

Confluent Developer Skills for Building Apache Kafka®

Version 6.0.0-v1.1.0

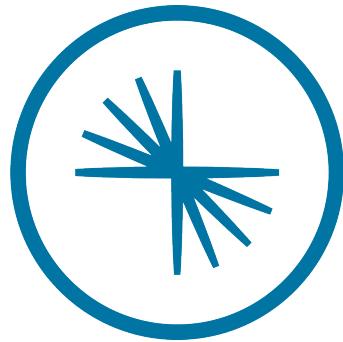


CONFLUENT

Table of Contents

01 Introduction	1
02 Fundamentals of Apache Kafka	9
03 Producing Messages to Kafka	37
04 Consuming Messages from Kafka	73
05 Schema Management In Kafka	113
06 Stream Processing with Kafka Streams	167
07 Data Pipelines with Kafka Connect	216
08 Event Streaming Apps with ksqlDB	274
09 Design Decisions	297
10 Confluent Cloud	359
11 Conclusion	376

01 Introduction



CONFLUENT

Copyright & Trademarks

Copyright © Confluent, Inc. 2014-2021. [Privacy Policy](#) | [Terms & Conditions](#).
Apache, Apache Kafka, Kafka and the Kafka logo are trademarks of the
[Apache Software Foundation](#)

Agenda



1. Introduction ... ←
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

Prerequisite

This course requires a working knowledge of the Apache Kafka architecture.

New to Kafka? Need a refresher?

Sign up for free ***Confluent Fundamentals for Apache Kafka*** course at
<https://confluent.io/training>

Course Objectives

Upon completion of this course, you should be able to:

- Write Producers and Consumers to send data to and read data from Apache Kafka
- Create schemas, describe schema evolution, and integrate with Confluent Schema Registry
- Integrate Kafka with external systems using Kafka Connect
- Write streaming applications with Kafka Streams & ksqlDB
- Describe common issues faced by Kafka developers and some ways to troubleshoot them
- Make design decisions about acks, keys, partitions, batching, replication, and retention policies

Throughout the course, Hands-On Exercises will reinforce the topics being discussed

Class Logistics



- Start and end times
- Can I come in early/stay late?
- Breaks
- Lunch
- Restrooms
- Wi-Fi and other information
- Emergency procedures



No video & recording please!

How to get the courseware?



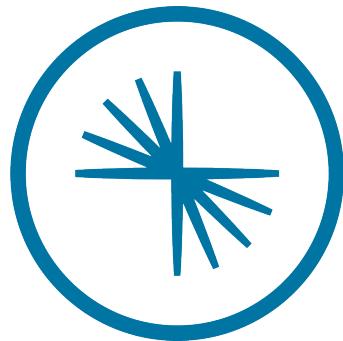
1. Register at **training.confluent.io/signup**
2. Verify your email
3. Log in to **training.confluent.io/login** and enter your **license activation key**
4. Go to **My Learning** dashboard and select your course

Introductions



- About you:
 - What is your experience with Kafka?
 - What is your name, your company, and your role?
 - What are some other distributed systems you like to work with?
 - What technology most excited you when you were a kid or early in your career?
- About your instructor

02 Fundamentals of Apache Kafka



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka ... ←
3. Kafka Producers
4. Kafka Consumers
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

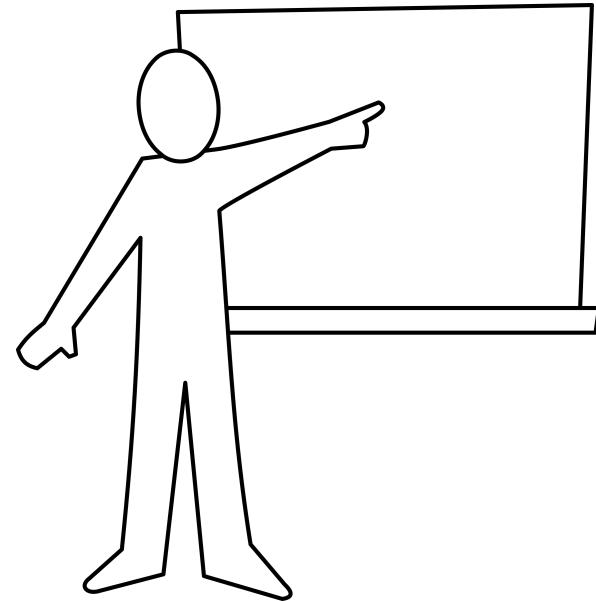
Learning Objectives



After this module you will be able to:

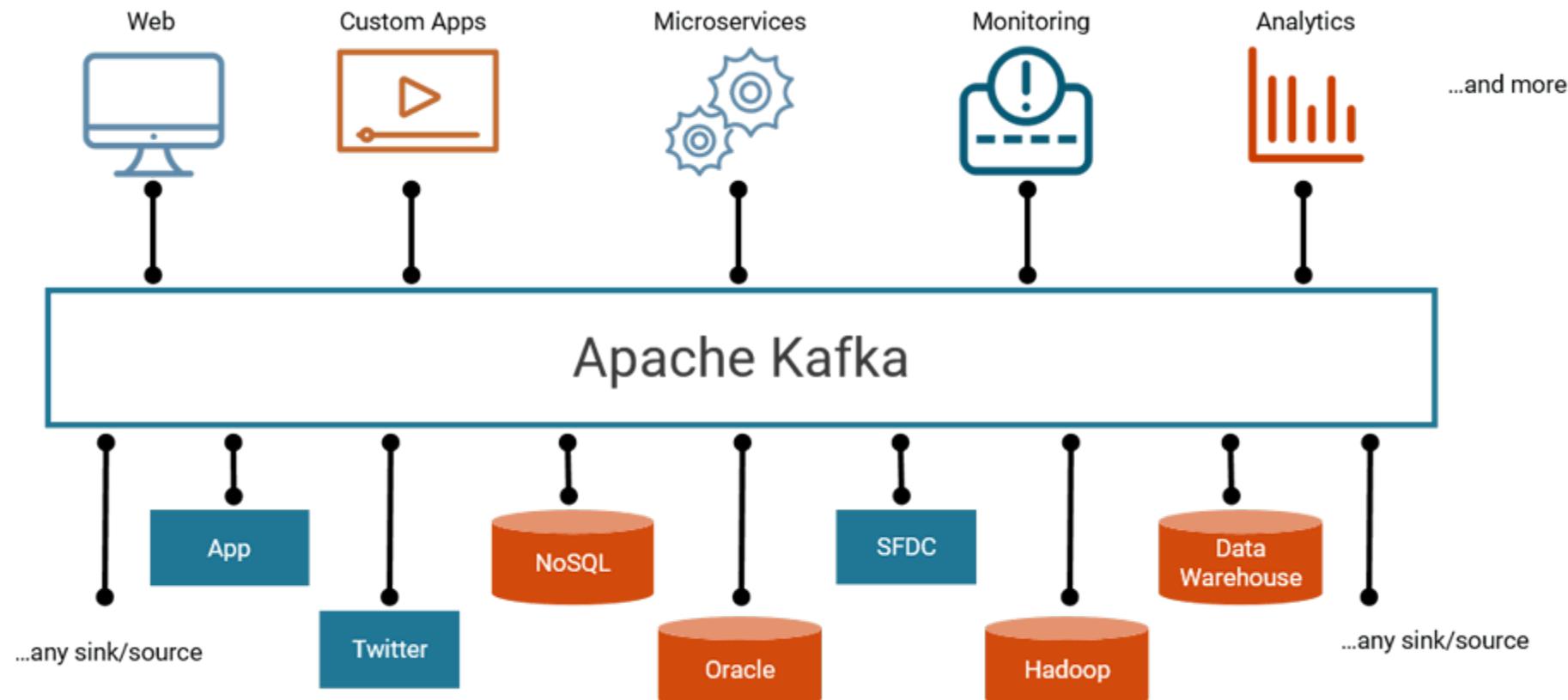
- explain the value of a **Distributed Event Streaming Platform**
- explain how the “log” abstraction enables a distributed event streaming platform
- explain the basic concepts of:
 - Brokers, Topics, Partitions, and Segments
 - Records (a.k.a. Messages, Events)
 - Retention Policies
 - Producers, Consumers, and Serialization
 - Replication

Module Map

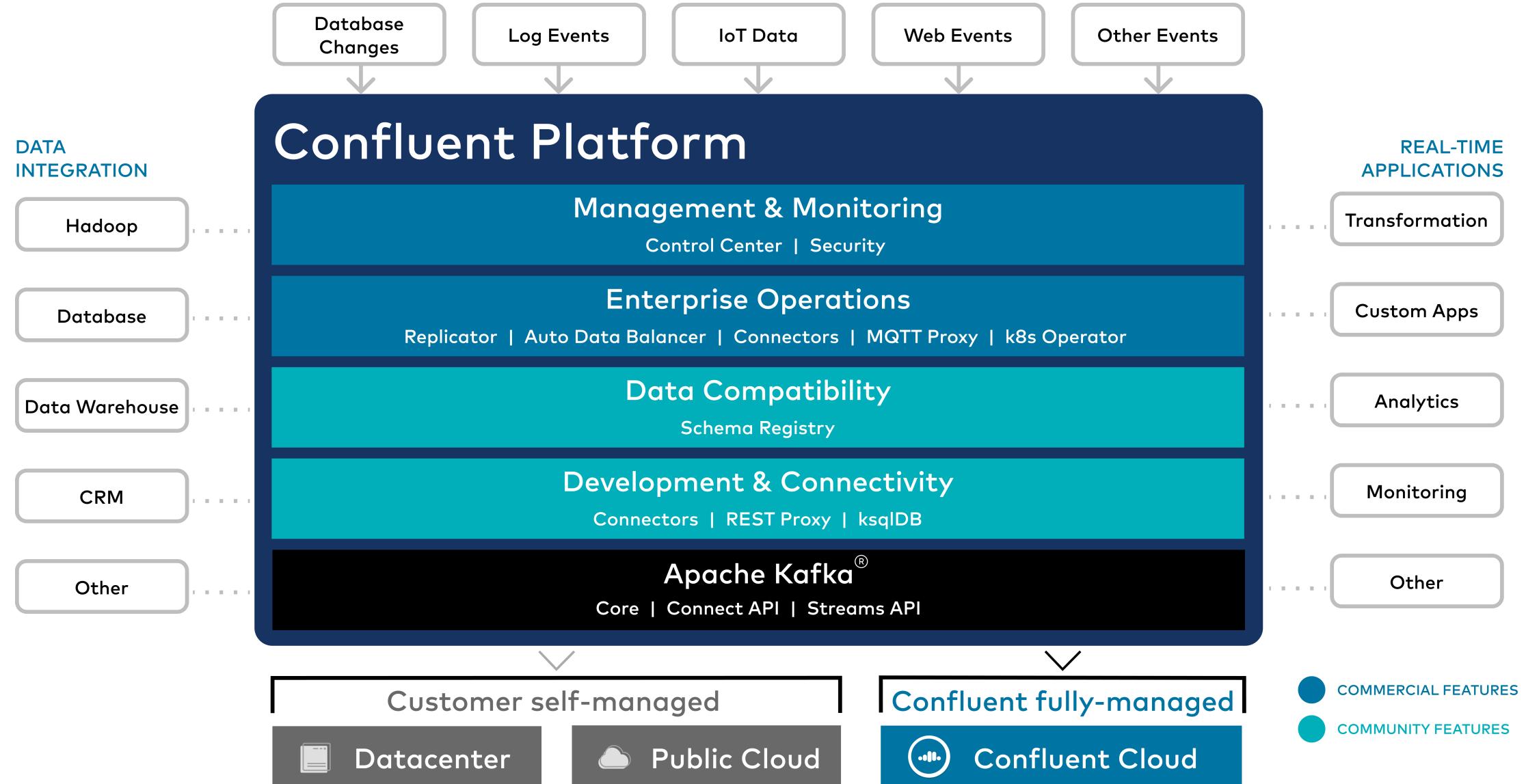


- Event Streaming Platform ... ←
- Event Streaming Architecture
- Log Retention
- Replication
- 🌐 Hands-on Lab

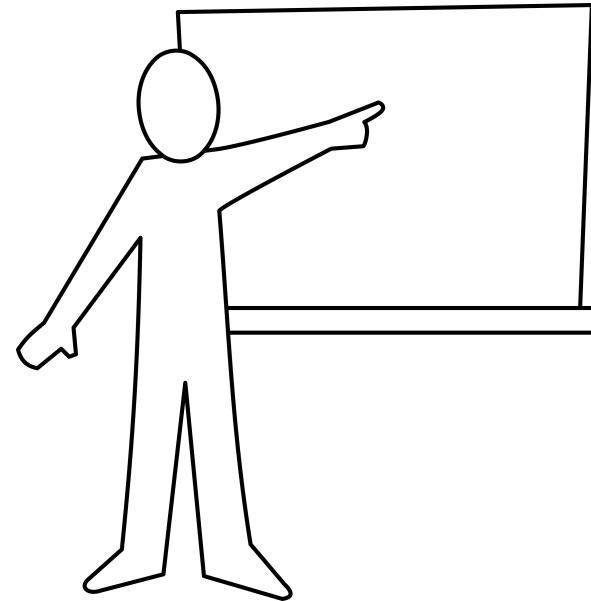
The Streaming Platform



Confluent Platform, Built on Apache Kafka

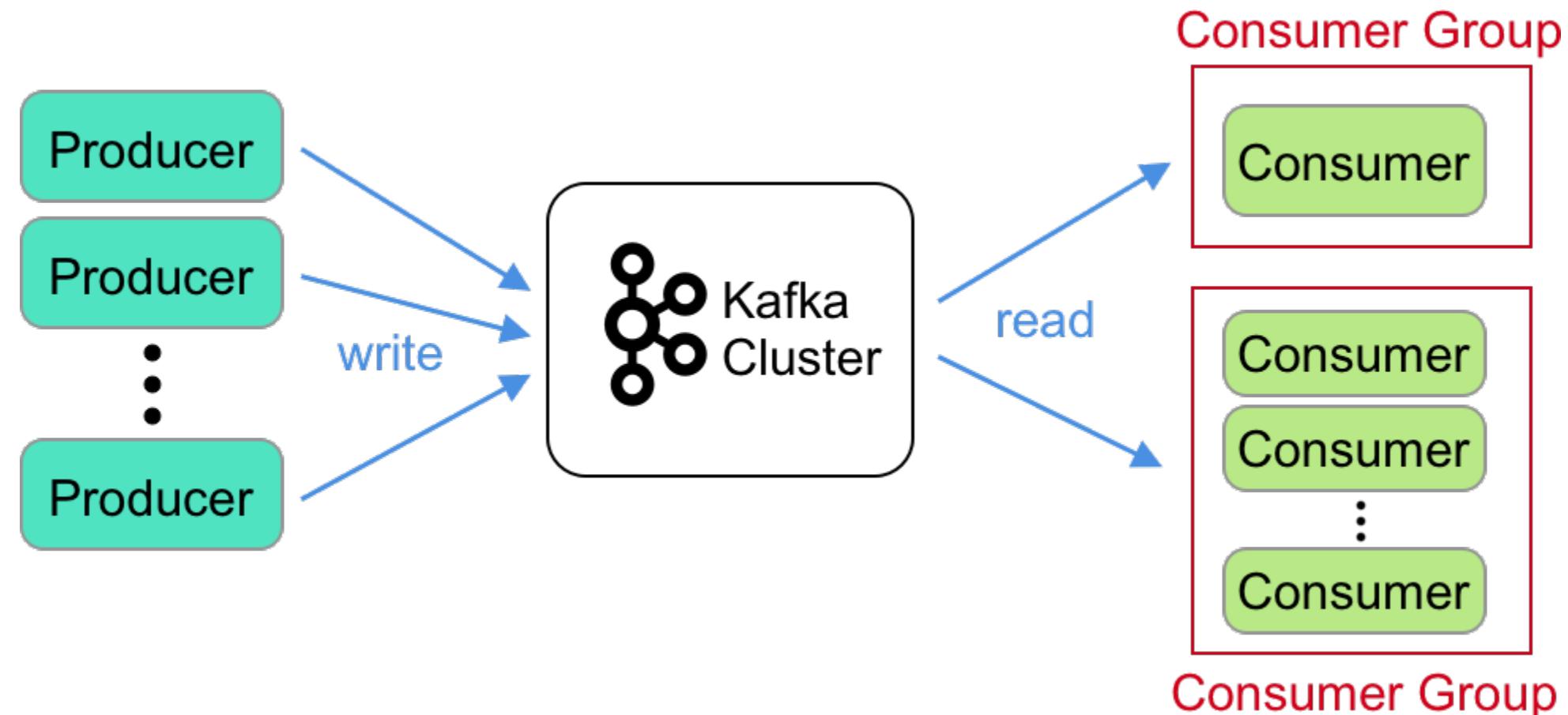


Module Map

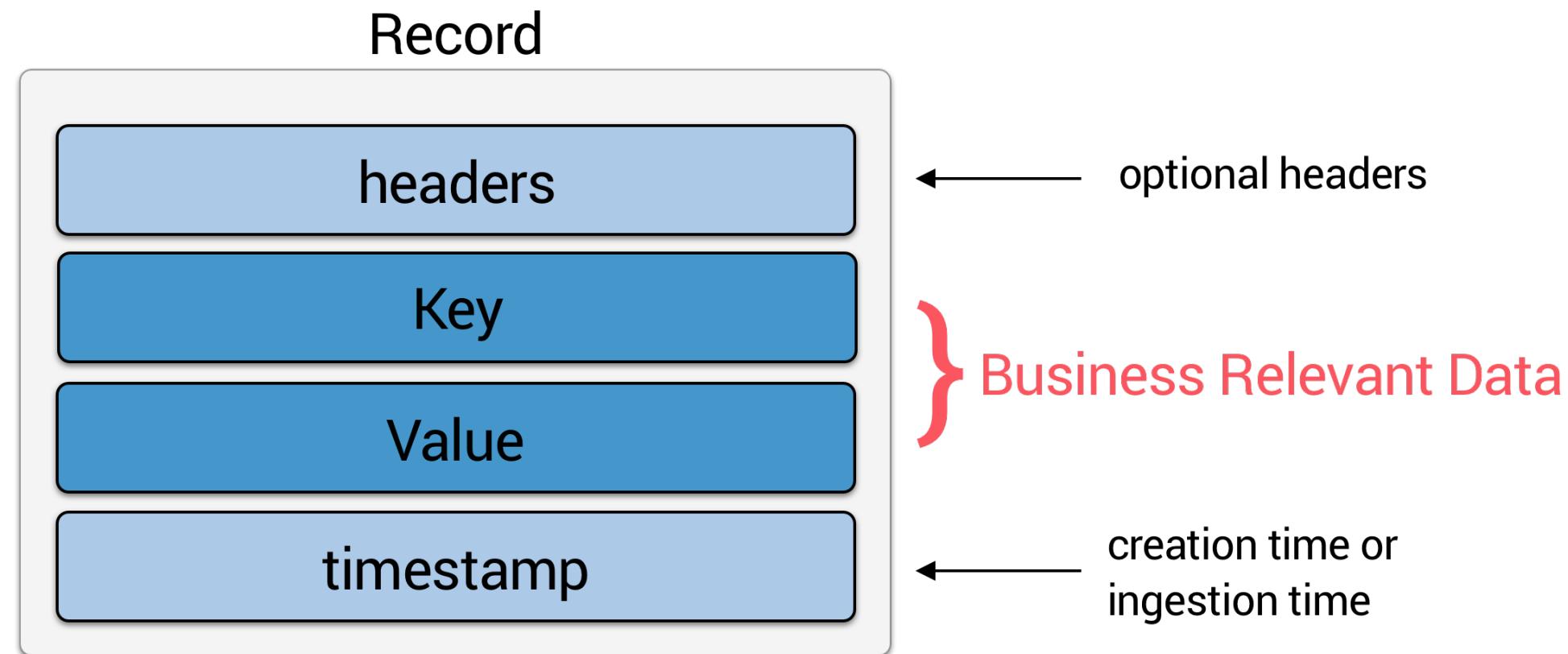


- Event Streaming Platform
- Event Streaming Architecture ... ←
- Log Retention
- Replication
- 🌐 Hands-on Lab

Kafka Clients - Producers & Consumers



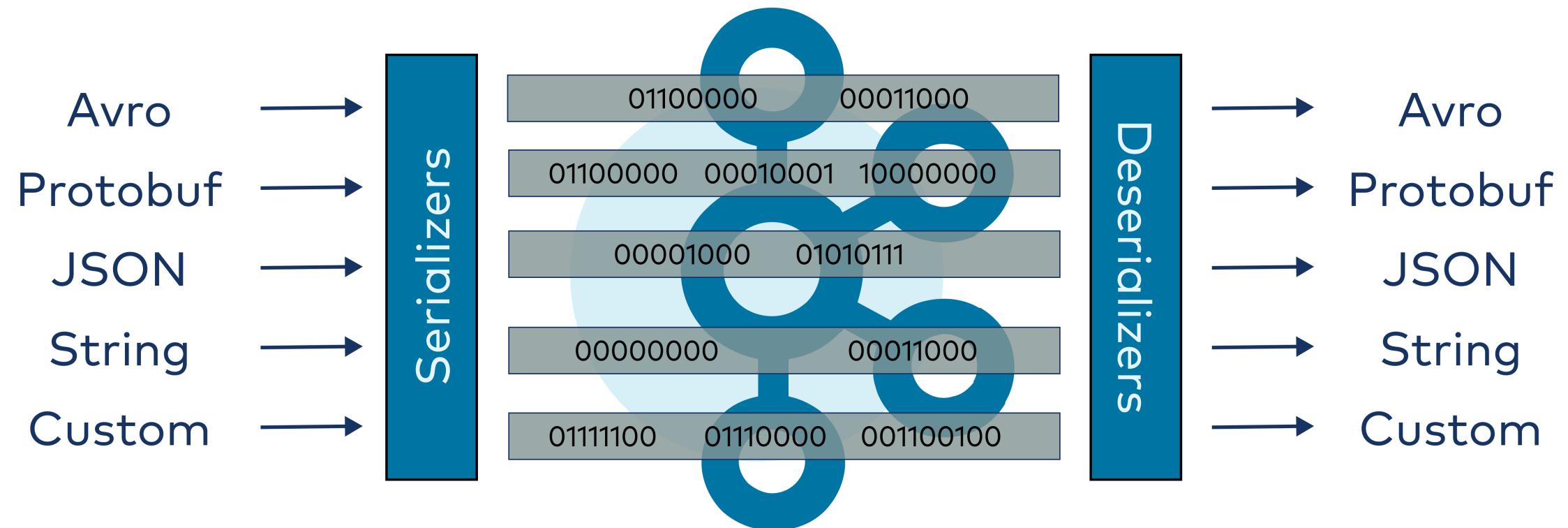
The Record—The Atomic Unit of Kafka



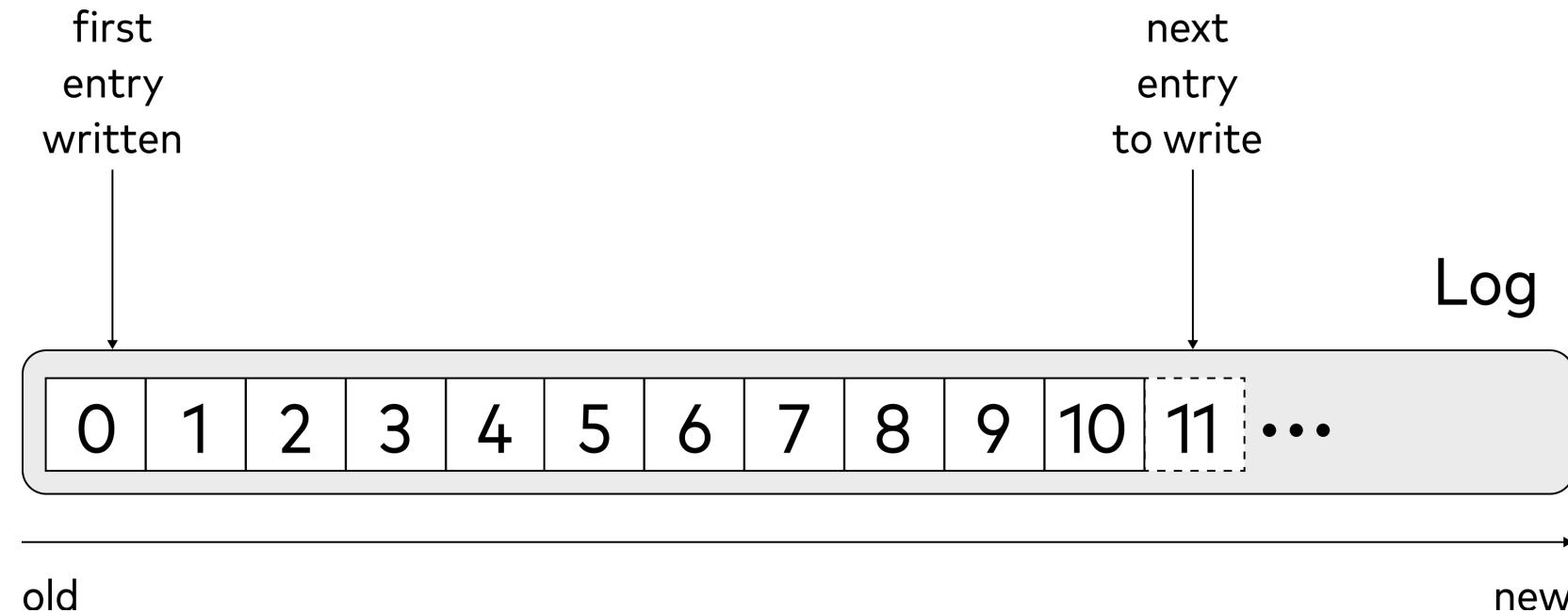
Kafka Records are also known as "Messages" or "Events"

Serialization

- Kafka stores byte arrays

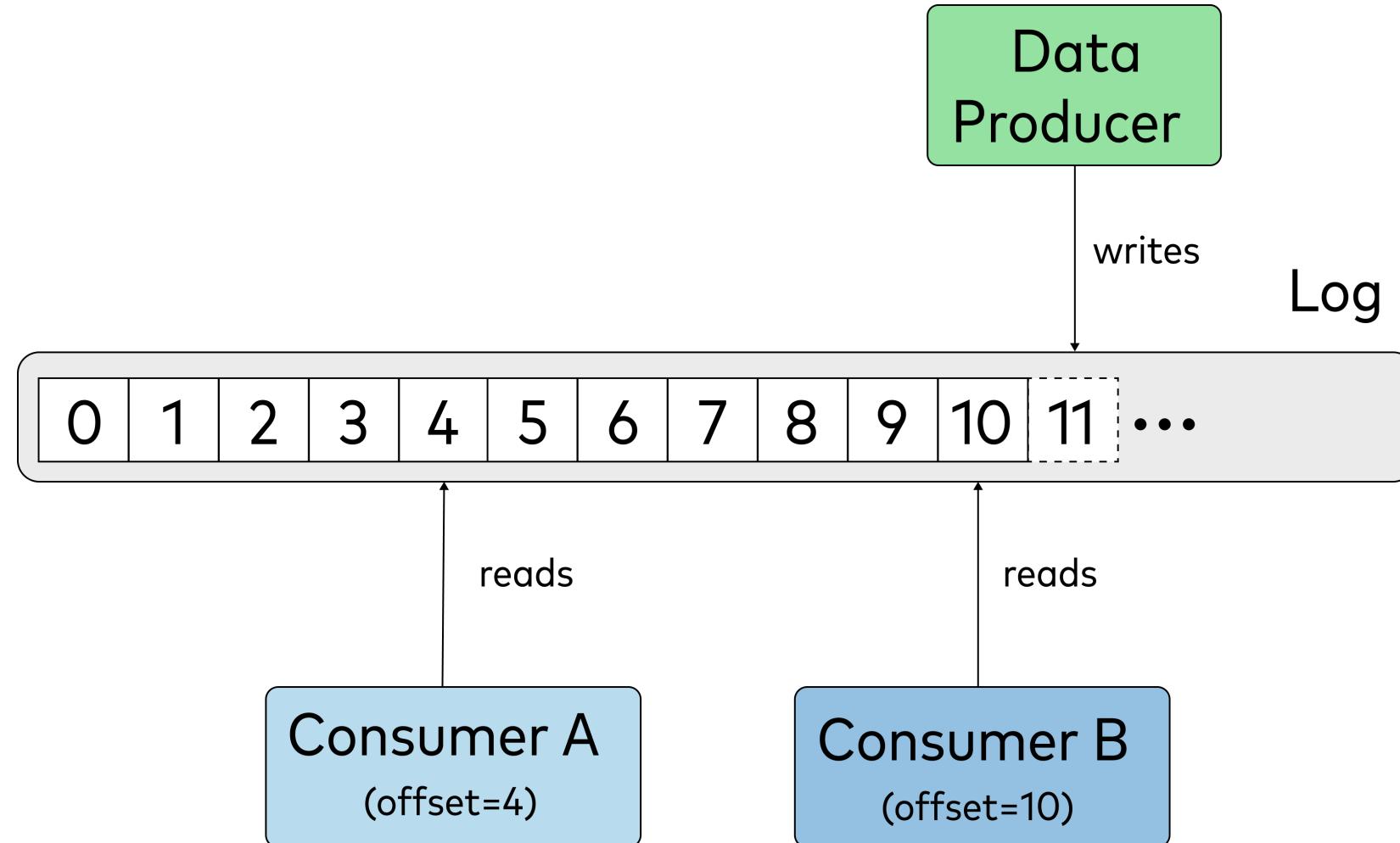


The Kafka Commit Log

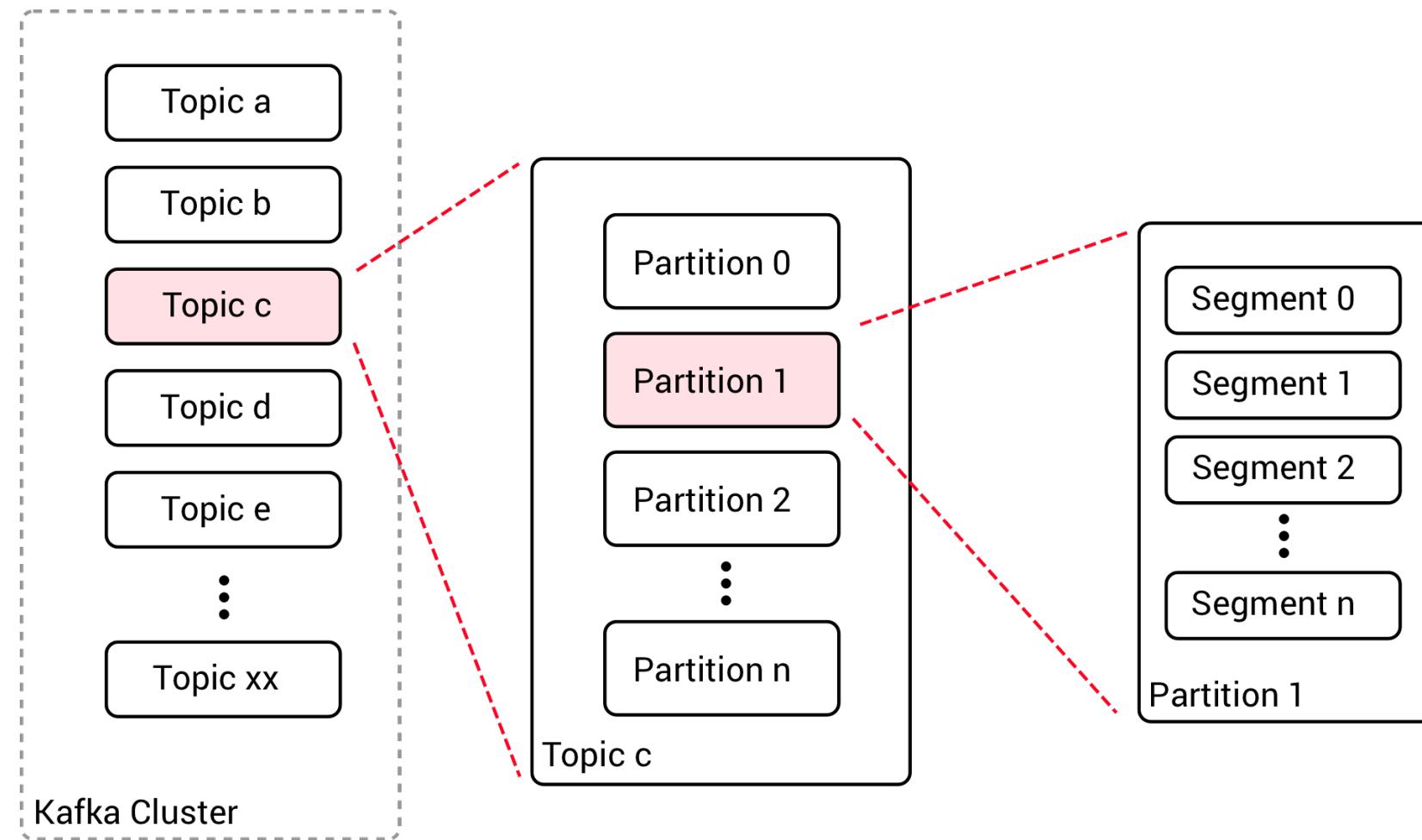


A position in the log is called an "offset." Here we see offsets 0 through 11.

Decoupling Data Producers from Data Consumers

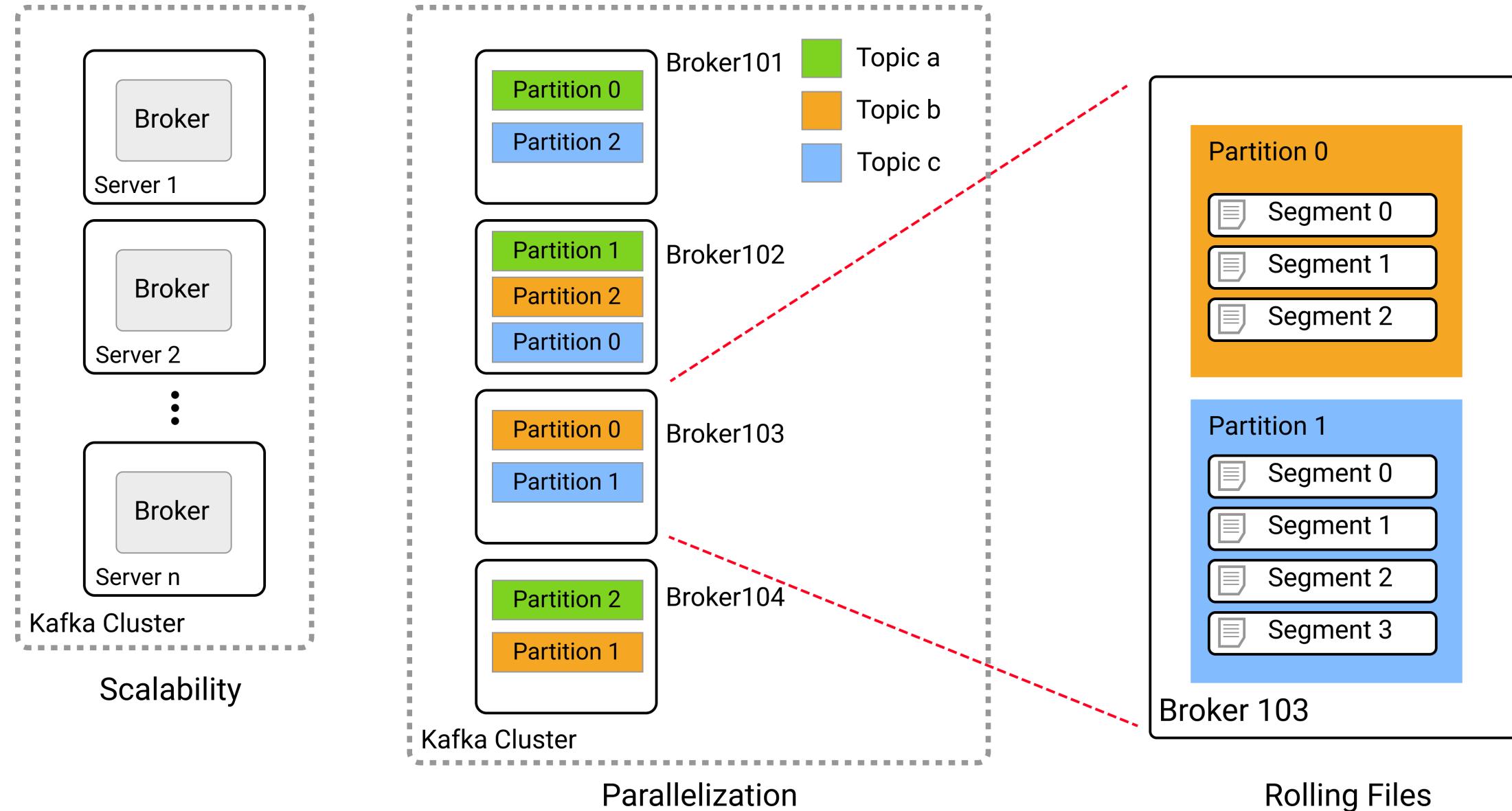


Kafka—Logical View of Topics, Partitions, and Segments

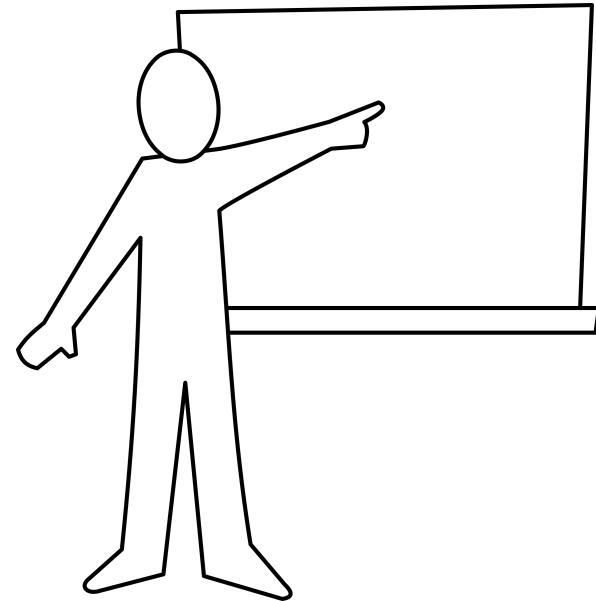


Each partition is a commit log. We often use "partition" and "log" interchangeably.

Kafka—Physical View of Topics, Partitions and Segments



Module Map



- Event Streaming Platform
- Event Streaming Architecture
- Log Retention ... ←
- Replication
- 🌐 Hands-on Lab

Managing Log File Growth



- We don't want logs to grow forever!
- `cleanup.policy`
 - `delete`
 - `compact`
 - **both:** `delete,compact`

The Delete Retention Policy

Delete segments when they get **too old**:

- **Too old** means newest message is older than `retention.ms`
 - Default: 7 days

Delete oldest segments when **partition gets too large**:

- `retention.bytes`
 - Default: -1 (unlimited)

The **Compact** Retention Policy

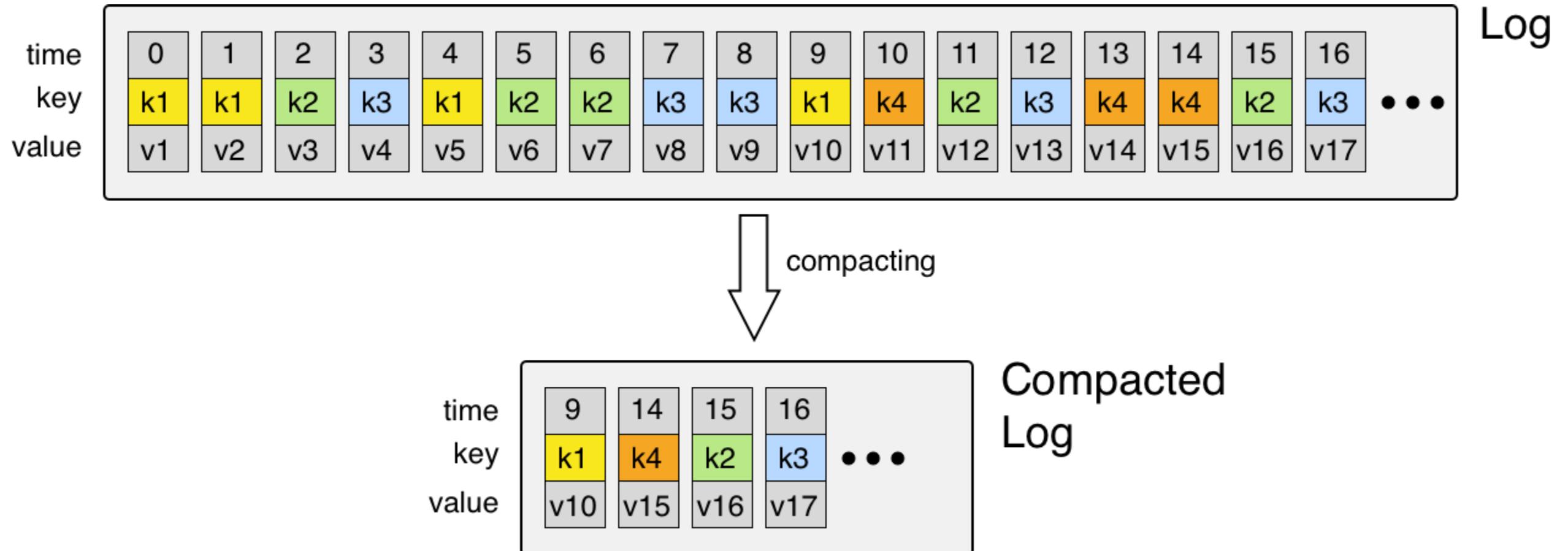
Keep only the **freshest** value of each key

Use cases:

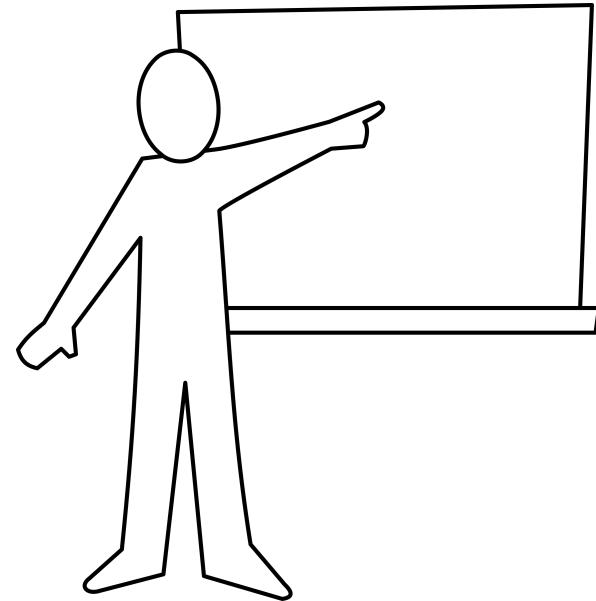
- **Real-time table lookups and joins during stream processing**
- Recovering latest state after failure of an in-memory application
- Database change capture



Log Compaction

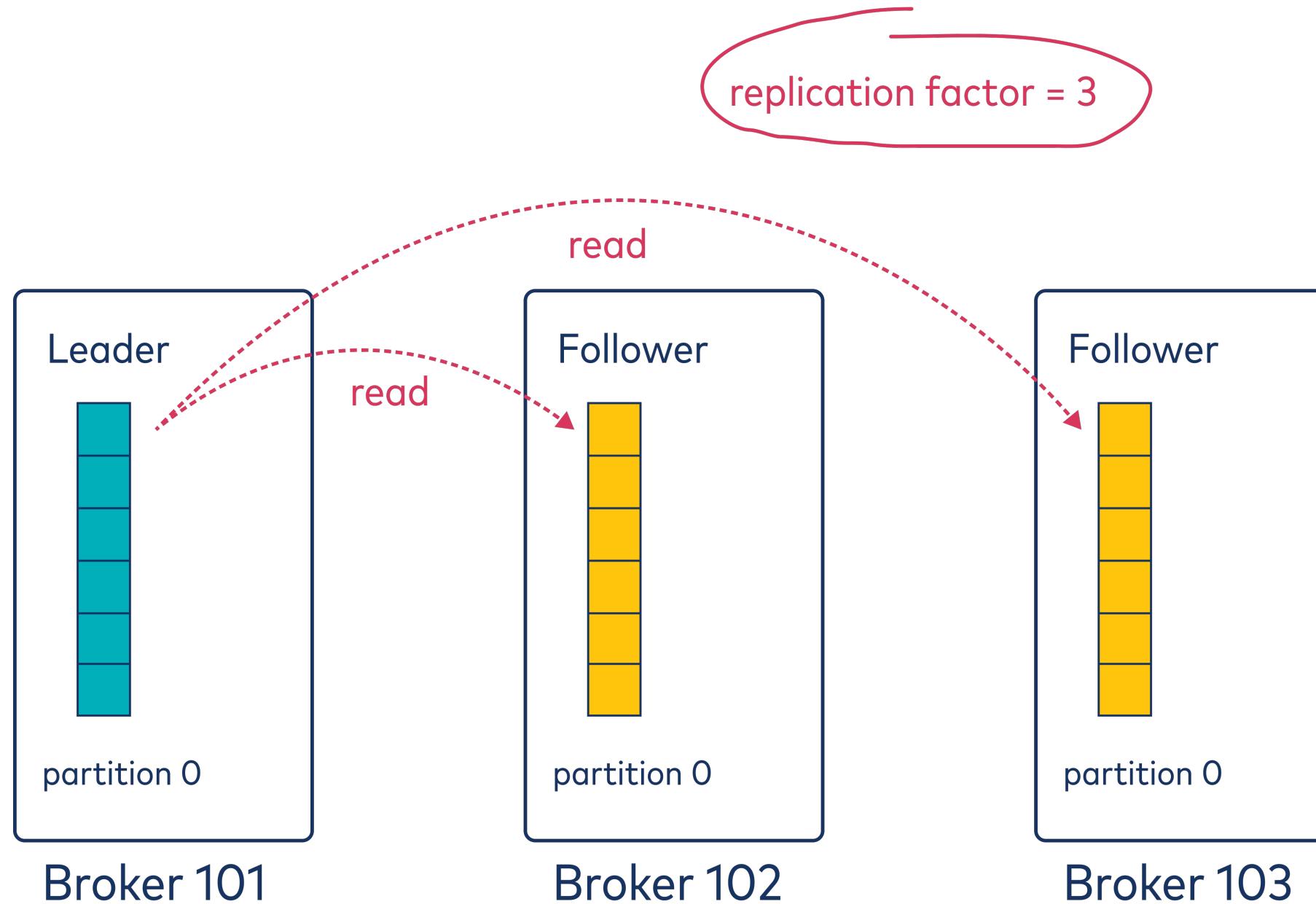


Module Map

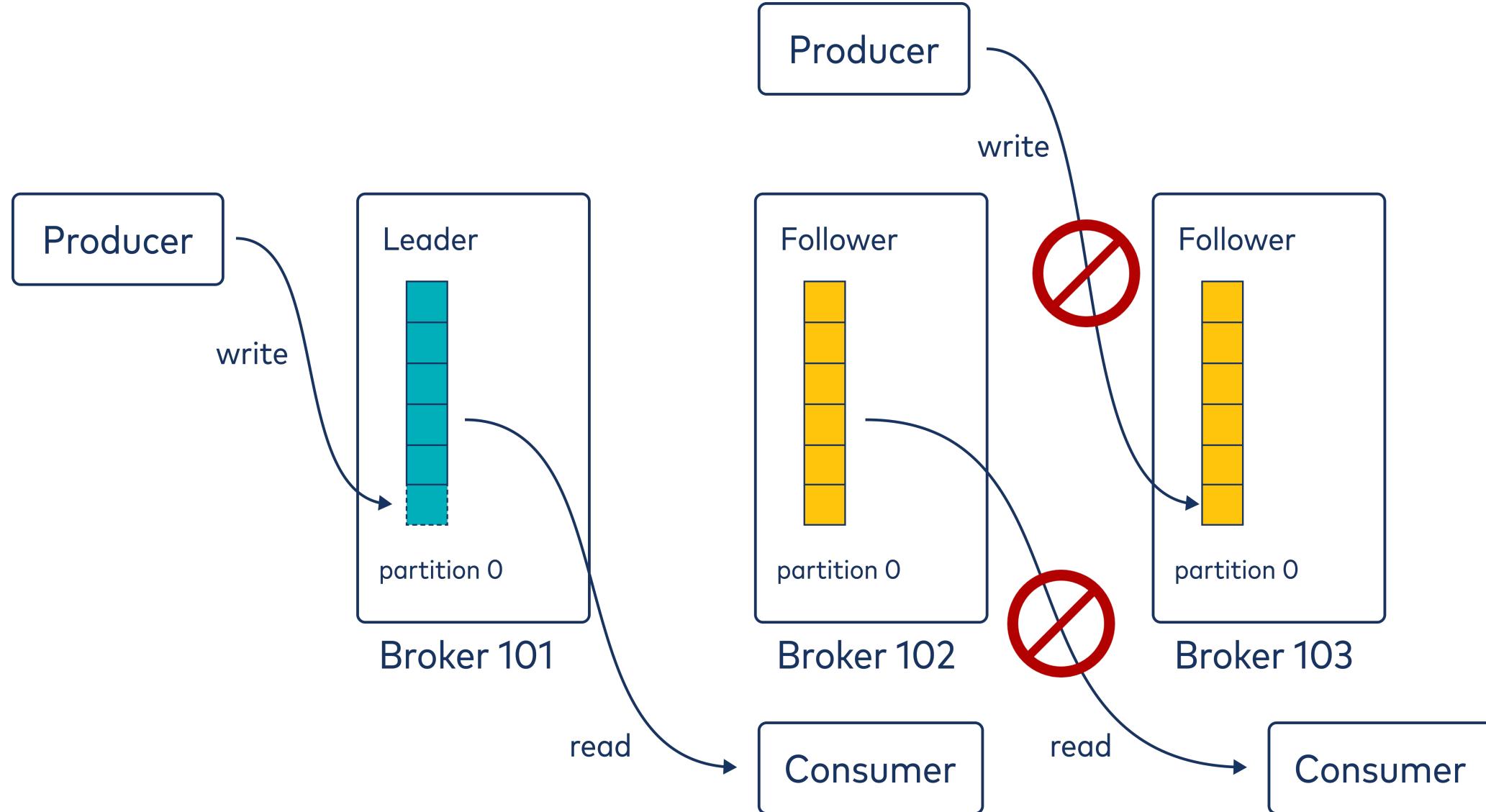


- Event Streaming Platform
- Event Streaming Architecture
- Log Retention
- Replication ... ←
- 🌐 Hands-on Lab

Replication of Partitions

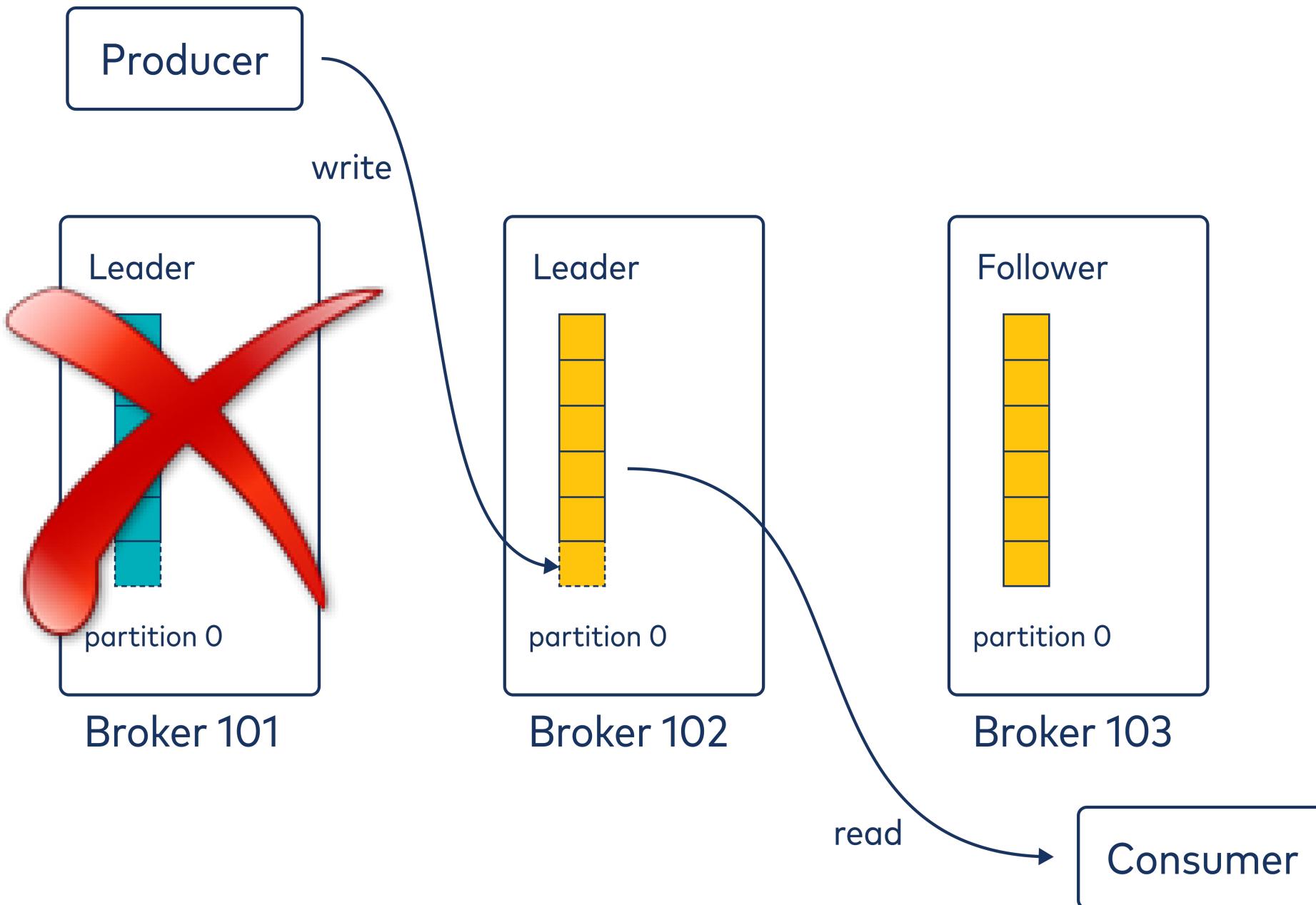


Clients Interact with Leaders



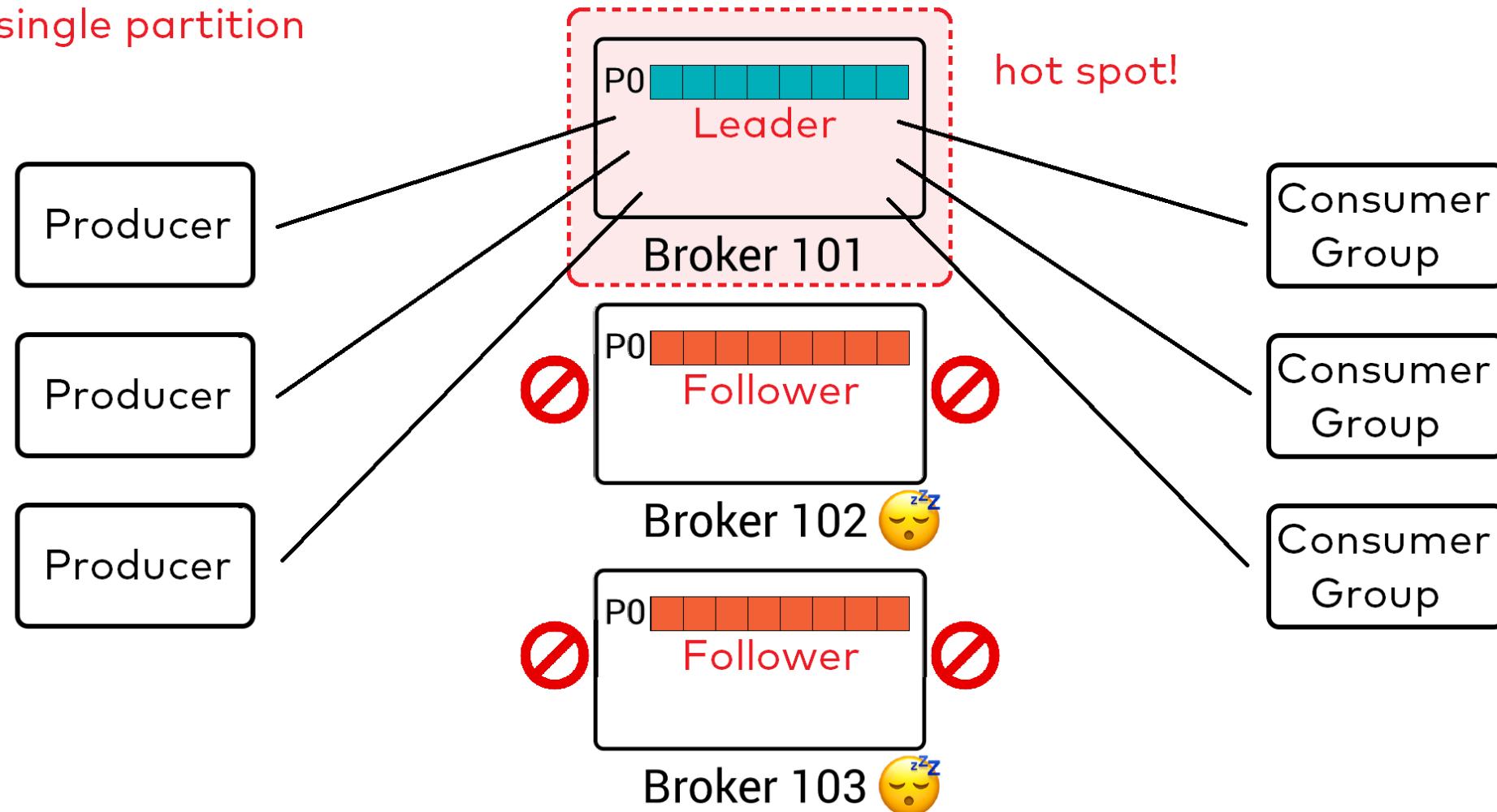
Caveat: Apache Kafka 2.4 introduced follower fetching for consumers.

Leader Failover



Load Balancing Partitions Leadership (1)

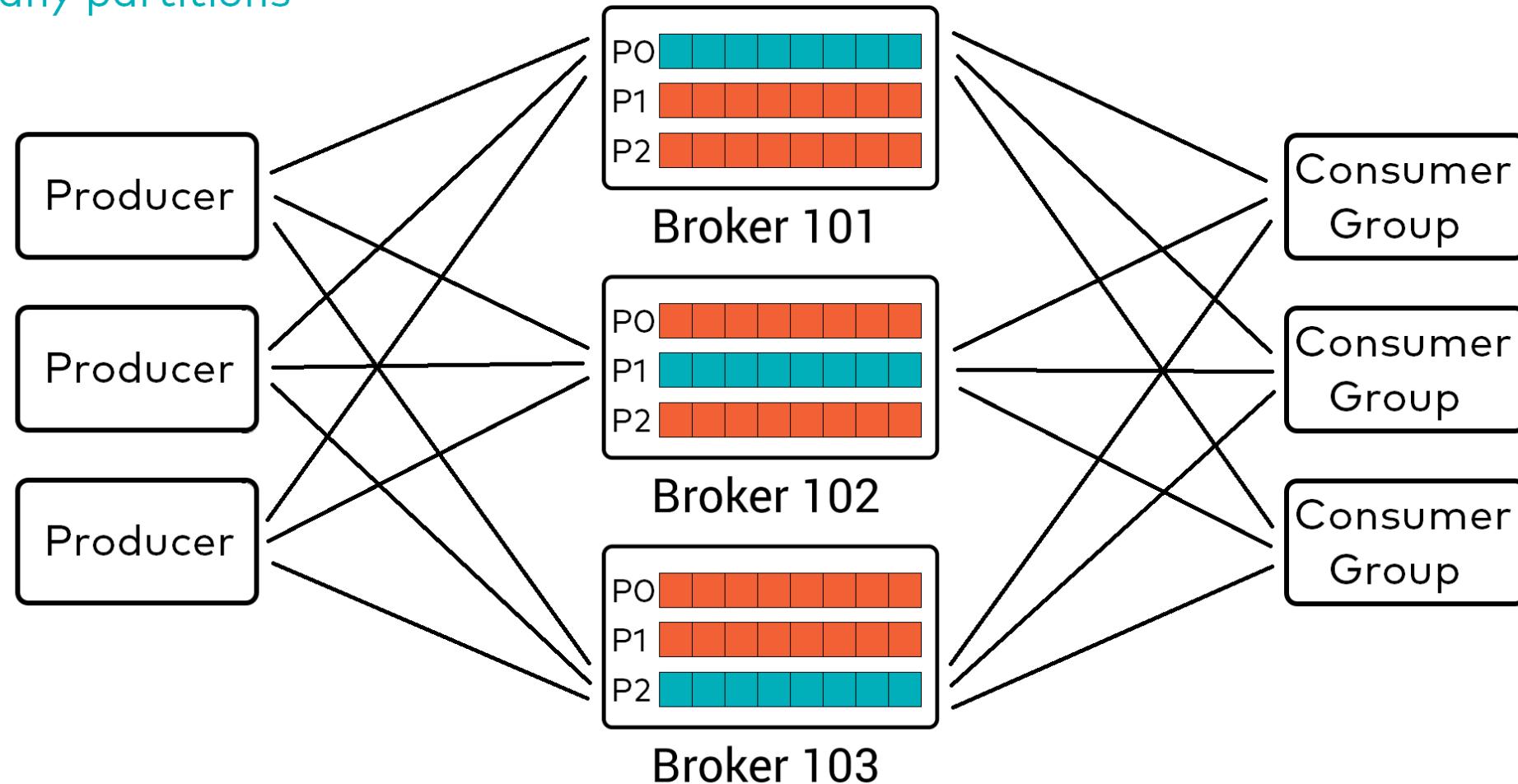
many clients producing to
and consuming from
a single partition



Load Balancing Partitions Leadership (2)

many clients producing to
and consuming from
many partitions

no hotspots!

