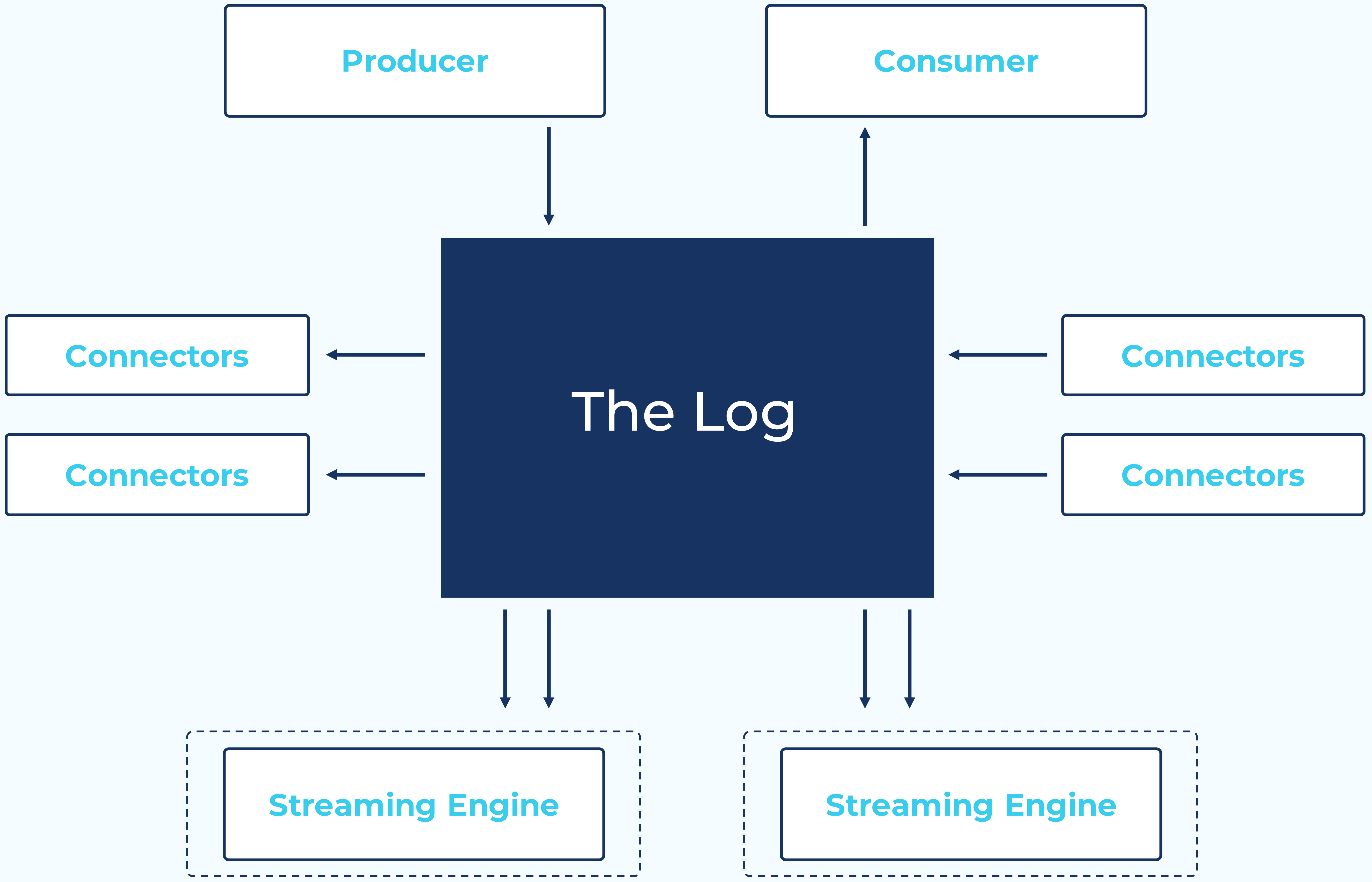
Kafka Streams and KTables

What's Covered

Getting Started with Kafka Streams

Apache Kafka®



How to Process Events in Kafka


```
{
 “reading_ts”: “2020-02-14T12:19:27Z”,
 “sensor_id”: “aa-101”,
 “production_line”: “w01”,
 “widget_type”: “acme94”,
 “temp_celcius”: 23,
 “widget_type”: 100
}
```


