



---

# Working with Apache Kafka for Developers

# Table of Content



## Session 1: Introduction to Streaming Systems

1. The genesis of big data: 7
2. Important concepts from MapReduce: 12
3. Introducing stream processing: 21
4. Kafka Streams as a graph of processing nodes: 34

# Table of Content



## Session 2: Introducing Kafka

1. Kafka Overview: 46
2. What is Kafka: 56
3. Kafka Architecture: 67
4. Comparing Kafka with other queue systems (JMS / MQ): 81
5. Kafka Topics: 86
6. Producing messages: 97
7. Consuming messages: 102
8. Using Kafka Single Node: 114
9. Kafka Cluster and Failover: 126
10. Kafka Ecosystem: 147

# Table of Content



## Session 3: Using Kafka APIs

1. Intro to Producers: 157
2. Advanced Producers: 173
3. About the App: 221
4. Producer Shutdown: 242
5. Kafka Low Level Design: 247
6. Log Compaction: 286
7. Introduction to Consumers: 298
8. Advanced Consumers: 322

# Table of Content



## Session 4: Kafka Streams API

1. The Streams Processor API: 368
2. Hello World for Kafka Streams: 371
3. Streams concepts: KStream / Ktable: 424
4. The relationship between streams and tables: 426
5. Record updates and KTable configuration: 434
6. Aggregations and windowing operations: 440

# Table of Content



## Session 5: Monitoring and troubleshooting kafka

1. Monitoring Tools Overview: 481
2. Monitoring Kafka: 495
3. Identifying performance bottlenecks: 512
4. Troubleshooting common kafka issues: 421

# Session 1

# Introduction to Streaming Systems

# Welcome to Kafka Streams

This lesson covers

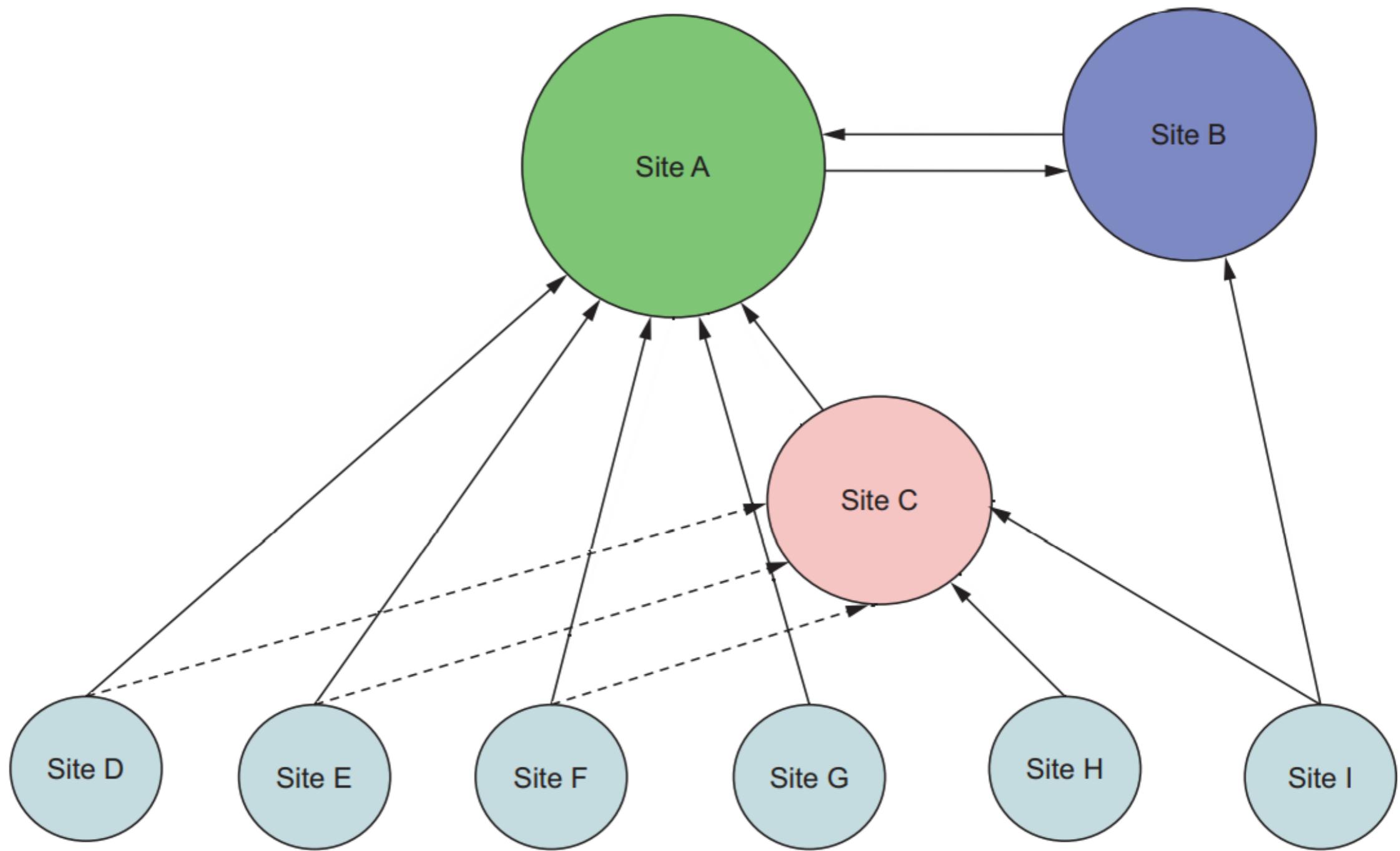
- Understanding how the big data movement changed the programming landscape
- Getting to know how stream processing works and why we need it
- Introducing Kafka Streams
- Looking at the problems solved by Kafka Streams

# 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.

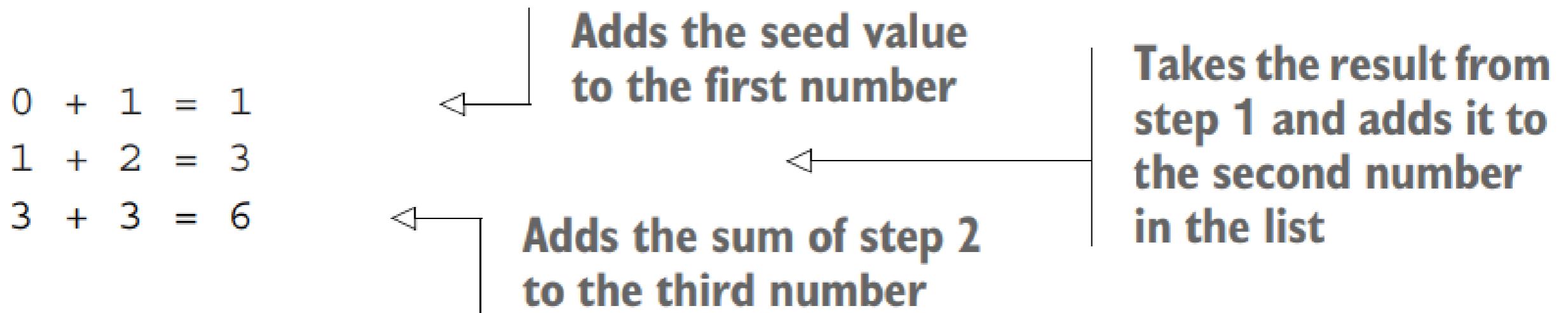
# Important concepts from MapReduce

- 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:



# 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

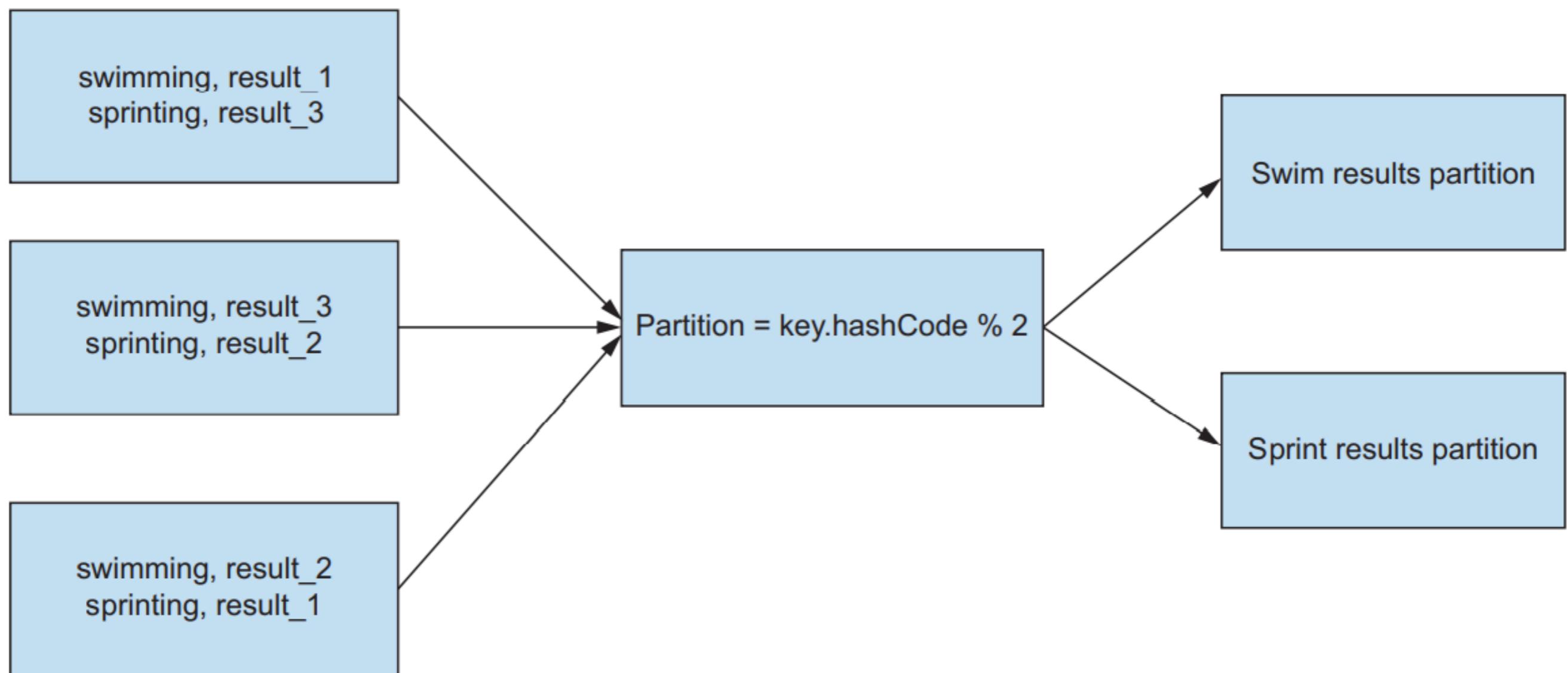
Number of machines	Amount of data processed per server
10	500 GB
100	50 GB
1000	5 GB
5000	1 GB

# 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

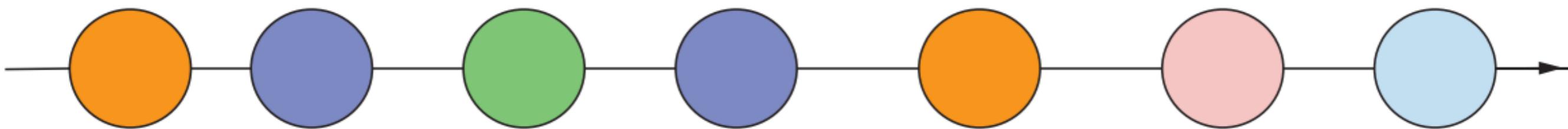


Figure represents a stream of data, with each circle on the line representing data at a point in time. Data is continuously flowing, as data in stream processing is unbounded

# 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

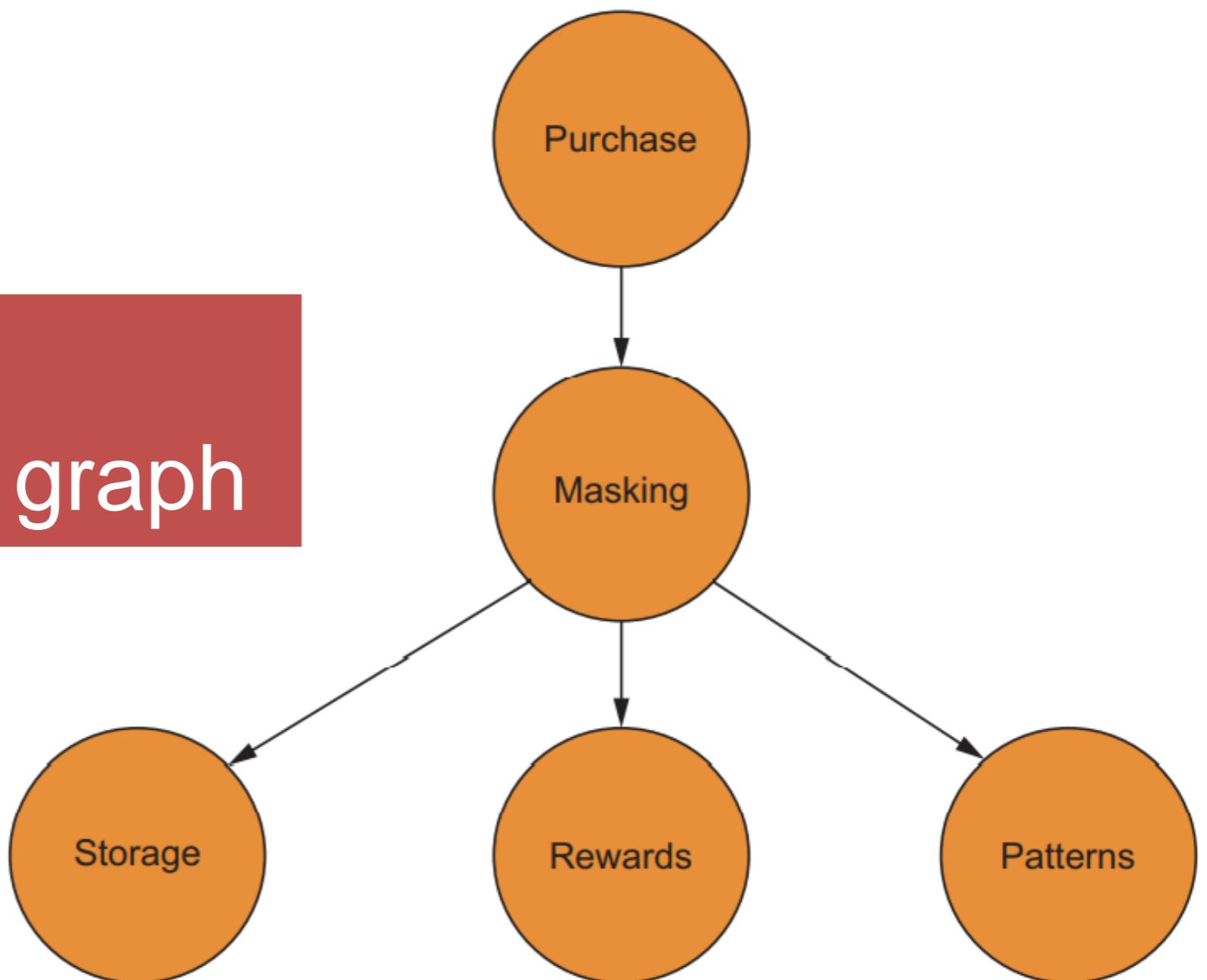
- Suppose you're the lead developer for ZMart's streaming-data team.
- ZMart is a big box retail store with several locations across the country.
- ZMart does great business, with total sales for any given year upwards of \$1 billion.
- You'd like to start mining the data from your company's transactions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

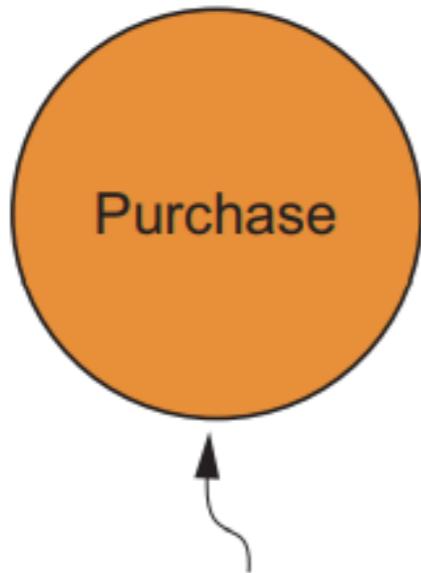
# Deconstructing the requirements into a graph



# 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.

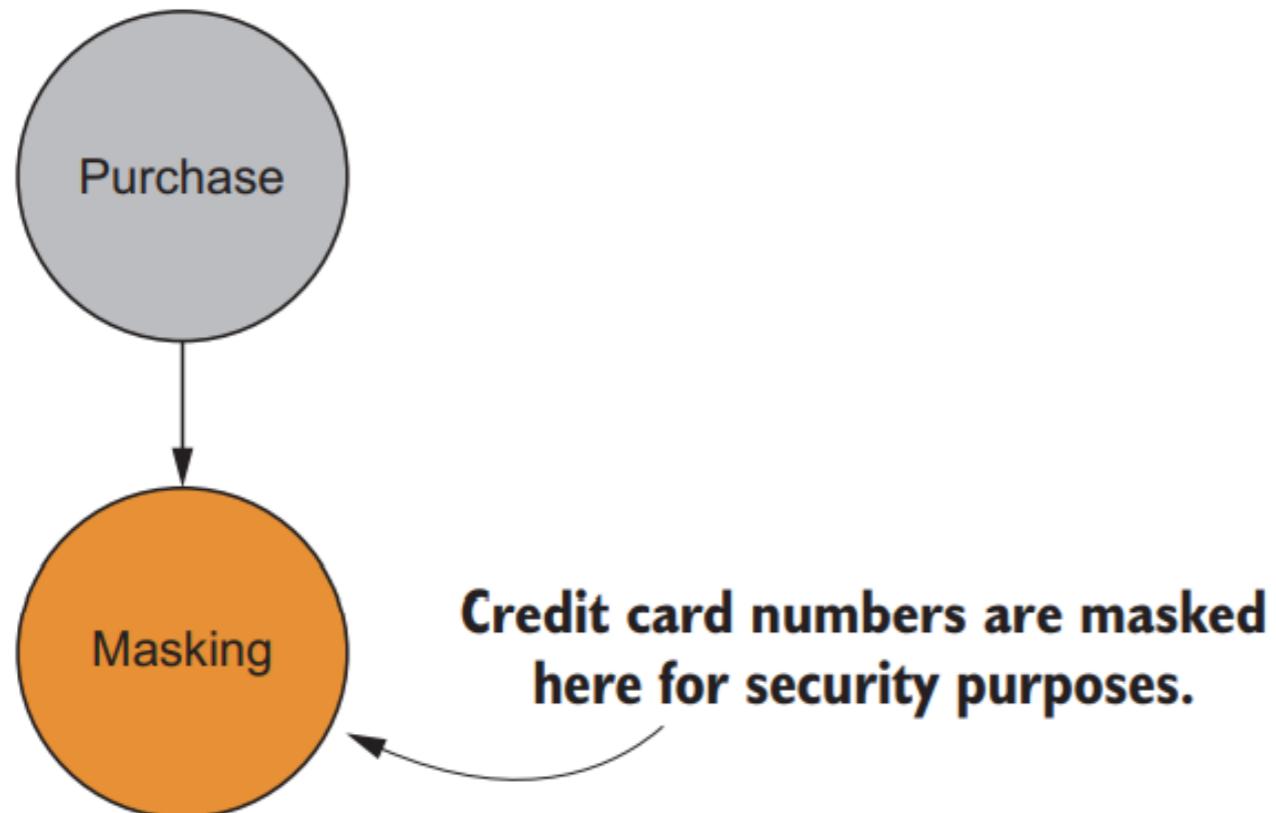
# Source node



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

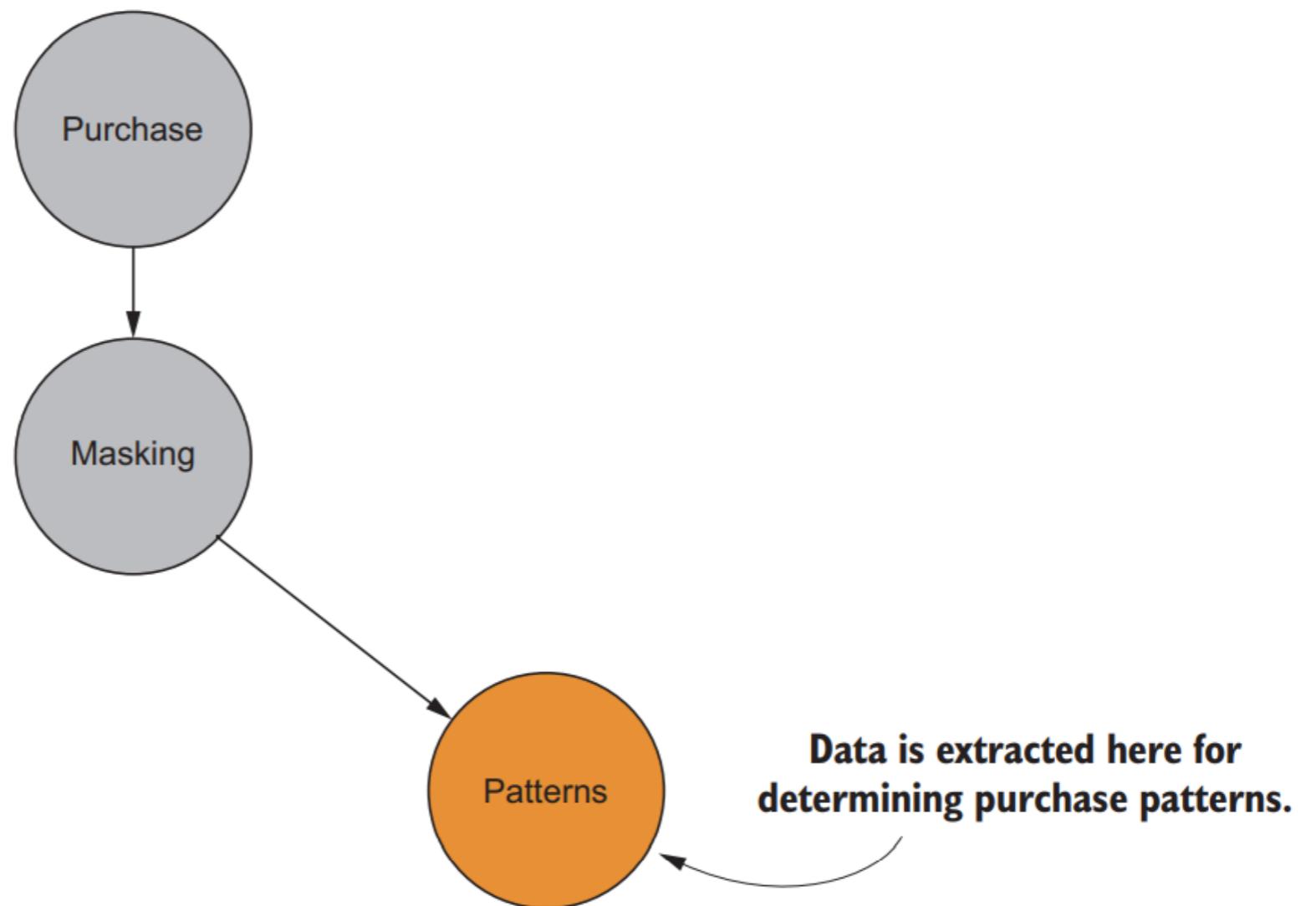
**Figure** The simple start for the sales transaction graph. This node is the source of raw sales transaction information that will flow through the graph.

# Credit card masking node



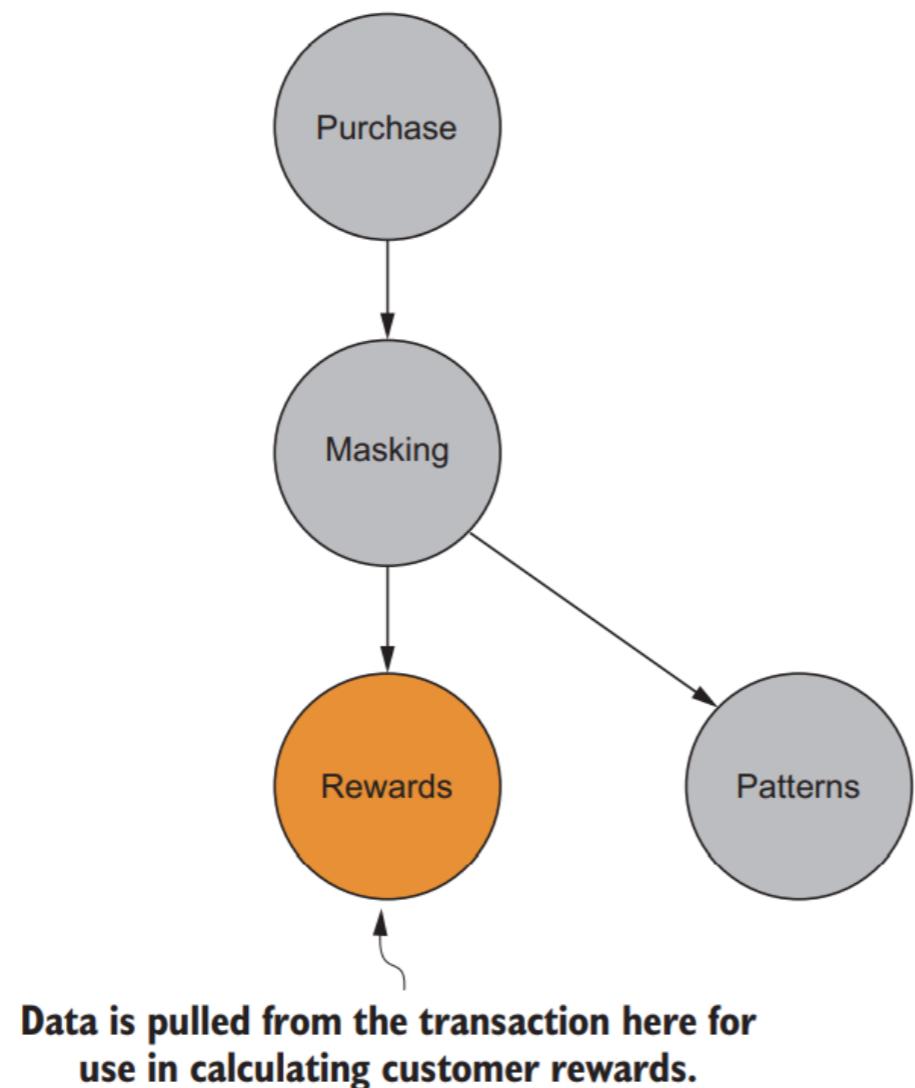
**Figure** The first node in the graph that represents the business requirements. This node is responsible for masking credit card numbers and is the only node that receives the raw sales data from the source node, effectively making it the source for all other nodes connected to it.

## Patterns node

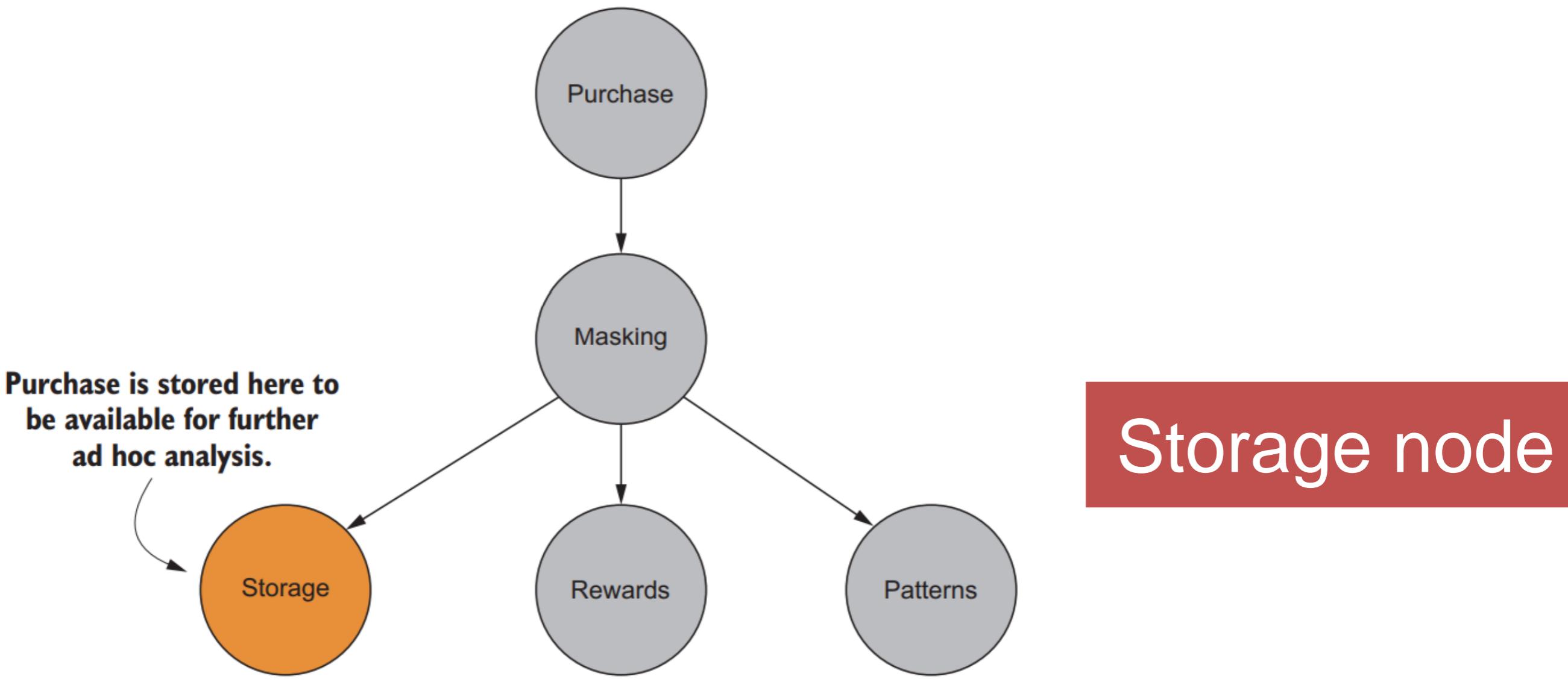


**Figure** The patterns node consumes purchase information from the masking node and converts it into a record showing when a customer purchased an item and the ZIP code where the customer completed the transaction.

# 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.



**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

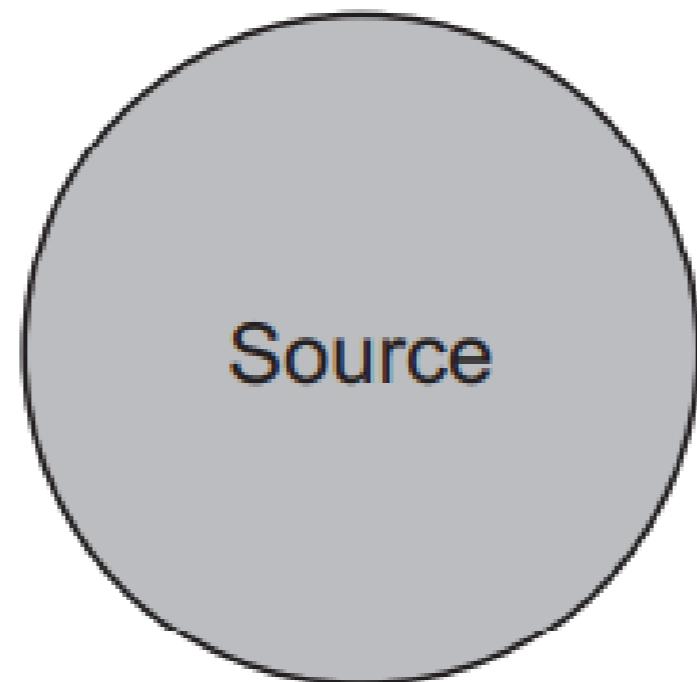
# Defining the source

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

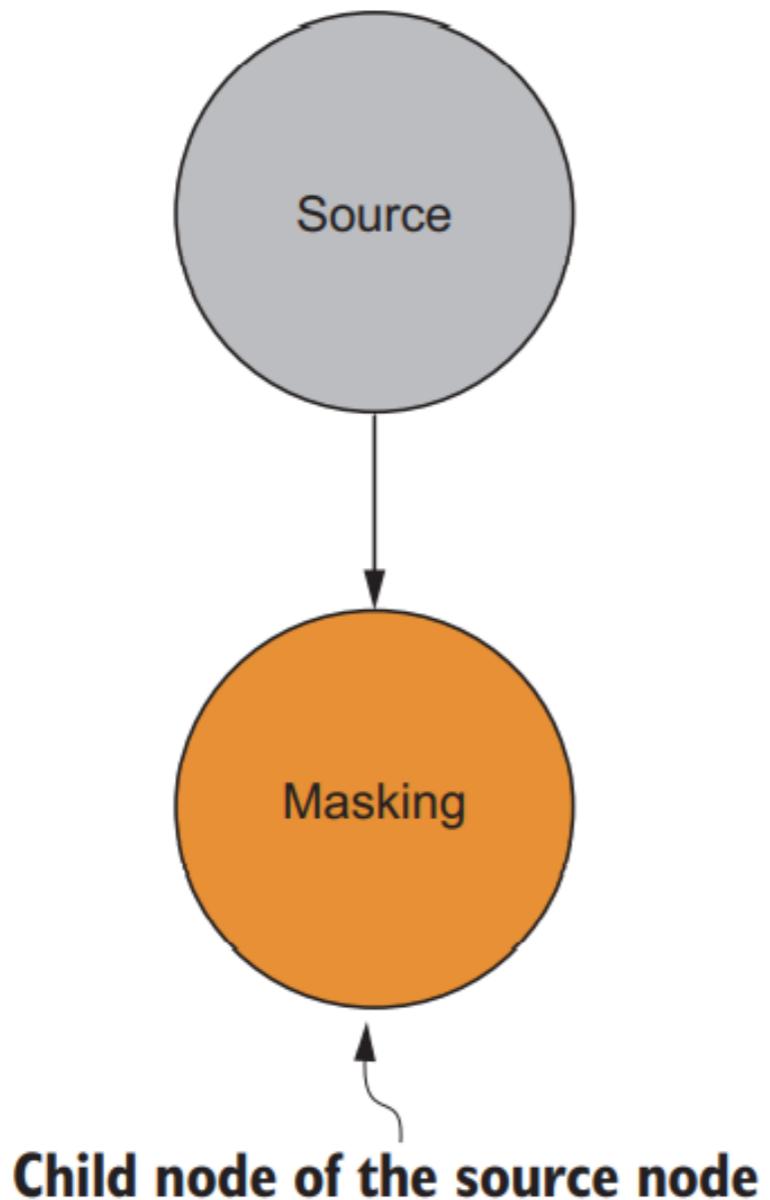
Figure represents the source node in the topology.



# 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**

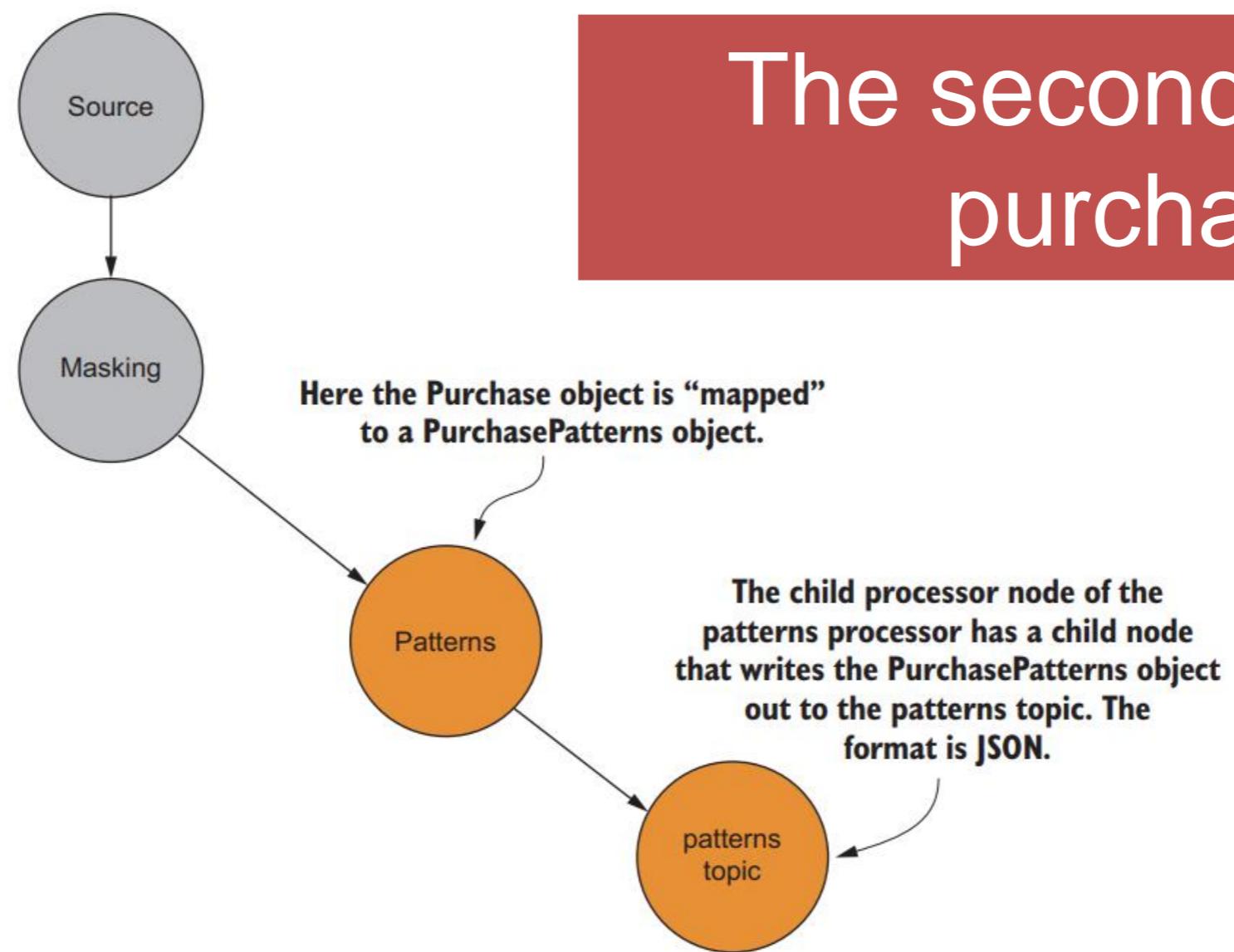


**Figure** The masking processor is a child of the main source node. It receives all the raw sales transactions and emits new records with the credit card number masked.

# 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

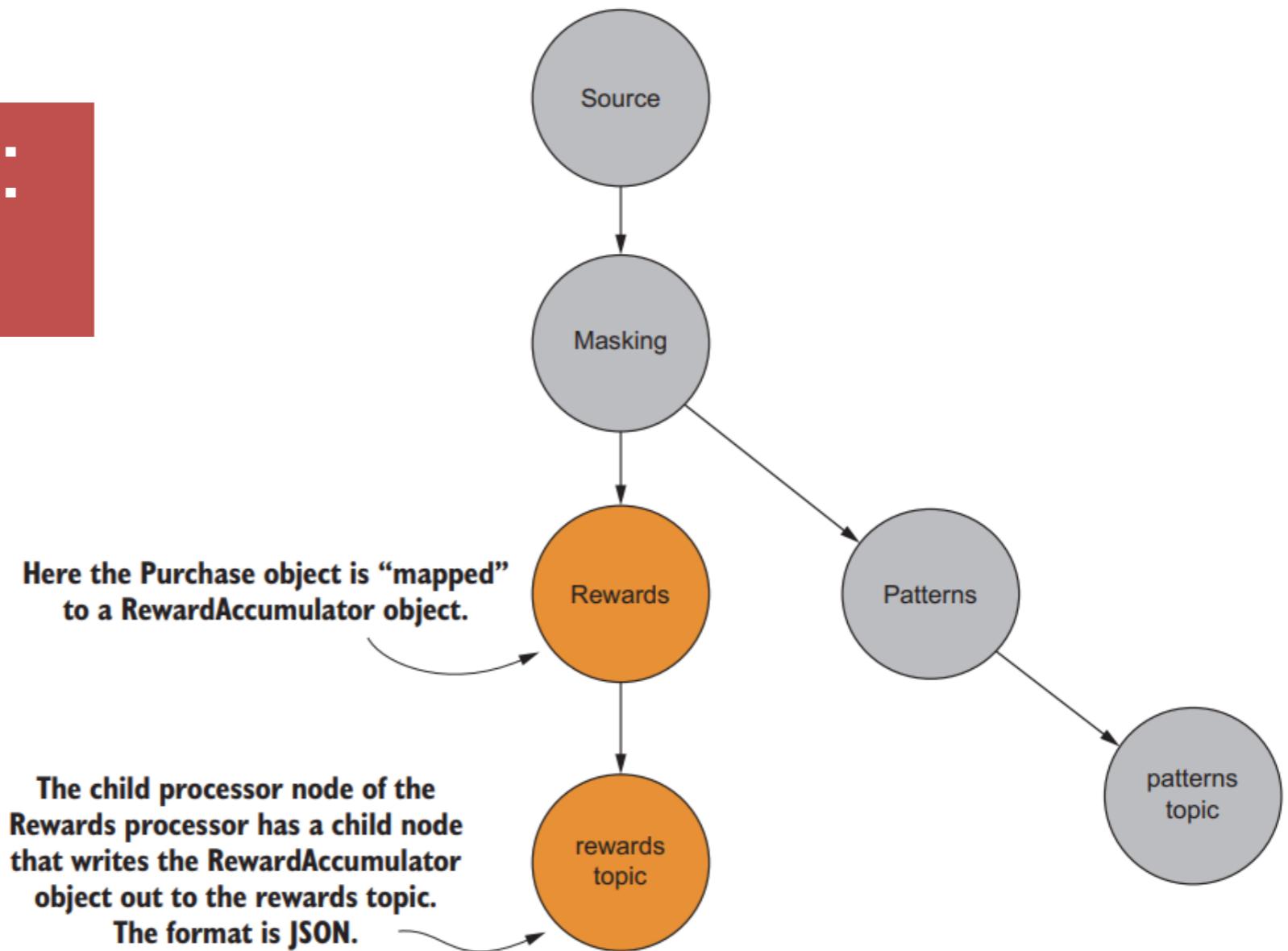
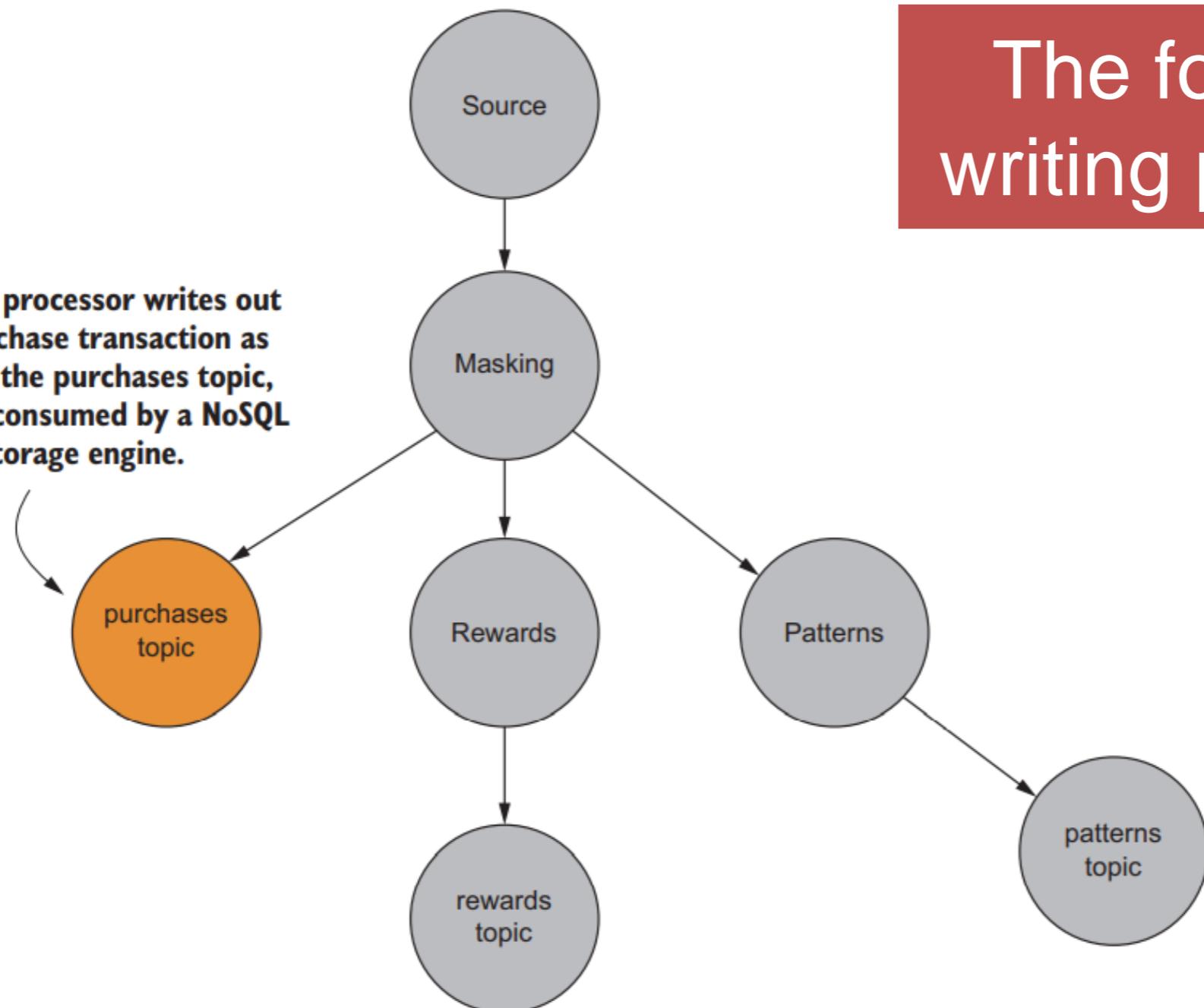


Figure The customer rewards processor is responsible for transforming Purchase objects into a RewardAccumulator object containing the customer ID, date, and dollar amount of the transaction. A child processor writes the Rewards objects to another Kafka topic.

## 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

# Session 2

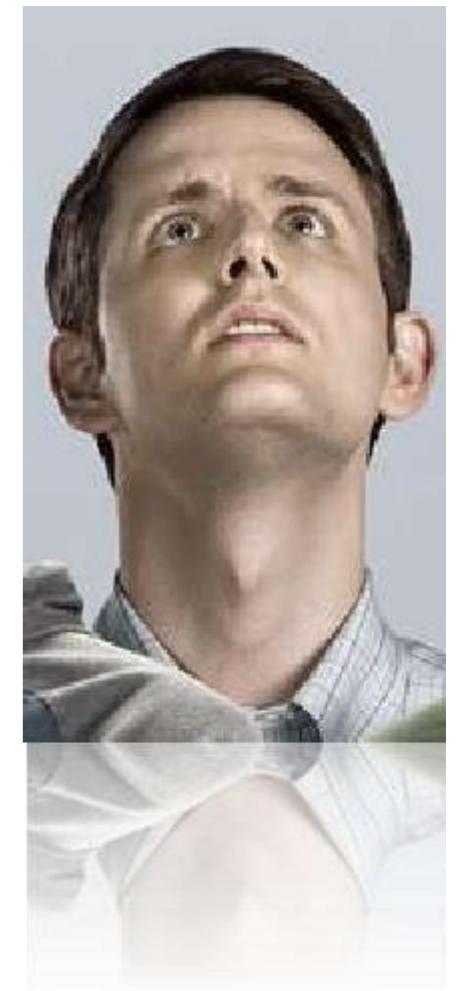
# Introducing Kafka

# Kafka Overview

# Kafka growth exploding

- ❖ Kafka growth exploding
- ❖ 1/3 of all Fortune 500 companies
- ❖ Top ten travel companies, 7 of ten top banks, 8 of ten top insurance companies, 9 of ten top telecom companies
- ❖ LinkedIn, Microsoft and Netflix process 4 comma message a day with Kafka (1,000,000,000,000)
- ❖ Real-time streams of data, used to collect big data or to do real time analysis (or both)

4  
commas!



# Why Kafka is Needed?

- ❖ Real time streaming data processed for real time analytics
  - ❖ Service calls, track every call, IOT sensors
- ❖ Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
- ❖ Kafka is often used instead of JMS, RabbitMQ and AMQP
  - ❖ higher throughput, reliability and replication

# Why is Kafka needed? 2

- ❖ Kafka can work in combination with
  - ❖ Flume/Flafka, Spark Streaming, Storm, HBase and Spark for real-time analysis and processing of streaming data
  - ❖ Feed your data lakes with data streams
- ❖ Kafka brokers support massive message streams for follow-up analysis in Hadoop or Spark

# Kafka Use Cases

- ❖ Stream Processing
- ❖ Microservices + Cassandra
- ❖ Website Activity Tracking
- ❖ Metrics Collection and Monitoring
- ❖ Log Aggregation
- ❖ Real time analytics
- ❖ Capture and ingest data into Spark / Hadoop
- ❖ CRQS, replay, error recovery
- ❖ Guaranteed distributed commit log for in-memory computing

# Who uses Kafka?

- ❖ ***LinkedIn***
- ❖ ***Twitter***
- ❖ ***Square***
- ❖ Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc.

