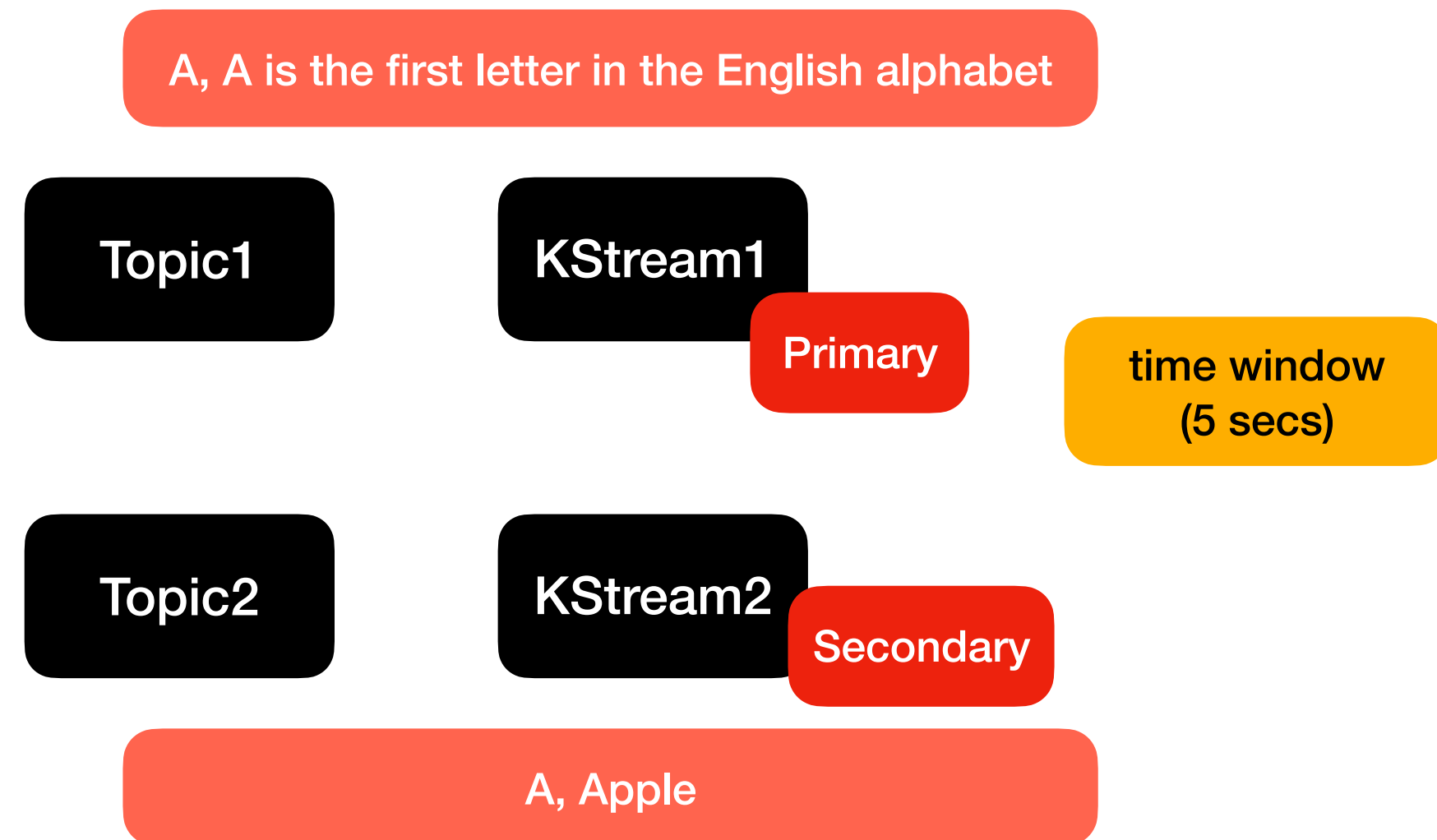
- Joins will get triggered if there is a matching record for the same key.
- To achieve a resulting data model like this , we would need a **ValueJoiner**.
- This is also called **innerJoin**.
- Join won't happen if the records from topics don't share the same key.

Types of Joins in Kafka Streams

- KStream - KTable ✓
- KStream - GlobalKTable ✓
- Ktable - KTable ✓
- KStream - KStream ✓

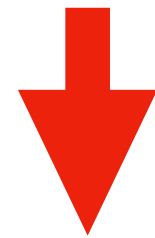
Join KStream - KStream

- The KStream-KStream join is little different compared to the other ones.
- A KStream is an infinite stream which represents a log of everything that happened.



leftJoin

- Join is triggered when a record on the left side of the join is received.



```
var joinedStream = alphabetsAbbreviation.leftJoin  
    (alphabetsStream,  
     valueJoiner,  
     tenSecondWindow  
     , joinedParams  
    );
```

- If there is no matching record on the right side , then the join will be triggered with null value for the right side value.

outerJoin

- Join will be triggered if there is a record on either side of the join.



```
var joinedStream = alphabetsAbbreviation.outerJoin(alphabetsStream,  
                                                    valueJoiner,  
                                                    tenSecondWindow  
                                                    , joinedParams  
                                                    );
```

- When a record is recieved on either side of the join, and If there is no matching record on the other side then the join will populate the null value to the combined result.

Co-Partitioning or Prerequisites in Joins

- The source topics used for Joins should have the same number of partitions.
 - Each partition is assigned to a task in Kafka Streams.
 - This guarantees that the related tasks are together and join will work as expected.
 - We can use `selectKey` or `map` operator to meet these requirements.
- Related records should be keyed on the same key.
 - Same partition strategy should be used.

CoPartitioning

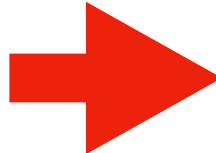
	Join Types	CoPartitioning	Operators
	KStream-KStream	✓	Join,leftJoin,outerJoin
	KStream-KTable	✓	Join,leftJoin
	KTable-KTable	✓	Join,leftJoin,outerJoin
➡	KStream-GlobalKTable	✗	Join,leftJoin

Joins in OrderManagement Service

Current

```
[general-orders-revenue]: store_1234, TotalRevenue[locationId=store_1234, runnuingOrderCount=4, runningRevenue=108.00]
```

New Requirement



```
[general-orders-revenue]: store_1234, TotalRevenueWithAddress[totalRevenue=TotalRevenue[locationId=store_1234, runnuingOrderCount=6, runningRevenue=90.00],  
store=Store[locationId=store_1234, address=Address[addressLine1=1234 Street 1 , addressLine2=, city=City1, state=State1, zip=12345], contactNum=1234567890]]
```

Windowing

- Group events by time windows/buckets that share the same key.
 - Calculating total **Number of orders** placed every **minute/hour/day**.
 - Calculating total **Revenue** generated every **minute/hour/day**.
- In order to group events by time, we need to extract the time from the Kafka record.

Time Concepts

- Event Time
 - Represents the time the record gets produced from the Kafka Producer.
- Ingestion Time
 - Represents the time the record gets appended to the Kafka topic in the log.
- Processing Time
 - This is the time the record gets processed by the consumer.

TimeStamp Extractor in Kafka Streams

- FailOnInvalidTimestamp
 - This implementation extracts the timestamp from the Consumer Record.
 - Throws a StreamsException if the timestamp is invalid.
- LogAndSkipOnInvalidTimestamp
 - Parses the record timestamp, similar to the implementation of FailOnInvalidTimestamp.
 - Simply logs the error incase of an invalid timestamp.
- WallclockTimestampExtractor
 - This ignores the timestamp from the consumer record and just uses the time the record got processed by the streams app.

Window Types

- Tumbling Window
- Hopping Window
- SlidingWindow
- Session Window
- Sliding Join Window
- Sliding Aggregation Window

Tumbling Window

- This type is a fixed sized window.
- Windows never overlap.
- Windows group records by matching keys.
- Time Windows/Buckets are created from the clock of the app's running machine.

