

Next Level Kafka

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or professional environment.



Table of Contents

1. Welcome to Kafka Streams: 3
 2. Kafka quickly: 43
 3. Developing Kafka Streams: 114
 4. Streams and state: 170
 5. The KTable API: 231
 6. The Processor API: 288
 7. Monitoring and performance: 333
 8. Testing a Kafka Streams application: 377
 9. Advanced applications with Kafka Streams: 407
- 

Part 1: Getting started with Kafka Streams

A photograph of a person's hands typing on a silver laptop keyboard. The background is slightly blurred, showing an office environment with a desk, papers, and a chair. The overall image has a professional and technical feel.

1. Welcome to Kafka Streams

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or development environment.

Welcome to Kafka Streams

This lesson covers

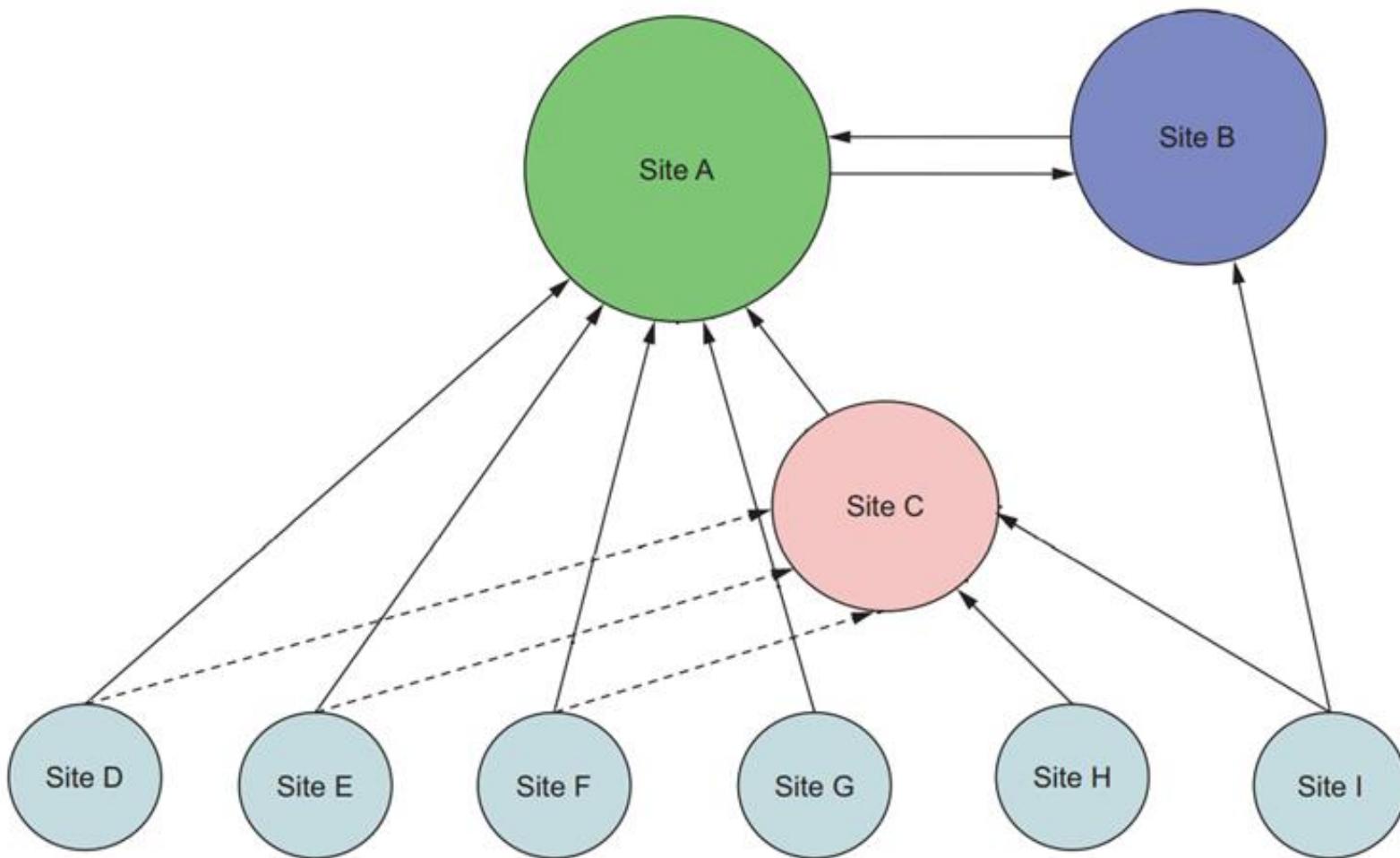
- Understanding how the big data movement changed the programming landscape
- Getting to know how stream processing works and why we need it
- Introducing Kafka Streams
- Looking at the problems solved by Kafka Streams

The big data movement

- The modern programming landscape has exploded with big data frameworks and technologies.
- Sure, client-side development has undergone transformations of its own, and the number of mobile device applications has exploded as well.
- But no matter how big the mobile device market gets or how client-side technologies evolve, there's one constant

The genesis of big data

- The internet started to have a real impact on our daily lives in the mid-1990s.
- Since then, the connectivity provided by the web has given us unparalleled access to information and the ability to communicate instantly with anyone, anywhere in the world.
- An unexpected byproduct of all this connectivity emerged: the generation of massive amounts of data



Important concepts from MapReduce

- The map and reduce functions weren't new concepts when Google developed MapReduce.
- What was unique about Google's approach was applying those simple concepts at a massive scale across many machines.
- At its heart, MapReduce has roots in functional programming.

Important concepts from MapReduce

- A simple example in Java 8, where a LocalDate object is mapped into a String message, while the original LocalDate object is left unmodified:

```
Function<LocalDate, String> addDate =  
    (date) -> "The Day of the week is " + date.getDayOfWeek();
```

Important concepts from MapReduce

- The steps to reduce a List<Integer> containing the values 1, 2, and 3:

$$\begin{array}{rcl} 0 & + & 1 = 1 \\ 1 & + & 2 = 3 \\ 3 & + & 3 = 6 \end{array}$$

Adds the seed value
to the first number

Adds the sum of step 2
to the third number

Takes the result from
step 1 and adds it to
the second number
in the list

Important concepts from MapReduce

- The following example shows an implementation of a simple reduce function using a Java 8 lambda:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);  
  
int sum = numbers.reduce(0, (i, j) -> i + j );
```

Important concepts from MapReduce

DISTRIBUTING DATA ACROSS A CLUSTER TO ACHIEVE SCALE IN PROCESSING

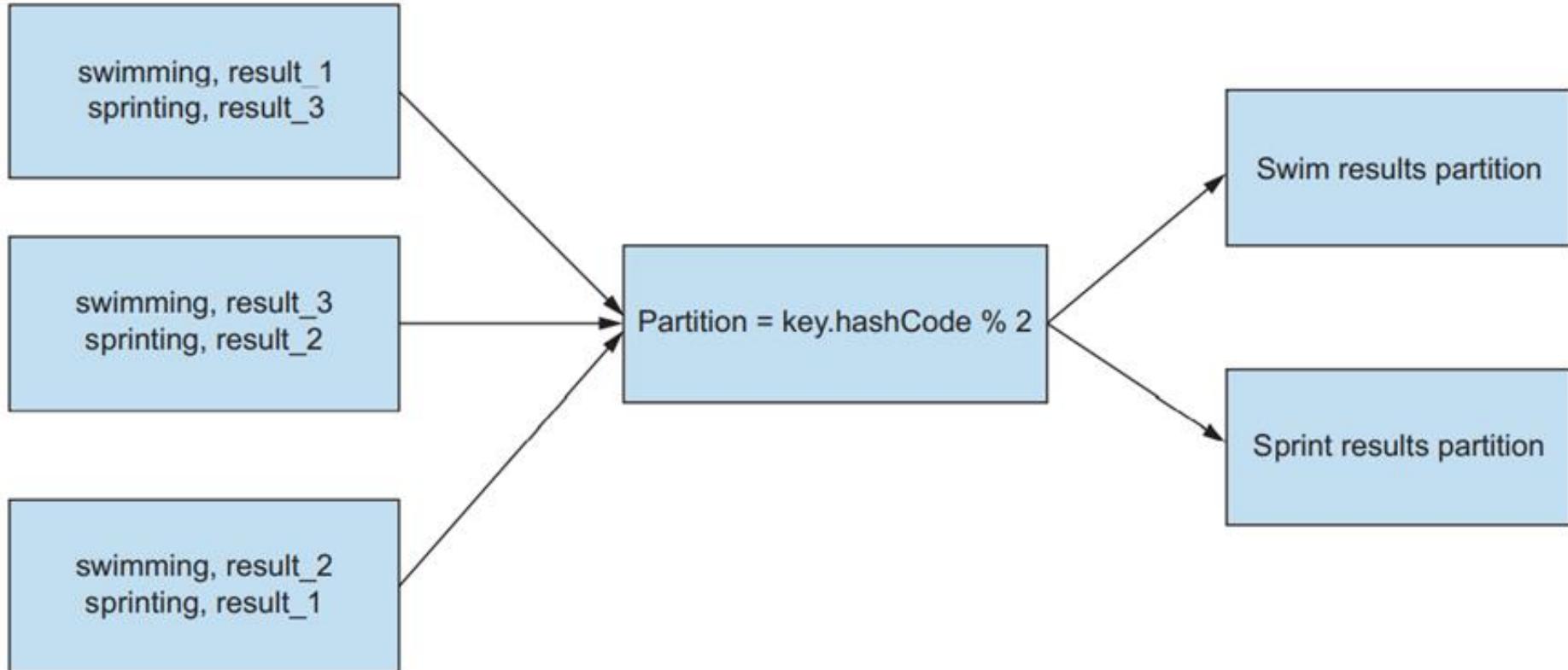
Number of machines	Amount of data processed per server
10	500 GB
100	50 GB
1000	5 GB
5000	1 GB

Important concepts from MapReduce

USING KEY/VALUE PAIRS AND PARTITIONS TO GROUP DISTRIBUTED DATA

- To regroup distributed data, you can use the keys from the key/value pairs to partition the data.
- The term partition implies grouping, but I don't mean grouping by identical keys, but rather by keys that have the same hash code.
- To split data into partitions by key, you can use the following formula:

```
int partition = key.hashCode() % numberOfPartitions
```



Important concepts from MapReduce

EMBRACING FAILURE BY USING REPLICATION

- Another key component of Google's MapReduce is the Google File System (GFS). Just as Hadoop is the open-source implementation of MapReduce, Hadoop File System (HDFS) is the open-source implementation of GFS.
- At a very high level, both GFS and HDFS split data into blocks and distribute those blocks across a cluster

Batch processing is not enough

- Hadoop caught on with the computing world like wildfire.
- It allowed people to process vast amounts of data and have fault tolerance while using commodity hardware (cost savings).
- But Hadoop/MapReduce is a batch-oriented process, which means you collect large amounts of data, process it, and then store the output for later use

Introducing stream processing

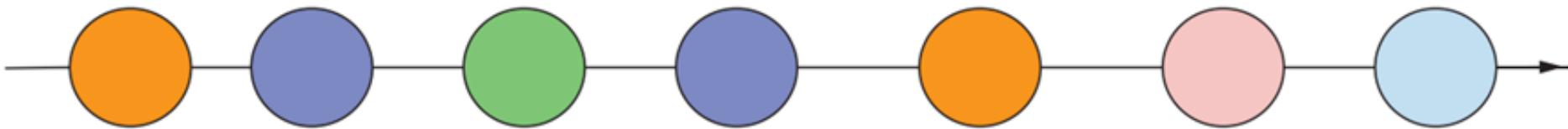


Figure represents a stream of data, with each circle on the line representing data at a point in time. Data is continuously flowing, as data in stream processing is unbounded

When to use stream processing, and when not to use it

Like any technical solution, stream processing isn't a one-size-fits-all solution. The need to quickly respond to or report on incoming data is a good use case for stream processing. Here are a few examples:

- Credit card fraud
- Intrusion detection
- A large race, such as the New York City Marathon
- The financial industry

When to use stream processing, and when not to use it

The focus is on analyzing data over time, rather than just the most current data:

- Economic forecasting
- School curriculum changes

Handling a purchase transaction

- Let's start by applying a general stream-processing approach to a retail sales example.
- Then we'll look at how you can use Kafka Streams to implement the stream-processing application.
- Suppose Jane Doe is on her way home from work and remembers she needs toothpaste.
- She stops at a ZMart, goes in to pick up the toothpaste, and heads to the checkout to pay.

Weighing the stream-processing option

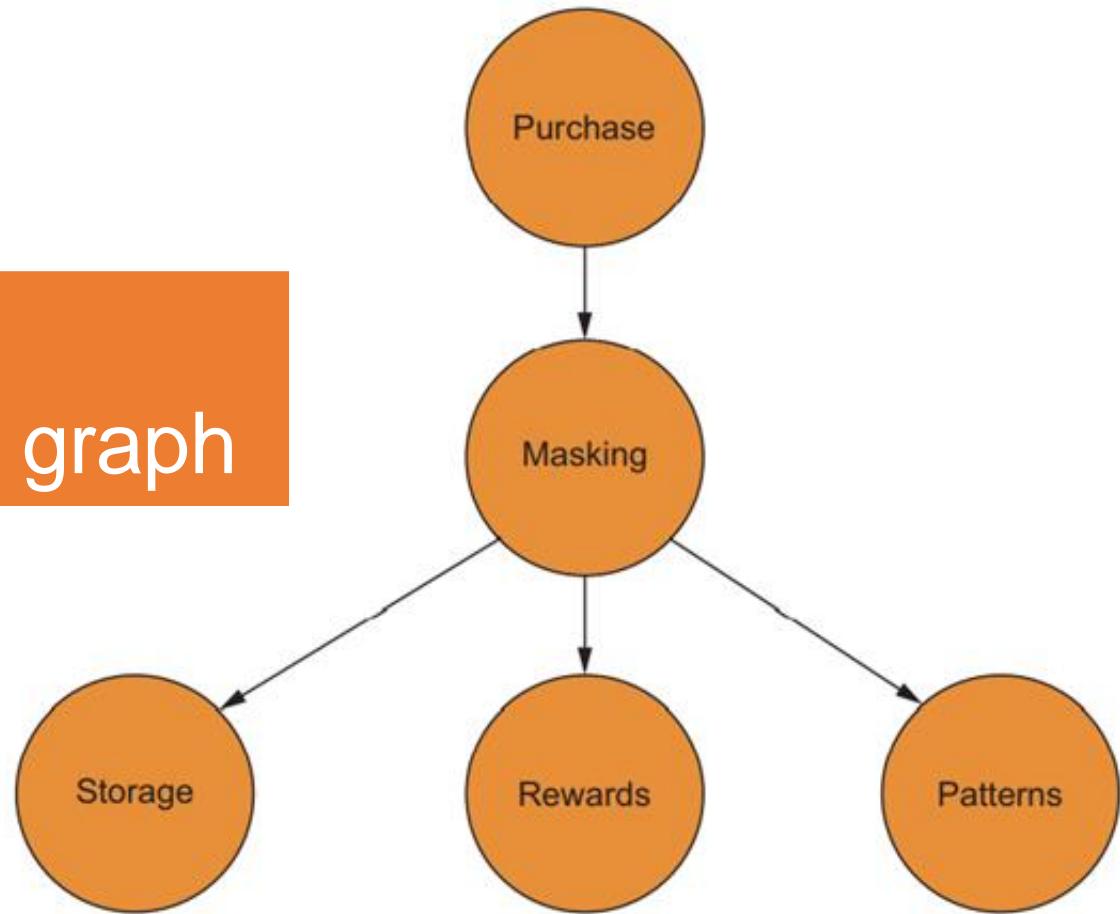
- Suppose you're the lead developer for ZMart's streaming-data team.
- ZMart is a big box retail store with several locations across the country.
- ZMart does great business, with total sales for any given year upwards of \$1 billion.
- You'd like to start mining the data from your company's transactions to make the business more efficient.

Weighing the stream-processing option

You get together with management and your team and produce the following four primary requirements for the stream-processing initiative to succeed:

- Privacy
- Customer rewards
- Sales data
- Storage

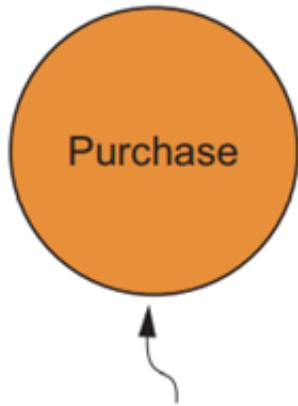
Deconstructing the requirements into a graph



Changing perspective on a purchase transaction

- In this section, we'll walk through the steps of a purchase and see how it relates, at a high level, to the requirements graph from previous figure.

Source node



The point of purchase is the source or parent node for the entire graph.

Figure The simple start for the sales transaction graph. This node is the source of raw sales transaction information that will flow through the graph.

Credit card masking node

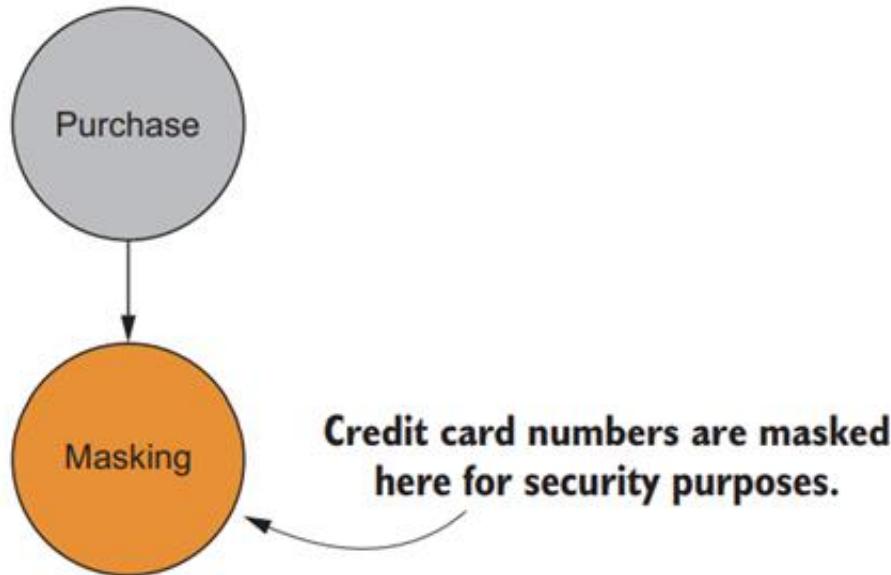


Figure The first node in the graph that represents the business requirements. This node is responsible for masking credit card numbers and is the only node that receives the raw sales data from the source node, effectively making it the source for all other nodes connected to it.

Patterns node

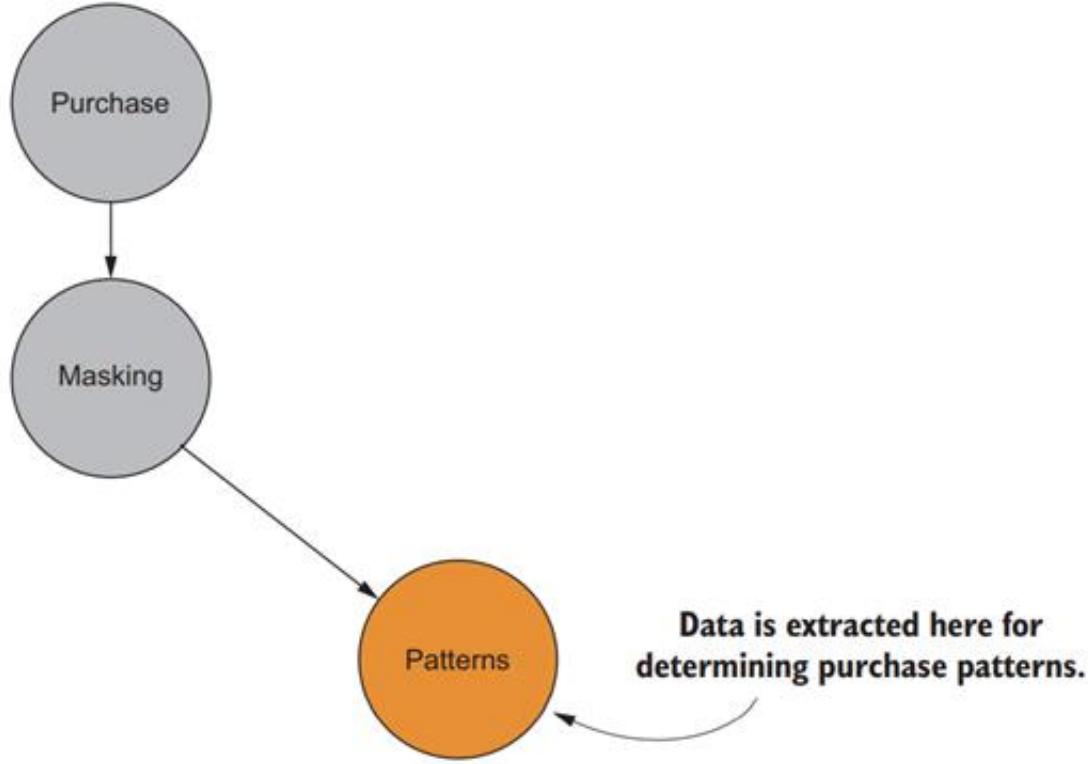


Figure : The patterns node consumes purchase information from the masking node and converts it into a record showing when a customer purchased an item and the ZIP code where the customer completed the transaction.

Rewards node

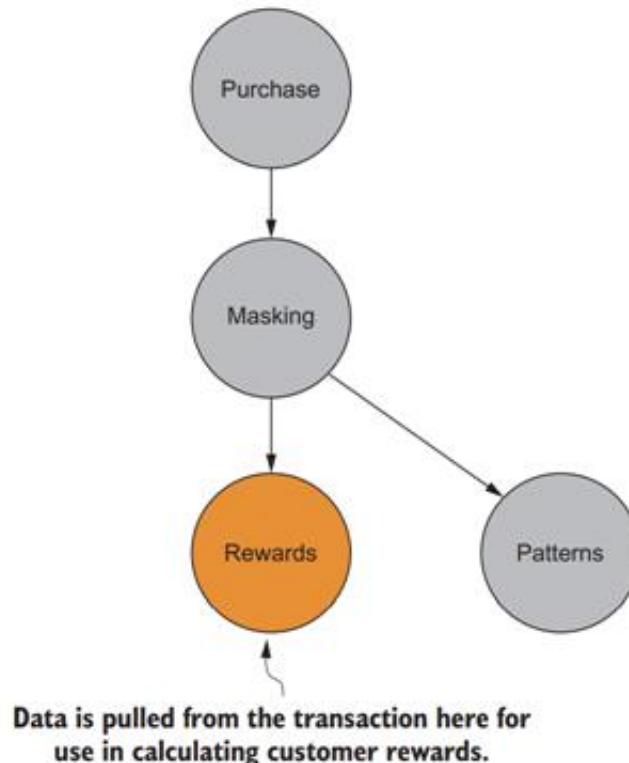
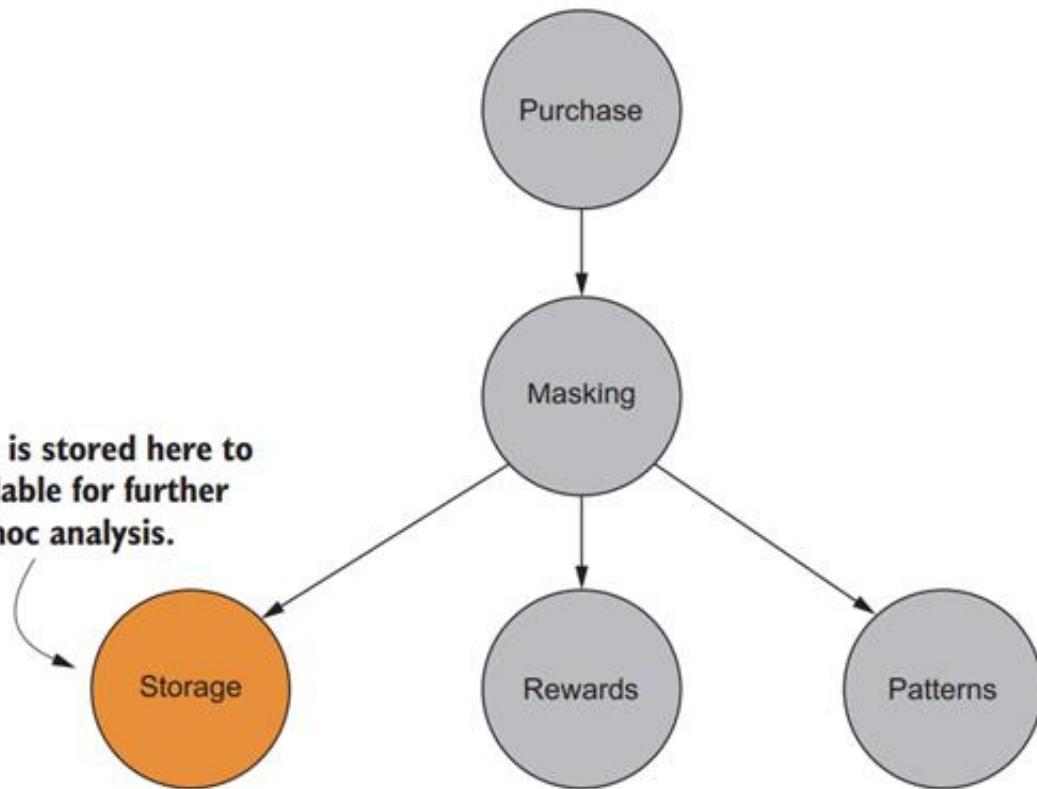


Figure The rewards node is responsible for consuming sales records from the masking node and converting them into records containing the total of the purchase and the customer ID.



Storage node

Figure The storage node consumes records from the masking node as well. These records aren't converted into any other format but are stored in a NoSQL data store for ad hoc analysis later.

Kafka Streams as a graph of processing nodes

- Kafka Streams is a library that allows you to perform per-event processing of records.
- You can use it to work on data as it arrives, without grouping data in microbatches.
- You process each record as soon as it's available.

Applying Kafka Streams to the purchase transaction flow

- Let's build a processing graph again, but this time we'll create a Kafka Streams program.
- Remember, the vertexes are processing nodes that handle data, and the edges show the flow of data

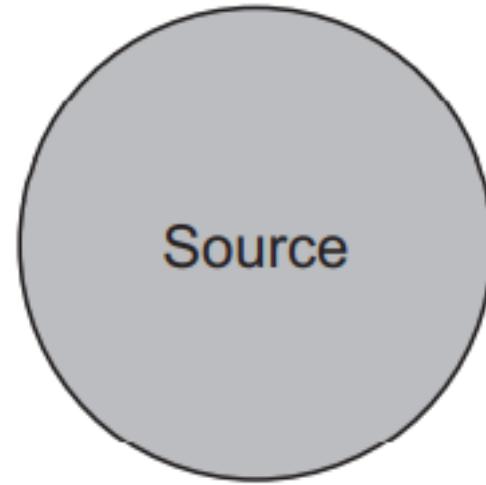
Defining the source

The first step in any Kafka Streams program is to establish a source for the stream. The source could be any of the following:

- A single topic
- Multiple topics in a comma-separated list
- A regex that can match one or more topics

Defining the source

Figure represents the source node in the topology.



The first processor: masking credit card numbers

- Now that you have a source defined, you can start creating processors that will work on the data.
- Your first goal is to mask the credit card numbers recorded in the incoming purchase records.
- The first processor will convert credit card numbers from something like 1234-5678-9123-2233 to xxxx-xxxx-xxxx-2233

Source node consuming message from the Kafka transaction topic

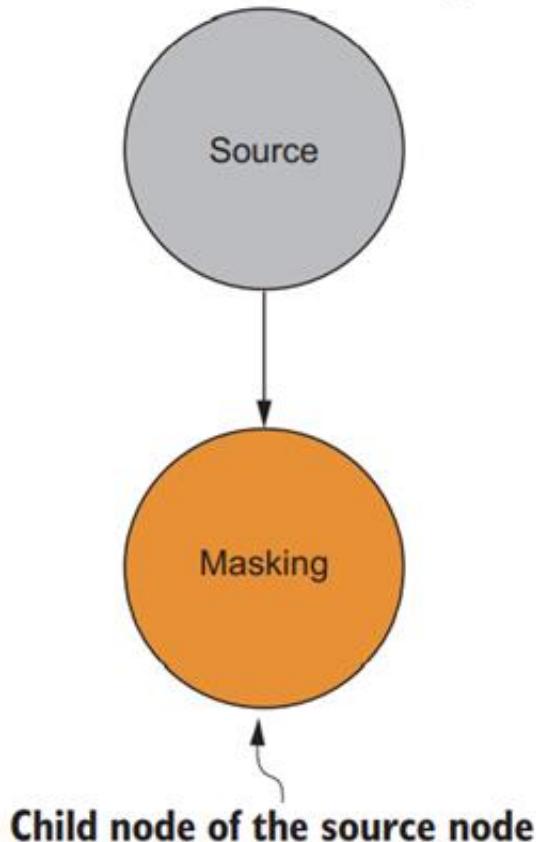


Figure The masking processor is a child of the main source node. It receives all the raw sales transactions and emits new records with the credit card number masked.

CREATING PROCESSOR TOPOLOGIES

- Each time you create a new KStream instance by using a transformation method, you're in essence building a new processor that's connected to the other processors already created.
- By composing processors, you can use Kafka Streams to create complex data flows elegantly

The second processor: purchase patterns

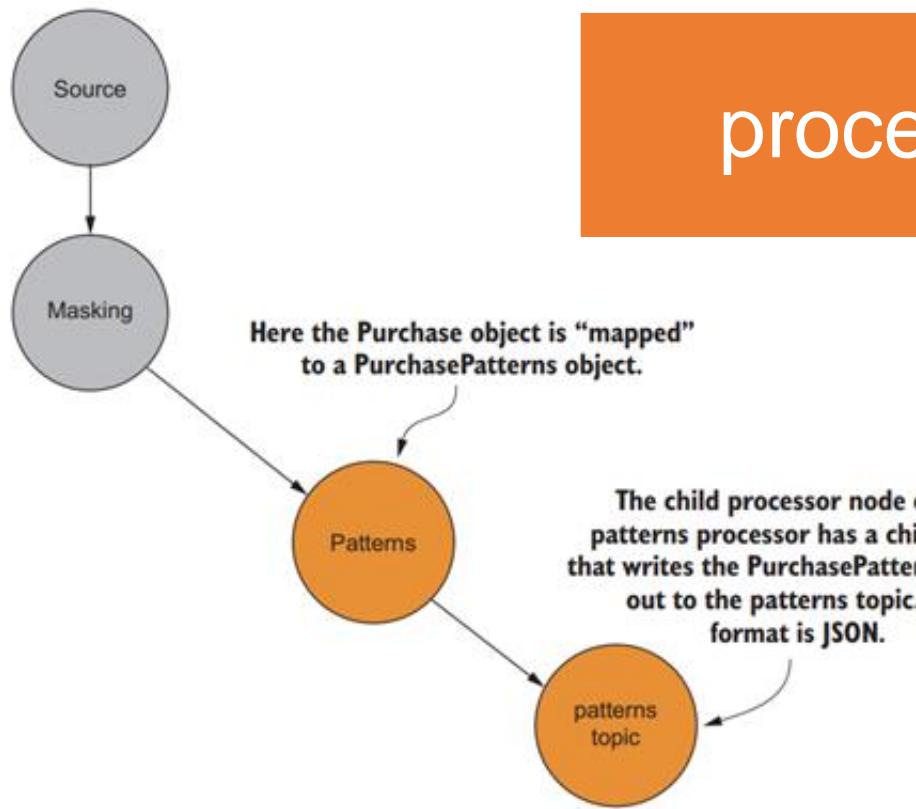


Figure : The purchase-pattern processor takes Purchase objects and converts them into PurchasePattern objects containing the items purchased and the ZIP code where the transaction took place. A new processor takes records from the patterns processor and writes them out to a Kafka topic.

The third processor: customer rewards

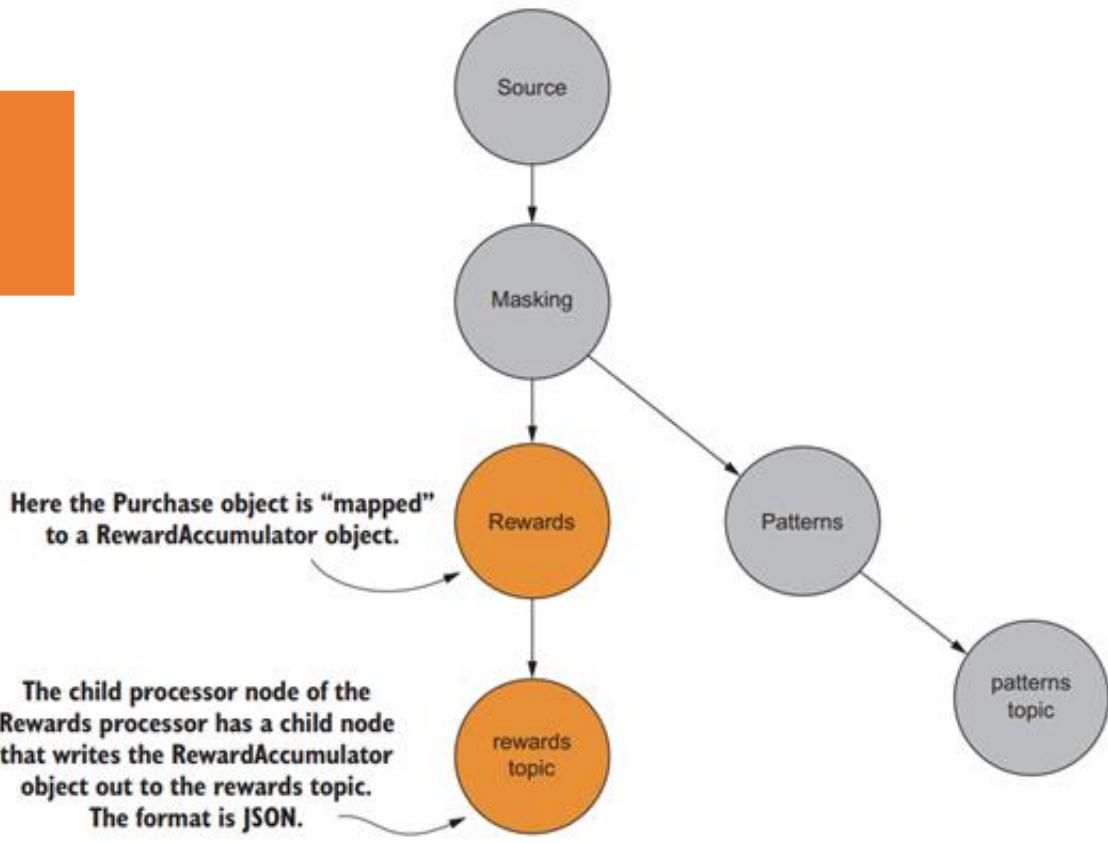
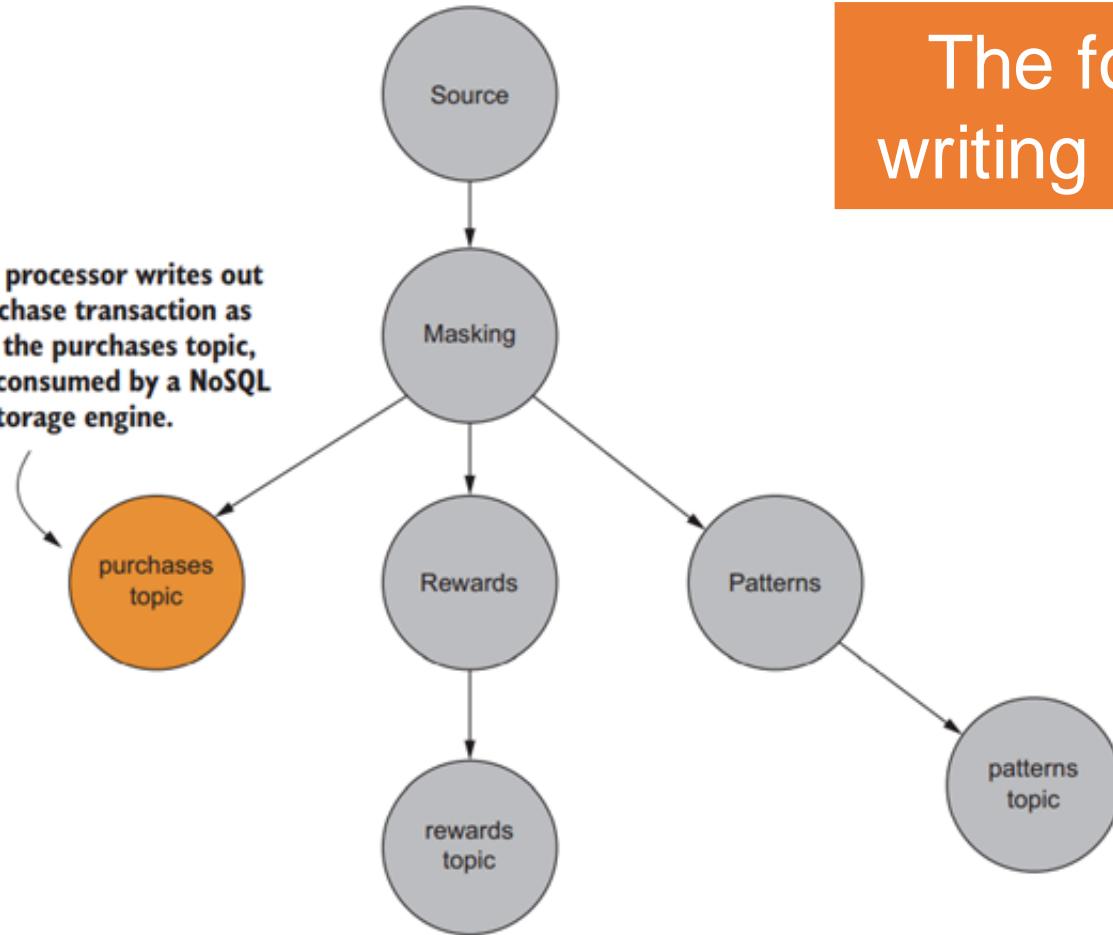


Figure The customer rewards processor is responsible for transforming Purchase objects into a RewardAccumulator object containing the customer ID, date, and dollar amount of the transaction. A child processor writes the Rewards objects to another Kafka topic.

The fourth processor—writing purchase records

This last processor writes out the purchase transaction as JSON to the purchases topic, which is consumed by a NoSQL storage engine.

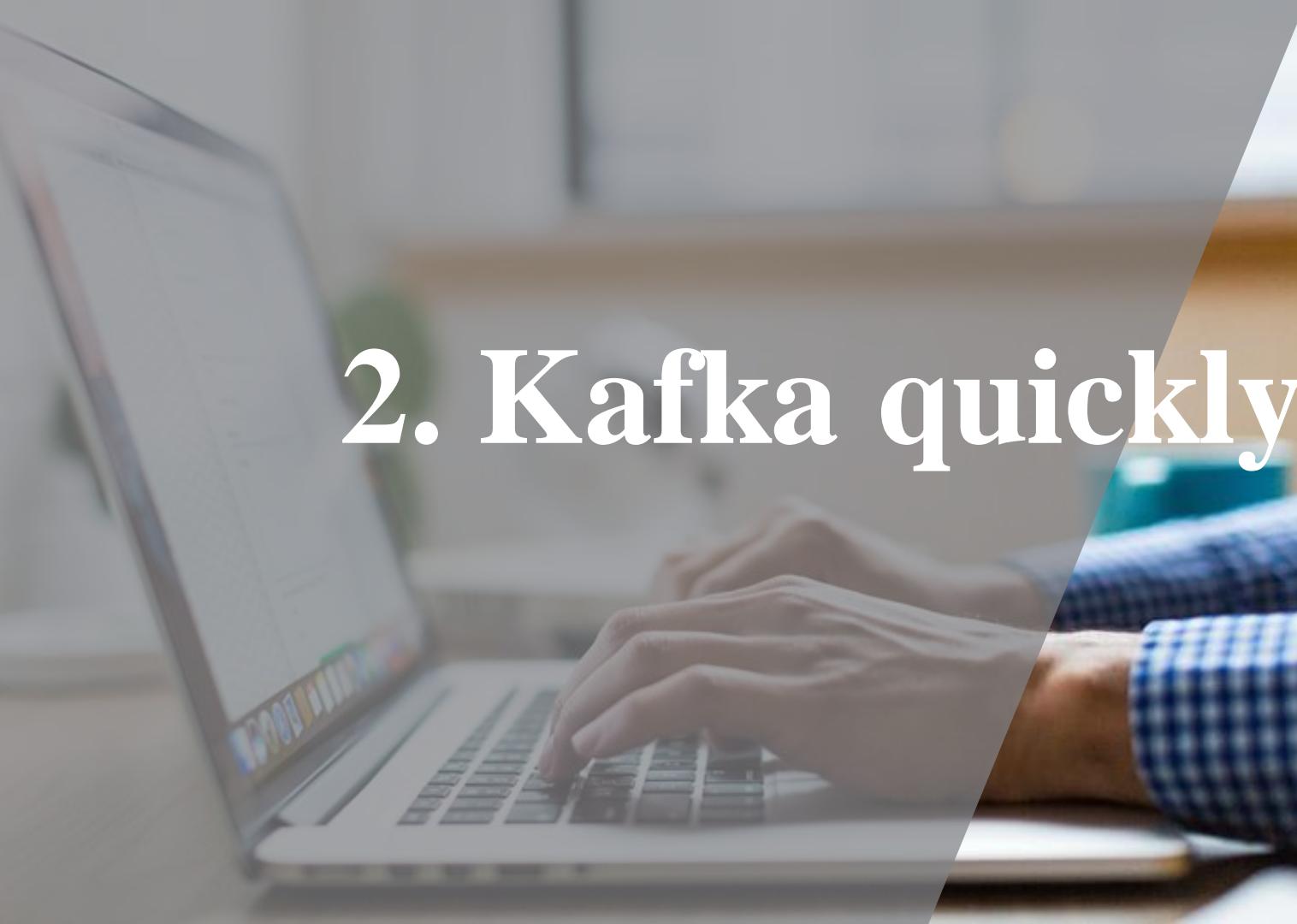


Summary

- Kafka Streams is a graph of processing nodes that combine to provide powerful and complex stream processing.
- Batch processing is powerful, but it's not enough to satisfy real-time needs for working with data.
- Distributing data, key/value pairs, partitioning, and data replication are critical for distributed applications

COMPLETE LAB 1

2. Kafka quickly



Kafka quickly

This lesson covers

- Examining the Kafka architecture
- Sending messages with producers
- Reading messages with consumers
- Installing and running Kafka

The data problem

- Organizations today are swimming in data. Internet companies, financial businesses, and large retailers are better positioned now than ever to use this data
- Both to serve their customers better and to find more efficient ways of conducting business. (We're going to take a positive outlook on this situation and assume only good intentions when looking at customer data.)

Using Kafka to handle data

- In lesson 1, you were introduced to the large retail company ZMart.
- At that point, ZMart wanted a streaming platform to use the company's sales data in order to offer better customer service and improve sales overall.
- But six months before that, Zmart was looking to get a handle on its data situation.

ZMart's original data platform

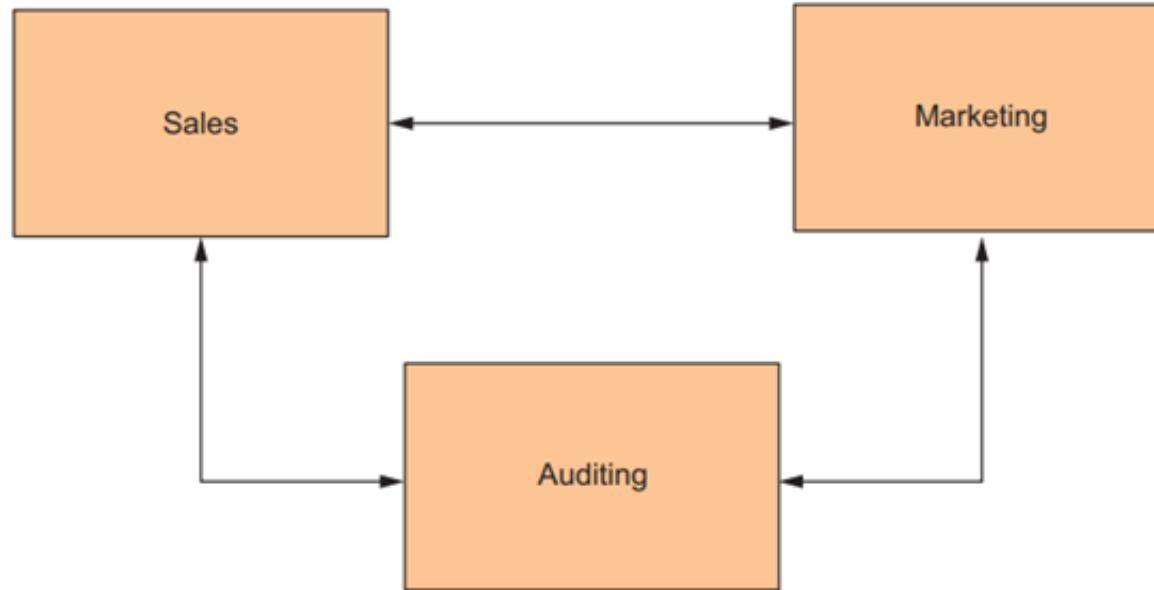
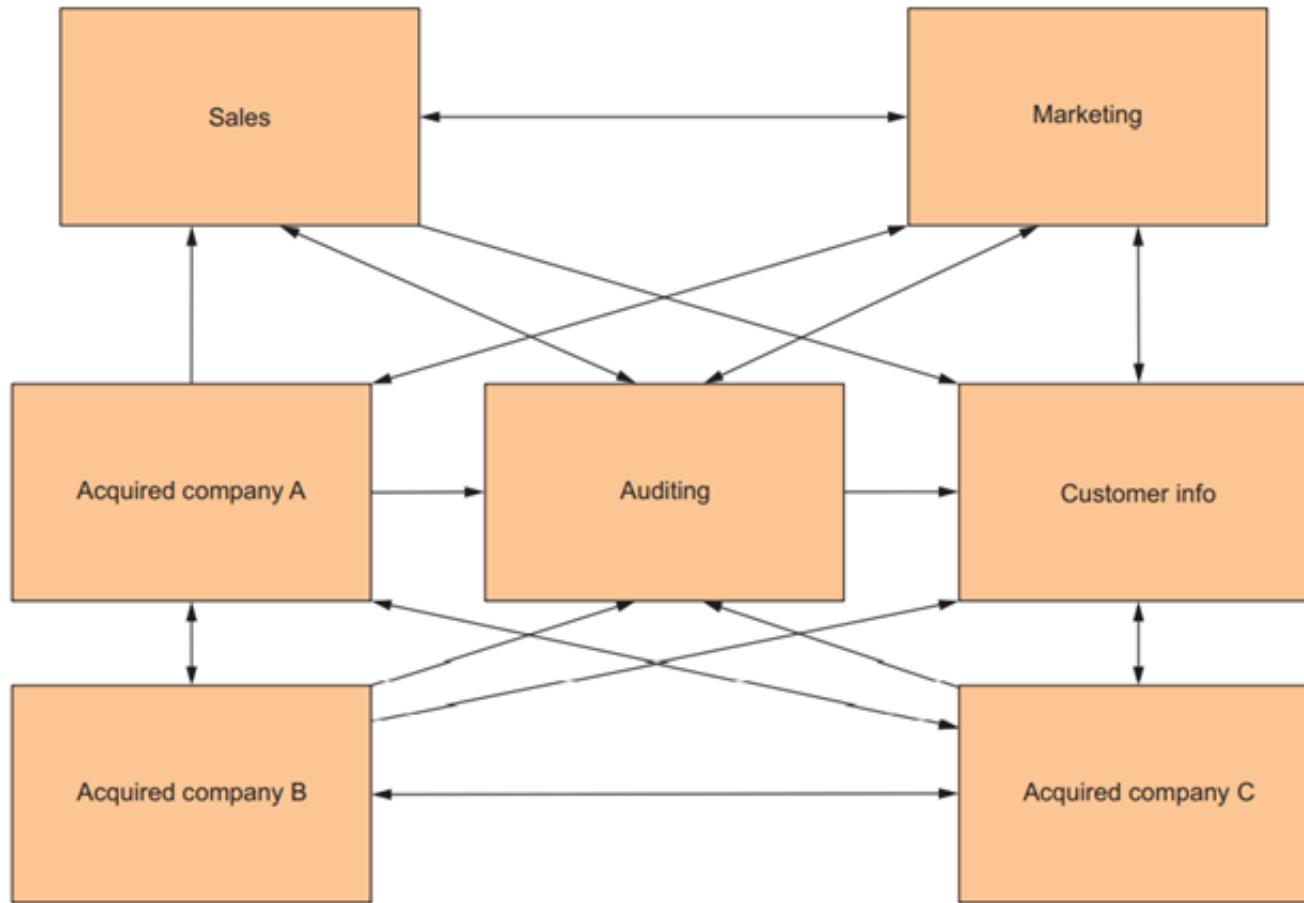
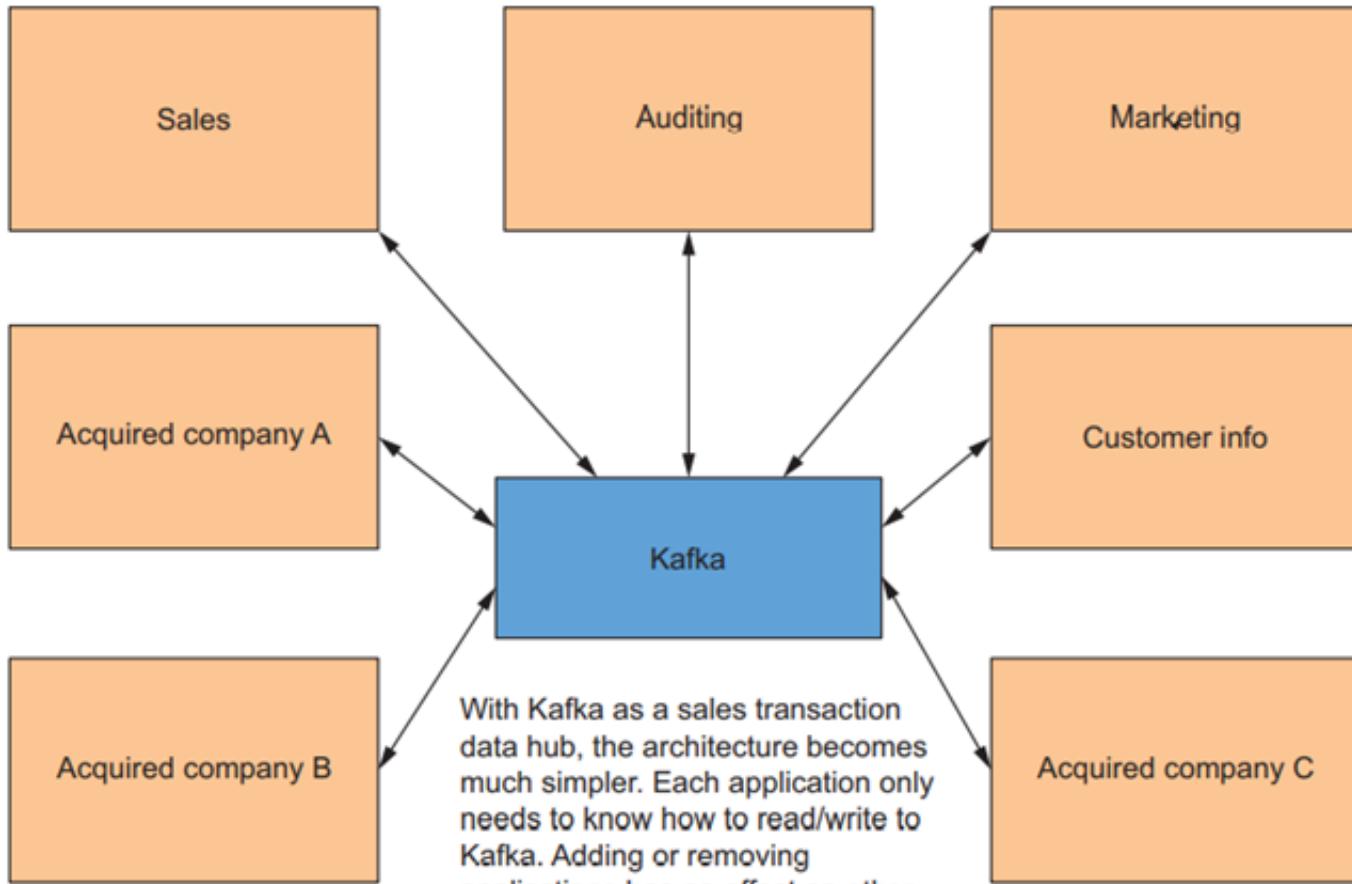


Figure The original data architecture for ZMart was simple enough to have information flowing to and from each source of information.

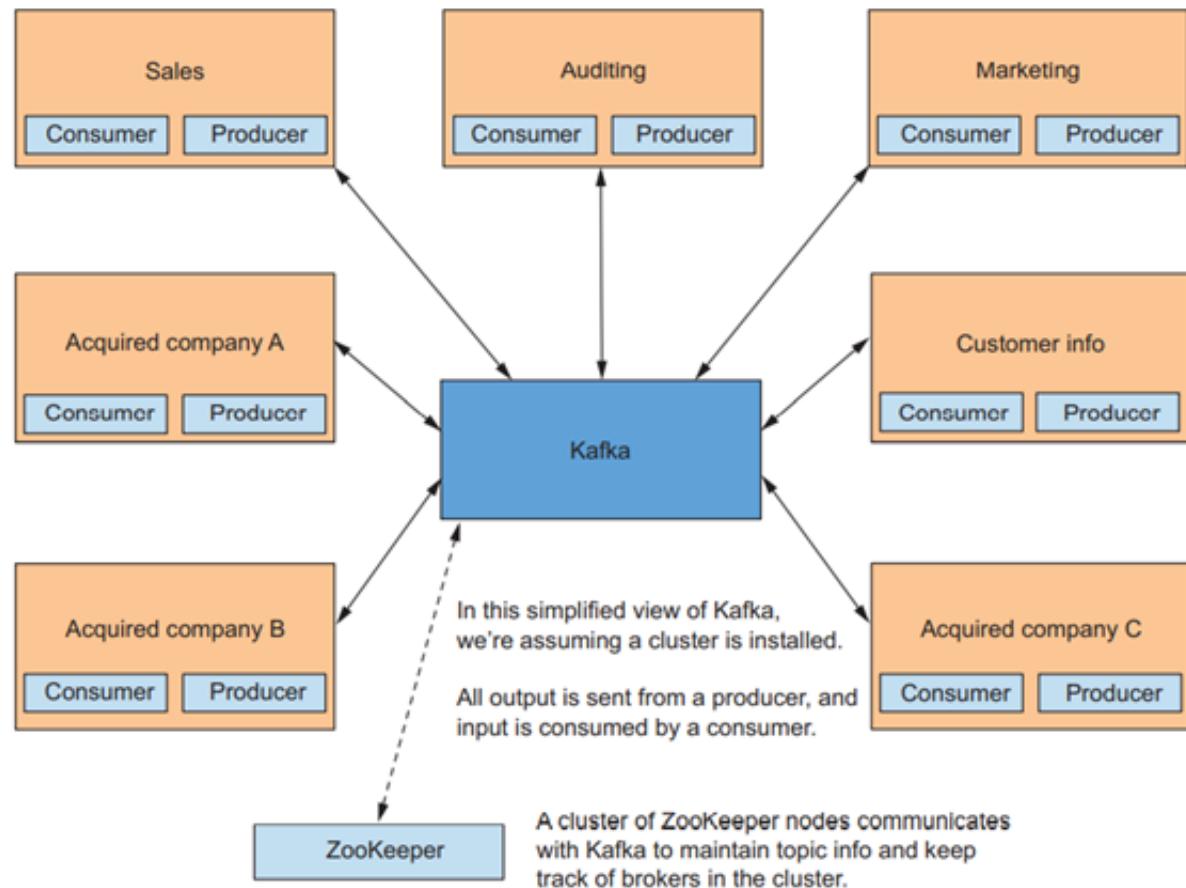


A Kafka sales transaction data hub

- A solution to ZMart's problem is to create one intake process to hold all transaction data—a transaction data hub.
- This transaction data hub should be stateless, accepting transaction data and storing it in such a fashion that any consuming application can pull the information it needs.



Kafka is a message broker



Kafka is a log

- The mechanism underlying Kafka is the log. Most software engineers are familiar with logs that track what an application's doing.
- If you're having performance issues or errors in your application, the first place to check is the application logs, But that's a different sort of log.

Kafka is a log

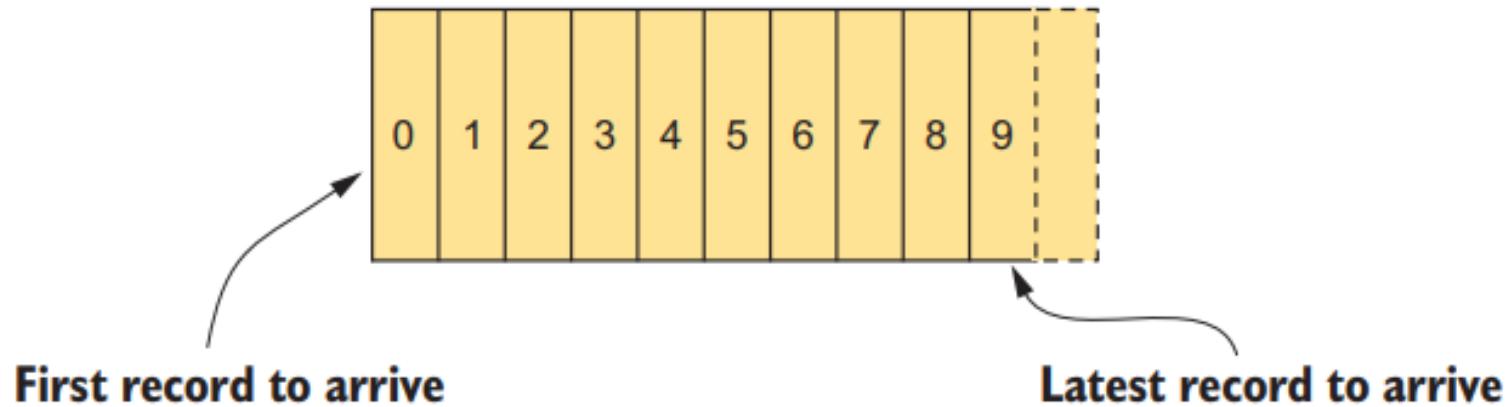


Figure A log is a file where incoming records are appended—each newly arrived record is placed immediately after the last record received. This process orders the records in the file by time.

How logs work in Kafka

The logs directory is configured in the root at /logs.

/logs

/logs/topicA_0 topicA has one partition.

/logs/topicB_0 topicB has three partitions.

/logs/topicB_1

/logs/topicB_2

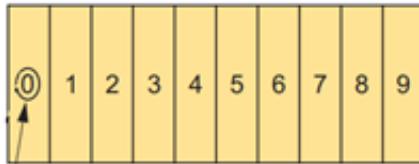
Figure The logs directory is the base storage for messages. Each directory under /logs represents a topic partition. Filenames within the directory start with the name of the topic, followed by an underscore, which is followed by a partition number.

Kafka and partitions

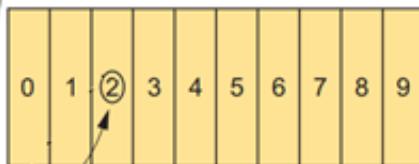
Topic with partitions

Data comes into a single topic but is placed into individual partitions either (0, 1, or 2). Because there are no keys with these messages, partitions are assigned in round-robin fashion.

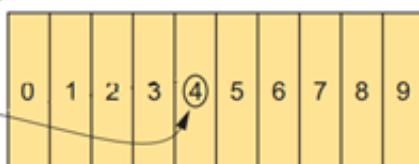
Partition 0



Partition 1



Partition 2



The numbers shown in the rectangles are the offsets for the messages.

As messages or records come in, they are written to a partition (assigned by producer) and appended in time order to the end of the log.

Each partition is in strictly increasing order, but there's no order across partitions.

Partitions group data by key

Incoming messages:

```
{foo, message data}  
{bar, message data}
```

Message keys are used to determine which partition the message should go to. These keys are not null.

The bytes of the key are used to calculate the hash.

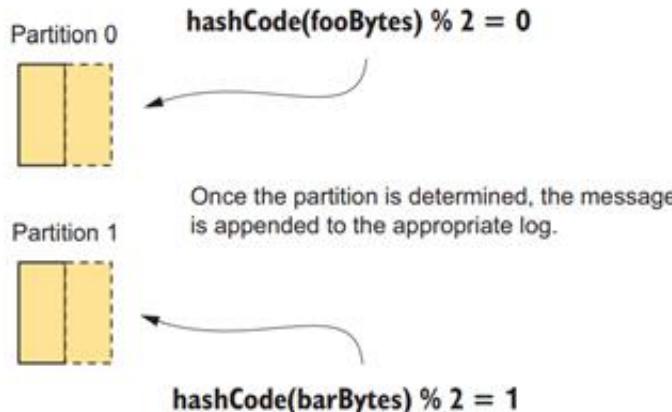


Figure “foo” is sent to partition 0, and “bar” is sent to partition 1. You obtain the partition by hashing the bytes of the key, modulus the number of partitions.

Partitions group data by key

- If the keys aren't null, Kafka uses the following formula (shown in pseudocode) to determine which partition to send the key/value pair to:

HashCode.(key) % number of partitions

Writing a custom partitioner

- Why would you want to write a custom partitioner? Of the several possible reasons, we'll look at one simple case here—the use of composite keys.
- Suppose you have purchase data flowing into Kafka, and the keys contain two values: a customer ID and a transaction date.

Listing 2.1 PurchaseKey composite key

```
public class PurchaseKey {  
  
    private String customerId;  
    private Date transactionDate;  
  
    public PurchaseKey(String customerId, Date transactionDate) {  
        this.customerId = customerId;  
        this.transactionDate = transactionDate;  
    }  
  
    public String getCustomerId() {  
        return customerId;  
    }  
  
    public Date getTransactionDate() {  
        return transactionDate;  
    }  
}
```

Listing 2.2 PurchaseKeyPartitioner custom partitioner

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {  
  
    @Override  
    public int partition(String topic, Object key,  
                         byte[] keyBytes, Object value,  
                         byte[] valueBytes, Cluster cluster) {  
        Object newKey = null;  
        if (key != null) {  
            PurchaseKey purchaseKey = (PurchaseKey) key;  
            newKey = purchaseKey.getCustomerId();  
            keyBytes = ((String) newKey).getBytes();  
        }  
        return super.partition(topic, newKey, keyBytes, value,  
                             valueBytes, cluster);  
    }  
}
```

If the key isn't null, extracts the customer ID

Sets the key bytes to the new value

Returns the partition with the updated key, delegating to the superclass

Specifying a custom partitioner

- Now that you've written a custom partitioner, you need to tell Kafka you want to use it instead of the default partitioner.
- Although we haven't covered producers yet, you specify a different partitioner when configuring the Kafka producer:

`partitioner.class=bbejeck_2.partition.PurchaseKeyPartitioner`

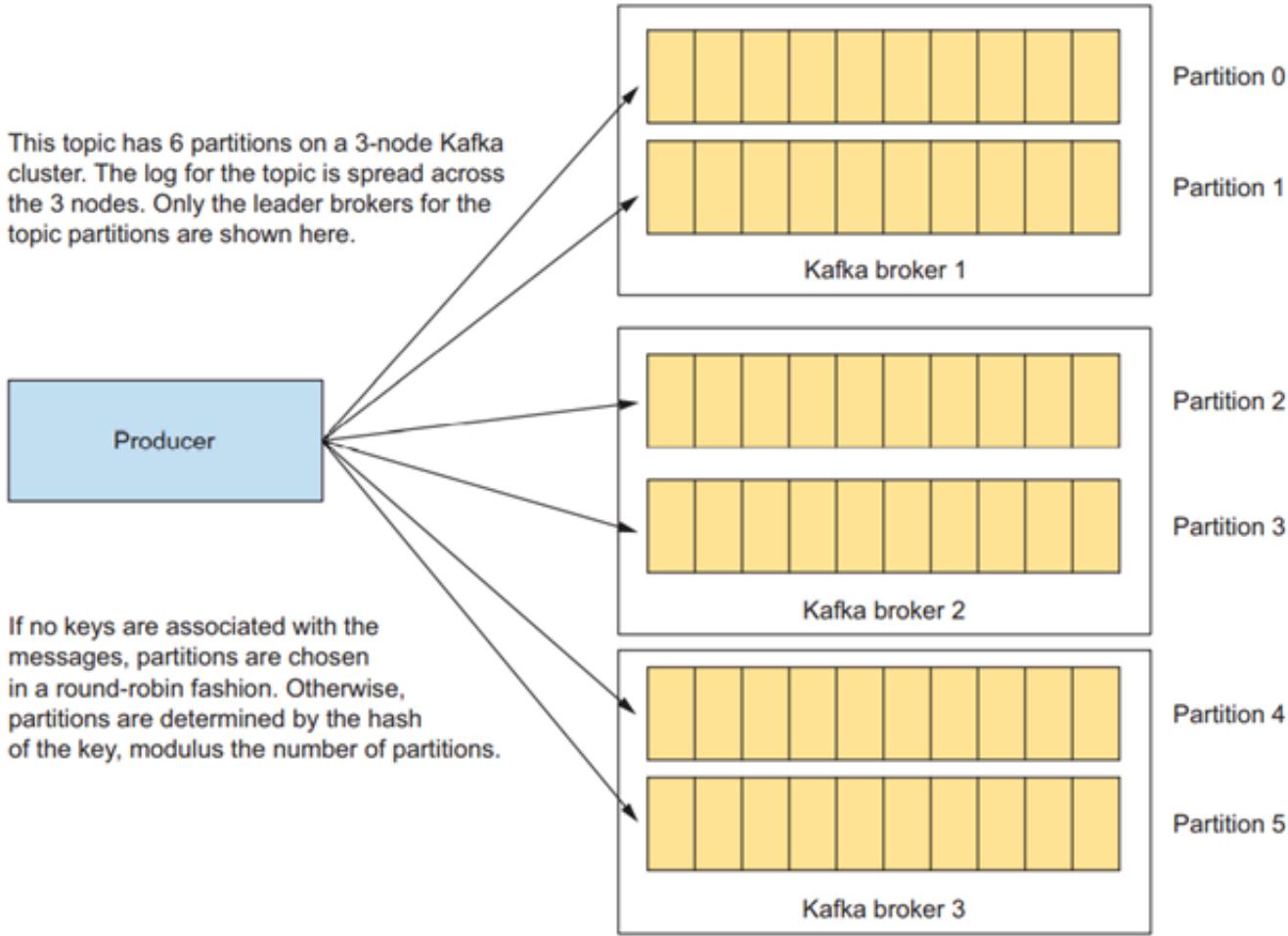
Determining the correct number of partitions

- Choosing the number of partitions to use when creating a topic is part art and part science.
- One of the key considerations is the amount of data flowing into a given topic.
- More data implies more partitions for higher throughput.

The distributed log

- We've discussed the concepts of logs and partitioned topics.
- Let's take a minute to look at those two concepts together to demonstrate distributed logs.
- So far, we've focused on logs and topics on one Kafka server or broker, but typically a Kafka production cluster environment includes several machines.

This topic has 6 partitions on a 3-node Kafka cluster. The log for the topic is spread across the 3 nodes. Only the leader brokers for the topic partitions are shown here.



ZooKeeper: leaders, followers, and replication

- Now it's time to look at how Kafka provides data availability in the face of machine failures.
- Kafka has the notion of leader and follower brokers.
- In Kafka, for each topic partition, one broker is chosen as the leader for the other brokers (the followers).

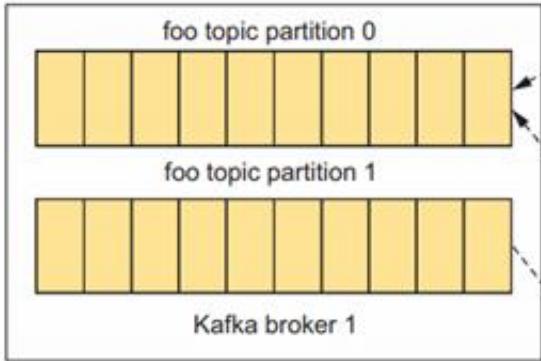
Apache ZooKeeper

- If you're a complete Kafka newbie, you may be asking yourself, "Why are we talking about Apache ZooKeeper in a Kafka course?" Apache ZooKeeper is integral to Kafka's architecture
- It's ZooKeeper that enables Kafka to have leader brokers and to do such things as track the replication of topics (<https://zookeeper.apache.org>)

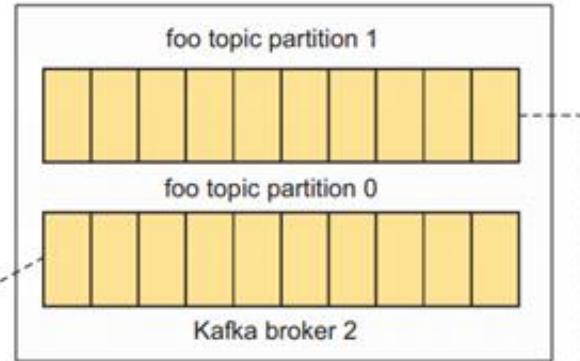
Electing a controller

- Kafka uses ZooKeeper to elect the controller broker.
- Discussing the consensus algorithms involved is way beyond the scope of this course, so we'll take the 50,000-foot view and just state that ZooKeeper elects a broker from the cluster to be the controller.
- If the controlling broker fails or becomes unavailable for any reason, ZooKeeper elects a new controller from a set of brokers that are considered to be caught up with the leader (an in-sync replica [ISR]).

The topic foo has 2 partitions and a replication level of 3. Dashed lines between partitions point to the leader of the given partition. Producers write records to the leader of a partition, and the followers read from the leader.

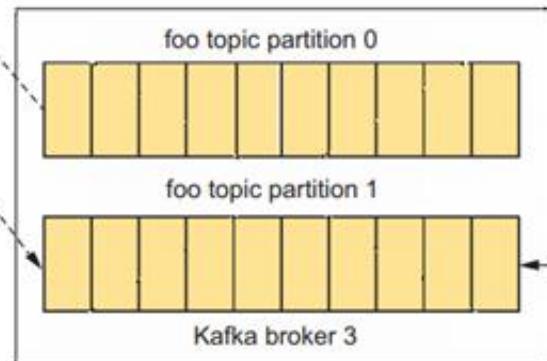


Broker 1 is the leader for partition 0 and is a follower for partition 1 on broker 3.



Broker 2 is a follower for partition 0 on broker 1 and a follower for partition 1 on broker 3.

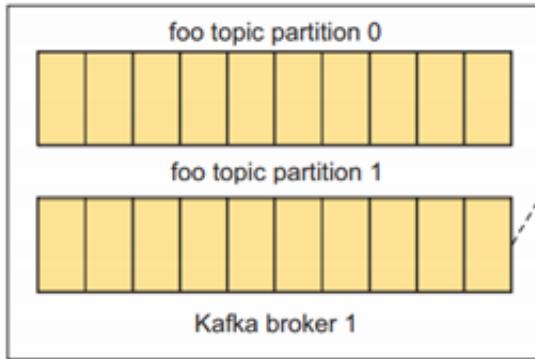
Broker 3 is a follower for partition 0 on broker 1 and the leader for partition 1.



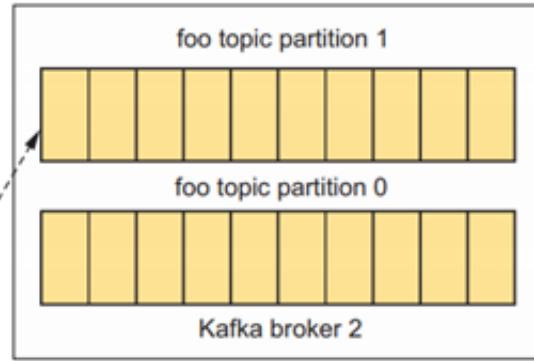
The topic foo has 2 partitions and a replication level of 3. These are the initial leaders and followers:

Broker 1 leader partition 0, follower partition 1
Broker 2 follower partition 0, follower partition 1
Broker 3 follower partition 0, leader partition 1

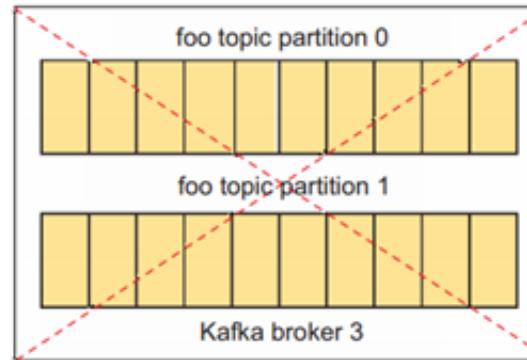
Broker 3 has become unresponsive.



Step 1: As the leader, broker 1 has detected that broker 3 has failed.



Step 2: The controller has reassigned the leadership of partition 1 from broker 3 to broker 2. All records for partition 1 will go to broker 2, and broker 1 will now consume messages for partition 1 from broker 2.



Controller responsibilities

ZooKeeper is also involved in the following aspects of Kafka operations:

- Cluster membership—Joining a cluster and maintaining membership in a cluster. If a broker becomes unavailable, ZooKeeper removes the broker from cluster membership.
- Topic configuration—Keeping track of the topics in a cluster, which broker is the leader for a topic, how many partitions there are for a topic, and any specific configuration overrides for a topic.
- Access control—Identifying who can read from and write to particular topics.

Deleting logs

Log rolling is a configuration setting you can specify when setting up a Kafka broker. There are two options for log rolling:

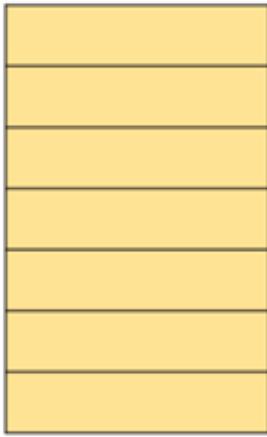
- `log.roll.ms`—This is the primary configuration, but there's no default value.
- `log.roll.hours`—This is the secondary configuration, which is only used if `log.role.ms` isn't set. It defaults to 168 hours

Deleting logs

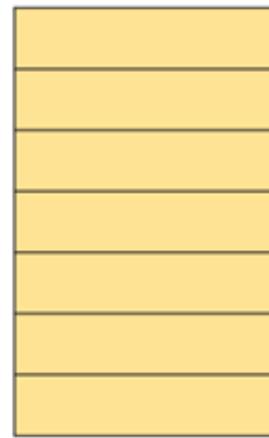
There are three settings, listed in order of priority, meaning that configurations earlier in the list override the later entries:

- log.retention.ms—How long to keep a log file in milliseconds
- log.retention.minutes—How long to keep a log file in minutes
- log.retention.hours—Log file retention in hours

The records are appended to this current log.



Older log segment files that have been rolled. The bottom segment is still in use.



This segment log has been deleted.

Figure . On the left are the current log segments. On the upper right is a deleted log segment, and the one below it is a recently rolled segment still in use.

Compacting logs

- Consider the case where you have keyed data, and you're receiving updates for that data over time, meaning a new record with the same key will update the previous value.
- For example, a stock ticker symbol could be the key, and the price per share would be the regularly updated value.

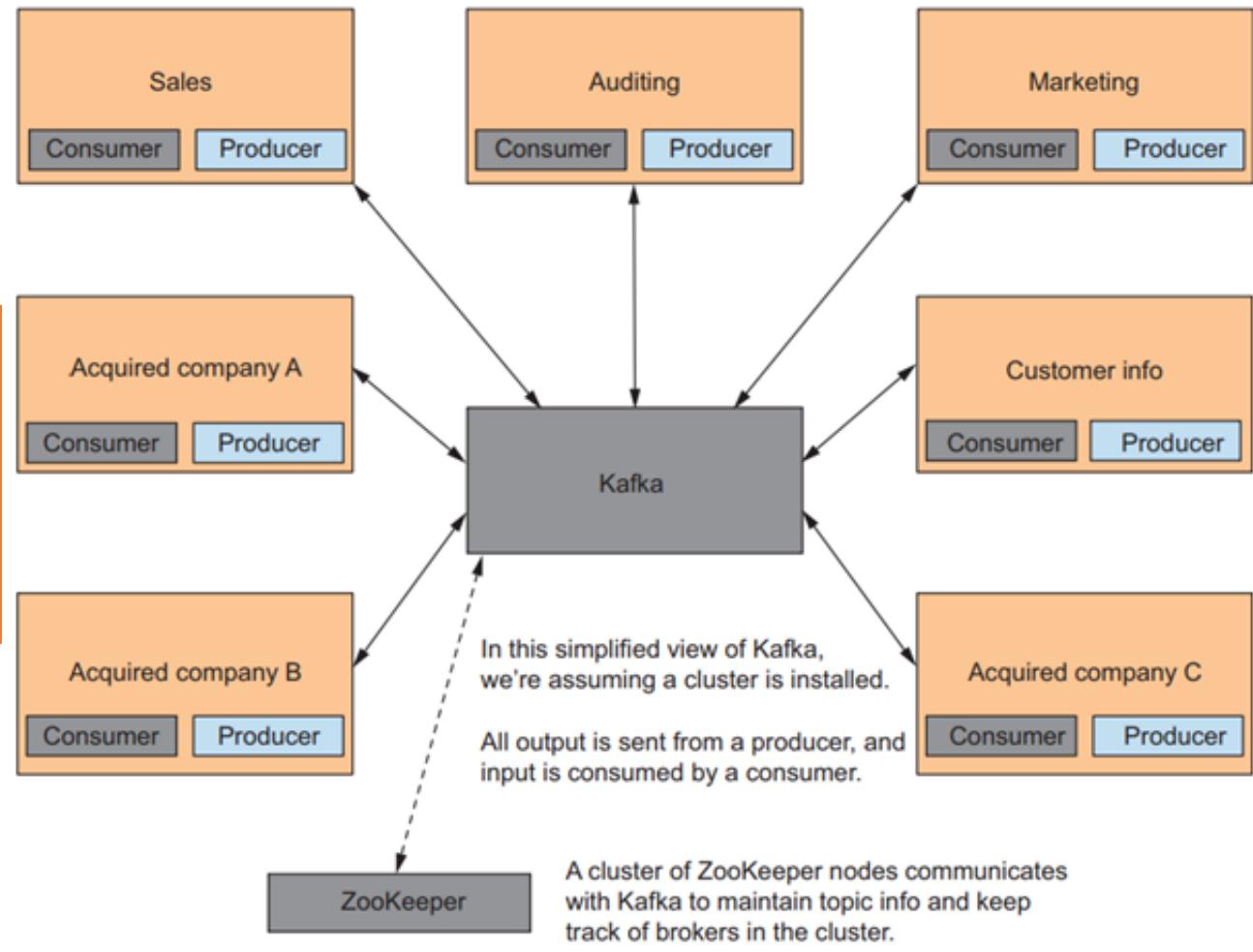
Before compaction

Offset	Key	Value
10	foo	A
11	bar	B
12	baz	C
13	foo	D
14	baz	E
15	boo	F
16	foo	G
17	baz	H

After compaction

Offset	Key	Value
11	bar	B
15	boo	F
16	foo	G
17	baz	H

Sending messages with producers



Listing 2.3 SimpleProducer example

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("value.serializer",
  => "org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
properties.put("partitioner.class",
  => PurchaseKeyPartitioner.class.getName());           ← Properties for
                                                       configuring a
                                                       producer

PurchaseKey key = new PurchaseKey("12334568", new Date());

try(Producer<PurchaseKey, String> producer =
  => new KafkaProducer<>(properties)) {
  ProducerRecord<PurchaseKey, String> record =
  => new ProducerRecord<>("transactions", key, "{\"item\":\"book\",
    \"price\":10.99}");

  Callback callback = (metadata, exception) -> {
    if (exception != null) {
      System.out.println("Encountered exception "
        => + exception);
    }
  };
}

Future<RecordMetadata> sendFuture =
  => producer.send(record, callback);           ← Creates the
                                               KafkaProducer
                                                       Builds a
                                                       callback
}                                              ← Sends the record and sets the
                                               returned Future to a variable
```

Instantiates
the Producer-
Record

Creates the
KafkaProducer

Builds a
callback

Sends the record and sets the
returned Future to a variable

Producer properties

- When you created the KafkaProducer instance, you passed a java.util.Properties parameter containing the configuration for the producer.
- The configuration of a KafkaProducer isn't complicated, but there are key properties to consider when setting it up

Specifying partitions and timestamps

- When you instantiated the ProducerRecord in listing 2.3, you used one of four overloaded constructors.
- Other constructors allow for setting a partition and timestamp, or just a partition

```
ProducerRecord(String topic, Integer partition, String key, String value)
ProducerRecord(String topic, Integer partition,
              Long timestamp, String key,
              String value)
```

Specifying a partition

- the distribution of the keys might not be even, and you want to ensure that all partitions receive roughly the same amount of data.
- Here's a rough implementation that would do this

Listing 2.4 Manually setting the partition

```
→ AtomicInteger partitionIndex = new AtomicInteger(0);  
  
int currentPartition = Math.abs(partitionIndex.getAndIncrement()) %  
→ numberPartitions;  
ProducerRecord<String, String> record =  
→ new ProducerRecord<>("topic", currentPartition, "key", "value");
```

Creates an AtomicInteger instance variable

Gets the current partition and uses it as a parameter

Timestamps in Kafka

- Kafka version 0.10 added timestamps to records.
- You set the timestamp when you create a ProducerRecord via this overloaded constructor call:

```
ProducerRecord(String topic, Integer partition,  
               Long timestamp, K key, V value)
```

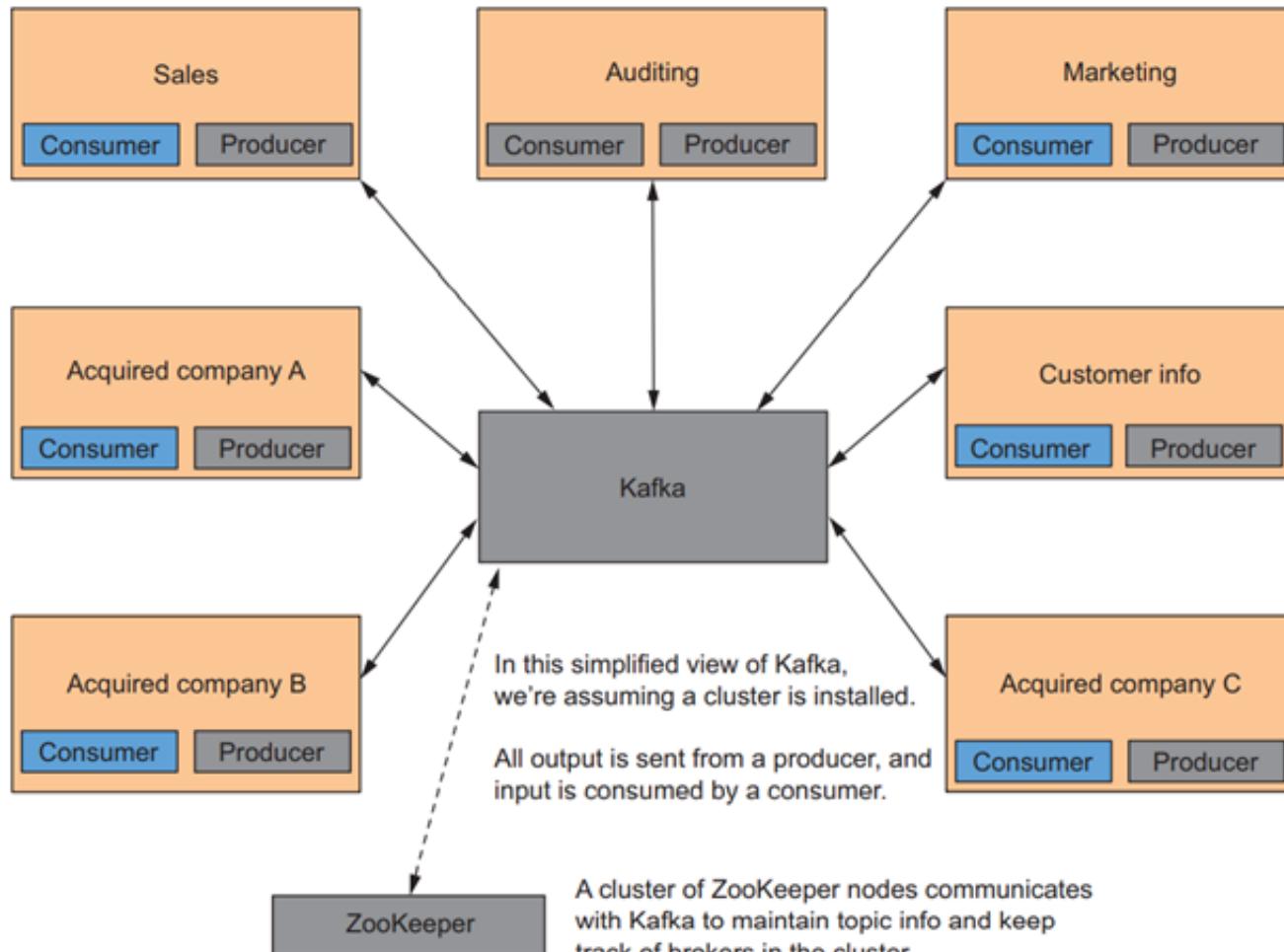
Reading messages with consumers

- You've seen how producers work; now it's time to look at consumers in Kafka.
- Suppose you're building a prototype application to show the latest ZMart sales statistics.
- For this example, you'll consume the message you sent in the previous producer example.

Managing offsets

Committing an offset has two implications for a consumer:

- Committing implies the consumer has fully processed the message.
- Committing also represents the starting point for that consumer in the case of failure or a restart

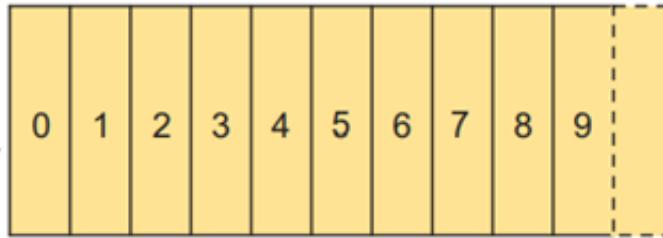


Managing offsets

- `auto.offset.reset="earliest"`—Messages will be retrieved starting at the earliest available offset. Any messages that haven't yet been removed by the logmanagement process will be retrieved.
- `auto.offset.reset="latest"`—Messages will be retrieved from the latest offset, essentially only consuming messages from the point of joining the cluster.
- `auto.offset.reset="none"`—No reset strategy is specified. The broker throws an exception to the consumer

Managing offsets

Ten messages have been sent to a topic.
A new consumer starts up, so it doesn't
have a last offset committed.



A config setting of “earliest”
means messages starting
from offset 0 will be sent to
the consumer.

A config setting of “latest”
means the consumer will
get the next message when
it is appended to the log.

Automatic offset commits

- Automatic offset commits are enabled by default, and they're represented by the `enable.auto.commit` property.
- There's a companion configuration option, `auto.commit.interval.ms`, which specifies how often the consumer will commit offsets (the default value is 5 seconds).

Manual offset commits

- There are two types of manually committed offsets—
synchronous and asynchronous.
- These are the synchronous commits:

```
consumer.commitSync()
```

```
consumer.commitSync(Map<TopicPartition, OffsetAndMetadata>)
```

Creating the consumer

- Creating a consumer is similar to creating a producer. You supply a configuration in the form of a Java `java.util.Properties` object, and you get back a `KafkaConsumer` instance.
- This instance then subscribes to topics from a supplied list of topic names or by specifying a regular expression.

Consumers and partitions

- You'll generally want multiple consumer instances—one for each partition of a topic.
- It's possible to have one consumer read from multiple partitions, but it's not uncommon to have a thread pool with as many threads as there are partitions, and with each thread running a consumer that's assigned to one partition.

Rebalancing

- The process of adding and removing topic-partition assignments to consumers described in the previous section is called rebalancing.
- Topic-partition assignments to a consumer aren't static—they're dynamic

Finer-grained consumer assignment

- KafkaConsumer has methods that allow you to subscribe to a particular topic and partition:

```
TopicPartition fooTopicPartition_0 = new TopicPartition("foo", 0);  
TopicPartition barTopicPartition_0 = new TopicPartition("bar", 0);
```

Finer-grained consumer assignment

- Failures won't result in topic partitions being reassigned, even for consumers with the same group ID. Any changes in assignments will require another call to `consumer.assign`.
- The group specified by the consumer is used for committing, but because each consumer will be acting independently, it's a good idea to give each consumer a unique group ID.

Consumer example

Listing 2.5 ThreadedConsumerExample example

```
public void startConsuming() {  
    executorService = Executors.newFixedThreadPool(numberPartitions);  
    Properties properties = getConsumerProps();  
  
    for (int i = 0; i < numberPartitions; i++) {  
        Runnable consumerThread = getConsumerThread(properties);      ←  
        executorService.submit(consumerThread);  
    }  
}  
  
private Runnable getConsumerThread(Properties properties) {  
    return () -> {  
        // Consumer logic  
    };  
}
```

Builds a consumer thread

```
Consumer<String, String> consumer = null;
try {
    consumer = new KafkaConsumer<>(properties);
    consumer.subscribe(Collections.singletonList(
    ↪ "test-topic"));
    while (!doneConsuming) {
        ConsumerRecords<String, String> records =
    ↪ consumer.poll(5000);
        for (ConsumerRecord<String, String> record : records) {
            String message = String.format("Consumed: key =
    ↪ %s value = %s with offset = %d partition = %d",
                record.key(), record.value(),
                record.offset(), record.partition());
            System.out.println(message);
        }
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (consumer != null) {
        consumer.close(); ←
    }
}
};
```

Subscribes to the topic

Polls for 5 seconds

Prints a formatted message

Closes the consumer—will leak resources otherwise

Installing and running Kafka

- As I write this, Kafka 1.0.0 is the most recent version. Because Kafka is a Scala project, each release comes in two versions: one for Scala 2.11 and another for Scala 2.12.
- I use the 2.12 Scala version of Kafka in this course.
- Although you can download the release, the course's source code includes a binary distribution of Kafka that will work with Kafka Streams as demonstrated and described in this course.

Kafka local configuration

- Running Kafka locally on your machine requires minimal configuration if you accept the default values.
- By default, Kafka uses port 9092, and ZooKeeper uses port 2181.
- Assuming you have no applications already using those ports, you're all set.
- Kafka writes its logs to /tmp/kafka-logs, and ZooKeeper uses /tmp/zookeeper for its log storage.

Running Kafka

- Kafka is simple to get started.
- Because ZooKeeper is essential for the Kafka cluster to function properly (ZooKeeper determines the leader broker, holds topic information, performs health checks on cluster members, and so on), you'll need to start Zookeeper before starting Kafk

STARTING ZOOKEEPER

- To start ZooKeeper, open a command prompt and enter the following command:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

STARTING KAFKA

- To start Kafka, open another command prompt and type this command:

```
bin/Kafka-server-start.sh config/server.properties
```


MacBook-Pro:~ user

java -jar /tmp/zookeeper-3.4.6.jar

[2016-08-13 10:32:54,235] INFO Client environment:java.library.path=/Users/bbejeck/Library/Java/Extensions:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java/. (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:java.io.tmpdir=/var/folders/t/k/d/9..._mr55sdqhyberty4rls880gn/ (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:os.name=Mac OS X (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:os.name=Mac OS X (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:os.arch=x86_64 (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:os.version=10.10.5 (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:user.name=bbejeck (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Client environment:user.dir=/usr/local/kafka_2.11-0.10.1.0-SNAPSHOT (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,235] INFO Initiating client connection, connectingLocalHost:2181 sessionTimeout=6000 watcher=org.IBItec.zkclient.ZkClient@8358bf (org.apache.zookeeper.ZooKeeper)

[2016-08-13 10:32:54,252] INFO Waiting for keeper state SyncConnected (org.IBItec.zkclient.ZkClient)

[2016-08-13 10:32:54,250] INFO Opening socket connection to server localhost/0:0:0:0:0:1:2181. Will not attempt to authenticate using SASL (unknown error) (org.apache.zookeeper.ClientCnxn)

[2016-08-13 10:32:54,318] INFO Socket connection established to localhost/0:0:0:0:0:1:2181, initiating session (org.apache.zookeeper.ClientCnxn)

[2016-08-13 10:32:54,386] INFO Session establishment complete on server localhost/0:0:0:0:0:1:2181, sessionid = 0x168427245000000, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)

[2016-08-13 10:32:54,387] INFO zookeeper state changed (SyncConnected) (org.IBItec.zkclient.ZkClient)

[2016-08-13 10:32:54,452] INFO Log directory '/tmp/kafka-logs' not found, creating it. (kafka.log.LogManager)

[2016-08-13 10:32:54,478] INFO Loading logs. (kafka.log.LogManager)

[2016-08-13 10:32:54,484] INFO Logs loading complete. (kafka.log.LogManager)

[2016-08-13 10:32:54,561] INFO Starting log cleanup with a period of 300000 ms. (kafka.log.LogManager)

[2016-08-13 10:32:54,565] INFO Starting log flusher with a default period of 92237/285885475387 ms. (kafka.log.LogManager)

[2016-08-13 10:32:54,567] WARN No meta.properties file under dir /tmp/kafka-logs/meta.properties (kafka.server.BrokerMetadataCheckpoint)

[2016-08-13 10:32:54,607] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)

[2016-08-13 10:32:54,609] INFO [Socket Server on Broker @], Started 1 acceptor threads (kafka.network.SocketServer)

[2016-08-13 10:32:54,628] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPungitory\$ExpireOperationReaper)

[2016-08-13 10:32:54,629] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPungitory\$ExpireOperationReaper)

[2016-08-13 10:32:54,669] INFO Creating /controller (is it secure? false) (kafka.utils.ZKCheckedphemeral)

[2016-08-13 10:32:54,694] INFO Result of node creation is: OK (kafka.utils.ZKCheckedphemeral)

[2016-08-13 10:32:54,695] INFO @ successfully elected as leader (kafka.server.ZookeeperLeaderElector)

[2016-08-13 10:32:54,748] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPungitory\$ExpireOperationReaper)

[2016-08-13 10:32:54,749] INFO [ExpirationReaper-0], Starting (kafka.server.DelayedOperationPungitory\$ExpireOperationReaper)

[2016-08-13 10:32:54,759] INFO [GroupCoordinator @]: Starting up. (kafka.coordinator.GroupCoordinator)

[2016-08-13 10:32:54,760] INFO [GroupCoordinator @]: Startup complete. (kafka.coordinator.GroupCoordinator)

[2016-08-13 10:32:54,763] INFO [GroupMetadataManager on Broker @]: Removed 0 expired offsets in 7 milliseconds. (kafka.coordinator.GroupMetadataManager)

[2016-08-13 10:32:54,777] INFO [ThrottledRequestReaper-Producer], Starting (kafka.server.ClientQuotaManager\$ThrottledRequestReaper)

[2016-08-13 10:32:54,778] INFO [ThrottledRequestReaper-Fetch], Starting (kafka.server.ClientQuotaManager\$ThrottledRequestReaper)

[2016-08-13 10:32:54,783] INFO Will not load MM4J, mm4j-tools.jar is not in the classpath (kafka.utils.MM4JLoader\$)

[2016-08-13 10:32:54,800] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedphemeral)

[2016-08-13 10:32:54,802] INFO Result of node creation is: OK (kafka.utils.ZKCheckedphemeral)

[2016-08-13 10:32:54,883] INFO Registered broker @ at path /brokers/ids/0 with addresses: PLAINTEXT => EndPoint(localhost,9092,PLAINTEXT) (kafka.utils.ZKUtils)

[2016-08-13 10:32:54,894] WARN No meta.properties file under dir /tmp/kafka-logs/meta.properties (kafka.server.BrokerMetadataCheckpoint)

[2016-08-13 10:32:54,895] INFO New leader is @ (kafka.server.ZookeeperLeaderElector\$LeaderChangeListener)

[2016-08-13 10:32:54,826] INFO Kafka version : 0.10.1.0-SNAPSHOT (org.apache.kafka.common.utils.AppInfoParser)

[2016-08-13 10:32:54,826] INFO Kafka commitId : 7ded9a29ec14bde (org.apache.kafka.common.utils.AppInfoParser)

[2016-08-13 10:32:54,827] INFO [Kafka Server @], started (kafka.server.KafkaServer)

Sending your first message

- Now that you have Kafka up and running, it's time to use Kafka for what it's meant to do: sending and receiving messages.
- But before you send a message, you'll need to define a topic for a producer to send a message to.

YOUR FIRST TOPIC

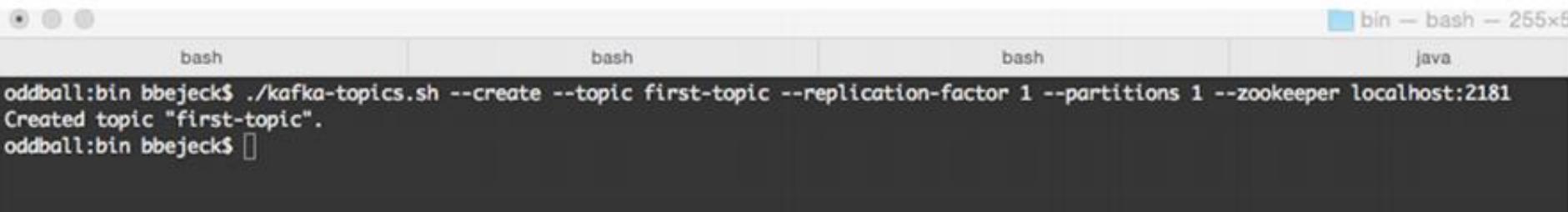
- Creating a topic in Kafka is simple.
- It's just a matter of running a script with some configuration parameters.
- The configuration is easy, but the settings you provide have broad performance implications.
- By default, Kafka is configured to autorecreate topics, meaning that if you attempt to send to or read from a nonexistent topic, the Kafka broker will create one for you (using default configurations in the server.properties file).

CREATING A TOPIC

- To create a topic, you need to run the `kafka-topics.sh` script.
- Open a terminal window and run this command:

```
bin/kafka-topics.sh --create --topic first-topic --replication-factor 1  
→ --partitions 1 --zookeeper localhost:2181
```

Sending your first message



A screenshot of a terminal window titled "bin - bash - 255x6". The window contains four tabs: "bash", "bash", "bash", and "java". The "bash" tab is active and shows the command: `./kafka-topics.sh --create --topic first-topic --replication-factor 1 --partitions 1 --zookeeper localhost:2181`. The output of the command is: "Created topic "first-topic". The "java" tab is visible but empty.

```
oddball:bin bbejeck$ ./kafka-topics.sh --create --topic first-topic --replication-factor 1 --partitions 1 --zookeeper localhost:2181
Created topic "first-topic".
oddball:bin bbejeck$
```

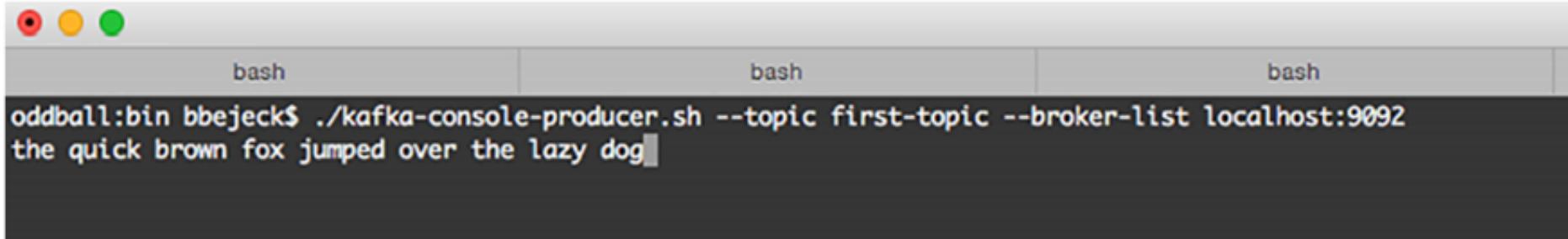
SENDING A MESSAGE

- Sending a message in Kafka generally involves writing a producer client, but Kafka also comes with a handy script called `kafka-console-producer` that allows you to send a message from a terminal window
- We'll use the console producer in this example, but we've covered how to use the `KafkaProducer`

SENDING A MESSAGE

To send your first message, run the following command

```
# command assumes running from bin directory  
.kafka-console-producer.sh --topic first-topic --broker-list localhost:9092
```



The screenshot shows a terminal window with three tabs, each labeled "bash". The active tab displays the command `./kafka-console-producer.sh --topic first-topic --broker-list localhost:9092` followed by the message "the quick brown fox jumped over the lazy dog". The terminal has a dark background and light-colored text.

READING A MESSAGE

- Kafka also provides a console consumer for reading messages from the command line.
- The console consumer is similar to the console producer: once started, it will keep reading messages from the topic until the script is stopped by you (with Ctrl-C).
- To launch the console consumer, run this command

```
bin/kafka-console-consumer.sh --topic first-topic  
    --bootstrap-server localhost:9092 --from-beginning
```

READING A MESSAGE

```
oddball:bin bbejeck$ ./kafka-console-consumer.sh --topic first-topic --bootstrap-server localhost:9092 --from-beginning
the quick brown fox jumped over the lazy dog
```

Summary

- Kafka uses partitions for achieving high throughput and to provide a means for grouping messages with the same keys in order.
- Producers are used for sending messages to Kafka.
- Null keys mean round-robin partition assignment; otherwise, the producer uses the hash of the key, modulus the number of partitions, for partition assignment.

COMPLETE LAB 2

Part 2: Kafka Streams development

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a development or coding environment.

3. Developing Kafka Streams



Developing Kafka Streams

This lesson covers

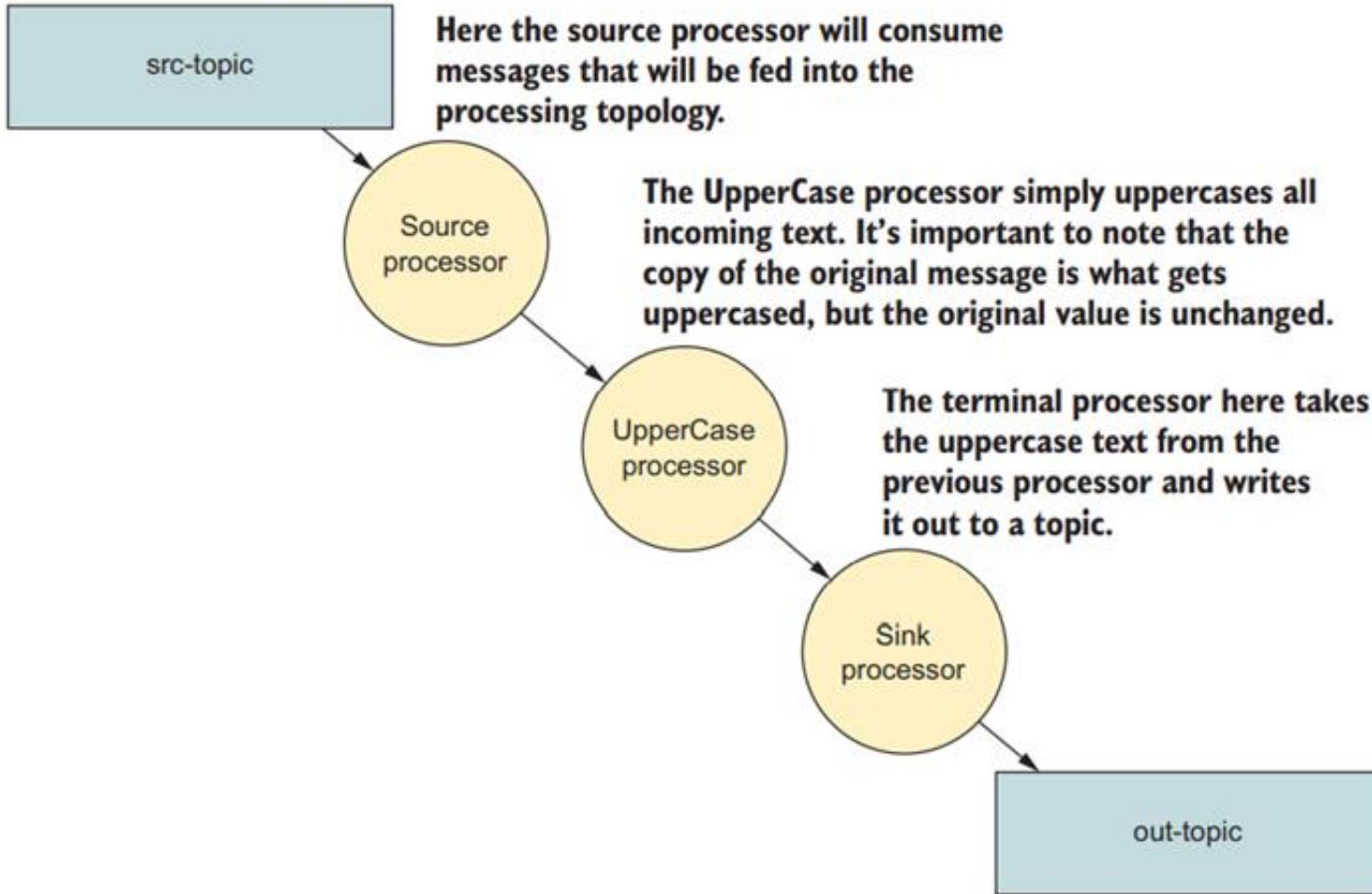
- Introducing the Kafka Streams API
- Building Hello World for Kafka Streams
- Exploring the ZMart Kafka Streams application in depth
- Splitting an incoming stream into multiple streams

The Streams Processor API

- The Kafka Streams DSL is the high-level API that enables you to build Kafka Streams applications quickly.
- The high-level API is very well thought out, and there are methods to handle most stream-processing needs out of the box, so you can create a sophisticated stream-processing program without much effort.

Hello World for Kafka Streams

- Your first program will be a toy application that takes incoming messages and converts them to uppercase characters, effectively yelling at anyone who reads the message. You'll call this the Yelling App.
- Before diving into the code, let's take a look at the processing topology you'll assemble for this application

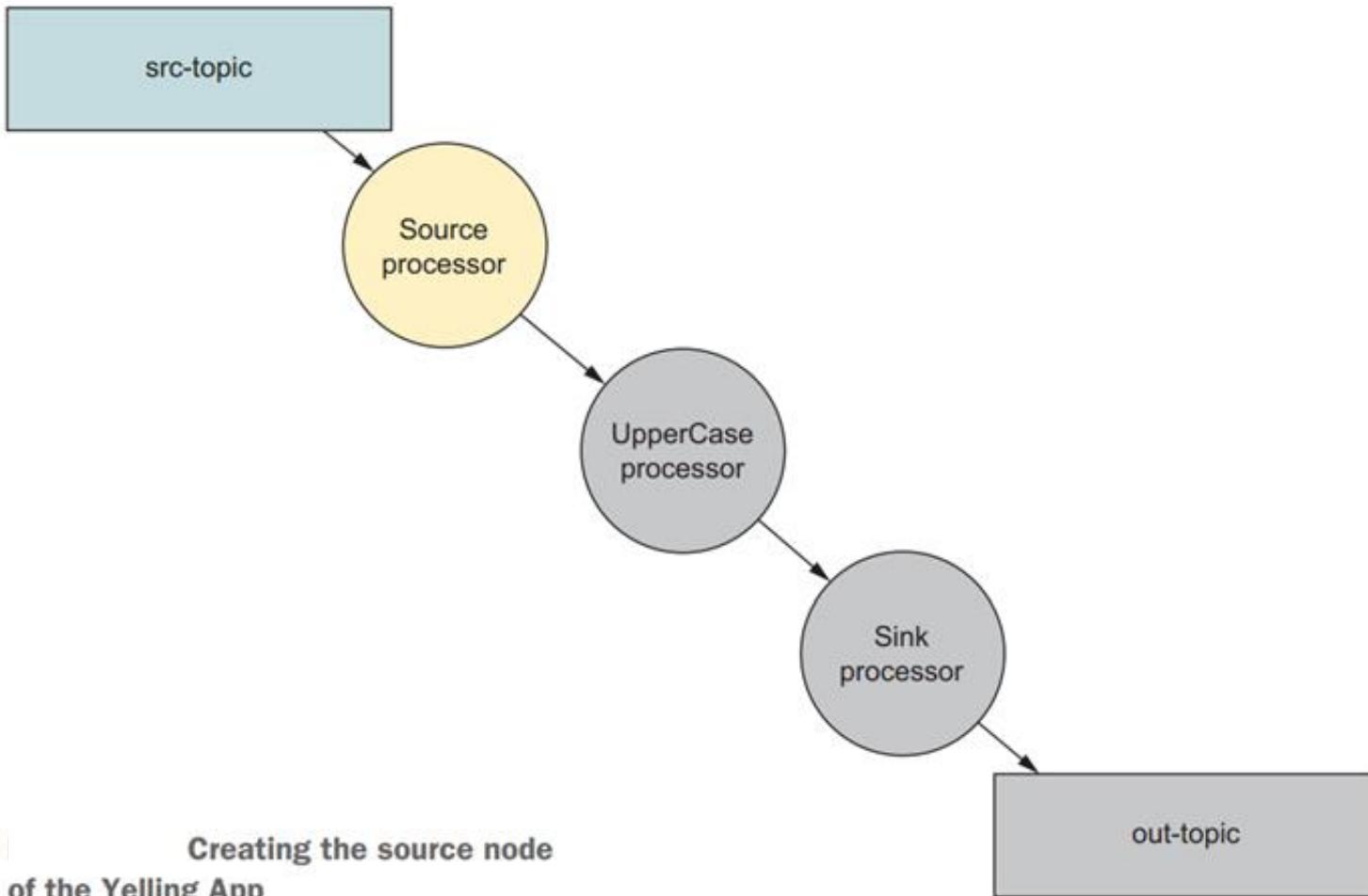


Creating the topology for the Yelling App

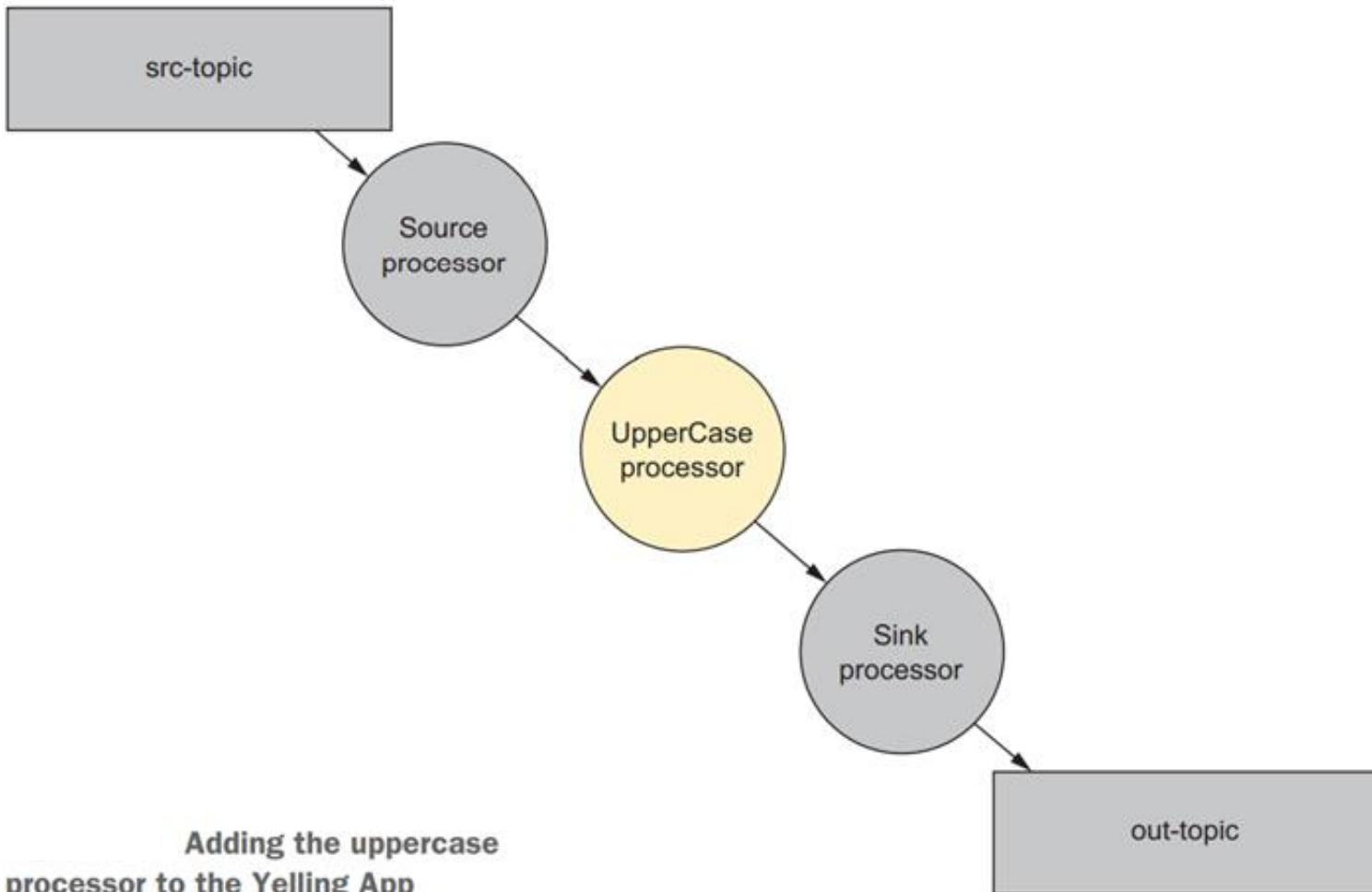
- The following line of code creates the source, or parent, node of the graph

Listing 3.1 Defining the source for the stream