# Why is Kafka Popular?

- ❖ ***Great performance***
- ❖ Operational Simplicity, easy to setup and use, easy to reason
- ❖ Stable, Reliable Durability,
- ❖ Flexible Publish-subscribe/queue (scales with N-number of consumer groups),
- ❖ Robust Replication,
- ❖ Producer Tunable Consistency Guarantees,
- ❖ Ordering Preserved at shard level (Topic Partition)
- ❖ Works well with systems that have data streams to process, aggregate, transform & load into other stores

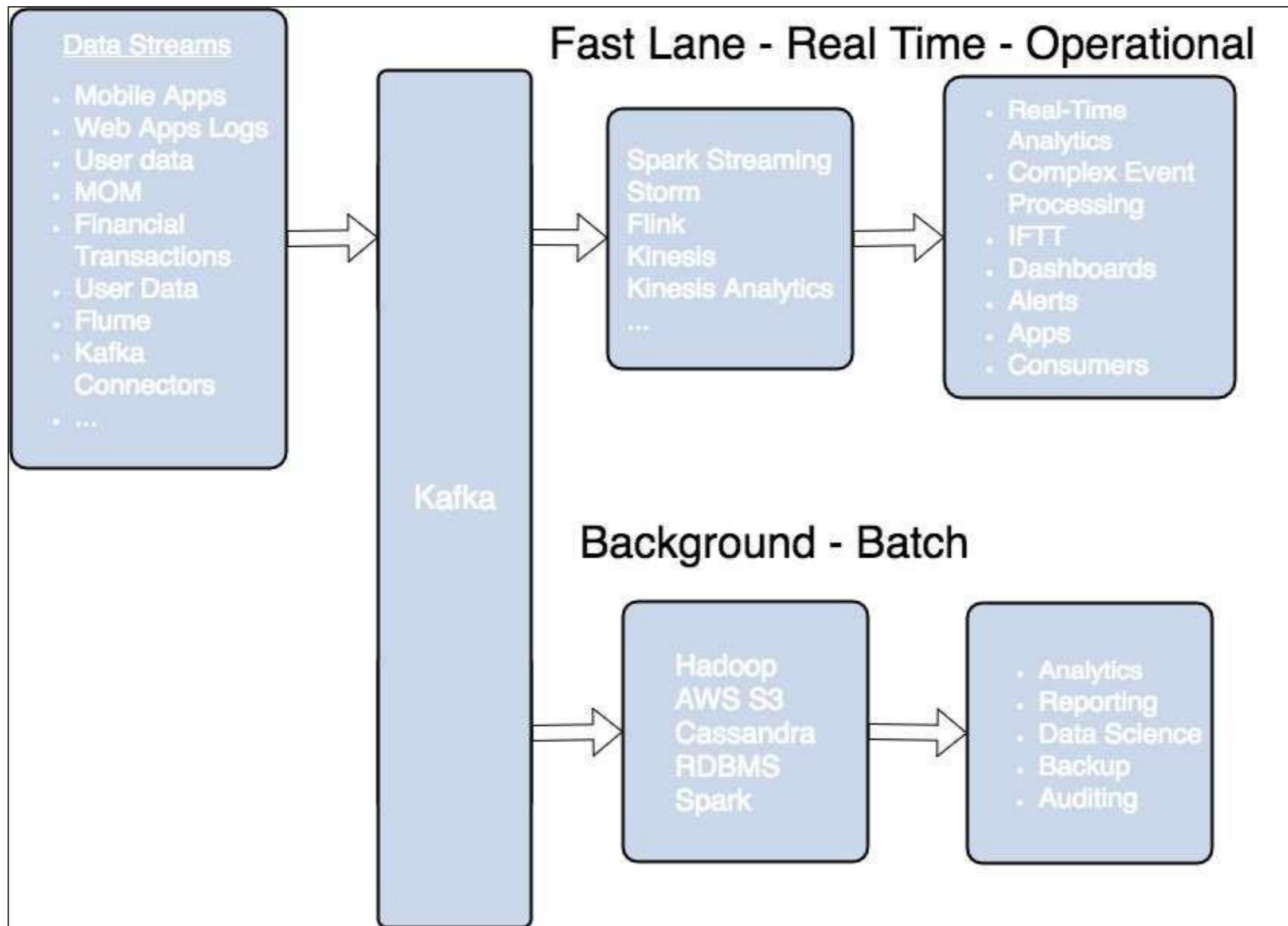
Most important reason: ***Kafka's great performance***: throughput, latency, obtained through great engineering

# Why is Kafka so fast?

- ❖ **Zero Copy**
- ❖ **Batch Data in Chunks**
- ❖ **Sequential Disk Writes**
- ❖ **Horizontal Scale**



# Kafka Streaming Architecture



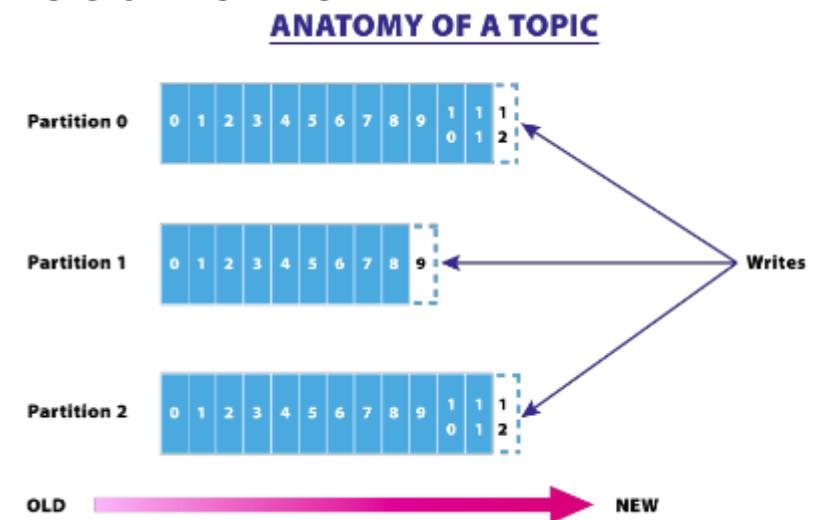


# Why Kafka Review

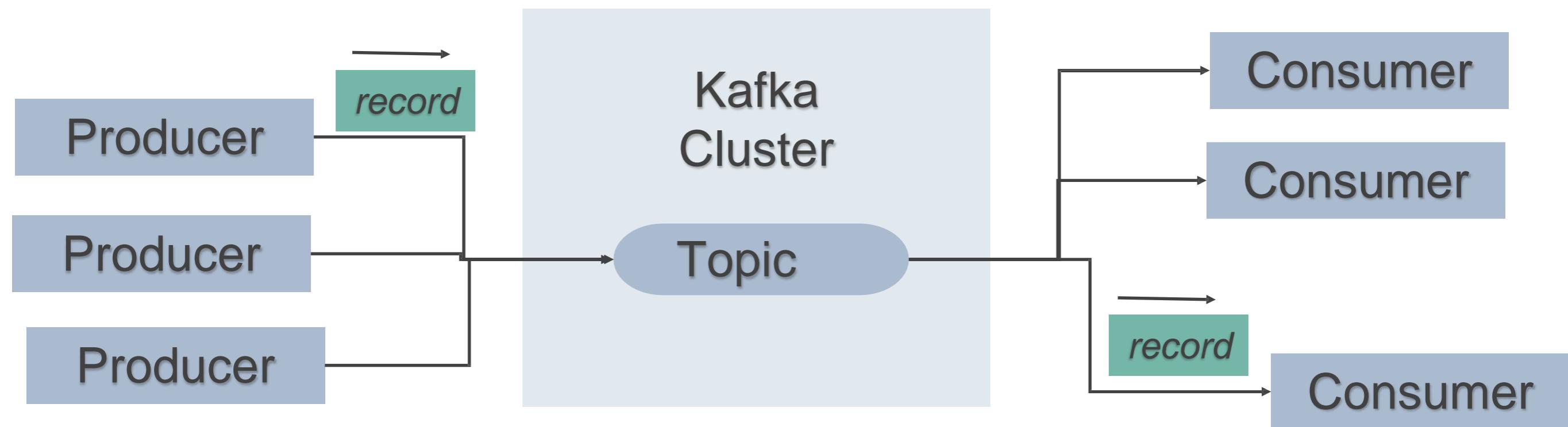
- ❖ Why is Kafka so fast?
- ❖ How fast is Kafka usage growing?
- ❖ How is Kafka getting used?
- ❖ Where does Kafka fit in the Big Data Architecture?
- ❖ How does Kafka relate to real-time analytics?
- ❖ Who uses Kafka?

# Kafka Fundamentals

- ❖ **Records** have a **key (optional)**, **value** and **timestamp**; **Immutable**
- ❖ **Topic** a stream of records (“/orders”, “/user-signups”), feed name
- ❖ **Log** topic storage on disk
- ❖ **Partition** / Segments (parts of Topic Log)
- ❖ **Producer API** to produce a streams or records
- ❖ **Consumer API** to consume a stream of records
- ❖ **Broker**: Kafka server that runs in a Kafka Cluster. Brokers form a cluster. Cluster consists on many Kafka Brokers on many servers.
- ❖ **ZooKeeper**: Does coordination of brokers/cluster topology. Consistent file system for configuration information and leadership election for Broker Topic Partition Leaders



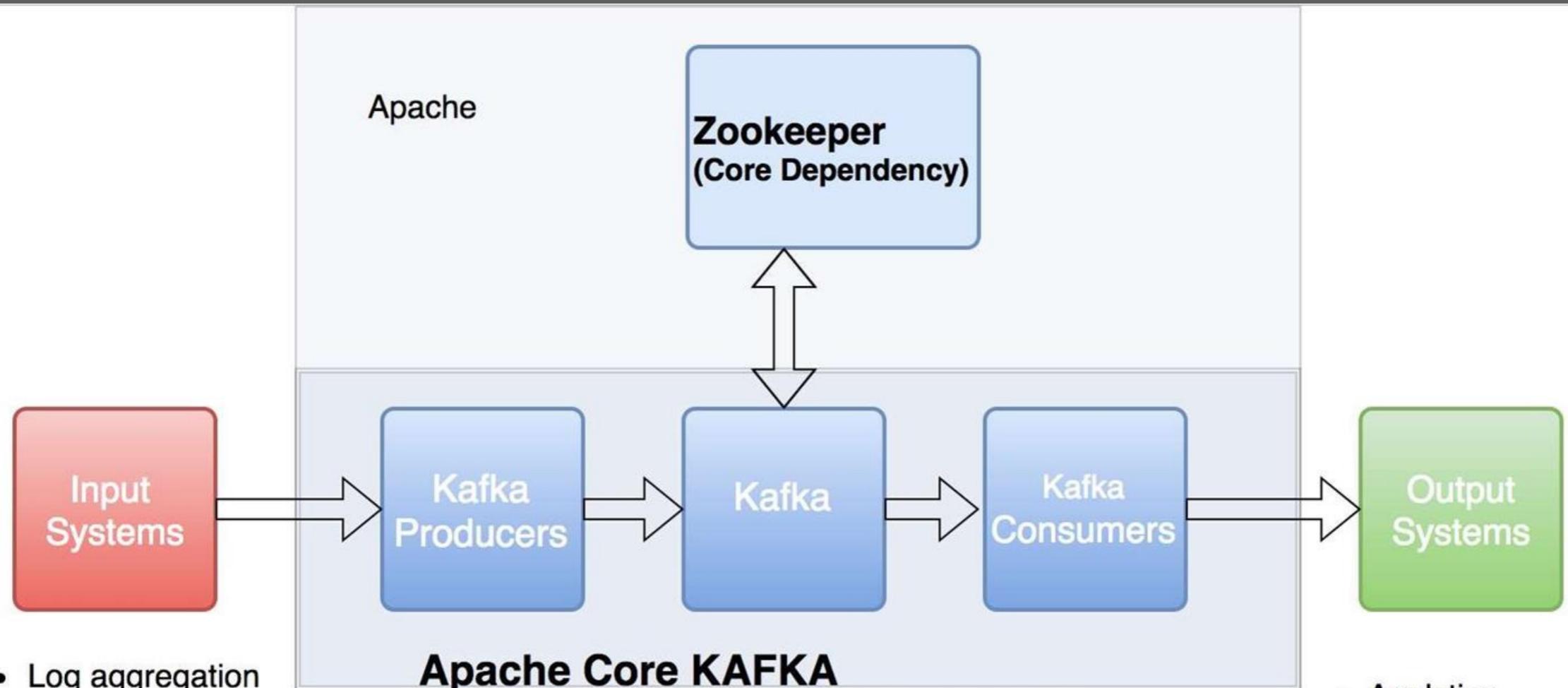
# Kafka: Topics, Producers, and Consumers



# Apache Kafka - Core Kafka

- ❖ Kafka gets conflated with Kafka ecosystem
- ❖ Apache Core Kafka consists of Kafka Broker, startup scripts for ZooKeeper, and client APIs for Kafka

# Apache Kafka



- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs

*Not* part of core

- Schema Registry
- Avro
- Kafka REST Proxy
- Kafka Connect
- Kafka Streams

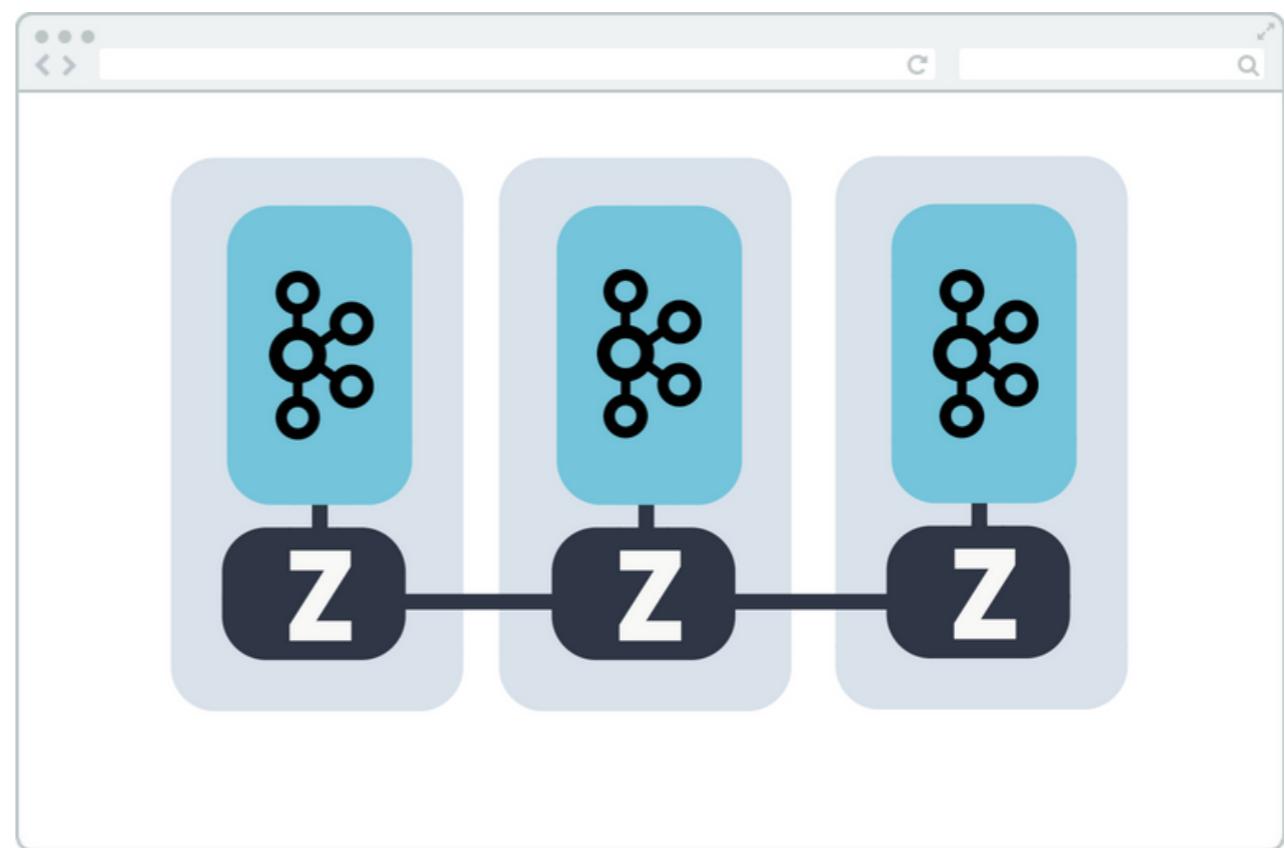
Apache Kafka Core

- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligence

# Kafka needs Zookeeper

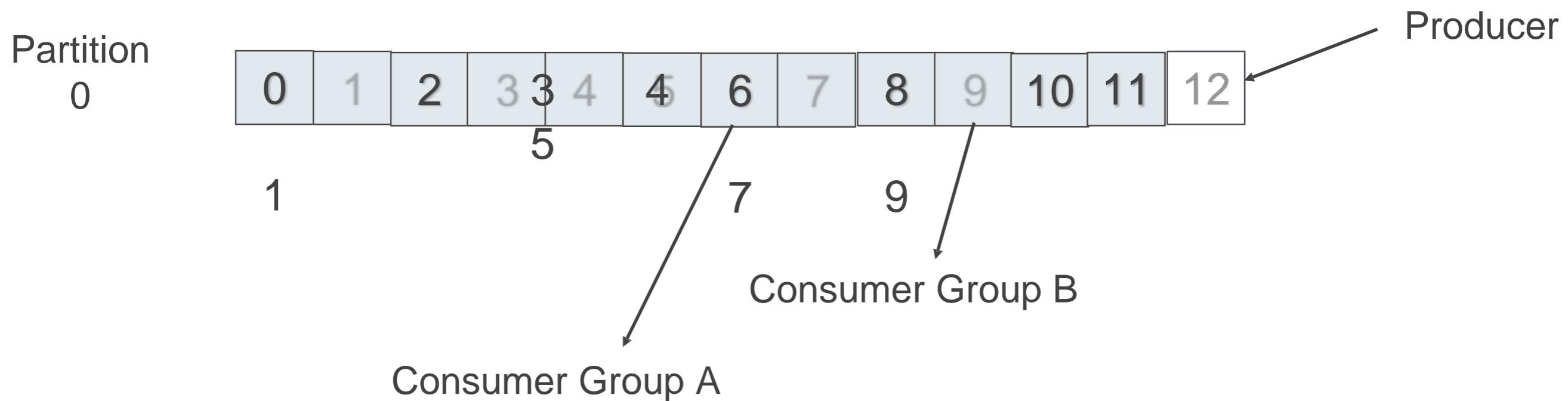
- ❖ Zookeeper helps with leadership election of Kafka Broker and Topic Partition pairs
- ❖ Zookeeper manages service discovery for Kafka Brokers that form the cluster
- ❖ Zookeeper sends changes to Kafka
  - ❖ New Broker join, Broker died, etc.
  - ❖ Topic removed, Topic added, etc.
- ❖ Zookeeper provides in-sync view of Kafka Cluster configuration



# Kafka Producer/Consumer Details

- ❖ **Producers** write to and **Consumers** read from **Topic(s)**
- ❖ **Topic** associated with a log which is data structure on disk
- ❖ **Producer(s)** append **Records** at end of Topic log
- ❖ Topic **Log** consist of Partitions -
  - ❖ Spread to multiple files on multiple nodes
- ❖ **Consumers** read from Kafka at their own cadence
  - ❖ Each Consumer (Consumer Group) tracks offset from where they left off reading
- ❖ **Partitions** can be distributed on different machines in a cluster
  - ❖ high performance with horizontal scalability and failover with replication

# Kafka Topic Partition, Consumers, Producers



Consumer groups remember offset where they left off.  
Consumers groups each have their own offset.

Producer writing to offset 12 of Partition 0 while...  
Consumer Group A is reading from offset 6.  
Consumer Group B is reading from offset 9.

# Kafka Scale and Speed



- ❖ How can Kafka scale if multiple producers and consumers read/write to same Kafka Topic log?
- ❖ Writes fast: Sequential writes to filesystem are **fast** (700 MB or more a second)
- ❖ Scales writes and reads by **sharding**:
  - ❖ Topic logs into **Partitions** (parts of a Topic log)
  - ❖ Topics logs can be split into multiple Partitions **different machines/different disks**
  - ❖ Multiple Producers can write to different Partitions of the same Topic
  - ❖ Multiple Consumers Groups can read from different partitions efficiently

---

# Kafka Brokers



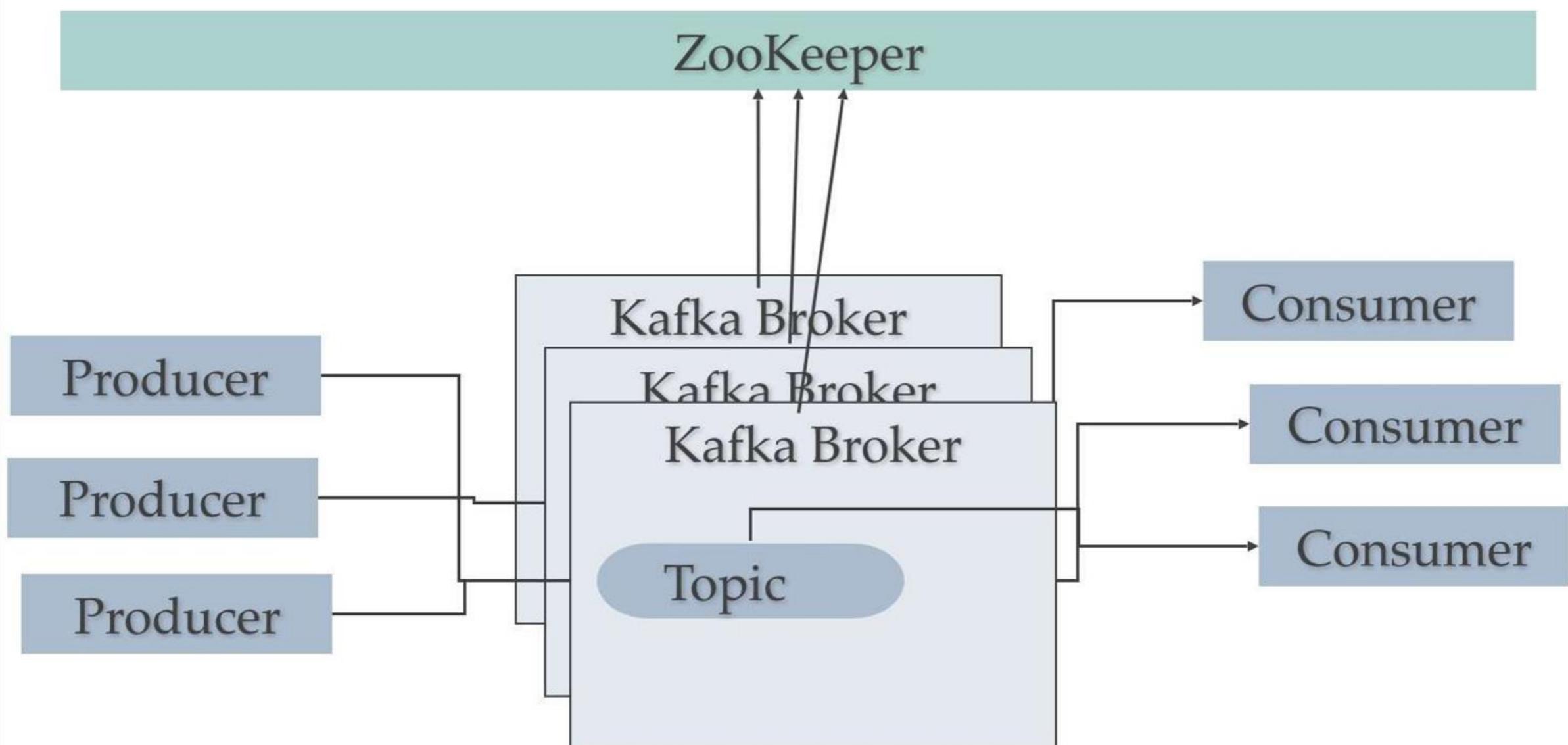
- ❖ Kafka Cluster is made up of multiple Kafka Brokers
- ❖ Each Broker has an ID (number)
- ❖ Brokers contain topic log partitions
- ❖ Connecting to one broker bootstraps client to entire cluster
- ❖ Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

# Kafka Cluster, Failover, ISRs



- ❖ Topic ***Partitions*** can be ***replicated***
  - ❖ across ***multiple nodes*** for failover
- ❖ Topic should have a replication factor greater than 1
  - ❖ (2, or 3)
- ❖ ***Failover***
  - ❖ if one Kafka Broker goes down then Kafka Broker with ISR (in-sync replica) can serve data

# ZooKeeper does coordination for Kafka Cluster



# Failover vs. Disaster Recovery



- ❖ Replication of Kafka Topic Log partitions allows for failure of a rack or AWS availability zone
  - ❖ You need a replication factor of at least 3
- ❖ ***Kafka Replication*** is for ***Failover***
- ❖ ***Mirror Maker*** is used for ***Disaster Recovery***
- ❖ Mirror Maker replicates a Kafka cluster to another data-center or AWS region
  - ❖ Called mirroring since replication happens within a cluster



# Kafka Review



- ❖ How does Kafka decouple streams of data?
- ❖ What are some use cases for Kafka where you work?
- ❖ What are some common use cases for Kafka?
- ❖ What is a Topic?
- ❖ What is a Broker?
- ❖ What is a Partition? Offset?
- ❖ Can Kafka run without Zookeeper?
- ❖ How do implement failover in Kafka?
- ❖ How do you implement disaster recovery in Kafka?

# Comparing Kafka with other queue systems (JMS / MQ)

# Kafka vs JMS, SQS, RabbitMQ

## Messaging



- ❖ Is Kafka a Queue or a Pub/Sub/Topic?
- ❖ Kafka is like a Queue per consumer group
- ❖ Kafka is like Topics in JMS, RabbitMQ, MOM
- ❖ MOM = JMS, ActiveMQ, RabbitMQ, IBM MQ Series, Tibco, etc.

# Kafka vs MOM



- ❖ By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log
- ❖ Also by moving location (partition offset) in log to client/consumer side of equation instead of the broker, less tracking required by Broker and more flexible consumers
- ❖ Kafka written with mechanical sympathy, modern hardware, cloud in mind

# Kinesis and Kafka are similar



- ❖ Kinesis Streams is like Kafka Core
- ❖ Kinesis Analytics is like Kafka Streams
- ❖ Kinesis Shard is like Kafka Partition
- ❖ Similar and get used in similar use cases
- ❖ In Kinesis, data is stored in shards. In Kafka, data is stored in partitions
- ❖ Kinesis Analytics allows you to perform SQL like queries on data streams
- ❖ Kafka Streaming allows you to perform functional aggregations and mutations
- ❖ Kafka integrates well with Spark and Flink which allows SQL like queries on streams

# Kafka vs. Kinesis



- ❖ Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
- ❖ Kafka records default stored for 7 days
- ❖ With Kinesis data can be analyzed by Lambda before it gets sent to S3 or RedShift
- ❖ With Kinesis you pay for use, by buying read and write units.
- ❖ Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going
- ❖ Kinesis is sold as a service and does not require a DevOps team to keep it going (can be more expensive and less flexible, but much easier to setup and run)

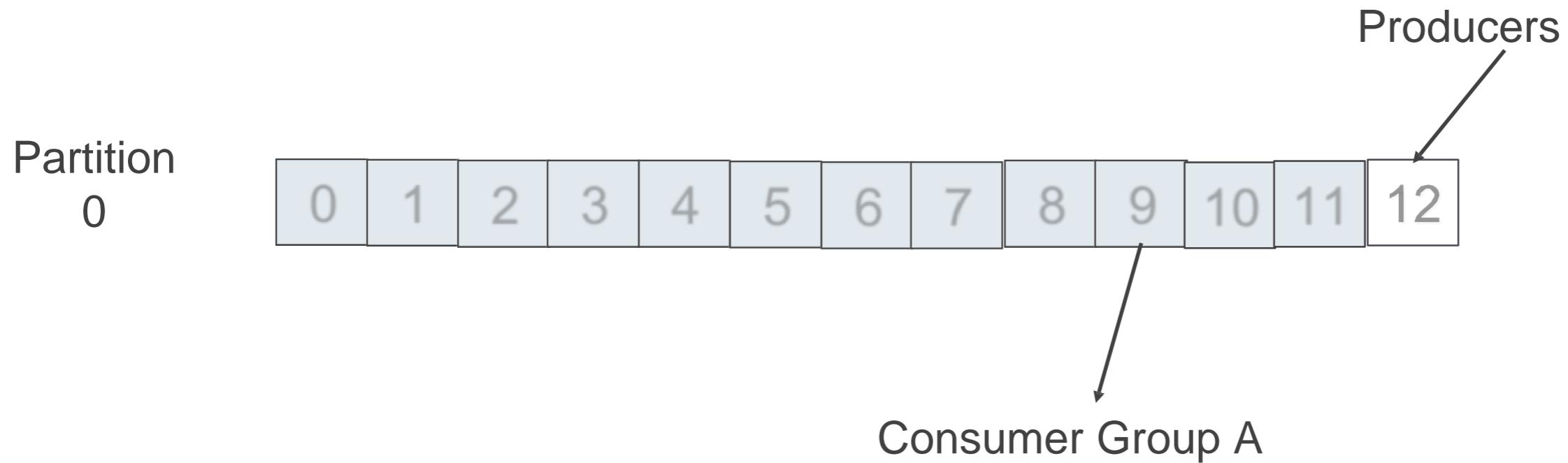
# Producing messages

# Kafka Producers



- ❖ ***Producers*** send records to topics
- ❖ ***Producer*** picks which partition to send record to per topic
- ❖ Important: *Producer picks partition*

# Kafka Producers and Consumers



Producers are writing at Offset 12

Consumer Group A is Reading from Offset 9.

# Kafka Producers



- ❖ **Producers** write at their own cadence so order of Records cannot be guaranteed across partitions
- ❖ **Producer** configures consistency level (ack=0, ack=all, ack=1)
- ❖ **Producers** pick the **partition** such that Record/messages goes to a given same partition based on the data



# Producer Review



- ❖ Can Producers occasionally write faster than consumers?
- ❖ What is the default partition strategy for Producers without using a key?
- ❖ What is the default partition strategy for Producers using a key?
- ❖ What picks which partition a record is sent to?

# Consuming messages

# Kafka Consumer Groups



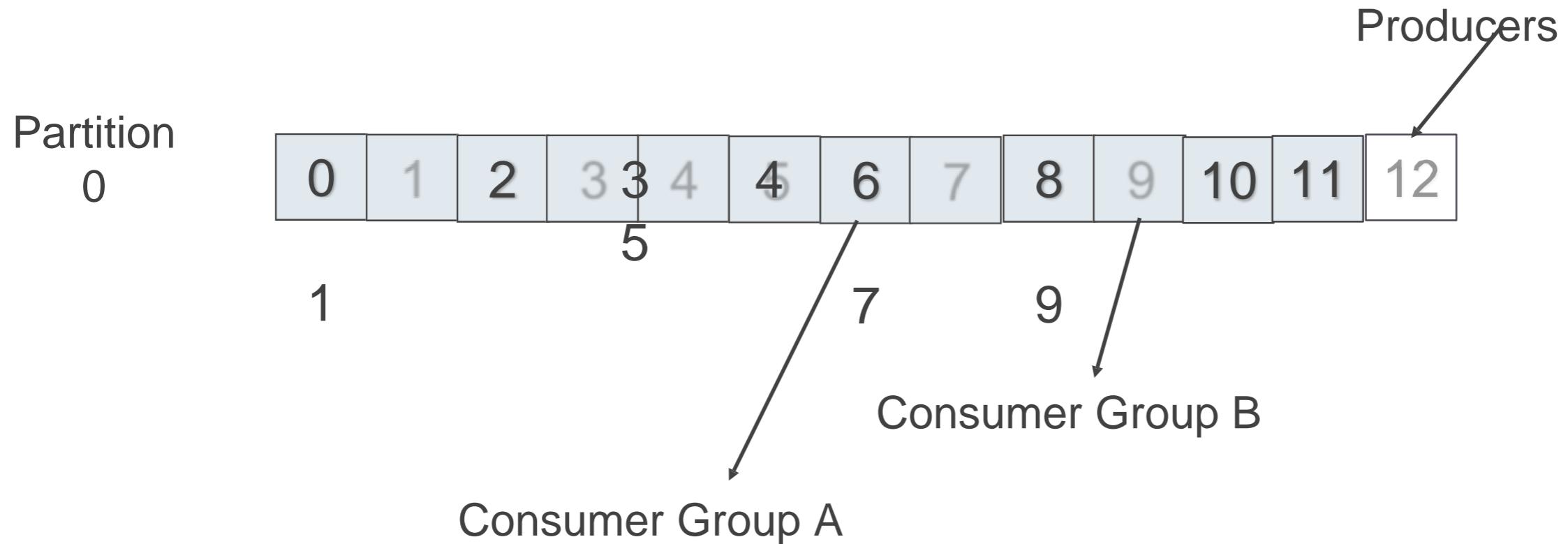
- ❖ Consumers are grouped into a ***Consumer Group***
  - ❖ ***Consumer group*** has a unique id
  - ❖ Each ***consumer group*** is a subscriber
  - ❖ Each ***consumer group*** maintains its own offset
  - ❖ Multiple subscribers = multiple consumer groups
  - ❖ Each has different function: one might delivering records to microservices while another is streaming records to Hadoop
- ❖ A ***Record*** is delivered to one ***Consumer*** in a ***Consumer Group***
- ❖ Each consumer in consumer groups takes records and only one consumer in group gets same record
- ❖ Consumers in Consumer Group ***load balance*** record consumption

# Kafka Consumer Load Share



- ❖ Kafka **Consumer** consumption **divides** partitions over consumers in a Consumer Group
- ❖ Each **Consumer** is exclusive consumer of a "**fair share**" of **partitions**
- ❖ This is Load Balancing
- ❖ **Consumer** membership in **Consumer Group** is handled by the Kafka protocol dynamically
- ❖ If new Consumers **join** Consumer group, it gets a share of partitions
- ❖ If Consumer **dies**, its partitions are split among remaining live Consumers in Consumer Group

# Kafka Consumer Groups



Consumers remember offset where they left off.

Consumers groups each have their own offset per partition.

# Kafka Consumer Groups Processing



- ❖ How does Kafka divide up topic so multiple **Consumers** in a **Consumer Group** can process a topic?
- ❖ You group consumers into consumers group with a group id
- ❖ **Consumers** with same id belong in same **Consumer Group**
- ❖ One **Kafka broker** becomes **group coordinator** for Consumer Group
- ❖ When **Consumer group** is created, offset set according to reset policy of topic

# Kafka Consumer Failover



- ❖ **Consumers** notify broker when it successfully processed a record
  - ❖ advances offset
- ❖ If **Consumer** fails before sending commit offset to Kafka broker,
  - ❖ different **Consumer** can continue from the last committed offset
  - ❖ some Kafka records could be reprocessed
  - ❖ **at least once behavior**
  - ❖ **messages should be idempotent**

# Kafka Consumer Offsets and Recovery

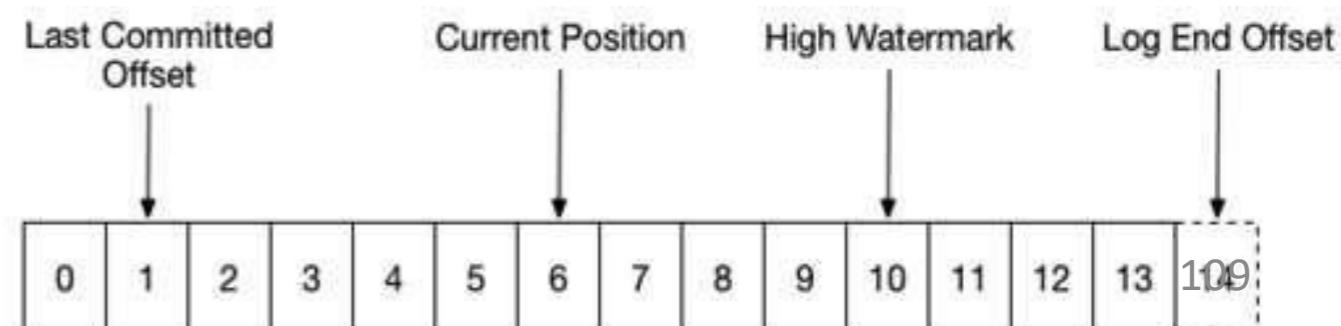


- ❖ Kafka stores offsets in topic called “`__consumer_offset`”
  - ❖ Uses Topic Log Compaction
- ❖ When a consumer has processed data, it should commit offsets
- ❖ If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in “`__consumer_offset`”

# Kafka Consumer: What can be consumed?



- ❖ "**Log end offset**" is offset of last record written to log partition and where **Producers** write to next
- ❖ "**High watermark**" is offset of last record successfully replicated to all partitions followers
- ❖ **Consumer** only reads up to "high watermark".  
**Consumer can't read un-replicated data**



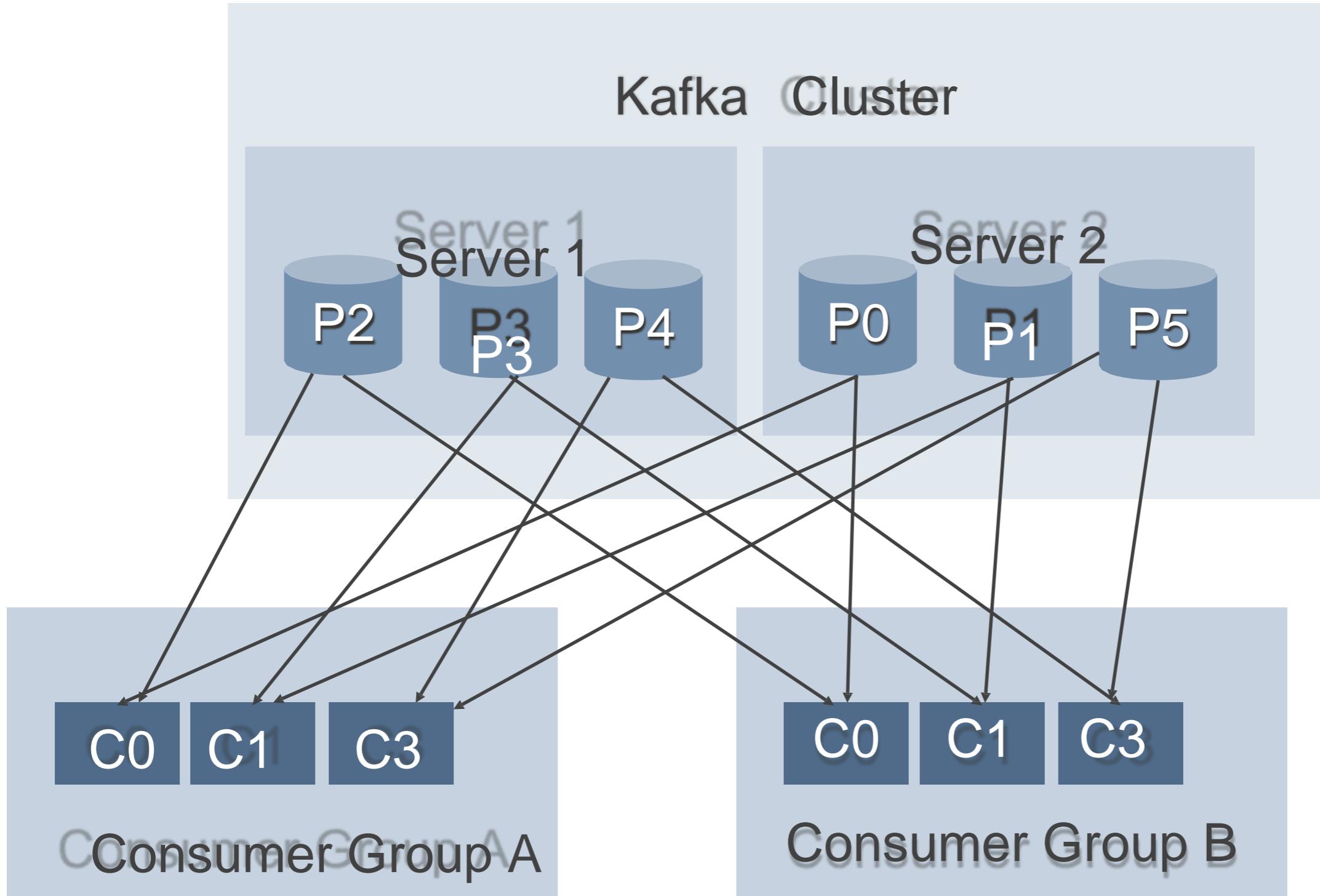
# Consumer to Partition Cardinality



- ❖ Only a single ***Consumer*** from the same ***Consumer Group*** can access a single ***Partition***
- ❖ If ***Consumer Group*** count **exceeds** Partition count:
  - ❖ Extra Consumers remain idle; can be used for failover
- ❖ If more Partitions than Consumer Group instances,
  - ❖ Some Consumers will read from more than one partition



## 2 server Kafka cluster hosting 4 partitions (P0-P5)



# Multi-threaded Consumers



- ❖ You can run more than one Consumer in a JVM process
- ❖ If processing records takes a while, a single Consumer can run multiple threads to process records
  - ❖ Harder to manage offset for each Thread/Task
  - ❖ One Consumer runs multiple threads
  - ❖ 2 messages on same partitions being processed by two different threads
  - ❖ Hard to guarantee order without threads coordination
- ❖ **PREFER:** Multiple Consumers can run each processing record batches in their own thread
  - ❖ Easier to manage offset
  - ❖ Each Consumer runs in its thread
  - ❖ Easier to manage failover (each process runs X num of Consumer threads)



# Consumer Review



- ❖ What is a consumer group?
- ❖ Does each consumer have its own offset?
- ❖ When can a consumer see a record?
- ❖ What happens if there are more consumers than partitions?
- ❖ What happens if you run multiple consumers in many thread in the same JVM?

# Using Kafka Single Node



# Run Kafka

- ❖ Run ZooKeeper start up script
- ❖ Run Kafka Server/Broker start up script
- ❖ Create Kafka Topic from command line
- ❖ Run producer from command line
- ❖ Run consumer from command line

# Run ZooKeeper



run-zookeeper.sh x

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/zookeeper-server-start.sh \
5   kafka/config/zookeeper.properties
6
```

```
$ ./run-zookeeper.sh
[2017-05-13 13:34:52,489] INFO Reading configuration from: kafka/config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,491] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)
[2017-05-13 13:34:52,491] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)
[2017-05-13 13:34:52,491] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)
[2017-05-13 13:34:52,491] WARN Either no config or no quorum defined in config, running in stand-alone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2017-05-13 13:34:52,504] INFO Reading configuration from: kafka/config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,504] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2017-05-13 13:34:57,609] INFO Server environment:zookeeper.version=3.4.9-1757313, built on 08/23/2016 06:50 GMT (org.apache.zookeeper.server.ZooKeeperServer)
[2017-05-13 13:34:57,609] INFO Server environment:host.name=10.0.0.115 (org.apache.zookeeper.serve
```



# Run Kafka Server

run-kafka.sh x

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-server-start.sh \
5     kafka/config/server.properties
```

```
$ ./run-kafka.sh
[2017-05-13 13:47:01,497] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
    auto.create.topics.enable = true
    auto.leader.rebalance.enable = true
    background.threads = 10
    broker.id = 0
    broker.id.generation.enable = true
    broker.rack = null
    compression.type = producer
    connections.max.idle.ms = 600000
    controlled.shutdown.enable = true
    controlled.shutdown.max.retries = 3
    controlled.shutdown.retry.backoff.ms = 5000
    controller.socket.timeout.ms = 30000
```



# Create Kafka Topic

```
create-topic.sh >
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # Create a topic
6 kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
7 --replication-factor 1 --partitions 13 --topic my-topic
```

```
$ ./create-topic.sh
Created topic "my-topic".
```



# List Topics

```
> list-topics.sh ×
```

```
1 #!/usr/bin/env bash  
2  
3 cd ~/kafka-training  
4  
5 # List existing topics  
6 kafka/bin/kafka-topics.sh --list \  
7 --zookeeper localhost:2181  
8
```

```
~/kafka-training/lab1/solution  
$ ./list-topics.sh  
__consumer_offsets  
_schemas  
my-example-topic  
my-example-topic2  
my-topic  
new-employees
```



# Run Kafka Producer

```
>_ start-producer-console.sh x
1  #!/usr/bin/env bash
2  cd ~/kafka-training
3
4  kafka/bin/kafka-console-producer.sh --broker-list \
5  localhost:9092 --topic my-topic
```

# Run Kafka Consumer



start-consumer-console.sh x

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
5 --topic my-topic --from-beginning
6
```

# Running Kafka Producer and Consumer



```
new-employees  
~/kafka-training/lab1/solution  
[$ ./start-producer-console.sh  
This is message 1  
This is message 2  
This is message 3  
Message 4  
Message 5  
Message 6  
Message 7
```

```
Last login: Sat May 13 13:57:09 on ttys004  
~/kafka-training/lab1/solution  
[$ ./start-consumer-console.sh  
Message 4  
This is message 2  
This is message 1  
This is message 3  
Message 5  
Message 6  
Message 7
```

# ? Kafka Single Node Review



- ❖ What server do you run first?
- ❖ What tool do you use to create a topic?
- ❖ What tool do you use to see topics?
- ❖ What tool did we use to send messages on the command line?
- ❖ What tool did we use to view messages in a topic?
- ❖ Why were the messages coming out of order?
- ❖ How could we get the messages to come in order from the consumer?



*Use Kafka to send and receive messages*

# Lab Use Kafka

Use single server version of Kafka.  
Setup single node.  
Single ZooKeeper.  
Create a topic.  
Produce and consume messages from the command line.

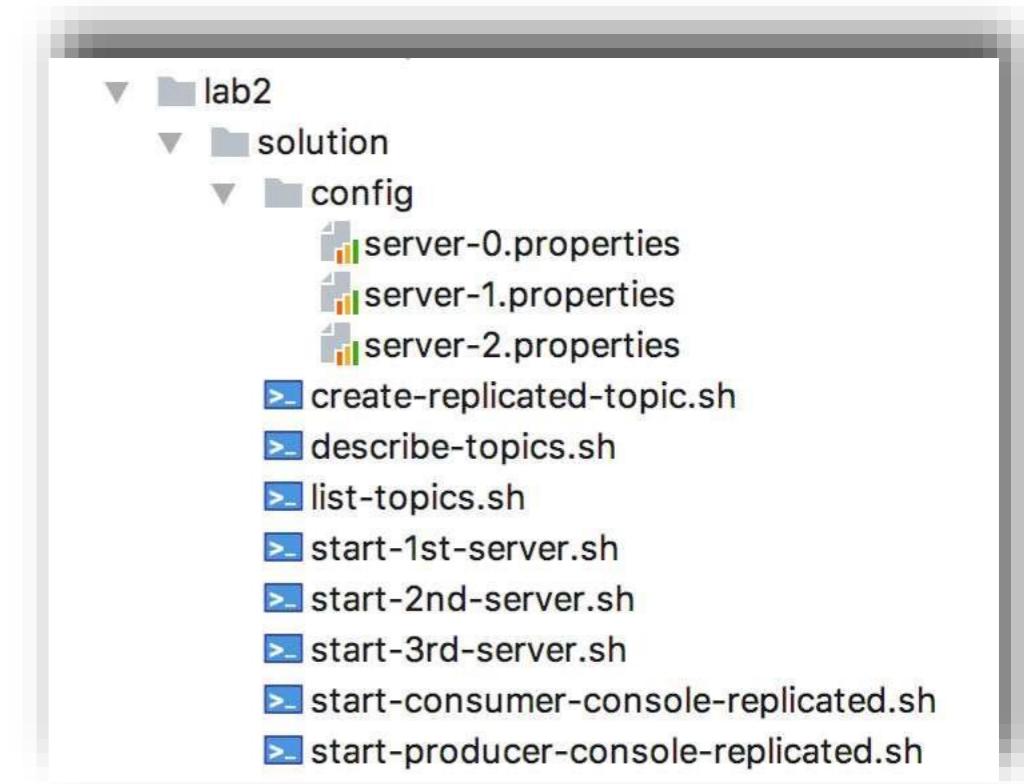
# Complete Lab 1

# Kafka Cluster and Failover



# Objectives

- ❖ Run many Kafka Brokers
- ❖ Create a replicated topic
- ❖ Demonstrate Pub / Sub
- ❖ Demonstrate load balancing consumers
- ❖ Demonstrate consumer failover
- ❖ Demonstrate broker failover



# Running many nodes



- ❖ If not already running, start up ZooKeeper
  - ❖ Shutdown Kafka from first lab
- ❖ Copy server properties for three brokers
  - ❖ Modify properties files, Change port, Change Kafka log location
- ❖ Start up many Kafka server instances
- ❖ Create Replicated Topic
- ❖ Use the replicated topic

```
~/kafka-training  
$ ./run-zookeeper.sh
```

# Create three new server- n.properties files



- ❖ Copy existing ***server.properties*** to ***server-0.properties*, *server-1.properties*, *server-2.properties***
- ❖ Change ***server-0.properties*** to use ***log.dirs* “./logs/kafka-logs-0”**
- ❖ Change ***server-1.properties*** to use ***port 9093*, *broker id 1*, and *log.dirs* “./logs/kafka-logs-1”**
- ❖ Change ***server-2.properties*** to use ***port 9094*, *broker id 2*, and *log.dirs* “./logs/kafka-logs-2”**

# Modify server-x.properties



```
server-0.properties x
1 broker.id=0
2 port=9092
3 log.dirs=./logs/kafka-0
.
.
.
server-1.properties x
1 broker.id=1
2 port=9093
3 log.dirs=./logs/kafka-1
4
.
.
.
server-2.properties x
1 broker.id=2
2 port=9094
3 log.dirs=./logs/kafka-2
.
```

- ❖ Each have different *broker.id*
- ❖ Each have different *log.dirs*
- ❖ Each had different *port*

# Create Startup scripts for three Kafka servers



```
start-1st-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3
4 cd ~/kafka-training
5
6 ## Run Kafka
7 kafka/bin/kafka-server-start.sh \
8     "$CONFIG/server-0.properties"
9

start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-1.properties"
8

start-3rd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-2.properties"
8
```



```
start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-1.properties"
8
9
```

- ❖ Passing properties files from last step



# Run Servers

```
$ ./start-1st-server.sh
[2017-05-15 11:18:00,168] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
```

```
$ ./start-2nd-server.sh
[2017-05-15 11:18:24,980] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
```

```
~/kafka-training/lab2/solution
$ ./start-3rd-server.sh
[2017-05-15 11:19:04,129] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
```



# Create Kafka replicated topic my-failsafe-topic

```
create-replicated-topic.sh >
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 kafka/bin/kafka-topics.sh --create \
6   --zookeeper localhost:2181 \
7   --replication-factor 3 \
8   --partitions 13 \
9   --topic my-failsafe-topic
10
```

- ❖ **Replication Factor** is set to 3
- ❖ Topic name is ***my-failsafe-topic***
- ❖ **Partitions** is 13

```
$ ./create-replicated-topic.sh
Created topic "my-failsafe-topic".
```

# Start Kafka Consumer



```
start-consumer-console-replicated.sh x
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-console-consumer.sh \
5   --bootstrap-server localhost:9094,localhost:9092 \
6   --topic my-failsafe-topic \
7   --from-beginning
8
```

- ❖ Pass list of Kafka servers to bootstrap-server
- ❖ We pass two of the three
- ❖ Only one needed, it learns about the rest

# Start Kafka Producer



```
start-producer-console-replicated.sh >

1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-console-producer.sh \
5 --broker-list localhost:9092,localhost:9093 \
6 --topic my-failsafe-topic
7
```

- ❖ Start producer
- ❖ Pass list of Kafka Brokers

# Kafka 1 consumer and 1 producer running



```
Last login: Mon May 15 11:25:19 on ttys007  
~/kafka-training/lab2/solution  
$ ./start-producer-console-replicated.sh  
Hi mom  
How are you?  
How are things going?  
Good!
```

```
Last login: Mon May 15 11:19:27 on ttys006  
~/kafka-training/lab2/solution  
$ ls  
config                                         start-2  
create-replicated-topic.sh                      start-3  
list-topics.sh                                 start-4  
start-1st-server.sh                            start-p  
~/kafka-training/lab2/solution  
$ ./start-consumer-console-replicated.sh  
Hi mom  
How are you?  
How are things going?  
Good!
```

# Start a second and third consumer



```
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:28:21 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are thin
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:35 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

- ❖ Acts like pub/sub
- ❖ Each consumer in its own group
- ❖ Message goes to each
- ❖ How do we load share?

# Running consumers in same group



start-consumer-console-replicated.sh ×

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-console-consumer.sh \
5   --bootstrap-server localhost:9094,localhost:9092 \
6   --topic my-failsafe-topic \
7   --consumer-property group.id=mygroup
8   --from-beginning
9
```

- ❖ Modify start consumer script
- ❖ Add the consumers to a group called mygroup
- ❖ Now they will share load

# Start up three consumers again

A screenshot of a Mac OS X desktop showing four terminal windows side-by-side. Each window has a title bar indicating the path as ~/kafka-training/lab2/solution and the command ./start-consumer-console-replicated.sh. The output shows different sets of messages (m1-m7) being consumed by different processes. The top-left window shows m1, m2, m3, m4, m5, m6, m7. The top-right window shows m3, m5. The bottom-left window shows m2, m6. The bottom-right window shows m1, m4, m7.