Processing Events with Producer and Consumer Clients

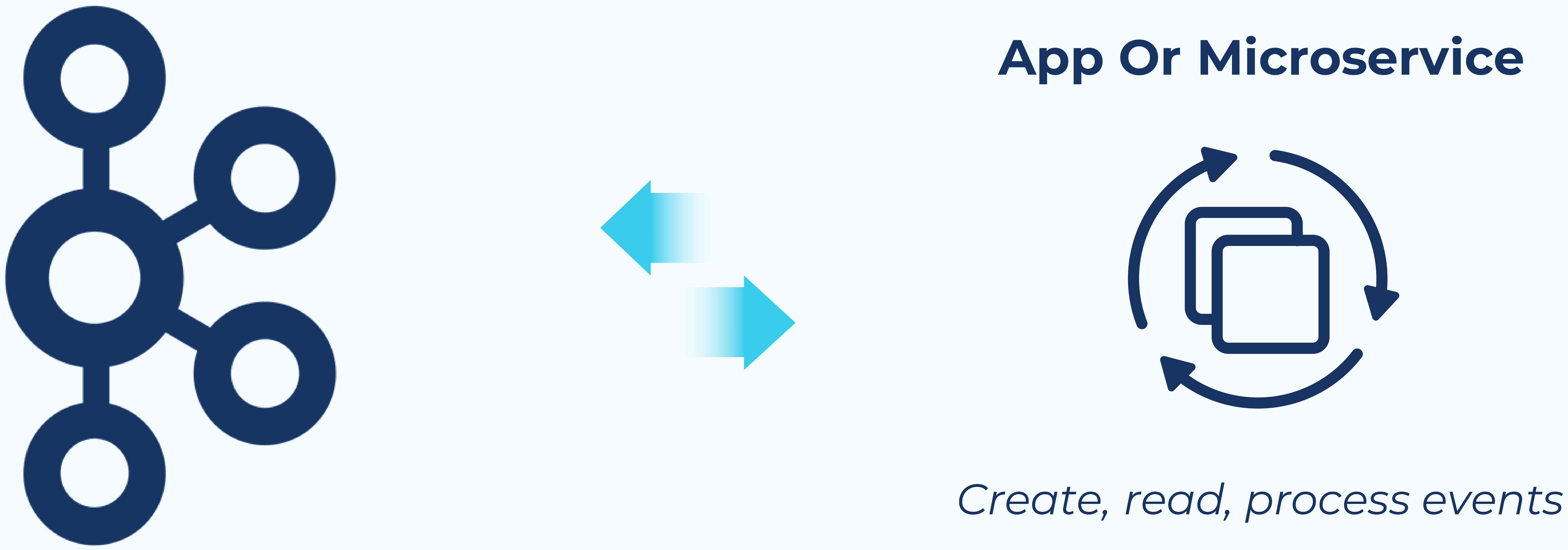
```
public static void main(String[] args) {
  try(Consumer<String, Widget> consumer = new KafkaConsumer<>(consumerProperties());
     Producer<String, Widget> producer = new KafkaProducer<>(producerProperties())) {
         consumer.subscribe(Collections.singletonList("widgets"));
         while (true) {
             ConsumerRecords<String, Widget> records =    consumer.poll(Duration.ofSeconds(5));
                 for (ConsumerRecord<String, Widget> record : records) {
                     Widget widget = record.value();
                     if (widget.getColour().equals("red") {
                         ProducerRecord<String, Widget> producerRecord = new ProducerRecord<>(
                             "widgets-red", record.key(), widget);
                         producerRecord.send(producerRecord, (metadata, exception)-> {…….} );
                 ...
```

Processing Events with Kafka Streams

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

builder.stream(“widgets”, Consumed.with(stringSerde, widgetsSerde))
        .filter( (key, widget) -> widget.getColour.equals("red"))
        .to("widgets-red", Produced.with(stringSerde, widgetsSerde));
```

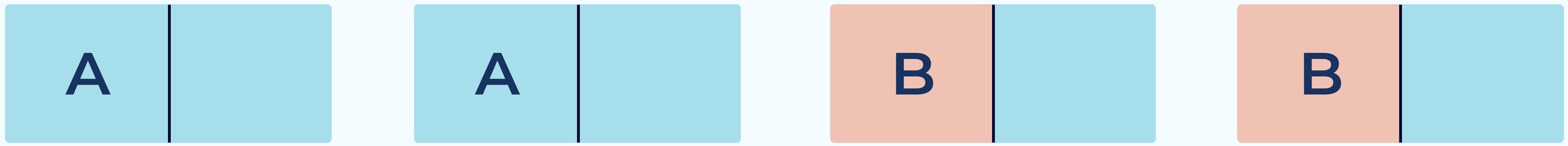
Kafka Streams is a Java Library



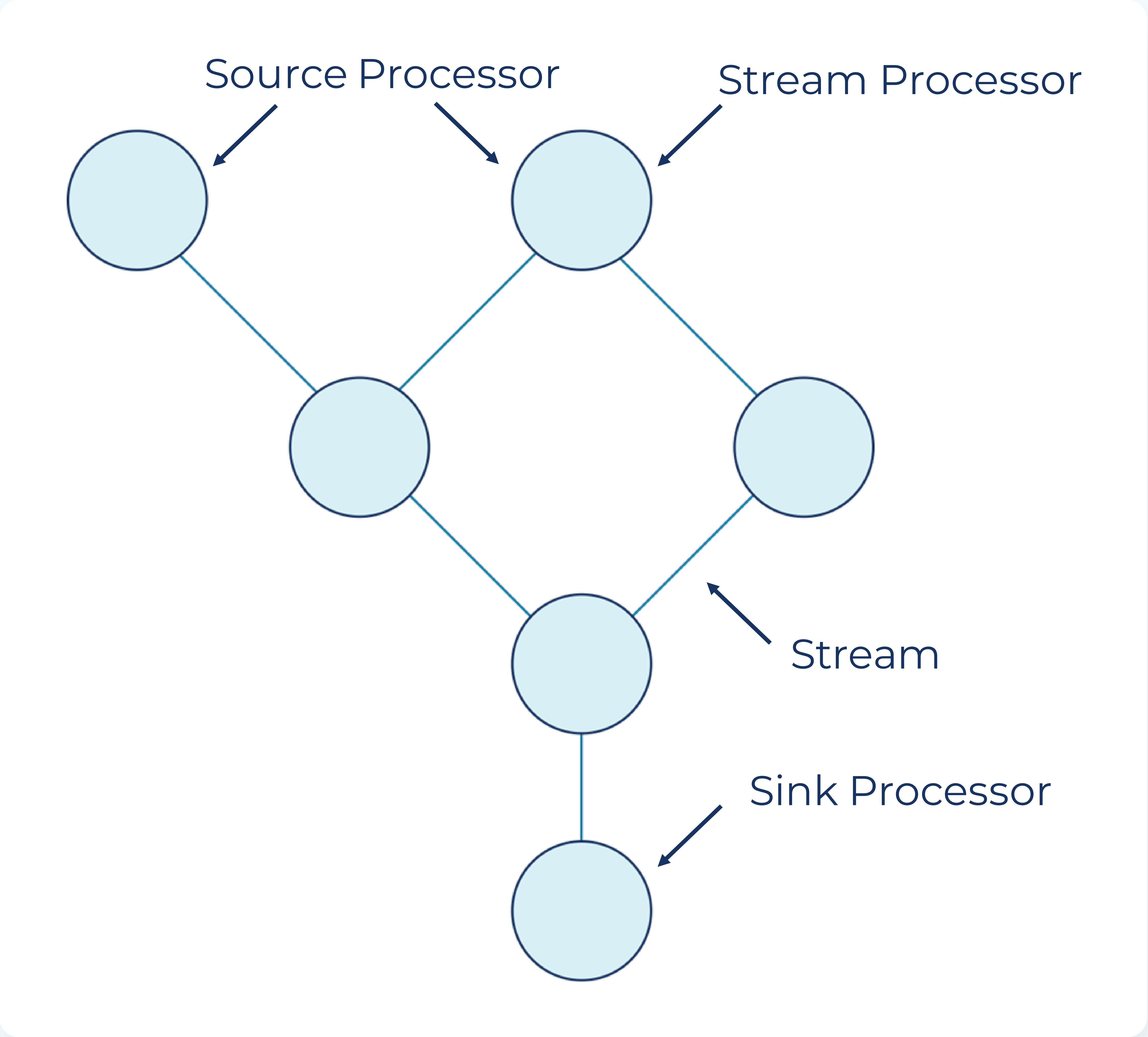
Use the library to write standard Java/JVM applications that process data in Kafka. Kafka Streams makes your applications elastic, distributed, scalable, and fault tolerant.

Basic Operations

Event Streams



Processor Topology



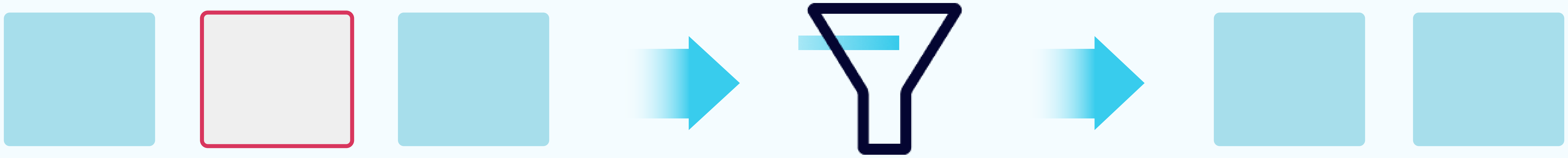
Defining a Stream

