Confluent Developer Skills for Building Apache Kafka® Applications

Version 7.0.0-v1.0.5

CONFLUENT

Table of Contents

Introduction

CONFLUENT
**Global Education**

Class Logistics and Overview

Copyright & Trademarks

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

---

Attendees should have a working knowledge of the Kafka architecture, either from prior experience or the recommended prerequisite course Confluent Fundamentals for Apache Kafka®.

This free course is available at https://training.confluent.io/learningpath/apache-kafka-fundamentals for anyone who needs to catch up.

Agenda

This course consists of these major parts:

1. **Core Kafka Development**
   a. Bridging from Fundamentals
   b. Producers
   c. Consumers

2. **Other Components of a Kafka Deployment**
   a. Schema management
   b. A taste of stream processing
   c. Integrating with other systems

3. **Additional Challenges in Core Kafka Components**
   a. Advanced matters
   b. Design decisions

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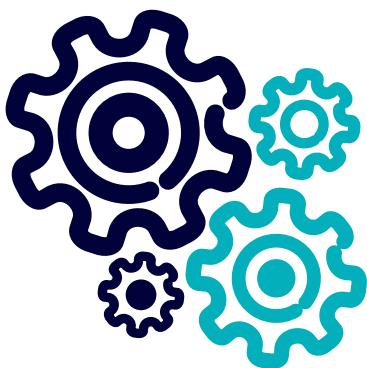
Upon completion of this course, you should be able to:

- 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 polices

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

Class Logistics

- Timing
  - Start and end times
  - Can I come in early/stay late?
  - Breaks
  - Lunch
- Physical Class Concerns
  - Restrooms
  - Wi-Fi and other information
  - Emergency procedures
  - Don't leave belongings unattended

No recording, please!

Expanding on the rule at the bottom: You are not permitted to record via any medium, or stream via any medium any of the content from this class.

How to get the courseware?

1. Register at **training.confluent.io**

2. Verify your email

3. Log in to **training.confluent.io** and enter your **license activation key**

4. Go to the **Classes** dashboard and select your class

Your instructor may choose to have you do this now, combine it with the first lab, or do it before class begins.

Introductions

- About you:

  ◦ What is your name, your company, and your role?

  ◦ Where are you located (city, timezone)?

  ◦ What is your experience with Kafka?

  ◦ Which other Confluent courses have you attended, if any?

  ◦ Optional talking points:

    ▪ What are some other distributed systems you like to work with?

    ▪ What technology most excited you early in your life?

    ▪ Anything else you want to share?

- About your instructor

Fundamentals Review

Discussion

Question Set 1 [6 mins]

Determine if each statement is true or false and why:

1. All messages in a topic are on the same broker.

2. All messages in a partition are on the same broker.

3. All messages that have the same key will be on the same broker.

4. The more partitions a topic has, the better.

Question Set 2 [3 mins]

Determine the best answer to each question.

1. What are the roles of a producer and a consumer?

2. How is it decided which messages consumers read?

3. Who initiates the reading of messages: consumers or the Kafka cluster?

How your instructor approaches this section may vary depending on the particular class.

Discussion, Cont'd.

Question 3 [1 min]

Suppose there is a message in our Kafka cluster about my breakfast purchase of $12.73. Consumer c_0 has consumed it to process the charge. Could consumer c_7 consume this same message this afternoon?

Question 4 [2 mins]

Kafka has a transactions API. When we know that all messages that are part of a transaction successfully made it to the cluster, we want to tag those messages as "good." When we know that not all messages in a transaction made it, we want to tag those messages that did make it as "bad." Kafka uses markers in the logs, that are effectively new messages written after existing messages to do this. Why not just put something in the metadata? Why not delete "bad" messages?

Instructor-Led Review

Some time is allocated here for an instructor-led review/Q&A on prerequisite concepts from Fundamentals.

Core Overview

CONFLUENT
Global Education

Agenda

This is a branch of our developer content on core developer concepts. It is broken down into the following modules:

1. Introductory Concepts

2. Starting with Producers

3. Preparing Producers for Practical Uses

4. Starting with Consumers

5. Groups, Consumers, and Partitions in Practice

Here is an expanded version of the outline on the slide, including the lessons that make up each module:

1. Introductory Concepts

   a. How Can I Connect to a Cluster?

   b. How Do I Control How Kafka Retains Messages?

   c. How Can I Leverage Replication?

2. Starting with Producers

   a. Basic Concepts of Kafka Producers

   b. Producers: Code Basics

3. Preparing Producers for Practical Uses

   a. How Can Producers Leverage Message Batching?

   b. How Do Producers Know Brokers Received Messages?

   c. How Can a Producer React to Failed Delivery?

4. Starting with Consumers

   a. Fetch Requests

   b. Basic Concepts of Kafka Consumers

   c. Consumers: Code Basics

5. Groups, Consumers, and Partitions in Practice

a. How Do Groups Distribute Workload Across Partitions?

b. How Does Kafka Manage Groups?

c. How Do Consumer Offsets Work with Groups?

After this, you are prepared to go to either of these branches:

• Other Components of a Kafka Deployment

• Additional Challenges in Core Kafka Components

01: Introductory Concepts

CONFLUENT
Global Education

Module Overview

This module contains three lessons:

  a. How Can You Connect to a Cluster?

  b. How Do You Control How Kafka Retains Messages?

  c. How Can You Leverage Replication?

Where this fits in:

• Hard Prerequisite: Fundamentals Course

• Recommended Follow-Up: Starting with Producers

01a: How Can You Connect to a Cluster?

Description

How brokers are interconnected and how this allows us to leverage bootstrap servers. Best practice for bootstrap servers and examples of how to configure for various clients and in a CLI example as well.
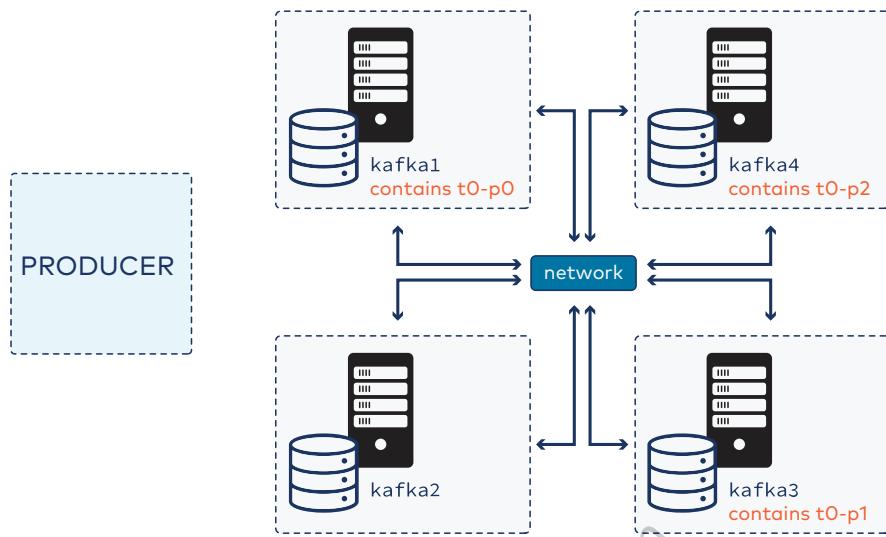
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain what a bootstrap server is

- Given a hypothetical cluster, provide a list - not just one of bootstrap servers

- Write a CLI command that requires connecting to a Kafka cluster

- Write code to connect to a cluster from a producer or consumer or Kafka Streams application explicitly

- Describe how to specify how to make a Connect worker connect to a cluster

View of a Cluster and Producer
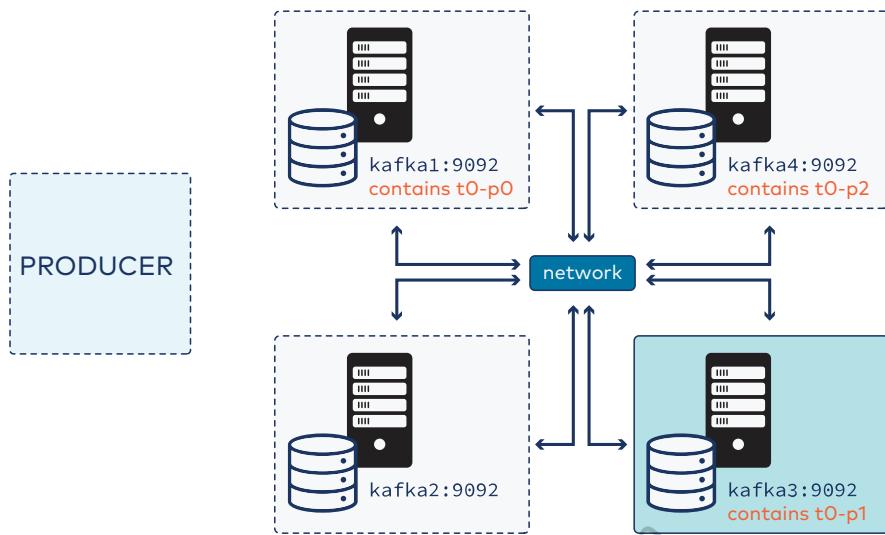
- Producer wants to send new message with key 10
- Producer chooses partition (how?)

Note that all of the brokers are interconnected.
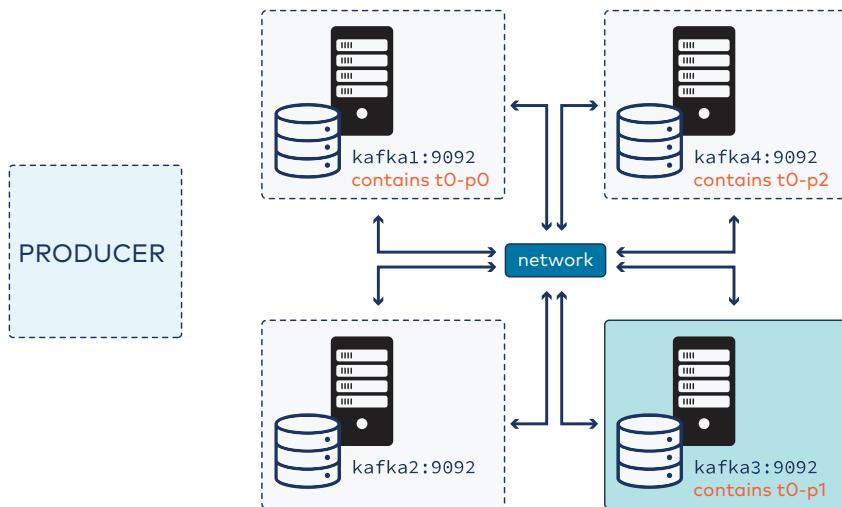
Note that this graphic shows only the **leader** replicas for each partition. Writing still only happens to leaders, never to followers. Partitioning and replication are independent, yet related, things.

Identifying Brokers

- Brokers identified by `host:port` pairs

- Which broker do we write our new message to?

Bootstrap Server

- Clients, like producers, need to specify a **bootstrap server**

- This is the **initial** connection to the cluster and all you need to specify in your code, configuration

---

We use the term **client** to refer to producers and consumers that are external to the cluster.

As long as a client connected to any broker in the cluster, it is indirectly connected to all brokers and thus can send or receive messages.

Bootstrap Servers in Practice (1)

Specify bootstrap server in CLI commands, e.g.,

```
kafka-topics
    --bootstrap-server kafka:9092
    --create
    --partitions 1
    --replication-factor 1
    --topic testing
```

Specify bootstrap server in client code, e.g.,

```
props.put("bootstrap.servers", "kafka:9092");
```

Specify bootstrap server in a properties file, e.g. for a Connect worker, e.g.,

```
bootstrap.servers = kafka:9092
```
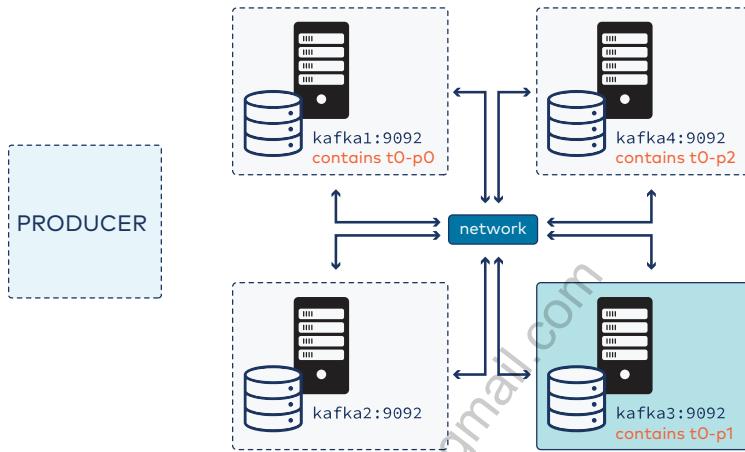
You'll learn more about setting properties in client code in the modules on the basics of producer and consumer code.

Bootstrap Servers in Practice (2)

In practice, specify a comma-separated list of bootstrap servers:

```
props.put("bootstrap.servers", "kafka1:9092, kafka2:9092, kafka3:9092");
```

Why?

Let's say we try to make the inital connection to `kafka1:9092` as a producer, but it is down. Then we're in trouble! But we can provide a backup! In this example, when `kafka1:9092` is down, the producer will then try to connect to `kafka2:9092`. If it can't succeed with that, it'll try the next bootstrap server in the list, `kafka3:9092`.

Confluent Cloud Considerations

If you are using Confluent Cloud, you can use the CLI command `confluent kafka cluster describe` to get info about your cluster, which will include the bootstrap servers value. The output will look like this:

```
+---------------+----------------------------------------------------------+
| ID            | lkc-38dy12                                               |
| Name          | cluster-2                                                |
| Type          | BASIC                                                    |
| Ingress       |                                                      100 |
| Egress        |                                                      100 |
| Storage       | 5 TB                                                     |
| Provider      | aws                                                      |
| Availability  | single-zone                                              |
| Region        | eu-west-2                                                |
| Status        | UP                                                       |
| Endpoint      | SASL_SSL://pkc-41wq6.eu-west-2.aws.confluent.cloud:9092  |
| REST Endpoint | https://pkc-41wq6.eu-west-2.aws.confluent.cloud:443      |
+---------------+----------------------------------------------------------+
```

You'd want the `host:port` part of the `Endpoint` value, (minus the `SASL_SSL://` part), so `pkc-41wq6.eu-west-2.aws.confluent.cloud:9092` in this case.

The bootstrap servers value will also be included in the file that is downloaded if you download an API key and API secret.

Also, the value may be found in the web UI's "Cluster Overview" section. You'll find it under "Cluster Settings" in the "General" tab under **Endpoints**.

Finally, while this looks like a single bootstrap server, Confluent Cloud takes care of the fault tolerance that would be achieved by providing a comma-sepaated list of `host:port` pairs as this slide suggests.

01b: How Do You Control How Kafka Retains Messages?

Description

Review of the basics of the two retention policies with concrete examples. Active vs. inactive segments and how they affect retention.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Given a log with ages of each message, describe what deletion would do

- Distinguish between an active and an inactive segment

- Describe one way an active segment becomes inactive

- Describe how active vs. inactive affects deletion and compaction

- Given a log, describe what it will look like after compaction

Retention Policies

- `delete`

- `compact`

- `delete, compact`

Retention policies are also called cleanup policies, and that term is used in the property name. These are values of the setting `cleanup.policy`.

Deletion 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)

Scenarios

Each picture shows the **age in days** of each message.

What will be left after deletion runs in each case?