```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7

~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
m6
m7

~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m6

~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m1
m4
m7
```

- ❖ Start up producer and three consumers
- ❖ Send 7 messages
- ❖ Notice how messages are spread among 3 consumers



# Consumer Failover

```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
m17
m18
m19
m20
m21
m22
m23
m24
m25
m26
m27
m28
m29
m30
m31
m32
m33
m34
m35
m36
m37
m38
m39
m40
m41
m42
m43
m44
m45
m46
m47
m48
m49
m50
m51
m52
m53
m54
m55
m56
m57
m58
m59
m60
m61
m62
m63
m64
m65
m66
m67
m68
m69
m70
m71
m72
m73
m74
m75
m76
m77
m78
m79
m80
m81
m82
m83
m84
m85
m86
m87
m88
m89
m90
m91
m92
m93
m94
m95
m96
m97
m98
m99
m100
m101
m102
m103
m104
m105
m106
m107
m108
m109
m110
m111
m112
m113
m114
m115
m116
m117
m118
m119
m120
m121
m122
m123
m124
m125
m126
m127
m128
m129
m130
m131
m132
m133
m134
m135
m136
m137
m138
m139
m140
m141
m142
m143
m144
m145
m146
m147
m148
m149
m150
m151
m152
m153
m154
m155
m156
m157
m158
m159
m160
m161
m162
m163
m164
m165
m166
m167
m168
m169
m170
m171
m172
m173
m174
m175
m176
m177
m178
m179
m180
m181
m182
m183
m184
m185
m186
m187
m188
m189
m190
m191
m192
m193
m194
m195
m196
m197
m198
m199
m200
m201
m202
m203
m204
m205
m206
m207
m208
m209
m210
m211
m212
m213
m214
m215
m216
m217
m218
m219
m220
m221
m222
m223
m224
m225
m226
m227
m228
m229
m230
m231
m232
m233
m234
m235
m236
m237
m238
m239
m240
m241
m242
m243
m244
m245
m246
m247
m248
m249
m250
m251
m252
m253
m254
m255
m256
m257
m258
m259
m260
m261
m262
m263
m264
m265
m266
m267
m268
m269
m270
m271
m272
m273
m274
m275
m276
m277
m278
m279
m280
m281
m282
m283
m284
m285
m286
m287
m288
m289
m290
m291
m292
m293
m294
m295
m296
m297
m298
m299
m299
m300
m301
m302
m303
m304
m305
m306
m307
m308
m309
m310
m311
m312
m313
m314
m315
m316
m317
m318
m319
m320
m321
m322
m323
m324
m325
m326
m327
m328
m329
m329
m330
m331
m332
m333
m334
m335
m336
m337
m338
m339
m339
m340
m341
m342
m343
m344
m345
m346
m347
m348
m349
m349
m350
m351
m352
m353
m354
m355
m356
m357
m358
m359
m359
m360
m361
m362
m363
m364
m365
m366
m367
m368
m369
m369
m370
m371
m372
m373
m374
m375
m376
m377
m378
m379
m379
m380
m381
m382
m383
m384
m385
m386
m387
m388
m389
m389
m390
m391
m392
m393
m394
m395
m396
m397
m398
m399
m399
m400
m401
m402
m403
m404
m405
m406
m407
m408
m409
m409
m410
m411
m412
m413
m414
m415
m416
m417
m418
m419
m419
m420
m421
m422
m423
m424
m425
m426
m427
m428
m429
m429
m430
m431
m432
m433
m434
m435
m436
m437
m438
m439
m439
m440
m441
m442
m443
m444
m445
m446
m447
m448
m449
m449
m450
m451
m452
m453
m454
m455
m456
m457
m458
m459
m459
m460
m461
m462
m463
m464
m465
m466
m467
m468
m469
m469
m470
m471
m472
m473
m474
m475
m476
m477
m478
m479
m479
m480
m481
m482
m483
m484
m485
m486
m487
m488
m489
m489
m490
m491
m492
m493
m494
m495
m496
m497
m498
m499
m499
m500
m501
m502
m503
m504
m505
m506
m507
m508
m509
m509
m510
m511
m512
m513
m514
m515
m516
m517
m518
m519
m519
m520
m521
m522
m523
m524
m525
m526
m527
m528
m529
m529
m530
m531
m532
m533
m534
m535
m536
m537
m538
m539
m539
m540
m541
m542
m543
m544
m545
m546
m547
m548
m549
m549
m550
m551
m552
m553
m554
m555
m556
m557
m558
m559
m559
m560
m561
m562
m563
m564
m565
m566
m567
m568
m569
m569
m570
m571
m572
m573
m574
m575
m576
m577
m578
m579
m579
m580
m581
m582
m583
m584
m585
m586
m587
m588
m589
m589
m590
m591
m592
m593
m594
m595
m596
m597
m598
m599
m599
m600
m601
m602
m603
m604
m605
m606
m607
m608
m609
m609
m610
m611
m612
m613
m614
m615
m616
m617
m618
m619
m619
m620
m621
m622
m623
m624
m625
m626
m627
m628
m629
m629
m630
m631
m632
m633
m634
m635
m636
m637
m638
m639
m639
m640
m641
m642
m643
m644
m645
m646
m647
m648
m649
m649
m650
m651
m652
m653
m654
m655
m656
m657
m658
m659
m659
m660
m661
m662
m663
m664
m665
m666
m667
m668
m669
m669
m670
m671
m672
m673
m674
m675
m676
m677
m678
m679
m679
m680
m681
m682
m683
m684
m685
m686
m687
m688
m689
m689
m690
m691
m692
m693
m694
m695
m696
m697
m698
m699
m699
m700
m701
m702
m703
m704
m705
m706
m707
m708
m709
m709
m710
m711
m712
m713
m714
m715
m716
m717
m718
m719
m719
m720
m721
m722
m723
m724
m725
m726
m727
m728
m729
m729
m730
m731
m732
m733
m734
m735
m736
m737
m738
m739
m739
m740
m741
m742
m743
m744
m745
m746
m747
m748
m749
m749
m750
m751
m752
m753
m754
m755
m756
m757
m758
m759
m759
m760
m761
m762
m763
m764
m765
m766
m767
m768
m769
m769
m770
m771
m772
m773
m774
m775
m776
m777
m778
m779
m779
m780
m781
m782
m783
m784
m785
m786
m787
m788
m789
m789
m790
m791
m792
m793
m794
m795
m796
m797
m798
m799
m799
m800
m801
m802
m803
m804
m805
m806
m807
m808
m809
m809
m810
m811
m812
m813
m814
m815
m816
m817
m818
m819
m819
m820
m821
m822
m823
m824
m825
m826
m827
m828
m829
m829
m830
m831
m832
m833
m834
m835
m836
m837
m838
m839
m839
m840
m841
m842
m843
m844
m845
m846
m847
m848
m849
m849
m850
m851
m852
m853
m854
m855
m856
m857
m858
m859
m859
m860
m861
m862
m863
m864
m865
m866
m867
m868
m869
m869
m870
m871
m872
m873
m874
m875
m876
m877
m878
m879
m879
m880
m881
m882
m883
m884
m885
m886
m887
m888
m889
m889
m890
m891
m892
m893
m894
m895
m896
m897
m898
m899
m899
m900
m901
m902
m903
m904
m905
m906
m907
m908
m909
m909
m910
m911
m912
m913
m914
m915
m916
m917
m918
m919
m919
m920
m921
m922
m923
m924
m925
m926
m927
m928
m929
m929
m930
m931
m932
m933
m934
m935
m936
m937
m938
m939
m939
m940
m941
m942
m943
m944
m945
m946
m947
m948
m949
m949
m950
m951
m952
m953
m954
m955
m956
m957
m958
m959
m959
m960
m961
m962
m963
m964
m965
m966
m967
m968
m969
m969
m970
m971
m972
m973
m974
m975
m976
m977
m978
m979
m979
m980
m981
m982
m983
m984
m985
m986
m987
m988
m989
m989
m990
m991
m992
m993
m994
m995
m996
m997
m998
m999
m999
m1000
m1001
m1002
m1003
m1004
m1005
m1006
m1007
m1008
m1009
m1009
m1010
m1011
m1012
m1013
m1014
m1015
m1016
m1017
m1018
m1019
m1019
m1020
m1021
m1022
m1023
m1024
m1025
m1026
m1027
m1028
m1029
m1029
m1030
m1031
m1032
m1033
m1034
m1035
m1036
m1037
m1038
m1039
m1039
m1040
m1041
m1042
m1043
m1044
m1045
m1046
m1047
m1048
m1049
m1049
m1050
m1051
m1052
m1053
m1054
m1055
m1056
m1057
m1058
m1059
m1059
m1060
m1061
m1062
m1063
m1064
m1065
m1066
m1067
m1068
m1069
m1069
m1070
m1071
m1072
m1073
m1074
m1075
m1076
m1077
m1078
m1079
m1079
m1080
m1081
m1082
m1083
m1084
m1085
m1086
m1087
m1088
m1089
m1089
m1090
m1091
m1092
m1093
m1094
m1095
m1096
m1097
m1098
m1098
m1099
m1099
m1100
m1101
m1102
m1103
m1104
m1105
m1106
m1107
m1108
m1109
m1109
m1110
m1111
m1112
m1113
m1114
m1115
m1116
m1117
m1118
m1119
m1119
m1120
m1121
m1122
m1123
m1124
m1125
m1126
m1127
m1128
m1129
m1129
m1130
m1131
m1132
m1133
m1134
m1135
m1136
m1137
m1138
m1139
m1139
m1140
m1141
m1142
m1143
m1144
m1145
m1146
m1147
m1148
m1149
m1149
m1150
m1151
m1152
m1153
m1154
m1155
m1156
m1157
m1158
m1159
m1159
m1160
m1161
m1162
m1163
m1164
m1165
m1166
m1167
m1168
m1169
m1169
m1170
m1171
m1172
m1173
m1174
m1175
m1176
m1177
m1178
m1178
m1179
m1180
m1181
m1182
m1183
m1184
m1185
m1186
m1187
m1188
m1188
m1189
m1190
m1191
m1192
m1193
m1194
m1195
m1196
m1197
m1198
m1198
m1199
m1199
m1200
m1201
m1202
m1203
m1204
m1205
m1206
m1207
m1208
m1209
m1209
m1210
m1211
m1212
m1213
m1214
m1215
m1216
m1217
m1218
m1219
m1219
m1220
m1221
m1222
m1223
m1224
m1225
m1226
m1227
m1228
m1229
m1229
m1230
m1231
m1232
m1233
m1234
m1235
m1236
m1237
m1238
m1239
m1239
m1240
m1241
m1242
m1243
m1244
m1245
m1246
m1247
m1248
m1249
m1249
m1250
m1251
m1252
m1253
m1254
m1255
m1256
m1257
m1258
m1259
m1259
m1260
m1261
m1262
m1263
m1264
m1265
m1266
m1267
m1268
m1269
m1269
m1270
m1271
m1272
m1273
m1274
m1275
m1276
m1277
m1278
m1278
m1279
m1280
m1281
m1282
m1283
m1284
m1285
m1286
m1287
m1288
m1288
m1289
m1290
m1291
m1292
m1293
m1294
m1295
m1296
m1297
m1297
m1298
m1299
m1299
m1300
m1301
m1302
m1303
m1304
m1305
m1306
m1307
m1308
m1309
m1309
m1310
m1311
m1312
m1313
m1314
m1315
m1316
m1317
m1318
m1319
m1319
m1320
m1321
m1322
m1323
m1324
m1325
m1326
m1327
m1328
m1329
m1329
m1330
m1331
m1332
m1333
m1334
m1335
m1336
m1337
m1338
m1339
m1339
m1340
m1341
m1342
m1343
m1344
m1345
m1346
m1347
m1348
m1349
m1349
m1350
m1351
m1352
m1353
m1354
m1355
m1356
m1357
m1358
m1359
m1359
m1360
m1361
m1362
m1363
m1364
m1365
m1366
m1367
m1368
m1369
m1369
m1370
m1371
m1372
m1373
m1374
m1375
m1376
m1377
m1378
m1378
m1379
m1380
m1381
m1382
m1383
m1384
m1385
m1386
m1387
m1388
m1388
m1389
m1390
m1391
m1392
m1393
m1394
m1395
m1396
m1397
m1398
m1398
m1399
m1399
m1400
m1401
m1402
m1403
m1404
m1405
m1406
m1407
m1408
m1409
m1409
m1410
m1411
m1412
m1413
m1414
m1415
m1416
m1417
m1418
m1419
m1419
m1420
m1421
m1422
m1423
m1424
m1425
m1426
m1427
m1428
m1429
m1429
m1430
m1431
m1432
m1433
m1434
m1435
m1436
m1437
m1438
m1439
m1439
m1440
m1441
m1442
m1443
m1444
m1445
m1446
m1447
m1448
m1449
m1449
m1450
m1451
m1452
m1453
m1454
m1455
m1456
m1457
m1458
m1459
m1459
m1460
m1461
m1462
m1463
m1464
m1465
m1466
m1467
m1468
m1469
m1469
m1470
m1471
m1472
m1473
m1474
m1475
m1476
m1477
m1478
m1478
m1479
m1480
m1481
m1482
m1483
```

# Create Kafka Describe Topic



```
>_ describe-topics.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic my-failsafe-topic \
8   --zookeeper localhost:2181
9
```

- ❖ —describe will show list partitions, ISRs, and partition leadership

# Use Describe Topics



```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13      ReplicationFactor:3      Configs:
Topic: my-failsafe-topic      Partition: 0      Leader: 2      Replicas: 2,0,1 Isr: 2,0,1
Topic: my-failsafe-topic      Partition: 1      Leader: 0      Replicas: 0,1,2 Isr: 0,1,2
Topic: my-failsafe-topic      Partition: 2      Leader: 1      Replicas: 1,2,0 Isr: 1,2,0
Topic: my-failsafe-topic      Partition: 3      Leader: 2      Replicas: 2,1,0 Isr: 2,1,0
Topic: my-failsafe-topic      Partition: 4      Leader: 0      Replicas: 0,2,1 Isr: 0,2,1
Topic: my-failsafe-topic      Partition: 5      Leader: 1      Replicas: 1,0,2 Isr: 1,0,2
Topic: my-failsafe-topic      Partition: 6      Leader: 2      Replicas: 2,0,1 Isr: 2,0,1
Topic: my-failsafe-topic      Partition: 7      Leader: 0      Replicas: 0,1,2 Isr: 0,1,2
Topic: my-failsafe-topic      Partition: 8      Leader: 1      Replicas: 1,2,0 Isr: 1,2,0
Topic: my-failsafe-topic      Partition: 9      Leader: 2      Replicas: 2,1,0 Isr: 2,1,0
Topic: my-failsafe-topic      Partition: 10     Leader: 0      Replicas: 0,2,1 Isr: 0,2,1
Topic: my-failsafe-topic      Partition: 11     Leader: 1      Replicas: 1,0,2 Isr: 1,0,2
Topic: my-failsafe-topic      Partition: 12     Leader: 2      Replicas: 2,0,1 Isr: 2,0,1
```

- ❖ Lists which broker owns (leader of) which partition
- ❖ Lists Replicas and ISR (replicas that are up to date)
- ❖ Notice there are 13 topics

# Test Broker Failover: Kill 1st server



Kill the first server

```
~/kafka-training/lab2/solution
```

```
[$ kill `ps aux | grep java | grep server-0.properties | tr -s " " | cut -d " " -f2`
```

use Kafka topic describe to see that a new leader was elected!

```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13      ReplicationFactor:3      Configs:
Topic: my-failsafe-topic      Partition: 0      Leader: 2      Replicas: 2,0,1 Isr: 2,1
Topic: my-failsafe-topic      Partition: 1      Leader: 1      Replicas: 0,1,2 Isr: 1,2
Topic: my-failsafe-topic      Partition: 2      Leader: 1      Replicas: 1,2,0 Isr: 1,2
Topic: my-failsafe-topic      Partition: 3      Leader: 2      Replicas: 2,1,0 Isr: 2,1
Topic: my-failsafe-topic      Partition: 4      Leader: 2      Replicas: 0,2,1 Isr: 2,1
Topic: my-failsafe-topic      Partition: 5      Leader: 1      Replicas: 1,0,2 Isr: 1,2
Topic: my-failsafe-topic      Partition: 6      Leader: 2      Replicas: 2,0,1 Isr: 2,1
Topic: my-failsafe-topic      Partition: 7      Leader: 1      Replicas: 0,1,2 Isr: 1,2
Topic: my-failsafe-topic      Partition: 8      Leader: 1      Replicas: 1,2,0 Isr: 1,2
Topic: my-failsafe-topic      Partition: 9      Leader: 2      Replicas: 2,1,0 Isr: 2,1
Topic: my-failsafe-topic      Partition: 10     Leader: 2      Replicas: 0,2,1 Isr: 2,1
Topic: my-failsafe-topic      Partition: 11     Leader: 1      Replicas: 1,0,2 Isr: 1,2
Topic: my-failsafe-topic      Partition: 12     Leader: 2      Replicas: 2,0,1 Isr: 2,1
```

# Show Broker Failover Worked



```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
m8
m9
m11
m14
[2017-05-15 12:00:58,462] WARN Auto-commit
ta='', my-failsafe-topic-3=OffsetAndMetad
ffset=1, metadata=''}, my-failsafe-topic-1
etAndMetadata{offset=1, metadata=''}, my-f
fe-topic-5=OffsetAndMetadata{offset=2, met
triable exception. You should retry commit
Coordinator)
m16
```

- ❖ Send two more messages from the producer
- ❖ Notice that the consumer gets the messages
- ❖ Broker Failover WORKS!



# Kafka Cluster Review



- ❖ Why did the three consumers not load share the messages at first?
- ❖ How did we demonstrate failover for consumers?
- ❖ How did we demonstrate failover for producers?
- ❖ What tool and option did we use to show ownership of partitions and the ISRs?

# Complete Lab 1.2

# Kafka Ecosystem

# Kafka Universe



- ❖ ***Ecosystem is Apache Kafka Core plus these (and community Kafka Connectors)***
- ❖ ***Kafka Streams***
  - ❖ ***Streams*** API to transform, aggregate, process records from a stream and produce derivative streams
- ❖ ***Kafka Connect***
  - ❖ ***Connector*** API reusable producers and consumers
    - ❖ (e.g., stream of changes from DynamoDB)
- ❖ ***Kafka REST Proxy***
  - ❖ Producers and Consumers over REST (HTTP)
- ❖ ***Schema Registry*** - Manages schemas using Avro for Kafka Records
- ❖ ***Kafka MirrorMaker*** - Replicate cluster data to another cluster

# What comes in Apache Kafka Core?



Apache Kafka Core Includes:

- ❖ ZooKeeper and startup scripts
- ❖ Kafka Server (Kafka Broker), Kafka Clustering
- ❖ Utilities to monitor, create topics, inspect topics, replicated (mirror) data to another datacenter
- ❖ Producer APIs, Consumer APIs
- ❖ Part of Apache Foundation
- ❖ Packages / Distributions are free do download with no registry

# What comes in Kafka Extensions?



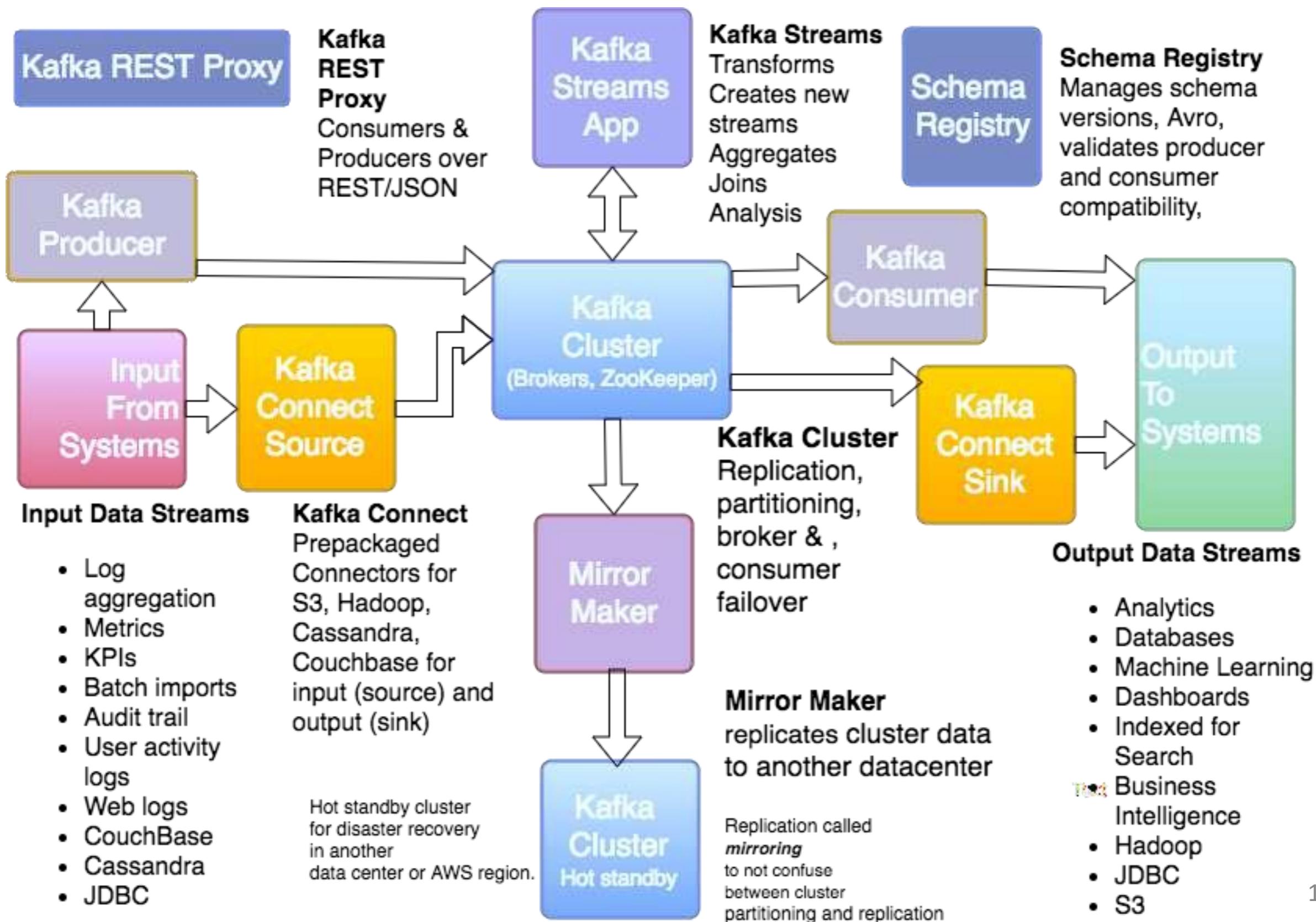
Confluent.io:

- ❖ All of Kafka Core
- ❖ ***Schema Registry*** (schema versioning and compatibility checks) ( [Confluent project](#))
- ❖ ***Kafka REST Proxy ([Confluent project](#))***
- ❖ ***Kafka Streams*** (aggregation, joining streams, mutating streams, creating new streams by combining other streams) ( [Confluent project](#))
- ❖ ***Not*** Part of Apache Foundation controlled by Confluent.io
- ❖ Code hosted on GitHub
- ❖ Packages / Distributions are free do download ***but you must register with [Confluent](#)***

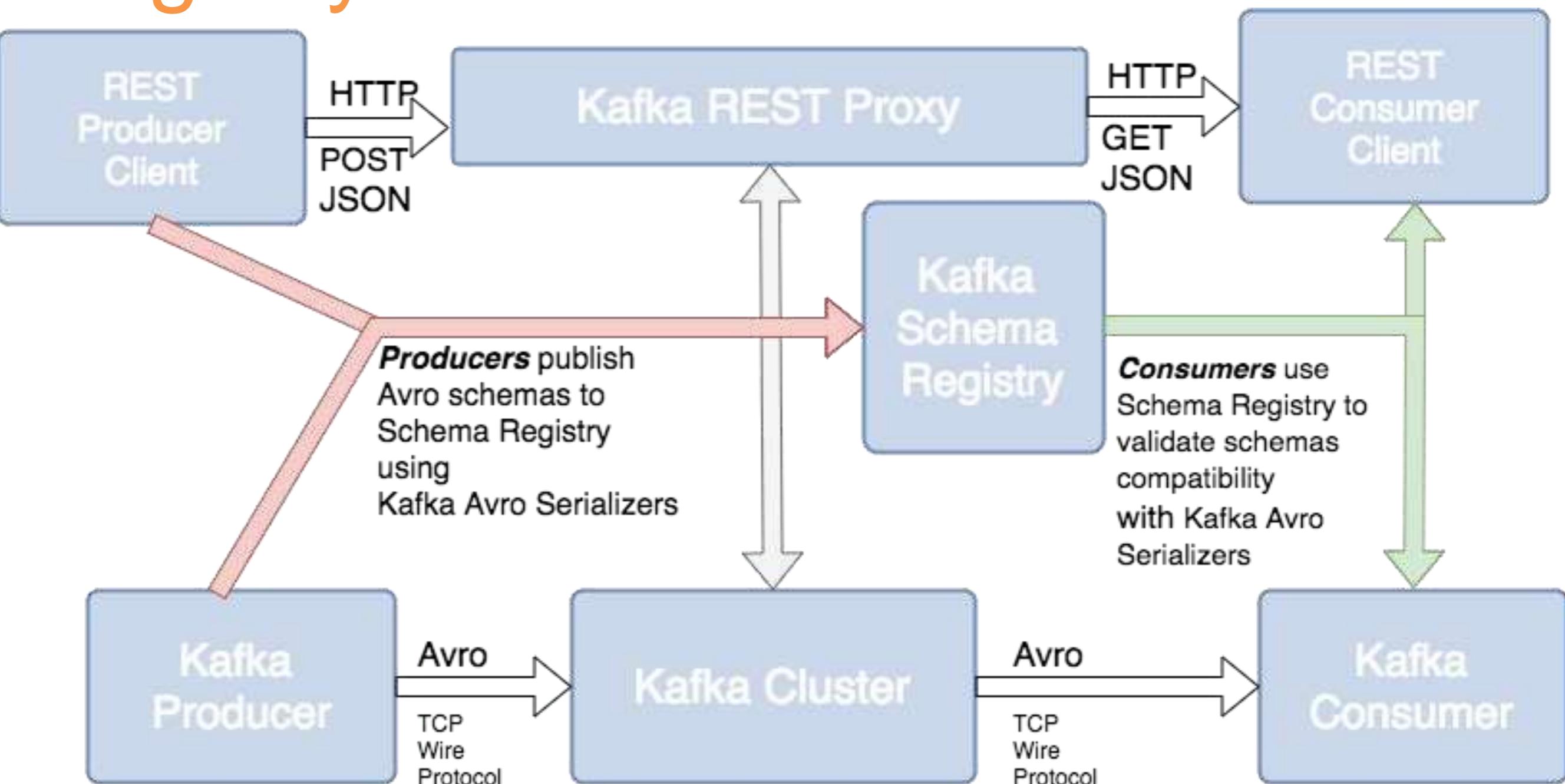
Community of Kafka Connectors from 3rd parties and [Confluent](#)



# Kafka Universe



# Kafka REST Proxy and Schema Registry

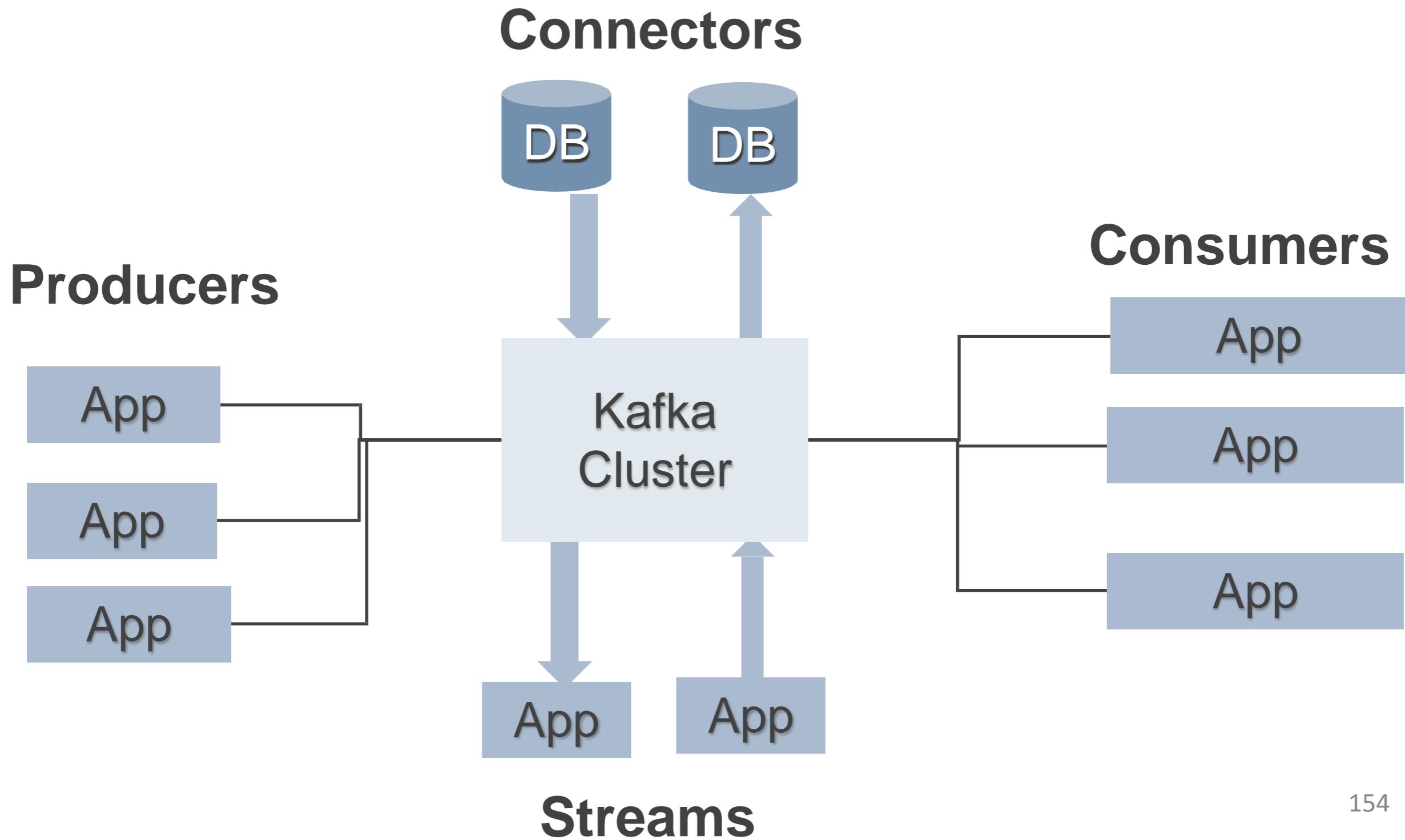


# Kafka Stream : Stream Processing



- ❖ **Kafka Streams** for Stream Processing
  - ❖ Kafka enable **real-time** processing of streams.
- ❖ Kafka Streams supports **Stream Processor**
  - ❖ processing, transformation, aggregation, and produces 1 to \* output streams
- ❖ Example: video player app sends events videos watched, videos paused
  - ❖ output a new stream of user preferences
  - ❖ can gear new video recommendations based on recent user activity
  - ❖ can aggregate activity of many users to see what new videos are hot
- ❖ Solves hard problems: out of order records, aggregating/joining across streams, stateful computations, and more

# Kafka Connectors and Streams



# ? Kafka Ecosystem review



- ❖ What is Kafka Streams?
- ❖ What is Kafka Connect?
- ❖ What is the Schema Registry?
- ❖ What is Kafka Mirror Maker?
- ❖ When might you use Kafka REST Proxy?

# Session 3

## Using Kafka APIs

# Intro to Producers

# Objectives Create Producer



- ❖ Create simple example that creates a ***Kafka Producer***
- ❖ Create a new replicated ***Kafka topic***
- ❖ ***Create Producer*** that uses topic to send records
- ❖ ***Send records with Kafka Producer***
- ❖ ***Send records asynchronously.***
- ❖ ***Send records synchronously***

# Create Replicated Kafka Topic



create-topic.sh ×

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 ## Create topics
5 kafka/bin/kafka-topics.sh --create \
6   --replication-factor 3 \
7   --partitions 13 \
8   --topic my-example-topic \
9   --zookeeper localhost:2181
10
11 ## List created topics
12 kafka/bin/kafka-topics.sh --list \
13   --zookeeper localhost:2181
```

```
$ ./create-topic.sh
Created topic "my-example-topic".
EXAMPLE_TOPIC
__consumer_offsets
kafkatopic
my-example-topic
my-failsafe-topic
my-topic
```

# Gradle Build script



kafka-training x

```
1 group 'cloudurable-kafka'  
2 version '1.0-SNAPSHOT'  
3  
4 apply plugin: 'java'  
5  
6 sourceCompatibility = 1.8  
7  
8 repositories {  
9     mavenCentral()  
10 }  
11  
12 dependencies {  
13     compile 'org.apache.kafka:kafka-clients:0.10.2.0'  
14     compile 'ch.qos.logback:logback-classic:1.2.2'  
15 }
```

# Create Kafka Producer to send records



- ❖ Specify bootstrap servers
- ❖ Specify client.id
- ❖ Specify Record Key serializer
- ❖ Specify Record Value serializer

# Common Kafka imports and constants



```
KafkaProducerExample.java x

KafkaProducerExample

1 package com.cloudurable.kafka;
2
3 import org.apache.kafka.clients.producer.*;
4 import org.apache.kafka.common.serialization.LongSerializer;
5 import org.apache.kafka.common.serialization.StringSerializer;
6
7 import java.util.Properties;
8
9 public class KafkaProducerExample {
10
11     private final static String TOPIC = "my-example-topic";
12     private final static String BOOTSTRAP_SERVERS =
13         "localhost:9092,localhost:9093,localhost:9094";
14 }
```

# Create Kafka Producer to send records



```
KafkaProducerExample.java x
KafkaProducerExample

14
15     private static Producer<Long, String> createProducer() {
16         Properties props = new Properties();
17         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
18                   BOOTSTRAP_SERVERS);
19         props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
20         props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
21                   LongSerializer.class.getName());
22         props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
23                   StringSerializer.class.getName());
24         return new KafkaProducer<>(props);
25     }
26 }
```

# Send sync records with Kafka Producer



```
KafkaProducerExample.java x
KafkaProducerExample runProducer()

27
28 static void runProducer(final int sendMessageCount) throws Exception {
29     final Producer<Long, String> producer = createProducer();
30     long time = System.currentTimeMillis();
31
32     try {
33         for (long index = time; index < time + sendMessageCount; index++) {
34             final ProducerRecord<Long, String> record =
35                 new ProducerRecord<>(TOPIC, index,
36                                         value: "Hello Mom " + index);
37
38             RecordMetadata metadata = producer.send(record).get();
39
40             long elapsedTime = System.currentTimeMillis() - time;
41             System.out.printf("sent record(key=%s value=%s) " +
42                               "meta(partition=%d, offset=%d) time=%d\n",
43                               record.key(), record.value(), metadata.partition(),
44                               metadata.offset(), elapsedTime);
45
46         }
47     } finally {
48         producer.flush();
49         producer.close();
50     }
51 }
```



# Running the Producer

```
public static void main(String... args) throws Exception {
    if (args.length == 0) {
        runProducer( sendMessageCount: 5 );
    } else {
        runProducer(Integer.parseInt(args[0]));
    }
}
```

Run KafkaExample

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See <http://www.slf4j.org/codes.html#StaticLoggerBinder> for further details.

```
sent record(key=1492463982402 value=Hello Mom 1492463982402) meta(partition=0, offset=380) time=139
sent record(key=1492463982403 value=Hello Mom 1492463982403) meta(partition=0, offset=381) time=141
sent record(key=1492463982404 value=Hello Mom 1492463982404) meta(partition=0, offset=382) time=141
sent record(key=1492463982405 value=Hello Mom 1492463982405) meta(partition=0, offset=383) time=141
sent record(key=1492463982406 value=Hello Mom 1492463982406) meta(partition=0, offset=384) time=141
Got Record: (1492463982402, Hello Mom 1492463982402) at offset 380
Got Record: (1492463982403, Hello Mom 1492463982403) at offset 381
Got Record: (1492463982404, Hello Mom 1492463982404) at offset 382
Got Record: (1492463982405, Hello Mom 1492463982405) at offset 383
Got Record: (1492463982406, Hello Mom 1492463982406) at offset 384
DONE
```

# Send async records with Kafka Producer

```
static void runProducer(final int sendMessageCount) throws InterruptedException {
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();
    final CountDownLatch countDownLatch = new CountDownLatch(sendMessageCount);

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                new ProducerRecord<>(TOPIC, index, value: "Hello Mom " + index);
            producer.send(record, (metadata, exception) -> {
                long elapsedTime = System.currentTimeMillis() - time;
                if (metadata != null) {
                    System.out.printf("sent record(key=%s value=%s) " +
                        "meta(partition=%d, offset=%d) time=%d\n",
                        record.key(), record.value(), metadata.partition(),
                        metadata.offset(), elapsedTime);
                } else {
                    exception.printStackTrace();
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await( timeout: 25, TimeUnit.SECONDS );
    }finally {
        producer.flush();
        producer.close();
    }
}
```



# Async Interface Callback

org.apache.kafka.clients.producer

## Interface Callback

---

**public interface Callback**

A callback interface that the user can implement to allow code to execute when the request is complete. This callback will generally execute in the background I/O thread so it should be fast.

### Method Summary

#### Methods

Modifier and Type	Method and Description
void	<b>onCompletion(RecordMetadata metadata, Exception exception)</b> A callback method the user can implement to provide asynchronous handling of request completion.



# Async Send Method

## send

```
public Future<RecordMetadata> send(ProducerRecord<K,V> record,  
                                     Callback callback)
```

Asynchronously send a record to a topic and invoke the provided callback when the send has been acknowledged.

The send is asynchronous and this method will return immediately once the record has been stored in the buffer of records waiting to be sent. This allows sending many records in parallel without blocking to wait for the response after each one.

- ❖ Used to send a record to a topic
- ❖ provided callback gets called when the send is acknowledged
- ❖ Send is asynchronous, and method will return immediately
  - ❖ once the record gets stored in the buffer of records waiting to post to the Kafka broker
- ❖ Allows sending many records in parallel without blocking

# Checking that replication is working



```
$ kafka/bin/kafka-replica-verification.sh --broker-list localhost:9092 --topic-white-list my-example-topic  
2017-05-17 14:06:46,446: verification process is started.  
2017-05-17 14:07:16,416: max lag is 0 for partition [my-example-topic,12] at offset 197 among 13 partitions  
2017-05-17 14:07:46,417: max lag is 0 for partition [my-example-topic,12] at offset 201 among 13 partitions
```

- ❖ Verify that replication is working with
  - ❖ **kafka-replica-verification**
  - ❖ Utility that ships with Kafka
  - ❖ If lag or outage you will see it as follows:

```
2017-05-17 14:36:47,497: max lag is 11 for partition [my-example-topic,5] at offset 272 among 13 partitions
```

```
2017-05-17 14:37:19,408: max lag is 15 for partition [my-example-topic,5] at offset 272 among 13 partitions
```

...

```
2017-05-17 14:38:49,607: max lag is 0 for partition [my-example-topic,12] at offset 272 among 13 partitions
```

# Java Kafka Simple Producer recap



- ❖ Created simple example that creates a ***Kafka Producer***
- ❖ Created a new replicated ***Kafka topic***
- ❖ ***Created Producer*** that uses topic to send records
- ❖ ***Sent records with Kafka Producer using async and sync send***

# ? Kafka Producer Review



- ❖ What does the Callback lambda do?
- ❖ What will happen if the first server is down in the bootstrap list? Can the producer still connect to the other Kafka brokers in the cluster?
- ❖ When would you use Kafka async send vs. sync send?
- ❖ Why do you need two serializers for a Kafka record?

# Complete Lab 2

# Advanced Producers

# Objectives Create Producer



- ❖ Cover advanced topics regarding Java Kafka Producer
- ❖ Custom Serializers
- ❖ Custom Partitioners
- ❖ Batching
- ❖ Compression
- ❖ Retries and Timeouts

# Kafka Producer



- ❖ Kafka client that publishes records to Kafka cluster
- ❖ Thread safe
- ❖ Producer has pool of buffer that holds to-be-sent records
  - ❖ background I/O threads turning records into request bytes and transmit requests to Kafka
- ❖ Close producer so producer will not leak resources

# Kafka Producer Send, Acks and Buffers



- ❖ **send()** method is asynchronous
  - ❖ adds the record to output buffer and return right away
  - ❖ buffer used to batch records for efficiency IO and compression
- ❖ acks config controls Producer record durability. "all" setting ensures full commit of record, and is most durable and least fast setting
- ❖ Producer can retry failed requests
- ❖ Producer has buffers of unsent records per topic partition (sized at **batch.size**)

# Kafka Producer: Buffering and batching



- ❖ Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight ***max.in.flight.requests.per.connection***)
- ❖ To reduce requests count, set ***linger.ms*** > 0
  - ❖ wait up to ***linger.ms*** before sending or until batch fills up whichever comes first
  - ❖ Under heavy load ***linger.ms*** not met, under light producer load used to increase broker IO throughput and increase compression
- ❖ ***buffer.memory*** controls total memory available to producer for buffering
  - ❖ If records sent faster than they can be transmitted to Kafka then this buffer gets exceeded then additional send calls block. If period blocks (***max.block.ms***) after then Producer throws a `TimeoutException`



# Producer Acks

- ❖ Producer Config property ***acks***
  - ❖ **(default all)**
- ❖ Write Acknowledgment received count required from partition leader before write request deemed complete
- ❖ Controls ***Producer*** sent records durability
- ❖ Can be all (-1), none (0), or leader (1)



# Acks 0 (NONE)

- ❖ acks=0
- ❖ Producer does not wait for any ack from broker at all
- ❖ Records added to the socket buffer are considered sent
- ❖ No guarantees of durability - maybe
- ❖ Record Offset returned is set to -1 (unknown)
- ❖ Record loss if leader is down
- ❖ Use Case: maybe log aggregation



# Acks 1 (LEADER)

- ❖ acks=1
- ❖ Partition leader wrote record to its local log but responds without followers confirmed writes
- ❖ If leader fails right after sending ack, record could be lost
  - ❖ Followers might have not replicated the record
- ❖ Record loss is rare but possible
- ❖ Use Case: log aggregation



# Acks -1 (ALL)

- ❖ acks=all or acks=-1
- ❖ Leader gets write confirmation from full set of ISRs before sending ack to producer
- ❖ Guarantees record not be lost as long as one ISR remains alive
- ❖ Strongest available guarantee
- ❖ Even stronger with broker setting ***min.insync.replicas*** (specifies the minimum number of ISRs that must acknowledge a write)
- ❖ Most Use Cases will use this and set a ***min.insync.replicas > 1***

# KafkaProducer config Acks



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer
17 > public class StockPriceKafkaProducer {
18     private static final Logger logger = LoggerFactory.getLogger(StockPriceKafkaProducer.class);
19
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         setupBatchingAndCompression(props);
25         setupRetriesInFlightTimeout(props);
26
27
28         //Set number of acknowledgments - acks - default is all
29         props.put(ProducerConfig.ACKS_CONFIG, "all");
30
31         return new KafkaProducer<>(props);
32     }
}
```

# Producer Buffer Memory Size



- ❖ Producer config property: ***buffer.memory***
  - ❖ ***default 32MB***
  - ❖ Total memory (bytes) producer can use to buffer records to be sent to broker
  - ❖ Producer blocks up to ***max.block.ms*** if ***buffer.memory*** is exceeded
    - ❖ if it is sending faster than the broker can receive, exception is thrown



# Batching by Size

- ❖ Producer config property: ***batch.size***
  - ❖ Default 16K
- ❖ Producer batch records
  - ❖ fewer requests for multiple records sent to same partition
  - ❖ Improved IO throughput and performance on both producer and server
- ❖ If record is larger than the batch size, it will not be batched
- ❖ Producer sends requests containing multiple batches
  - ❖ batch per partition
- ❖ Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

# Batching by Time and Size - 1



- ❖ **Producer** config property: *linger.ms*
  - ❖ Default 0
  - ❖ Producer groups together any records that arrive before they can be sent into a batch
    - ❖ good if records arrive faster than they can be sent out
  - ❖ **Producer** can reduce requests count even under moderate load using *linger.ms*

# Batching by Time and Size - 2



- ❖ ***linger.ms*** adds delay to wait for more records to build up so larger batches are sent
  - ❖ ***good brokers throughput at cost of producer latency***
- ❖ If ***producer*** gets records whose size is ***batch.size*** or more for a broker's leader partitions, then it is sent right away
- ❖ If ***Producers*** gets less than ***batch.size*** but ***linger.ms*** interval has passed, then records for that partition are sent
- ❖ Increase to improve throughput of Brokers and reduce broker load (common improvement)

# Compressing Batches



- ❖ **Producer** config property: ***compression.type***
  - ❖ Default 0
  - ❖ Producer compresses request data
  - ❖ By default producer does not compress
  - ❖ Can be set to none, gzip, snappy, or lz4
  - ❖ Compression is by batch
    - ❖ improves with larger batch sizes
  - ❖ End to end compression possible if Broker config “**compression.type**” set to producer. Compressed data from producer sent to log and consumer by broker

# Batching and Compression Example



```
14 > public class StockPriceKafkaProducer {  
15  
16     private static Producer<String, StockPrice> createProducer() {  
17         final Properties props = new Properties();  
18         setupBootstrapAndSerializers(props);  
19         setupBatchingAndCompression(props);  
  
57  
58     private static void setupBatchingAndCompression(Properties props) {  
59         //Wait up to 50 ms to batch to Kafka - linger.ms  
60         props.put(ProducerConfig.LINGER_MS_CONFIG, 200);  
61  
62         //Holds up to 64K per partition default is 16K - batch.size  
63         props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);  
64  
65         //Holds up to 64 MB default is 32MB for all partition buffers  
66         // - "buffer.memory"  
67         props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33_554_432 * 2);  
68  
69         //Set compression type to snappy - compression.type  
70         props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");  
71     }  
}
```



# Custom Serializers

- ❖ You don't have to use built in serializers
- ❖ You can write your own
- ❖ Just need to be able to convert to/fro a byte[]
- ❖ Serializers work for keys and values
- ❖ ***value.serializer*** and ***key.serializer***

# Custom Serializers Config



```
14 > public class StockPriceKafkaProducer {  
15  
16     private static Producer<String, StockPrice> createProducer() {  
17         final Properties props = new Properties();  
18         setupBootstrapAndSerializers(props);  
19     }  
  
30  
31     private static void setupBootstrapAndSerializers(Properties props) {  
32         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,  
33                 StockAppConstants.BOOTSTRAP_SERVERS);  
34         props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");  
35         props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,  
36                 StringSerializer.class.getName());  
37  
38         //Custom Serializer - config "value.serializer"  
39         props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,  
40                 StockPriceSerializer.class.getName());  
41     }  
}
```



# Custom Serializer

c StockPriceSerializer.java x

```
1 package com.cloudurable.kafka.producer;  
2  
3 import com.cloudurable.kafka.producer.model.StockPrice;  
4 import org.apache.kafka.common.serialization.Serializer;  
5  
6 import java.nio.charset.StandardCharsets;  
7 import java.util.Map;  
8  
9 public class StockPriceSerializer implements Serializer<StockPrice> {  
10  
11     @Override  
12     public byte[] serialize(String topic, StockPrice data) {  
13         return data.toJson().getBytes(StandardCharsets.UTF_8);  
14     }  
15  
16     @Override  
17     public void configure(Map<String, ?> configs, boolean isKey) {  
18     }  
19  
20     @Override  
21     public void close() {  
22     }  
23 }  
24 }
```



# StockPrice

```
c StockPrice.java x
StockPrice
1 package com.cloudurable.kafka.producer.model;
2
3 import io.advantageous.boon.json.JsonFactory;
4
5 public class StockPrice {
6
7     private final int dollars;
8     private final int cents;
9     private final String name;
10
11    public String toJson() {
12        return "{" +
13            "\"dollars\": " + dollars +
14            ", \"cents\": " + cents +
15            ", \"name\": \"\" + name + '\"' +
16            '}';
17    }
18}
```

# Broker Follower Write Timeout



- ❖ *Producer* config property: ***request.timeout.ms***
  - ❖ Default 30 seconds (30,000 ms)
  - ❖ Maximum time broker waits for confirmation from followers to meet Producer acknowledgment requirements for ***ack=all***
  - ❖ Measure of broker to broker latency of request
  - ❖ 30 seconds is high, long process time is indicative of problems

# Producer Request Timeout



- ❖ *Producer* config property: ***request.timeout.ms***
  - ❖ Default 30 seconds (30,000 ms)
  - ❖ Maximum time producer waits for request to complete to broker
  - ❖ Measure of producer to broker latency of request
  - ❖ 30 seconds is very high, long request time is an indicator that brokers can't handle load



# Producer Retries

- ❖ Producer config property: ***retries***
  - ❖ ***Default 0***
- ❖ Retry count if ***Producer*** does not get ack from Broker
  - ❖ only if record send fail deemed a transient error ([API](#))
  - ❖ as if your producer code resent record on failed attempt
- ❖ timeouts are retried, ***retry.backoff.ms*** (default to 100 ms) to wait after failure before ***retry***

# Retry, Timeout, Back-off Example



```
13
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
20          setupRetriesInFlightTimeout(props);
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40      private static void setupRetriesInFlightTimeout(Properties props) {
41
42          //Only two in-flight messages per Kafka broker connection
43          // - max.in.flight.requests.per.connection (default 5)
44          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
45                  1);
46
47
48          //Set the number of retries - retries
49          props.put(ProducerConfig.RETRIES_CONFIG, 2);
50
51
52          //Request timeout - request.timeout.ms
53          props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);
54
55          //Only retry after one second.
56          props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
57      }
```

# Producer Partitioning



- ❖ **Producer** config property: ***partitioner.class***
  - ❖ org.apache.kafka.clients.producer.internals.DefaultPartitioner
- ❖ Partitioner class implements Partitioner interface
- ❖ Default Partitioner partitions using hash of key if record has key
- ❖ Default Partitioner partitions uses round-robin if record has no key

# Configuring Partitioner



```
StockPriceKafkaProducer.java x
StockPriceKafkaProducer createProducer()
17 > public class StockPriceKafkaProducer {
18     private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         setupBatchingAndCompression(props);
25         setupRetriesInFlightTimeout(props);
26
27         //Install partitioner -- "partitioner.class"
28         props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
29                   StockPricePartitioner.class.getName());
30
31         props.put("importantStocks", "IBM,UBER");
```

# StockPricePartitioner



```
StockPriceKafkaProducer.java x StockPricePartitioner.java x
StockPricePartitioner StockPricePartitioner()
1 package com.cloudurable.kafka.producer;
2
3 import org.apache.kafka.clients.producer.Partitioner;
4 import org.apache.kafka.common.Cluster;
5 import org.apache.kafka.common.PartitionInfo;
6
7 import java.util.*;
8
9 public class StockPricePartitioner implements Partitioner{
10
11     private final Set<String> importantStocks;
12     public StockPricePartitioner() { importantStocks = new HashSet<>(); }
13
14     @Override
15     public int partition(final String topic,
16                         final Object objectKey,
17                         byte[] keyBytes, final Object value,
18                         final byte[] valueBytes,
19                         final Cluster cluster) {...}
20
21     @Override
22     public void close() {
23     }
24
25     @Override
26     public void configure(Map<String, ?> configs) {...}
27
28 }
```

# StockPricePartitioner

## partition()



StockPriceKafkaProducer.java x

c StockPricePartitioner.java x

StockPricePartitioner partition()