```
KStream<String, String> simpleFirstStream = builder.stream("src-topic",  
    Consumed.with(stringSerde, stringSerde));
```



Creating the source node
of the Yelling App



Adding the uppercase
processor to the Yelling App

Creating the topology for the Yelling App

Listing 3.2 Mapping incoming text to uppercase

```
KStream<String, String> upperCasedStream =  
    simpleFirstStream.mapValues(String::toUpperCase);
```

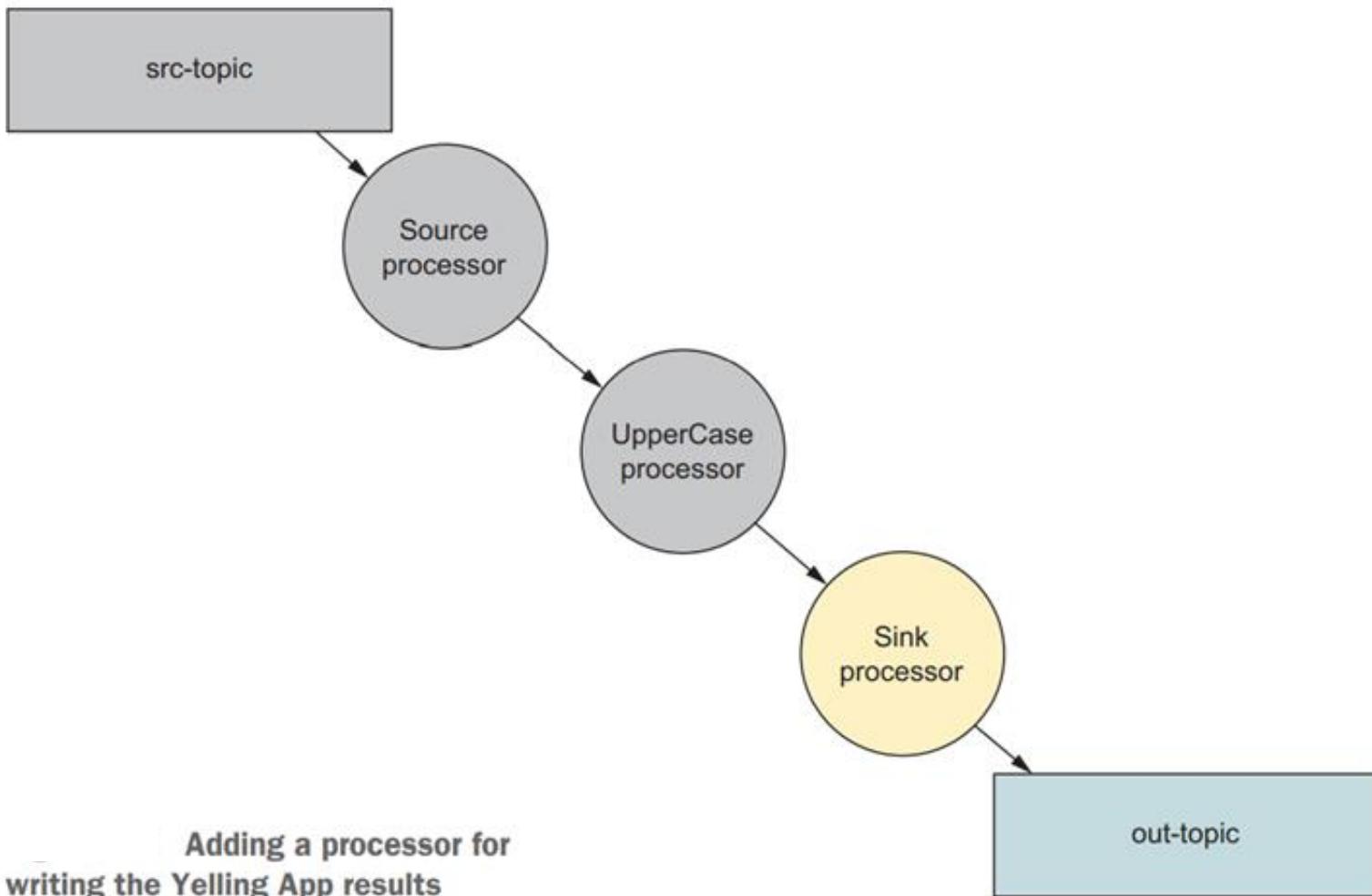
By calling the KStream.mapValues function, you're creating a new processing node whose inputs are the results of going through the mapValues call

Creating the topology for the Yelling App

- The final step is to add a sink processor that writes the results out to a topic.
- Next Figure shows where you are in the construction of the topology.
- The following code line adds the last processor in the graph.

Listing 3.3 Creating a sink node

```
upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```



Adding a processor for
writing the Yelling App results

Creating the topology for the Yelling App

- The preceding example uses three lines to build the topology:

```
KStream<String, String> simpleFirstStream =  
  builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));  
KStream<String, String> upperCasedStream =  
  simpleFirstStream.mapValues(String::toUpperCase);  
  upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

Creating the topology for the Yelling App

- To demonstrate this idea, here's another way you could construct the Yelling App topology:

```
builder.stream("src-topic", Consumed.with(stringSerde, stringSerde))  
  ➔ .mapValues(String::toUpperCase)  
  ➔ .to("out-topic", Produced.with(stringSerde, stringSerde));
```

Kafka Streams configuration

- Although Kafka Streams is highly configurable, with several properties you can adjust for your specific needs, the first example uses only two configuration settings, APPLICATION_ID_CONFIG and BOOTSTRAP_SERVERS_CONFIG:

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

Serde creation

- In Kafka Streams, the Serdes class provides convenience methods for creating Serde instances, as shown here:

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

Serde creation

The Serdes class provides default implementations for the following types:

- String
- Byte array
- Long
- Integer
- Double

Listing 3.4 Hello World: the Yelling App

```
public class KafkaStreamsYellingApp {  
    public static void main(String[] args) {  
  
        Properties props = new Properties();  
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id"); ←  
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
  
        StreamsConfig streamingConfig = new StreamsConfig(props); ←  
  
        Serde<String> stringSerde = Serdes.String();  
        StreamsBuilder builder = new StreamsBuilder(); ←  
  
        KStream<String, String> simpleFirstStream = builder.stream("src-topic",  
            => Consumed.with(stringSerde, stringSerde));  
        KStream<String, String> upperCasedStream =  
            => simpleFirstStream.mapValues(String::toUpperCase);  
        upperCasedStream.to("out-topic",  
            => Produced.with(stringSerde, stringSerde));  
  
        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);  
  
        kafkaStreams.start();  
        Thread.sleep(35000);  
        LOG.info("Shutting down the Yelling APP now");  
        kafkaStreams.close();  
  
    }  
}
```

Creates the StreamsConfig with the given properties

Properties for configuring the Kafka Streams program

Creates the Serdes used to serialize/deserialize keys and values

Creates the StreamsBuilder instance used to construct the processor topology

Creates the actual stream with a source topic to read from (the parent node in the graph)

A processor using a Java 8 method handle (the first child node in the graph)

Writes the transformed output to another topic (the sink node in the graph)

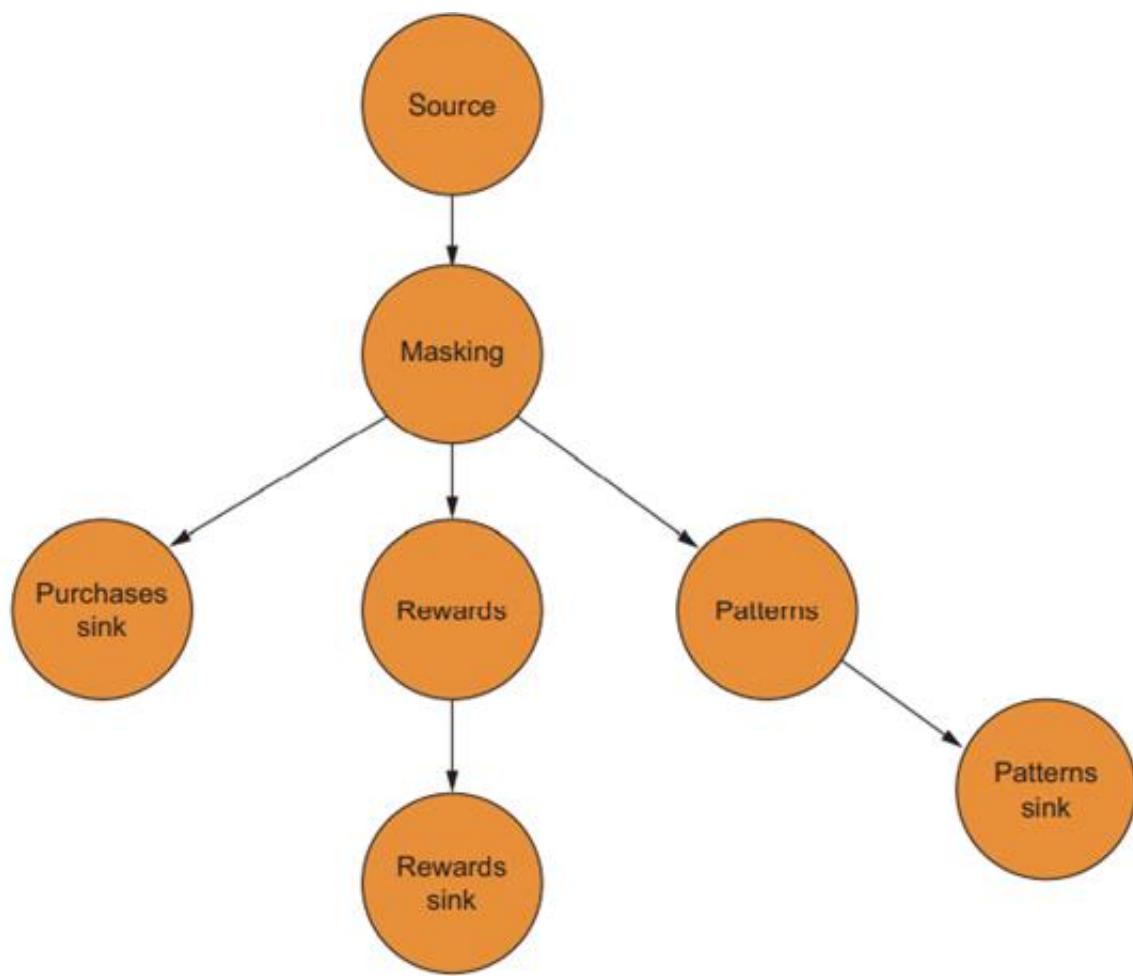
Kicks off the Kafka Streams threads

Serde creation

You've now constructed your first Kafka Streams application. Let's quickly review the steps involved, as it's a general pattern you'll see in most of your Kafka Streams applications:

- Create a StreamsConfig instance.
- Create a Serde object
- Construct a processing topology.
- Start the Kafka Streams program.

Working with customer data



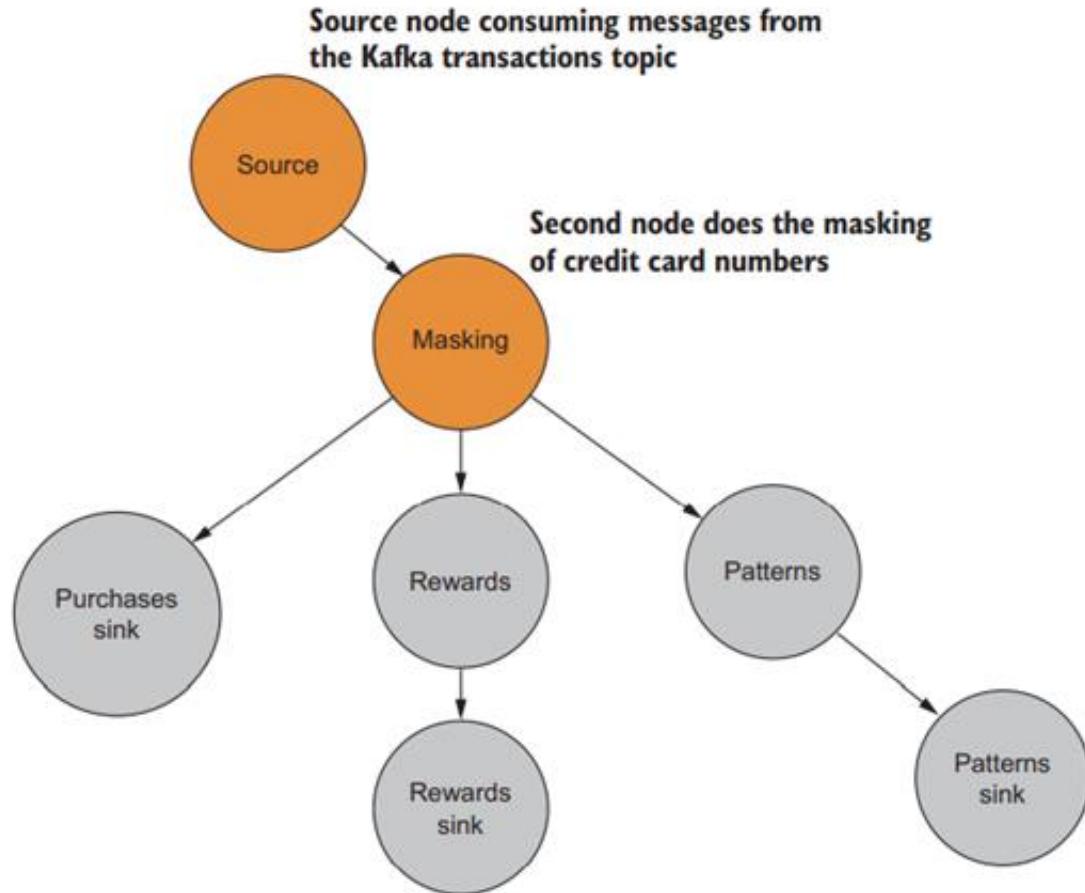
Working with customer data

Let's briefly review the requirements for the streaming program, which will also serve as a good description of what the program will do:

- All records need to have credit card numbers protected, in this case by masking the first 12 digits.
- You need to extract the items purchased and the ZIP code to determine purchase patterns. This data will be written out to a topic.

Constructing a topology

BUILDING THE SOURCE NODE



Constructing a topology

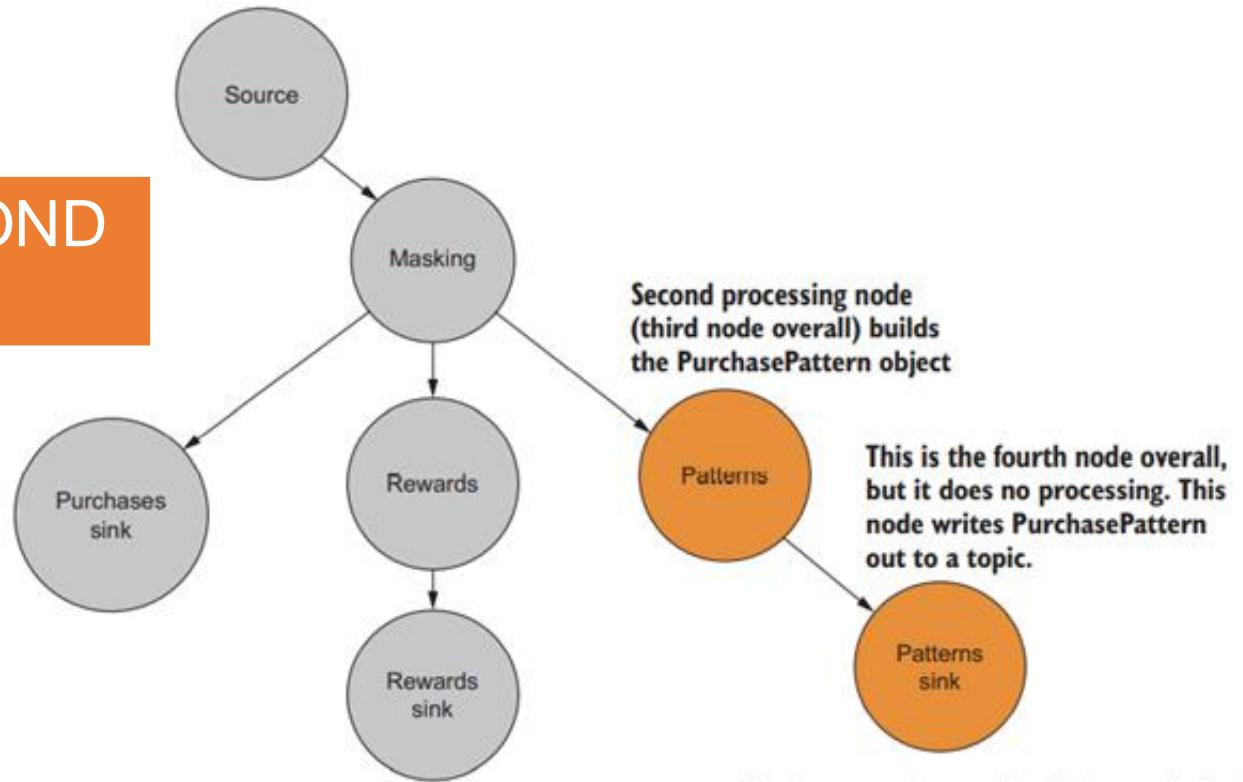
Listing 3.5 Building the source node and first processor

```
KStream<String, Purchase> purchaseKStream =  
    streamsBuilder.stream("transactions",  
    Consumed.with(stringSerde, purchaseSerde))  
    .mapValues(p -> Purchase.builder(p).maskCreditCard().build());
```

HINTS ABOUT FUNCTIONAL PROGRAMMING

- An important concept to keep in mind with the map and mapValues functions is that they're expected to operate without side effects, meaning the functions don't modify the object or value presented as a parameter.
- This is because of the functional programming aspects in the KStream API.

BUILDING THE SECOND PROCESSOR



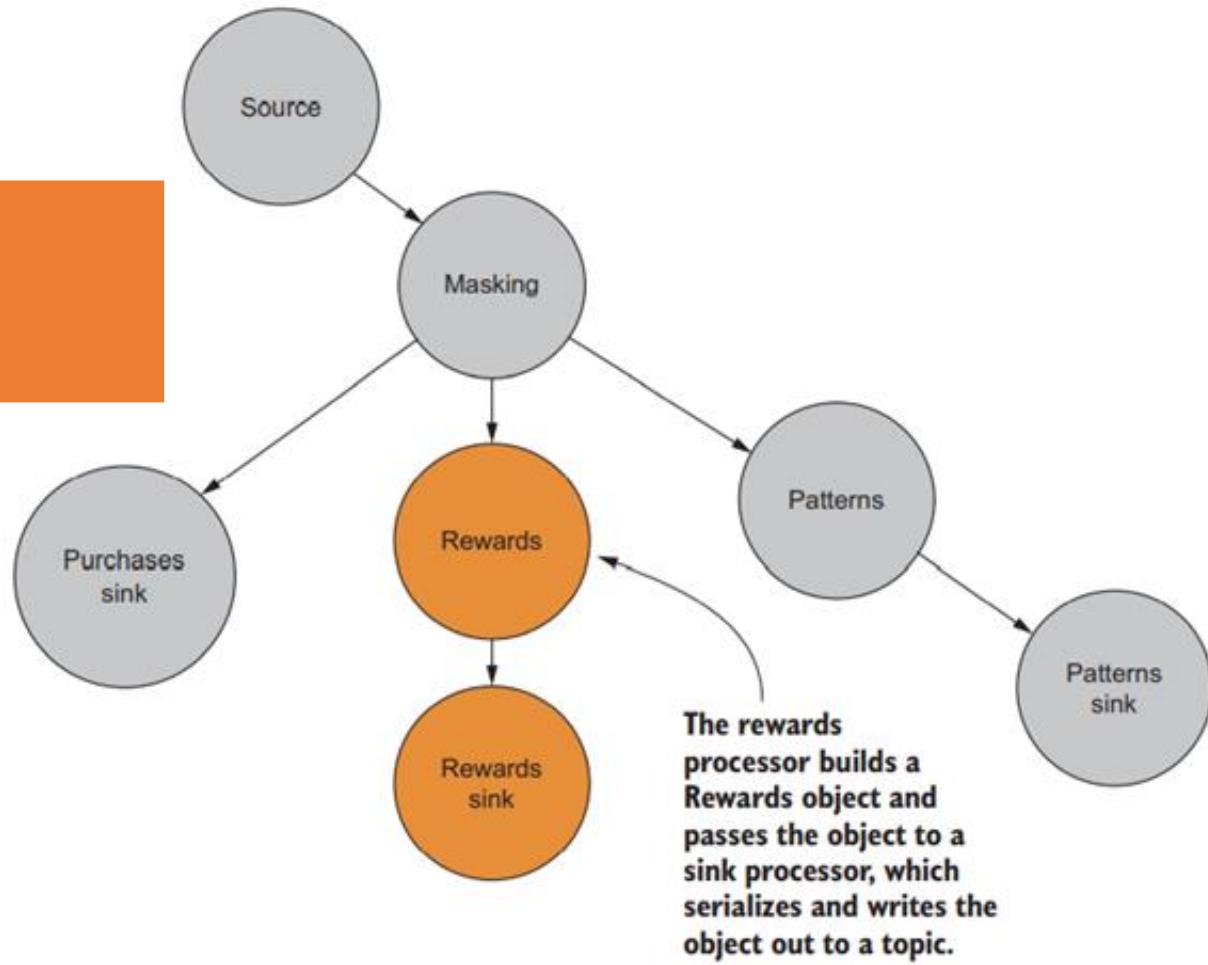
Again you see two nodes in the graph, but with the fluent style of programming in Kafka Streams, sometimes it's easy to overlook the fact that you're creating two nodes.

BUILDING THE SECOND PROCESSOR

Listing 3.6 Second processor and a sink node that writes to Kafka

```
KStream<String, PurchasePattern> patternKStream =  
    → purchaseKStream.mapValues(purchase ->  
    → PurchasePattern.builder(purchase).build());  
  
patternKStream.to("patterns",  
    → Produced.with(stringSerde, purchasePatternSerde));
```

BUILDING THE THIRD PROCESSOR

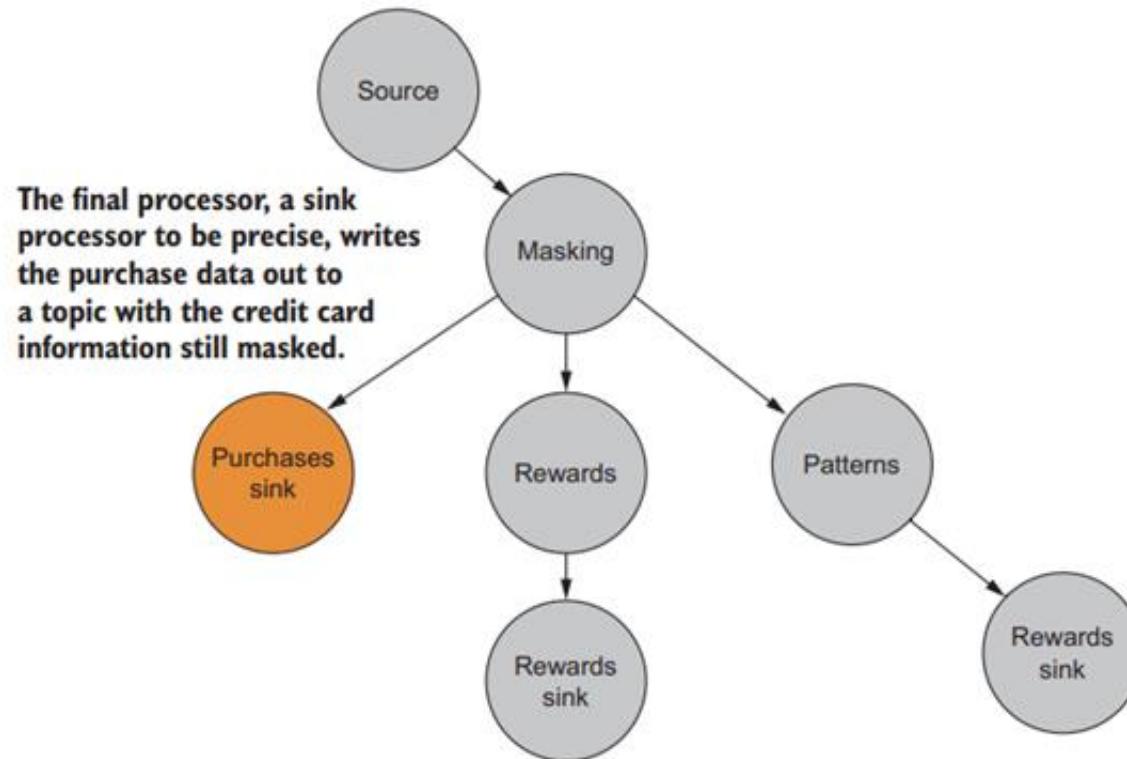


BUILDING THE THIRD PROCESSOR

Listing 3.7 Third processor and a terminal node that writes to Kafka

```
KStream<String, RewardAccumulator> rewardsKStream =  
    purchaseKStream.mapValues(purchase ->  
        RewardAccumulator.builder(purchase).build());  
    rewardsKStream.to("rewards",  
        Produced.with(stringSerde, rewardAccumulatorSerde));
```

BUILDING THE LAST PROCESSOR



BUILDING THE LAST PROCESSOR

Listing 3.8 Final processor

```
purchaseKStream.to("purchases", Produced.with(stringSerde, purchaseSerde));
```

Listing 3.9 ZMart customer purchase KStream program

```
public class ZMartKafkaStreamsApp {  
  
    public static void main(String[] args) {  
        // some details left out for clarity  
  
        StreamsConfig streamsConfig = new StreamsConfig(getProperties());  
  
        JsonSerializer<Purchase> purchaseJsonSerializer = new  
        ↪ JsonSerializer<>();  
        JsonDeserializer<Purchase> purchaseJsonDeserializer = ↪ Creates the Serde; the  
        ↪ new JsonDeserializer<>(Purchase.class);  
        Serde<Purchase> purchaseSerde =  
        ↪ Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);  
        //Other Serdes left out for clarity  
  
        Serde<String> stringSerde = Serdes.String();  
  
        StreamsBuilder streamsBuilder = new StreamsBuilder();  
  
        KStream<String, Purchase> purchaseKStream =  
        ↪ streamsBuilder.stream("transactions",  
        ↪ Consumed.with(stringSerde, purchaseSerde))  
        ↪ .mapValues(p -> Purchase.builder(p).maskCreditCard().build());  
    }  
}
```

```
    patternKStream.to("patterns",
→ Produced.with(stringSerde, purchasePatternSerde)) ;

    KStream<String, RewardAccumulator> rewardsKStream =
→ purchaseKStream.mapValues(purchase ->
→ RewardAccumulator.builder(purchase).build()) ; | Builds the
                                                 RewardAccumulator processor

    rewardsKStream.to("rewards",
→ Produced.with(stringSerde, rewardAccumulatorSerde)) ;

    purchaseKStream.to("purchases",
→ Produced.with(stringSerde, purchaseSerde)) ; | Builds the storage sink, the topic
                                                 used by the storage consumer

    KafkaStreams kafkaStreams =
→ new KafkaStreams(streamsBuilder.build(), streamsConfig) ;
    kafkaStreams.start() ;

}
```

Creating a custom Serde

- Kafka transfers data in byte array format. Because the data format is JSON, you need to tell Kafka how to convert an object first into JSON and then into a byte array when it sends data to a topic.
- Conversely, you need to specify how to convert consumed byte arrays into JSON, and then into the object type your processors will use.
- This conversion of data to and from different formats is why you need a Serde.

Listing 3.10 Generic serializer

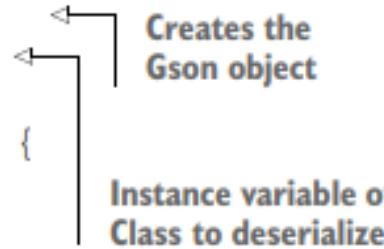
```
public class JsonSerializer<T> implements Serializer<T> {  
    private Gson gson = new Gson();  
    @Override  
    public void configure(Map<String, ?> map, boolean b) {  
    }  
    @Override  
    public byte[] serialize(String topic, T t) {  
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));  
    }  
    @Override  
    public void close() {  
    }  
}
```

Creates the
Gson object

Serializes an
object to bytes

Listing 3.11 Generic deserializer

```
public class JsonDeserializer<T> implements Deserializer<T> {  
  
    private Gson gson = new Gson();  
    private Class<T> deserializedClass;  
  
    public JsonDeserializer(Class<T> deserializedClass) {  
        this.deserializedClass = deserializedClass;  
    }  
  
    public JsonDeserializer() {  
    }  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public void configure(Map<String, ?> map, boolean b) {  
        if(deserializedClass == null) {  
            deserializedClass = (Class<T>) map.get("serializedClass");  
        }  
    }  
}
```



The code includes two annotations:

- A callout arrow points from the line `private Gson gson = new Gson();` to the text "Creates the Gson object".
- A callout arrow points from the line `private Class<T> deserializedClass;` to the text "Instance variable of Class to deserialize".

```
@Override  
public T deserialize(String s, byte[] bytes) {  
    if(bytes == null){  
        return null;  
    }  
  
    return gson.fromJson(new String(bytes),deserializedClass);    ←  
}  
  
@Override  
public void close() {  
}  
}
```

Deserializes bytes to an
instance of expected Class

Creating a custom Serde

- Now, let's go back to the following lines from listing 3.9:

```
JsonDeserializer<Purchase> purchaseJsonDeserializer =  
    new JsonDeserializer<>(Purchase.class);  
JsonSerializer<Purchase> purchaseJsonSerializer =  
    new JsonSerializer<>();  
Serde<Purchase> purchaseSerde =  
    Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);
```

Creates the Deserializer for the Purchase class

Creates the Serializer for the Purchase class

Creates the Serde for Purchase objects

Interactive development

```
[patterns]: null , PurchasePattern{zipCode='21842', item='beer', date=Thu Feb 18 12:07:10 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Doe,Andrew', purchaseTotal=18.5500}
[purchases]: null , Purchase{firstName='Andrew', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-3820', itemPurchased='beer', quantity=4, price=4.6377, purchaseDate=Thu Feb 18 12:07:10 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='10005', item='eggs', date=Thu Feb 11 22:03:37 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Eric', purchaseTotal=20.8000}
[purchases]: null , Purchase{firstName='Eric', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-3820', itemPurchased='eggs', quantity=2, price=10.4043, purchaseDate=Thu Feb 11 22:03:37 EST 2016, zipCode='10005'}
[patterns]: null , PurchasePattern{zipCode='20852', item='batteries', date=Tue Feb 16 14:17:45 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Andrew', purchaseTotal=5.8758}
[purchases]: null , Purchase{firstName='Andrew', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-3820', itemPurchased='batteries', quantity=1, price=5.8758, purchaseDate=Tue Feb 16 14:17:45 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='20852', item='shampoo', date=Sat Feb 13 04:58:42 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Loxly,Eric', purchaseTotal=18.7134}
[purchases]: null , Purchase{firstName='Eric', lastName='Loxly', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='shampoo', quantity=2, price=5.3567, purchaseDate=Sat Feb 13 04:58:42 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='19971', item='diapers', date=Mon Feb 15 21:23:01 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Black,Andrew', purchaseTotal=11.7635}
[purchases]: null , Purchase{firstName='Andrew', lastName='Black', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='diapers', quantity=1, price=11.7635, purchaseDate=Mon Feb 15 21:23:01 EST 2016, zipCode='19971'}
[patterns]: null , PurchasePattern{zipCode='20852', item='eggs', date=Thu Feb 18 17:31:14 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Eric', purchaseTotal=6.4234}
[purchases]: null , Purchase{firstName='Eric', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='eggs', quantity=1, price=6.4234, purchaseDate=Thu Feb 18 17:31:14 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='21842', item='diapers', date=Fri Feb 19 10:23:28 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Bob', purchaseTotal=48.81}
[purchases]: null , Purchase{firstName='Bob', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-8111', itemPurchased='diapers', quantity=4, price=10.2025, purchaseDate=Fri Feb 19 10:23:28 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='20852', item='batteries', date=Thu Feb 18 23:18:06 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Steve', purchaseTotal=29.1552}
[purchases]: null , Purchase{firstName='Steve', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='batteries', quantity=4, price=7.2888, purchaseDate=Thu Feb 18 23:18:06 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='21842', item='doughnuts', date=Sat Feb 13 21:20:45 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Smith,Bob', purchaseTotal=13.3516}
[purchases]: null , Purchase{firstName='Bob', lastName='Smith', creditCardNumber='xxxx-xxxx-xxxx-1939', itemPurchased='doughnuts', quantity=2, price=6.6759, purchaseDate=Sat Feb 13 21:20:45 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='10005', item='beer', date=Fri Feb 12 19:55:06 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Eric', purchaseTotal=8.2866}
[purchases]: null , Purchase{firstName='Eric', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-3858', itemPurchased='beer', quantity=2, price=4.1433, purchaseDate=Fri Feb 12 19:55:06 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='10005', item='beer', date=Fri Feb 12 02:26:32 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Doe,Sarah', purchaseTotal=13.8466}
[purchases]: null , Purchase{firstName='Sarah', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-8783', itemPurchased='beer', quantity=1, price=13.8466, purchaseDate=Fri Feb 12 02:26:32 EST 2016, zipCode='10005'}
```

Interactive development

- There's a method on the KStream interface that can be useful during development: the KStream.print method, which takes an instance of the Printed<K, V> class Printed provides two static methods allowing you print to stdout, Printed.toSysOut(), or to write results to a file, Printed.toFile(filePath).
- Additionally, you can label your printed results by chaining the withLabel() method, allowing you to print an initial header with the records.

Interactive development

Sets up to print the RewardAccumulator transformation to the console

```
patternKStream.print(Printed.<String, PurchasePattern>toSysOut()  
    .withLabel("patterns"));
```

Sets up to print the PurchasePattern transformation to the console

```
rewardsKStream.print(Printed.<String, RewardAccumulator>toSysOut()  
    .withLabel("rewards"));
```

```
purchaseKStream.print(Printed.<String, Purchase>toSysOut()  
    .withLabel("purchases"));
```

Prints the purchase data to the console

Interactive development

Name(s) given to the print statement, helpful to make this the same as the topic

The values for the records. Note that these are JSON strings and the Purchase, PurchasePattern, and RewardAccumulator objects defined `toString` methods to get this rendering on the console.

Note the masked credit card number!

```
[purchases]: null Purchase{firstName='Andrew', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-3020', itemPurchased='beer', quantity=1, purchaseDate=Thu Feb 11 22:03:37 EST 2016}
[patterns]: null PurchasePattern{zipCode='10005', item='eggs', date=Thu Feb 11 22:03:37 EST 2016}
[rewards]: null RewardAccumulator{customerName='Grange, Eric', purchaseTotal=20.8086}
```

The keys for the records, which are null in this case

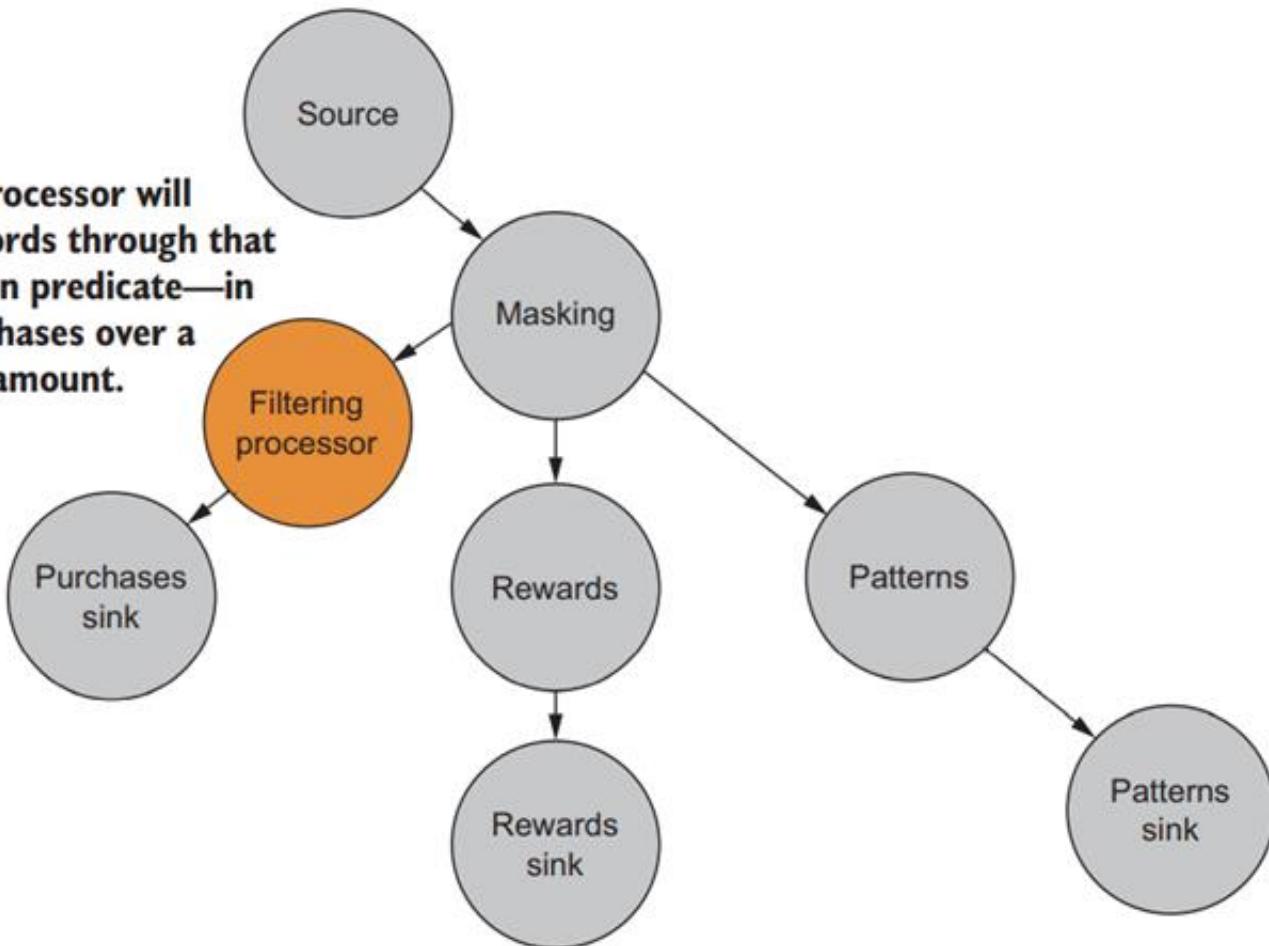
Next steps

- At this point, you have your Kafka Streams purchase-analysis program running well.
- Other applications have also been developed to consume the messages written to the patterns, rewards, and purchases topics, and the results for ZMart have been good.

New requirements

- Purchases under a certain dollar amount need to be filtered out.
- Upper management isn't much interested in the small purchases for general daily articles.
- ZMart has expanded and has bought an electronics chain and a popular coffee house chain.
- All purchases from these new stores will flow through the streaming application you've set up.
- You need to send the purchases from these new subsidiaries to their topics.

The filtering processor will only allow records through that match the given predicate—in this case, purchases over a certain dollar amount.



FILTERING PURCHASES

- You can use the KStream method, which takes a Predicate<K,V> instance as a parameter.
- Although you're chaining method calls together here, you're creating a new processing node in the topology

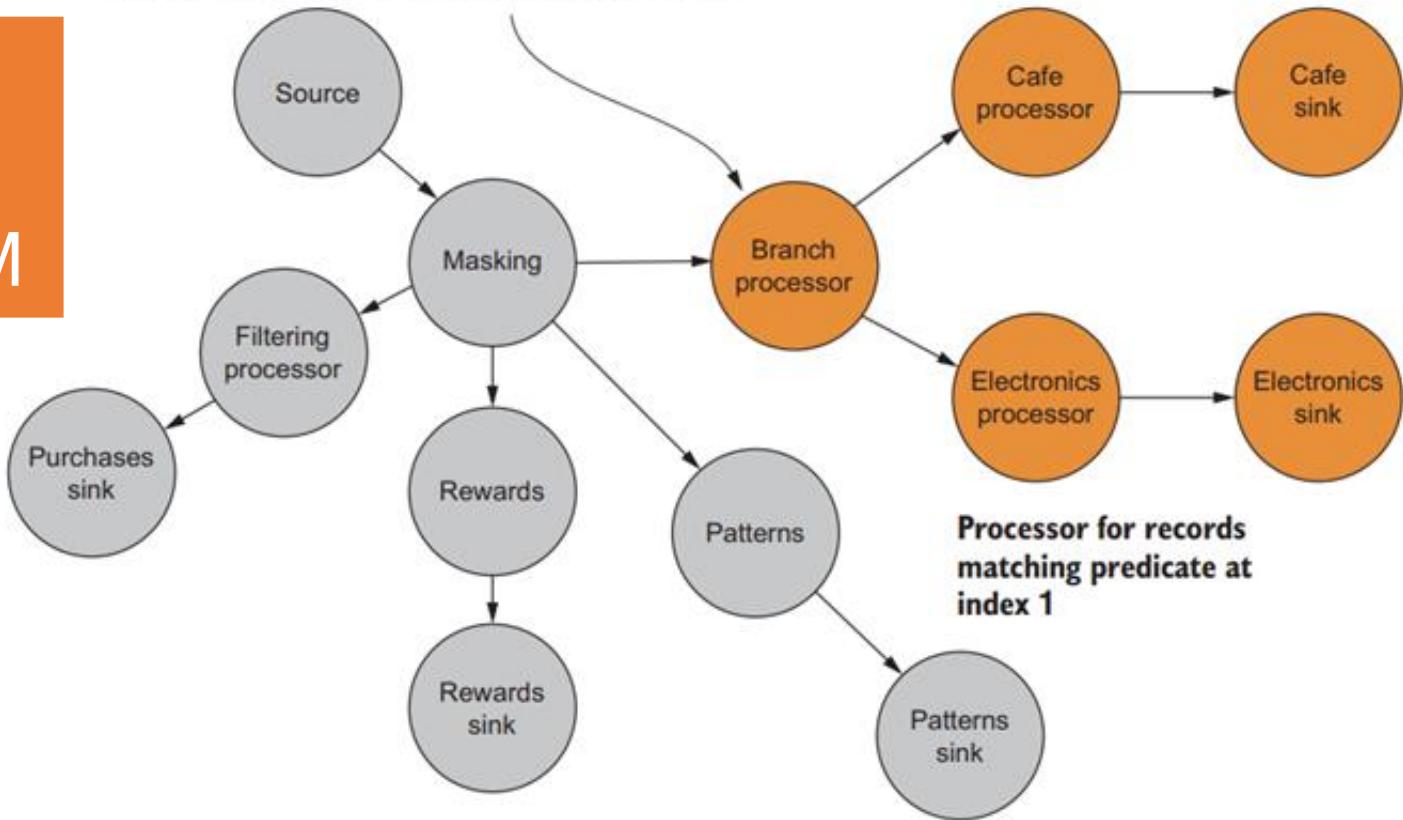
Listing 3.12 Filtering on KStream

```
KStream<Long, Purchase> filteredKStream =  
    ➔ purchaseKStream( key, purchase ) ->  
    ➔ purchase.getPrice() > 5.00 .selectKey( purchaseDateAsKey );
```

SPLITTING/ BRANCHING THE STREAM

The KStream.branch method takes an array of predicates and returns an array containing an equal number of KStream instances, each one accepting records matching the corresponding predicate.

Processor for records matching predicate at index 0



Processor for records matching predicate at index 1

SPLITTING/BRANCHING THE STREAM

- As records from the original stream flow through the branch processor, each record is matched against the supplied predicates in the order that they're provided.
- The processor assigns records to a stream on the first match; no attempts are made to match additional predicates.

Listing 3.13 Splitting the stream

```
Predicate<String, Purchase> isCoffee =  
  ➔ (key, purchase) ->  
  ➔ purchase.getDepartment().equalsIgnoreCase("coffee");
```

Creates the predicates as Java 8 lambdas

```
Predicate<String, Purchase> isElectronics =  
  ➔ (key, purchase) ->  
  ➔ purchase.getDepartment().equalsIgnoreCase("electronics");
```

```
int coffee = 0;           ← Labels the expected indices  
int electronics = 1;     of the returned array
```

```
KStream<String, Purchase>[] kstreamByDept =  
  ➔ purchaseKStream.branch(isCoffee, isElectronics);
```

Calls branch to split the original stream into two streams

```
kstreamByDept [coffee].to( "coffee",  
  Produced.with(stringSerde, purchaseSerde));  
kstreamByDept [electronics].to("electronics",  
  ➔ Produced.with(stringSerde, purchaseSerde));
```

Writes the results of each stream out to a topic

GENERATING A KEY

- Kafka messages are in key/value pairs, so all records flowing through a Kafka Streams application are key/value pairs as well.
- But there's no requirement stating that keys can't be null.
- In practice, if there's no need for a particular key, having a null key will reduce the overall amount of data that travels the network

GENERATING A KEY

Listing 3.14 Generating a new key

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey =  
    (key, purchase) -> purchase.getPurchaseDate().getTime();
```

The **KeyValueMapper** extracts the purchase date and converts to a long.