➡ `Duration windowSize = Duration.ofSeconds(5);`

➡ `TimeWindows tumblingWindow = TimeWindows.ofSizeWithNoGrace(windowSize);`

`wordsStream`

`.groupByKey()`

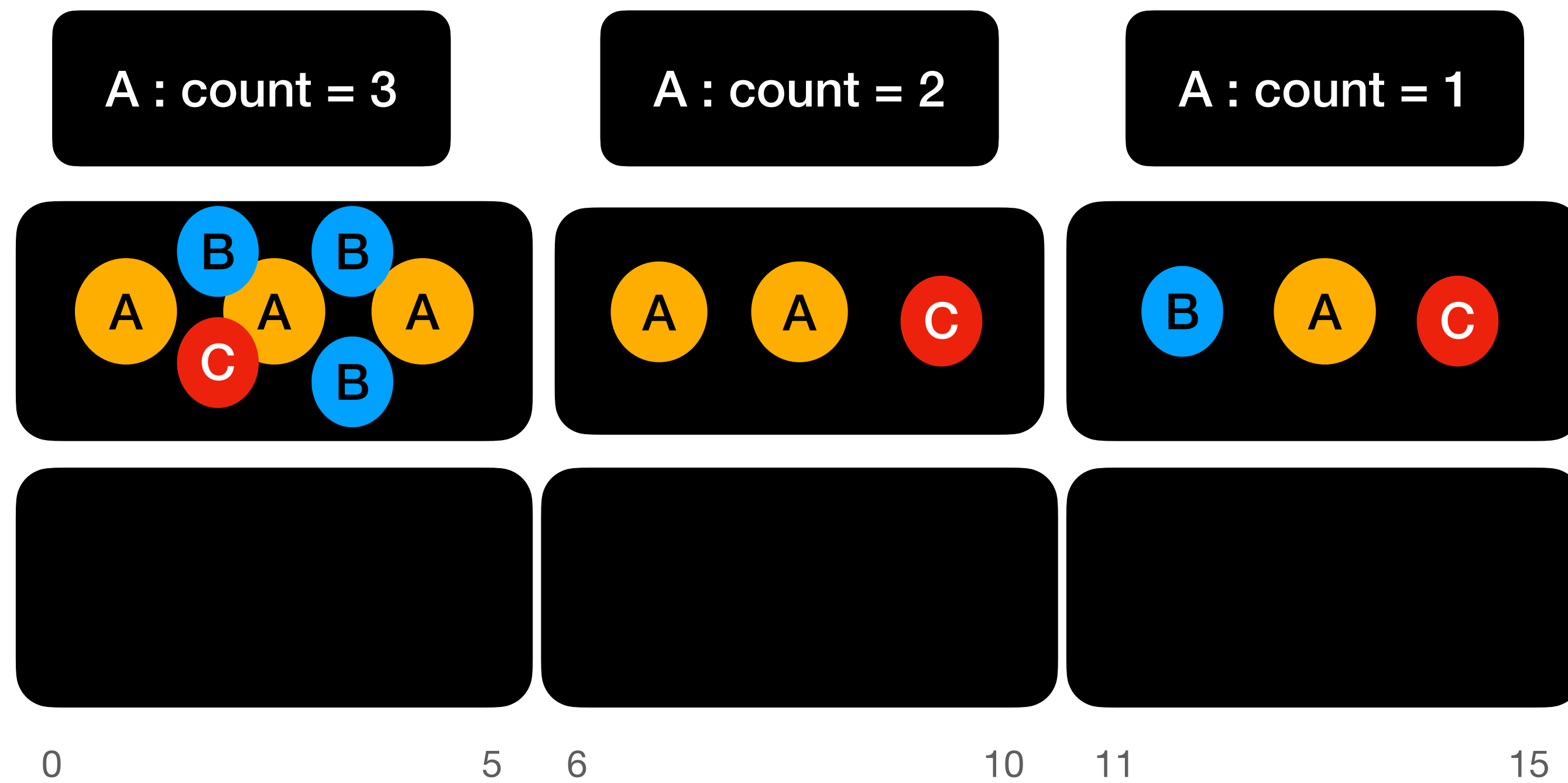
➡ `.windowedBy(tumblingWindow)`

`.count()`

Tumbling Window

```
Duration windowSize = Duration.ofSeconds(5);
```

```
TimeWindows tumblingWindow = TimeWindows.ofSizeWithNoGrace(windowSize);
```



RealTime Examples

- Entertainment:
 - Total number of tickets sold on the opening day.
 - Total revenue made on the opening day.
- We can achieve this by defining a tumbling window for a day.

```
Duration windowSize = Duration.ofDays(1);
```

```
TimeWindows tumblingWindow = TimeWindows.ofSizeWithNoGrace(windowSize);
```

Controlling Emission of Intermediate Results

- By default, aggregated window results are emitted based on the **commit.interval.ms**.
- suppress
 - This operator can be used to **buffer** the records until the time interval for the window is complete.
 - Suppression Config, Buffer Config and BufferFull Config

Suppression Config
(Config 1)

- Suppressed.untilWindowCloses
 - Only emit the results after the defined window is exhausted.
- Suppressed.untilTimeLimit
 - Only emit the results if there are no successive events for the defined time.

Suppression config for Buffer

Buffer Config (Config 2)

- `BufferConfig.maxBytes()`
 - Represents the total number of bytes the buffer can hold.
- `BufferConfig.maxRecords()`
 - Represents the total number of records the buffer can hold.
- `BufferConfig.unbounded()`
 - Represents unbounded memory to hold the records.

BufferFull Strategy (Config 3)

- `shutdownWhenFull`
 - Shutdown the app if the buffer is full
- `emitEarlyWhenFull`
 - Emit the intermediate results if the buffer is full.

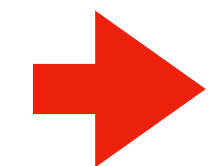
suppression Implementation

```
wordsStream
```

```
    .groupByKey( )
```

```
    .windowedBy( hoppingWindow )
```

```
    .count( )
```



```
    .suppress( Suppressed
```

```
                .untilWindowCloses( Suppressed.BufferConfig.unbounded( ) .shutdownWhenFull( ) )
```

```
    )
```

Hopping Windows

- Hopping windows are fixed time windows that may overlap.
- Time Windows/Buckets are created from the clock of the app's running machine.

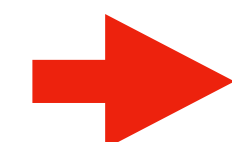
Hopping Windows

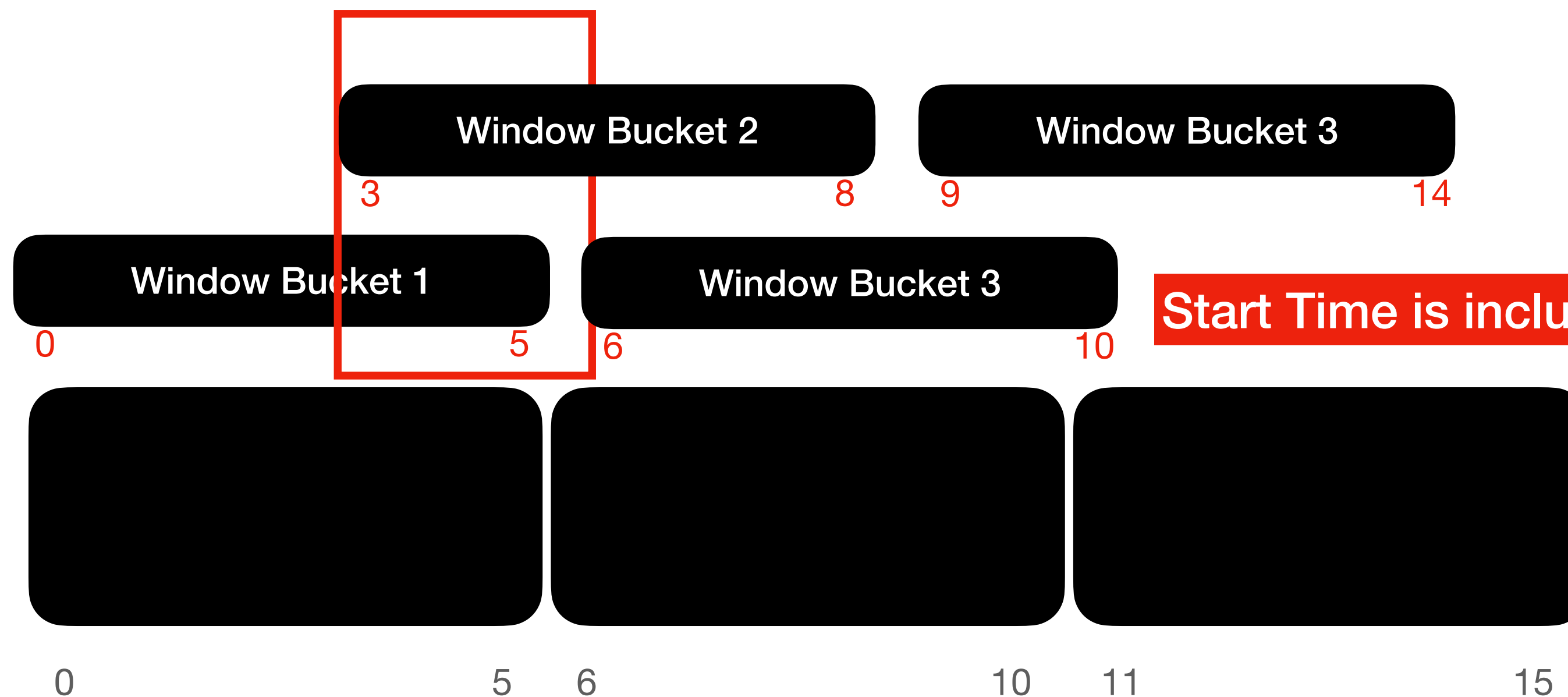
1 `Duration windowSize = Duration.ofSeconds(5);`