```
17 ①     public int partition(final String topic,
18                           final Object objectKey,
19                           final byte[] keyBytes,
20                           final Object value,
21                           final byte[] valueBytes,
22                           final Cluster cluster) {
23
24     final List<PartitionInfo> partitionInfoList =
25         cluster.availablePartitionsForTopic(topic);
26     final int partitionCount = partitionInfoList.size();
27     final int importantPartition = partitionCount -1;
28     final int normalPartitionCount = partitionCount -1;
29
30     final String key = ((String) objectKey);
31
32     if (importantStocks.contains(key)) {
33         return importantPartition;
34     } else {
35         return Math.abs(key.hashCode()) % normalPartitionCount;
36     }
37
38 }
```



# StockPricePartitioner

```
StockPriceKafkaProducer.java x StockPricePartitioner.java x
StockPricePartitioner configure()
39
40
41 ①↑   public void close() {
42    }
43
44
45 ①↑   public void configure(Map<String, ?> configs) {
46    final String importantStocksStr = (String) configs.get("importantStocks");
47    Arrays.stream(importantStocksStr.split( regex: "," ))
48      .forEach(importantStocks::add);
49
50 }
```

# Producer Interception



- ❖ **Producer** config property: ***interceptor.classes***
  - ❖ empty (you can pass an comma delimited list)
- ❖ interceptors implementing [ProducerInterceptor](#) interface
- ❖ intercept records producer sent to broker and after acks
- ❖ you could mutate records

# KafkaProducer - Interceptor Config



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer getStockSenderList()
17 > public class StockPriceKafkaProducer {
18     private static final Logger logger = LoggerFactory.getLogger(Stock
19
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         setupBatchingAndCompression(props);
25         setupRetriesInFlightTimeout(props);
26
27         //Install interceptor list - config "interceptor.classes"
28         props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
29                   StockProducerInterceptor.class.getName());
30
31         //Set number of acknowledgments - acks - default is all
32         props.put(ProducerConfig.ACKS_CONFIG, "all");
33
34         return new KafkaProducer<>(props);
35     }
}
```

# KafkaProducer ProducerInterceptor



```
c StockProducerInterceptor.java x
StockProducerInterceptor
10
11 public class StockProducerInterceptor implements ProducerInterceptor {
12
13     private final Logger logger = LoggerFactory
14             .getLogger(StockProducerInterceptor.class);
15     private int onSendCount;
16     private int onAckCount;
17
18     @Override
19     public ProducerRecord onSend(final ProducerRecord record) {...}
36
37     @Override
38     public void onAcknowledgement(final RecordMetadata metadata,
39                                  final Exception exception) {...}
54
55     @Override
56     public void close() {...}
59
60     @Override
61     public void configure(Map<String, ?> configs) {...}
64 }
```

# ProducerInterceptor onSend



```
c StockProducerInterceptor.java x
StockProducerInterceptor
18
19 @Override
20 public ProducerRecord onSend(final ProducerRecord record) {
21     onSendCount++;
22     if (logger.isDebugEnabled()) {
23         logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
24             record.topic(), record.key(), record.value().toString(),
25             record.partition()
26         ));
27     } else {
28         if (onSendCount % 100 == 0) {
29             logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
30                 record.topic(), record.key(), record.value().toString(),
31                 record.partition()
32             ));
33         }
34     }
35     return record;
}
```

## Output

```
topic=stock-prices2 key=UBER value=StockPrice{dollars=737, cents=78, name='
```

# ProducerInterceptor onAck



```
c StockProducerInterceptor.java x
StockProducerInterceptor onAcknowledgement()
37
38 ① @Override
39 ② public void onAcknowledgement(final RecordMetadata metadata,
40 ③ ④ final Exception exception) {
41 ⑤     onAckCount++;
42
43     if (logger.isDebugEnabled()) {
44         logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
45             metadata.topic(), metadata.partition(), metadata.offset()
46         ));
47     } else {
48         if (onAckCount % 100 == 0) {
49             logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
50                 metadata.topic(), metadata.partition(), metadata.offset()
51             ));
52         }
53     }
}
```

## Output

```
onAck topic=stock-prices2, part=0, offset=18360
```

# ProducerInterceptor the rest



```
c StockProducerInterceptor.java x
StockProducerInterceptor configure()
48     } metadata.topic(), metadata.partition
49     });
50   });
51 }
52 }
53 }
54
55 @Override
56 public void close() {
57 }
58
59 @Override
60 public void configure(Map<String, ?> configs) {
61 }
62 }
```

# KafkaProducer send() Method



- ❖ Two forms of send with callback and with no callback both return Future
  - ❖ Asynchronously sends a record to a topic
  - ❖ Callback gets invoked when send has been acknowledged.
- ❖ send is asynchronous and return right away as soon as record has added to send buffer
- ❖ Sending many records at once without blocking for response from Kafka broker
- ❖ Result of send is a RecordMetadata
  - ❖ record partition, record offset, record timestamp
- ❖ Callbacks for records sent to same partition are executed in order

# KafkaProducer send() Exceptions



- ❖ ***InterruptedException*** - If the thread is interrupted while blocked ([API](#))
- ❖ ***SerializationException*** - If key or value are not valid objects given configured serializers ([API](#))
- ❖ ***TimeoutException*** - If time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc. ([API](#))
- ❖ ***KafkaException*** - If Kafka error occurs not in public API.  
([API](#))



# Using send method

```
c StockSender.java x
StockSender run()
51   try {
52
53     final Future<RecordMetadata> future = producer.send(record);
54
55     if (sentCount % 100 == 0) {
56       displayRecordMetaData(record, future);
```

```
c StockSender.java x
StockSender displayRecordMetaData()
74   private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
75                                     final Future<RecordMetadata> future)
76                                     throws InterruptedException, ExecutionException {
77     final RecordMetadata recordMetadata = future.get();
78     logger.info(String.format("\n\t\tkey=%s, value=%s " +
79                           "\n\t\tsent to topic=%s part=%d off=%d at time=%s",
80                           record.key(),
81                           record.value().toJson(),
82                           recordMetadata.topic(),
83                           recordMetadata.partition(),
84                           recordMetadata.offset(),
85                           new Date(recordMetadata.timestamp())
86                           ));
87 }
```

# KafkaProducer flush() method



- ❖ flush() method sends all buffered records now (even *linger.ms > 0*)
  - ❖ blocks until requests complete
- ❖ Useful when consuming from some input system and pushing data into Kafka
- ❖ **flush()** ensures all previously sent messages have been sent
  - ❖ you could mark progress as such at completion of flush

# KafkaProducer close()



- ❖ close() closes producer
  - ❖ frees resources (threads and buffers) associated with producer
- ❖ Two forms of method
- ❖ both block until all previously sent requests complete or duration passed in as args is exceeded
- ❖ close with no params equivalent to close(Long.MAX\_VALUE, TimeUnit.MILLISECONDS).
- ❖ If producer is unable to complete all requests before the timeout expires, all unsent requests fail, and this method fails

# Orderly shutdown using close



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
95  Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97      executorService.shutdown();
98      try {
99          executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS );
100         logger.info("Shutting down executorService for workers nicely");
101     } catch (InterruptedException e) {
102         logger.warn("shutting down", e);
103     }
104
105     logger.info("Flushing producer");
106     producer.flush();
107     logger.info("Closing producer");
108     producer.close();
109
110     if (!executorService.isShutdown()) {
111         logger.info("Forcing shutdown of workers");
112         executorService.shutdownNow();
113     }
114 });

});
```

# Wait for clean close



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
95     Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97         executorService.shutdown();
98
99         try {
100             executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS );
101             logger.info("Flushing and closing producer");
102             producer.flush();
103             producer.close( timeout: 10_000, TimeUnit.MILLISECONDS );
104         } catch (InterruptedException e) {
105             logger.warn("shutting down", e);
106         }
107     }));
108 }
```

# KafkaProducer partitionsFor() method



- ❖ partitionsFor(topic) returns meta data for partitions
- ❖ **public** List<PartitionInfo> partitionsFor(String topic)
- ❖ Get partition metadata for give topic
- ❖ Produce that do their own partitioning would use this
  - ❖ for custom partitioning
  - ❖ PartitionInfo(String topic, int partition, Node leader, Node[] replicas, Node[] inSyncReplicas)
    - ❖ Node(int id, String host, int port, optional String rack)

# KafkaProducer metrics() method



- ❖ metrics() method get map of metrics
- ❖ **public Map<MetricName,? extends Metric>** metrics()
- ❖ Get the full set of producer metrics

MetricName(

```
String name,  
String group,  
String description,  
Map<String, String> tags
```

)

**public interface Metric**  
A numerical metric tracked for monitoring purposes

## Method Summary

### Methods

#### Modifier and Type

**MetricName**

**double**

#### Method and Description

**metricName()**

A name for this metric

**value()**

The value of the metric

# Metrics producer.metrics()



```
c MetricsProducerReporter.java x
MetricsProducerReporter
25
26     final Map<MetricName, ? extends Metric> metrics = producer.metrics();
27
28     metrics.forEach((metricName, metric) ->
29         logger.info(
30             String.format("\nMetric\t %s, \t %s, \t %s, " +
31                         "\n\t\t%s\n",
32             metricName.group(),
33             metricName.name(),
34             metric.value(),
35             metricName.description())
36     );

```

- ❖ Call producer.metrics()
- ❖ Prints out metrics to log

# Metrics producer.metrics() output



Metric producer-metrics, record-queue-time-max, 508.0,  
The maximum time in ms record batches spent in the record accumulator.

17:09:22.721 [pool-1-thread-9] INFO  
c.c.k.p.MetricsProducerReporter - Metric producer-node-  
metrics, request-rate, 0.025031289111389236,  
The average number of requests sent per second.

17:09:22.721 [pool-1-thread-9] INFO c.c.k.p.MetricsProducerReporter -  
Metric producer-metrics, records-per-request-avg,  
205.55263157894737, The average number of records per  
request.

17:09:22.722 [pool-1-thread-9] INFO  
c.c.k.p.MetricsProducerReporter - Metric producer-  
metrics, record-size-avg, 71.02631578947368,  
The average record size

17:09:22.722 [pool-1-thread-9] INFO  
c.c.k.p.MetricsProducerReporter - Metric producer-node-  
metrics, request-size-max, 56.0,  
The maximum size of any request sent in the window.

17:09:22.723 [pool-1-thread-9] INFO c.c.k.p.MetricsProducerReporter -  
Metric producer-metrics, request-size-max, 12058.0,  
The maximum size of any request sent in the window.

# Metrics via JMX



```
~/kafka-training  
$ jconsole
```

Java Monitoring & Management Console

Connection Window Help

pid: 31636 com.intellij.rt.execution.application.AppMain com.cloudurable.kafka.producer.StockPriceKafkaProducer

Overview Memory Threads Classes VM Summary MBeans

Attribute value

Name	Value
compression-rate	0.4308939902619882

Refresh

MBeanAttributeInfo

Name	Value
Attribute:	compression-rate
Name	compression-rate
Description	
Readable	true
Writable	false

Descriptor

Name	Value

request-latency-max  
request-latency-avg  
incoming-byte-rate  
request-size-avg  
outgoing-byte-rate  
request-size-max

- ▶ node--2
- ▶ node--3
- ▶ node-0
- ▶ node-2

▼ producer-topic-metrics

  ▼ StockPriceKafkaProducer

    ▼ stock-prices2

      ▼ Attributes

        record-retry-rate  
        record-send-rate  
        **compression-rate**  
        byte-rate  
        record-error-rate

219

# Complete Lab 5.1

# About the App (DEMO)

# StockPrice App to demo Advanced Producer



- ❖ **StockPrice** - holds a stock price has a name, dollar, and cents
- ❖ **StockPriceKafkaProducer** - Configures and creates **KafkaProducer<String, StockPrice>**, **StockSender** list, ThreadPool (ExecutorService), starts **StockSender** runnable into thread pool
- ❖ **StockAppConstants** - holds topic and broker list
- ❖ **StockPriceSerializer** - can serialize a **StockPrice** into **byte[]**
- ❖ **StockSender** - generates somewhat random stock prices for a given **StockPrice** name, Runnable, 1 thread per StockSender
  - ❖ Shows using **KafkaProducer** from many threads

# StockPrice domain object



c StockPrice.java x

StockPrice

```
1 package com.cloudurable.kafka.producer.model;
2
3 import io.advantageous.boon.json.JsonFactory;
4
5 public class StockPrice {
6
7     private final int dollars;
8     private final int cents;
9     private final String name;
10
11    public String toJson() {
12        return "{" +
13            "\"dollars\": " + dollars +
14            ", \"cents\": " + cents +
15            ", \"name\": \"\" + name + '\"' +
16            '}';
17    }
18}
```

- ❖ has name
- ❖ dollars
- ❖ cents
- ❖ converts itself to JSON

# StockPriceKafkaProducer



- ❖ Import classes and setup logger
- ❖ Create ***createProducer*** method to create ***KafkaProducer*** instance
- ❖ Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- ❖ Write ***main()*** method - creates ***producer***, create ***StockSender*** list passing each instance a ***producer***, creates a thread pool so every stock sender gets its own thread, runs each *stockSender* in its own thread



# StockPriceKafkaProducer imports, createProducer

StockPriceKafkaProducer main()

```
1 package com.cloudurable.kafka.producer;
2
3 import com.cloudurable.kafka.StockAppConstants;
4 import com.cloudurable.kafka.producer.model.StockPrice;
5 import io.advantageous.boon.core.Lists;
6 import org.apache.kafka.clients.producer.*;
7 import org.apache.kafka.common.serialization.StringSerializer;
8 import org.slf4j.Logger;
9 import org.slf4j.LoggerFactory;
10
11 import java.util.List;
12 import java.util.Properties;
13 import java.util.concurrent.ExecutorService;
14 import java.util.concurrent.Executors;
15 import java.util.concurrent.TimeUnit;
16
17 public class StockPriceKafkaProducer {
18     private static final Logger logger =
19         LoggerFactory.getLogger(StockPriceKafkaProducer.class);
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         return new KafkaProducer<>(props);
25     }
}
```

Import classes and  
setup logger

createProducer  
used to create a  
KafkaProducer

createProducer()  
calls  
setupBootstrapAnd  
Serializers()

# Configure Producer Bootstrap and Serializer



StockPriceKafkaProducer

```
27     private static void setupBootstrapAndSerializers(Properties props) {  
28         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,  
29                     StockAppConstants.BOOTSTRAP_SERVERS);  
30         props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");  
31         props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,  
32                     StringSerializer.class.getName());  
33  
34         //Custom Serializer - config "value.serializer"  
35         props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,  
36                     StockPriceSerializer.class.getName());  
37     }
```

- ❖ Create **setupBootstrapAndSerializers** to initialize bootstrap servers, client id, key serializer and custom serializer (**StockPriceSerializer**)
- ❖ **StockPriceSerializer** will serialize **StockPrice** into bytes

# StockPriceKafkaProducer.main()



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
39 >     public static void main(String... args)
40             throws Exception {
41
42             //Create Kafka Producer
43             final Producer<String, StockPrice> producer = createProducer();
44
45             //Create StockSender list
46             final List<StockSender> stockSenders = getStockSenderList(producer);
47
48             //Create a thread pool so every stock sender gets its own.
49             final ExecutorService executorService =
50                 Executors.newFixedThreadPool(stockSenders.size());
51
52             //Run each stock sender in its own thread.
53             stockSenders.forEach(executorService::submit);
54
55 }
```

- ❖ **main** method - creates **producer**,
- ❖ create **StockSender** list passing each instance a **producer**
- ❖ creates a thread pool (executorService)
- ❖ every StockSender runs in its own thread

# StockAppConstants



```
1 package com.cloudurable.kafka;  
2  
3 public class StockAppConstants {  
4     public final static String TOPIC = "stock-prices";  
5     public final static String BOOTSTRAP_SERVERS =  
6         "localhost:9092,localhost:9093,localhost:9094";  
7  
8 }
```

- ❖ topic name for Producer example
- ❖ List of bootstrap servers

# StockPriceKafkaProducer.getStockSenderList



```
StockPriceKafkaProducer getStockSenderList()
57 @     private static List<StockSender> getStockSenderList(
58         final Producer<String, StockPrice> producer) {
59     return Lists.list(
60         new StockSender(StockAppConstants.TOPIC,
61             new StockPrice( name: "IBM",    dollars: 100,    cents: 99),
62             new StockPrice( name: "IBM",    dollars: 50,     cents: 10),
63             producer,
64             delayMinMs: 1,    delayMaxMs: 10
65         ),
66         new StockSender(
67             StockAppConstants.TOPIC,
68             new StockPrice( name: "SUN",    dollars: 100,    cents: 99),
69             new StockPrice( name: "SUN",    dollars: 50,     cents: 10),
70             producer,
71             delayMinMs: 1,    delayMaxMs: 10
72         ),
73         new StockSender(
74             StockAppConstants.TOPIC,
75             new StockPrice( name: "GOOG",   dollars: 500,    cents: 99),
76             new StockPrice( name: "GOOG",   dollars: 400,    cents: 10),
77             producer,
78             delayMinMs: 1,    delayMaxMs: 10
79         ),
80         new StockSender(
81             StockAppConstants.TOPIC,
82             new StockPrice( name: "INEL",   dollars: 100,    cents: 99),
83             new StockPrice( name: "INEL",   dollars: 50,     cents: 10),
84             producer,
85             delayMinMs: 1,    delayMaxMs: 10
)
```



# StockPriceSerializer

```
1 package com.cloudurable.kafka.producer;
2 import com.cloudurable.kafka.producer.model.StockPrice;
3 import org.apache.kafka.common.serialization.Serializer;
4 import java.nio.charset.StandardCharsets;
5 import java.util.Map;
6
7 public class StockPriceSerializer implements Serializer<StockPrice> {
8
9     @Override
10    public byte[] serialize(String topic, StockPrice data) {
11        return data.toJson().getBytes(StandardCharsets.UTF_8);
12    }
13
14    @Override
15    public void configure(Map<String, ?> configs, boolean isKey) {
16    }
17
18    @Override
19    public void close() {
20    }
21}
```

- ❖ Converts **StockPrice** into byte array



# StockSender

- ❖ Generates random stock prices for a given ***StockPrice*** name,
- ❖ StockSender is Runnable
- ❖ 1 thread per StockSender
- ❖ Shows using ***KafkaProducer*** from many threads
- ❖ Delays random time between delayMin and delayMax,
  - ❖ then sends random StockPrice between ***stockPriceHigh*** and ***stockPriceLow***

# StockSender imports, Runnable



```
StockSender
1 package com.cloudurable.kafka.producer;
2
3 import com.cloudurable.kafka.producer.model.StockPrice;
4 import org.apache.kafka.clients.producer.Producer;
5 import org.apache.kafka.clients.producer.ProducerRecord;
6 import org.apache.kafka.clients.producer.RecordMetadata;
7 import org.slf4j.Logger;
8 import org.slf4j.LoggerFactory;
9
10 import java.util.Date;
11 import java.util.Random;
12 import java.util.concurrent.ExecutionException;
13 import java.util.concurrent.Future;
14
15 public class StockSender implements Runnable{
```

- ❖ Imports Kafka *Producer*, *ProducerRecord*, *RecordMetadata*, *StockPrice*
- ❖ Implements Runnable, can be submitted to *ExecutionService*



# StockSender constructor

## StockSender

```
17     private final StockPrice stockPriceHigh;
18     private final StockPrice stockPriceLow;
19     private final Producer<String, StockPrice> producer;
20     private final int delayMinMs;
21     private final int delayMaxMs;
22     private final Logger logger = LoggerFactory.getLogger(StockSender.class);
23     private final String topic;
24
25     public StockSender(final String topic, final StockPrice stockPriceHigh,
26                         final StockPrice stockPriceLow,
27                         final Producer<String, StockPrice> producer,
28                         final int delayMinMs,
29                         final int delayMaxMs) {
30         this.stockPriceHigh = stockPriceHigh;
31         this.stockPriceLow = stockPriceLow;
32         this.producer = producer;
33         this.delayMinMs = delayMinMs;
34         this.delayMaxMs = delayMaxMs;
35         this.topic = topic;
36     }
```

- ❖ takes a topic, high & low stockPrice, producer, delay min & max



# StockSender run()

```
StockSender run()

38
39 ①↑ public void run() {
40     final Random random = new Random(System.currentTimeMillis());
41     int sentCount = 0;
42
43     while (true) {
44         sentCount++;
45         final ProducerRecord <String, StockPrice> record =
46             createRandomRecord(random);
47         final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);
48
49         try {
50             final Future<RecordMetadata> future = producer.send(record);
51             if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
52             Thread.sleep(delay);
53         } catch (InterruptedException e) {
54             if (Thread.interrupted()) {
55                 break;
56             }
57         } catch (ExecutionException e) {
58             logger.error("problem sending record to producer", e);
59         }
60     }
61 }
```

- ❖ In loop, creates random record, **send** record, waits random time

# StockSender

## createRandomRecord



```
StockSender createRandomRecord()

77
78     private final int randomIntBetween(final Random random,
79                                         final int max,
80                                         final int min) {
81         return random.nextInt( bound: max - min + 1 ) + min;
82     }
83
84     private ProducerRecord<String, StockPrice> createRandomRecord(
85         final Random random) {
86
87         final int dollarAmount = randomIntBetween(random,
88             stockPriceHigh.getDollars(), stockPriceLow.getDollars());
89
90         final int centAmount = randomIntBetween(random,
91             stockPriceHigh.getCents(), stockPriceLow.getCents());
92
93         final StockPrice stockPrice = new StockPrice(
94             stockPriceHigh.getName(), dollarAmount, centAmount);
95
96         return new ProducerRecord<>(topic, stockPrice.getName(),
97             stockPrice);
98     }
```

- ❖ *createRandomRecord* uses *randomIntBetween*
- ❖ creates **StockPrice** and then wraps **StockPrice** in **ProducerRecord**



# StockSender

## displayRecordMetaData

```
StockSender displayRecordMetaData()
```

```
62
63     private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
64                                         final Future<RecordMetadata> future)
65                                         throws InterruptedException, ExecutionException {
66         final RecordMetadata recordMetadata = future.get();
67         logger.info(String.format("\n\t\tkey=%s, value=%s " +
68                         "\n\t\tsent to topic=%s part=%d off=%d at time=%s",
69                         record.key(),
70                         record.value().toJson(),
71                         recordMetadata.topic(),
72                         recordMetadata.partition(),
73                         recordMetadata.offset(),
74                         new Date(recordMetadata.timestamp())
75                     ));
76     }
```

- ❖ Every 100 records `displayRecordMetaData` gets called
- ❖ Prints out record info, and recordMetadata info:
  - ❖ key, JSON value, topic, partition, offset, time
  - ❖ uses Future from call to `producer.send()`

# Run it

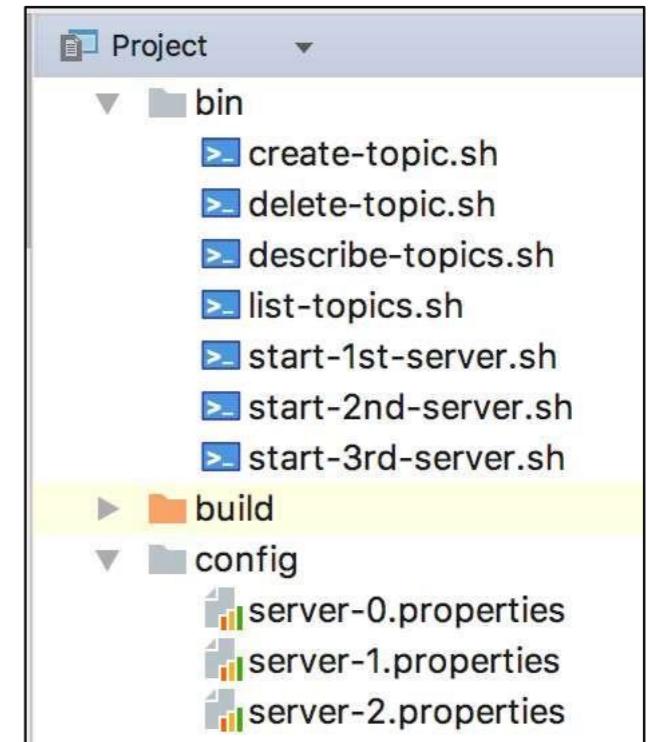


- ❖ Run ZooKeeper
- ❖ Run three Brokers
- ❖ run `create-topic.sh`
- ❖ Run ***StockPriceKafkaProducer***



# Run scripts

- ❖ run ZooKeeper from ~/kafka-training
- ❖ use ***bin/create-topic.sh*** to create topic
- ❖ use ***bin/delete-topic.sh*** to delete topic
- ❖ use ***bin/start-1st-server.sh*** to run Kafka Broker 0
- ❖ use ***bin/start-2nd-server.sh*** to run Kafka Broker 1
- ❖ use ***bin/start-3rd-server.sh*** to run Kafka Broker 2



Config is under directory called config  
server-0.properties is for Kafka Broker 0  
server-1.properties is for Kafka Broker 1  
server-2.properties is for Kafka Broker 2



# Run All 3 Brokers

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-0.properties"
7
```

```
start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-1.properties"
7
```

```
start-3rd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-2.properties"
7
```

```
server-0.properties x
1 broker.id=0
2 port=9092
3 log.dirs=./logs/kafka-0
4 min.insync.replicas=3
5 compression.type=producer
6 auto.create.topics.enable=false
7 message.max.bytes=65536
8 replica.lag.time.max.ms=5000
9 delete.topic.enable=true
```

```
server-1.properties x
1 broker.id=1
2 port=9093
3 log.dirs=./logs/kafka-1
4 min.insync.replicas=3
5 compression.type=producer
```

```
server-2.properties x
1 broker.id=2
2 port=9094
3 log.dirs=./logs/kafka-2
4 min.insync.replicas=3
5 compression.type=producer
6 auto.create.topics.enable=false
7 message.max.bytes=65536
8 replica.lag.time.max.ms=5000
9 delete.topic.enable=true
```

# Run create-topic.sh script



```
create-topic.sh >_ x  
1 #!/usr/bin/env bash  
2  
3 cd ~/kafka-training  
4  
5 kafka/bin/kafka-topics.sh \  
6   --create \  
7   --zookeeper localhost:2181 \  
8   --replication-factor 3 \  
9   --partitions 3 \  
10  --topic stock-prices  
11
```

Name of the topic  
is stock-prices

Three partitions

Replication factor  
of three

```
Terminal  
+ Local Local (1) Local (2) Local (3) Local (4)  
✖ $ ./bin/create-topic.sh  
Created topic "stock-prices".
```

# Run StockPriceKafkaProducer



Run StockPriceKafkaProducer

```
key=ABC, value={"dollars": 68, "cents": 37, "name": "ABC"}  
sent to topic=stock-prices part=2 off=41360 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.024 [pool-1-thread-6] INFO c.c.kafka.producer.StockSender -  
key=G00G, value={"dollars": 498, "cents": 37, "name": "G00G"}  
sent to topic=stock-prices part=0 off=48256 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.049 [pool-1-thread-3] INFO c.c.kafka.producer.StockSender -  
key=IBM, value={"dollars": 68, "cents": 37, "name": "IBM"}  
sent to topic=stock-prices part=0 off=48287 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.049 [pool-1-thread-1] INFO c.c.kafka.producer.StockSender -  
key=DEF, value={"dollars": 63, "cents": 26, "name": "DEF"}  
sent to topic=stock-prices part=2 off=41399 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.171 [pool-1-thread-8] INFO c.c.kafka.producer.StockSender -
```

- ❖ Run ***StockPriceKafkaProducer*** from the IDE
  - ❖ You should see log messages from ***StockSender(s)*** with ***StockPrice*** name, JSON value, partition, offset, and time

# Producer Shutdown

# Shutdown Producer nicely



- ❖ Handle ctrl-C shutdown from Java
- ❖ Shutdown thread pool and wait
- ❖ Flush producer to send any outstanding batches if using batches (***producer.flush()***)
- ❖ Close Producer (***producer.close()***) and wait

# Nice Shutdown producer.close()



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
41 > 1 public static void main(String... args)
42           throws Exception {
43             //Create Kafka Producer
44             final Producer<String, StockPrice> producer = createProducer();
45             //Create StockSender list
46             final List<StockSender> stockSenders = getStockSenderList(producer);
47             //Create a thread pool so every stock sender gets its own.
48             final ExecutorService executorService =
49               Executors.newFixedThreadPool(stockSenders.size());
50             //Run each stock sender in its own thread.
51             stockSenders.forEach(executorService::submit);
52
53             //Register nice shutdown of thread pool, then flush and close producer.
54             Runtime.getRuntime().addShutdownHook(new Thread(() -> {
55               executorService.shutdown();
56               try {
57                 executorService.awaitTermination(timeout: 200, TimeUnit.MILLISECONDS);
58                 logger.info("Flushing and closing producer");
59                 producer.flush();
60                 producer.close(timeout: 10_000, TimeUnit.MILLISECONDS);
61               } catch (InterruptedException e) {
62                 logger.warn("shutting down", e);
63               }
64             }));
65           });
}
```

# Restart Producer then shut it down



- ❖ Add shutdown hook
- ❖ Start StockPriceKafkaProducer
- ❖ Now stop it (CTRL-C or hit stop button in IDE)

```
Run StockPriceKafkaProducer
00:06:52.709 [pool-1-thread-4] INFO c.c.kafka.producer.StockSender -
    key=INEL, value={"dollars": 73, "cents": 10, "name": "INEL"}
    sent to topic=stock-prices part=0 off=93522 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.710 [pool-1-thread-5] INFO c.c.kafka.producer.StockSender -
    key=UBER, value={"dollars": 547, "cents": 90, "name": "UBER"}
    sent to topic=stock-prices part=2 off=80143 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.717 [pool-1-thread-2] INFO c.c.kafka.producer.StockSender -
    key=SUN, value={"dollars": 73, "cents": 10, "name": "SUN"}
    sent to topic=stock-prices part=2 off=80146 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.718 [pool-1-thread-1] INFO c.c.kafka.producer.StockSender -
    key=IBM, value={"dollars": 73, "cents": 10, "name": "IBM"}
    sent to topic=stock-prices part=2 off=80148 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.967 [Thread-1] INFO c.c.k.p.StockPriceKafkaProducer - Flushing and closing producer
00:06:52.968 [Thread-1] INFO o.a.k.clients.producer.KafkaProducer - Closing the Kafka producer
?
Process finished with exit code 130 (interrupted by signal 2: SIGINT)
```

Complete Lab 5.2 (just read  
but don't complete) and  
Complete Labs 5.3-5.8

# Kafka Low Level Design

# Kafka Design Motivation Goals



- ❖ Kafka built to support real-time analytics
  - ❖ Designed to feed analytics system that did real-time processing of streams
  - ❖ Unified platform for real-time handling of streaming data feeds
- ❖ Goals:
  - ❖ high-throughput streaming data platform
  - ❖ supports high-volume event streams like log aggregation, user activity, etc.

# Kafka Design Motivation Scale



- ❖ To scale Kafka is
  - ❖ distributed,
  - ❖ supports sharding
  - ❖ load balancing
- ❖ Scaling needs inspired Kafka partitioning and consumer model
- ❖ Kafka scales writes and reads with partitioned, distributed, commit logs

# Kafka Design Motivation Use Cases



- ❖ Also designed to support these Use Cases
  - ❖ Handle periodic large data loads from offline systems
  - ❖ Handle traditional messaging use-cases, low-latency.
- ❖ Like MOMs, Kafka is fault-tolerance for node failures through replication and leadership election
- ❖ Design more like a distributed database transaction log
- ❖ Unlike MOMs, replication, scale not afterthought

# Persistence: Embrace filesystem



- ❖ Kafka relies heavily on filesystem for storing and caching messages/records
- ❖ Disk performance of hard drives performance of sequential writes is fast
  - ❖ JBOD with six 7200rpm SATA RAID-5 array clocks at 600MB/sec
  - ❖ Heavily optimized by operating systems
- ❖ Ton of cache: Operating systems use available of main memory for disk caching
- ❖ JVM GC overhead is high for caching objects OS file caches are almost free
- ❖ Kafka greatly simplifies code for cache coherence by using OS page cache
- ❖ Kafka disk does sequential reads easily optimized by OS page cache

# Big fast HDDs and long sequential access



- ❖ Like Cassandra, LevelDB, RocksDB, and others, Kafka uses long sequential disk access for read and writes
- ❖ Kafka uses tombstones instead of deleting records right away
- ❖ Modern Disks have somewhat unlimited space and are fast
- ❖ Kafka can provide features not usually found in a messaging system like holding on to old messages for a really long time
  - ❖ This flexibility allows for interesting application of Kafka

# Kafka Record Retention Redux



- ❖ Kafka cluster retains all published records
  - ❖ Time based – configurable retention period
  - ❖ Size based - configurable based on size
  - ❖ Compaction - keeps latest record
- ❖ Kafka uses Topic ***Partitions***
- ❖ Partitions are broken down into ***Segment*** files



# Broker Log Config

Kafka Broker Config for Logs

NAME	DESCRIPTION	DEFAULT
log.dir	Log Directory will topic logs will be stored use this or log.dirs.	/tmp/kafka-logs
log.dirs	The directories where the Topics logs are kept used for JBOD.	
log.flush.interval.messages	Accumulated messages count on a log partition before messages are flushed to disk.	9,223,372,036,854,780,000
log.flush.interval.ms	Maximum time that a topic message is kept in memory before flushed to disk. If not set, uses log.flush.scheduler.interval.ms.	
log.flush.offset.checkpoint.interval.ms	Interval to flush log recovery point.	60,000
log.flush.scheduler.interval.ms	Interval that topic messages are periodically flushed from memory to log.	9,223,372,036,854,780,000 254

# Broker Log Retention Config



Kafka Broker Config for Logs

log.retention.bytes	Delete log records by size. The maximum size of the log before deleting its older records.	long	-1
log.retention.hours	Delete log records by time hours. Hours to keep a log file before deleting older records (in hours), tertiary to log.retention.ms property.	int	168
log.retention.minutes	Delete log records by time minutes. Minutes to keep a log file before deleting it, secondary to log.retention.ms property. If not set, use log.retention.hours is used.	int	null
log.retention.ms	Delete log records by time milliseconds. Milliseconds to keep a log file before deleting it, If not set, use log.retention.minutes.	long	null

# Broker Log Segment File Config



Kafka Broker Config - Log Segments

NAME	DESCRIPTION	TYPE	DEFAULT
log.roll.hours	Time period before rolling a new topic log segment. (secondary to log.roll.ms property)	int	168
log.roll.ms	Time period in milliseconds before rolling a new log segment. If not set, uses log.roll.hours.	long	
log.segment.bytes	The maximum size of a single log segment file.	int	1,073,741,824
log.segment.delete.delay.ms	Time period to wait before deleting a segment file from the filesystem.	long	60,000

# Kafka Producer Load Balancing



- ❖ Producer sends records directly to Kafka broker partition leader
- ❖ Producer asks Kafka broker for metadata about which Kafka broker has which topic partitions leaders - thus no routing layer needed
- ❖ Producer client controls which partition it publishes messages to
- ❖ Partitioning can be done by key, round-robin or using a custom semantic partitioner

# Kafka Producer Record Batching



- ❖ Kafka producers support record batching. by the size of records and auto-flushed based on time
- ❖ Batching is good for network IO throughput.
- ❖ Batching speeds up throughput drastically.
- ❖ Buffering is configurable
  - ❖ lets you make a tradeoff between additional latency for better throughput.
  - ❖ Producer sends multiple records at a time which equates to fewer IO requests instead of lots of one by one sends

[QBit a microservice library](#) uses message batching in an identical fashion as K to send messages over WebSocket between nodes and from client to QBit server.

# More producer settings for performance



```
KafkaExample.java x
KafkaExample
21  private static Producer<Long, String> createProducer() {
22      Properties props = new Properties();
23      props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
24      props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
25      props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
26      props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
27
28      //The batch.size in bytes of record size, 0 disables batching
29      props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
30
31      //Linger how much to wait for other records before sending the batch over the network.
32      props.put(ProducerConfig.LINGER_MS_CONFIG, 20);
33
34      // The total bytes of memory the producer can use to buffer records waiting to be sent
35      // to the Kafka broker. If records are sent faster than broker can handle than
36      // the producer blocks. Used for compression and in-flight records.
37      props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67_108_864);
38
39      //Control how much time Producer blocks before throwing BufferExhaustedException.
40      props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 1000);
41
```

For higher throughput, Kafka Producer allows buffering based on time and size.

Multiple records can be sent as a batches with fewer network requests.

Speeds up throughput drastically.



# Kafka Compression

- ❖ Kafka provides ***End-to-end Batch Compression***
- ❖ Bottleneck is not always CPU or disk but often network bandwidth
  - ❖ especially in cloud, containerized and virtualized environments
  - ❖ especially when talking datacenter to datacenter or WAN
- ❖ Instead of compressing records one at a time, compresses whole batch
- ❖ Message batches can be compressed and sent to Kafka broker/server in one go
- ❖ Message batch get written in compressed form in log partition
  - ❖ don't get decompressed until they consumer
- ❖ GZIP, Snappy and LZ4 compression protocols supported

# Kafka Compression Config



## Kafka Broker Compression Config

compression.type	<p>Configures compression type for topics. Can be set to codecs 'gzip', 'snappy', 'lz4' or 'uncompressed'. If set to 'producer' then it retains compression codec set by the producer (so it does not have to be uncompressed and then recompressed).</p>	Default: producer
------------------	---	----------------------

# Pull vs. Push/Streams: Pull



- ❖ With Kafka consumers ***pull*** data from brokers
- ❖ Other systems are push based or stream data to consumers
- ❖ Messaging is usually a pull-based system (SQS, most MOM is pull)
  - ❖ if consumer fall behind, it catches up later when it can
- ❖ Pull-based can implement aggressive batching of data
- ❖ Pull based systems usually implement some sort of ***long poll***
  - ❖ long poll keeps a connection open for response after a request for a period
- ❖ Pull based systems have to pull data and then process it
  - ❖ There is always a pause between the pull

# Pull vs. Push/Streams: Push



- ❖ Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka)
  - ❖ push-based have problems dealing with slow or dead consumers
  - ❖ push system consumer can get overwhelmed
  - ❖ push based systems use back-off protocol (back pressure)
    - ❖ consumer can indicate it is overwhelmed, (<http://www.reactive-streams.org/>)
- ❖ Push-based streaming system can
  - ❖ send a request immediately or accumulate request and send in batches
- ❖ Push-based systems are always pushing data or streaming data
  - ❖ Advantage: Consumer can accumulate data while it is processing data already sent
  - ❖ Disadvantage: If consumer dies, how does broker know and when does data get resent to another consumer (harder to manage message acks; more complex)

# MOM Consumer Message State



- ❖ With most MOM it is brokers responsibility to keep track of which messages have been consumed
- ❖ As message is consumed by a consumer, broker keeps track
  - ❖ broker may delete data quickly after consumption
  - ❖ Trickier than it sounds (acknowledgement feature), lots of state to track per message, sent, acknowledge

# Kafka Consumer Message State



- ❖ Kafka topic is divided into ordered partitions - A topic partition gets read by only one ***consumer*** per ***consumer group***
- ❖ Offset data is not tracked per message - ***a lot less data to track***
  - ❖ just stores offset of each ***consumer group, partition pairs***
  - ❖ Consumer sends offset Data periodically to Kafka Broker
  - ❖ Message acknowledgement is cheap compared to MOM
- ❖ Consumer can rewind to older offset (replay)
  - ❖ If bug then fix, rewind consumer and replay

# Message Delivery Semantics



- ❖ At most once
  - ❖ Messages may be lost but are never redelivered
- ❖ At least once
  - ❖ Messages are never lost but may be redelivered
- ❖ Exactly once
  - ❖ this is what people actually want, each message is delivered once and only once

# Consumer: Message Delivery Semantics



- ❖ "at-most-once" - Consumer reads message, save offset, process message
  - ❖ Problem: consumer process dies after saving position but before processing message - consumer takes over starts at last position and message never processed
- ❖ "at-least-once" - Consumer reads message, process messages, saves offset
  - ❖ Problem: consumer could crash after processing message but before saving position - consumer takes over receives already processed message
- ❖ "exactly once" - need a two-phase commit for consumer position, and message process output - or, store consumer message process output in same location as last position
- ❖ Kafka offers the first two and you can implement the third

# Kafka Producer Acknowledgement



- ❖ Kafka's offers operational predictable semantics
- ❖ When publishing a message, message get **committed** to the log
  - ❖ Durable as long as at least one replica lives
- ❖ If Producer connection goes down during of send
  - ❖ Producer not sure if message sent; resends until message sent ack received (log could have duplicates)
  - ❖ Important: use message keys, idempotent messages
  - ❖ Not guaranteed to not duplicate from producer retry

# Producer Durability Levels



- ❖ Producer can specify durability level
- ❖ Producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message
- ❖ Producer can send with no acknowledgments (0)
- ❖ Producer can send with acknowledgment from partition leader (1)
- ❖ Producer can send and wait on acknowledgments from all replicas (-1) (default)
- ❖ As of June 2017: producer can ensure a message or group of messages was sent "exactly once"

# Improved Producer (coming soon)



- ❖ New feature:
  - ❖ exactly once delivery from producer, atomic write across partitions, (coming soon),
  - ❖ producer sends sequence id, broker keeps track if producer already sent this sequence
  - ❖ if producer tries to send it again, it gets ack for duplicate, but nothing is save to log
  - ❖ NO API changed

# Coming Soon: Kafka Atomic Log Writes



## New Producer API for transactions

```
producer.initTransaction();

try {
    producer.beginTransaction();
    producer.send(debitAccountMessage);
    producer.send(creditOtherAccountMessage);
    producer.sentOffsetsToTxn(...);
    producer.commitTransaction();
} catch (ProducerFencedTransactionException pfte) {
    ...
    producer.close();
} catch (KafkaException ke) {
    ...
    producer.abortTransaction();
}
```

Consumer only see committed logs

Marker written to log to sign what has been successful transacted

Transaction coordinator and transaction log maintain sta

New producer API for transactions

# Kafka Replication



- ❖ Kafka replicates each topic's partitions across a configurable number of Kafka brokers
- ❖ Kafka is replicated by default not a bolt-on feature
- ❖ Each topic partition has one leader and zero or more followers
  - ❖ leaders and followers are called replicas
  - ❖ replication factor = 1 leader + N followers
- ❖ Reads and writes always go to leader
- ❖ Partition leadership is evenly shared among Kafka brokers
  - ❖ logs on followers are in-sync to leader's log - identical copy - sans un-replicated offsets
- ❖ Followers pull records in batches records from leader like a regular Kafka consumer

# Kafka Broker Failover



- ❖ Kafka keeps track of which Kafka Brokers are alive (in-sync)
  - ❖ To be alive Kafka Broker must maintain a ZooKeeper session (heart beat)
  - ❖ Followers must replicate writes from leader and not fall "too far" behind
- ❖ Each leader keeps track of set of "in sync replicas" aka ISRs
- ❖ If ISR/follower dies, falls behind, leader will removes follower from ISR set - falling behind ***replica.lag.time.max.ms > lag***
- ❖ Kafka guarantee: committed message not lost, as long as one live ISR - "committed" when written to all ISRs logs
- ❖ Consumer only reads committed messages

# Replicated Log Partitions

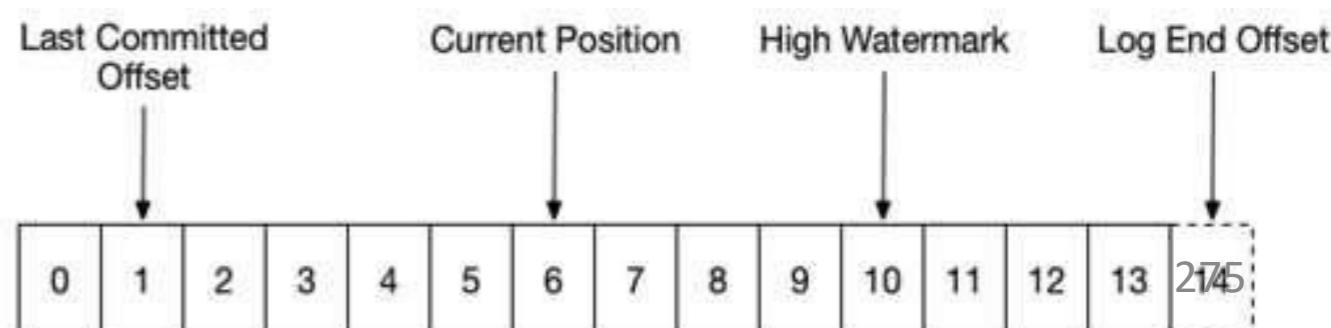


- ❖ A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- ❖ Replicated log useful for building distributed systems using state-machines
- ❖ A replicated log models “coming into consensus” on ordered series of values
  - ❖ While leader stays alive, all followers just need to copy values and ordering from leader
- ❖ When leader does die, a new leader is chosen from its in-sync followers
- ❖ If producer told a message is committed, and then leader fails, new elected leader must have that committed message
- ❖ More ISRs; more to elect during a leadership failure

# Kafka Consumer Replication Redux



- ❖ What can be consumed?
- ❖ "**Log end offset**" is offset of last record written to log partition and where **Producers** write to next
- ❖ "**High watermark**" is offset of last record successfully replicated to all partitions followers
- ❖ **Consumer** only reads up to "high watermark".  
**Consumer can't read un-replicated data**





# Kafka Broker ReplicationConfig

Kafka Broker Config

NAME	DESCRIPTION	TYPE	DEFAULT
<b>auto.leader.rebalance.enable</b>	Enables auto leader balancing.	boolean	TRUE
<b>leader.imbalance.check.interval.seconds</b>	The interval for checking for partition leadership balancing.	long	300
<b>leader.imbalance.per.broker.percentage</b>	Leadership imbalance for each broker. If imbalance is too high then a rebalance is triggered.	int	10
<b>min.insync.replicas</b>	When a producer sets acks to all (or -1), This setting is the minimum replicas count that must acknowledge a write for the write to be considered successful. If not met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).	int	1
<b>num.replica.fetchers</b>	Replica fetcher count. Used to replicate messages from a broker that has a leadership partition. Increase this if followers are falling behind.	int	1

# Kafka Replication Broker Config 2



Kafka Broker Config

NAME	DESCRIPTION
<b>replica.high.watermark.checkpoint.interval.ms</b>	The frequency with which the high watermark is saved out to disk used for knowing what consumers can consume. <b>Consumer</b> only reads up to “high watermark”. <b>Consumer can’t read un-replicated data.</b>
<b>replica.lag.time.max.ms</b>	Determines which Replicas are in the ISR set and which are not. ISR is important for acks and quorum.
<b>replica.socket.receive.buffer.bytes</b>	The socket receive buffer for network requests
<b>replica.socket.timeout.ms</b>	The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms
<b>unclean.leader.election.enable</b>	What happens if all of the nodes go down?  Indicates whether to enable replicas not in the ISR. Replicas that are not in-sync. Set to be elected as leader as a last resort, even though doing so may result in data loss. Availability over Consistency. True is the default.

# Kafka and Quorum



- ❖ Quorum is number of acknowledgements required and number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap
- ❖ Most systems use a majority vote - Kafka does not use a majority vote
- ❖ Leaders are selected based on having the most complete log
- ❖ Problem with majority vote Quorum is it does not take many failure to have inoperable cluster

# Kafka and Quorum 2



- ❖ If we have a replication factor of 3
  - ❖ Then at least two ISRs must be in-sync before the leader declares a sent message committed
  - ❖ If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages
  - ❖ Among the followers there must be at least one replica that contains all committed messages

# Kafka Quorum Majority of ISRs



- ❖ Kafka maintains a set of ISRs
- ❖ Only this set of ISRs are eligible for leadership election
- ❖ Write to partition is not committed until all ISRs ack write
- ❖ ISRs persisted to ZooKeeper whenever ISR set changes

# Kafka Quorum Majority of ISRs 2



- ❖ Any replica that is member of ISRs are eligible to be elected leader
- ❖ Allows producers to keep working with out majority nodes
- ❖ Allows a replica to rejoin ISR set
  - ❖ must fully re-sync again
  - ❖ even if replica lost un-flushed data during crash

# All nodes die at same time. Now what?



- ❖ Kafka's guarantee about data loss is only valid if at least one replica being in-sync
- ❖ If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid.
- ❖ If all replicas are down for a partition, Kafka chooses first replica (not necessarily in ISR set) that comes alive as the leader
  - ❖ Config ***unclean.leader.election.enable=true*** is default
  - ❖ If ***unclean.leader.election.enable=false***, if all replicas are down for a partition, Kafka waits for the ISR member that comes alive as new leader.

# Producer Durability Acks



- ❖ Producers can choose durability by setting **acks** to - 0, 1 or all replicas
- ❖ **acks=all** is **default**, acks happens when all current in-sync replicas (ISR) have received the message
- ❖ If durability over availability is prefer
  - ❖ Disable unclean leader election
  - ❖ Specify a minimum ISR size
    - ❖ trade-off between consistency and availability
    - ❖ higher minimum ISR size guarantees better consistency
    - ❖ but higher minimum ISR reduces availability since partition won't be unavailable for writes if size of ISR set is less than threshold



# Quotas

- ❖ Kafka has quotas for Consumers and Producers
- ❖ Limits bandwidth they are allowed to consume
- ❖ Prevents Consumer or Producer from hogging up all Broker resources
- ❖ Quota is by client id or user
- ❖ Data is stored in ZooKeeper; changes do not necessitate restarting Kafka

# ? Kafka Low-Level Review



- ❖ How would you prevent a denial of service attack from a poorly written consumer?
- ❖ What is the default producer durability (acks) level?
- ❖ What happens by default if all of the Kafka nodes go down at once?
- ❖ Why is Kafka record batching important?
- ❖ What are some of the design goals for Kafka?
- ❖ What are some of the new features in Kafka as of June 2017?
- ❖ What are the different message delivery semantics?

# Log Compaction

# Kafka Log Compaction Overview



- ❖ Recall Kafka can delete older records based on
  - ❖ time period
  - ❖ size of a log
- ❖ Kafka also supports log compaction for record key compaction
- ❖ Log compaction: keep latest version of record and delete older versions



# Log Compaction

- ❖ Log compaction retains last known value for each record key
- ❖ Useful for restoring state after a crash or system failure, e.g., in-memory service, persistent data store, reloading a cache
- ❖ Data streams is to log changes to keyed, mutable data,
  - ❖ e.g., changes to a database table, changes to object in in-memory microservice
- ❖ Topic log has full snapshot of final values for every key - not just recently changed keys
- ❖ Downstream consumers can restore state from a log compacted topic

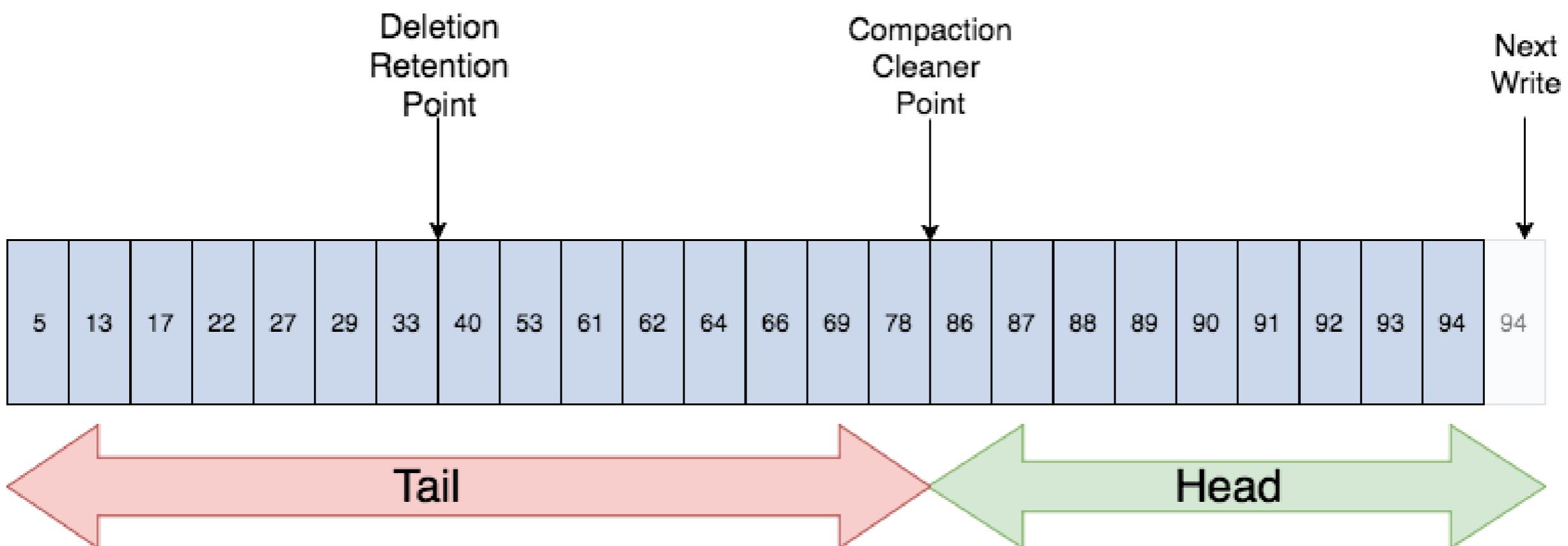
# Log Compaction Structure



- ❖ Log has head and tail
- ❖ Head of compacted log identical to a traditional Kafka log
- ❖ New records get appended to the head
- ❖ Log compaction works at tail of the log
- ❖ Tail gets compacted
- ❖ Records in tail of log retain their original offset when written after compaction



# Compaction Tail/Head



# Log Compaction Basics



- ❖ All offsets remain valid, even if record at offset has been compacted away (next highest offset)
- ❖ Compaction also allows for deletes. A message with a key and a null payload acts like a tombstone (a delete marker for that key)
  - ❖ Tombstones get cleared after a period.
- ❖ Log compaction periodically runs in background by recopying log segments.
- ❖ Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers

# Log Compaction Cleaning



## Before Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

## Cleaning

Only keeps latest version  
of key. Older duplicates not  
needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

## After Compaction

# Log Compaction Guarantees



- ❖ If consumer stays caught up to head of the log, it sees every record that is written.
  - ❖ Topic config ***min.compaction.lag.ms*** used to guarantee minimum period that must pass before message can be compacted.
- ❖ Consumer sees all tombstones as long as the consumer reaches head of log in a period less than the topic config ***delete.retention.ms*** (the default is 24 hours).
- ❖ Compaction will never re-order messages, just remove some.
- ❖ Offset for a message never changes.
- ❖ Any consumer reading from start of the log, sees at least final state of all records in order they were written



# Log Cleaner

- ❖ Log cleaner does log compaction.
  - ❖ Has a pool of background compaction threads that recopy log segments, removing records whose key appears in head of log
- ❖ Each compaction thread works as follows:
  - ❖ Chooses topic log that has highest ratio: log head to log tail
  - ❖ Recopies log from start to end removes records whose keys occur later
- ❖ As log partition segments cleaned, they get swapped into log partition
  - ❖ Additional disk space required: only one log partition segment
  - ❖ not whole partition

# Topic Config for Log Compaction



- ❖ To turn on compaction for a topic
  - ❖ topic config ***log.cleanup.policy=compact***
- ❖ To start compacting records after they are written
  - ❖ topic config ***log.cleaner.min.compaction.lag.ms***
  - ❖ Records wont be compacted until after this period

# Broker Config for Log Compaction



## Kafka Broker Log Compaction Config

NAME	DESCRIPTION	TYPE	DEFAULT
<b>log.cleaner.backoff.ms</b>	Sleep period when no logs need cleaning	long	15,000
<b>log.cleaner.dedupe.buffer.size</b>	The total memory for log dedupe process for all cleaner threads	long	134,217,728
<b>log.cleaner.delete.retention.ms</b>	How long record delete markers (tombstones) are retained.	long	86,400,000
<b>log.cleaner.enable</b>	Turn on the Log Cleaner. You should turn this on if any topics are using clean.policy=compact.	boolean	TRUE
<b>log.cleaner.io.buffer.size</b>	Total memory used for log cleaner I/O buffers for all cleaner threads	int	524,288
<b>log.cleaner.io.max.bytes.per.second</b>	This is a way to throttle the log cleaner if it is taking up too much time.	double	1.7976931348623157E308
<b>log.cleaner.min.cleanable.ratio</b>	The minimum ratio of dirty head log to total log (head and tail) for a log to get selected for cleaning.	double	0.5
<b>log.cleaner.min.compaction.lag.ms</b>	Minimum time period a new message will remain uncompacted in the log.	long	0
<b>log.cleaner.threads</b>	Threads count used for log cleaning. Increase this if you have a lot of log compaction going on across many topic log partitions.	int	1
<b>log.cleanup.policy</b>	The default cleanup policy for segment files that are beyond their retention window. Valid policies are: "delete" and "compact". You could use log compaction just for older segment files. instead of deleting them, you could just compact them.	list	[delete]

# ? Log Compaction Review



- ❖ What are three ways Kafka can delete records?
- ❖ What is log compaction good for?
- ❖ What is the structure of a compacted log? Describe the structure.
- ❖ After compaction, do log record offsets change?
- ❖ What is a partition segment?

# Introduction to Consumer

# Objectives Create a Consumer



- ❖ Create simple example that creates a ***Kafka Consumer***
  - ❖ that consumes messages from the ***Kafka Producer*** we wrote
- ❖ ***Create Consumer*** that uses topic from first example to receive messages
- ❖ ***Process messages*** from Kafka with ***Consumer***
- ❖ Demonstrate how Consumer Groups work

# Create Consumer using Topic to Receive Records



- ❖ Specify bootstrap servers
- ❖ Specify Consumer Group
- ❖ Specify Record Key deserializer
- ❖ Specify Record Value deserializer
- ❖ Subscribe to Topic from last session

# Common Kafka imports and constants