```
StreamBuilder builder = new StreamBuilder();
KStream<String, String> firstStream =
builder.stream(inputTopic, Consumed.with(Serdes.String(), Serdes.String()));
```

Mapping


```
mapValues(value -> value.substring(5))
map((key, value) -> ..)
```

Filtering



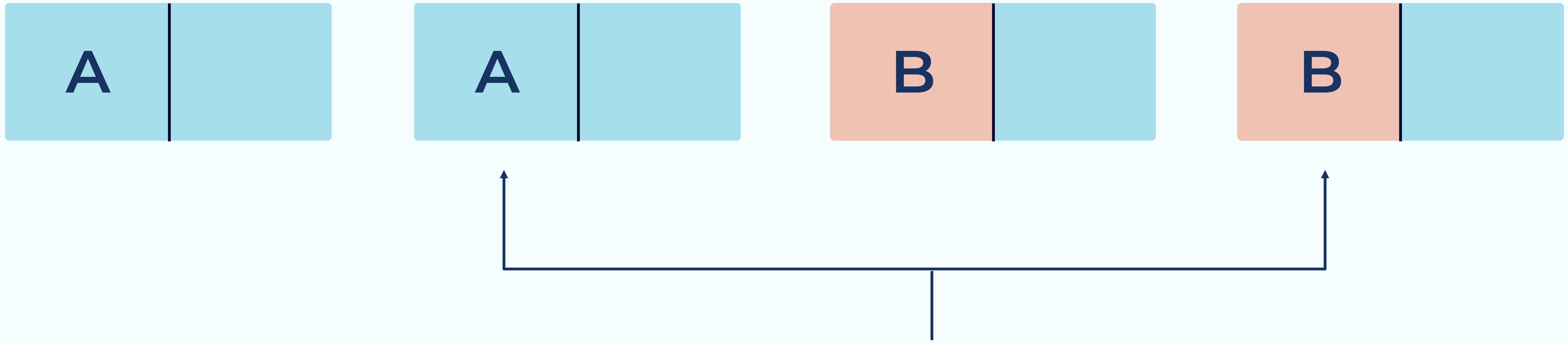
```
filter((key, value) -> Long.parseLong(value) > 1000)
```

Hands On: Basic Operations

Hands On: Basic Operations

KTable

Update Streams



Events with the same key are considered updates to previous records with the same key.

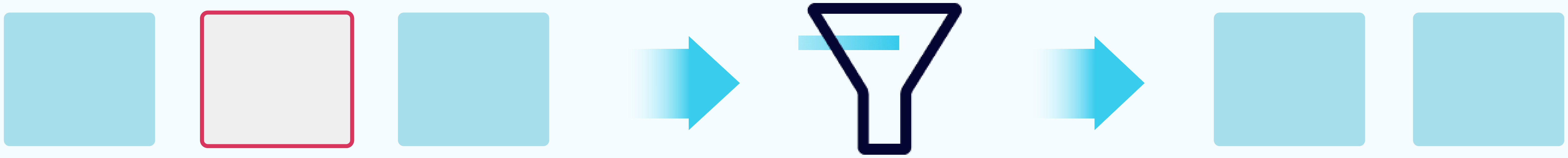
Defining a KTable