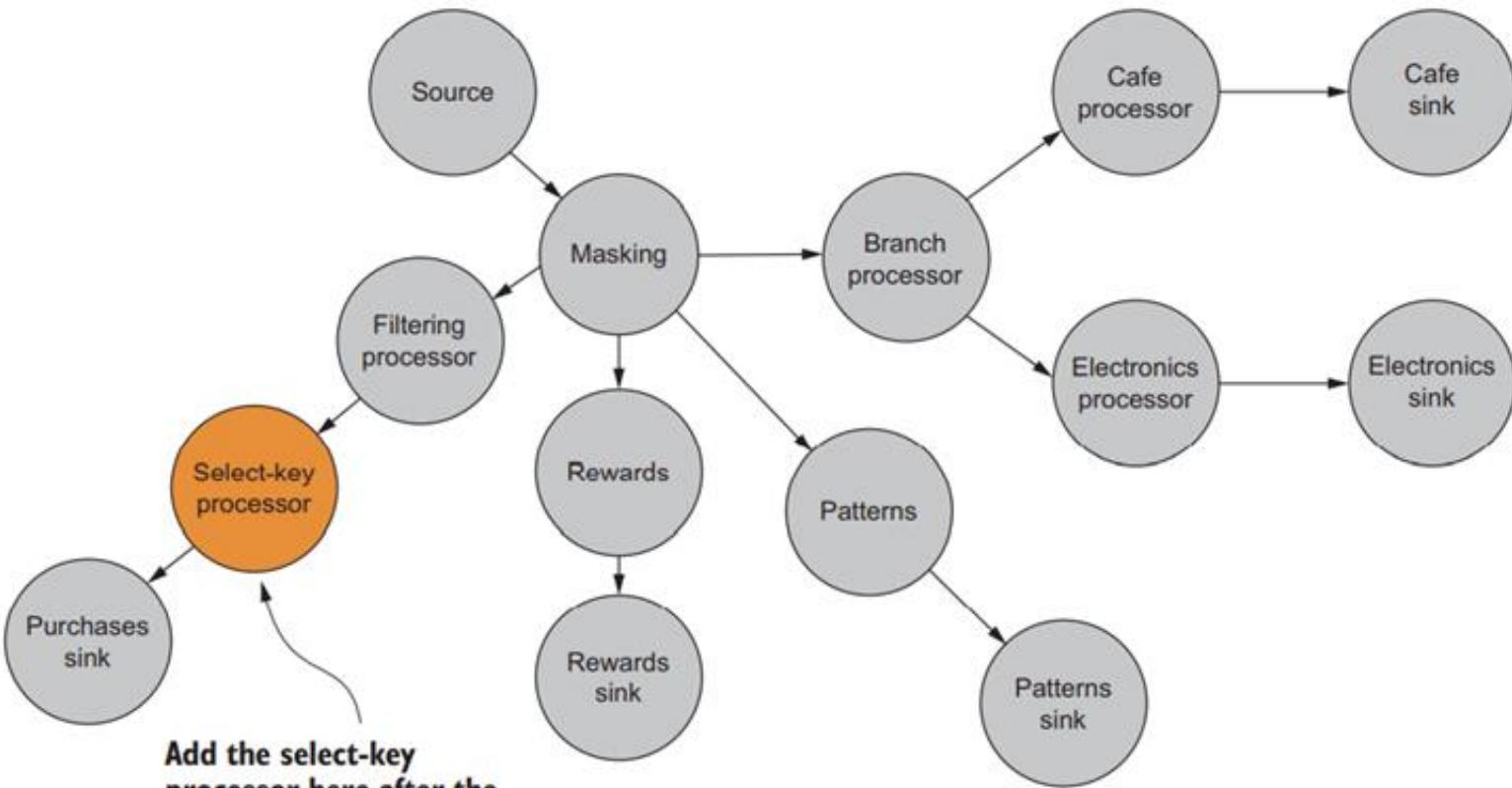
```
KStream<Long, Purchase> filteredKStream =  
    purchaseKStream((key, purchase) ->  
        purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);
```

Filters out purchases and selects the key in one statement

```
filteredKStream.print(Printed.<Long, Purchase>  
    .toSysOut().withLabel("purchases"));  
filteredKStream.to("purchases",  
    Produced.with(Serdes.Long(), purchaseSerde));
```

Prints the results to the console

Materializes the results to a Kafka topic



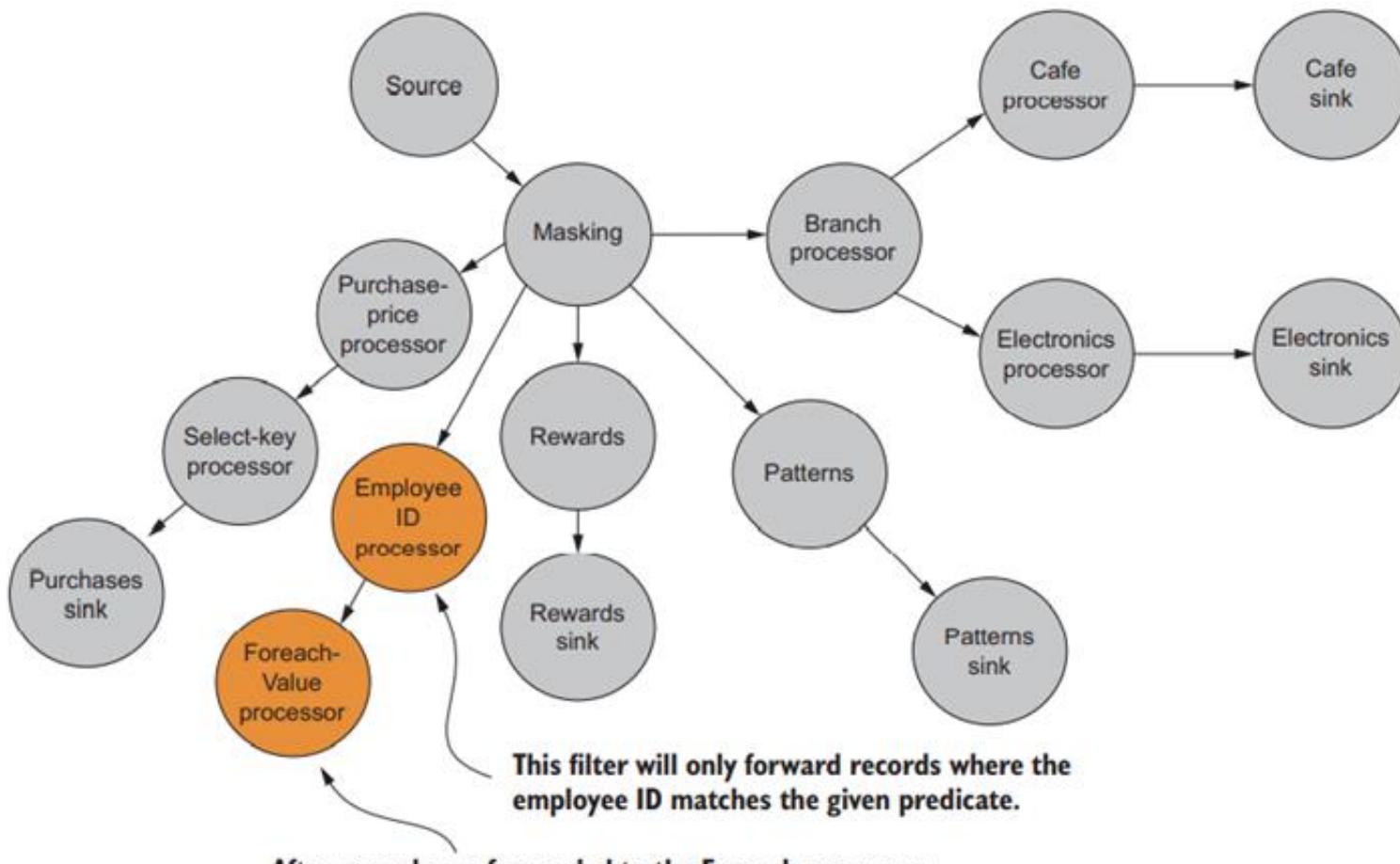
Add the select-key processor here after the filtering, as you only need to generate keys for records that will be written out to the purchases topic.

Writing records outside of Kafka

- The security department at ZMart has approached you.
- Apparently, in one of the stores, there's a suspicion of fraud.
- There have been reports that a store manager is entering invalid discount codes for purchases.
- Security isn't sure what's going on, but they're asking for your help.

FOREACH ACTIONS

- The first thing you need to do is create a new KStream that filters results down to a single employee ID.
- Even though you have a large amount of data flowing through your topology, this filter will reduce the volume to a tiny amount.
- Here, you'll use KStream with a predicate that looks to match a specific employee ID.



This filter will only forward records where the employee ID matches the given predicate.

After records are forwarded to the Foreach processor, the value of each record is written to an external database.

FOREACH ACTIONS

Listing 3.15 Foreach operations

```
ForeachAction<String, Purchase> purchaseForeachAction = (key, purchase) ->
    ➔ SecurityDBService.saveRecord(purchase.getPurchaseDate(),
    ➔ purchase.getEmployeeId(), purchase.getItemPurchased());

purchaseKStream.filter((key, purchase) ->
    ➔ purchase.getEmployeeId()
    ➔ .equals("source code has 000000"))
    ➔ .foreach(purchaseForeachAction);
```

Summary

- You can use the KStream.mapValues function to map incoming record values to new values, possibly of a different type.
- You also learned that these mapping changes shouldn't modify the original objects.
- Another method, KStream.map, performs the same action but can be used to map both the key and the value to something new.

COMPLETE LAB 3

4. Streams and state



Streams and state

This lesson covers

- Applying stateful operations to Kafka Streams
- Using state stores for lookups and remembering previously seen data
- Joining streams for added insight
- How time and timestamps drive Kafka Streams

Thinking of events

- When it comes to event processing, events sometimes require no further information or context.
- At other times, an event on its own may be understood in a literal sense, but without some added context, you might miss the significance of what is occurring; you might think of the event in a whole new light, given some additional information

Thinking of events

Timeline

9:30 a.m.

9:50 a.m.

10:30 a.m.

10,000 shares of XYZ
Pharmaceutical
purchased

12,000 shares of XYZ
Pharmaceutical
purchased

15,000 shares of XYZ
Pharmaceutical
purchased

Thinking of events

Timeline

9:30 a.m.

9:50 a.m.

10:30 a.m.

11:00 a.m.

10,000 shares of XYZ
Pharmaceutical
purchased

12,000 shares of XYZ
Pharmaceutical
purchased

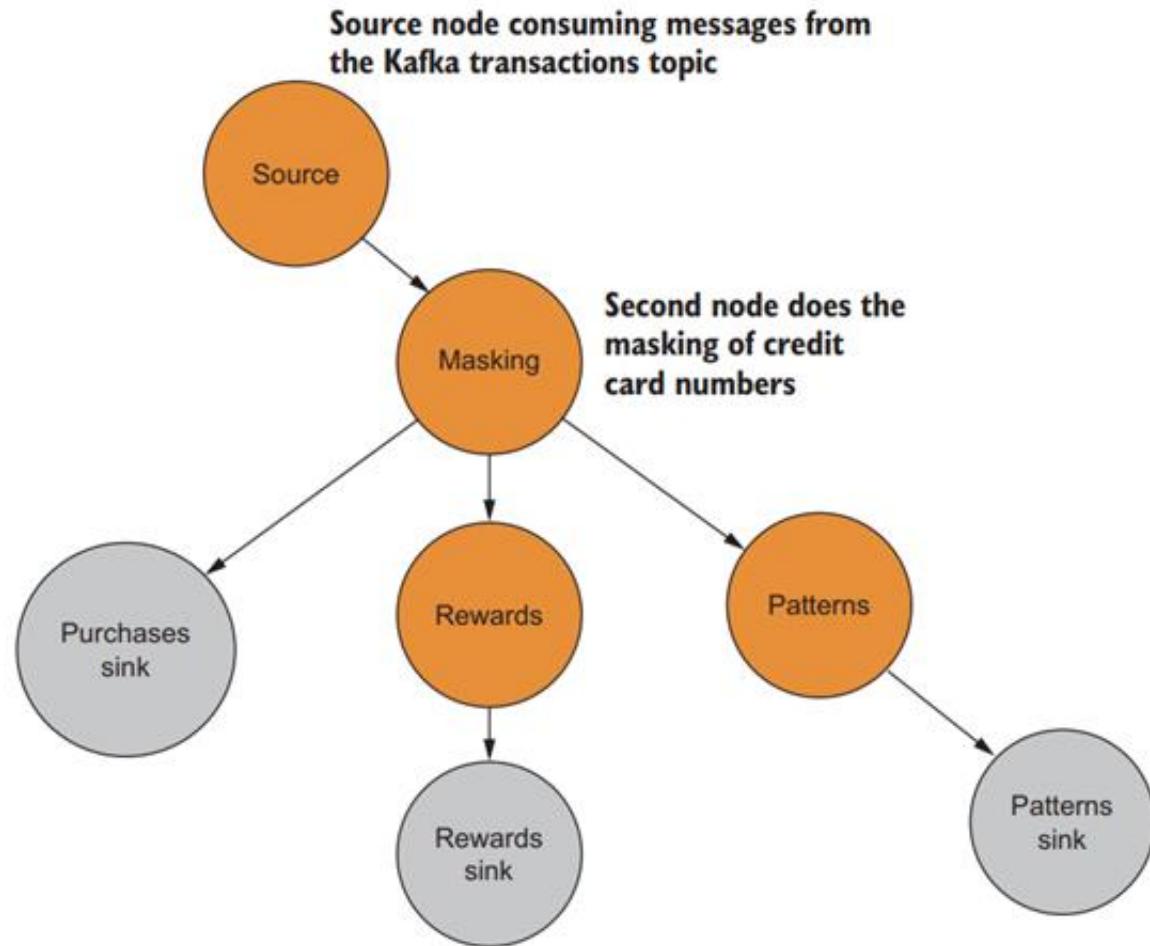
15,000 shares of XYZ
Pharmaceutical
purchased

FDA announces approval
of experimental drug
developed by XYZ
Pharmaceutical. Stock
price soars 30%.

Streams need state

- The preceding fictional scenario illustrates something that most of us already know instinctively.
- Sometimes it's easy to reason about what's going on, but usually you need some context to make good decisions.
- When it comes to stream processing, we call that added context state.
- At first glance, the notions of state and stream processing may seem to be at odds with each other

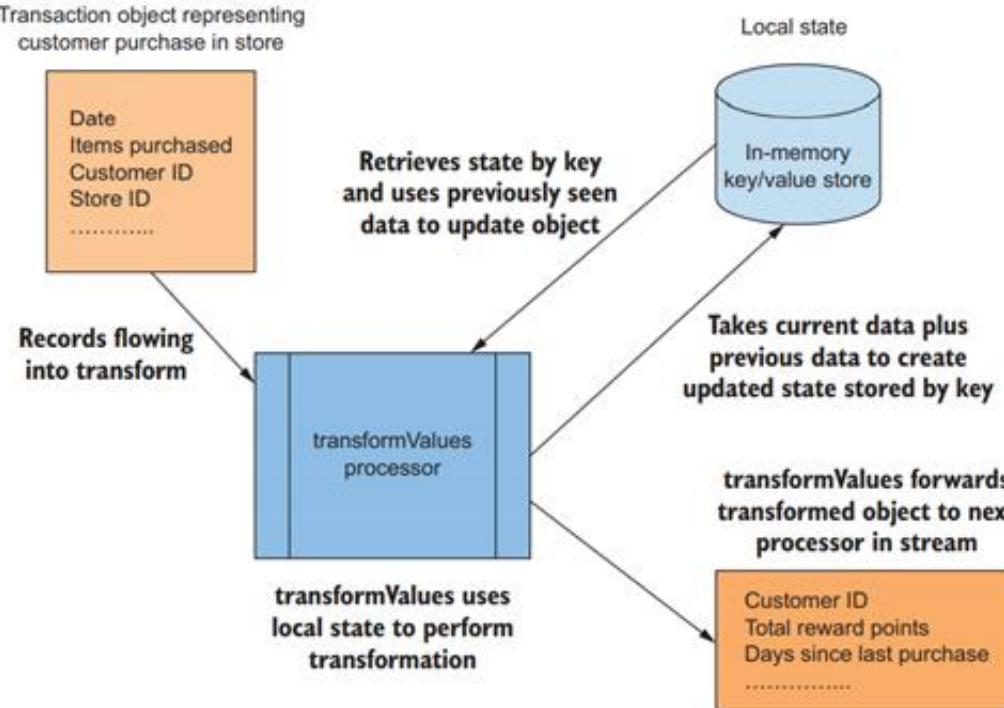
Applying stateful operations to Kafka Streams



Applying stateful operations to Kafka Streams

- In this topology, you produced a stream of purchase-transaction events.
- One of the processing nodes in the topology calculated reward points for customers based on the amount of the sale.
- But in that processor, you just calculated the total number of points for the single transaction and forwarded the results.

The transformValues processor



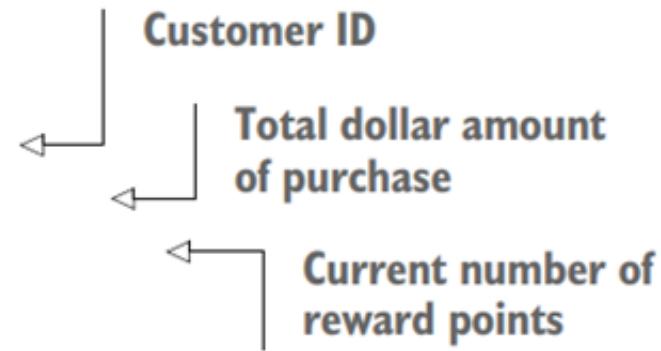
Stateful customer rewards

- The rewards processor from the lesson 3 topology for ZMart extracts information for customers belonging to ZMart's rewards program.
- Initially, the rewards processor used the KStream.mapValues() method to map the incoming Purchase object into a RewardAccumulator object.

Stateful customer rewards

- Now, the requirements have changed some, and points are being associated with the ZMart rewards program:

```
public class RewardAccumulator {  
  
    private String customerId;  
    private double purchaseTotal;  
    private int currentRewardPoints;  
  
    //details left out for clarity  
}
```



Stateful customer rewards

Listing 4.1 Refactored RewardAccumulator object

```
public class RewardAccumulator {  
  
    private String customerId;  
    private double purchaseTotal;  
    private int currentRewardPoints;  
    private int daysFromLastPurchase;  
    private long totalRewardPoints;  
  
    //details left out for clarity  
}
```

Field added for tracking total points

Stateful customer rewards

The difference lies in the ability to use local state to perform the transformation.
Specifically, you'll take two main two steps:

- Initialize the value transformer.
- Map the Purchase object to a RewardAccumulator using state

Initializing the value transformer

Listing 4.2 init() method

```
private KeyValueStore<String, Integer> stateStore;           ↪ Instance variables  
private final String storeName;  
private ProcessorContext context;  
  
public void init(ProcessorContext context) {  
    this.context = context;  
    stateStore = (KeyValueStore)  
        this.context.getStateStore(storeName);  
}
```

Sets a local reference to ProcessorContext

Retrieves the StateStore instance by storeName variable. storeName is set in the constructor.

Mapping the Purchase object to a RewardAccumulator using state

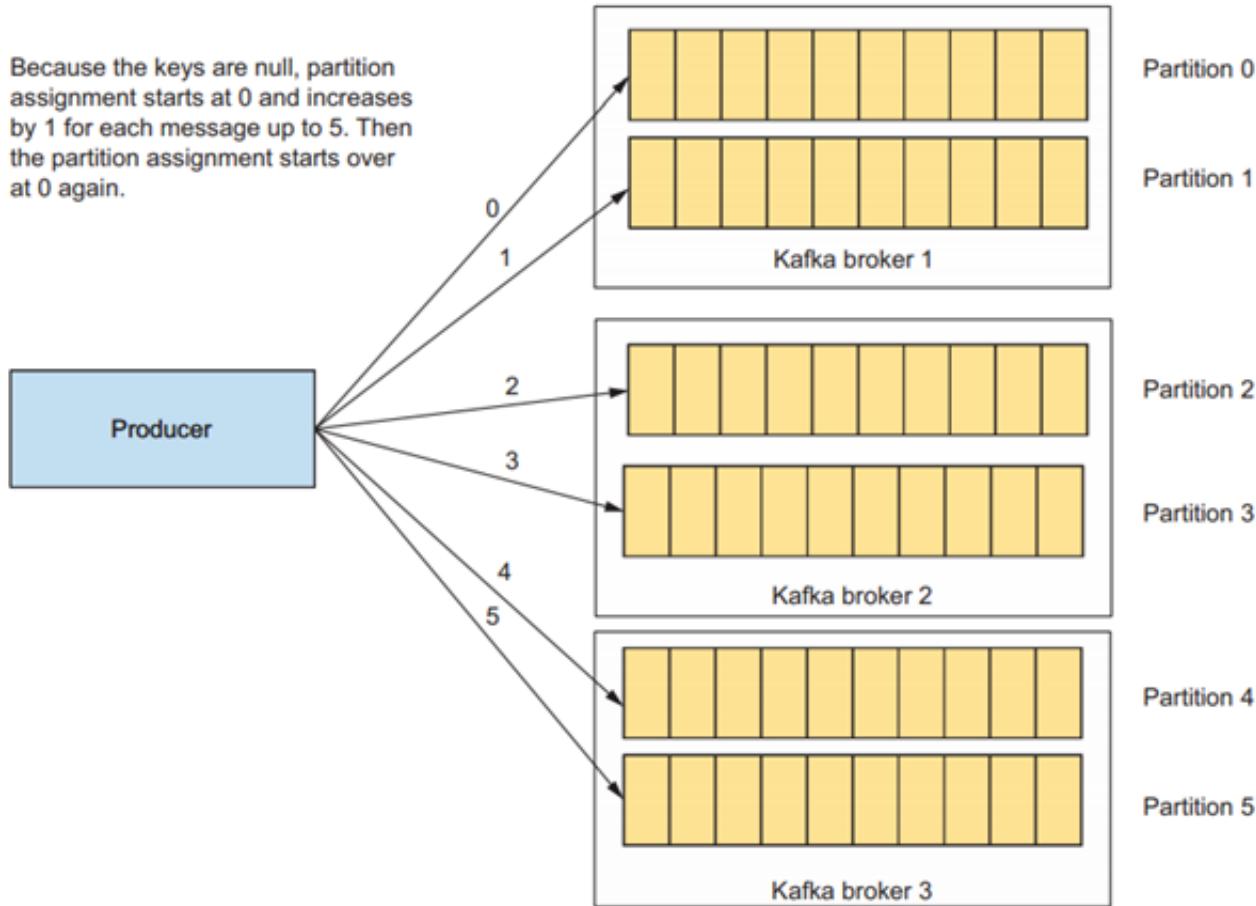
1. Check for points accumulated so far by customer ID.
1. Sum the points for the current transaction and present the total.
1. Set the reward points on the RewardAccumulator to the new total amount.
1. Save the new total points by customer ID in the local state store.

Mapping the Purchase object to a RewardAccumulator using state

Listing 4.3 Transforming Purchase using state

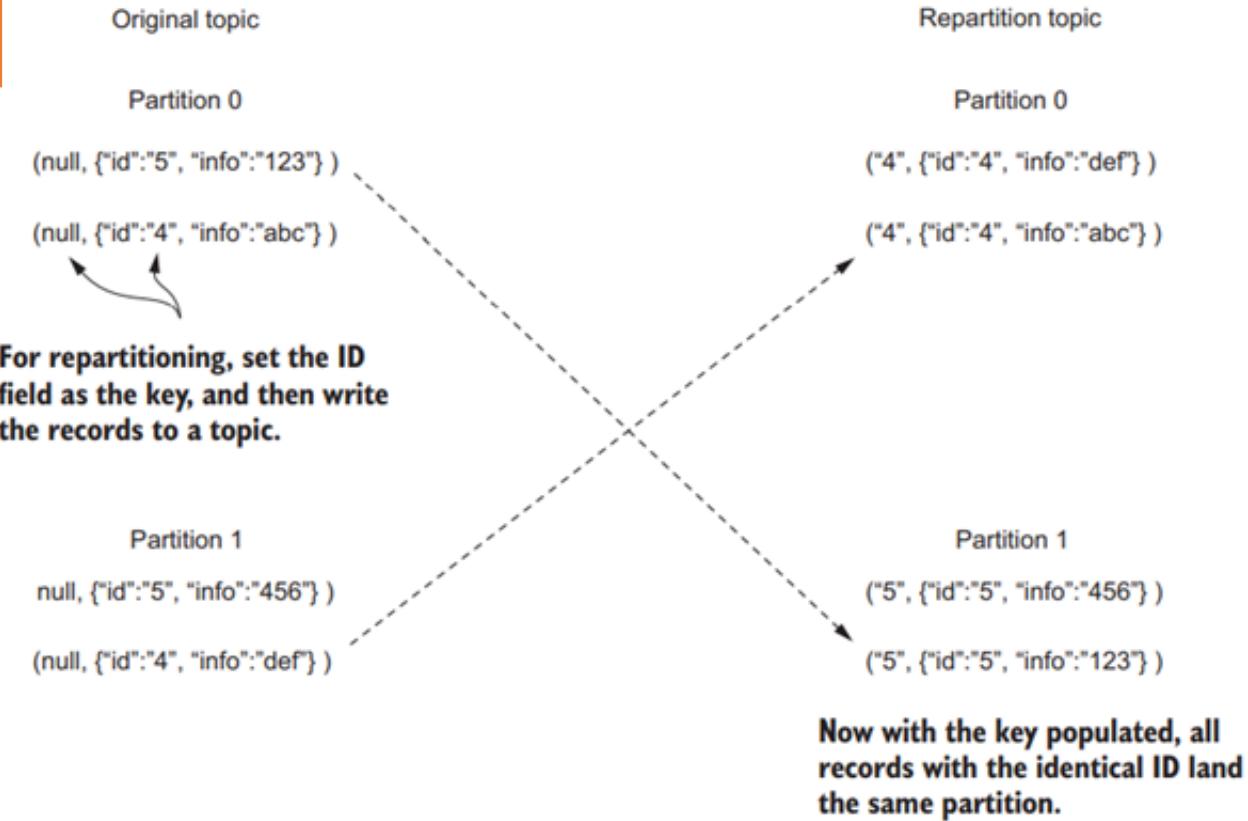
```
public RewardAccumulator transform(Purchase value) {  
    RewardAccumulator rewardAccumulator =  
        RewardAccumulator.builder(value).build(); ← Builds the Reward-  
    Integer accumulatedSoFar =  
        stateStore.get(rewardAccumulator.getCustomerId()); ← Retrieves the latest  
    if (accumulatedSoFar != null) {  
        rewardAccumulator.addRewardPoints(accumulatedSoFar); ← If an accumulated number exists,  
    }  
        adds it to the current total  
    stateStore.put(rewardAccumulator.getCustomerId(),  
        rewardAccumulator.getTotalRewardPoints()); ← Stores the new  
    return rewardAccumulator; ← Returns the new  
}← accumulated  
    rewards points
```

Because the keys are null, partition assignment starts at 0 and increases by 1 for each message up to 5. Then the partition assignment starts over at 0 again.



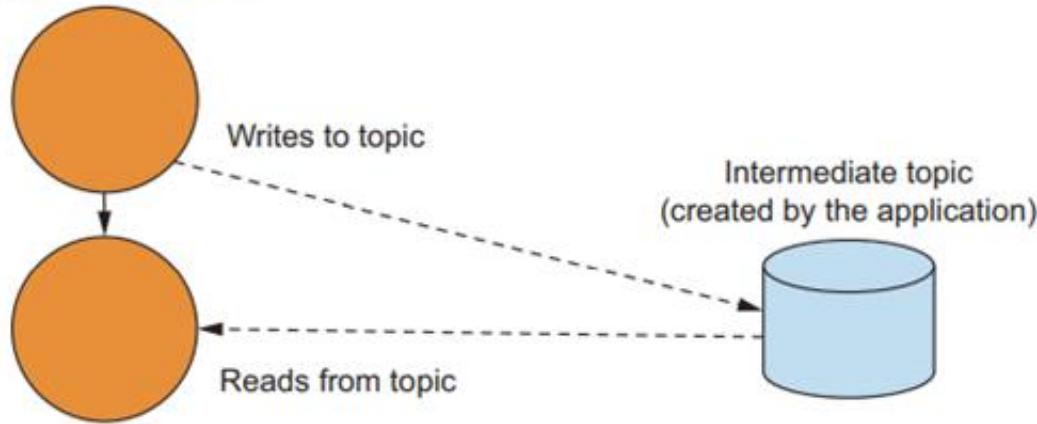
REPARTITIONING THE DATA

The keys are originally null, so distribution is done round-robin, resulting in records with the same ID across different partitions.



REPARTITIONING IN KAFKA STREAMS

Original KStream node
making the "through" call



The returned KStream instance immediately starts to consume from the intermediate topic.

Writing out to an intermediate topic and then reading from it in a new KStream instance

REPARTITIONING IN KAFKA STREAMS

- Under the covers, Kafka Streams creates a sink and source node.
- The sink node is a child processor of the calling KStream instance, and the new KStream instance uses the new source node for its source of records.
- You could write the same type of sub topology yourself using the DSL, but using the KStream.through() method is more convenient.

REPARTITIONING IN KAFKA STREAMS

Listing 4.4 Using the KStream.through method

```
RewardsStreamPartitioner streamPartitioner =  
    new RewardsStreamPartitioner();  
  
KStream<String, Purchase> transByCustomerStream =  
    purchaseKStream.through("customer_transactions",  
        Produced.with(stringSerde,  
                      purchaseSerde,  
                      streamPartitioner));
```

↳ Instantiates the concrete StreamPartitioner instance

↳ Creates a new KStream with the KStream.through method

USING A STREAMPARTITIONER

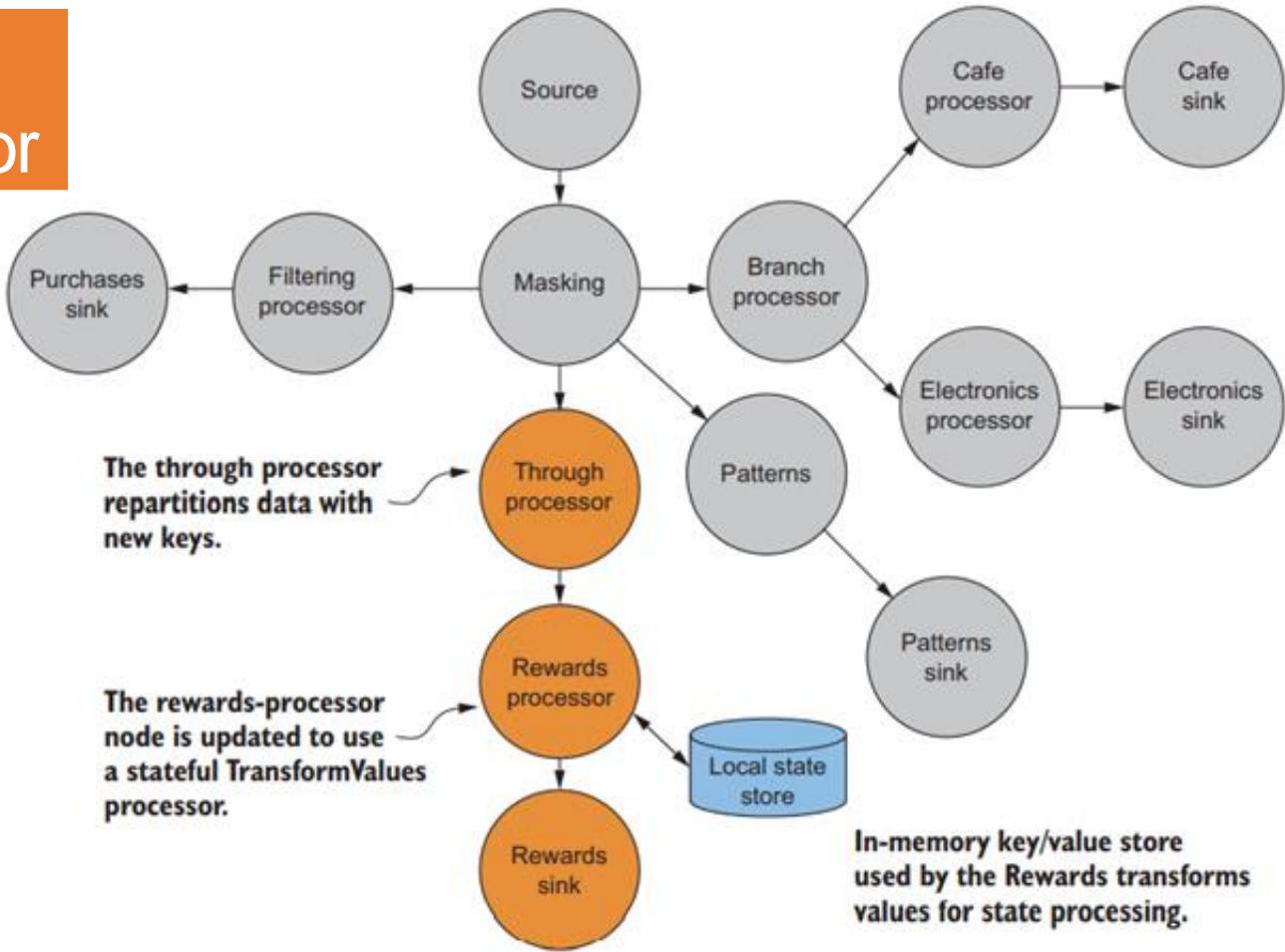
Listing 4.5 RewardsStreamPartitioner

```
public class RewardsStreamPartitioner implements
→ StreamPartitioner<String, Purchase> {

    @Override
    public Integer partition(String key,
                            Purchase value,
                            int numPartitions) {
        return value.getCustomerId().hashCode() % numPartitions;
    }
}
```

Determines the
partition by
customer ID

Updating the rewards processor



Updating the rewards processor

- Now, you'll use the new Stream instance (created by the KStream.through() method) to update the rewards processor and use the stateful transform approach with the following code

Listing 4.6 Changing the rewards processor to use stateful transformation

```
KStream<String, RewardAccumulator> statefulRewardAccumulator =  
  transByCustomerStream.transformValues(() ->  
    new PurchaseRewardTransformer(rewardsStateStoreName),  
    rewardsStateStoreName);  
  
statefulRewardAccumulator.to("rewards",  
  Produced.with(stringSerde,  
    rewardAccumulatorSerde));
```

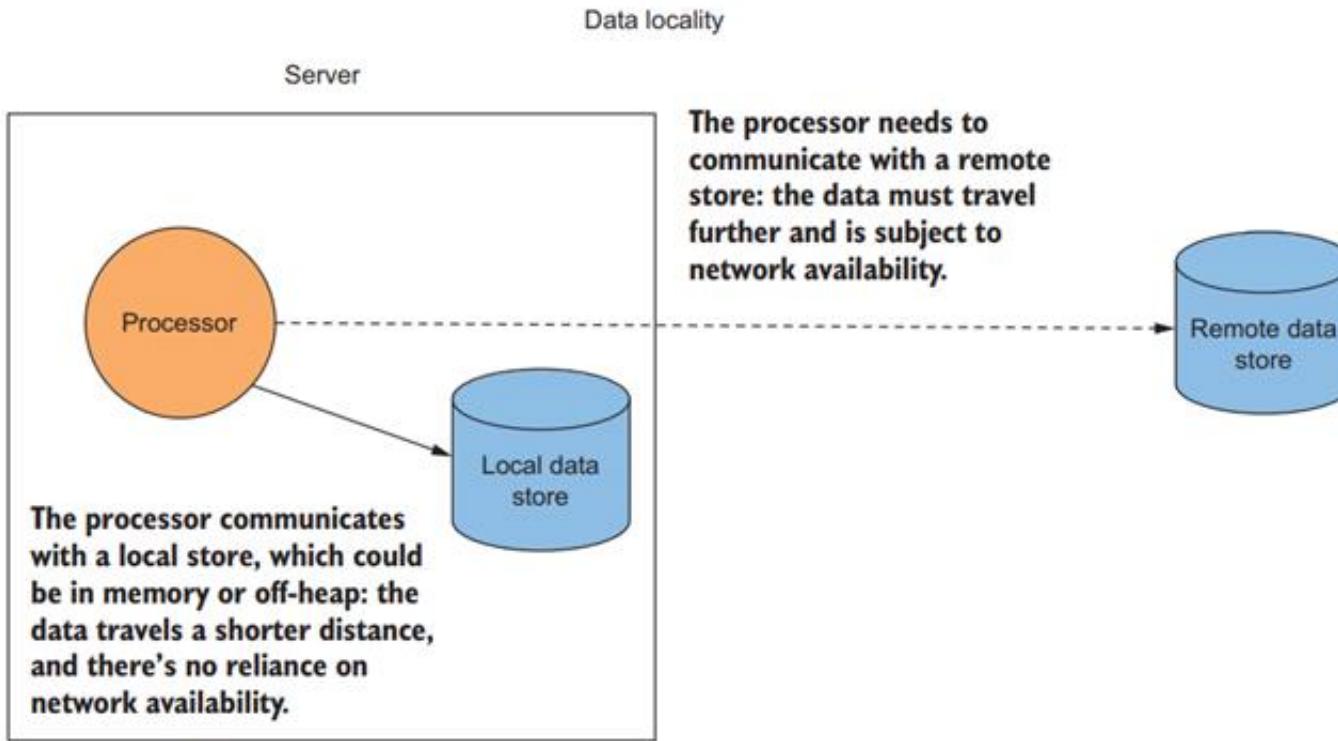
Uses a stateful transformation

Writes the results out to a topic

Using state stores for lookups and previously seen data

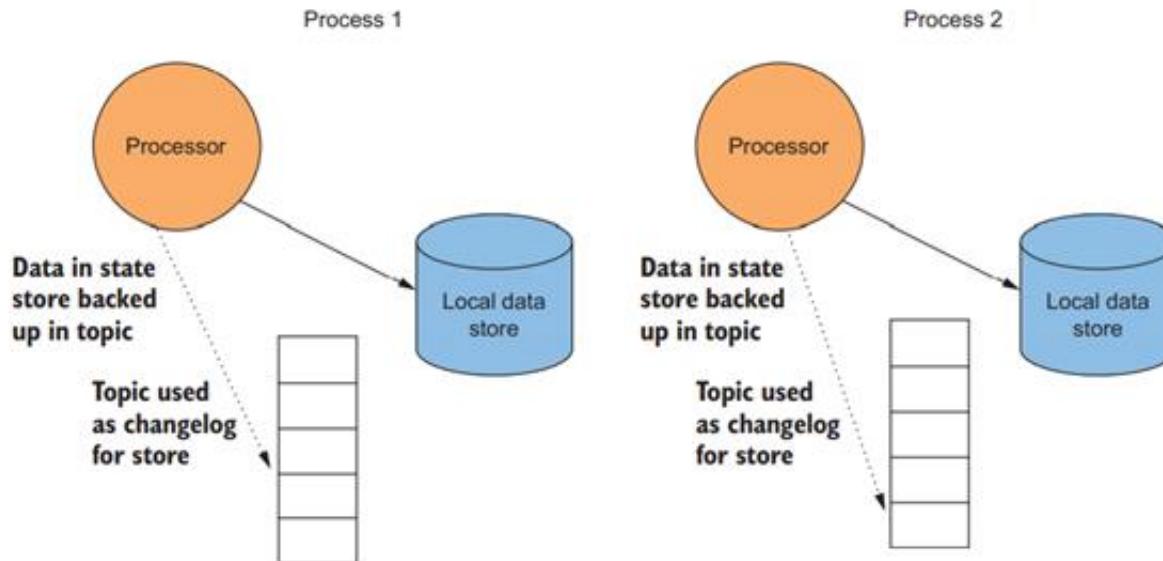
- We've discussed the need for using state with streams, and you've seen an example of one of the more basic stateful operations available in Kafka Streams.
- Before we get into more detail about using state stores in Kafka Streams, let's briefly look at two important attributes of state: data locality and failure recovery.

Data locality



Failure recovery and fault tolerance

Fault tolerance and failure recovery:
two Kafka Streams processes running on the same server



Because each process has its own local state store and a shared-nothing architecture, if either process fails, the other process will be unaffected. Also, each store has its keys/values replicated to a topic, which is used to recover values lost when a process fails or restarts.

Using state stores in Kafka Streams

- Adding a state store is a simple matter of creating a `StoreSupplier` instance with one of the static factory methods on the `Stores` class.
- There are two additional classes for customizing the state store: the `Materialized` and `StoreBuilder` classes.
- Which one you'll use depends on how you add the store to the topology

Using state stores in Kafka Streams

Listing 4.7 Adding a state store

```
String rewardsStateStoreName = "rewardsPointsStore";  
KeyValueBytesStoreSupplier storeSupplier =  
    Stores.inMemoryKeyValueStore(rewardsStateStoreName);    ← Creates the StateStore supplier  
  
StoreBuilder<KeyValueStore<String, Integer>> storeBuilder =  
    Stores.keyValueStoreBuilder(storeSupplier,  
        Serdes.String(),  
        Serdes.Integer());    ← Creates the StoreBuilder and specifies the key and value types  
  
builder.addStateStore(storeBuilder);    ← Adds the state store to the topology
```

Additional key/value store suppliers

In addition to the `Stores.inMemoryKeyValueStore` method, you can use these other static factory methods for producing store suppliers:

- `Stores.persistentKeyValueStore`
- `Stores.lruMap`
- `Stores.persistentWindowStore`
- `Stores.persistentSessionStore`

StateStore fault tolerance

- All the StateStoreSupplier types have logging enabled as a default.
- Logging, in this context, means a Kafka topic used as a changelog to back up the values in the store and provide fault tolerance.
- For example, suppose you lost a machine running Kafka Streams.

Configuring changelog topics

- The changelogs for state stores are configurable via the `withLoggingEnabled(Map config)` method.
- You can use any configuration parameters available for topics in the map.
- The configuration of changelogs for state stores is important when building a Kafka Streams application.

Configuring changelog topics

- Let's first take a look at how you can configure your changelog topic to have a retention size of 10 GB and a retention time of 2 days.

Listing 4.8 Setting changelog properties

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms", "172800000");
changeLogConfigs.put("retention.bytes", "10000000000");

// to use with a StoreBuilder
storeBuilder.withLoggingEnabled(changeLogConfigs);

// to use with Materialized
Materialized.as(Stores.inMemoryKeyValueStore("foo")
    .withLoggingEnabled(changeLogConfigs));
```

Configuring changelog topics

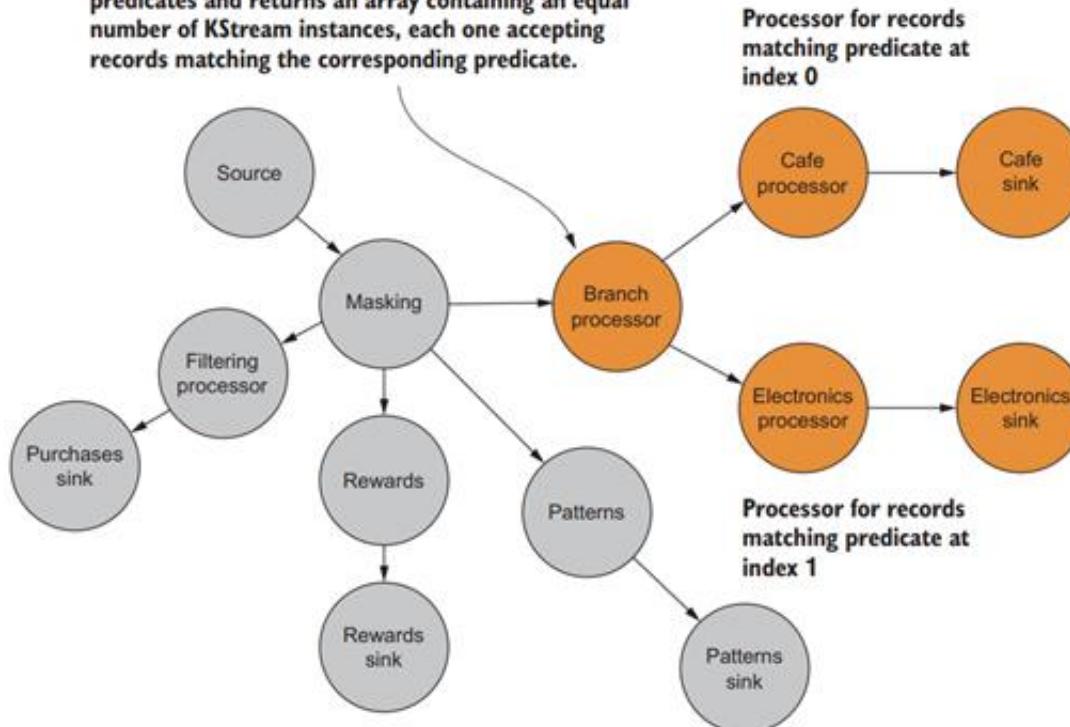
- You specify a cleanup policy of delete and compact.

Listing 4.9 Setting a cleanup policy

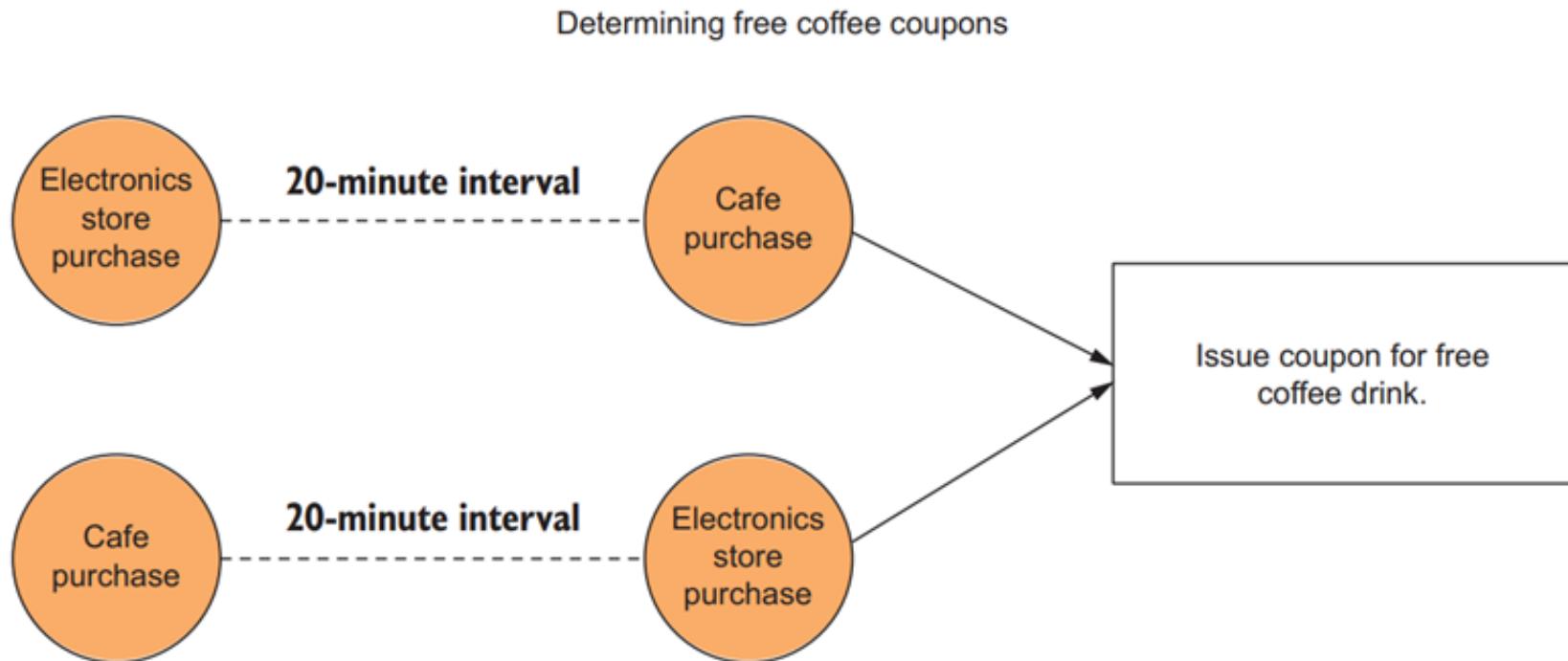
```
Map<String, String> changeLogConfigs = new HashMap<>();  
changeLogConfigs.put("retention.ms", "172800000");  
changeLogConfigs.put("retention.bytes", "10000000000");  
changeLogConfigs.put("cleanup.policy", "compact,delete");
```

Joining streams for added insight

The `KStream.branch` method takes an array of predicates and returns an array containing an equal number of `KStream` instances, each one accepting records matching the corresponding predicate.

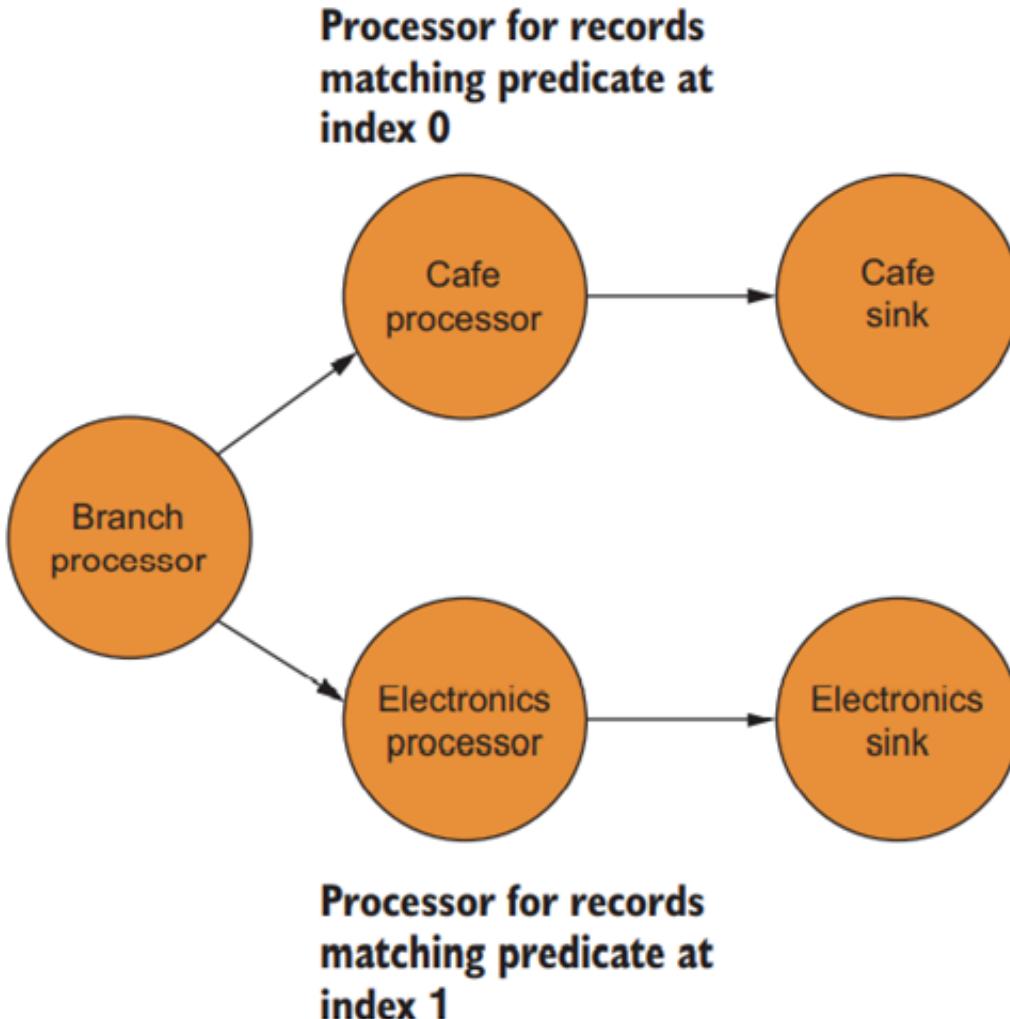


Joining streams for added insight



Data setup

This processor contains the array of predicates and returns an equal number of KStream instances, accepting records that match the given predicate.



Data setup

Listing 4.10 Branching into two streams

```
Predicate<String, Purchase> coffeePurchase = (key, purchase) ->  
    purchase.getDepartment().equalsIgnoreCase("coffee");
```

Defines the predicates for matching records

```
Predicate<String, Purchase> electronicPurchase = (key, purchase) ->  
    purchase.getDepartment().equalsIgnoreCase("electronics");
```

```
final int COFFEE_PURCHASE = 0;  
final int ELECTRONICS_PURCHASE = 1;
```

Uses labeled integers for clarity when accessing the corresponding array

```
KStream<String, Purchase>[] branchedTransactions =  
    transactionStream.branch(coffeePurchase, electronicPurchase);
```

Creates the branched stream

Generating keys containing customer IDs to perform joins

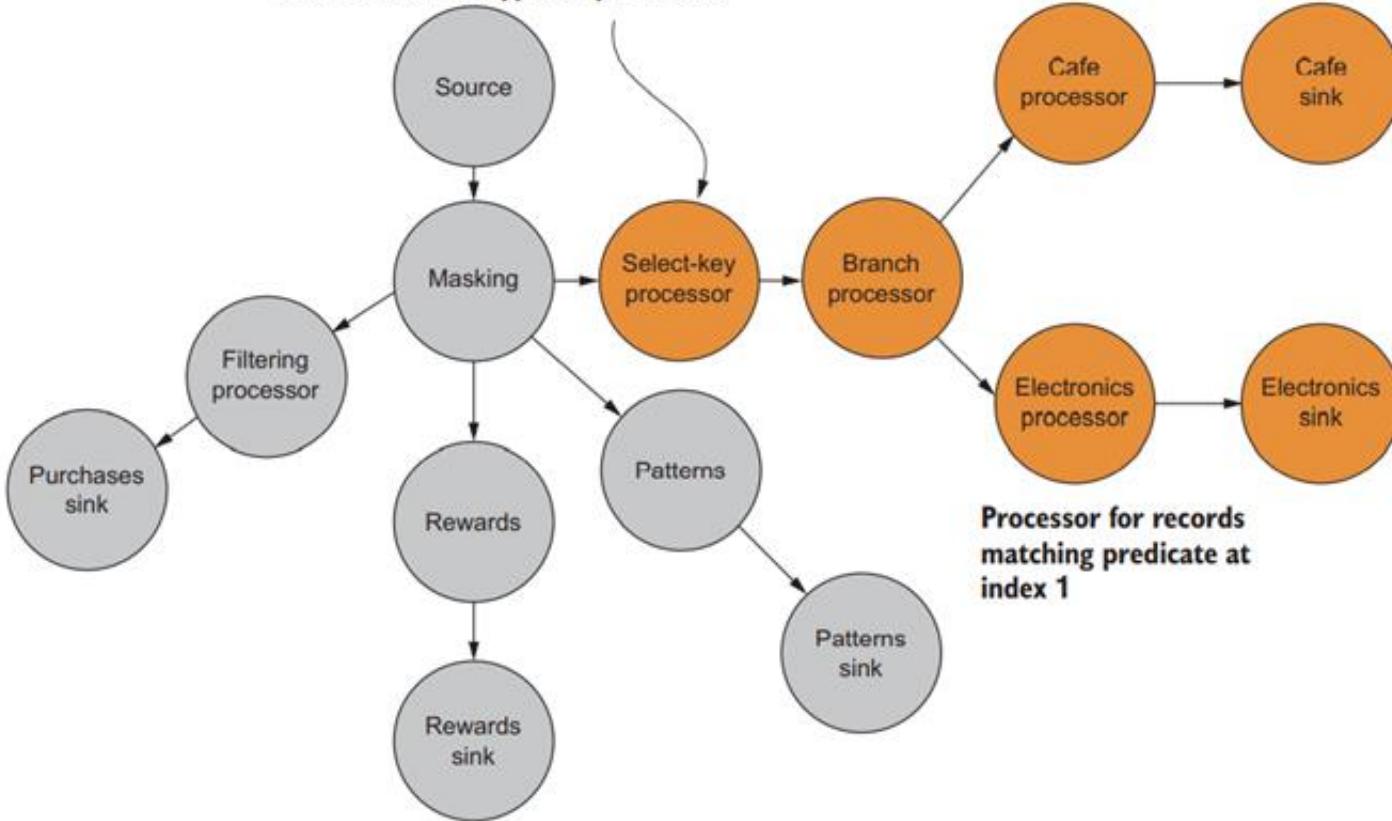
- To generate a key, you select the customer ID from the purchase data in the stream.
- To do so, you need to update the original KStream instance (`transactionStream`) and create another processing node between it and the branch node

Listing 4.11 Generating new keys

