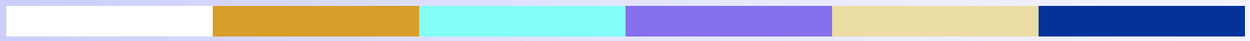
Kafka Streams Intro



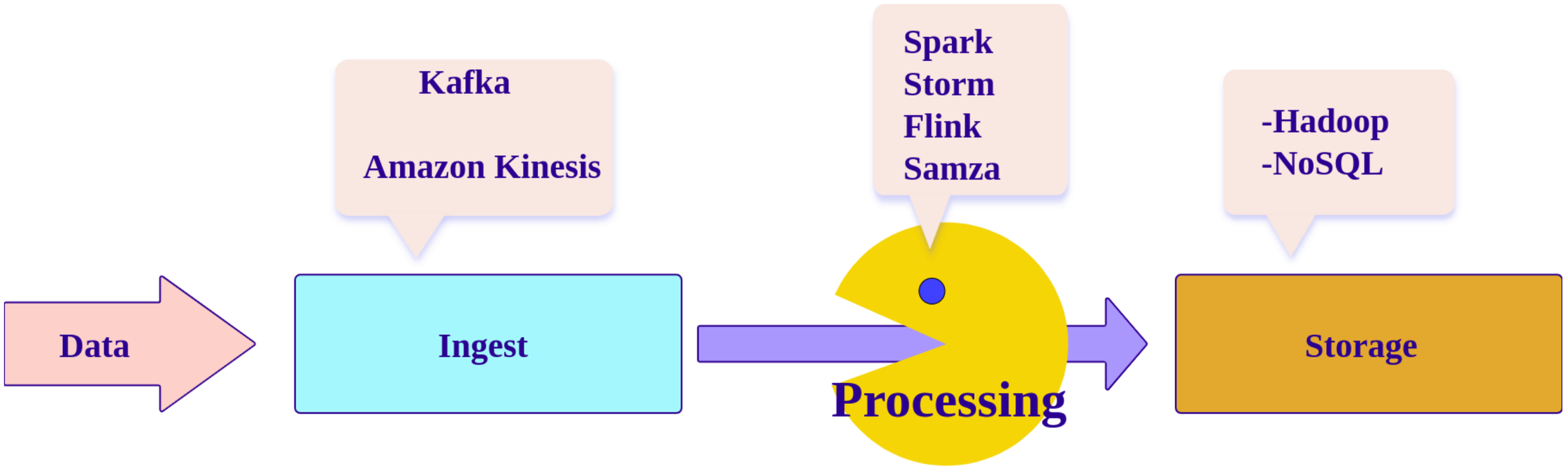
Lesson Objectives

- Learn Kafka Streams architecture
- Learn Kafka Streams API

Kafka Streams Intro

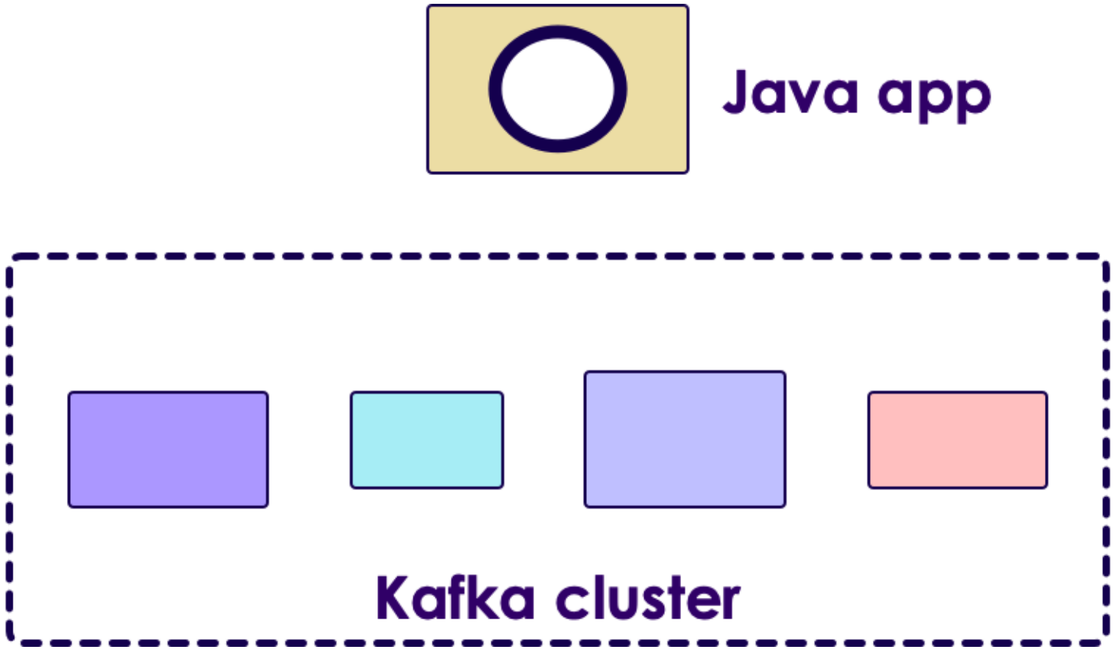
Streaming Platforms

- Kafka is a messaging bus
- The 'processing' portion was done outside Kafka



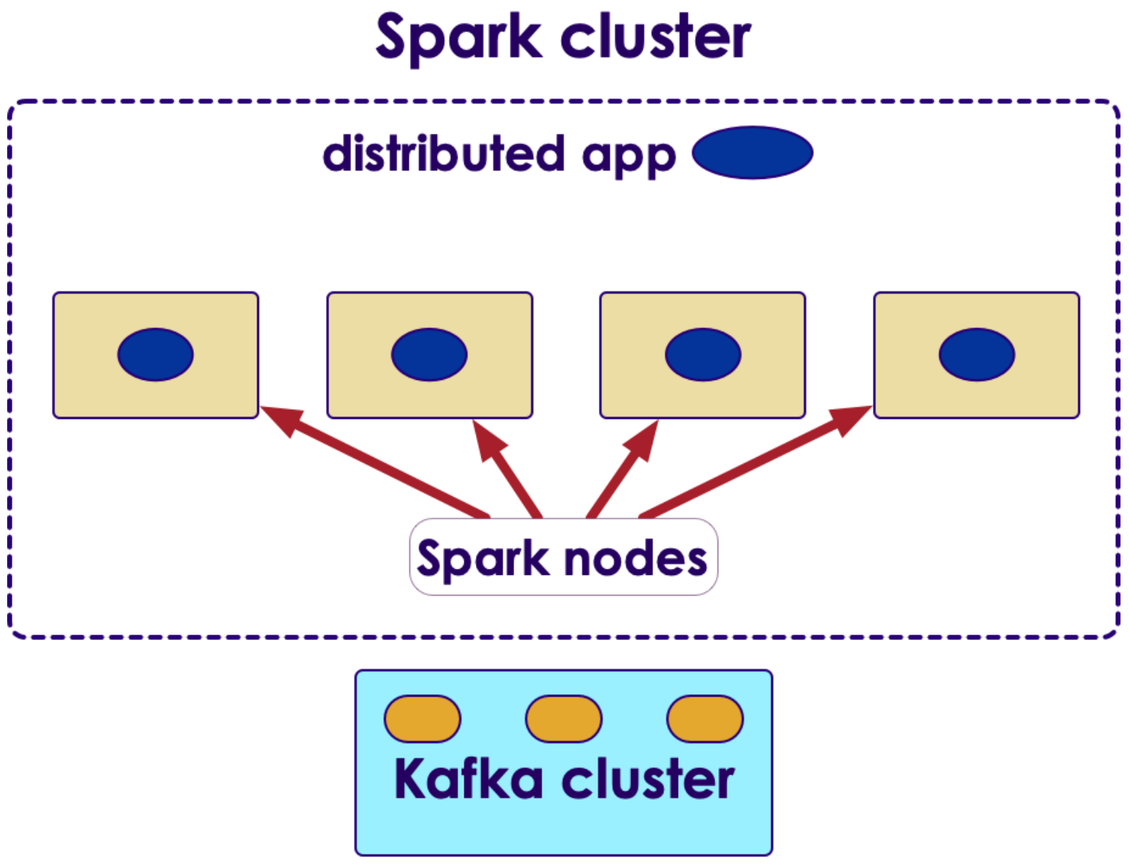
Kafka Application Using Java

- Pros: easy, simple
- Cons: not scalable, not fault tolerant



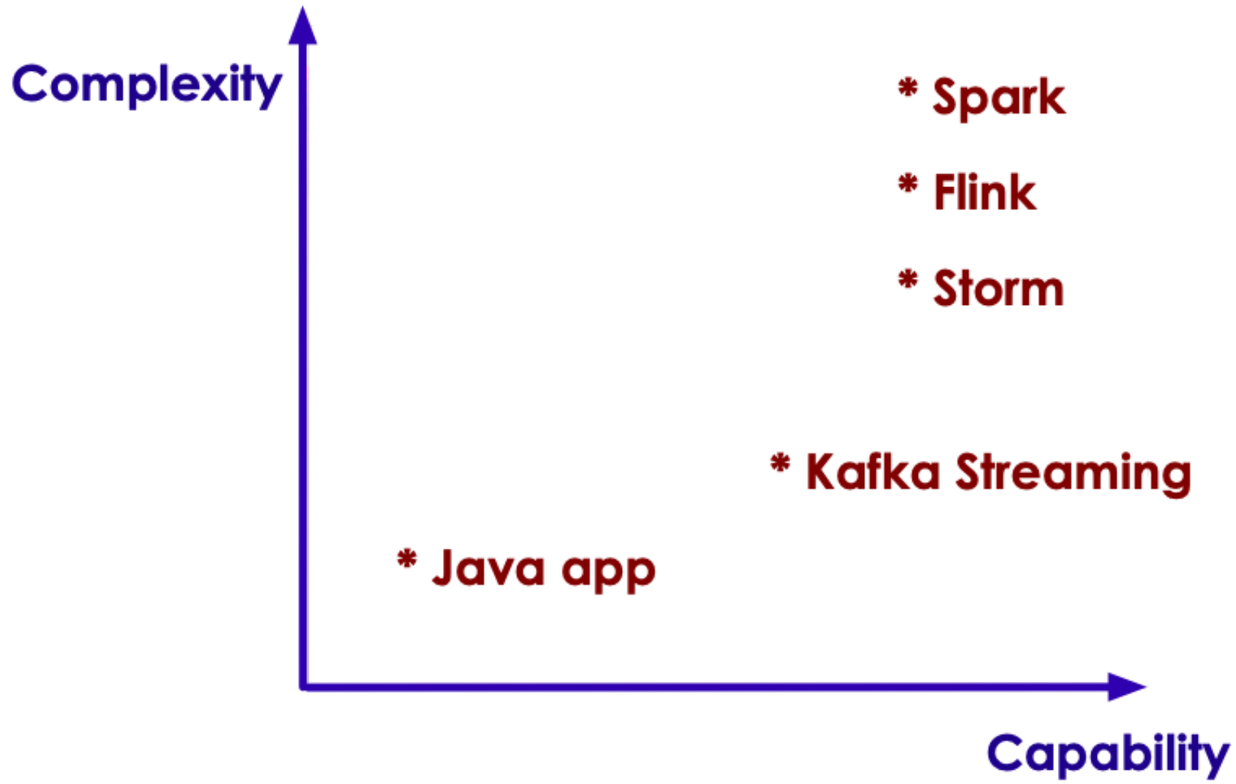
Kafka Application Using Spark

- Pros: distributed, fault tolerant, lots of functionality
- Cons:
    - Need to setup and maintain a Spark cluster
    - Not so simple



Case for Kafka Streams

- Java / Python application using Kafka Producer/Consumer APIs
  - Simple
  - Limited capability
- Distributed systems like Spark / Flink
  - Excellent capabilities
  - Complex
- 'Kafka Streams' aims to fill the sweet spot between capabilities & complexities



Kafka Streams Features