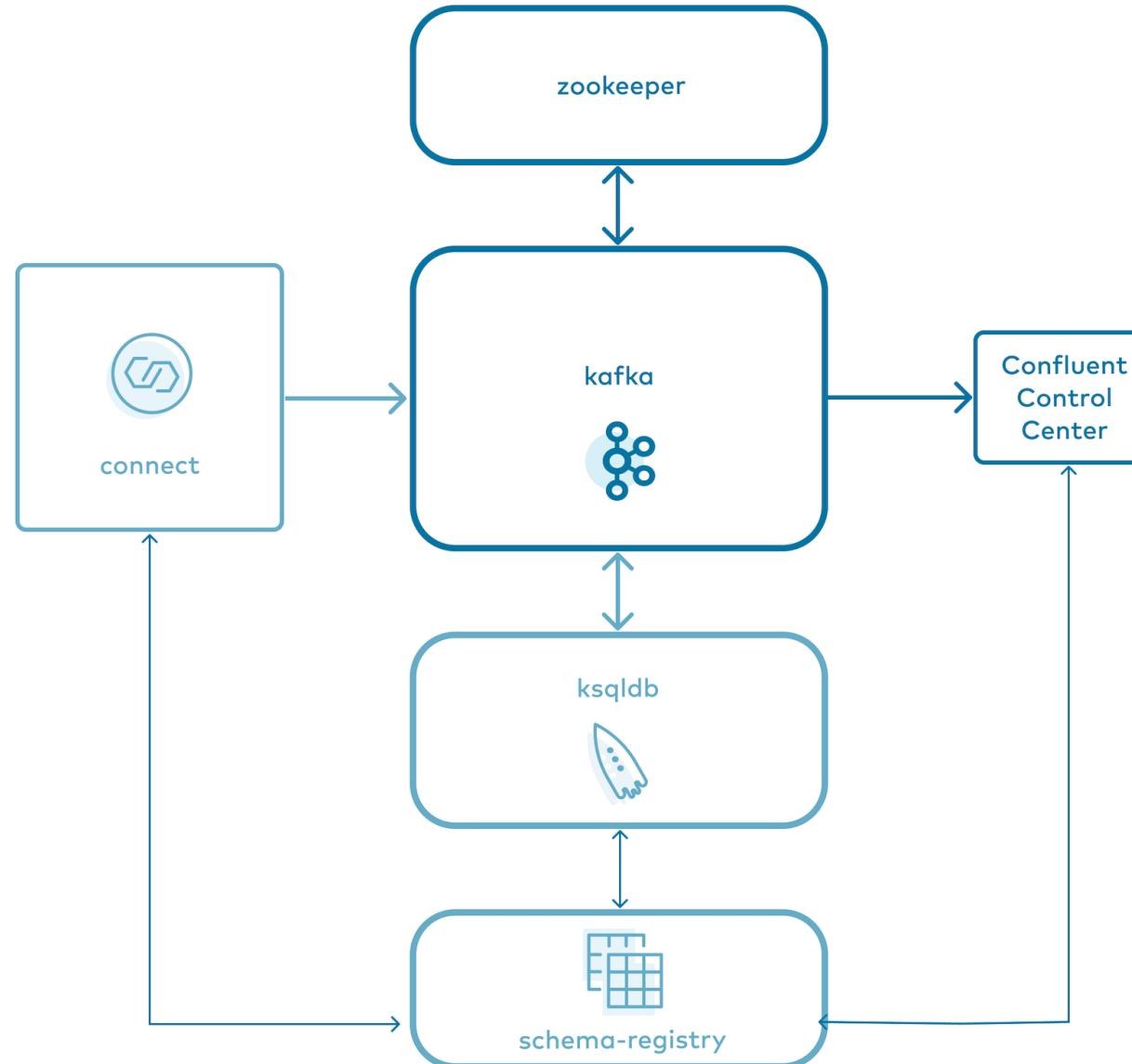


Hands-On Lab

- In this Hands-On Exercise, you will run a simple Kafka cluster and use some of the Kafka command line tools to do basic operations on the cluster.
- Please refer to **Lab 02 Fundamentals of Apache Kafka** in the Exercise Book:
 - a. **Introduction**
 - b. **Using Kafka's Command-Line Tools**



Preparing the Labs



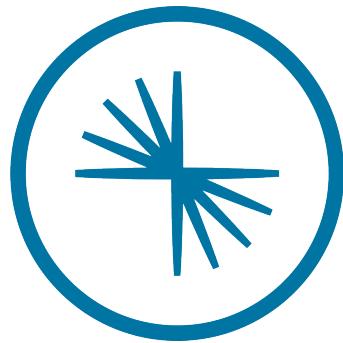
Module Review



Questions:

1. Explain the relationships between **records**, **topics**, and **partitions**.
2. How does Kafka achieve **high availability**?
3. What is the **easiest** way to get data from a database into Kafka?

03 Producing Messages to Kafka



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka ... ←
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

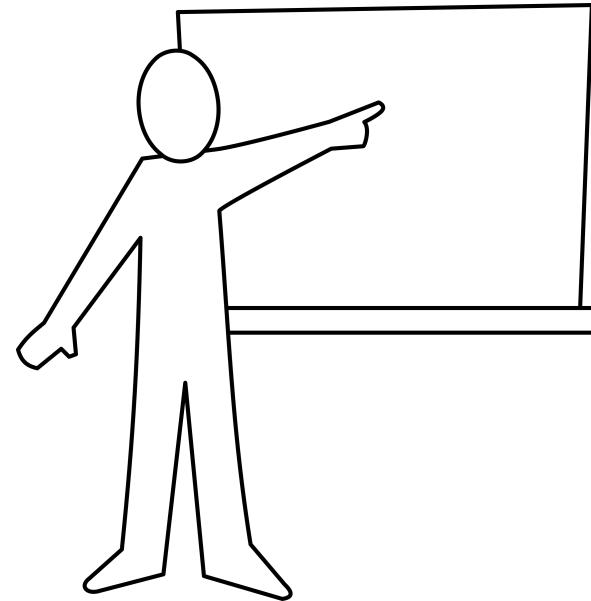
Learning Objectives



After this module you will be able to:

- sketch the high level architecture of a Kafka producer
- illustrate key-based partitioning
- explain the difference between `acks=0`, `acks=1`, and `acks=all`
- configure `delivery.timeout.ms` to control retry behavior
- create a custom `producer.properties` file
- tune throughput and latency using batching
- create a producer using the Java Client API
- create a producer with Confluent REST Proxy

Module Map



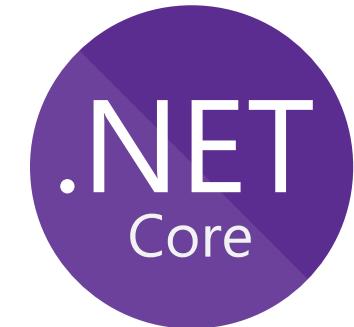
- Design and Configuration ... ←
- Life of a KafkaProducer
- Confluent REST Proxy
- 🌐 Hands-on Lab

Kafka Clients Supported by Confluent

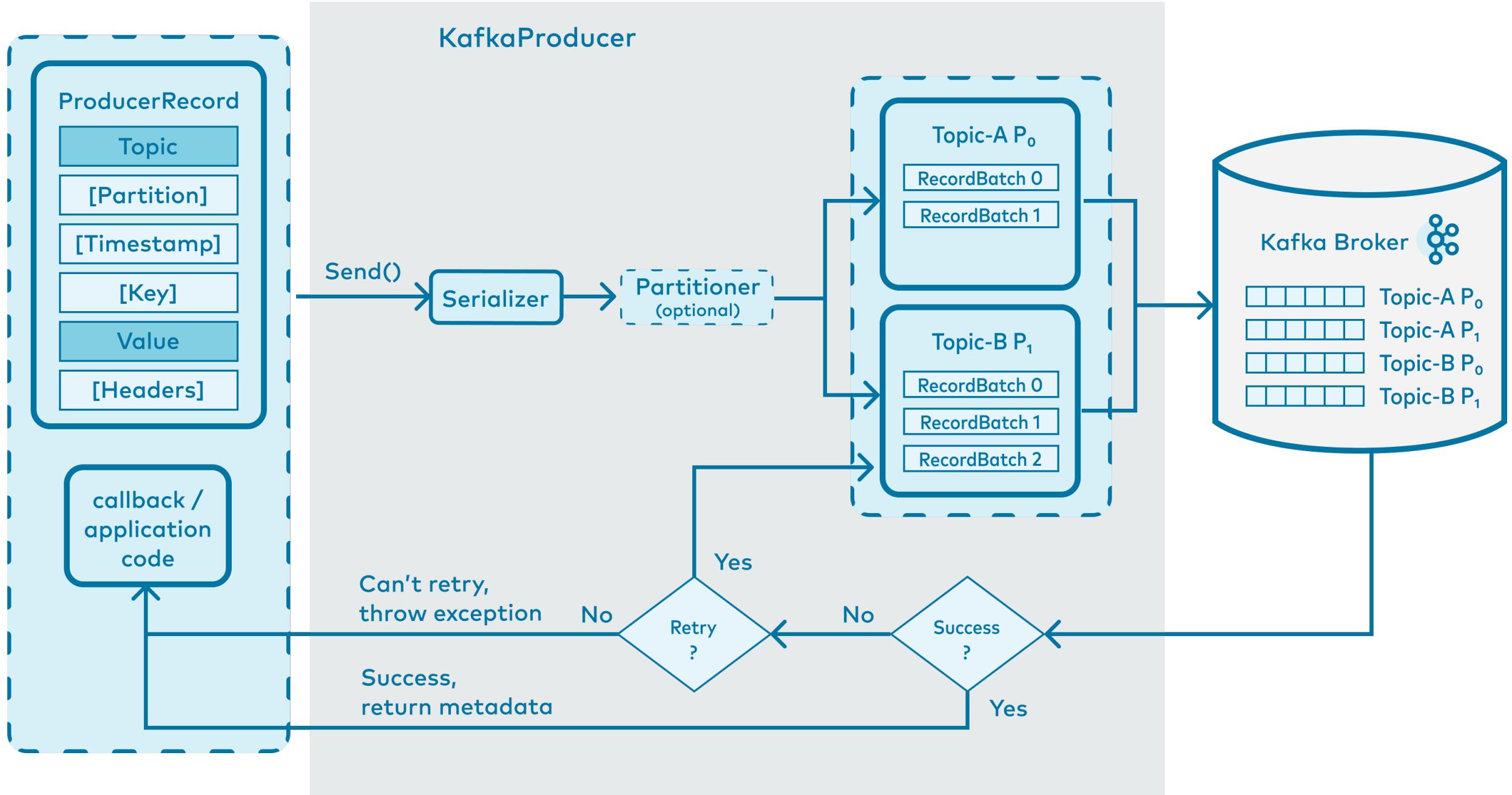
JVM



librdkafka (C library)

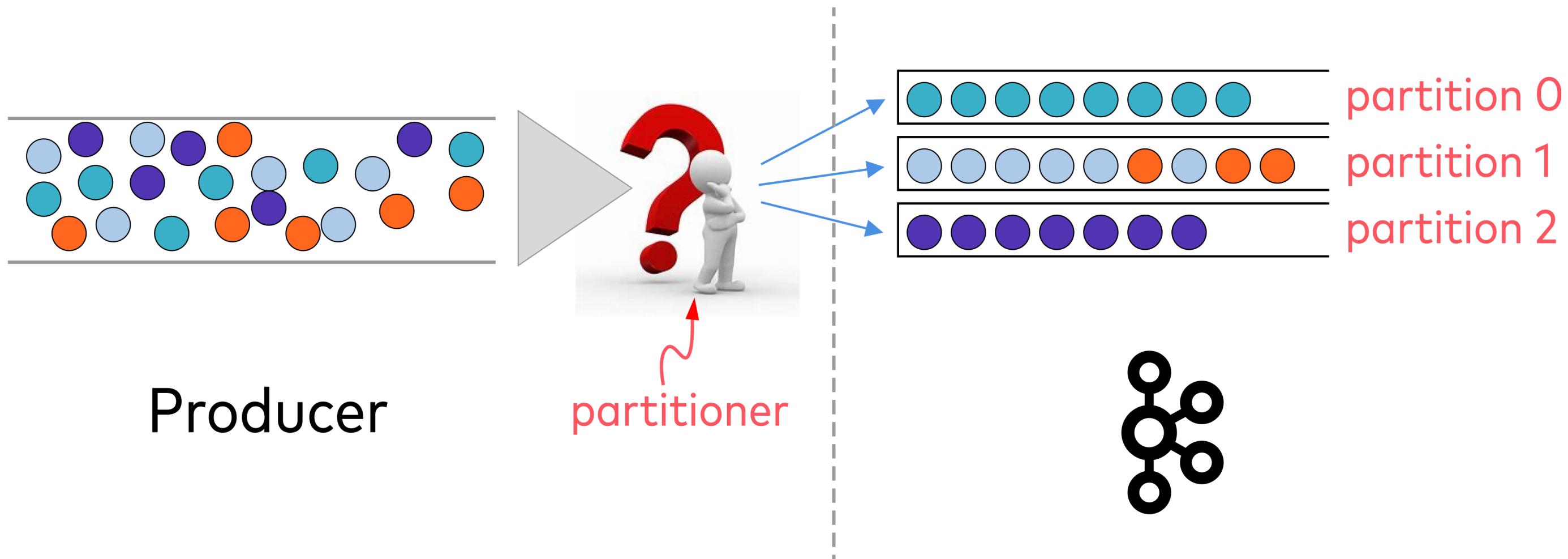


KafkaProducer Design



Default Partitioner

Partition = $\text{hash}(\text{key}) \% \text{Number of Partitions}$



Important Producer Properties (1)

Name	Description
<code>bootstrap.servers</code>	<p>Comma separated list of Broker host/port pairs used to establish the initial connection to the cluster. Example:</p> <pre>kafka-1:9092, kafka-2:9092, kafka-3:9092</pre>
<code>key.serializer</code>	<p>Class used to serialize the key. Must implement Kafka's <code>Serializer</code> interface. Example:</p> <pre>StringSerializer.class</pre>
<code>value.serializer</code>	<p>Class used to serialize the value. Must implement Kafka's <code>Serializer</code> interface. Example:</p> <pre>KafkaAvroSerializer.class</pre>

Important Producer Properties (2)

Name	Description
client.id	String to identify this producer uniquely; used in monitoring and logs. Example: producer1

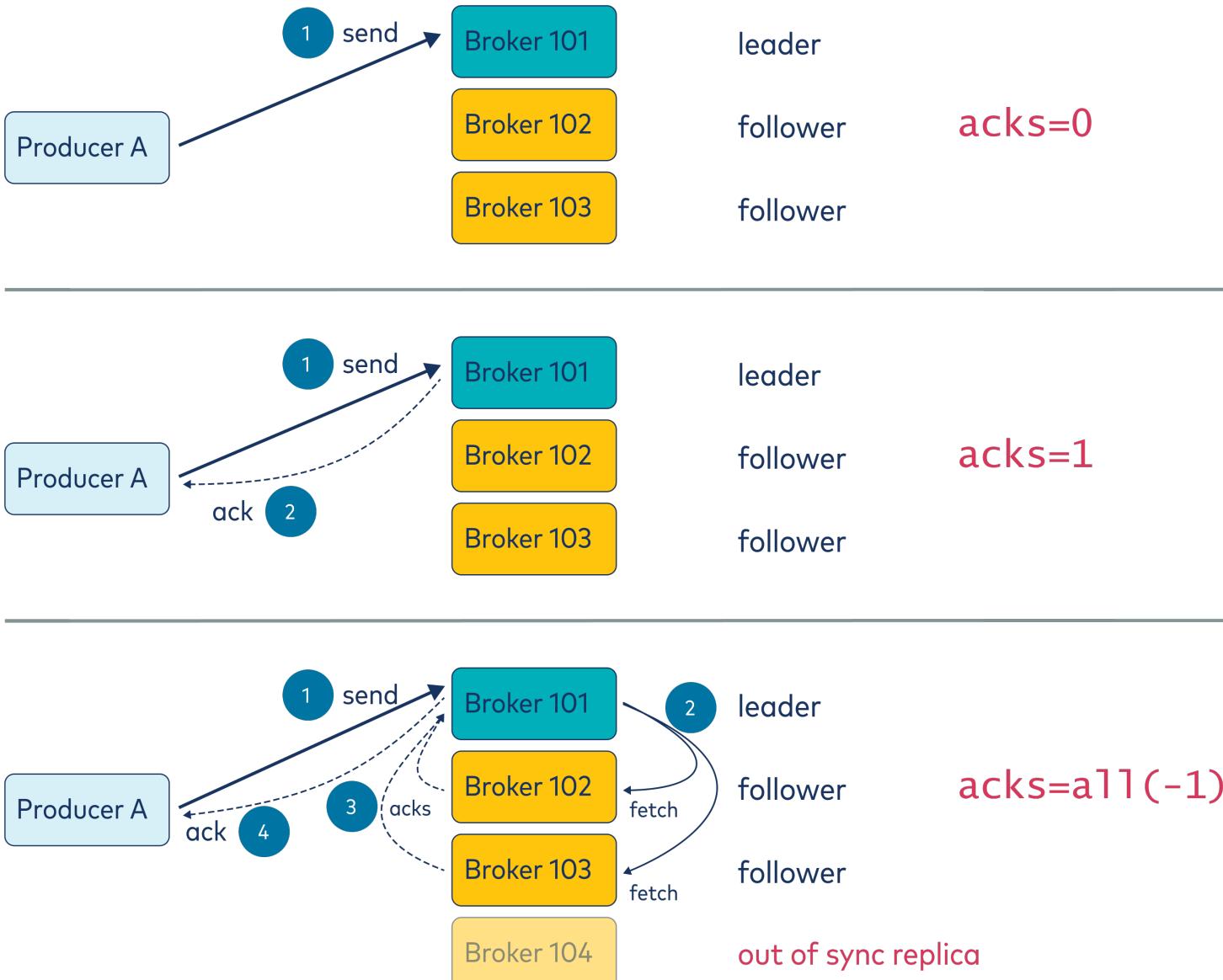
Important Producer Properties (3)

Name	Description
<code>compression.type</code>	How data should be compressed. Values are <code>none</code> , <code>snappy</code> , <code>gzip</code> , <code>lz4</code> , <code>zstd</code> . Compression is performed on batches of records. Default: <code>none</code>
<code>acks</code>	Used to determine when a write request is successful. Can be 0, 1, or all (-1). If <code>acks</code> is not zero, then the producer will retry failed requests. Default: <code>1</code>

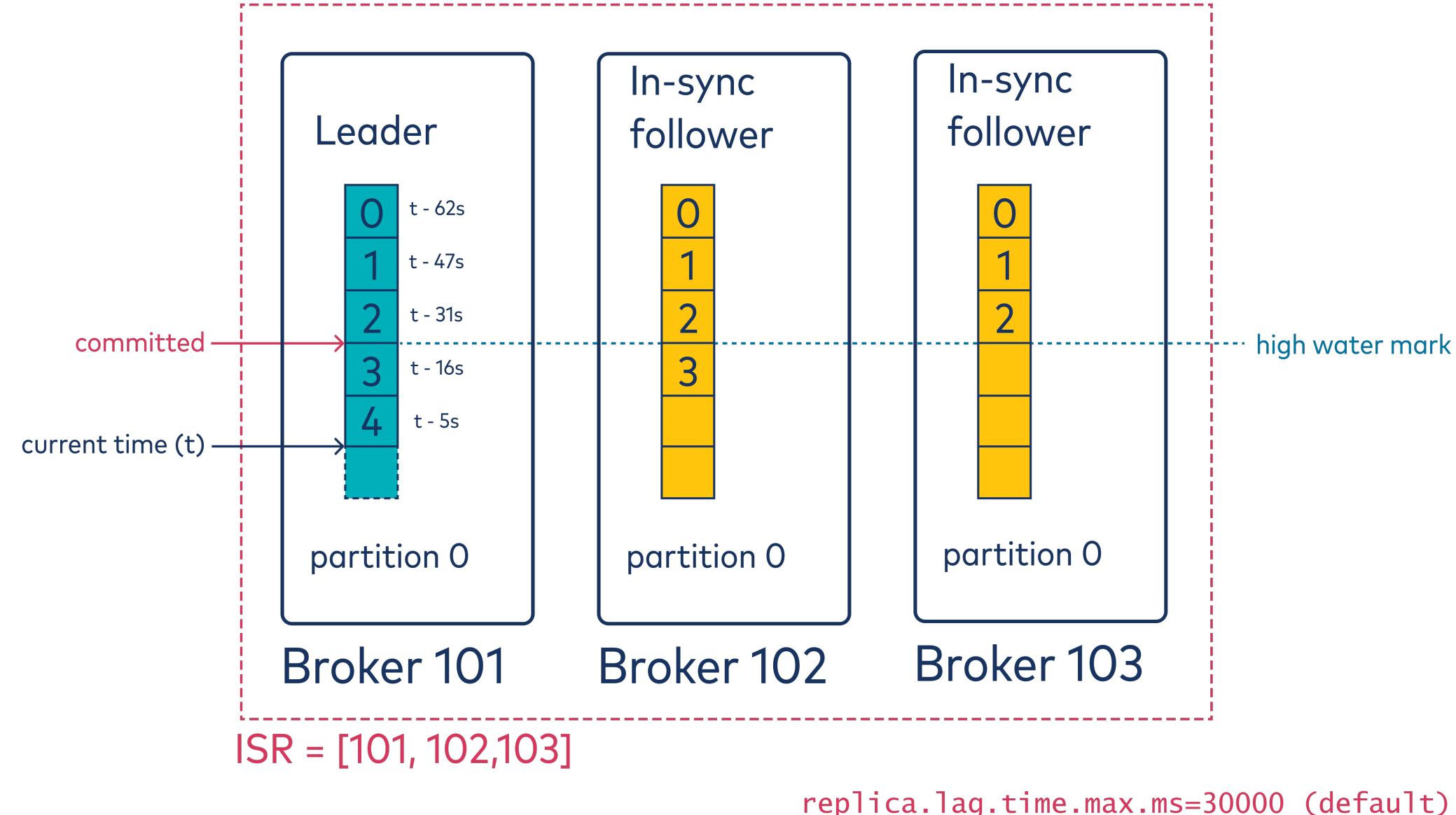
Important Producer Properties (4)

Name	Description
<code>delivery.timeout.ms</code>	Puts an upper bound on the time to report success or failure after a call to <code>send()</code> returns. Use this to control producer retries. Default: <code>120000</code>
<code>batch.size</code>	Max number of bytes for a batch of messages sent to a partition. Default: <code>16384</code>
<code>linger.ms</code>	Time a batch will wait to accumulate before sending. Default: <code>0</code>

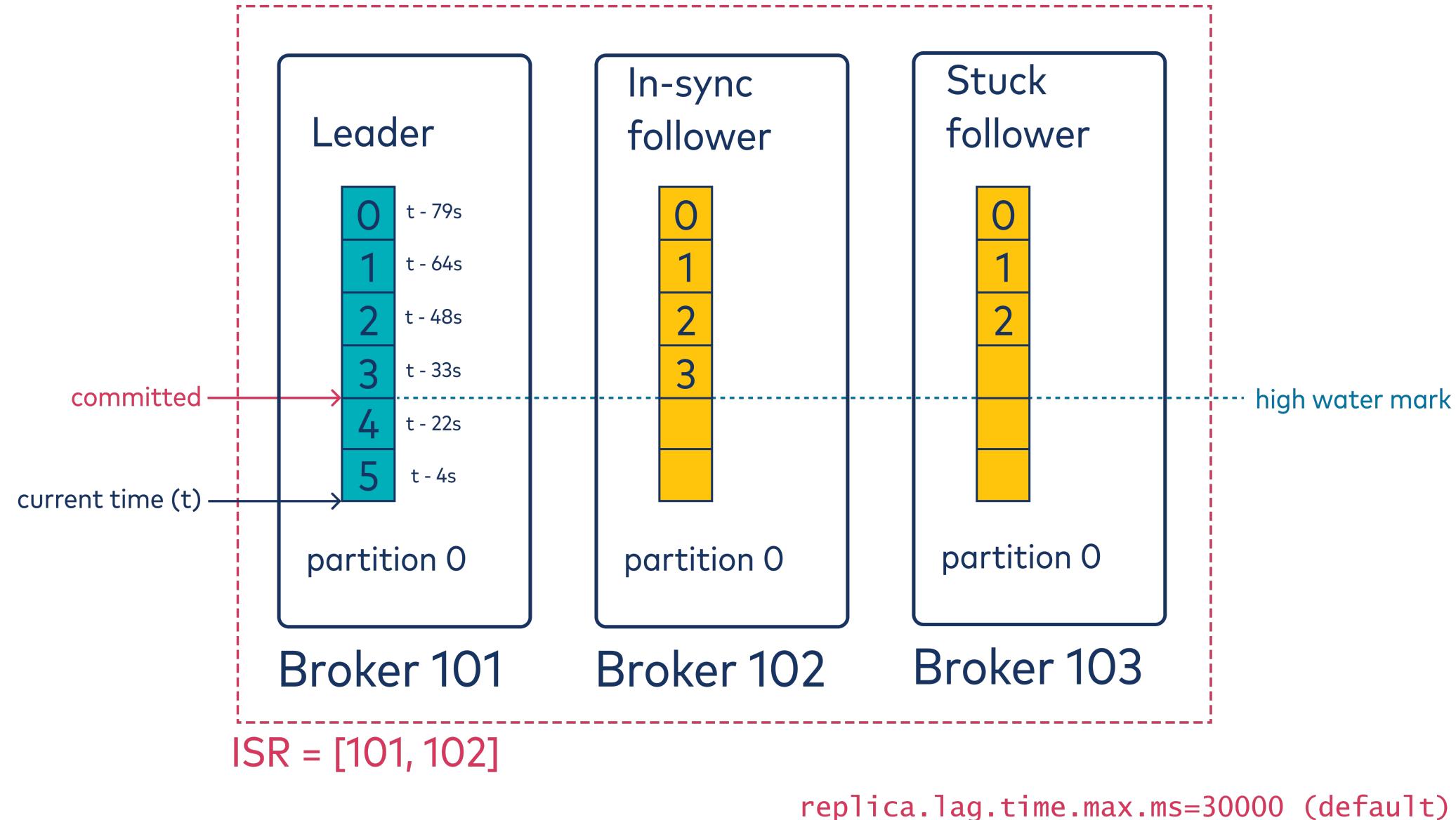
A Closer Look at Acknowledgments



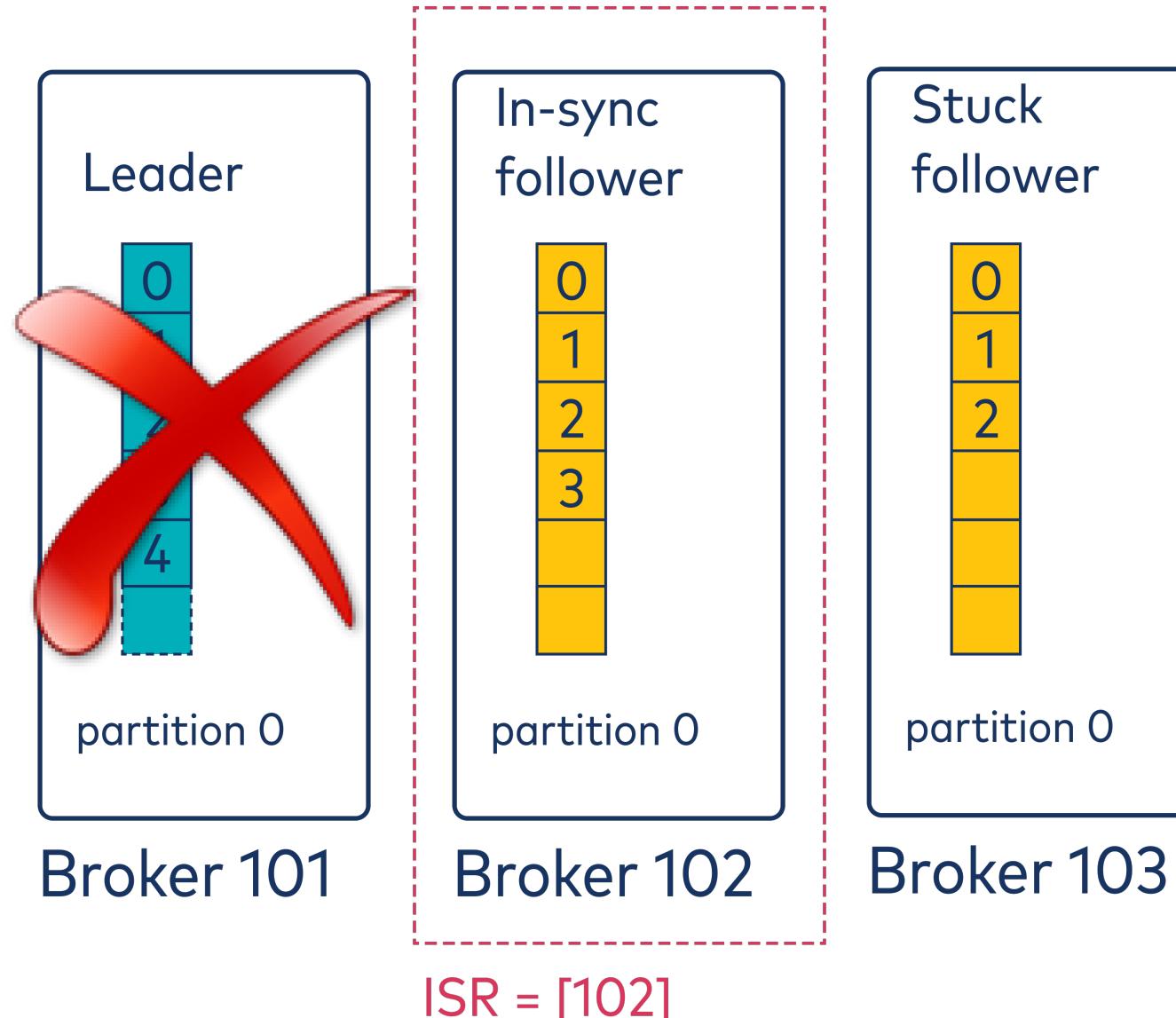
In-Sync Replicas



Stuck Follower



Leader Failure with a Stuck Follower



When Does `acks=ALL` Equal `acks=1`?



ISR = [102]

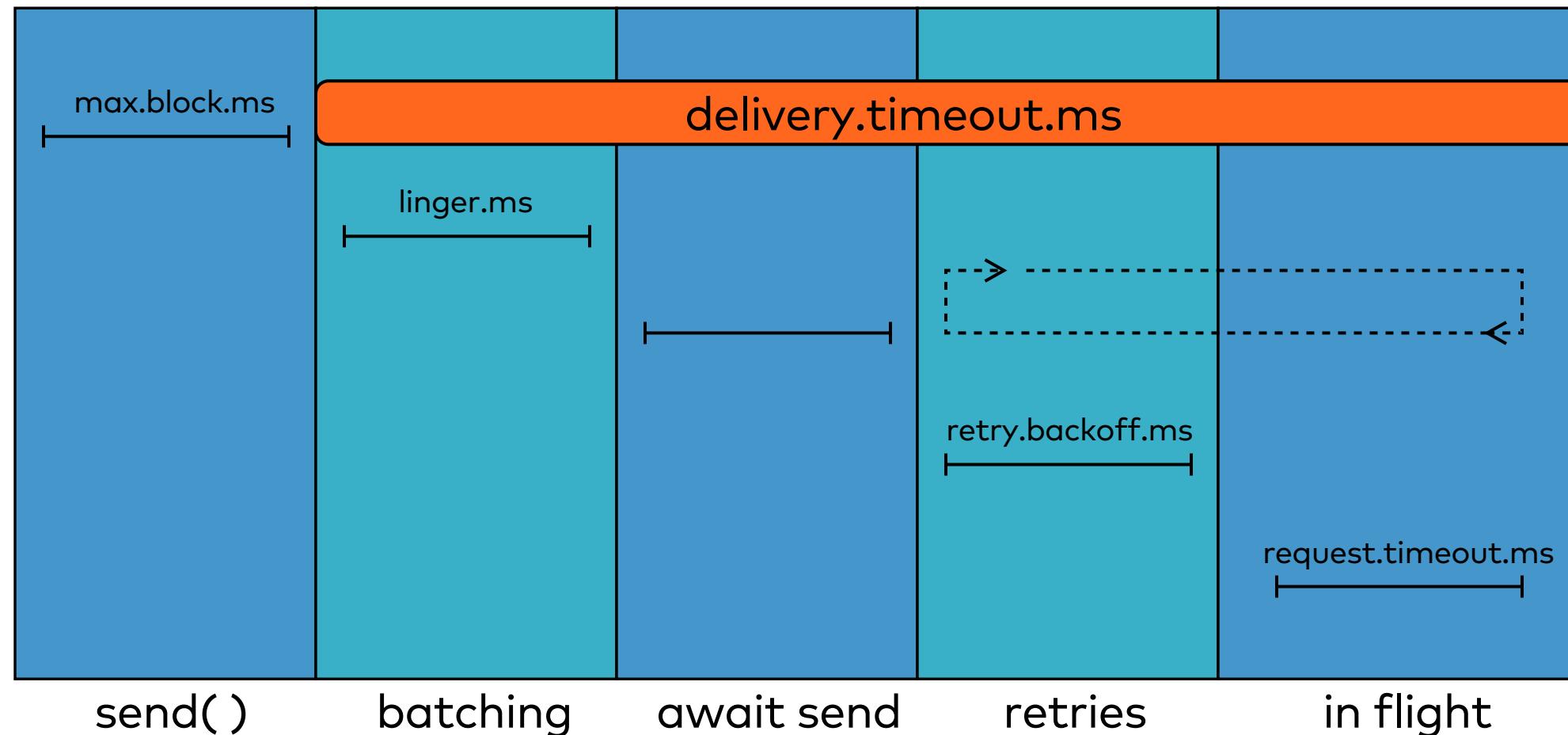
Enforce `acks=ALL` Guarantee



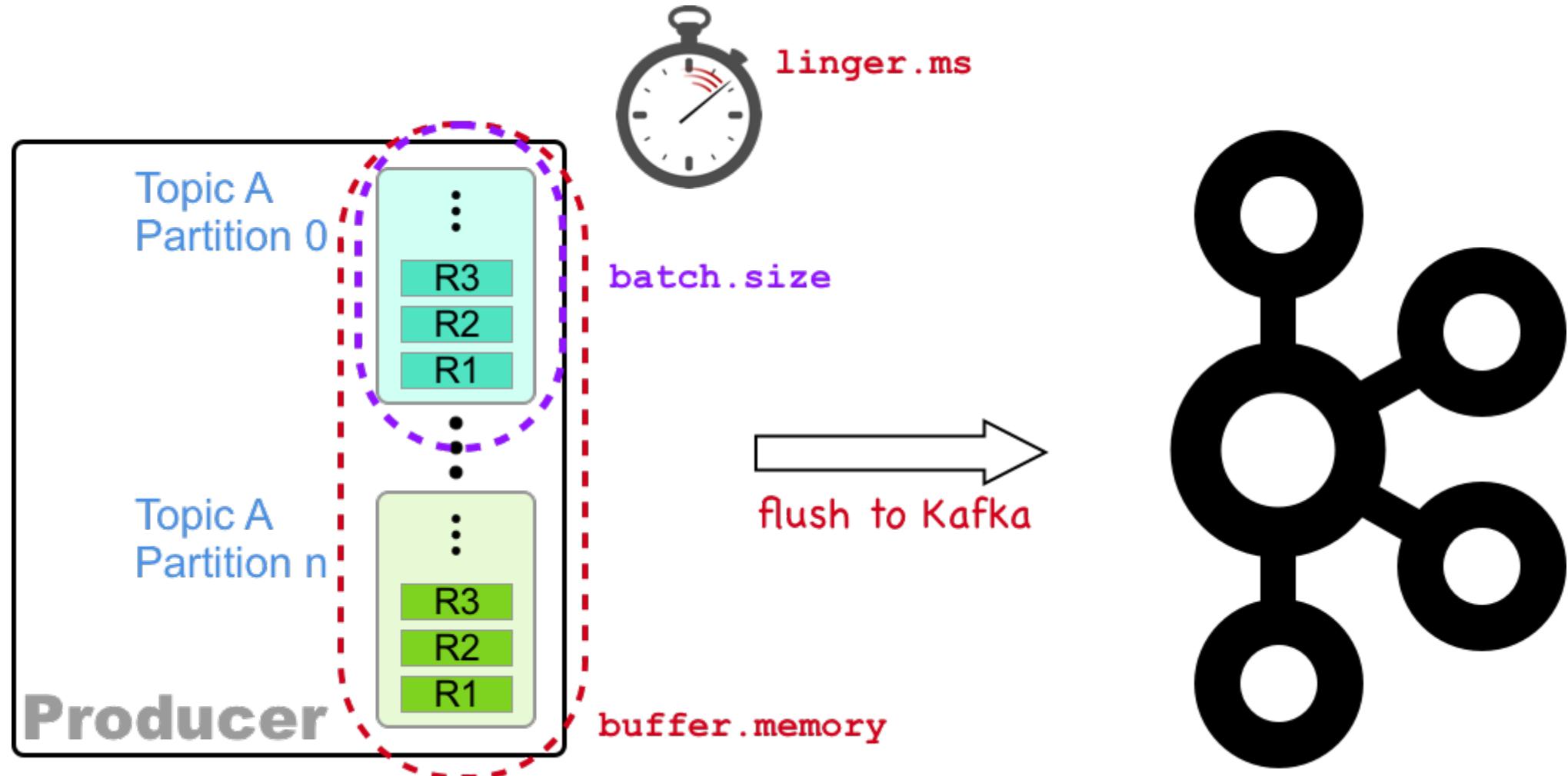
ISR = [102]

Using Delivery Timeout to Control Retries

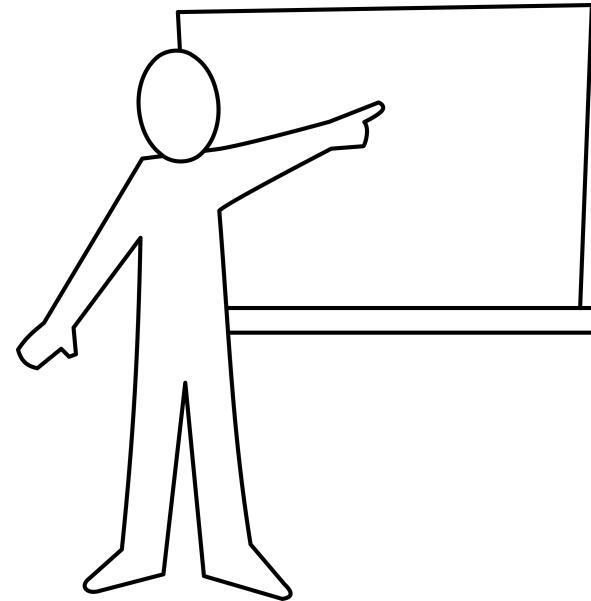
- Producer property `delivery.timeout.ms` puts an upper bound on the time to report success or failure after a call to `send()` returns. Use this to control retry behavior.



Performance Tuning Batching



Module Map



- Design and Configuration
- Life of a KafkaProducer ... ←
- Confluent REST Proxy
- 🌐 Hands-on Lab

Creating the Properties and KafkaProducer Objects

```
1 Properties props = new Properties();
2 props.put("bootstrap.servers", "broker-1:9092");
3 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer.class");
4 props.put("value.serializer", "org.apache.kafka.common.serialization.KafkaAvroSerializer.class");
5
6 KafkaProducer<String, MyObject> producer = new KafkaProducer<>(props);
```

- `ByteArraySerializer`, `IntegerSerializer`, `LongSerializer`, and more are included in the client
- `StringSerializer` encoding defaults to UTF8
 - Can be customized by setting the property `serializer.encoding`

Loading Properties in Practice

- Using Helper Classes

```
1 final Properties props = new Properties();  
2  
3 props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");  
4 ... // Load other properties  
5  
6 KafkaProducer<String, MyObject> producer = new KafkaProducer<>(props);
```

- Using a Properties File

```
1 final Properties props = new Properties();  
2  
3 InputStream propsFile = new FileInputStream("src/main/resources/producer.properties");  
4 props.load(propsFile);  
5  
6 KafkaProducer<String, MyObject> producer = new KafkaProducer<>(props);
```

Sending Messages to Kafka

```
1 String k = "mykey";  
2 String v = "myvalue";  
3 ProducerRecord<String, String> record = new ProducerRecord<String, String>("my_topic", k, v);  
4 producer.send(record);
```

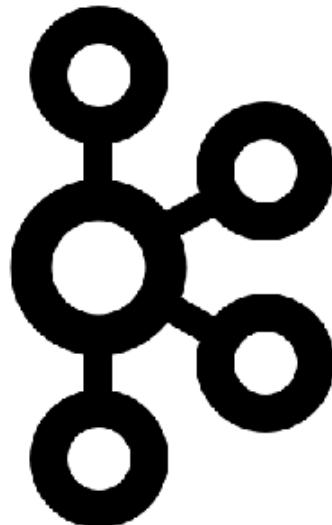
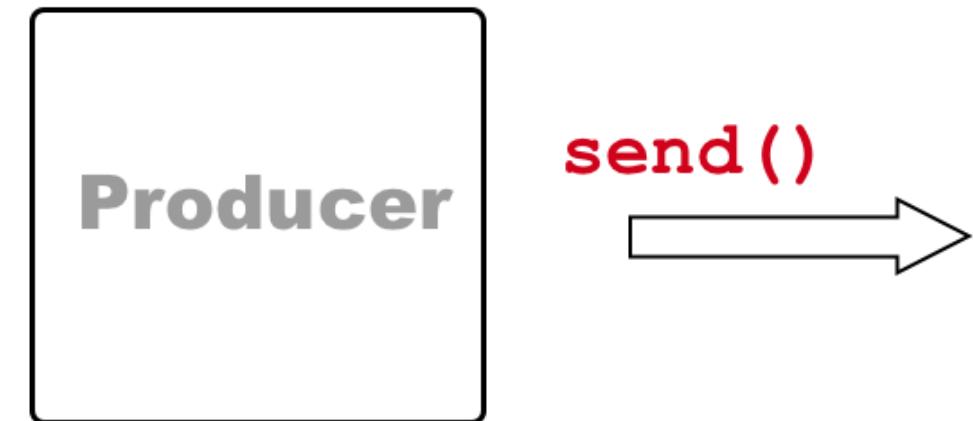


`ProducerRecord` can take an optional timestamp if you don't want to use current system time

Asynchronous Sending

`KafkaProducer::send()...`

- does not wait for broker response before returning
- returns `Future` with `RecordMetadata`
- force blocking:
`producer.send(...).get()`



When Do Producers Actually Send?

- A Producer `send()` returns immediately after it has added the message to a local buffer of pending record sends
 - This allows the Producer to send records in batches for better performance
- Then the Producer flushes multiple messages to the Brokers based on batching configuration properties
- You can also manually flush by calling the synchronous Producer method `flush()`

send() and Callbacks (1)

- `send(record)` is equivalent to `send(record, null)`
- Instead, it is possible to supply a `Callback` as the second parameter
 - This is invoked when the send has been acknowledged or when the transfer fails
 - It is an Interface with an `onCompletion` method:

```
onCompletion(RecordMetadata metadata, java.lang.Exception exception) {...}
```

- `Callback` parameters
 - `exception` will be `null` if no error occurred



When `exception` is not `null` in the callback, `metadata` will contain the special `-1` value for all fields except for `topicPartition`, which will be valid.

send() and Callbacks (2)

- The record being sent is in the scope of the callback implementation
- Example code, with lambda expression:

```
1 producer.send(record, (recordMetadata, e) -> {
2     if (e != null) {
3         e.printStackTrace();
4     } else {
5         System.out.println("Message String = " + record.value() +
6                             ", Offset = " + recordMetadata.offset());
7     }
8 });
```

Closing the Producer

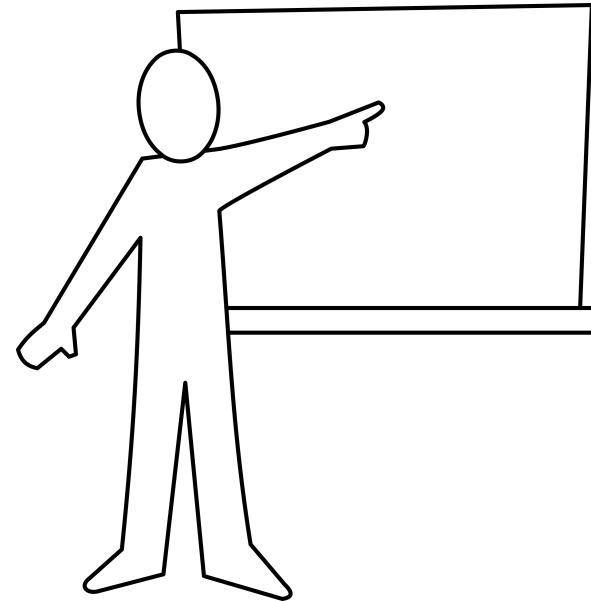
```
producer.close();
```

- Blocks until all previously sent requests complete

```
producer.close(timeout, timeUnit);
```

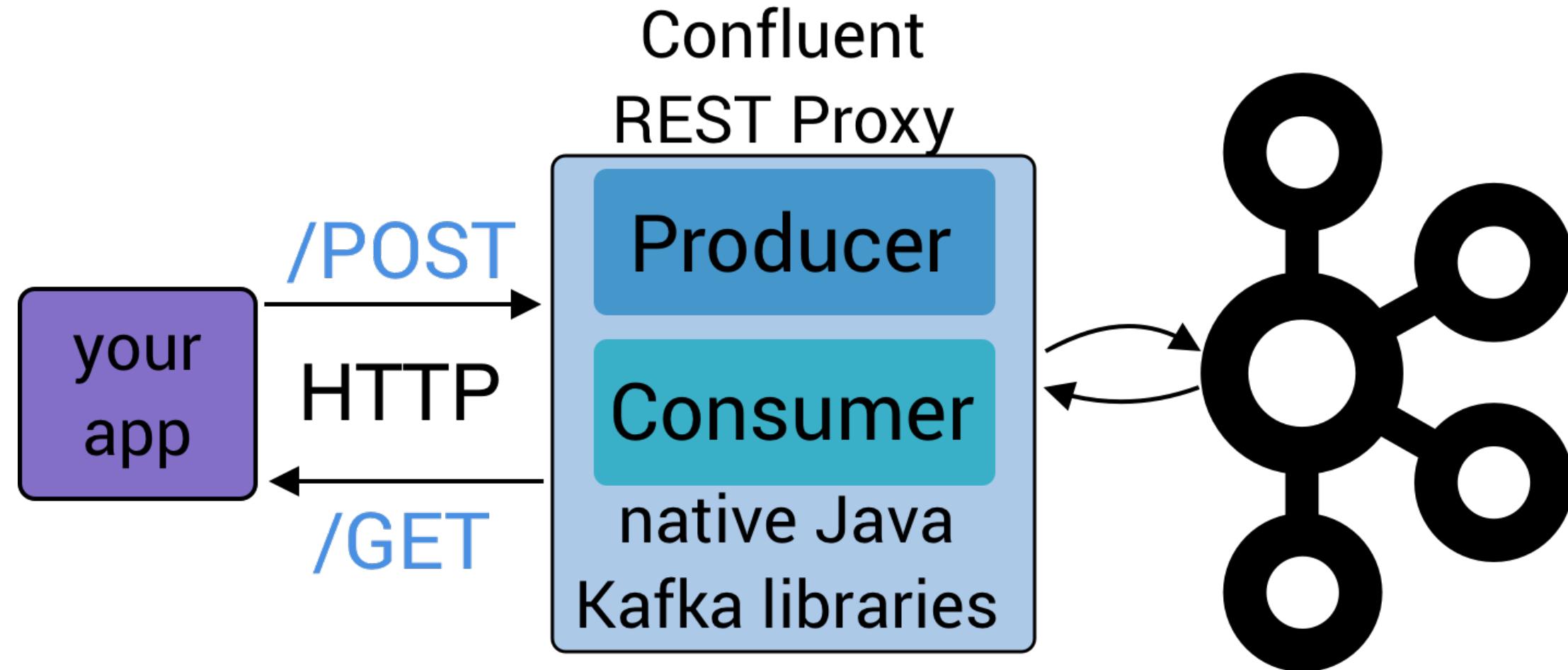
- Waits until complete or given timeout expires

Module Map



- Design and Configuration
- Life of a KafkaProducer
- Confluent REST Proxy ... ←
- 🌐 Hands-on Lab

Confluent REST Proxy (1)



Confluent REST Proxy (2)

- The Confluent REST Proxy allows you to use HTTP to perform actions on the Kafka cluster
- The REST calls are translated into Java Kafka client calls
- This allows virtually any language to access Kafka
- Uses POST to send data to Kafka:
 - JSON, binary, Avro, Protobuf and JSON Schema
- Uses GET to retrieve data from Kafka

Creating a Producer with REST Proxy

```
1 import requests
2 import json
3
4 url = "http://restproxy:8082/topics/my_topic"
5 headers = {"Content-Type" : "application/vnd.kafka.json.v2+json"}
6 # Create one or more messages
7 payload = {"records":
8     [
9         {"key": "firstkey",
10          "value": "firstvalue"
11     }]
12 # Send the message
13 r = requests.post(url, data=json.dumps(payload), headers=headers)
14 if r.status_code != 200:
15     print "Status Code: " + str(r.status_code)
16     print r.text
```

Hands-On Exercise Environment

- **Visual Studio Code**—development environment
- Exercises available in:
 - Java
 - Python
 - C#
- Front end webserver is written with a community NodeJS client



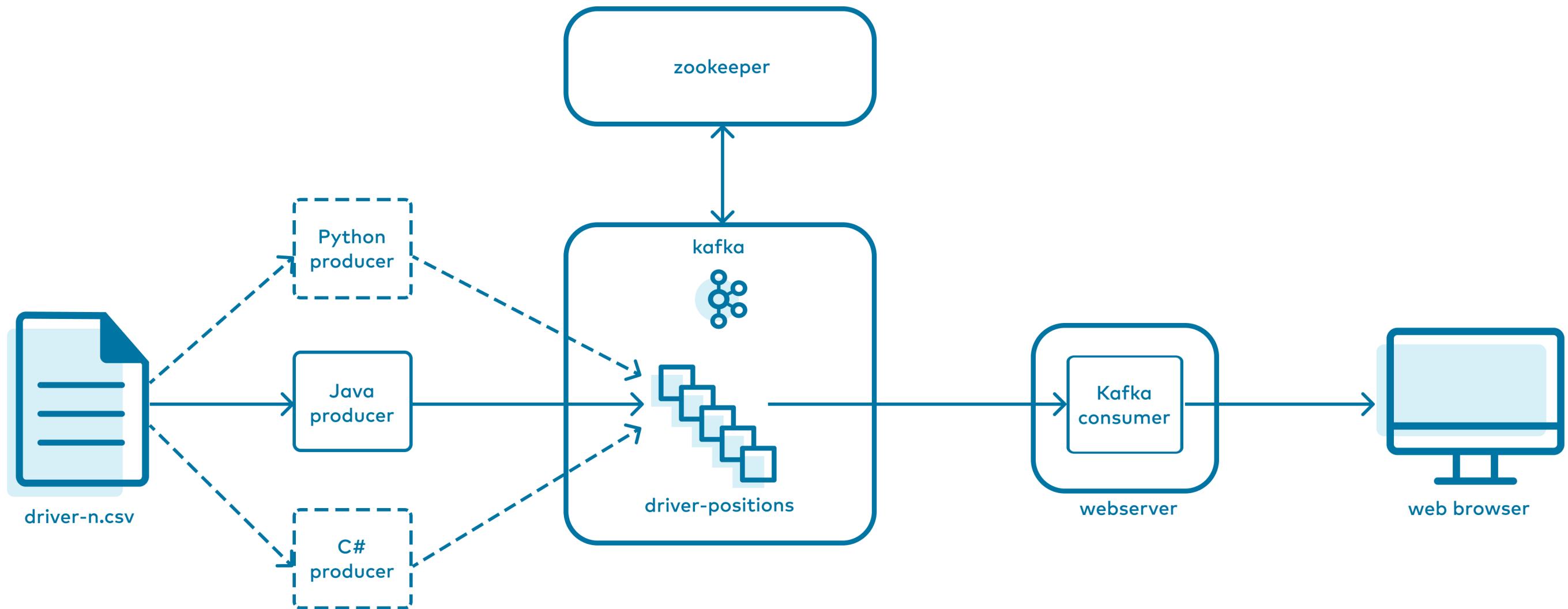
You are encouraged to try the exercises in multiple languages!

Hands-On Lab

- In this Hands-On Exercise, you will write a producer that produces driver location data to Kafka
- Please refer to **Lab 03 Producing Messages to Kafka** in the Exercise Book:
 - a. **Kafka Producer (Java, C#, Python)**



Kafka Producer (Java, C#, Python)



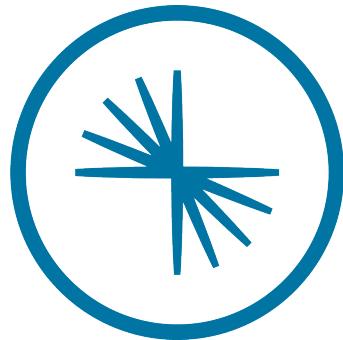
Module Review



Questions:

1. What is the difference between `acks=0`, `acks=1`, and `acks=all`? When might `acks=0` be appropriate?
2. Think of a particular use case where you want to produce data to Kafka. Which configuration settings might you customize for this use case?