Example 1		Example 2	
12 10 9 8 — segment 1		22 16 13 12 — segment 1	
4 3 — segment 2		10 9 5 — segment 2	
		5 4 3 3 — segment 3	

Use deletion for topics that contain events.

Food for Thought Question: We noted immutability in Fundamentals—we can't change nor delete a message once it's written to a log. How is it that deletion doesn't conflict with this?

Answer: Deletion deletes entire **segments**, not individual **messages**.

Alternative answer: Immutability means a message cannot be changed once written, but it doesn't impose any restriction on the expiration of said message (either because it's old - delete, or because it's been superseded - compact).

Compaction

- Keep only the latest value per key

- When is this useful?
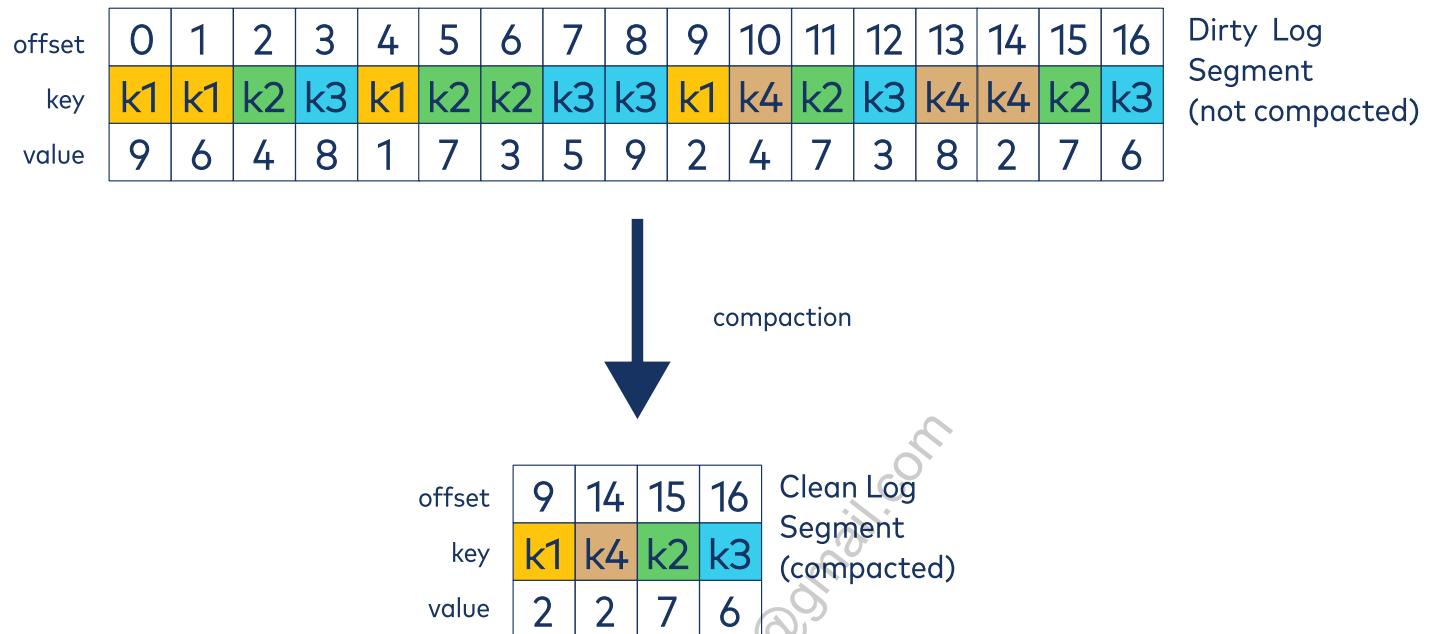
Compaction is the second retention policy. It does not apply in all cases, only some.

Two that will come up again:

- Consider a topic that keeps track of the current status on an order. When the order has been shipped, that status is fresher than a status saying the order was placed, so we probably only care about the current status, not the old one.

- Consider a topic used for temperature readings used to power a current temperature display (not to study temperature changes over time). That it's currently 66 degrees at my location now renders the information that it was 62 degrees an hour ago irrelevant. We can compact away old, "stale" data.

In short, use compaction for topics that contain state.

Compaction Example

offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Dirty Log
--------	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	
key	k1	k1	k2	k3	k1	k2	k2	k3	k3	k1	k4	k2	k3	k4	k4	k2	k3	Segment (not compacted)
value	9	6	4	8	1	7	3	5	9	2	4	7	3	8	2	7	6	

compaction

offset	9	14	15	16	Clean Log
--------	---	----	----	----	
key	k1	k4	k2	k3	Segment (compacted)
value	2	2	7	6	

Notice here that we keep only the latest value for each key.

Notice also that after compaction, messages retain their offsets. In other words, offsets will likely be non-contiguous after compaction.

Back to that idea that we can't delete individual messages? Aren't we doing that here? It sure looks like it. However, as per implementation details beyond the scope of this course, when compaction runs, it creates new copies of log segments with only the messages that are kept. Then it moves file pointers around and deletes old segments as a whole. It does not alter the existing segments directly.

Active and Inactive Segments

- Messages are written to the **active** segment of logs

- Active segment **rolls** to become inactive

- Consumers read from all segments

ℹ️	Retention policies do not alter the active segment

With deletion, even if all of the messages in the active segment are older than the retention threshold, the active segment is never deleted. The active segment must roll and become inactive to be a candidate for deletion.

Compaction ignores the active segment entirely. Even if there are duplicate keys in a log due to their appearance in the active segment, compaction does not remove the duplicates. We will explore this more deeply in an advanced module.

Clean and Dirty Segments

One more distinction about log segments that applies to compaction:

- Log segments that have been compacted are called **clean** segments

- Log segments that have not been compacted are called **dirty** segments

Activity: Compacting an Uncompacted Log

Consider the log shown. Sketch out what the log will look like after compaction.

offset	0	1	2	3	4	5	6	7	8	9	10
key	y	j	y	x	k	z	x	q	x	w	a

inactive segment 1   inactive segment 2   inactive segment 3   active segment

> ⚠️ Note that colors denote *segments* here, not keys as in a prior example.

01c: How Can You Leverage Replication?

Description

Review of leaders vs. followers. Replication factor. How messages get from leaders to followers and config. ISRs. Leader failover / leader election.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:
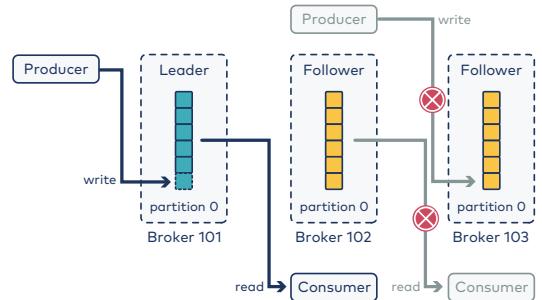
- Distinguish between leaders and followers

- Interpret a replication factor or choose a replication factor given a desired number of followers

- Explain which of leaders/followers producers/consumers access

- Explain at a high level how and when messages get from leaders to followers

- Distinguish between an in-sync replica and one that is not

- Describe what happens to trigger leader election and what results

Review: Basics of Replication

- Ensure high availability of data with backup copies

- Replicas:

  Leader

  Clients write to and read from, always one leader

  Follower

  Backup copies, keep up with leader, generally multiple followers
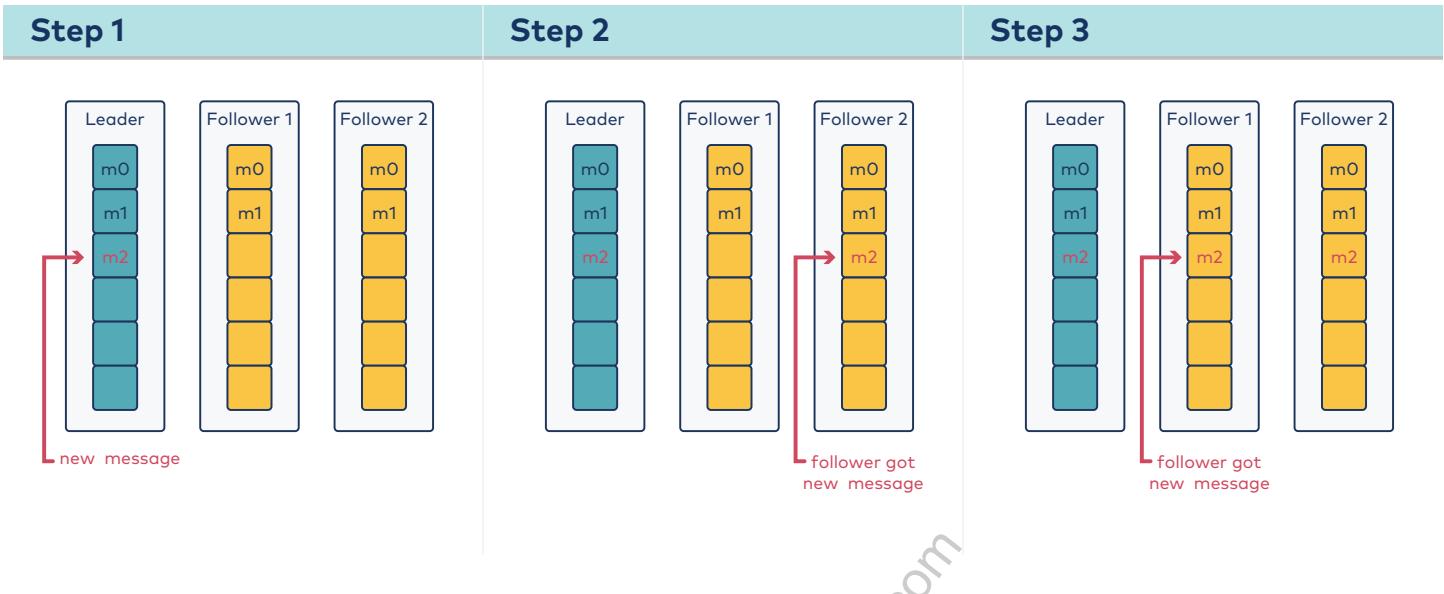
- Topic setting `replication.factor`

In the picture, `replication.factor = 3`.

Observe that producers write **only** to the leader, never followers.

Observe that consumers read **only** from the leader, never followers (under normal circumstances).

"Follow the Leader"

Step 1	Step 2	Step 3
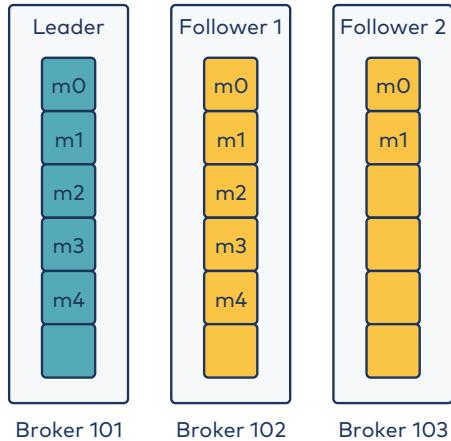
Observe the three steps that are illustrated:

1. We start with a leader that has two messages. The two followers shown are caught up, i.e. they have the same two messages. A new message is written to this partition, and it goes to the leader.

2. The followers are monitoring the leader, periodically checking for new messages. One follower gets the new message.

3. Then a second follower gets the new message.

(Both followers may be perfectly timed with each other and Steps 2 and 3 happen simultaneously, but they do not have to.)

But…

Leader — Broker 101 (m0, m1, m2, m3, m4)
Follower 1 — Broker 102 (m0, m1, m2, m3, m4)
Follower 2 — Broker 103 (m0, m1)

Observe:

• Follower 1 (on Broker 102) is an **in-sync replica** (ISR)

• Follower 2 (on Broker 103) is **not**
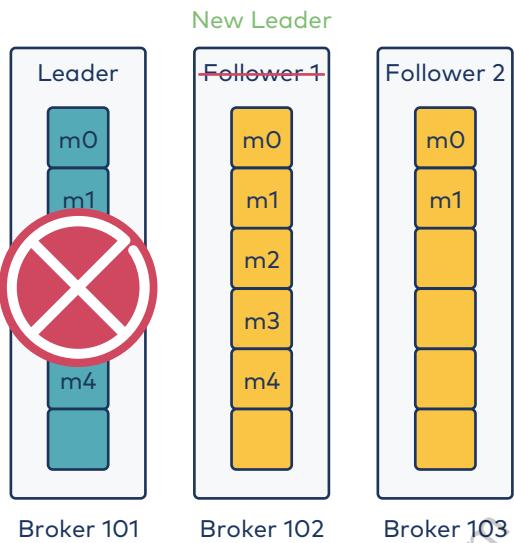
Note that the leader is always in-sync with itself and is always considered an in-sync replica.

Followers that are not ISRs are often referred to as stuck followers.

Note that replicas **are** partitions. It wouldn't make sense to have more than one replica on the same broker, so tools reporting replica information can use broker IDs to reference replicas. Given that, we would often see the ISR list for this picture written as [101, 102].

Leader Failover

New Leader

Leader	Follower 1	Follower 2
m0	m0	m0
m1	m1	m1
	m2	
	m3	
m4	m4	

Broker 101   Broker 102   Broker 103

Question: Would either choice of follower have been equally good to replace the leader that had died?

Answer: No, an in-sync replica is the best choice of a new leader. We want the (new) leader to be caught up with the old leader. It's stepping in to do the job the old leader was doing.

How Does Kafka Choose Leaders?

- Leader election happens automatically

- Kafka will generally choose an in-sync follower to become leader

- Leader election does not happen in parallel

- Background processes manage balance of leadership

It's important for you to know that when a broker containing a leader goes down, a follower will become the new leader. But, you don't need to worry about how this happens as a developer; Kafka will take care of this for you.

One thing that's important to know, however, is that the leader election process cannot be parallelized. So, if a broker that contains the leaders for four different partitions goes down, Kafka will need to choose a new leader for the first, then the second, then the third, and then the fourth—in succession and not in parallel. Having leaders spread evenly across brokers is thus good. You administrators should monitor for this, but Kafka has processes in place to monitor for it.

In case you're curious:

- For each partition, there is a preferred replica, or a broker where having its leader would yield the best balance. You might see this indicated in bold in command-line tools. Kafka has background processes that monitor how many leaders are not in the preferred place, and, when a threshold has been crossed, Kafka balances this.

- A thread called the Controller handles leader election.

- If there is no in-sync follower able to become leader, Kafka will not select an out-of-sync follower, as this could lead to data loss. Administrators may choose to turn this on, though, understanding the implications, and allowing for something called "unclean leader election."

Activity: Exploring Replica Placement & Replication Behavior

Say we have 5 brokers - b_0, b_1, …, b_4.

Say we have a replication factor of 4.

1. How many followers would we have?

2. Say leader is on broker b_4.

   a. Where could the followers be?

   b. Where could a follower **not** be?

3. Say we have 3 successfully written messages that have been properly replicated. It's time to write the fourth message.

   a. Where does it go?

   b. What happens next?