```
KafkaConsumerExample.java x
KafkaConsumerExample
1 package com.cloudurable.kafka;
2 import org.apache.kafka.clients.consumer.*;
3 import org.apache.kafka.clients.consumer.Consumer;
4 import org.apache.kafka.common.serialization.LongDeserializer;
5 import org.apache.kafka.common.serialization.StringDeserializer;
6
7 import java.util.Collections;
8 import java.util.Properties;
9
10 public class KafkaConsumerExample {
11
12     private final static String TOPIC = "my-example-topic";
13     private final static String BOOTSTRAP_SERVERS =
14         "localhost:9092,localhost:9093,localhost:9094";
15 }
```

# Create Consumer using Topic to Receive Records



```
KafkaConsumerExample.java x
KafkaConsumerExample createConsumer()
16     private static Consumer<Long, String> createConsumer() {
17         final Properties props = new Properties();
18         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
19                   BOOTSTRAP_SERVERS);
20         props.put(ConsumerConfig.GROUP_ID_CONFIG,
21                   "KafkaExampleConsumer");
22         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
23                   LongDeserializer.class.getName());
24         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
25                   StringDeserializer.class.getName());
26
27         // Create the consumer using props.
28         final Consumer<Long, String> consumer =
29             new KafkaConsumer<>(props);
30
31         // Subscribe to the topic.
32         consumer.subscribe(Collections.singletonList(TOPIC));
33         return consumer;
34 }
```

# Process messages from Kafka with Consumer



KafkaConsumerExample.java x

KafkaConsumerExample

```
40     static void runConsumer() throws InterruptedException {
41         final Consumer<Long, String> consumer = createConsumer();
42
43         final int giveUp = 100;    int noRecordsCount = 0;
44
45         while (true) {
46             final ConsumerRecords<Long, String> consumerRecords =
47                 consumer.poll( timeout: 1000 );
48
49             if (consumerRecords.count()==0) {
50                 noRecordsCount++;
51                 if (noRecordsCount > giveUp) break;
52                 else continue;
53             }
54
55             consumerRecords.forEach(record -> {
56                 System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
57                     record.key(), record.value(),
58                     record.partition(), record.offset());
59             });
60
61             consumer.commitAsync();
62         }
63         consumer.close();
64         System.out.println("DONE");
```

# Consumer poll



- ❖ poll() method returns fetched records based on current partition offset
- ❖ Blocking method waiting for specified time if no records available
- ❖ When/If records available, method returns straight away
- ❖ Control the maximum records returned by the poll() with  
props.put(ConsumerConfig.**MAX\_POLL\_RECORDS\_CONFIG**, 100);
- ❖ poll() is not meant to be called from multiple threads

# Running both Consumer then Producer



```
67  
68 >   public static void main(String... args) throws Exception {  
69     runConsumer();  
70   }  
71 }
```

Run KafkaConsumerExample

ssl.protocol = TLS  
ssl.provider = null  
ssl.secure.random.implementation = null  
ssl.trustmanager.algorithm = PKIX  
ssl.truststore.location = null  
ssl.truststore.password = null  
ssl.truststore.type = JKS  
value.deserializer = class org.apache.kafka.common.serialization.StringDeserializer

15:17:35.267 [main] INFO o.a.kafka.common.utils.AppInfoParser - Kafka version : 0.10.2.0  
15:17:35.267 [main] INFO o.a.kafka.common.utils.AppInfoParser - Kafka commitId : 576d93a8dc0cf421  
15:17:35.384 [main] INFO o.a.k.c.c.i.AbstractCoordinator - Discovered coordinator 10.0.0.115:9093  
15:17:35.391 [main] INFO o.a.k.c.c.i.ConsumerCoordinator - Revoking previously assigned partitions  
15:17:35.391 [main] INFO o.a.k.c.c.i.AbstractCoordinator - (Re-)joining group KafkaExampleConsumer  
15:17:42.257 [main] INFO o.a.k.c.c.i.AbstractCoordinator - Successfully joined group KafkaExampleC  
15:17:42.259 [main] INFO o.a.k.c.c.i.ConsumerCoordinator - Setting newly assigned partitions [my-e  
Consumer Record:(1494973064716, Hello Mom 1494973064716, 6, 4)  
Consumer Record:(1494973064719, Hello Mom 1494973064719, 10, 6)  
Consumer Record:(1494973064718, Hello Mom 1494973064718, 9, 9)  
Consumer Record:(1494973064717, Hello Mom 1494973064717, 12, 9)  
Consumer Record:(1494973064720, Hello Mom 1494973064720, 4, 8)



# Logging

```
logback.xml x

1 <configuration>
2   <appender name="STDOUT"
3     class="ch.qos.logback.core.ConsoleAppender">
4     <encoder>
5       <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
6         %logger{36} - %msg%n</pattern>
7     </encoder>
8   </appender>
9
10  <logger name="org.apache.kafka" level="INFO"/>
11  <logger name="org.apache.kafka.common.metrics" level="INFO"/>
12
13  <root level="debug">
14    <appender-ref ref="STDOUT" />
15  </root>
16</configuration>
```

- ❖ Kafka uses sl4j
- ❖ Set level to DEBUG to see what is going on

# Try this: Consumers in Same Group



- ❖ Three consumers and one producer sending 25 records
- ❖ Run three consumers processes
- ❖ Change Producer to send 25 records instead of 5
- ❖ Run one producer
- ❖ What happens?

# Outcome 3 Consumers Load Share



Consumer 0 (key, value, partition, offset)

```
Consumer Record: (1495042369488, Hello Mom 1495042369488, 0, 9)
Consumer Record: (1495042369490, Hello Mom 1495042369490, 3, 9)
Consumer Record: (1495042369498, Hello Mom 1495042369498, 3, 10)
Consumer Record: (1495042369504, Hello Mom 1495042369504, 3, 11)
Consumer Record: (1495042369508, Hello Mom 1495042369508, 3, 12)
Consumer Record: (1495042369491, Hello Mom 1495042369491, 4, 9)
Consumer Record: (1495042369503, Hello Mom 1495042369503, 4, 10)
Consumer Record: (1495042369505, Hello Mom 1495042369505, 4, 11)
Consumer Record: (1495042369494, Hello Mom 1495042369494, 2, 9)
Consumer Record: (1495042369499, Hello Mom 1495042369499, 2, 10)
```

Consumer 1 (key, value, partition, offset)

```
key=1495042369509 value=Hello Mom 1495042369509) meta(partition=6, offset=6) t
key=1495042369487 value=Hello Mom 1495042369487) meta(partition=9, offset=12)
key=1495042369486 value=Hello Mom 1495042369486) meta(partition=12, offset=10)
key=1495042369493 value=Hello Mom 1495042369493) meta(partition=12, offset=11)
key=1495042369507 value=Hello Mom 1495042369507) meta(partition=12, offset=12)
key=1495042369488 value=Hello Mom 1495042369488) meta(partition=0, offset=9)
key=1495042369490 value=Hello Mom 1495042369490) meta(partition=3, offset=9)
key=1495042369498 value=Hello Mom 1495042369498) meta(partition=3, offset=10)
key=1495042369504 value=Hello Mom 1495042369504) meta(partition=3, offset=11)
key=1495042369508 value=Hello Mom 1495042369508) meta(partition=3, offset=12)
key=1495042369497 value=Hello Mom 1495042369497) meta(partition=7, offset=11)
key=1495042369500 value=Hello Mom 1495042369500) meta(partition=7, offset=12)
sent record(key=1495042369492 value=Hello Mom 1495042369492) meta(partition=10, offset=7)
sent record(key=1495042369495 value=Hello Mom 1495042369495) meta(partition=10, offset=8)
sent record(key=1495042369491 value=Hello Mom 1495042369491) meta(partition=4, offset=9)
sent record(key=1495042369503 value=Hello Mom 1495042369503) meta(partition=4, offset=10)
key=1495042369505 value=Hello Mom 1495042369505) meta(partition=4, offset=11)
key=1495042369496 value=Hello Mom 1495042369496) meta(partition=5, offset=7)
key=1495042369510 value=Hello Mom 1495042369510) meta(partition=5, offset=8)
key=1495042369489 value=Hello Mom 1495042369489) meta(partition=8, offset=9)
key=1495042369502 value=Hello Mom 1495042369502) meta(partition=8, offset=10)
key=1495042369501 value=Hello Mom 1495042369501) meta(partition=11, offset=8)
key=1495042369506 value=Hello Mom 1495042369506) meta(partition=11, offset=9)
key=1495042369494 value=Hello Mom 1495042369494) meta(partition=2, offset=9)
key=1495042369499 value=Hello Mom 1495042369499) meta(partition=2, offset=10)
```

Consumer 2 (key, value, partition, offset)

```
Consumer Record: (1495042369509, Hello Mom 1495042369509, 6, 6)
Consumer Record: (1495042369497, Hello Mom 1495042369497, 7, 11)
Consumer Record: (1495042369500, Hello Mom 1495042369500, 7, 12)
Consumer Record: (1495042369496, Hello Mom 1495042369496, 5, 7)
Consumer Record: (1495042369510, Hello Mom 1495042369510, 5, 8)
Consumer Record: (1495042369489, Hello Mom 1495042369489, 8, 9)
Consumer Record: (1495042369502, Hello Mom 1495042369502, 8, 10)
```

Producer

Which consumer owns partition 10?  
How many ConsumerRecords objects did Consumer 0 get?  
What is the next offset from Partition 5 that Consumer 2 should get?  
Why does each consumer get

# Try this: Consumers in Different Groups



- ❖ Three consumers with unique group and one producer sending 5 records
- ❖ Modify Consumer to have unique group id
- ❖ Run three consumers processes
- ❖ Run one producer
- ❖ What happens?

# Pass Unique Group Id



```
KafkaConsumerExample.java x
KafkaConsumerExample createConsumer()
16
17  private static Consumer<Long, String> createConsumer() {
18      final Properties props = new Properties();
19
20      props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
21                  BOOTSTRAP_SERVERS);
22
23      props.put(ConsumerConfig.GROUP_ID_CONFIG,
24                  "KafkaExampleConsumer" +
25                  System.currentTimeMillis());
26
27      props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
28                  LongDeserializer.class.getName());
29      props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
30                  StringDeserializer.class.getName());
31
32
33      //Take up to 100 records at a time
34      props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);
```



# Outcome 3 Subscribers

Consumer 0 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Producer

```
sent (key=1495043832401 ) meta(partition=7, offset=14)
sent (key=1495043832400 ) meta(partition=1, offset=11)
sent (key=1495043832404 ) meta(partition=6, offset=7)
sent (key=1495043832402 ) meta(partition=0, offset=11)
sent (key=1495043832403 ) meta(partition=3, offset=13)
```

Consumer 1 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Which consumer(s) owns partition 10?

How many ConsumerRecords objects did Consumer 0 get?

What is the next offset from Partition 2 that Consumer 2 should get?

Consumer 2 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Why does each consumer get the same messages?

# Try this: Consumers in Different Groups



- ❖ Modify consumer: change group id back to non-unique value
- ❖ Make the batch size 5
- ❖ Add a 100 ms delay in the consumer after each message poll and print out record count and partition count
- ❖ Modify the Producer to run 10 times with a 30 second delay after each run and to send 50 messages each run
- ❖ Run producer



# Modify Consumer

```
KafkaConsumerExample.java x
KafkaConsumerExample
8 import java.util.Properties;
9
10 public class KafkaConsumerExample {
11
12     private final static String TOPIC = "my-example-topic";
13     private final static String BOOTSTRAP_SERVERS =
14         "localhost:9092,localhost:9093,localhost:9094";
15
16
17     private static Consumer<Long, String> createConsumer() {
18         final Properties props = new Properties();
19
20         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
21                  BOOTSTRAP_SERVERS);
22
23         props.put(ConsumerConfig.GROUP_ID_CONFIG,
24                  "KafkaExampleConsumer");
25
26         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
27                  LongDeserializer.class.getName());
28         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
29                  StringDeserializer.class.getName());
30
31         props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 5);
```

- ❖ Change group name to common name
- ❖ Change batch size to 5

# Add a 100 ms delay to Consumer after poll



```
47     try {
48         final int giveUp = 1000; int noRecordsCount = 0;
49
50         while (true) {
51             final ConsumerRecords<Long, String> consumerRecords =
52                 consumer.poll( timeout: 1000);
53
54             if (consumerRecords.count() == 0) {
55                 noRecordsCount++;
56                 if (noRecordsCount > giveUp) break;
57                 else continue;
58             }
59
60             System.out.printf("New ConsumerRecords par count %d count %d\n",
61                             consumerRecords.partitions().size(),
62                             consumerRecords.count());
63
64             consumerRecords.forEach(record -> {
65                 System.out.printf("Consumer Record: (%d, %s, %d, %d)\n",
66                                 record.key(), record.value(),
67                                 record.partition(), record.offset());
68             });
69             Thread.sleep( millis: 100);
70             consumer.commitAsync();
71         }
72     }
73     finally {
74         consumer.close();
75     }
```

# Modify Producer: Run 10 times, add 30 second delay



```
KafkaProducerExample.java x
KafkaProducerExample main()
104
105 >   public static void main(String... args)
106       throws Exception {
107           for (int index = 0; index < 10; index++) {
108               runProducer( sendMessageCount: 50 );
109               Thread.sleep( millis: 30_000 );
110           }
111       }
112   }
113 }
```

- ❖ Run 10 times
- ❖ Add 30 second delay
- ❖ Send 50 records

# Notice one or more partitions per ConsumerRecords



```
KafkaConsumerExample KafkaConsumerExample KafkaConsumerExample
New ConsumerRecords par count 1 count 4
Consumer Record:(1495055263352, Hello Mom 1495055263352, 2, 189)
Consumer Record:(1495055263355, Hello Mom 1495055263355, 2, 190)
Consumer Record:(1495055263365, Hello Mom 1495055263365, 2, 191)
Consumer Record:(1495055263368, Hello Mom 1495055263368, 2, 192)
New ConsumerRecords par count 2 count 4
Consumer Record:(1495055263340, Hello Mom 1495055263340, 0, 175)
Consumer Record:(1495055263362, Hello Mom 1495055263362, 0, 176)
Consumer Record:(1495055263335, Hello Mom 1495055263335, 4, 176)
Consumer Record:(1495055263358, Hello Mom 1495055263358, 4, 177)
New ConsumerRecords par count 2 count 5
Consumer Record:(1495055263338, Hello Mom 1495055263338, 1, 219)
Consumer Record:(1495055263341, Hello Mom 1495055263341, 1, 220)
Consumer Record:(1495055263339, Hello Mom 1495055263339, 3, 185)
Consumer Record:(1495055263354, Hello Mom 1495055263354, 3, 186)
Consumer Record:(1495055263371, Hello Mom 1495055263371, 3, 187)
New ConsumerRecords par count 1 count 5
Consumer Record:(1495055263351, Hello Mom 1495055263351, 1, 221)
Consumer Record:(1495055263353, Hello Mom 1495055263353, 1, 222)
Consumer Record:(1495055263356, Hello Mom 1495055263356, 1, 223)
Consumer Record:(1495055263366, Hello Mom 1495055263366, 1, 224)
Consumer Record:(1495055263367, Hello Mom 1495055263367, 1, 225)
```



# Now run it again but..

- ❖ Run the consumers and producer again
- ❖ Wait 30 seconds
- ❖ While the producer is running kill one of the consumers and see the records go to the other consumers
- ❖ Now leave just one consumer running, all of the messages should go to the remaining consumer
  - ❖ Now change consumer batch size to 500

```
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG,  
500)
```
  - ❖ and run it again

# Output from batch size 500



```
New ConsumerRecords par count 7 count 28
Consumer Record:(1495056566578, Hello Mom 1495056566578, 5, 266)
Consumer Record:(1495056566591, Hello Mom 1495056566591, 5, 267)
Consumer Record:(1495056566603, Hello Mom 1495056566603, 5, 268)
Consumer Record:(1495056566605, Hello Mom 1495056566605, 5, 269)
Consumer Record:(1495056566581, Hello Mom 1495056566581, 8, 238)
Consumer Record:(1495056566592, Hello Mom 1495056566592, 8, 239)
Consumer Record:(1495056566597, Hello Mom 1495056566597, 8, 240)
Consumer Record:(1495056566598, Hello Mom 1495056566598, 8, 241)
Consumer Record:(1495056566607, Hello Mom 1495056566607, 8, 242)
Consumer Record:(1495056566609, Hello Mom 1495056566609, 8, 243)
Consumer Record:(1495056566625, Hello Mom 1495056566625, 8, 244)
Consumer Record:(1495056566626, Hello Mom 1495056566626, 8, 245)
Consumer Record:(1495056566584, Hello Mom 1495056566584, 10, 253)
Consumer Record:(1495056566585, Hello Mom 1495056566585, 10, 254)
Consumer Record:(1495056566594, Hello Mom 1495056566594, 10, 255)
Consumer Record:(1495056566601, Hello Mom 1495056566601, 10, 256)
Consumer Record:(1495056566618, Hello Mom 1495056566618, 10, 257)
Consumer Record:(1495056566619, Hello Mom 1495056566619, 10, 258)
Consumer Record:(1495056566593, Hello Mom 1495056566593, 11, 230)
Consumer Record:(1495056566600, Hello Mom 1495056566600, 11, 231)
Consumer Record:(1495056566586, Hello Mom 1495056566586, 2, 265)
Consumer Record:(1495056566596, Hello Mom 1495056566596, 1, 296)
Consumer Record:(1495056566624, Hello Mom 1495056566624, 1, 297)
Consumer Record:(1495056566595, Hello Mom 1495056566595, 4, 242)
Consumer Record:(1495056566604, Hello Mom 1495056566604, 4, 243)
Consumer Record:(1495056566610, Hello Mom 1495056566610, 4, 244)
Consumer Record:(1495056566622, Hello Mom 1495056566622, 4, 245)
Consumer Record:(1495056566623, Hello Mom 1495056566623, 4, 246)
New ConsumerRecords par count 5 count 22
Consumer Record:(1495056566599, Hello Mom 1495056566599, 6, 236)
Consumer Record:(1495056566613, Hello Mom 1495056566613, 6, 237)
Consumer Record:(1495056566620, Hello Mom 1495056566620, 6, 238)
Consumer Record:(1495056566579, Hello Mom 1495056566579, 9, 267)
Consumer Record:(1495056566583, Hello Mom 1495056566583, 9, 268)
```

# Java Kafka Simple Consumer Example Recap



- ❖ Created simple example that creates a ***Kafka Consumer*** to consume messages from our ***Kafka Producer***
- ❖ Used the replicated ***Kafka topic*** from first example
- ❖ ***Created Consumer*** that uses topic to receive messages
- ❖ ***Processed records*** from Kafka with ***Consumer***
- ❖ ***Consumers*** in same group divide up and share partitions
- ❖ ***Each Consumer groups gets a copy of the same data (really has a unique set of offset partition pairs per Consumer Group)***

# ? Kafka Consumer Review



- ❖ How did we demonstrate Consumers in a Consumer Group dividing up topic partitions and sharing them?
- ❖ How did we demonstrate Consumers in different Consumer Groups each getting their own offsets?
- ❖ How many records does poll get?
- ❖ Does a call to poll ever get records from two different partitions?

# Complete Labs 3

# Advanced Consumers

# Objectives Advanced Kafka Producers



- ❖ Using auto commit / Turning auto commit off
- ❖ Managing a custom partition and offsets
  - ❖ ***ConsumerRebalanceListener***
- ❖ Manual Partition Assignment (**assign()** vs. **subscribe()**)
- ❖ Consumer Groups, aliveness
  - ❖ **poll()** and **session.timeout.ms**
- ❖ Consumers and message delivery semantics
  - ❖ at most once, at least once, exactly once
- ❖ Consumer threading models
  - ❖ thread per consumer, consumer with many threads (both)

# Java Consumer Examples Overview



- ❖ Rewind Partition using ***ConsumerRebalanceListener*** and ***consumer.seekX()*** full Java example
- ❖ At-Least-Once Delivery Semantics full Java Example
- ❖ At-Most-Once Delivery Semantics full Java Example
- ❖ Exactly-Once Delivery Semantics full Java Example
  - ❖ full example storing topic, partition, offset in RDBMS with JDBC
  - ❖ Uses ***ConsumerRebalanceListener*** to restore to correct location in log partitions based off of database records
  - ❖ Uses transactions, rollbacks if commitSync fails
- ❖ Threading model Java Examples
  - ❖ Thread per Consumer
  - ❖ Consumer with multiple threads (***TopicPartition*** management of ***commitSync()***)
- ❖ Implement full “priority queue” behavior using Partitioner and manual partition assignment
  - ❖ Manual Partition Assignment (***assign()*** vs. ***subscribe()***)



# KafkaConsumer

- ❖ A Consumer client
  - ❖ consumes records from Kafka cluster
- ❖ Automatically handles Kafka broker failure
  - ❖ adapts as topic partitions leadership moves in Kafka cluster
- ❖ Works with Kafka broker to form consumers groups and load balance consumers
- ❖ Consumer maintains connections to Kafka brokers in cluster
- ❖ Use ***close()*** method to not leak resources
- ❖ NOT thread-safe

# StockPrice App to demo Advanced Producer



- ❖ ***StockPrice*** - holds a stock price has a name, dollar, and cents
- ❖ ***StockPriceConsumer*** - Kafka ***Consumer*** that consumes ***StockPrices***
- ❖ ***StockAppConstants*** - holds topic and broker list
- ❖ ***StockPriceDeserializer*** - can deserialize a ***StockPrice*** from ***byte[]***

# Kafka Consumer Example



- ❖ ***StockPriceConsumer*** to consume StockPrices and display batch lengths for ***poll()***
- ❖ Shows using ***poll()***, and basic configuration (review)
- ❖ Also Shows how to use a Kafka Deserializer
- ❖ How to configure the ***Deserializer*** in Consumer config
- ❖ All the other examples build on this and this builds on Advanced Producer StockPrice example

# Consumer: createConsumer / Consumer Config



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

11
12 > public class SimpleStockPriceConsumer {
13
14     private static Consumer<String, StockPrice> createConsumer() {
15         final Properties props = new Properties();
16         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
17             StockAppConstants.BOOTSTRAP_SERVERS);
18         props.put(ConsumerConfig.GROUP_ID_CONFIG,
19             "KafkaExampleConsumer");
20         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
21             StringDeserializer.class.getName());
22         //Custom Deserializer
23         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
24             StockDeserializer.class.getName());
25         props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
26         // Create the consumer using props.
27         final Consumer<String, StockPrice> consumer =
28             new KafkaConsumer<>(props);
29         // Subscribe to the topic.
30         consumer.subscribe(Collections.singletonList(
31             StockAppConstants.TOPIC));
32         return consumer;
33     }
}
```

- ❖ Similar to other Consumer examples so far
- ❖ Subscribes to **stock-prices** topic
- ❖ Has custom serializer

# SimpleStockPriceConsumer.runConsumer



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

35
36     static void runConsumer() throws InterruptedException {
37         final Consumer<String, StockPrice> consumer = createConsumer();
38         final Map<String, StockPrice> map = new HashMap<>();
39         try {
40             final int giveUp = 1000; int noRecordsCount = 0;
41             int readCount = 0;
42             while (true) {
43                 final ConsumerRecords<String, StockPrice> consumerRecords =
44                     consumer.poll( timeout: 1000 );
45                 if (consumerRecords.count() == 0) {
46                     noRecordsCount++;
47                     if (noRecordsCount > giveUp) break;
48                     else continue;
49                 }
50                 readCount++;
51                 consumerRecords.forEach(record -> {
52                     map.put(record.key(), record.value());
53                 });
54                 if (readCount % 100 == 0) {
55                     displayRecordsStatsAndStocks(map, consumerRecords);
56                 }
57                 consumer.commitAsync();
58             }
59         }
```

- ❖ Drains topic; Creates map of current stocks; Calls ***displayRecordsStatsAndStocks()***



# Using ConsumerRecords : SimpleStockPriceConsumer.displayRecordsStatsAndStocks()

SimpleStockPriceConsumer.java x

```
SimpleStockPriceConsumer displayRecordsStatsAndStocks()
66     private static void displayRecordsStatsAndStocks(
67         final Map<String, StockPrice> stockPriceMap,
68         final ConsumerRecords<String, StockPrice> consumerRecords) {
69     System.out.printf("New ConsumerRecords par count %d count %d\n",
70         consumerRecords.partitions().size(),
71         consumerRecords.count());
72     stockPriceMap.forEach((s, stockPrice) ->
73         System.out.printf("ticker %s price %d.%d \n",
74             stockPrice.getName(),
75             stockPrice.getDollars(),
76             stockPrice.getCents()));
77     System.out.println();
78 }
```

- ❖ Prints out size of each partition read and total record count
- ❖ Prints out each stock at its current price

# Consumer Deserializer: StockDeserializer



```
c SimpleStockPriceConsumer.java x c StockDeserializer.java x
StockDeserializer
3 import ...
8
9 public class StockDeserializer implements Deserializer<StockPrice> {
10
11     @Override
12     public StockPrice deserialize(final String topic, final byte[] data) {
13         return new StockPrice(new String(data, StandardCharsets.UTF_8));
14     }
15
16     @Override
17     public void configure(Map<String, ?> configs, boolean isKey) {
18     }
19
20     @Override
21     public void close()
22     }
23 }
```

```
c SimpleStockPriceConsumer.java x c StockDeserializer.java x c StockPrice.java x
StockPrice
1 package com.cloudurable.kafka.producer.model;
2
3 import io.advantageous.boon.json.JsonFactory;
4
5 public class StockPrice {
6
7     private final int dollars;
8     private final int cents;
9     private final String name;
10
11     public StockPrice(final String json) {
12         this(JsonFactory.fromJson(json, StockPrice.class));
13     }
14 }
```

# Kafka Consumer: Offsets and Consumer Position

- ❖ Consumer position is offset and partition of last record per partition consuming from
  - ❖ offset for each record in a partition as a unique identifier record location in partition
- ❖ Consumer position gives offset of next record that it consume (next highest)
  - ❖ position advances automatically for each call to ***poll(..)***
- ❖ Consumer committed position is last offset that has been stored to broker
  - ❖ If consumer fails, it picks up at last committed position
- ❖ Consumer can auto commit offsets (***enable.auto.commit***) periodically (***auto.commit.interval.ms***) or do commit explicitly using ***commitSync()*** and ***commitAsync()***

# KafkaConsumer: Consumer Groups



- ❖ Consumers organized into consumer groups (Consumer instances with same **group.id**)
  - ❖ Pool of consumers divide work of consuming and processing records
  - ❖ Processes or threads running on same box or distributed for scalability/fault tolerance
- ❖ Kafka shares topic partitions among all consumers in a consumer group
  - ❖ each partition is assigned to exactly one consumer in consumer group
  - ❖ Example topic has six partitions, and a consumer group has two consumer processes, each process gets consume three partitions
- ❖ Failover and Group rebalancing
  - ❖ if a consumer fails, Kafka reassigned partitions from failed consumer to other consumers in same consumer group
  - ❖ if new consumer joins, Kafka moves partitions from existing consumers to new one

# Consumer Groups and Topic Subscriptions



- ❖ Consumer group form ***single logical subscriber*** made up of multiple consumers
- ❖ Kafka is a multi-subscriber system, Kafka supports N number of ***consumer groups*** for a given topic without duplicating data
- ❖ To get something like a MOM queue all consumers would be in single consumer group
  - ❖ Load balancing like a MOM queue
- ❖ Unlike a MOM, you can have multiple such consumer groups, and price of subscribers does not incur duplicate data
- ❖ To get something like MOM pub-sub each process would have its own consumer group

# KafkaConsumer: Partition Reassignment



- ❖ Consumer partition reassignment in a consumer group happens automatically
- ❖ Consumers are notified via ***ConsumerRebalanceListener***
  - ❖ Triggers consumers to finish necessary clean up
- ❖ Consumer can use API to assign specific partitions using ***assign(Collection)***
  - ❖ disables dynamic partition assignment and consumer group coordination
- ❖ Dead client may see CommitFailedException thrown from a call to `commitSync()`
  - ❖ Only active members of consumer group can commit offsets.

# Controlling Consumers Position



- ❖ You can control consumer position
  - ❖ moving consumer forward or backwards - consumer can re-consume older records or skip to most recent records
- ❖ Use ***consumer.seek(TopicPartition, long)*** to specify new position
  - ❖ ***consumer.seekToBeginning(Collection) and seekToEnd(Collection) respectively***
- ❖ Use Case Time-sensitive record processing: Skip to most recent records
- ❖ Use Case Bug Fix: Reset position before bug fix and replay log from there
- ❖ Use Case Restore State for Restart or Recovery: Consumer initialize position on start-up to whatever is contained in local store and replay missed parts (cache warm-up or replacement in case of failure assumes Kafka retains sufficient history or you are using log compaction)

# Storing Offsets Outside: Managing Offsets



- ❖ For the consumer to manage its own offset you just need to do the following:
  - ❖ Set ***enable.auto.commit=false***
  - ❖ Use offset provided with each ConsumerRecord to save your position (partition/offset)
  - ❖ On restart restore consumer position using ***kafkaConsumer.seek(TopicPartition, long)***.
- ❖ Usage like this simplest when the partition assignment is also done manually using ***assign()*** instead of ***subscribe()***

# Storing Offsets Outside: Managing Offsets



- ❖ If using automatic partition assignment, you must handle cases where partition assignments change
  - ❖ Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe(Collection, ConsumerRebalanceListener)*** and ***kafkaConsumer.subscribe(Pattern, ConsumerRebalanceListener)***.
  - ❖ when partitions taken from consumer, commit its offset for partitions by implementing ***ConsumerRebalanceListener.onPartitionsRevoked(Collection)***
  - ❖ When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ***ConsumerRebalanceListener.onPartitionsAssigned(Collection)***

```
// Subscribe to the topic.  
consumer.subscribe(Collections.singletonList(  
    StockAppConstants.TOPIC),  
    new SeekToConsumerRebalanceListener(consumer, seekTo, location));
```



# Controlling Consumers Position Example

```
SeekToConsumerRebalanceListener onPartitionsAssigned()  
o  
9  
10 public class SeekToConsumerRebalanceListener implements ConsumerRebalanceListener {  
11     private final Consumer<String, StockPrice> consumer;  
12     private final SeekTo seekTo; private boolean done;  
13     private final long location;  
14     private final long startTime = System.currentTimeMillis();  
15     public SeekToConsumerRebalanceListener(final Consumer<String, StockPrice> consumer,  
16  
21         @Override  
22     ⚡ public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {  
23         if (done) return;  
24         else if (System.currentTimeMillis() - startTime > 30_000) {  
25             done = true;  
26             return;  
27         }  
28         switch (seekTo) {  
29             case END: //Seek to end  
30                 consumer.seekToEnd(partitions);  
31                 break;  
32             case START: //Seek to start  
33                 consumer.seekToBeginning(partitions);  
34                 break;  
35             case LOCATION: //Seek to a given location  
36                 partitions.forEach(topicPartition ->  
37                     consumer.seek(topicPartition, location));  
38                 break;  
39             }  
40         }
```

# Kafka Consumer: Consumer Alive Detection

- ❖ Consumers join consumer group after subscribe and then **poll()** is called
- ❖ Automatically, consumer sends periodic heartbeats to Kafka brokers server
- ❖ If consumer crashes or unable to send heartbeats for a duration of **session.timeout.ms**, then consumer is deemed dead and its partitions are reassigned



- ❖ Instead of subscribing to the topic using `subscribe`, you can call **`assign(Collection)`** with the full topic partition list

```
String topic = "log-replication";  
  
TopicPartition part0 = new TopicPartition(topic, 0);  
  
TopicPartition part1 = new TopicPartition(topic, 1);  
  
consumer.assign(Arrays.asList(part0, part1));
```

- ❖ Using consumer as before with **`poll()`**
- ❖ Manual partition assignment negates use of group coordination, and auto consumer fail over - Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion)
- ❖ You have to use **`assign()`** or **`subscribe()`** but not both

# KafkaConsumer: Consumer Alive if Polling



- ❖ Calling ***poll()*** marks consumer as alive
  - ❖ If consumer continues to call ***poll()***, then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every ***max.poll.interval.ms interval***)
  - ❖ Not calling ***poll()***, even if consumer is sending heartbeats, consumer is still considered dead
- ❖ Processing of records from ***poll*** has to be faster than ***max.poll.interval.ms*** interval or your consumer could be marked dead!
- ❖ ***max.poll.records*** is used to limit total records returned from a poll call - easier to predict max time to process records on each poll interval

# Message Delivery Semantics

---



- ❖ ***At most once***
  - ❖ Messages may be lost but are never redelivered
- ❖ ***At least once***
  - ❖ Messages are never lost but may be redelivered
- ❖ ***Exactly once***
  - ❖ this is what people actually want, each message is delivered once and only once

# “At-Least-Once” - Delivery Semantics



```
SimpleStockPriceConsumer pollRecordsAndProcess()
```

```
76  
77     final ConsumerRecords<String, StockPrice> consumerRecords =  
78         consumer.poll( timeout: 1000 );  
79  
80     try {  
81         startTransaction();           //Start DB Transaction  
82  
83             //Process the records  
84         processRecords(map, consumerRecords);  
85  
86             //Commit the Kafka offset  
87         consumer.commitSync();  
88  
89         commitTransaction();        //Commit DB Transaction  
90     } catch( CommitFailedException ex ) {  
91         logger.error("Failed to commit sync to log", ex);  
92         rollbackTransaction();       //Rollback Transaction  
93     } catch ( DatabaseException dte ) {  
94         logger.error("Failed to write to DB", dte);  
95         rollbackTransaction();       //Rollback Transaction  
96     }
```

# “At-Most-Once” - Delivery Semantics



```
SimpleStockPriceConsumer pollRecordsAndProcess()  
 76  
 77    final ConsumerRecords<String, StockPrice> consumerRecords =  
 78        consumer.poll( timeout: 1000 );  
 79  
 80    try {  
 81        startTransaction();                      //Start DB Transaction  
 82  
 83        consumer.commitSync();                  //Commit the Kafka offset  
 84  
 85        processRecords(map, consumerRecords);   //Process the records  
 86  
 87    } catch( CommitFailedException ex ) {  
 88        logger.error("Failed to commit sync to log", ex);  
 89        commitTransaction();                     //Commit DB Transaction  
 90    } catch( DatabaseException dte ) {  
 91        logger.error("Failed to write to DB", dte);  
 92        rollbackTransaction();                  //Rollback Transaction  
 93    }  
 94 }
```



# Fine Grained “At-Most-Once”

```
80  ↗    consumerRecords.forEach(record -> {
81      ⚡    try {
82
83          startTransaction();           //Start DB Transaction
84
85          processRecord(record);
86
87          // Commit Kafka at exact location for record, and only this record.
88          final TopicPartition recordTopicPartition =
89              new TopicPartition(record.topic(), record.partition());
90
91          final Map<TopicPartition, OffsetAndMetadata> commitMap =
92              Collections.singletonMap(recordTopicPartition,
93                  new OffsetAndMetadata( offset: record.offset() + 1));
94
95          consumer.commitSync(commitMap);
96
97          commitTransaction();           //Commit DB Transaction
98      } catch (CommitFailedException ex) {
99          logger.error("Failed to commit sync to log", ex);
100         rollbackTransaction();        //Rollback Transaction
101     } catch (DatabaseException dte) {
102         logger.error("Failed to write to DB", dte);
103         rollbackTransaction();        //Rollback Transaction
104     }
105 });
});
```



# Fine Grained “At-Least-Once”

```
SimpleStockPriceConsumer pollRecordsAndProcess()  
78  
79  
80  ↗ consumerRecords.forEach(record -> {  
81      try {  
82          startTransaction();           //Start DB Transaction  
83  
84          // Commit Kafka at exact location for record, and only this record.  
85          final TopicPartition recordTopicPartition =  
86              new TopicPartition(record.topic(), record.partition());  
87  
88          final Map<TopicPartition, OffsetAndMetadata> commitMap =  
89              Collections.singletonMap(recordTopicPartition,  
90                  new OffsetAndMetadata( offset: record.offset() + 1));  
91  
92          consumer.commitSync(commitMap); //Kafka Commit  
93  
94          processRecord(record);       //Process the record  
95  
96          commitTransaction();       //Commit DB Transaction  
97      } catch (CommitFailedException ex) {  
98          logger.error("Failed to commit sync to log", ex);  
99          rollbackTransaction();       //Rollback Transaction  
100     } catch (DatabaseException dte) {  
101         logger.error("Failed to write to DB", dte);  
102         rollbackTransaction();       //Rollback Transaction  
103     }  
104});
```

# Consumer: Exactly Once, Saving Offset



- ❖ Consumer do not have to use Kafka's built-in offset storage
- ❖ Consumers can choose to store offsets with processed record output to make it “exactly once” message consumption
- ❖ If Consumer output of record consumption is stored in RDBMS then storing offset in database allows committing both process record output and location (partition/offset of record) in a single transaction implementing “exactly once” messaging.
- ❖ Typically to achieve “exactly once” you store record location with output of record

# Saving Topic, Offset, Partition in DB



```
DatabaseUtilities saveStockPrice()  
22  
23     public static void saveStockPrice(final StockPriceRecord stockRecord,  
24                                     final Connection connection) throws SQLException {  
25  
26         final PreparedStatement preparedStatement = getUpsertPreparedStatement(  
27             stockRecord.getName(), connection);  
28  
29  
30         //Save partition, offset and topic in database.  
31         preparedStatement.setLong( parameterIndex: 1, stockRecord.getOffset());  
32         preparedStatement.setLong( parameterIndex: 2, stockRecord.getPartition());  
33         preparedStatement.setString( parameterIndex: 3, stockRecord.getTopic());  
34  
35         //Save stock price, name, dollars, and cents into database.  
36         preparedStatement.setInt( parameterIndex: 4, stockRecord.getDollars());  
37         preparedStatement.setInt( parameterIndex: 5, stockRecord.getCents());  
38         preparedStatement.setString( parameterIndex: 6, stockRecord.getName());  
39  
40         //Save the record with offset, partition, and topic.  
41         preparedStatement.execute();  
42  
43     }  
44 }
```

to exactly once, you need to save the offset and partition with the output of the consumer process.

# “Exactly-Once” - Delivery Semantics



SimpleStockPriceConsumer pollRecordsAndProcess()

```
92
93     //Get rid of duplicates and keep only the latest record.
94     consumerRecords.forEach(record -> currentStocks.put(record.key(),
95                               new StockPriceRecord(record.value(), saved: false, record)));
96
97     final Connection connection = getConnection();
98     try {
99         startJdbcTransaction(connection);                      //Start DB Transaction
100        for (StockPriceRecord stockRecordPair : currentStocks.values()) {
101            if (!stockRecordPair.isSaved()) {                     //Save the record
102                saveStockPrice(stockRecordPair, connection);      // with partition/offset to DB.
103                //Mark the record as saved
104                currentStocks.put(stockRecordPair.getName(), new
105                                StockPriceRecord(stockRecordPair, saved: true));
106            }
107        }
108        consumer.commitSync();                                //Commit the Kafka offset
109        connection.commit();                                //Commit DB Transaction
110    } catch (CommitFailedException ex) {
111        logger.error("Failed to commit sync to log", ex);
112        connection.rollback();                            //Rollback Transaction
113    } catch (SQLException sqle) {
114        logger.error("Failed to write to DB", sqle);
115        connection.rollback();                            //Rollback Transaction
116    } finally {
117        connection.close();
118    }
119}
```

# Move Offsets past saved Records



- ❖ If implementing “**exactly once**” message semantics, then you have to manage offset positioning
  - ❖ Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe(Collection, ConsumerRebalanceListener)*** and ***kafkaConsumer.subscribe(Pattern, ConsumerRebalanceListener)***.
  - ❖ when partitions taken from consumer, commit its offset for partitions by implementing ***ConsumerRebalanceListener.onPartitionsRevoked(Collection)***
  - ❖ When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ***ConsumerRebalanceListener.onPartitionsAssigned(Collection)***

# Exactly Once - Move Offsets past saved Records



```
SeekToLatestRecordsConsumerRebalanceListener onPartitionsAssigned()  
16  
17 public class SeekToLatestRecordsConsumerRebalanceListener  
18     implements ConsumerRebalanceListener {  
19  
20     private final Consumer<String, StockPrice> consumer;  
21     private static final Logger logger = getLogger(SimpleStockPriceConsumer.class);  
22  
23     public SeekToLatestRecordsConsumerRebalanceListener(  
24         final Consumer<String, StockPrice> consumer) {  
25         this.consumer = consumer;  
26     }  
27  
28     @Override  
29     public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {  
30         final Map<TopicPartition, Long> maxOffsets = getMaxOffsetsFromDatabase();  
31         maxOffsets.entrySet().forEach(  
32             entry -> partitions.forEach(topicPartition -> {  
33                 if (entry.getKey().equals(topicPartition)) {  
34                     long maxOffset = entry.getValue();  
35  
36                     // Call to consumer.seek to move to the partition.  
37                     consumer.seek(topicPartition, offset: maxOffset + 1);  
38  
39                     displaySeekInfo(topicPartition, maxOffset);  
40                 }  
41             }));  
42     }  
}
```

# Kafka Consumer: Consumption Flow Control

- ❖ You can control consumption of topics using by using ***consumer.pause(Collection)*** and ***consumer.resume(Collection)***
  - ❖ This pauses or resumes consumption on specified assigned partitions for future ***consumer.poll(long)*** calls
- ❖ Use cases where consumers may want to first focus on fetching from some subset of assigned partitions at full speed, and only start fetching other partitions when these partitions have few or no data to consume
  - ❖ Priority queue like behavior from traditional MOM
  - ❖ Other cases is stream processing if preforming a join and one topic stream is getting behind another.

# Kafka Consumer: MultiThreaded Process

- ❖ Kafka consumer is NOT thread-safe
- ❖ All network I/O happens in thread of the application making call
- ❖ Only exception thread safe method is  
***consumer.wakeup()***
  - ❖ **forces** WakeupException to be thrown from thread blocking on operation
  - ❖ Use Case to shutdown consumer from another thread

# Kafka Consumer: One Consumer Per Thread

- ❖ Pro
  - ❖ Easiest to implement
  - ❖ Requires no inter-thread co-ordination
  - ❖ In-order processing on a per-partition basis easy to implement
    - ❖ Process in-order that you receive them
- ❖ Con
  - ❖ More consumers means more TCP connections to the cluster (one per thread) - low cost has Kafka uses async IO and handles connections efficiently

# One Consumer Per Thread: Runnable



## StockPriceConsumerRunnable

```
13 import java.util.Map;
14 import java.util.concurrent.atomic.AtomicBoolean;
15
16 import static com.cloudurable.kafka.StockAppConstants.TOPIC;
17
18 
19 public class StockPriceConsumerRunnable implements Runnable{
20     private static final Logger logger =
21         LoggerFactory.getLogger(StockPriceConsumerRunnable.class);
22
23     private final Consumer<String, StockPrice> consumer;
24     private final int readCountStatusUpdate;
25     private final int threadIndex;
26     private final AtomicBoolean stopAll;
27     private boolean running = true;
28
29     @Override
30     public void run() {
31         try {
32             runConsumer();
33         } catch (Exception ex) {
34             logger.error("Run Consumer Exited with", ex);
35             throw new RuntimeException(ex);
36         }
37     }
38 }
```

# One Consumer Per Thread: runConsumer



c StockPriceConsumerRunnable.java x

StockPriceConsumerRunnable pollRecordsAndProcess()

```
48
49     void runConsumer() throws Exception {
50         // Subscribe to the topic.
51         consumer.subscribe(Collections.singletonList(TOPIC));
52         final Map<String, StockPriceRecord> lastRecordPerStock = new HashMap<>();
53         try {
54             int readCount = 0;
55             while (isRunning()) {
56                 pollRecordsAndProcess(lastRecordPerStock, readCount);
57             }
58         } finally {
59             consumer.close();
60         }
61     }
```

# One Consumer Per Thread: runConsumer



```
StockPriceConsumerRunnable pollRecordsAndProcess()  
64     private void pollRecordsAndProcess(  
65         final Map<String, StockPriceRecord> currentStocks,  
66         final int readCount) throws Exception {  
67  
68     final ConsumerRecords<String, StockPrice> consumerRecords =  
69         consumer.poll( timeout: 100 );  
70  
71     if (consumerRecords.count() == 0) {  
72         if (stopAll.get()) this.setRunning(false);  
73         return;  
74     }  
75  
76     consumerRecords.forEach(record -> currentStocks.put(record.key(),  
77                     new StockPriceRecord(record.value(), saved: true, record)));  
78  
79  
80     try {  
81         startTransaction();                                //Start DB Transaction  
82  
83         processRecords(currentStocks, consumerRecords);  
84         consumer.commitSync();                            //Commit the Kafka offset  
85         commitTransaction();                            //Commit DB Transaction  
86     } catch (CommitFailedException ex) {  
87         logger.error("Failed to commit sync to log", ex);  
88         rollbackTransaction();                         //Rollback Transaction  
89     }
```



# One Consumer Per Thread: Thread Pool

```
ConsumerMain main()
48
49 ► ⚡ public static void main(String... args) throws Exception {
50     final int threadCount = 5;
51     final ExecutorService executorService = newFixedThreadPool(threadCount);
52     final AtomicBoolean stopAll = new AtomicBoolean();
53
54 ⚡ IntStream.range(0, threadCount).forEach(index -> {
55     final StockPriceConsumerRunnable stockPriceConsumer =
56         new StockPriceConsumerRunnable(createConsumer(),
57                                         readCountStatusUpdate: 10, index, stopAll);
58     executorService.submit(stockPriceConsumer);
59});
```

# KafkaConsumer: One Consumer with Worker Threads



- ❖ Decouple Consumption and Processing: One or more consumer threads that consume from Kafka and hands off to ConsumerRecords instances to a blocking queue processed by a processor thread pool that process the records.
- ❖ PROs
  - ❖ This option allows independently scaling consumers count and processors count. Processor threads are independent of topic partition count
- ❖ CONS
  - ❖ Guaranteeing order across processors requires care as threads execute independently a later record could be processed before an earlier record and then you have to do consumer commits somehow
  - ❖ How do you committing the position unless there is some ordering? You have to provide the ordering. (Concurrently HashMap of BlockingQueues where topic, partition is the key (TopicPartition)?)

# One Consumer with Worker Threads



## StockPriceConsumerRunnable