04 Consuming Messages from Kafka



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka ... ←
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

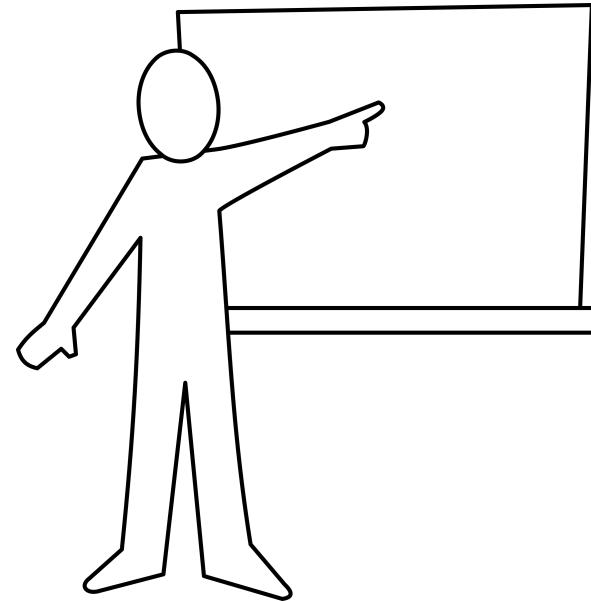
Learning Objectives



After this module you will be able to:

- illustrate how consumer groups and partitions provide scalability and fault tolerance
- tune consumers to avoid excessive rebalances
- explain the difference between "range" and "round robin" partition assignment strategies
- create a custom `consumer.properties` file
- tune fetch requests
- create a consumer using the Java Client API
- create a consumer with Confluent REST Proxy

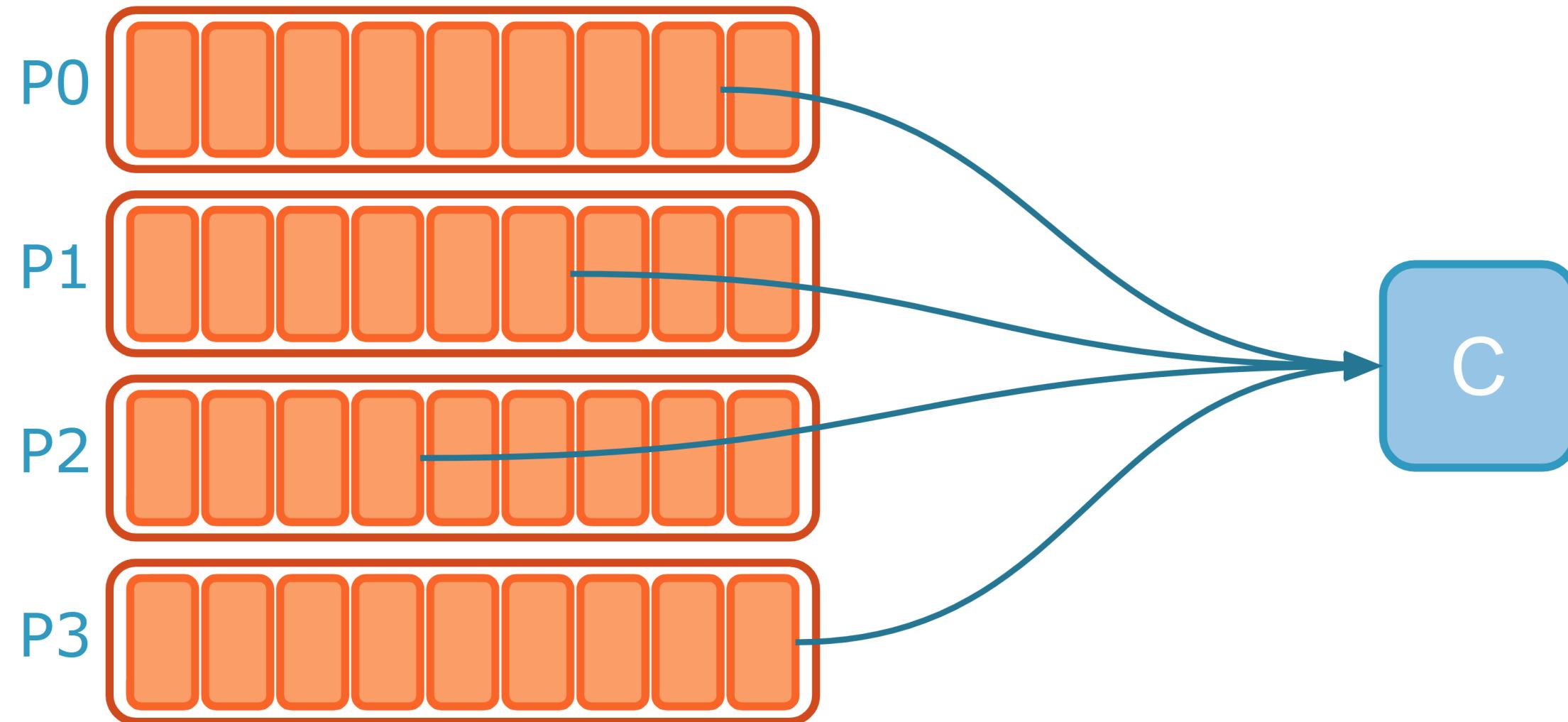
Module Map



- Consumer Groups and Partitions ... ←
- Consumer Group Rebalances
- Life of a KafkaConsumer
- Offset Management
- Performance Tuning
- Confluent REST Proxy
- 🚧 Hands-on Lab

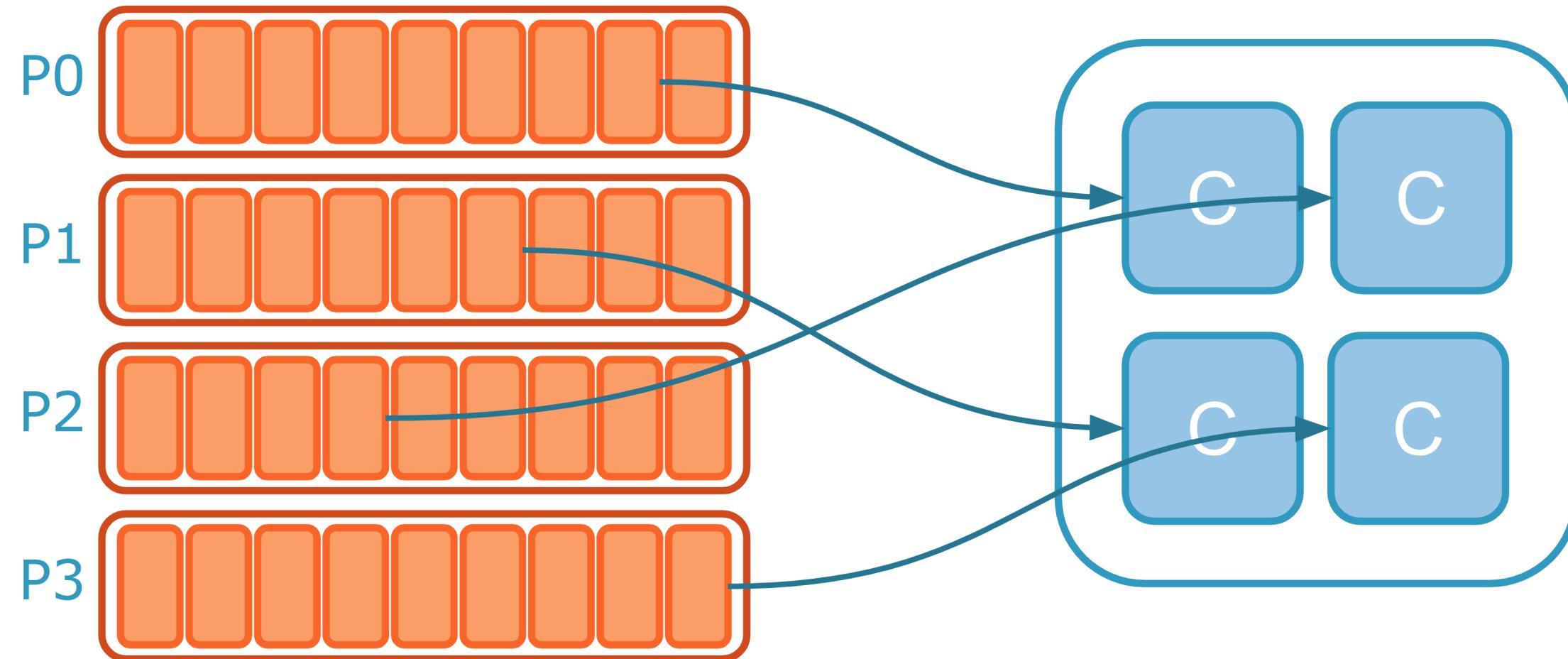
Consuming from Kafka - Single Consumer

Topic driver-positions

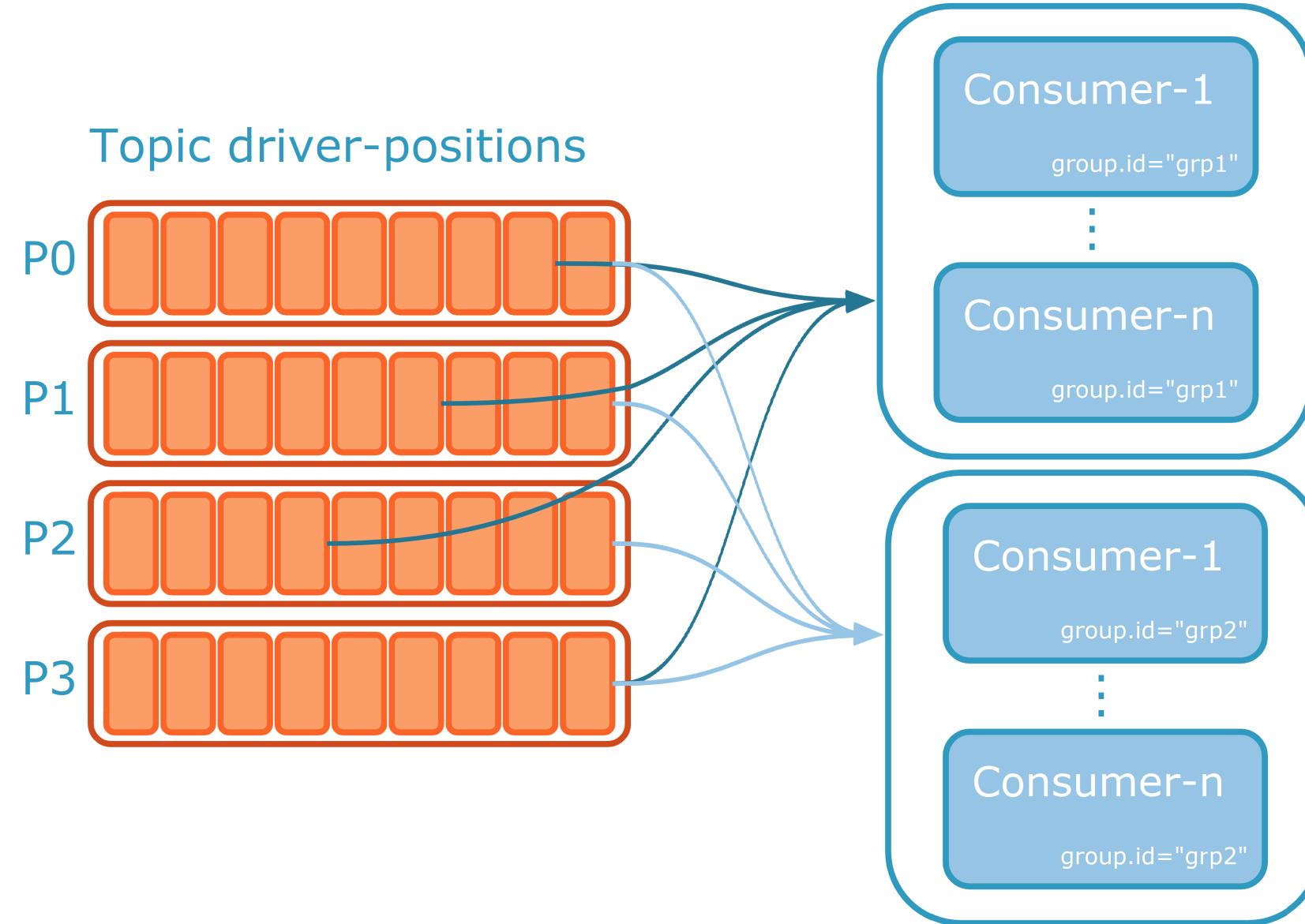


Consuming as a Group

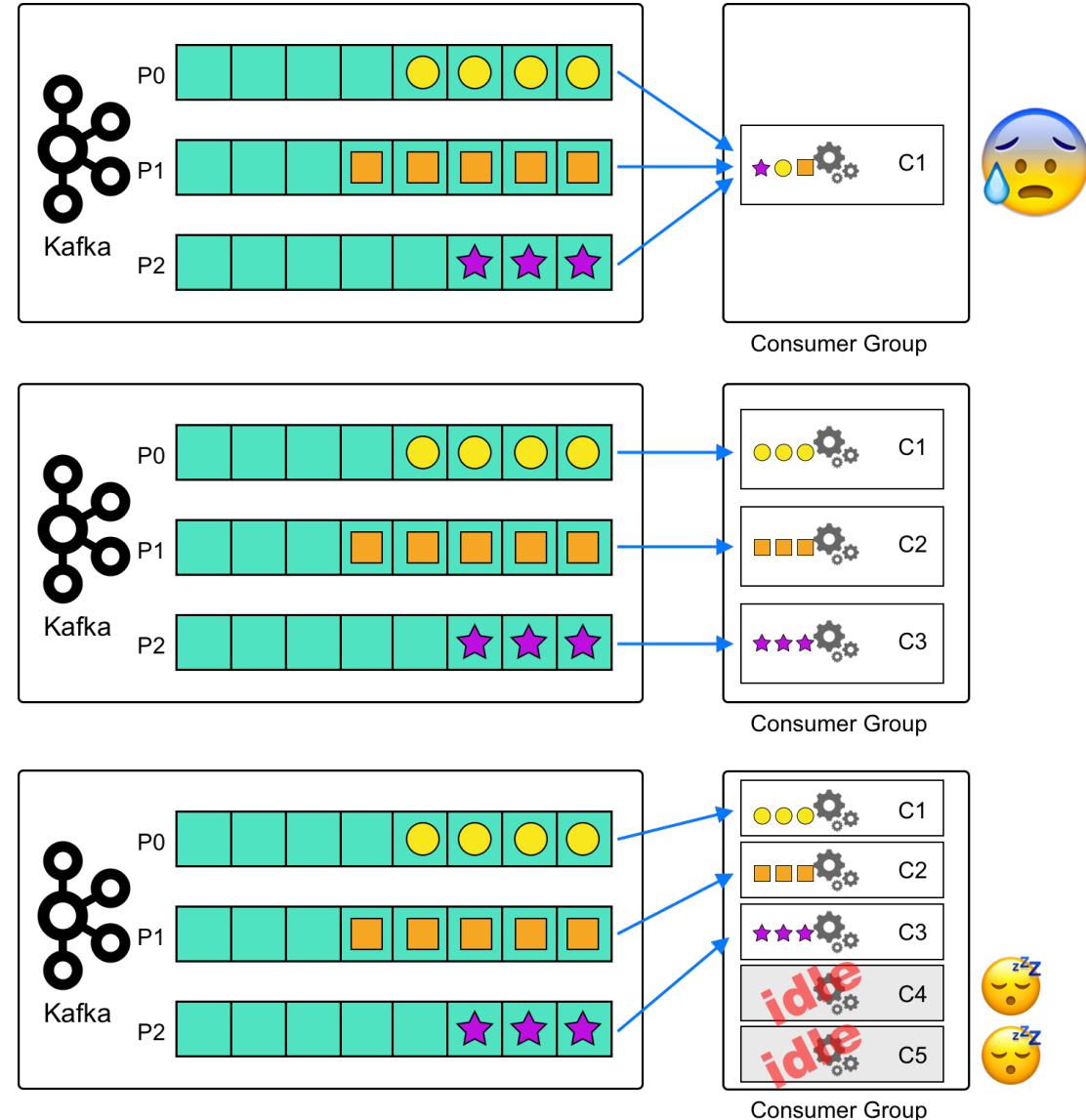
Topic driver-positions



Multiple Consumer Groups



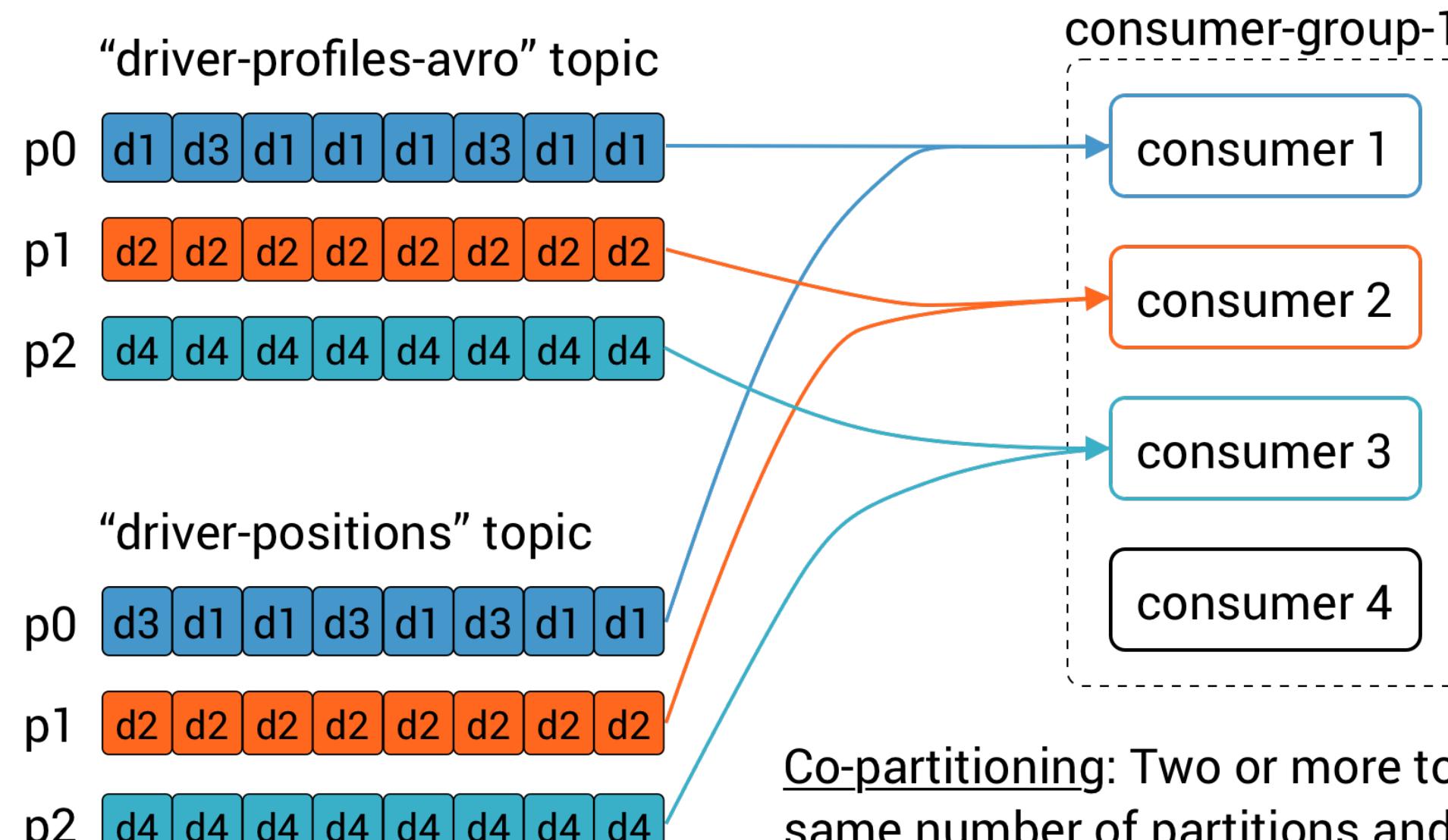
Scalability Limited by Number of Partitions



How are Partitions Assigned to Consumers in a Group?

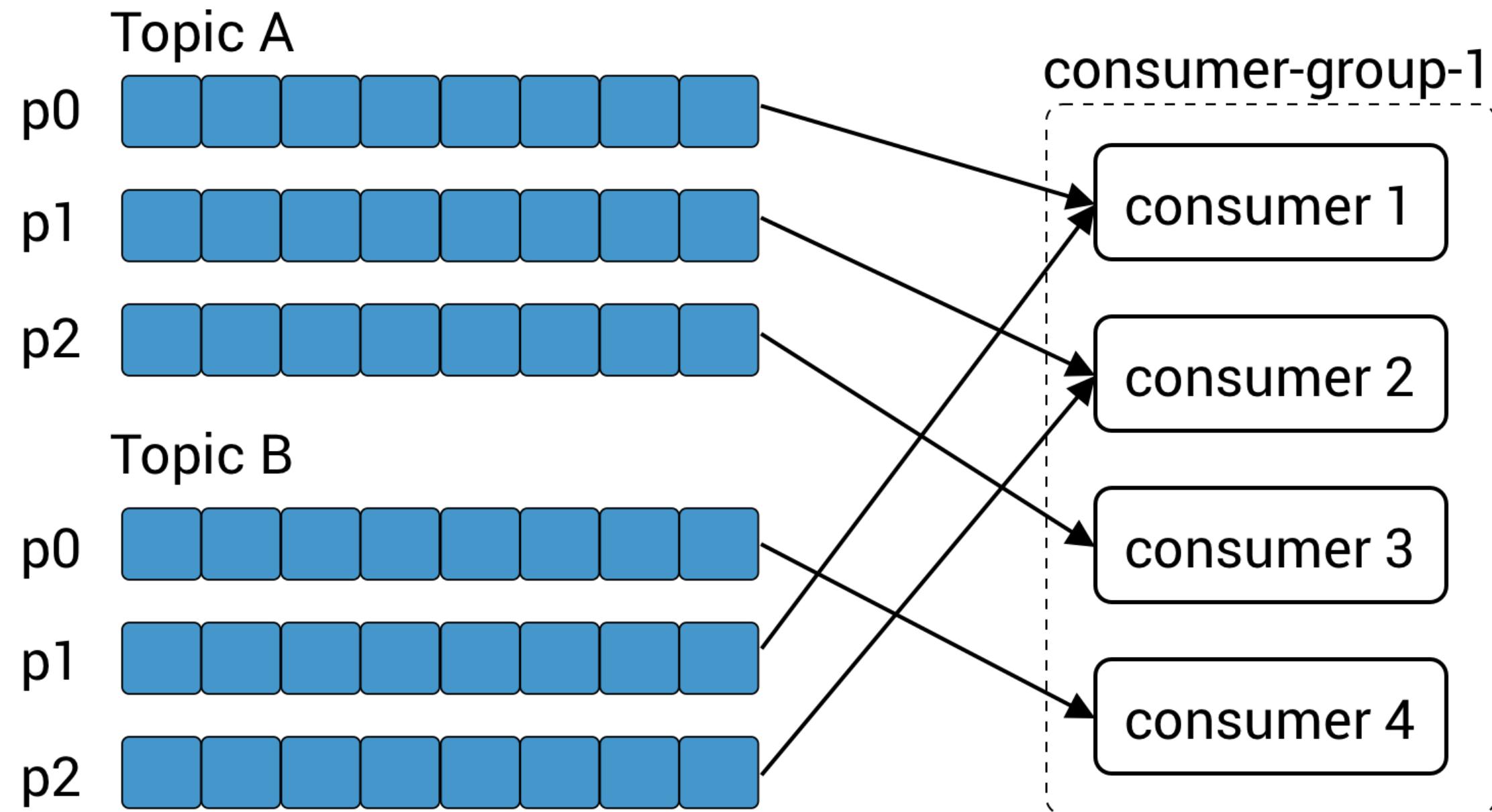
- Use RangeAssignor when joining data from multiple topics (this is the **default**):
`partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor`
- Use RoundRobin when performing stateless operations on records from many topics:
`partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor`
- Sticky is RoundRobin with a best effort to maintain assignments across rebalances:
`partition.assignment.strategy=org.apache.kafka.clients.consumer.StickyAssignor`
- CooperativeSticky is Sticky but it uses consecutive rebalances rather than the single stop-the-world used by Sticky:
`partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor`

Range Partition Assignment

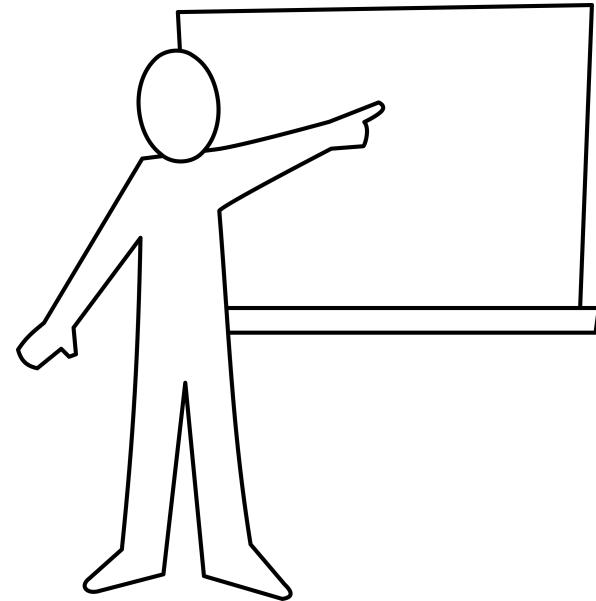


Co-partitioning: Two or more topics have same number of partitions and same partitioner, so that all records of a given key are consumed by the same consumer.

Round Robin and Sticky Partition Assignment

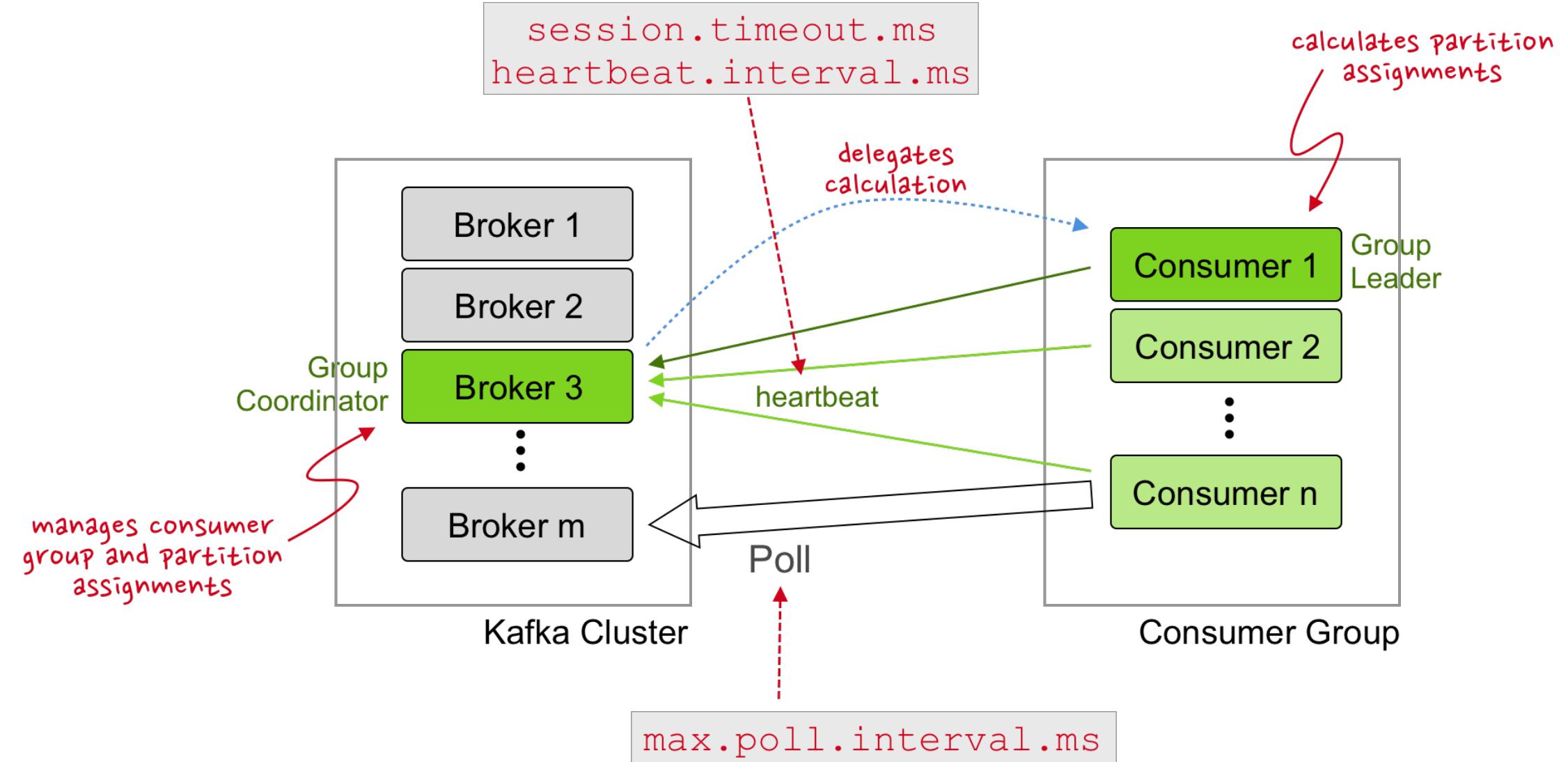


Module Map



- Consumer Groups and Partitions
- Consumer Group Rebalances ... ←
- Life of a KafkaConsumer
- Offset Management
- Performance Tuning
- Confluent REST Proxy
- 🚧 Hands-on Lab

Consumer Liveness



Consumer Rebalancing

Rebalance occurs when:

- Consumer **leaves or joins** group
- Consumer **changes topic subscription**
- Number of partitions changes

The Pros and Cons of Rebalance



- Consumer Group is **resilient to failures** of individual consumers
- Consumption **scales automatically** when consumers are added



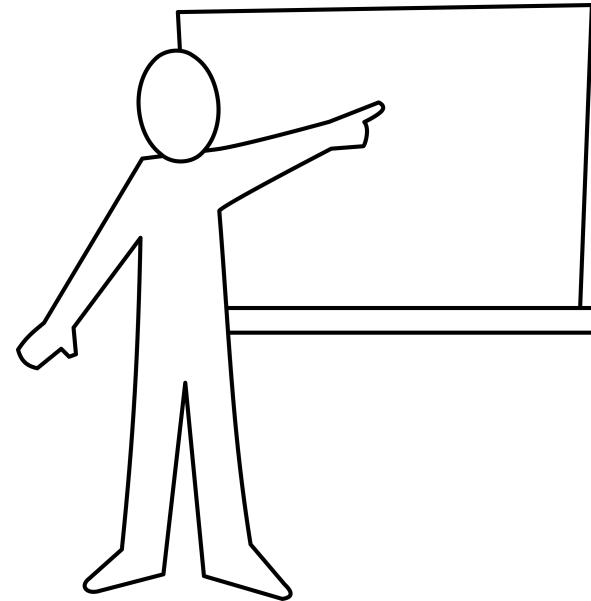
- Customize what consumers do when a rebalance occurs:

```
1 public interface ConsumerRebalanceListener {  
2     void onPartitionsAssigned(Collection<TopicPartition> partitions);  
3     void onPartitionsRevoked(Collection<TopicPartition> partitions);  
4 }
```

Avoiding Excessive Rebalances

- Tune `session.timeout.ms`:
 - set `heartbeat.interval.ms` to 1/3 `session.timeout.ms`
 - Pro: gives more time for Consumer to rejoin
 - Con: takes longer to detect hard failures
- Tune `max.poll.interval.ms`
 - Give Consumers enough time to process data from `poll()`
- Static group membership:
 - Assign each Consumer in Group unique `group.instance.id`
 - Consumers do not send `LeaveGroupRequest`
 - Rejoin doesn't trigger rebalance for known `group.instance.id`

Module Map



- Consumer Groups and Partitions
- Consumer Group Rebalances
- Life of a KafkaConsumer ... ←
- Offset Management
- Performance Tuning
- Confluent REST Proxy
- 🚧 Hands-on Lab

Important Consumer Properties (1)

Name	Description
<code>bootstrap.servers</code>	List of Broker host/port pairs used to establish the initial connection to the cluster
<code>key.deserializer</code>	Class used to deserialize the key. Must implement the <code>Deserializer</code> interface
<code>value.deserializer</code>	Class used to deserialize the value. Must implement the <code>Deserializer</code> interface

Important Consumer Properties (2)

Name	Description
<code>client.id</code>	String to identify this consumer uniquely; used in monitoring and logs
<code>group.id</code>	A unique string that identifies the Consumer Group this Consumer belongs to.
<code>enable.auto.commit</code>	When set to <code>true</code> (the default), the Consumer will trigger offset commits based on the value of <code>auto.commit.interval.ms</code> (default 5000ms)

Consumer Configuration

- Using Helper Classes

```
1 final Properties props = new Properties();
2
3 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
4 ... // Load other properties
5
6 KafkaConsumer<String, MyObject> consumer = new KafkaConsumer<>(props);
```

- Using a Properties File

```
1 final Properties props = new Properties();
2
3 InputStream propsFile = new FileInputStream("src/main/resources/consumer.properties");
4 props.load(propsFile);
5
6 KafkaConsumer<String, MyObject> consumer = new KafkaConsumer<>(props);
```

Subscribing to Topics

```
7 consumer.subscribe(Arrays.asList("my_topic", "my_other_topic"));
```

- Can subscribe to `1...n` Topics
- Calling `subscribe` again will replace existing list of topics
- Can use regular expressions for topic subscription

Reading Messages from Kafka with `poll()`

```
8 while (true) {  
9     ConsumerRecords<String, MyObject> records = consumer.poll(Duration.ofMillis(100));  
10    for (ConsumerRecord<String, MyObject> record : records)  
11        System.out.printf("offset = %d, key = %s, value = %s\n",  
12                           record.offset(), record.key(), record.value());  
13 }
```

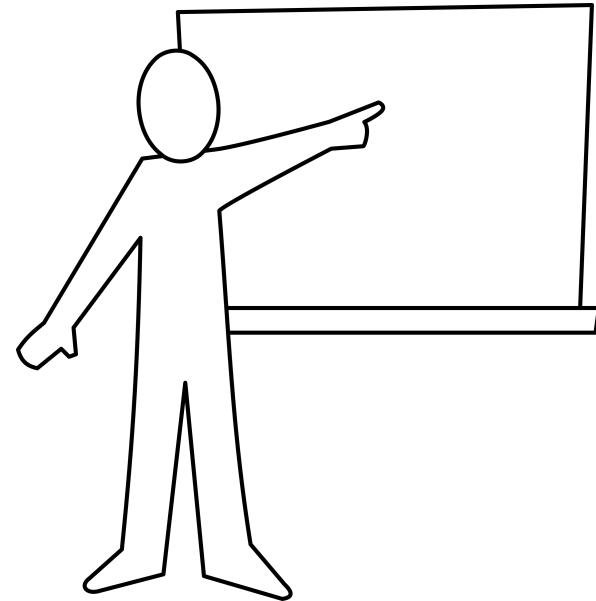
Closing the Consumer

```
14 try {
15     // loop forever
16     while (true) {
17         // poll for records
18         ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
19         // process each record
20         for (ConsumerRecord<String, String> record : records) {
21             System.out.printf("offset = %d, key = %s, value = %s\n",
22                               record.offset(), record.key(), record.value());
23         }
24     }
25 } finally {
26     // avoid resource leaks
27     consumer.close();
28 }
```



KafkaConsumer is **not thread-safe**

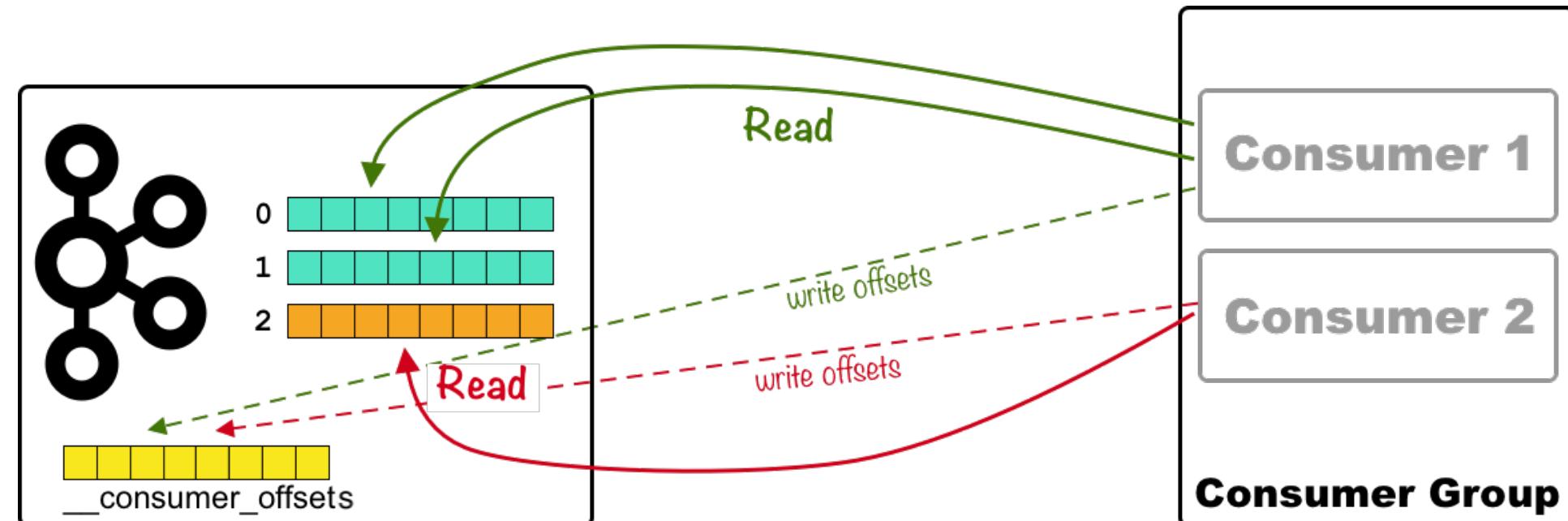
Module Map



- Consumer Groups and Partitions
- Consumer Group Rebalances
- Life of a KafkaConsumer
- Offset Management ... ←
- Performance Tuning
- Confluent REST Proxy
- 🛠️ Hands-on Lab

Consumers and Offsets

- **Offset:** Position of a message in the partition
- Kafka tracks where consumers are in each partition
- Consumer offsets **auto-committed** by default
- `(group.id, topic, partition)` is tracked in topic called `__consumer_offsets`
- Consumer Offset topic tracks which message **should be read next**



Manually Committing Offsets

- Set: `enable.auto.commit=false`

Synchronous

- `commitSync()`
- **blocks** until success or exception

Asynchronous

- `commitAsync()`
- **Non-blocking**
- Optional: add **callback**

Question

- What might be some tradeoffs between sync vs. async manual offset commits?

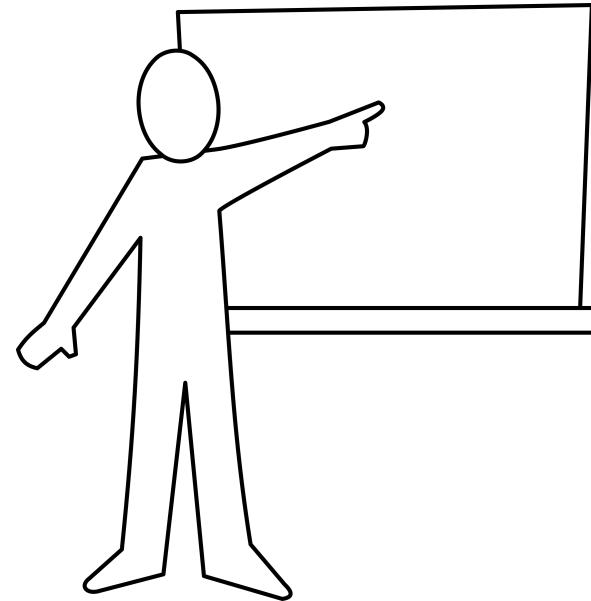
Processing Guarantees

```
1 records=consumer.poll();  
2 consumer.commitSync();  
3 ... // process records
```

```
1 records=consumer.poll();  
2 ... // process records  
3 consumer.commitSync();
```

Question: Which code block represents an "at most once" guarantee, and which represents "at least once"?

Module Map

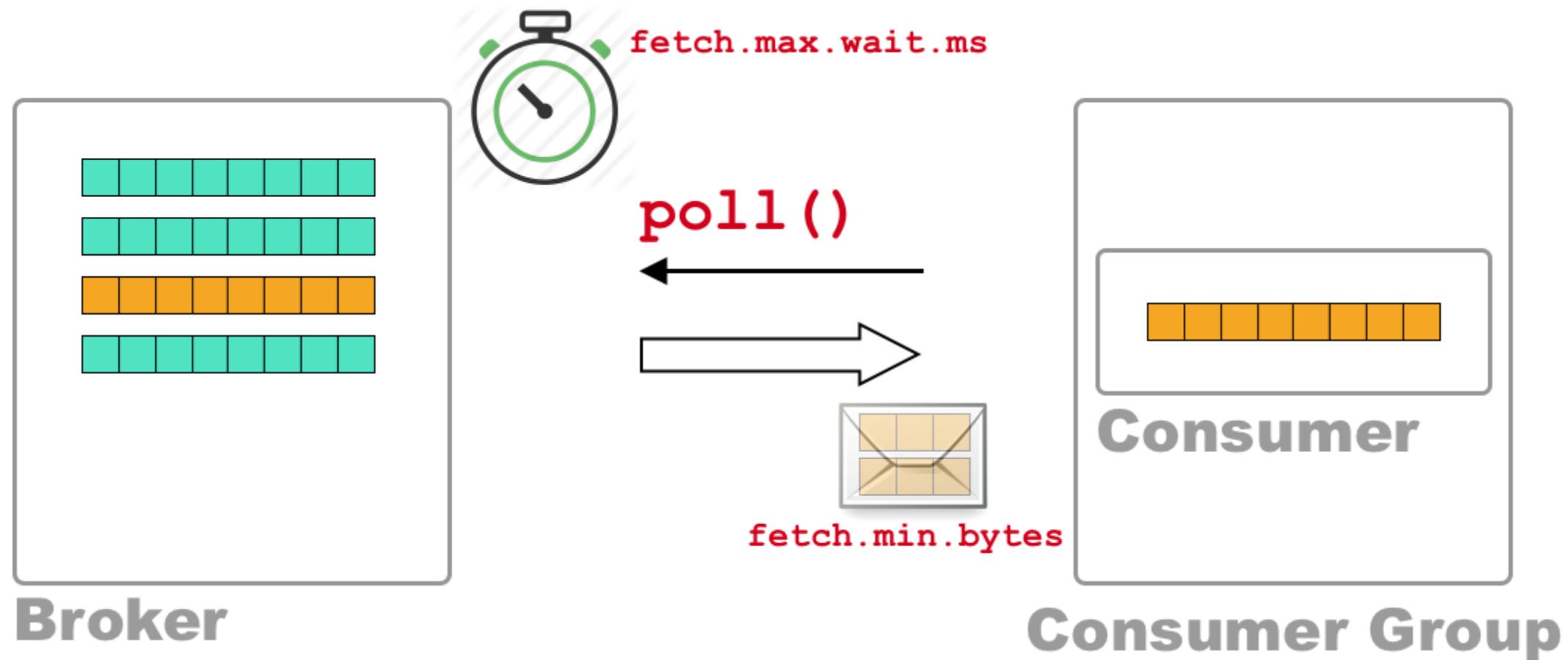


- Consumer Groups and Partitions
- Consumer Group Rebalances
- Life of a KafkaConsumer
- Offset Management
- Performance Tuning ... ←
- Confluent REST Proxy
- 🛠️ Hands-on Lab

Controlling the Number of Messages Returned

- `poll(Duration)` fetches from **multiple partitions** across **multiple brokers**
- Max. amount of data fetched **per partition**:
 - `max.partition.fetch.bytes` (default `1MB`)
- Alternatively set max number of records per poll across **all partitions** with:
 - `max.poll.records` (default `500`)

Performance Tuning Consumer Fetch Requests (1)



`fetch.max.wait.ms` defaults to 500 ms and `fetch.min.bytes` defaults to 1 byte

Performance Tuning Consumer Fetch Requests (2)

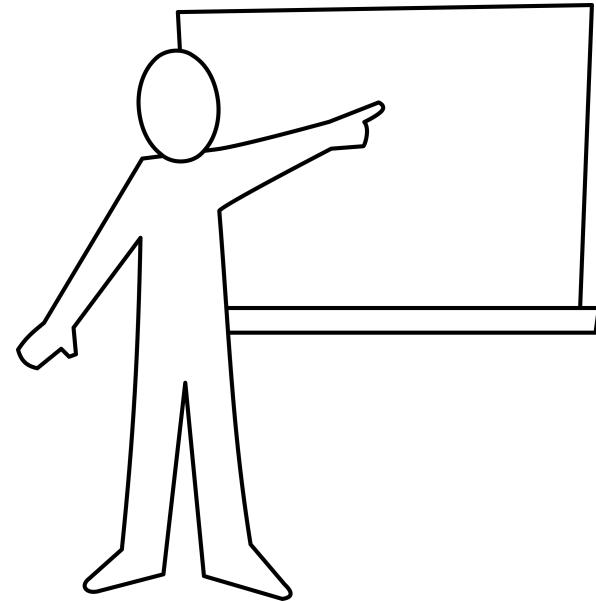
High throughput

- get more data in one batch
- large `fetch.min.bytes`
- reasonable `fetch.max.wait.ms`

Low latency

- get data as quickly as possible
- set `fetch.min.bytes=1` (default)

Module Map



- Consumer Groups and Partitions
- Consumer Group Rebalances
- Life of a KafkaConsumer
- Offset Management
- Performance Tuning
- Confluent REST Proxy ... ←
- 🛠️ Hands-on Lab

Creating a Consumer with REST Proxy (1)

- Main Logic

```
1 import requests
2 import json
3 import sys
4
5 FORMAT = "application/vnd.kafka.v2+json"
6 POST_HEADERS = { "Content-Type": FORMAT }
7 GET_HEADERS = { "Accept": FORMAT }
8
9 base_uri = create_consumer_instance("group1", "my_consumer")
10 subscribe_to_topic(base_uri, "hello_world_topic")
11 consume_messages(base_uri)
12 delete_consumer(base_uri)
```



This is just a Confluent REST Proxy example. This is not the Python consumer client.

Creating a Consumer with REST Proxy (2)

- Creating the Consumer Instance

```
13 def create_consumer_instance(group_name, instance_name):
14     url = f'http://rest-proxy:8082/consumers/{group_name}'
15     payload = {
16         "name": instance_name,
17         "format": "json"
18     }
19     r = requests.post(url, data=json.dumps(payload), headers=POST_HEADERS)
20
21     if r.status_code != 200:
22         print ("Status Code: " + str(r.status_code))
23         print (r.text)
24         sys.exit("Error thrown while creating consumer")
25
26     return r.json()["base_uri"]
```

Creating a Consumer with REST Proxy (3)

- Subscribing to a topic

```
27 def subscribe_to_topic(base_uri, topic_name):  
28     payload = {  
29         "topics": [topic_name]  
30     }  
31  
32     r = requests.post(base_uri + "/subscription",  
33                         data=json.dumps(payload),  
34                         headers=POST_HEADERS)  
35  
36     if r.status_code != 204:  
37         print("Status Code: " + str(r.status_code))  
38         print(r.text)  
39         delete_consumer(base_uri)  
40         sys.exit("Error thrown while subscribing the consumer to the topic")
```

Creating a Consumer with REST Proxy (4)

- Consuming and processing messages

```
41 def consume_messages(base_uri):  
42     r = requests.get(base_uri + "/records", headers=GET_HEADERS, timeout=20)  
43  
44     if r.status_code != 200:  
45         print ("Status Code: " + str(r.status_code))  
46         print (r.text)  
47         sys.exit("Error thrown while getting message")  
48  
49     for message in r.json():  
50         if message["key"] is not None:  
51             print ("Message Key:" + message["key"])  
52             print ("Message Value:" + message["value"])
```

Creating a Consumer with REST Proxy (5)

- Deleting the consumer

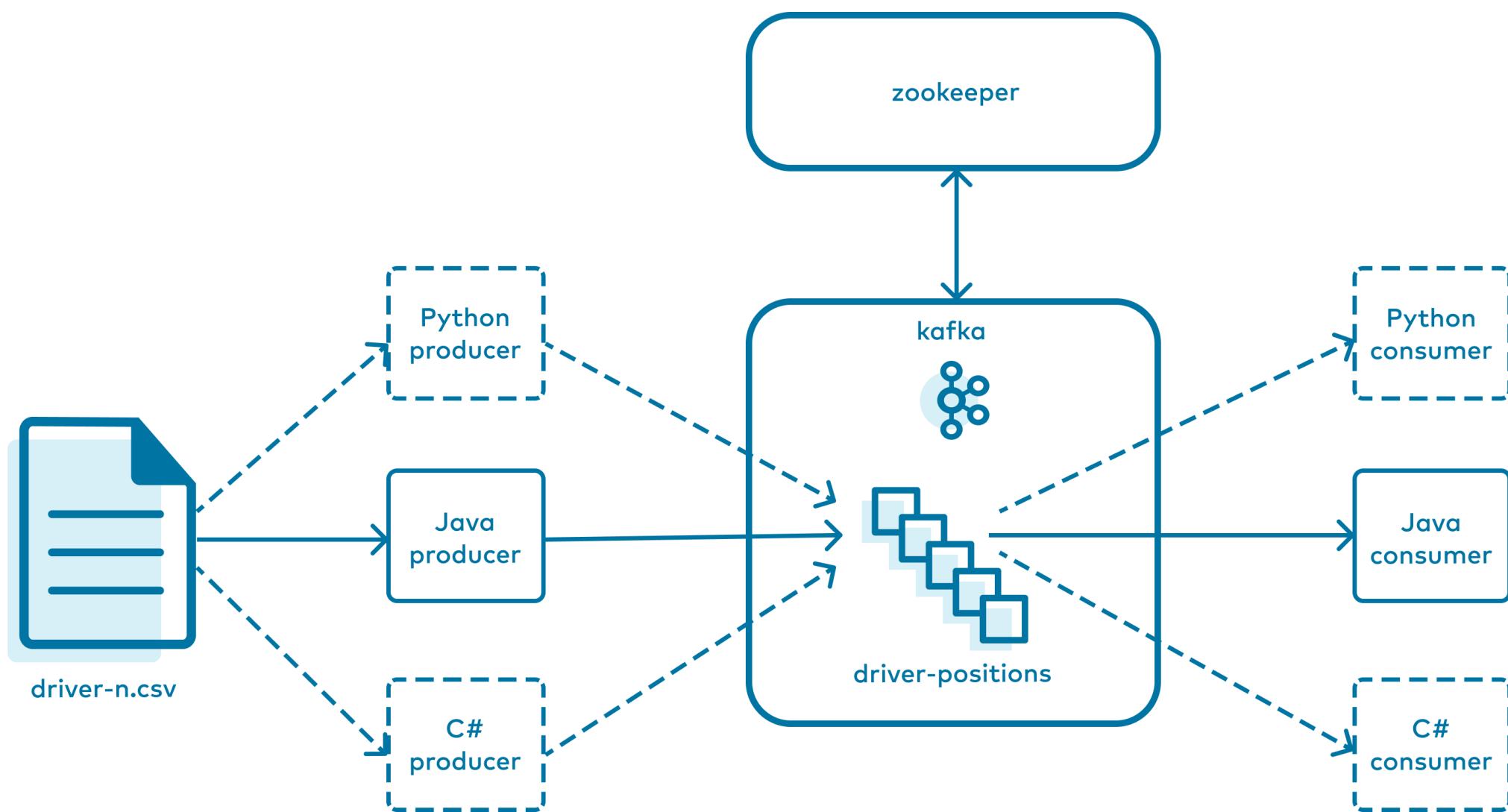
```
53 def delete_consumer(base_uri):  
54     r = requests.delete(base_uri, headers=POST_HEADERS)  
55  
56     if r.status_code != 204:  
57         print ("Status Code: " + str(r.status_code))  
58         print (r.text)
```

Hands-On Lab

- In this Hands-On Exercise, you will create a consumer to read driver location data
- Please refer to **Lab 04 Consuming Messages from Kafka** in the Exercise Book:
 - a. **Kafka Consumer**



Kafka Consumer (Java, C#, Python)



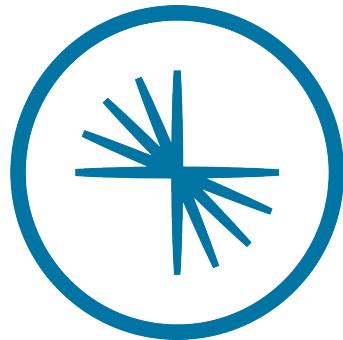
Module Review



Questions:

1. What could cause a consumer group rebalance?
2. How can you program what a consumer does upon rebalance?
3. A "temperatures" topic streams temperature readings for cities all around the world. An "AvgTemp" consumer application provides a running average for temperatures by city. How do consumer group rebalances affect this application?

05 Schema Management In Kafka



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka ... ←
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

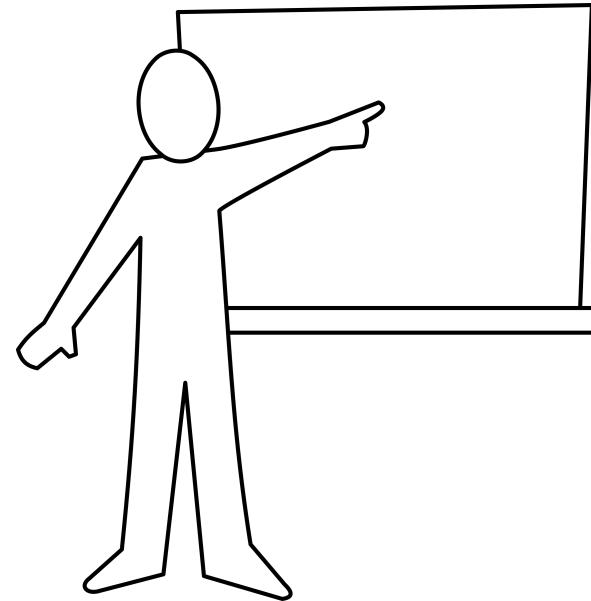
Learning Objectives



After this module you will be able to:

- describe Kafka schemas and how they work
- use the Confluent Schema Registry to guide schema evolution
- write and read messages using schema-enabled Kafka client tools

Module Map



- An Introduction to Schemas ... ←
- Avro, Protobuf, and JSON Schema formats
- Using Confluent Schema Registry
- 🌐 Hands-on Lab

The Need for a More Complex Serialization System

So far, all our data has been **plain text**...



PROS

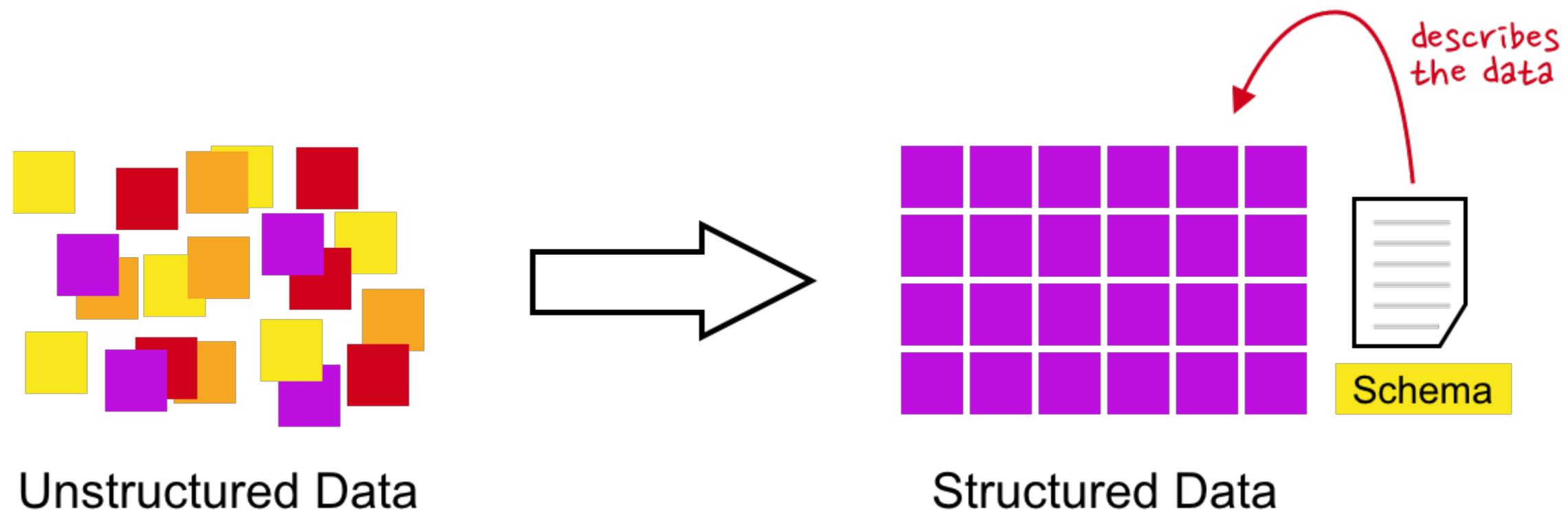
- Supported by most programming languages
- **Easy to inspect** files for debugging



CONS

- Data is stored **inefficiently**
- Non-text data must be converted to strings
 - No type checking is performed
 - It is inefficient to convert binary data to strings
- **No schema evolution**

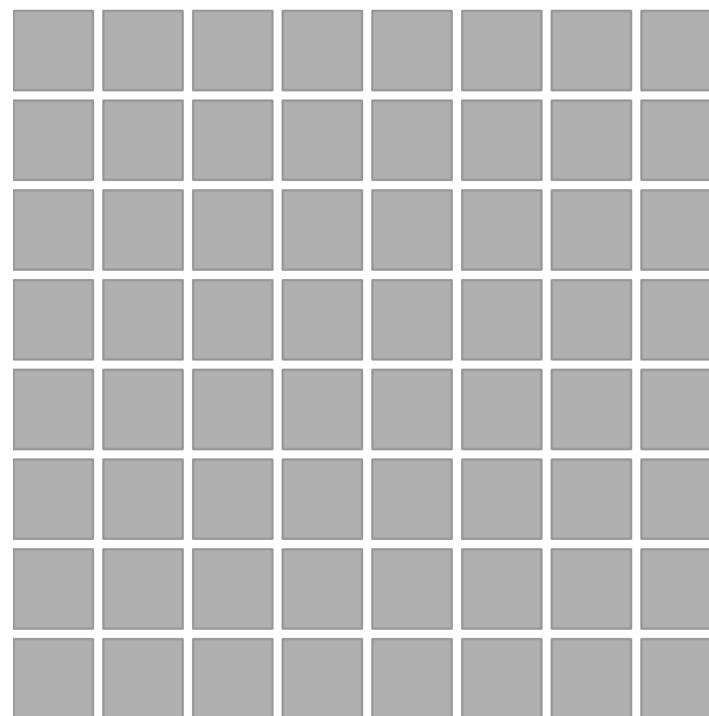
The Data Schema



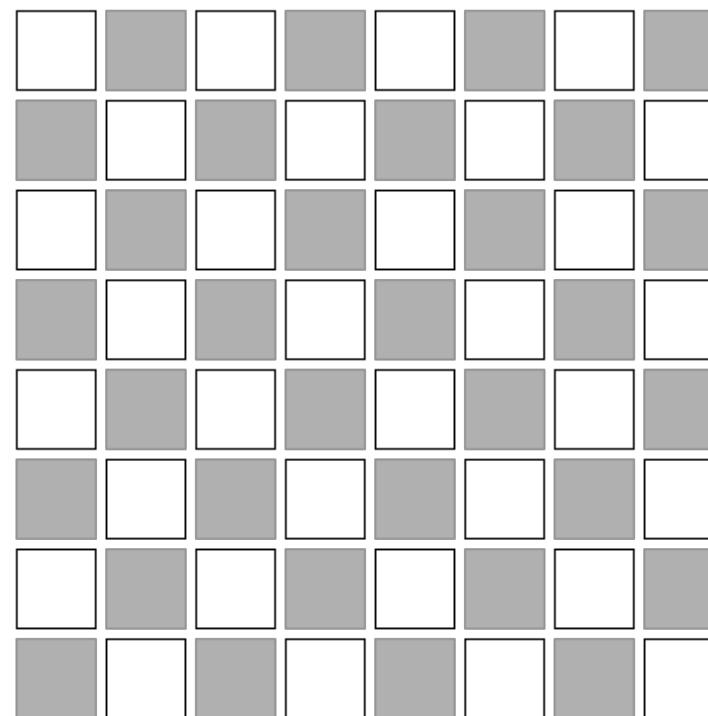
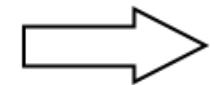
Data Schema Evolution

Always have a schema evolution plan in place...

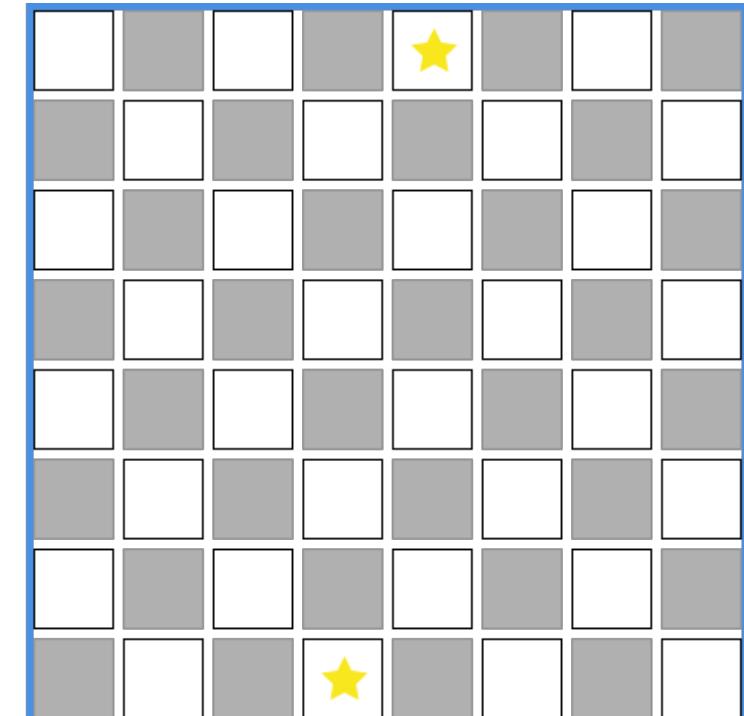
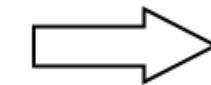
Chris Smith, VP, Engineering Data Science, Ticketmaster



Data Schema V1

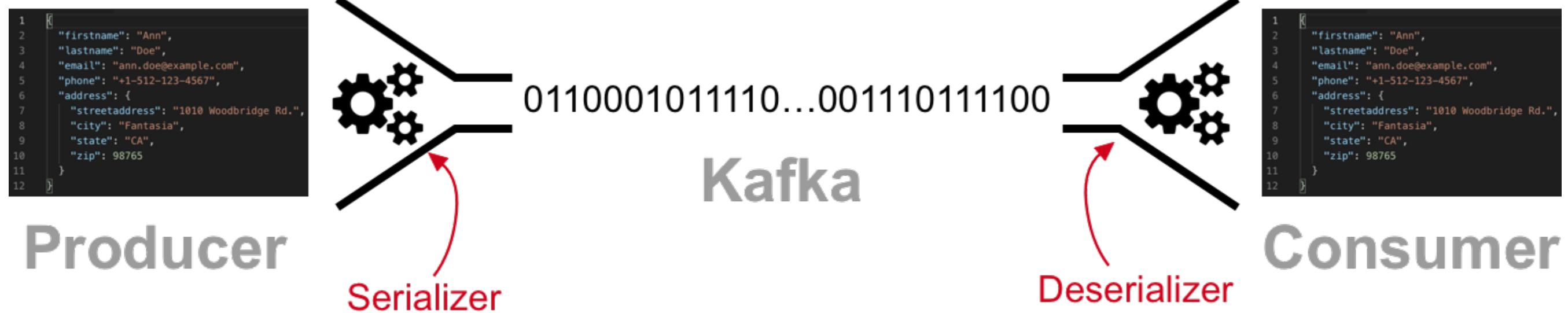


Data Schema V2



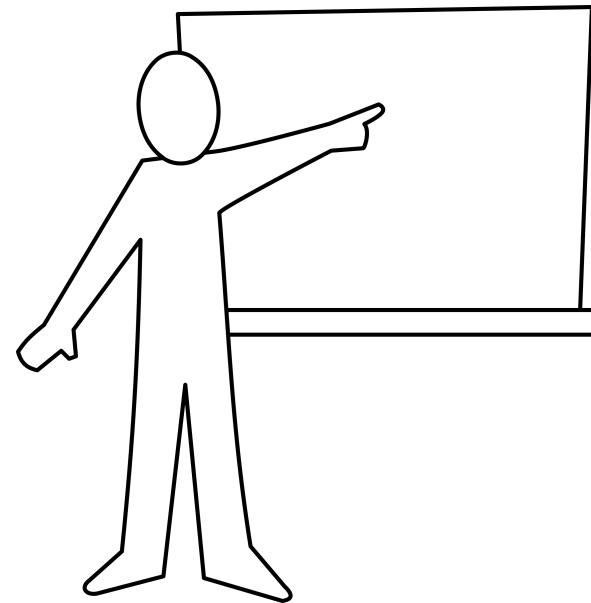
Data Schema V3

Serialization Review



Kafka has its own serialization classes in
[org.apache.kafka.common.serialization](https://github.com/apache/kafka/blob/trunk/avro-serializer/src/main/java/org/apache/kafka/common/serialization/AvroSerializer.java)

Module Map



- An Introduction to Schemas
- Avro, Protobuf, and JSON Schema ... ←
- Using Confluent Schema Registry
- 🌐 Hands-on Lab

Avro: An Efficient Data Serialization System

- Apache open source project
- Compact, fast, binary data format
- Supported by many programming languages, including Java
- Define schemas in JSON
- Supports code generation for objects



```
{  
  "namespace": "clients.avro",  
  "type": "record",  
  "name": "PositionValue",  
  "fields": [  
    {"name": "latitude", "type": "double"},  
    {"name": "longitude", "type": "double"}  
  ]  
}
```

Avro Schemas

- Avro schemas can be used to define the structure of your data
- Schemas are represented in **JSON** or **IDL** format
- Avro supports three different ways of creating records:

Generic

1. Manually write schema
2. Manually create `GenericRecord` object from schema

Use for quickly changing or highly variable schemas

Reflection

1. Manually create data type
2. Generate schema from code

Specific

1. Manually write schema
2. Generate classes from schema to include in your program

Most common way to use Avro classes

Simple Avro Data Types (1)

Name	Description	Java equivalent
boolean	True or false	boolean
int	32-bit signed integer	int
long	64-bit signed integer	long
float	Single-precision floating-point number	float
double	Double-precision floating-point number	double

Simple Avro Data Types (2)

Name	Description	Java equivalent
string	Sequence of Unicode characters	<code>java.lang.CharSequence</code>
bytes	Sequence of bytes	<code>java.nio.ByteBuffer</code>
null	The absence of a value	<code>null</code>

Complex Avro Data Types

Name	Description
record	A user-defined field comprising one or more simple or complex data types, including nested records
enum	Exactly one value from a specified set of values
union	Exactly one data type from a specified set of types
array	Zero or more values, each of the same type
map	Set of key/value pairs; key is always a <code>string</code> , value is the specified type
fixed	A fixed number of bytes
logical	Avro primitive or complex type with extra attributes to represent a derived type

Example Avro Schema (1)

- Schema of a car's PositionValue:

```
1 {
2   "namespace": "clients.avro",
3   "type": "record",
4   "name": "PositionValue",
5   "fields": [
6     {"name": "latitude", "type": "double" },
7     {"name": "longitude", "type": "double" }
8   ]
9 }
```

Example Avro Schema (2)

- Schema of a `TrainArrived` event:

```
1 {
2   "namespace": "com.traincompany.examples",
3   "type": "record",
4   "name": "TrainArrived",
5   "fields": [
6     { "name": "trainId", "type": "int" },
7     { "name": "stationId", "type": "int" },
8     { "name": "arrivalTime",
9       "type": "int",
10      "doc": "Time in ms since the epoch"}
11   ]
12 }
```

Schema Naming Conventions

- By default, the schema definition is placed in `src/main/avro`
 - File extension is `.avsc`
- The `namespace` is the Java package name, which you will import into your code

Example Avro Schema: array

```
1 {
2   "namespace": "example.avro",
3   "type": "record",
4   "name": "Salesperson",
5   "fields": [
6     { "name": "name", "type": "string",
7       "doc": "employee name"},,
8     { "name": "AccountList",
9       "type": {
10         "type": "array",
11         "items":{
12           "name": "Account",
13           "type": "record",
14           "fields": [
15             { "name": "id", "type": "string" },
16             { "name": "email", "type": "string" }
17           ...
18         }
19       }
20     }
21   ]
22 }
```

Example Avro Schema: map

```
1 {
2   "namespace": "example.avro",
3   "type": "record",
4   "name": "Log",
5   "fields": [
6     { "name": "ip",
7       "type": "string" },
8     { "name": "timestamp",
9       "type": "string" },
10    { "name": "message",
11      "type": "string" },
12    {
13      "name": "additional",
14      "type": {
15        "type": "map",
16        "values": "string"
17      ...
}
```

Example Avro Schema: enum

```
1  {
2      "type": "record",
3      "name": "Card",
4      "namespace": "com.acme",
5      "fields": [
6          {"name": "face_value",
7              "type": "int"},
8          {"name": "suit_type",
9              "type" : {
10                  "type" : "enum",
11                  "name" : "Suit",
12                  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]},
13                  "doc" : "The suit of the card"
14          }
15      ]
16 }
```

Logical Data Types

Decimal

```
{  
  "name": "myDecimal",  
  "type": {  
    "type": "bytes",  
    "logicalType": "decimal",  
    "precision": 4,  
    "scale": 2  
  }  
}
```

Timestamp (ms precision)

```
{  
  "name": "longTime",  
  "type" : {  
    "type": "long",  
    "logicalType": "timestamp-millis"  
  }  
}
```

Default and Null Values

Default Value

```
{  
  "name": "suit_type",  
  "type" : {  
    "type" : "enum",  
    "name" : "Suit",  
    "symbols" : ["SPADES", "HEARTS",  
                "DIAMONDS", "CLUBS"],  
    "default": "SPADES"  
  },  
}
```

Null Value

```
{  
  "name" : "experience",  
  "type": ["null", "int"]  
}
```

Protobuf and JSON Schema

- In addition to Avro, as of Confluent Platform 5.5, Confluent Schema Registry now supports the use of Protobuf and JSON Schema formats

Protobuf:

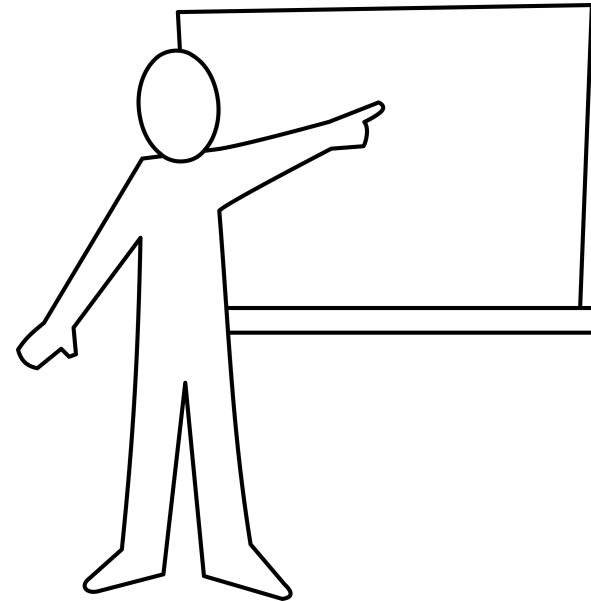
```
syntax = "proto3";

option java_package =
"clients.proto";
message PositionValue {
    double latitude = 1;
    double longitude = 2;
}
```

JSON Schema:

```
{
    "type": "object",
    "title": "driverposition",
    "properties": {
        "latitude": { "type": "number" },
        "longitude": { "type": "number" }
    }
}
```

Module Map



- An Introduction to Schemas
- Avro, Protobuf, and JSON Schema
- Using Confluent Schema Registry ... ←
- 🌐 Hands-on Lab

What Is Confluent Schema Registry?