```
StreamBuilder builder = new StreamBuilder();
KTable<String, String> firstKTable =
builder.table(inputTopic, Materialized.with(Serdes.String(), Serdes.String()));
```

Mapping


```
firstKTable.mapValues(value -> ..)
firstKTable.map((key,value) -> ..)
```

Filtering



```
firstKTable.filter((key, value) -> ..)
```

Global KTable

```
StreamBuilder builder = new StreamBuilder();
GlobalKTable<String, String> globalKTable =
Builder.globalTable(inputTopic, Materialized.with(Serdes.String(), Serdes.String()));
```

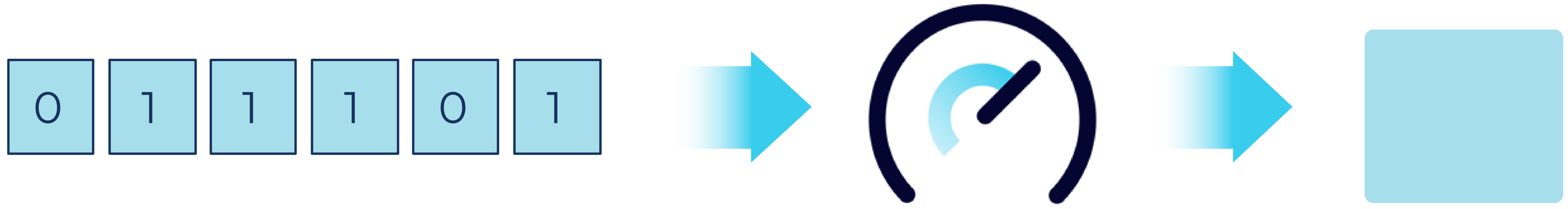
Hands On: KTable

Hands On: KTable

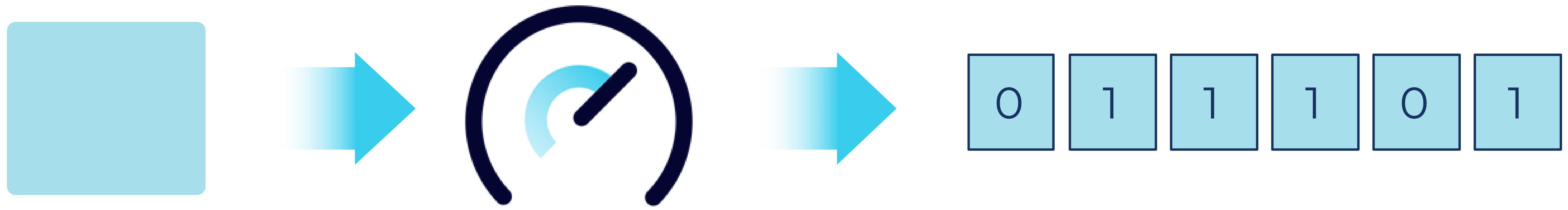
Data Serialization

Data Serialization

Bytes to object -
deserialize



Object to bytes -
serialize



Data Serialization - Serdes (Serializer/Deserializer)

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, MyObject> stream =
builder.stream("topic", Consumed.with(Serdes.String(), customObjectSerde)
```

Serdes used by Kafka Streams for reading input bytes into expected object types

Data Serialization - Serdes (Serializer/Deserializer)

```
KStream<String, CustomObject> modifiedStream =
    stream.filter( (key, value) -> value.startsWith(“ID5”))
    .mapValues( value -> new CustomObject(value));

modifiedStream.to(“output-topic”, Produced.with(Serdes.String(), customObjectSerde);
```

Serdes used by Kafka Streams
for serializing objects into bytes

Data Serialization - Serdes (Serializer/Deserializer)

```
Serde<T> serde = Serdes.serdeFrom( new CustomSerializer<T>,
                                           new
                                   CustomDeserializer<T>);
```

Data Serialization - Serdes (Serializer/Deserializer)

- Pre-existing serdes: String, Integer, Double, Long, Float, Bytes, ByteArray, ByteBuffer, UUID, and Void
- Additional Serdes available for working with Avro, Protobuf, and JSONSchema
    - Avro
        - SpecificAvroSerde
        - GenericAvroSerde
    - Protobuf
        - KafkaProtobufSerde
    - JSONSchema
        - KafkaJsonSchemaSerde

Joins

Joins

- Kafka Streams offers join operations
- Stream-Stream joins
  - Combine two event streams into a new event stream
  - Join of events based on a common key
  - Records arrive within a defined window of time
  - Possible to compute a new value type
  - Keys are available in read-only mode can be used in computing the new value
- Stream-Table joins
  - KStream-KTable
  - KStream-GlobalKTable
- Table-Table joins