- Kafka Streams is a **client-side** library for building distributed applications for Kafka
- Event-based processing (one event at a time). Not micro batch
- Stateful processing for joins / aggregations
- High level operations (map, filter, reduce)
- **Not** designed for analytics like Spark or Hive

Comparing Streams

- **Kafka Streams motto - "Build apps, not clusters"**

	Simple Java App	Kafka Streams	Distributed Streaming Frameworks
	Using Java/ Python	Java	Spark / Flink / Samza
Pros	-Simple to implement - Simple to deploy	-Simple to implement, - Simple to deploy	- Distributed out of the box, - Very good scaling, - Excellent capabilities like windowing / machine learning
Cons	- Hard to scale	- Medium difficulty	- Heavy weight, - Complex operations, - Need to build a cluster, - Monitor / maintain

Kafka Streams API

Kafka Streams (Abbreviated)

```
// ** 1 : configure **
Properties config = new Properties();
config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "kafka-streams-consumer1");
config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

// ** 2 : define processing **
final StreamsBuilder builder = new StreamsBuilder();
final KStream < String, String > clickstream = builder.stream("topic1");// topic

clickstream.print(Printed.toSysOut());


// ** 3 : start the stream **
final KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.cleanUp();
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

Lab: Kafka Streams Intro

- **Overview:**
  - Getting started with Kafka Streams
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.1
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Streams Operations

Function	Description
ForEach	Process one record at a time
Filter	Filter stream event by event
map	Transform the stream, (key1, value1) => (key2, value2)
groupBy	Group the stream by key
count	Count the stream

Kafka Streams: ForEach

```
final StreamBuilder builder = new StreamBuilder();
final KStream < String, String > clickstream = builder.stream("topic1");