```
KStream<String, Purchase>[] branchesStream =  
  transactionStream.selectKey((k, v) ->  
    v.getCustomerId() ).branch(coffeePurchase, electronicPurchase);
```

Inserts the selectKey
processing node

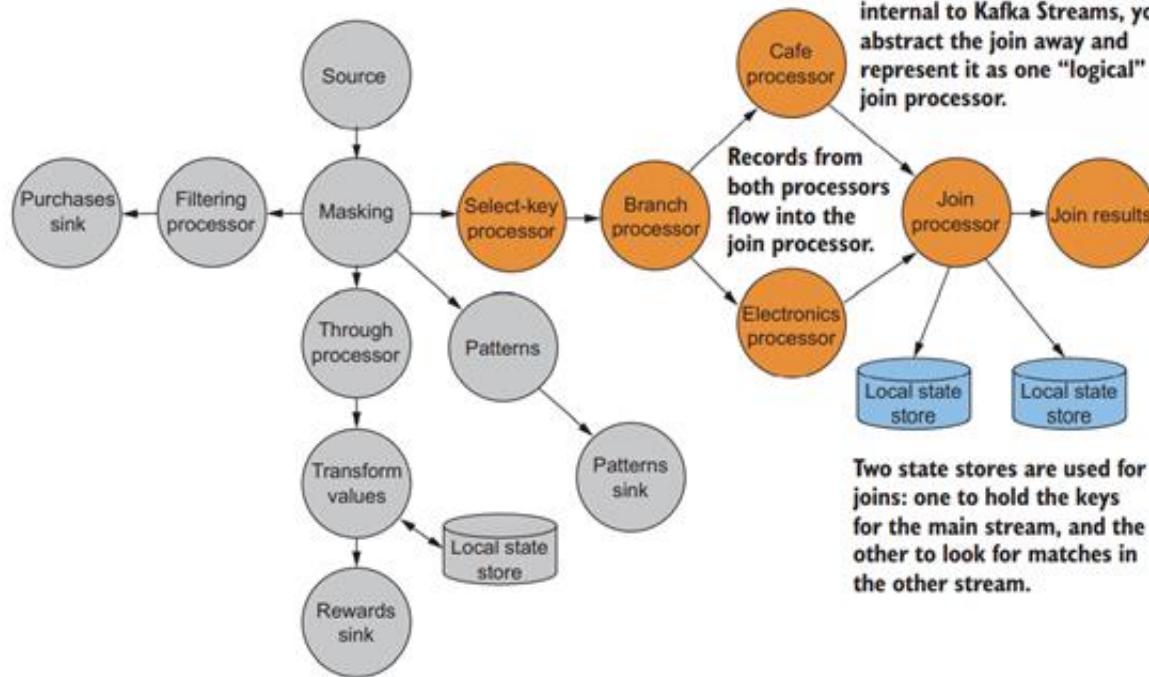
This processor is inserted to extract the customer ID from the transaction object to be used for the key. This sets up the join between the two types of purchases.



Processor for records matching predicate at index 0

Processor for records matching predicate at index 1

Constructing the join



JOINING PURCHASE RECORDS

Listing 4.12 ValueJoiner implementation

```
public class PurchaseJoiner
  ↪ implements ValueJoiner<Purchase, Purchase, CorrelatedPurchase> {

    @Override
    public CorrelatedPurchase apply(Purchase purchase, Purchase purchase2) {
        CorrelatedPurchase.Builder builder =
    ↪ CorrelatedPurchase.newBuilder();           ← Instantiates
                                              the builder
        Date purchaseDate =
    ↪ purchase != null ? purchase.getPurchaseDate() : null;

        Double price = purchase != null ? purchase.getPrice() : 0.0;
```

```

        String itemPurchased =
        => purchase != null ? purchase.getItemPurchased() : null; ← Handles a null
                                                Purchase in the case
                                                of an outer join

        Date otherPurchaseDate =
        => otherPurchase != null ? otherPurchase.getPurchaseDate() : null;

        Double otherPrice =
        => otherPurchase != null ? otherPurchase.getPrice() : 0.0;

        String otherItemPurchased =
        => otherPurchase != null ? otherPurchase.getItemPurchased() : null; ← Handles a null
                                                Purchase in the
                                                case of a left
                                                outer join

        List<String> purchasedItems = new ArrayList<>();

        if (itemPurchased != null) {
            purchasedItems.add(itemPurchased);
        }

        if (otherItemPurchased != null) {
            purchasedItems.add(otherItemPurchased);
        }

        String customerId =
        => purchase != null ? purchase.getCustomerId() : null;

        String otherCustomerId =
        => otherPurchase != null ? otherPurchase.getCustomerId() : null;

        builder.withCustomerId(customerId != null ? customerId : otherCustomer
rId)
            .withFirstPurchaseDate(purchaseDate)
            .withSecondPurchaseDate(otherPurchaseDate)
            .withItemsPurchased(purchasedItems)
            .withTotalAmount(price + otherPrice);

        return builder.build(); ← Returns the new
                                CorrelatedPurchase
                                object
    }
}

```

IMPLEMENTING THE JOIN

Listing 4.13 Using the join() method

```
KStream<String, Purchase> coffeeStream =  
↳ branchesStream[COFFEE_PURCHASE];  
KStream<String, Purchase> electronicsStream =  
↳ branchesStream[ELECTRONICS_PURCHASE];  
  
ValueJoiner<Purchase, Purchase, CorrelatedPurchase> purchaseJoiner =  
↳ new PurchaseJoiner();  
  
JoinWindows twentyMinuteWindow = JoinWindows.of(60 * 1000 * 20);  
  
KStream<String, CorrelatedPurchase> joinedKStream =  
↳ coffeeStream.join(electronicsStream,  
                     purchaseJoiner,  
                     twentyMinuteWindow,  
                     Joined.with(stringSerde,  
                               purchaseSerde,  
                               purchaseSerde));  
  
joinedKStream.print("joinedStream");
```

Extracts the branched streams

ValueJoiner instance used to perform the join

Calls the join method, triggering automatic repartitioning of coffeeStream and electronicsStream

Constructs the join

Prints the join results to the console

CO-PARTITIONING

- In order to perform a join in Kafka Streams, you need to ensure that all join participants are co-partitioned, meaning that they have the same number of partitions and are keyed by the same type.
- As a result, when you call the join() method in listing 4.13, both KStream instances will be checked to see if a repartition is required

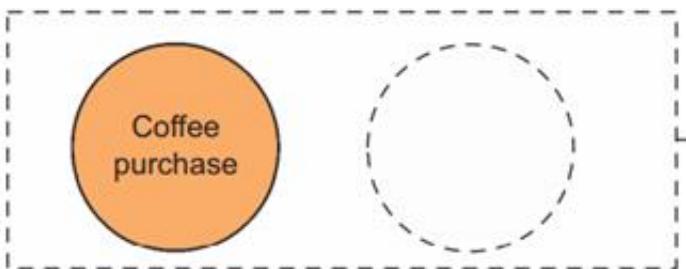
Other join options

- The join in listing 4.13 is an inner join.
- With an inner join, if either record isn't present, the join doesn't occur, and you don't emit a CorrelatedPurchase object.
- There are other options that don't require both records.

OUTER JOINS

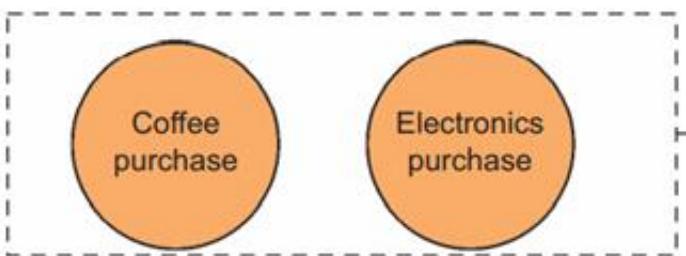
- Outer joins always output a record, but the forwarded join record may not include both of the events specified by the join.
- If either side of the join isn't present when the time window expires, an outer join sends the record that's available downstream.
- Of course, if both events are present within the window, the issued record contains both events.
- For example, if you wanted to use an outer join in listing 4.13, you'd do so like this:
`coffeeStream.outerJoin(electronicsStream,...)`

Join time window



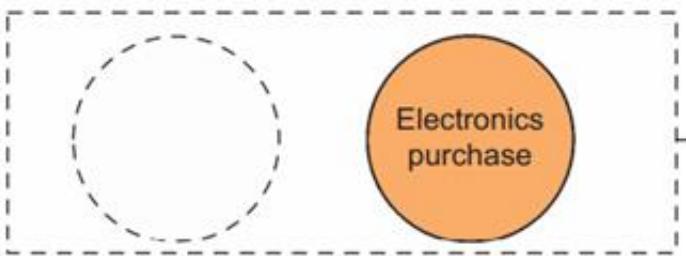
Only the calling stream's event is available in the time window, so that's the only record included.

→ (Coffee purchase, null)



Both streams' events are available in the time window, so both are included in the join record.

→ (Coffee purchase, electronics purchase)



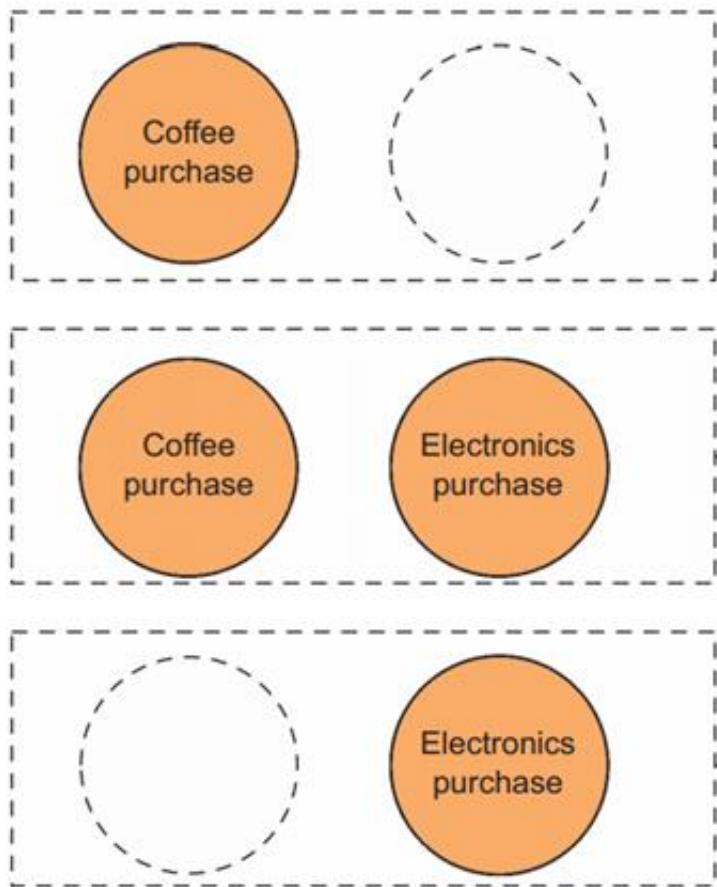
Only the other stream's event is available in the time window, so nothing is sent downstream.

→ (Null, electronics purchase)

LEFT-OUTER JOIN

- The records sent downstream from a left-outer join are similar to an outer join, with one exception.
- When the only event available in the join window is from the other stream, there's no output at all.
- If you wanted to use a left-outer join in listing 4.13, you'd do so like this:
`coffeeStream.leftJoin(electronicsStream..)`

Join time window



Only the calling stream's event is available in the time window, so that's the only record included.

→ (Coffee purchase, null)

Both streams' events are available in the time window, so both are included in the join record.

→ (Coffee purchase, electronics purchase)

Only the other stream's event is available in the time window, so nothing is sent downstream.

→ No output

Timestamps in Kafka Streams

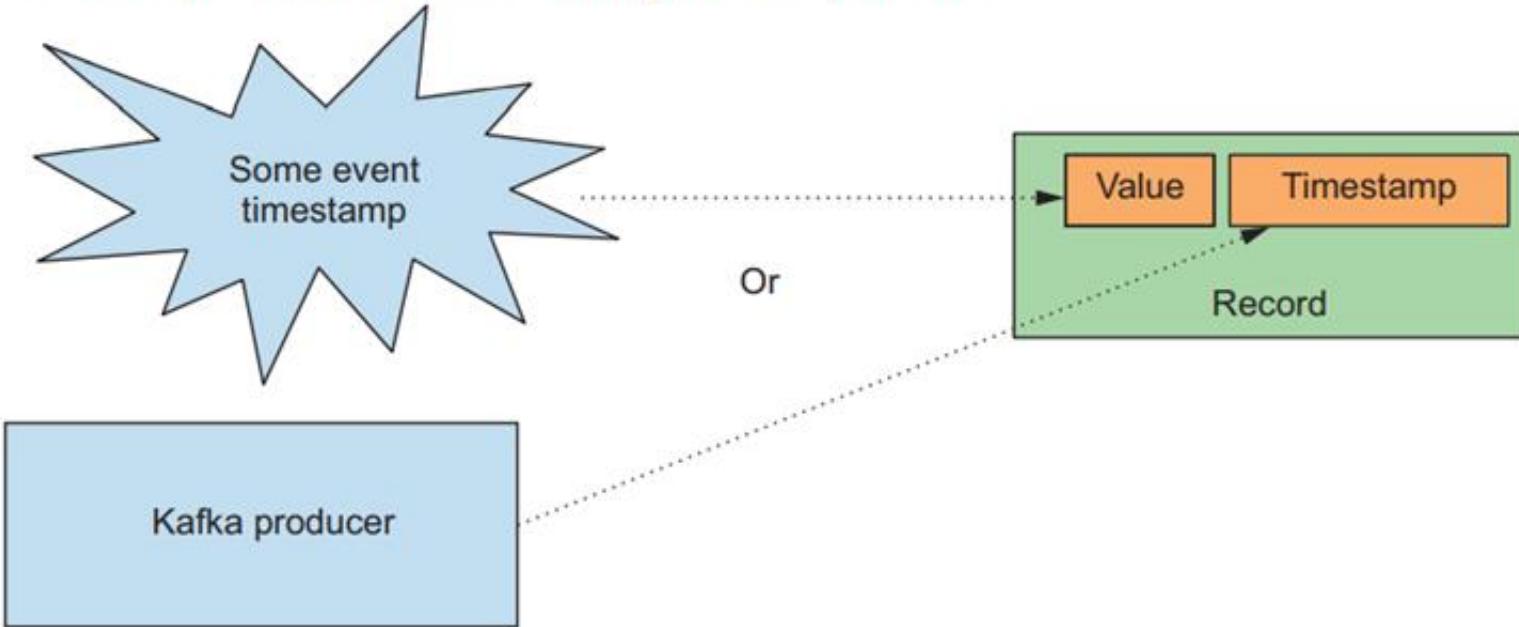
In this section, we'll discuss the use of timestamps in Kafka Streams. Timestamps play a role in key areas of Kafka Streams functionality:

- Joining streams
- Updating a changelog (KTable API)
- Deciding when the Processor.punctuate() method is triggered (Processor API)

Timestamps in Kafka Streams

Timestamp embedded in data object at time of event, or timestamp set in ProducerRecord by a Kafka producer

Event time



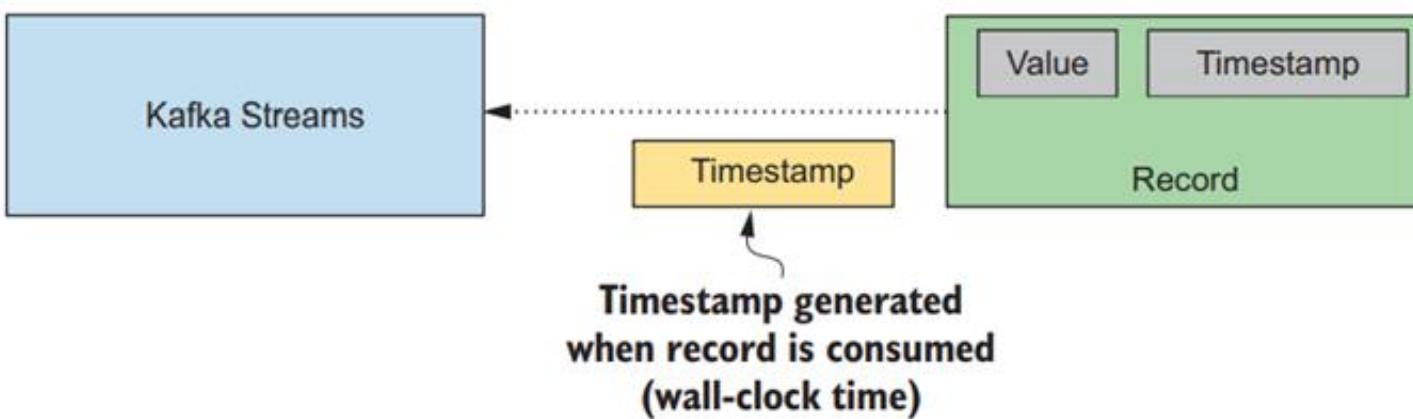
Timestamp set at time record is appended to log (topic)

Ingest time



Timestamp generated at the moment when record is consumed, ignoring timestamp embedded in data object and ConsumerRecord

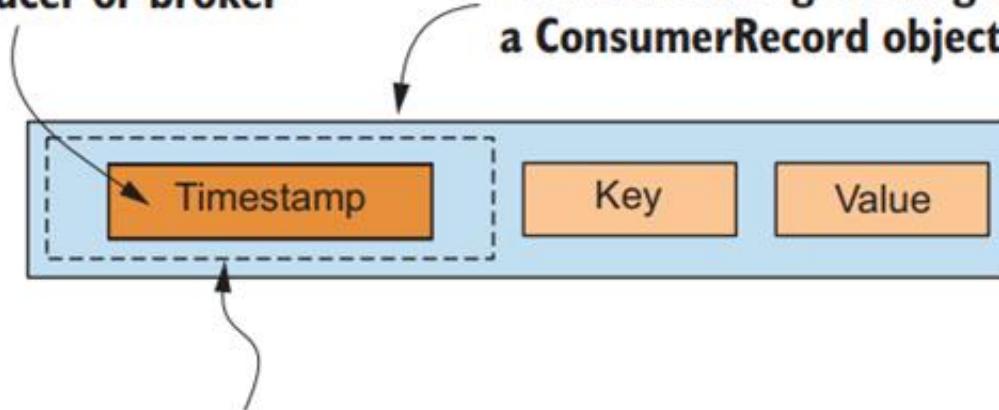
Processing time



Provided TimestampExtractor implementations

**Consumer timestamp extractor
retrieves timestamp set by
Kafka producer or broker**

**Entire enclosing rectangle represents
a ConsumerRecord object**



**Dotted rectangle represents
ConsumerRecord metadata**

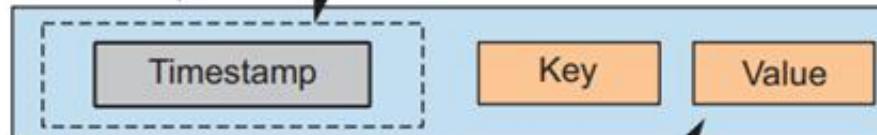
WallclockTimestampExtractor

- WallclockTimestampExtractor provides process-time semantics and doesn't extract any timestamps.
- Instead, it returns the time in milliseconds by calling the System.currentTimeMillis() method.

Custom TimestampExtractor

Entire enclosing
rectangle represents a
ConsumerRecord object

ConsumerRecord metadata



Custom TimestampExtractor knows where
to pull the timestamp from the value in a
ConsumerRecord object

Record in JSON format

```
{ "recordType" = "purchase",  
  "amount" = 500.00,  
  "timestamp" = 1502041889179 }
```

Custom TimestampExtractor

Listing 4.14 Custom TimestampExtractor

```
public class TransactionTimestampExtractor implements TimestampExtractor {  
  
    @Override  
    public long extract(ConsumerRecord<Object, Object> record,  
    long previousTimestamp) {  
        Purchase purchaseTransaction = (Purchase) record.value();  
        return purchaseTransaction.getPurchaseDate().getTime();  
    }  
}
```

Retrieves the Purchase object from
the key/value pair sent to Kafka

Returns the timestamp
recorded at the point of sale

Specifying a TimestampExtractor

- If no property is set, the default setting is FailOnInvalidTimestamp.class.
- For example, the following code would configure the TransactionTimestampExtractor via properties when setting up the application:

```
props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,  
        TransactionTimestampExtractor.class);
```

Specifying a TimestampExtractor

The second option is to provide a `TimestampExtractor` instance via a `Consumed` object:

```
Consumed.with(Serdes.String(), purchaseSerde)
    .withTimestampExtractor(new TransactionTimestampExtractor()))
```

Summary

- Stream processing needs state. Sometimes events can stand on their own, but usually you'll need additional information to make good decisions.
- Kafka Streams provides useful abstractions for stateful transformations, including joins

COMPLETE LAB 4

5. The KTable API



The KTable API

This lesson covers

- Defining the relationship between streams and tables
- Updating records, and the KTable abstraction
- Aggregations, and windowing and joining
- KStreams and KTables
- Global KTables
- Queryable state stores

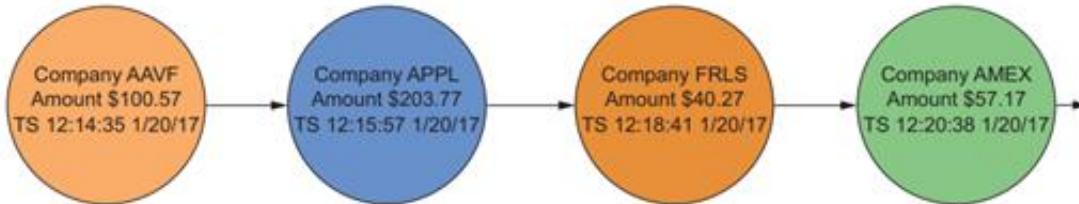
The relationship between streams and tables

The record stream

- Suppose you want to view a series of stock price updates.
- You can recast the generic marble diagram from lesson 1 to look like next figure.
- You can see that each stock price quote is a discrete event, and they aren't related to each other.
- Even if the same company accounts for many price quotes, you're only looking at them one at a time.

Time
→

Each circle on the line represents a publicly traded stock's share price adjusting to market forces.



Imagine that you are observing a stock ticker displaying updated share prices in real time.

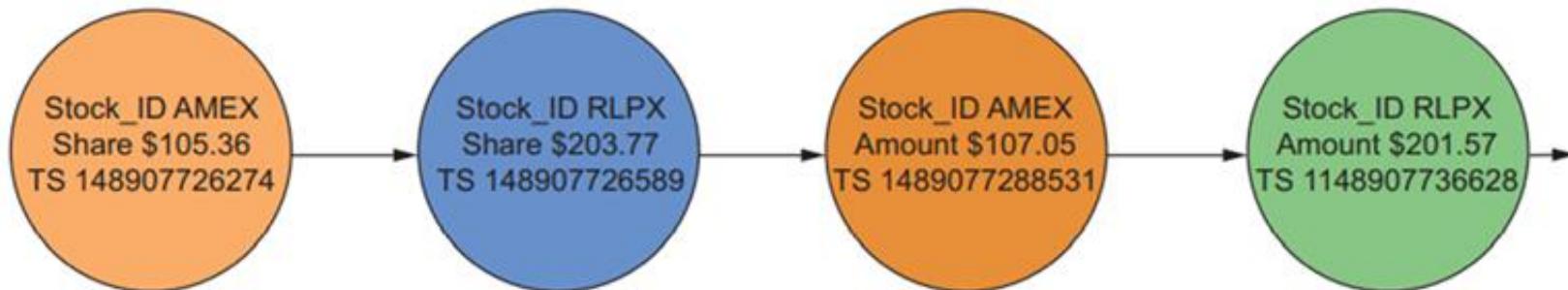
Figure 5.1 A marble diagram for an unbounded stream of stock quotes

Key ↓ Value ↗ ↘

Stock_ID	Timestamp	Share_Price
ABVF	32225544289	105.36
APPL	32225544254	333.66

The rows from table above can be recast as key/value pairs.
For example, the first row in the table can be converted
to this key/value pair:

{key:{stockid:1235588}, value:{ts:32225544289, price:105.36}}

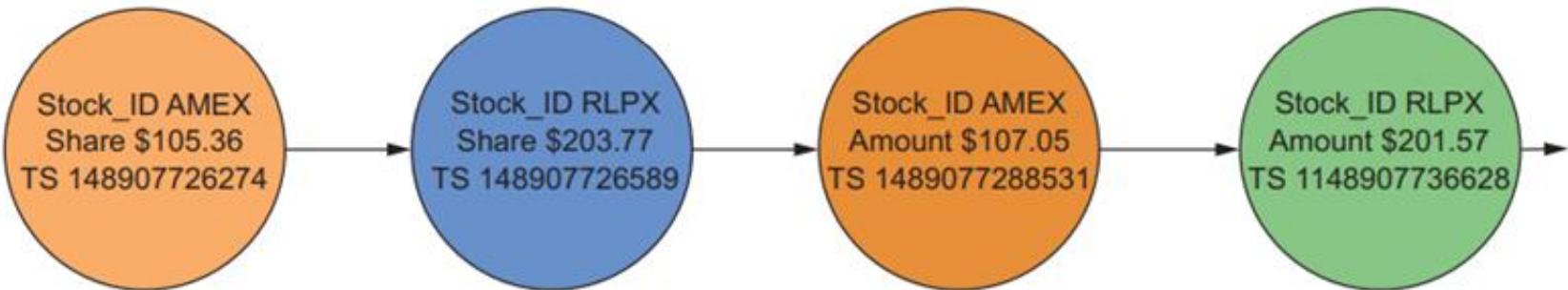


Key	Stock_ID	Timestamp	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

This shows the relationship between events and inserts into a database. Even though it's stock prices for two companies, it counts as four events because we're considering each item on the stream as a singular event.

As a result, each event is an insert, and we increment the key by one for each insert into the table.

With that in mind, each event is a new, independent record or insert into a database table.



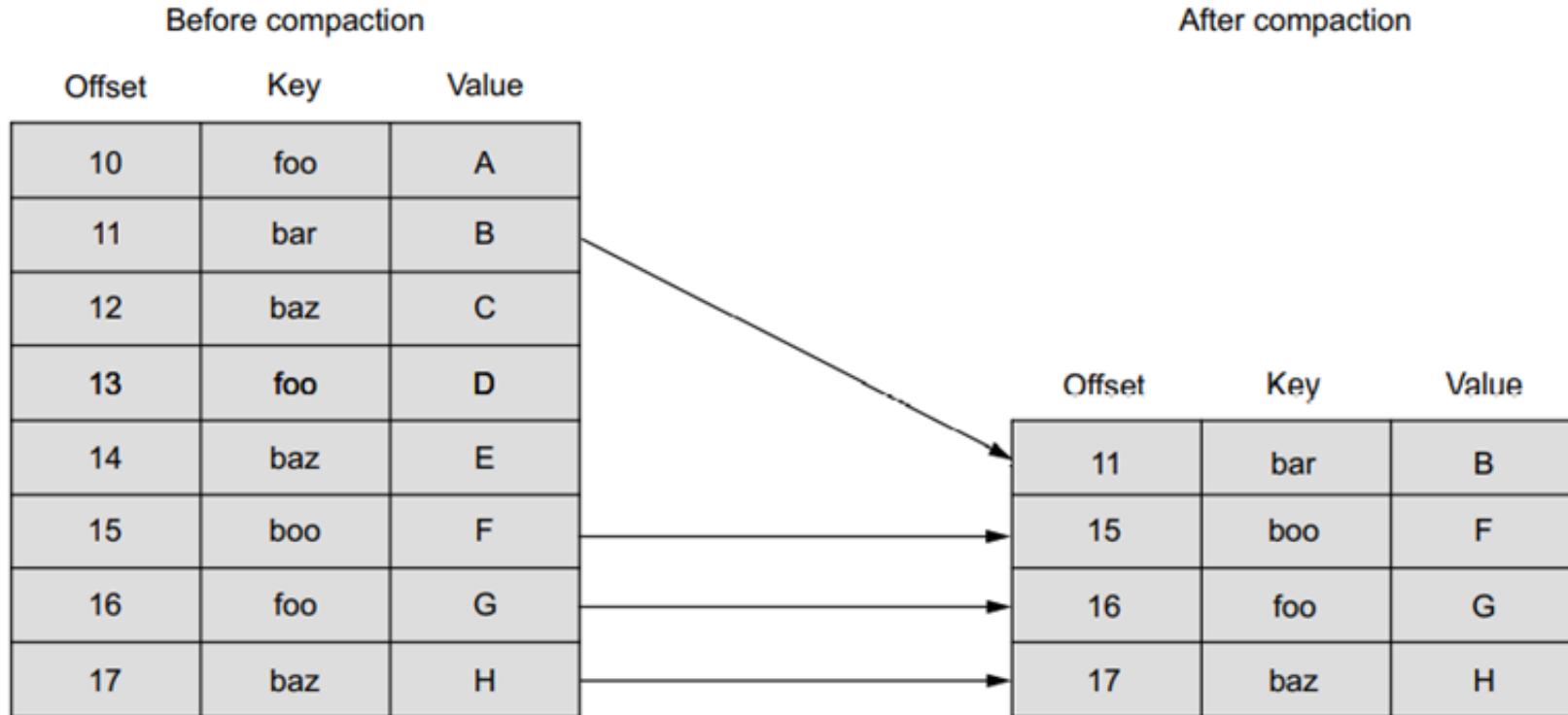
Stock_ID	Timestamp	Share_Price
AMEX	148907726274	105.36
RLPX	148907726589	203.77
AMEX	1489077288531	107.05
RLPX	148907736628	201.57

The previous records for these stocks have been overwritten with updates.

Latest records from event stream

If you use the stock ID as a primary key, subsequent events with the same key are updates in a changelog. In this case, you only have two records, one per company. Although more records can arrive for the same companies, the records won't accumulate.

Updates to records or the changelog



Event streams vs. update streams

- We'll use the KStream and the KTable to drive our comparison of event streams versus update streams.
- We'll do this by running a simple stock ticker application that writes the current share price for three (fictitious!) companies.
- It will produce three iterations of stock quotes for a total of nine records.

A simple stock ticker for three fictitious companies with a data generator producing three updates for the stocks. The KStream printed all records as they were received.

The KTable only printed the last batch of records because they were the latest updates for the given stock symbol.

Here are all three events/records for the KStream.

```
Initializing the producer
Producer initialized
KTable vs KStream output started
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.25, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.19, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=91.97, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.74, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.78, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.53, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
[Stocks-KTable]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KTable]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KTable]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
Shutting down the Kafka Streams Application now
Shutting down data generation
```

Here is the last update record for the KTable.

As expected, the values for the last KStream event and KTable update are the same.

Event streams vs. update streams

Listing 5.1 KTable and KStream printing to the console

```
KTable<String, StockTickerData> stockTickerTable =  
    builder.table(STOCK_TICKER_TABLE_TOPIC);           ← Creates the  
KStream<String, StockTickerData> stockTickerStream =  
    builder.stream(STOCK_TICKER_STREAM_TOPIC);          ← Creates the  
  
stockTickerTable.toStream()  
    .print(Printed.<String, StockTickerData>toSysOut()  
    .withLabel("Stocks-KTable"));                         ← KTable prints results  
                                                       to the console  
  
stockTickerStream  
    .print(Printed.<String, StockTickerData>toSysOut()  
    .withLabel("Stocks-KStream"));                        ← KStream prints results  
                                                       to the console
```

Record updates and KTable configuration

To figure out how the KTable functions, we should ask the following two questions:

- Where are records stored?
- How does a KTable make the determination to emit records?

Record updates and KTable configuration

- To answer the first question, let's look at the line that creates the KTable:

```
builder.table(STOCK_TICKER_TABLE_TOPIC);
```

Setting cache buffering size

- The KTable cache serves to deduplicate updates to records with the same key.
- This deduplication allows child nodes to receive only the most recent update instead of all updates, reducing the amount of data processed.

Incoming stock ticker record

YERB	105.36
------	--------

As records come in, they are also placed in the cache, with new records replacing older ones.

Cache

YERB	105.24
NDLE	33.56
YERB	105.36

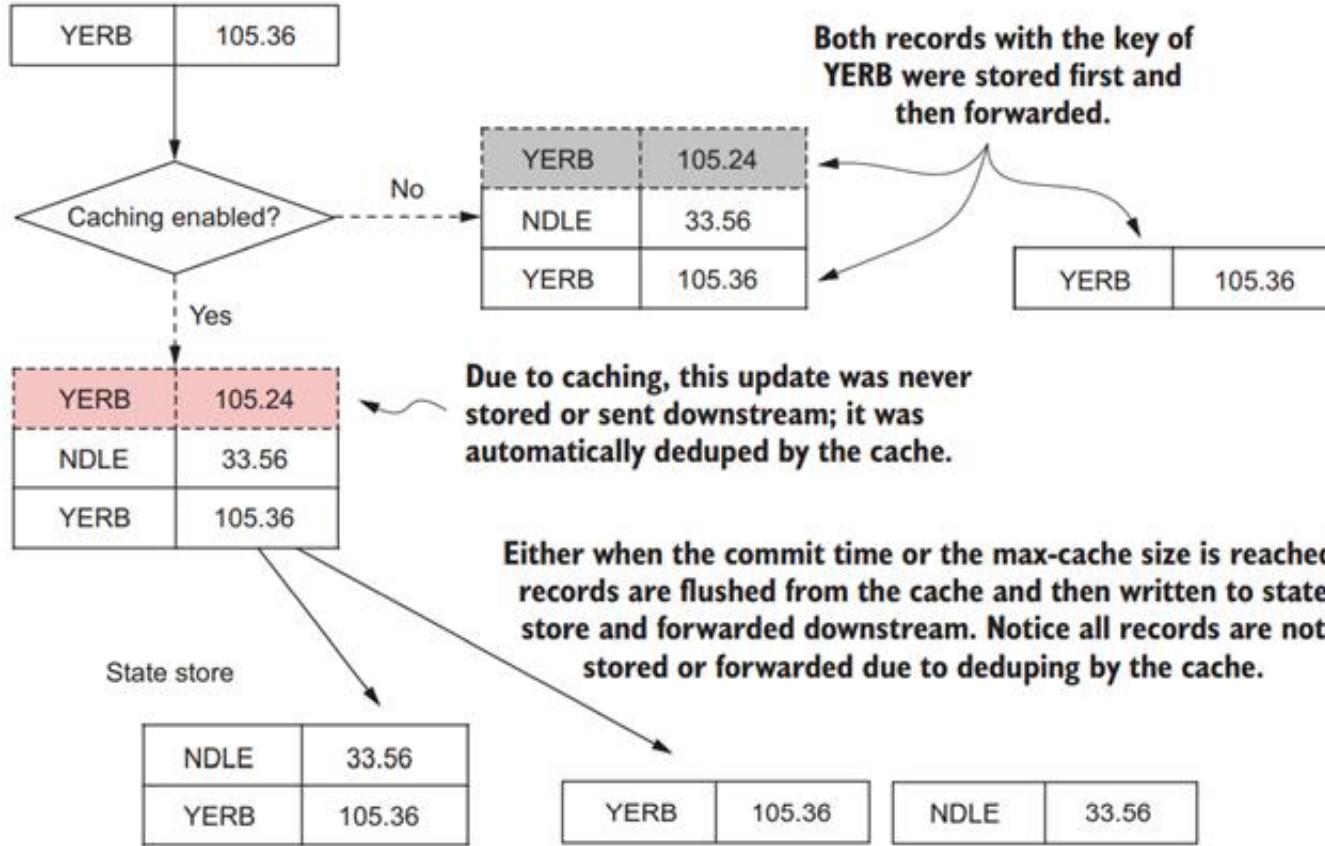
KTable caching deduplicates updates to records with the same key, preventing a flood of consecutive updates to child nodes of the KTable in the topology.

Setting the commit interval

- The other setting is the commit.interval.ms parameter.
- The commit interval specifies how often (in milliseconds) the state of a processor should be saved.
- When the state of the processor is saved (committing), it forces a cache flush, sending the latest updated, deduplicated records downstream.
- In the full caching workflow (next figure), you can see two forces at play when it comes to sending records downstream.

Incoming stock ticker record

With caching disabled (setting cache.max.bytes.buffering= 0), incoming updates are immediately written to the state store and sent downstream.



Aggregations and windowing operations

In this section, we'll move on to cover some of the most potent parts of Kafka Streams. So far, we've looked at several aspects of Kafka Streams:

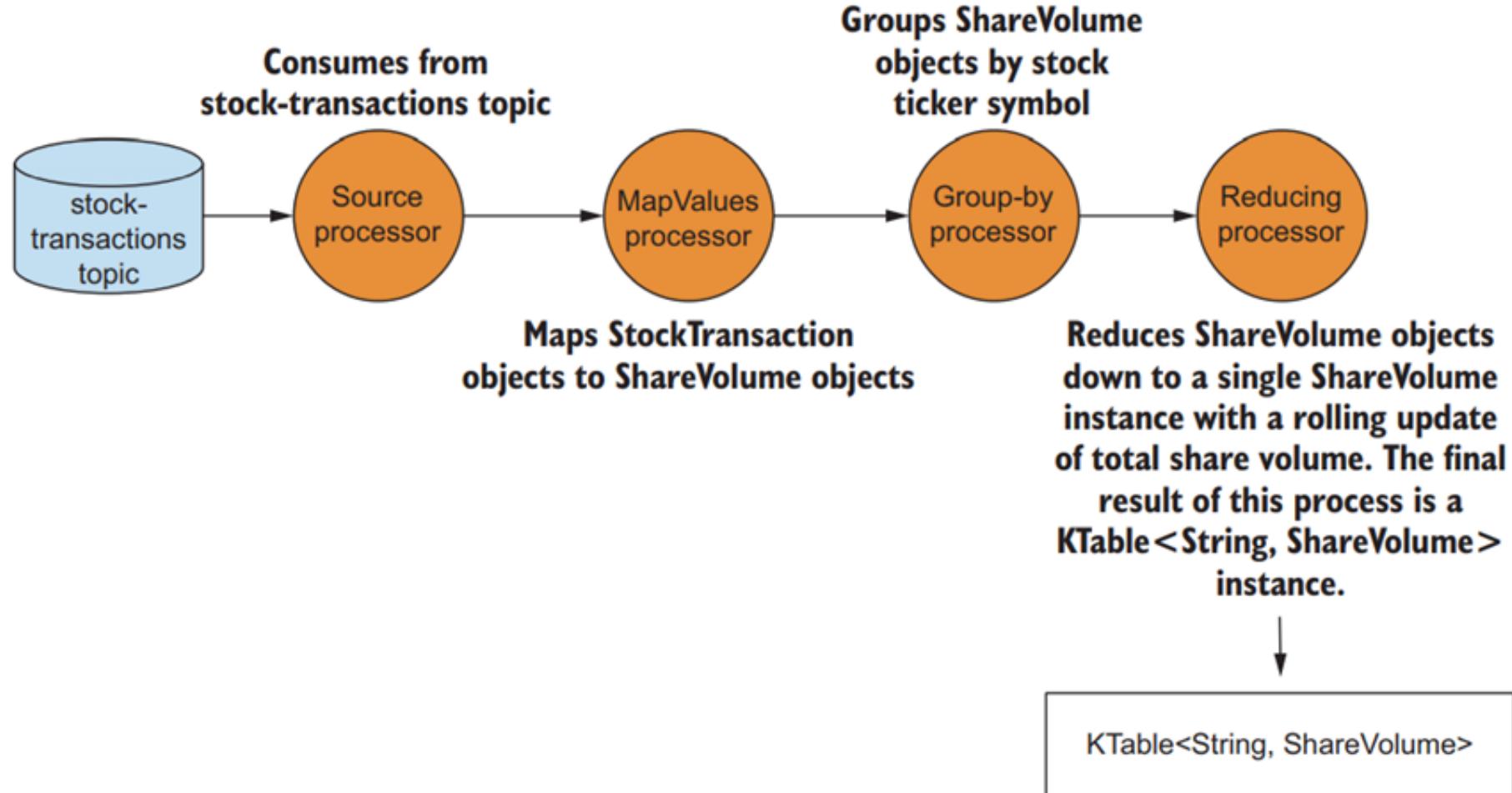
- How to set up a processing topology
- How to use state in your streaming application
- How to perform joins between streams
- The difference between an event stream (KStream) and an update stream (Ktable)

Aggregating share volume by industry

Aggregation and grouping are necessary tools when you're working with streaming data. Reviewing single records as they arrive is often not enough. To gain any insight, you'll need grouping and combining of some sort

To do this aggregation, a few steps will be required to set up the data in the correct format. At a high level, these are the steps:

1. Create a source from a topic publishing raw stock-trade information.
2. Group ShareVolume by its ticker symbol



Aggregating share volume by industry

Listing 5.2 Source for the map-reduce of stock transactions

```
KTable<String, ShareVolume> shareVolume =  
    builder.stream(STOCK_TRANSACTIONS_TOPIC,  
                  Consumed.with(stringSerde, stockTransactionSerde)  
    .withOffsetResetPolicy(EARLIEST))  
    .mapValues(st -> ShareVolume.newBuilder(st).build())  
    .groupBy((k, v) -> v.getSymbol(),  
             Serialized.with(stringSerde, shareVolumeSerde))  
    .reduce(ShareVolume::reduce);
```

The source processor consumes from a topic.

Maps StockTransaction objects to ShareVolume objects

Reduces ShareVolume objects to contain a rolling aggregate of share volume

Groups the ShareVolume objects by their stock ticker symbol

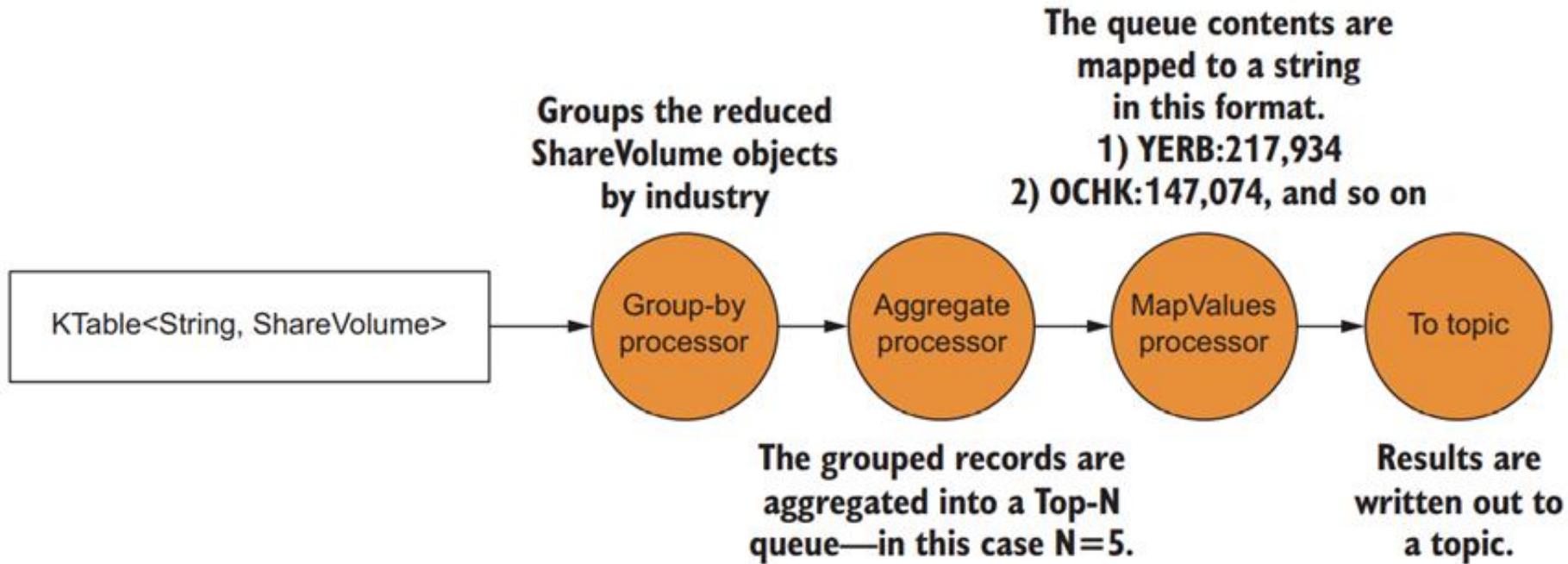
Aggregating share volume by industry

- It's clear what mapValues and groupBy are doing, but
- let's look into the sum() method

Listing 5.3 The ShareVolume.sum method

```
public static ShareVolume sum(ShareVolume csv1, ShareVolume csv2) {  
    Builder builder = newBuilder(csv1);  
    builder.shares = csv1.shares + csv2.shares;  
    return builder.build();  
}  
  
Calls build and returns a  
new ShareVolume object  
Sets the number of  
shares to the total of  
both ShareVolume  
objects  
Uses a Builder  
for a copy  
constructor
```

Aggregating share volume by industry



Aggregating share volume by industry

Listing 5.4 KTable groupBy and aggregation

```
Comparator<ShareVolume> comparator =  
  ➔ (sv1, sv2) -> sv2.getShares() - sv1.getShares()
```

The Aggregate initializer is
an instance of the
`FixedSizePriorityQueue`
class (for demonstration
purposes only!).

```
FixedSizePriorityQueue<ShareVolume> fixedQueue =  
  ➔ new FixedSizePriorityQueue<>(comparator, 5);
```

Groups by industry and
provides required serdes

```
shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),  
  ➔ Serialized.with(stringSerde, shareVolumeSerde))  
  .aggregate(() -> fixedQueue,
```

Aggregate
adder adds
new updates

```
    ➔ (k, v, agg) -> agg.add(v),  
    ➔ (k, v, agg) -> agg.remove(v),  
    Materialized.with(stringSerde, fixedSizePriorityQueueSerde))
```

Aggregate remover
removes old updates

Serde for the aggregator

Aggregating share volume by industry

```
.mapValues(valueMapper)
.toStream().peek((k, v) ->
  LOG.info("Stock volume by industry {} {}", k, v))
.to("stock-volume-by-company", Produced.with(stringSerde,
stringSerde));
```

Writes the results to
the stock-volume-by-
company topic

Calls `toStream()` to log the
results (to the console) via
the `peek` method

ValueMapper
instance converts
aggregator to a
string that's used
for reporting

Aggregating share volume by industry

You've now learned how to do two important things:

- Group values in a KTable by a common key
- Perform useful operations like reducing and aggregation with those grouped values

Windowing operations

- Sometimes, you'll want an ongoing aggregation and reduction of results like this.
- At other times, you'll only want to perform operations for a given time range.
- For example, how many stock transactions have involved a particular company in the last 10 minutes?
- How many users have clicked to view a new advertisement in the last 15 minutes?

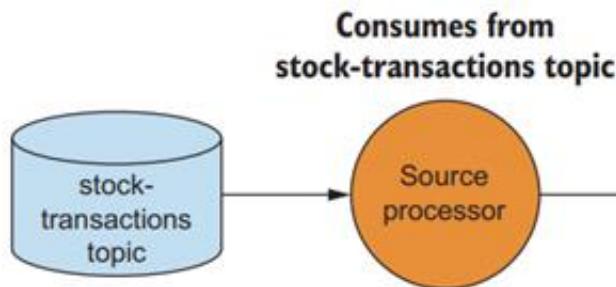
COUNTING STOCK TRANSACTIONS BY CUSTOMER

- In the next example, you'll track stock transactions for a handful of traders.
- These could be large institutional traders or financially savvy individuals & There are two likely reasons for doing this tracking.
- One reason is that you may want to see where the market leaders are buying and selling

COUNTING STOCK TRANSACTIONS BY CUSTOMER

Here are the steps to do this tracking:

1. Create a stream to read from the stock-transactions topic.
2. Group incoming records by the customer ID and stock ticker symbol. The groupBy call returns a KGroupedStream instance.
3. Use the KGroupedStream.windowedBy method to return a windowed stream, so you can perform some sort of windowed aggregation.

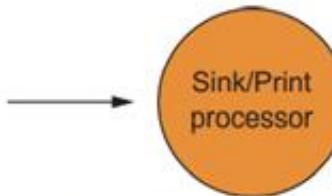


Consumes from stock-transactions topic

Counts number of transactions by TransactionSummary (customer ID and stock symbol). You'll use a windowed approach to contain the counts. The window could be a tumbling, hopping, or session window.

Groups StockTransactions by customer ID and stock ticker symbol. These are encapsulated in a TransactionSummary object.

The final object returned from the count processor is a KTable<Windowed<TransactionSummary>, Long>.



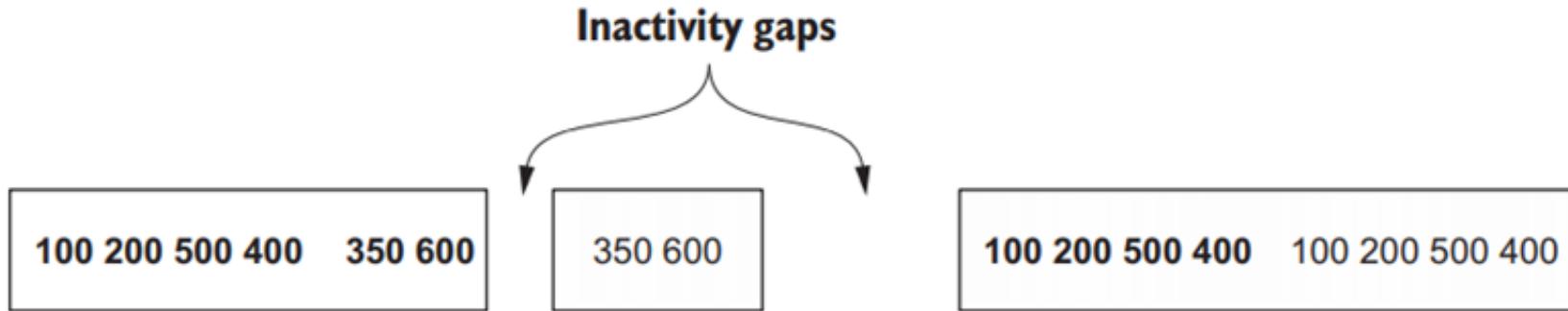
You'll write out the results to a topic (or print to the console during development).

WINDOW TYPES

In Kafka Streams, three types of windows are available:

- Session windows
- Tumbling windows
- Sliding/hopping windows

SESSION WINDOWS



There's a small inactivity gap here, so these sessions would probably be merged into one larger session.

This inactivity gap is large, so new events go into a separate session.

Session windows are different because they aren't strictly bound by time but represent periods of activity. Specified inactivity gaps demarcate the sessions.

USING SESSION WINDOWS TO TRACK STOCK TRANSACTIONS

Listing 5.5 Tracking stock transactions with session windows

```
Serde<String> stringSerde = Serdes.String();
Serde<StockTransaction> transactionSerde =
  StreamsSerdes.StockTransactionSerde();

Serde<TransactionSummary> transactionKeySerde =
  StreamsSerdes.TransactionSummarySerde();
```

Creates the stream from the STOCK_TRANSACTIONS_TOPIC (a String constant). Uses the offset-reset-strategy enum of LATEST for this stream.

```
long twentySeconds = 1000 * 20;
long fifteenMinutes = 1000 * 60 * 15;
KTable<Windowed<TransactionSummary>, Long>
  customerTransactionCounts =
  builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
  transactionSerde)
  .withOffsetResetPolicy(LATEST))
  .groupBy((noKey, transaction) ->
  TransactionSummary.from(transaction),
  Serialized.with(transactionKeySerde, transactionSerde))
  .windowedBy(SessionWindows.with(twentySeconds))
  .until(fifteenMinutes)).count();
```

KTable resulting from the groupBy and count calls

Groups records by customer ID and stock symbol, which are stored in the TransactionSummary object.

Windows the groups with SessionWindow with an inactivity time of 20 seconds and a retention time of 15 minutes. Then performs an aggregation as a count.

```
customerTransactionCounts.toStream()
  .print(Printed.<Windowed<TransactionSummary>, Long>.toSysOut())
  .withLabel("Customer Transactions Counts");
```

Converts the KTable output to a KStream and prints the result to the console

USING SESSION WINDOWS

- The code in listing 5.5 does a count over session windows.
- Figure breaks it down

The `with` call creates
the inactivity gap of
20 seconds.

The `until` method creates
the retention period—
15 minutes, in this case.

```
SessionWindows.with(twentySeconds).until(fifteenMinutes)
```

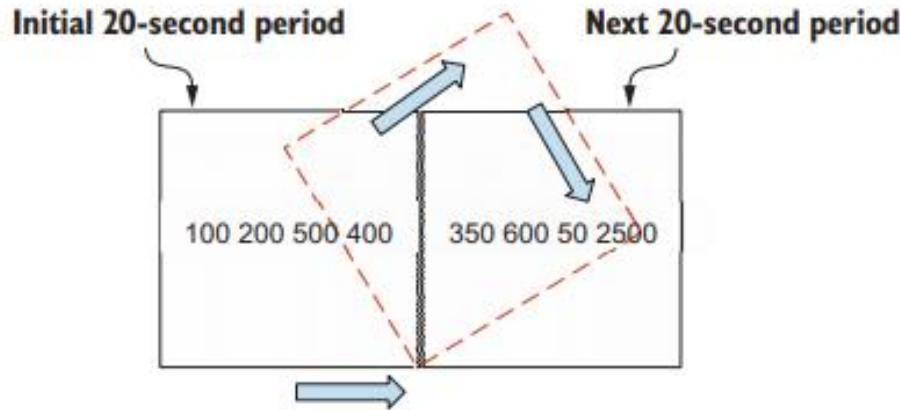
USING SESSION WINDOWS

- Let's walk through a few records from the count method to see sessions in action: see table

Arrival order	Key	Timestamp
1	{123-345-654,FFBE}	00:00:00
2	{123-345-654,FFBE}	00:00:15
3	{123-345-654,FFBE}	00:00:50
4	{123-345-654,FFBE}	00:00:05

TUMBLING WINDOWS

The current time period “tumbles” (represented by the dashed box) into the next time period completely with no overlap.



The box on the left is the first 20-second window. After 20 seconds, it “tumbles” over or updates to capture events in a new 20-second period.

There is no overlapping of events. The first event window contains [100, 200, 500, 400] and the second event window contains [350, 600, 50, 2500].

TUMBLING WINDOWS

Listing 5.6 Using tumbling windows to count user transactions

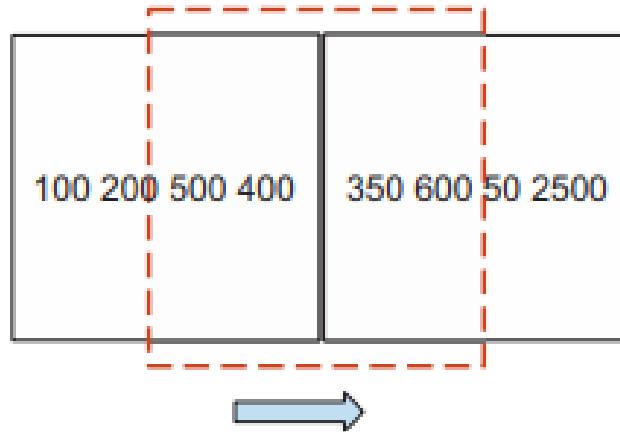
```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
  builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
                           transactionSerde)  
  .withOffsetResetPolicy(LATEST))  
    .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),  
  Serialized.with(transactionKeySerde, transactionSerde))  
    .windowedBy(TimeWindows.of(twentySeconds)).count();
```

Specifies a tumbling window of 20 seconds

SLIDING OR HOPPING WINDOWS

- Sliding or hopping windows are like tumbling windows but with a small difference.
- A sliding window doesn't wait the entire time before launching another window to process recent events.
- Sliding windows perform a new calculation after waiting for an interval smaller than the duration of the entire window.

SLIDING OR HOPPING WINDOWS



The box on the left is the first 20-second window, but the window “slides” over or updates after 5 seconds to start a new window. Now you see an overlapping of events. Window 1 contains [100, 200, 500, 400], window 2 contains [500, 400, 350, 600], and window 3 is [350, 600, 50, 2500].

SLIDING OR HOPPING WINDOWS

- Here's how to specify hopping windows with code

Listing 5.7 Specifying hopping windows to count transactions

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
  builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
  transactionSerde)  
  .withOffsetResetPolicy(LATEST))  
  .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),  
  Serialized.with(transactionKeySerde, transactionSerde))  
  .windowedBy(TimeWindows.of(twentySeconds)  
  .advanceBy(fiveSeconds).until(fifteenMinutes)).count();
```

← **Uses a hopping window of 20 seconds, advancing every 5 seconds**

Joining KStreams and KTables

Let's take the stock transaction counts and join them with financial news from relevant industry sectors. Here are the steps to make this happen with the existing code:

1. Convert the KTable of stock transaction counts into a KStream where you change the key to the industry of the count by ticker symbol.
2. Create a KTable that reads from a topic of financial news. The new KTable will be categorized by industry.
3. Join the news updates with the stock transaction counts by industry

CONVERTING THE KTABLE TO A KSTREAM

To do the KTable-to-KStream conversion, you'll take the following steps:

1. Call the KTable.toStream() method.
1. Use the KStream.map call to change the key to the industry name, and extract the TransactionSummary object from the Windowed instance.