4. Say broker b_4 fails. What happens? Why?

Review the examples on the preceding slides and apply the ideas to this problem to check your knowledge.
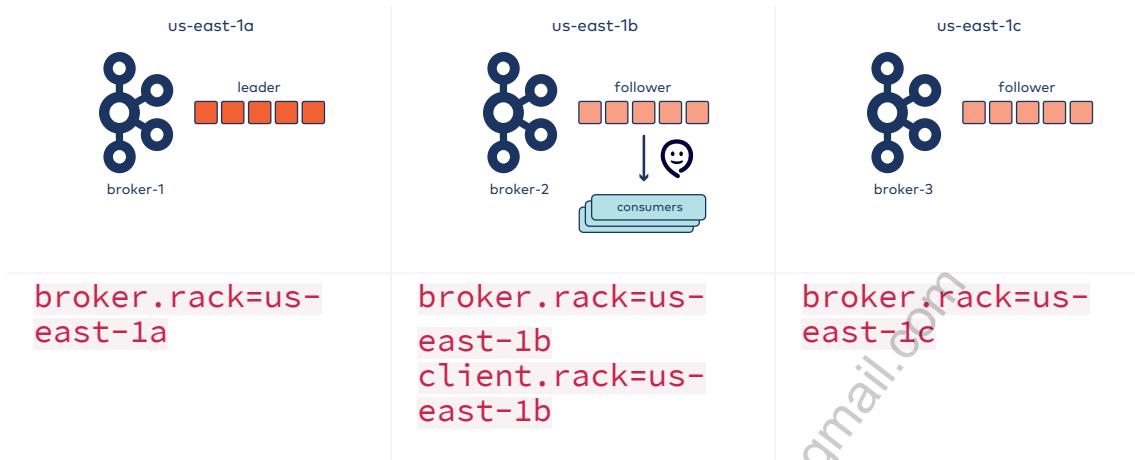
Follow up food-for-thought:

a. What if, in #4, broker b_2 failed before the action of #3b completed?

b. What if, in #4, instead, broker b_3 failed?

Follower Fetching

In this lesson, we told you all clients must interact with the leader. But, it is possible to configure it so consumers fetch from followers in the same AZ to reduce costs...

us-east-1a	us-east-1b	us-east-1c
`broker.rack=us-east-1a`	`broker.rack=us-east-1b` `client.rack=us-east-1b`	`broker.rack=us-east-1c`

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

QUESTION: What is the tradeoff?

Answer to question: Followers follow the leader and fetch records from the leader periodically. Thus, if we fetch from a follower, we might not have the most up to date information. But the follower will catch up. The cost is increased latency.

Follower fetching was introduced in AK 2.4 with the `client.rack` property for consumers. Brokers must have rack awareness configured with `broker.rack` and `replica.selector.class` so that clients can discern where the closest replicas are.
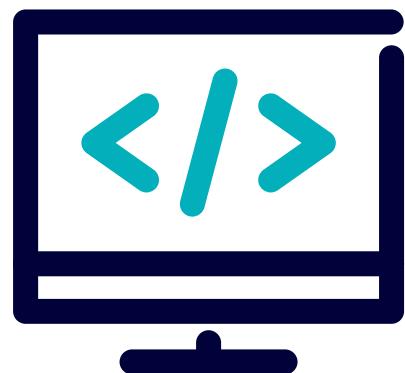
You may wonder whether producers also have a `client.rack` setting. They do not. Kafka's fault tolerance and strong ordering guarantees are due to the append-only nature of the log and the fact that producers write only to the leader. If producers could write to followers, then the leader and follower logs would diverge.

For more information, see [KIP-392](#).

Lab: Introduction

Please work on **Lab 1a: Introduction**

Refer to the Exercise Guide
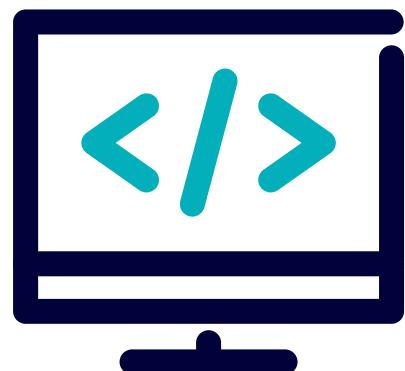
Lab: Using Kafka's Command-Line Tools

Please work on **Lab 1b: Using Kafka's Command-Line Tools**

Refer to the Exercise Guide

02: Starting with Producers

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

    a. What are the Basic Concepts of Kafka Producers?

    b. How Do You Write the Code for a Basic Kafka Producer?

Where this fits in:

- Hard Prerequisite: Fundamentals Course
- Recommended Prerequisite Module: Introductory Concepts
- Recommended Follow-Up Module: Preparing Producers for Practical Uses

02a: What are the Basic Concepts of Kafka Producers?

Description

Conceptually, basics of a producer. What is needed to specify a record. Objects one needs to instantiate to set up any producer. Basics of partitioning and serialization.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Be able to describe a basic producer

- List the most basic things needed to produce a message: key, value, topic

- Include and justify serialization in describing what a producer must do

- Given a hypothetical hash function, a number of partitions, and a key of a message, tell the index of the partition that will receive said message

- Note what three objects (language agnostic) one must instantiate to specify a producer

Specifying a Producer Record

- All records

  - Topic

  - Value

- Most records

  - Key

- Some records

  - Headers - custom metadata

  - Timestamp - override default

  - Partition - force a specific partition

All records need to go to a topic, so you must always specify that. All records need a value too.

You should have a key for most records—and before you start writing records without keys, you should think carefully about how keys may be used. However, if you don't supply a key, Kafka will fill it in as `NULL`.

The other three items are options. You don't really need to worry about them for now, but might find yourself needing to use them for certain use cases.
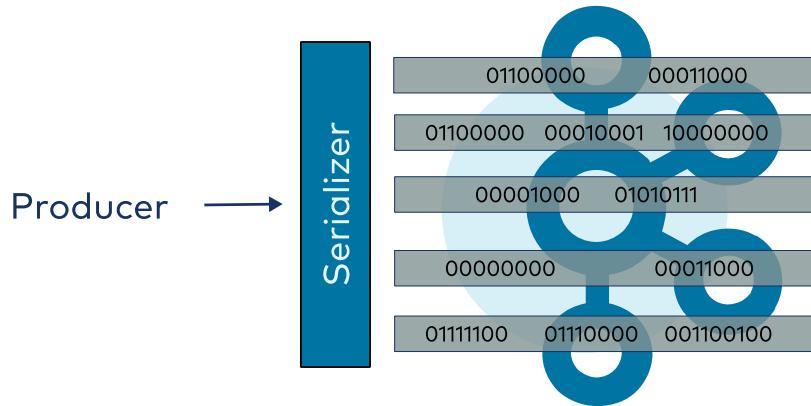
Core Objects for Setting Up a Producer

To specify a producer, you must instantiate each of the following:

Name	Example in One Client	Description
Configuration	`Properties`	A map of configuration settings to their values, e.g., bootstrap servers, serializers, client IDs, performance tuning settings
Producer Object	`KafkaProducer`	An abstraction of the producer itself
Record	`ProducerRecord`	An abstraction of an individual record, as per the previous slide

We'll see the specific code for this in the next lesson.

Serialization

- Kafka stores byte arrays

- Need byte arrays to send data across the network

- ... need to specify **serializer**

  ◦ Many built-in serializers
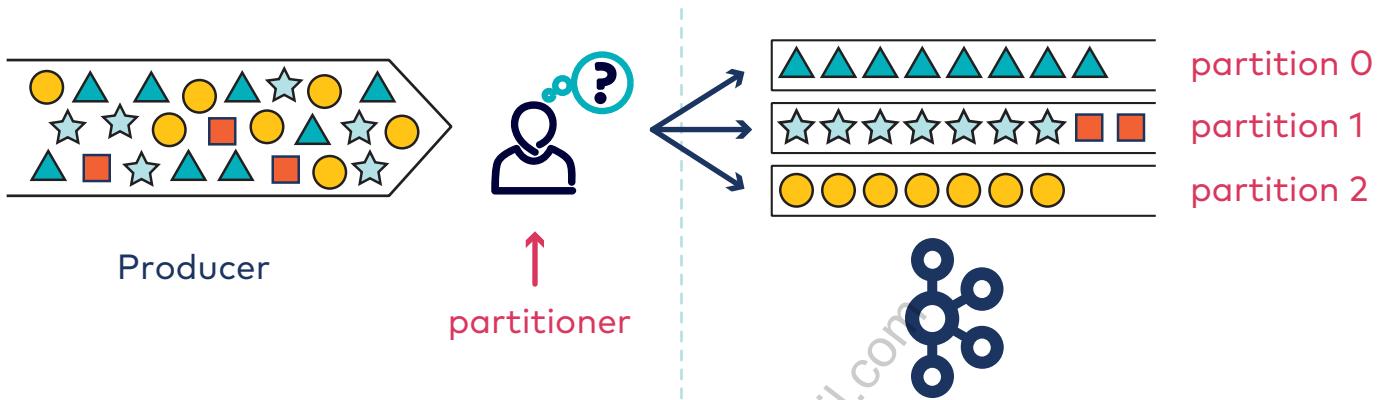
Both to send a message over the network and to store it in Kafka, it needs to be converted to zeroes and ones. A serializer is the tool that does this.

When you create a producer, you always need to specify what serializer to use. Many are provided for the standard kinds of data types. There are tools like Avro and Protobuf that can help with serialization for more complex objects; we learn about them in the Schema Management lessons.

Partitioning: Default

Partitions are indexed from 0 to `numberOfPartitions - 1`. When we have keyed messages...

$$partitionIndex = hash(key)\ \%\ numberOfPartitions$$

For now, we concern ourselves only with the default partitioner used when messages have keys.

When messages do not have keys, the Sticky Partitioner is used. This is described in [KIP (Kafka Improvement Proposal) 480](). In short, a random partition is chosen for messages, they accumulate in a buffer for that partition until batching requirements are satisfied, that batch is sent to Kafka, and then a new random partition is chosen and the process repeats. (Note that even with the default partitioner, batching happens; we leave the discussion of this important concept for a later lesson.)
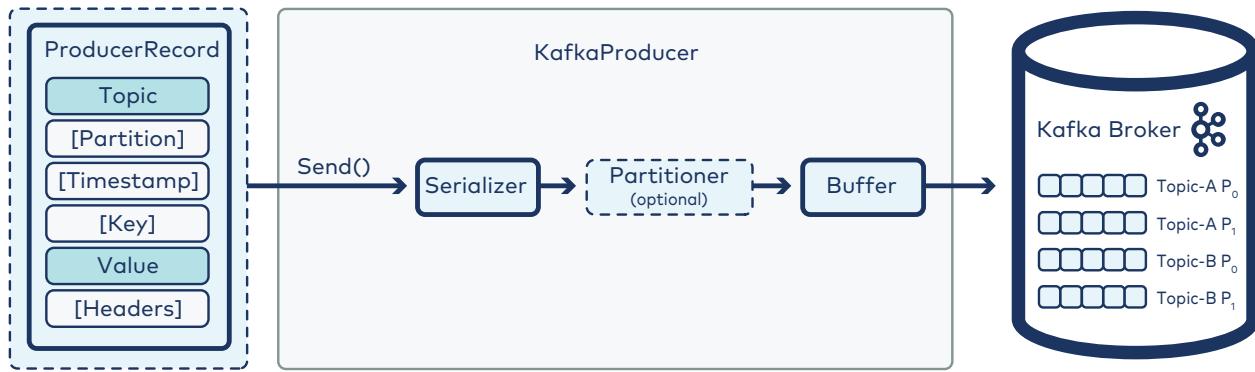
It is also possible to define a custom partitioner. Also, recall from a few slides back that you can bypass the partitioner entirely and specify a partition index. How to specify and use either of these less-common options is described in another lesson in this track.

> The `librdkafka` and Java clients use different defaults for key-based partitioning. The `librdkafka` clients use `consistent_random` which partitions using a CRC32 hash of key, whereas the Java client's `DefaultPartitioner` uses a murmur2 hash of the key. This can lead to situations where messages with the same key end up on different partitions, which has serious implications for downstream processing. If you must use both `librdkafka` and Java producers on the same topic, then configure the `librdkafka` client to use `murmur2_random`.

A Bigger Picture

There's more to come...

We've just discussed the very basics of producers here. This picture summarizes producer design we know so far.

Note that batching in standard on producers; we'll discuss that in a later lesson in this track. For now, note that messages go to a buffer after partitioning before being sent to brokers.

Further, note that things can go wrong; we have another module on how we can deal with that.

You'll see refinements of this picture as we go along.

To expand on the Producer Record on the left side... This is the data the producer wants to send to Kafka. Only the topic and value are required (pictured in darker blue). The other attributes are optional (pictured in lighter blue).

- Topic - Required. A topic name to which the record is being sent

- Partition - Optional. This is usually left to the partitioner to decide.

  ◦ If a valid partition number is specified, that partition will be used when sending the record.

  ◦ If no partition is specified but a key is present, a partition will be chosen using a hash of the key modulo the number of partitions in the topic.

  ◦ If neither key nor partition is present, a partition will be assigned using the StickyPartitioner.

- Timestamp - Optional. If a timestamp is not provided, the producer will stamp the record

with its current time. The timestamp eventually used by Kafka depends on the timestamp type configured for the topic.

- ◦ If the topic is configured to use CreateTime, the timestamp in the producer record will be used by the broker. This is the default behavior.

- ◦ If the topic is configured to use LogAppendTime, the timestamp in the producer record will be overwritten by the broker with the broker local time when it appends the message to its log.

- Key - Optional.

- Value - Required. The record contents.

- Headers - Optional. Key-value arrays.

Activity: First Impressions of Producers

Say you have just one minute to explain the basics of producers to a new teammate. What would you say?

Review the slides in this module as you answer this question.

02b: How Do You Write the Code for a Basic Kafka Producer?

Description

Turning a producer from the level of Producers Basic Concepts into code using the Java Client API. Quick overview of supported clients.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Configure properties required of al producers using the Java API

- Compare and contrast options for specifying properties

- Create producer and record objects using the Java API

- Send records to producers using the Java API

Basic Producer Properties

Name	Description
`bootstrap.servers`	Comma separated list of broker host/port pairs used to establish the initial connection to the cluster. Example: `kafka-1:9092, kafka-2:9092, kafka-3:9092`
`key.serializer`	Class used to serialize the key. Example: `StringSerializer.class`
`value.serializer`	Class used to serialize the value. Example: `KafkaAvroSerializer.class`
`client.id`	String to identify this producer uniquely; used in monitoring and logs. Example: `producer1`

- These are some of the producer configuration properties. These are the ones that are relevant so far. More will come in the Preparing Producers for Practical Uses module.

- While this lesson is nominally about the Java client API, these property names are universal.

- A key serializer must be specified even if you do not intend to use keys.

- Serializers must implement Kafka's `Serializer` interface.

- In the lessons on Schema Management, we'll learn about the Avro serializer, which can make your life very easy for serializing complex data types.

- More details on basic serializers:

  - `ByteArraySerializer`, `IntegerSerializer`, `LongSerializer`, and more are included in the client

  - `StringSerializer` encoding defaults to UTF8

    - Can be customized by setting the property `serializer.encoding`

Configuring a Producer in Java Code (1)

Instantiate and populate a **Properties** object:

```
1  Properties props;
2
3  props = new Properties();
4  props.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
5  props.put("key.serializer",
6            "org.apache.kafka.common.serialization.StringSerializer.class");
7  props.put("value.serializer",
8            "org.apache.kafka.common.serialization.KafkaAvroSerializer.class");
9  props.put("client.id", "my_first_producer");
```

Note that `Properties` is just a Java class; it is not specific to Kafka.

The `put` method expects two `string` arguments: a key and a value.

Configuring a Producer in Java Code (2)

Question: We know this code works to set bootstrap servers:

```
4 props.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

What about this code:

```
4 props.put("bootsrtap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

Will it compile?

Virtual Classroom Poll:

 it compiles

 it does not compile

Configuring a Producer in Java Code (3)

Question: We know this code works to set bootstrap servers:

```
4 props.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

What about this code:

```
4 props.put("bootsrtap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

Will it work?