2 `Duration advanceBySize = Duration.ofSeconds(3);`

`TimeWindows hoppingWindow = TimeWindows`

`.ofSizeWithNoGrace(windowSize)`

 `.advanceBy(advanceBySize);`



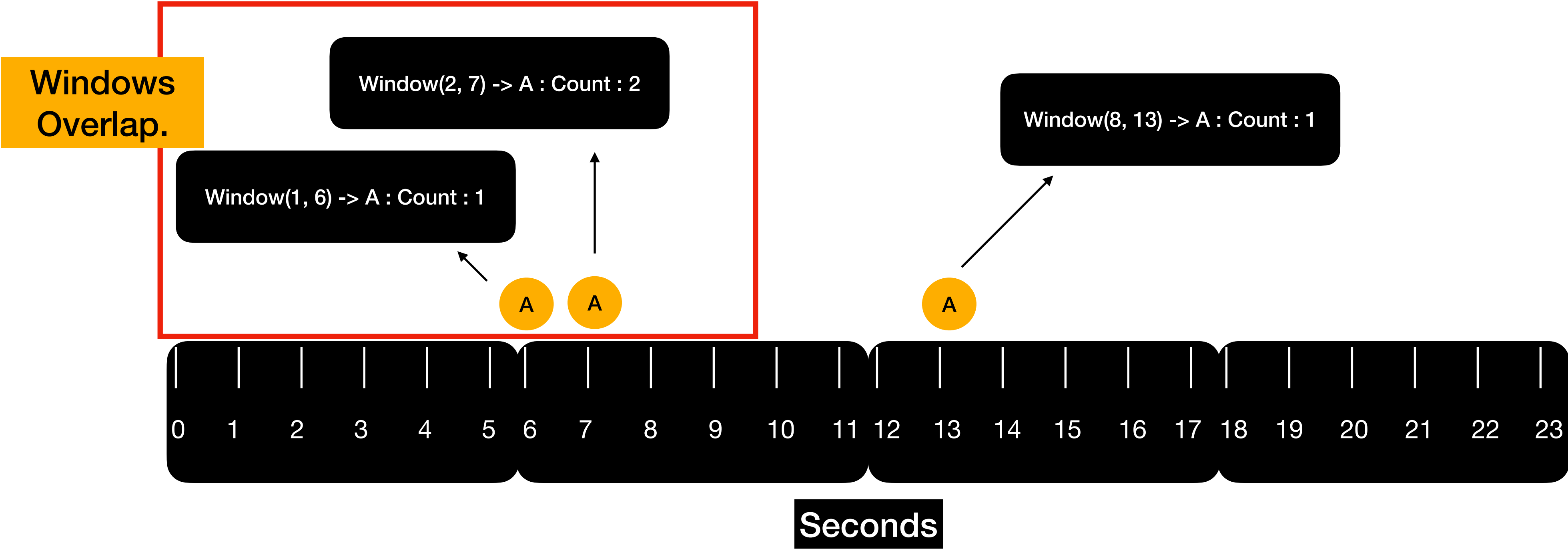
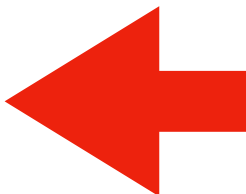
Start Time is inclusive and the end time is exclusive.

Sliding Windows

- This is a fixed time window, but the windows created are not defined by the machines running clock.
- It's the timestamp in the event that creates the window.
- Windows that are created overlap with one another.

Sliding Windows

```
SlidingWindows slidingWindow = SlidingWindows
    .ofTimeDifferenceWithNoGrace(Duration.ofSeconds(5));
```



Start and End times are inclusive.

New Requirements

- Aggregate the number of orders by grouping them in time windows.
- Aggregate the revenue of the orders by grouping them in time windows.
- Group the aggregations by **15 seconds**.
 - What type of Windowing function can we use for this one ?
 - Tumbling Window

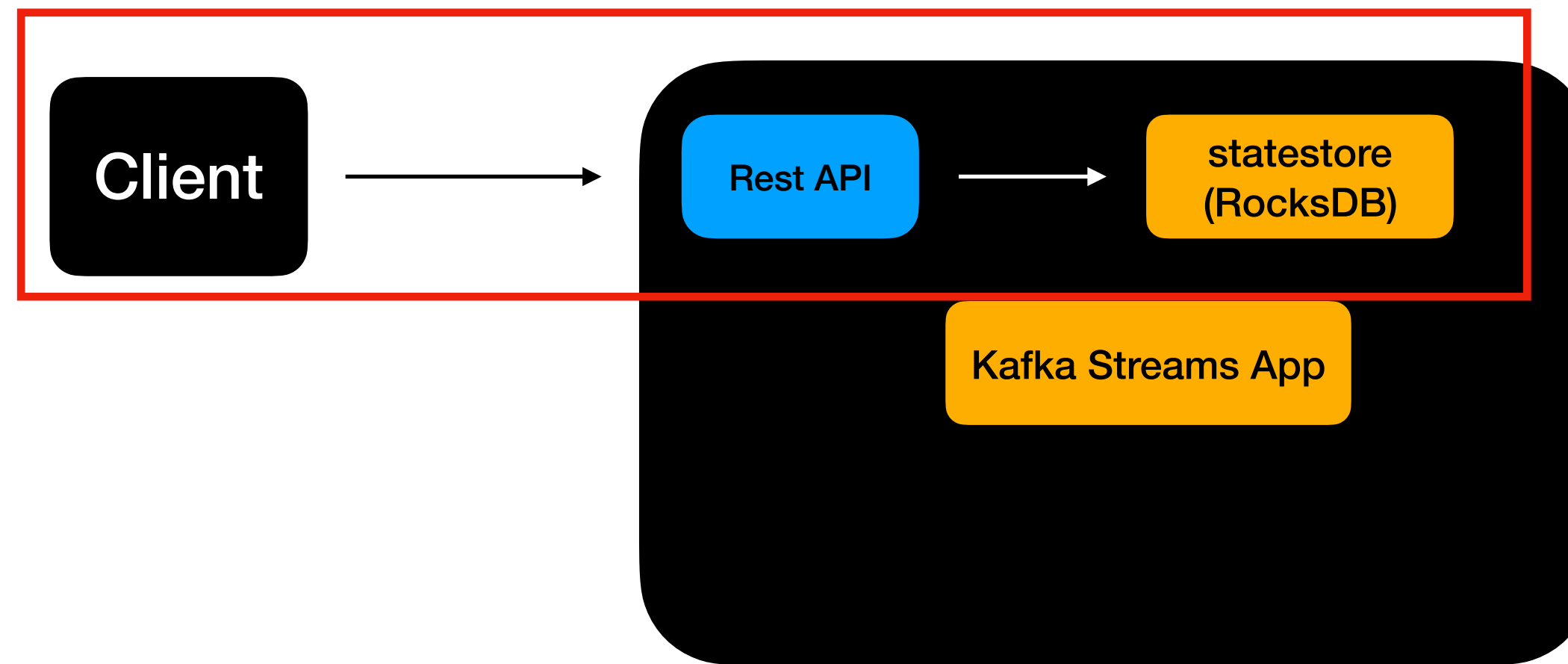
Order Message

```
{
  "orderId": 12345,
  "locationId": "store_1234",
  "finalAmount": 27.00,
  "orderType": "GENERAL",
  "orderLineItems": [
    {
      "item": "Bananas",
      "count": 2,
      "amount": 2.00
    },
    {
      "item": "Iphone Charger",
      "count": 1,
      "amount": 25.00
    }
  ],
  "orderedDateTime": "2022-12-05T08:55:27"
}
```

Kafka Streams Application using SpringBoot

- Spring framework is one the popular JVM frameworks.
- Build Kafka Streams application using SpringBoot.
 - Spring Framework is going to manage lifecycle of the Kafka Streams app.
- JSON Serialization/Deserialization.
 - SpringBoot has in-built generic Serdes to handle this functionality.
- Error Handling using SpringBoot in a Kafka Streams application.
- Query Materialized views and expose the data in a RestFul API.
- Unit and Integration tests for the Spring Kafka Streams using JUnit5.

Querying Materialized Views



OrdersManagement KafkaStreams App using SpringBoot

Testing Kafka Streams

- Automated testing is one of the key requirement for quality software development.
- So far, we have been manually testing.
 - Time consuming and error prone.

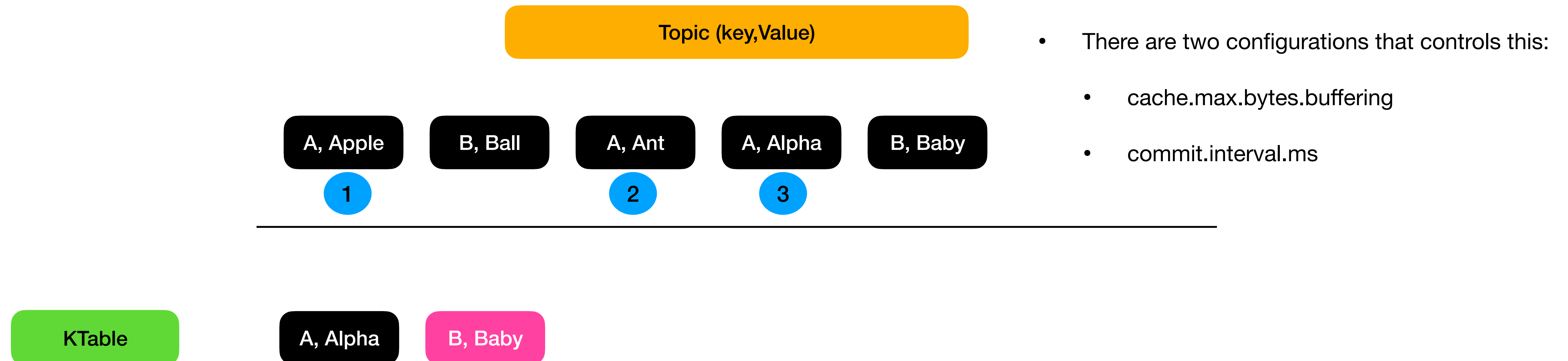
Testing Kafka Streams

- Business Logic resides in the Topology.
- JUnit Test-cases are going to test the Topology.
- Kafka Streams has a test library named **kafka-streams-test-utils**.
- Kafka Streams app does two things:
 - Writes the enriched data into an output topic.
 - Writes the enriched or aggregated data into a state store.