Joining KStreams and KTables

- Steps are chained together in the following manner

Listing 5.8 Converting a KTable to a KStream

Extracts the TransactionSummary object from the Windowed instance

```
KStream<String, TransactionSummary> countStream =  
  customerTransactionCounts.toStream().map((window, count) -> {  
    TransactionSummary transactionSummary = window.key();  
    String newKey = transactionSummary.getIndustry();  
    transactionSummary.setSummaryCount(count);  
    return KeyValue.pair(newKey, transactionSummary);  
  });
```

Sets the key to the industry segment of the stock purchase

Calls toStream, immediately followed by the map call

Returns the new KeyValue pair for the KStream

Takes the count value from the aggregation and places it in the TransactionSummary object

CREATING THE FINANCIAL NEWS KTABLE

- Fortunately, creating the KTable involves just one line of code.

Listing 5.9 KTable for financial news

```
KTable<String, String> financialNews =  
    builder.table( "financial-news", Consumed.with(EARLIEST)) ;
```

Creates the KTable with the EARLIEST
enum, topic financial-news

JOINING NEWS UPDATES WITH TRANSACTION COUNTS

- Setting up the join is very simple.
- You'll use a left join, in case there's no financial news for the industry involved in the transaction

Listing 5.10 Setting up the join between the KStream and KTable

```
ValueJoiner<TransactionSummary, String, String> valueJoiner =  
    ➔ (txacct, news) ->  
    ➔ String.format("%d shares purchased %s related news [%s]",  
    ➔ txacct.getSummaryCount(), txacct.getStockTicker(), news);  
  
KStream<String, String> joined =  
    ➔ countStream.leftJoin(financialNews, valueJoiner,  
    ➔ Joined.with(stringSerde, transactionKeySerde, stringSerde));  
  
joined.print(Printed.<String, String>toSysOut()  
    ➔ .withLabel("Transactions and News"));
```

Prints results to the console (in production this would be written to a topic with a `to("topic-name")` call)

ValueJoiner combines the values from the join result.

The leftJoin statement for the countStream KStream and the financial news KTable, providing serdes with a Joined instance

GlobalKTables

- We've established the need to enrich or add context to our event streams.
- You've also seen joins between two KStreams in lesson 4, and the previous section demonstrated a join between a KStream and a KTable.
- In all of these cases, when you map the keys to a new type or value, the stream needs to be repartitioned.

REPARTITIONING HAS A COST

- Repartitioning isn't free.
- There's additional overhead in this process: creating intermediate topics, storing duplicate data in another topic, and increased latency due to writing to and reading from another topic.

JOINING WITH SMALLER DATASETS

- In some cases, the lookup data you want to join against will be relatively small, and entire copies of the lookup data could fit locally on each node.
- For situations where the lookup data is reasonably small, Kafka Streams provides the GlobalKTable.

JOINING KSTREAMS WITH GLOBALKTABLES

- You performed a windowed aggregation of stock transactions per customer.
- The output of the aggregation looked like this:

```
{customerId='074-09-3705', stockTicker='GUTM'}, 17  
{customerId='037-34-5184', stockTicker='CORK'}, 16
```

JOINING KSTREAMS WITH GLOBALKTABLES

Listing 5.11 Aggregating stock transactions using session windows

```
KStream<String, TransactionSummary> countStream =  
    builder.stream( STOCK_TRANSACTIONS_TOPIC,  
    ➔ Consumed.with(stringSerde, transactionSerde)  
    ➔ .withOffsetResetPolicy(LATEST))  
        .groupBy( (noKey, transaction) ->  
    ➔ TransactionSummary.from(transaction),  
  
    ➔ Serialized.with(transactionSummarySerde, transactionSerde))  
        .windowedBy(SessionWindows.with(twentySeconds)) .count()  
        .toStream() .map(transactionMapper);
```

JOINING KSTREAMS WITH GLOBALKTABLES

- The next step is to define two GlobalKTable instances

Listing 5.12 Defining the GlobalKTables for lookup data

```
GlobalKTable<String, String> publicCompanies =  
  ➔ builder.globalTable(COMPANIES.topicName());
```

The publicCompanies lookup is
for finding companies by their
stock ticker symbol.

```
GlobalKTable<String, String> clients =  
  ➔ builder.globalTable(CLIENTS.topicName());
```

The clients lookup is for getting
customer names by customer ID.

JOINING KSTREAMS WITH GLOBAKTABLES

- Now that the components are in place, you need to construct the join.

Listing 5.13 Joining a KStream with two GlobalKTables

Sets up the leftJoin with the publicCompanies table, keys by stock ticker symbol, and returns the transactionSummary with the company name added

```
countStream.leftJoin(publicCompanies, (key, txn) ->
    ↪ txn.getStockTicker(), TransactionSummary::withCompanyName)
    .leftJoin(clients, (key, txn) ->
        ↪ txn.getCustomerId(), TransactionSummary::withCustomerName)
        .print(Printed.<String, TransactionSummary>toSysOut()
    ↪ .withLabel("Resolved Transaction Summaries"));
```

Prints the results out to the console

Sets up the leftJoin with the clients table, keys by customer ID, and returns the transactionSummary with the customer named added

JOINING KSTREAMS WITH GLOBALKTABLES

- Although there are two joins here, they're chained together because you don't use any of the results alone.
- You print the results at the end of the entire operation.
- If you run the join operation, you'll now get results like this:

```
{customer='Barney, Smith' company="Exxon", transactions= 17}
```

JOINING KSTREAMS WITH GLOBALKTABLES

Left join	Inner join	Outer join
KStream-KStream	KStream-KStream	KStream-KStream
KStream-KTable	KStream-KTable	N/A
KTable-KTable	KTable-KTable	KTable-KTable
KStream-GlobalKTable	KStream-GlobaKTable	N/A

Queryable state

- You've performed several operations involving state, and you've always printed the results to the console (for development) or written them to a topic (for production).
- When you write the results to a topic, you need to use a Kafka consumer to view the results.
- Reading the data from these topics could be considered a form of materialized views.

Queryable state

There are also some gains in efficiency resulting from not writing the data out again:

- Because the data is local, you can access it quickly.
- You avoid duplicating data by not copying it to an external store.

Summary

- KStreams represent event streams that are comparable to inserts into a database.
- KTables are update streams and are more akin to updates to a database.
- The size of a KTable doesn't continue to grow; older records are replaced with newer records.
- KTables are essential for performing aggregation operations.

COMPLETE LAB 5

6. The Processor API



The Processor API

This lesson covers

- Evaluating higher-level abstractions versus more control
- Working with sources, processors, and sinks to create a topology
- Digging deeper into the Processor API with a stock analysis processor
- Creating a co-grouping processor
- Integrating the Processor API and the Kafka Streams API

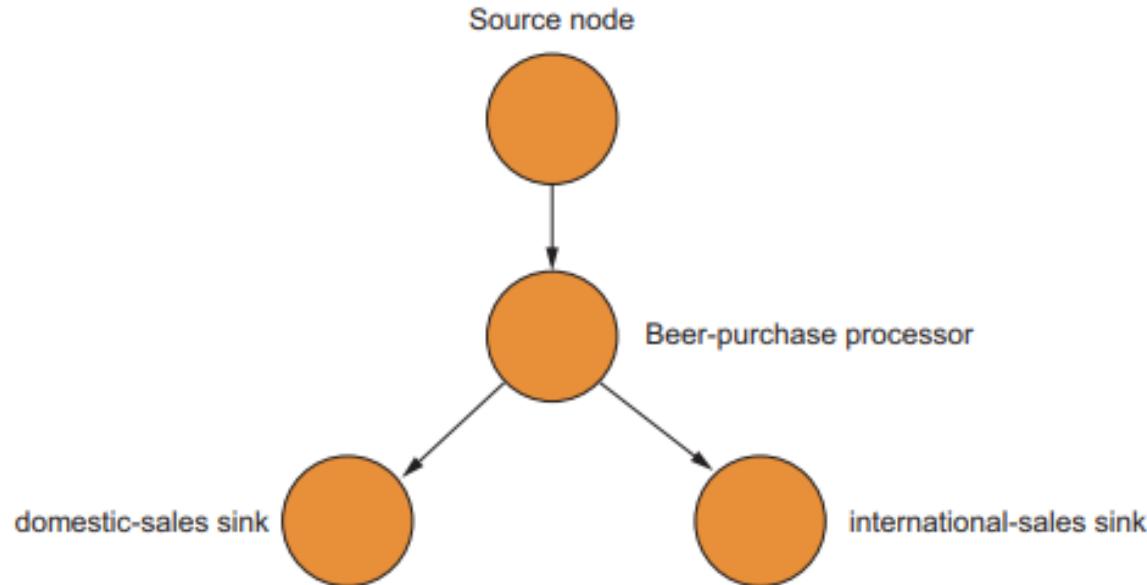
The trade-offs of higher-level abstractions vs. more control

- A classic example of trading off higher-level abstractions versus gaining more control is using object-relational mapping (ORM) frameworks.
- A good ORM framework maps your domain objects to database tables and creates the correct SQL queries for you at runtime.

Working with sources, processors, and sinks to create a topology

- Let's say you're the owner of a successful brewery (Pops Hops) with several locations.
- You've recently expanded your business to accept online orders from distributors, including international sales to Europe

Working with sources, processors, and sinks to create a topology



Adding a source node

Listing 6.1 Creating the beer application source node

```
topology.addSource(LATEST,  
                   purchaseSourceNodeName,  
                   new UsePreviousTimeOnInvalidTimestamp(),  
                   stringDeserializer,  
                   beerPurchaseDeserializer,  
                   Topics.POPS_HOPS_PURCHASES.topicName())
```

Specifies the offset reset to use

Specifies the name of this node

Specifies the TimestampExtractor to use for this source

Sets the key deserializer

Sets the value deserializer

Specifies the name of the topic to consume data from

Adding a source node

- Next, you specify the timestamp extractor to use with this source.
- We discussed the different timestamp extractors available to use for each stream source.
- Here, you're using the `UsePreviousTimeOnInvalidTimestamp` class; all other sources in the application will use the default `FailOnInvalidTimestamp` class.

Adding a processor node

- Now, you'll add a processor to work with the records coming in from the source node

Listing 6.2 Adding a processor node

```
BeerPurchaseProcessor beerProcessor =  
    new BeerPurchaseProcessor(domesticSalesSink, internationalSalesSink);  
  
topology.addSource(LATEST,  
    purchaseSourceNodeName,  
    new UsePreviousTimeOnInvalidTimestamp(),  
    stringDeserializer,  
    beerPurchaseDeserializer,  
    Topics.POPS_HOPS_PURCHASES.topicName());  
.addProcessor(purchaseProcessor,  
    () -> beerProcessor,  
    purchaseSourceNodeName);
```

Adds the processor defined above

Names the processor node

Specifies the name of the parent node or nodes

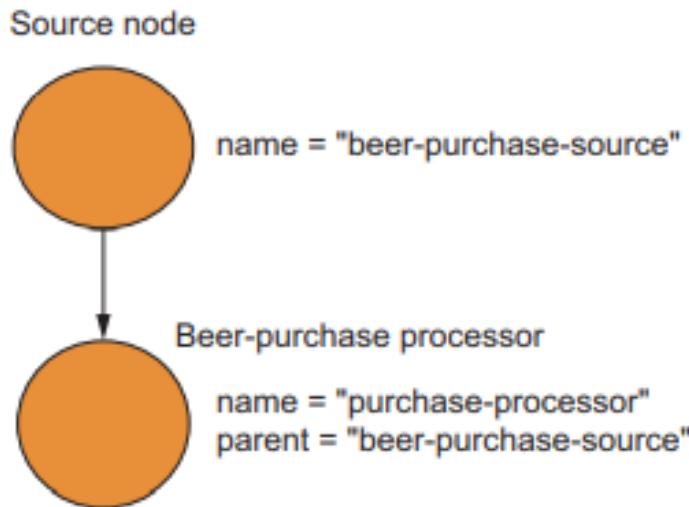
Adding a processor node

```
builder.addSource(LATEST,  
    purchaseSourceNodeName,  
    new UsePreviousTimeOnInvalidTimestamp()  
    stringDeserializer,  
    beerPurchaseDeserializer,  
    "pops-hops-purchases");
```

```
builder.addProcessor(purchaseProcessor,  
    () -> beerProcessor,  
    purchaseSourceNodeName);
```

The **name of the source node (above)** is used
for the **parent name of the processing node
(below)**. This establishes the parent-child
relationship, which directs data flow
in **Kafka Streams**.

Adding a processor node



The Processor API topology so far,
including node names and parent names

Listing 6.3 BeerPurchaseProcessor

```
public class BeerPurchaseProcessor extends  
    => AbstractProcessor<String, BeerPurchase> {  
  
    private String domesticSalesNode;  
    private String internationalSalesNode;  
  
    public BeerPurchaseProcessor(String domesticSalesNode,  
                                String internationalSalesNode) {  
        this.domesticSalesNode = domesticSalesNode;  
        this.internationalSalesNode = internationalSalesNode;    ← Sets the  
    }                                              names for  
  
    @Override  
    public void process(String key, BeerPurchase beerPurchase) {    ← different  
        ← domestic child node  
        ← international child node  
        Currency transactionCurrency = beerPurchase.getCurrency();  
  
        if (transactionCurrency != DOLLARS) {  
            BeerPurchase dollarBeerPurchase;  
            BeerPurchase.Builder builder =  
            => BeerPurchase.newBuilder(beerPurchase);  
            double internationalSaleAmount = beerPurchase.getTotalSale();  
            String pattern = "###.##";  
            DecimalFormat decimalFormat = new DecimalFormat(pattern);  
            builder.currency(DOLLARS);  
            builder.totalSale(Double.parseDouble(decimalFormat.  
                => format(transactionCurrency  
                => .convertToDollars(internationalSaleAmount)));  
                dollarBeerPurchase = builder.build();  
                context().forward(key,  
                => dollarBeerPurchase, internationalSalesNode);  
                } else {  
                    context().forward(key, beerPurchase, domesticSalesNode);  
                }  
        }  
    }  
}
```

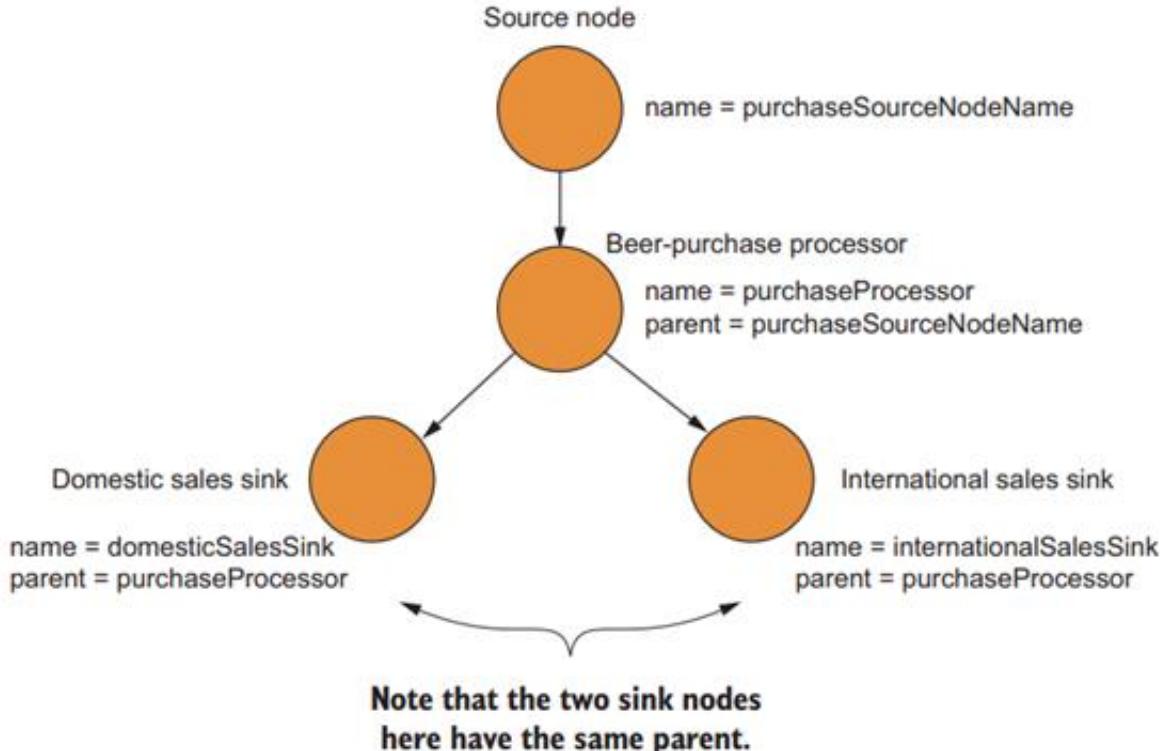
Sends records for domestic sales to the domestic child node

The process() method, where the action takes place

Converts international sales to US dollars

Uses the ProcessorContext (returned from the context() method) and forwards records to the international child node

Adding a sink node



Listing 6.4 Adding a sink node

```
topology.addSource(LATEST,
                    purchaseSourceNodeName,
                    new UsePreviousTimeOnInvalidTimestamp(),
                    stringDeserializer,
                    beerPurchaseDeserializer,
                    Topics.POPS_HOPS_PURCHASES.topicName())
    .addProcessor(purchaseProcessor,
                  () -> beerProcessor,
                  purchaseSourceNodeName)
    .addSink(internationalSalesSink,
             "international-sales",
             stringSerializer,
             beerPurchaseSerializer,
             purchaseProcessor)
    .addSink(domesticSalesSink,
             "domestic-sales",
             stringSerializer,
             beerPurchaseSerializer,
             purchaseProcessor);
```

Serializer for the key →

Name of the sink ↘

The topic this sink represents ↗

Serializer for the value ↗

Parent node for this sink ↗

Name of the sink ↗

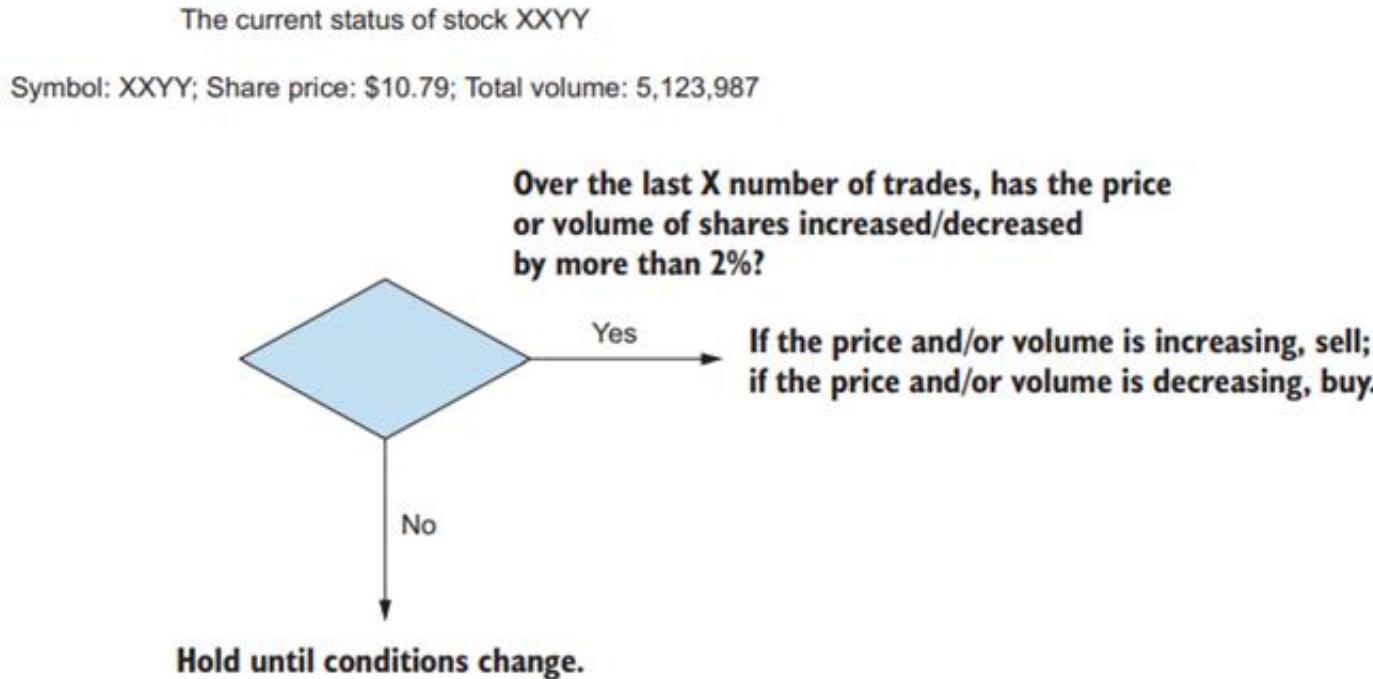
The topic this sink represents ↗

Serializer for the value ↗

Digging deeper into the Processor API with a stock analysis processor

- You'll now return to the world of finance and put on your day trading hat.
- As a day trader, you want to analyze how stock prices are changing with the intent of picking the best time to buy and sell.
- The goal is to take advantage of market fluctuations and make a quick profit.

- Figure shows the sort of decision tree you'll want to create to help make decisions.



The stock-performance processor application

Here's the topology for the stock-performance application

Listing 6.5 Stock-performance application with custom processor

```
Topology topology = new Topology();
String stocksStateStore = "stock-performance-store";
double differentialThreshold = 0.02; ← Sets the percentage
                                         differential for forwarding
                                         stock information

Creates an in-memory key/value state store
↳ KeyValueBytesStoreSupplier storeSupplier =
    ↳ Stores.inMemoryKeyValueStore(stocksStateStore);
    StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder
    ↳ = Stores.keyValueStoreBuilder(
        ↳ storeSupplier, Serdes.String(), stockPerformanceSerde); ← Creates the
                                                               StoreBuilder to place in the
                                                               topology

topology.addSource("stocks-source",
    stringDeserializer,
    stockTransactionDeserializer,
    "stock-transactions")
    .addProcessor("stocks-processor",
    ↳ () -> new StockPerformanceProcessor(
        ↳ stocksStateStore, differentialThreshold), "stocks-source") ← Adds the
                                                               processor to the topology
    .addStateStore(storeBuilder, "stocks-processor") ← Adds the state store to the stocks
                                                     processor
    .addSink("stocks-sink",
        "stock-performance",
        stringSerializer,
        stockPerformanceSerializer,
        "stocks-processor");

Adds a sink for writing results out, although you could use a printing sink as well
```

- Let's look first at the init() method in the processor

Listing 6.6 init() method tasks

```
@Override  
public void init(ProcessorContext processorContext) {  
    super.init(processorContext);  
    keyValueStore =  
        (KeyValueStore) context().getStateStore(stateStoreName);  
    StockPerformancePunctuator punctuator =  
        new StockPerformancePunctuator(differentialThreshold,  
                                         context(),  
                                         keyValueStore);  
    context().schedule(10000, PunctuationType.WALL_CLOCK_TIME,  
                      punctuator);  
}
```

Initializes ProcessorContext via the AbstractProcessor superclass

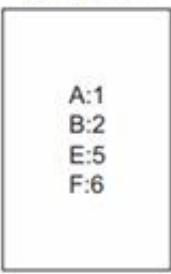
Retrieving state store created when building topology

Initializing the Punctuator to handle the scheduled processing

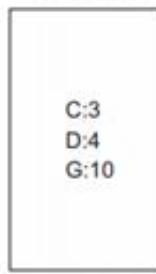
Schedules Punctuator.punctuate() to be called every 10 seconds

In the two partitions below, the letter represents the record, and the number is the timestamp. For this example, we'll assume that punctuate is scheduled to run every five seconds.

Partition A



Partition B



Because partition A has the smallest timestamp, it's chosen first:

- 1) process called with record A
- 2) process called with record B

Now partition B has the smallest timestamp:

- 3) process called with record C
- 4) process called with record D

Switch back to partition A, which has the smallest timestamp again:

- 5) process called with record E
- 6) punctuate called because time elapsed from timestamps is 5 seconds
- 7) process called with record F

Finally, switch back to partition B:

- 8) process called with record G
- 9) punctuate called again as 5 more seconds have elapsed, according to the timestamps

PUNCTUATION SEMANTICS

- The key point here is that the application advances timestamps via the `TimestampExtractor`, so `punctuate()` calls are consistent only if data arrives at a constant rate.
- If your flow of data is sporadic, the `punctuate()` method won't get executed at the regularly scheduled intervals.

The process() method

1. Check the state store to see if you have a corresponding StockPerformance object for the record's stock ticker symbol.
2. If the store doesn't contain the StockPerformance object, one is created. Then, the StockPerformance instance adds the current share price and share volume and updates your calculations.
3. Start performing calculations once you hit 20 transactions for any given stock.

The process() method

- 1) Price: \$10.79, Number shares: 5,000
- 2) Price: \$11.79, Number shares: 7,000



- 20) Price: \$12.05, Number shares: 8,000

As stocks come in, you keep a rolling average of share price and volume of shares over the last 20 trades. You also record the timestamp of the last update.

Before you have 20 trades, you take the average of the number of trades you've collected so far.

- 1) Price: \$10.79, Number shares: 5,000
- 2) Price: \$11.79, Number shares: 7,000



- 20) Price: \$12.05, Number shares: 8,000
- 21) Price: \$11.75, Number shares: 6,500
- 22) Price: \$11.95, Number shares: 7,300

After you hit 20 trades, you drop the oldest trade and add the newest one. You also update the rolling average by removing the old value from the average.

The process() method

Listing 6.7 process() implementation

```
@Override  
public void process(String symbol, StockTransaction transaction) {  
    StockPerformance stockPerformance = keyValueStore.get(symbol); ← Retrieves previous performance stats, possibly null  
  
    if (stockPerformance == null) {  
        stockPerformance = new StockPerformance(); ← Creates a new StockPerformance object if one isn't in the state store  
    }  
    stockPerformance.updatePriceStats(transaction.getSharePrice()); ← Updates the price statistics for this stock  
    stockPerformance.updateVolumeStats(transaction.getShares()); ← Sets the timestamp of the last update  
    stockPerformance.setLastUpdateSent(Instant.now()); ←  
    keyValueStore.put(symbol, stockPerformance); ← Places the updated StockPerformance object into the state store  
}
```

The punctuator execution

Listing 6.8 Punctuation code

```
@Override  
public void punctuate(long timestamp) {  
    KeyValueIterator<String, StockPerformance> performanceIterator =  
    ↪ keyValueStore.all();  
  
    while (performanceIterator.hasNext()) {  
        KeyValue<String, StockPerformance> keyValue =  
        ↪ performanceIterator.next();  
        String key = keyValue.key;  
        StockPerformance stockPerformance = keyValue.value;  
  
        if (stockPerformance != null) {  
            if (stockPerformance.priceDifferential()  
            ↪ >= differentialThreshold ||  
                stockPerformance.volumeDifferential()  
            ↪ >= differentialThreshold) {  
                context.forward(key, stockPerformance);  
            }  
        }  
    }  
}
```

Retrieves the iterator to go over all key values in the state store

Checks the threshold for the current stock

If you've met or exceeded the threshold, forwards the record

The co-group processor

Cafe purchases

Stream A key/value pairs
A:1 A:2 A:3 A:4 A:5

Joins

Records A and B are individually joined by a key and combined to produce a single joined record.



AB:1 AB:2 AB:3 AB:4 AB:5

Electronics purchases

Stream B key/value pairs
B:1 B:2 B:3 B:4 B:5

For this example, you assume the window time lines up to give you only one-to-one joins so that each record uniquely matches up with only one other record.

The co-group processor

- Now, let's imagine that you want to do a similar type of analysis, but instead of using a one-to-one join by key, you want two collections of data joined by a common key, a cogrouping of data.
- Suppose you're the manager of a popular online day-trading application.

The co-group processor

- What you need is a tuple with two collections of each event type by the company trading symbol, as shown in figure.

ClickEvents key/value pairs

A:1 A:2 A:3 A:4 A:5

Records A (click events from the day-trading application) and B (purchase transactions) are co-grouped by key (stock symbol) and produce a key/value pair where the key is K, and the value is Tuple, containing a collection of click events and a collection of stock transactions.



K, Tuple ([A1, A2, A3, A4, A5], [B1, B2, B3, B4, B5])

StockPurchase key/value pairs

B:1 B:2 B:3 B:4 B:5

For this example, each collection is populated with what's available at the time punctuate is called. There might be an empty collection at any point.

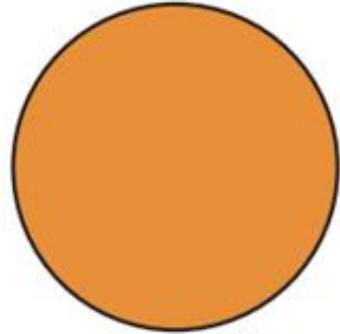
Building the co-grouping processor

To create the co-grouping processor, you need to tie a few pieces together:

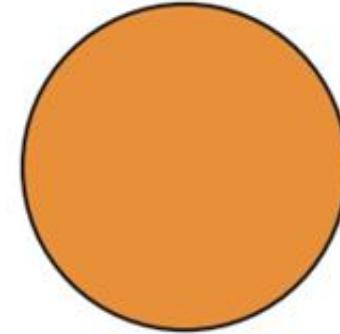
1. Define two topics (stock-transactions, events).
2. Add two processors to consume records from the topics.
3. Add a third processor to act as an aggregator/co-grouping for the two preceding processors.

DEFINING THE SOURCE NODES

transaction-source topic



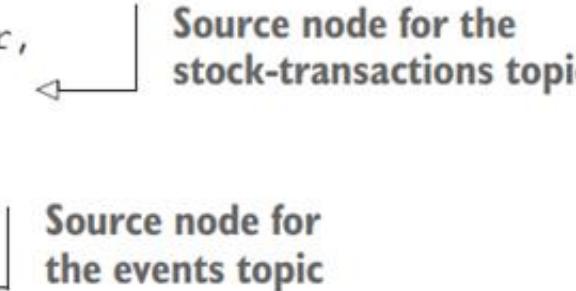
Click events source node



DEFINING THE SOURCE NODES

Listing 6.9 Source nodes for co-grouping processor

```
//I've left out configuration and (de)serializer creation for clarity.  
  
topology.addSource("Txn-Source",  
                    stringDeserializer,  
                    stockTransactionDeserializer,  
                    "stock-transactions")  
.addSource("Events-Source",  
          stringDeserializer,  
          clickEventDeserializer,  
          "events")
```

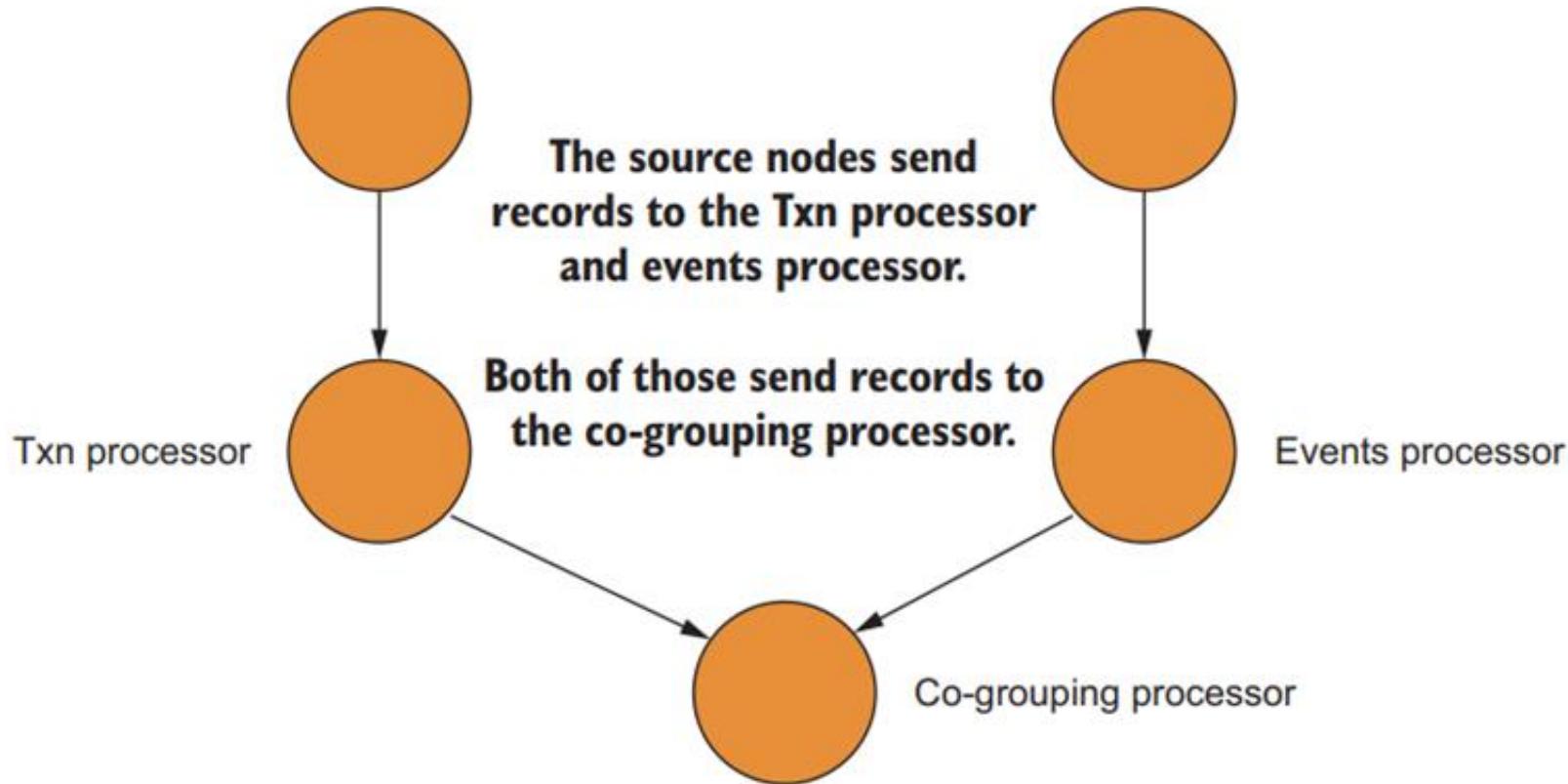


Source node for the stock-transactions topic

Source node for the events topic

transaction-source topic

Click event source node



ADDING THE PROCESSOR NODES

Listing 6.10 The processor nodes

```
.addProcessor("Txn-Processor",
    StockTransactionProcessor::new,
    "Txn-Source")
```

↳ Adds the StockTransactionProcessor


```
.addProcessor("Events-Processor",
    ClickEventProcessor::new,
    "Events-Source")
```

↳ Adds the ClickEventProcessor


```
.addProcessor("CoGrouping-Processor",
    CogroupingProcessor::new,
    "Txn-Processor",
    "Events-Processor")
```

↳ Adds the CogroupingProcessor, which
is a child node of both processors

ADDING THE PROCESSOR NODES

- Let's look at why you've set up the processors in this manner.
- This example is an aggregation operation, and the roles of the StockTransactionProcessor and ClickEventProcessor are to wrap their respective objects into smaller aggregate objects and then forward them to another processor for a total aggregation.

Listing 6.11 StockTransactionProcessor

```
public class StockTransactionProcessor extends  
→ AbstractProcessor<String, StockTransaction> {  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public void init(ProcessorContext context) {  
        super.init(context);  
    }  
  
    @Override  
    public void process(String key, StockTransaction value) {  
        if (key != null) {  
            Tuple<ClickEvent, StockTransaction> tuple =  
→ Tuple.of(null, value);  
            context().forward(key, tuple);  
        }  
    }  
}
```

Creates an aggregate object with the StockTransaction

Forwards the tuple to the CogroupingProcessor

Listing 6.12 ClickEventProcessor

```
public class ClickEventProcessor extends  
    AbstractProcessor<String, ClickEvent> {  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public void init(ProcessorContext context) {  
        super.init(context);  
  
    }  
  
    @Override  
    public void process(String key, ClickEvent clickEvent) {  
        if (key != null) {  
            Tuple<ClickEvent, StockTransaction> tuple =  
                Tuple.of(clickEvent, null);  
            context().forward(key, tuple);  
        }  
    }  
}
```

Adds the ClickEvent to the initial aggregator object

Forwards the tuple to the CogroupingProcessor

ADDING THE PROCESSOR NODES

Listing 6.13 The CogroupingProcessorinit() method

```
public class CogroupingProcessor extends  
  ↪ AbstractProcessor<String, Tuple<ClickEvent, StockTransaction>> {  
  
    private KeyValueStore<String,  
  ↪ Tuple<List<ClickEvent>, List<StockTransaction>>> tupleStore;  
    public static final String TUPLE_STORE_NAME = "tupleCoGroupStore";  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public void init(ProcessorContext context) {  
        super.init(context);  
        tupleStore = (KeyValueStore)  
  ↪ context().getStateStore(TUPLE_STORE_NAME);  
        CogroupingPunctuator punctuator =  
  ↪ new CogroupingPunctuator(tupleStore, context());  
        context().schedule(15000L, STREAM_TIME, punctuator);  
    }  
}
```

Retrieves the configured state store

Creates a Punctuator instance, CogroupingPunctuator, which handles all scheduled calls

Schedules a call to the Punctuator.punctuate() method every 15 seconds

Listing 6.14 The CogroupingProcessorprocess() method

```
@Override
    public void process(String key,
    ➔ Tuple<ClickEvent, StockTransaction> value) {

        Tuple<List<ClickEvent>, List<StockTransaction>> cogroupedTuple
    ➔ = tupleStore.get(key);
        if (cogroupedTuple == null) {
            cogroupedTuple =
    ➔ Tuple.of(new ArrayList<>(), new ArrayList<>()); ←
            }

        if (value._1 != null) {
            cogroupedTuple._1.add(value._1); ←
        }

        if (value._2 != null) {
            cogroupedTuple._2.add(value._2); ←
        }

        tupleStore.put(key, cogroupedTuple); ←
    }
}
```

Initializes the total aggregation if it doesn't exist yet

If the ClickEvent is not null, adds it to the list of click events

If the StockTransaction is not null, adds it to the list of stock transactions

Places the updated aggregation into the state store

Listing 6.15 The CogroupingPunctuator.punctuate() method

```
// leaving out class declaration and constructor for clarity
@Override
public void punctuate(long timestamp) {
    KeyValueIterator<String, Tuple<List<ClickEvent>, List<StockTransaction>> iterator = tupleStore.all();           ← Gets iterator of all co-groupings in the store

    while (iterator.hasNext()) {
        KeyValue<String, Tuple<List<ClickEvent>, List<StockTransaction>> cogrouping = iterator.next();           ← Retrieves the next co-grouping

        // if either list contains values forward results
        if (cogrouping.value != null &&
            (!cogrouping.value._1.isEmpty() ||           ← Ensures that the value is not null, and
            !cogrouping.value._2.isEmpty())) {           ← that either collection contains data

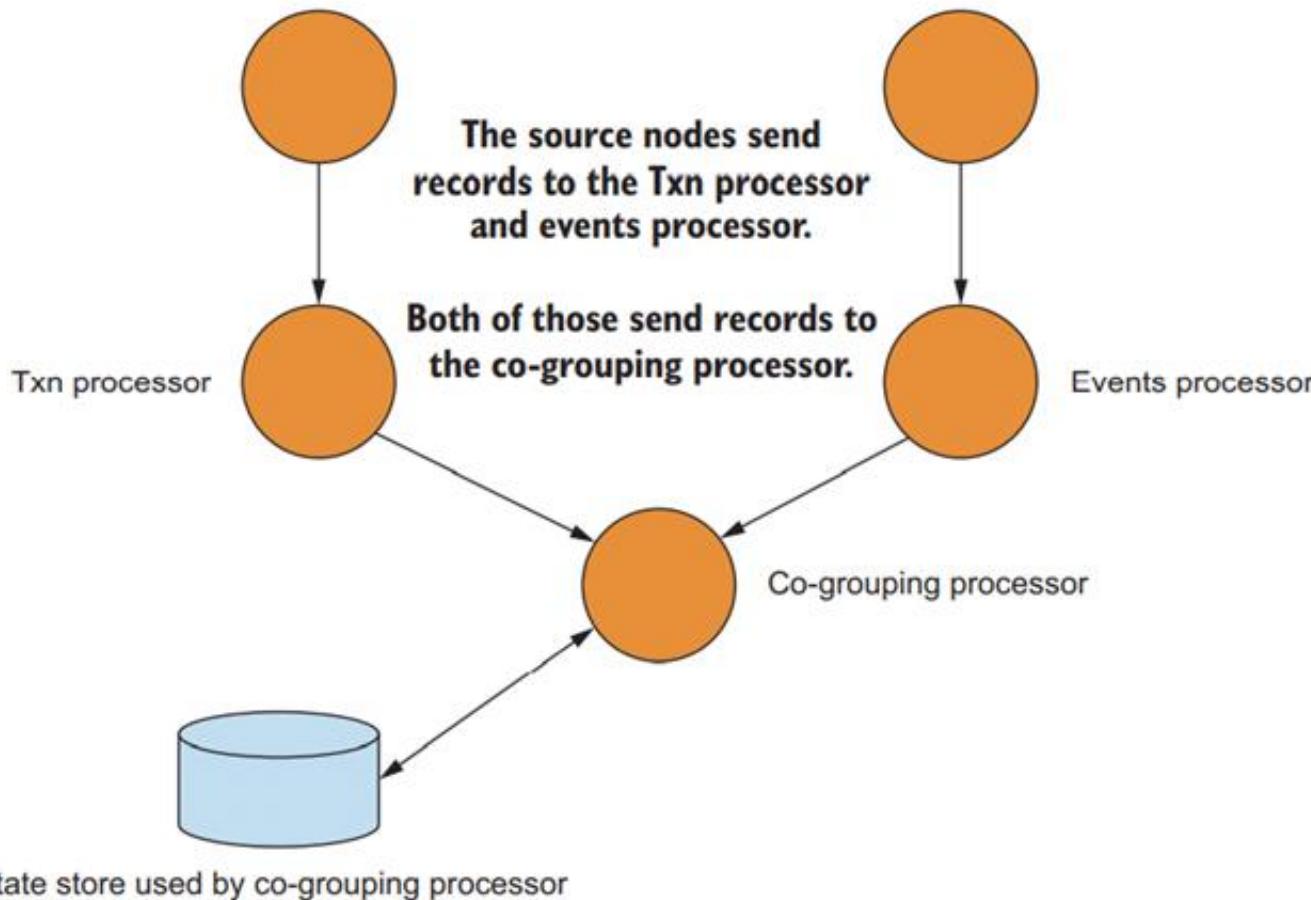
            List<ClickEvent> clickEvents =           ← Makes defensive
            new ArrayList<>(cogrouping.value._1);       ← copies of co-grouped
            List<StockTransaction> stockTransactions =   ←
            new ArrayList<>(cogrouping.value._2);           ← collections

            context.forward(cogrouping.key,
            Tuple.of(clickEvents, stockTransactions));     ← Forwards the key and aggregated co-grouping
            cogrouped.value._1.clear();
            cogrouped.value._2.clear();
            tupleStore.put(cogrouped.key, cogrouped.value); ←

        }
    }
    iterator.close();                                ← Puts the cleared-out tuple back into the store
}
```

Transaction source node

Click event source node



Listing 6.16 Adding a state store node

```
// this comes earlier in source code, included here for context
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms", "120000");
changeLogConfigs.put("cleanup.policy", "compact,delete");

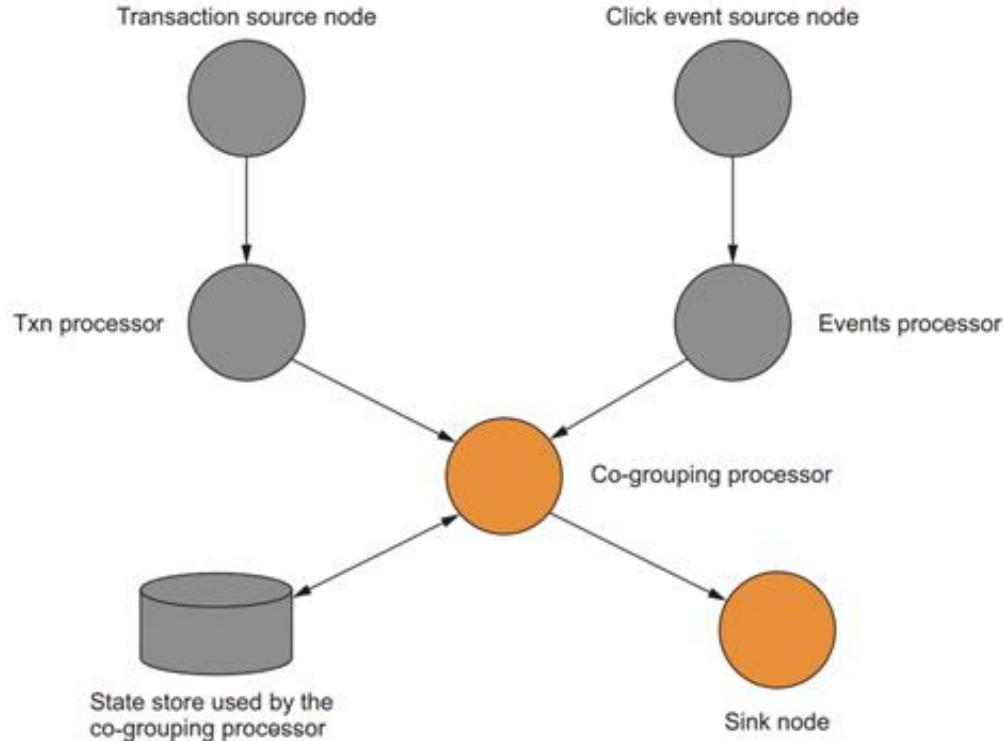
KeyValueBytesStoreSupplier storeSupplier =
    Stores.persistentKeyValueStore(TUPLE_STORE_NAME);           ← Creates the store supplier for
                                                               a persistent store (RocksDB)
    ↳ Tuple<List<ClickEvent>, List<StockTransaction>>> storeBuilder =   ← Creates the
                                                               store builder
    ↳ Stores.keyValueStoreBuilder(storeSupplier,
        Serdes.String(),
        eventPerformanceTuple)
    ↳ .withLoggingEnabled(changeLogConfigs);

.addStateStore(storeBuilder, "CoGrouping-Processor")           ← Adds the store to the
                                                               topology with the name
                                                               of the processor that
                                                               will access the store
```

Adds the changelog configs to the store builder

Specifies how long to keep records, and uses compaction and delete for cleanup

ADDING THE SINK NODE



ADDING THE SINK NODE

- Now, the co-grouped aggregation results are written out to a topic for use in further analysis.
- Here's the code

Listing 6.17 The sink node and a printing processor

```
.addSink("Tuple-Sink",
        "cogrouped-results",
        stringSerializer,
        tupleSerializer,
        "CoGrouping-Processor");
```

The sink node writes
the co-grouped tuples
out to a topic.

```
topology.addProcessor("Print",
        new KStreamPrinter("Co-Grouping"),
        "CoGrouping-Processor");
```

This processor prints
results to stdout for use
during development.

Integrating the Processor API and the Kafka Streams API

- Let's say you've used both the KStream and Processor APIs for a while.
- You've come to prefer the KStream approach, but you want to include some of your previously defined processors in a KStream application, because they provide some of the lower-level control you need

Integrating the Processor API and the Kafka Streams API

- The KStream.process method creates a terminal node, whereas the KStream.transform (or KStream.transformValues) method returns a new KStream instance allowing you to continue adding processors to that node.
- Note also that the transform methods are stateful, so you also provide a state store name when using them

Summary

- The Processor API gives you more flexibility at the cost of more code.
- Although the Processor API is more verbose than the Kafka Streams API, it's still easy to use, and the Processor API is what the Kafka Streams API, itself, uses under the covers.
- When faced with a decision about which API to use, consider using the Kafka Streams API and integrating lower-level methods (`process()`, `transform()`, `transformValues()`) when needed.

Part 3: Administering Kafka Streams

A photograph of a person's hands typing on a silver laptop keyboard. The background is slightly blurred, showing a desk with some papers and a blue and white checkered shirt cuff. The overall image has a professional, tech-oriented feel.

7: Monitoring and performance



Monitoring and performance

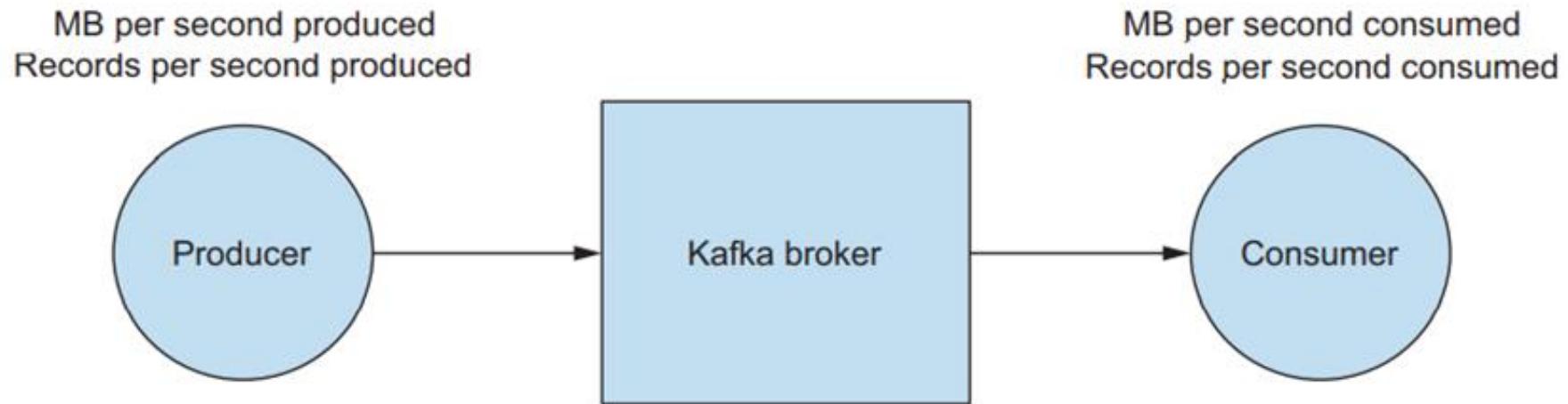
This lesson covers

- Looking at basic Kafka monitoring
- Intercepting messages
- Measuring performance
- Observing the state of the application

Basic Kafka monitoring

- Because the Kafka Streams API is a part of Kafka, it goes without saying that monitoring your application will require some monitoring of Kafka as well.
- Full-blown surveillance of a Kafka cluster is a big topic, so we'll limit our discussion of Kafka performance to where the two meet.

Measuring consumer and producer performance



Measuring consumer and producer performance

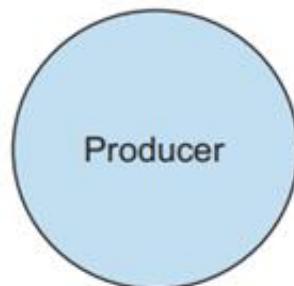
How fast are records being produced to topic A?



How fast are records consumed from topic A?

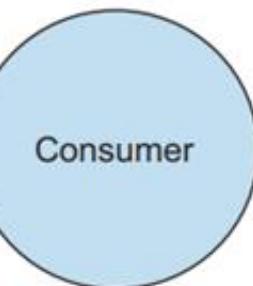
Does the consumer keep up with the rate of records coming from the producer?

Most recent message produced has an offset of 1000



994
995
996
997
998
999
1000

Last message consumed at offset 994



Last offset produced

Last offset consumed

1000	994
------	-----

The difference between the most recent offset produced and the last offset consumed (from the same topic) is known as consumer lag.

In this case, the consumer lags behind the producer by six records.

Checking for consumer lag

- First, use the list command to find all active consumer groups.
- Figure shows the results of running this command.

```
<kafka-install-dir>/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --list
```

```
oddball:bin bbejeck$ ./kafka-consumer-groups.sh --list --bootstrap-server localhost:9092
Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

console-consumer-59026
```

- With this information, you can choose a consumer group name and run the following command:

```
<kafka-install-dir>/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --group <GROUP-NAME> \
    --describe
```

shows the results: the status of how this consumer is performing.