Virtual Classroom Poll:

✅ it works   ❌ it does not work

Configuring a Producer in Java Code (4)

Wanted:

```
4  props.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

Typo - compiles but doesn't run:

```
4  props.put("bootsrtap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
```

Fix - **helper classes**:

```
4  props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-1:9092, kafka-2:9092,
   kafka-3:9092");
```

Helper classes just define some constants for config settings. If the settings aren't valid, you get compiler errors, which can save you debugging time. Plus, you can get help from most IDEs in suggesting names, so you don't need to look up the names of the options in documentation. (But you should use the documentation to be aware of what the configurations *are*).

Effectively, by using helper classes, we take what was a semantic mistake and turn it into a syntactic mistake. Developer tools can help us detect such mistakes more easily.

Configuring a Producer in Java Code (5)

We can also use a properties file to specify configuration:

```java
1  InputStream propsFile;
2
3  propsFile = new FileInputStream("src/main/resources/producer.properties");
4  props.load(propsFile);
```

In production, it is important to separate configuration from application logic so that the same logic can be run in many different environments with different properties. The typical way this is done in Java is with `.properties` files. Kafka's Java clients are no different.

A `.properties` file is a text file where properties are defined with the "=" symbol. Here is an example of a line from a properties file:

producer.properties

```
bootstrap.servers = kafka-1:9092, kafka-2:9092, kafka-3:9092
key.serializer = org.apache.kafka.common.serialization.StringSerializer.class
value.serializer = org.apache.kafka.common.serialization.KafkaAvroSerializer.class
client.id = my_first_producer
```

If you are use Confluent Cloud, you'll also need to include security settings in your properties file. Here's a partial properties file template:

```
bootstrap.servers=pkc-ep9mm.us-east-2.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 }}';
sasl.mechanism=PLAIN
```

Creating a Producer Object

We must create an instance of `KafkaProducer`, e.g.

```
KafkaProducer<String, MyObject> producer;
```

And we must pass the configuration `Properties` object to it during initialization:

```
producer = new KafkaProducer<>(props);
```

The arguments in `<..>` are the data types of the keys and values of messages, respectively, this producer will produce.

The name `props` is taken from the example a few slides back.

Creating and Sending a Record

We must create an instance of `ProducerRecord` and instantiate it with a topic, key, and value. Here's an example:

```java
ProducerRecord<String, String> record;

record = new ProducerRecord<String, String>("my_topic", "my_key", "my_value");
```

Then we tell the producer send it:

```java
producer.send(record);
```

We may not need to name the `ProducerRecord`, so you might, succinctly, do this:

```java
producer.send(new ProducerRecord<String, String>("my_topic", "my_key", "my_value"));
```

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

The `ProducerRecord` constructor encapsulates the provided key and value into a record (message) complete with headers. When `send()` is called, the `KafkaProducer` serializes the key and value and runs the default Partitioner (to determine which partition the message will store the message in the topic) using the data provided in the `ProducerRecord`.

Cleaning Up

- `producer.close();`

  Blocks until all previously sent requests complete

- `producer.close(Duration.ofMillis(…));`

  Waits until complete or given timeout expires

  Other `Duration` units are allowed

This can be done in a `finally` block with the core producer code in a `try` block. Here's the full producer application code with that idea:

```java
 1  Properties props;
 2  KafkaProducer<String, String> producer;
 3
 4  try
 5  {
 6      props = new Properties();
 7      props.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092, kafka-3:9092");
 8      props.put("key.serializer",
 9              "org.apache.kafka.common.serialization.StringSerializer.class");
10      props.put("value.serializer",
11              "org.apache.kafka.common.serialization.KafkaAvroSerializer.class");
12      props.put("client.id", "my_first_producer");
13
14      producer = new KafkaProducer<>(props);
15
16      producer.send(new ProducerRecord<String, String>("my_topic",
17                                                  "my_key", "my_value"));
18  }
19  finally
20  {
21      producer.close();
22  }
```

More Clients...

Here are the Kafka clients supported by Confluent:

JVM	librdkafka (C library)

💡	For other languages, consider the REST Proxy.

This is a summary of all clients supported by Confluent.

Many more clients are available from members of the Kafka community.

Java is a first class citizen in the Kafka ecosystem, so these course materials will focus mainly on Java. All of the JVM languages listed use Kafka's Java API directly.
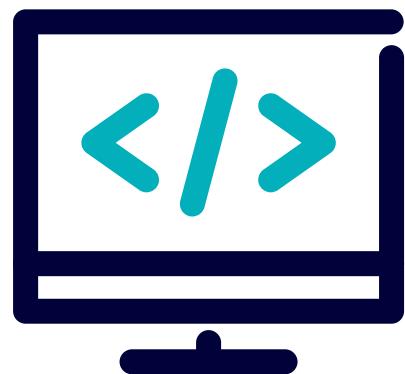
The other family of clients are derived from a C library called `librdkafka`.

See the clients documentation for a list of features supported by each client.

While these are the main clients that Confluent supports, there are many other clients created by various members of the Kafka community. For example client code in many languages (C, Clojure, C#, Go, Groovy, Java, Kotlin, NodeJS, Python, Ruby, Rust, Scala, and more), see https://github.com/confluentinc/examples/tree/5.5.0-post/clients/cloud.

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!
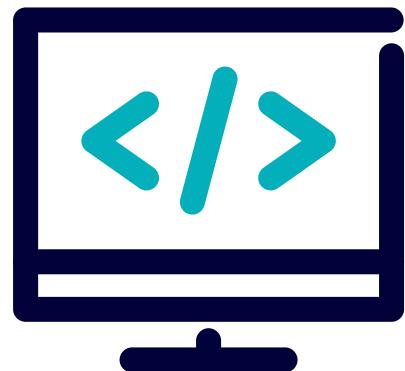
Java is bolded since this is the language that trainers will be able to support during class.

Trying the exercises in multiple languages may help distinguish what is Kafka specific vs. what is language specific. You are also encouraged to inspect the NodeJS client code!

Lab: Basic Kafka Producer

Please work on **Lab 2a: Basic Kafka Producer**

Refer to the Exercise Guide

03: Preparing Producers for Practical Uses

Module Overview

This module contains three lessons:

    a. How Can Producers Leverage Message Batching?

    b. How Do Producers Know Brokers Received Messages?

    c. How Can a Producer React to Failed Delivery?

Where this fits in:

- Hard Prerequisite: Starting with Producers

- Recommended Follow-Up: Starting with Consumers

03a: How Can Producers Leverage Message Batching?

Description

The motivation for batching on producers, configuration of batching and buffers, and understanding the shared buffer and implications. Setting compression on batches.
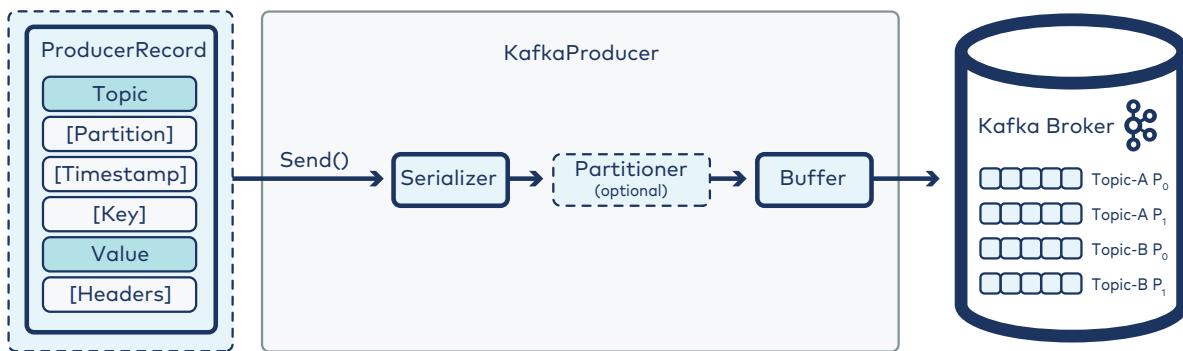
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe how messages go to buffers upon sending

- Describe what causes messages to leave buffers and get sent, explicitly noting two configuration settings

- Configure the size of the shared buffer

- Describe what could cause a send to fail—full buffer—and how to use configuration to deal with that

Producers So Far...

Let's go back to this picture...

Let's now see what's going on with the buffers...

When we use `KafkaProducer` to `send()` a message, the message is serialized and then (likely) run through the partitioner to decide which partition of the topic will receive the message. But it doesn't go directly to Kafka. Now let's see what happens first...

Diversion

Say it takes you 10 minutes to get to your nearest grocery store.

Say you need three things:

- ice cream
- cereal
- cheese

You could...
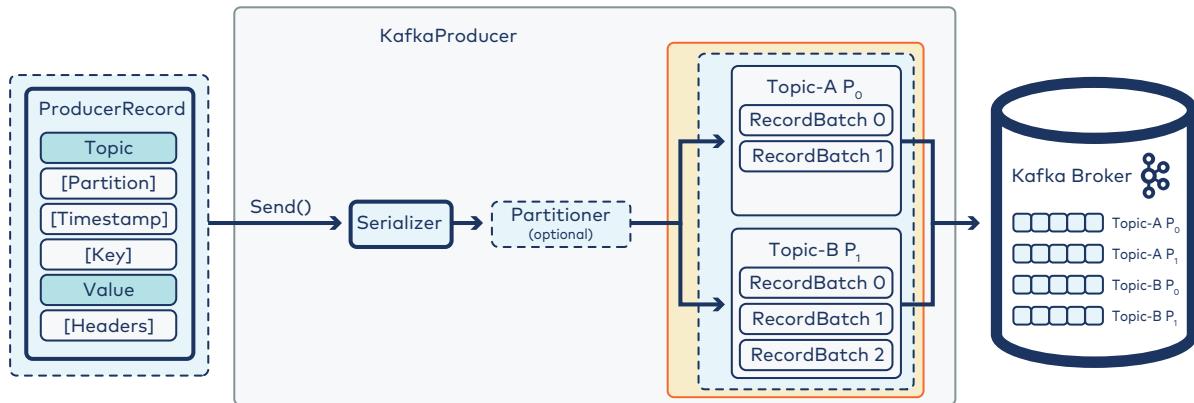
1. Go to the store, buy the ice cream, go home, and put the ice cream away.

2. Go to the store again, buy the cereal, go home, and put the cereal away.

3. Go to the store again, buy the cheese, go home, and put the cheese away.

Thoughts?

Big takeaway: This process involves 60 minutes of travel time. We could have gotten everything at once and done this in 20 minutes of travel time. In Kafka, we often want to send messages in **batches**, rather than one at a time. This generally gets us better performance and certainly makes better use of shared resources, e.g., the network.

Refining Our Producer Design Picture

ℹ️	Messages are stored in the buffer **per topic-partition**.

Hopefully you see the value in batching messages rather than sending each message one at a time.

We see here a more-detailed version of the earlier picture. The producer has space allocated for buffering. In there, messages are accumulated per topic-partition. Accumulation continues until some criterion is met to cause a batch of messages to be sent.

Back to that Grocery Store Diversion

- Before:

  - Grocery store 10 minutes away

  - Needed ice cream, cereal, cheese

  - We agreed we could batch, i.e., get all three at once

- Clarifying constraint:

  - Walking to store

  - Carrying everything

- Now: Additional things to buy:

  - 5 cases of soda

  - Bulk pack of 10 rolls of paper towels

  - Bulk pack of 8 boxes of tissues

Now what? Same batching?

We decided earlier batching is good, but... not so fast. Can you carry all of this stuff? Probably not. Should you send a batch that's really big? No. You can compromise shared resources and hurt performance. We need to pick an appropriate batch size.

What Do You Care About?

- Throughput?
  - Want to send many messages at once
- Latency?
  - Want to have messages consumed as quickly as possible after they are produced
- We can specify
  - How big batches can get
  - How long batches can accumulate
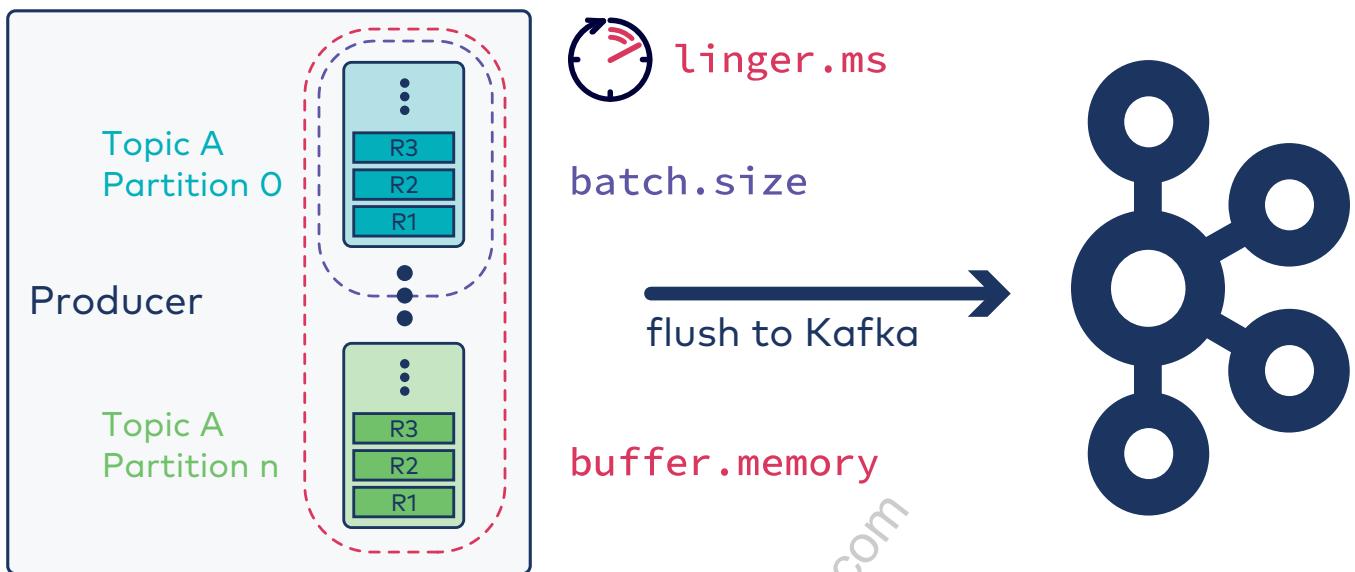
ℹ️	Balance! Probably we care about both matters

So, batching is good in general, but we don't want to let batches get too large. In deciding appropriate settings, it's likely throughput or latency your organization is looking to optimize.

Visualizing the Buffer

A few slides back, we saw the shared buffer. Here we see for the first time `buffer.memory`, the size of that shared buffer.

We also see the two thresholds that, when met, can cause the buffer to be flushed.

See the last slide of this lesson for spelled-out definitions of these settings and their defaults.

Not So Fast...

- `send()` returns when a message has been added to the buffer

- The shared buffer is of size `buffer.memory`

- What if, when we send, there is not enough room in the buffer? Are we doomed?

Memory is finite, so we have to consider the size of our shared buffer in thinking about what happens when we send a message.

We're Okay After All! (Maybe)

- `send()` returns when a message has been added to the buffer

- The shared buffer is of size `buffer.memory`

- What if, when we send, there is not enough room in the buffer? Are we doomed?

  - **NO**!

  - Maybe a batch accumulating for some partition is big enough to meet `batch.size` or `linger.ms` is flushed and space is freed

- Config setting `max.block.ms` puts a limit on how long a producer will wait for there to be space in the buffer before a `send()` fails.
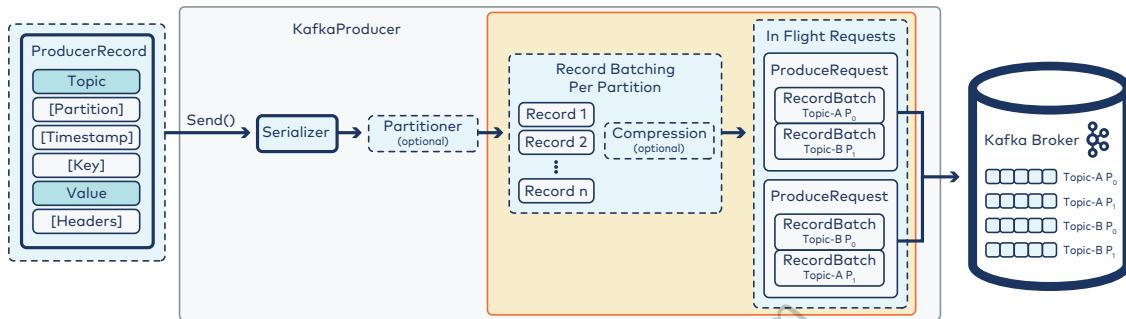
If our shared buffer is full, we cannot write to it. A producer will wait up to `max.block.ms` for some space to free up in the buffer. If this time passes, the `send()` will fail.

Note that we say here `send()` returns when a message has been added to the buffer, in other words, `send()` is asynchronous. But, if you'd like to send synchronously, i.e., force `send()` to block until the producer learns of success or failure (more in the next lesson on how to do that), append `.get()` to the call, e.g. `producer.send().get()`.

Compression 101

- It is possible to turn on compression on producers

- Records are sent as compressed batches

New picture:

Compressing records can help save network utilization and disk space on the brokers.

But, compression is not free; there is a time cost associated with compression that could compromise latency. We discuss the tradeoffs and how to make decisions in our Confluent Advanced Optimization course.

Compression algorithm choices supported by Kafka are `none`, `snappy`, `gzip`, `lz4`, `zstd`.

Compressed batches of records get stored on the brokers and decompressed by the consumers.

Summarizing new Configurations

Name	Description	Default
`batch.size`	Minimum number of bytes needed to accumulate in a batch for it to be considered complete and ready to send.	`16384`
`linger.ms`	Maximum time a batch will wait to accumulate before sending.	`0`
`buffer.memory`	Maximum size of the producer's buffer, shared across all partitions.	32 MB
`max.block.ms`	How long producer will wait for a full buffer to have free space before failing a `send()`	1 min
`compression.type`	How data should be compressed. Values are `none`, `snappy`, `gzip`, `lz4`, `zstd`. Compression is performed on batches of records.	`none`

This slide gives the four new settings introduced in this lesson.

Kafka: The Definitive Guide is a great resource. Note that there's a nice explanation on p. 5 on batching.

Activity: Applying Batching Configurations

Consider the following settings:

- `batch.size`
- `linger.ms`
- `buffer.memory`
- `max.block.ms`

Which setting(s) would be important to adjust to accomodate each of the following scenarios?

a. You want high throughput, but don't care about latency

b. Latency is a major goal but we don't want to forget about throughput entirely

c. You are finding the overall buffer gets full a lot in (a)

03b: How Do Producers Know Brokers Received Messages?

Description

The various levels of producer acknowledgements and implications of each. Using callbacks in producer code to react to acks.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- List the 3 different values of `acks` and describe when to use each

- Explain how `acks=all` could be equivalent to `acks=1` and what to do about that

- Take code for a producer that does not look for acknowledgements and make it, i.e. add a Callback

- Explain the tradeoffs of the 3 levels of `acks`

The Notion of an Acknowledgment

A producer might want to know if a request was successfully delivered to the Kafka cluster.

We add the following configuration setting to our list of producer properties:
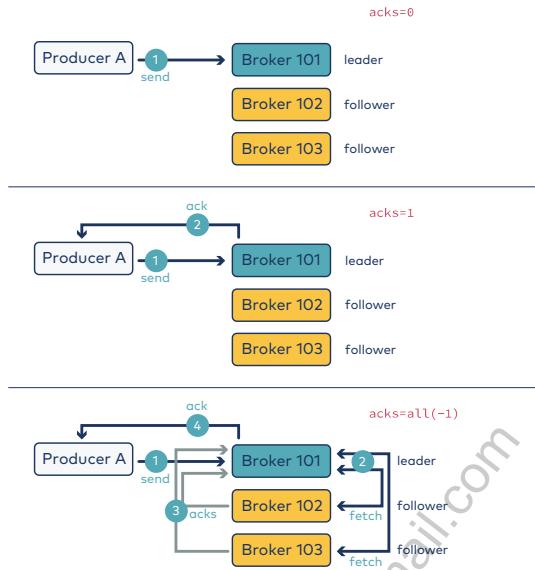
Name	Description
`acks`	Used to determine when a write request is successful. Can be 0, 1, or all (-1). If `acks` is not zero, then the producer will retry failed requests. Default: `all`

Note that the default for `acks` was changed to `all` in Apache Kafka 3.0, but had been `1` in all prior versions.

Details on this to come in the coming slides.

acks - Three Cases, Ideal Performance

We have three choices for the `acks` setting:

- `0` means the Kafka cluster does **not** communicate back to the producer whether a message has been received.

- `1` means that once the leader has persisted the message, it communicates an acknowledgement back to the producer.

- `all` means that once leader **and** all followers have persisted the message, the leader communicates an acknowledgement back to the producer.

Put differently, `1` means that an ack is sent after record is stored in "1" member of the ISR, whereas `all` means that an ack is sent after the record is stored in "all" members of the ISR.
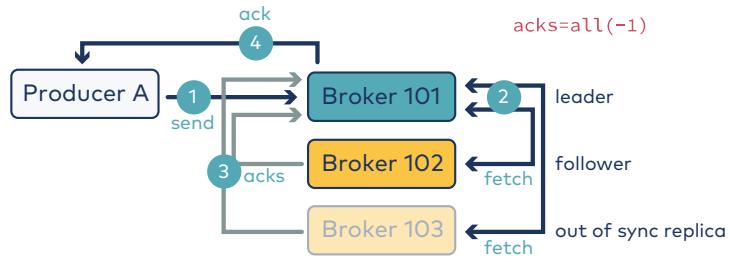
This loosely relates to message delivery guarantees. Note that there are three cases:

- At most once: Messages may be lost but are never redelivered.

- At least once: Messages are never lost but may be redelivered.

- Exactly once: this is what people generally want; each message is delivered once and only once.

There is a later full lesson on delivery guarantees in the Advanced Concepts branch of the course.

…But not all followers are in-sync…
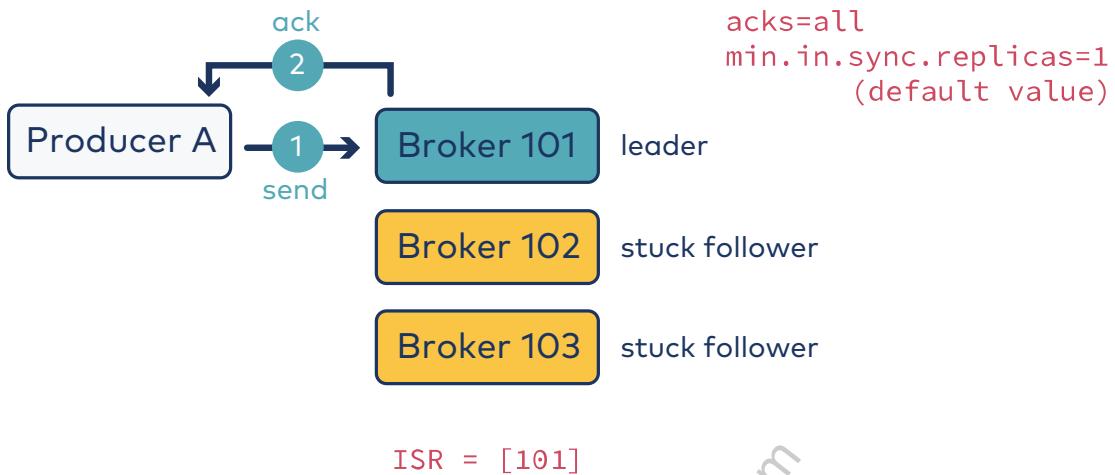
So… when the leader gets a new message and `acks=all`…

1. The leader notes which followers are **in sync** with the leader at the time it receives the message

2. Followers fetch from the leader and send acks to the leader

3. When the leader receives acks from all of the followers from (1), it sends an ack to the producer

---

If we took the `acks = all` requirement from the last slide literally, Kafka would require stuck followers not only to get the new message, but also catch up on prior messages in order to satisfy the `acks` request. Instead, Kafka does **not** require stuck followers to catch up to satisfy `acks = all`; the new message must only be received by followers that were in-sync with the leader at the time the leader received the new message.

What if We Don't Have Any In-Sync Followers?