Limitations of TopologyTestDriver

- It cannot not simulate the caching behavior that comes with KTable and StateStore.

```
var wordsTable = streamsBuilder
    .table("words", Consumed.with(Serdes.String(), Serdes.String())
        , Materialized.as("words-store")
    );
```

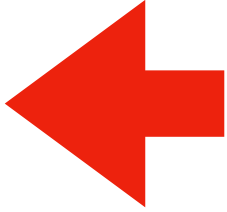


- Kafka Cluster to simulate this behavior.

Integration Tests in Spring Kafka Streams

- Integration test is a type of test in which the app interacts with the external source or system.
- An Integration test in our Kafka Streams app is to interact with the real Kafka environment.
 - This will simulate the behavior of the app that's running in a production environment.

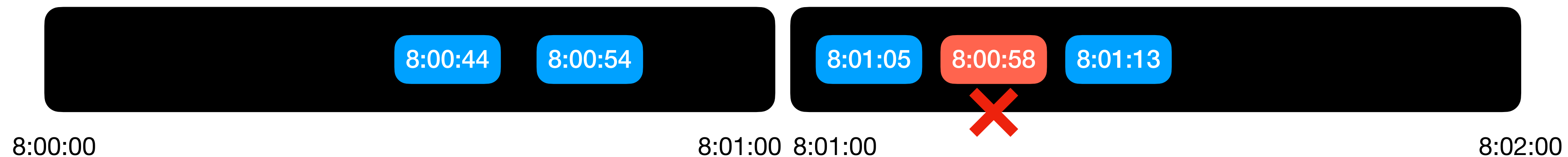
Options for Integration Tests

- EmbeddedKafka 
- TestContainers

Grace Period

- Data delays are a common concern in streaming applications.
 - Events could be delayed because the producing app has some issue or the Kafka broker is down.
- What is a data delay in the first place ?

Data Delay Example



Grace Period to the Rescue

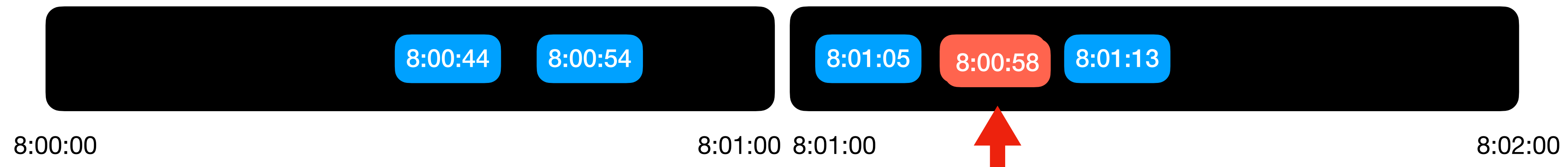
Grace Period Example

```
Duration windowSize = Duration.ofSeconds(60);
```

➔

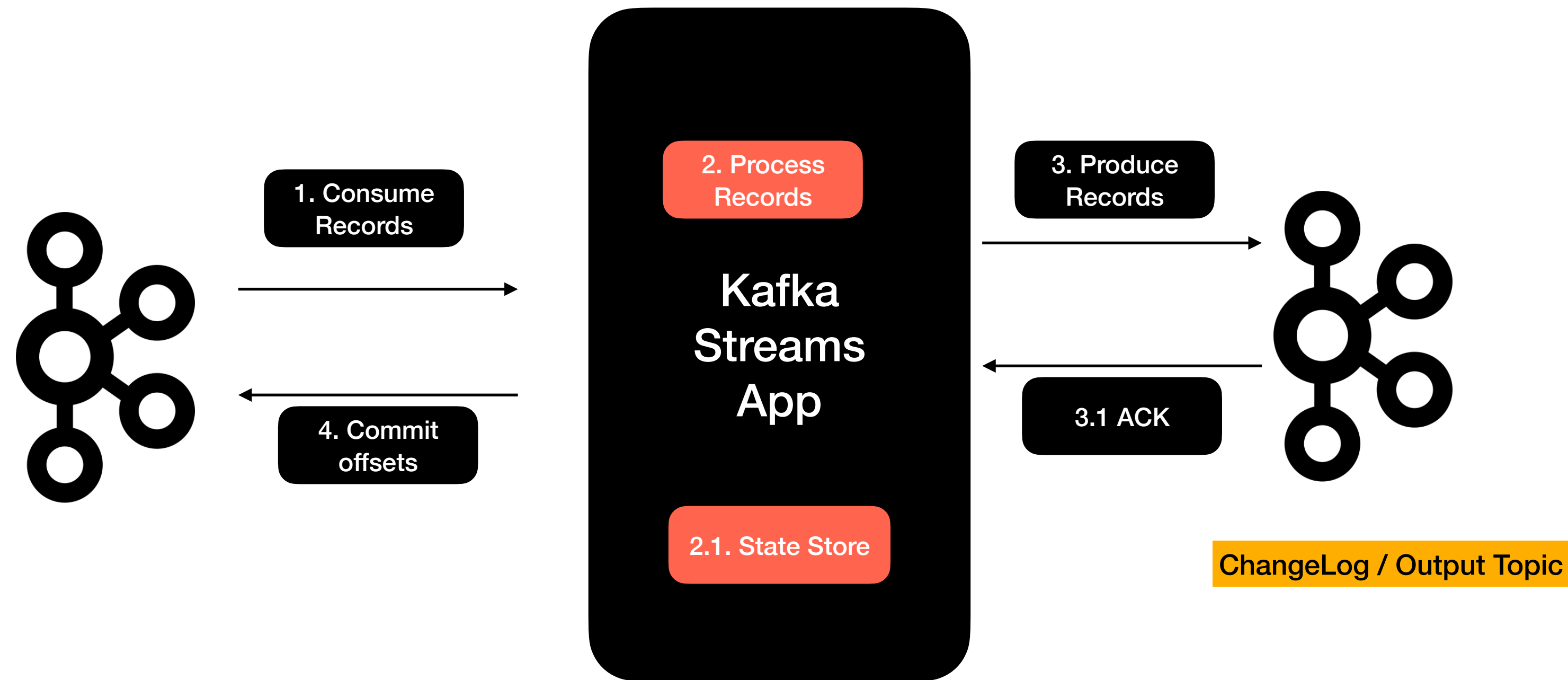
```
Duration graceWindowSize = Duration.ofSeconds(15);
```

```
TimeWindows hoppingWindow = TimeWindows.ofSizeAndGrace(windowSize, graceWindowSize);
```