Number of messages read = 3

Number of messages sent to topic = 10

```
oddball:bin bbejeck$ ./kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group console-consumer-59026 --describe
Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).
```

```
Consumer group 'console-consumer-59026' has no active members.
```

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENT-ID
consumer_log_demo	0	3	10	7	-	-	-

$$10 \text{ (messages sent)} - 3 \text{ (messages read)} = 7 \text{ (lag, or records behind)}$$

Intercepting the producer and consumer

- Early in 2016, Kafka Improvement Proposal 42 (KIP-42) introduced the ability to monitor or “intercept” information on client (consumer and producer) behavior.
- The goal of the KIP was to provide “the ability to quickly deploy tools to observe, measure, and monitor Kafka client behavior, down to the message level.”

CONSUMER INTERCEPTOR

- The following pseudocode will give you an idea of where the consumer interceptor is doing its work:

```
ConsumerRecords<String, String> poll(long timeout) {  
    ConsumerRecords<String, String> consumerRecords =  
    ➔ ...consuming records  
    return interceptors.onConsume(consumerRecords);
```

Fetches new records from the broker

Runs records through the interceptor chain and returns the results

CONSUMER INTERCEPTOR

- A ConsumerInterceptor accepts and returns a ConsumerRecords instance.
- If there are multiple interceptors, the returned ConsumerRecords from one interceptor serves as the input parameter for the next interceptor in the chain.
- Thus, any modifications made by one interceptor are propagated to the next interceptor in the chain.

Listing 7.1 Logging consumer interceptor

```
public class StockTransactionConsumerInterceptor implements
  => ConsumerInterceptor<Object, Object> {

    // some details left out for clarity
    private static final Logger LOG =
  => LoggerFactory.getLogger(StockTransactionConsumerInterceptor.class);

    public StockTransactionConsumerInterceptor() {
        LOG.info("Built StockTransactionConsumerInterceptor");
    }

    @Override
    public ConsumerRecords<Object, Object>
  => (ConsumerRecords<Object, Object> consumerRecords) {

        LOG.info("Intercepted ConsumerRecords {}",
            buildMessage(consumerRecords.iterator()));
        return consumerRecords;
    }                                Logs the consumer records and metadata
                                    before the records are processed
    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> map) {
        LOG.info("Commit information {}", map);
    }                                Logs the commit information once
                                    the Kafka Streams consumer
                                    commits offsets to the broker
```

PRODUCER INTERCEPTOR

- The ProducerInterceptor works similarly and has two access points: ProducerInterceptor.onSend() and ProducerInterceptor.onAcknowledgement().
- With the onSend method, the interceptor can perform any action, including mutating the ProducerRecord.
- Each producer interceptor in the chain receives the returned object from the previous interceptor.

Here's a simple logging ProducerInterceptor example

Listing 7.2 Logging producer interceptor

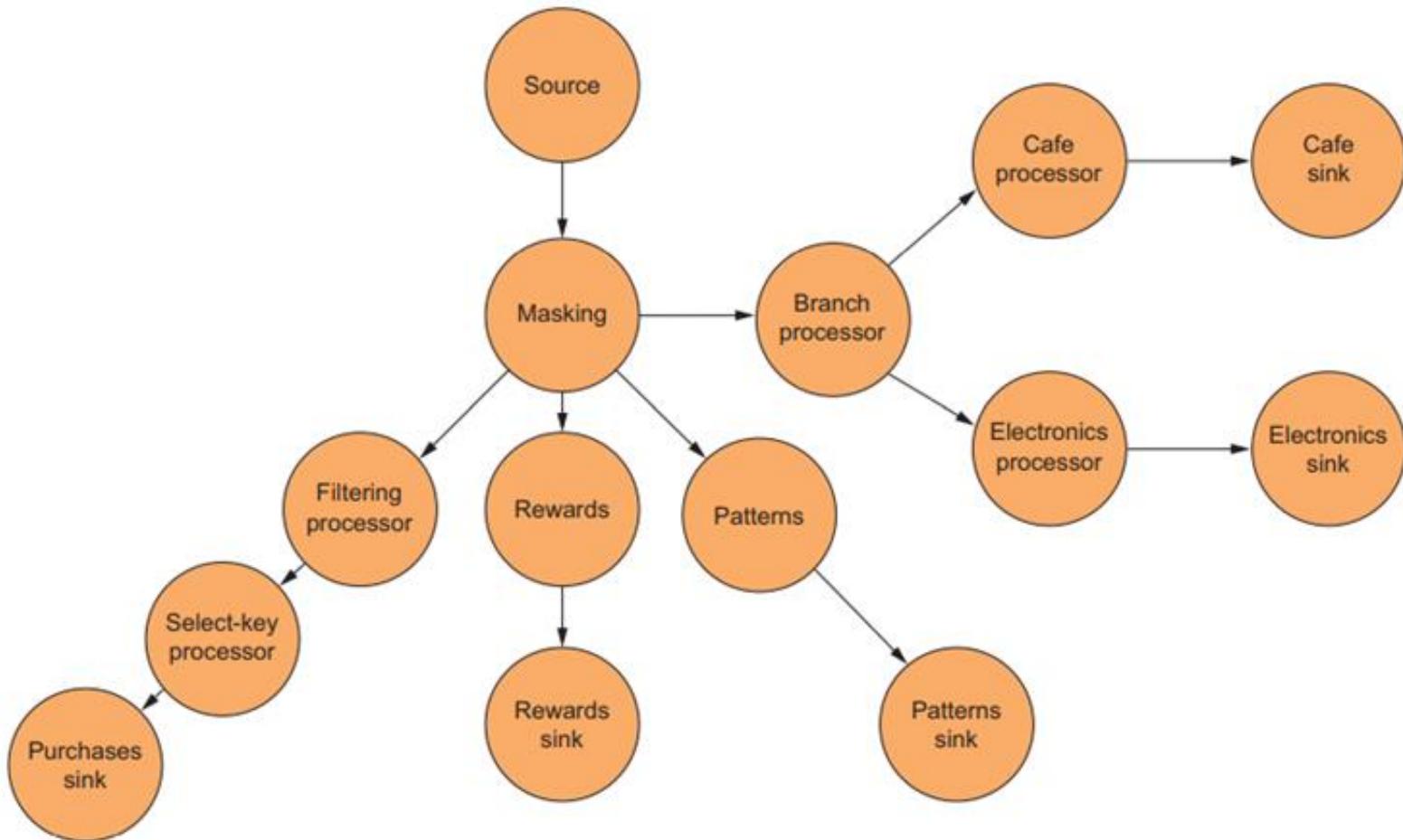
```
public class ZMartProducerInterceptor implements  
↳ ProducerInterceptor<Object, Object> {  
    // some details left out for clarity  
    private static final Logger LOG =  
↳ LoggerFactory.getLogger(ZMartProducerInterceptor.class);  
  
    @Override  
    public ProducerRecord<Object, Object> onSend(ProducerRecord<Object,  
↳ Object> record) {  
        LOG.info("ProducerRecord being sent out {}", record); ← Logs right before  
        return record;  
    }  
  
    @Override  
    public void onAcknowledgement(RecordMetadata metadata,Exception exception) {  
        if (exception != null) {  
            LOG.warn("Exception encountered producing record {}", ← Logs broker  
            exception); acknowledgement or whether error  
        } else {  
            LOG.info("record has been acknowledged {}", metadata);  
        }  
    }  
}
```

Logs right before
the message is sent
to the broker

Logs broker
acknowledgement
or whether error
occurred (broker-
side) during the
produce phase

Application metrics

- When it comes to measuring the performance of an application, you can get a sense of how long it takes to process one record, and measuring end-to-end latency is undoubtedly a good indicator of overall performance.
- But if you want to improve performance, you'll need to know exactly where things are slowing down.



Keeping the ZMart topology in mind, let's take a look at the categories of metrics:

- Thread metrics
 - Average time for commits, poll, process operations
 - Tasks created per second, tasks closed per second
- Task metrics
 - Average number of commits per second
 - Average commit time
- Processor node metrics
 - Average and max processing time
 - Average number of process operations per second
 - Forward rate
- State store metrics
 - Average execution time for put, get, and flush operations
 - Average number put, get, and flush operations per second

Metrics configuration

- Kafka Streams already provides the mechanism for collecting performance metrics.
- For the most part, you just need to provide some configuration values.
- Because the collection of metrics does incur a performance cost, there are two levels, INFO and DEBUG

Metrics configuration

Metrics category	DEBUG	INFO
Thread	X	X
Task	X	
Processor node	X	
State store	X	
Record cache	X	

- Let's update the configs in the advanced ZMart application and turn on the collection of all metrics.

Listing 7.3 Updating the configs for DEBUG metrics

```
private static Properties getProperties() {  
    Properties props = new Properties();  
    props.put(StreamsConfig.CLIENT_ID_CONFIG,  
    ↪ "metrics-client-id");                                ← Client ID  
  
    props.put(ConsumerConfig.GROUP_ID_CONFIG,  
    ↪ "metrics-group-id");                                ← Group ID  
    props.put(StreamsConfig.APPLICATION_ID_CONFIG,  
    ↪ "metrics-app-id");                                ← Application ID  
    props.put(StreamsConfig.METRICS_RECORDING_LEVEL_CONFIG,  
    ↪ "DEBUG");  
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,  
    ↪ "localhost:9092");  
    return props;  
}
```

Sets the connection for the brokers

Sets the metrics recording level to DEBUG

How to hook into the collected metrics

- The metrics in a Kafka Streams application are collected and distributed to metrics reporters.
- As you might have guessed, Kafka Streams provides a default reporter via Java Management Extensions (JMX).
- Once you've enabled collecting metrics at the DEBUG level, you have nothing left to do but observe them.

Using JMX

- JMX is a standard way of viewing the behavior of programs running on the Java VM.
- You can also use JMX to see how the Java Virtual Machine (Java VM) is performing.
- In a nutshell, JMX gives you the infrastructure to expose parts of your running program.
- Fortunately, you won't need to write any code to do this monitoring.

STARTING JCONSOLE



STARTING TO MONITOR A RUNNING PROGRAM

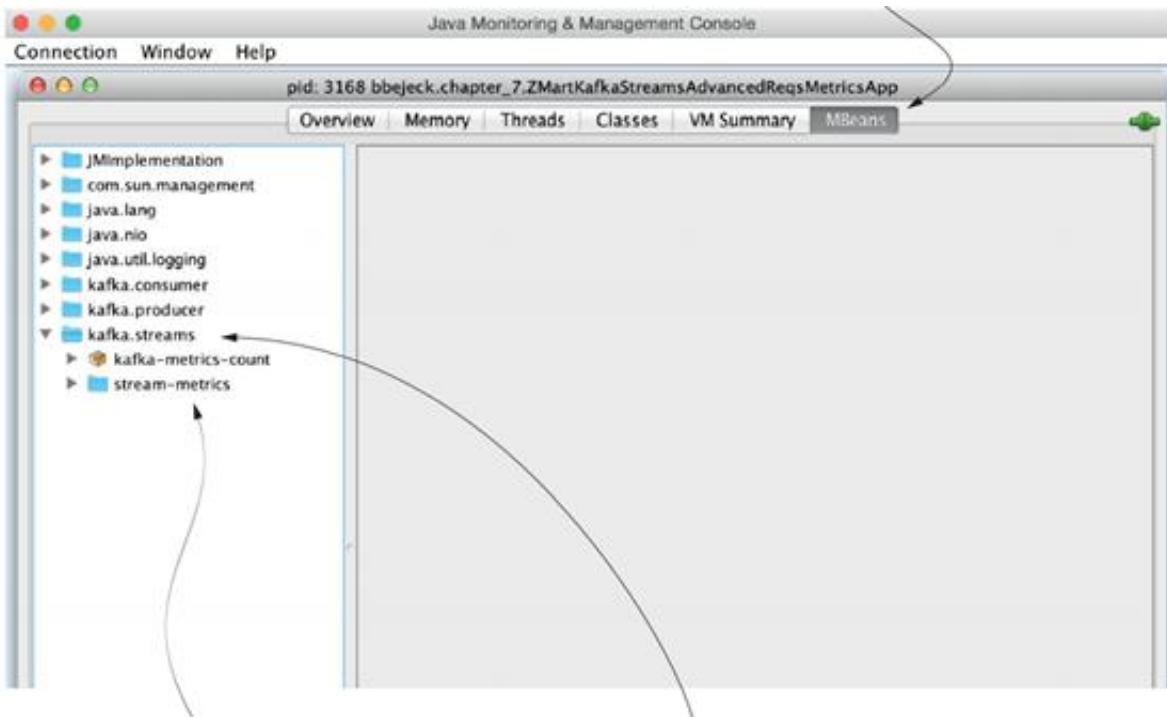


STARTING TO MONITOR A RUNNING PROGRAM



You're running on your local development
machine, so select this button.

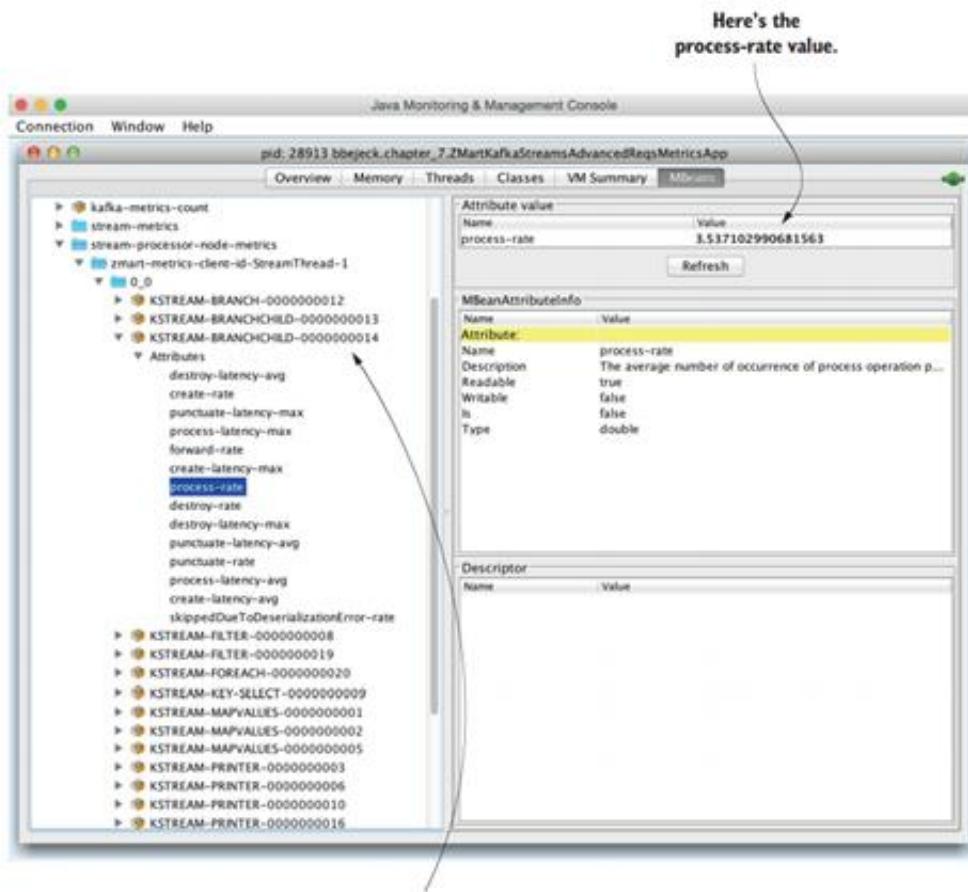
VIEWING THE INFORMATION



You're going to use the metrics found under the kafka.streams folder.

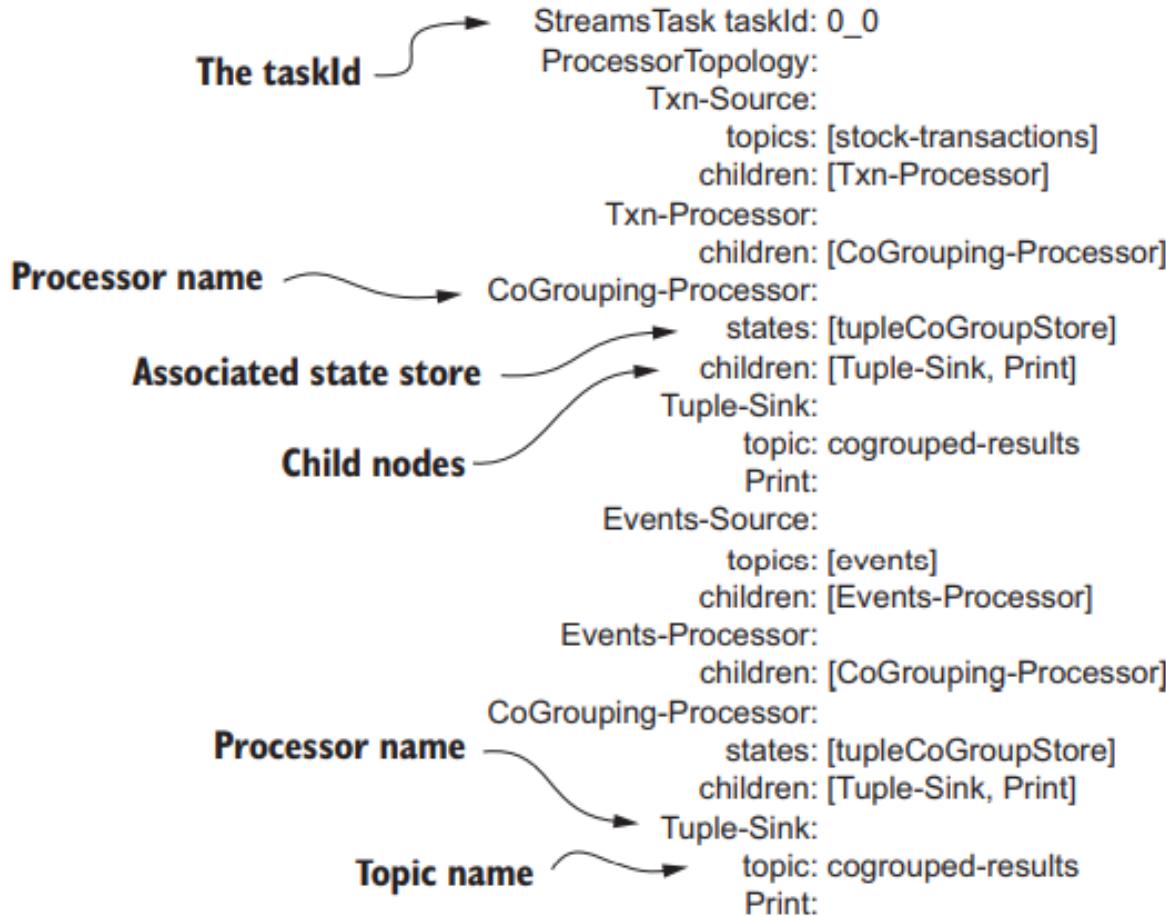
Here, you can see a good selection of MBeans for different metrics.

Viewing metrics



More Kafka Streams debugging techniques

- We'll now look at some more ways you can observe and debug Kafka streaming applications.
- The previous section was more about performance; the techniques we'll look at in this section focus on getting notified about various states of the application and viewing the structure of the topology.



Viewing a representation of the application

- With the Kafka Streams API, the names of the nodes are a little more generic:

```
KSTREAM-SOURCE-0000000000:  
    topics:      [transactions]  
    children:   [KSTREAM-MAPVALUES-0000000001]
```

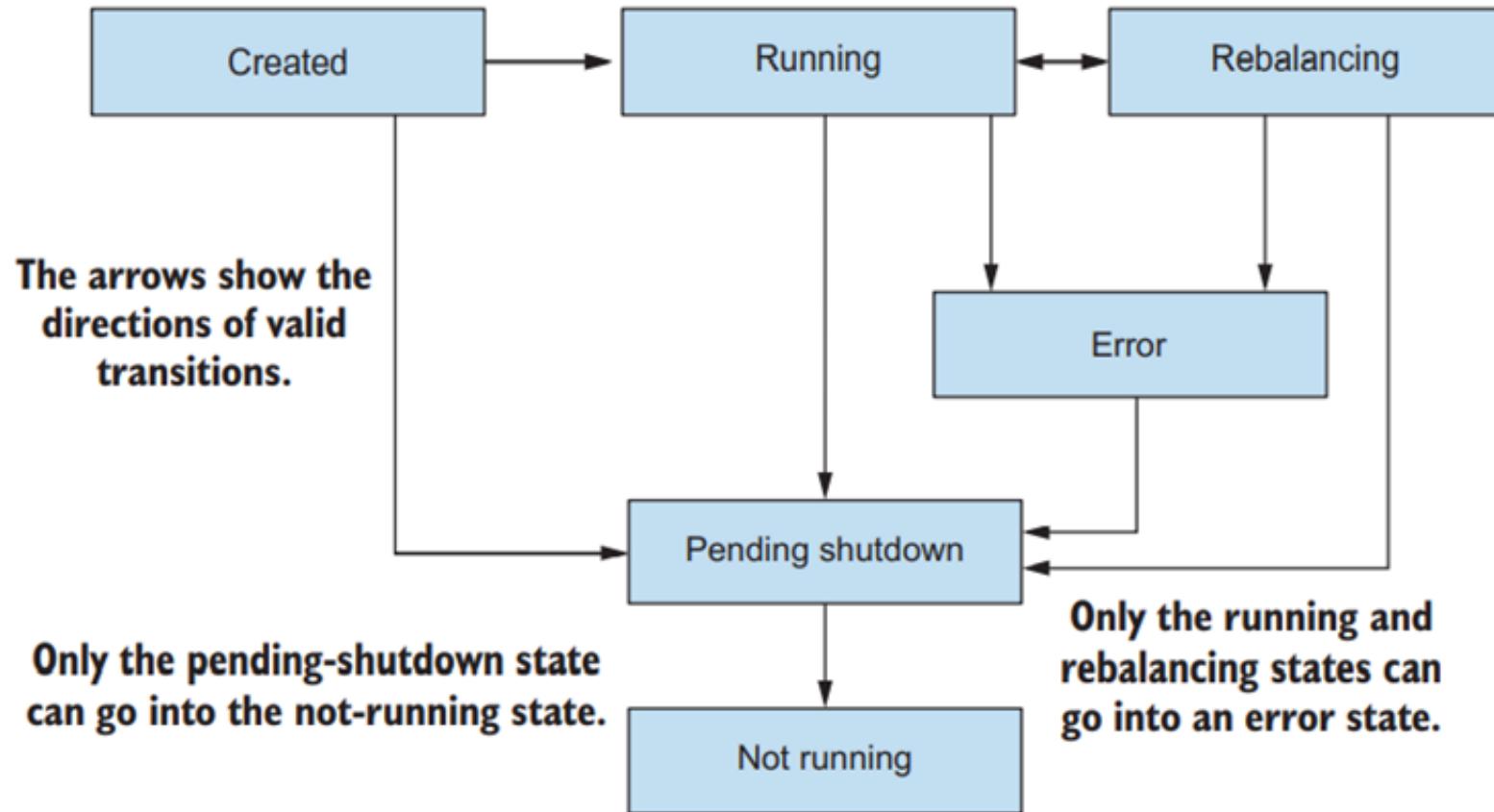
Getting notification on various states of the application

- When you start your Kafka Streams application, it doesn't automatically begin processing data—some coordination has to happen first.
- The consumer needs to fetch metadata and subscription information; the application needs to start StreamThread instances and assign TopicPartitions to StreamTasks.

Getting notification on various states of the application

- After running the app for a little while, you decide you want to process records more quickly.
- All you need to do is start another version of the application with the same application ID, and the rebalance process will distribute the load across the new application thread, resulting in the two tasks being assigned across both threads.

Six states of a Kafka Streams application



Using the StateListener

Listing 7.4 Adding a state listener

```
KafkaStreams.StateListener stateListener = (newState, oldState) -> {  
    if (newState == KafkaStreams.State.RUNNING &&  
        oldState == KafkaStreams.State.REBALANCING) {  
        LOG.info("Application has gone from REBALANCING to RUNNING ");  
        LOG.info("Topology Layout {}");  
        streamsBuilder.build().describe();  
    }  
};
```

**Prints out the structure
of the topology**

**Checks that you're
transitioning from
REBALANCING to
RUNNING**

Using the StateListener

Listing 7.5 Updating the state listener when REBALANCING

```
KafkaStreams.StateListener stateListener = (newState, oldState) -> {
    if (newState == KafkaStreams.State.RUNNING &&
        oldState == KafkaStreams.State.REBALANCING) {
        LOG.info("Application has gone from REBALANCING to RUNNING ");
        LOG.info("Topology Layout {}", streamsBuilder.build().describe());
    }

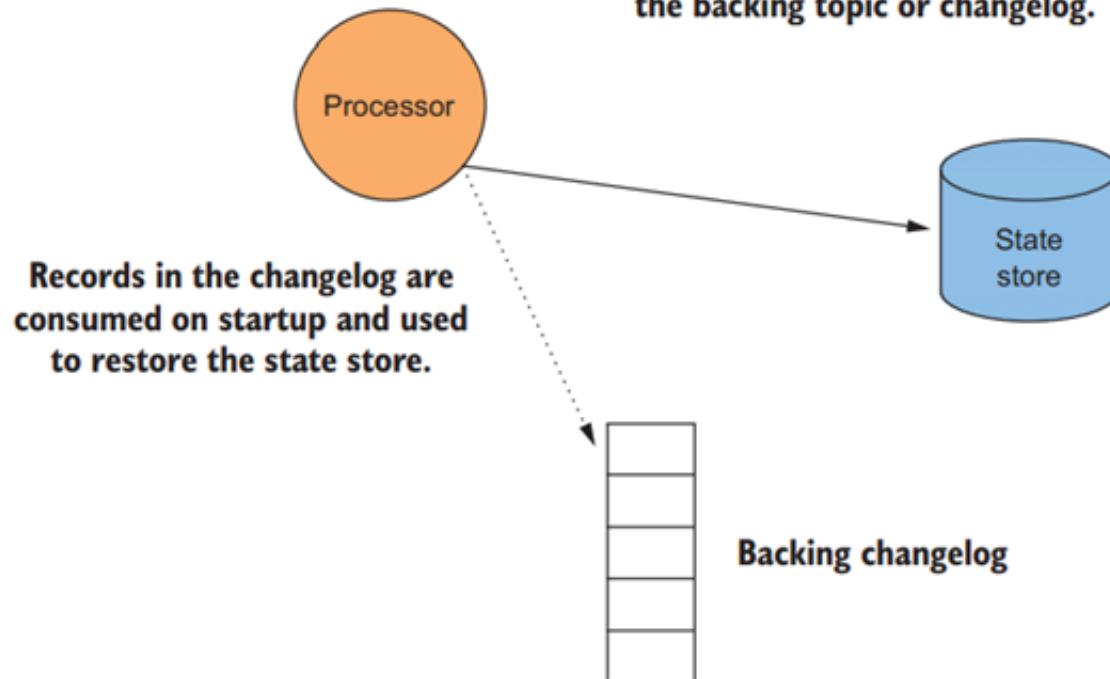
    if (newState == KafkaStreams.State.REBALANCING) {
        LOG.info("Application is entering REBALANCING phase");
    }
};
```

↳ Adds an action when entering
the rebalancing phase

State restore listener

State store restoration

On a clean startup with no persisted local state, the state is fully restored from the backing topic or changelog.



State restore listener

- In some circumstances, however, you may need to do a full recovery of state store from the changelog, such as if you're running your Kafka Streams application in a stateless environment like Mesos, or if you encounter a severe failure and the files on local disk are wiped out.
- Depending on the amount of data you have to restore, this restoration process could take a non-trivial amount of time.

Listing 7.6 A logging restore listener

```
public class LoggingStateRestoreListener implements StateRestoreListener {  
  
    private static final Logger LOG =  
        LoggerFactory.getLogger(LoggingStateRestoreListener.class);  
    private final Map<TopicPartition, Long> totalToRestore =  
        new ConcurrentHashMap<>();  
    private final Map<TopicPartition, Long> restoredSoFar =  
        new ConcurrentHashMap<>();  
  
    @Override  
    public void onRestoreStart(TopicPartition topicPartition,  
        String store, long start, long end) {  
        long toRestore = end - start;  
        totalToRestore.put(topicPartition, toRestore);  
        LOG.info("Starting restoration for {} on topic-partition {}  
            total to restore {}", store, topicPartition, toRestore);  
    }  
  
    // other methods left out for clarity covered below  
}
```

Creates ConcurrentHashMap instances for keeping track of restore progress

Stores the total amount to restore for the given TopicPartition

State restore listener

Listing 7.7 Handling onBatchRestored

```
@Override  
public void onBatchRestored(TopicPartition topicPartition,  
    String store, long start, long batchCompleted) {  
    NumberFormat formatter = new DecimalFormat("#.##");  
  
    long currentProgress = batchCompleted +  
        restoredSoFar.getOrDefault(topicPartition, 0L);  
    double percentComplete =  
        (double) currentProgress / totalToRestore.get(topicPartition);  
  
    LOG.info("Completed {} for {}% of total restoration for {} on {}",  
        batchCompleted,  
        formatter.format(percentComplete * 100.00),  
        store, topicPartition);  
    restoredSoFar.put(topicPartition, currentProgress);  
}
```

Stores the number of records restored so far

Calculates the total number of records restored

Determines the percentage of restoration completed

Logs the percent restored

State restore listener

- The last step we'll cover is when the restoration process completes

Listing 7.8 Method called when restoration is completed

```
@Override
public void onRestoreEnd(TopicPartition topicPartition,
    String store, long totalRestored) {
    LOG.info("Restoration completed for {} on
    topic-partition {}", store, topicPartition);
    restoredSoFar.put(topicPartition, 0L);      ←
}
    
```

Keeps track of restore progress
for a TopicPartition

State restore listener

- Finally, you can use the LoggingStateRestoreListener in your application as follows

Listing 7.9 Specifying the global restore listener

```
kafkaStreams.setGlobalStateRestoreListener(new LoggingStateRestoreListener());
```

Uncaught exception handler

- Kafka Streams provides `KafkaStreams.setUncaughtExceptionHandler` for dealing with these unexpected error

Listing 7.10 Basic uncaught exception handler

```
kafkaStreams.setUncaughtExceptionHandler((thread, exception) -> {  
    CONSOLE_LOG.info("Thread [" + thread + "]  
    ➔ encountered [" + exception.getMessage() +"]");  
});
```

Summary

- To monitor Kafka Streams, you'll need to look at the Kafka brokers as well.
- You should enable metrics reporting from time to time to see the how the performance of the application is doing.
- Peeking under the hood is required, and sometimes you'll need to go to lower levels and use command-line tools included with Java, such as jstack (thread dumps) and jmap/jhat (for heap dumps) to understand what your application is doing.

COMPLETE LAB 7

8. Testing a Kafka Streams application

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a developer environment. The laptop screen shows some code or terminal output. The image is partially covered by a large, semi-transparent gray overlay containing the title text.

Testing a Kafka Streams application

This lesson covers

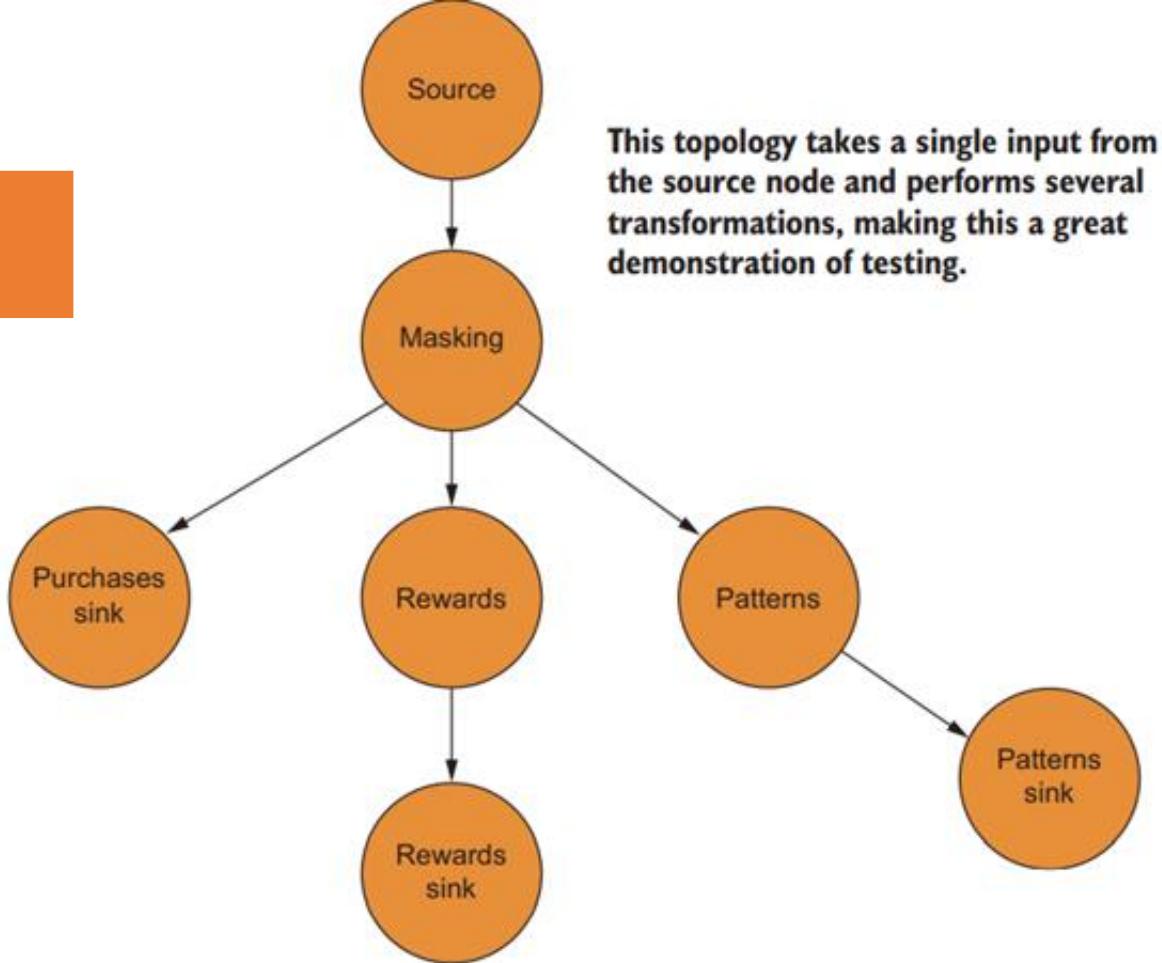
- Testing a topology
- Testing individual processors and transformers
- Integration testing with an embedded Kafka cluster

Testing a Kafka Streams application

Table summarizes the differences between unit and integration testing.

Test type	Purpose	Testing speed	Level of use
Unit	Testing individual parts of functionality in isolation	Fast	Large majority
Integration	Testing integration points between whole systems	Longer to run	Small minority

Testing a topology



Building the test

Listing 8.1 Setup method for topology test

```
@Before  
public void setUp() {  
  
    // properties construction left out for clarity  
    StreamsConfig streamsConfig = new StreamsConfig(props);  
    Topology topology = ZMartTopology.build();  
  
    topologyTestDriver =  
    ➔ new ProcessorTopologyTestDriver(streamsConfig, topology);  
}
```

Refactored ZMart topology: now you can get the topology from the method call.

Creates the ProcessorTopologyTestDriver

Listing 8.2 Testing the topology

```
@Test
public void testZMartTopology() {
    // serde creation left out for clarity

    Purchase purchase = DataGenerator.generatePurchase(); ← Creates a test object; reuses the generation code from running the topology

    topologyTestDriver.process("transactions", ← Sends an initial record
                               null,           into the topology
                               purchase,
                               stringSerde.serializer(),
                               purchaseSerde.serializer());;

    ProducerRecord<String, Purchase> record = ← Reads a record from
    → topologyTestDriver.readOutput("purchases",   the purchases topic
                                     stringSerde.deserializer(),
                                     purchaseSerde.deserializer());;

    Purchase expectedPurchase = ← Converts the
    → Purchase.builder(purchase).maskCreditCard().build(); test object to the expected format
    → assertEquals(record.value(), equalTo(expectedPurchase));;

    Verifies that the record from the
    topology matches the expected record
```

Listing 8.3 Testing the rest of the topology

```
@Test
public void testZMartTopology() {

    // continuing test from the previous section

    RewardAccumulator expectedRewardAccumulator =
        RewardAccumulator.builder(expectedPurchase).build();
```

↳ Reads a record from the rewards topic

```
    ProducerRecord<String, RewardAccumulator> accumulatorProducerRecord =
        topologyTestDriver.readOutput("rewards",
            stringSerde.deserializer(),
            rewardAccumulatorSerde.deserializer());
```

```
    assertThat(accumulatorProducerRecord.value(),
        equalTo(expectedRewardAccumulator));
```

↳ Verifies the rewards topic output matches expectations

```
    PurchasePattern expectedPurchasePattern =
        PurchasePattern.builder(expectedPurchase).build();
```

```
    ProducerRecord<String, PurchasePattern> purchasePatternProducerRecord =
        topologyTestDriver.readOutput("patterns",
            stringSerde.deserializer(),
            purchasePatternSerde.deserializer());
```

```
    assertThat(purchasePatternProducerRecord.value(),
        equalTo(expectedPurchasePattern));
}
```

↳ Verifies the patterns topic output matches expectations

↳ Reads a record from the patterns topic

Testing a state store in the topology

Listing 8.4 Testing the state store

```
StockTransaction stockTransaction =  
    DataGenerator.generateStockTransaction();           ↪ Generates a test record  
  
topologyTestDriver.process("stock-transactions",  
    stockTransaction.getSymbol(),  
    stockTransaction,  
    stringSerde.serializer(),  
    stockTransactionSerde.serializer());                ↪ Processes the record with the test driver  
  
KeyValueStore<String, StockPerformance> store =  
    topologyTestDriver.getKeyValueStore("stock-performance-store");    ↪ Retrieves the state store from the test topology  
  
assertThat(store.get(stockTransaction.getSymbol()),  
    notNullValue());                                     ↪ Asserts the store contains the expected value
```

Testing processors and transformers

- To verify the behavior inside a single class requires a true unit test, where there's only one class under test.
- Writing a unit test for a Processor or Transformer shouldn't be very challenging, but remember that both classes have a dependency on the Processor Context for obtaining any state stores and scheduling punctuation actions.

Testing processors and transformers

- If you need the mock object strictly as a placeholder for a real dependency, concrete mocks (mocks not created from a framework) are a good choice.
- But if you want to verify the parameters passed to a mock, the value returned, or any other behavior, using a mock object generated by a framework is a good choice.

Listing 8.5 Testing the init method

```
// some details left out for clarity
private ProcessorContext processorContext =
    mock(ProcessorContext.class);                                | Mocks the ProcessorContext
                                                               | with Mockito
private MockKeyValueStore<String, Tuple<List<ClickEvent>,
    List<StockTransaction>>> keyValueStore =
    new MockKeyValueStore<>();                                    | A mock
                                                               | KeyValueStore
                                                               | object
private AggregatingMethodHandleProcessor processor =
    new AggregatingMethodHandleProcessor();                      | The class under test

@Test
@DisplayName("Processor should initialize correctly")
public void testInitializeCorrectly() {
    processor.init(processorContext);                            | Calls the init method on the
                                                               | processor, triggering method
                                                               | calls on ProcessorContext
    verify(processorContext).schedule(eq(15000L), eq(STREAM_TIME),
        isA(Punctuator.class));                                |
    verify(processorContext).getStateStore(TUPLE_STORE_NAME);  | Verifies retrieving
                                                               | the state store
}

Verifies the parameters for the
ProcessorContext.schedule method
```

Listing 8.6 Testing the punctuate method

```
@Test
@DisplayName("Punctuate should forward records")
public void testPunctuateProcess() {
    when(processorContext.getStateStore(TUPLE_STORE_NAME))
        .thenReturn(keyValueStore);
```

Calls init method on processor

```
    ▶ processor.init(processorContext);
    processor.process("ABC", Tuple.of(clickEvent, null));
    processor.process("ABC", Tuple.of(null, transaction));
```

Processes a ClickEvent and a StockTransaction

Calls the co-group method, which is the method used to schedule punctuate

```
    Tuple<List<ClickEvent>, List<StockTransaction>> tuple =
        keyValueStore.innerStore().get("ABC");
    List<ClickEvent> clickEvents = new ArrayList<>(tuple._1);
    List<StockTransaction> stockTransactions = new ArrayList<>(tuple._2);

    ▶ processor.cogroup(124722348947L);
    verify(processorContext).forward("ABC",
        ▶ Tuple.of(clickEvents, stockTransactions));
```

Extracts the entries put into the state store in the process method

Validates that the ProcessorContext forwards the expected records

```
    assertThat(tuple._1.size(), equalTo(0));
    assertThat(tuple._2.size(), equalTo(0));
}
```

Validates that the collections within the tuple are cleared out

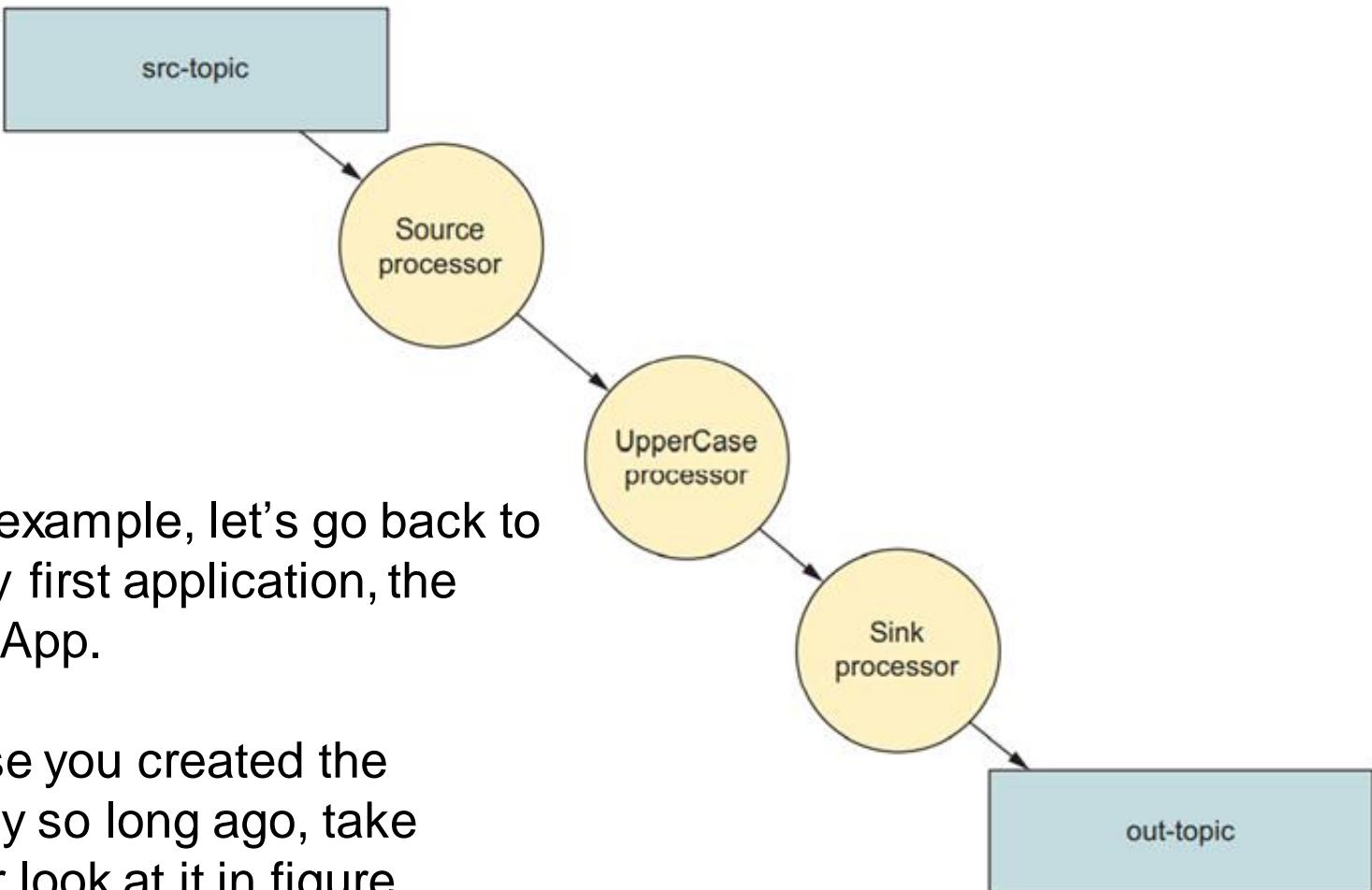
Testing processors and transformers

- Now, you verify the main point of the test: that the expected key and value are forwarded downstream via the `ProcessorContext.forward` method.
- This portion of the test demonstrates the usefulness of a generated mock object.

Integration testing

- There are times where you'll need to test all the working parts together, end to end: in other words, an integration test.
- Usually, an integration test is required when you have some functionality that can't be covered in a unit test.

- For an example, let's go back to our very first application, the Yelling App.
- Because you created the topology so long ago, take another look at it in figure



Integration testing

- Suppose you've decided to change the source from a single named topic to any topic matching a regex:

yell-[A-Za-z0-9-]

Integration testing

- You won't update the original Yelling App, since it's so small.
- Instead you'll use the following modified version directly in the test

Listing 8.7 Updating the Yelling App

```
streamsBuilder.<String, String>stream(Pattern.compile("yell.*"))
    .mapValues(String::toUpperCase)
    .to(OUT_TOPIC);
```

Converts all text to uppercase

Writes out to topic, or yells at people!



Subscribes to any topic starting with "yell"

Building an integration test

- The first step to using the embedded Kafka server requires you to add three more testing dependencies—`scala-library-2.12.4.jar`, `kafka_2.12-1.0.0-test.jar`, and `kafka_2.12- 1.0.0.jar`—to your `build.gradle` or `pom.xml` file.
- We've already covered the syntax for providing a test JAR.

ADDING THE EMBEDDEDKAFKA CLUSTER

- Adding the embedded Kafka broker to the test is a matter of adding one line, as shown in the following listing

Listing 8.8 Adding the embedded Kafka broker

```
private static final int NUM_BROKERS = 1;           ← Defines the number  
                                                 of brokers  
  
@ClassRule  
public static final EmbeddedKafkaCluster EMBEDDED_KAFKA  
= new EmbeddedKafkaCluster(NUM_BROKERS);           ← The JUnit ClassRule  
                                                 annotation  
                                                 ← Creates an instance of the  
                                                 EmbeddedKafkaCluster
```

JUNIT RULES

- JUnit introduced the concept of rules to apply some common logic JUnit tests.
- Here's a brief definition, from <https://github.com/junit-team/junit4/wiki/Rules#rules>: “Rules allow very flexible addition or redefinition of the behavior of each test method in a test class.”

CREATING TOPICS

- Now that your embedded Kafka cluster is available, you can use it to create topics, as follows

Listing 8.9 Creating the topics for testing

```
@BeforeClass  
public static void setUpAll() throws Exception {  
    EMBEDDED_KAFKA.createTopic(YELL_A_TOPIC);  
    EMBEDDED_KAFKA.createTopic(OUT_TOPIC);  
}
```

BeforeClass annotation

Creates the first source topic

Creates the output topic

CREATING TOPICS

For that, you can use one of the following overloaded `createTopic` methods:

- `EmbeddedKafkaCluster.createTopic(String topic, int partitions, int replication)`
- `EmbeddedKafkaCluster.createTopic(String topic, int partitions, int replication, Properties topicConfig)`

TESTING THE TOPOLOGY

All the pieces are in place. Now you can follow these steps to execute the integration test:

1. Start the Kafka Streams application.
2. Write some records to the source topic and assert the correct results.
3. Create a new topic matching your pattern.
4. Write some additional records to the newly created topic and assert the correct results.

TESTING THE TOPOLOGY

- Let's start with the first two parts of the test

Listing 8.10 Starting the application and asserting the first set of values

```
// some setup code left out for clarity

kafkaStreams = new KafkaStreams(streamsBuilder.build(), streamsConfig);
kafkaStreams.start();                                ← Starts the Kafka Streams application

List<String> valuesToSendList =
    ↪ Arrays.asList("this", "should", "yell", "at", "you");      ← Specifies the list of values to send

List<String> expectedValuesList =
    ↪ valuesToSendList.stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());                                ← Creates the list of expected values

    ↪ IntegrationTestUtils.produceValuesSynchronously(YELL_A_TOPIC,
        valuesToSendList,
        producerConfig,
        mockTime);

    int expectedNumberOfRecords = 5;
    List<String> actualValues =
        ↪ IntegrationTestUtils.waitUntilMinValuesRecordsReceived(
        ↪ consumerConfig, OUT_TOPIC, expectedNumberOfRecords);      ← Consumes records from Kafka

    assertThat(actualValues, equalTo(expectedValuesList));          ← Asserts the values read are equal to the expected values

    Produces the values to embedded Kafka
```

PRODUCING AND CONSUMING RECORDS IN A TEST

- The `IntegrationTestUtils.produceValuesSynchronously` method creates a Producer Record for each item in the collection with a null key.
- This method is synchronous, as it takes the resulting `Future<RecordMetadata>` from the `Producer.send` call and immediately calls `Future.get()`, which blocks until the produce request returns.

Listing 8.11 Starting the application and asserting values

```
EMBEDDED_KAFKA.createTopic(YELL_B_TOPIC);           ← Creates the new topic
valuesToSendList = Arrays.asList("yell", "at", "you", "too"); ← Specifies a new list of values to send

expectedValuesList = valuesToSendList.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());                         ← Creates the expected values

→ IntegrationTestUtils.produceValuesSynchronously(YELL_B_TOPIC,
                                                 valuesToSendList,
                                                 producerConfig,
                                                 mockTime);

expectedNumberOfRecords = 4;
List<String> actualValues =
    ↵IntegrationTestUtils.waitUntilMinValuesRecordsReceived(
    ↵consumerConfig, OUT_TOPIC, expectedNumberOfRecords);   ← Consumes the results of the streaming application

assertThat(actualValues, equalTo(expectedValuesList));   ← Asserts that the expected results match the actual results

Produces the values to the source topic of the streaming application
```

DYNAMICALLY ADDING A TOPIC

- You create a new topic matching the pattern for the source node of the streaming application.
- After that, you go through the same steps of populating the new topic with data, and consuming records from the topic backing the sink node of the streaming application.
- At the end of the test, you verify that the consumed results match the expected results

Summary

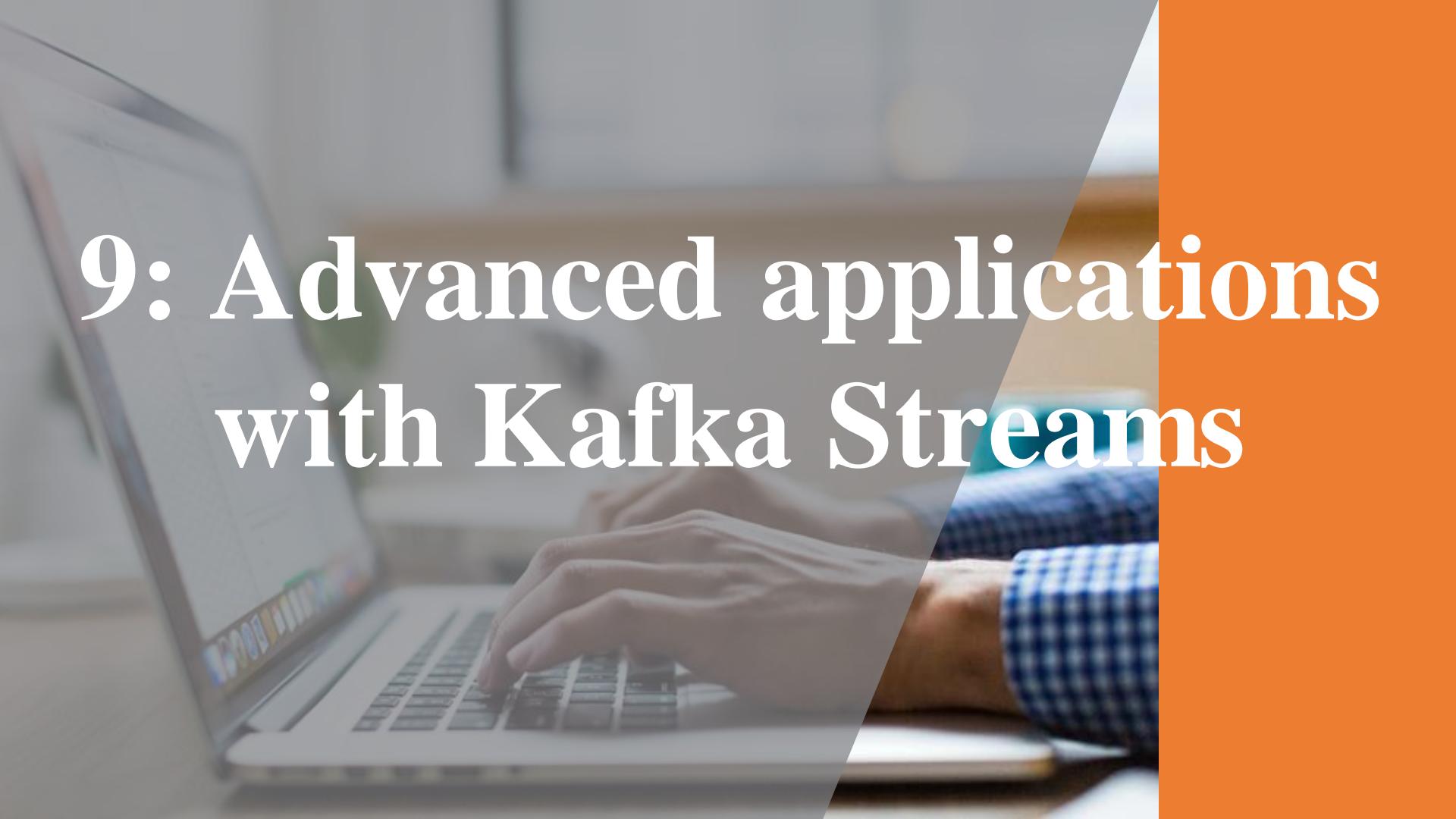
- Strive to keep business logic in standalone classes that are entirely independent of your Kafka Streams application. This makes them easy to unit test.
- It's useful to have at least one test using ProcessorTopologyTestDriver to test your topology from end to end. This type of test doesn't use a Kafka broker, so it's fast, and you can see end-to-end results.