```
ack
 2
```

Producer A — **1** → Broker 101   leader
send

Broker 102   stuck follower

Broker 103   stuck follower

ISR = [101]

```
acks=all
min.in.sync.replicas=1
        (default value)
```
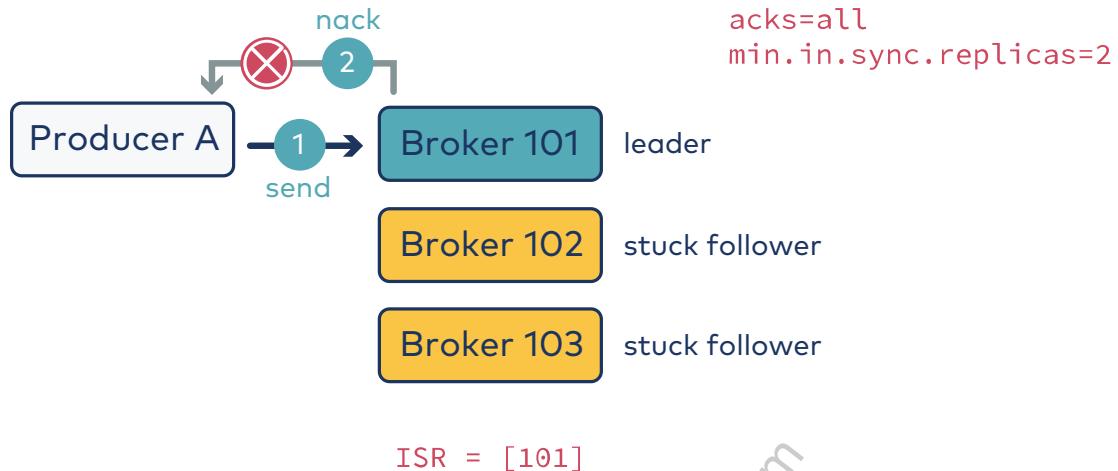
Let's be literal about what `acks = all` requires:

1. The leader must persist the new message

2. All followers that were in sync with the leader at the time the leader received the message must also receive the message.

What if no followers were in sync with the leader? Then the second condition is vacuously true and `acks = all` is met. But if one has set `acks = all`, that means the producer wants certainty that the message has been delivered to one or more followers. This isn't so good...

Guaranteeing Meaningful **acks=all**

...so we'd often like `acks = all` to be stronger and only be met when a new message has made it to at least one follower. We can strengthen it by setting `min.in.sync.replicas` to 2 or greater. (Remember, the leader always counts as one in-sync replica, so this value must be strictly greater than 1.)

Also, note that while `acks` is a producer setting, `min.in.sync.replicas` is a broker setting.

What Happens When Producers Send? [A Review]

- When we call `send()` on a producer...

  - Message is serialized

  - Message runs through partitioner (in most cases)

  - Message is written to the buffer

- `send()` call returns once the message has been written to the buffer

  - Producer does not wait to find out if Kafka brokers have received the message

  - More code, likely more `send()` calls, runs (i.e., producer does not block further execution) as a batch may get more messages after the `send()` has returned

  - Maybe we want to know if a `send()` was successful...

It is possible to add `.get()` to the `send()` call, e.g.,

```
producer.send().get();
```

The `send()` call returns an instance of `Future`. The method above causes the `Future` to be blocked, effectively making it so that one record is being sent at a time.

While you could work with `Future`s directly, the next slide provides a cleaner solution.

Callbacks

- We can provide a `Callback` as the second argument to a `send()` call

- The `Callback` is an interface

- The key method is `onCompletion`:

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

- Typically, `onCompletion` is implemented with a lambda expression

- Parmeters of `onCompletion`:

  ◦ Send in uninitialized objects

  ◦ `exception` is `null` when we have success

    ▪ Test for this first

ℹ️	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.

So, how can we as a producer do something with acknowledgements? The answer is to provide a Callback.

The next slide gives an example of this in code.

⚠️	Note that some older documentation may say `metadata` is `null` in the case of an exception, but that is not correct. Always test for the `exception` being `null` first.

send() with Callback Example