Joins - Types Available

- Stream-Stream
  - Inner - Only if both sides are available within the defined window is a joined result emitted
  - Outer - Both sides always produce an output record
    - Left-value + Right-value
    - Left-value + Null
    - Null + Right-value
  - Left-Outer - The left side always produces an output record
    - Left-value + Right-value
    - Left-value + Null

Joins - Types Available

- Stream-Table
  - Non-windowed joins
  - Inner - Only if both sides are available is a record emitted
  - Left-Outer - The left side (KStream) always produces an output record
    - Left-value + Right-value
    - Left-value + Null
  - Only the Stream side drives the join - new records arriving to the table (right-side) don't result in outputting a join result
  - Applies to both KTable and GlobalKTable joins
  - GlobalKTable joins provide mechanism for determining the join-key from the Stream side key and/or value
  - KTables are timestamp driven but GlobalKTables are bootstrapped - results in different join semantics

Joins - Example

```
KStream<String, String> leftStream = builder.stream("topic-A");
KStream<String, String> rightStream = builder.stream("topic-B");

 ValueJoiner<String, String, String> valueJoiner = (leftValue, rightValue) -> {
    return leftValue + rightValue;
};
  leftStream.join(rightStream,
                  valueJoiner,
                  JoinWindows.of(Duration.ofSeconds(10)));
```

Hands On: Joins

Hands On: Joins

Stateful Operations

Stateful Operations

- Stateless operations are great for operations where you don't need to remember
  - Filter - drop records that don't match a condition
- Other times you need to remember previous results
  - How many times has a particular customer logged in?
  - What's the total sum of tickets sold?
- For those situations where the previous state of an event is important, Kafka Streams offers stateful operations

Stateful Operations - Reduce

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Long> myStream = builder.stream("topic-A");

Reducer<Long> reducer = (longValueOne, longValueTwo) -> longValueOne + longValueTwo;
myStream.groupByKey().reduce(reducer,
                         Materialized.with(Serdes.String(), Serdes.Long()))
                        .toStream().to("output-topic");
```

Stateful Operations - Aggregation

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> myStream = builder.stream("topic-A");

Aggregator<String, String, Long> characterCountAgg =
                    (key, value, charCount) -> value.length() + charCount;
myStream.groupByKey().aggregate(() -> 0L,
                                      characterCountAgg,
                                      Materialized.with(Serdes.String(),
Serdes.Long()))
                                     .toStream().to("output-topic");
```

Stateful Operations - Considerations

- In Kafka Streams stateful operations don’t emit results immediately
- Internal caching buffer results
  - Factors controlling when cache emits records
    - Cache is full (10MB by default)
    - Commit interval (30 seconds)
  - To see all updates, set cache size to zero (also for debugging)

Hands On: Aggregations

Hands On: Aggregations

Windowing

Windowing

- Aggregations continue to build up over time
- Windowing gives snapshot of an aggregate within a given timeframe
- Four types of windows we'll discuss
  - Tumbling
  - Hopping
  - Session
  - Sliding

Windowing

```
KStream<String, String> myStream = builder.stream("topic-A");
myStream.groupByKey().count().toStream().to("output")
```

Windowing - Hopping Window

```
KStream<String, String> myStream = builder.stream("topic-A");
Duration windowSize = Duration.ofMinutes(5);
Duration advanceSize = Duration.ofMinutes(1);
TimeWindows hoppingWindow =
  TimeWindows.of(windowSize).advanceBy(advanceSize);
myStream.groupByKey()
       .windowedBy(hoppingWindow)
       .count();
```

Windowing - Tumbling Window

```
KStream<String, String> myStream = builder.stream("topic-A");
Duration windowSize = Duration.ofSeconds(30);
TimeWindows tumblingWindow = TimeWindows.of(windowSize);

 myStream.groupByKey()
          .windowedBy(tumblingWindow)
          .count();
```

Windowing - Session Window

```
KStream<String, String> myStream = builder.stream("topic-A");
Duration inactivityGap = Duration.ofMinutes(5);

myStream.groupByKey()
        .windowedBy(SessionWindows.with(inactivityGap))
        .count();
```

Windowing - Sliding Window

```
KStream<String, String> myStream = builder.stream("topic-A");
Duration timeDifference = Duration.ofSeconds(2);
Duration gracePeriod = Duration.ofMillis(500);
myStream.groupByKey()
       .windowedBy(SlidingWindows.withTimeDifferenceAndGrace(
                                  timeDifference, gracePeriod))
       .count();
```

Windowing - Grace Period

```
TimeWindows.of(Duration.ofSeconds(30)).grace(Duration.ofSeconds(5));
```

Hands On: Windowing

Hands On: Windowing

Time Concepts

Time Concepts

- Timestamps are a critical component of Kafka.
- The Kafka message format has a dedicated **timestamp** field.
- **Event-time:** A producer, including the Kafka Streams library, automatically sets this timestamp field if user does not
  - Timestamp is the current wall-clock time of the Producer environment when the event is created
- **Ingestion-time:** can configure the Kafka broker to set this timestamp field when an event is appended to (stored in) the topic
  - Timestamp is the current wall-clock time of the Broker environment