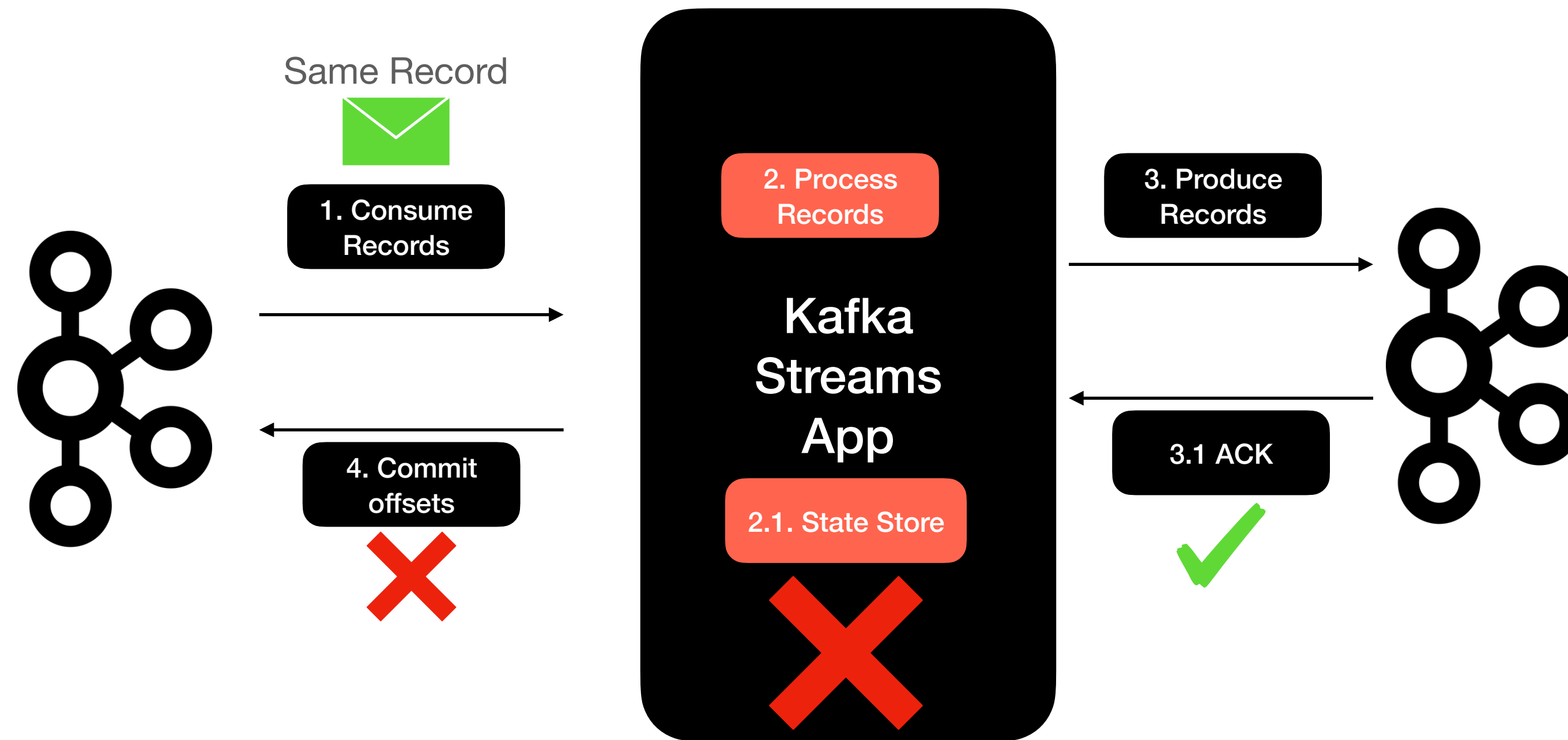


Exactly Once Processing

A Typical Kafka Streams App

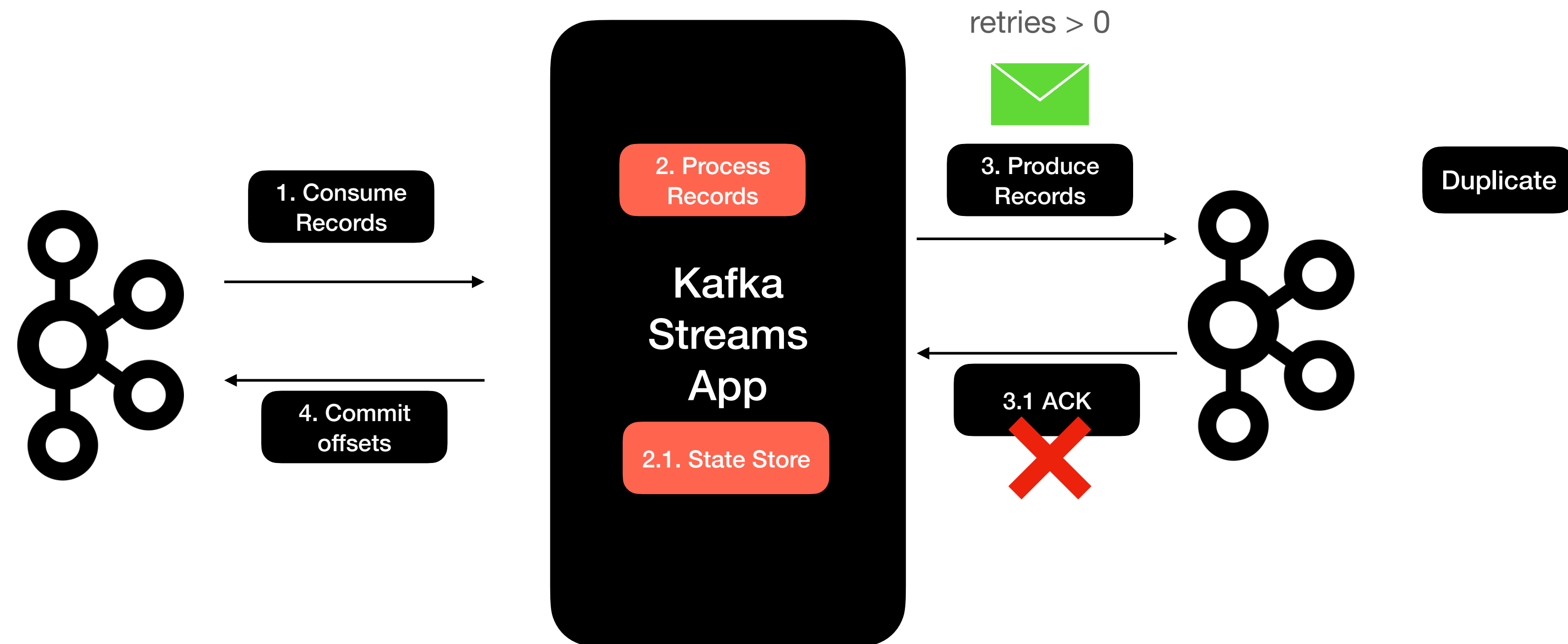


Error 1 : Application Crash After Step 3



Reprocess the same record due to application crash and results in Duplicate writes.

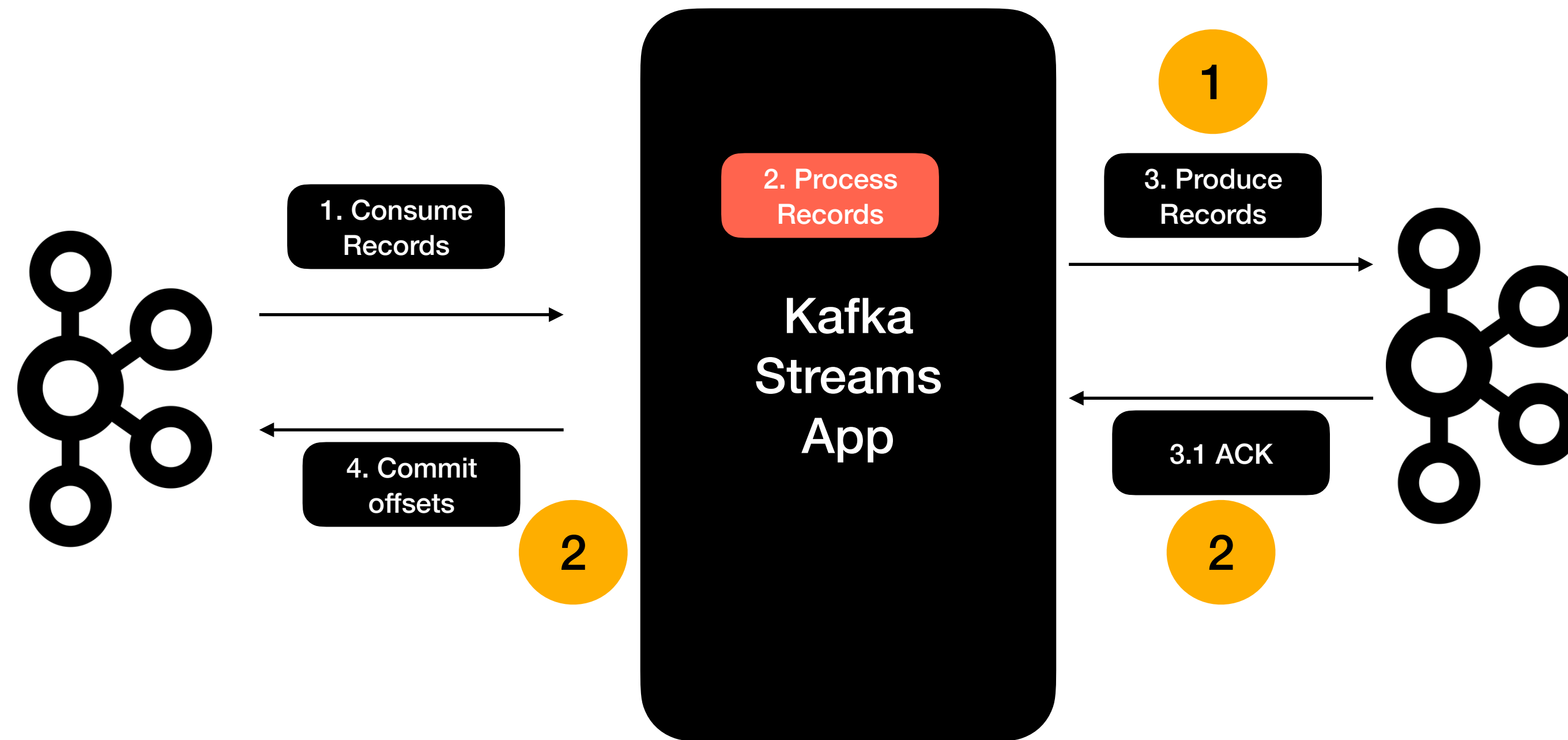
Error 2 : Duplicate Writes due to Network Partition



Are there any issues with Duplicates?

- It depends on the use case.
- **Duplicates not Allowed:**
 - Finance related transactions
 - Revenue Calculation
- **Duplicated Allowed:**
 - Likes on a post
 - No of viewers on a live stream

How to guarantee Exactly Once Processing ?