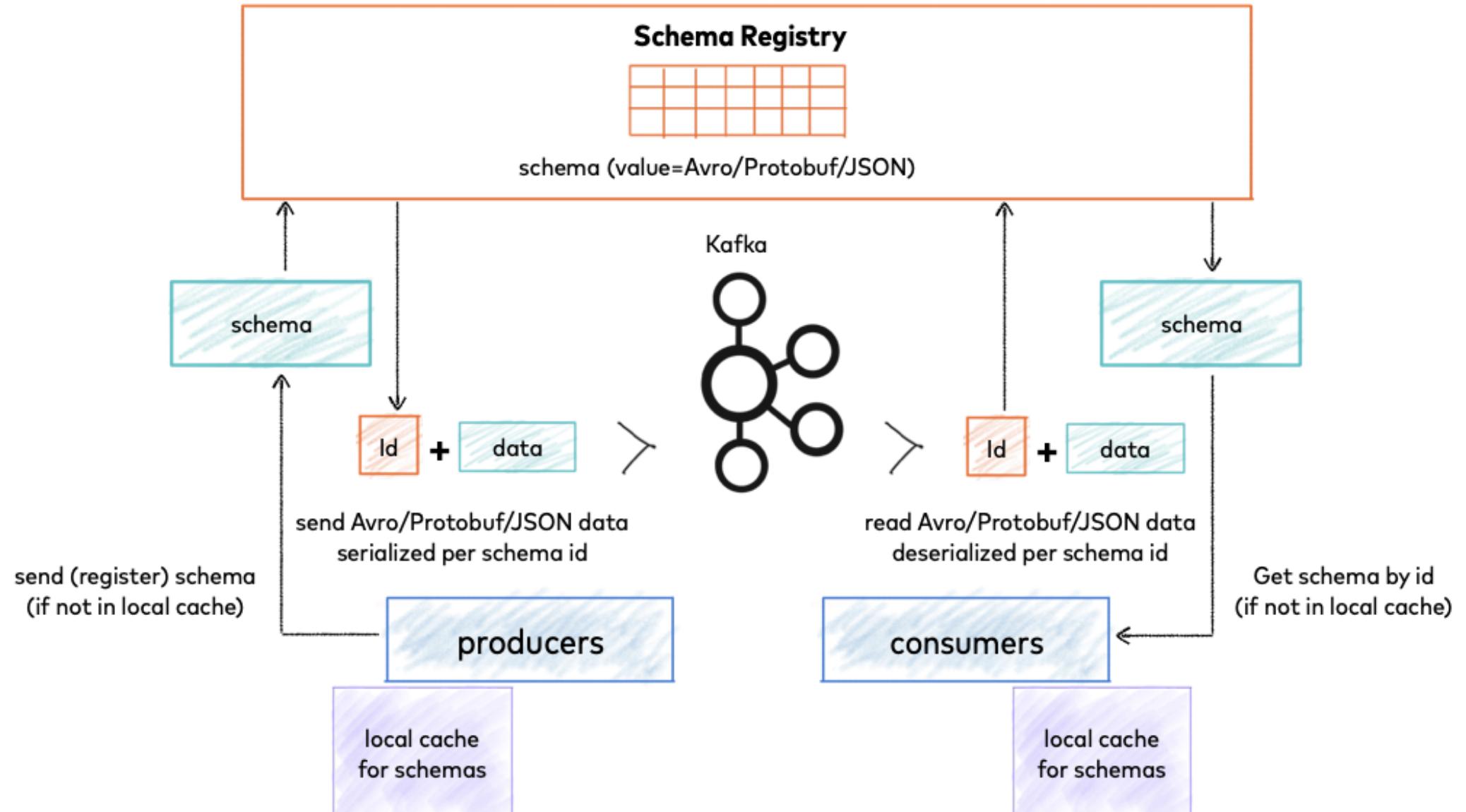
- Confluent Schema Registry provides centralized management of schemas
 - Stores a **versioned history** of all schemas
 - Provides a **RESTful interface** for storing and retrieving schemas
 - **Checks schemas** and throws an exception if data does not conform to the schema
 - Allows **evolution** of schemas according to the configured compatibility setting
- Sending the schema with each message would be inefficient
 - Instead, a globally unique ID representing the schema is sent with each message
- The Schema Registry stores schema information in a special Kafka topic
- The Schema Registry is accessible both via a REST API and a Java API
- There are also command-line tools:

`kafka-avro-console-producer` and `kafka-avro-console-consumer`

`kafka-protobuf-console-producer` and `kafka-protobuf-console-consumer`

`kafka-json-schema-console-producer` and `kafka-json-schema-console-consumer`

Schema Registration and Data Flow



Integration with Schema Registry

- Java clients (producer, consumer)
- ksqlDB
- Kafka Streams
- Kafka Connect
- Confluent REST Proxy
- Non-Java clients based on `librdkafka`
(except Go)

Schema Registry's REST API (POST)

```
# Register a schema for keys
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}"}' \
http://schemaregistry1:8081/subjects/<topic-name>-key/versions
```

HTTP/1.1 200 OK

Content-Type: application/vnd.schemaregistry.v1+json
{"id":1}

```
# Register another schema for values
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\"type\": \"string\"}"}' \
http://schemaregistry1:8081/subjects/<topic-name>-value/versions
```

HTTP/1.1 200 OK

Content-Type: application/vnd.schemaregistry.v1+json
{"id":2}

Schema Registry's REST API (GET)

```
# Retrieve the schema with id 1
$ curl -X GET http://schemaregistry1:8081/schemas/ids/1
```

HTTP/1.1 200 OK

Content-Type: application/vnd.schemaregistry.v1+json
{"schema": "{\"type\": \"string\"}"}

```
# Check all the different schema subjects stored in Schema Registry
$ curl -X GET http://schemaregistry1:8081/subjects
```

HTTP/1.1 200 OK

Content-Type: application/vnd.schemaregistry.v1+json
["my_topic-key", "my_topic-value"]

Schema Evolution

- Schemas may evolve as updates to code happen
 - Schema Registry allows schema evolution
- Schema Registry enforces the Subject compatibility setting when a new schema version is registered
 - Versions can be backward, forward, or fully compatible
- Set Subject schema compatibility based upon its anticipated schema evolution
 - Default is BACKWARD
- Different schema formats have different compatibility rules. For more details, see:
<https://docs.confluent.io/current/schema-registry/serdes-develop/index.html#sr-serdes-schemas-compatibility-checks>

Configuring Schema Compatibility

```
# This example disables compatibility checking using "NONE"  
$ curl -X PUT -i -H "Content-Type: application/vnd.schemaregistry.v1+json" \  
--data '{"compatibility": "NONE"}' \  
http://schemaregistry1:8081/config/my_topic-value
```

HTTP/1.1 200 OK

Content-Type: application/vnd.schemaregistry.v1+json
{"compatibility": "NONE"}

Compatibility Types (1)

Compatibility Type	Schema Resolution	Upgrade first
BACKWARD	Consumers using the new schema can read data produced with the previous schema	Consumers
FORWARD	Consumers using the previous schema can read data produced with the new schema	Producers
FULL	The combination of BACKWARD and FORWARD compatibility	Any order
NONE	Compatibility checking disabled	Any order

Compatibility Types (2)

Compatibility Type	Schema Resolution	Upgrade first
BACKWARD_TRANSITIVE	Consumers using the new schema can read data produced with all previous schema	Consumers
FORWARD_TRANSITIVE	Consumers using a previous schema can read data produced with all new schema	Producers
FULL_TRANSITIVE	The combination of BACKWARD_TRANSITIVE and FORWARD_TRANSITIVE compatibility	Any order

Schema Compatibility Challenge 1

Schema V1

```
{  
    "namespace": "example.avro",  
    "type": "record",  
    "name": "user",  
    "fields": [  
        {"name": "firstname", "type": "string"},  
        {"name": "lastname", "type": "string"},  
        {"name": "age", "type": "int", "default": -1}  
    ]  
}
```

Backward



Schema V2

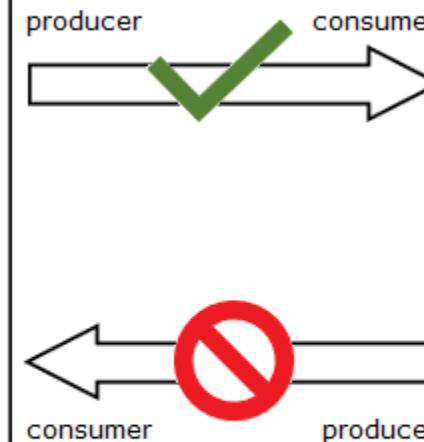
```
{  
    "namespace": "example.avro",  
    "type": "record",  
    "name": "user",  
    "fields": [  
        {"name": "lastname", "type": "string"},  
        {"name": "age", "type": "int", "default": -1},  
        {  
            "name": "hobby",  
            "type": "string",  
            "default": ""  
        }  
    ]  
}
```

BACKWARD Compatibility

Schema V1

```
{  
    "namespace": "example.avro",  
    "type": "record",  
    "name": "user",  
    "fields": [  
        {"name": "firstname", "type": "string"},  
        {"name": "lastname", "type": "string"},  
        {"name": "age", "type": "int", "default": -1}  
    ]  
}
```

Backward



Schema V2

```
{  
    "namespace": "example.avro",  
    "type": "record",  
    "name": "user",  
    "fields": [  
        {"name": "lastname", "type": "string"},  
        {"name": "age", "type": "int", "default": -1},  
        {  
            "name": "hobby",  
            "type": "string",  
            "default": ""  
        }  
    ]  
}
```

Forward

- Default value for `hobby` allows consumer using V2 to process messages produced with V1, i.e. schema V2 is **BACKWARD** compatible with V1
- No default value for `firstname` means consumer using V1 cannot process messages produced with V2, i.e. schema V2 is **not FORWARD** compatible with V1

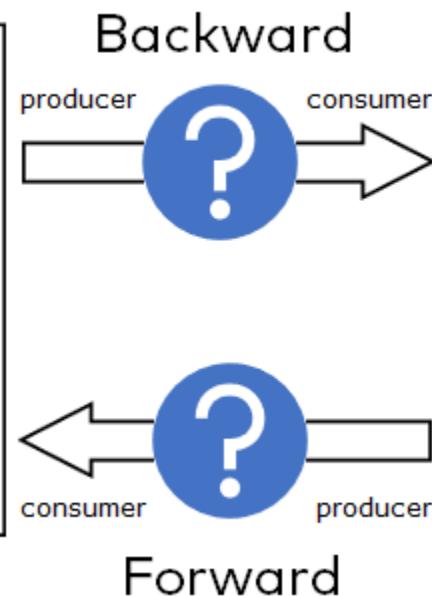
Schema Compatibility Challenge 2

Schema V1

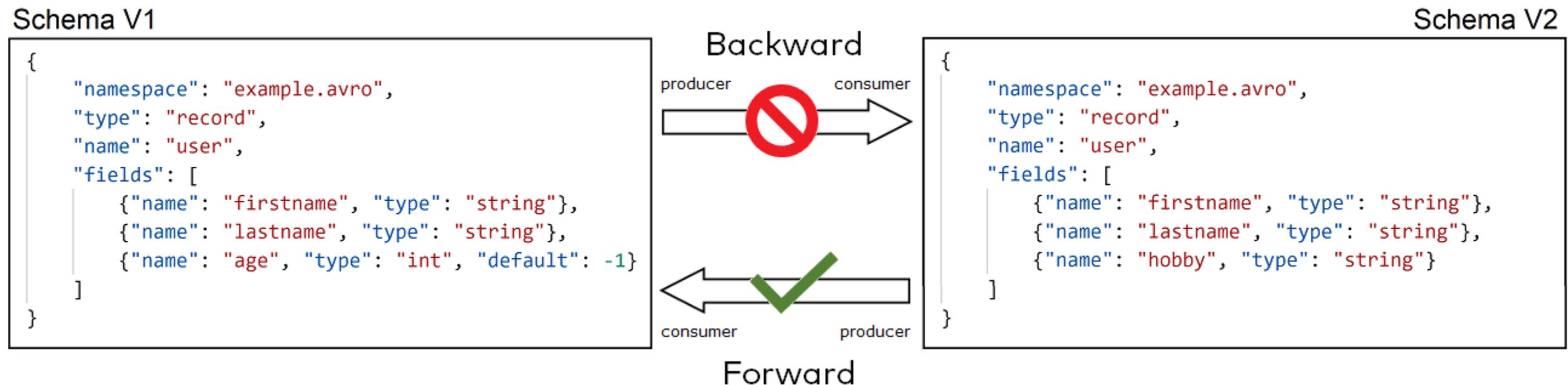
```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "user",  
  "fields": [  
    {"name": "firstname", "type": "string"},  
    {"name": "lastname", "type": "string"},  
    {"name": "age", "type": "int", "default": -1}  
  ]  
}
```

Schema V2

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "user",  
  "fields": [  
    {"name": "firstname", "type": "string"},  
    {"name": "lastname", "type": "string"},  
    {"name": "hobby", "type": "string"}  
  ]  
}
```



FORWARD Compatibility



- Default value for `age` allows consumer using V1 to process messages produced with V2, i.e. schema V2 is **FORWARD** compatible with V1
- No default value for `hobby` means consumer using V2 cannot process messages produced with V1, i.e. schema V2 is **not BACKWARD** compatible with V1

Schema Compatibility Challenge 3

Schema V1

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "user",  
  "fields": [  
    {"name": "firstname", "type": "string"},  
    {"name": "lastname", "type": "string"},  
    {"name": "age", "type": "int", "default": -1}  
  ]  
}
```

Backward

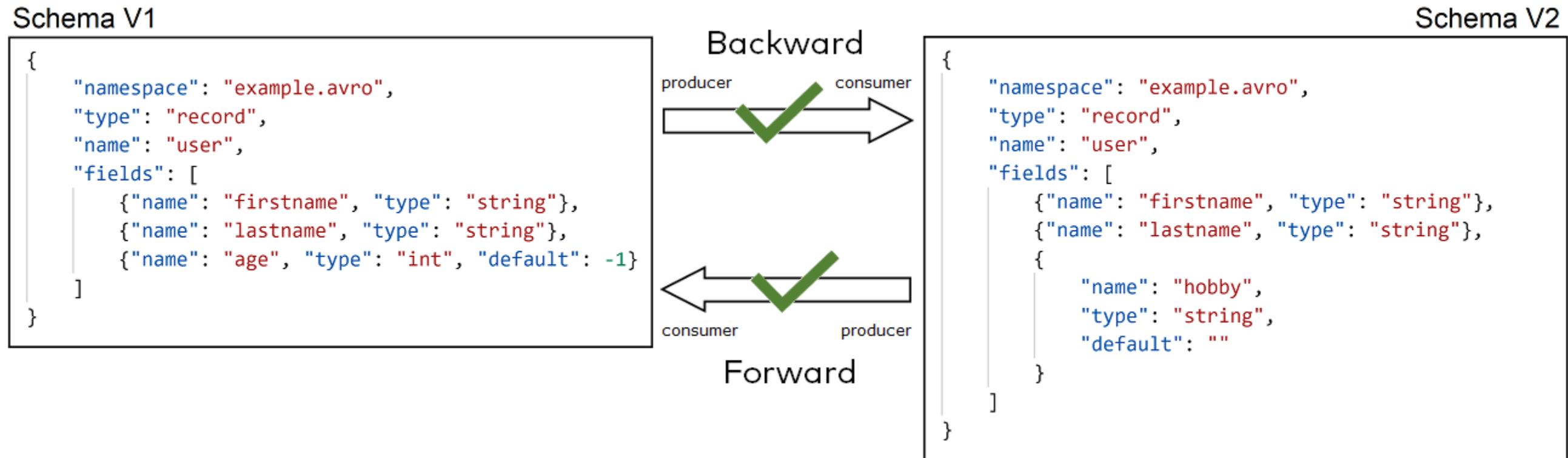


Schema V2

```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "user",  
  "fields": [  
    {"name": "firstname", "type": "string"},  
    {"name": "lastname", "type": "string"},  
    {  
      "name": "hobby",  
      "type": "string",  
      "default": ""  
    }  
  ]  
}
```

Forward

FULL Compatibility



- Default value for `age` allows consumer using V1 to process messages produced with V2, i.e. schema V2 is **FORWARD** compatible with V1
- Default value for `hobby` means consumer using V2 can process messages produced with V1, i.e. schema V2 is **BACKWARD** compatible with V1
- Thus, schemas V1 and V2 are **FULL** compatible

Subject Naming Strategies

- **Subject:** scope where schemas can evolve in Schema Registry

Naming Strategies	Configurations
<ul style="list-style-type: none">• TopicNameStrategy (default)• RecordNameStrategy• TopicRecordNameStrategy	<ul style="list-style-type: none">• key.subject.name.strategy• value.subject.name.strategy

- TopicNameStrategy example

Topic: driver-positions

Subjects: driver-positions-key
driver-positions-value

Java Avro Producer Example

```
1 Properties props = new Properties();
2 props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
3 // Configure serializer classes
4 props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
5 props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
6 // Configure schema repository server
7 props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schema-registry1:8081");
8 // Create the producer, which expects PositionValue object generated from Avro schema
9 final KafkaProducer<String, PositionValue> producer = new KafkaProducer<>(props);
10 // Create the key (String) and value (PositionValue object generated from Avro schema)
11 final String key = "driver-1";
12 final PositionValue value = new PositionValue(47.618580396045445, -122.35454111509547);
13 // Create the ProducerRecord and send it
14 final ProducerRecord<String, PositionValue> record = new ProducerRecord<>(
15     "driver-positions-avro", key, value);
16 producer.send(record)
```

Java Avro Consumer Example

```
1 Properties props = new Properties();
2 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
3 props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
4 props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
5 props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class);
6 props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schemaregistry1:8081");
7 props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
8
9 KafkaConsumer<String, PositionValue> consumer = new KafkaConsumer<>(props);
10 consumer.subscribe(Arrays.asList("driver-positions-avro"));
11
12 while (true) {
13     ConsumerRecords<String, PositionValue> records = consumer.poll(Duration.ofMillis(100));
14     for (ConsumerRecord<String, PositionValue> record : records) {
15         System.out.printf("Key:%s Latitude:%s Longitude:%s [partition %s]\n",
16             record.key(), record.value().getLatitude(),
17             record.value().getLongitude(), record.partition());}}
```

REST Proxy Avro Producer Example

```
1 # Read in the Avro schema files
2 key_schema = open("my_key.avsc", 'rU').read()
3 value_schema = open("my_value.avsc", 'rU').read()
4
5 producerurl = "http://rest-proxy:8082/topics/my_avro_topic"
6 headers = {"Content-Type" : "application/vnd.kafka.avro.v2+json"}
7 payload = {
8     "key_schema": key_schema,
9     "value_schema": value_schema,
10    "records": [
11        {"key": {"suit": "spades"}, 
12         "value": {"suit": "spades", "denomination": "ace"}}
13    ]
14 }
15 # Send the message
16 r = requests.post(producerurl, data=json.dumps(payload), headers=headers)
17 print("Status Code: ", r.status_code, "\n", r.text if r.status_code != 200)
```

REST Proxy Avro Consumer Example

```
1 headers = {"Accept" : "application/vnd.kafka.avro.v2+json"}  
2 # Request records from the consumer instance  
3 r = requests.get(base_uri + "/records", headers=headers, timeout=20)  
4 if r.status_code != 200:  
5     print("Status Code: ", r.status_code, "\n", r.text)  
6     sys.exit("Error thrown while getting message")  
7 # Process the records  
8 for message in r.json():  
9     keysuit = message["key"]["suit"]  
10    valuesuit = message["value"]["suit"]  
11    denomination = message["value"]["denomination"]  
12    if keysuit != valuesuit:  
13        print("This card has two different suits! Impossible!")  
14        continue  
15    else:  
16        doStuff(keysuit, denomination)
```

AVRO Command-line Producer Tool

```
$ my_schema='{
  "namespace": "clients.avro",
  "type": "record",
  "name": "PositionValue",
  "fields": [
    {"name": "latitude", "type": "double" },
    {"name": "longitude", "type": "double" }
  ]
}'
```

```
$ kafka-avro-console-producer \
  --bootstrap-server kafka:9092 \
  --property schema.registry.url=schema-registry:8081 \
  --property value.schema="$my_schema" \
  --topic driver-positions-avro
```

AVRO Command-line Consumer Tool

```
$ kafka-console-consumer \
--bootstrap-server kafka:9092 \
--property schema.registry.url=schema-registry:8081 \
--from-beginning \
--topic driver-positions-avro
```

Protobuf Command-line Producer Tool

```
$ my_schema='
syntax = "proto3";
option java_package = "clients.proto";
message PositionValue {
    double latitude = 1;
    double longitude = 2;
}'
```

```
$ kafka-protobuf-console-producer \
--bootstrap-server kafka:9092 \
--property schema.registry.url=schema-registry:8081 \
--property value.schema=$my_schema \
--topic driver-positions-protobuf
```

Protobuf Command-line Consumer Tool

```
$ kafka-protobuf-console-consumer \
  --bootstrap-server kafka:9092 \
  --property schema.registry.url=http://schema-registry:8081 \
  --from-beginning \
  --topic driver-positions-protobuf
```

JSON Schema Command-line Producer Tool

```
$ my_schema='{
  "type": "object",
  "title": "driverposition",
  "properties": {
    "latitude": { "type": "number" },
    "longitude": { "type": "number" }
  }
}'
```

```
$ kafka-json-schema-console-producer \
  --bootstrap-server kafka:9092 \
  --property schema.registry.url=schema-registry:8081 \
  --property value.schema=$my_schema \
  --topic driver-positions-json
```

JSON Schema Command-line Consumer Tool

```
$ kafka-json-schema-console-consumer \
  --bootstrap-server kafka:9092 \
  --property schema.registry.url=http://schema-registry:8081 \
  --from-beginning \
  --topic driver-positions-json
```

Viewing Schemas in Confluent Control Center

The screenshot shows the Confluent Control Center interface. On the left, there's a sidebar with a navigation menu:

- Overview
- Brokers
- Topics** (selected)
- Connect
- ksqlDB
- Consumers
- Replicators
- Cluster settings

The main content area shows the details for the topic **driver-positions-avro**. At the top, there are tabs for Overview, Messages, Schema (which is selected), and Configuration. Below the tabs, there are two buttons: Value and Key. The Value button is highlighted.

Below the buttons, there are three actions: Edit schema, Version history, and Download. To the right of these actions, it says Format: AVRO Version: 1.

The schema itself is displayed as a multi-line code block:

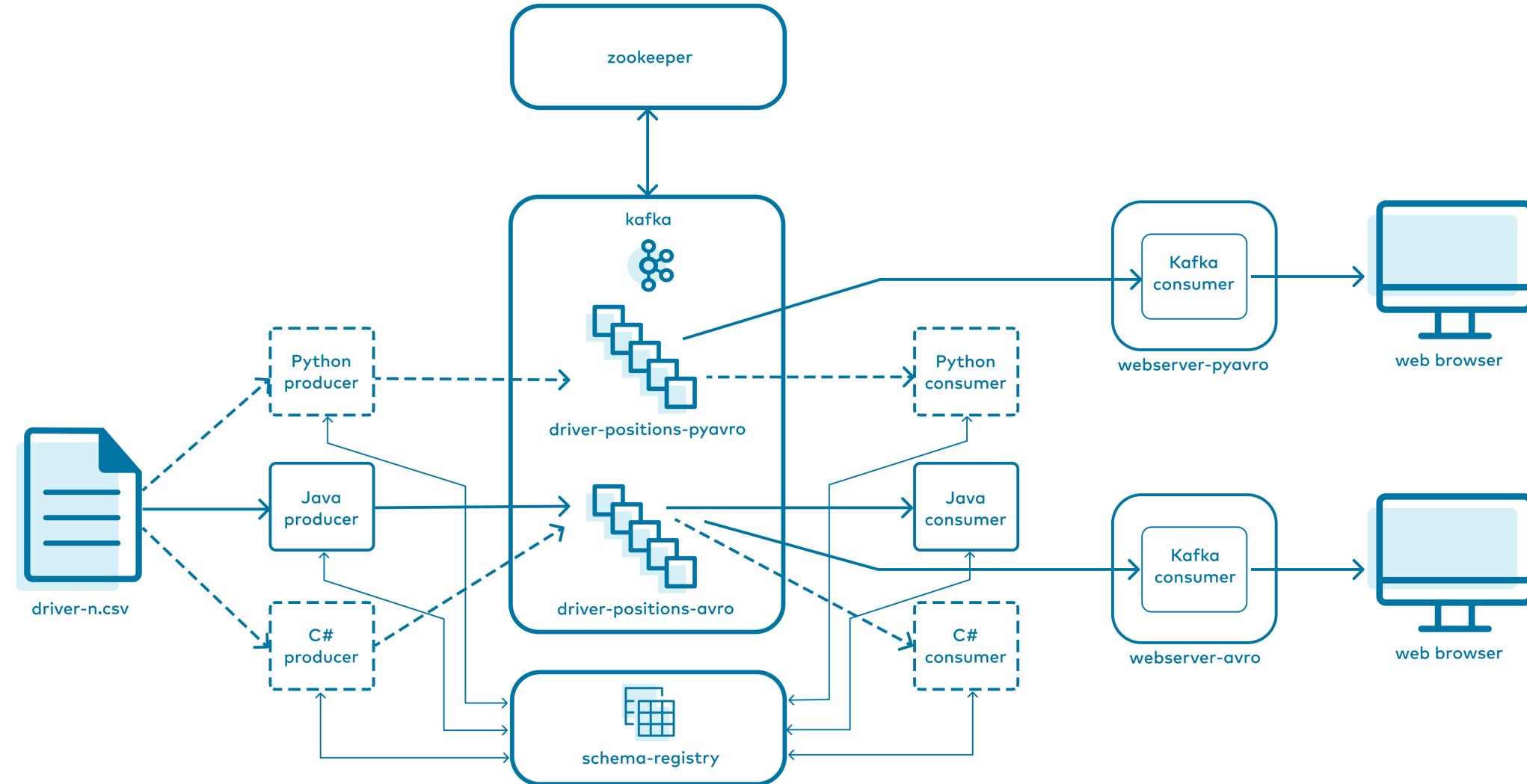
```
1  {
2   "fields": [
3     {
4       "name": "latitude",
5       "type": "double"
6     },
7     {
8       "name": "longitude",
9       "type": "double"
10    }
11  ],
12  "name": "PositionValue",
13  "namespace": "clients.avro",
14  "type": "record"
15 }
```

Hands-On Lab

- In this Hands-On Exercise, you will write and read Kafka data with Avro and interact with the Schema Registry REST API
- Please refer to **Lab 05 Schema Management in Kafka** in the Exercise Book:
 - a. **Schema Registry, Avro Producer and Consumer**



Schema Registry, Avro Producer and Consumer (Java, C#, Python)



Module Review

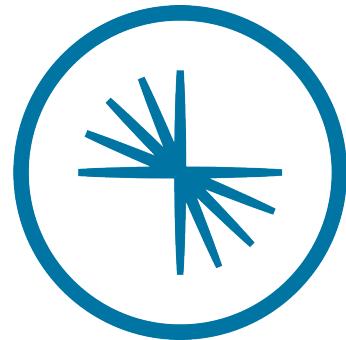


Questions:



1. Here is a Kafka project that uses Avro. Why might the developer have chosen **not** to include `PositionValue.java` in source control?
2. The Confluent documentation suggests disabling automatic schema registration in production. Why might this be the recommended approach?

06 Stream Processing with Kafka Streams



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams ... ←
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

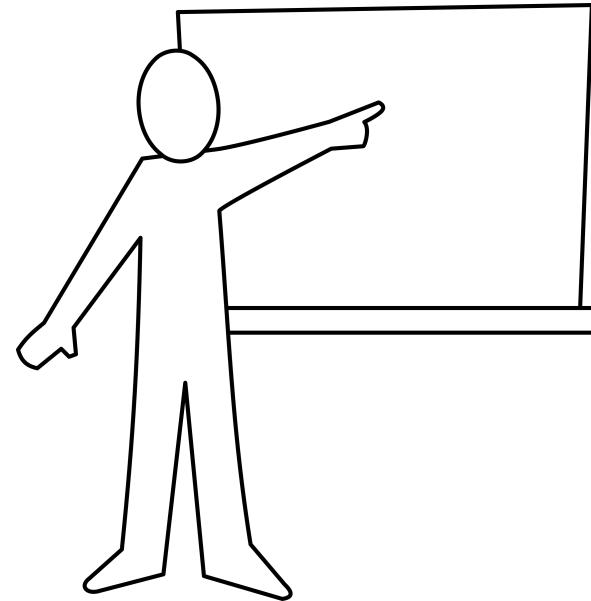
Learning Objectives



After this module you will be able to:

- Compare KStreams to KTables
- Create a Custom `streams.properties` file
- Explain what co-partitioning is and why it is important
- Write an application using the Streams DSL (Domain-Specific Language)

Module Map



- An Introduction to the Kafka Streams API ... ←
- Kafka Streams Concepts
- Creating a Kafka Streams Application
- Kafka Streams by Example
-  Hands-on Lab

Kafka Streams

Kafka Streams is a Java **client library** for building applications and microservices, where the **input and output data are stored in Kafka clusters**

- Just a Java library—No separate resource management technology required
- Deploy to containers, VMs, bare metal, cloud
- Powered by Kafka: elastic, scalable, distributed, battle-tested
- Perfect for small, medium, large use cases
- Fully integrated with Kafka security
- Exactly-once processing semantics
- Part of the Apache Kafka project

What Is the Kafka Streams API?

- **Transforms** and **enriches** data
 - per-record stream processing
 - millisecond latency
 - stateless & stateful processing
 - windowing operations
- **Fault-tolerant** and **distributed processing**
- Has **Domain-Specific Language** (DSL)
 - High level operations: `map`, `filter`, `count`, etc.
- **Processor API** for even more flexibility

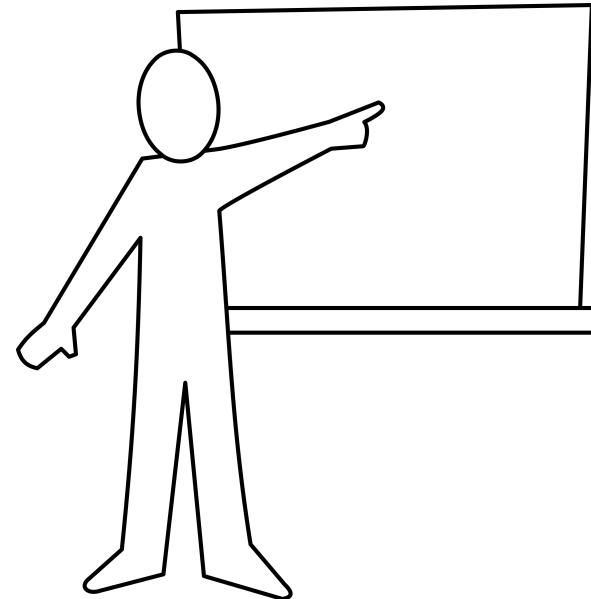
A Library, Not a Framework

- Kafka Streams provides **stream processing capabilities**
- Other streaming frameworks:
 - Apache Flink
 - Spark Streaming
 - Apache Storm
 - Apache Samza
 - etc.
- **BUT:** Kafka Streams does not require separate resource management technology
 - It's just a **Java library**
 - Runs on **1...n** machines
 - Failover, scalability, recovery, all handled by the Kafka cluster

Why Not Just Build Your Own?

- You can build your own stream processing apps
 - Using the Producer and Consumer APIs
- But Kafka Streams API is much easier
 - Well-designed, well-tested, robust
 - Powerful DSL
 - Designed for scalability
 - Designed for failure recovery, even for stateful processing
 - So you can focus on the application logic, not the low-level plumbing

Module Map



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts ... ←
- Creating a Kafka Streams Application
- Kafka Streams by Example
-  Hands-on Lab

What is a Stream?

- Think of a stream as an unbounded, real-time flow of records
 - You don't need to explicitly request new records, you just receive them
- Records are key-value pairs

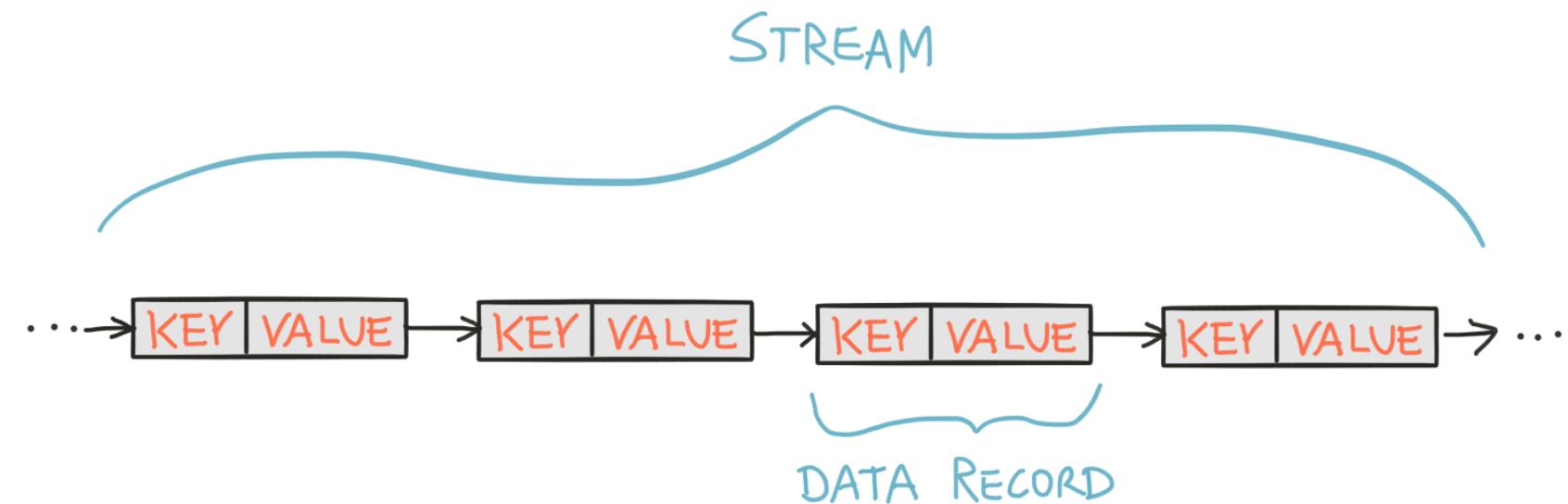
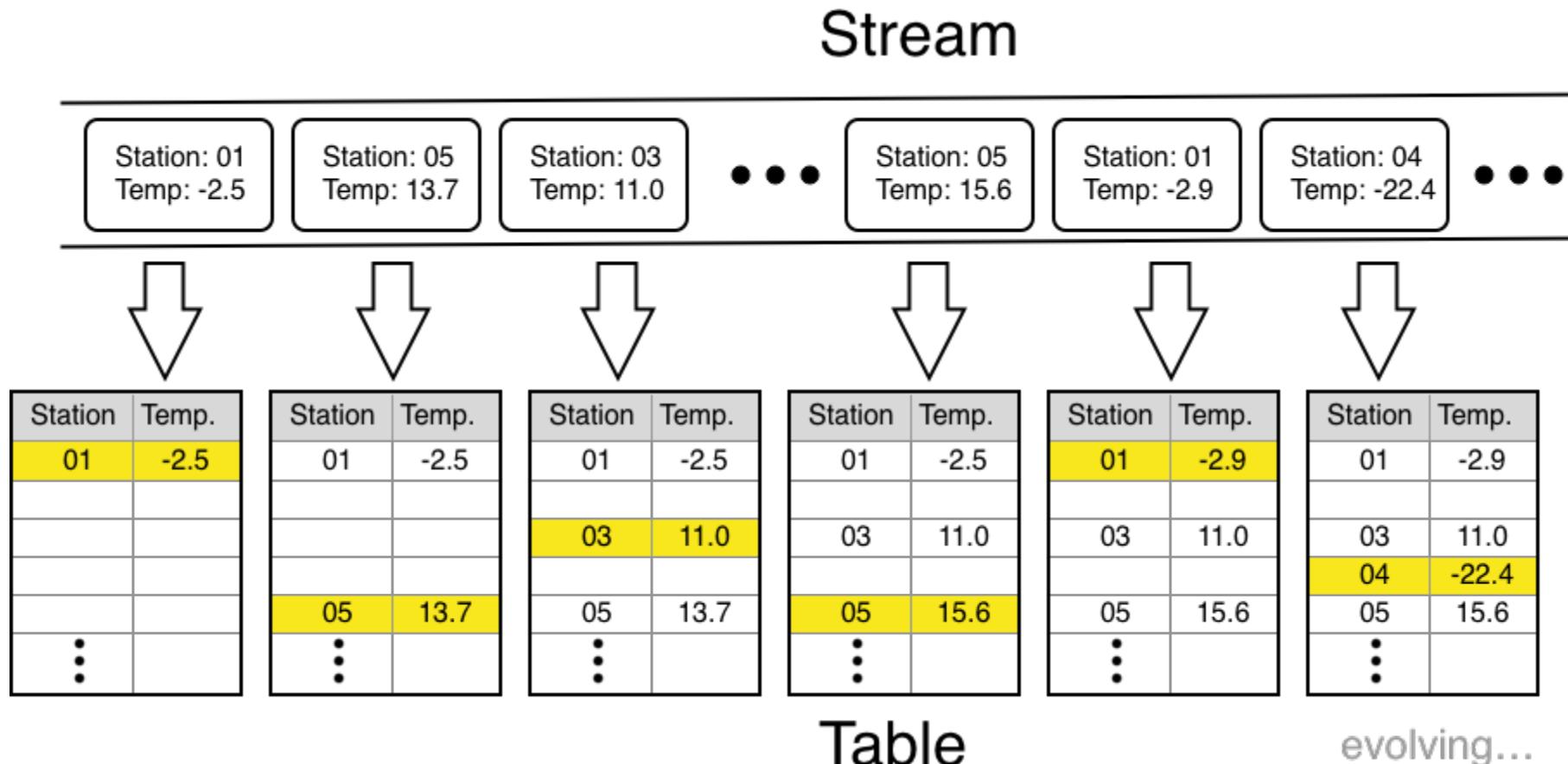
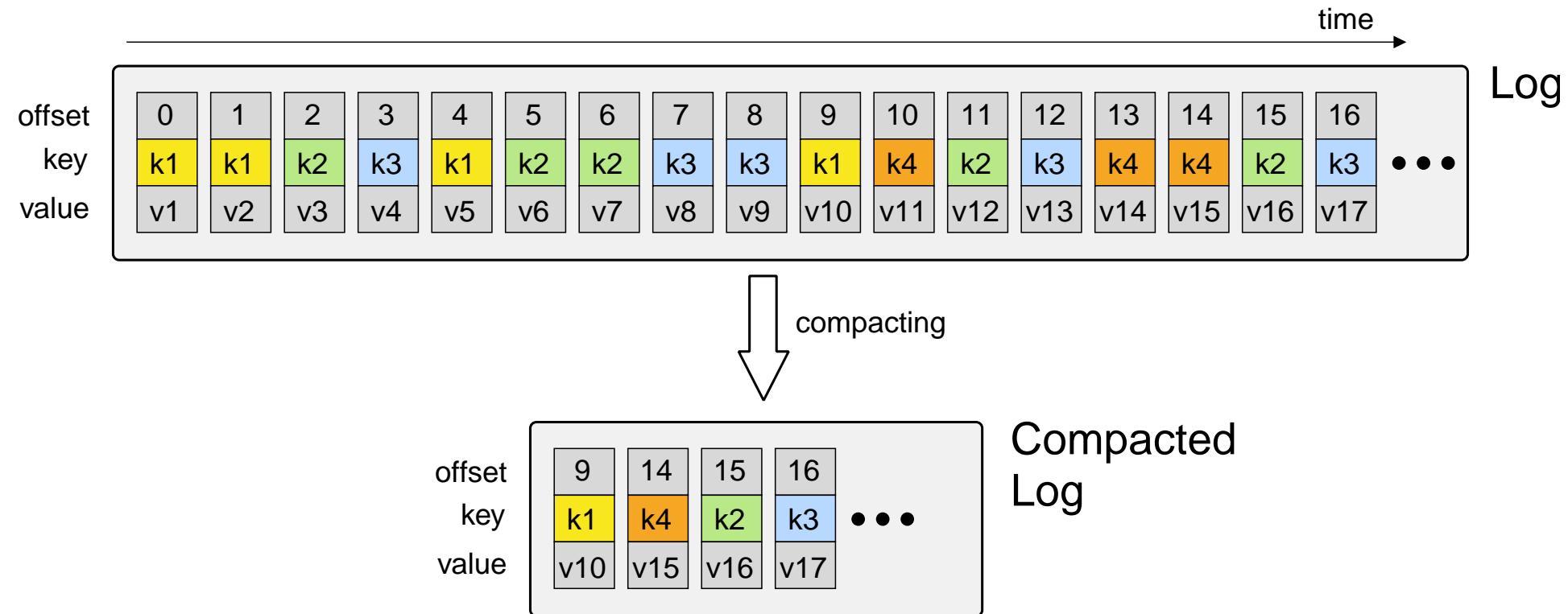


Table as an Evolving Set of Facts



In the Kafka Streams API, the "KStream" object represents a stream, and the "KTable" object represents a table.

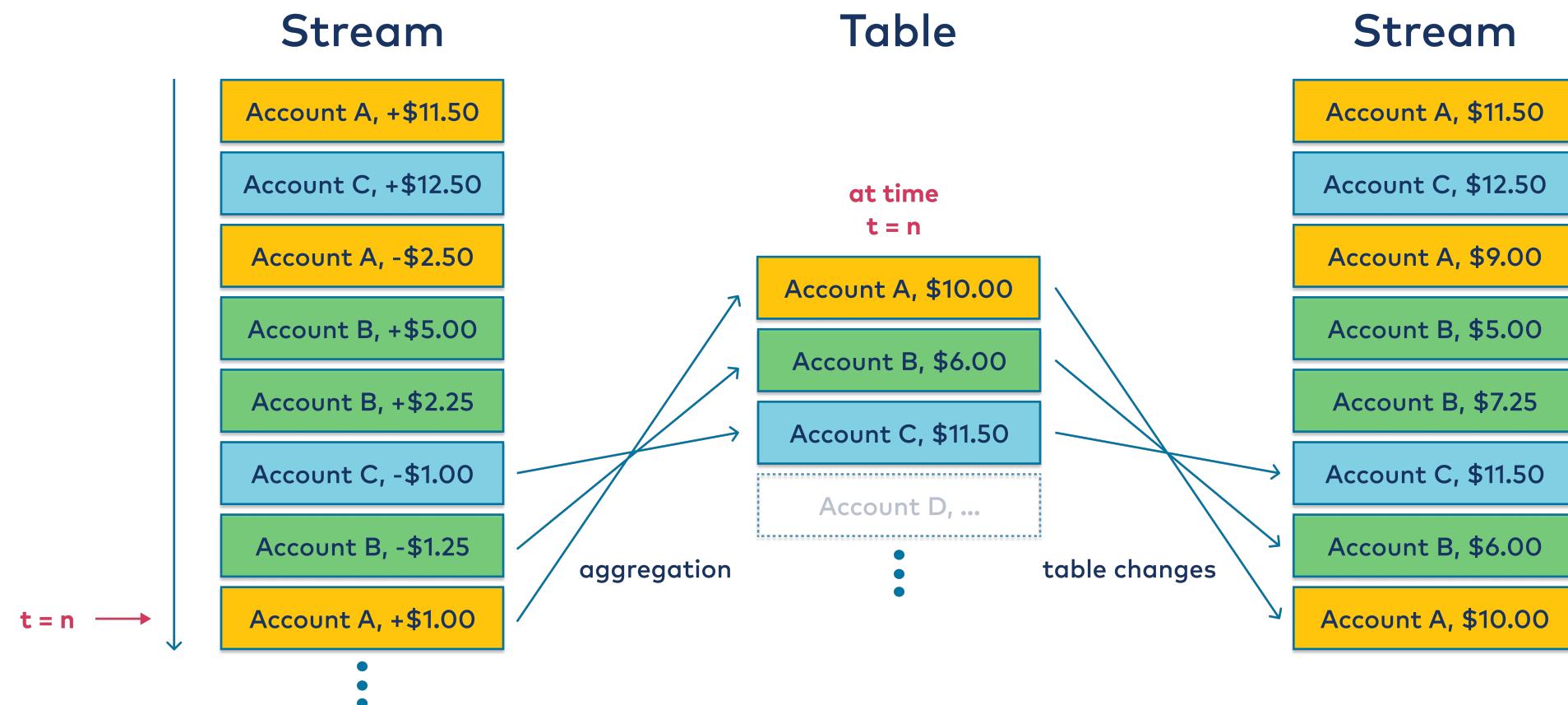
Tables and Compacted Topics



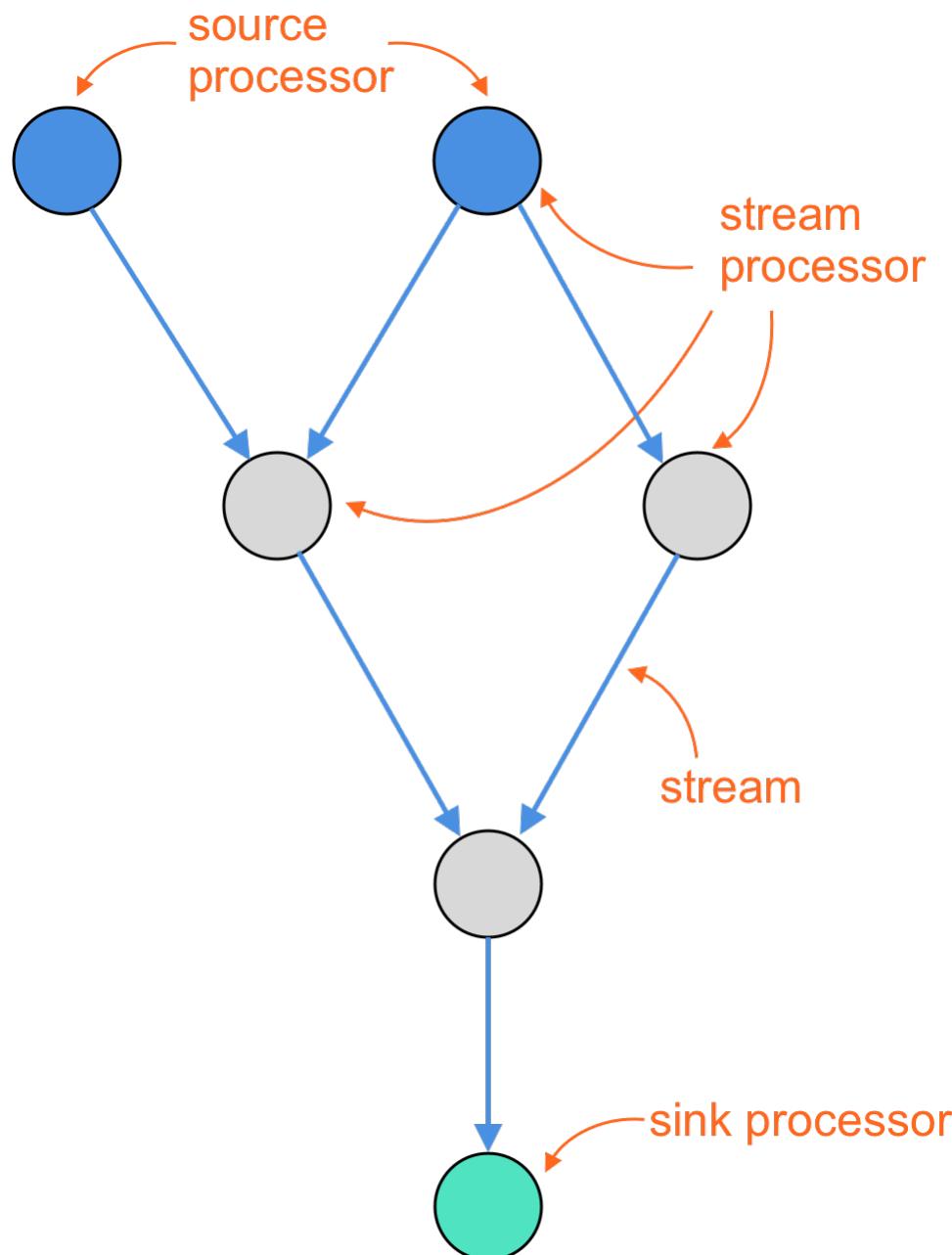
Question: How might compacted topics and tables be related?

Stream - Table Duality with Aggregation

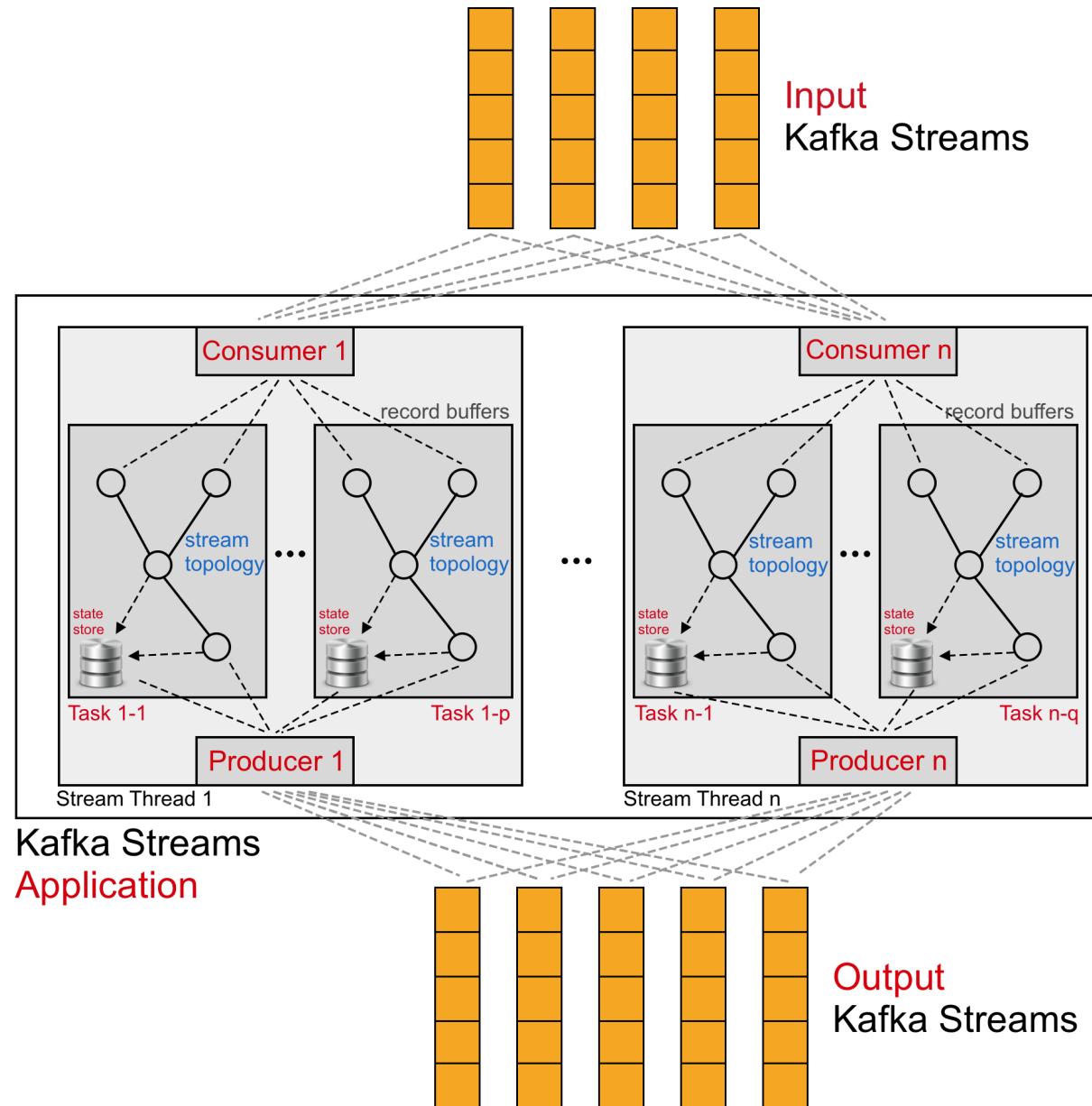
- We can turn a stream into a table by aggregating the stream
- We can turn a table into a stream by capturing the changes made to the table



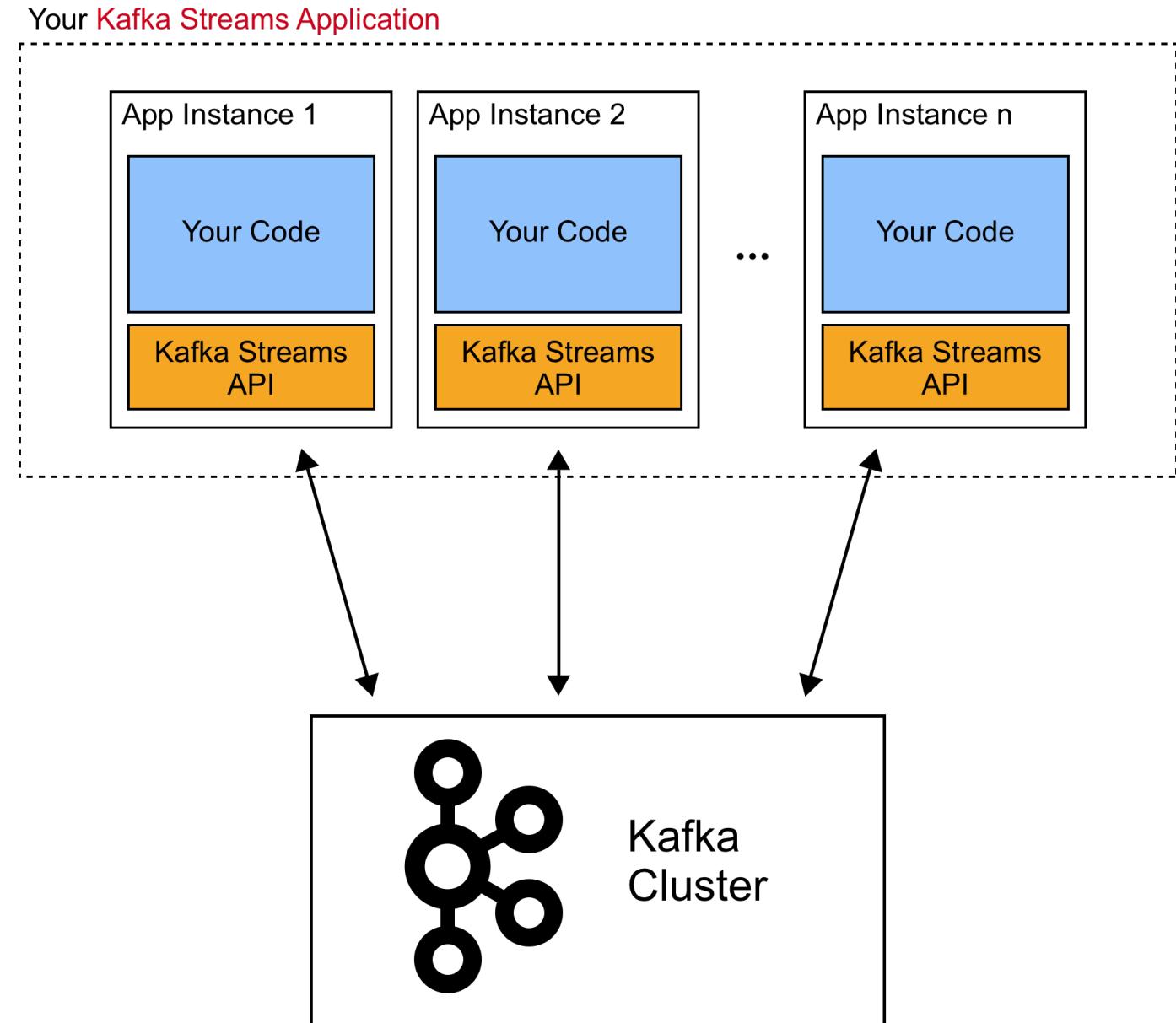
Stream Processor Topology



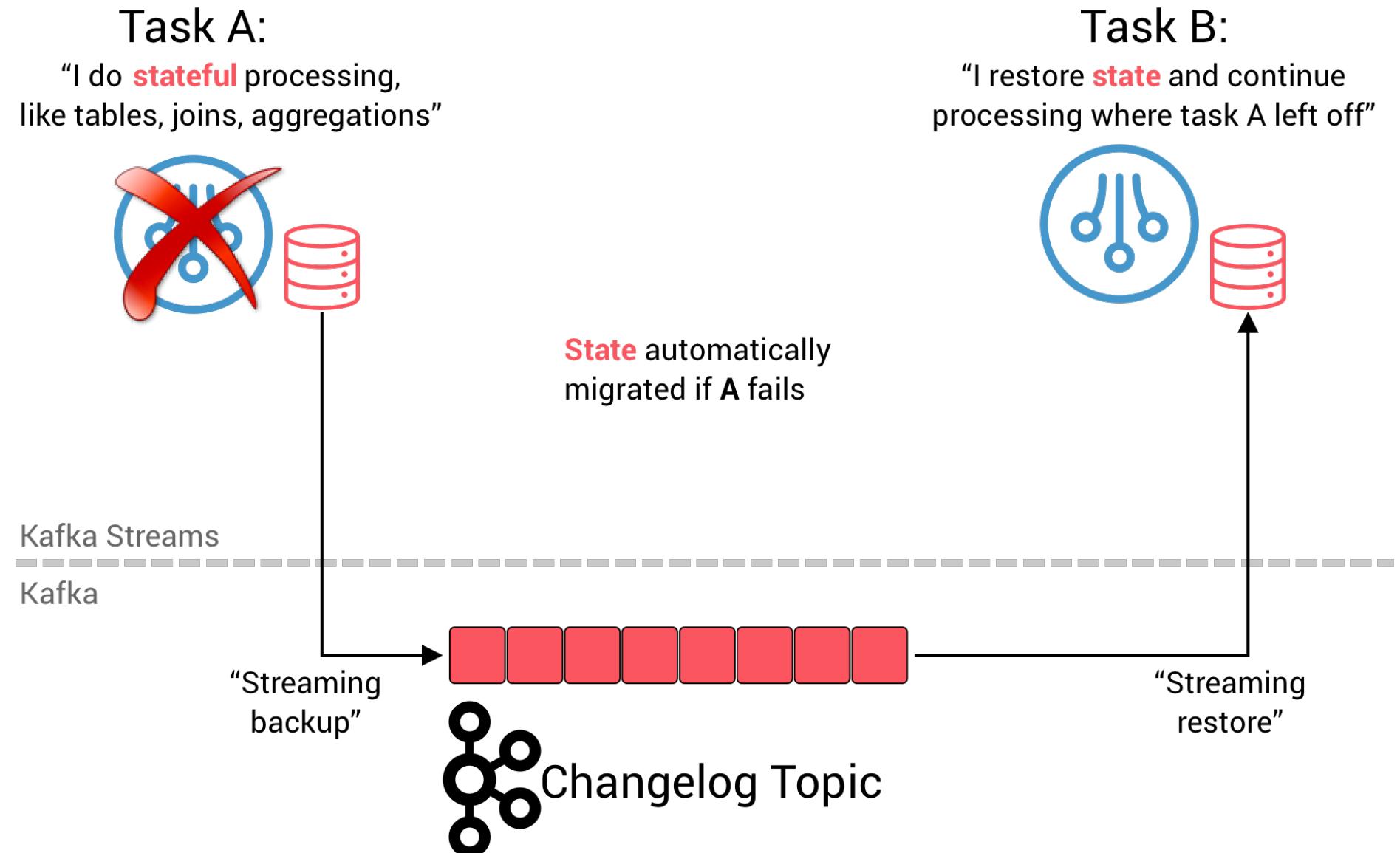
Logical View of Kafka Streams Application



Kafka Streams Application Architecture



Task State Recovery Powered by Kafka



Standby Replicas for Faster Task Recovery

- **Standby replicas** are "shadow copies" of local state stores
 - kept up-to-date from changelog Kafka topics
- Set `num.standby.replicas` to 1 or greater (Default: 0)
- A recovered task is scheduled to an instance with a standby replica of its state
- Nearly immediate task recovery

Sizing and Scaling

- Stateful client applications may use Streams operations (e.g., aggregates, joins, windows)
- Clients keep state in local state stores (*i.e.*, RocksDB)
 - The state stores use local disk on the client machines
 - The state stores have 50MB - 100MB memory overhead
 - Clients persist state stores to a compacted Kafka Topic
 - If the client state store fails, the data is recoverable
 - Standby replicas for Kafka Streams tasks results in extra copies of state stores
- If client performance is bound by CPU
 - Add cores or machines, and increase the number of threads
 - `num.stream.threads` (Default: 1)
- If client performance is bound by network, memory, or disk
 - Scale out the clients to additional machines

Windows, Aggregations, and Joins

Windows:

- Divide stream into "time buckets"
- Tumbling, Hopping, and Session windows

Aggregations

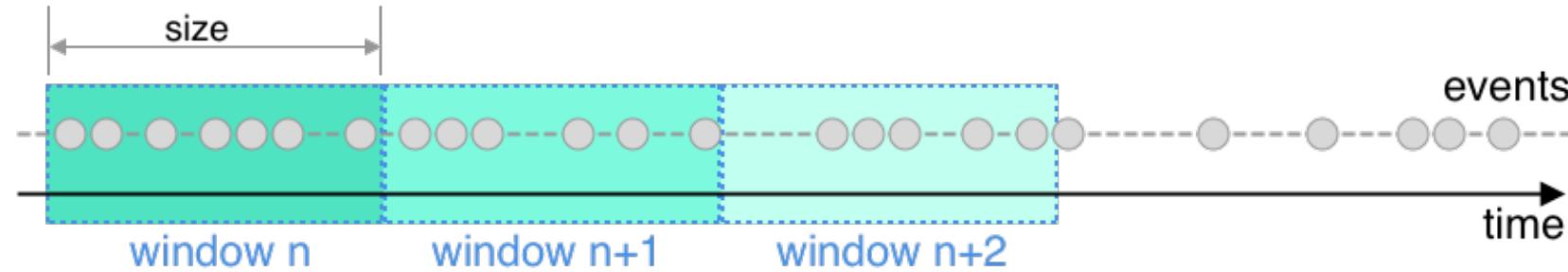
- Accumulate some value as new records come in
- Usually windowed
- Examples: sum, count, max, min

Joins:

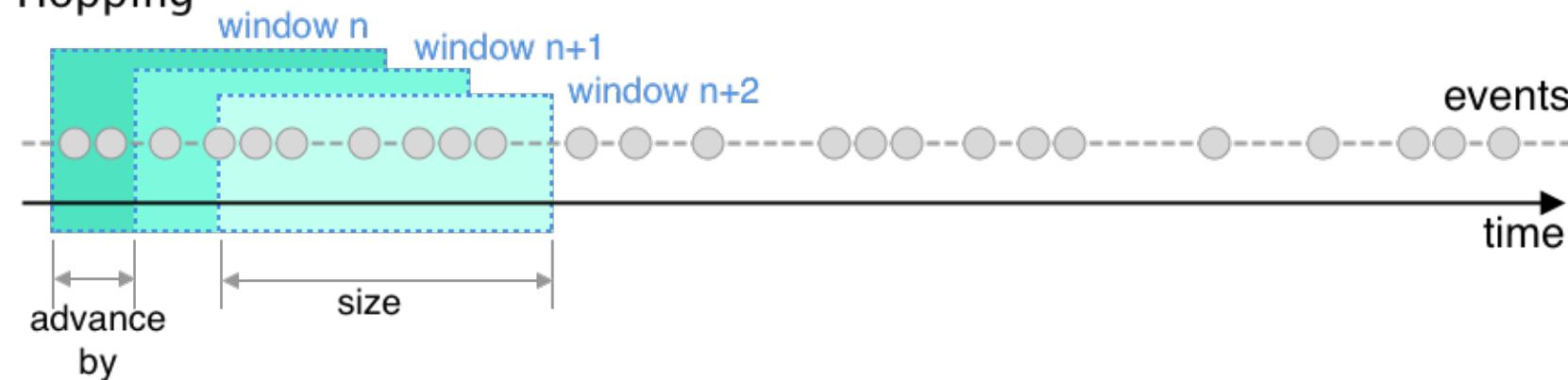
- Combine different streams/tables together on a key
- Example: Join driver location stream with driver profile table on driver ID key to create enriched stream with both location and profile info
- Can be windowed with a "sliding window"

Window Types

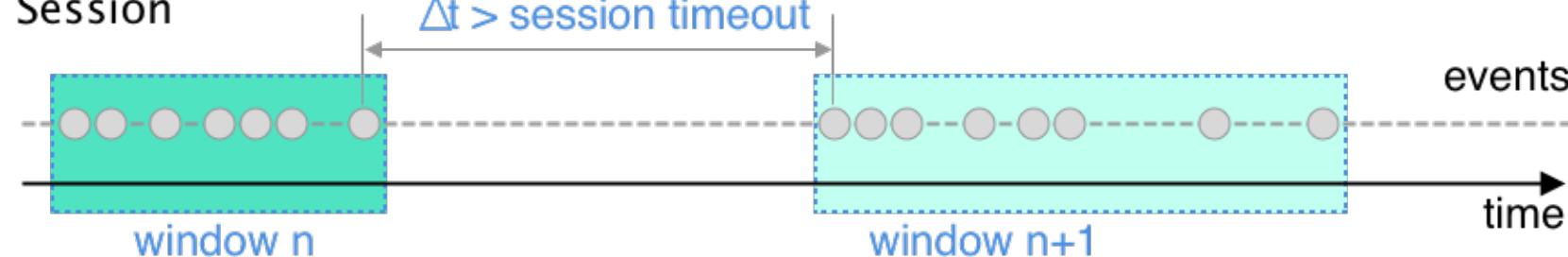
Tumbling



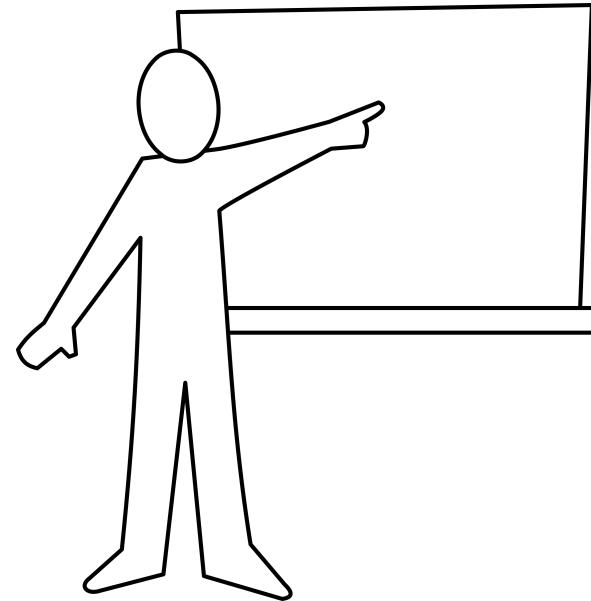
Hopping



Session



Module Map



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts
- Creating a Kafka Streams Application ... ←
- Kafka Streams by Example
-  Hands-on Lab

Configuring a Kafka Streams Application (1)

Option 1: Use `StreamsConfig` helper class with `consumerPrefix()`, `producerPrefix()`,
`adminClientPrefix()` methods for client-specific properties:

StreamsApp.java

```
....  
Properties config = new Properties();  
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-app-v1.0.0");  
config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092,broker-2:9092");  
config.put(StreamsConfig.consumerPrefix(FETCH_MIN_BYTES_CONFIG), "1024");  
config.put(StreamsConfig.producerPrefix(BATCH_SIZE_CONFIG), "32768");  
...
```

Configuring a Kafka Streams Application (2)

Option 2: Load `.properties` file like usual, but using `consumer`, `producer`, and `admin` prefixes to configure the app's consumer, producer, and admin Kafka clients:

streams.properties

```
...
application.id=my-app-v1.0.0
bootstrap.servers=broker-1:9092,broker-2:9092,broker-3:9092
consumer.fetch.min.bytes=1024
producer.batch.size=32768
...
```

Serializers and Deserializers (SerDes)

- Use Serializers and Deserializers (SerDes) to convert bytes of the record to a specific type
 - SERializer
 - DESerializer
- Key SerDes can be independent from value SerDes
- There are many built-in SerDes (e.g. `Serdes.String()`, etc.)
- You can also define your own
- Configuration example:

```
config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
          Serdes.String().getClass().getName());  
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
          Serdes.Long().getClass().getName());
```

Available SerDes

- Kafka includes a variety of SerDes in the `kafka-clients` Maven artifact

Data type	SerDes
<code>byte[]</code>	<code>Serdes.ByteArray()</code> , <code>Serdes.Bytes()</code> (<code>Bytes</code> wraps Java's <code>byte[]</code> and supports equality and ordering semantics)
<code>ByteBuffer</code>	<code>Serdes.ByteBuffer()</code>
<code>Double</code>	<code>Serdes.Double()</code>
<code>Integer</code>	<code>Serdes.Integer()</code>
<code>Long</code>	<code>Serdes.Long()</code>
<code>String</code>	<code>Serdes.String()</code>

- In addition, `kafka-streams-avro-serde`, `kafka-streams-json-schema-serde` and `kafka-streams-protobuf-serde` Maven artifacts are also available

Creating the Processing Topology

- Create a `KStream` or `KTable` object from one or more Kafka Topics, using `StreamsBuilder`
- Example:

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, Long> purchases = builder.stream("PurchaseTopic");

KTable<Integer, String> users = builder.table("UsersTopic");
```

Transforming a Stream

- Data can be transformed using a number of different operators
- Some operations result in a new `KStream` object
 - For example, `filter` or `map`
- Some operations result in a `KTable` object
 - For example, an aggregation operation

Stateless Transformation Operations (1)

filter

Creates a new `KStream` containing only records from the previous `KStream` which meet some specified criteria

```
KStream<K,V> smallPurchases = purchases  
    .filter((key,value) -> value.amount < 50.0)
```

Stateless Transformation Operations (2)

mapValues

Creates a new `KStream` by transforming the value of each element in the current stream into a different element in the new stream. Use when only transforming the value.