```java
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 });
```
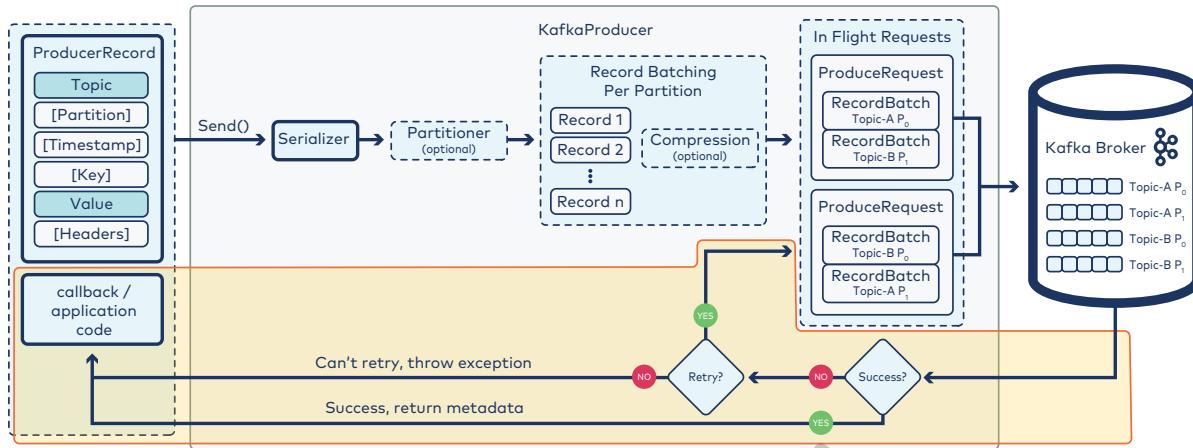
This code shows how to equip `send()` with a `Callback` to receive an acknowledgment, or lack thereof. Notes:

- Since the `Exception` will be `null` when we get a successful acknowledgment, we should test for that first. In this case, we look for the inverse condition in Line 2 and print a report in Line 3.

- Lines 5-6 handle the successful case. In this case, we just show some information about the delivery.

Note that we have more to say about failed records. There is a retry mechanism and timeouts. If the send of a record fails the first time, that does not mean the Callback—and specifically the `Exception` branch of it—will get activated; there will likely be retries. The next lesson is all about retries and timeouts.
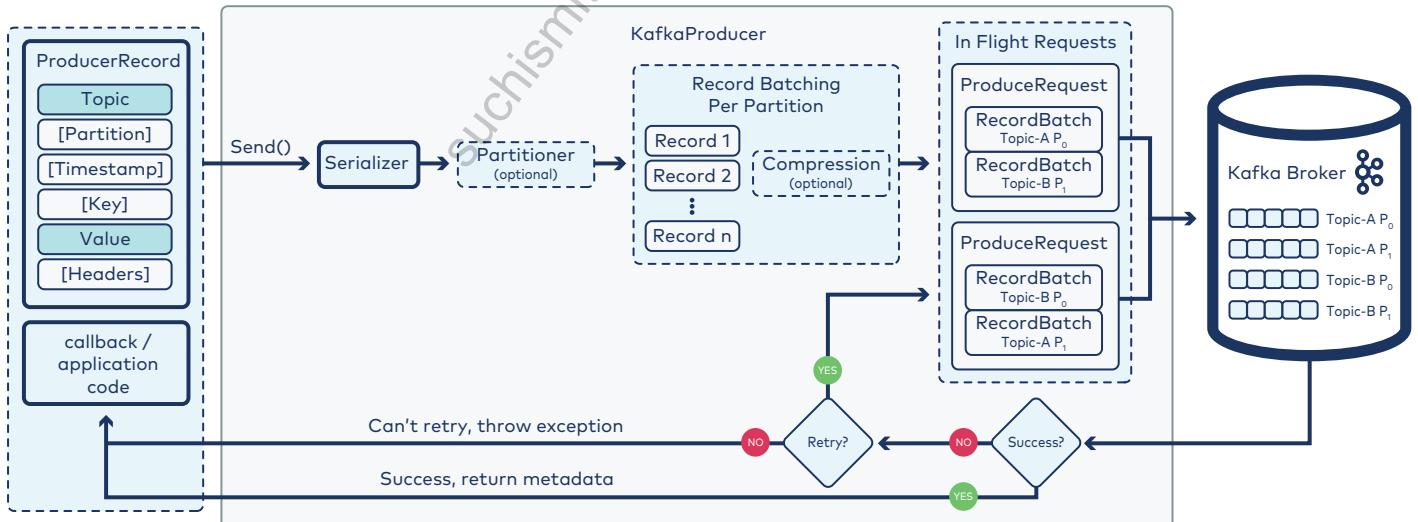
Refining Our Producer Design Picture

Here we come back to the producer design illustration we've been building up. Note the addition of the check for success.

Note also that if retries are enabled and we've not timed out (more on that in the next lesson), the producer will retry before going to the Callback with an exception.

Here's a final version of the image:
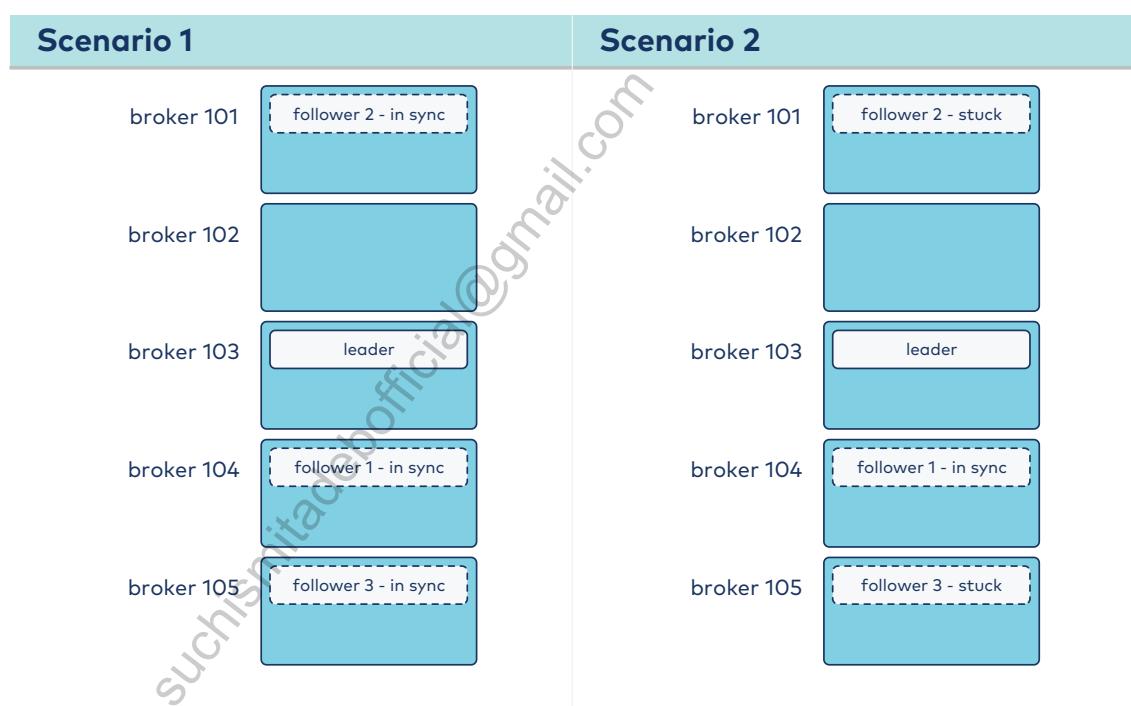
Activity: Tracing Message Writes with Acks

Discuss each scenario with a partner or a small group before we discuss as a class.

Suppose a producer produces a new message to the partition shown. Describe a possible sequence of what could happen from the moment the cluster receives the message until the producer receives an acknowledgment

1. when `acks = 1`

2. when `acks = all`

Scenario 1	Scenario 2
broker 101 follower 2 - in sync	broker 101 follower 2 - stuck
broker 102	broker 102
broker 103 leader	broker 103 leader
broker 104 follower 1 - in sync	broker 104 follower 1 - in sync
broker 105 follower 3 - in sync	broker 105 follower 3 - stuck

03c: How Can a Producer React to Failed Delivery?

Description

Retrying failed messages, different components of time from send until delivery and configuration, and best practices.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe what happens when a message fails to be accepted by a broker.

- Explain some timeout settings that can be tuned for producers and decide which might affect various real-life scenarios.

- Make decisions about reacting to failed delivery in line with Confluent's recommended best practices.

Big Picture

- We can configure producers with `acks = all` or `acks = 1` to make the Kafka brokers respond to the producer know if messages make it successfully.

- What if a message fails?

  ◦ Give up?

  ◦ Try again?

  ◦ How long to keep trying?

In this lesson, we will go deeper into what happens when a producer finds out a message send attempt fails.

Retries and Timeouts

- We can configure a number of retries a producer has

  - Setting `retries` with default `MAX_INT`

- We can also limit how much time a producer spends waiting overall at at various stages after sending

- **Best practice**: Limit delivery with timeouts, not number of retries

---

We can control retry behavior through number of retries or through timeouts.
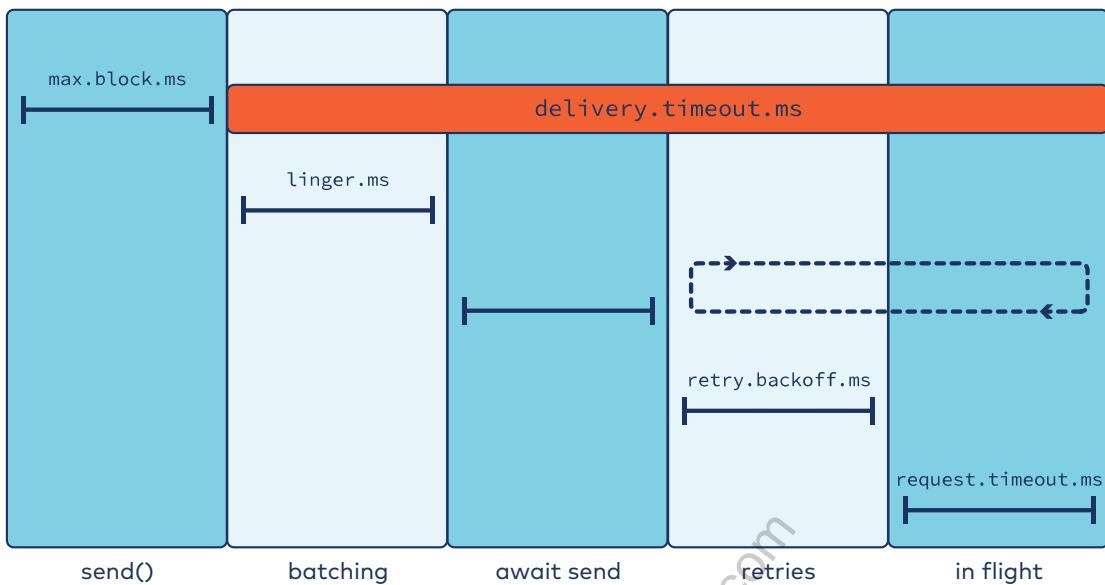
Some Time Limits

Name	Description	Default
`max.block.ms`	An upper bound on the time to wait for a buffer that doesn't have enough room to accept a new message to gain more space.	1 min.
`linger.ms`	Time a batch will wait to accumulate before sending.	`0`
`request.timeout.ms`	An upper bound on the time a producer will wait to hear acknowledgments back from the cluster.	30 sec.
`retry.backoff.ms`	How much time is added after a failed request before retrying it.	`100`
`delivery.timeout.ms`	An upper bound on the time to report success or failure after a call to `send()` returns. Use this to control producer retries.	2 mins.

You can review [a larger list of producer configurations on our web site](#).

You'll do an activity using this information in two slides. This slide is intended as reference for that activity and beyond.

Visualizing Those Times
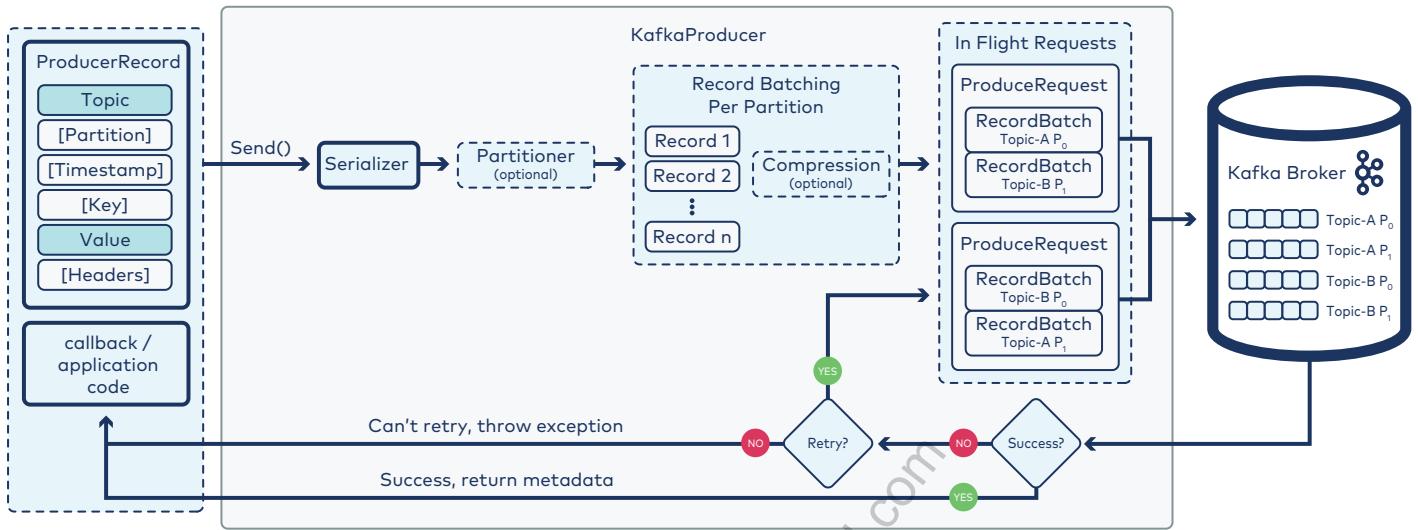
Activity: Interpreting and Applying Time Limits

Assume you're working with all settings starting at their defaults. Work with a classmate or small group to interpret the configurations on the last few slides and answer these questions:

1. Someone reports to you that while `batch.size` is set rather high, only one message is ever being sent at a time. Batching never occurs. What setting can we change to fix this?

2. You have messages that are extremely time sensitive. No matter what happens, if they don't make it to the broker within 30 seconds of `send()` returning, there's no point. How can you enforce this?

3. Suppose you fixed the last problem correctly and have also implemented a callback, but in fact, some messages don't fail until 90 seconds after the producer tries to `send()`. Where could this extra time be coming from?

Summarizing Producer Design

This figure brings together everything about producer design.

Here are a few other miscellaneous notes:

- For more details about the system metadata included in RecordBatch and Record objects, see https://kafka.apache.org/documentation/#messageformat.

- The "await send" portion of the diagram two slides back reflects the fact that the batch has to wait for a transmission opportunity to the broker. A ready batch can only be sent out if the leader broker is in a sendable state (i.e., if a connection exists, current inflight requests are less than max.inflight.requests, etc.).

- For more details on delivery timeout, see KIP 91 at https://cwiki.apache.org/confluence/display/KAFKA/KIP-91+Provide+Intuitive+User+Timeouts+in+The+Producer

04: Starting with Consumers

CONFLUENT
Global Education

Module Overview

This module contains three lessons:

a. How Do You Request Data to Fetch from Kafka?

b. What are the Basic Concepts of Kafka Consumers?

c. How Do You Write the Code for a Basic Kafka Consumer?

Where this fits in:

- Hard Prerequisite: Starting with Producers

- Recommended Prerequisite: Preparing Producers for Practical Uses

- Recommended Follow-Up: Groups, Consumers, and Partitions in Practice

04a: How Do You Request Data to Fetch from Kafka?

Description

The consumer parallel of producer batching. Configuring consumer properties affecting fetch requests. Configuring broker properties that affect fetch requests from followers of leaders.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe the options for tuning a fetch request from a consumer

- Describe the options for tuning a fetch request from a follower

- Relate consumer and follower fetch requests

- Relate fetch request decisions with optimization goals, i.e. throughput and latency

Fetching Data: Quick Overview

- Data has been produced to brokers

- Who fetches it?

  ◦ Consumers

  ◦ Followers

We've spent the last several lessons talking about getting data into Kafka - specifically onto the brokers - via producers. So, we now turn to how the data gets out. You might expect the answer to be consumers, and that will be our focus, but along the way, we'll note how some ideas central to consumers are used by other aspects of Kafka.

The first thing we need to look at is the idea of a broker being asked for data. This is called fetching. We'll look into how to configure a fetch request. A consumer could make a fetch request—from the leader replica of a partition. So could a follower replica, also making the request of the leader replica of the same partition.

Back to Producers

Recall:

- A producer `send()` call writes a message to a local buffer

- Buffer accumulates batches of messages per partition

- A batch is sent from the producer to the cluster when it

    ◦ Reaches a size threshold

    ◦ Reaches a time threshold

Fetch requests generally work with batches of messages too

You may recall from Fundamentals and our warm-up discussion that consumers subscribe to topics and that consumers make pull requests, asking Kafka for messages (rather than Kafka pushing messages). When producers sent messages to Kafka, they were sent in batches for efficiency. So, it stands to reason that the same happens when consumers are asking for records or followers are asking for records. Either requests a batch, and not just a single record, in general. We can tune these requests, as we will see in the slides to come.

How to Control Fetch Requests

size of data in a fetch   **time** to wait for a fetch

These are the two general ways we can tune fetch requests.

Consumer Fetch Request Settings

minimum size of data in a fetch	maximum amount of data in a fetch	maximum time to wait for a fetch
`fetch.min.bytes`	• `max.partition.fetch.bytes` - size per partition • `max.poll.records` - # records across all partitions	`fetch.max.wait.ms`

Here we give the names of the specific properties we can set to tune fetch requests.

Consumer properties are set in consumer code, directly or indirectly. We'll see how in a later lesson in this module.

Defaults:

• `fetch.min.bytes`: 1 byte

• `max.partition.fetch.bytes`: 1 MB

• `max.poll.records`: 500 records

• `fetch.max.wait.ms`: 500 ms

This table transposes and expands on the one on the slide, bringing in the equivalent follower settings too:

Fetch Request Settings

...	Consumer	Follower
minimum size of data in a fetch	`fetch.min.bytes`	`replica.fetch.min.bytes`
maximum amount of data in a fetch	• `max.partition.fetch.bytes` - size per partition • `max.poll.records` - # records across all partitions	-
maximum time to wait for a fetch	`fetch.max.wait.ms`	`replica.fetch.wait.max.ms`

Replica settings are set on the `server.properties` file on each broker.

Tuning Consumer Fetch Requests

Goal	High Throughput	Low Latency
Meaning	• Get more records at once	• Consume records as quickly as possible after they were produced
Tune...	• Large `fetch.min.bytes` • Reasonable `fetch.max.wait.ms`	• Low `fetch.max.wait.ms`

💡	Often a balancing act

This is specific to consumers and not about followers.

What is our goal in terms of performance? Here we give general tips on meeting it. (Naturally, these recommendations are qualitative. The specific numbers that work best depend on the particular use cases and desires, and one could spend a lot of time studying and experimenting with these settings in a given context.)

Activity: Applying Fetch Settings to a Scenario

Back to a scenario from the producer end. Say you want high throughput and don't care about latency. To achieve this on the consumer end, which settings would you adjust and how (qualitatively)?

04b: What are the Basic Concepts of Kafka Consumers?

Description

Conceptual view of a basic consumer.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Relate what is necessary to set up a basic producer to what is necessary to set up a basic consumer conceptually

- Highlight where topics fit in on both extremes of a message's life cycle

Reviewing Producers

What did you need to do to set up a basic producer and send a message?
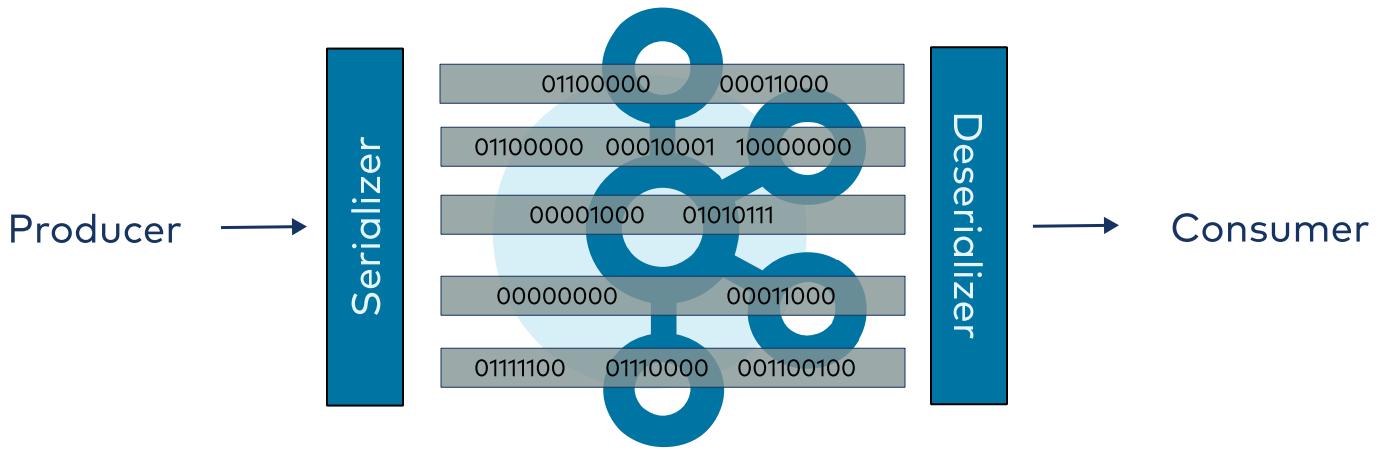
Parallel Ideas with Consumers

Producer	Consumer
Configure properties	Configure properties
Create producer	Create consumer
Send records	Retrieve & process records
Close & clean up	Close & clean up

If we look at all of the big parts of coding a basic producer, we can see that each either appears identically on the consumer end or with a parallel.

One Config Note: Deserialization

Producer → Serializer | 01100000   00011000 | 01100000   00010001   10000000 | 00001000   01010111 | 00000000   00011000 | 01111100   01110000   001100100 | Deserializer → Consumer

Remember that on the producer end, we had to convert messages to 0s and 1s, a.k.a. serialization. Since messages were stored in Kafka serialized and we receive them that way on the consumer end, we need to **deserialize** now. Like on the producer end, this is just a matter of setting the correct deserializer.

How Do Consumers Know What Messages to Read?

You must take action	Handled automatically by Kafka/Consumer API
• subscribe to topic(s)	• choose partition to read from
	• maintain consumer offset in partition
	More on both of these in the recommended next module

On the producer end, we specified the topic to which to write with each record. On the consumer end, consumers subscribe to topics. A need step in defining a consumer is thus to make that subscription.

That's all you need to do as a developer for a basic consumer!

Of course, each topic is (likely) broken up into partitions and each partition consists of several messages (with offsets associated). Which partition and which message(s) in it are important things, but Kafka will take care of all of that routing for you. We'll learn more about how it all works in the next module.

Polling and Processing

- Consumers **poll** the cluster for messages

  - i.e., fetch request

- Consumers get batches of records back to **process**

- Repeat this process indefinitely

On the producer side, we had to send a message. On the consumer side, we must poll Kafka. This generates a fetch request.

Then we get back, potentially, a batch of records. We must process them.

After setup, a consumer will be running an infinite loop of polling and processing. That's it!

Summary

Producer	Consumer
Configure properties	Configure properties
Create producer	Create consumer
(n/a)	Subscribe to topics
Prepare records	(n/a)
Send records	Poll for records
(n/a)	Process records
Close & clean up	Close & clean up

So, here we bring everything together!

In the next lesson, we'll look at some specific code.

04c: How Do You Write the Code for a Basic Kafka Consumer?

Description

Coding a basic consumer using the Java client API.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Configure properties required of all consumers using the Java API

- Create a consumer and work with consumer record objects using the Java API

- Subscribe to topics in Java consumers

- Retrieve and process messages using the Java API

Configuration (1)

Important properties:

Name	Description
`bootstrap.servers`	List of broker host/port pairs used to establish the initial connection to the cluster
`key.deserializer`	Class used to deserialize the key. Must implement the `Deserializer` interface
`value.deserializer`	Class used to deserialize the value. Must implement the `Deserializer` interface
`client.id`	String to identify this consumer uniquely; used in monitoring and logs

All of these properties are like those we saw on the producer end. See the Producers Code Basics lesson for more details if you need them.

As with the producers, even if the key is not used by your environment, the clients must be configured as if you are using keys. Thus, `key.deserializer` is a required setting.

Configuration (2)

We specify properties in code just like with producers.

We need a `Properties` object:

```
1 Properties props;
```

Here we show using a helper class:

```
6 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-1:9092, kafka-2:9092,
  kafka-3:9092");
7 // other properties
```

You can specify properties with helper classes, or hardcode the string names, or read from a properties file—just like with producers. Revisit your notes from the Producers Code Basics module on the pros and cons of each.

Creating a Consumer

We need a `KafkaConsumer` object:

```
2 KafkaConsumer<String, String> consumer;
```

Initialization is just like with a producer:

```
10 consumer = new KafkaConsumer<>(props);
```

This part is almost the same as on the producer side; the name is all that's different.

Subscribing to Topics

This is an additional step:

```
14      consumer.subscribe(Arrays.asList("my_topic", "my_second_topic"));
```

Note that calling `subscribe` again **replaces** an existing topic list.

We noted in our conceptual overview we must also subscribe to a topic or topics. Note and emulate the syntax. Typically, you do this right after constructing your consumer, but it's possible for a consumer to change its subscription, so this can happen later too.

An important note is that calling `subscribe` is not additive.

Note also that you could use regular expressions to specify topics.

Consumer Record and Polling

We need an object to hold what we get back:

```
3  ConsumerRecord record;
```

Then we call `poll()`:

```
18         record = consumer.poll();
```

Let's build up that big part of consumers where we poll and process...

Consumer Records & Processing

But, remember, records are batched and we must process the batches so...

```
4 ConsumerRecords records;
```

We must process the batch of records we receive:

```
18        records = consumer.poll();
19
20        for(record : records)
21        {
22            System.out.printf("offset: %d, key: %s, value, %s\n",
23                              record.offset(), record.key(), record.value());
24        }
```

Since a fetch request gives back batches (which could be a single record or be empty), we need a `ConsumerRecords` object to hold our result, not just a single `ConsumerRecord`.

Then we process the records. Here we just output details.

Consumers Run Indefinitely

```
16    while(true)
17    {
18        records = consumer.poll();
19
20        for(record : records)
21        {
22            System.out.printf("offset: %d, key: %s, value, %s\n",
23                              record.offset(), record.key(); record.value());
24        }
25    }
```

Since a consumer's job, after setup, is to poll and process indefinitely, we simply loop what we did on the last slide forever.

Timeout on Polling

Let's tweak the poll:

```
16    while(true)
17    {
18        records = consumer.poll(Duration.ofMillis(100));
19
20        for(record : records)
21        {
22            System.out.printf("offset: %d, key: %s, value, %s\n",
23                              record.offset(), record.key(); record.value());
24        }
25    }
```

We should also provide a timeout on how long to wait, at most, for a batch. We do so with a `Duration` object as an argument to `poll()`.

Cleaning Up

Like with producers, we must `close` to clean up open resources:

```
29    consumer.close();
```

We could supply a timeout, as with the producer:

```
consumer.close(Duration.ofMillis(100));
```

More on `close` and timeouts:

- [The documentation](#)

- [The KIP proposing timeouts](#)

Coding Safely and Putting It All Together

To do this safely, we should employ exception handling. Here's the full consumer:

```java
 1 Properties                      props;
 2 KafkaConsumer<String, String> consumer;
 3 ConsumerRecord                  record;
 4 ConsumerRecords                 records;
 5
 6 props = new Properties();
 7 props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
 8 // other properties
 9
10 consumer = new KafkaConsumer<>(props);
11
12 try
13 {
14     consumer.subscribe(Arrays.asList("my_topic", "my_second_topic"));
15
16     while(true)
17     {
18         records = consumer.poll(Duration.ofMillis(100));
19
20         for(record : records)
21         {
22             System.out.printf("offset: %d, key: %s, value, %s\n",
23                             record.offset(), record.key(); record.value());
24         }
25     }
26 }
27 finally
28 {
29     consumer.close();
30 }
```
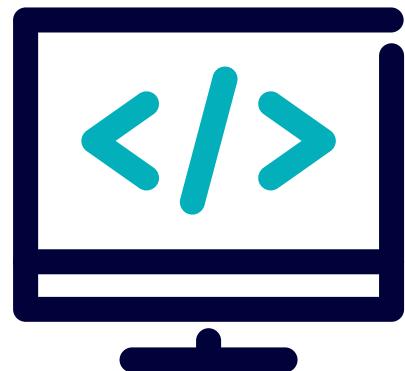
Now we show everything together. Note the addition of a `try`...`finally`.

Note that Consumer is not thread-safe. Parallel processing is meant to be managed by consumers in groups along with partitions in Kafka; more on this in the next module. There is, however, a Confluent blog post on multi-thread Consumers: https://www.confluent.io/blog/kafka-consumer-multi-threaded-messaging/.

Lab: Basic Kafka Consumer

Please work on **Lab 4a: Basic Kafka Consumer**

Refer to the Exercise Guide

05: Groups, Consumers, and Partitions in Practice

CONFLUENT

Global Education

Module Overview

This module contains three lessons:

a. How Do Groups Distribute Workload Across Partitions?

b. How Does Kafka Manage Groups?

c. How Do Consumer Offsets Work with Groups?

Where this fits in:

- Hard Prerequisite: Starting with Consumers

- Recommended Follow-Up: Either of these branches of developer content:
  - Other Components of a Kafka Deployment
  - Additional Challenges in Core Kafka Components

05a: How Do Groups Distribute Workload Across Partitions?

Description

Consumers in groups, what distinguishes consumers in same group and why we'd use more than one group. Valid assignments of consumers to partitions, range and round robin partition assignment strategies and when to use each.
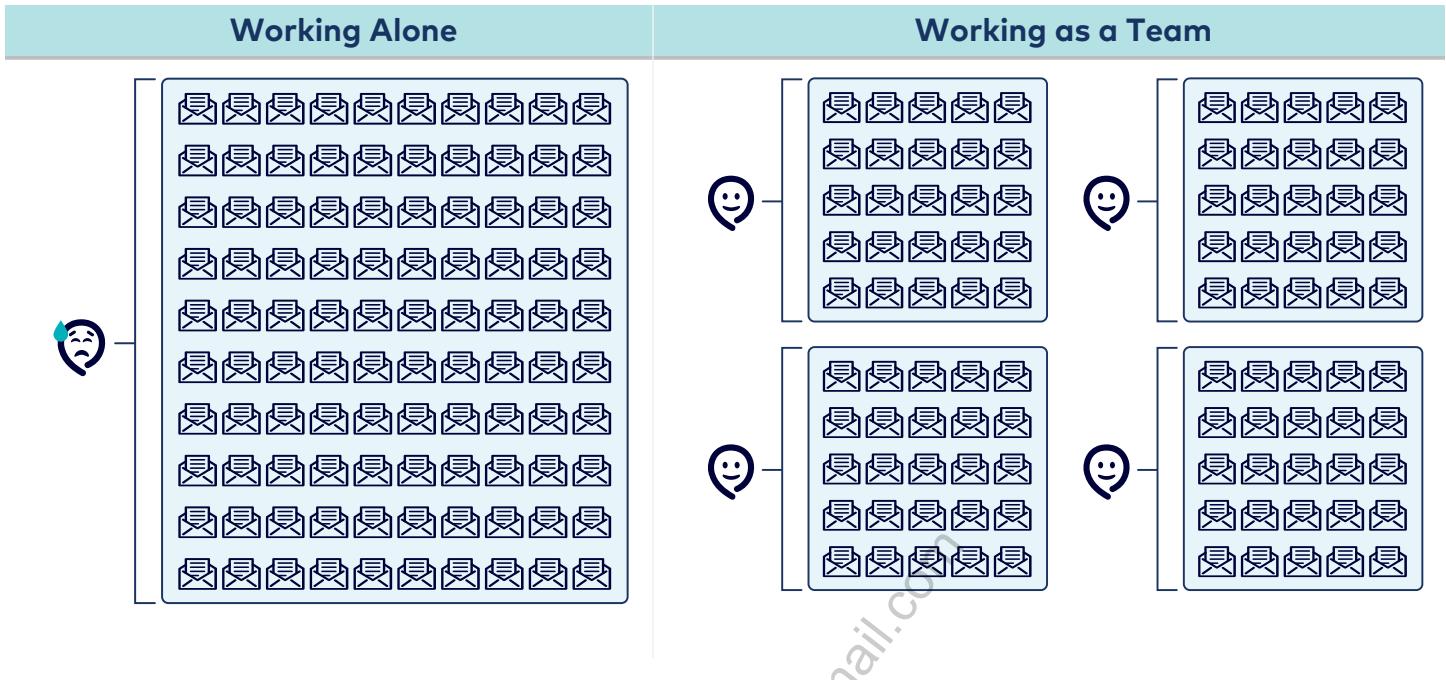
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe what's different and same about consumers in a group

- Distinguish valid assignments of consumers to partitions from invalid

- Compare and contrast range and round robin

- Describe a scenario when range would be appropriate, including prerequisites

- Describe a scenario when round robin would be appropriate

Teamwork in Real Life

Working Alone	Working as a Team

Imagine you're the person on the left, tasked with folding each of the 100 letters by yourself. Whoa! That's a lot of work.

Now imagine you're part of a team of four, doing the same thing. You have help. You can each fold 25 letters. Much better! Less work for everyone, you can work in parallel, and those letters will get folded faster.

In many manual tasks, it's much better if we share the workload among a team. In Kafka, consumers can work together as a group and share the workload of processing messages, in parallel.

Grouping Consumers

- Best practice is for consumers to operate in **consumer groups**

- Consumers in a group…

  ◦ …all do the same thing

  ◦ …using different data

- Why?

  ◦ Share workload

  ◦ Consume in parallel

  ◦ Scale up and down

Kafka consumers work in groups just like our real-life parallel on the last slide.

Groups Aren't Just for Consumers!

- Groups allow for parallel processing and scaling up and down

- Kafka has built automatic group management

  → next lesson!

- Groups are leveraged in many places in Kafka and Confluent Platform:
  - consumers
  - Kafka Connect workers
  - Kafka Streams applications
  - ksqlDB servers

The idea of groups is not just for consumers. Consumers are just one area where Kafka leverages groups.

How do we Configure Consumer Groups?

Set consumer property `group.id` like any other property, e.g.,