```
18  public class StockPriceConsumerRunnable implements Runnable{  
19      private static final Logger logger =  
20          LoggerFactory.getLogger(StockPriceConsumerRunnable.class);  
21  
22      private final Consumer<String, StockPrice> consumer;  
23      private final int readCountStatusUpdate;  
24      private final int threadIndex;  
25      private final AtomicBoolean stopAll;  
26      private boolean running = true;  
27  
28      //Store blocking queue by TopicPartition.  
29      private final Map<TopicPartition, BlockingQueue<ConsumerRecord>>  
30          commitQueueMap = new ConcurrentHashMap<>();  
31  
32      //Worker pool.  
33      private final ExecutorService threadPool;  
34  
35
```



# Worker

```
StockPriceConsumerRunnable pollRecordsAndProcess()
72     private void pollRecordsAndProcess(
73         final Map<String, StockPriceRecord> currentStocks,
74         final int readCount) throws Exception {
75
76     final ConsumerRecords<String, StockPrice> consumerRecords =
77         consumer.poll( timeout: 100 );
78
79     if (consumerRecords.count() == 0) {
80         if (stopAll.get()) this.setRunning(false);
81         return;
82     }
83
84     consumerRecords.forEach(record ->
85         currentStocks.put(record.key(),
86             new StockPriceRecord(record.value(), saved: true, record)
87         ));
88
89     threadPool.execute(() ->
90         processRecords(currentStocks, consumerRecords));
91
92
```

# processCommits

```
StockPriceConsumerRunnable processCommits()
120     private void processCommits() {
121
122     commitQueueMap.entrySet().forEach(queueEntry -> {
123         final BlockingQueue<ConsumerRecord> queue = queueEntry.getValue();
124         final TopicPartition topicPartition = queueEntry.getKey();
125
126         ConsumerRecord consumerRecord = queue.poll();
127         ConsumerRecord highestOffset = consumerRecord;
128
129         while (consumerRecord != null) {
130             if (consumerRecord.offset() > highestOffset.offset()) {
131                 highestOffset = consumerRecord;
132             }
133             consumerRecord = queue.poll();
134         }
135
136         if (highestOffset != null) {
137             logger.info(String.format("Sending commit %s %d",
138                                     topicPartition, highestOffset.offset()));
139             try {
140                 consumer.commitSync(Collections.singletonMap(topicPartition,
141                                               new OffsetAndMetadata(highestOffset.offset())));
142             } catch (CommitFailedException cfe) {
143                 logger.info("Failed to commit record", cfe);
144             }
145         }
146     }
147 }
```

- ❖ Creating Priority processing queue
- ❖ Use **consumer.partitionsFor(TOPIC)** to get a list of partitions
- ❖ Usage like this simplest when the partition assignment is also done manually using **assign()** instead of **subscribe()**
  - ❖ Use assign(), pass TopicPartition to worker
  - ❖ Use Partitioner from earlier example for Producer

# Using partitionsFor() for Priority Queue



ConsumerMain main()

```
49
50  public static void main(String... args) throws Exception {
51
52      final AtomicBoolean stopAll = new AtomicBoolean();
53      final Consumer<String, StockPrice> consumer = createConsumer();
54
55      //Get the partitions.
56      final List<PartitionInfo> partitionInfos = consumer.partitionsFor(TOPIC);
57
58      final int threadCount = partitionInfos.size();
59      final int numWorkers = 5;
60      final ExecutorService executorService = newFixedThreadPool(threadCount);
61
62
63      IntStream.range(0, threadCount).forEach(index -> {
64          final PartitionInfo partitionInfo = partitionInfos.get(index);
65          final boolean leader = partitionInfo.partition() == partitionInfos.size() -1;
66          final int workerCount = leader ? numWorkers * 3 : numWorkers;
67          final StockPriceConsumerRunnable stockPriceConsumer =
68              new StockPriceConsumerRunnable(partitionInfo, createConsumer(),
69                  readCountStatusUpdate: 10, index, stopAll, numWorkers );
70          executorService.submit(stockPriceConsumer);
71      });
72 }
```

# Using assign() for Priority Queue



StockPriceConsumerRunnable runConsumer()

```
61     void runConsumer() throws Exception {
62         // Assign a partition.
63         consumer.assign(Collections.singleton(topicPartition));
64         final Map<String, StockPriceRecord> lastRecordPerStock = new HashMap<String, StockPriceRecord>();
65         try {
66             int readCount = 0;
67             while (isRunning()) {
68                 pollRecordsAndProcess(lastRecordPerStock, readCount);
69             }
70         } finally {
71             consumer.close();
72         }
73     }
```

# **Complete Labs 6.1-6.7**

# Session 4

# Kafka Streams API

# Developing Kafka Streams

This lesson covers

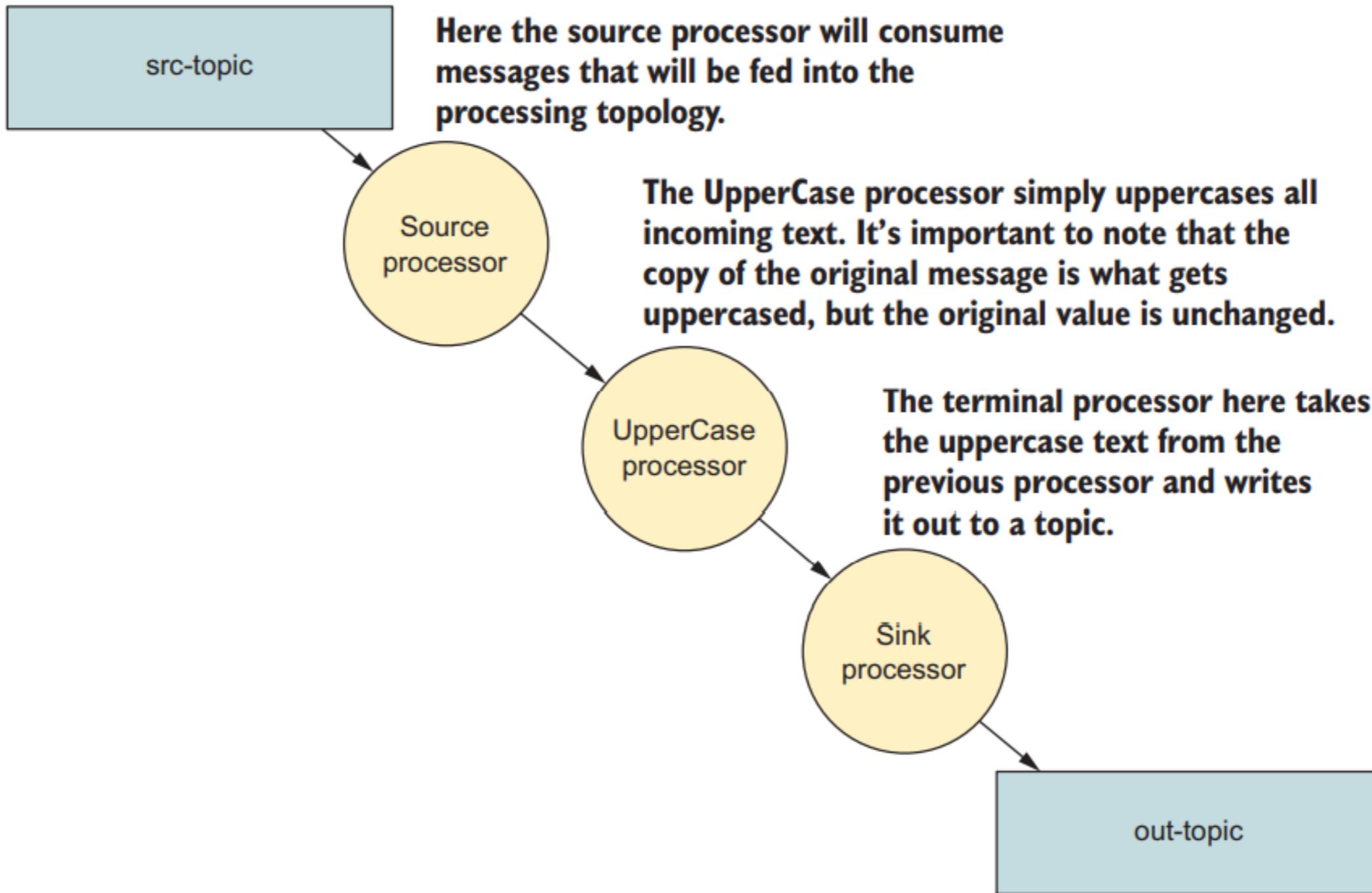
- Introducing the Kafka Streams API
- Building Hello World for Kafka Streams
- Exploring the ZMart Kafka Streams application in depth
- Splitting an incoming stream into multiple streams

# 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

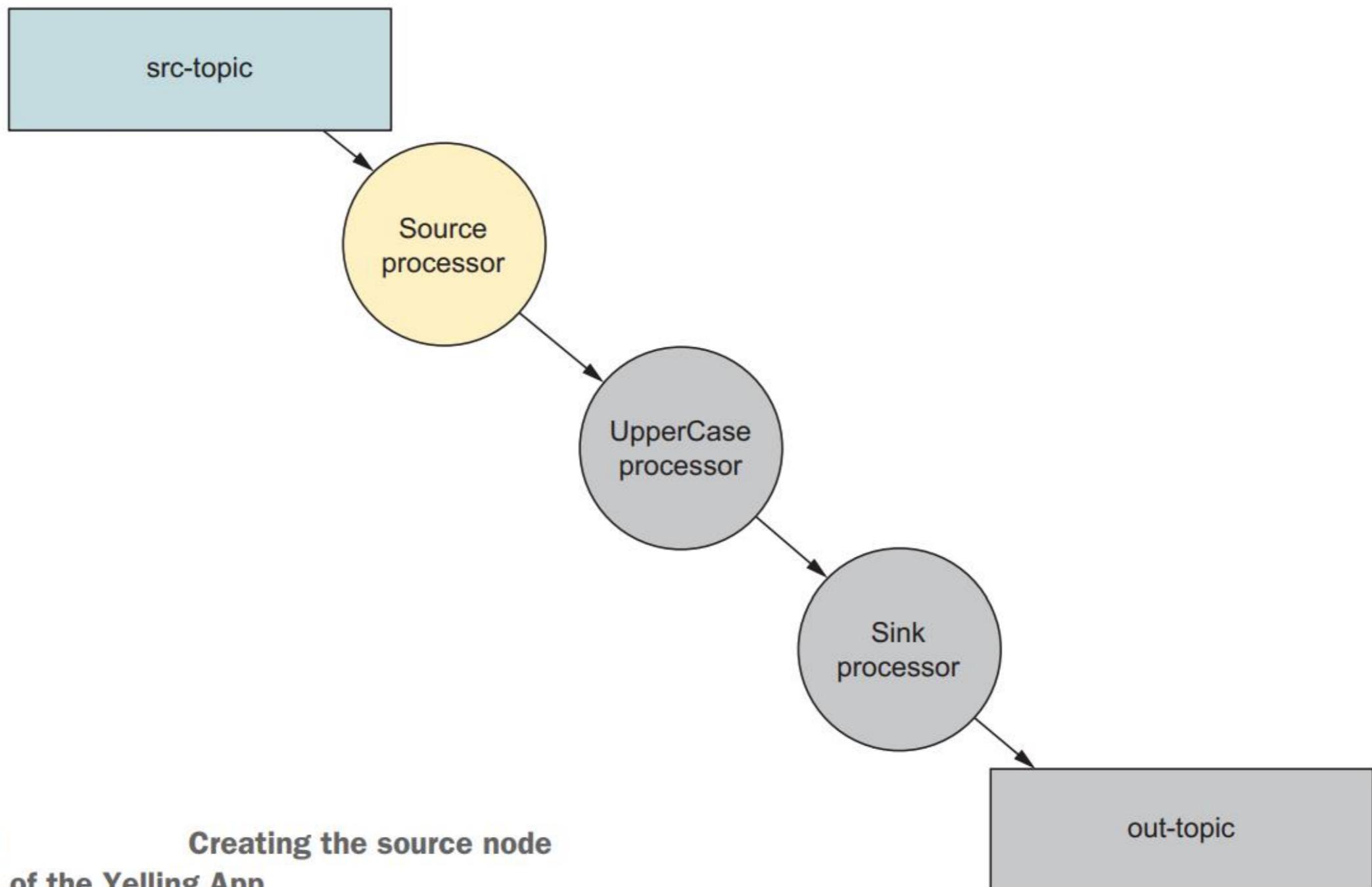


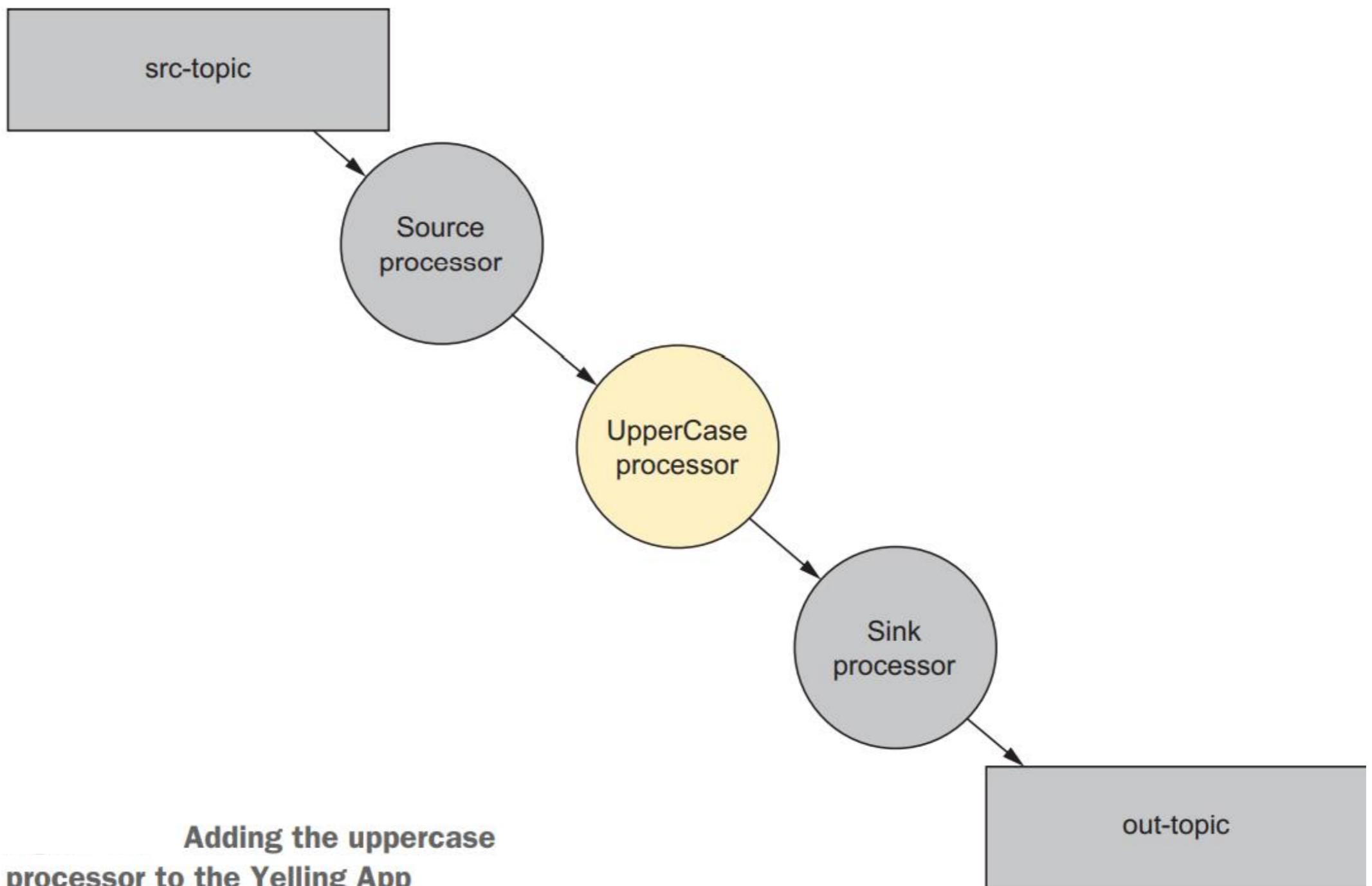
# Creating the topology for the Yelling App

- 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 topology for the Yelling App

## **Listing 3.2 Mapping incoming text to uppercase**

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

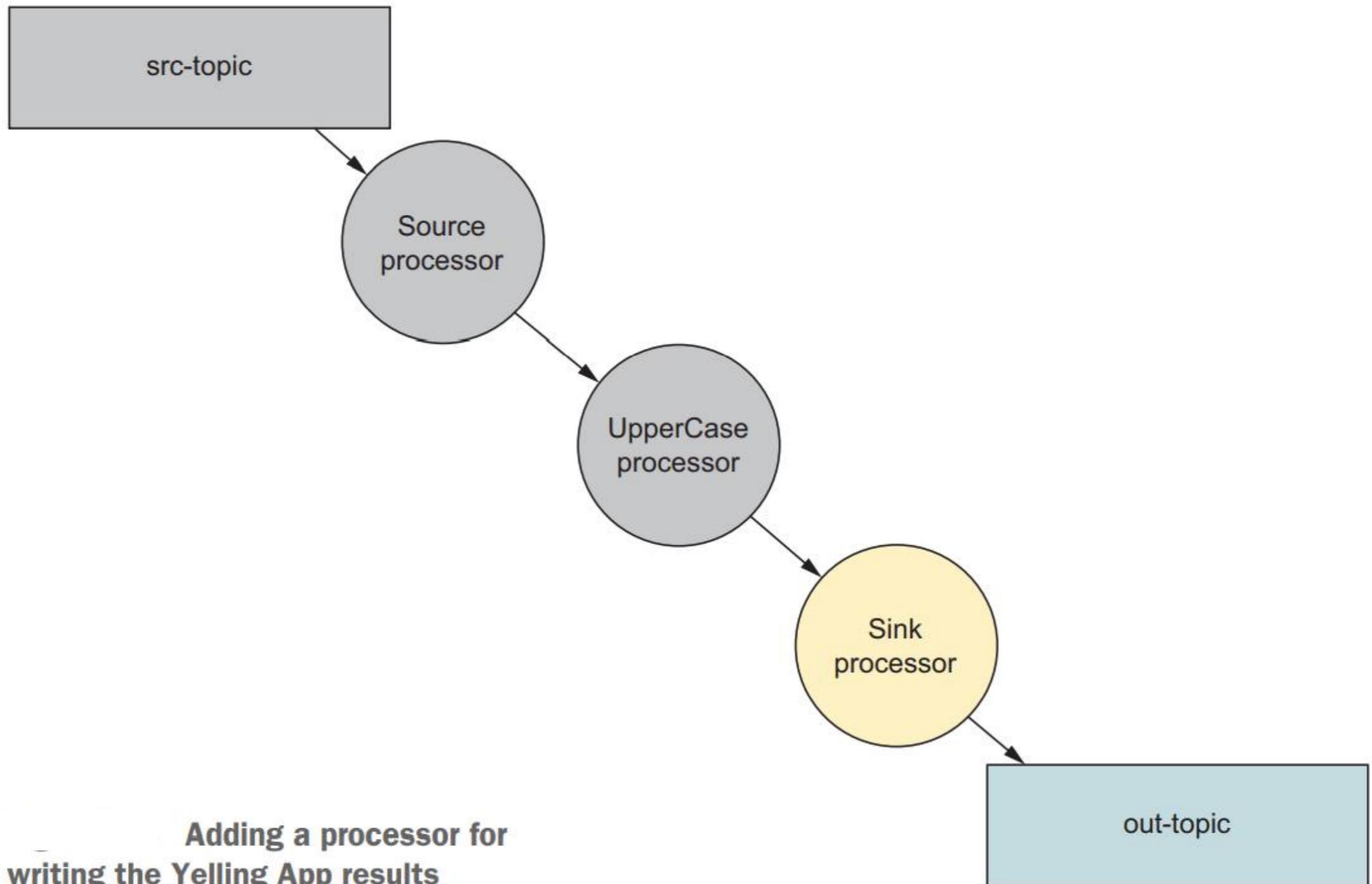
By calling the KStream.mapValues function, you're creating a new processing node whose inputs are the results of going through the mapValues call

# 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));
```



# 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); ←  
            A processor using a Java 8  
            method handle (the first  
            child node in the graph)  
        upperCasedStream.to("out-topic",  
            Produced.with(stringSerde, stringSerde)); ←  
            Writes the transformed  
            output to another topic  
            (the sink node in the graph)  
  
        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)**

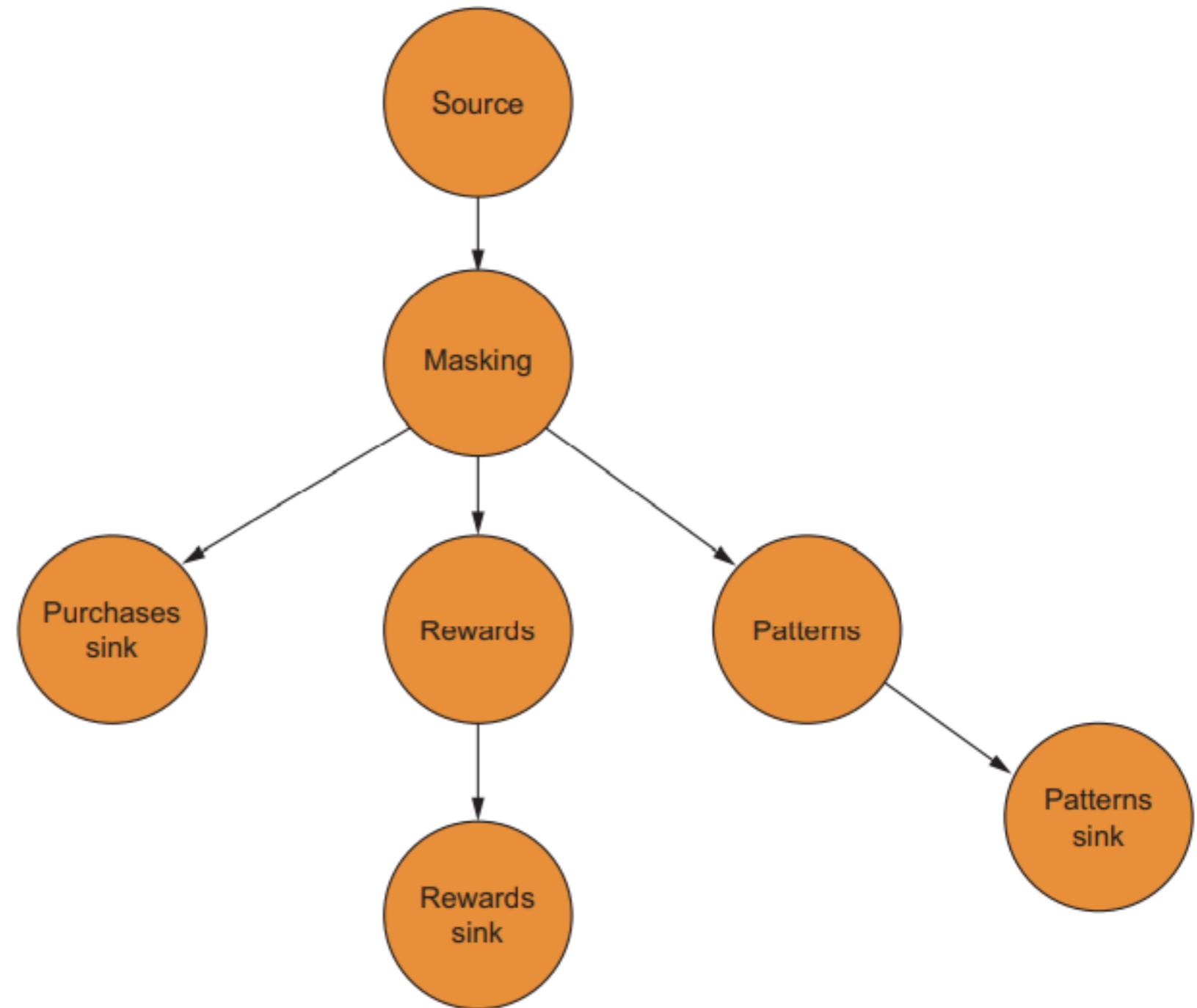
**Kicks off the Kafka Streams threads**

# 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



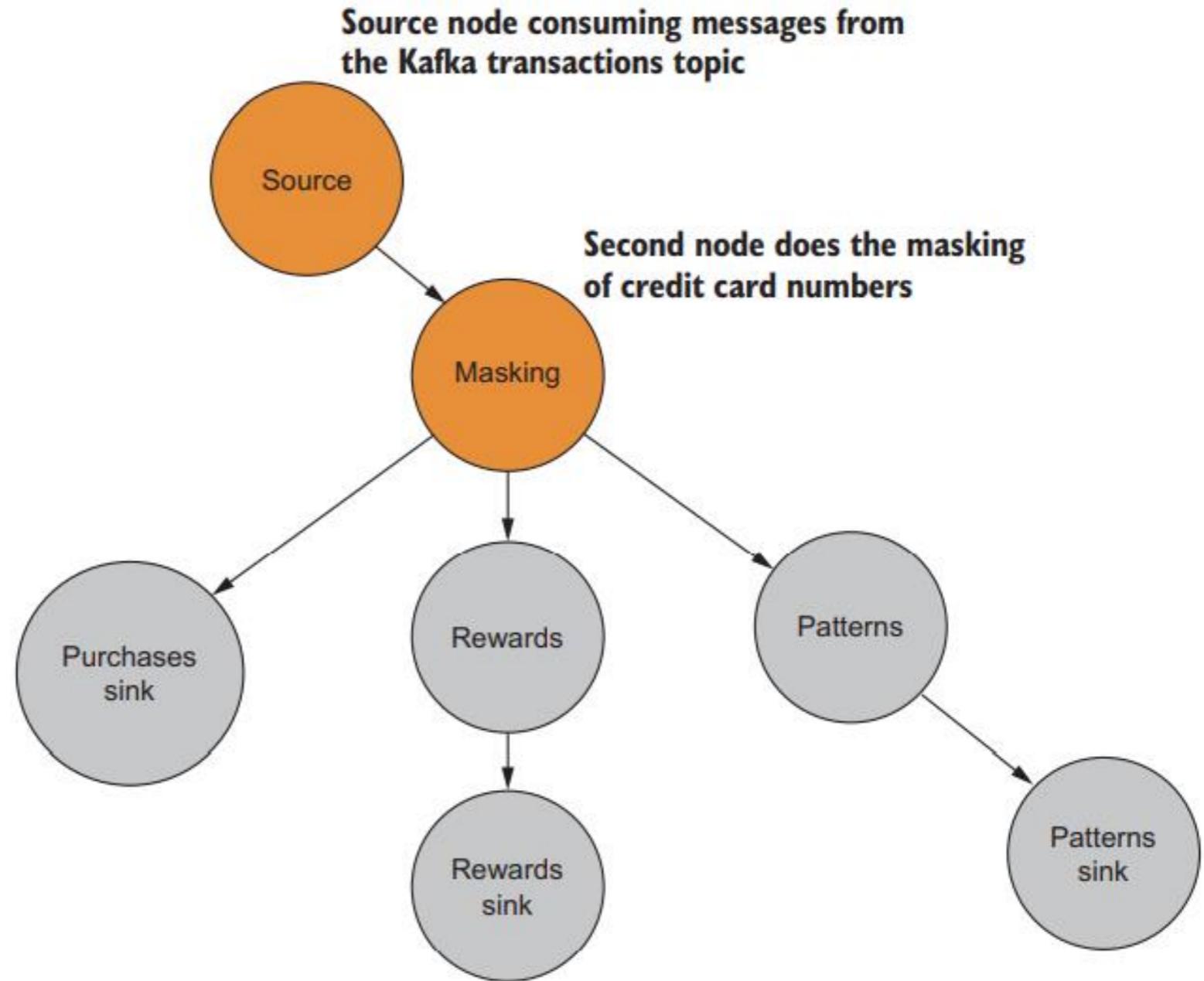
# Working with customer data

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

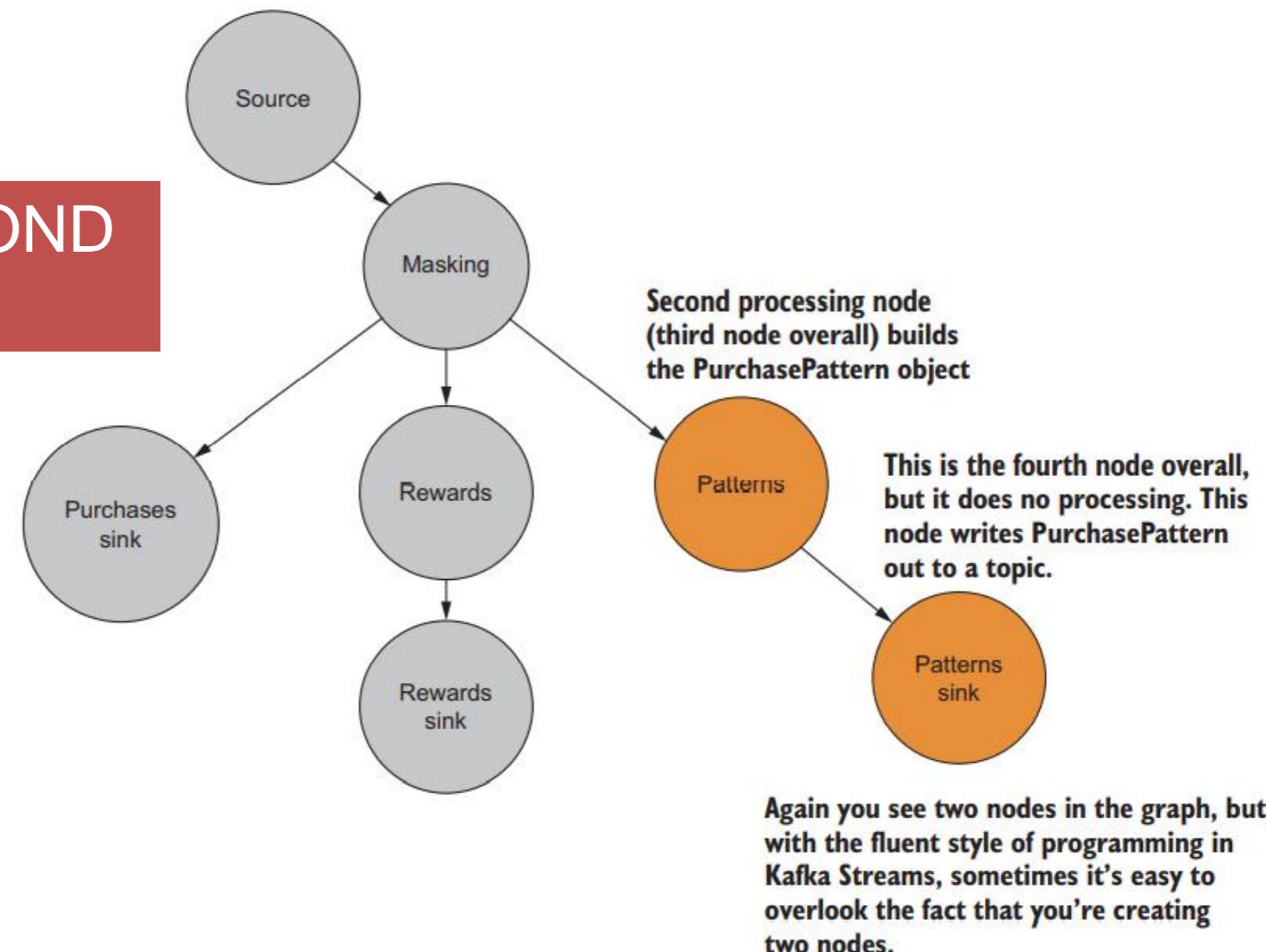
## **Listing 3.5 Building the source node and first processor**

```
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

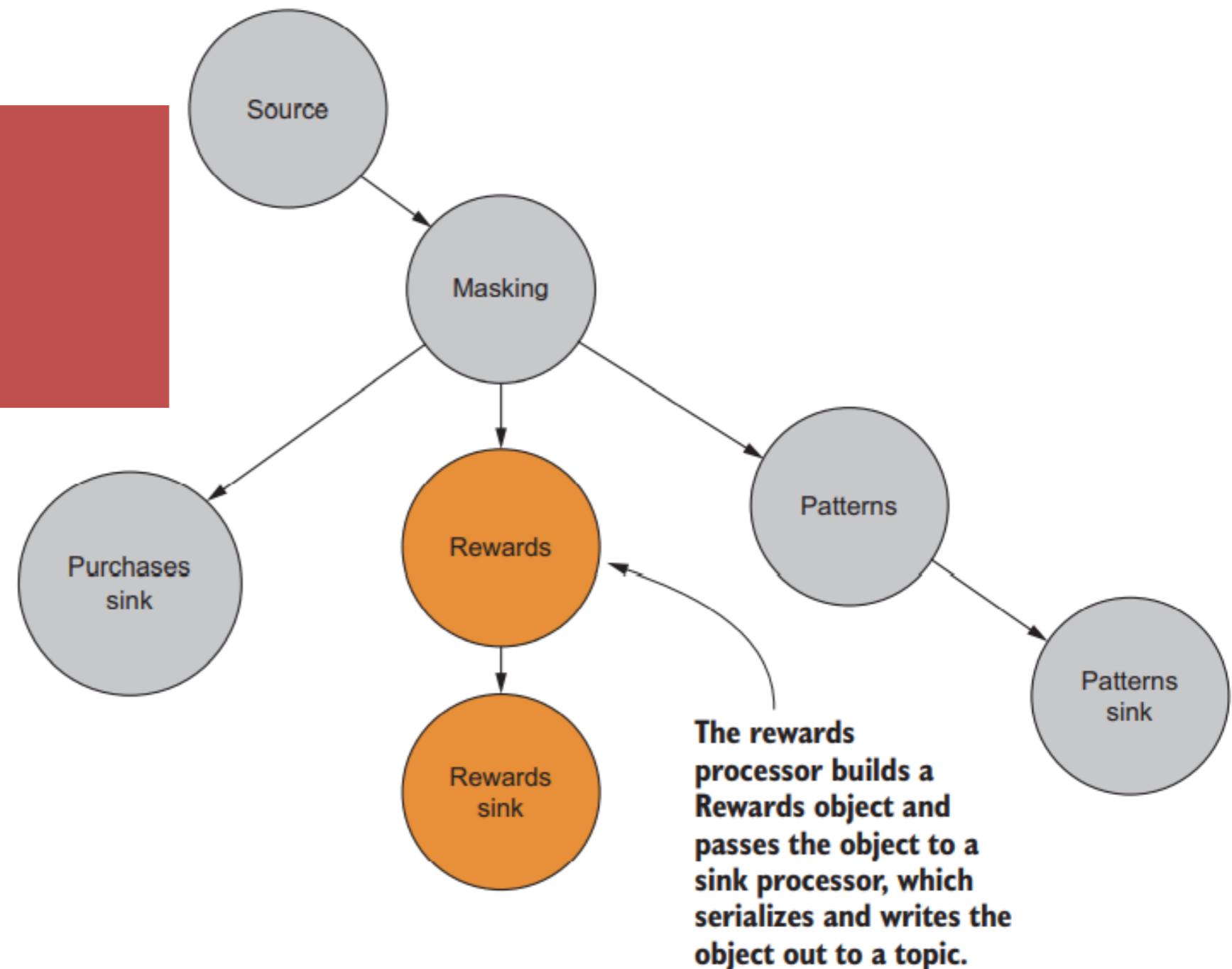


# BUILDING THE SECOND PROCESSOR

## **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

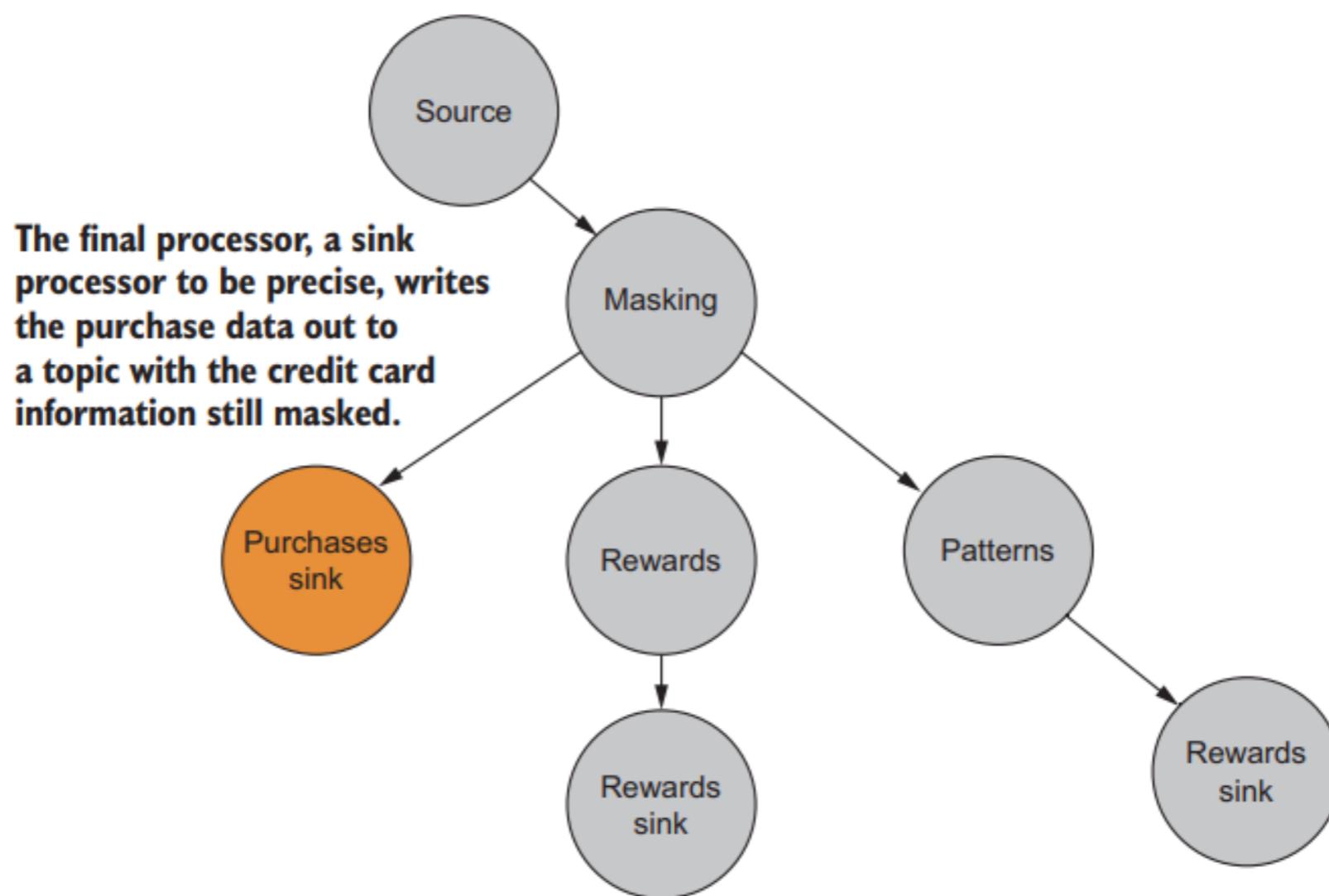


# BUILDING THE THIRD PROCESSOR

## **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<>();  
        JsonSerializer<Purchase> purchaseJsonDeserializer = | Creates the Serde; the  
        ➔ new JsonSerializer<>(Purchase.class); | data format is JSON.  
        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 = | Builds the  
        ➔ streamsBuilder.stream("transactions", | source and first  
        ➔ Consumed.with(stringSerde, purchaseSerde)) | processor  
        ➔ .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());
→ rewardsKStream.to("rewards",
→ Produced.with(stringSerde, rewardAccumulatorSerde)) ;

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

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

}
```

**Builds the RewardAccumulator processor**

**Builds the storage sink, the topic used by the storage consumer**

# 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**

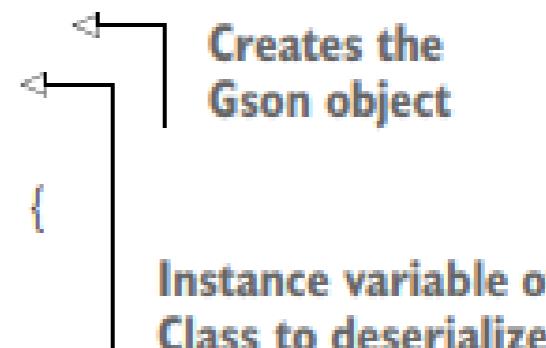
```
public class JsonSerializer<T> implements Serializer<T> {  
  
    private Gson gson = new Gson();  
  
    @Override  
    public void configure(Map<String, ?> map, boolean b) {  
  
    }  
  
    @Override  
    public byte[] serialize(String topic, T t) {  
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));  
    }  
  
    @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 diagram consists of two callout arrows pointing from text elements to specific parts of the code. The first arrow points from the annotation `@Override` to the `configure` method. The second arrow points from the variable `deserializedClass` to its declaration in the constructor.

**Creates the Gson object**

**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=18.5508}
[purchases]: null , Purchase{firstName='Andrew', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-3020', 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=20.8086}
[purchases]: null , Purchase{firstName='Eric', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-3020', 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-3020', 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=10.7134}
[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=40.81}
[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='Roh', lastName='Smith', creditCardNumber='vvvv-vvvv-vvvv-1938', itemPurchased='doughnuts', quantity=2, price=6.6758, purchaseDate=Sat Feb 13 21:20:45 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='20852', 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-3058', 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"));
```

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

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

Sets up to print the PurchasePattern transformation to the console

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, purchaseTotal=20.8086}
[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