Time Concepts

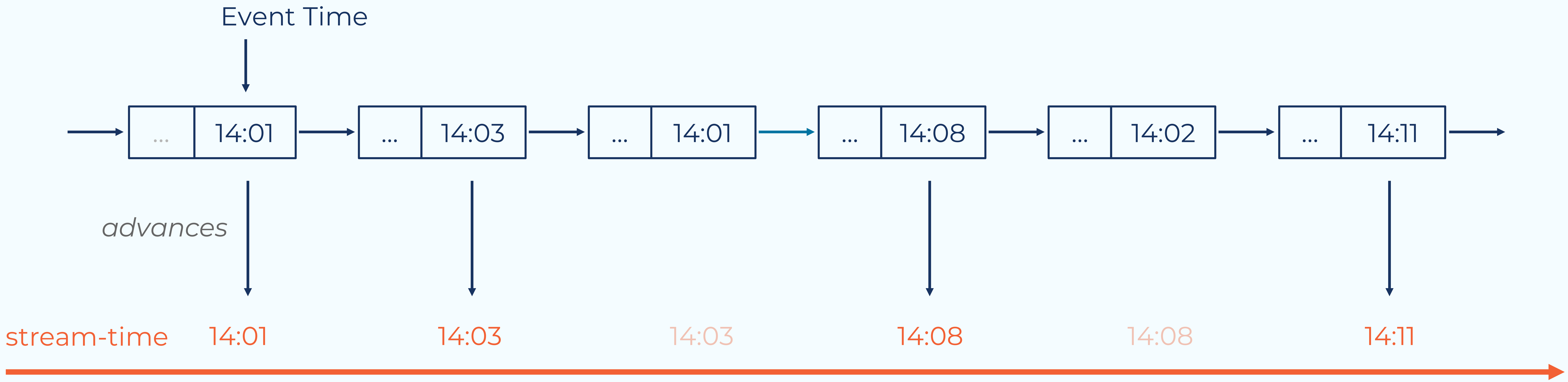
- Timestamps of events drive the action in Kafka Streams
- Earliest timestamp across all input partitions chosen first for processing
- Kafka Streams uses the **TimestampExtractor** interface to get timestamp
  - Default behavior is to use the event timestamp (set by either the event producer or the Kafka broker, see previous slide)
  - Default extractor is the **FailOnInvalidTimestamp**
  - Timestamp set by producer (event-time) or broker (ingestion-time)
- If it's desired to use a timestamp *embedded* in the event "payload" (i.e., the event key or the event value), provide a custom **TimestampExtractor** interface implementation

Time Concepts

- Time moves forward in Kafka Streams by these timestamps.
- For windowing operations this means the timestamps govern the opening and closing of windows.
  - How long a window remains open depends on timestamps only; it’s completely detached from wall-clock time
- Kafka Streams has a concept of **Stream Time**

Time Concepts - Stream Time

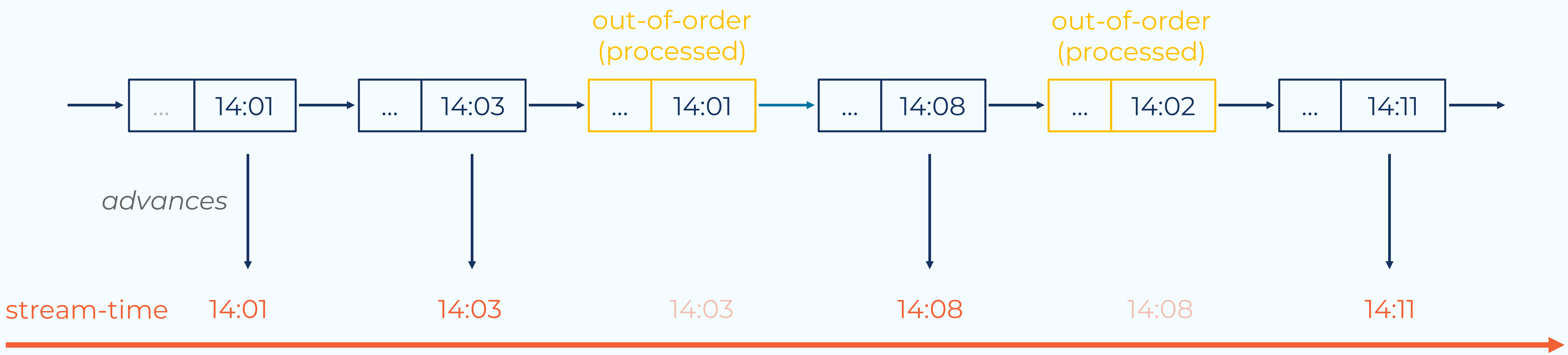
- Stream Time definition (based on “slack time” [1])
  - Largest timestamp seen so far
  - Only moves forward, never backward
  - If an out-of-order event arrives, stream-time stays where it is



[1] cf. Beyond Analytics: The Evolution of Stream Processing Systems (SIGMOD 2020), Aurora: a new model and architecture for data stream management (VLDB Journal 2003)