```
KStream<K,V> upper = words
    .mapValues(value -> value.toUpperCase());
```

map

Creates a new `KStream` by transforming each element in the current stream into a different element in the new stream. Use this if you must change the key.

```
KStream<K,V> upper_new_key = words
    .map((key,value)->new KeyValue<>(value.toLowerCase(),
                                              value.toUpperCase()));
```

Stateless Transformation Operations (3)

flatMapValues

Creates a new `KStream` by transforming the value of each element in the current stream into zero or more different elements in the new stream. Only modify the value.

```
Pattern pattern = Pattern.compile("\\w+", ...);
```

```
KStream<byte[], String> words = textLines
    .flatMapValues(sentence -> Arrays.asList(pattern.split(sentence)));
```

flatMap

Creates a new `KStream` by transforming each element in the current stream into zero or more different elements in the new stream. You can modify the record keys and values, including their types.

Stateful Transformation Operations

count

Counts the number of instances of each key in the stream; results in a new, ever-updating `KTable`

```
stream.groupByKey()  
    .count()
```

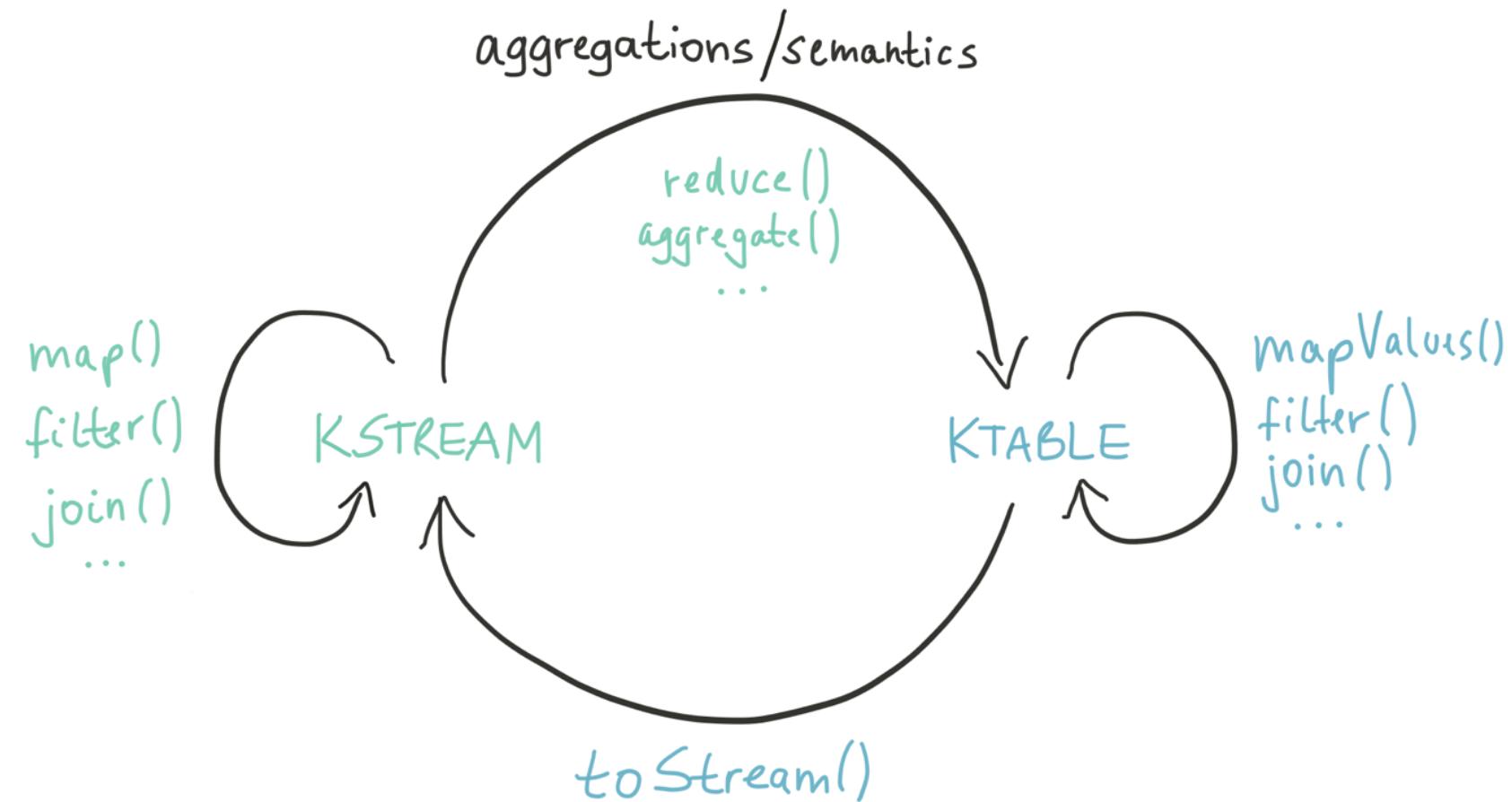
reduce

Combines values of the stream using a supplied Reducer into a new, ever-updating `KTable`

```
stream.groupByKey()  
    .reduce(...)
```

Mixed Processing

- You can build stream processing topologies with mixed stateless and stateful processing



Writing Streams Back to Kafka

- Streams usually terminate by producing back to a Kafka topic
- This is accomplished with the `KStream#to()` method:

```
myNewStream.to("NewTopic");
```

Printing a Stream's Contents

- It is sometimes useful to be able to see what a stream contains
 - Especially when testing and debugging
- Use `print()` to write the contents of the stream

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

Running the Application

- To start processing the stream, create a `KafkaStreams` object
 - Configure it using `StreamsBuilder`
- Then call the `start()` method
- Example:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

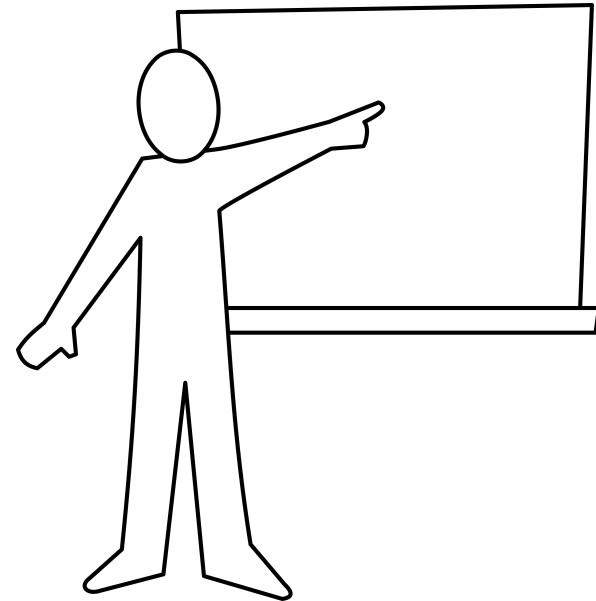
Application Graceful Shutdown

- Allow your application to gracefully shutdown
 - For example, in response to SIGTERM
- Add a shutdown hook to stop the application

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

- After an application is stopped, Kafka migrates any tasks that had been running in this instance to available remaining instances

Module Map



- An Introduction to the Kafka Streams API
- Kafka Streams Concepts
- Creating a Kafka Streams Application
- Kafka Streams by Example ... ←
- Hands-on Lab

A Simple Kafka Streams Example (Configuration)

```
1 public class SimpleStreamsExample {  
2  
3     public static void main(String[] args) throws Exception {  
4         Properties config = new Properties();  
5         // Give the Streams application a unique name.  
6         // The name must be unique in the Kafka cluster.  
7         config.put(StreamsConfig.APPLICATION_ID_CONFIG,  
8             "simple-streams-example");  
9         config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,  
10            "broker-1:9092,broker-2:9092");  
11         config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
12         // Specify default (de)serializers for record keys and for record values.  
13         config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
14             Serdes.ByteArray().getClass());  
15         config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
16             Serdes.String().getClass());
```

A Simple Kafka Streams Example (Topology)

```
17  StreamsBuilder builder = new StreamsBuilder();
18
19  // Construct a KStream from the input Topic "TextLinesTopic"
20  KStream<byte[], String> textLines = builder.stream("TextLinesTopic");
21  // Convert to upper case
22  KStream<byte[], String> uppercasedWithMapValues =
23      textLines.mapValues(value -> value.toUpperCase());
24  // Write the results to a new Kafka Topic called "UppercasedTextLinesTopic".
25  uppercasedWithMapValues.to("UppercasedTextLinesTopic");
26  // Run the Streams application via `start()`
27  KafkaStreams streams = new KafkaStreams(builder.build(), config);
28  streams.start();
29  // Stop the application gracefully
30  Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
31 }
32 }
```

Kafka Streams With Stateful Processing (Configuration)

```
1 public class SimpleStreamsExample {  
2  
3     public static void main(String[] args) throws Exception {  
4         Properties config = new Properties();  
5         // Give the Streams application a unique name.  
6         // The name must be unique in the Kafka cluster  
7         config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-stateful-example");  
8         config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");  
9         config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
10        // Specify default (de)serializers for record keys and for record values.  
11        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
12                    Serdes.ByteArray().getClass());  
13        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
14                    Serdes.String().getClass());
```

Kafka Streams With Stateful Processing (Topology)

```
15 // Create a topology builder
16 StreamsBuilder builder = new StreamsBuilder();
17
18 // Stream sentences from the input topic "TextLinesTopic"
19 KStream<byte[], String> textLines = builder.stream("TextLinesTopic");
20
21 // Count the occurrences of each word from the streaming sentences
22 KTable<String, Long> wordCounts = textLines
23     .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
24     .groupBy((key, word) -> word, Grouped.withKeySerde(Serdes.String()))
25     .count(Materialized.as("Counts"));
26
27 // Write the word counts to an output topic "WordsWithCountsTopic"
28 wordCounts
29     .toStream()
30     .to("WordsWithCountsTopic", Produced.with(Serdes.String(), Serdes.Long()));
```

Kafka Streams With Stateful Processing (Build and Run)

```
31    // Build the topology and run the Streams application via `start()`
32    Topology topology = builder.build();
33    KafkaStreams streams = new KafkaStreams(topology, config);
34    streams.start();
35
36    // Prevent resource leaks with a shutdown hook
37    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
38 }
39 }
```

Join Example

- You can enrich records by **joining data** from different sources
- The example below does a **KStream-KStream inner join**:
 - When record occurs in either stream, look up all records in other stream with matching key within time window and create output event with same key and a new value `doStuff(leftValue, rightValue)`

```
1 KStream<Long, String> myJoinedStream =  
2     myLeftStream.join(myRightStream, (leftValue, rightValue) ->  
3         doStuff(leftValue, rightValue),  
4         JoinWindows.of(Duration.ofMinutes(5)),  
5         Joined.with(Serdes.Long(), Serdes.String(), Serdes.String()));
```

- Many other kinds of joins possible (e.g. KStream-KTable inner and left joins)

Questions:

- Why must a stream-stream join be windowed?
- Why must input topics be co-partitioned for a join?

Kafka Streams Processing Guarantees

- Supports **at-least-once** processing
 - With no failures, it will process data exactly once
 - Upon failure, some records may be processed **more than once**
- Supports **Exactly Once** processing semantics (EOS)
 - Set `processing.guarantee=exactly_once` (Default: `at_least_once`)



We will discuss EOS in deeper detail in an upcoming module

More Kafka Streams API Features

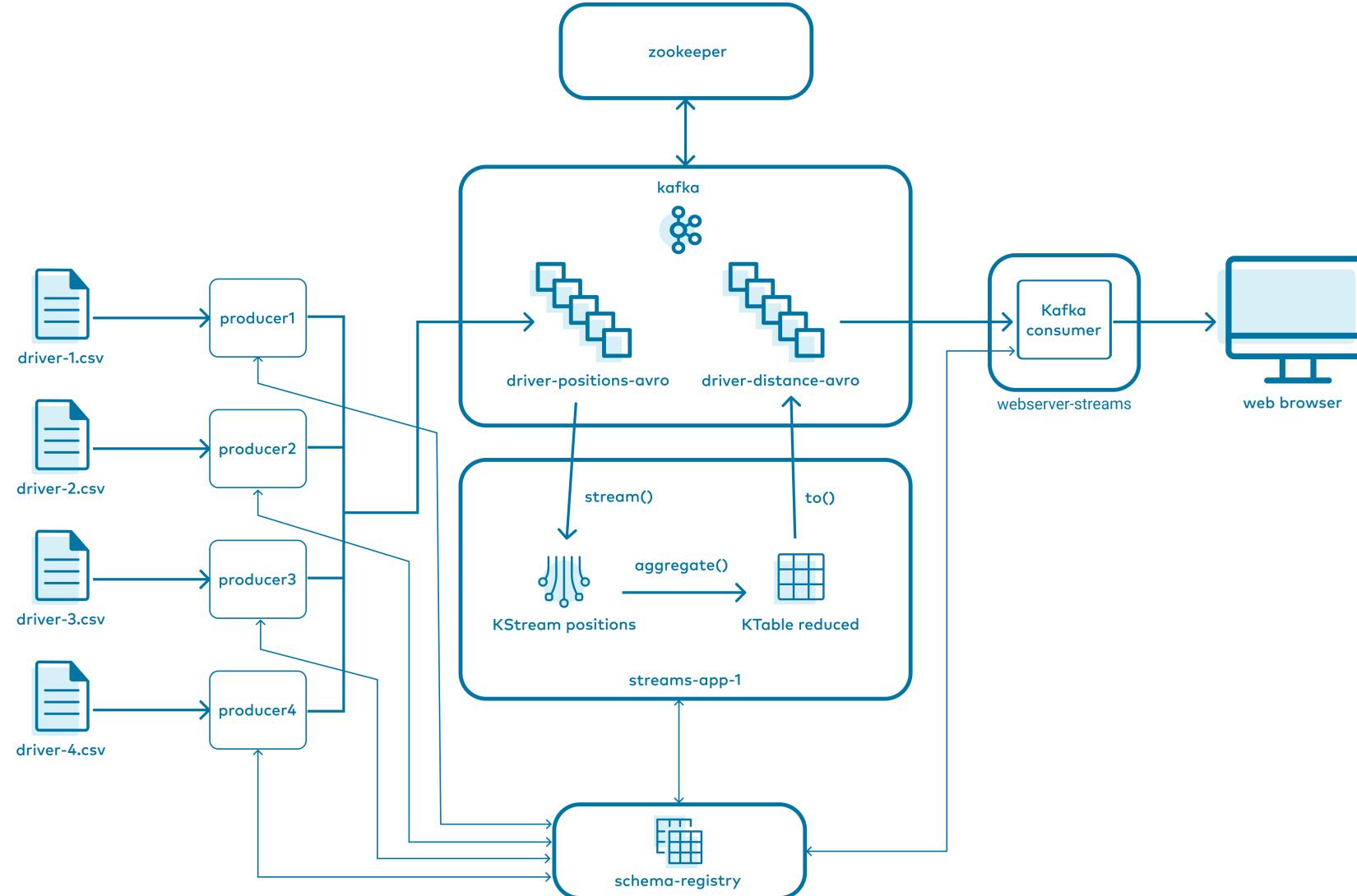
- We have only scratched the surface of Kafka Streams
- Check the **documentation** and sign up for more **training** to learn about processing windowed tables, joining tables, joining streams and tables, and much more!

Hands-On Lab

- In this Hands-On Exercise, you will investigate a Kafka Streams application that aggregates the distance each driver has driven.
- Please refer to **Lab 06 Writing a Kafka Streams Application** in the Exercise Book:
 - a. **Kafka Streams**



Kafka Streams (Java)



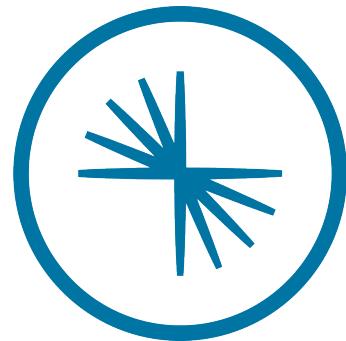
Module Review



Questions:

What are some advantages of Kafka Streams versus other stream processing software?

07 Data Pipelines with Kafka Connect



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect ... ←
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

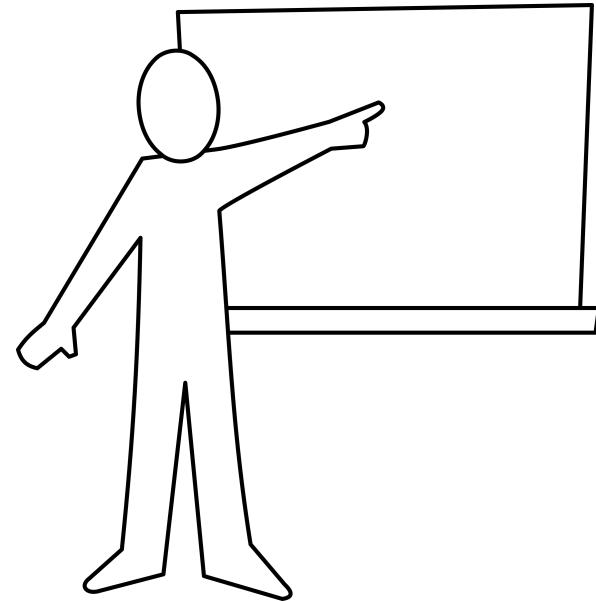
Learning Objectives



After this module you will be able to:

- explain the motivation for Kafka Connect
- list commonly used Connectors
- explain the differences between standalone and distributed mode
- configure and use Kafka Connect
- use Single Message Transforms (SMTs)

Module Map



- The Motivation for Kafka Connect ... ←
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab

GOALS



1. Focus on copying
2. Batteries included
3. Standardize
4. Parallelism
5. Scale

What is Kafka Connect?

- Kafka Connect is a framework for streaming data between Apache Kafka and other data systems
- Kafka Connect is open source, and is part of the Apache Kafka distribution
- It is simple, scalable, and reliable



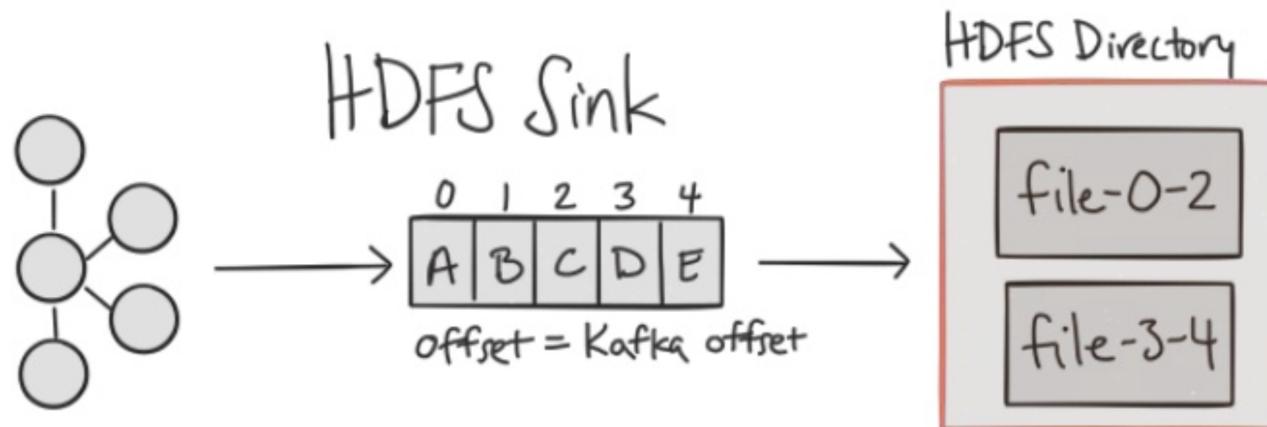
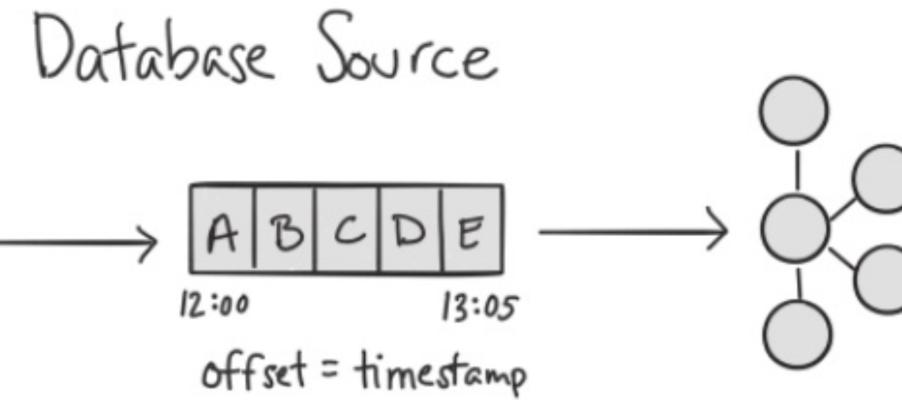
Connect Basics

- **Connectors** are logical jobs that copy data between Kafka and another system
- **Source** Connectors read data *from* an external data system into Kafka
 - Internally, a source connector is a Kafka Producer
- **Sink** Connectors write Kafka data *to* an external data system
 - Internally, a sink connector is a Kafka Consumer Group

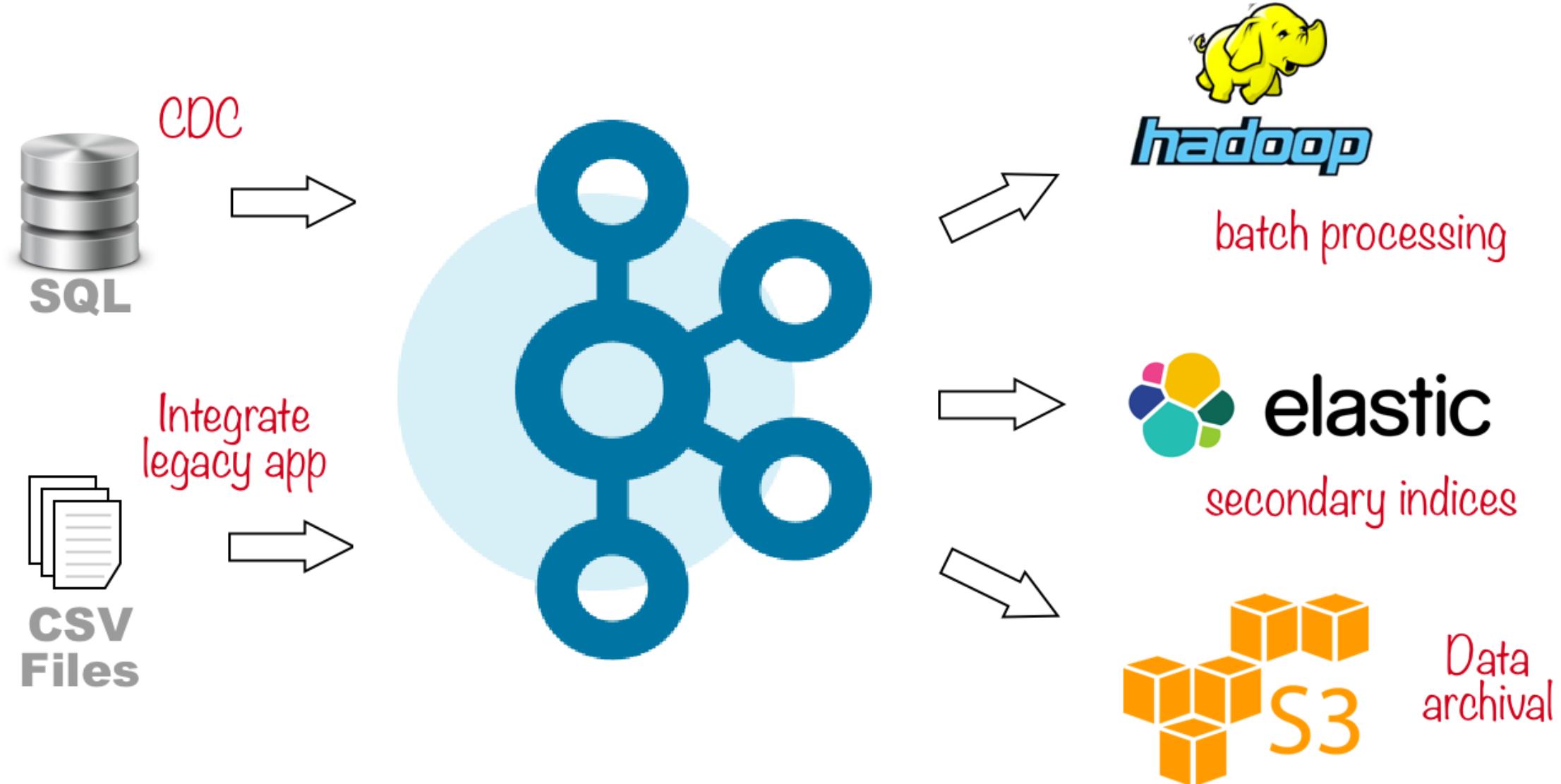
Connect Illustrated

Table

TS	Data
12:00	A
12:20	B
12:30	C
13:00	D
13:05	E



Example Use Cases



Why Not Just Use Producers and Consumers?

Not Optimal

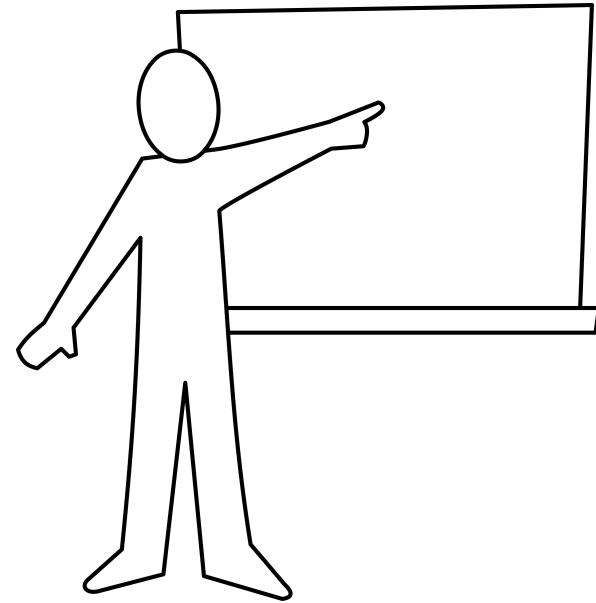
- One-off tools
- All-purpose tools
- Stream processing frameworks



Kafka Connect

- Off-the-shelf
- Tested
- Connectors for common sources & sinks
- Fault tolerance
- Automatic load balancing
- No coding required
- Pluggable/extendable by developers

Module Map



- The Motivation for Kafka Connect
- Types of Connectors ... ←
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
-  Hands-on Lab

Confluent Hub

CONFLUENT

Product Cloud Developers Blog Docs DOWNLOAD

Confluent Hub

Discover Kafka® connectors and more

What plugin are you looking for?

Filters

Plugin type:

- Sink
- Source
- Transform
- Converter

Enterprise support:

- Confluent supported
- Partner supported
- None

Verification:

- Confluent built
- Confluent Tested
- Verified gold
- Verified standard
- None

License:

- Commercial
- Free

Results (158)

+ Submit a plugin

Kafka Connect GCP Pub-Sub

A Kafka Connect plugin for GCP Pub-Sub

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported
Verification: Confluent built
License: Commercial
Author: Confluent, Inc.
Version: 1.0.2

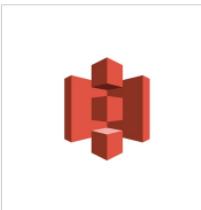


Kafka Connect S3

The S3 connector, currently available as a sink, allows you to export data from Kafka topics to S3 objects in either Avro or JSON formats

Available fully-managed on Confluent Cloud

Enterprise support: Confluent supported
Verification: Confluent built
License: Free
Author: Confluent, Inc.
Version: 5.5.1



Installing New Connectors

From Confluent Hub

- Use `confluent-hub` client included with Confluent Platform

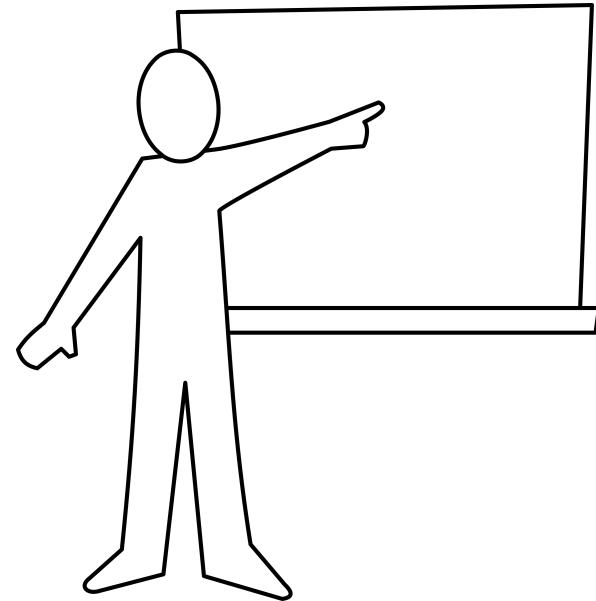
```
$ confluent-hub install debezium/debezium-connector-mysql:latest
```

From other sources

- Package Connector in JAR file
- Install JAR file on **all** Kafka Connect worker machines
 - Connectors are installed as plugins
 - There is **library isolation** between plugins

```
plugin.path=/path/to/my/plugins
```

Module Map



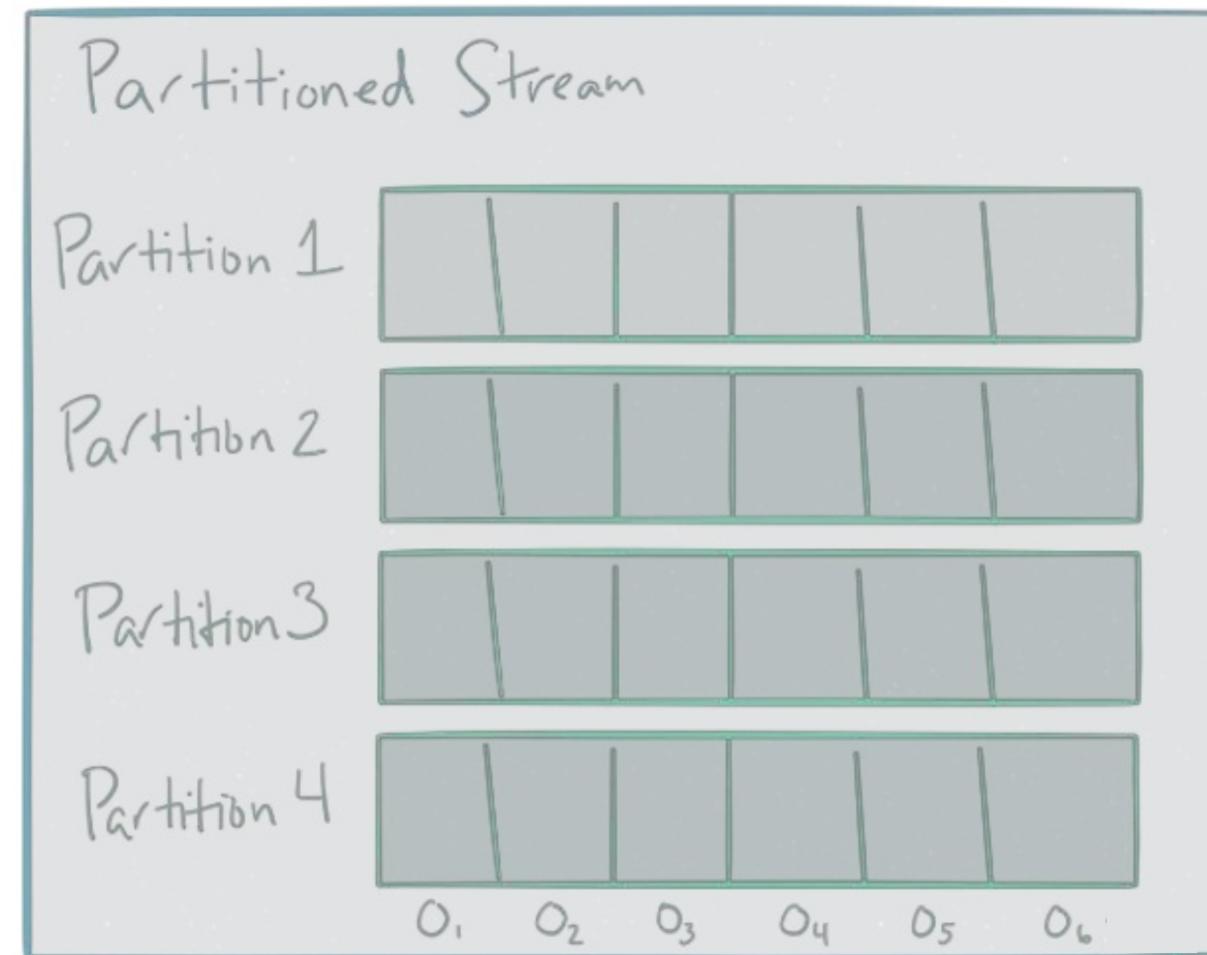
- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation ... ←
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
- 🤖 Hands-on Lab

Kafka Connect Reliability and Scalability

- Elasticity and scalability
 - To add resources to Connect simply start more workers
 - Connect Workers discover each other and load-balance the work automatically
 - Allows running more connectors and more tasks
- Reliability and fault-tolerance
 - If a worker fails, then all its tasks and connectors will automatically restart on another worker
 - If individual tasks or connectors fail, then the user can decide how to respond
- ALL connectors support at-least-once semantics
- SOME connectors support exactly-once semantics
- MANY connectors support schema evolution
 - Data format changes in source system will reflect in Kafka and all target systems

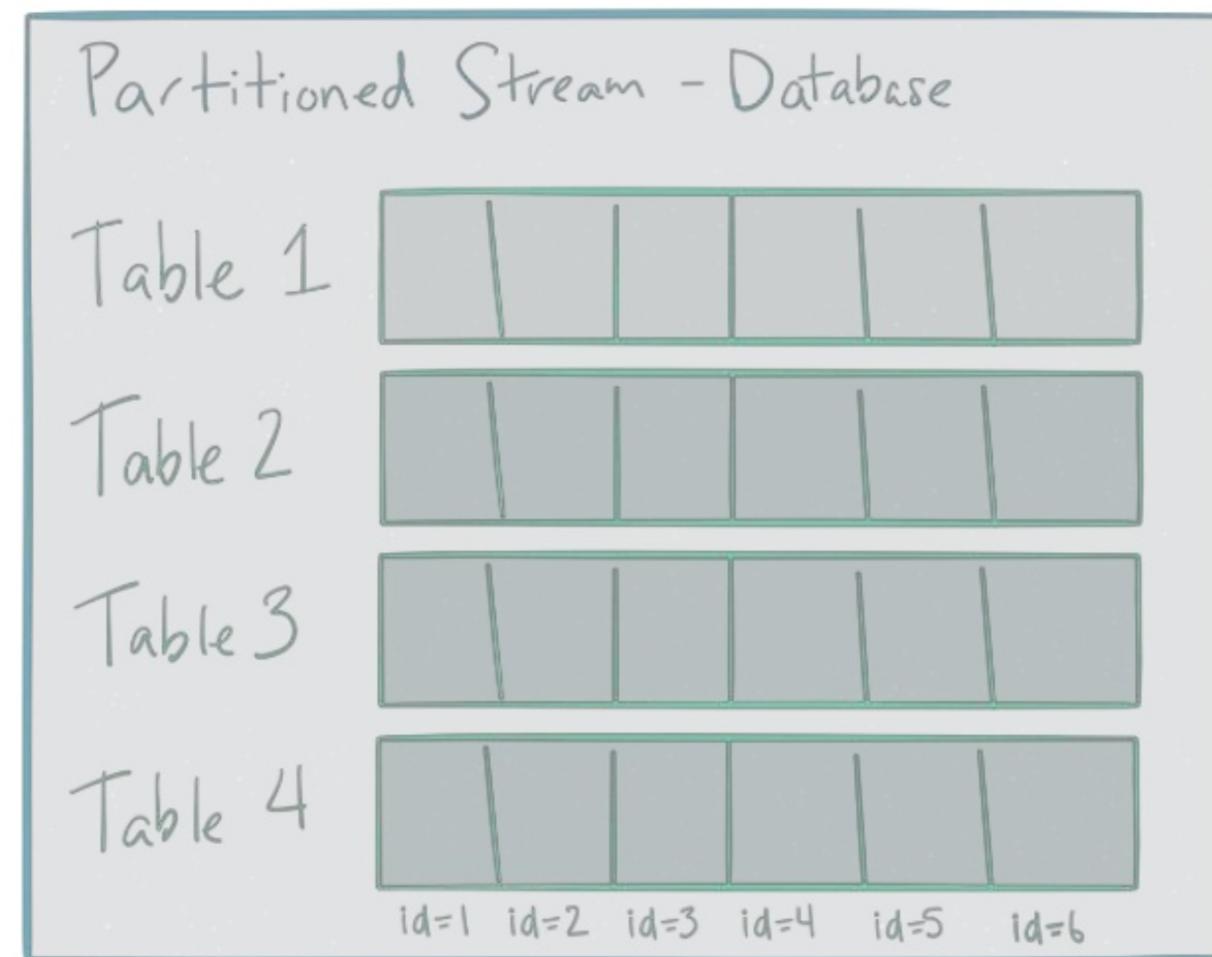
Providing Parallelism and Scalability (1)

- In Kafka, we split a topic (stream) into partitions to parallelize work.

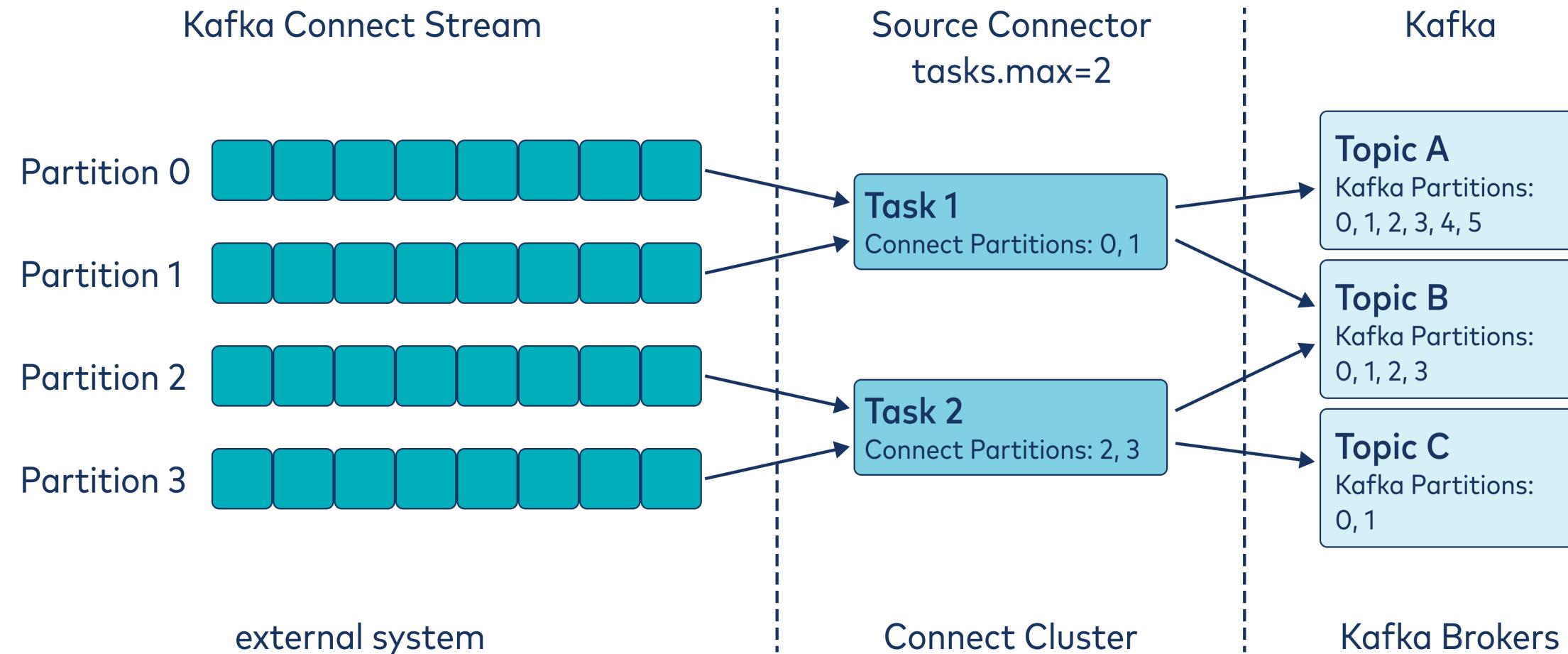


Providing Parallelism and Scalability (2)

- For a Source Connector, a partition is a subset of data in the source system.



Providing Parallelism and Scalability (3)



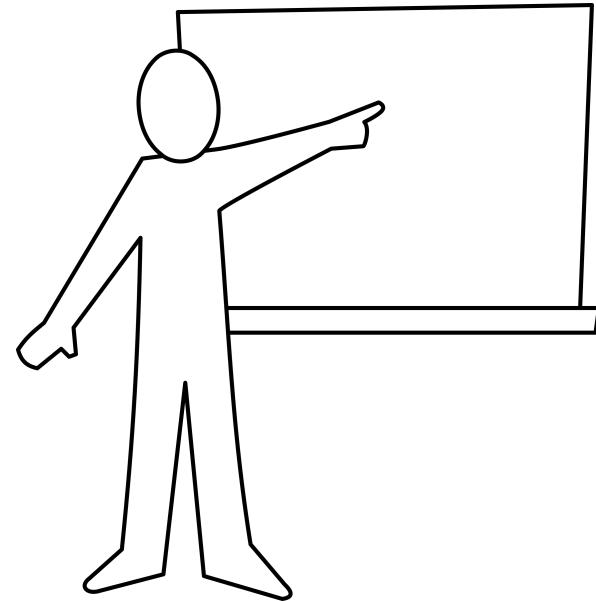
Providing Parallelism and Scalability (4)

- Partitioning → parallelism & scalability
- 1 Connector job → 1..n **tasks**
- 1 Worker → 1..n **tasks**
- 1 Task per Thread

Source and Sink Offsets

- Kafka Connect tracks the produced and consumed offsets so it can restart at the correct place in case of failure
- What the offset corresponds to depends on the Connector, for example:
 - File input: offset → position in file
 - Database input: offset → timestamp or sequence id
- The method of tracking the offset depends on the specific Connector
 - Each Connector can determine its own way of doing this
- Examples of source and sink offset tracking methods:
 - JDBC source in distributed mode: a special Kafka Topic
 - HDFS sink: an HDFS file
 - FileStream source: a separate local file

Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes ... ←
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples
- 🤖 Hands-on Lab

Two Modes: Standalone and Distributed

Standalone mode

- 1 worker on 1 machine
- When to use:
 - testing and development
 - non distributable process

Distributed mode

- n workers on m machines
- When to use:
 - production environments
 - for **fault tolerance**
 - and **scalability**

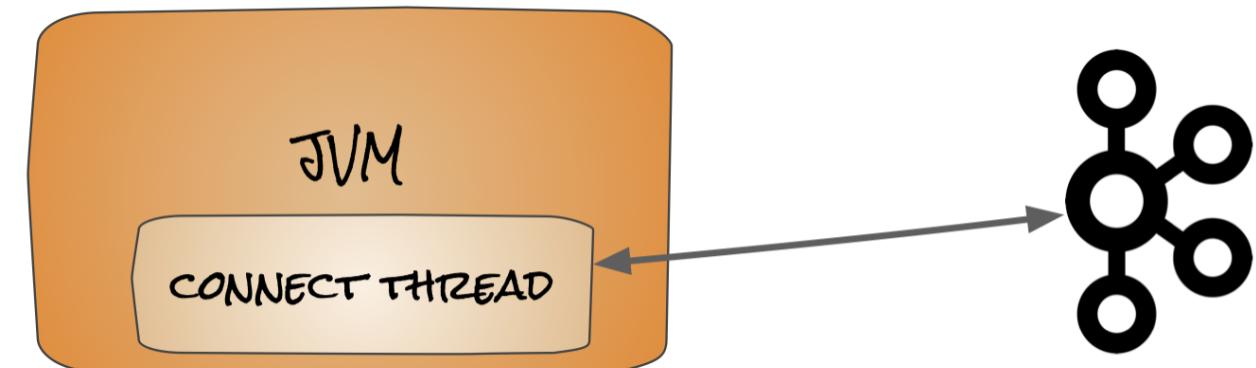
Running in Standalone Mode

To run in standalone mode, start a process by providing as arguments

- Standalone config properties file
- 1...n connector config files



Each connector instance runs in own thread