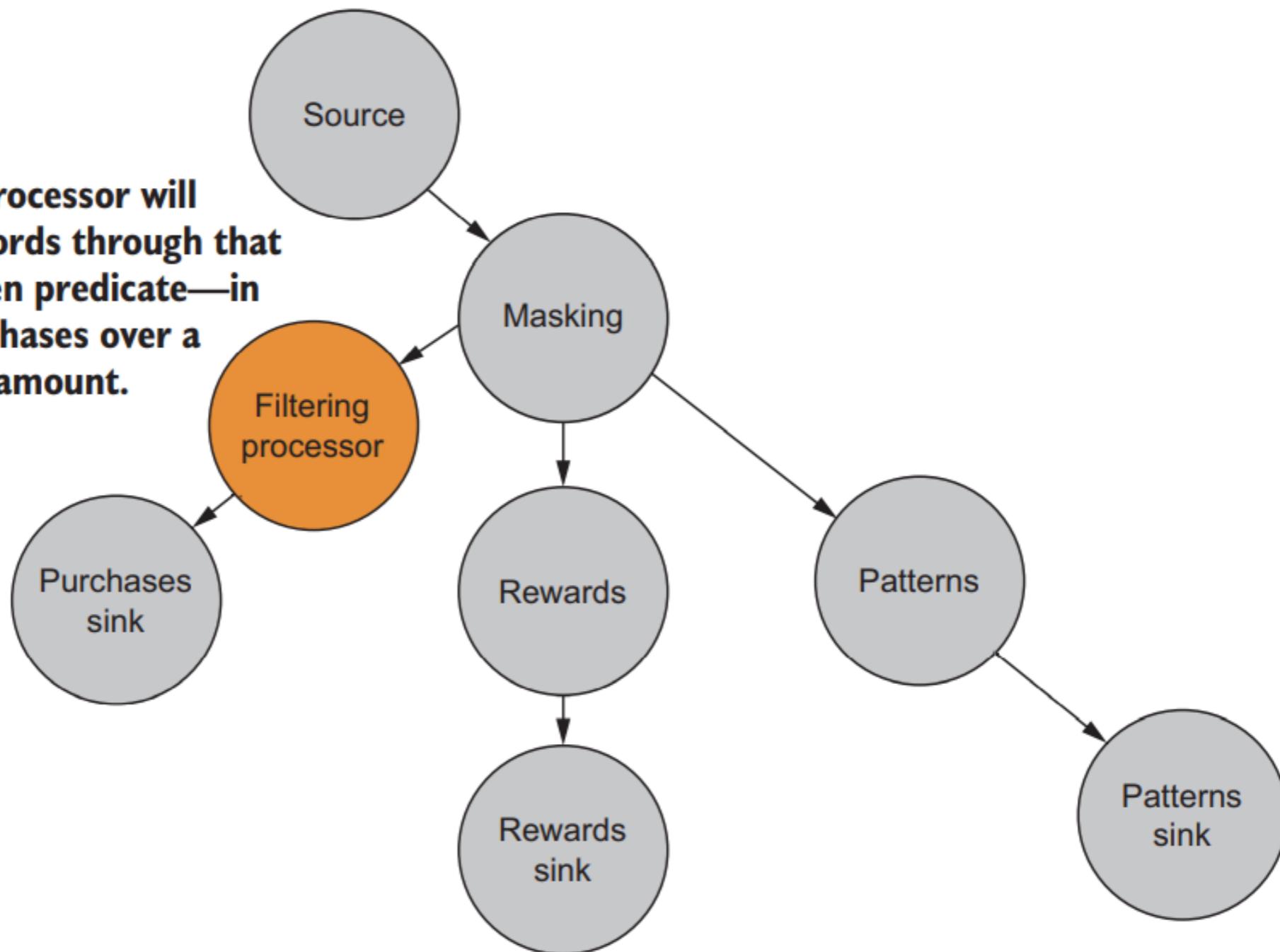
# Next steps

- 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

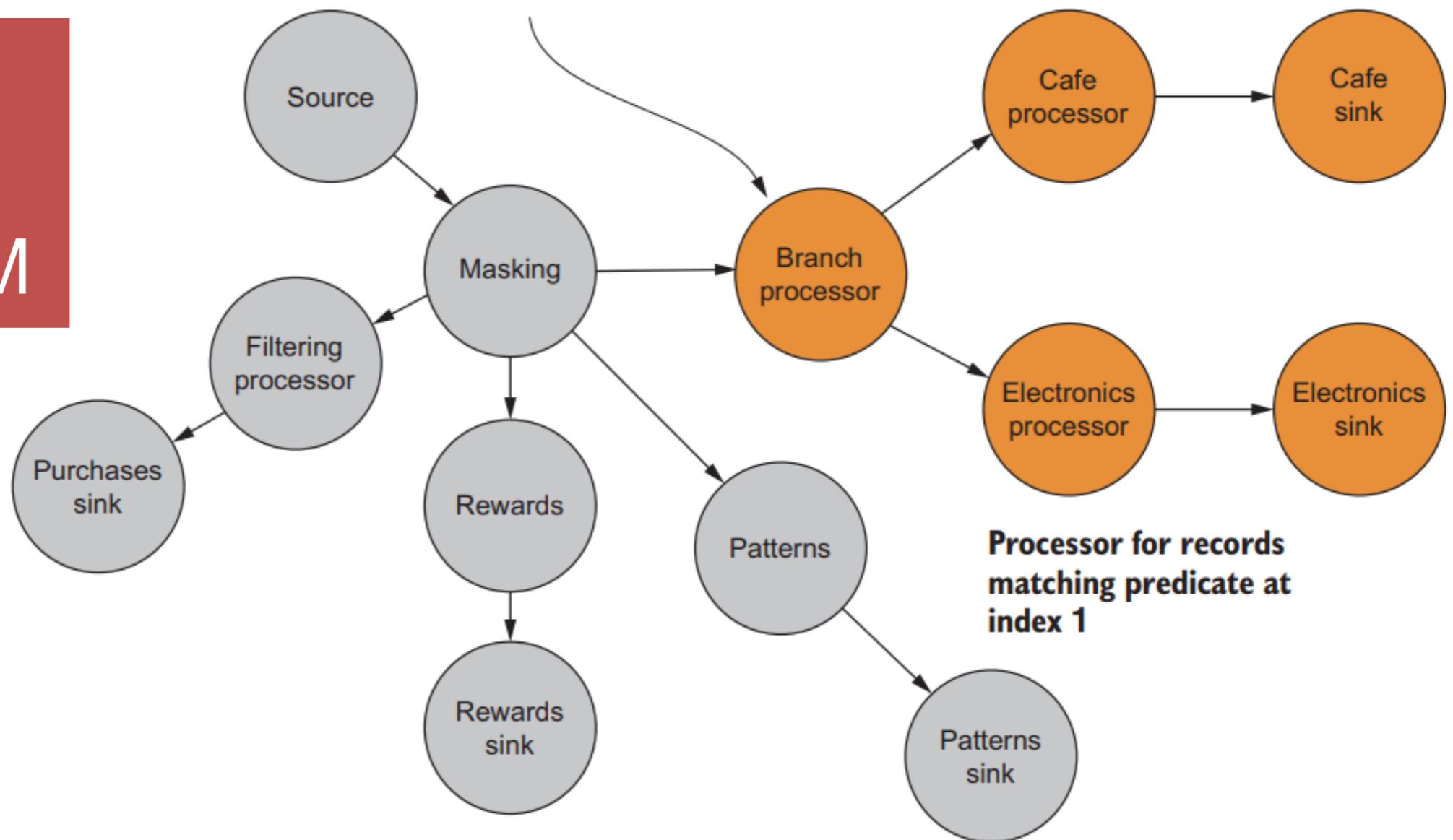
## Listing 3.12 Filtering on KStream

```
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.

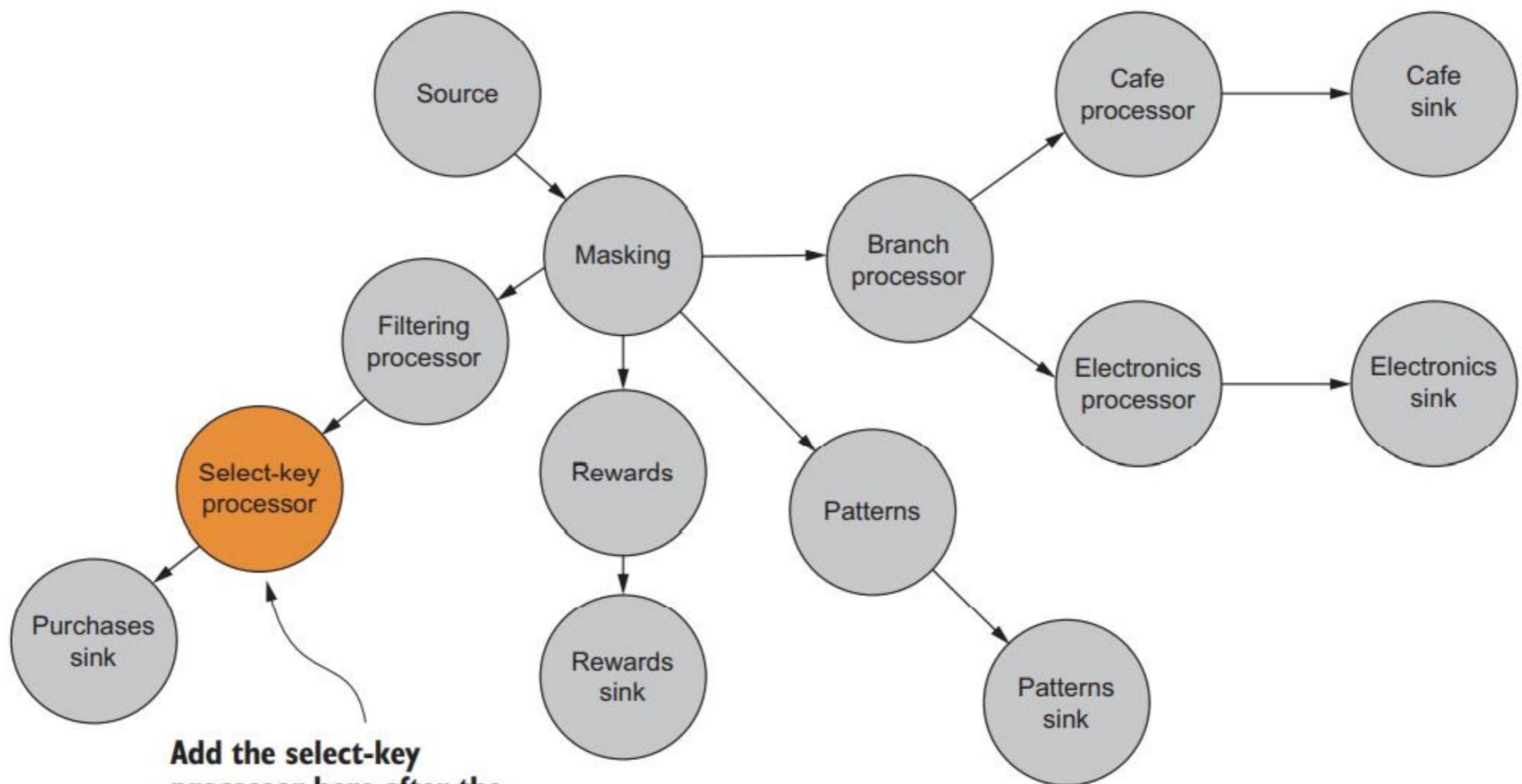
```
KStream<Long, Purchase> filteredKStream =  
    → purchaseKStream( (key, purchase) ->  
    → purchase.getPrice() > 5.00) .selectKey(purchaseDateAsKey);
```

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



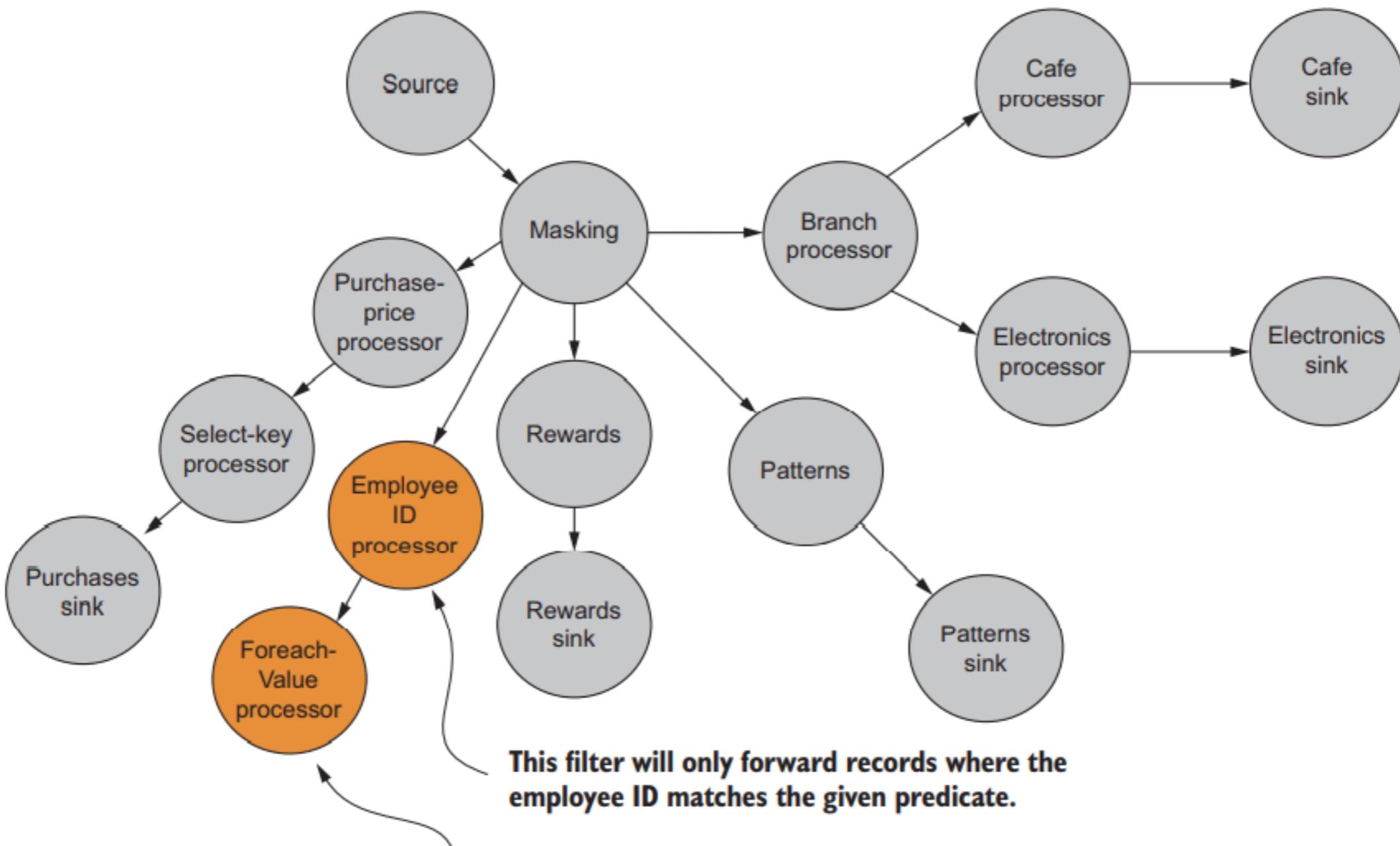
**Add the select-key processor here after the filtering, as you only need to generate keys for records that will be written out to the purchases topic.**

# 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 STREAM LAB 1& 2

# Streams concepts: KStream / KTable

# The KTable API

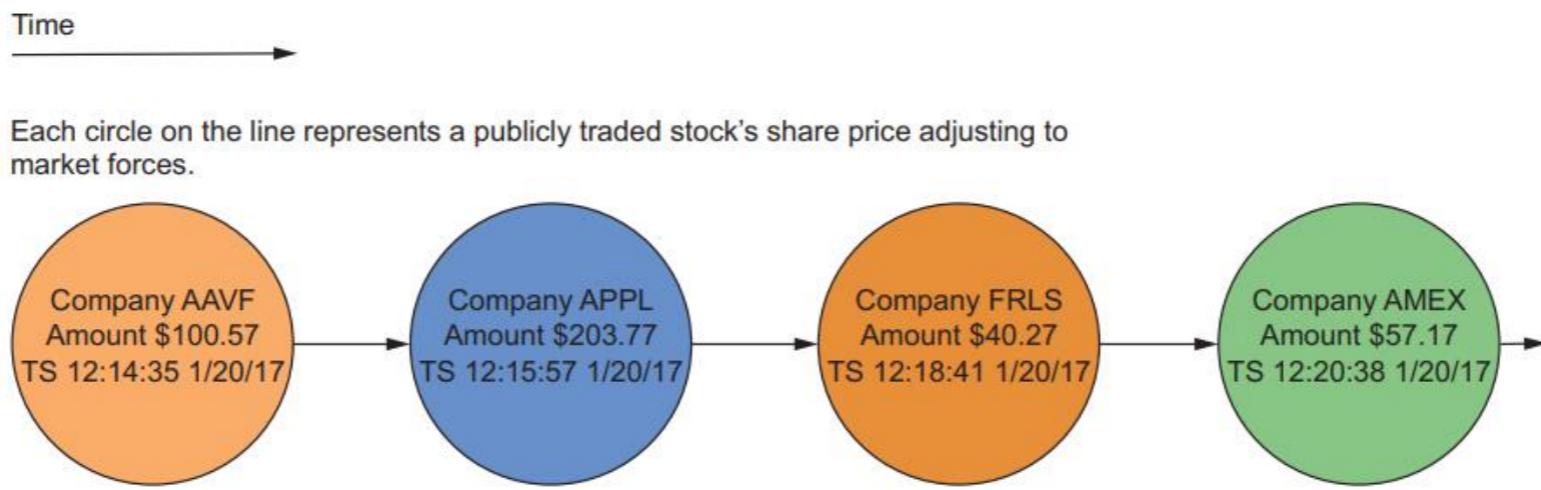
This lesson covers

- 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

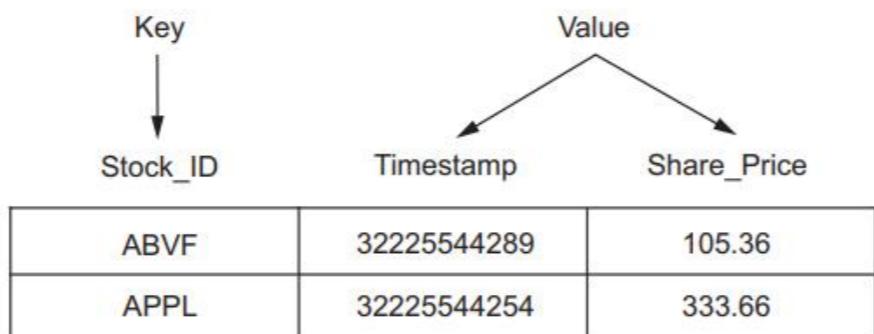
## The record stream

- 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.



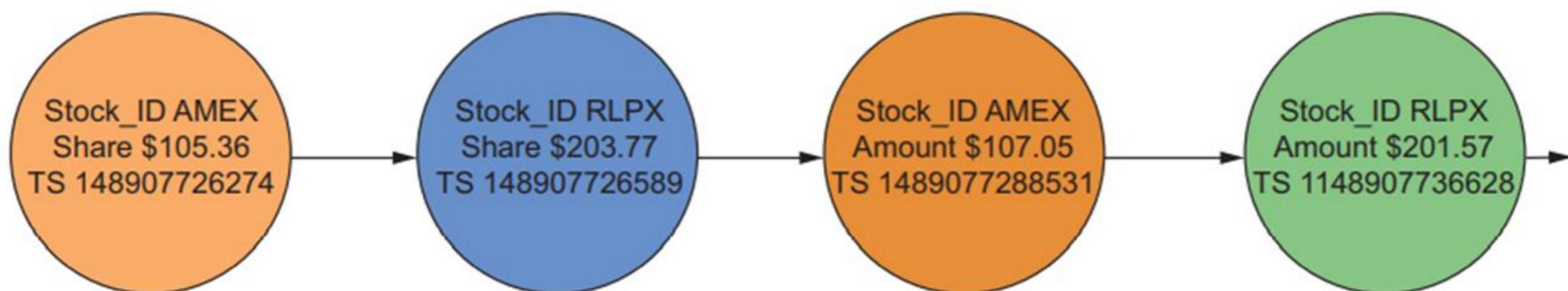
Imagine that you are observing a stock ticker displaying updated share prices in real time.

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



**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}}`

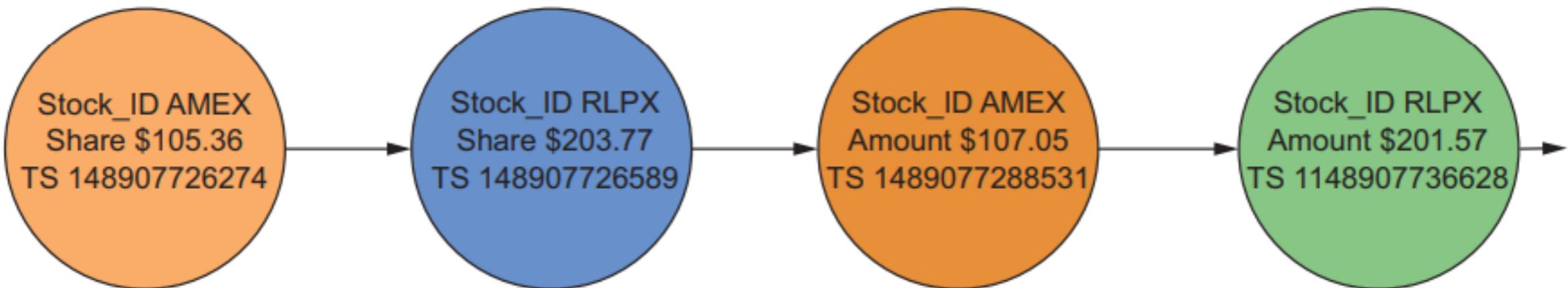


Key	Stock_ID	Timestamp	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

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.



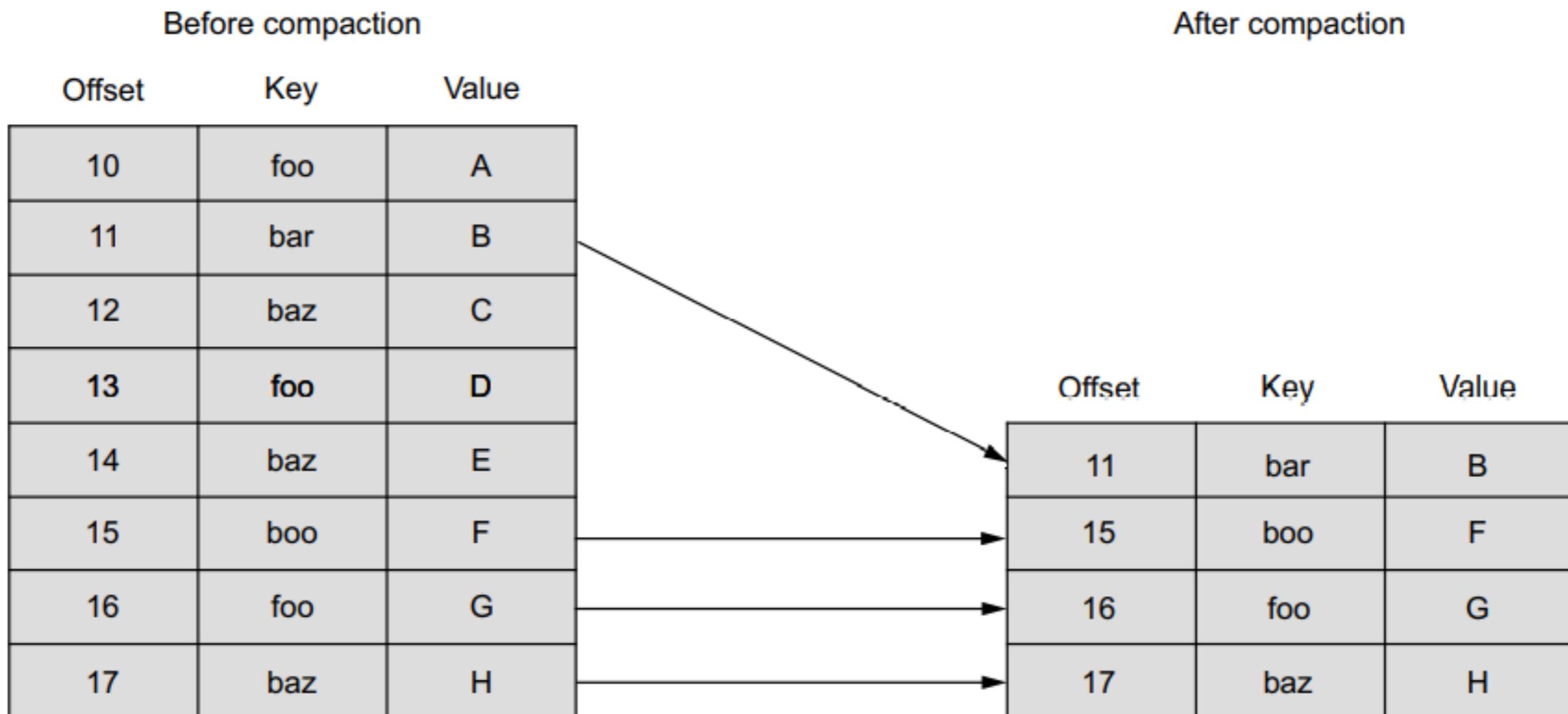
Stock_ID	Timestamp	Share_Price
AMEX	148907726274	105.36
RLPX	148907726589	203.77
AMEX	1489077288531	107.05
RLPX	148907736628	201.57

**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.

Here is the last update record for the KTable.

```
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
```

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);  
  
KStream<String, StockTickerData> stockTickerStream =  
    builder.stream(STOCK_TICKER_STREAM_TOPIC);  
  
stockTickerTable.toStream()  
    .print(Printed.<String, StockTickerData>toSysOut())  
    .withLabel("Stocks-KTable"));  
  
stockTickerStream  
    .print(Printed.<String, StockTickerData>toSysOut())  
    .withLabel("Stocks-KStream"));
```

Creates the KTable instance

Creates the KStream instance

KTable prints results to the console

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

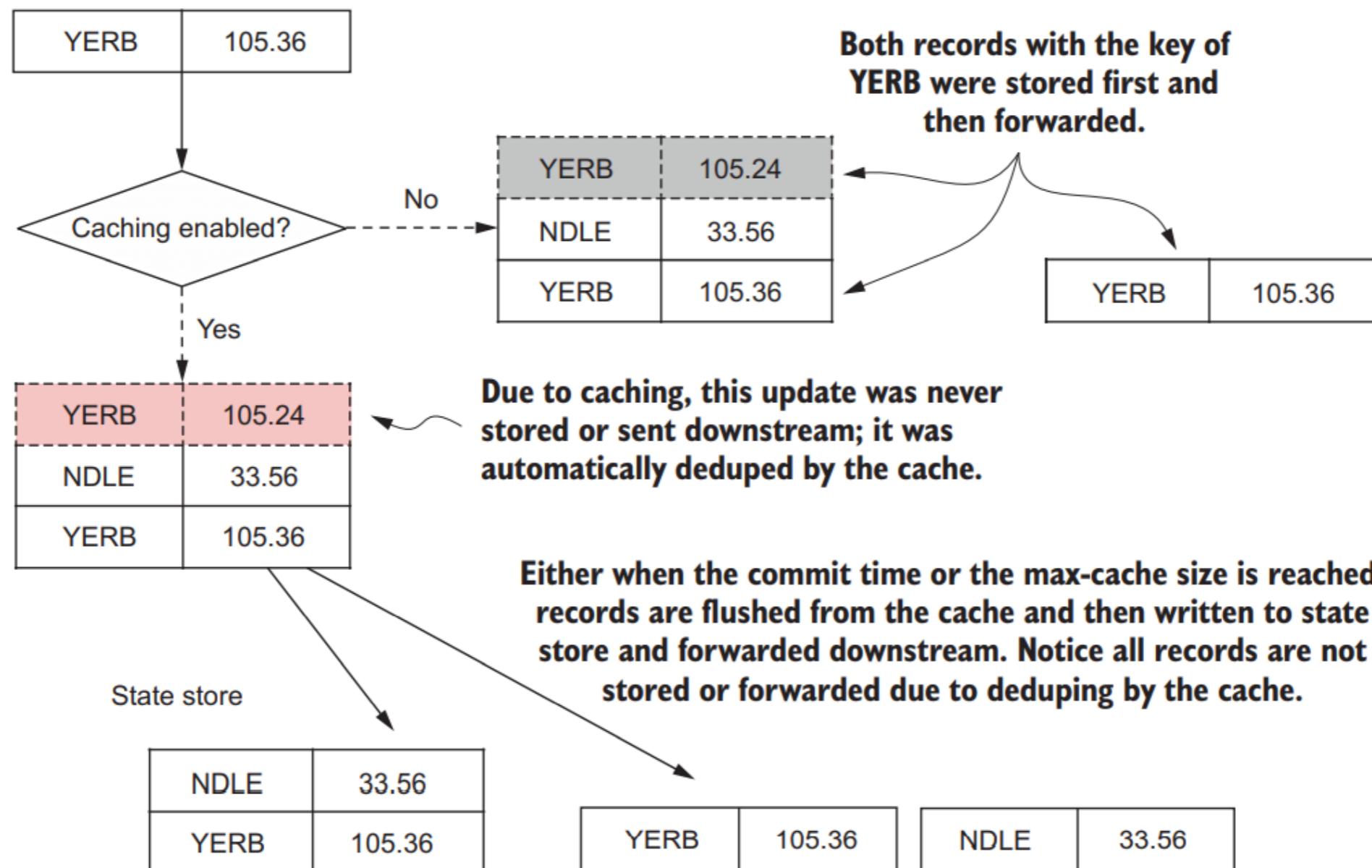
**KTable caching deduplicates updates to records with the same key, preventing a flood of consecutive updates to child nodes of the KTable in the topology.**

# 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:

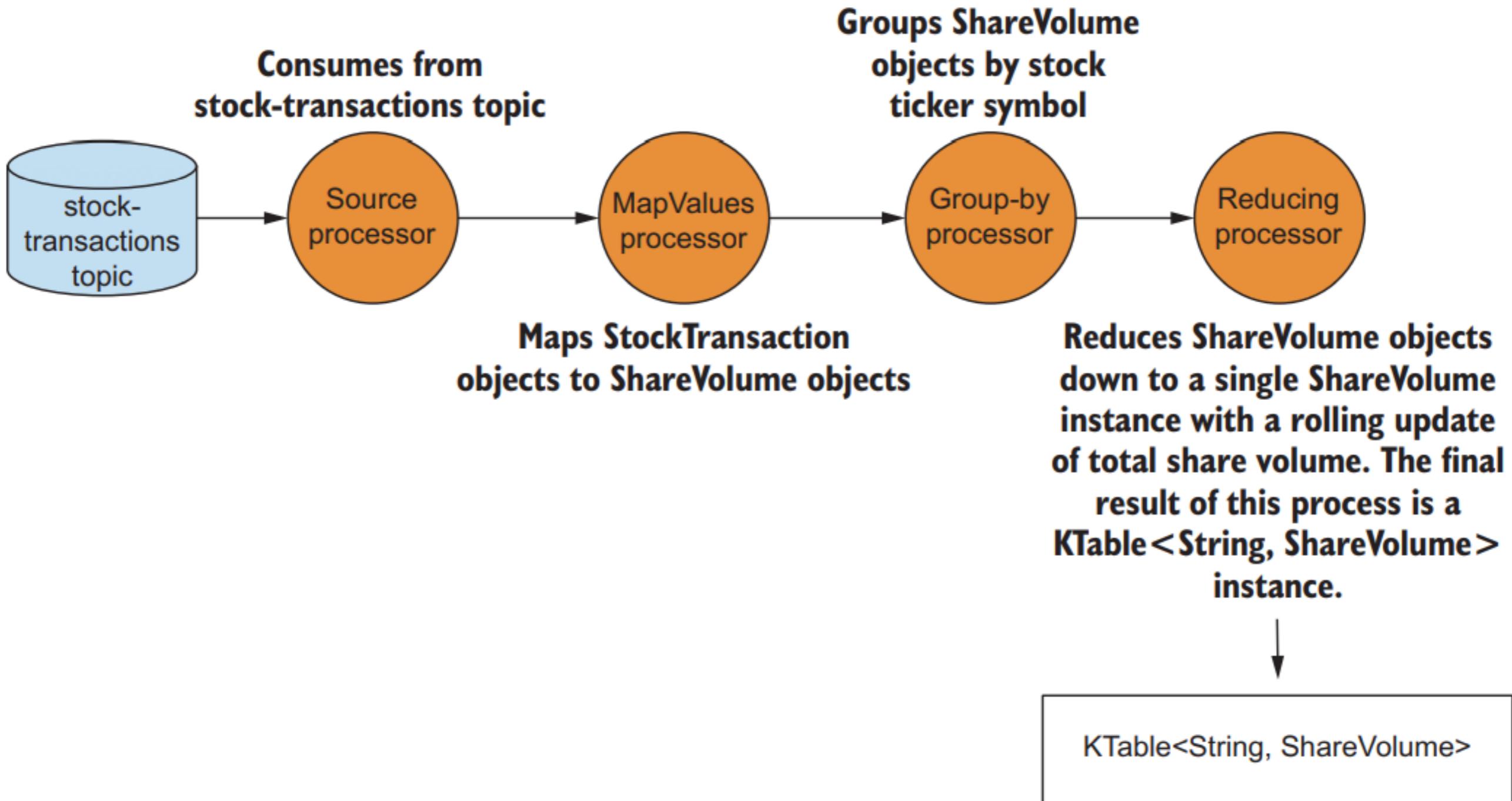
- How to set up a processing topology
- How to use state in your streaming application
- How to perform joins between streams
- The difference between an event stream (`KStream`) and an update stream (`Ktable`)

# 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);
```

Maps StockTransaction  
objects to ShareVolume  
objects

Reduces  
ShareVolume  
objects to contain  
a rolling aggregate  
of share volume

The source processor  
consumes from a topic.

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

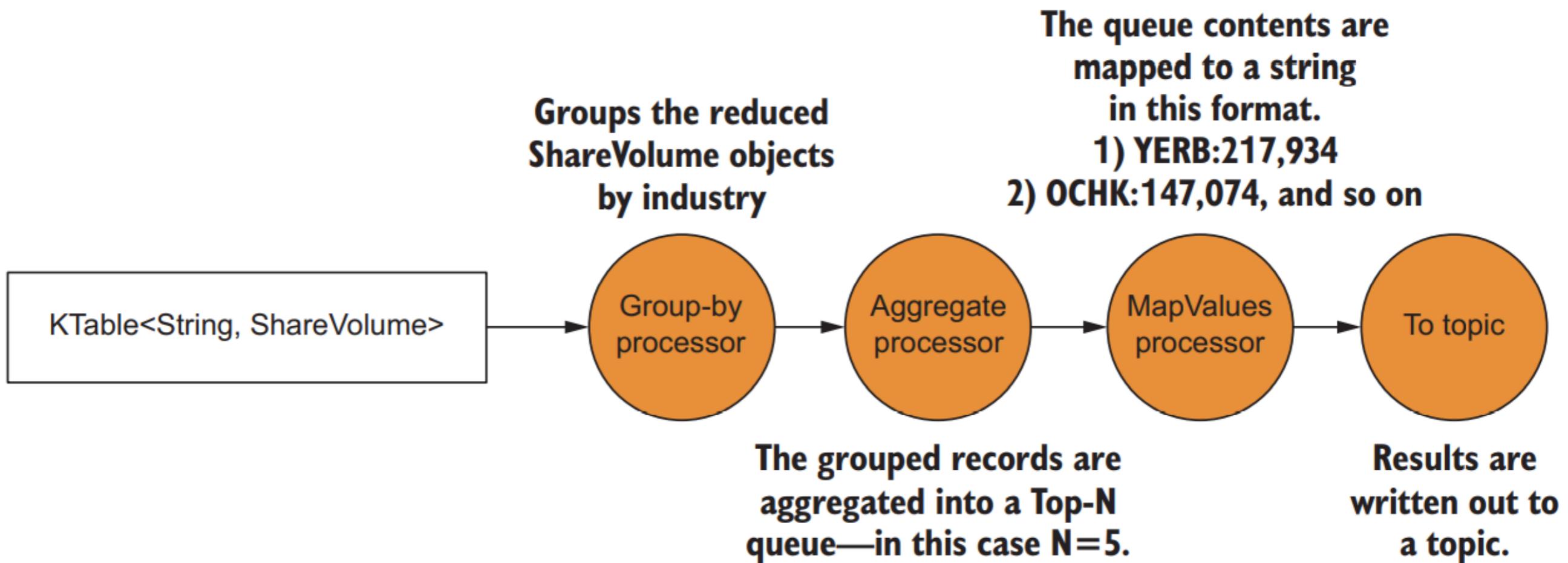
## **Listing 5.3 The ShareVolume.sum method**

```
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()  
  
FixedSizePriorityQueue<ShareVolume> fixedQueue =  
  ➔ new FixedSizePriorityQueue<>(comparator, 5);  
  
shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),  
  ➔ Serialized.with(stringSerde, shareVolumeSerde))  
    .aggregate(() -> fixedQueue,  
      ➔ (k, v, agg) -> agg.add(v),  
      ➔ (k, v, agg) -> agg.remove(v),  
      ➔ Materialized.with(stringSerde, fixedSizePriorityQueueSerde))
```

Aggregate adder adds  
new updates

Aggregate remover  
removes old updates

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

Groups by industry and  
provides required serdes

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?

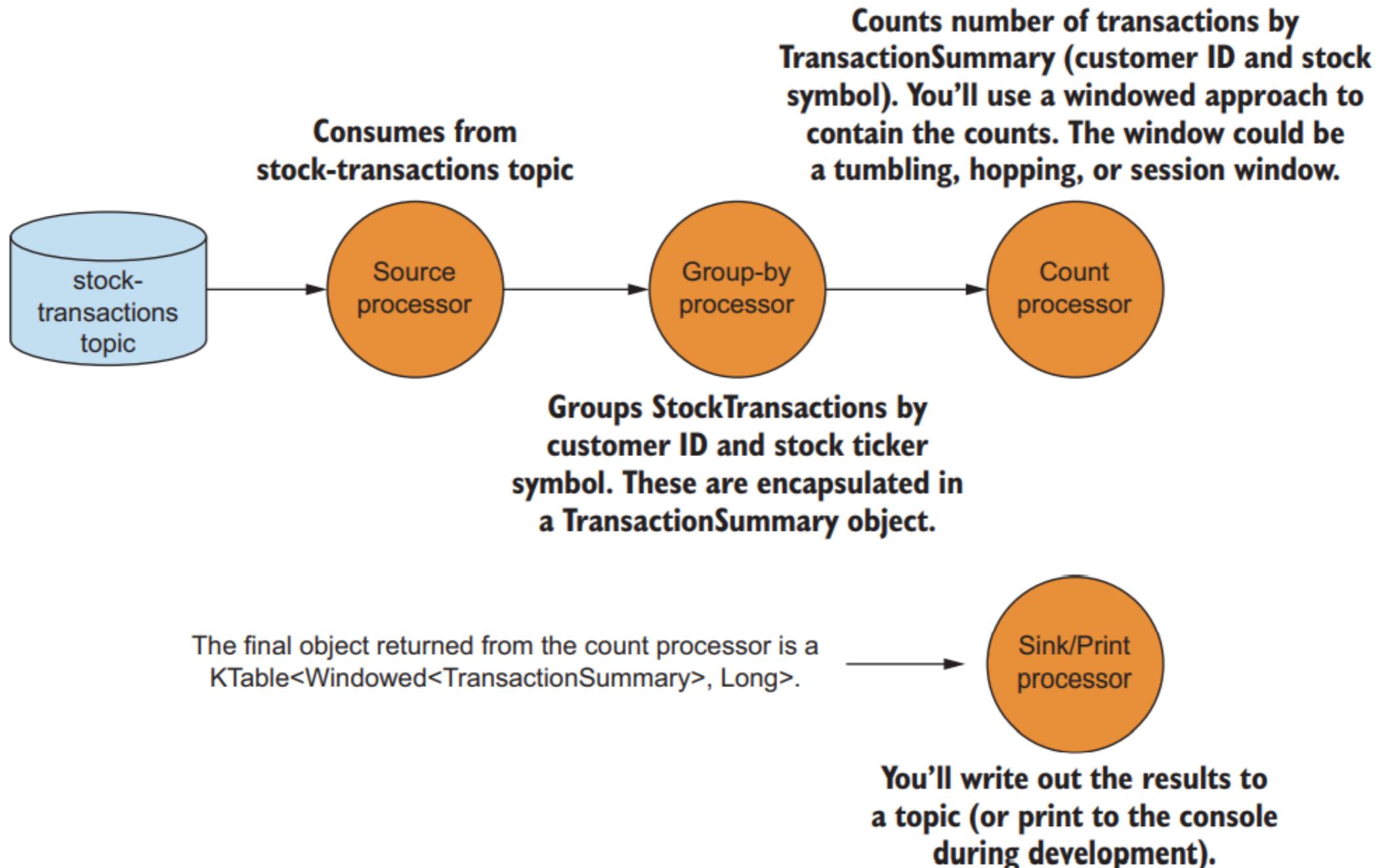
# COUNTING STOCK TRANSACTIONS BY CUSTOMER

- 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.

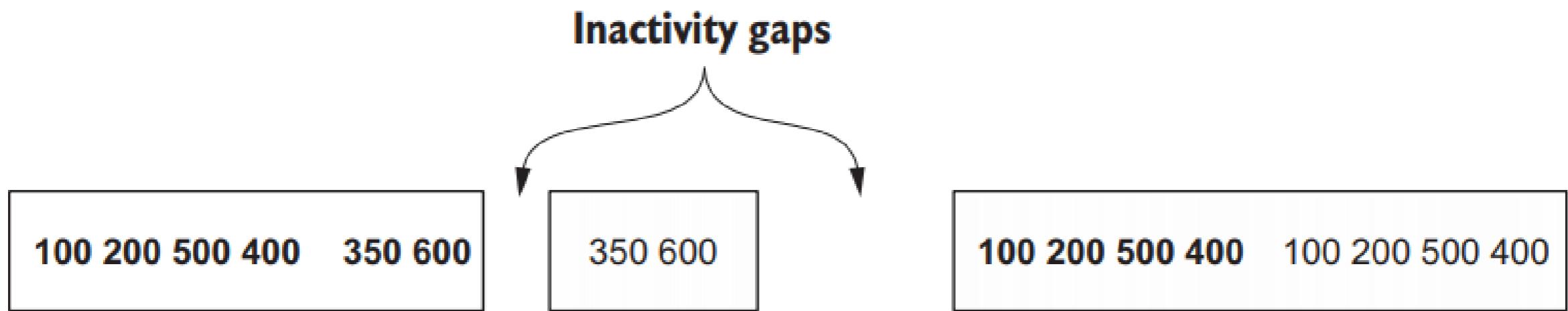


# WINDOW TYPES

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

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");
```

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

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

# USING SESSION WINDOWS

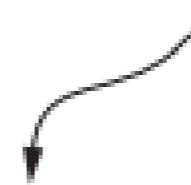
- The code in listing 5.5 does a count over session windows.
- Figure breaks it down

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



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

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



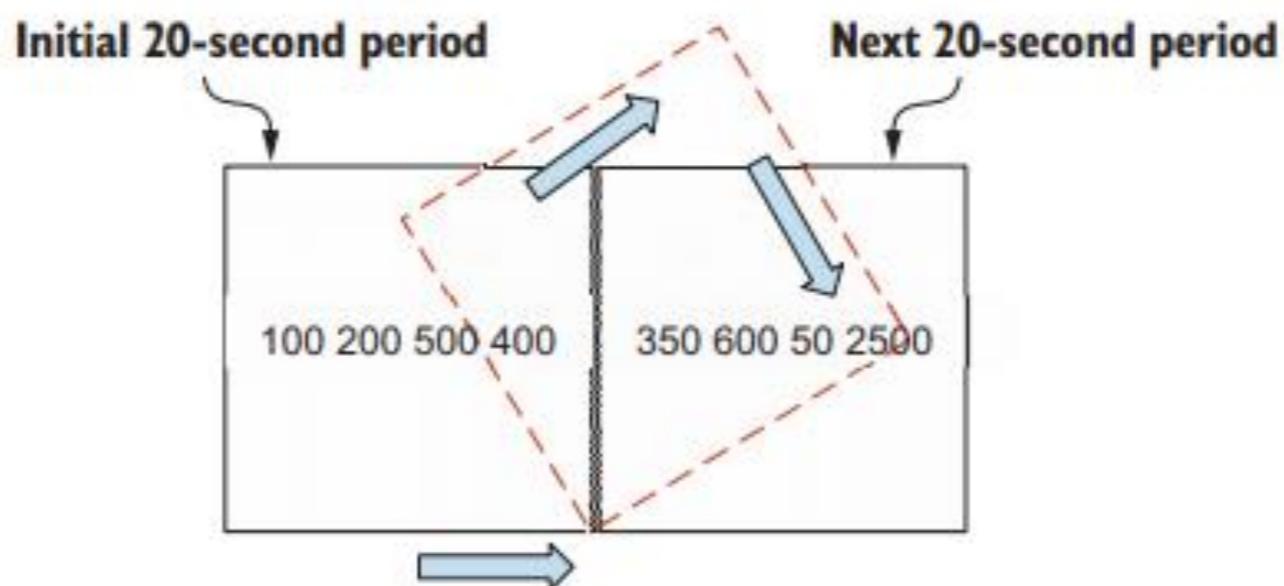
# USING SESSION WINDOWS

- 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

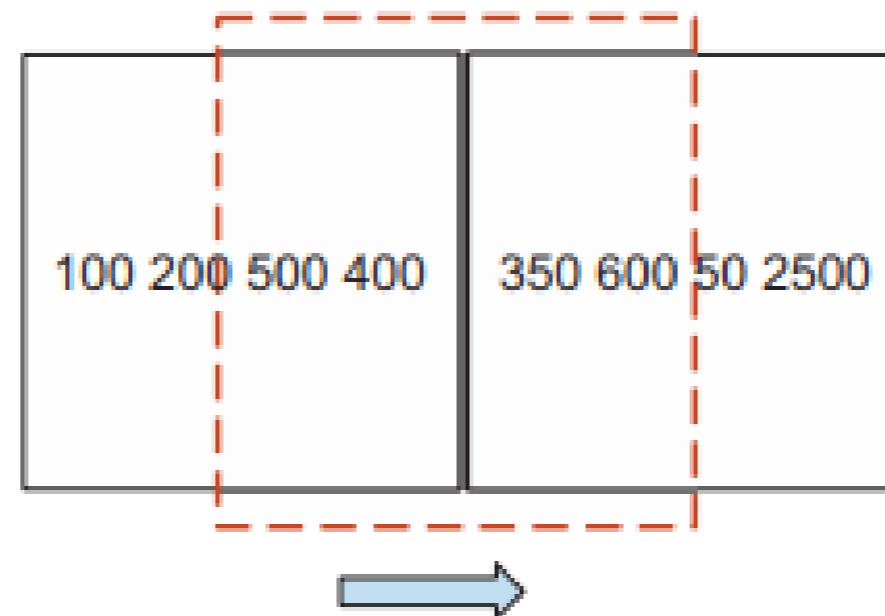
```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
  builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
                                                       transactionSerde)  
  .withOffsetResetPolicy(LATEST))  
    .groupBy(noKey, transaction) -> TransactionSummary.from(transaction),  
  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  
  
Takes the count value  
from the aggregation  
and places it in the  
TransactionSummary  
object  
  
Returns the new  
KeyValue pair for  
the KStream
```

# 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 =  
    ➔ (txnct, news) ->  
    ➔ String.format("%d shares purchased %s related news [%s]",  
    ➔ txnct.getSummaryCount(), txnct.getStockTicker(), news);      ↪ ValueJoiner  
                                                               combines the  
                                                               values from the  
                                                               join result.  
  
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)  
  
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='037-34-5104', 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());  
  