- 1. Idempotent Producer
 - The retries should not result in duplicates
- 2. Transactions (atomicity)
 - State Update to Internal Topics (changelog)
 - Producer the result to output topics
 - Committing offsets

Exactly Once Processing in Kafka Streams

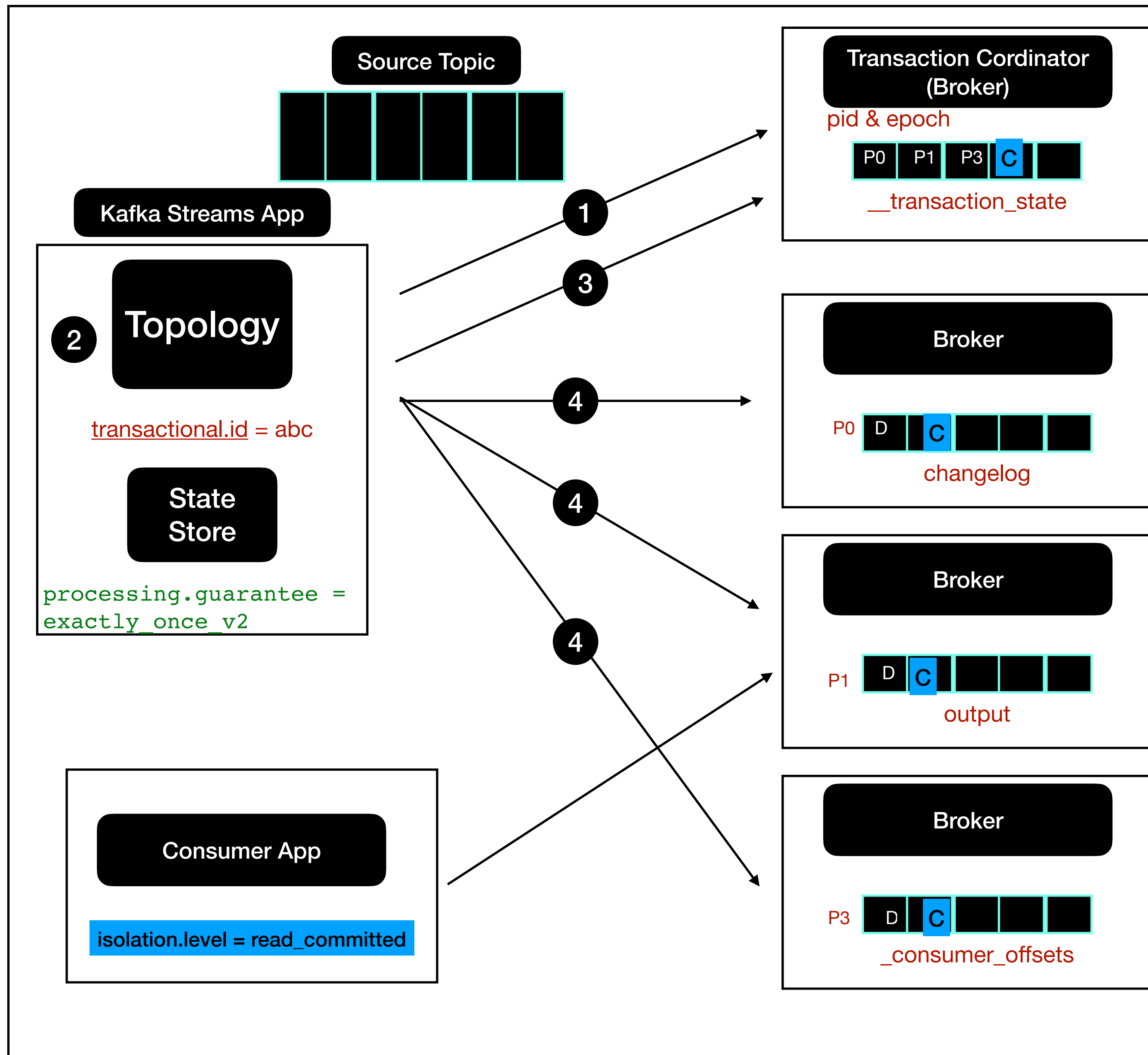
- Idempotent Producer 1
 - The retries should not result in duplicates
- Transactions 2
 - State Update to Internal Topics (changelog)
 - Producer the result to output topics
 - Committing offsets

Implement Exactly Once Processing in Kafka Streams

```
processing.guarantee = exactly_once_v2
```

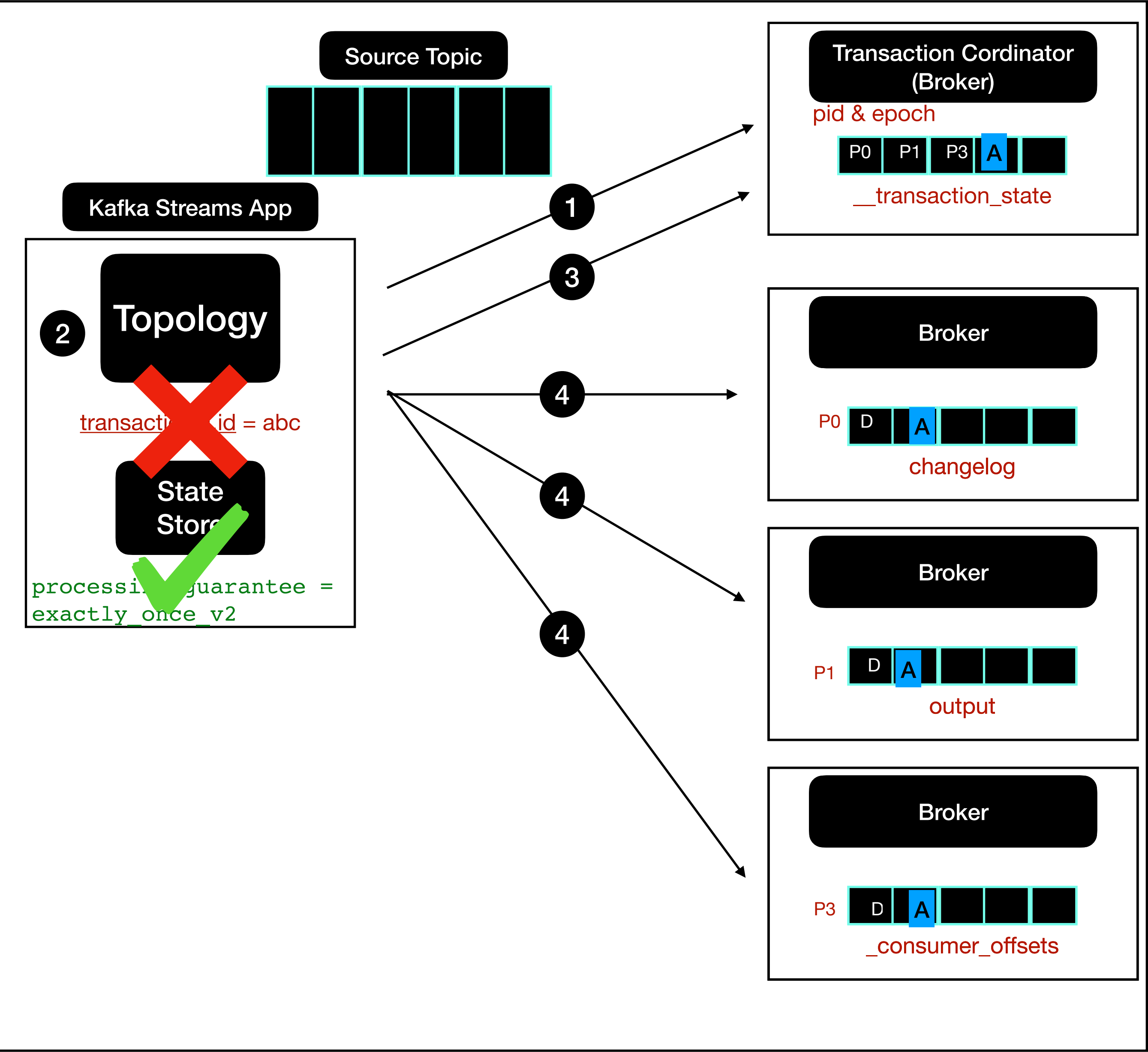
- Pass this config to the Kafka Streams application.
- The idea is to keep it simple for the developers.