```
$ connect-standalone connect-standalone.properties \
  connector1.properties [connector2.properties connector3.properties ...]
```

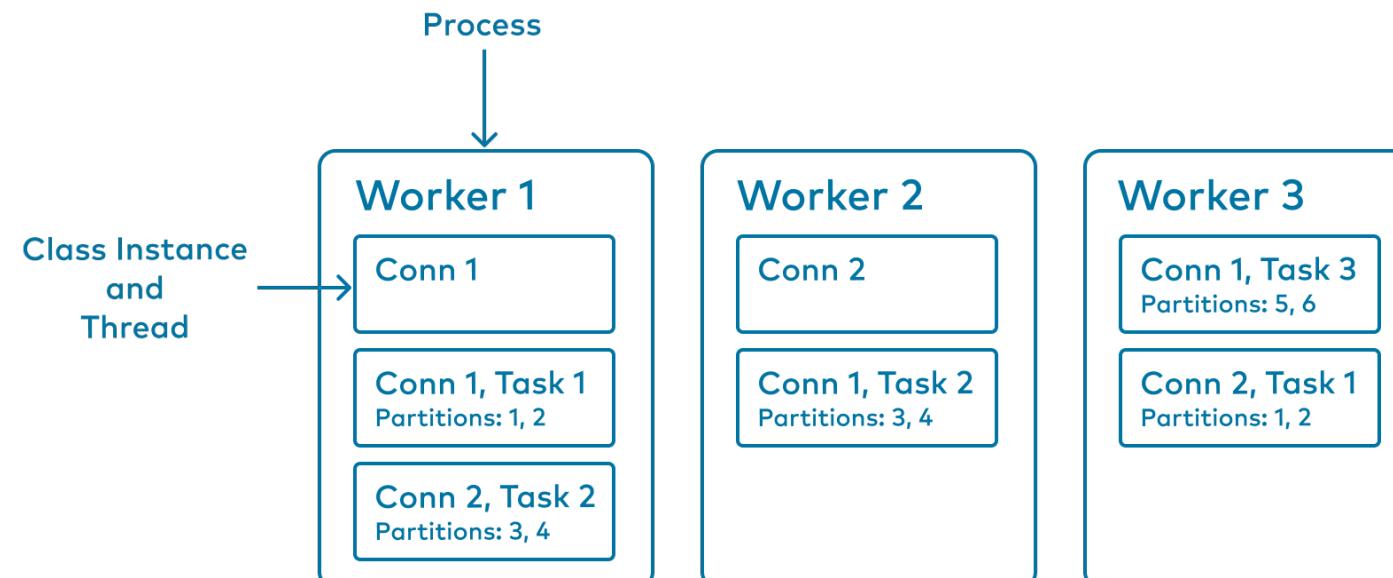
Running in Distributed Mode

Start Kafka Connect **on each** worker node

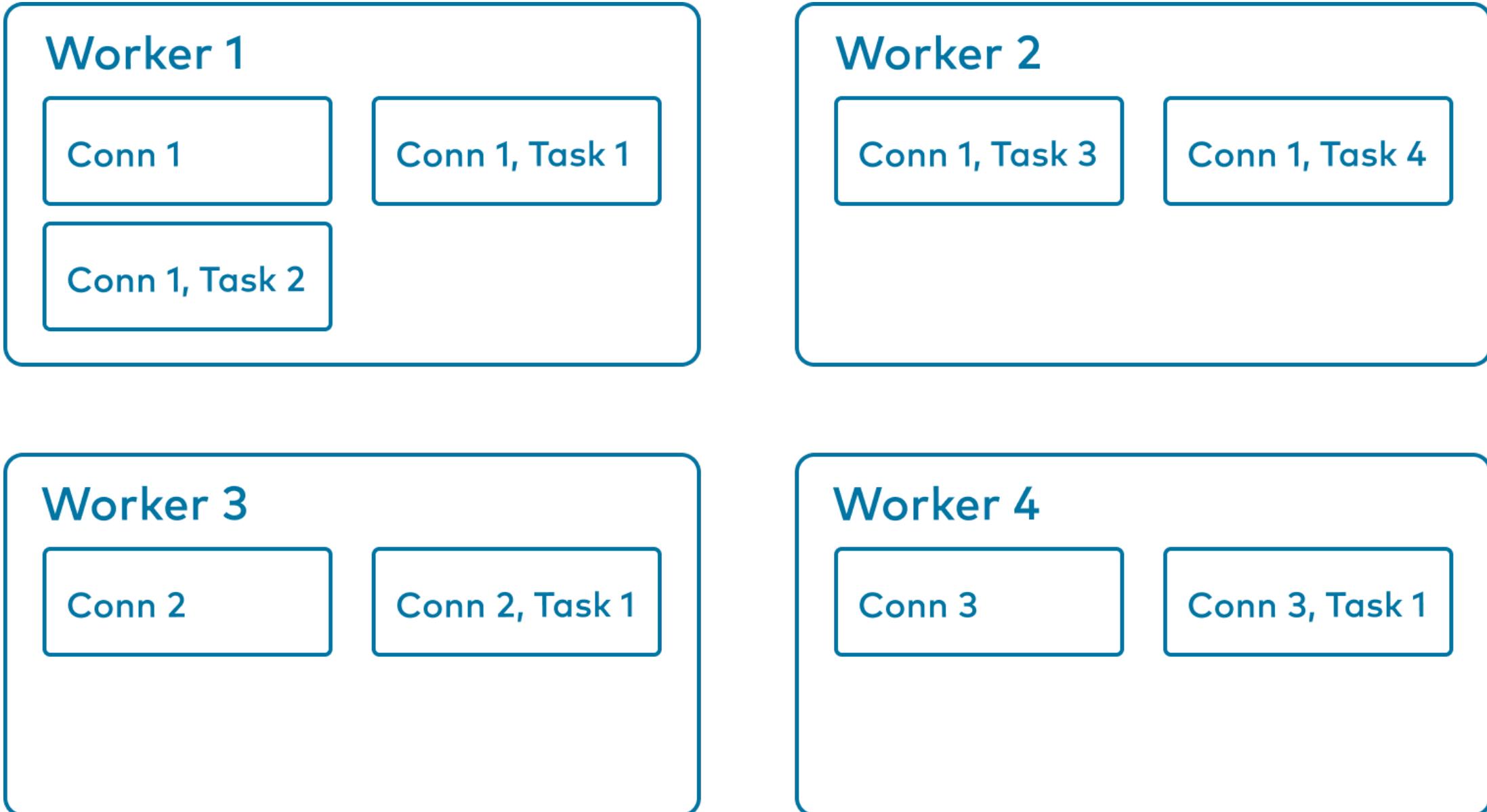
```
$ connect-distributed connect-distributed.properties
```

Group coordination

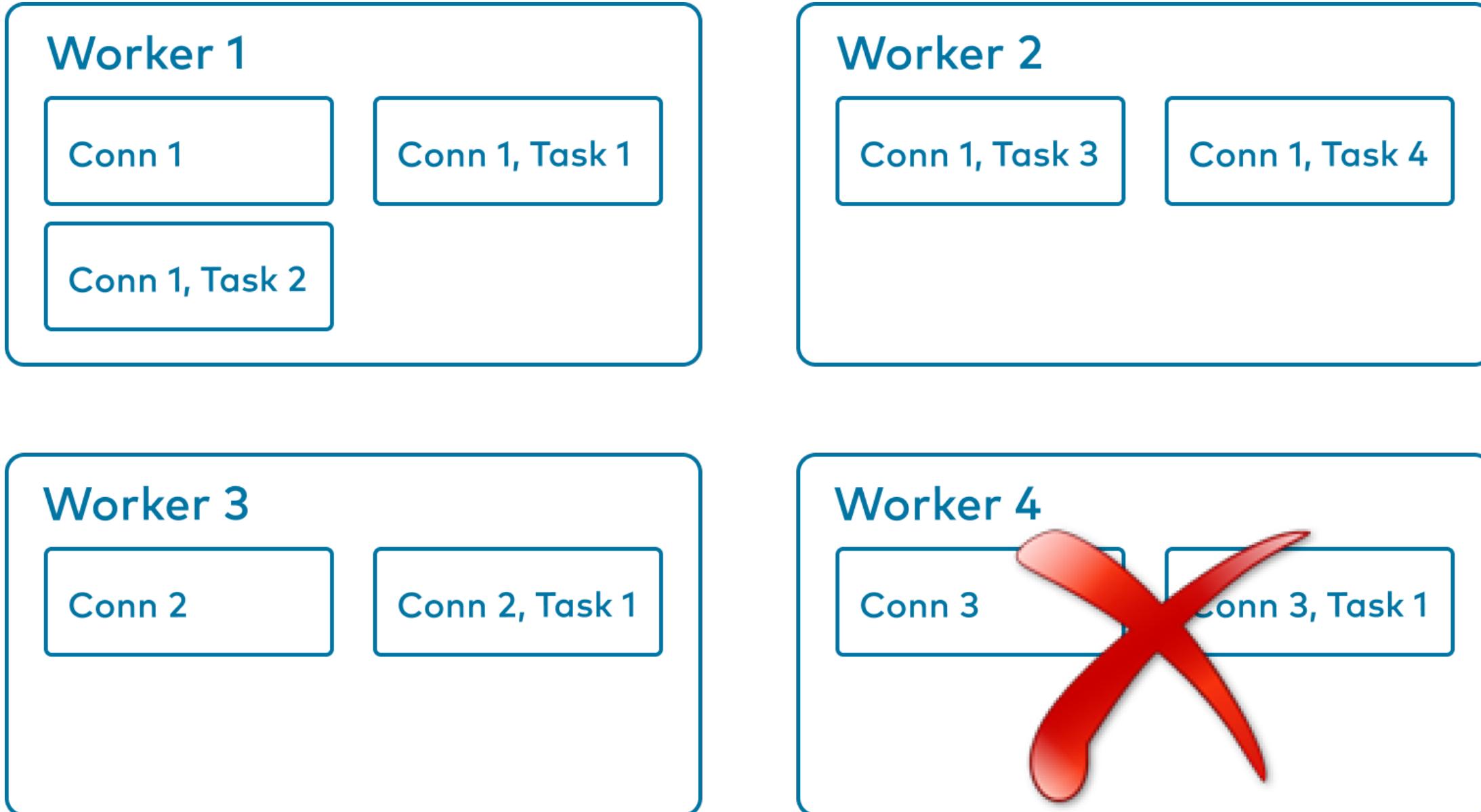
- Connect leverages Kafka's group membership protocol
 - Configure workers with the same `group.id`
- Workers distribute load within this Kafka Connect "cluster"



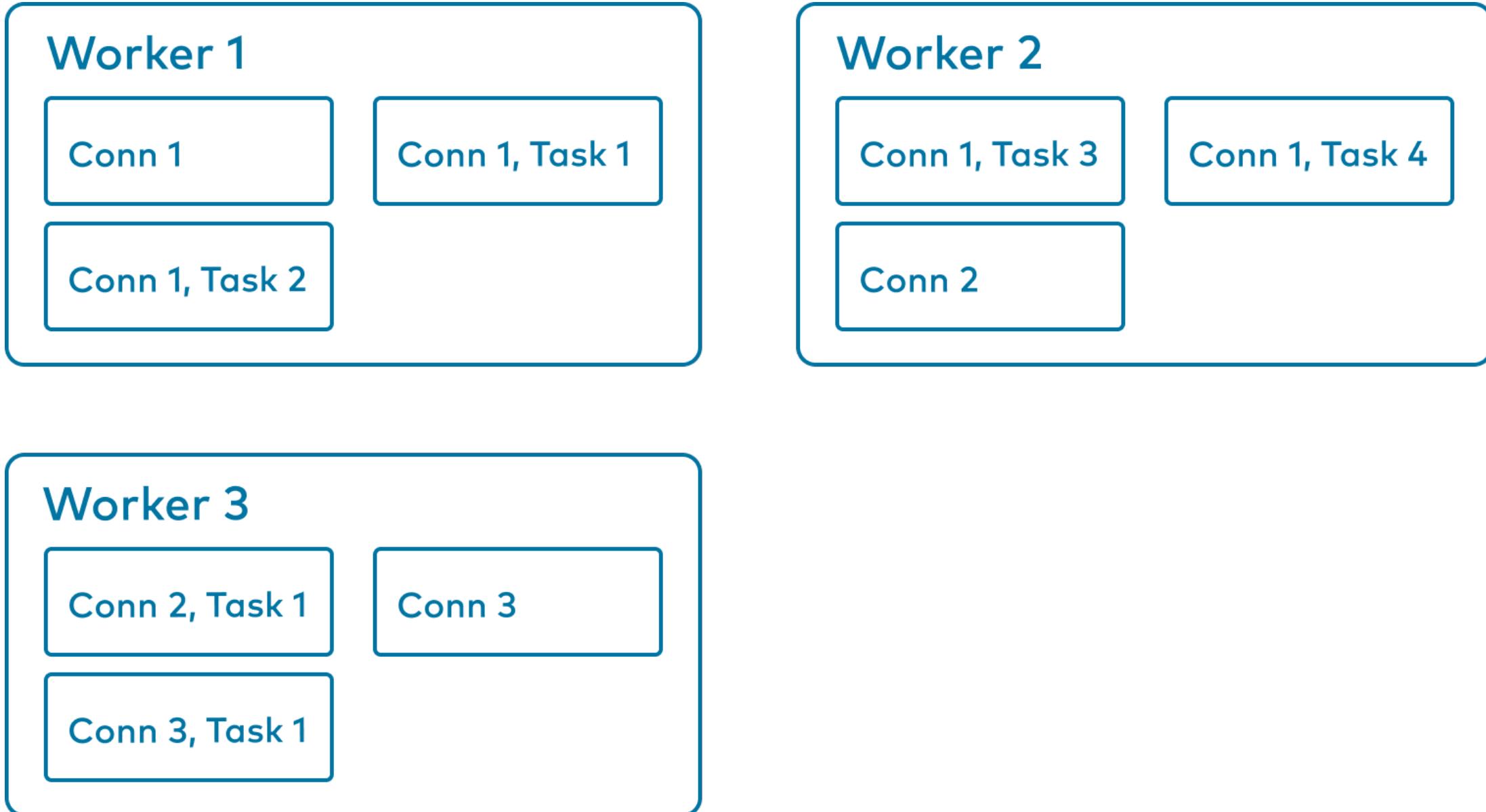
Fail-over 1/3



Fail-over 2/3



Fail-over 3/3



Configuring Workers: Both Modes

- You can modify Connect configuration settings
 - Distributed mode in `/etc/kafka/connect-distributed.properties`
 - Standalone mode in `/etc/kafka/connect-standalone.properties`
- Important configuration options common to all workers:

Property	Description
<code>bootstrap.servers</code>	A list of host/port pairs to use to establish the initial connection to the Kafka cluster
<code>key.converter</code>	Converter class for the key
<code>value.converter</code>	Converter class for the value



Customize producer (source) or consumer (sink) properties using prefixes, for example: `producer.batch.size=2048`

Configuring Workers: Standalone Mode

Property	Description
<code>offset.storage.file.filename</code>	The filename in which to store offset data for the Connectors (Default: ""). This enables a standalone process to be stopped and then resume where it left off.

Configuring Workers: Distributed Mode

Property	Description
<code>group.id</code>	A unique string that identifies the Kafka Connect cluster group the worker belongs to
<code>session.timeout.ms</code>	Timeout used to detect worker failures
<code>heartbeat.interval.ms</code>	Time between heartbeats to the group coordinator. Must be smaller than <code>session.timeout.ms</code>
<code>config.storage.topic</code>	Topic in which to store Connector & task config. data
<code>offset.storage.topic</code>	Topic in which to store offset data for Connectors
<code>status.storage.topic</code>	Topic in which to store connector & task status
<code>topic.creation.enable</code>	Allow source connector configurations to define topic creation settings (Default: true)

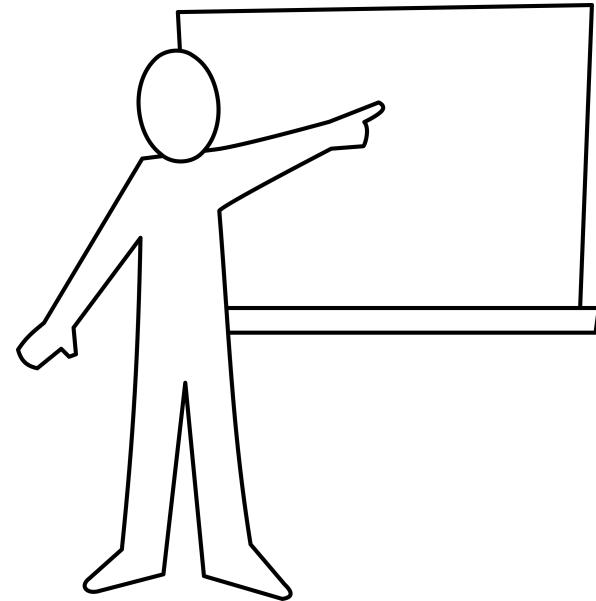
Creating Kafka Connect's Topics Manually

- The Topics used in Kafka Connect distributed mode will be **automatically created**
- They are created with **recommended values** for
 - replication factor
 - partition counts
 - cleanup policy

The following number of Partitions:

Topic	Partitions
config.storage.topic	1
offset.storage.topic	25
status.storage.topic	5

Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors ... ←
- Single Message Transforms (SMTs)
- Examples
- 🤖 Hands-on Lab

Configuring Connectors with Confluent Control Center

The image displays two screenshots of the Confluent Control Center interface, illustrating the process of adding a new connector.

Browse Page: Shows a list of connectors categorized by source and sink. The categories include ActiveMQSource Connector, JdbcSource Connector, and FileStreamSink. Each category has a "Connect" button.

Add Connector Page: Shows the configuration steps for a JdbcSource Connector.

- Step 01: SETUP CONNECTION**
 - Connector class:** io.confluent.connect.jdbc.JdbcSourceConnector
 - name:** JdbdSourceConnector
- Step 02: TEST AND VERIFY** (This step is currently not visible in the screenshot)

Right Sidebar (Topics):

- Common
- Transforms
- Error Handling
- Database
- Mode
- Connector
- Additional Properties

Configuring Connectors with the REST API

- Add, modify delete connectors
- Distributed Mode:
 - Config **only** via REST API
 - Config stored in Kafka topic
 - REST call to **any** worker
- Standalone Mode:
 - Config also via REST API
 - Changes **not persisted!**
- Control Center uses REST API

Using the REST API

Some important REST endpoints

Method	Path	Description
GET	/connectors	Get a list of active connectors
POST	/connectors	Create a new Connector
GET	/connectors/(string: name)/config	Get configuration information for a Connector
PUT	/connectors/(string: name)/config	Create a new Connector, or update the configuration of an existing Connector

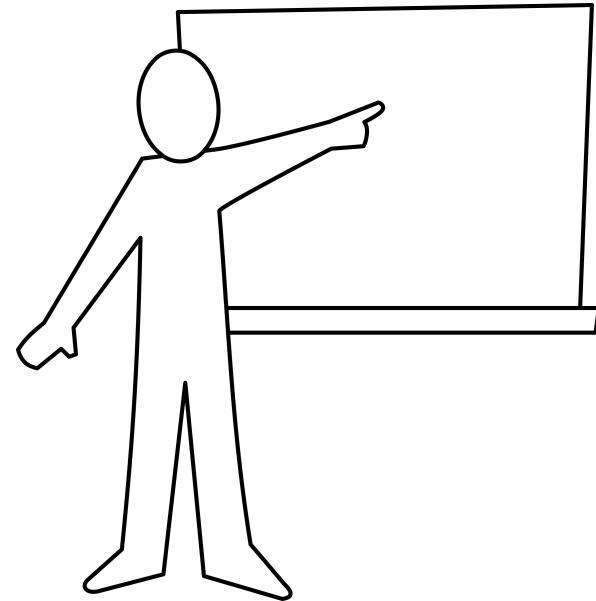
Configuring the Connector

Property	Description
<code>name</code>	Connector's unique name
<code>connector.class</code>	Connector's Java class
<code>tasks.max</code>	Maximum tasks to create. The Connector may create fewer if it cannot achieve this level of parallelism (Default: 1)
<code>key.converter</code>	(optional) Override the worker key converter
<code>value.converter</code>	(optional) Override the worker value converter
<code>topics</code> (Sink connectors only)	List of input topics to consume from

Overriding Connect Worker Properties

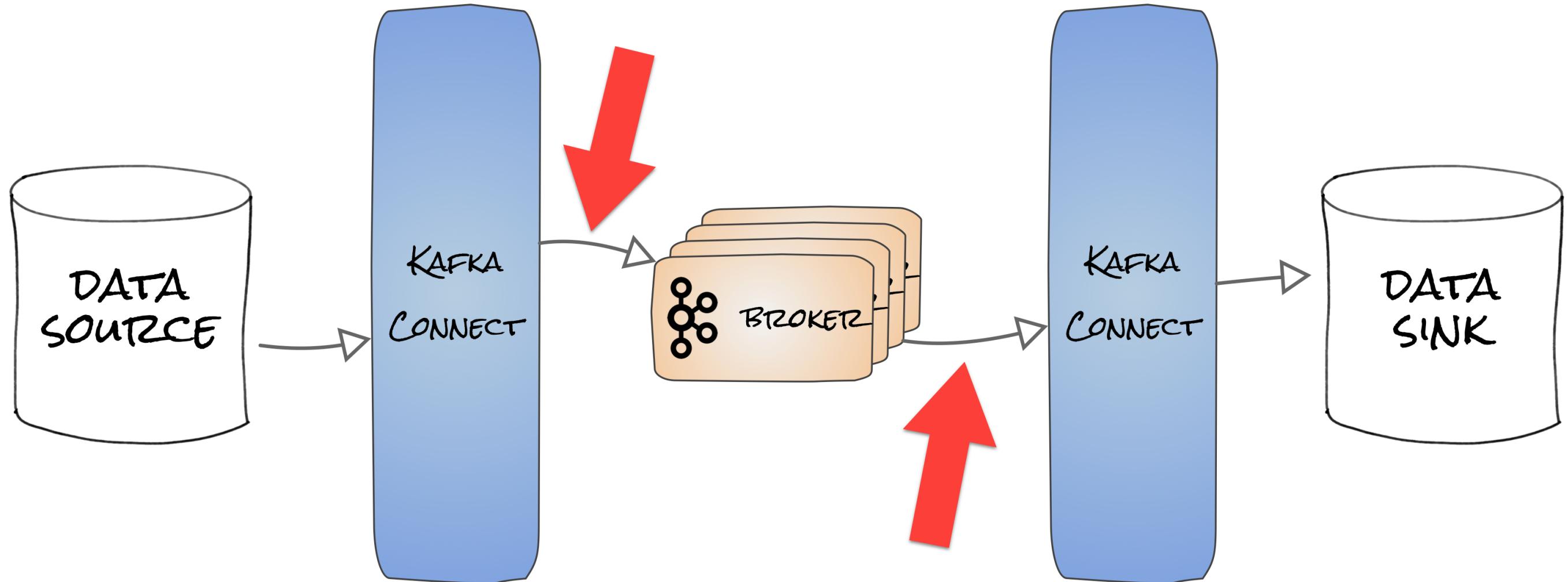
- By default, a worker's configurations apply to **all** connectors running on the worker
- Customize a connector's settings with `connector.client.config.override.policy`
 - Overrides worker configurations with `producer`, `consumer`, or `admin` prefixes
 - `None` (default): use worker settings
 - `Principal`: override security settings
 - `All`: override any prefixed settings
 - You can create your own custom `ConnectorClientConfigOverridePolicy` class
- Granular control over security and performance for individual connectors

Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs) ... ←
- Examples
-  Hands-on Lab

Transforming Data with Connect



Single Message Transform (SMT)

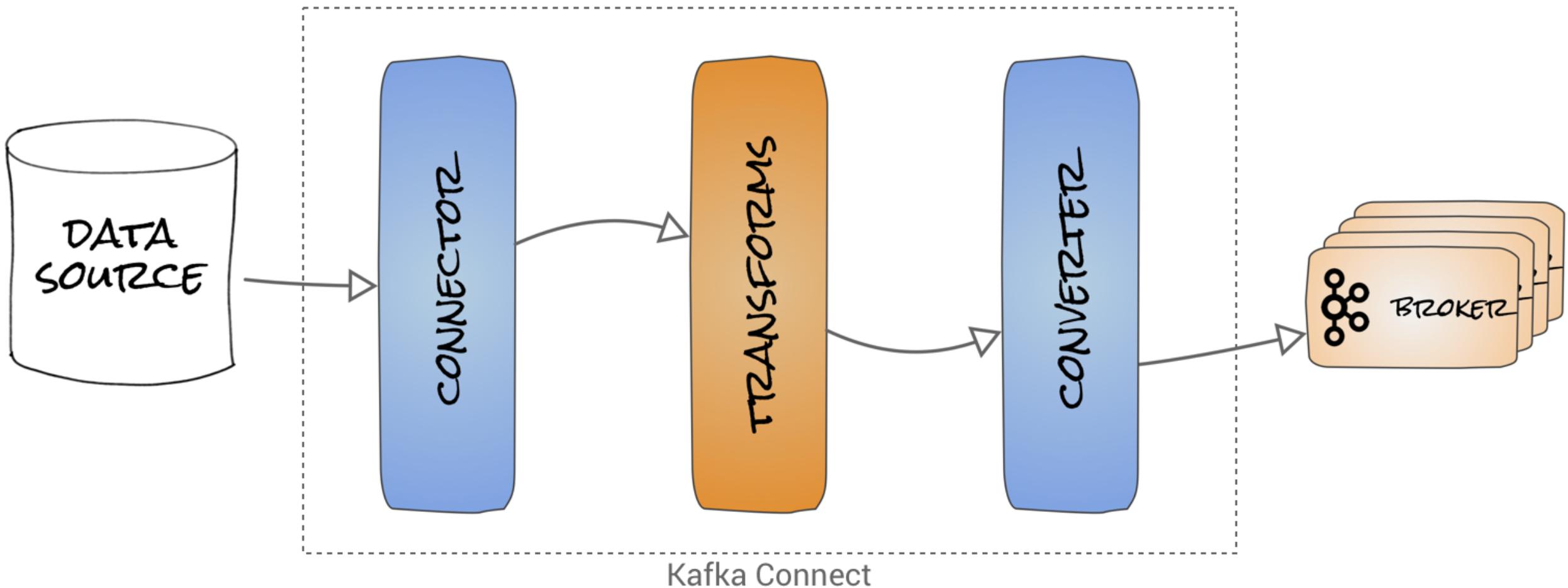
Modify events before storing in Kafka:

- Mask sensitive information
- Add identifiers
- Tag events
- Lineage/provenance
- Remove unnecessary columns

Modify events going out of Kafka:

- Route high priority events to faster data stores
- Direct events to different Elasticsearch indices
- Cast data types to match destination
- Remove unnecessary columns

Where SMTs Live



Single Message Transform - Details (1)

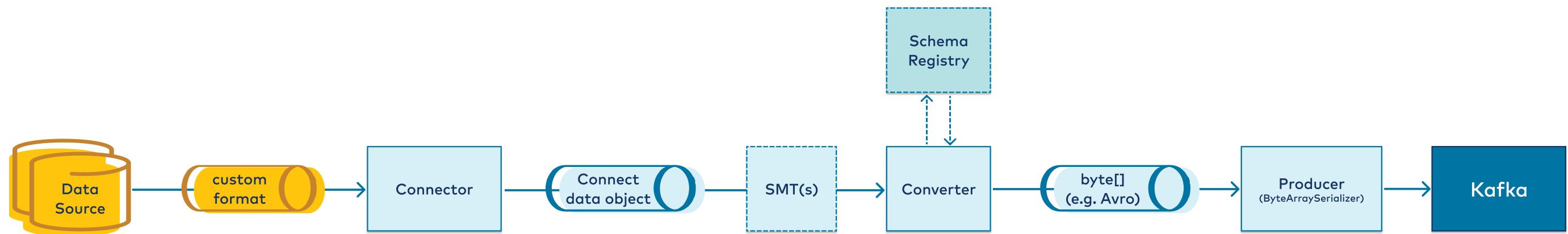
Transform	Description
InsertField	insert a field using attributes from message metadata or from a configured static value
ReplaceField	rename fields, or apply a blacklist or whitelist to filter
MaskField	replace field with valid <code>null</code> value for the type (0, empty string, etc)
ValueToKey	replace the key with a new key formed from a subset of fields in the value payload
HoistField	wrap the entire event as a single field inside a <code>Struct</code> or a <code>Map</code>
ExtractField	extract a specific field from <code>Struct</code> and <code>Map</code> and include only this field in results
SetSchemaMetadata	modify the schema name or version
TimestampRouter	modify the topic of a record based on the original topic name and timestamp
RegexRouter	update a record topic using the configured regular expression and replacement string

Single Message Transform - Details (2)

Transform	Description
Flatten	flatten a nested data structure, generating names for each field by concatenating the field names at each level with a configurable delimiter character.
Cast	cast fields or the key/value to a specific type (e.g. to force an integer field to a smaller width)
TimestampConverter	convert timestamps between different formats
Filter	conditionally to filter out records matching (or not matching) a particular predicate
HasHeaderKey	a predicate which is true for records with at least one header with the configured name
RecordIsTombstone	a predicate which is true for records which are tombstones (i.e. have null value)
TopicNameMatches	a predicate which is true for records with a topic name that matches the configured regular expression

Converting Data

- Converters provide the data format written to or read from Kafka (like Serializers)
- Converters are decoupled from Connectors so they can be reused
 - Allows any connector to work with any serialization format
- Example of format conversion in a source converter (sink converter is the reverse)



Converter Data Formats

- Converters apply to both the key and value of the message
 - Key and value converters can be set independently
 - `key.converter`
 - `value.converter`
- Pre-defined data formats for Converter
 - Avro: `AvroConverter`
 - Byte Array: `ByteArrayConverter`
 - JSON: `JsonConverter`
 - JSON Schema: `JsonSchemaConverter`
 - Protobuf: `ProtobufConverter`
 - String: `StringConverter`

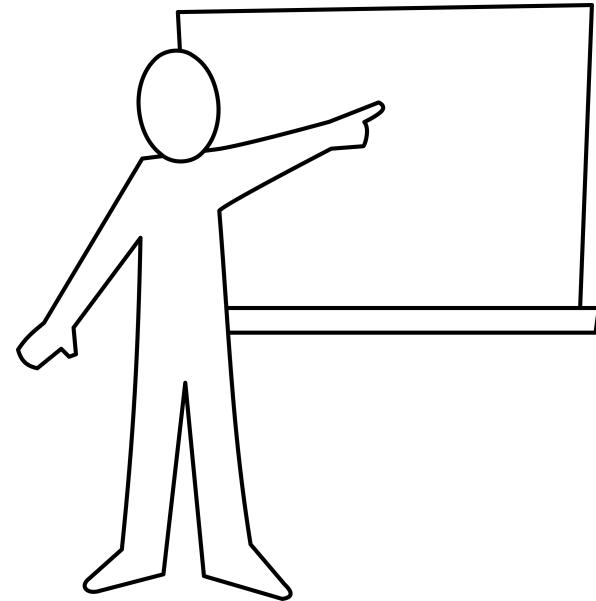
Converter Best Practice

- Use a converter that supports Schema Registry!
 - Avro, JSON Schema, and Protobuf converters
- This gives connectors all the Schema Registry benefits that regular producers and consumers enjoy:
 - Planned schema evolution
 - Schemas durably persisted in Kafka
 - Each message is sent with a schema id instead of the whole schema

connect-distributed.properties

```
...
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://schemaregistry1:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://schemaregistry1:8081
```

Module Map



- The Motivation for Kafka Connect
- Types of Connectors
- Kafka Connect Implementation
- Standalone and Distributed Modes
- Configuring the Connectors
- Single Message Transforms (SMTs)
- Examples ... ←
- 🤖 Hands-on Lab

Example: JDBC Source Connector

- Java Database Connectivity (JDBC) API is common amongst databases
- JDBC Source Connector is a great way to get database tables into Kafka topics
- JDBC Source periodically polls a relational database for new or recently modified rows
 - Creates a record for each row, and Produces that record as a Kafka message
- Each table gets its own Kafka topic
- New and deleted tables are handled automatically

Query Mode (1)

Incremental query mode	Description
Incrementing column	Check a single column where newer rows have a larger, auto-incremented ID. Does not capture updates to existing rows.
Timestamp column	Checks a single 'last modified' column to capture new rows and updates to existing rows. If task crashes before all rows with the same timestamp have been processed, some updates may be lost.
Timestamp and incrementing column	Detects new rows and updates to existing rows with fault tolerance. Uses timestamp column, but reprocesses current timestamp upon task failure. Incrementing column then prevents duplicate processing.

Query Mode (2)

Incremental query mode	Description
Custom query	Used in conjunction with the options above for custom filtering.
Bulk	Load all rows in the table. Does not detect new or updated rows.

Configuration

Property	Description
<code>connection.url</code>	The JDBC connection URL for the database
<code>topic.prefix</code>	The prefix to prepend to table names to generate the Kafka Topic name
<code>mode</code>	The mode for detecting table changes. Options are <code>bulk</code> , <code>incrementing</code> , <code>timestamp</code> , <code>timestamp+incrementing</code>
<code>query</code>	The custom query to run, if specified
<code>poll.interval.ms</code>	The frequency in milliseconds to poll for new data in each table (Default: 5000)
<code>table.blacklist</code>	A list of tables to ignore and not import. If specified, <code>tables.whitelist</code> cannot be specified
<code>table.whitelist</code>	A list of tables to import. If specified, <code>tables.blacklist</code> cannot be specified

JDBC Source Connector with SMTs (1)

```
1 {
2   "name": "Driver-Connector",
3   "config": {
4     "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
5     "connection.url": "jdbc:postgresql://postgres:5432/postgres",
6     "connection.user": "postgres",
7     "table.whitelist": "driver",
8     "topic.prefix": "",
9     "mode": "timestamp+incrementing",
10    "incrementing.column.name": "id",
11    "timestamp.column.name": "timestamp",
12    "table.types": "TABLE",
13    "numeric.mapping": "best_fit",
```

JDBC Source Connector with SMTs (2)

```
14  "transforms": "suffix,createKey,extractKey",
15  "transforms.suffix.type": "org.apache.kafka.connect.transforms.RegexRouter",
16  "transforms.suffix.regex": "(.*)",
17  "transforms.suffix.replacement": "$1-profiles-avro",
18  "transforms.createKey.type": "org.apache.kafka.connect.transforms.ValueToKey",
19  "transforms.createKey.fields": "driverkey",
20  "transforms.extractKey.type": "org.apache.kafka.connect.transforms.ExtractField$Key",
21  "transforms.extractKey.field": "driverkey"
22 }
23 }
```

Example: HDFS Sink Connector

- Continuously polls from Kafka and writes to HDFS (Hadoop Distributed File System)
- Integrates with Hive
 - Auto table creation
 - Schema evolution with Avro
- Works with secure HDFS and the Hive Metastore, using Kerberos
- Provides exactly once delivery
- Data format is extensible
 - Avro, Parquet, custom formats
- Pluggable Partitioner, supporting:
 - Kafka Partitioner (default)
 - Field Partitioner
 - Time Partitioner
 - Custom Partitioners

Example: FileStream Connector

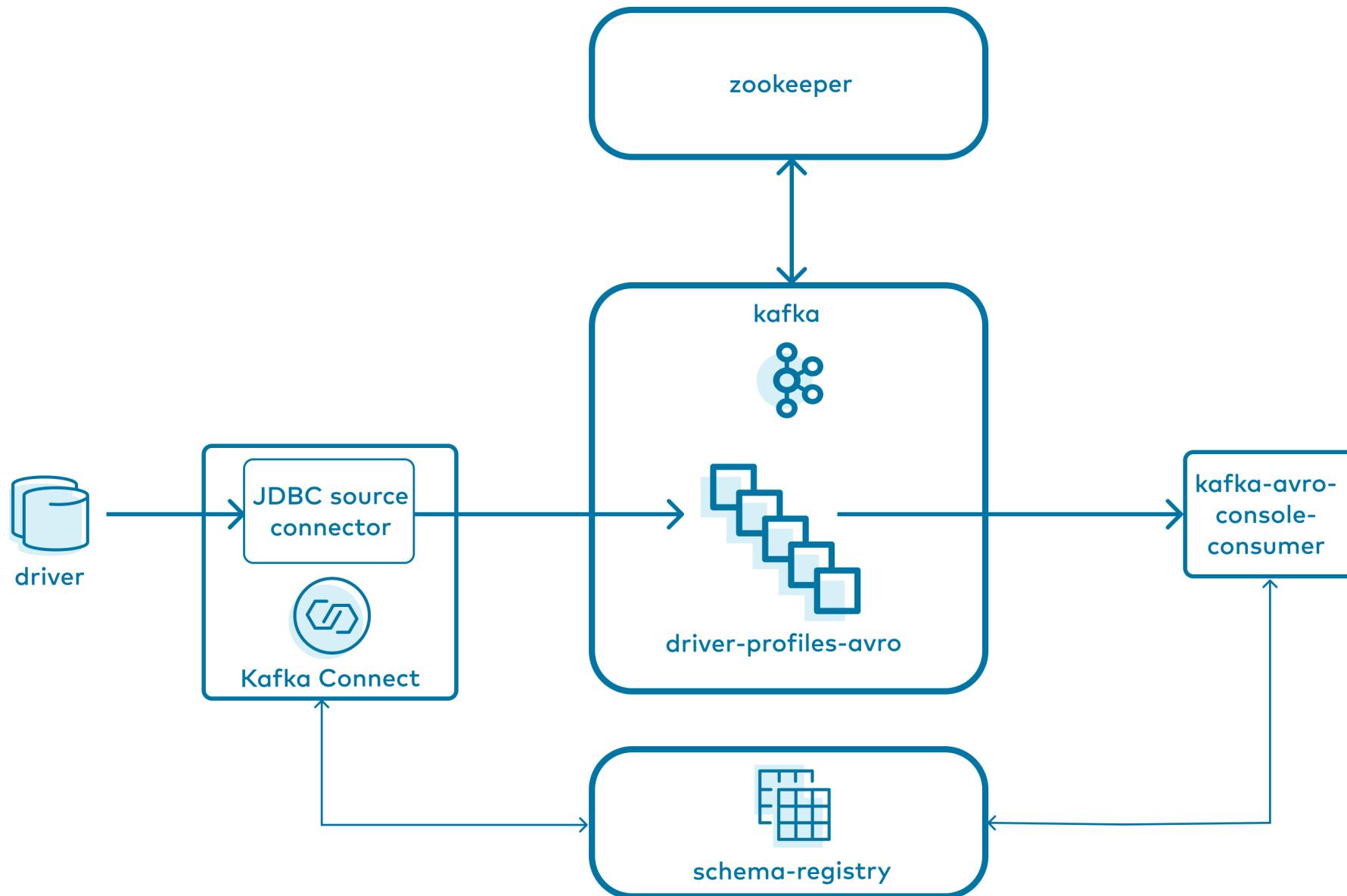
- Great tool for learning about connectors
- FileStream Connector acts on a local file
 - Local file Source Connector: tails local file and sends each line as a Kafka message
 - Local file Sink Connector: Appends Kafka messages to a local file

Hands-On Lab

- In this lab, you will enrich driver position data with driver profile data from a database in near-real time using the JDBC Connector.
- Please refer to **Lab 07 Data Pipelines with Kafka Connect** in the Exercise Book:
 - a. **Kafka Connect - Database to Kafka**



Kafka Connect - Database to Kafka



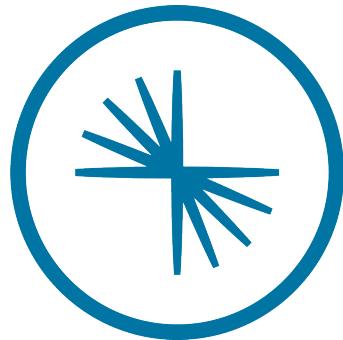
Module Review



Questions:

- Why would you use Kafka Connect?
- What kind of supported connectors are you aware of?
- What if no connector exists for your source or sink?

08 Event Streaming Apps with ksqlDB



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB ... ←
9. Design Decisions
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

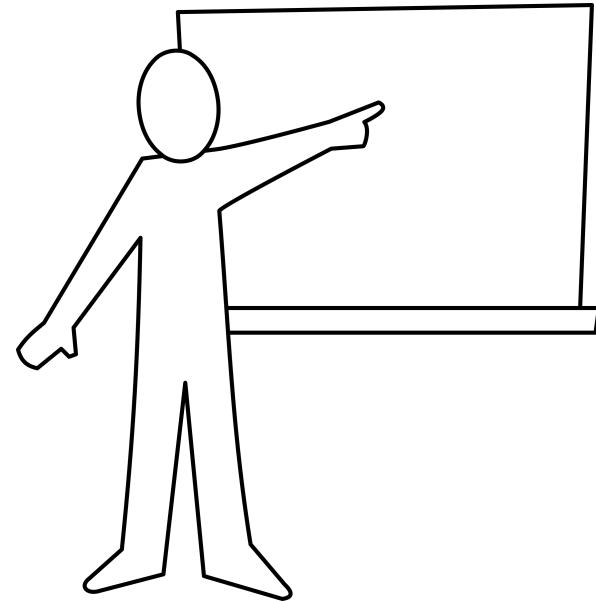
Learning Objectives



After this module you will be able to:

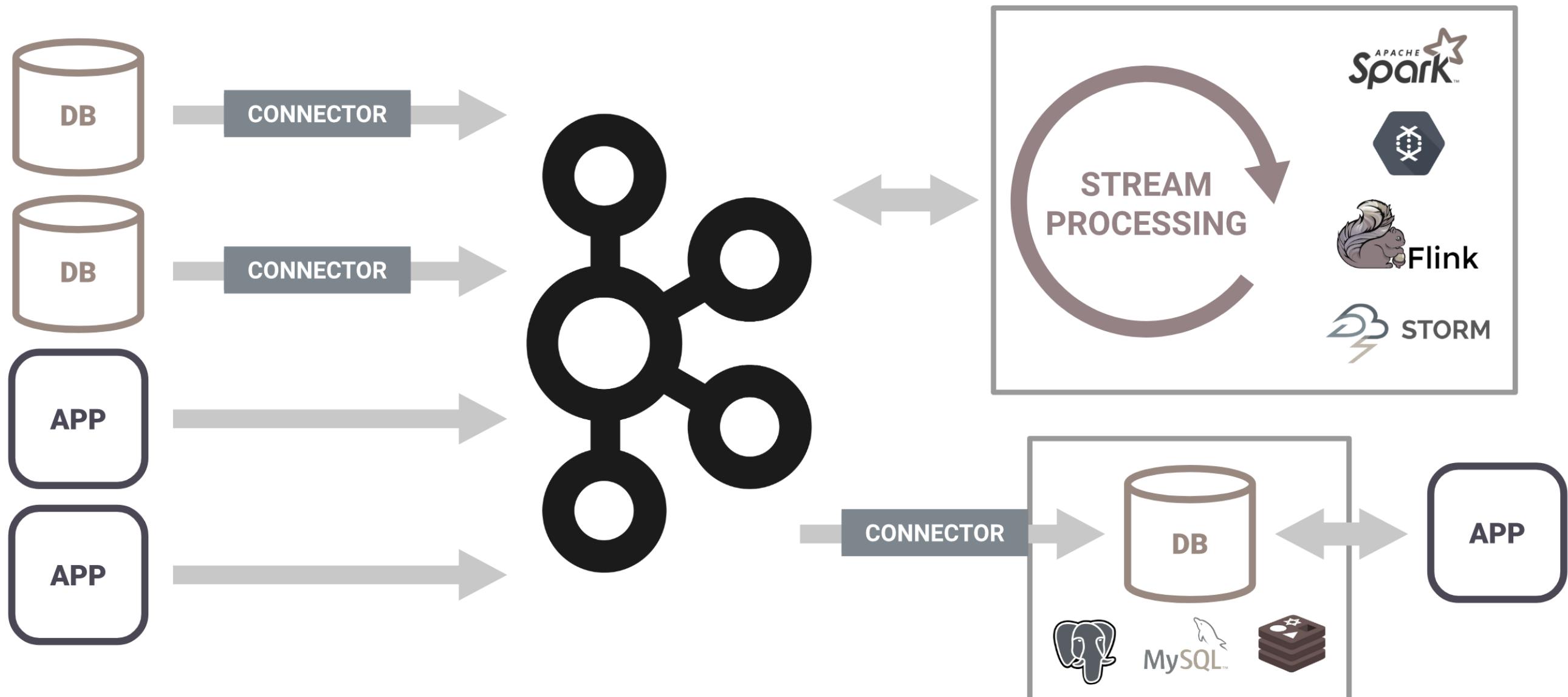
- use ksqlDB to filter and transform a stream
- write a ksqlDB query that joins two streams or a stream and a table
- write a ksqlDB query that aggregates values per key and time window
- write Push and Pull queries and explain the differences between them
- create a Connector with ksqlDB

Module Map



- What is ksqlDB? ... ←
- Queries
- Examples
- 🌐 Hands-on Lab

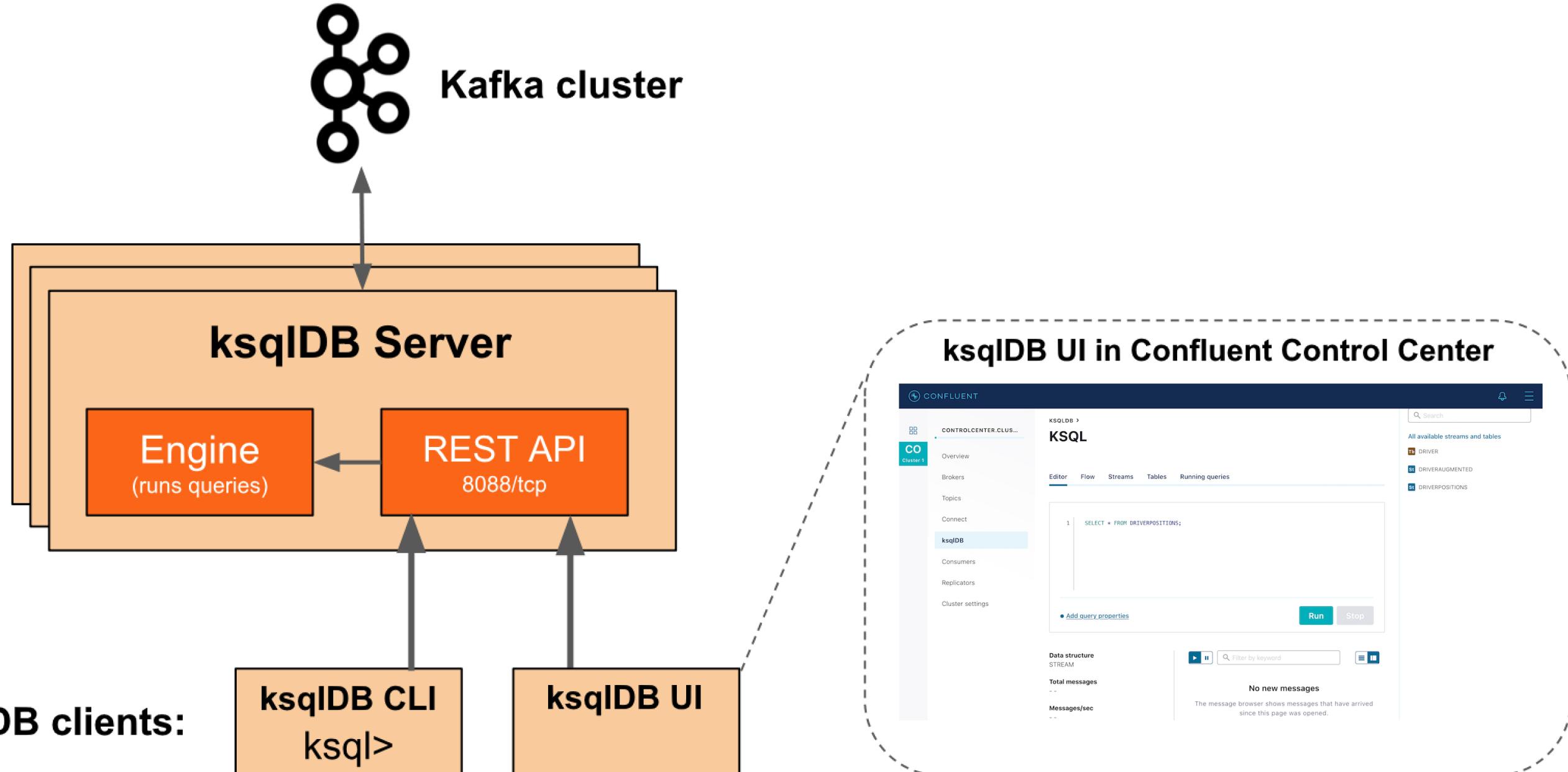
Stream Processing Apps Before ksqlDB



A Simpler Architecture for Event Streaming Apps

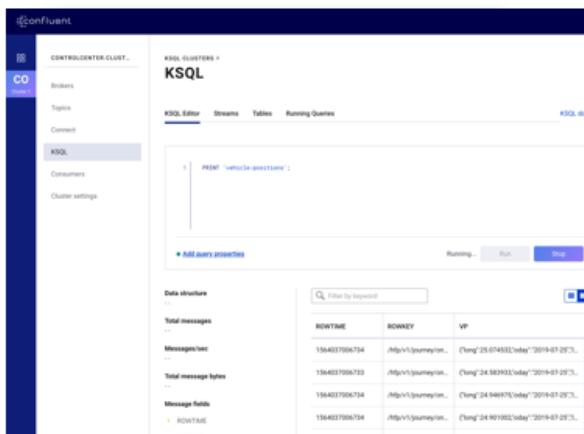


ksqldb Architecture and Components



ksqldb Can Be Accessed Interactively + Programmatically

1 UI



2 CLI



3 REST



4 Headless



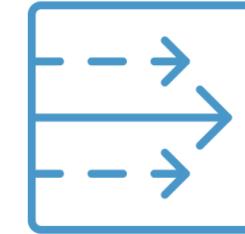
ksqldb Example Use Cases



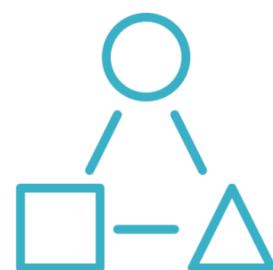
Data exploration



Data enrichment



Streaming ETL



Filter, cleanse, mask



Real-time monitoring



Anomaly detection

SQL-like Semantics

```
CREATE TABLE possible_fraud AS
  SELECT card_number, count(*)
  FROM authorization_attempts
  WINDOW HOPPING (SIZE 5 SECONDS, ADVANCE BY 1 SECOND)
  GROUP BY card_number
  HAVING count(*) > 3;
```

KSQL Statements

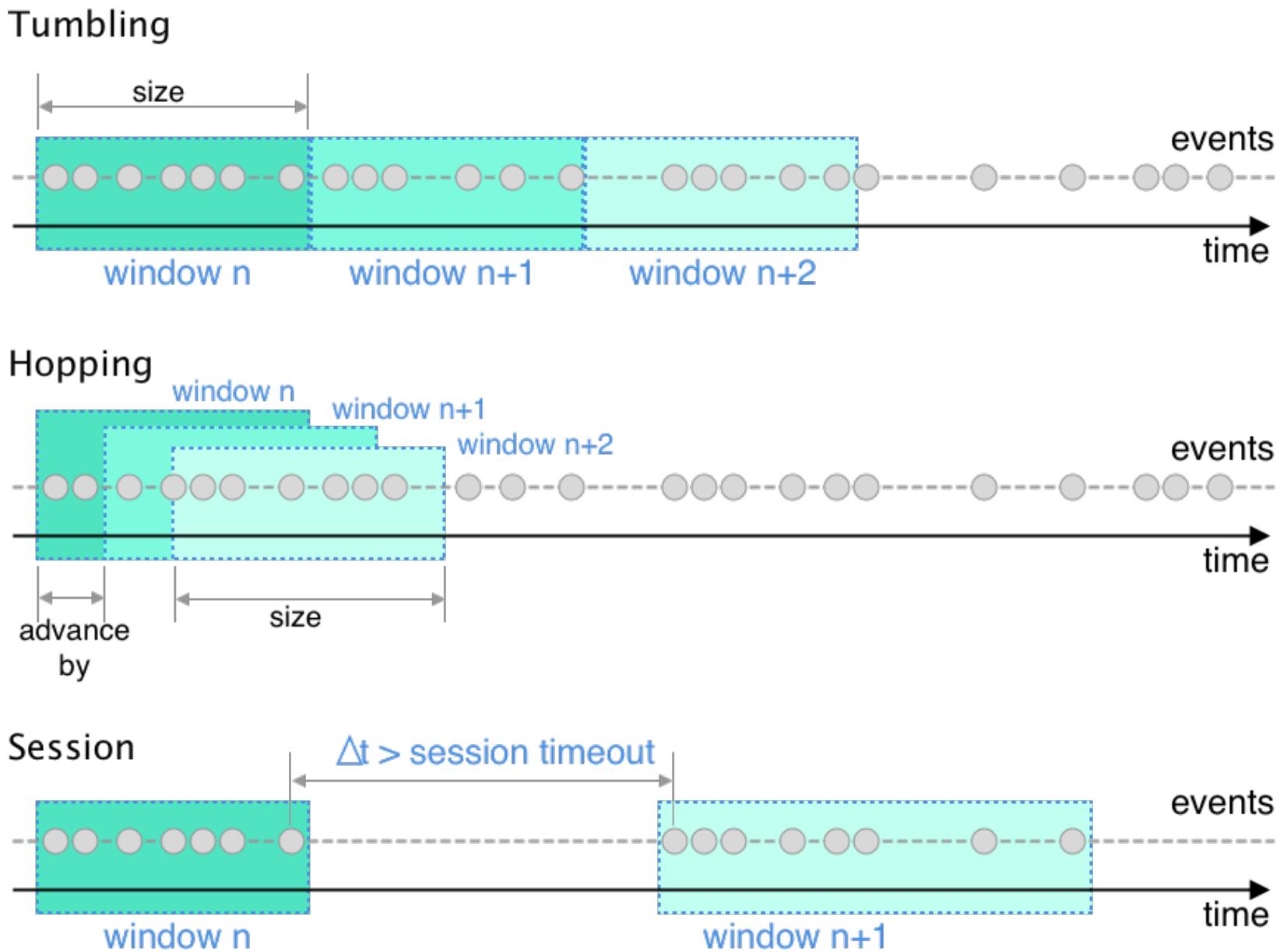
Command	Description
<code>CREATE</code>	Create streams, tables, Kafka Connect connectors, custom "types"
<code>SELECT</code>	Create push or pull queries
<code>SHOW</code>	Display information about streams, tables, types, properties, functions, topics queries, etc.
<code>DESCRIBE</code>	Display column names and types, metrics for streams and tables, or check status of connector
<code>DROP</code>	Remove an existing stream, table, connector, or type

ksqlDB Functions and Clauses

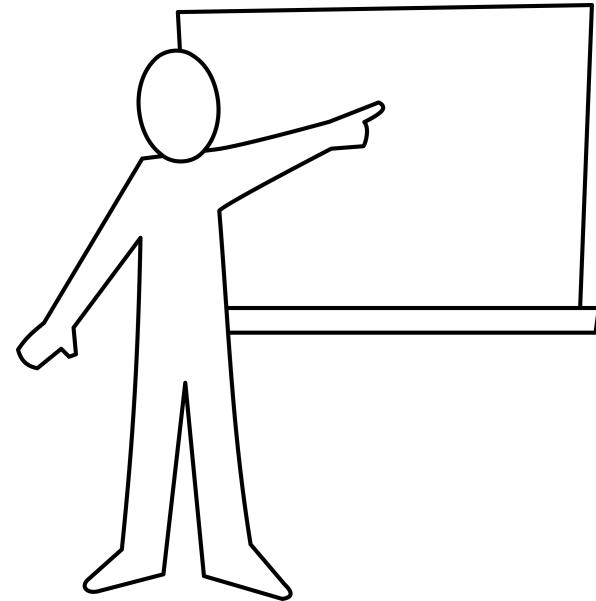
- Many popular scalar functions:
 - `ABS`, `CEIL`, `CONCAT`, `LEN`, `ROUND`, `TRIM`, `SUBSTRING`, and many more
 - `EXTRACTJSONFIELD`: Given a string column in JSON format, extract the field that matches
- You can `JOIN` streams and tables
- Aggregations
 - `COUNT`, `MAX`, `MIN`, `SUM`, etc
- Re-key the streams
 - `PARTITION BY column_name` → resulting stream will have new key set by specified column
 - `PARTITION BY MY_FUNCTION(x, 2)` → resulting stream will have new key set by a UDF response
- Extend ksqlDB with user-defined functions (UDFs), user-defined aggregate functions (UDAFs), and user-defined table functions (UDTFs)

Windowing in ksqlDB

- Windowing capabilities are the same as in Kafka Streams



Module Map



- What is ksqlDB?
- Queries ... ←
- Examples
- 🌐 Hands-on Lab

Non-persistent and Persistent Queries

- Non-persistent query
 - The result of a non-persistent query will not be persisted into a Kafka Topic and will only be printed out in the console
- Persistent query
 - The result of a persistent query will be persisted into a Kafka Topic

Command	Description
<i>SELECT</i>	Non-persistent
<i>CREATE STREAM AS SELECT</i>	Persistent
<i>CREATE TABLE AS SELECT</i>	Persistent

Pull Query

- Look up current value from a **materialized table**
- Single response returned in HTTP request

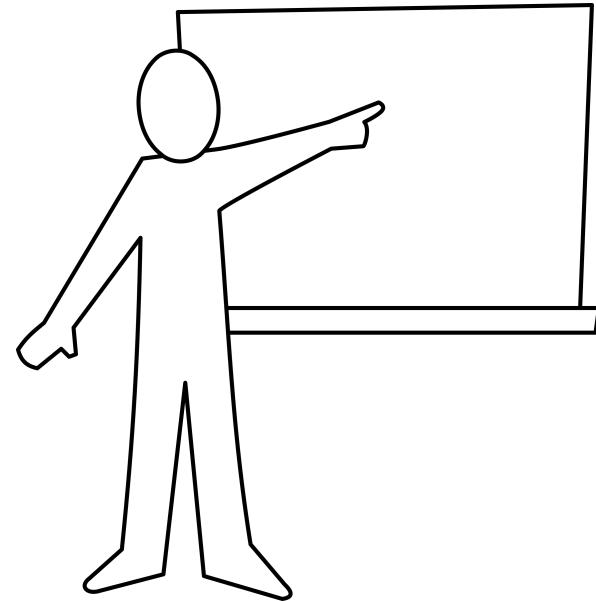
```
SELECT * FROM viewsPerUserSession
WHERE user_id = 'cool-kafka-user-5'
AND 1570051876000 <= WINDOWSTART
AND WINDOWSTART <= 1570123744000;
```

Push Query

- Subscribe to a **stream**
- Denoted by **EMIT CHANGES**
- Chunked HTTP response of indefinite length

```
SELECT * FROM user_table
EMIT CHANGES;
```

Module Map



- What is ksqlDB?
- Queries
- Examples ... ←
- 🌐 Hands-on Lab

CREATE Streams and Tables From Kafka Topics

- Create a stream from the Kafka topic `my-pageviews-topic`, which has a JSON payload

```
CREATE STREAM pageviews (viewtime BIGINT, user_id VARCHAR, page_id VARCHAR)
    WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-pageviews-topic');
```

- Create a table from the Kafka topic `my-users-topic`, which has a JSON payload

```
CREATE TABLE users (usertimestamp BIGINT, user_id VARCHAR PRIMARY KEY,
    gender VARCHAR, region_id VARCHAR)
    WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'my-users-topic');
```

Persistent Query With Join Example

- Enrich the `pageviews` stream by joining it with the `users` table on `user_id`