GlobalKTable<String, String> clients =  
    builder.globalTable(CLIENTS.topicName());
```

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

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

# JOINING KSTREAMS WITH GLOBALKTABLES

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

**Listing 5.13 Joining a KStream with two GlobalKTables**

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

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 GLOBALTABLES

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.

# Session 5

## Monitoring and troubleshooting kafka

# Monitoring Tools Overview

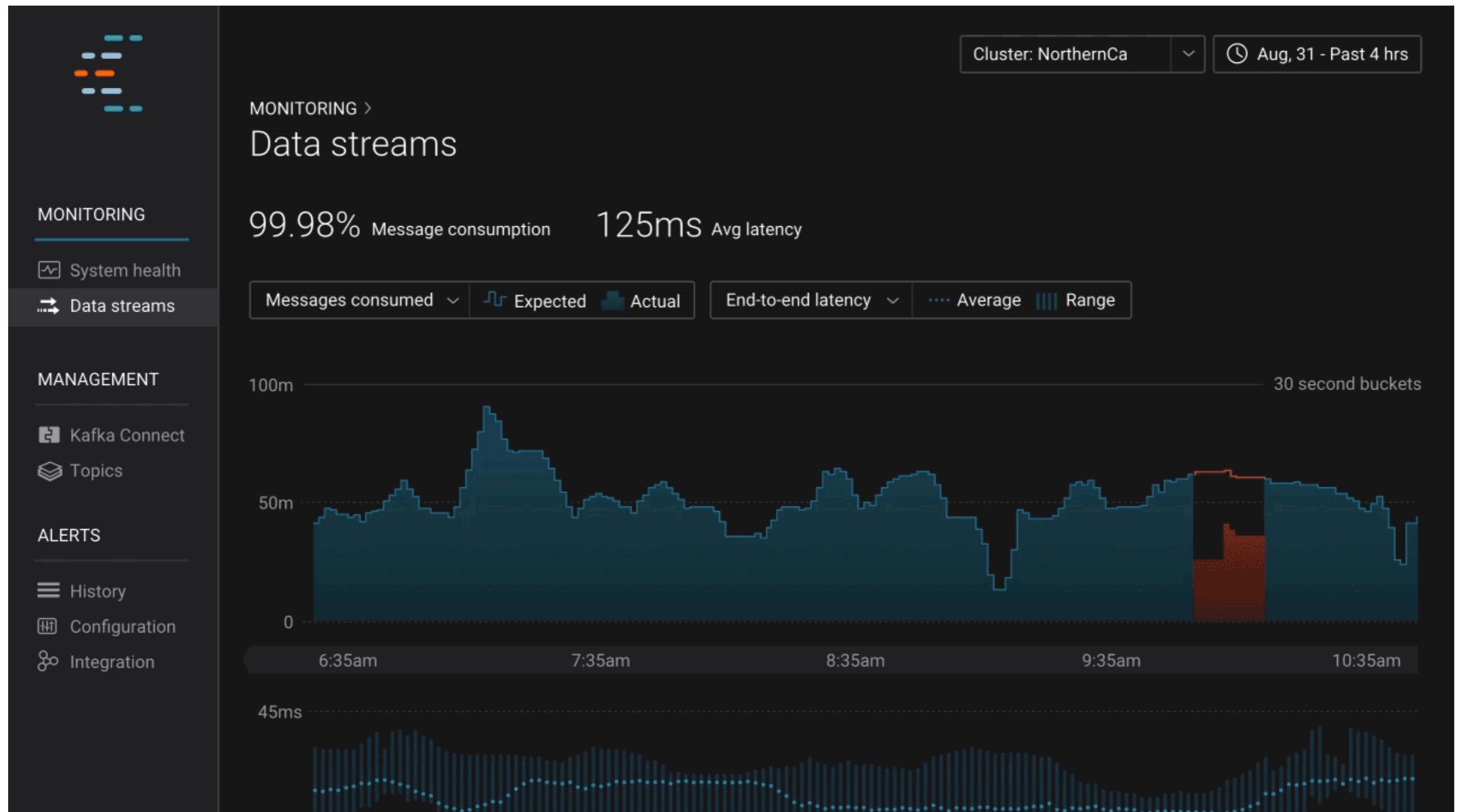
# Monitoring Tools Overview



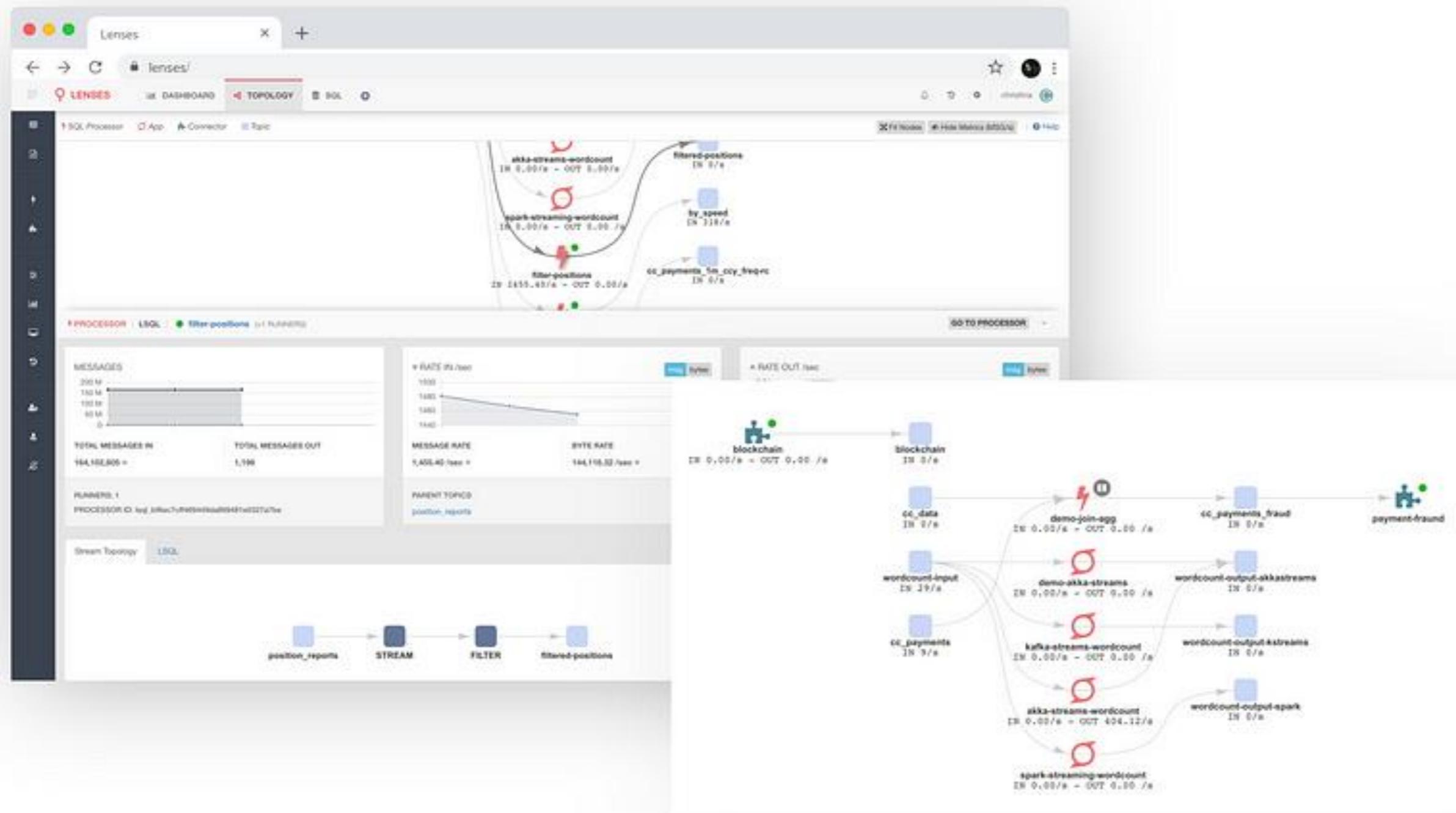
## BEST MONITORING TOOLS FOR APACHE KAFKA?

- ❖ Confluent Control Centre
- ❖ Lenses
- ❖ Datadog Kafka Dashboard
- ❖ Cloudera Manager
- ❖ Yahoo Kafka Manager
- ❖ KafDrop
- ❖ LinkedIn Burrow
- ❖ Kafka Tool

# Confluent Control Centre



# Lenses



# Lenses

- ❖ Lenses works with any Kafka distribution, delivers high quality enterprise features and monitoring, SQL for ALL and self-serviced real-time data access and flows on Kubernetes.
- ❖ The company also offers Lenses Box, which is a free all-in-one docker that can serve a single broker for up to 25M messages.

# Kafka Topics UI

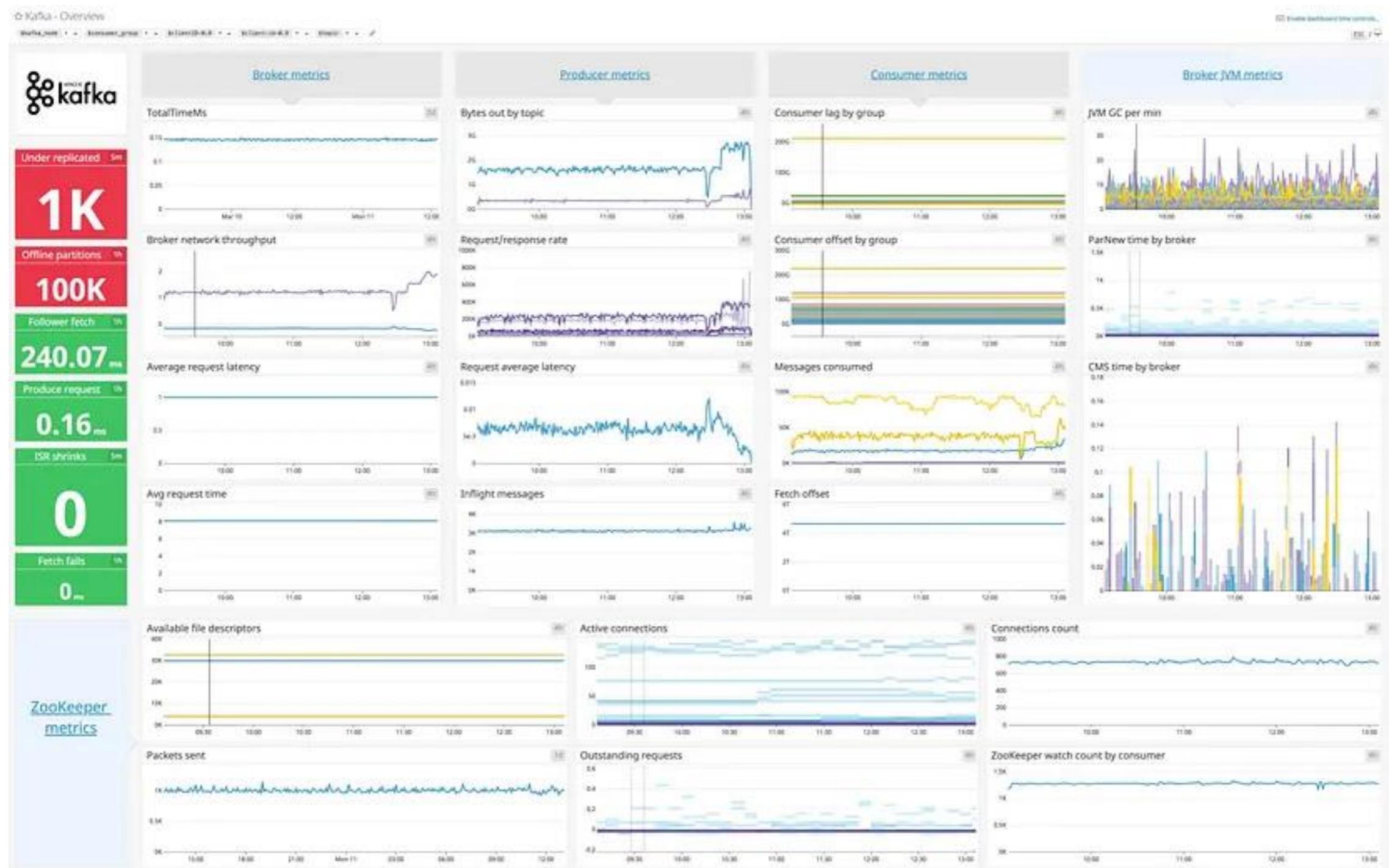
- ❖ Furthermore, Lenses also offers Kafka Topics UI, which is a web tool for managing Kafka Topics.

The screenshot shows the Kafka Topics UI interface. On the left, there's a sidebar with a dark header "KAFKA TOPICS" and a "4 TOPICS" summary. A "System Topics" toggle switch is off. Below it is a search bar labeled "Search topics:" followed by a list of topics: "position-reports", "rethink-fx-roundtrip", "yahoo-fx", and "yahoo-stocks". Each topic entry includes its partition count and replication factor. To the right, the main panel is focused on the "position-reports" topic, which has 1 partition and 1 replication. The "DATA" tab is selected, showing "Total Messages Fetched: 500, Data type: avro". Below this are buttons for "filter", "Partition 0", "Seek to offset", and "Seek to beginning". A table below lists raw data entries. The first entry is:

TOPIC	TABLE	RAW DATA	Actions
position-reports		<p>▶ Key: { MMSI: 265650970 }</p> <p>▶ Value: { Type: 1, Repeat: 0, MMSI: 265650970, Speed: 0, Accuracy: true, Longitude: 11.841745, Latitude: 57.66149333333333, location: "Bogota", timestamp: "2018-01-01T12:00:00Z" }</p>	Offset: 5000 Partition: 0
position-reports		<p>▶ Key: { MMSI: 265509180 }</p> <p>▶ Value: { Type: 1, Repeat: 0, MMSI: 265509180, Speed: 8.2, Accuracy: true, Longitude: 17.051526666666668, Latitude: 56.84044668 }</p>	Offset: 5001 Partition: 0
position-reports		<p>▶ Key: { MMSI: 265647200 }</p> <p>▶ Value: { Type: 1, Repeat: 0, MMSI: 265647200, Speed: 0.9, Accuracy: true, Longitude: 11.846096666666668, Latitude: 57.55822166 }</p>	Offset: 5002 Partition: 0

At the bottom left, there's a "Powered by Landoop" logo.

# Datadog Kafka Dashboard



# Cloudera Manager

The screenshot shows the Cloudera Manager interface for a Kafka cluster named 'KAFKA-1'. The top navigation bar includes links for Clusters, Hosts, Diagnostics, Audits, Charts, Backup, and Administration. The main content area displays the following components:

- Kafka functions such as stopping and starting the cluster:** A green checkmark icon next to 'KAFKA-1'.
- List of running processes:** Buttons for Status, Instances, and Configuration, with 'Status' being the active tab.
- Configuration options such as debugging, broker properties:** A section titled 'Health Tests' with a 'Create Trigger' button.
- Active charts for monitoring Kafka behaviors:** A large orange-bordered box containing a chart titled 'Active Controllers' and a 'Status Summary' table.
- Health monitor:** A section titled 'Health History' with a timestamp of 5:45:04 and a link to 'Kafka Broker Health'.
- Charts:** Two charts: 'Active Controllers' (showing 1 controller) and 'Total Messages Received Across Kafka Broke...' (showing 0 messages).

Annotations on the left side map specific features to the interface elements:

- Kafka functions such as stopping and starting the cluster: Points to the green checkmark icon next to 'KAFKA-1'.
- List of running processes: Points to the 'Status' button under the Kafka cluster name.
- Configuration options such as debugging, broker properties: Points to the 'Configuration' button under the Kafka cluster name.
- Active charts for monitoring Kafka behaviors: Points to the large orange-bordered box containing the 'Status Summary' table and the two charts.
- Health monitor: Points to the 'Health History' section at the bottom of the page.

# Yahoo Kafka Manager

Yahoo Kafka Manager is an open-source managing tool for Apache Kafka clusters. With Kafka Manager, you can:

- ❖ Manage multiple clusters
- ❖ Easy inspection of cluster state (topics, consumers, offsets, brokers, replica distribution, partition distribution)
- ❖ Run preferred replica election
- ❖ Generate partition assignments with option to select brokers to use
- ❖ Run reassignment of partition (based on generated assignments)
- ❖ Create a topic with optional topic configs
- ❖ Delete topics

# Yahoo Kafka Manager

Kafka Manager local Cluster ▾ Brokers Topic ▾ Consumers Preferred Replica Election Reassign Partitions

Clusters / local / Topics

## Operations

Generate Partition Assignments Manually Set Partition Assignments Run Partition Assignments Add Partitions

## Topics

Show 10 entries Search:

Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	# Replicas	Under Replicated %	Summed Recent Offsets
another-topic	24	3	100	0	3	0	0
test-topic	12	3	100	0	1	0	2066

# KafDrop

## Kafdrop

---

### Broker Id: 1

#### Broker Overview

Host	████████.homeadvisor.com:████
Start Time	2016-03-16 15:49:05.969-0600
Controller	Yes
# of Topics	116
# of Partitions	1625

#### Topic Detail

Topic	Total Partitions	Broker Partitions	Partition Ids
_consumer_offsets	50	13	0,4,8,12,16,20,24,28,32,36,40,44,48
address.geocode	53	13	2,6,10,14,18,22,26,30,34,38,42,46,50
address.geocode.error	53	13	3,7,11,15,19,23,27,31,35,39,43,47,51
api.log	53	13	2,6,10,14,18,22,26,30,34,38,42,46,50

# LinkedIn Burrow

- ❖ LinkedIn Burrow is an open-source monitoring companion for Apache Kafka that provides consumer lag checking as a service without the need for specifying thresholds.
- ❖ It monitors committed offsets for all consumers and calculates the status of those consumers on demand.
- ❖ An HTTP endpoint is provided to request status on demand, as well as provide other Kafka cluster information.
- ❖ There are also configurable notifiers that can send status out via email or HTTP calls to another service.

# Kafka Tool

Kafka Tool is a GUI application for managing and using Apache Kafka clusters. It provides an intuitive UI that allows one to quickly view objects within a Kafka cluster as well as the messages stored in the topics of the cluster. It contains features geared towards both developers and administrators. Using Kafka Tool, you can:

- ❖ View metrics on cluster, broker, topic and consumer level
- ❖ View contents of messages in your partitions and add new messages
- ❖ View offsets of the Kafka consumers, including Apache Storm Kafka spout consumers
- ❖ Show JSON and XML messages in a pretty-printed format

# Comparison and Conclusions

- ❖ If you cannot afford commercial licenses then your options are Yahoo Kafka Manager, LinkedIn Burrow, KafDrop and Kafka Tool.
- ❖ In my opinion, the former is a comprehensive solution that should do the trick for most of the use-cases.
- ❖ If you are running relatively big Kafka Clusters, then it is worth paying for a commercial license.
- ❖ Confluent and Lenses offer more rich functionality compared to the other monitoring tools we've seen in this post and I would highly recommend both of them.

# Monitoring Kafka

# Monitoring Kafka

- ❖ Kafka clusters are complex by definition, making it difficult to know at a glance if they're healthy.
- ❖ It is therefore essential to have an effective solution to monitor them and react.

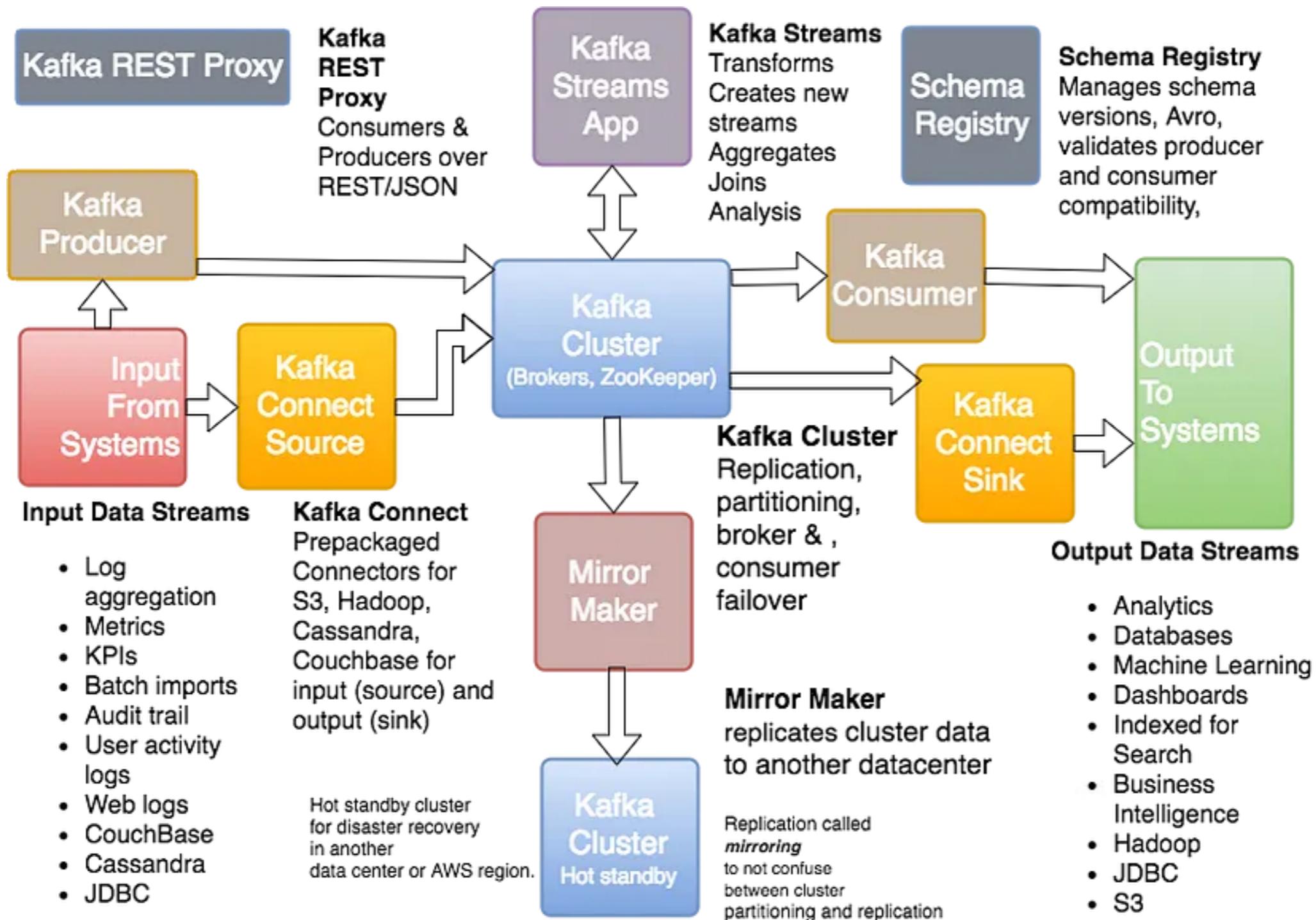


# Monitoring Kafka

Here we will focus on the following topics:

- ❖ **Domains:** the definition of the components of the Kafka ecosystem to which monitoring will apply
- ❖ **Metrics:** the set of metrics that we've selected to implement our monitoring process
- ❖ **Solution:** the description of monitoring implementation based on ELK stack, extendable to other tools or platforms (not only Kafka!)

# Domains



# Domains

To sum up, it was necessary to consider these components:

- ❖ Kafka brokers
- ❖ Zookeeper nodes
- ❖ Producers
- ❖ Consumers
- ❖ KSQL
- ❖ Kafka Streams
- ❖ Kafka Connect (sinks and sources)
- ❖ Schema Registry
- ❖ Host (CPU, disks, network)
- ❖ JVM

# Domains

- ❖ Stream Processing
- ❖ Microservices + Cassandra
- ❖ Website Activity Tracking
- ❖ Metrics Collection and Monitoring
- ❖ Log Aggregation
- ❖ Real time analytics
- ❖ Capture and ingest data into Spark / Hadoop
- ❖ CRQS, replay, error recovery
- ❖ Guaranteed distributed commit log for in-memory computing

# Domains

we've defined three macro-categories:

- ❖ Status: all about the availability of the system — How much memory is available for each cluster node? Is broker up & running? How many clients are connected to Zookeeper?
- ❖ Performance: How efficiently a component is doing its work? For example, the most common performance metric is latency, which represents the time required to complete a unit of work.
- ❖ Error: each metric that catches the number of erroneous results usually expressed as a rate of errors per unit time or normalized by the throughput to yield errors per unit of work

# Metrics

The idea here was to define two sets:

- ❖ A shortlist where we put the two-three most important metrics for each selected entity — this was useful to build a proof-of-concept
- ❖ A complete list where we added all relevant metrics

# Metrics

For each metric, we try to get each of them:

- ❖ Metric description: What's the meaning of the metric?  
What's its critical value?
- ❖ Category: Is metric about status, error or performance?
- ❖ Entity: Which component of the architecture is being monitored?
- ❖ Access type: MBean, REST, Beat or derived metrics?
- ❖ Access method: The specific way to access to metric and get its value — this should be an MBean/Beat attribute or a REST endpoint

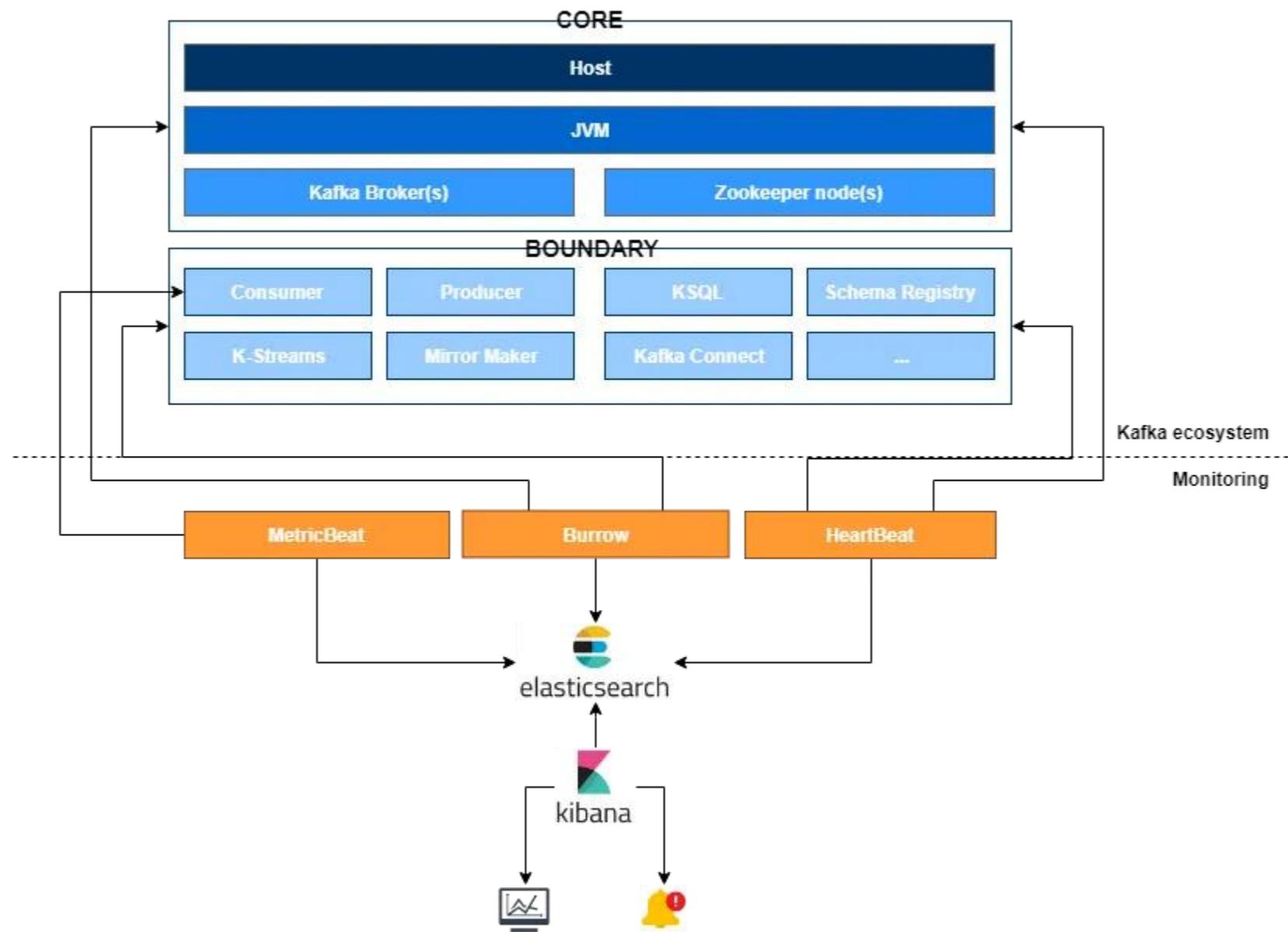
# Solution

Don't you know ELK? Essentially, it's a combination of four open-source products:

- ❖ **Elasticsearch (E)**: a distributed search and analytics engine built on Apache Lucene, useful for log analytics and search use cases
- ❖ **Logstash (L)**: a data ingestion tool that allows you to collect data from a variety of sources, transform it, and send it to your desired destination
- ❖ **Kibana (K)**: a data visualization and exploration tool for discovering logs and events
- ❖ **Beats**: a set of plugins that you install as agents on your servers to send specific types of operational data to Elasticsearch

# Solution

The following image summarizes the architecture we built:



# Monitoring

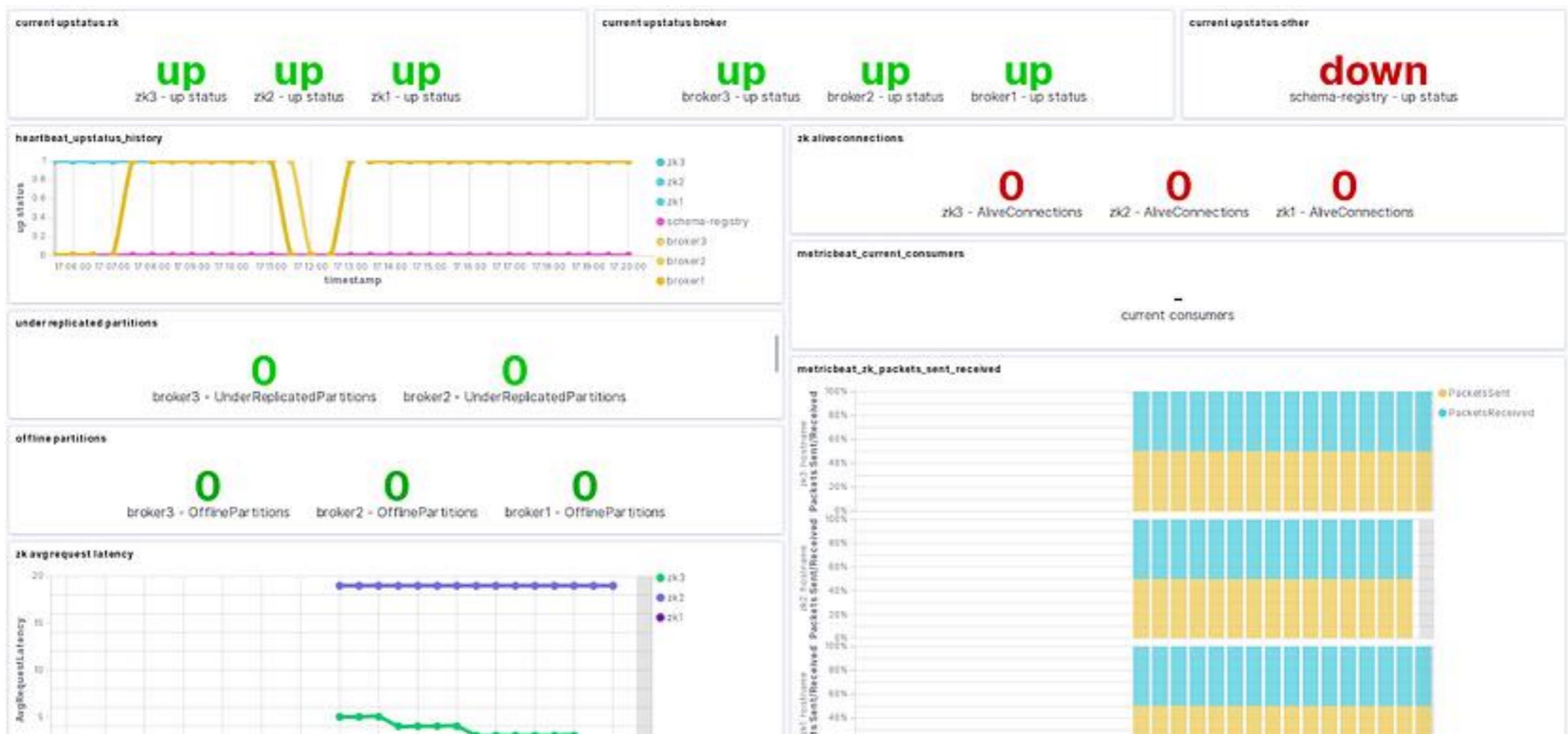
The infrastructure comprises ELK stack, some additional plugins, and open-source tools. In particular, we chose:

- MetricBeat to monitor servers by collecting metrics from the system about memory, disk, network, and CPU utilization
- Heartbeat for uptime monitoring. It executes periodic checks to verify whether the endpoint is up or down, then reports this information along with other useful metrics, to Elasticsearch

# Monitoring

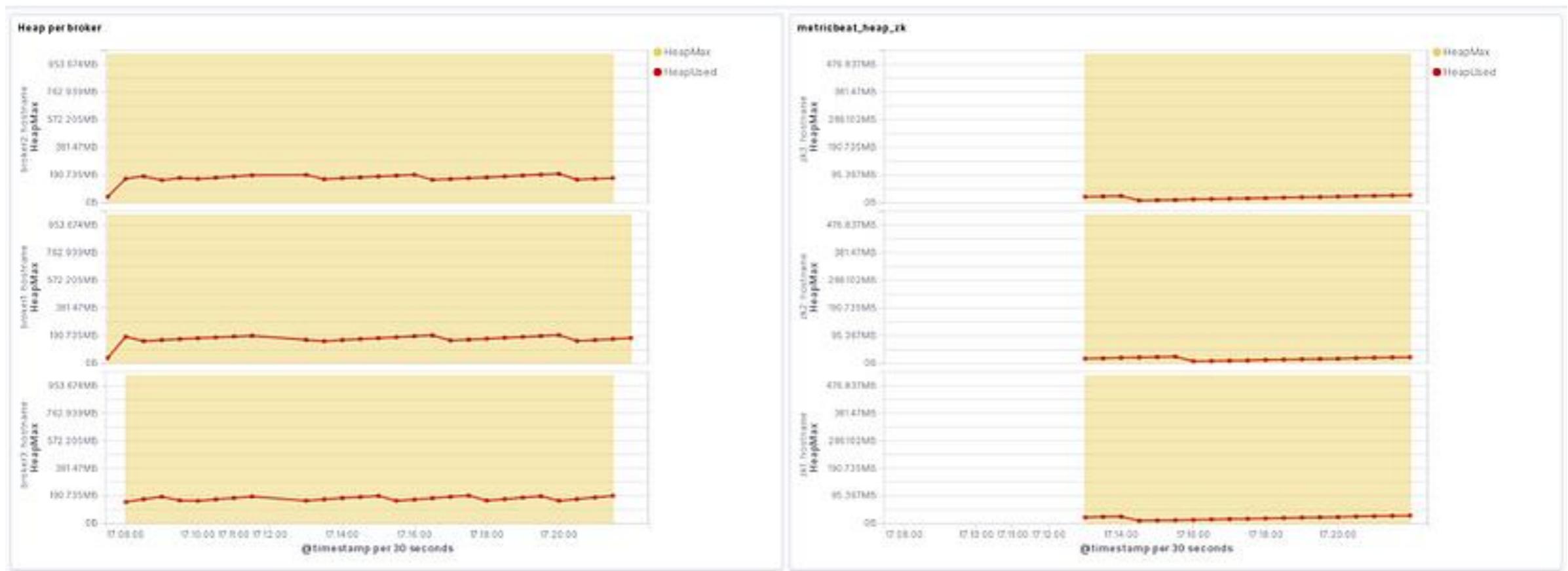
- ❖ Burrow to collect Kafka consumers' data. For example, it provides several HTTP request endpoints to get information about Kafka clusters and consumer groups, like current lag, offsets read and partition status. Normally, it's very hard to get these though Mbean, so Burrow simplifies our life, For more details, check out its Github repository!
- ❖ Elastalert for alerting on anomalies, spikes, or other patterns of interest from data in Elasticsearch. It is a framework written in Python, and it works by combining two types of components, rule types and alerts.
- ❖ Elasticsearch is periodically queried, and the data is passed to the rule type, which determines when a match is found

# cluster high-level monitoring



# Memory usage monitoring

- In the following example we deep dive into Kafka broker heap monitoring



# Memory usage monitoring



**K-Monitor** APP 3:02 PM

K-Connect is down

K-Connect is down

At least 3 events occurred between 2019-06-05 12:01 UTC and 2019-06-05 13:01 UTC

@timestamp: 2019-06-05T13:01:11.351Z

\_id: zc26J2sBl-uFQdmLp1li

\_index: heartbeat-7.0.1-2019.06.05-000001

\_type: \_doc

ecs: {

Show more

Kafka Broker is up

Kafka Broker is up

At least 3 events occurred between 2019-06-05 12:00 UTC and 2019-06-05 13:00 UTC

@timestamp: 2019-06-05T13:00:55Z

\_id: fc26J2sBl-uFQdmLZ1JL

\_index: heartbeat-7.0.1-2019.06.05-000001

\_type: \_doc

ecs: {

Show more

# Conclusion

- ❖ We all know monitoring Kafka cluster is quite challenging.
- ❖ There is no “right recipe” here.
- ❖ Nowadays the trend is to monitor a large number of metrics and to define many alerts, but the monitoring process must be simple and effective.

# Identifying performance bottlenecks

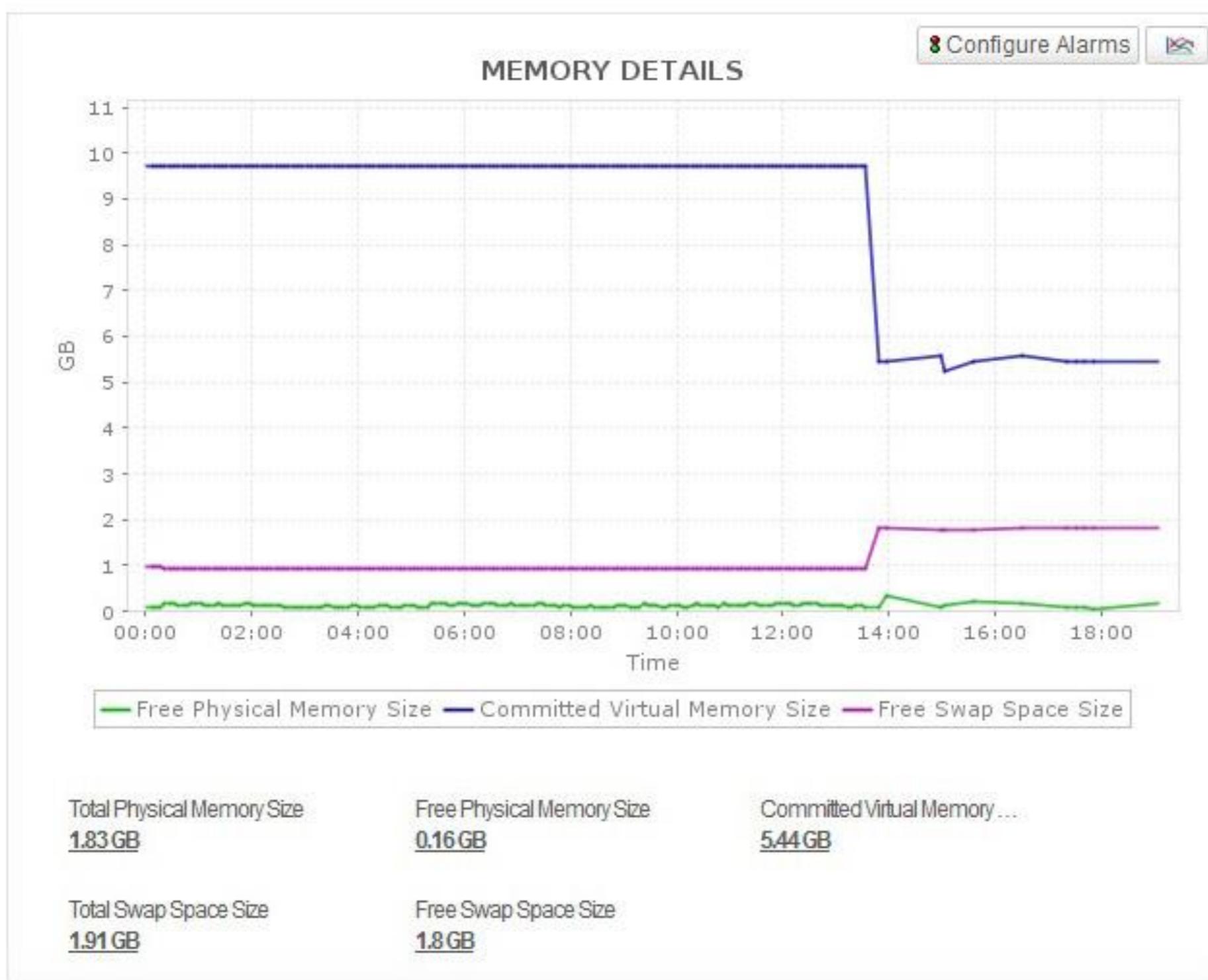
# Kafka Monitoring

- ❖ Gain insights with your Kafka clusters before they cause issues for your business.
- ❖ Monitor your Kafka ecosystem efficiently, get alerted to problems, and swiftly identify the root cause of production issues with Applications Manager's Kafka monitoring.
- ❖ Go beyond open source Kafka monitoring tools, streamline operations, and optimize the way your engineering teams use Kafka.

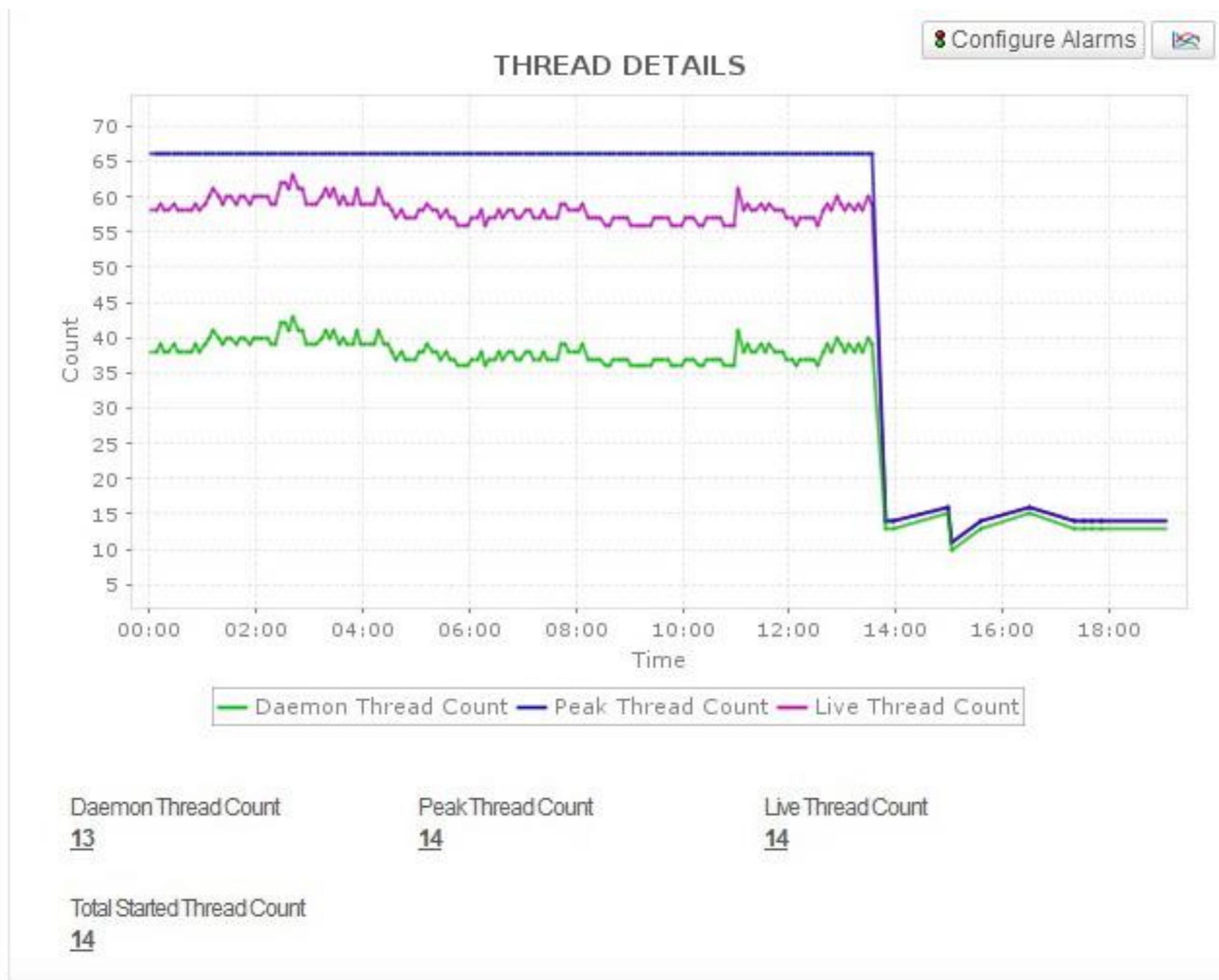
# Kafka monitoring dashboards

- ❖ In large enterprise environments, the data provided by Kafka can be both massive in quantity and uncovering the metrics is often a difficult task.
- ❖ A lot of businesses use Kafka in application development and delivery, while some use it to perform queuing and to replicate data between data centers (i.e DC-DC replication).
- ❖ If the health of the Kafka server is down, the development and queuing process may get affected and it causes a lag in the delivery of apps.

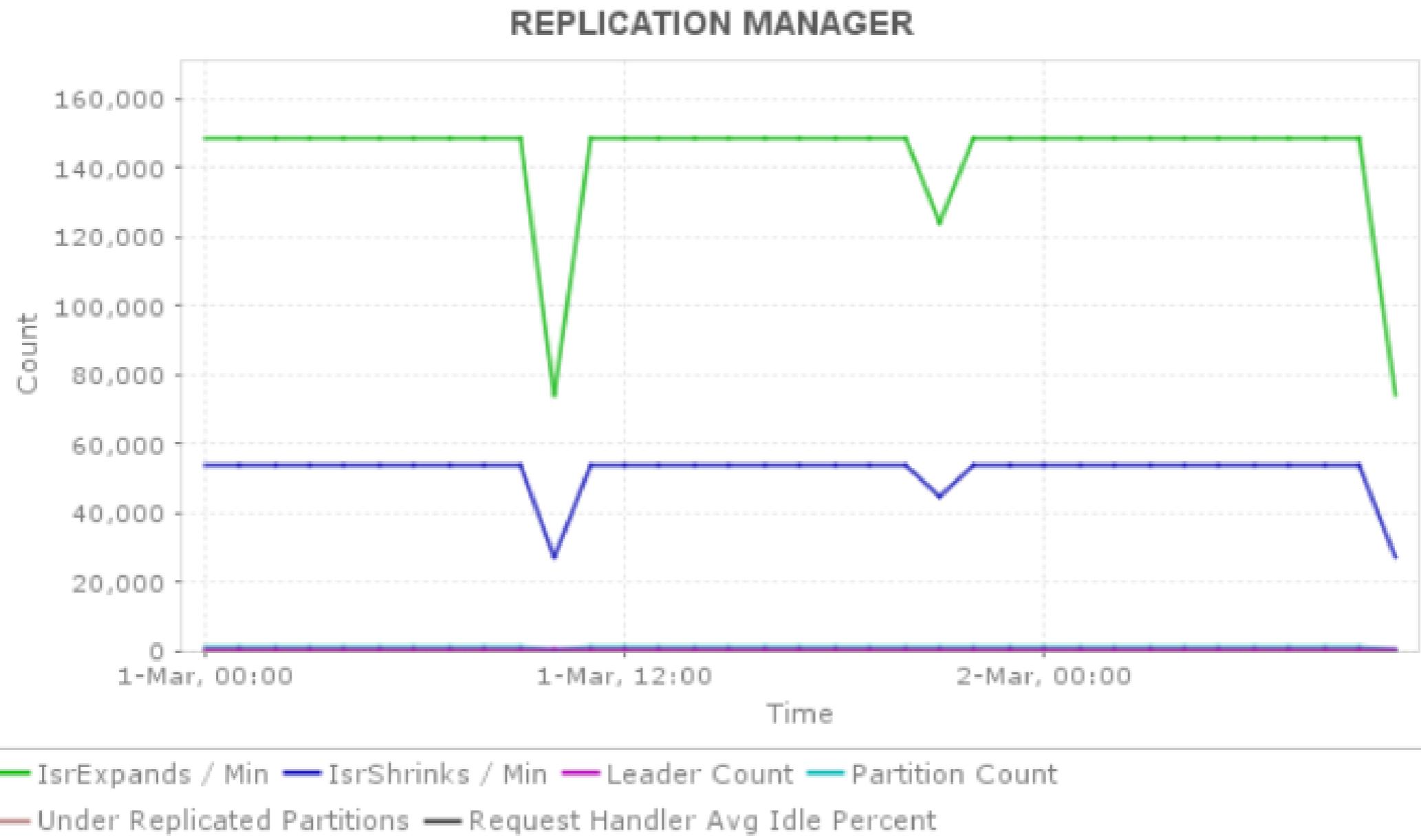
# Avoid overloading of resources



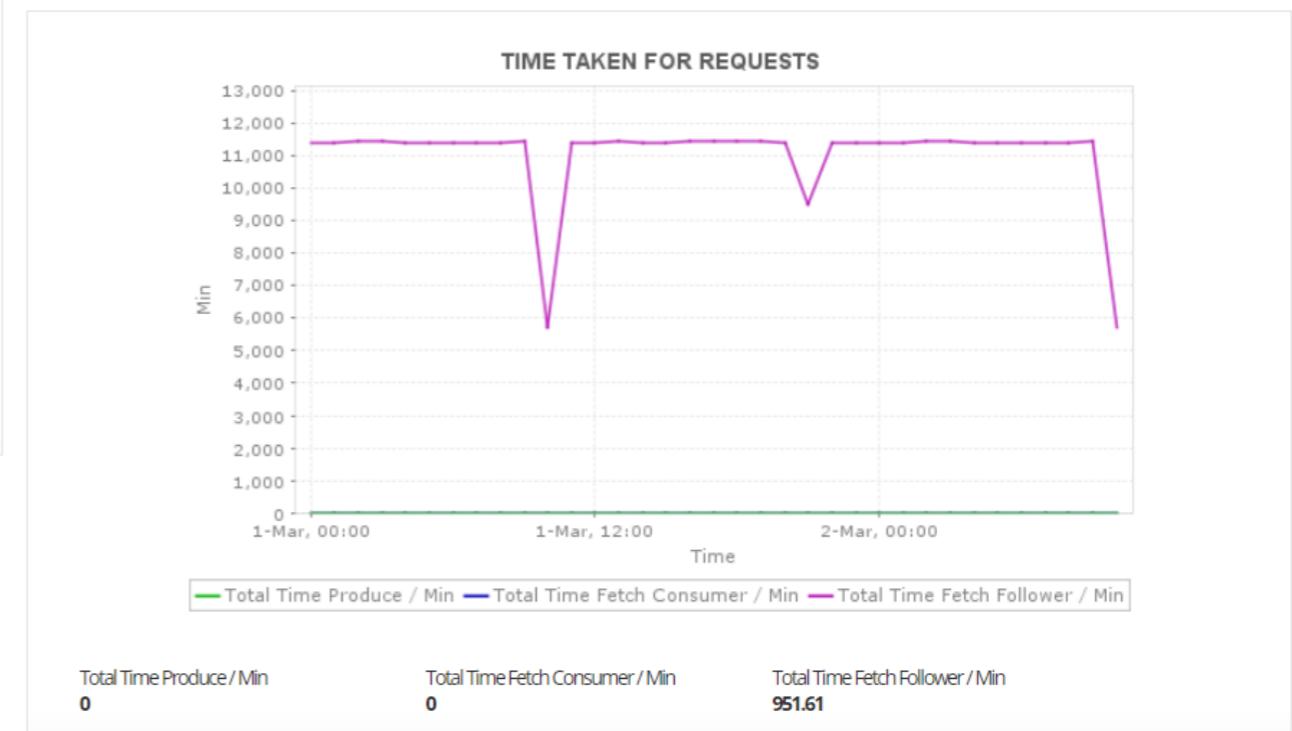
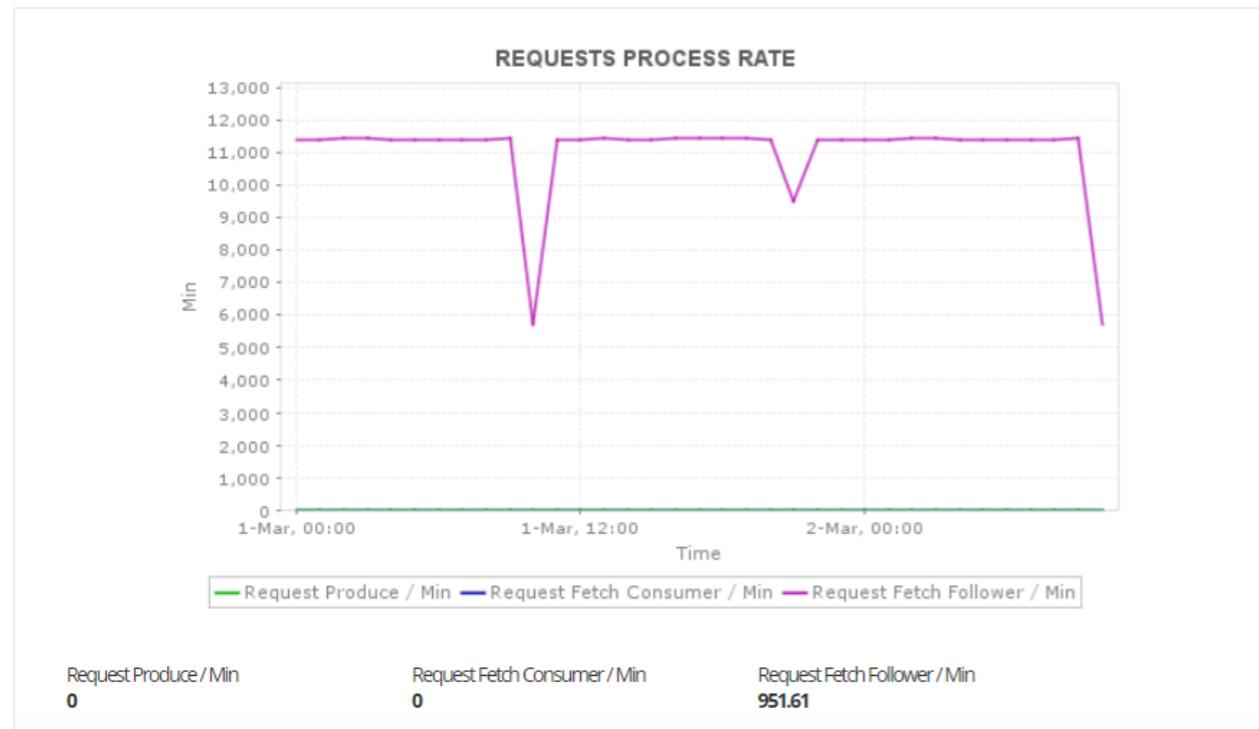
# Keep tabs on threads and JVM usage



# Gain insights into broker, controller, and replication statistics



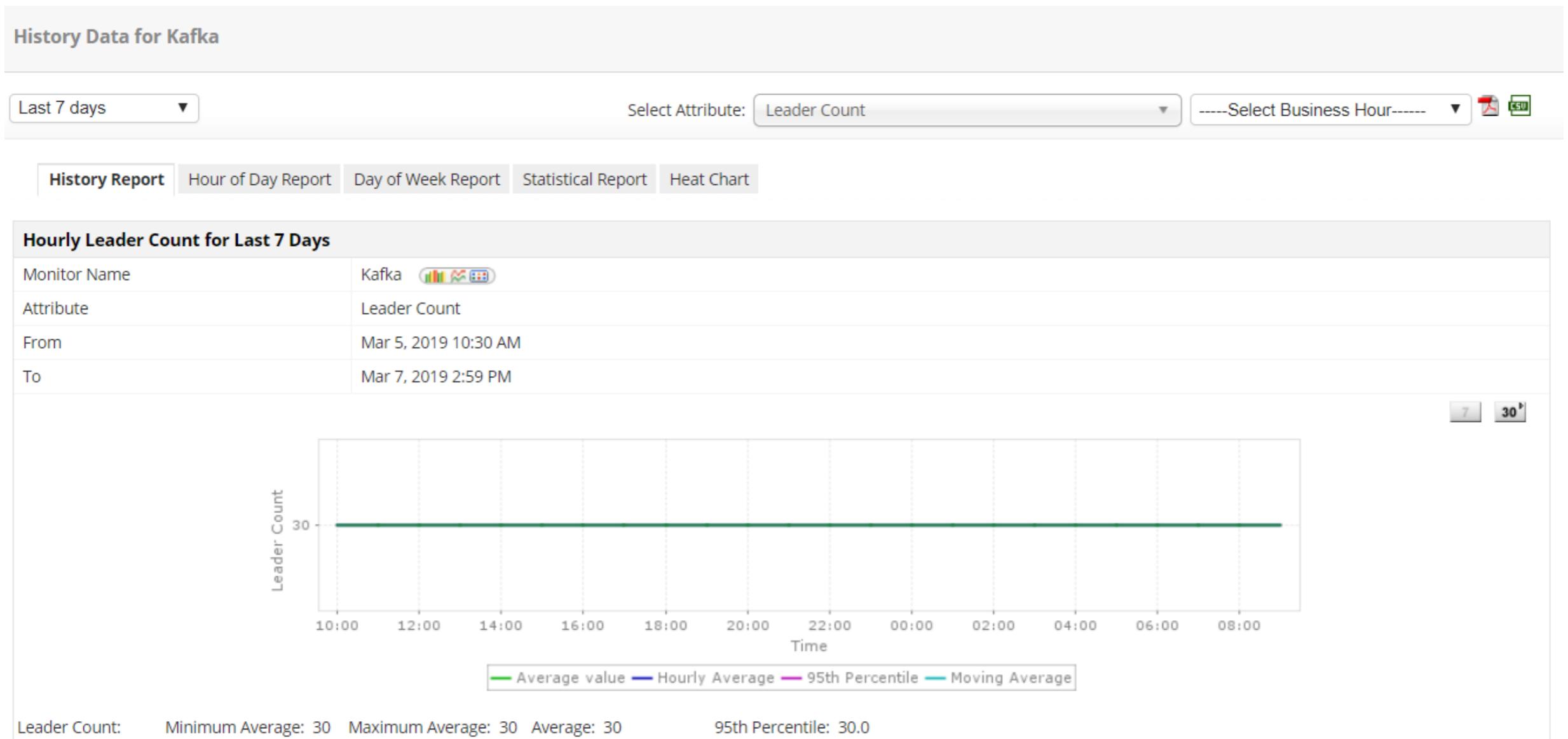
# Monitor network and topic details



# Get alerted about Kafka performance problems in real time

- ❖ Set up alerts quickly, Choose from threshold based alerts or alerts based on anomaly detection.
- ❖ Trigger alerts in real time when any Kafka component goes down or for problems such as consumer lag, offline partitions, and much more.
- ❖ Get notified in the medium of your choice (email, text or Slack channels).
- ❖ Propagate alerts to external incident management tools such as ServiceNow and ManageEngine ServiceDesk Plus.
- ❖ Respond quickly to incidents by automating corrective actions. Automate tasks using web hooks to start external actions.

# Gain insights into trends and predict resource growth



# Troubleshooting common kafka issues

# Troubleshooting kafka issues

- ❖ Apache Kafka is one of the most popularly used open-source distributed event streaming platforms.
- ❖ Its use cases range from enabling mission-critical apps to building and maintaining high-performance data pipelines.
- ❖ If you consider whether to use Apache Kafka for your future projects, you should know all about the pros and cons of using it.

# In Sync Replica Alerts

- ❖ Kafka In Sync Replica Alert tells you that some of the topics are under-replicated.
- ❖ The data is simply not being replicated to brokers, These alerts indicate a potentially serious problem because the probability of data being lost becomes higher.
- ❖ It can happen entirely unexpectedly, even if you do nothing on your side.
- ❖ It usually takes place when downlevel clients affect the volume of data.
- ❖ A spike in data volume causes the Kafka broker to back up message conversion.
- ❖ However, the problem has to be addressed as soon as possible.

# Kafka Liveness Check Problems and Automation

- ❖ The Kafka liveness check problems can quickly occur if the host where the liveness check is running cannot reach the host where the broker is running.
- ❖ If this happens, the broker will keep on restarting. Meanwhile, all the downlevel clients won't be able to run their apps. It can become a real nuance if you want to automate some of your tasks on Kafka.
- ❖ Why? Because you need to enable liveness check to streamline automation and make sure that the broker's client-serving port is open.

# New Brokers Can Impact the Performance

- ❖ Staging a new cluster and installing the broker software on Apache Kafka is straightforward.
- ❖ Adding new brokers should not cause any problems, right? Pushing a new Kafka broker into production can potentially impact the performance and cause serious latency and missing file problems.
- ❖ The broker can work properly before the partition reassign process is completed.
- ❖ Devs usually forget about it and use the default commands from the documentation.

# Questionable Long-Term Storage Solution

- ❖ If you are working with large sets of data, using Apache Kafka to store it might cause you several problems.
- ❖ The major problem comes from Kafka storing redundant copies of data.
- ❖ It can affect the performance, but, more importantly, it can significantly increase your storage costs.
- ❖ The best solution would be to use Kafka only for storing data for a brief period and migrate data to a relational or non-relational database, depending on your specific requirements.

# Finding Perfect Data Retention Settings

- ❖ While we are discussing long-term storage solution problems, let's point out one additional issue related to it.
- ❖ The downstream clients often have completely unpredictable data request patterns.
- ❖ This makes finding the perfect and most optimal data retention settings somewhat of a problem.
- ❖ Kafka stores messages in topics, This data can take up significant disk space on your brokers.

# Overly Complex Data Transformations on-Fly

- ❖ Using Apache Kafka on big data integration and migration projects can become too complex.
- ❖ How come? Kafka was built to streamline delivering messages, and the platform excels at it.
- ❖ However, you will run into some problems if you want to transform data on-fly.
- ❖ Even with Kafka Stream API, you will have to spend days building complex data pipelines and managing the interaction between data producers and data consumers.

# Upscaling and Topic Rebalancing

- ❖ The volume of your data streams can go in both directions.
- ❖ This is why it is crucial to choose a distributed messaging platform easy to scale up and down.
- ❖ With Kafka, this is a problem because you need to balance things manually to reduce resource bottlenecks.
- ❖ You will have to do it every time a major change in the data stream occurs.
- ❖ And do it both via partition leadership balancing and Kafka reassign partition script.

# MirrorMaker Doesn't Replicate the Topic Offsets

- ❖ MirrorMaker is one of Kafka's features that allows you to make copies of your clusters.
- ❖ This would be a great disaster recovery plan if it weren't for one downside.
- ❖ MirrorMaker doesn't replicate the topic offsets between the clusters.
- ❖ You will have to create unique keys in messages to overcome this problem which can become a daunting task when you are working at scale.

# Not All Messaging Paradigms Are Included

- ❖ While Apache Kafka comes with many messaging paradigms, some are still missing.
- ❖ This can turn into a real problem if you need to extend your infrastructure use case.
- ❖ It limits the Kafka capability to support building complex data pipelines.
- ❖ Two major messaging paradigms not supported in Kafka are point-to-point queues and request/reply queues

# Changing Messages Reduces Performance

- ❖ If you want to use Apache Kafka to deliver messages as they are, you will have no issues performance-wise.
- ❖ However, the problem occurs once you wish to modify the messages before you deliver them.
- ❖ Manipulating data on the fly is possible with Kafka, but the system it uses has some limits.
- ❖ It uses system calls to do it, and modifying messages makes the entire platform perform significantly slower.