Time Concepts - Out-of-Order Input

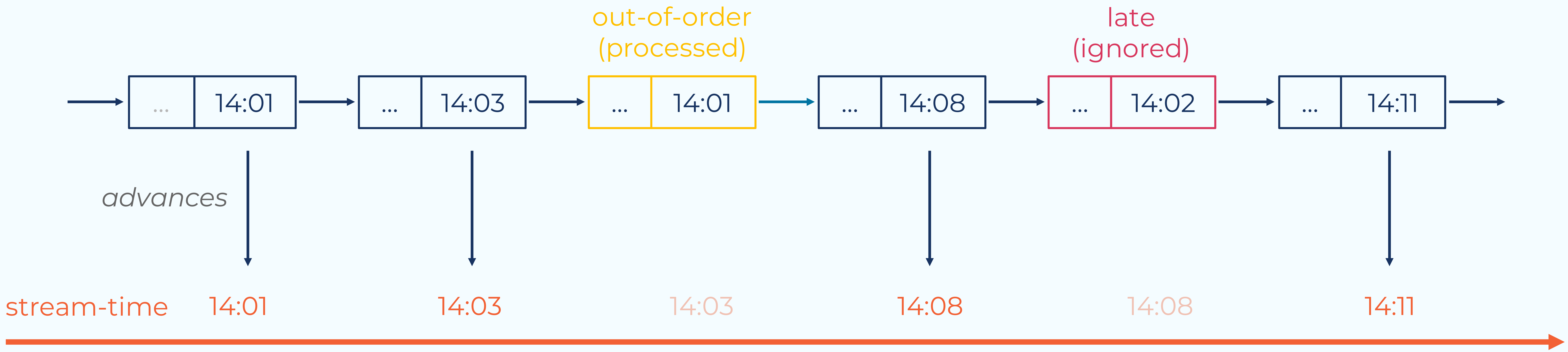
- Any input events with an **event-time** **<** **stream-time** are considered **out-of-order**
  - For windowed operations, this means the event-timestamp is less than the current stream-time, but is *within* the window time (window size plus grace period)
- Out-of-order records are accepted and processed



[1] cf. Beyond Analytics: The Evolution of Stream Processing Systems (SIGMOD 2020), Aurora: a new model and architecture for data stream management (VLDB Journal 2003)

Time Concepts - Late Input

- The grace period, a *per-window* setting, defines a **cut-off** for **out-of-order events**
- Any out-of-order events that arrive after the grace period are considered **(too) late**, and thus are ignored and not processed
- The delay of an event is determined by stream-time - event-timestamp



[1] cf. Beyond Analytics: The Evolution of Stream Processing Systems (SIGMOD 2020), Aurora: a new model and architecture for data stream management (VLDB Journal 2003)

Hands On: Time Concepts

Hands On: Time Concepts

Processor API

Processor API

- The DSL provides high-level operators then builds the topology for you
- For any options the DSL doesn't allow you can use Processor API
- Maximum flexibility
  - The developer is responsible for all the details
- Can build any type of stream processing application

Processor API

- Access to state stores for custom stateful operations
- You can call commit from processors
- Provides the ability to schedule arbitrary operations with Punctuation
  - Stream-time processing
  - Wall-clock time

Processor API

- Building streams applications with the Processor API follows this pattern
  - Add source node(s)
  - Add N number of processors that are child nodes of the source node(s)
  - Optionally create one or more StoreBuilder instances and attach them to the processing nodes
  - Add sink node(s) and make them child node of processor nodes

Processor API

- As you add nodes you'll use the name of one node as the parent name for another node
- Processors can have more than one parent node
- The Processor API gives you the flexibility to forward records to all child nodes or just select ones
- We'll see more details in the exercise for this module

Processor API - Example

```
Topology topology = new Topology();

topology.addSource(“source-node”, “topicA”, ”topicB”);

topology.addProcessor(“custom-processor”, new CustomProcessorSupplier(storeName), “source-node”);

toplogy.addSink(“sink-node”, “output-topic”, “custom-processor”);
```

Processor API

- It’s possible to “mix-in” the Processor API with the Streams DSL.
  - transform
  - transformValues
  - process
- This means you can e.g. the more convenient DSL for most processing steps of your application, and only need to use the Processor API for a few more complicated steps.

Hands On: Processor API

Hands On: Processor API

Testing

Testing Kafka Streams

- Testing is a critical part of software development
- Kafka Streams connects to brokers
- Expensive in unit tests to rely on broker connection
- The `TopologyTestDriver` solves the issue of unit testing a topology
- Note that integration testing with a live broker should still be done

Testing Kafka Streams TopologyTestDriver

- Build topology as usual including configurations
  - Use `MockSchemaRegistry` with `TopologyTestDriver`
- Instantiate `TopologyTestDriver` and use a `Topology and Properties` as constructor parameters
- Create `TestInputTopic` instances
- Call `TestInputTopic.pipeInput` with `KeyValue` objects
  - Overloaded methods allow for providing timestamps, `List` of records
- Executing `TestInputTopic.pipeInput` will trigger stream-time punctuation
  - Wall clock punctuations will fire only by calling the `advanceWallClockTime` method

Testing - Testable Application

- The Streams DSL has several operators that accept SAM interfaces
- Can easily use lambda expressions
- Downside - can’t easily test the lambda expressions in isolation
- Consider to write the concrete implementations of those interfaces
  - Can write single unit tests against ValueMapper, Reducer etc.

Testing - Integration Tests

- Good to have integration tests against a live broker
- Can see how stateful operations behave in real environment
  - **`TopologyTestDriver`** doesn't have the caching behavior or commits
  - Doesn't write to real topics
- Best choice for brokers in a test is to use **`TestContainers`**
  - Easily control broker life-cycle
  - Possible to share a container across multiple test classes
    - Improved testing time

Hands On: Testing

Hands On: Testing

Error Handling

Error Handling

- Errors are inevitable, especially in distributed systems and applications
- But not all errors are worthy of “stop the world”
  - Network partitions will *usually* resolve quickly
  - Change in partition ownership - very transient
- Need to provide a mechanism for gracefully handling errors
  - Recovery automatically when possible
  - Shut down when truly unrecoverable without intervention

Error Handling

- In Kafka Streams there are 3 broad categories where errors may occur
  - Entry - Consuming records - network or deserialization errors
  - Processing of records - not in expected format
  - Exit - Producing records - network or serialization errors
- For these broad categories there are handlers
  - Provide for users deciding to best deal with recoverable errors
  - In some cases the best approach is to acknowledge and continue
  - Other times more prudent to shut down
- We’ll cover the handlers Kafka Streams provides in all 3 areas