// Foreach : process events one by one
clickstream.foreach(new ForeachAction < String, String >() {

     public void apply(String key, String value) {

         logger.debug("key:" + key + ", value:" + value);
    }
});
```

- Using Java 8 Lambda functions

Lab: Kafka Streams Foreach

- **Overview:**
  - Kafka Streams: Foreach
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.2
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Kafka Streams: Filter

- Applying a filter to a stream produces another stream

key	value
facebook	{… action="clicked" …}
twitter	{… action="blocked" …}
facebook	{… action="viewed" …}
youtube	{… action="clicked" …}
cnn	{… action="viewed" …}

Filter "Clicked"

key	value
facebook	{… action="clicked" …}
youtube	{… action="clicked" …}

Kafka Streams : Filter

```
final StreamBuilder builder = new StreamBuilder();
final KStream < String, String > clickstream = builder.stream("topic1");


// filter clicks only
final KStream<String, String> actionClickedStream = clickstream.filter((key, value) -> {
            try {
                ClickstreamData clickstreamData = gson.fromJson(value, ClickstreamData.class);
                return ((clickstreamData.action != null) && (clickstreamData.action.equals("clicked")));

            } catch (Exception e) {
                return false;
            }

        });

// another quick filter
final KStream < String, String > actionClickedStream =
    clickstream.
    filter((k, v) -> v.contains("action:clicked"));

actionClickstream.print(Printed.toSysOut());
```

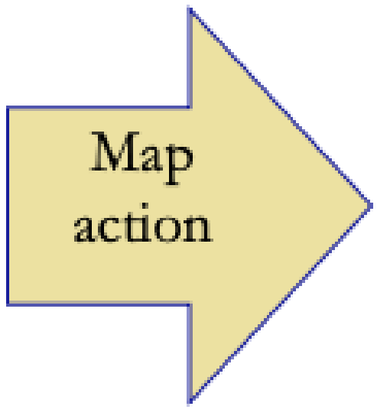
Lab: Kafka Streams Filter

- **Overview:**
  - Kafka Streams Filter
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.3
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Kafka Streams: Map

- Map **transforms** a stream into another stream
- KStream<key1, value1> -> KStream <key2, value2>
- Map action

key	value
facebook	{… action="clicked" …}
twitter	{… action="blocked" …}
facebook	{… action="viewed" …}
youtube	{… action="clicked" …}
cnn	{… action="viewed" …}



key	value
clicked	1
blocked	1
viewed	1
clicked	1
viewed	1

Kafka Streams: Map

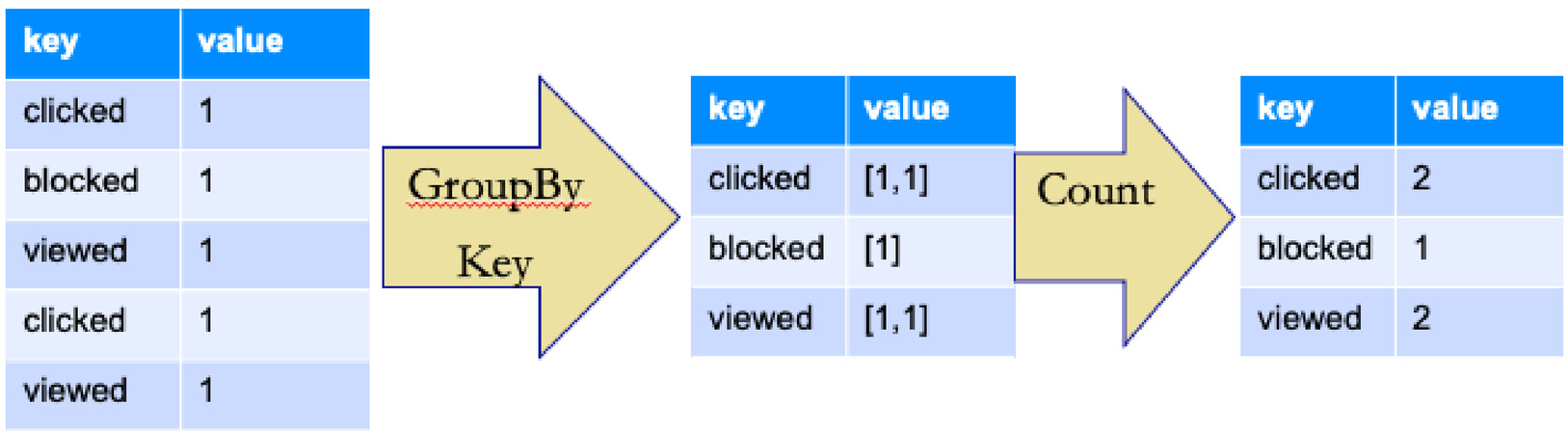
```
final StreamsBuilder builder = new StreamsBuilder();
final KStream < String, String > clickstream = builder.stream("topic1");

// map transform (String, String) to  (String, Integer)
final Gson gson = new Gson();