COMPLETE LAB 8

Part 4: Advanced concepts with Kafka Streams

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or data processing environment.

9: Advanced applications with Kafka Streams



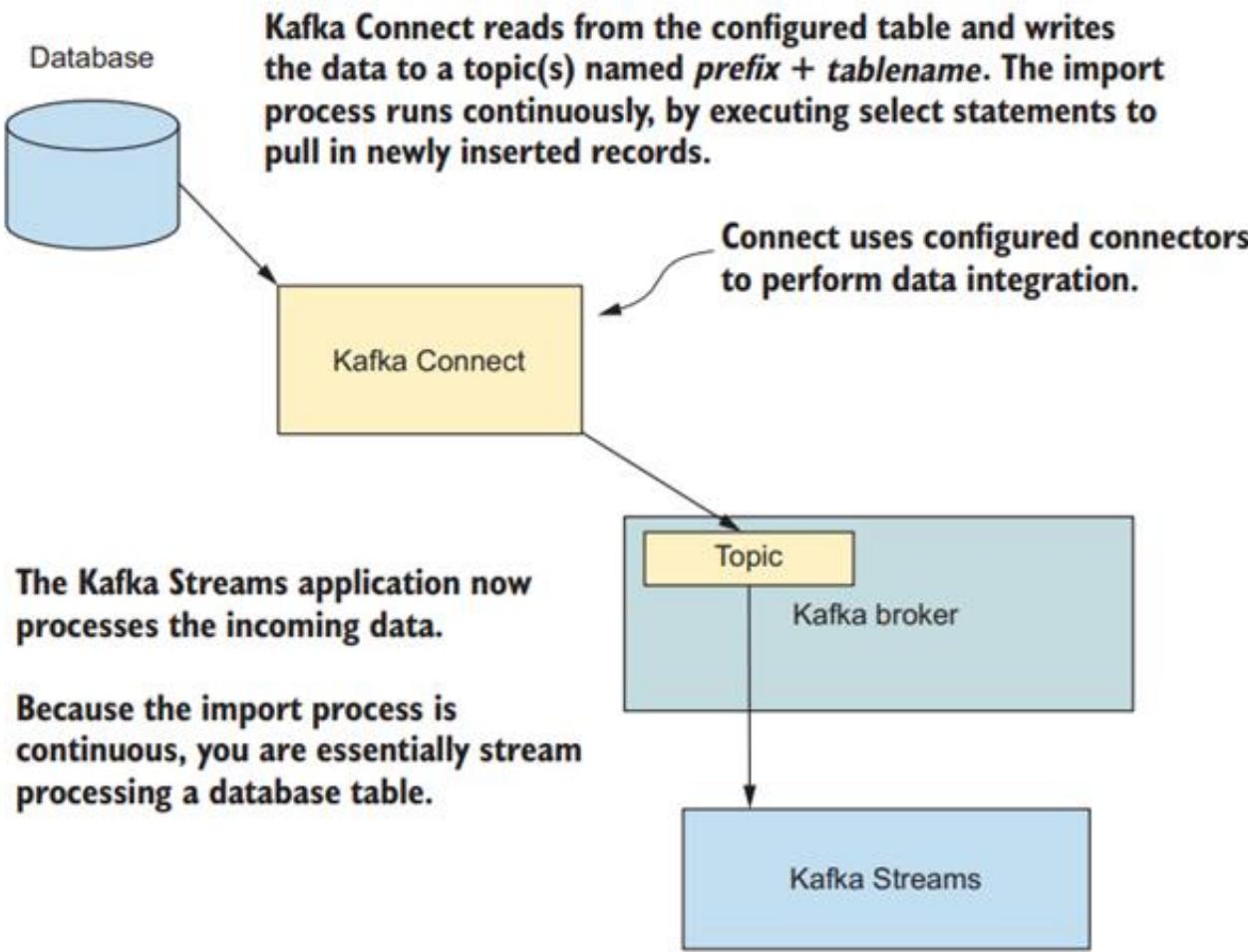
Advanced applications with Kafka Streams

This lesson covers

- Integrating outside data into Kafka Streams with Kafka Connect
- Kicking your database to the curb with interactive queries
- KSQL continuous queries in Kafka

Integrating Kafka with other data sources

- For the first advanced example application, let's suppose you work at an established financial services firm, Big Short Equity (BSE).
- BSE wants to migrate its legacy data operations into the modern era, and that plan includes using Kafka.
- The migration is part-way through, and you're tasked with updating the company's analytics.



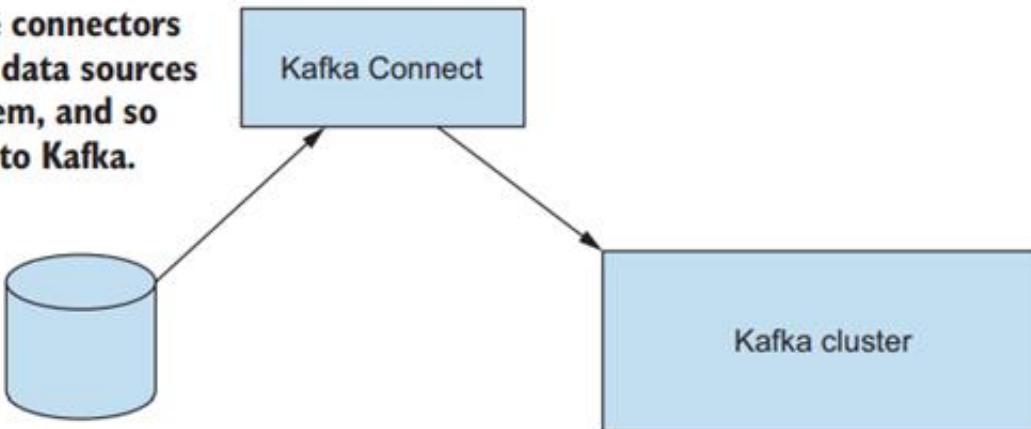
Using Kafka Connect to integrate data

- Kafka Connect is explicitly designed for streaming data from other systems into Kafka and for streaming data from Kafka into another data-storage application such as MongoDB (www.mongodb.com) or Elasticsearch (www.elastic.co).
- With Kafka Connect, it's possible to import entire databases into Kafka, or other data such as performance metrics.

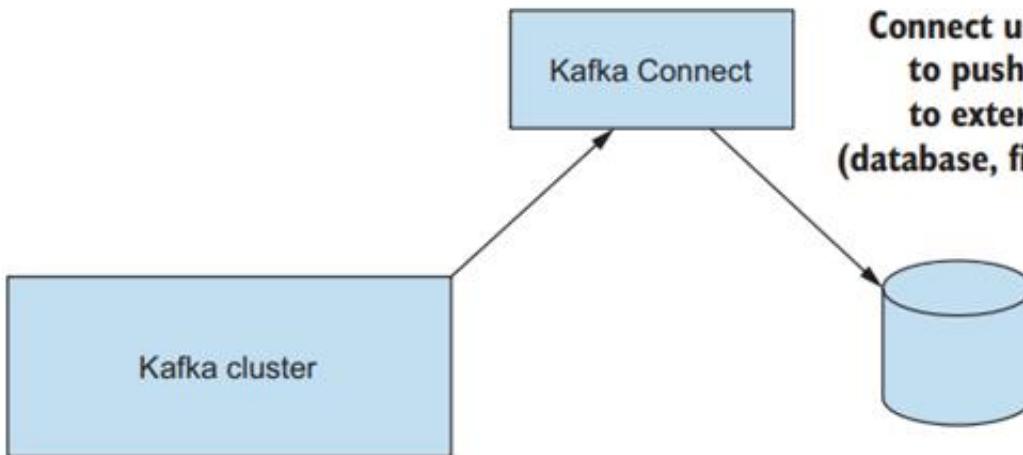
Setting up Kafka Connect

- Kafka Connect runs in two flavors: distributed mode and standalone mode.
- For most production environments, running in distributed mode makes sense, because you can take advantage of the parallelism and fault tolerance available when you run multiple Connect instances.
- I'm assuming you'll run the examples on your local machine, so everything is configured in standalone mode.

Connect uses source connectors to pull from external data sources (database, filesystem, and so on) to push data to Kafka.



Connect uses sink connectors to push data from Kafka to external data sources (database, filesystem, and so on).



Setting up Kafka Connect

- Most of these configurations are straightforward, but we need to discuss two of them—mode and incrementing.column.name—in a little more detail, because they play an active role in how the connector runs.
- The JDBC source connector uses mode to detect which rows it needs to load.

Transforming data

- Before getting this new assignment, you had already developed a Kafka Streams application using similar data.
- As a result, you have an existing model and Serde objects (using Gson underneath for JSON serialization and deserialization).
- To keep your development velocity high, you'd prefer not to write any new code to support working with Connect.

Original column names

Transforms take the original column names and map them to the new configured names.

SYMB
SCTR
INDSTY



Connect transforms

SYMB:symbol, SCTR:sector, INDSTY:industry,...

The JSON message sent to Kafka topics now has the expected field names needed for deserialization.

{"symbol": "xxxx", "sector": "xxxxx", "industry": "xxxxx", ...}

Transforming data

Listing 9.1 JDBC connector properties

Date field to convert

```
transforms=ConvertDate,Rename  
transforms.ConvertDate.type=  
  ↗ org.apache.kafka.connect.transforms.TimestampConverter$Value  
transforms.ConvertDate.field=TXNTS  
transforms.ConvertDate.target.type=string  
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400  
transforms.Rename.type=  
  ↗ org.apache.kafka.connect.transforms.ReplaceField$Value  
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
```

Aliases for the transformers

Type for the ConvertDate alias

Output type of the converted date field

Format of the date

Type for the Rename alias

List of column names (truncated for clarity) to replace. The format is Original:Replacement.

Transforming data

Listing 9.2 Kafka Streams source topic populated with data from Connect

```
Serde<StockTransaction> stockTransactionSerde =  
    StreamsSerdes.StockTransactionSerde();  
  
StreamsBuilder builder = new StreamsBuilder();  
builder.stream("dbTxnTRANSACTIONS",  
    Consumed.with(stringSerde, stockTransactionSerde))  
    .peek((k, v) ->  
        LOG.info("transactions from database key {}value {}", k, v));
```

Serde for the StockTransaction object

Uses the topic Connect writes to as the source for the stream

Prints messages out to the console

Transforming data

Listing 9.3 Updated JDBC connector properties

```
transforms=ConvertDate,Rename,ExtractKey
transforms.ConvertDate.type=
  ↪ org.apache.kafka.connect.transforms.TimestampConverter$Value
transforms.ConvertDate.field=TXNTS
transforms.ConvertDate.target.type=string
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400
transforms.Rename.type=
  ↪ org.apache.kafka.connect.transforms.ReplaceField$Value
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
transforms.ExtractKey.type=
  ↪ org.apache.kafka.connect.transforms.ValueToKey
transforms.ExtractKey.fields=symbol
```

← Adds the ExtractKey transform

← Specifies the class name of the ExtractKey transform

← Lists the field(s) to extract for the key

Transforming data

Listing 9.4 Adding a transform

```
transforms=ConvertDate,Rename,ExtractKey,FlattenStruct           ← Adds  
transforms.ConvertDate.type=                                     the last  
  ↪ org.apache.kafka.connect.transforms.TimestampConverter$Value  
transforms.ConvertDate.field=TXNTS  
transforms.ConvertDate.target.type=string  
transforms.ConvertDate.format=yyyy-MM-dd'T'HHI:mm:ss.SSS-0400  
transforms.Rename.type=                                         class  
  ↪ org.apache.kafka.connect.transforms.ReplaceField$Value  
transforms.renames=SMBL:symbol, SCTR:sector,...  
transforms.ExtractKey.type=org.apache.kafka.connect.transforms.ValueToKey  
transforms.ExtractKey.fields=symbol  
transforms.FlattenStruct.type=                                     ← Specifies the class  
  ↪ org.apache.kafka.connect.transforms.ExtractField$Key          for the transform  
transforms.FlattenStruct.field=symbol                            (ExtractField$Key)  
                                                               ← Name of the field  
                                                               to pull out
```

Listing 9.5 Processing transactions from a table in Kafka Streams via Connect

```
Serde<StockTransaction> stockTransactionSerde =  
    StreamsSerdes.StockTransactionSerde();  
StreamsBuilder builder = new StreamsBuilder();  
builder.stream("dbTxnTRANSACTIONS",  
    Consumed.with(stringSerde, stockTransactionSerde))  
    .peek((k, v) ->  
        LOG.info("transactions from database key {} value {}", k, v))  
        .groupByKey()  
    Serialized.with(stringSerde, stockTransactionSerde))  
        .aggregate(() -> 0L, (symb, stockTxn, numShares) ->  
            numShares + stockTxn.getShares(),  
            Materialized.with(stringSerde, longSerde)).toStream()  
    .peek((k, v) -> LOG.info("Aggregated stock sales for {} {}", k, v))  
    .to("stock-counts", Produced.with(stringSerde, longSerde));
```

Groups by key

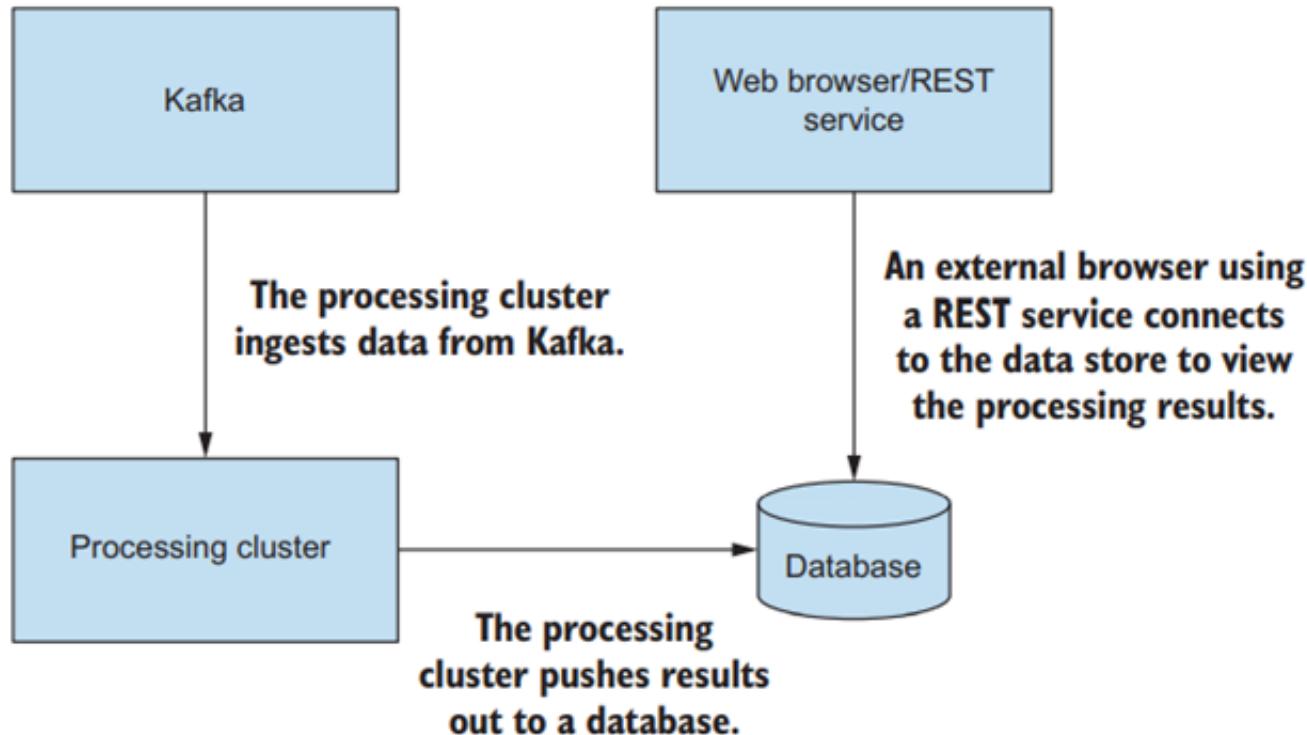
Performs an aggregation of the total number of shares sold

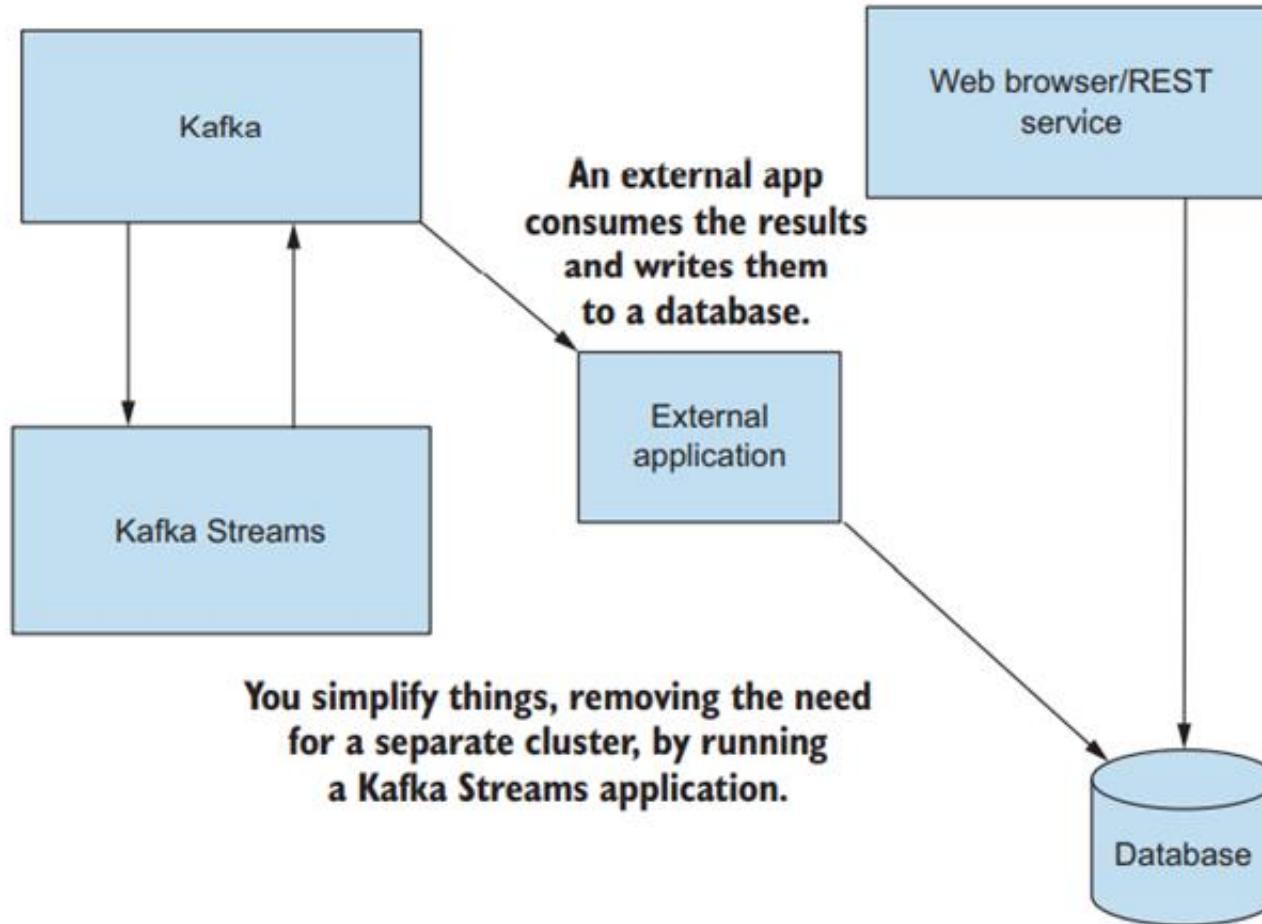
Kicking your database to the curb

Going back to the requirements from BSE, you've developed a Kafka Streams application that captures three categories of equity transactions:

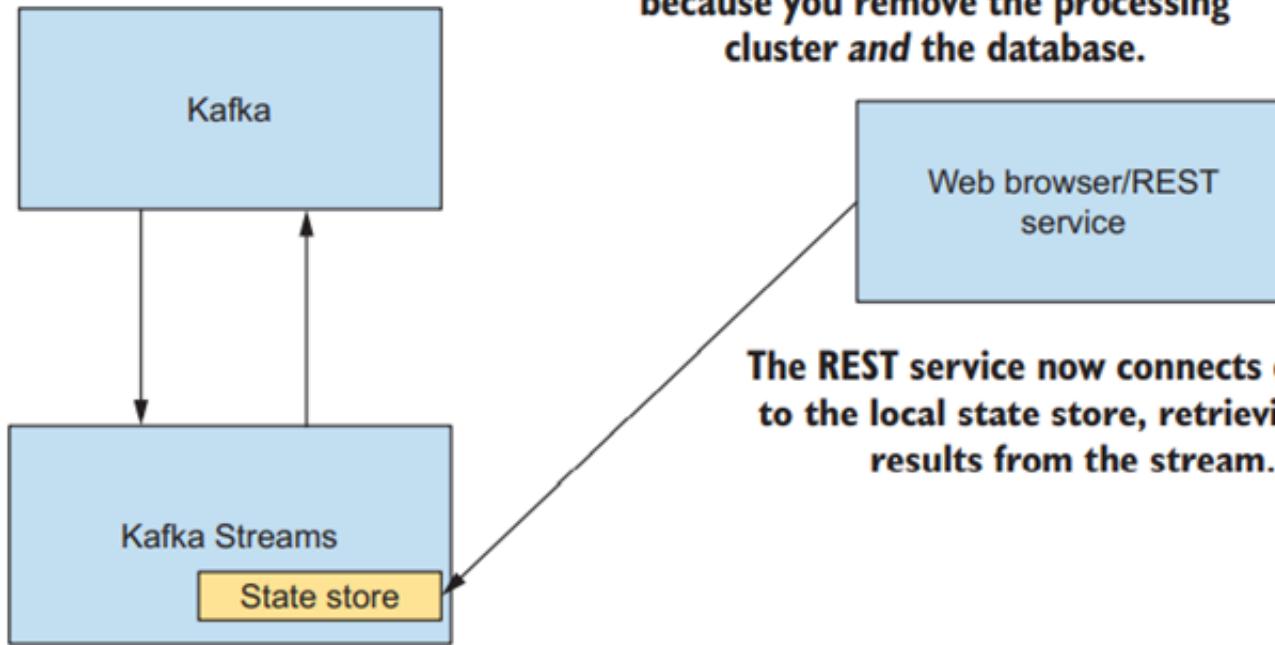
- Total transactions by market sector
- Customer purchases of shares per session
- Total number of shares traded by stock symbol, in tumbling windows of 10 seconds

Kicking your database to the curb





**Now, you really simplify your architecture,
because you remove the processing
cluster and the database.**



**The REST service now connects directly
to the local state store, retrieving live
results from the stream.**

**The Kafka Streams application consumes
from the broker, and local state captures
the current state of the stream.**

How interactive queries work

- For interactive queries to work, Kafka Streams exposes state stores in a read-only wrapper.
- It's important to understand that while Kafka Streams makes the stores available for queries, there's no updating or modifying the state store in any way.
- Kafka Streams exposes state stores from the `KafkaStreams.store` method.

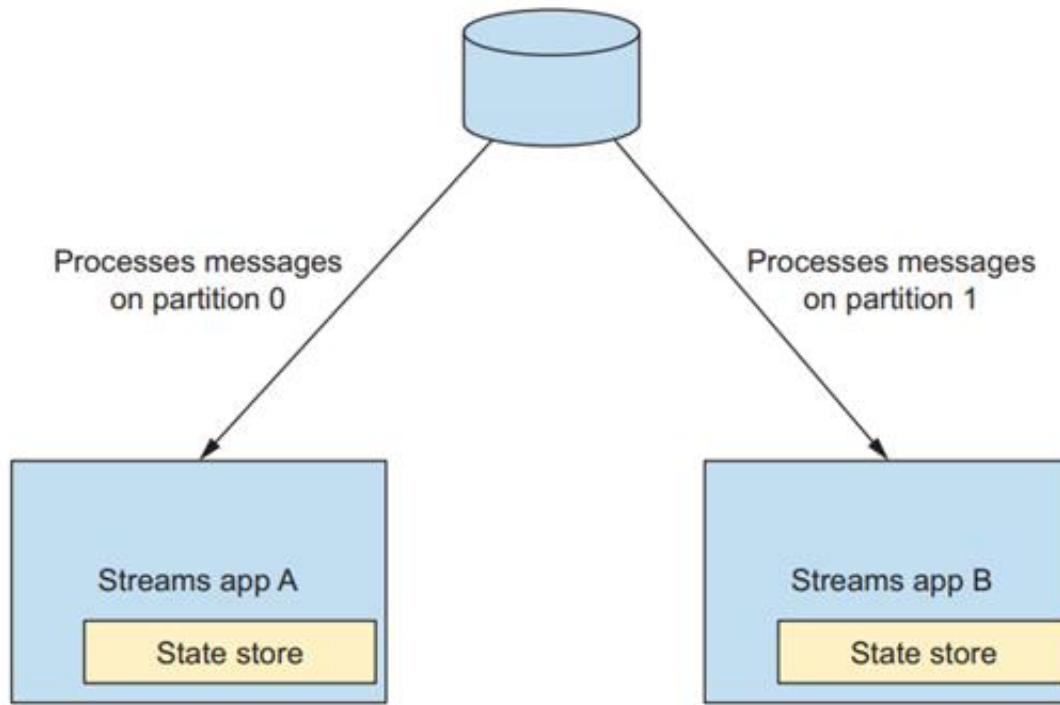
Here's how this method works

```
ReadOnlyWindowStore readOnlyStore =  
    kafkaStreams.store(storeName, QueryableStoreTypes.windowStore());
```

Distributing state stores

- Consider the first analytic: aggregating stock trades per market sector.
- Because you're doing an aggregation, state stores come into play.
- You want to expose the state stores to provide a real-time view of the number of trades per sector, to gain insight into which segment of the market is seeing the most action at the moment.

Kafka topic T with two partitions

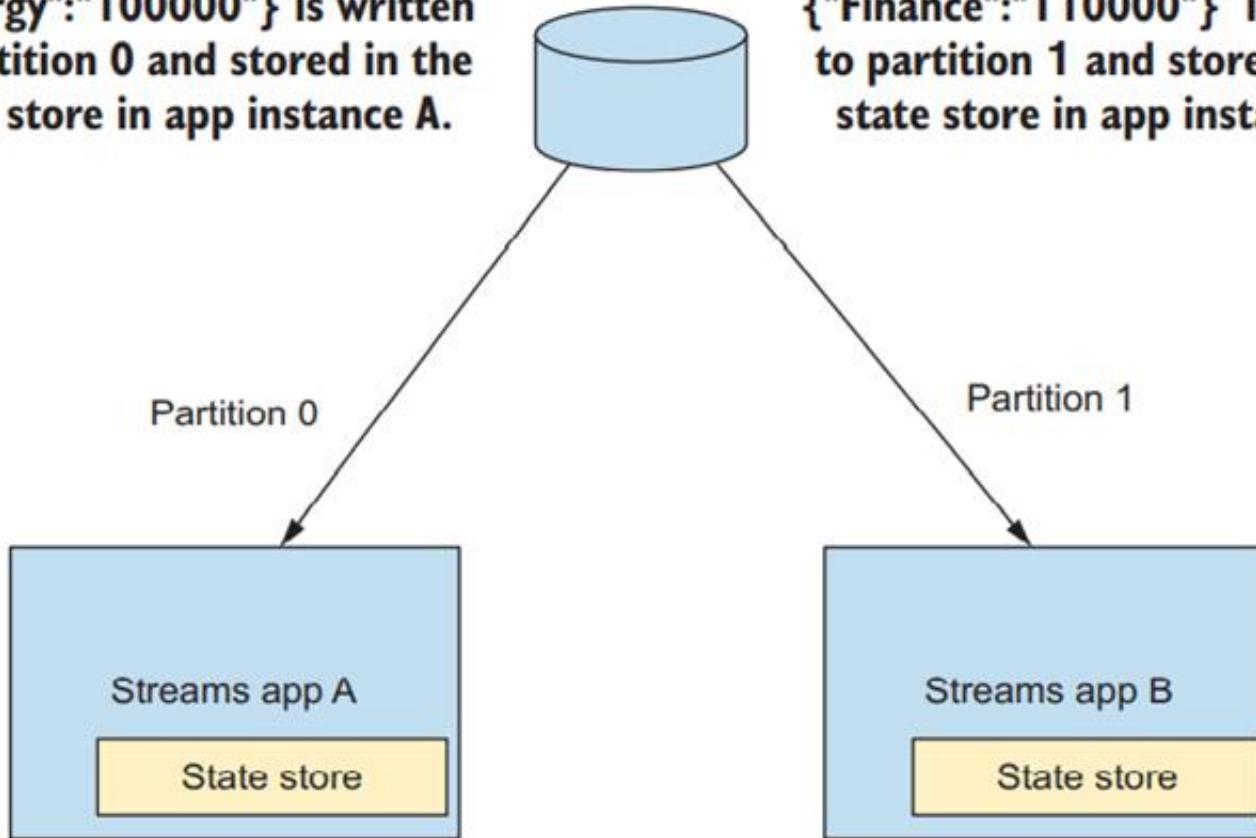


The assigned task for Kafka Streams application instance A is TopicPartition T0.

The assigned task for Kafka Streams application instance B is TopicPartition T1.

{"Energy": "100000"} is written to partition 0 and stored in the state store in app instance A.

{"Finance": "110000"} is written to partition 1 and stored in the state store in app instance B.



Setting up and discovering a distributed state store

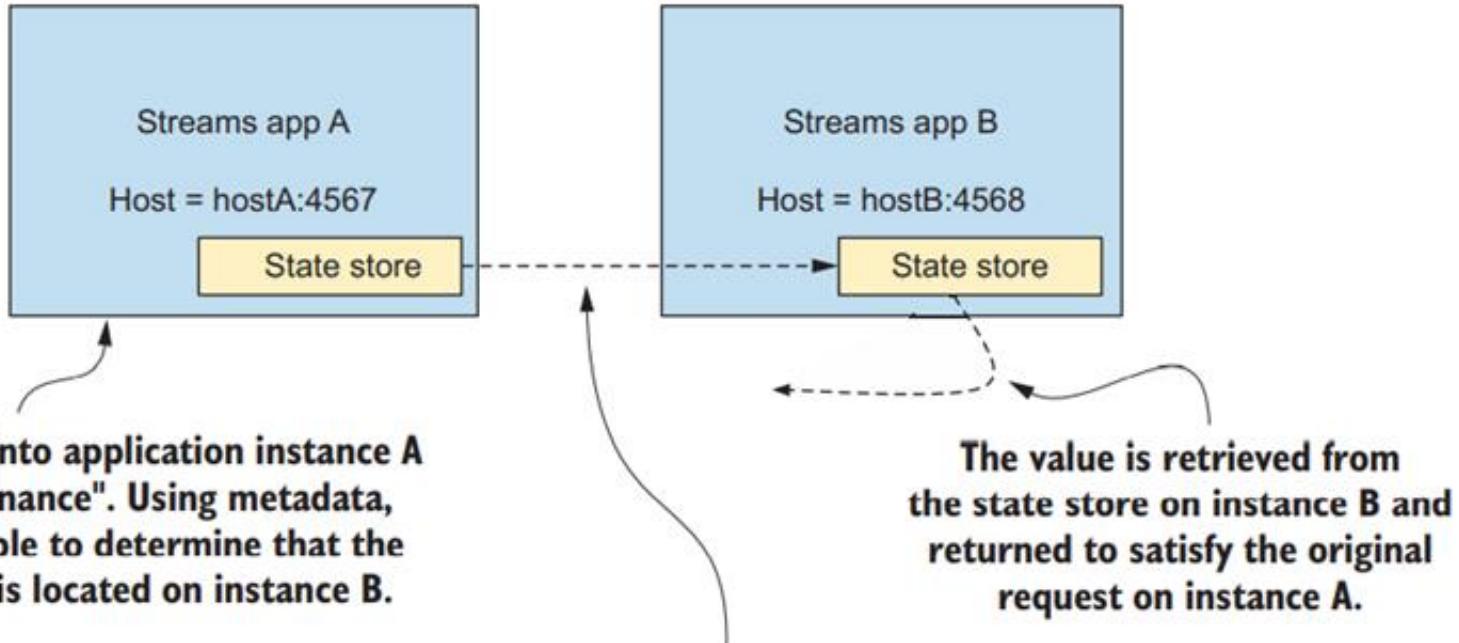
- To enable interactive queries, you need to set the StreamsConfig.APPLICATION_SERVER_CONFIG parameter.
- It consists of the hostname of the Kafka Streams application and the port that a query service will listen on, in hostname:port format.
- When a Kafka Streams instance receives a query for a given key, you'll need to find out whether the key is contained in the local store.

Setting up and discovering a distributed state store

Name	Parameter(s)	Usage
allMetadata	N/A	All instances, some possibly remote
allMetadataForStore	Store name	All instances (some remote) containing the named store
allMetadataForKey	Key, Serializer	All instances (some remote) with the store containing the key
allMetadataForKey	Key, StreamPartitioner	All instances (some remote) with the store containing the key

{"Energy": "100000"}

{"Finance": "110000"}



A query comes into application instance A for the key "Finance". Using metadata, instance A is able to determine that the key "Finance" is located on instance B.

The value is retrieved from the state store on instance B and returned to satisfy the original request on instance A.

Instance A uses host and port info from metadata to retrieve the value for "Finance" from instance B.

Listing 9.6 Setting the hostname and port

```
public static void main(String[] args) throws Exception {  
  
    if(args.length < 2){  
        LOG.error("Need to specify host and port");  
        System.exit(1);  
    }  
  
    String host = args[0];  
    int port = Integer.parseInt(args[1]);  
    final HostInfo hostInfo = new HostInfo(host, port);  
  
    Properties properties = getProperties();  
    properties.put(  
        StreamsConfig.APPLICATION_SERVER_CONFIG, host+":"+port);  
  
    // other details left out for clarity
```

Creates a HostInfo object for later use in the application

Sets the config for enabling interactive queries

Listing 9.7 Initializing the web server and setting its status

```
    Adds a StateListener to only enable
    queries to state stores until ready

    // details left out for clarity

    KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
    InteractiveQueryServer queryServer =
        new InteractiveQueryServer(kafkaStreams, hostInfo);
    queryServer.init();

    kafkaStreams.setStateListener(((newState, oldState) -> {
        if (newState == KafkaStreams.State.RUNNING && oldState ==
            KafkaStreams.State.REBALANCING) {
            LOG.info("Setting the query server to ready");
            queryServer.setReady(true);
        } else if (newState != KafkaStreams.State.RUNNING) {
            LOG.info("State not RUNNING, disabling the query server");
            queryServer.setReady(false);
        }
    }));
    kafkaStreams.setUncaughtExceptionHandler((t, e) -> {
        LOG.error("Thread {} had a fatal error {}", t, e, e);
        shutdown(kafkaStreams, queryServer);
    });

    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        shutdown(kafkaStreams, queryServer);
    }));
}

Creates the embedded
web server (actually a
wrapper class)

Enables queries to
state stores once
the Kafka Streams
application is in a
RUNNING state.
Queries are
disabled if the state
isn't RUNNING.

Sets an Uncaught-
ExceptionHandler to
log unexpected errors
and close everything
down

Adds a shutdown
hook to close
everything down
when the application
exits normally
```

Inside the query server

Listing 9.8 Mapping URL paths to methods

```
public void init() {  
    LOG.info("Started the Interactive Query Web server");  
  
    get("/kv/:store", (req, res) -> ready ?  
        ↪ fetchAllFromKeyValueStore(req.params()) :  
        ↪ STORES_NOT_ACCESSIBLE);  
    get("/session/:store/:key", (req, res) -> ready ?  
        ↪ fetchFromSessionStore(req.params()) :  
        ↪ STORES_NOT_ACCESSIBLE);  
    get("/window/:store/:key", (req, res) -> ready ?  
        ↪ fetchFromWindowStore(req.params()) :  
        ↪ STORES_NOT_ACCESSIBLE);  
    get("/window/:store/:key/:from/:to", (req, res) -> ready ?  
        ↪ fetchFromWindowStore(req.params()) :  
        ↪ STORES_NOT_ACCESSIBLE);  
}
```

- Annotation 1: "Mapping to retrieve all values from a plain key/value store" points to the first mapping for "/kv/:store".
- Annotation 2: "Mapping to return all sessions (from a session store) for a given key" points to the second mapping for "/session/:store/:key".
- Annotation 3: "Mapping for a window store with no times specified" points to the third mapping for "/window/:store/:key".
- Annotation 4: "Mapping for a window store with from and to times" points to the fourth mapping for "/window/:store/:key/:from/:to".

CHECKING FOR THE STATE STORE LOCATION

- You'll use the following mapping to walk through the example, from reviewing the request to returning the response:

```
get ("/window/:store/:key", (req, res) -> ready ?  
  ➔ fetchFromWindowStore(req.params()) : STORES_NOT_ACCESSIBLE);
```

Listing 9.9 Mapping the request and checking for the key location

```
private String fetchFromWindowStore(Map<String, String> params) {  
    String store = params.get(STORE_PARAM);  
    String key = params.get(KEY_PARAM);  
    String fromStr = params.get(FROM_PARAM);  
    String toStr = params.get(TO_PARAM);  
  
    HostInfo storeHostInfo = getHostInfo(store, key);  
  
    if(storeHostInfo.host().equals("unknown")) {  
        return STORES_NOT_ACCESSIBLE;  
    }  
  
    if(dataNotLocal(storeHostInfo)) {  
        LOG.info("{} located in state store on another instance", key);  
        return fetchRemote(storeHostInfo, "window", params);  
    }  
  
    Extracts the request parameters  
    Gets the HostInfo for the key  
    If the hostname is "unknown", returns an appropriate message  
    Checks whether the returned hostname matches the host of this instance
```

Listing 9.10 Retrieving and formatting the results

```
Instant instant = Instant.now();           ← Gets the current time in milliseconds
long now = instant.toEpochMilli();          ← Sets the window segment start time or, if not provided, the time as of one minute ago
long from = fromStr != null ? Long.parseLong(fromStr) : now - 60000;
long to = toStr != null ? Long.parseLong(toStr) : now;           ← Sets the window segment ending time or, if not provided, the current time

List<Integer> results = new ArrayList<>();

ReadOnlyWindowStore<String, Integer> readOnlyWindowStore =
    kafkaStreams.store(store,
    QueryableStoreTypes.windowStore());           ← Retrieves the ReadOnlyWindowStore

try(WindowStoreIterator<Integer> iterator =
    readOnlyWindowStore.fetch(key, from , to)){   ← Fetches the window segments
    while (iterator.hasNext()) {
        results.add(iterator.next().value);         ← Builds up the response
    }
}
return gson.toJson(results);                ← Converts the results to JSON and returns to the requestor
```

RUNNING THE INTERACTIVE QUERY EXAMPLE

To observe the results of this example, you need to run three commands:

- `./gradlew runProducerInteractiveQueries` produces the data needed for the examples.
- `./gradlew runInteractiveQueryApplicationOne` starts a Kafka Streams application with HostInfo using port 4567.
- `./gradlew runInteractiveQueryApplicationTwo` starts a Kafka Streams application with HostInfo using port 4568.

RUNNING A DASHBOARD APPLICATION FOR INTERACTIVE QUERIES

- A better example is a mini-dashboard web application that updates automatically (via Ajax) and displays the results from four different Kafka Streams aggregation operations.
- By running the commands listed in the previous subsection, you have everything set up; point your browser to localhost:4568/iq or localhost:4567/iq to run the dashboard application.

KSQL

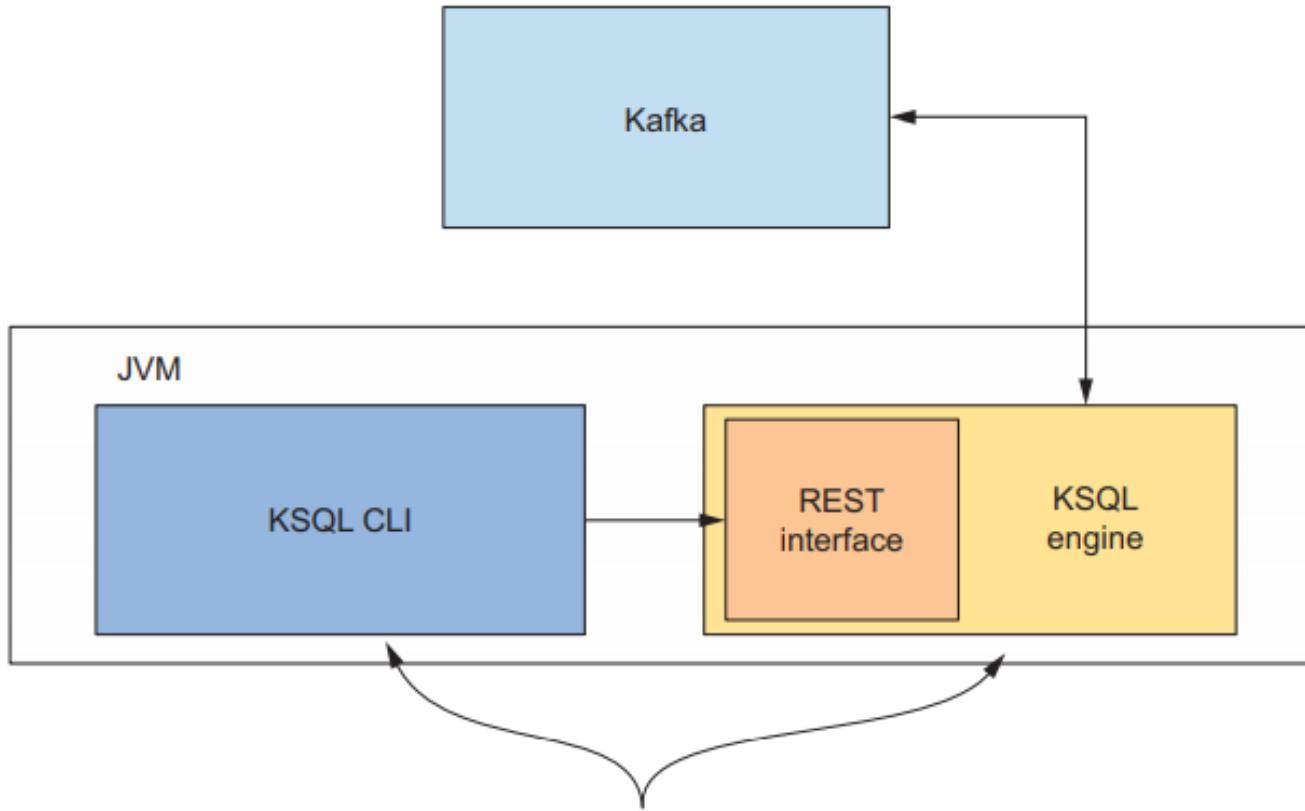
- Imagine you're working with business analysts at BSE.
- The analysts are interested in your ability to quickly write applications in Kafka Streams to perform real-time data analysis & This interest puts you in a bind.
- You want to work with the analysts and write applications for their requests, but you also have your normal workload—the additional work makes it hard to keep up with everything.

KSQL streams and tables

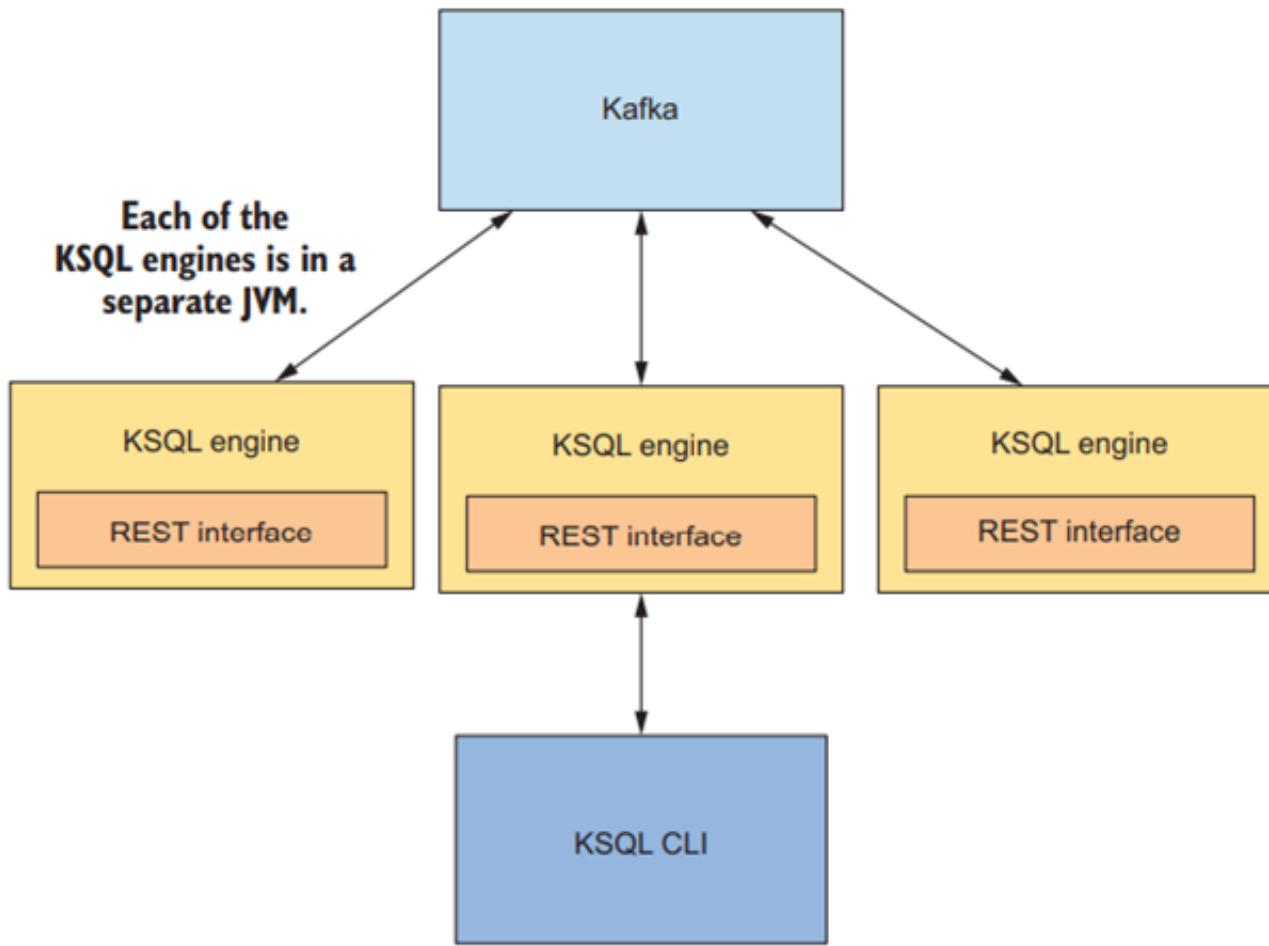
- We discussed the concept of an event stream versus an update stream.
- An event stream is an unbounded stream of individual independent events, where an update or record stream is a stream of updates to previous records with the same key.
- KSQL has a similar concept of querying from a Stream or a Table.

KSQL architecture

- KSQL uses Kafka Streams under the covers to build and fetch the results of queries. KSQL is made up of two components: a CLI and a server.
- Users of standard SQL tools such as MySQL, Oracle, and even Hive will feel right at home with the CLI when writing queries in KSQL.
- Best of all, KSQL is open source (Apache 2.0 licensed).



**The KSQL CLI and KSQL engine
are in one JVM locally.**



Installing and running KSQL

- To install KSQL, you'll clone the KSQL repo with the command `git clone git@github.com:confluentinc/ksql.git` and then `cd` into the `ksql` directory and execute `mvn clean package` to build the entire KSQL project.
- If you don't have git installed or don't want to build from source, you can download the KSQL release from <http://mng.bz/765U>

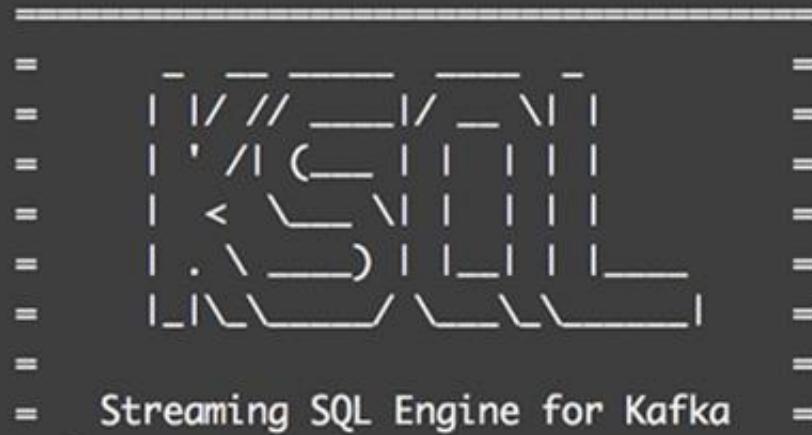
Installing and running KSQL

- Make sure you're in the base directory of the KSQL project before going any further.
- The next step is to start KSQL in local mode:

```
./bin/ksql-cli local
```

```
oddball:bin bbejeck$ ./ksql-cli local
```

```
Initializing KSQL...
```



```
= Streaming SQL Engine for Kafka =
```

```
Copyright 2017 Confluent Inc.
```

```
CLI v0.4, Server v0.4 located at http://localhost:9098
```

```
Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!
```

```
ksql> |
```

Creating a KSQL stream

- Getting back to your work at BSE, you've been approached by an analyst who is interested in one of the applications you've written and would like to make some tweaks to it.
- But instead of this request resulting in more work, you spin up a KSQL console and turn the analyst loose to reconstruct your application as a SQL statement!

Creating a KSQL stream

Listing 9.11 Creating a stream

CREATE STREAM statement to create a stream named stock_txn_stream

```
CREATE STREAM stock_txn_stream (symbol VARCHAR, sector VARCHAR, \
    industry VARCHAR, shares BIGINT, sharePrice DOUBLE, \
    customerId VARCHAR, transactionTimestamp STRING, purchase BOOLEAN) \
→ WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'stock-transactions');
```

Specifies the data format and the Kafka topic serving as the source of the stream (both required parameters)

Registers the fields of the StockTransaction object as columns

Creating a KSQL stream

- You can view all streams and verify that KSQL created the new stream as expected with the following commands:

```
show streams;  
describe stock_txn_stream;
```

```
ksql> show streams;
```

Stream Name	Kafka Topic	Format
-------------	-------------	--------

STOCK_TXN_STREAM	stock-transactions	JSON
------------------	--------------------	------

```
ksql> describe stock_txn_stream;
```

Field	Type
-------	------

ROWTIME	BIGINT
---------	--------

ROWKEY	VARCHAR(STRING)
--------	-----------------

SYMBOL	VARCHAR(STRING)
--------	-----------------

SECTOR	VARCHAR(STRING)
--------	-----------------

INDUSTRY	VARCHAR(STRING)
----------	-----------------

SHARES	BIGINT
--------	--------

SHAREPRICE	DOUBLE
------------	--------

CUSTOMERID	VARCHAR(STRING)
------------	-----------------

TRANSACTIONTIMESTAMP	VARCHAR(STRING)
----------------------	-----------------

PURCHASE	BOOLEAN
----------	---------

Writing a KSQL query

The SQL query for performing the stock analysis is as follows:

```
SELECT symbol, sum(shares) FROM stock_txn_stream  
→ WINDOW TUMBLING (SIZE 10 SECONDS) GROUP BY symbol;
```

Writing a KSQL query

ITZL		44694
KPAU		52858
NSTR		74110
ZERA		97959
MONA		29507
<u>MESG</u>		43474

Listing 9.12 Stock analysis application written in Kafka Streams

```
KStream<String, StockTransaction> stockTransactionKStream =  
    builder.stream(MockDataProducer STOCK_TRANSACTIONS_TOPIC,  
                    Consumed.with(stringSerde, stockTransactionSerde)  
                    .withOffsetResetPolicy(Topology.AutoOffsetReset.EARLIEST));  
  
Aggregator<String, StockTransaction, Integer> sharesAggregator =  
    (k, v, i) -> v.getShares() + i;  
  
stockTransactionKStream.groupByKey()  
    .windowedBy(TimeWindows.of(10000))  
    .aggregate(() -> 0, sharesAggregator,  
              Materialized.<String, Integer,  
              WindowStore<Bytes,  
              byte[]>>as("NumberSharesPerPeriod")  
              .withKeySerde(stringSerde)  
              .withValueSerde(Serdes.Integer()))  
    .toStream().  
    peek((k,v)->LOG.info("key is {} value is{}", k, v));
```

Creating a KSQL table

Listing 9.13 Creating a KSQL table

```
CREATE TABLE stock_txn_table (symbol VARCHAR, sector VARCHAR, \
    industry VARCHAR, shares BIGINT, \
    sharePrice DOUBLE, \
    customerId VARCHAR, transactionTimestamp \
    STRING, purchase BOOLEAN) \
    WITH (KEY='symbol', VALUE_FORMAT = 'JSON', \
    KAFKA_TOPIC = 'stock-transactions');
```

Creating a KSQL table

- A useful experiment is to pick a ticker symbol from the streaming stock-performance query, and then run the following queries in the KSQL console, and notice the difference in output:

```
select * from stock_txn_stream where symbol='CCLU';  
select * from stock_txn_table where symbol='CCLU';
```

Configuring KSQL

- You're free to override any settings as needed, and any of the stream, consumer, and producer configs that you can set for a Kafka Streams application are available.
- To view the properties that are currently set, run the show properties; command.
- As an example of setting a property, here's how you can change auto.offset.reset to earliest:

```
SET 'auto.offset.reset'='earliest';
```

Configuring KSQL

- But if you need to set several configurations, typing each one into the console isn't convenient.
- Instead, you can specify a configuration file on startup:

```
./bin/ksql-cli local --properties-file  
/path/to/configs.properties
```

Summary

- By using Kafka Connect, you can incorporate other data sources into your Kafka Streams applications.
- Interactive queries are a potent tool: they allow you to see data in a stream as it flows through your Kafka Streams application, without the need for a relational database.

COMPLETE LAB 9