Transactions Under the hood



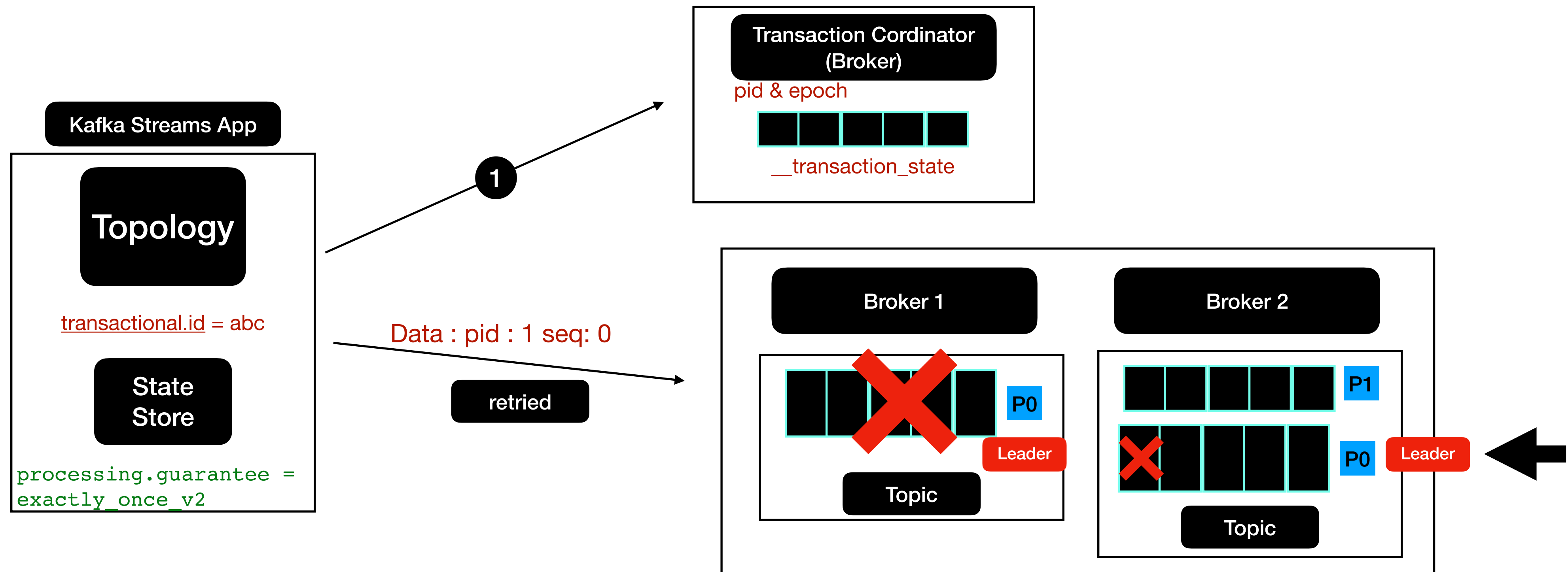
```
try {  
    producer.beginTransaction();  
  
    records = consumer.poll();  
  
    for(record : records ){  
  
        producer.send("changelog")  
        producer.send("output")  
        producer.sendConsumerOffsets()  
  
        //commit the txn  
        producer.commitTxn()  
    } catch (KafkaException e){  
    }  
}
```

Transactions Error Scenario



```
try {  
    producer.beginTransaction();  
  
    records = consumer.poll();  
  
    for(record : records ){  
  
        producer.send("changelog")  
        producer.send("output")  
        producer.sendConsumerOffsets()  
  
        //commit the txn  
        producer.commitTxn()  
    } catch (KafkaException e){  
  
        producer.abortTxn()   
    }  
}
```


Transactions : Idempotent Producer



Producer Retries are not going to cause message duplicates.

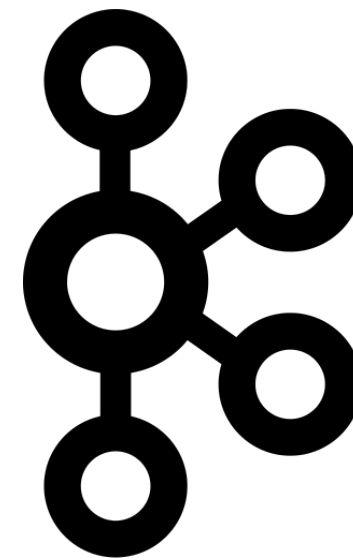
Limitations of Exactly Once Processing

- Deduplication is not a transaction feature.
 - If a duplicate record is published into source topic then the Kafka Streams app has no clue that the record was not processed already
 - For example : In the streams processing logic , if we are sending a text/email for each record then an email/text will be sent for the duplicate record too.
- Kafka Transactions are not applicable for Consumers that read from a Kafka Topic and write to a DB.
 - Reason being there is no producer involved in this whole flow.

Performance of Kafka Transactions

- Enabling Transactions adds a performance overhead to the Kafka Producer.
 - Additional call to the broker to register the producer id.
 - Additional call to add the register the partitions to the transaction coordinator.
 - Transaction initialization and Commit operations are synchronous, the app needs to wait and blocked until these operations complete.
- On the Consumer end, the consumer needs to wait for records until the transaction is complete because of the `read_committed` configuration.
 - If the transaction takes a longer time then the wait period for consumers goes up.

Running Kafka Streams as Multiple Instances



Orders



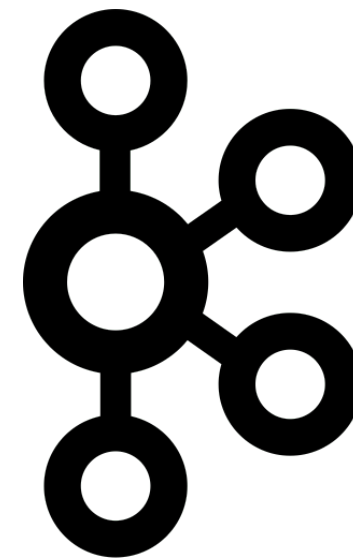
`application.id = orders-stream-app`

Kafka Streams App

Stream Thread

Running Kafka Streams as Multiple Instances

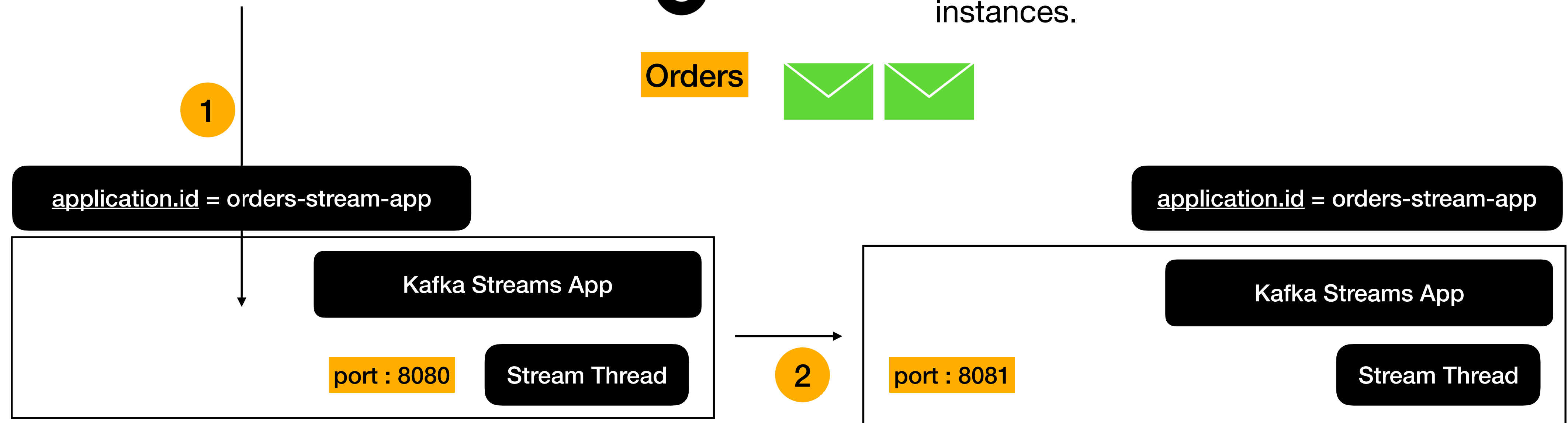
```
curl -i http://localhost:8080/v1/orders/count/general_orders
```



Orders



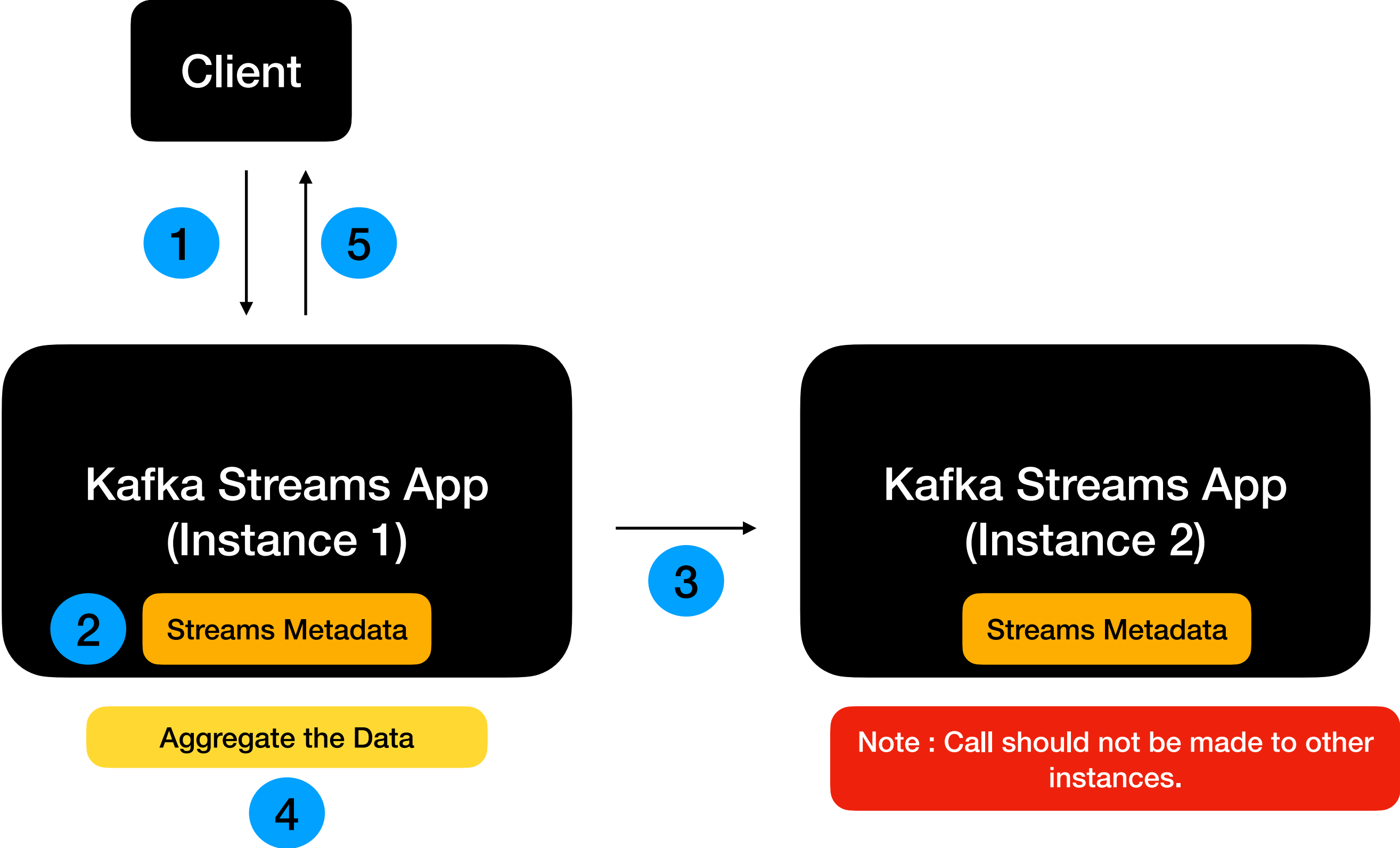
- Each instance needs to know the address of the other instance.
- Aggregate Data from multiple instance.
- Build REST Clients to interact with other instances.