// k1 = domain,   v1 = {json}
// k2 = action    v2 = 1
final KStream<String, Integer> actionStream = clickstream.map (
    new KeyValueMapper<String, String, KeyValue<String, Integer>>() {

        public KeyValue<String, Integer> apply(String key, String value) {
            try {
                ClickstreamData clickstreamData = gson.fromJson(value, ClickstreamData.class);

                String action = clickstreamData.action;

                KeyValue<String, Integer> actionKV = new KeyValue<>(action, 1);

                return actionKV;
            } catch (Exception ex) {
                logger.error("", ex);
                return new KeyValue<String, Integer>("unknown", 1);
            }
        }
    });

}
);
actionStream.print(Printed.toSysOut());
```

Lab: Kafka Streams Map

- **Overview:**
  - Kafka Streams: Map
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.4
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Kafka Streams: GroupBy

- GroupBy will aggregate KStream by key
- Think of it like 'group by' operator in SQL



Lab: Kafka Streams Foreach

- **Overview:**
  - Kafka Streams: Foreach
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.2
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Kafka Streams: Join Example

- Source

```
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)));
```

KTable

KStreams vs. KTables

- **Kstream**
  - Each record/message represents an independent entity/event irrespective of its key.
- **Ktable**
  - Messages with same key are treated as updates of previous message.

```
//-------- KStream example ------

// reading from Kafka
KStream < byte[], String > textLines = builder.stream("textlines-topic",
            Consumed.with(Serdes.ByteArray(), Serdes.String()));

// Transforming data
KStream < byte[], String > upperCaseLines = textLines.mapValues(String::toUpperCase));
```

```
// ------ KTable Example ----

KTable < String, Long > wordCounts = textLines.flatMapValues(
        textLine -> Arrays.asList(textLine.toLowerCase().split("\\W+"))).
        groupBy((key,word) -> word).
        count()
```

Joins on Kstream and KTables

Kstream + KStream	Ktable + KTable	KTable + KStream
It is a sliding window join., Results a KStream, Supports Left, Inner and Outer Joins	Symmetric non-window join., Results a continuously updating Ktable., Supports Left, Inner and Outer Joins, (think like 2 database tables)	Asymmetric non-window join., Results a KStream., Supports Left and Inner join

Kafka Streams: GroupBy

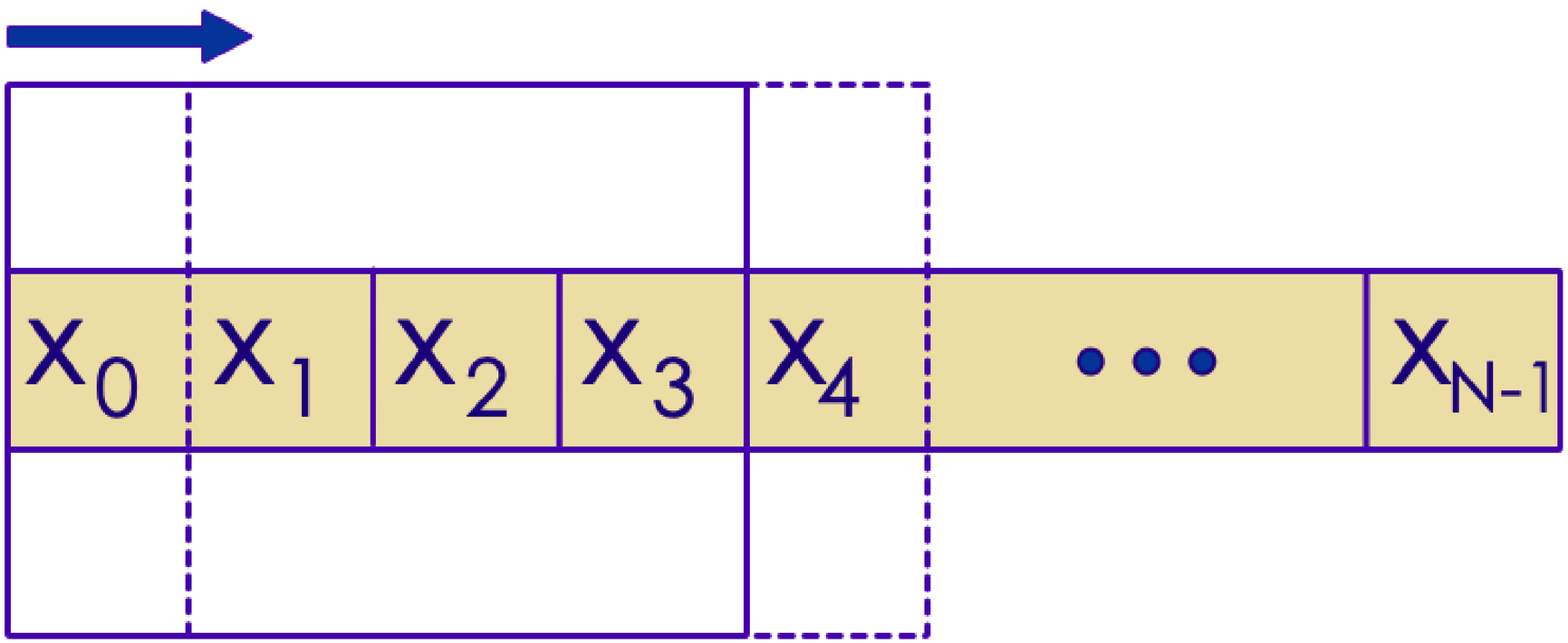
```
final StreamsBuilder builder = new StreamsBuilder();
final KStream < String, String > clickstream = builder.stream( "topic1");