```
CREATE STREAM pageviews_enriched AS
  SELECT pv.user_id AS user_id,
         pv.viewtime,
         pv.page_id,
         u.gender,
         u.region_id
    FROM pageviews pv
  LEFT JOIN users u
      ON pv.user_id = u.user_id
   EMIT CHANGES;
```

Persistent Query With Windowing Example

- Aggregate the enriched pageviews stream by gender and region

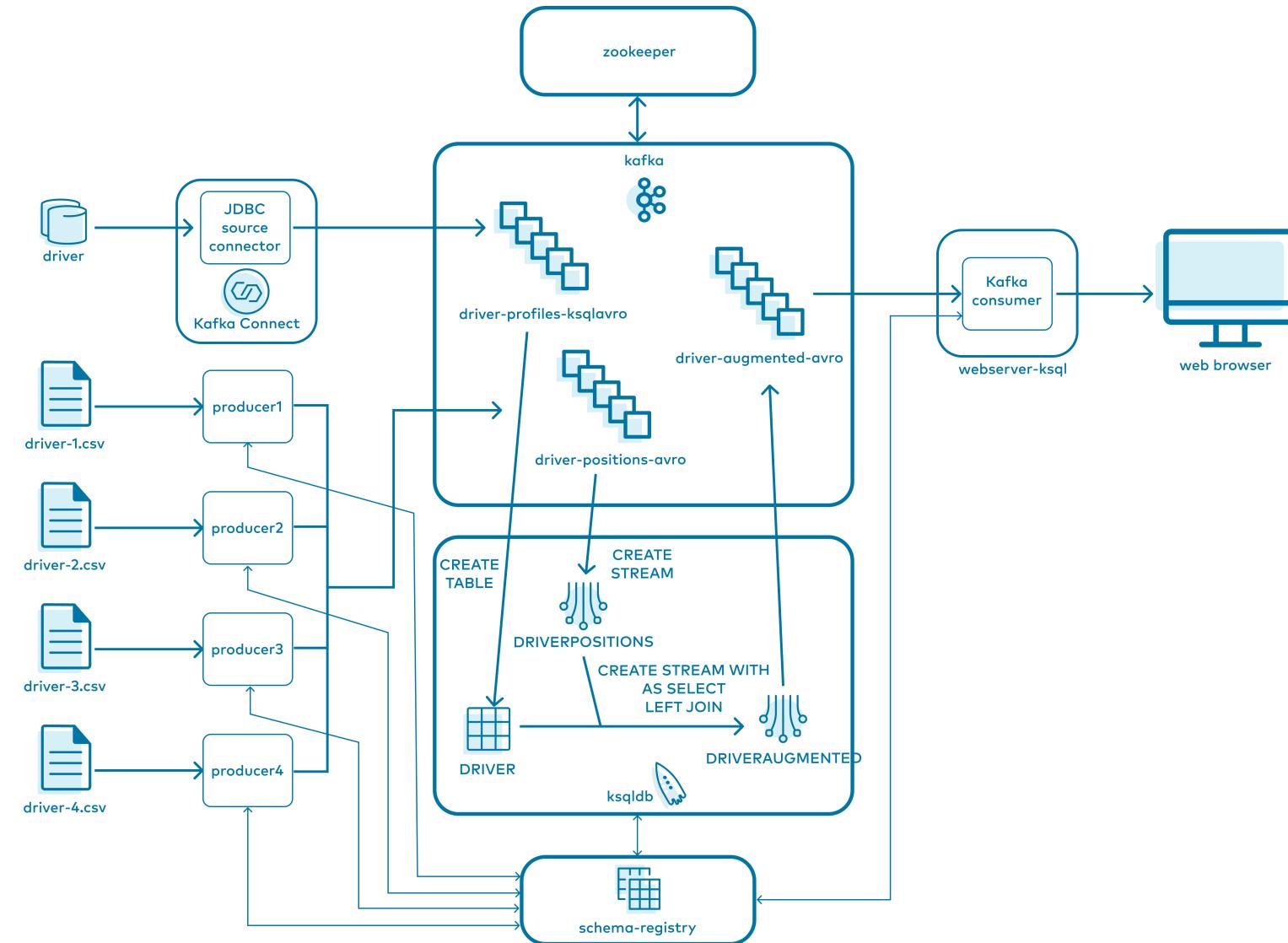
```
CREATE TABLE pageviews_count_by_region AS
  SELECT gender_id, region_id, COUNT(*) AS total
  FROM pageviews_enriched
  WINDOW TUMBLING (SIZE 30 SECONDS)
  GROUP BY gender_id, region_id
  HAVING COUNT(*) > 1
EMIT CHANGES;
```

Hands-On Lab

- In this Hands-On Exercise, you will use ksqlDB to create an entire event streaming application with only a few KSQL statements.
- Please refer to **Lab 08 Event Streaming Apps with ksqlDB** in the Exercise Book:
 - a. **ksqlDB - Join a Stream and a Table**



ksqldb - Join a Stream and a Table



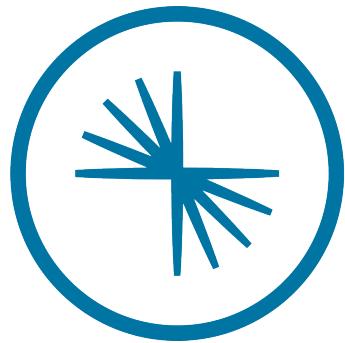
Module Review



Questions:

How does ksqlDB simplify the architecture for creating event streaming applications?

09 Design Decisions



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions ... ←
10. Confluent Cloud
11. Conclusion
12. Appendix: Basic Kafka Administration

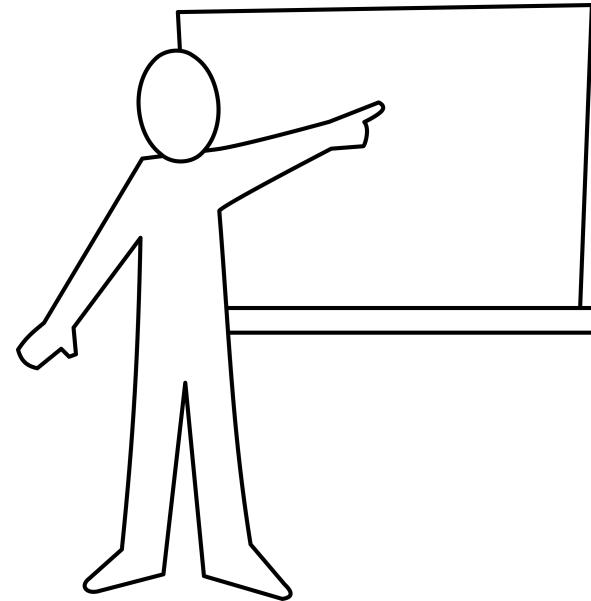
Learning Objectives



After this module you will be able to:

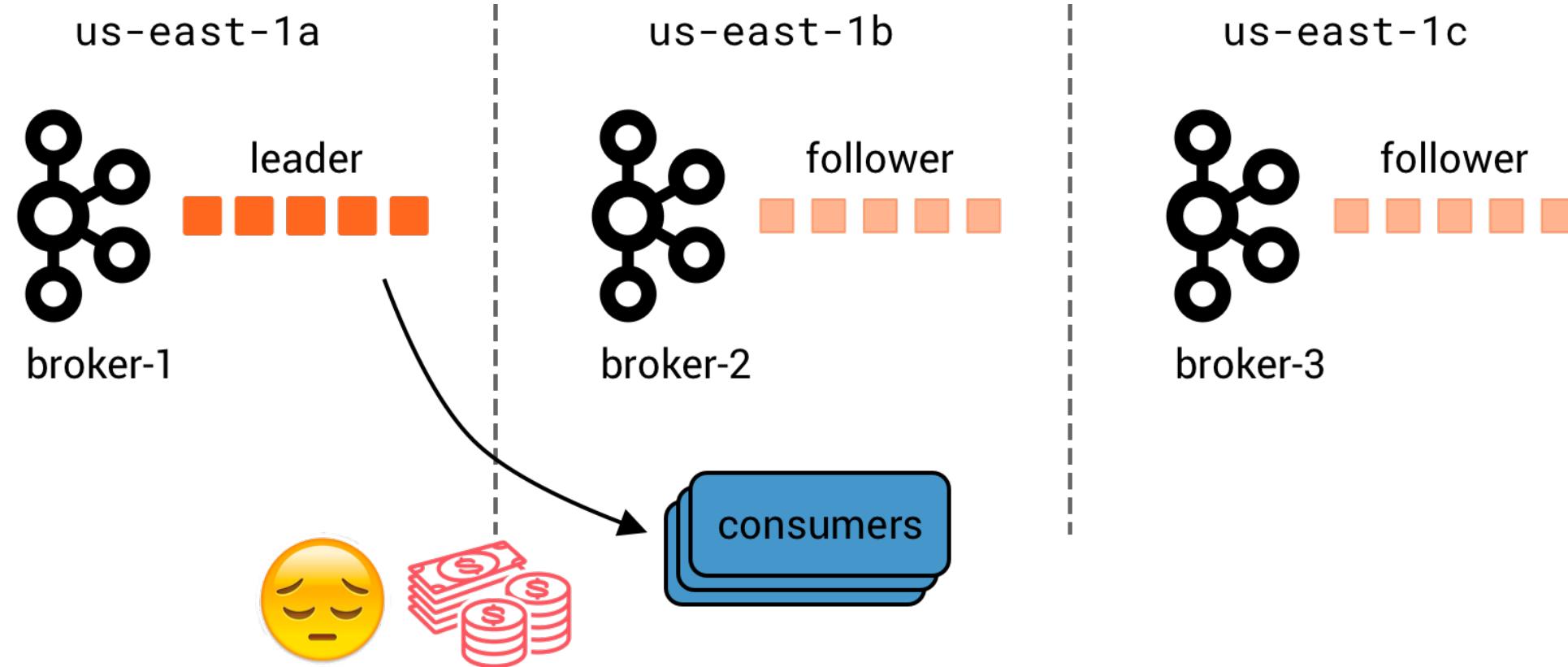
- list ways to avoid large message sizes
- decide when to use ksqlDB vs. Kafka Streams vs. Kafka Connect SMTs
- explain differences and tradeoffs between processing guarantees
- address decisions that arise from key-based partitioning
- authenticate a client app with a secure Kafka cluster

Module Map

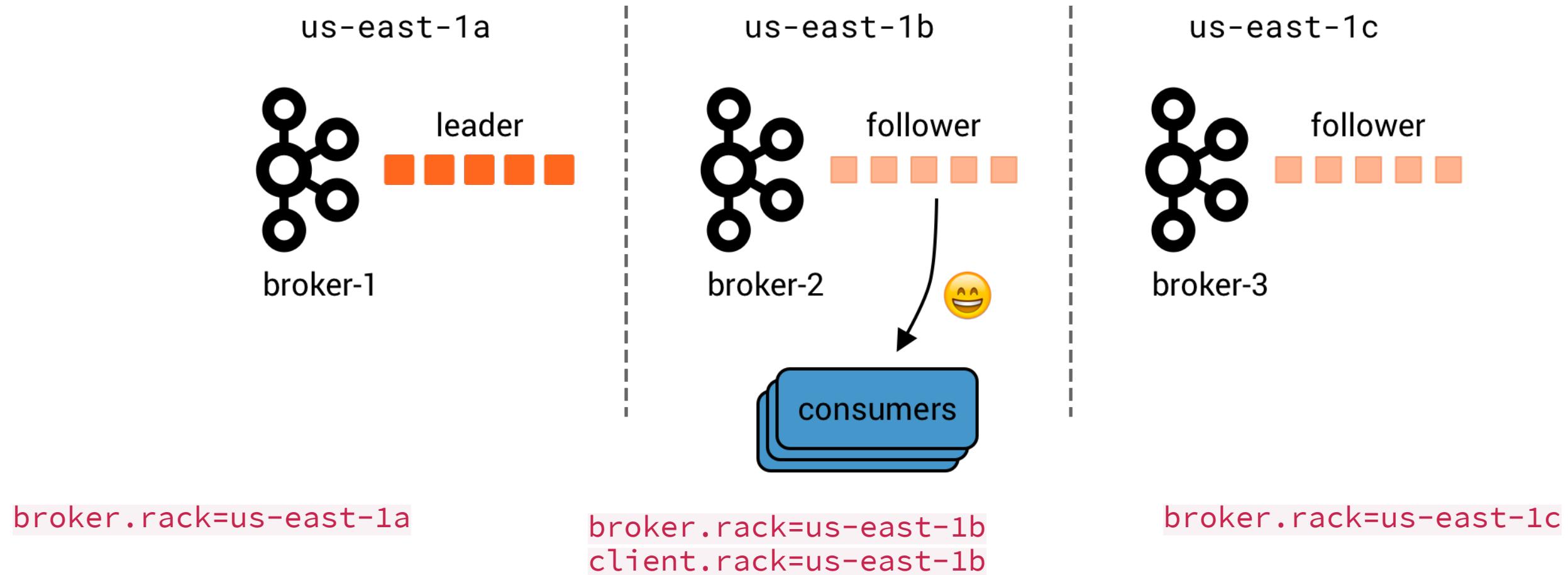


- "What could possibly go wrong?" ... ←
- Exactly Once Semantics (EOS)
- Decisions Around Keys and Partitions
- ksqlDB, Kafka Streams, or Kafka Connect SMTs?
- Authenticating to a secure cluster
- 🔐 Hands-on Lab

Problem: Cross-Zone Consumer Costs

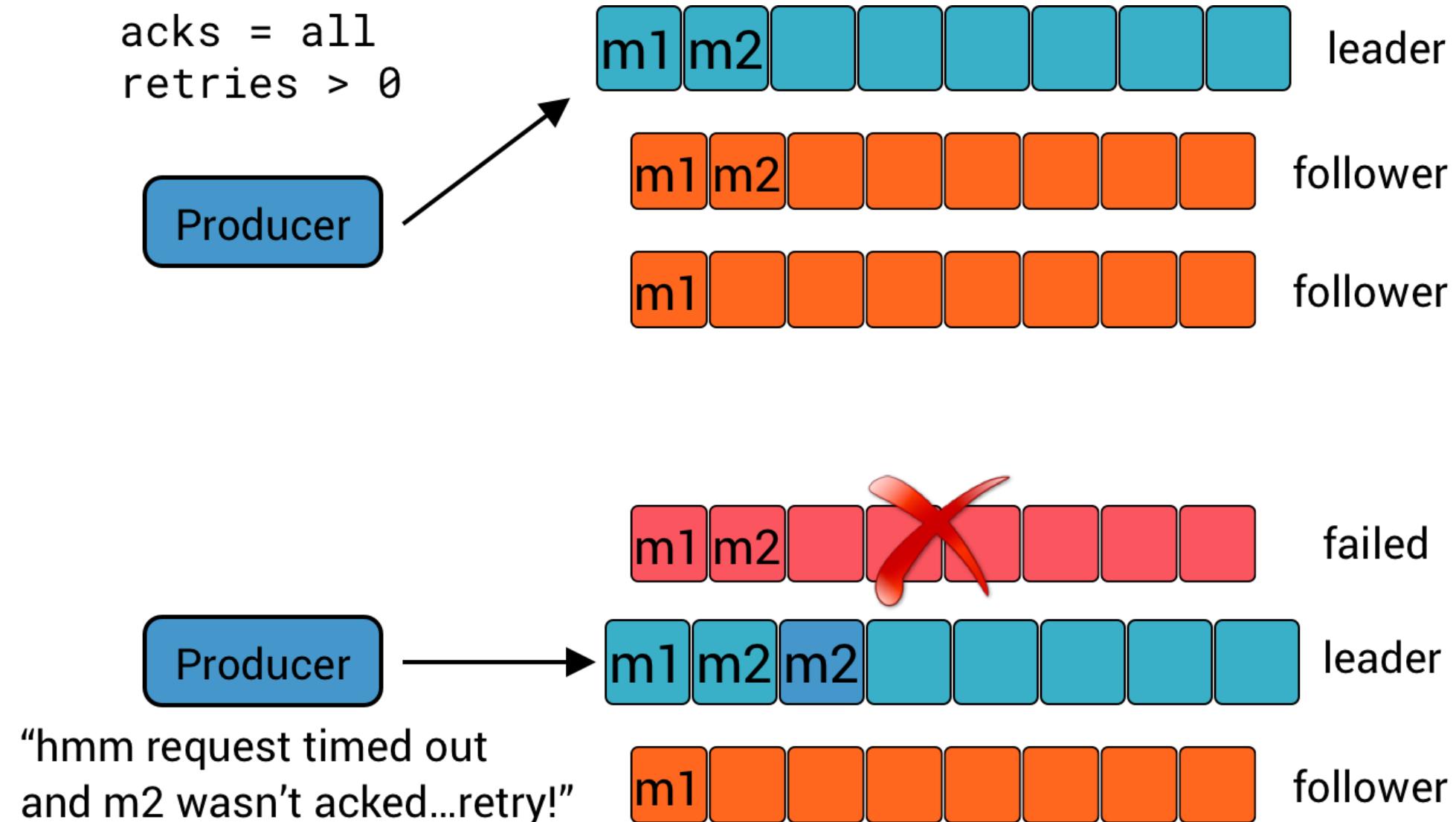


Solution: Follower Fetching



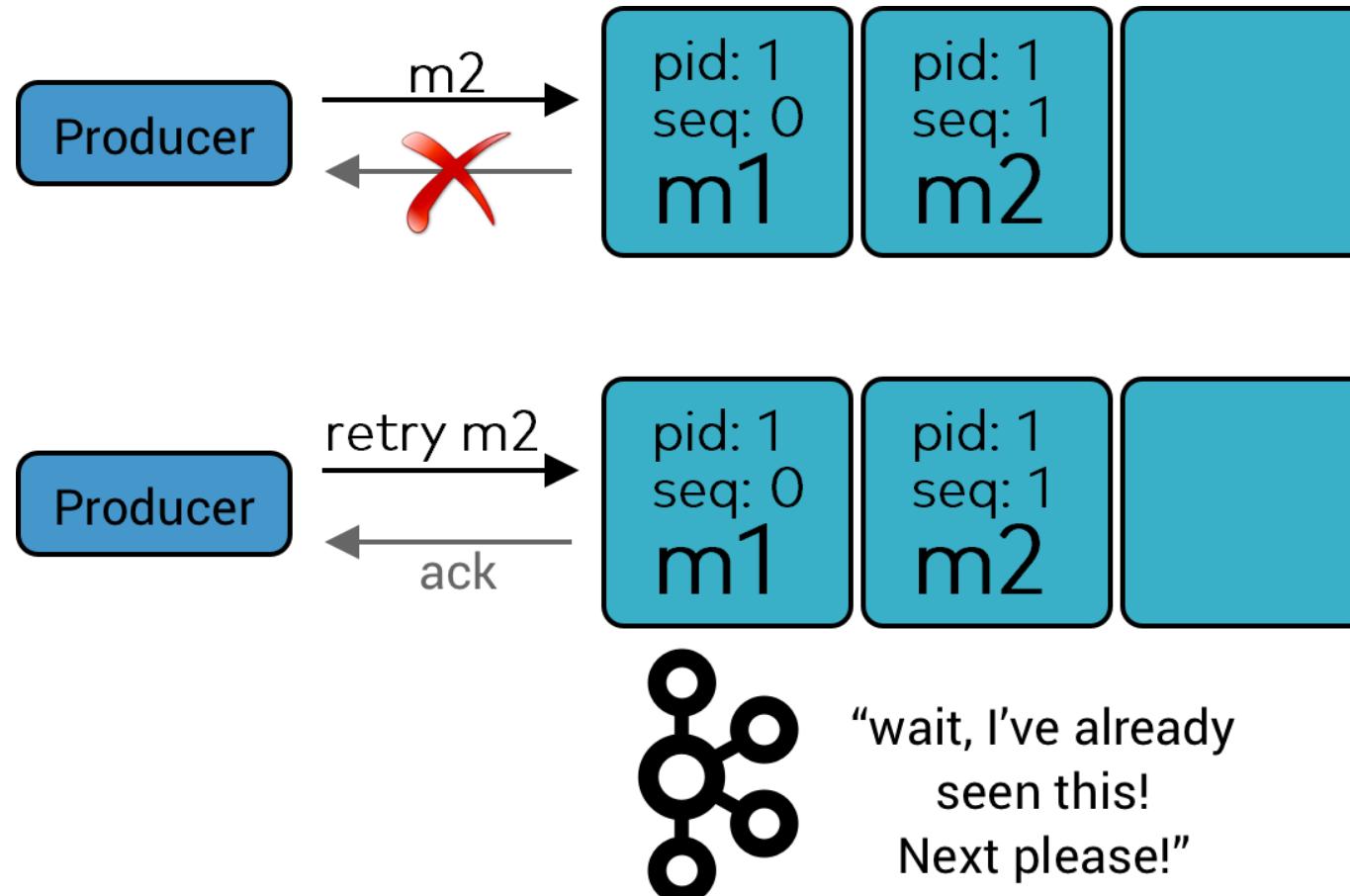
All brokers: `replica.selector.class=org.apache.kafka.common.replica.RackAwareReplicaSelector`

Problem: Producing Duplicates to the Log



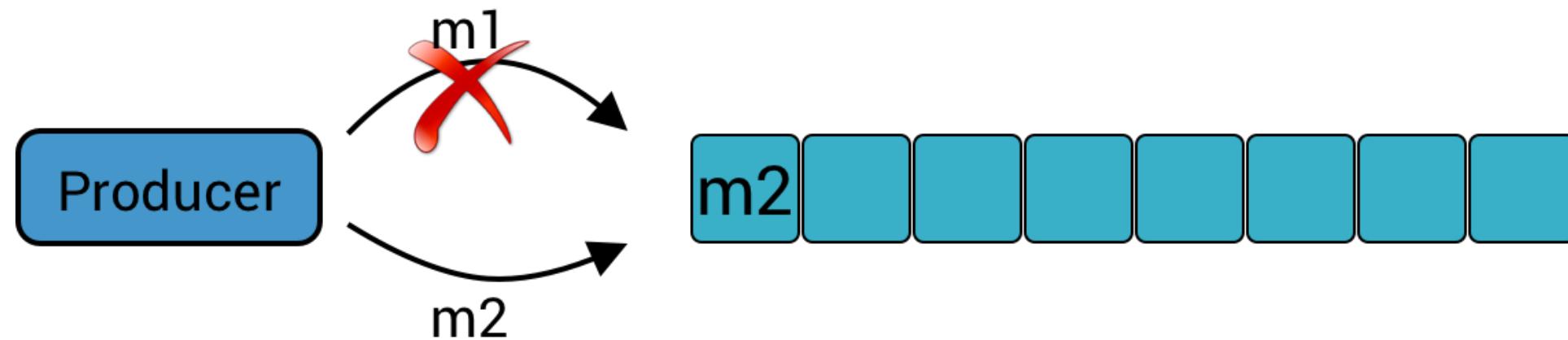
Solution: Idempotent Producers

- `enable.idempotence=true`
- `acks=all`

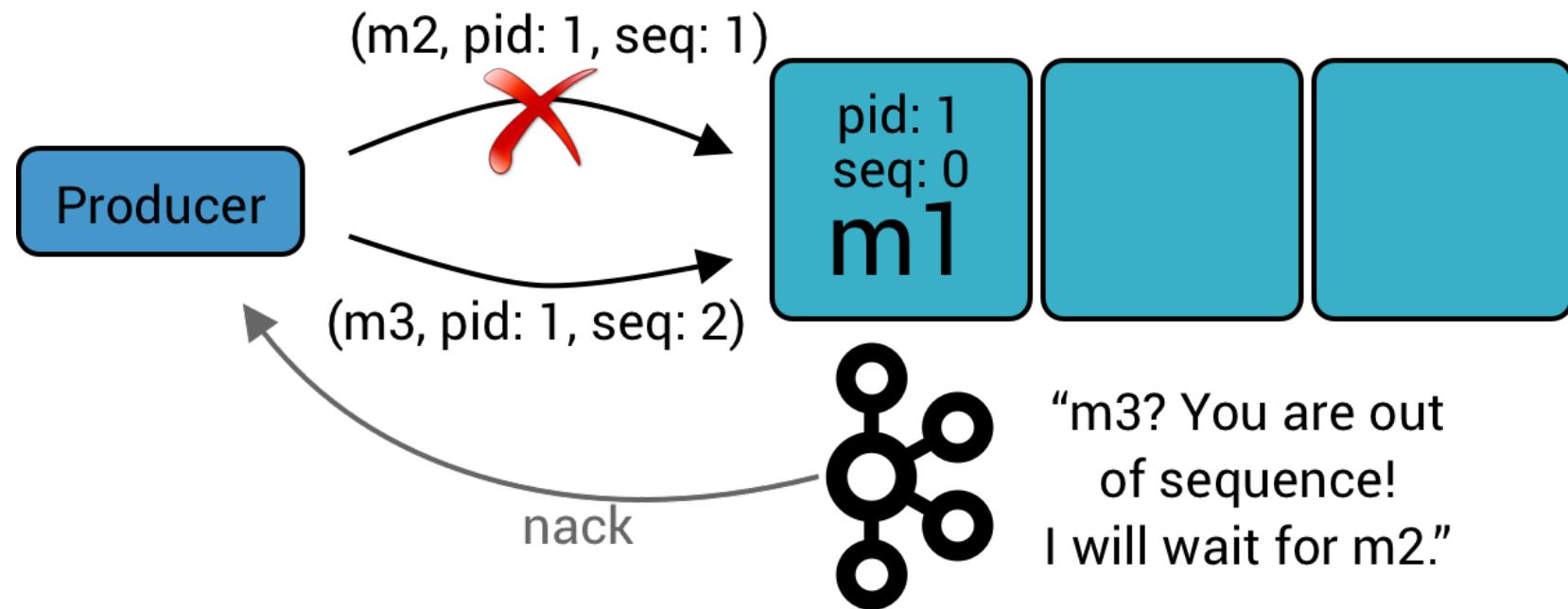


Problem: Producing Messages Out of Order

- `max.in.flight.requests.per.connection > 1`
- `retries > 0`



Solution: Idempotent Producers



Problem: Reprocessing Records is Required

- There are infrequent situations where a Consumer application must begin processing at an offset other than the default
 - The application was updated and records need to be reprocessed
 - A new release of an application had an error resulting in records being incorrectly processed
 - The previous release of the application must be restored
 - Already processed records must be reprocessed correctly

Determining the Offset When a Consumer Starts

- If a valid offset exists in `_consumer_offsets`, the consumer begins consumption from that point
- If no valid offset exists, `auto.offset.reset` determines how the consumer proceeds

3 Scenarios	Values for <code>auto.offset.reset</code>	
Consumer Group starts first time	<code>earliest</code>	Reset offset to earliest available
Consumer offset < smallest offset	<code>latest</code>	Reset offset to latest available
Consumer offset > last offset + 1	<code>none</code>	Throw exception if no previous offset found for Consumer Group

Solution: Reset the Current Offset (1)

- Getting Offsets

<code>position(TopicPartition)</code>	Offset of next record that will be fetched
<code>offsetsForTimes(Map<TopicPartition, Long> timestampsToSearch)</code>	Offsets for given Partitions by timestamp

- Changing Offsets

<code>seek(TopicPartition, offset)</code>	Seek to specific offset in specified Partition
<code>seekToBeginning(Collection<TopicPartition>)</code>	Seek to first offset of each specified Partition
<code>seekToEnd(Collection<TopicPartition>)</code>	Seek to last offset of each specified Partition

Solution: Reset the Current Offset (2)

- Example: Reset offset to **particular timestamp**:

```
1 for (TopicPartition partition : partitions) {  
2     timestampsToSearch.put(partition, MY_TIMESTAMP);  
3 }  
4  
5 Map<TopicPartition, OffsetAndTimestamp> result = consumer.offsetsForTimes(timestampsToSearch);  
6  
7 for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : result.entrySet()) {  
8     consumer.seek(entry.getKey(), entry.getValue().offset());  
9 }
```

- Add each Partition and timestamp to HashMap
- Get offset for each Partition
- Seek to specified offset for each Partition
- Question: what is a use case where this would be useful?

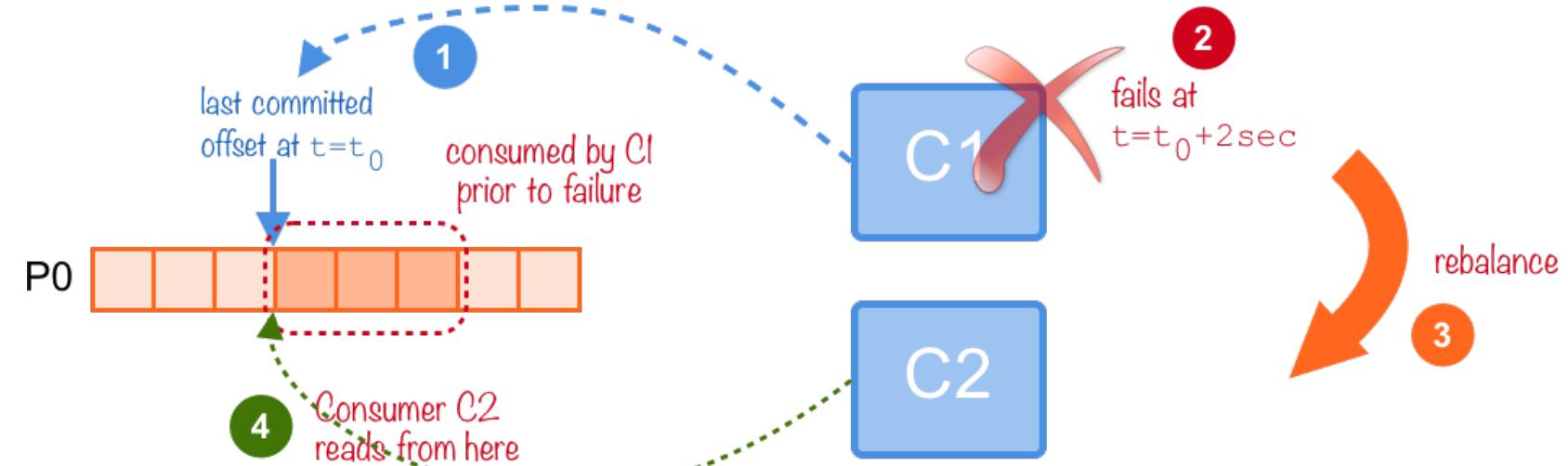
Solution: Reset the Current Offset (3)

- Example: **Seek to beginning** of all partitions for topic `my_topic`:

```
1 ConsumerRebalanceListener listener = new
2     ConsumerRebalanceListener() {
3         @Override
4         public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
5             // nothing to do...
6         }
7
8         @Override
9         public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
10            consumer.seekToBeginning(partitions);
11        }
12    };
13 consumer.subscribe(Arrays.asList("my_topic"), listener);
14 consumer.poll(Duration.ofMillis(0));
```

- Subscribe to topics & register `ConsumerRebalanceListener`
- `poll(Duration.ofMillis(0))` retrieves metadata from broker

Problem: Consumer Reprocessing after Failure



Solution 1: Handle Data and Offsets Atomically

pseudocode

```
SELECT partition, offset FROM database

consumer.seek(partition, offset)

while true:
    purchases = consumer.poll()
    for purchase in purchases:
        BEGIN TRANSACTION
        INSERT purchase.id, purchase.amount INTO database
        INSERT purchase.partition, purchase.offset INTO database
        COMMIT TRANSACTION
```

Solution 2: Use an Idempotent Sink

pseudocode

```
while true:  
    purchases = consumer.poll()  
    for purchase in purchases:  
        INSERT purchase.id, purchase.amount INTO database  
        ON DUPLICATE KEY UPDATE  
    consumer.commitAsync()
```

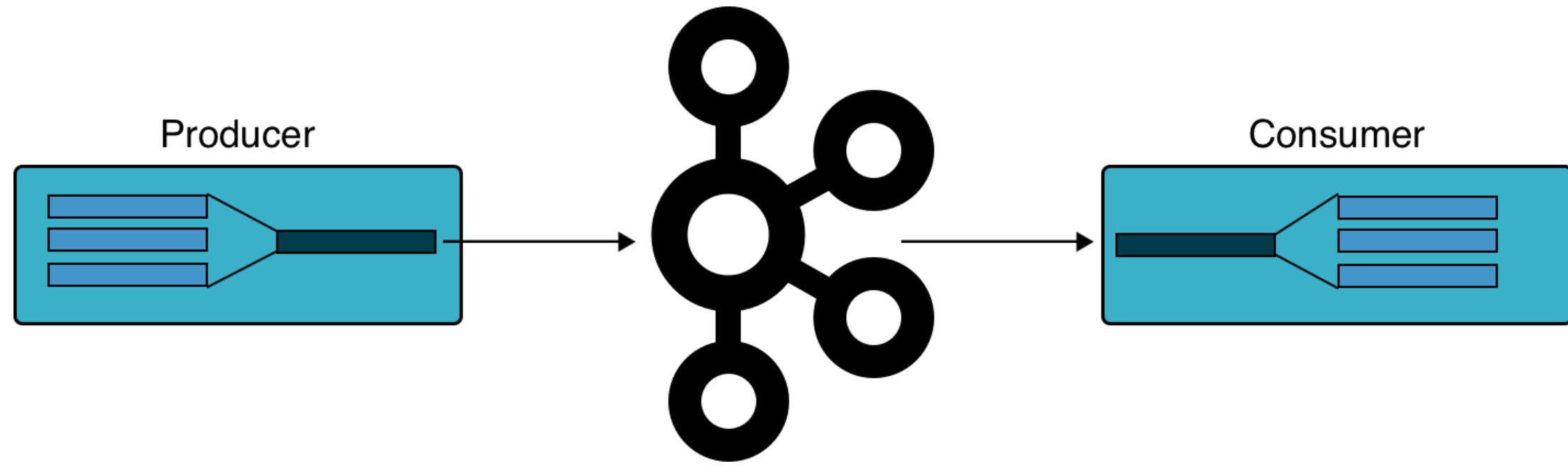
Examples:

- JDBC Sink Connector
- HDFS Sink Connector
- S3 Sink Connector
- Elasticsearch Sink Connector

Problem: Large Messages

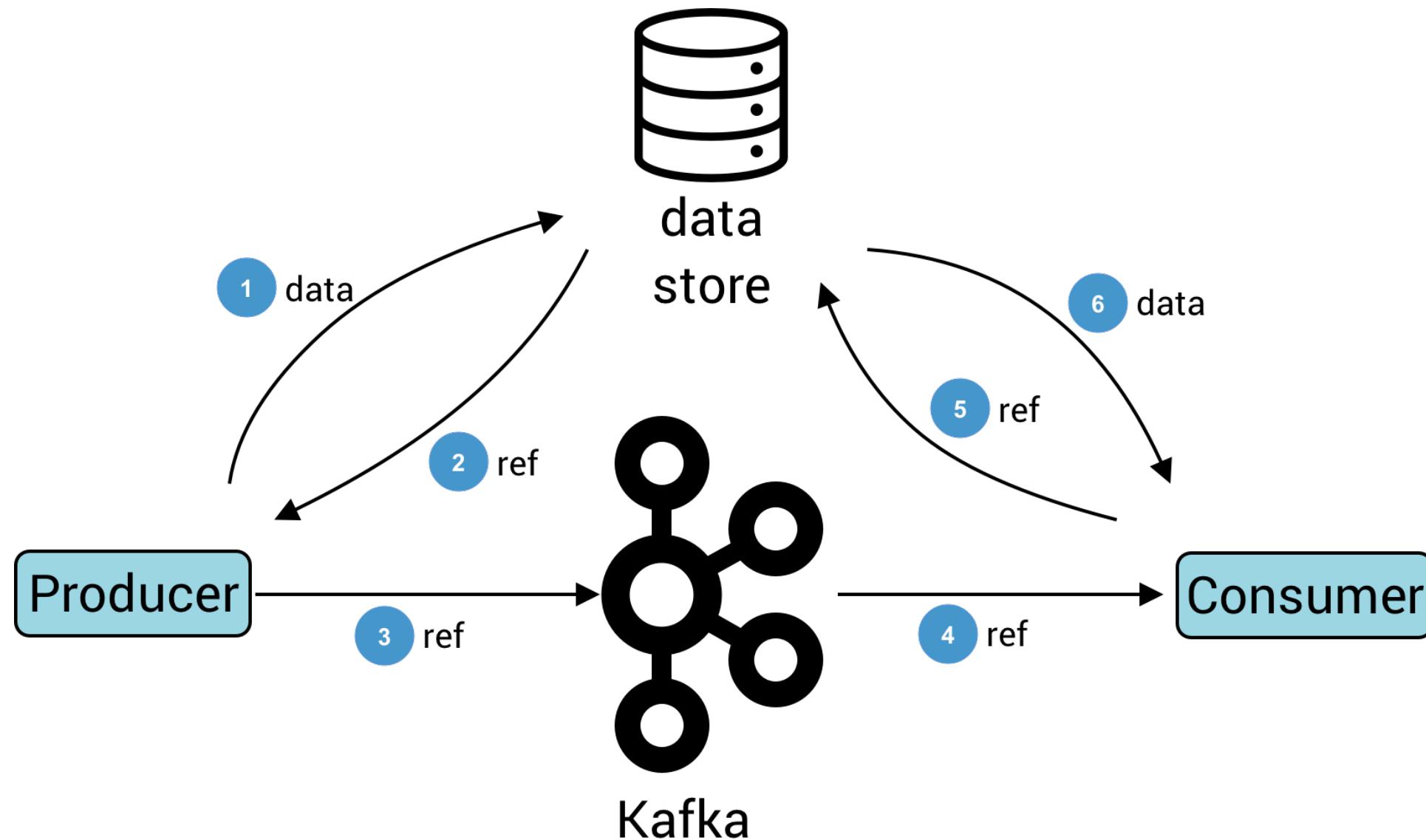
- Broker default for `message.max.bytes` is 1 MB
- Producer default for `max.request.size` is also 1 MB
- Increasing message size limit can lead to:
 - poor Garbage Collection performance
 - less memory available for other important Broker business
 - more resources needed to handle requests

Solution 1: Compression

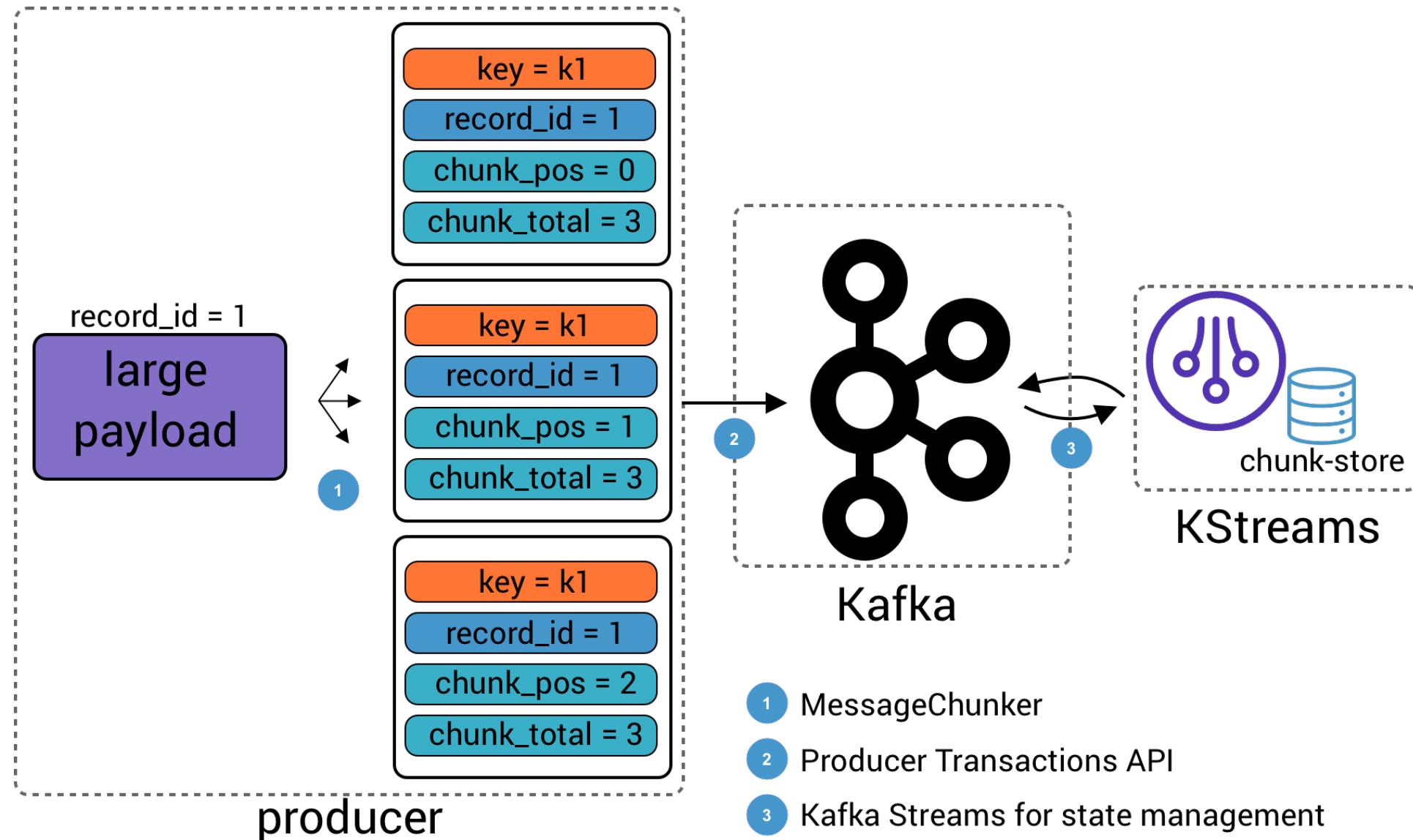


1. Producer batches and then compresses the record batch
2. Compressed record batch stored in Kafka
3. Consumer decompresses

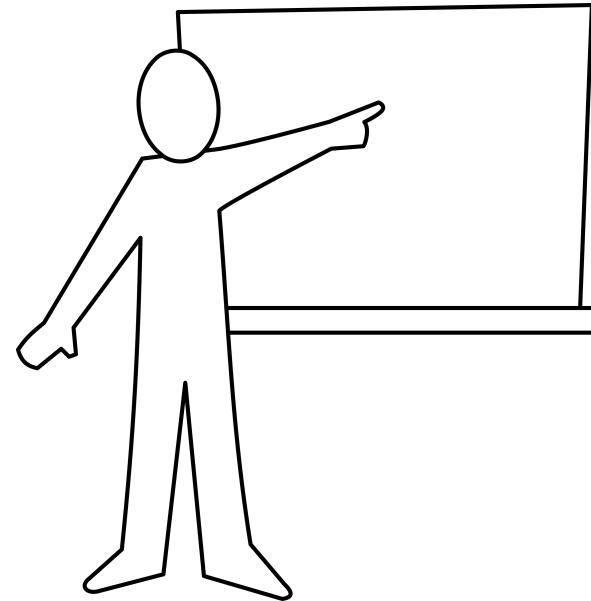
Solution 2: Pass Reference to External Store



Solution 3: Record Chunking



Module Map

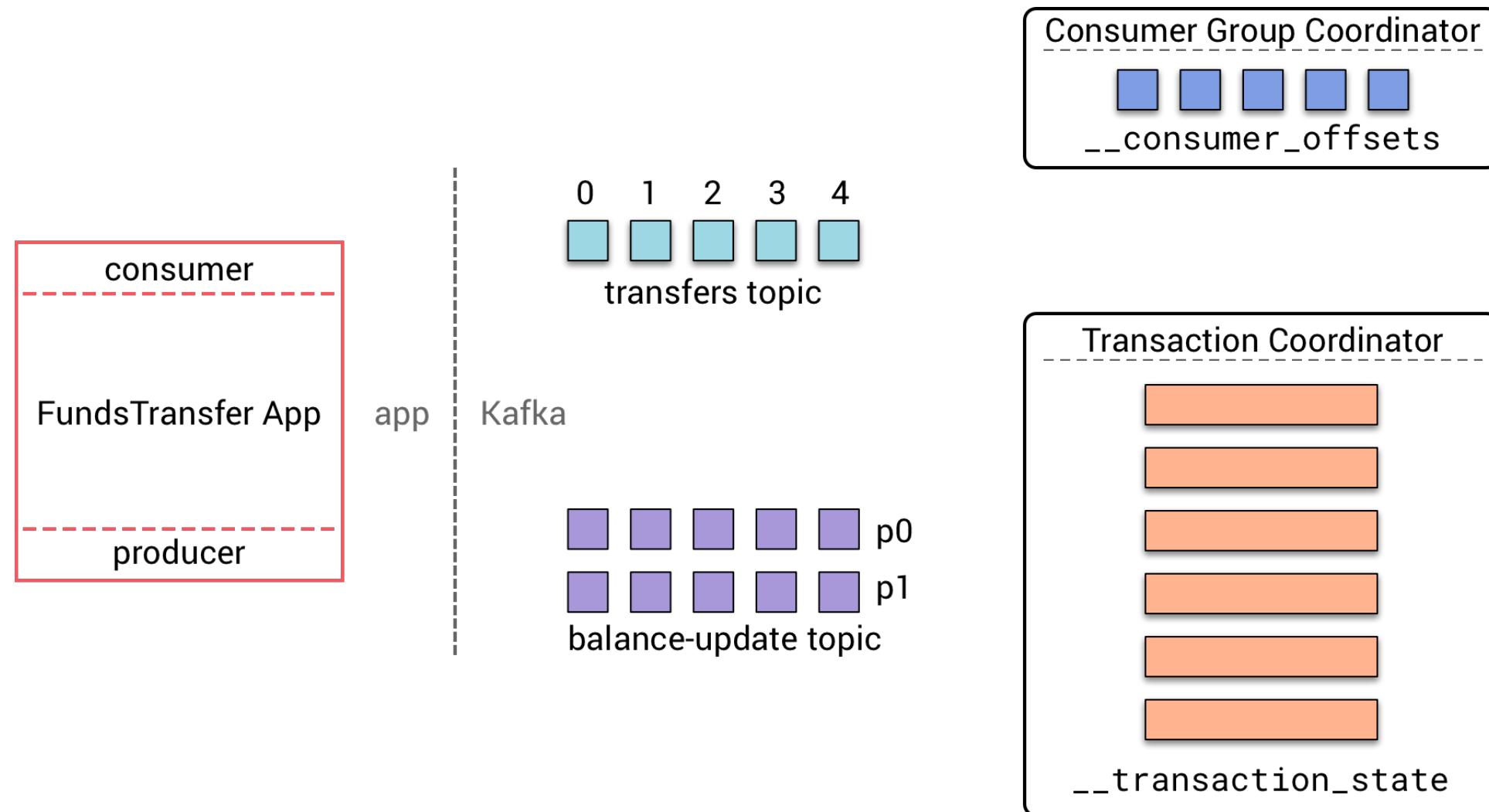


- "What could possibly go wrong?"
- Exactly Once Semantics (EOS) ... ←
- Decisions Around Keys and Partitions
- ksqlDB, Kafka Streams, or Kafka Connect SMTs?
- Authenticating to a secure cluster
- 🔐 Hands-on Lab

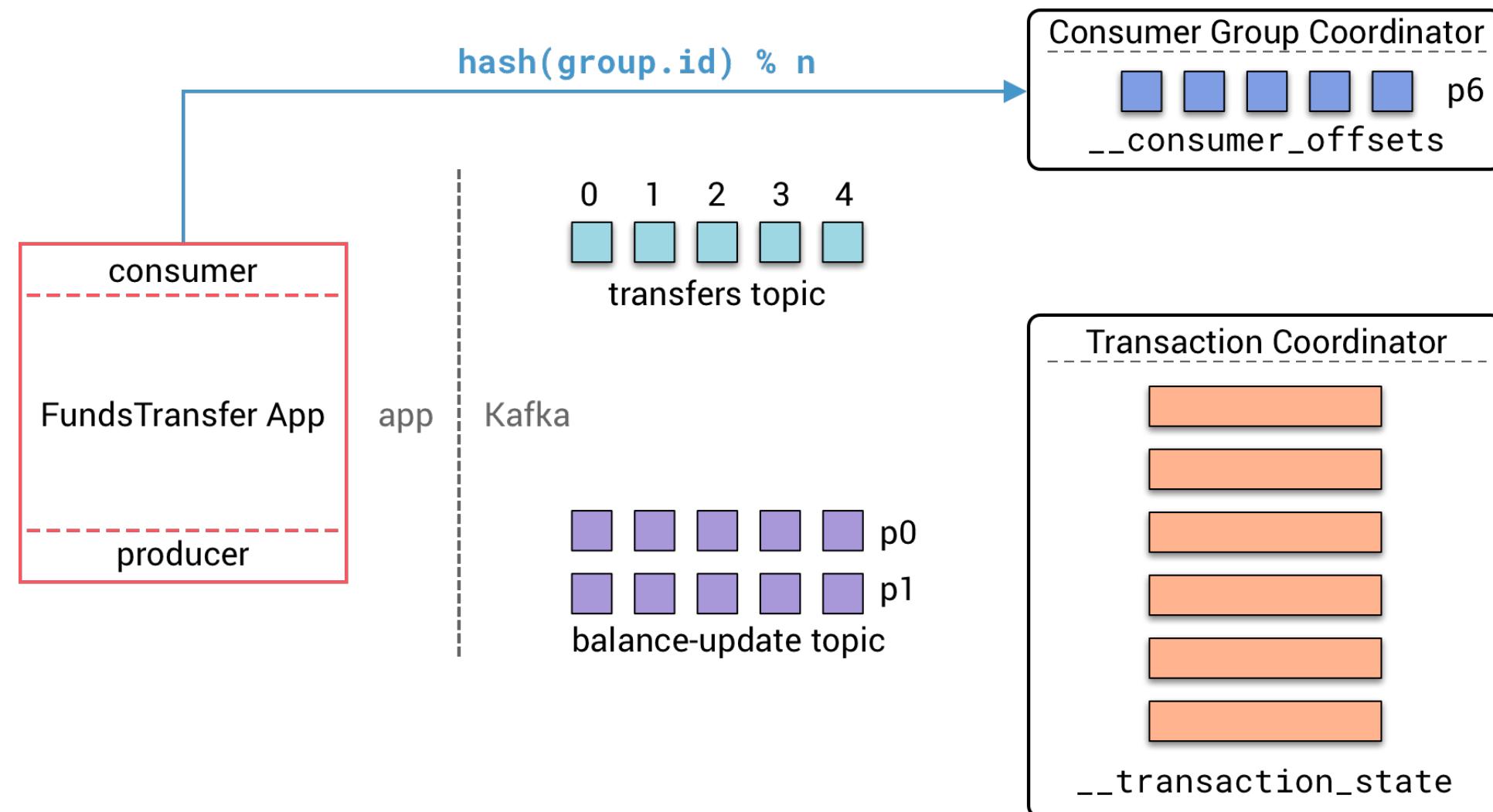
Transactions with Exactly Once Semantics (EOS)

- Feature designed for consume-process-produce use cases, like **Kafka Streams**
- How it works:
 1. Idempotent Producers
 - Set `enable.idempotence = true` on the producer
 - This ensures Kafka's partitions are free of duplicate records
 2. Transactions
 - Set a unique `transactional.id` for the producer
 - Use the transactions API in the producer code
- Downstream consumers read committed transactional records with `isolation.level = read_committed`
- librdkafka now has complete Exactly-Once-Semantics (EOS) functionality, supporting the idempotent producer (since v1.0.0), a transaction-aware consumer (since v1.2.0) and full producer transaction support (since v5.5.0).

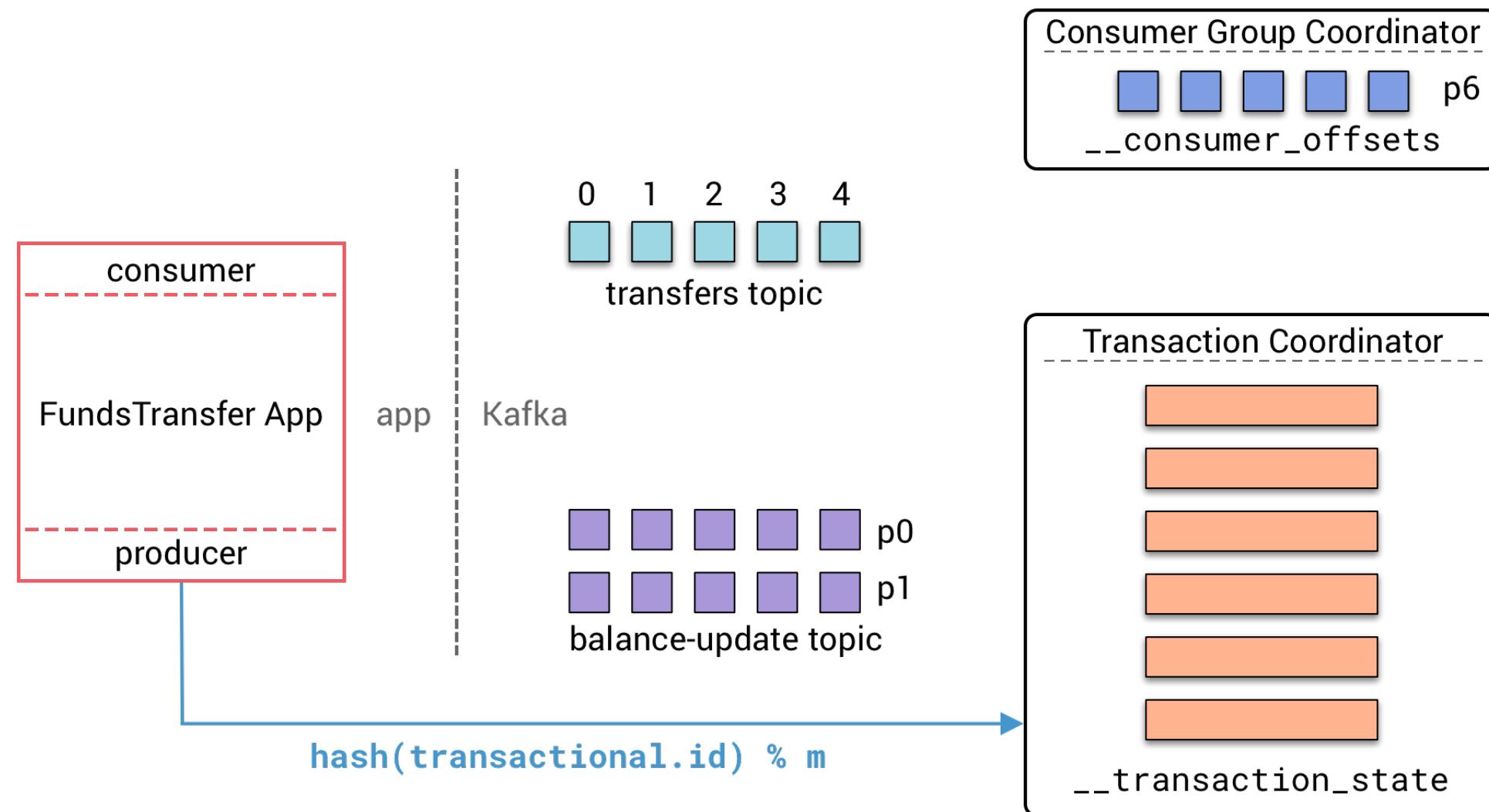
Transactions (1/14)



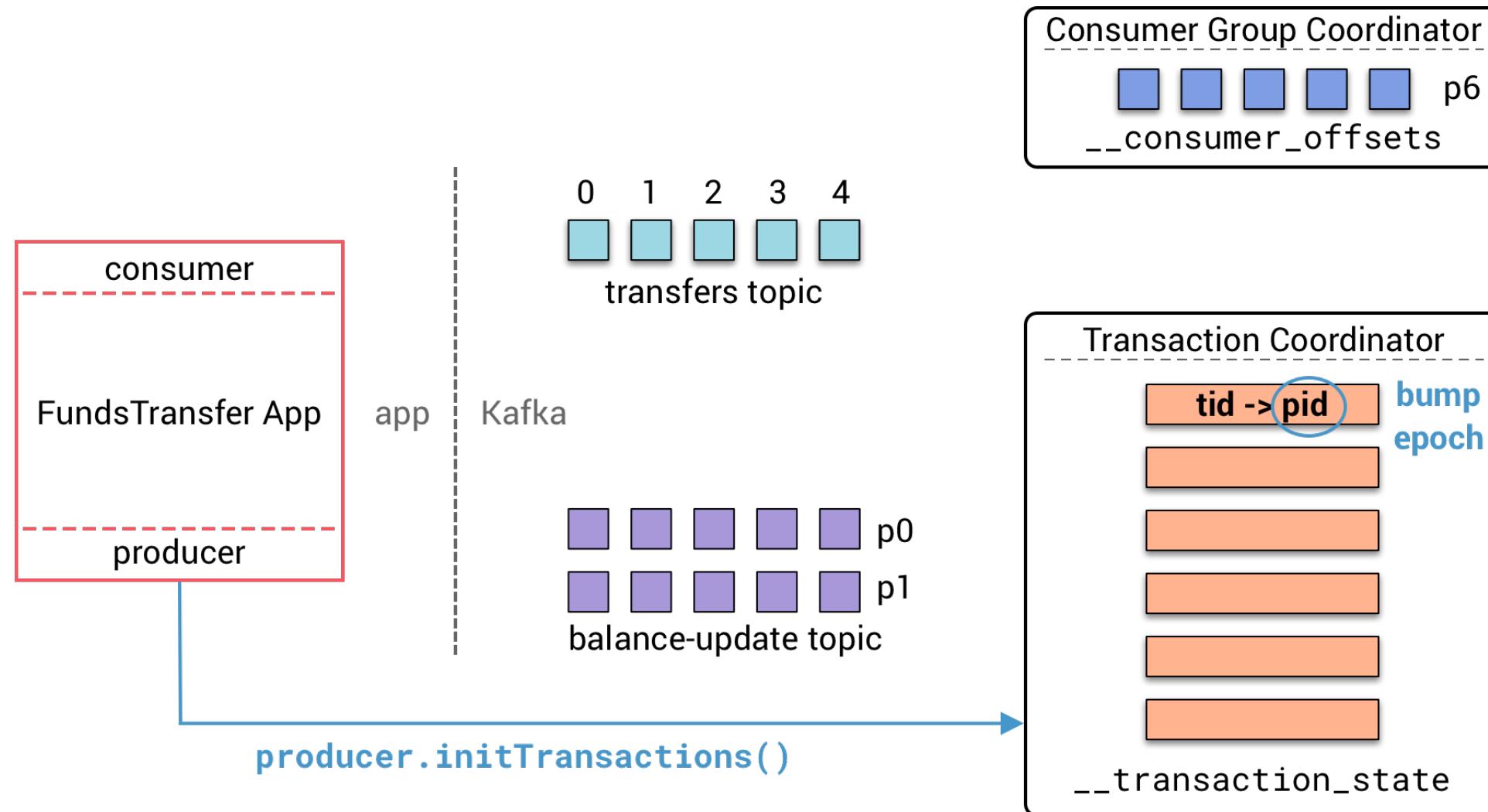
Transactions - Initialize Consumer Group (2/14)



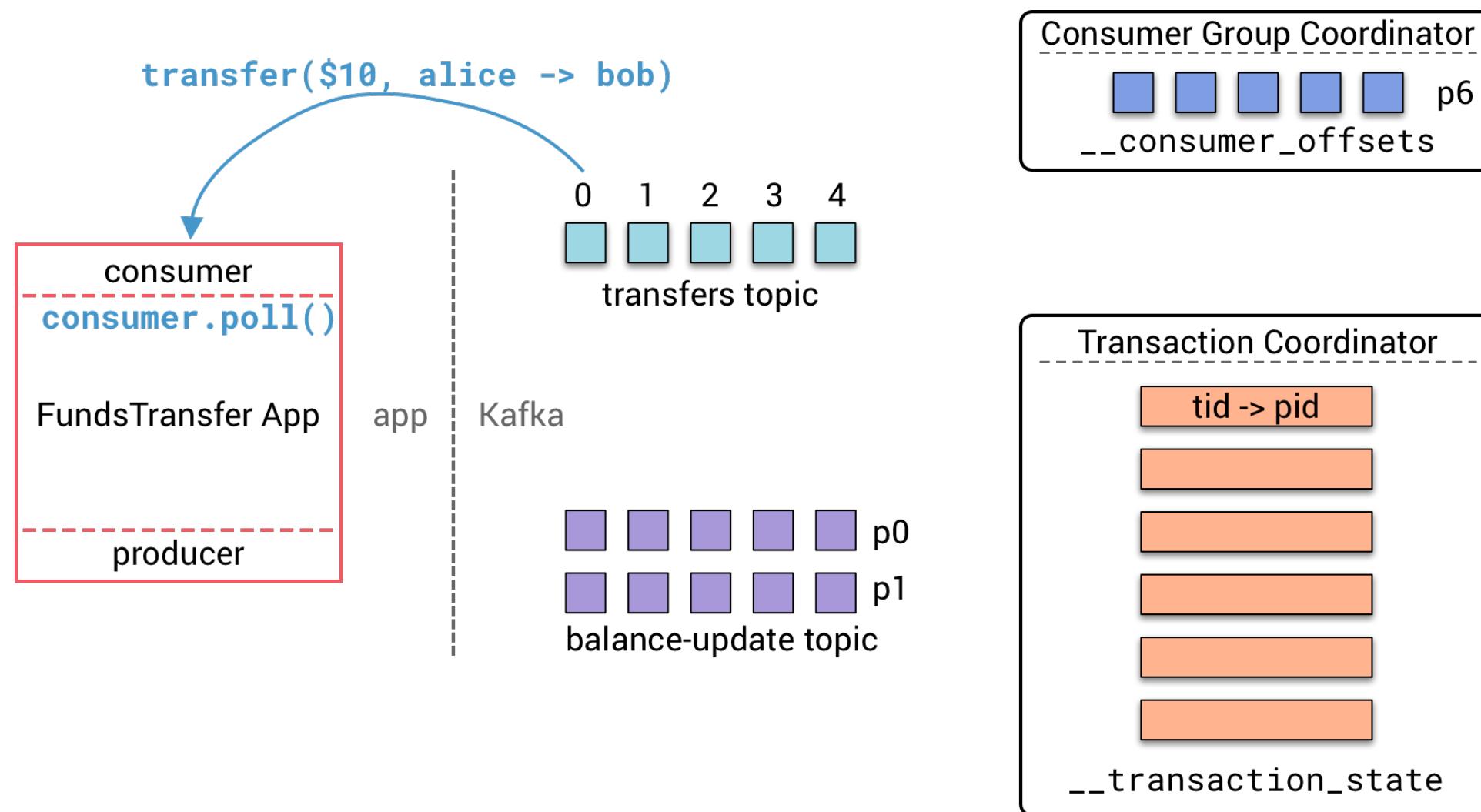
Transactions - Transaction Coordinator (3/14)



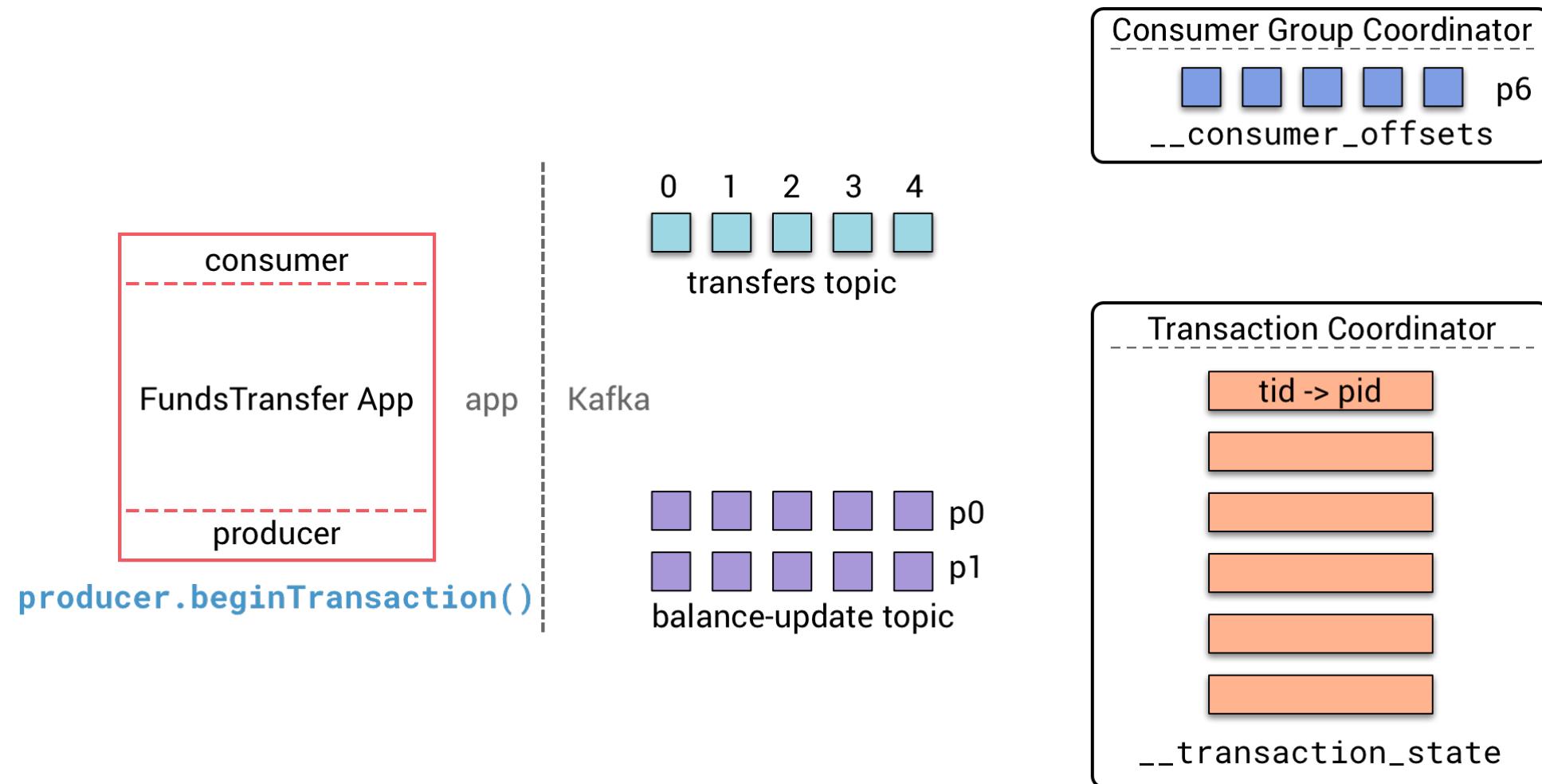
Transactions - Initialize (4/14)



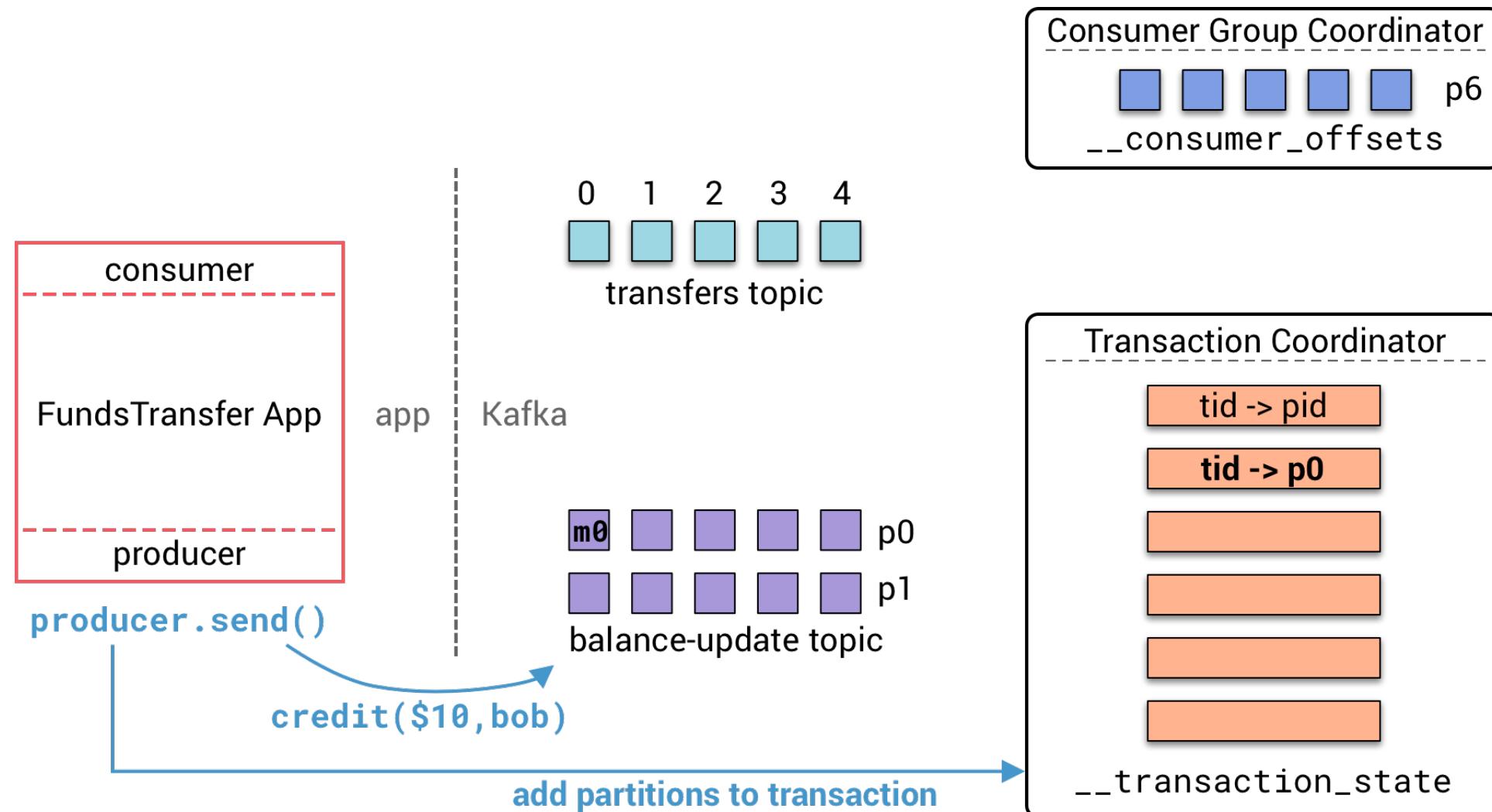
Transactions - Consume and Process (5/14)



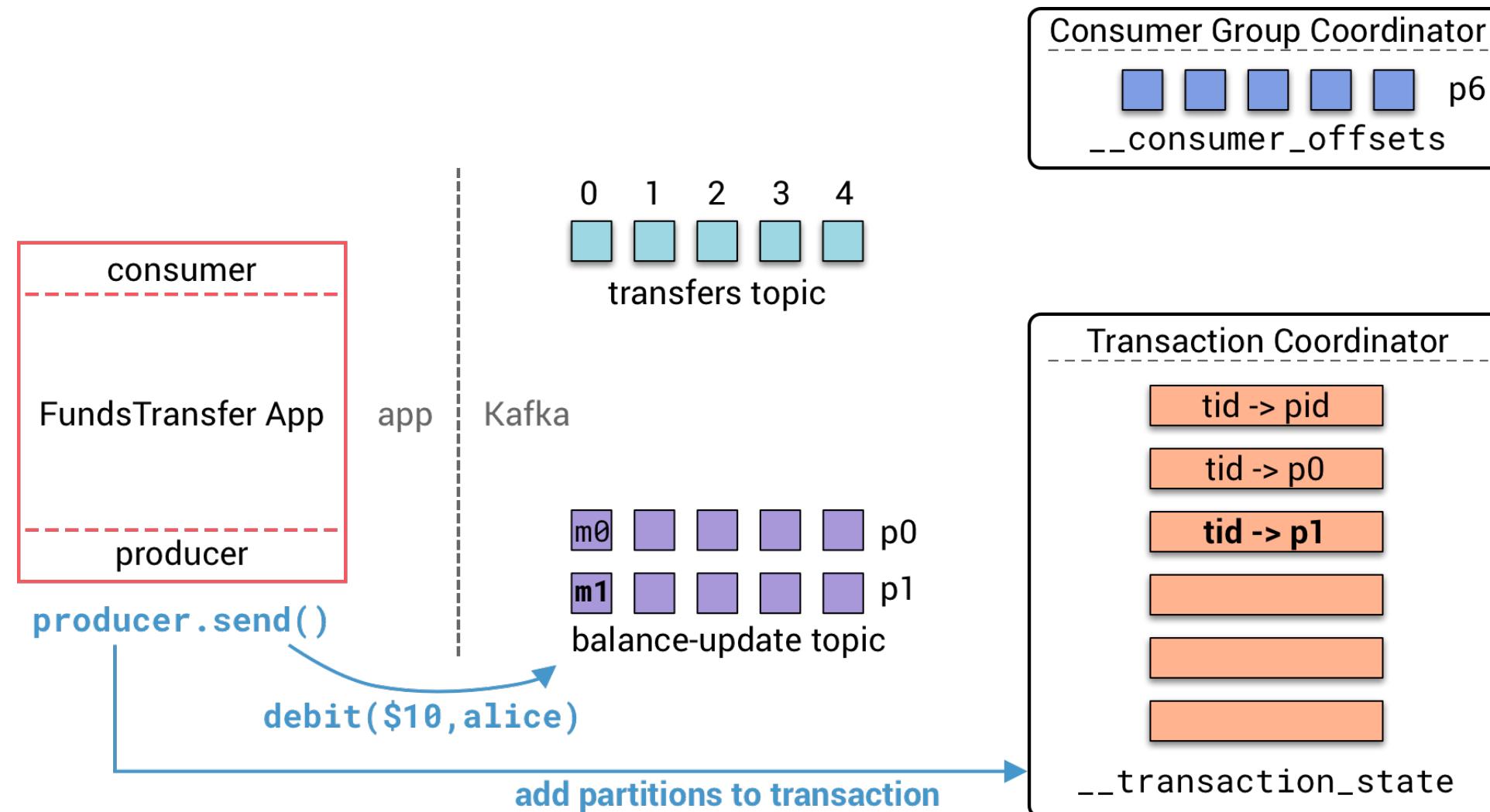
Transactions - Begin Transaction (6/14)



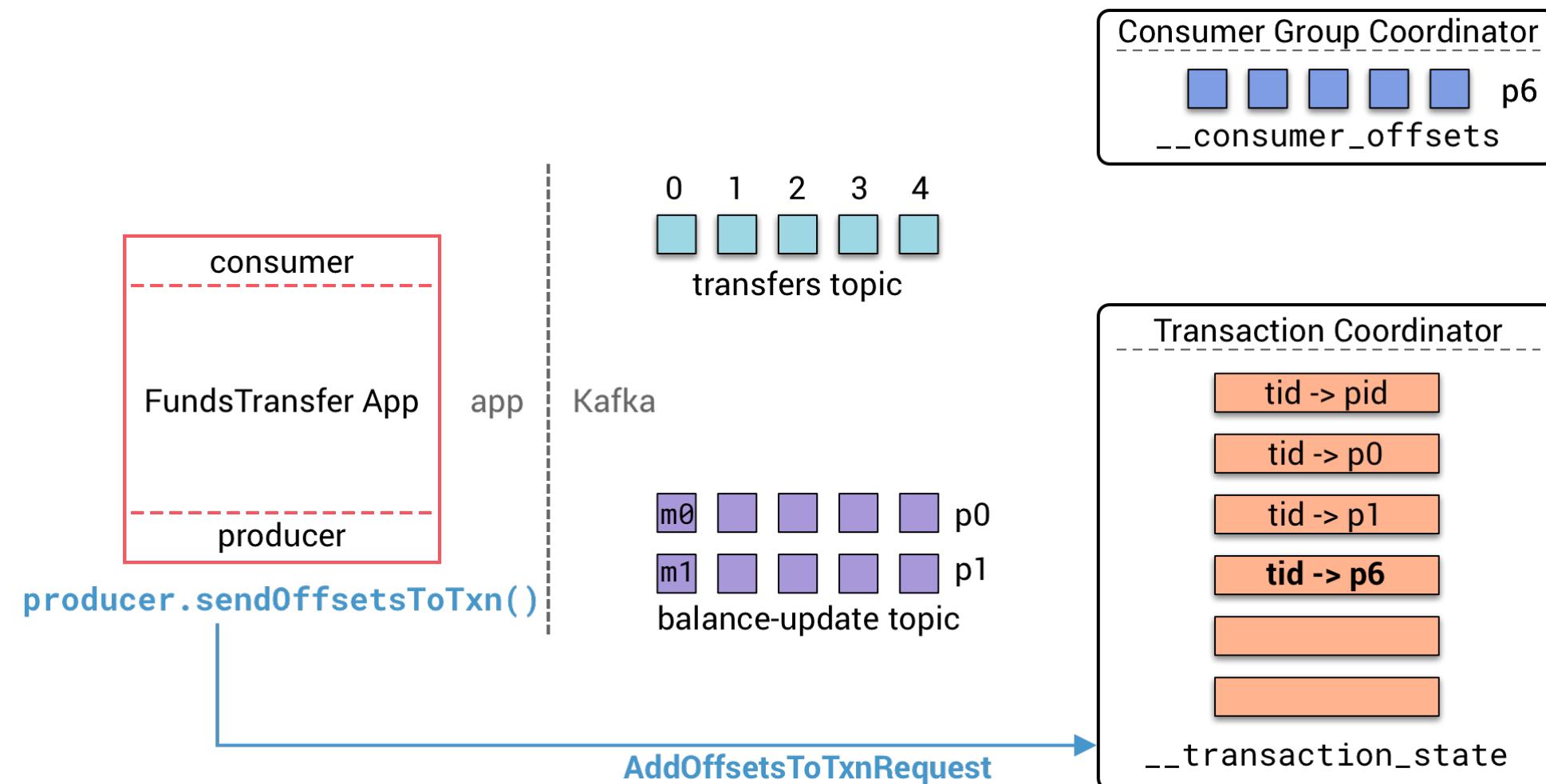
Transactions - Send (7/14)



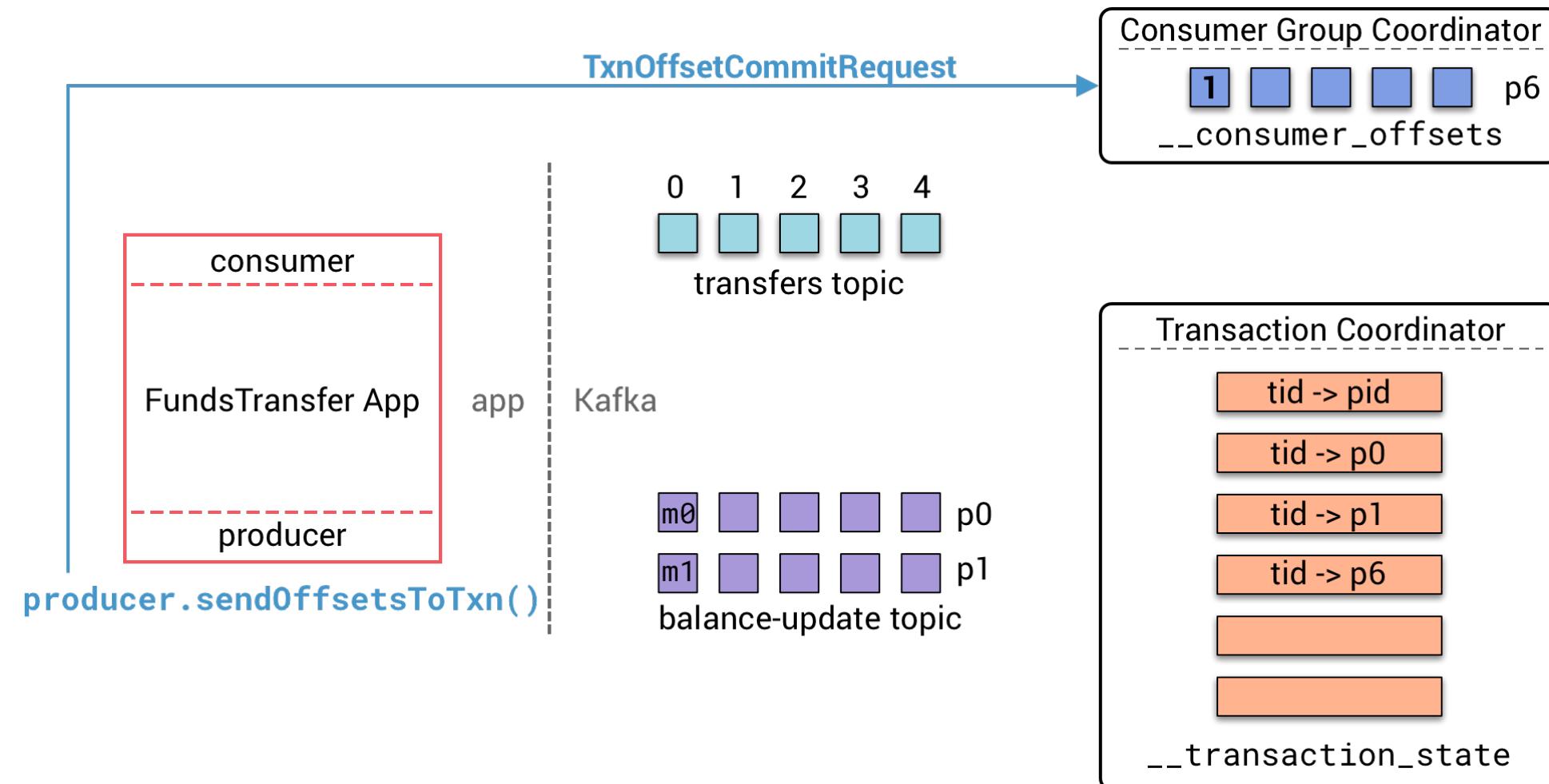
Transactions - Send (8/14)



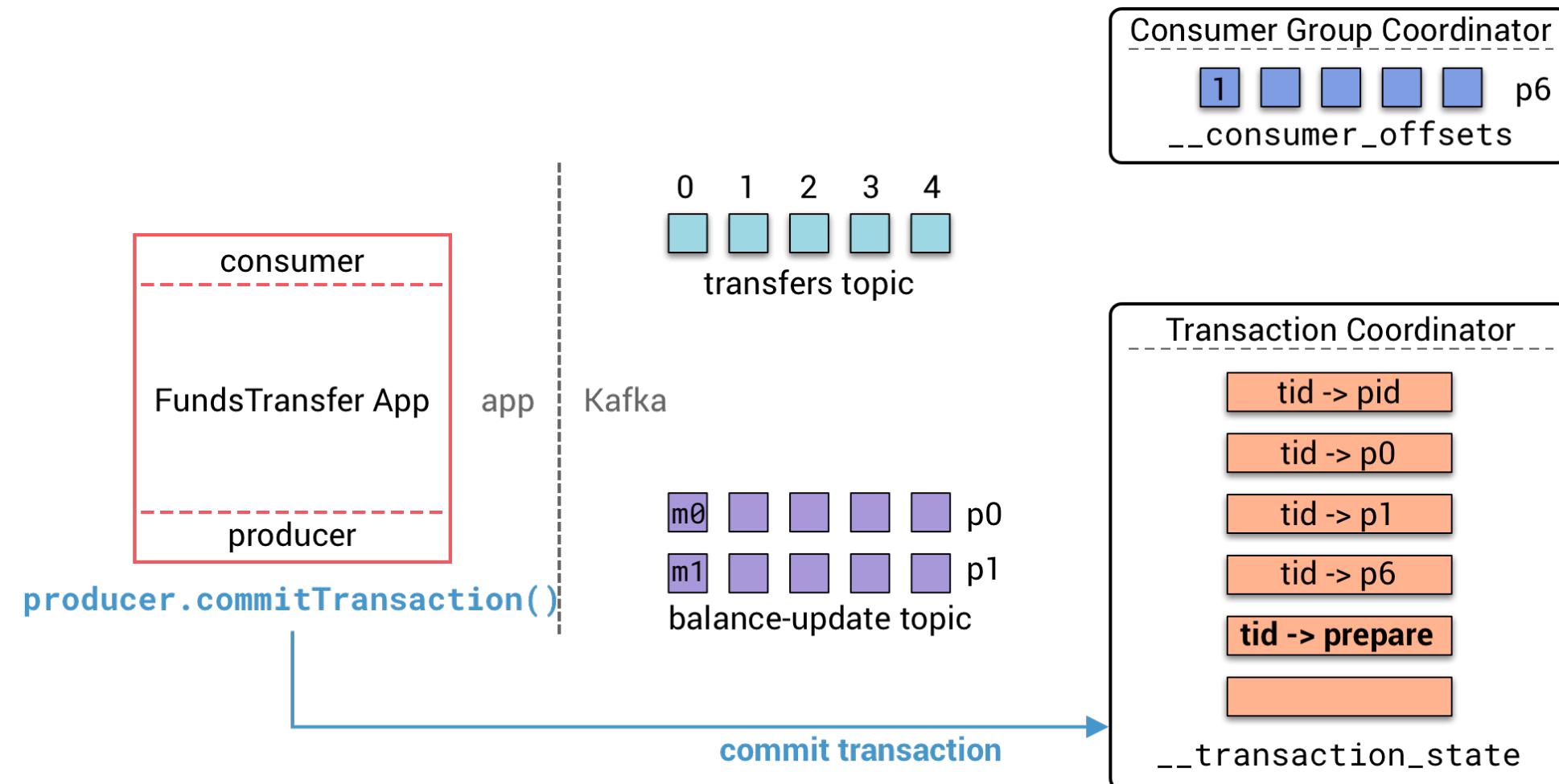
Transactions - Track Consumer Offset (9/14)



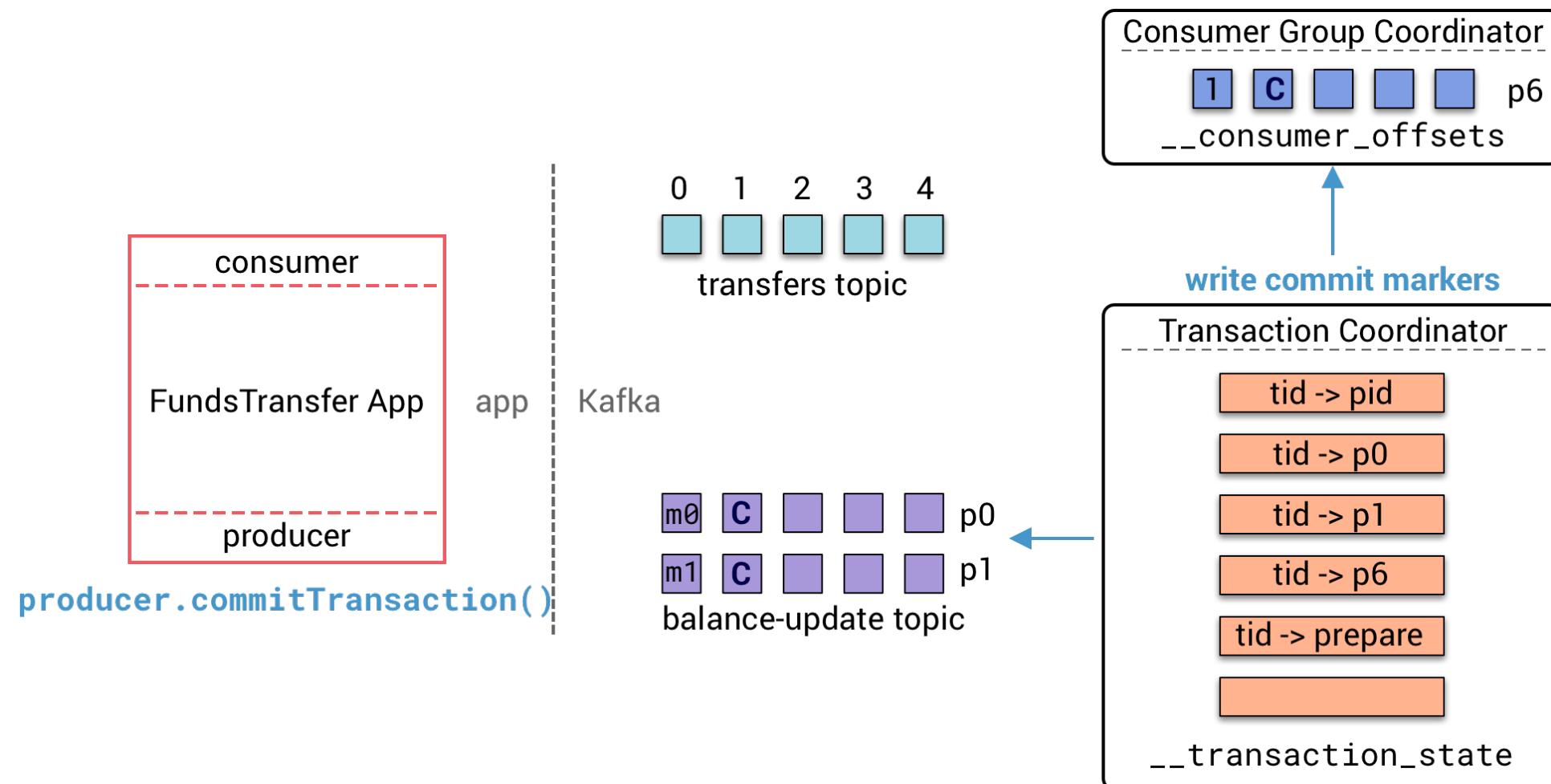
Transactions - Commit Consumer Offset (10/14)



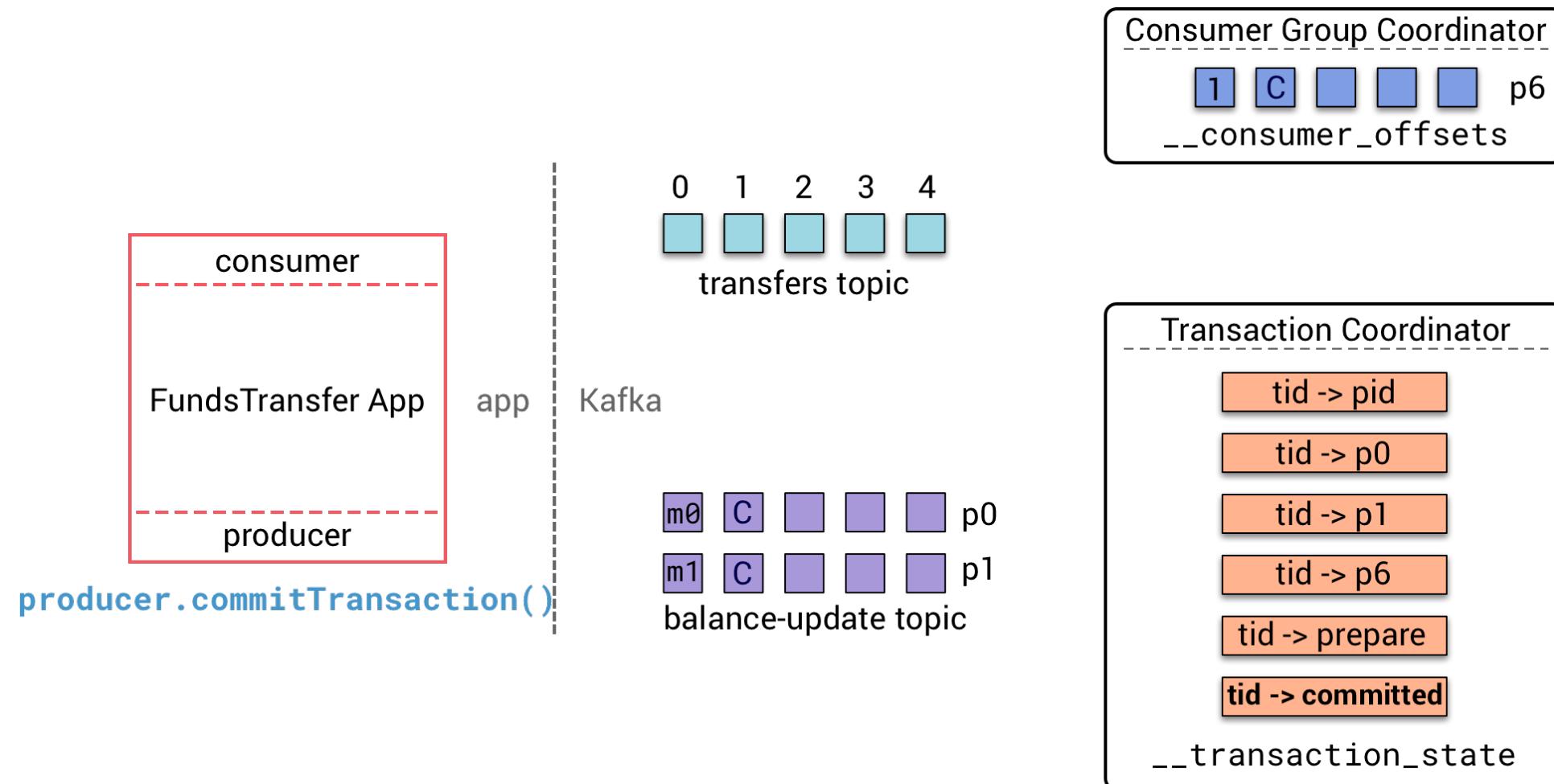
Transactions - Prepare Commit (11/14)



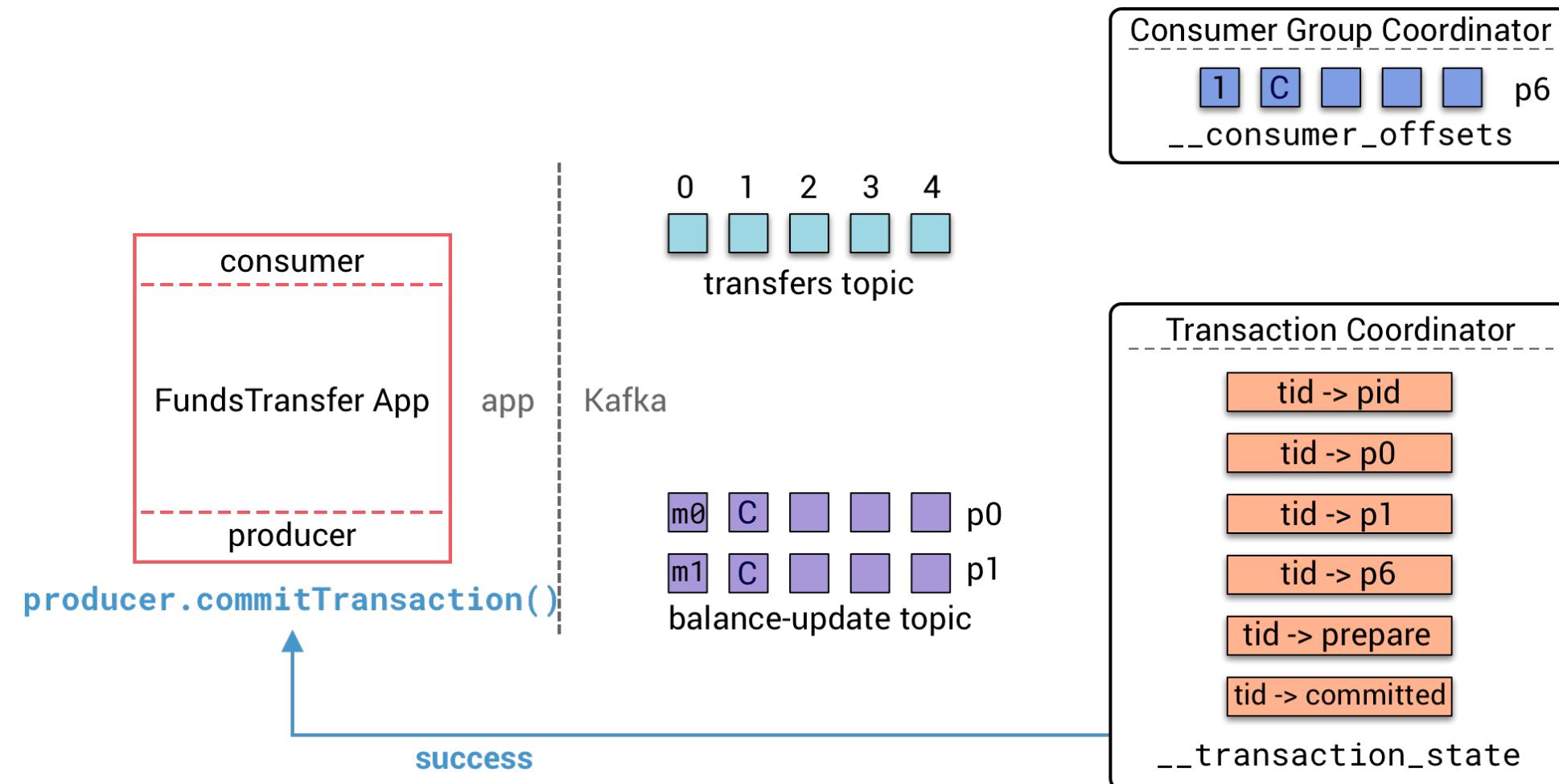
Transactions - Write Commit Markers (12/14)



Transactions - Commit (13/14)



Transactions - Success (14/14)



Transactions API Code Example (1)