Error Handling - Consuming

- For errors consuming into Kafka Streams
  - The `DeserializationExceptionHandler` interface
    - Default configuration is the `LogAndFailExceptionHandler`
    - Other option is to use the `LogAndContinueExceptionHandler`
    - Can provide a custom implementation and provide the classname via the Kafka Streams configuration

Error Handling - Producing

- For errors producing from Kafka Streams
  - The **`ProductionExceptionHandler`** interface
    - Can respond with continue processing or fail
    - Default configuration is the **`DefaultProductionExceptionHandler`**
      - The default option always returns fail
      - For any other option you'll need to implement your own implementation
    - Only applies to exceptions not handled by Kafka Streams
      - **`RecordTooLargeException`**

Error Handling - Client Related

- Kafka Streams uses embedded producer and consumer instances
  - Clients can experience intermittent temporary failures
    - Network partition
    - Lead broker changes
  - Clients have their own configurations these situations
    - Tough to get client configurations correct.
    - Optimizing for resilience means blocking by the clients which has adverse effects on the Kafka Streams application
    - Too loose and there’s a risk of application shut downs for transient issues

Error Handling - Processing

- For processing errors
  - Exceptions from user logic bubble up and shut down the application
  - Kafka Streams provides the `StreamsUncaughtExceptionHandler`
    - Works for Exceptions not handled by Kafka Streams
    - The implementation you provide has three options
      - Replace the thread
      - Shutdown the individual streams instance
      - Shutdown all streams instances (with the same application id)

Error Handling - Client Related

- Kafka Streams solution is the **`task.timeout.config`**
    - When clients experience an issue Kafka Stream starts a timer
    - Kafka Streams attempts to make progress with other tasks
    - The task with the failed operation is retried
    - Procedure continues until success or the timeout is reached

Hands On: Error Handling

Hands On: Error Handling

Elasticity, Scaling, Parallelism

Elasticity, Scaling, Parallelism

- Kafka Streams internally uses the concept of a task for a work unit
  - Driven by the number of input partitions
  - Kafka Streams takes the highest partition count from the source node and that determines the number of tasks
  - Kafka Streams assigns tasks to StreamThread(s) one is the default
  - Can have as many threads there are tasks
    - Threads beyond this count are idle

Elasticity, Scaling, Parallelism

- Each Task represents a Topic-Partition from the source
- By increasing the thread count - automatically increases Kafka Streams processing power.
- Spinning up new Kafka Streams instances (with the same application ID) provides the same increase in throughput
    - Same ceiling applies more instances than tasks are idle - but can be available for failover

Elasticity, Scaling, Parallelism

- Since Kafka Streams uses **`KafkaConsumer`** instances internally it automatically inherits powerful dynamic scaling properties
  - Consumer group protocol
    - When a member leaves a rebalance reassigns resources to current active members
    - When a new member joins a rebalance pulls resources from existing members and gives them to the new member

Elasticity, Scaling, Parallelism

- The same protocol applies to Kafka Streams
  - For more processing power spin up as many Kafka Streams instances until all tasks are accounted for
  - In times where the level of traffic is reduced then take down Kafka Stream instances and resources are automatically assigned to active applications
- This behavior is completely dynamic - processing continues after a brief delay for the rebalance to complete

Fault Tolerance

Fault Tolerance

- Kafka Streams has stateless (e.g., map, filter) as well as stateful operations
- For stateful operations, such as aggregations and joins:
  - State stores are either persistent or in-memory
  - Backed by changelog topics for durability
    - Records written to the changelog as they are persisted
  - Losing a machine with a State Store can happen
    - When starting up Kafka Streams will detect a stateful node
    - If the state doesn’t exist fully restored from the changelog topic

Fault Tolerance

- In-memory stores don't retain records across restarts
  - Fully restored to from changelog topic
- Changelog topics use compaction
  - Oldest records by key are deleted
  - Safely leaves most recent records for that key

Fault Tolerance

- With a stateful operation restoring full state can take time
  - Kafka Streams offers stand-by tasks
  - Configure **`num.standby.replicas`** to number greater than the default setting of zero
    - Kafka Streams will designate another application instance as the “standby”
    - The standby instance keeps a mirrored state store in-sync with the original
    - When the primary goes down, the standby takes over with complete or minor restoration - no downtime restoring

Interactive Queries

Interactive Queries

- Typical pattern with event streaming is reporting
  - Dashboard applications
  - Require the streaming system write results to database
  - UI layer queries the database for live views
- Kafka Streams stateful operations, such as aggregations, represent the present “state” of an event stream
- Interactive queries let you query this state as well as query **`KTable`** instances

Interactive Queries

- Eligible for interactive queries: KTable and aggregations
- To enable Interactive queries:
  - Name the state store via `Materialized` or using the `Stores` factory
  - Set the `application.server`configuration `(host:port)`
- Each application instance has metadata for all instances of the same application `(same application.id)`
- The developer needs to provide the serving layer

Interactive Queries

- Kafka Streams does it without the need for an external database by allowing direct, read-only access to state stores and **`KTables`**
  - Live - ongoing as it's happening
  - No need to write intermediate results - simplifies architecture
  - A materialized view of the operation in real-time
- ksqlDB: There's also the option to use SQL with ksqlDB to query tables.
  - Take a look at our ksqlDB courses to learn more