// map transform (String, String) --> (String, Integer)

final KStream < String, Integer > actionStream = clickstream.map( ... )

// Now aggregate and count actions
// we have to explicitly state the K,V serdes in groupby,
// as the types are changing

final KTable < String, Long > actionCount = actionStream
   .groupByKey(Serialized.with(Serdes.String(), Serdes.Integer()))
   .count ();

actionCount.toStream().print(Printed.toSysOut());
```

Wordcount in Kafka Streams

```
// Serializers/deserializers (serde) for String and Long types
final Serde < String > stringSerde = Serdes.String();
final Serde < Long > longSerde = Serdes.Long();

// Construct a `KStream` from the input topic "topic1", where message values
// represent lines of text (for the sake of this example, we ignore whatever may be stored
// in the message keys).
KStream < String, String > textLines = builder.stream("topic1",
.with(stringSerde, stringSerde);

KTable < String, Long > wordCounts = textLines
    // Split each text line, by whitespace, into words.
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

    // Group the text words as message keys
    .groupBy((key, value) -> value)

    // Count the occurrences of each word (message key).
    .count();

// Store the running counts as a changelog stream to the output topic.
wordCounts.toStream().to("topic1-out", Produced.with(Serdes.String(), Serdes.Long()));
```

Windowing Operations

- Windowing is a common function in event processing
  - What is the average CPU utilization?
    - Over the last 5 minutes?
- Create groups of records with the *same key* for aggregations or joins into " **windows** "



Windowing Parameters

- Retention Period
  - How long to wait for late-arriving records for a given window
- Advance Period/Interval
  - How much to move the window forward relative to the last one
- Window Size
  - Size of the window i.e. how long is the window in time units
- Maintain Period
  - How long to keep the window alive

Windowing Example

- TimeWindows.of("cpu-window", 60*1000)
  - Returns a time window of 1 min.
  - Advance period of 1 min.
  - Window maintained for 1 day
- Modify various parameters using functions in TimeWindows class
- https://kafka.apache.org/20/javadoc/org/apache/kafka/streams/kstream/TimeWindows.

Counts visits per hour

```
KStreamBuilder builder = new KStreamBuilder();
KStream < String, Long > visitsStream = builder.stream(Serdes.String(), Serdes.Long(),
                                        "visitsTopic");

// Group and count visits per URL/page
KGroupedStream < String, Long > groupedStream =
visitsStream.groupByKey();
KTable < String, Long > totalCount = groupedStream.count("totalVisitCount");

// Create window for visits per hour
KTable < Windowed < String >, Long > windowedCount =
groupedStream.count(TimeWindows.of(60 * 60 * 1000), "hourlyVisitCount");
```

Lab: Kafka Streams Windows

- **Overview:**
  - Kafka Streams: Windows
- **Approximate Time:**
  - 10 - 15 mins
- **Instructions:**
  - Please follow: lab 7.6
- **To Instructor:**
  - Please demo this lab on screen and do it together with students

Review and Q&A

- Let's go over what we have covered so far
- Any questions?

Backup Slides

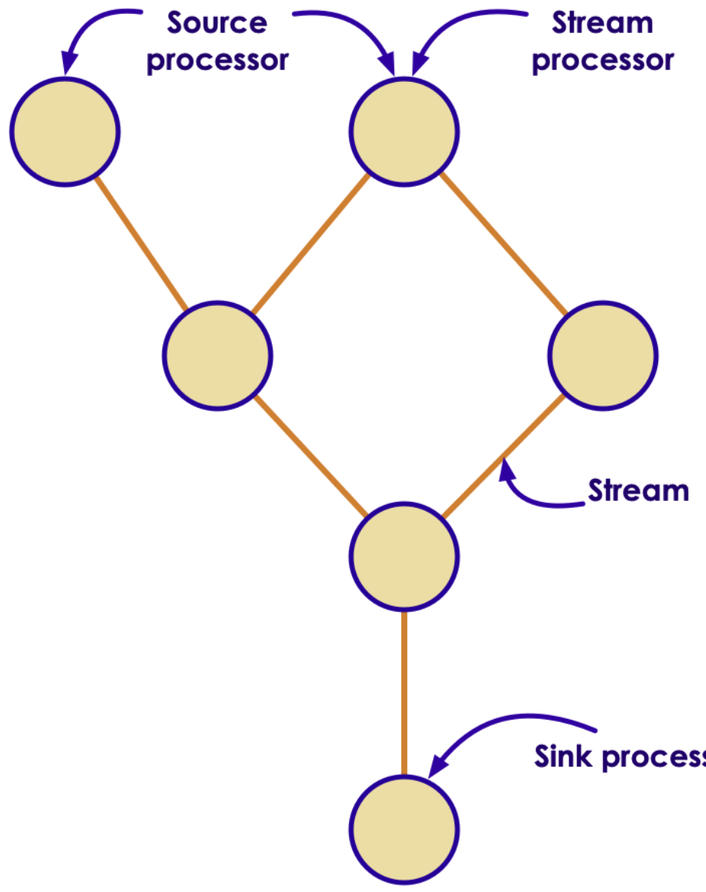
Modifying RocksDB Configuration

- Setting cache size to 16 Meg