```
1 while (true) {  
2     producer.initTransactions();  
3     ConsumerRecords<K,V> inputRecords = consumer.poll(...)  
4     Map<TopicPartition, OffsetAndMetadata> offsetMap = getOffsets(inputRecords)  
5  
6     try {  
7         producer.beginTransaction();  
8         for (ConsumerRecord<K,V> inputRecord : inputRecords) {  
9             List<ProducerRecord<K,V>> outputRecords = doStuff(inputRecord)  
10            for (ProducerRecord<K,V> outputRecord : outputRecords) {  
11                producer.send(outputRecord)  
12            }  
13        }  
14        producer.sendOffsetsToTxn(offsetMap, "consumer-group-id");  
15        producer.commitTransaction();
```

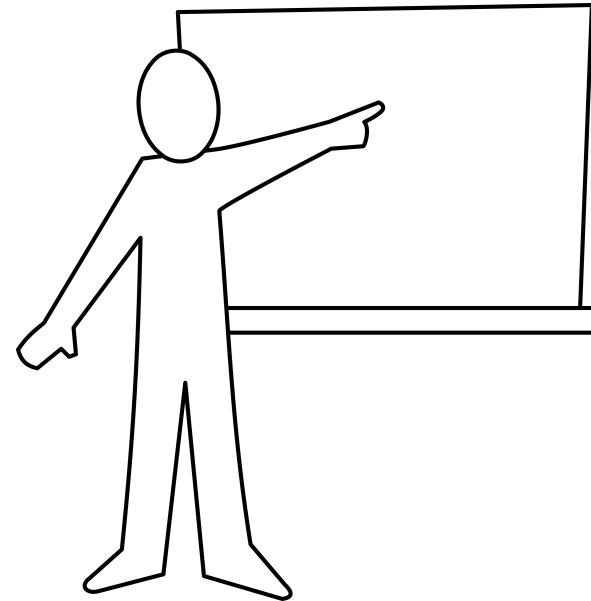
Transactions API Code Example (2)

```
16 } catch( ProducerFencedException e) {
17     producer.close();
18 } catch( KafkaException e ) {
19     producer.abortTransaction();
20 }
21 }
```

The Downstream Consumer

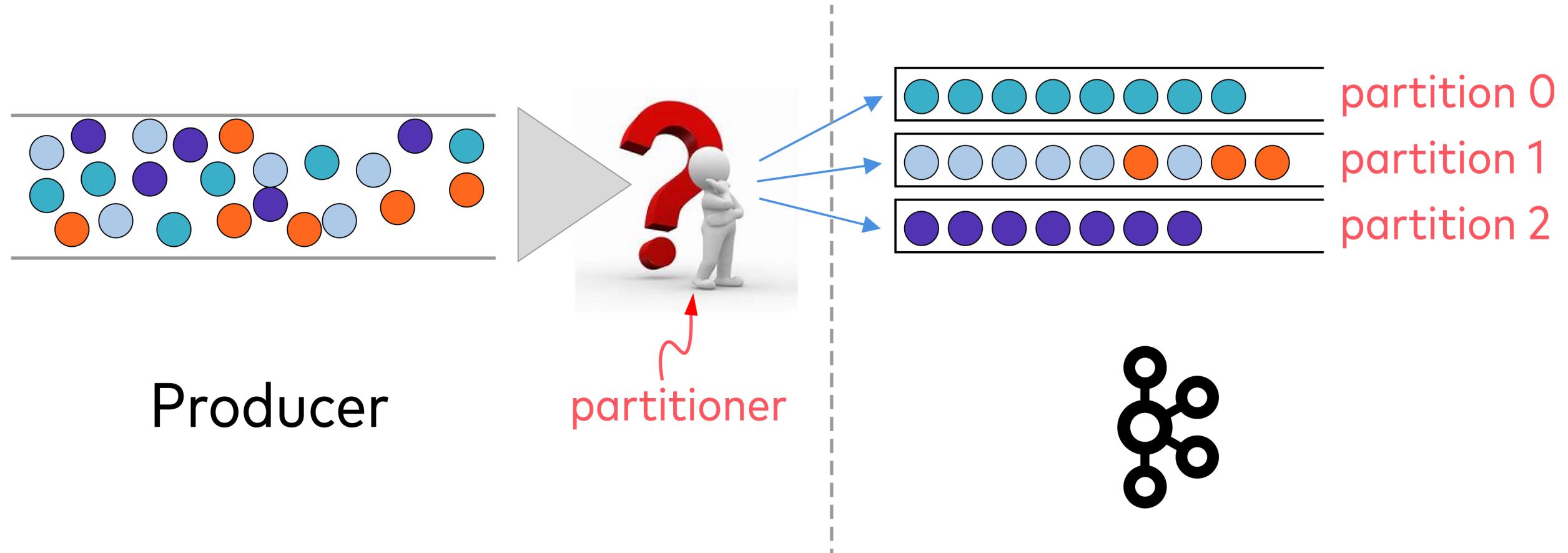
- Set `isolation.level = read_committed` to only read messages from committed transactions
 - Broker provides "abort index" so consumers can filter out aborted messages
- `read_committed` **does not protect against reprocessing** after rebalance
- For "exactly once" message handling downstream, we still need:
 - At-least-once + idempotent writes to external system
 - Ex: Hadoop Sink Connector
 - Ex: JDBC Sink Connector
 - Ex: S3 Sink Connector
 - Or, store **offsets and message together** atomically in external system

Module Map



- "What could possibly go wrong?"
- Exactly Once Semantics (EOS)
- Decisions Around Keys and Partitions ... ←
- ksqlDB, Kafka Streams, or Kafka Connect SMTs?
- Authenticating to a secure cluster
- 🔑 Hands-on Lab

Discussion: Deciding Number of Partitions



- How many partitions should you create for your topic?

Guideline for Choosing Number of Partitions

- Suggested number of partitions: $\max(t/p, t/c)$
 - t : target throughput
 - p : Producer throughput per Partition
 - c : Consumer throughput per Partition
-

- Vary Producer properties:
 - Replication factor
 - Message size
 - In flight requests per connection
(`max.in.flight.requests.per.connection`)
 - Batch size (`batch.size`)
 - Batch wait time (`linger.ms`)
- Vary Consumer properties:
 - Fetch size (`fetch.min.bytes`)
 - Fetch wait time (`fetch.max.wait.ms`)

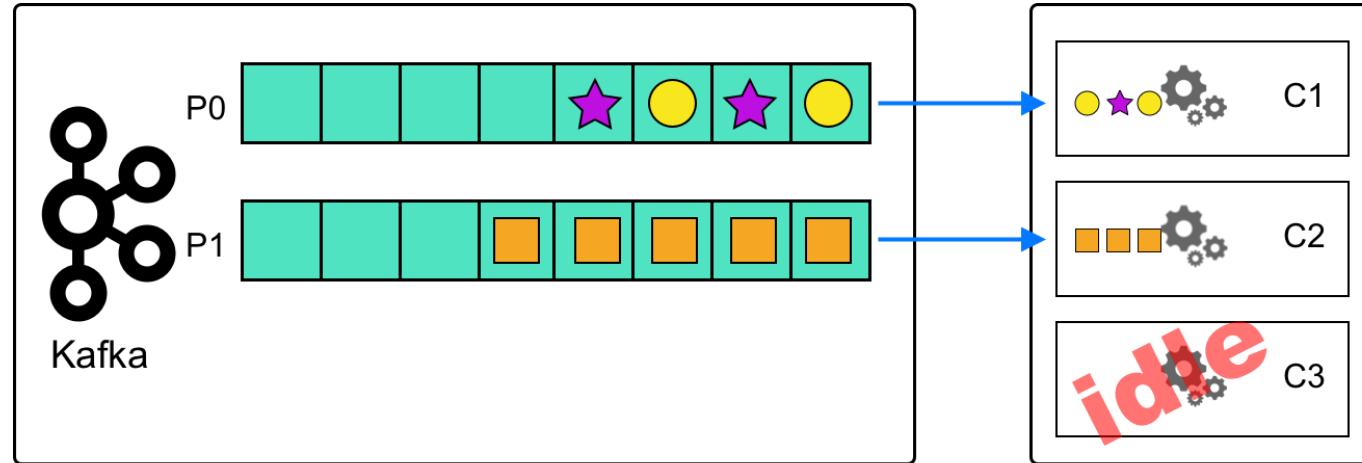
Discussion: Deciding the Number of Consumers

- How does a topic's partition count affect consumer group scalability?
- Why should all topics have a **highly divisible** number of partitions?
- With the default partition assignment strategy (RangeAssignor), what would happen if a consumer group of 10 consumers subscribed to 10 topics, each topic with 1 partition?

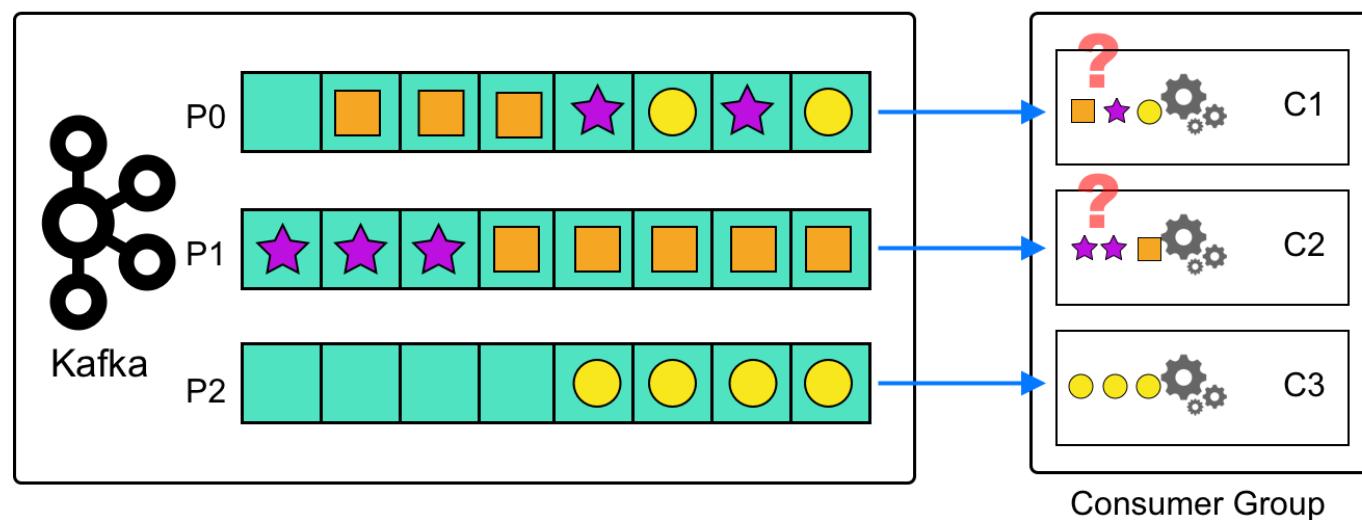
Discussion: What If I Need More Partitions?

What are the consequences and options around increasing a topic's number of partitions?

Increasing Partitions: Accept Your Fate



change nbr
of partitions



Increasing Partitions: Migrate to a New Topic

- ksqlDB makes it relatively simple to populate the new topic:

```
CREATE STREAM old WITH (KAFKA_TOPIC='old-topic',VALUE_FORMAT='JSON');

CREATE STREAM new WITH (KAFKA_TOPIC='new-topic',VALUE_FORMAT='JSON',PARTITIONS=12)
  AS SELECT * FROM old
  EMIT CHANGES;
```

Discussion: What If I want to Partition Differently?

- Can you think of any situations where `hash(key) % num partitions` might not be the most appropriate partitioning logic?

Custom Producer Partitioner

- Implement `Partitioner` interface and customize `partition()`

```
1 public interface Partitioner extends Configurable, ... {  
2     void configure(java.util.Map<java.lang.String,?> configs);  
3     void close();  
4     void onNewBatch(String topic, Cluster cluster, int prePartition);  
5  
6     int partition(java.lang.String topic,  
7                   java.lang.Object key,  
8                   byte[] keyBytes,  
9                   java.lang.Object value,  
10                  byte[] valueBytes,  
11                  Cluster cluster);  
12 }
```

Custom Partitioner: Example (1)

- In this example, we want to store all messages with a particular key in one partition and distribute all other messages across the remaining partitions. The following code sets the stage:

```
1 public class MyPartitioner implements Partitioner {  
2     public void configure(Map<String, ?> configs) {}  
3     public void close() {}  
4     public void onNewBatch() {}  
5  
6     public int partition(String topic, Object key, byte[] keyBytes,  
7                           Object value, byte[] valueBytes, Cluster cluster) {  
8         int numPartitions = cluster.partitionsForTopic(topic).size();  
9  
10        if ((keyBytes == null) || (!(key instanceof String)))  
11            throw new InvalidRecordException("Record did not have a string Key");
```

Custom Partitioner: Example (2)

- And the remaining code contains the decision logic:

```
12     if (((String) key).equals("OurBigKey"))
13         return 0; // This key will always go to Partition 0
14
15     // Other records will go to remaining partitions using a hashing function
16     return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;
17 }
18 }
```

An Alternative to a Custom Partitioner

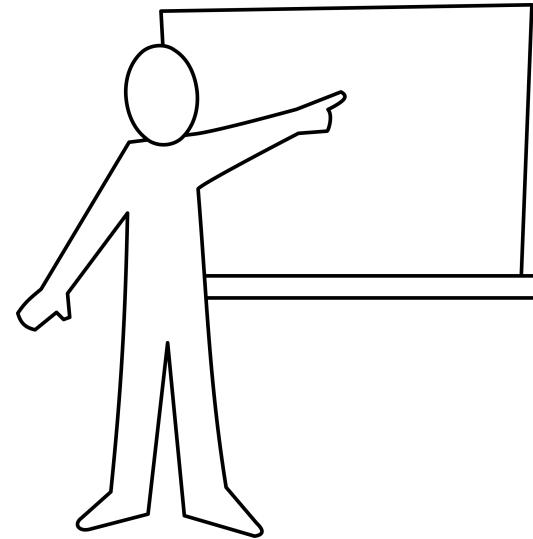
- Specify Partition when defining `ProducerRecord`

```
ProducerRecord<String, String> record  
    = new ProducerRecord<String, String>("my_topic", 0, key, value);
```

Writes message to Partition 0

Question: Which method is preferable?

Module Map



- "What could possibly go wrong?"
- Exactly Once Semantics (EOS)
- Decisions Around Keys and Partitions
- ksqlDB, Kafka Streams, or Kafka Connect SMTs? ... ←
- Authenticating to a secure cluster
- ⚒ Hands-on Lab

ksqlDB vs. KStreams

Start with ksqlDB when...

- You don't use Java/Scala
- You are new to streaming or Kafka
- You prefer a UI or REST API
- You can achieve stream processing and state querying use case with ksqlDB SQL syntax

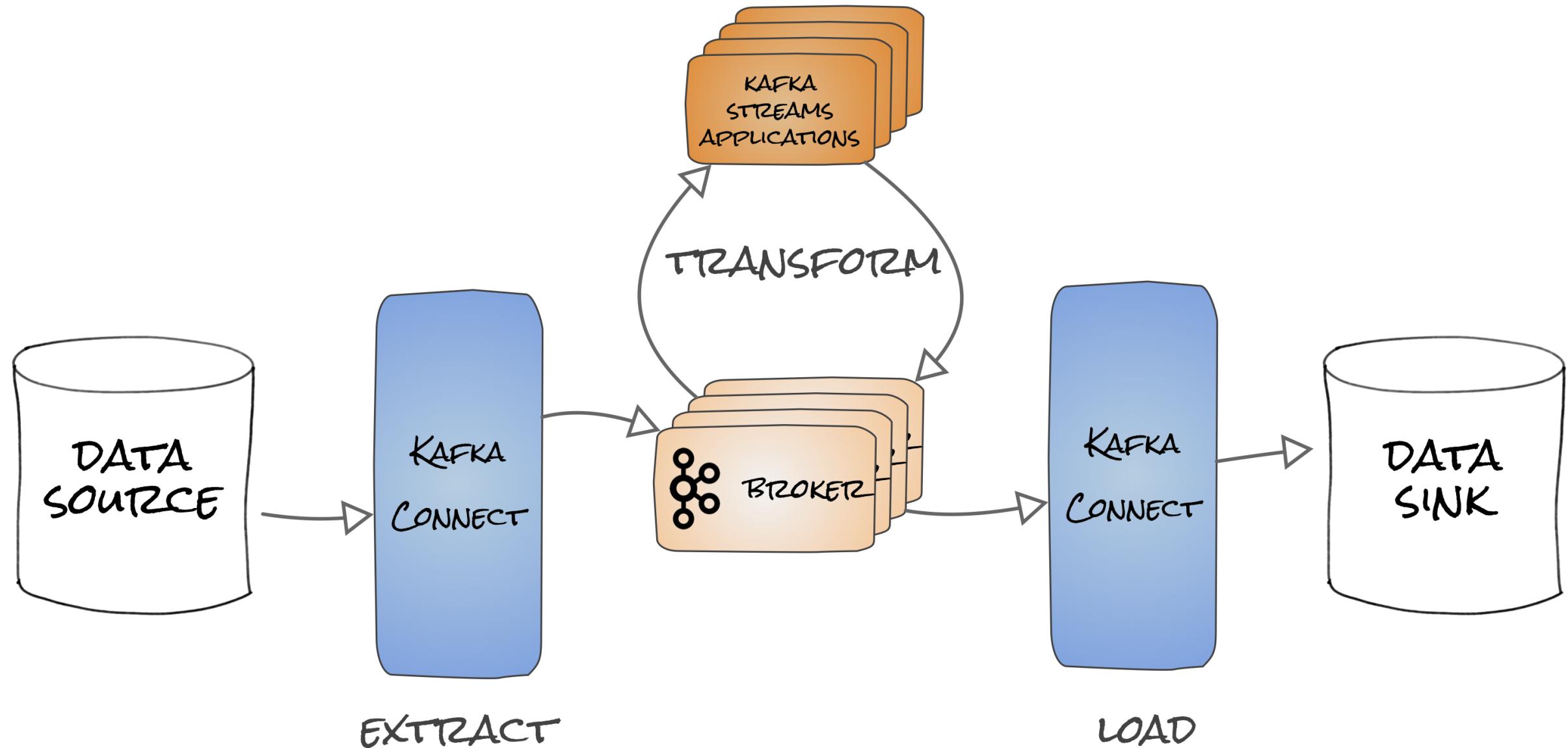
Start with Kafka Streams when ...

- You already use Java/Scala
- You need tight control over performance
- You prefer the deployment flexibility of using a Java Library
- You need custom logic that can't be described with ksqlDB SQL syntax

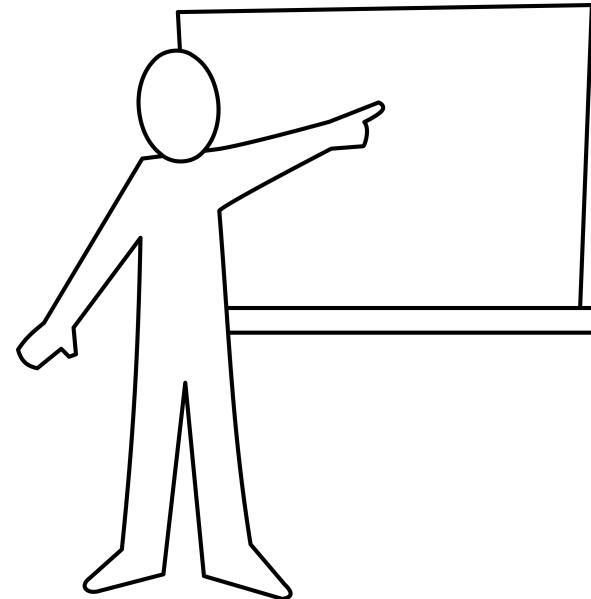


You can start with ksqlDB and expand functionality later with User Defined Functions (UDFs) in Java or Scala.

SMTs vs. Streaming Applications

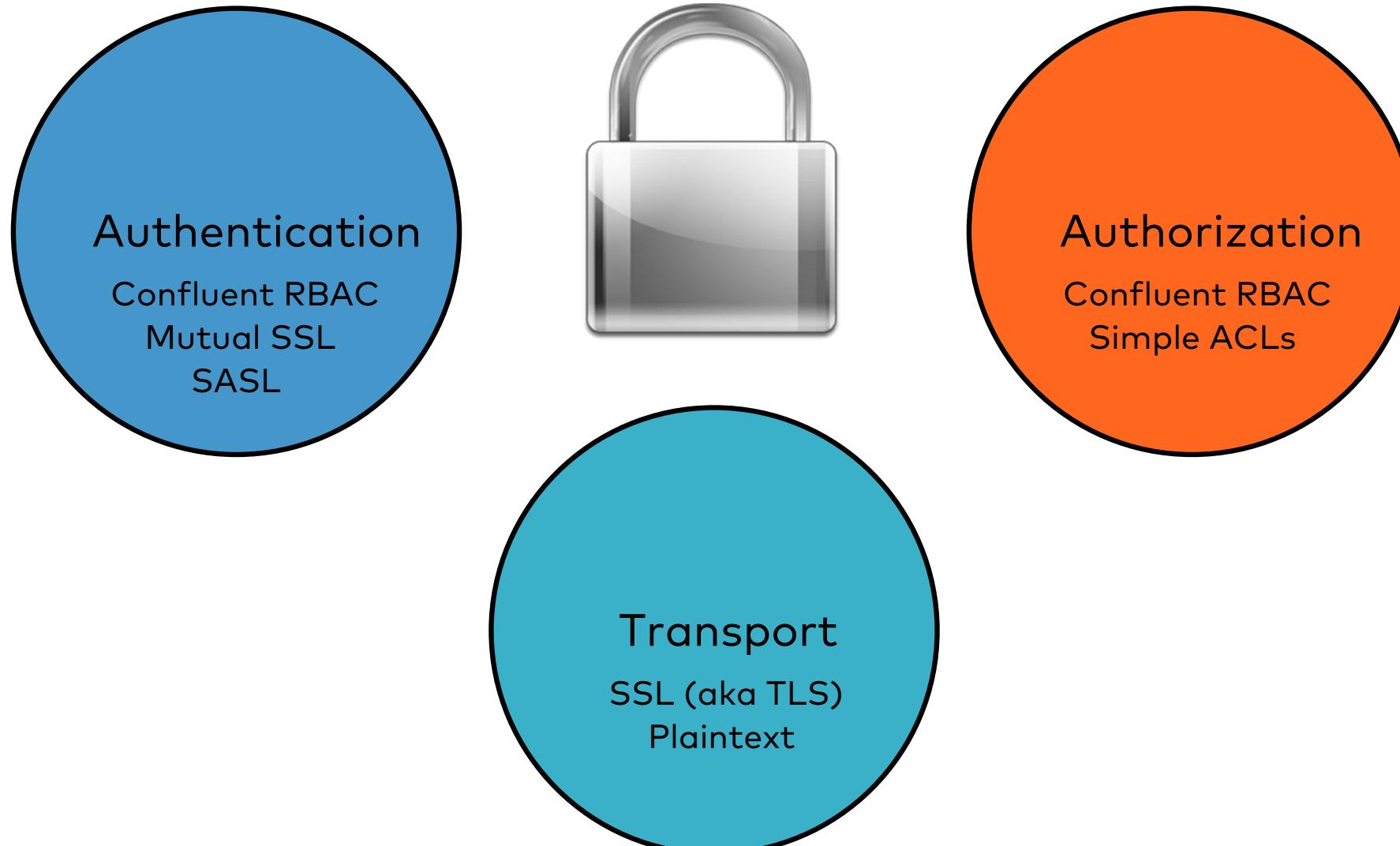


Module Map



- "What could possibly go wrong?"
- Exactly Once Semantics (EOS)
- Decisions Around Keys and Partitions
- ksqlDB, Kafka Streams, or Kafka Connect SMTs?
- Authenticating to a secure cluster ... ←
- 🤳 Hands-on Lab

Authenticating to a Secure Kafka Cluster



Configure Client for SSL

- Transport Encryption:

client.properties

```
1 ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks  
2 ssl.truststore.password=password-to-truststore-file  
3 security.protocol=SSL
```

- Add Mutual SSL for Authentication:

client.properties

```
4 ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks  
5 ssl.keystore.password=password-to-keystore-file  
6 ssl.key.password=password-to-private-key
```

Configure Client for SASL

client.properties

```
1 ...
2 security.protocol=SASL_SSL
3 sasl.mechanism=SCRAM-SHA-256
4
5 sasl.jaas.config= \
6   org.apache.kafka.common.security.scram.ScramLoginModule required \
7     username="alice" \
8     password="alice-secret";
9 ...
```

Hands-On Lab

- In this Hands-On Exercise, you will experiment with partitioning to observe how the key-partition guarantee affects your application design.
- Please refer to **Lab 09 Design Decisions** in the Exercise Book:
 - a. **Increasing the Topic Partitions**
 - b. **Kafka Consumer - offsetForTimes**



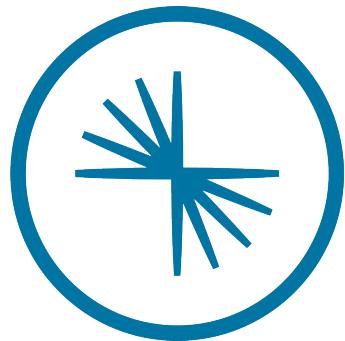
Module Review



Questions:

- What are the design decisions **you** need to make?
- Based on what you have learned, what are some design decisions you are already confident about?
- Which design decisions are you still undecided about, and how can you make progress in deciding?

10 Confluent Cloud



CONFLUENT

Agenda



1. Introduction
2. Fundamentals of Apache Kafka
3. Producing Messages to Kafka
4. Consuming Messages from Kafka
5. Schema Management in Kafka
6. Stream Processing with Kafka Streams
7. Data Pipelines with Kafka Connect
8. Event Streaming Apps with ksqlDB
9. Design Decisions
10. Confluent Cloud ... ←
11. Conclusion
12. Appendix: Basic Kafka Administration

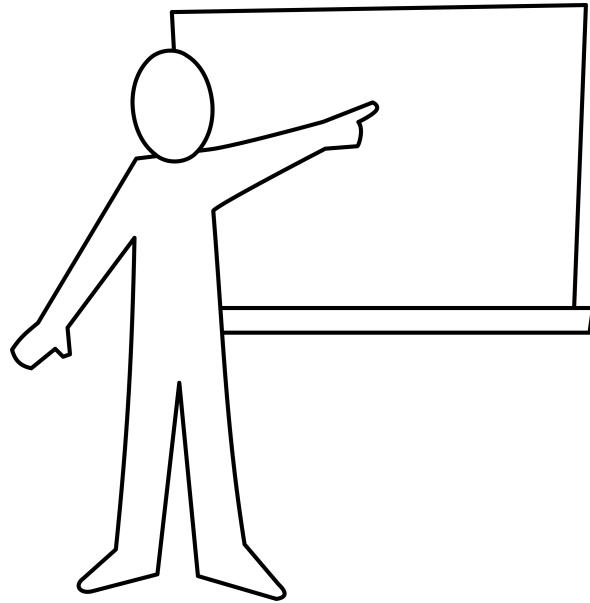
Learning Objectives



After this module you will be able to:

- explain what "fully-managed" means in the context of Confluent Cloud
- do basic operations with the `ccloud` CLI

Module Map



- Confluent Cloud Overview ... ←
- Confluent Cloud Web UI
- Confluent Cloud CLI

Freedom of Choice

Self-Managed Software

Confluent Platform

The Enterprise Distribution of
Apache Kafka



Deploy on any platform, on-prem or cloud



Fully-Managed Service

Confluent Cloud

Apache Kafka Re-engineered
for the Cloud



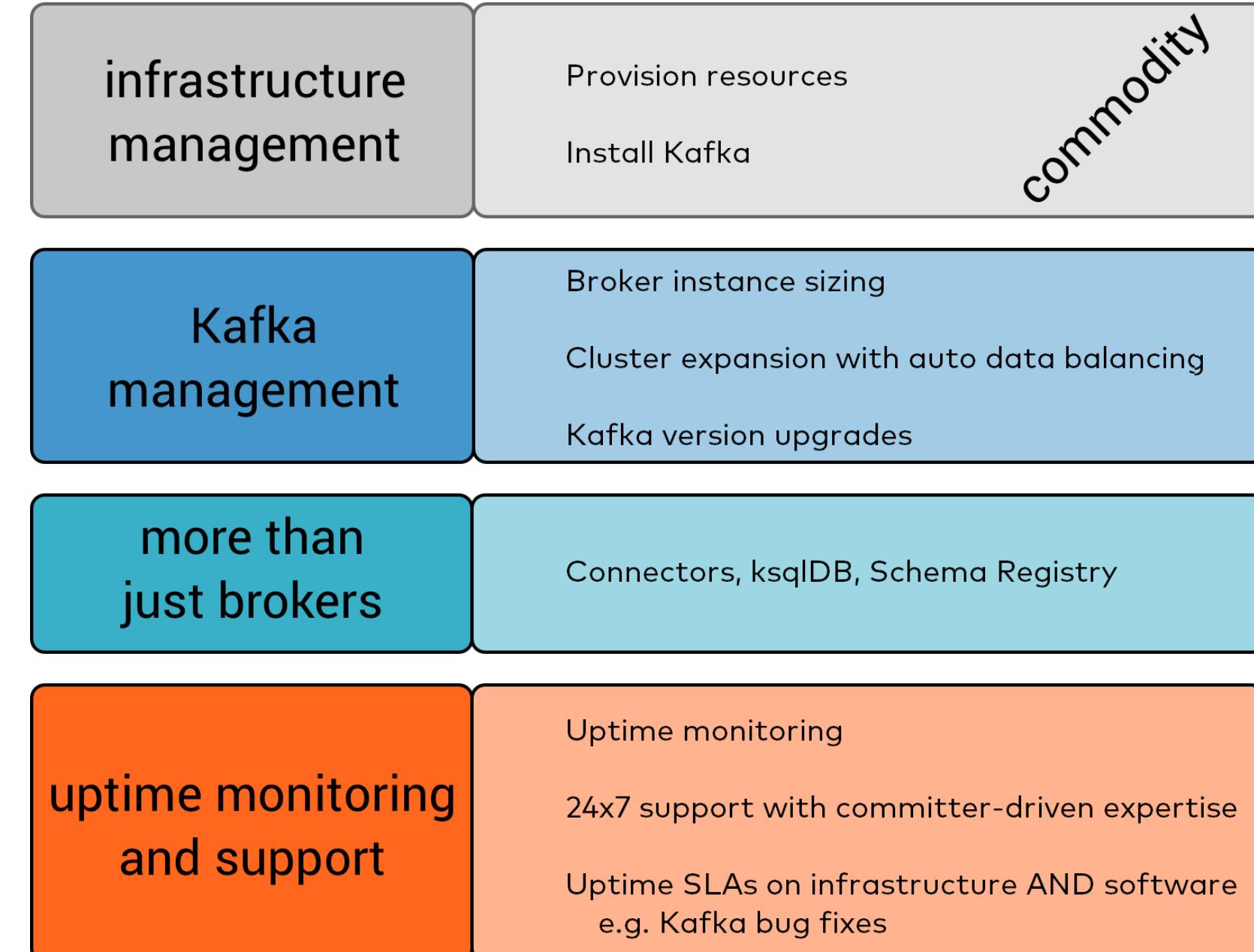
Available on the leading public clouds



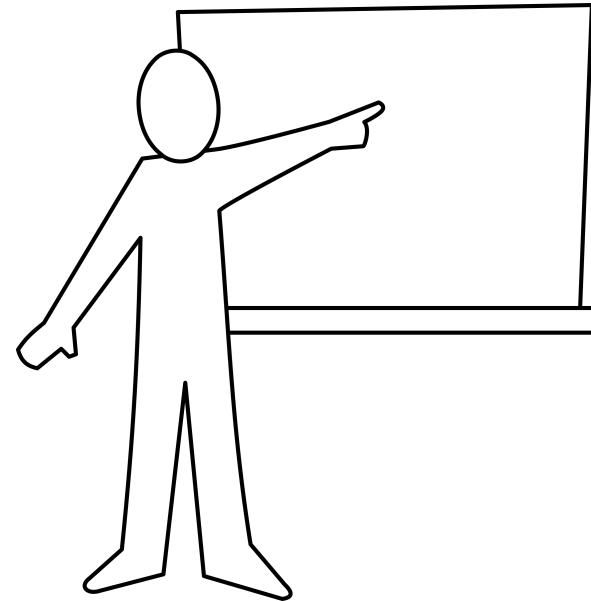
What is Confluent Cloud?

- **Fully managed** Streaming Platform built on Apache Kafka
- Manage via Web UI
- Automate via CLI
- All communication between users and Confluent Cloud is secured using TLS encryption.

What Does Fully-Managed Mean?



Module Map



- Confluent Cloud Overview
- Confluent Cloud Web UI ... ←
- Confluent Cloud CLI

Confluent Cloud Web Interface

The image displays three screenshots of the Confluent Cloud Web Interface:

- Cluster settings (Left):** Shows the Kafka section with cluster details like name, ID, provider, region, and type. It also includes usage limits for ingress, egress, storage, partitions, and uptime SLA.
- Topics (Top Right):** A list of topics with their partitions. Topics include PAGEVIEWSFEMALE, PAGEVIEWSFEMALELIKE_89, PAGEVIEWSEGREGATION, USERSORIGINAL, pageviews, pksqlc43do0processinglog, and pksqlc43do0PAGEVIEWSFEMAL...
- Topic Summary (Bottom Right):** A detailed view for a specific topic, showing configuration parameters such as name, partitions, replication.factor, cluster, min.insync.replicas, cleanup.policy, retention.ms, and retention.bytes.

Consumer Lag

CONFLUENT

Search default

ALL CONSUMER GROUPS > connect-replicator

916 +0 messages
Total Messages behind 5 second interval

Current progress in processing

pageviews Max lag / consumer: 171 messages

Consumer ID	Topic Partition	Partition	Lag	Messages behind	Current offset	End offset
	pageviews	4	156	156	97116	97272
	pageviews	5	145	145	97107	97252
	pageviews	2	154	154	96460	96614

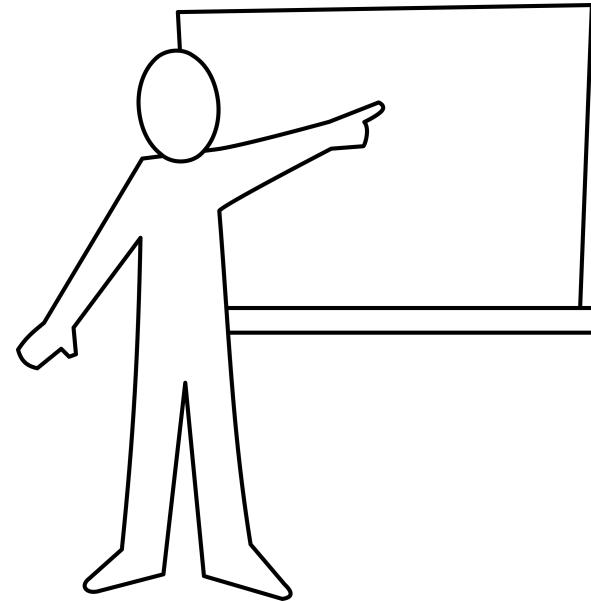
Client Configuration

The screenshot shows the Confluent Cloud interface for client configuration. On the left, there's a sidebar with three clusters: Cluster 1 (CC), Cluster 2 (DH), and Cluster 3 (RU). Under Cluster 3, 'Schemas' is selected. The main area is titled 'Java' and contains a code snippet for Kafka client configuration:

```
# Kafka
bootstrap.servers=pkcs-4nym6.us-east-1.aws.confluent.cloud:9092
security.protocol=SASL_SSL
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="{{ CLUSTER_API_KEY }}"
password="{{ CLUSTER_API_SECRET }}";
ssl.endpoint.identification.algorithm=https
sasl.mechanism=PLAIN
# Confluent Cloud Schema Registry
schema.registry.url=https://psrc-4nrnd.us-central1.gcp.confluent.cloud
basic.auth.credentials.source=USER_INFO
schema.registry.basic.auth.user.info={{ SR_API_KEY }}:{{ SR_API_SECRET }}
```

Below the code, there are two buttons: 'Copy to clipboard' and 'See example'. A note says 'If necessary, create a key/secret pair for your Schema Registry.' with a link to 'Create New Schema Registry API key & secret'. There are also sections for 'Kotlin', 'Node.js', and 'Python'.

Module Map



- Confluent Cloud Overview
- Confluent Cloud Web UI
- Confluent Cloud CLI ... ←

Confluent Cloud CLI - Getting Started

- Confluent Cloud Commands:

```
$ ccloud help
```

Manage your Confluent Cloud.

Usage:

```
ccloud [command]
```

Available Commands:

api-key	Manage API keys
environment	Manage and select ccloud environments
kafka	Manage Apache Kafka
login	Login to Confluent Cloud
logout	Logout of Confluent Cloud
service-account	Manage service accounts
...	

- Login to your account:

```
$ ccloud login
```

Enter your Confluent Cloud credentials:

Email: <YOUR_EMAIL>

Password: <YOUR_PASSWORD>

Logged in as xyz@confluent.io

Using environment tXXXX ("confluent-training")

- List Environments:

```
$ ccloud environment list
```

- Select your environment:

```
$ ccloud environment use <YOUR_ENVIRONMENT>
```

Confluent Cloud CLI - Manage Clusters

- List all clusters in your environment

```
$ ccloud kafka cluster list
  Id      ! Name   ! Provider ! Region     ! Durability ! Status
+-----+-----+-----+-----+-----+
  lkc-lovz9 ! Alpha ! gcp    ! europe-west3 ! LOW        ! UP
```

- Select your cluster

```
$ ccloud kafka cluster use <CLUSTER_ID>
```

Confluent Cloud CLI - Manage Topics

```
$ ccloud kafka topic list  
demo-topic  
other-topic  
my-topic
```

```
$ ccloud kafka topic create product-topic \  
--replication-factor 3 \  
--partitions 6
```

```
$ ccloud kafka topic describe product-topic  
Topic: products PartitionCount: 6 ReplicationFactor: 3  
Topic | Partition | Leader | Replicas | ISR  
+-----+-----+-----+-----+  
product-topic | 0 | 2 | [2 4 9] | [2 4 9]  
product-topic | 1 | 3 | [3 2 0] | [3 2 0]  
product-topic | 2 | 1 | [1 3 8] | [1 3 8]  
...
```

Module Review



Questions:

- What does "fully managed" mean in terms of Confluent Cloud?
- How is this different from other "Kafka as a Service" providers?
- How can you quickly authenticate your application to Confluent Cloud?



You are encouraged to try Confluent Cloud yourself! It's free to get started.

Further Reading

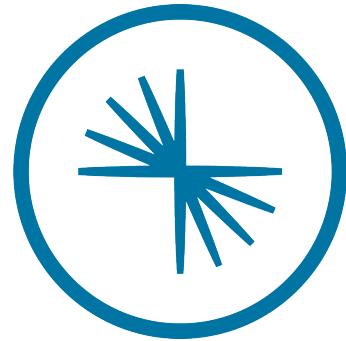
- Security Overview and Recommendations:

https://www.confluent.io/wp-content/uploads/CC_Security_WP.pdf

- What specific security features does Confluent Cloud offer?

<https://docs.confluent.io/current/cloud/faq.html#what-specific-security-features-does-ccloud-offer>

11 Conclusion



CONFLUENT

Course Contents



Now that you have completed this course, you should have the skills to:

- Write Producers and Consumers to send data to Apache Kafka
- Create schemas, describe schema evolution, and integrate with Confluent Schema Registry
- Integrate Kafka with external systems using Connect
- Write streaming apps with Kafka Streams & ksqlDB
- Describe common issues faced by Kafka developers and ways to troubleshoot
- Make design decisions about acks, keys, partitions, batching, replication, and retention policies

Other Confluent Training Courses

- Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB
 - recommended to take this next!
- Apache Kafka® Administration by Confluent
- Confluent Advanced Skills for Optimizing Apache Kafka®



For more details, see <https://confluent.io/training>

Confluent Certified Developer for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and Confluent products, and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours a day!

Cost: \$150

Register online: www.confluent.io/certification



Confluent Certified Administrator for Apache Kafka

Duration: 90 minutes

Qualifications: Solid work foundation in Confluent products and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours per day!

Cost: \$150

Register online: www.confluent.io/certification



We Appreciate Your Feedback!



- Thank you for attending the course!
- Please complete the course survey
- For additional feedback, email training-admin@confluent.io