Running Kafka Streams as Multiple Instances

1. Each instance needs to know the address of the other instance.
2. Build REST Clients to interact with other instances.
 1. Get the Data from other instances
 2. Aggregate the Data and respond to client

Aggregate Data from Multiple Instances



What's needed to aggregate the data ?

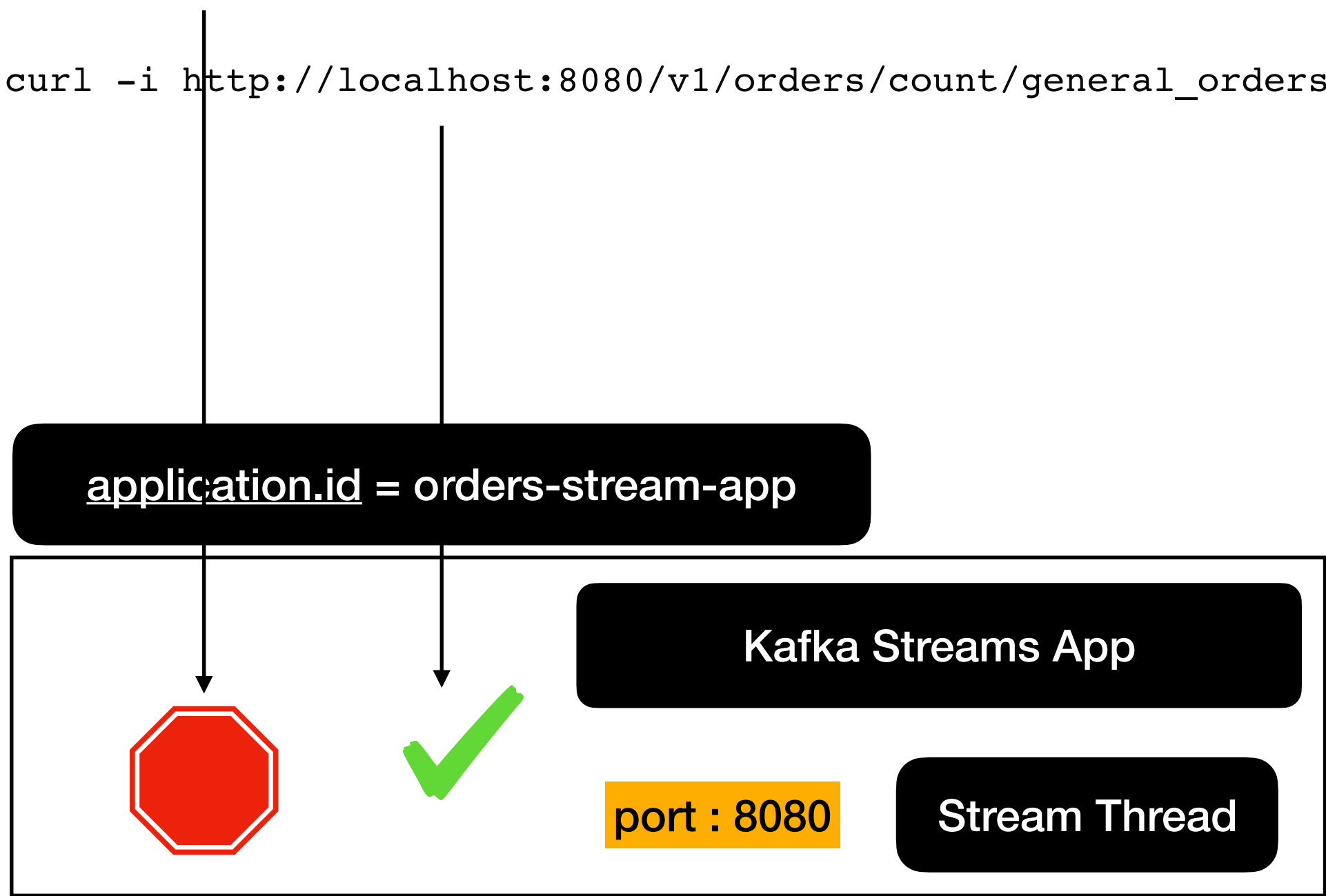
1. Use the `KafkaStreams` instance from `StreamsBuilderFactoryBean` to find out the metadata information about other Kafka Streams Instances.
2. Build a rest client to interact with other Kafka Streams Instances.
 - A. Spring WebClient
3. Build the logic to aggregate the data.

KeyBased Queries with Multiple Instances - Overview

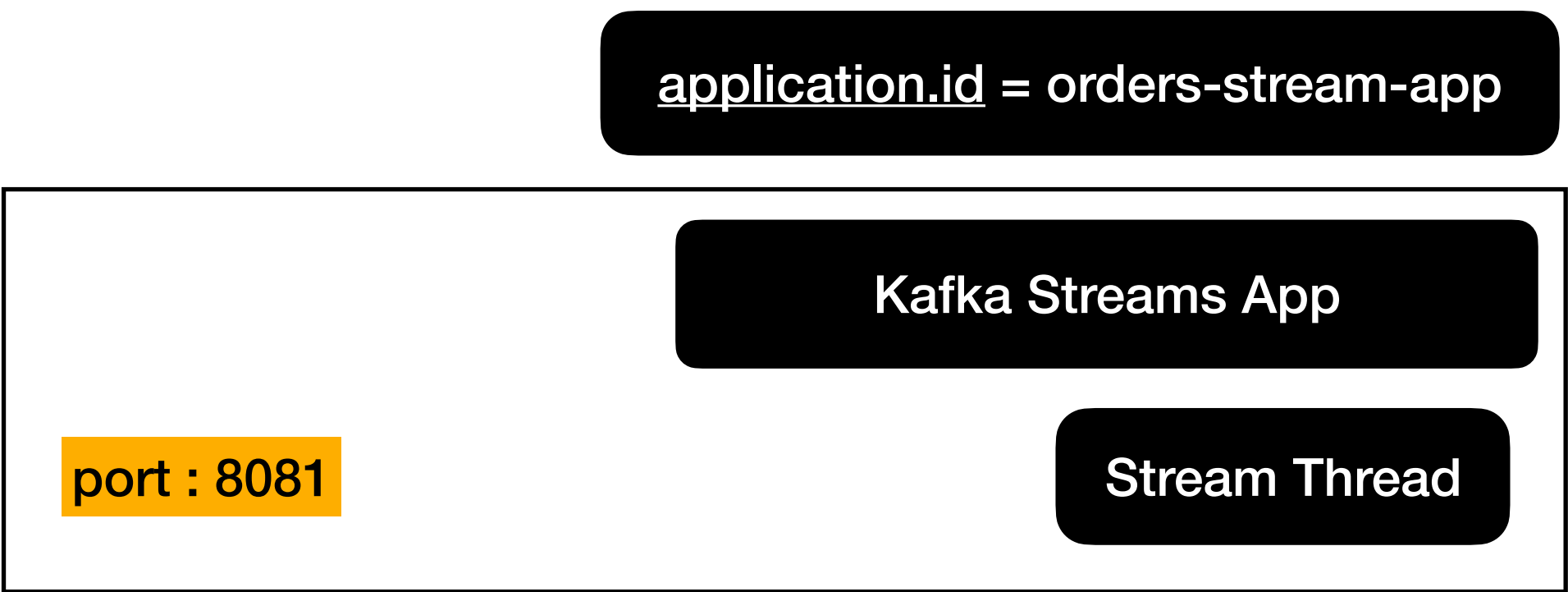
- Each instance needs to know the instance that holds the data for a given key.

```
curl -i http://localhost:8080/v1/orders/count/general_orders?location_id=store_4567
```

```
curl -i http://localhost:8080/v1/orders/count/general_orders?location_id=store_1234
```



`{"locationId":"store_1234","orderCount":1}`



`{"locationId":"store_4567","orderCount":1}`