```
public static class CustomRocksDBConfig implements RocksDBConfigSetter {
   @Override
   public void setConfig (final String storeName, final Options options,
   final Map < String, Object > configs) {

       BlockBasedTableConfig tableConfig = new
org.rocksdb.BlockBasedTableConfig();

     tableConfig.setBlockCacheSize(16 * 1024 * 1024L);
     /*
      * set more configuration here
      */
   }
}
Properties streamsSettings = new Properties();
streamsSettings.put(
  StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG,
  CustomRocksDBConfig.class);
```

Processor Topology

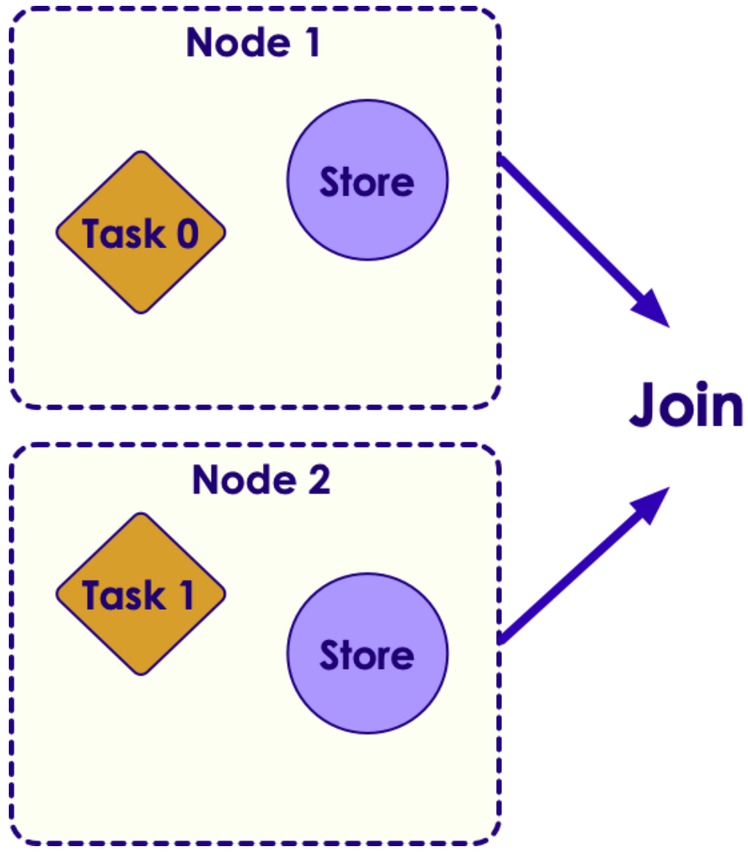
- Defines the logic for the application
- Topology is a graph
  - Nodes: Stream processors
  - Edges: Streams
- **Source processor**
  - Has no upstream processors. *Reads* topic
- **Sink processor**
  - Has no downstream processor. *Writes* topic



PROCESSOR TOPOLOGY

State Store

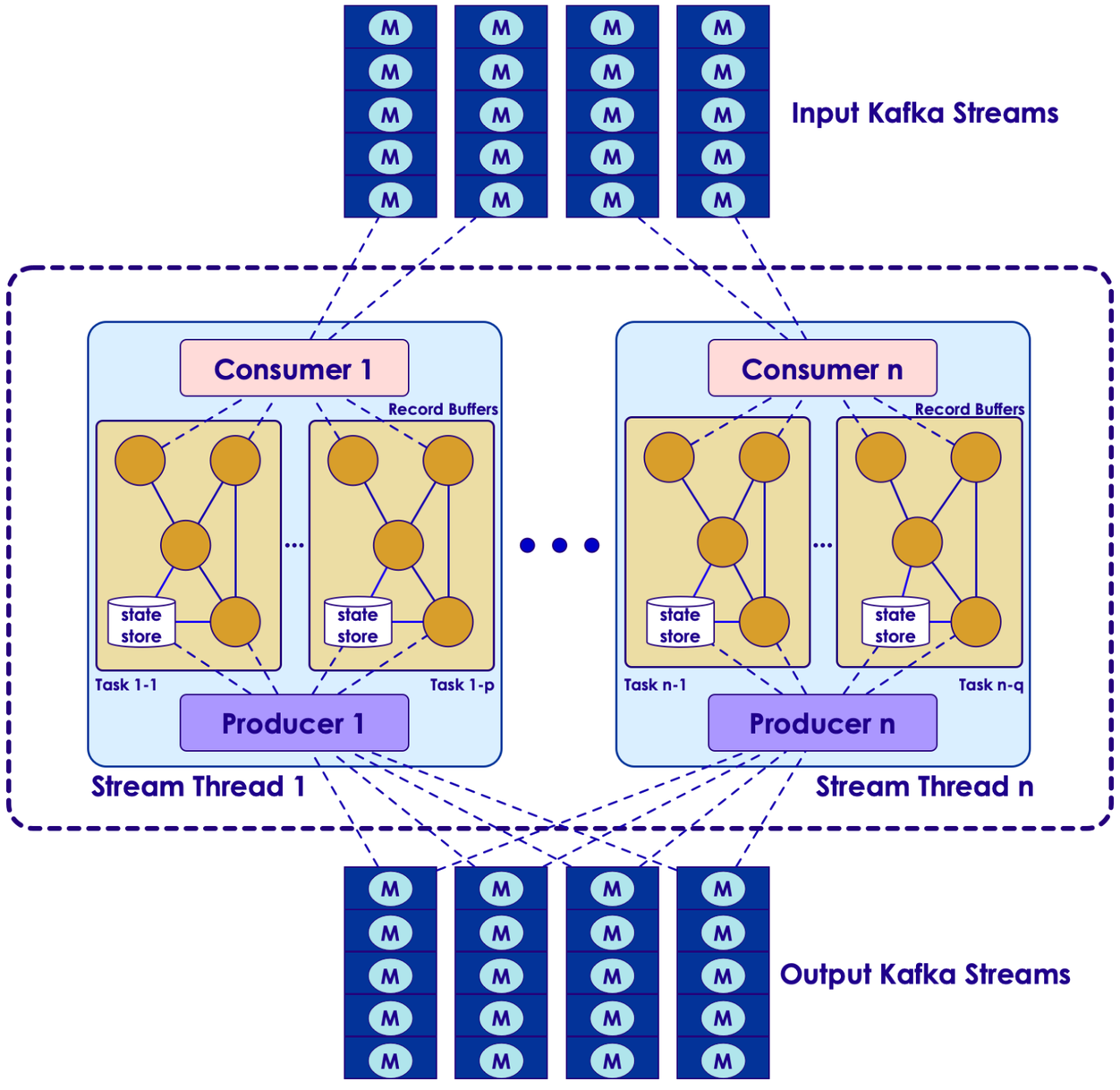
- Stateful operations like (Aggregations / Joins) require intermediate state storage
- Kafka Streams provides this storage at per node level
- Storage mediums
  - In memory cache
  - RocksDB (a very fast embedded DB, developed by Facebook)
    Stored on disk on each node
- Tasks uses it to store and query data
- Every task can have one or more state stores
- Fault tolerant
- Automatic recovery



Replication and Fault Tolerance

- Kafka Partitions are replicated and highly available
- If Streams task fails
  - Kafka will restart it on another running instance of the application
- Stream data persisted to Kafka is still available in case application fails and wants to re-process it
- Local state stores are replicated as a topic called **changelog**
  - **Changelog** has log compaction enabled

Overall Architecture



Writing a Streams Application

- Use Kafka Streams DSL
  - High level API
  - Provides most common required functions for transformation, grouping, aggregation
- Use Processor API
  - Low-level API
  - Create, connect processors in topology and interact with State Stores directly

Why Streaming from Database (CDC)?

- Integrations with Legacy Applications
  - Avoid dual writes when integrating with legacy systems
- Smart Cache Invalidation
  - Automatically invalidate entries in a cache as soon as the record(s) for entries change or are removed.
- Monitoring Data Changes
  - Immediately react to data changes committed by application/user.
- Data Warehousing
  - Atomic operation synchronizations for ETL-type solutions.
- Event Sourcing (CQRS)
- Totally ordered collection of events to asynchronously update the read-only views while writes can be recorded as normal