```
props.put(ConsumerConfig.GROUP_ID_CONFIG, "order_processor");
// other properties
```

`ConsumerConfig.GROUP_ID_CONFIG` resolves to `group.id`.

The `group.id` becomes another property we want for consumers.

Note that we use the exact same `group.id` property in configuring groups of Kafka Connect workers.

Consumers and Partitions

- Earlier, we said consumers in a group do the same thing on different data

- Given a subscribed topic, consumers in a group work together to consume all partitions of a topic

- Consumers get assigned partitions to consume

  ◦ Two major types of `partition.assignment.strategy`...
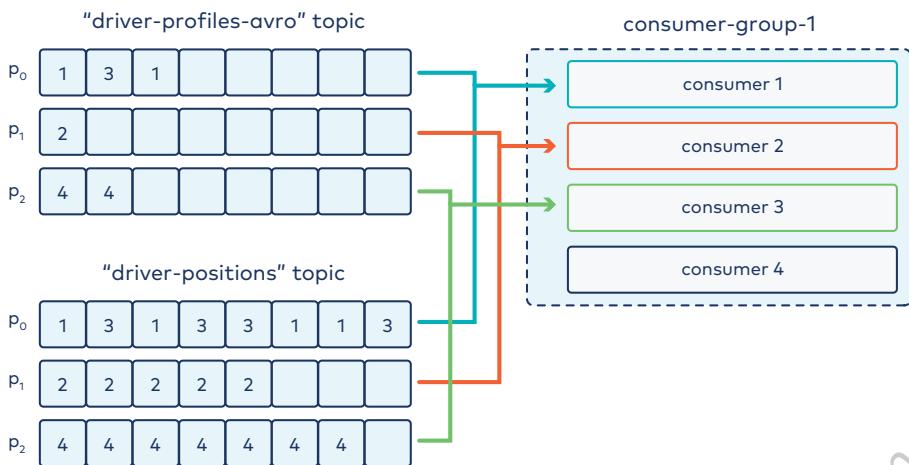
We now start to get into that detail of exactly which data in a subscribed topic a consumer will read. Partitions are the first step.

To reiterate, you as a developer **do not** say which partitions your consumer is reading from; Kafka handles that automatically. You **do** need to subscribe to a topic.

You can, however, decide which strategy Kafka will use for matching consumers with partitions. All consumers in a group must have the same value of `partition.assignment.strategy`.

Partition Assignment: Range

Why? Relate data across two or more co-partitioned topics.

	All topics must be **co-partitioned**, i.e., have
⚠️	1. same number of partitions 2. same partitioner 3. same set of keys

ℹ️	This is the default strategy.

Property config:

```
partition.assignment.strategy =
    org.apache.kafka.clients.consumer.*RangeAssignor*
```

This is the default partitioning assignment strategy, ultimately used when you want to relate data across topics. (If you don't and you're only consuming from a single topic, use the next strategy).

So, we want to relate data across topics, and we specifically want the same consumer to get records with the same key, regardless of which topic they came from. Well then, how we partition matters. Both (or all) topics need the exact same style of partitioning:

- same number of partitions

- same partitioning strategy (so far, we just know the default hash-mod, but keep this in mind if you choose to change this later, as you can learn about in the Message

Considerations module)

- use the same keys

Range assignment is done on a topic by topic basis. In general, it works like this: Order the partitions by number and the consumers lexicographically. Let p be the number of partitions in the given topic, and let c be the number of consumers in the consumer group. Attempt to divide the number of partitions by the number of consumers. Let a be `floor(p/c)` and b be the remainder `p % c`. The first b consumers will be assigned `a + 1` partitions, and the rest will each be assigned a partitions. This process happens for each topic the consumer group is subscribed to. Here are some examples:
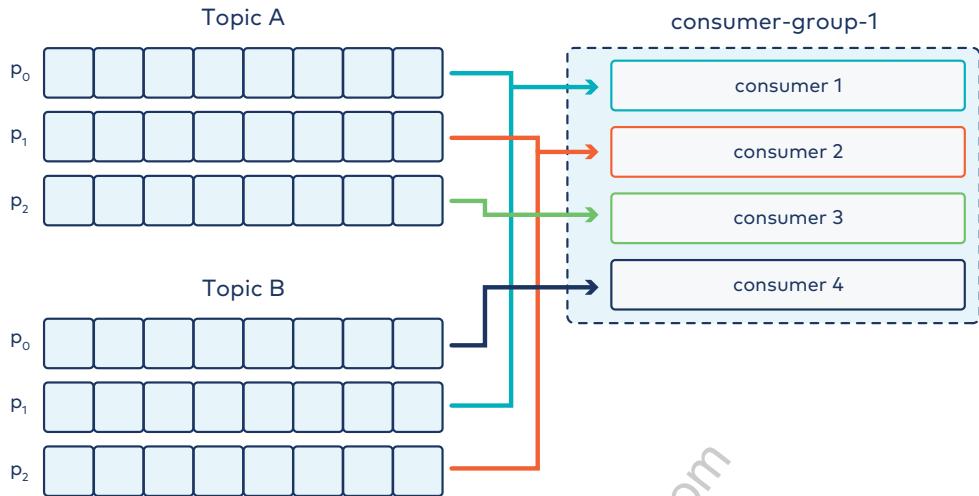
- For 16 partitions and 5 consumers, `16 = 5*3 + 1`. So each consumer will end up with at least 3 partitions. There is one left over, so the first consumer gets 1 extra partition.

- For 18 partitions and 7 consumers, `18 = 7*2 + 4`. So each consumer will end up with at least 2 partitions. There are 4 left over, so the first 4 consumers will get an extra partition.

- For 12 partitions and 20 consumers, `12 = 20*0 + 12`. So each consumer will end up with at least 0 partitions. The first 12 consumers get 1 extra partition (so the first 12 consumers each get a partition, and the remaining consumers are idle).

More information regarding the Range assignor is available at the following link: https://docs.confluent.io/current/clients/javadocs/org/apache/kafka/clients/consumer/RangeAssignor.html

Partition Assignment: Round Robin

Why? Balance load so each consumer has roughly the same number of partitions.

Property config:

```
partition.assignment.strategy =
    org.apache.kafka.clients.consumer.*RoundRobinAssignor*
```

If your consumer is subscribed to just one topic, you should choose the round-robin partition assignment strategy to get the most balanced assignment of consumers/partitions. Range would not make sense in that case.

Note that there are two different sub-types of round-robin, but we need to understand group management in the next lesson to make sense of them. So... let's leave it at this for now, review what we've learned, and get on to group management!

Partition Assignment: Round Robin, continued

There are two different sub-types of round robin:

• Sticky

• Cooperative Sticky

These both preserve some assignments after a rebalance and improve performance.

What's a rebalance? Let's move on to the next lesson and find out (after an activity)!

Activity: Choosing Partition Assignment Strategies

Your instructor will put you in groups and assign your group one of the scenarios below. In all scenarios, assume you'll be spawning a consumer group with more than one consumer to process the relevant data. For your scenario, describe...

- what partition assignment strategy makes the most sense and why

- if there are any requirements on how many partitions you should have for the topic(s) in play

Scenarios:

1. You are reading from one topic, `locations`, which tells geographic information about places where something is happening, and want to display all of that information.

2. You are reading from two topics and want to give both details of an order from an e-commerce retailer and its current status: `orders`, containing information about orders; `order_status`, containing time-tagged status of orders, e.g., `packed`, `shipped`, etc.

3. You are reading from one topic, `order_status`, containing time-tagged status of orders from an e-commerce retailer, e.g., `packed`, `shipped`, etc.; you want to display updates on a ticker observed by a fulfillment center manager who wants to see every time a package is packed.

05b: How Does Kafka Manage Groups?

Description

Heartbeats, what triggers rebalancing, pros and cons of rebalancing. Relating Kafka's group management protocol across various components, e.g., consumers and workers. Reasons to grow and shrink group size.
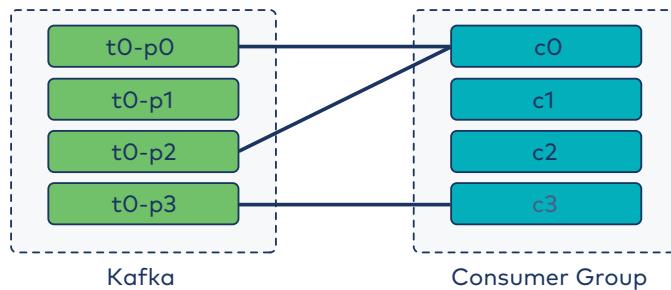
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe how "dead" consumers are detected

- List related configuration settings and their defaults

- Explain what a developer must do to trigger rebalance, i.e. nothing

- List the different things that could happen to trigger a rebalance

- List a pro and con of rebalance

- List other places in the Kafka ecosystem group management happens (workers, Streams, ksqlDB)

- Explain reasons one would deliberately grow or shrink a group

What's Wrong With This Picture?

Here, the consumers in the group are all subscribed to topic `t0`.

Kafka   Consumer Group

We've said that consumers subscribe to topics and we've said that all of the consumers in a group work together to do the same thing, but on different data, ultimately to process all of the messages in a topic. We last saw the two major kinds of strategies for how consumers could be assigned to partitions.

So, in this picture, we have two problems:

- One partition is not being consumed

- The workload is not balanced among the consumers in the group.

While this wouldn't arise from either of the two strategies we know, this could happen in a normal course of events in a real Kafka deployment. What can we do? Well, let's move on...

Rebalancing Partition/Consumer Assignments

In situations like on the last slide, the assignment isn't optimal. Kafka's group management protocol detects non-ideal assignments and **rebalances**.

Triggers of rebalance:

- **Number of consumers in play changes**

  - Consumer was added

  - Consumer was removed

    - Intentionally

    - Consumer died

- **Number of partitions in play changes**

  - Number of partitions for topic increased

  - Topic subscription changed

By "in play," we refer to relevant to the current group.

Remember that you cannot **decrease** the number of partitions for a topic.

If any of these things happens, Kafka will detect it and trigger a rebalance. The assignment of consumers to partitions gets assessed and everything gets reassigned optimally.

Note that rebalancing happens automatically—these are the triggers. You cannot force a rebalance, nor can you prevent one if one of these triggers occurs.
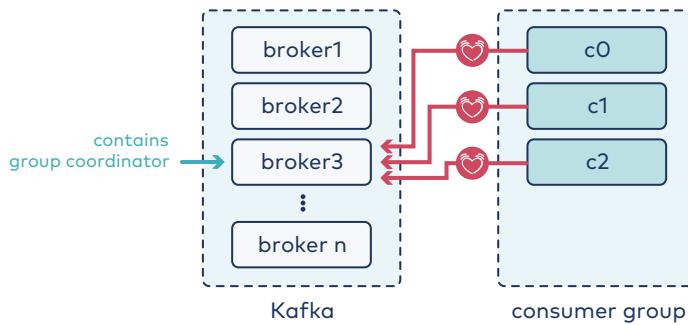
> ⚠️ Note that a consumer calling `pause` does **not** trigger a rebalance.

Note, for the second major bullet, behind the scenes, the number of partitions changing causes the metadata for the topic to change, and this metadata changing is what ultimately triggers the repartitioning.

How Does Kafka Detect Dead Consumers?

- Consumers send a heartbeat to Kafka every 3 s (or `heartbeat.interval.ms`)

- If 45 s (or `session.timeout.ms`) passes without a consumer heartbeating, it is deemed dead

Note that the default value of `session.timeout.ms` of 45 s shown applies in Apache Kafka 3.0 and above. In prior version, the default was 10 seconds.

Among our triggers for rebalance was that a consumer died. How would Kafka know that? Via this heartbeating mechanism.

Every 3 s (by default, or `heartbeat.interval.ms`), each consumer sends a heartbeat to Kafka. Consumers who fail to send a heartbeat for 45 s (by default, or `session.timeout.ms`) will be deemed dead and removed from the consumer group.

This triggers a rebalance.

The heartbeats go to a thread called the Group Coordinator running on one of the brokers. (This is per group; two different groups could have the coordinators running on different brokers).
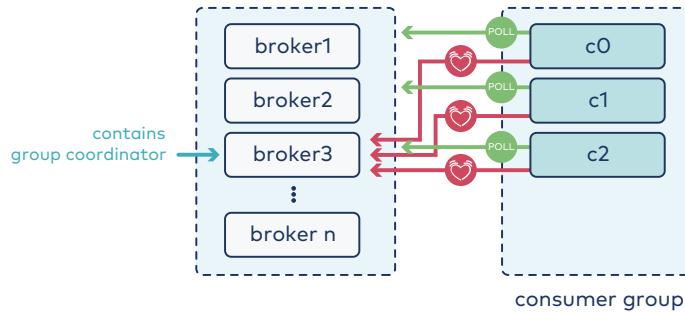
Remember, it isn't just consumers that are grouped. Kafka Connect has workers that copy data. We can say that every 3 s (by default, or `heartbeat.interval.ms`), each worker sends a heartbeat to Kafka. Workers who fail to send a heartbeat for `session.timeout.ms` will be deemed dead and removed from the worker group. The logic is exactly the same. (Note that, while it's inconsistent, the default for `session.timeout.ms` is 10 s for Connect workers, not the 45 s for consumers.)

By the way, the `heartbeat.interval.ms` and `session.timeout.ms` defaults work well for most Kafka deployments and are rarely changed. If you have a good reason to change them, always allow for two heartbeats to be missed in selecting the timeout. (In the old defaults,

`10 = 3*3 + 1`. The new default is more liberal.)

But wait…

contains group coordinator →

consumer group

- Consumers must `poll`

- If 5 mins (or `max.poll.interval.ms`) passes without `poll`, consumer is also deemed dead

---

Certainly, a consumer that fails to heartbeat cannot be contributing to the group, so that should indirectly trigger a rebalance. But what about a consumer that is heartbeating but not doing anything useful? That does not advance the group's progress, so it should also lead to a rebalance.

Remember that a consumer's job is to poll and process messages. If it isn't polling, it can't be processing. As such, the heartbeat thread also monitors if a consumer fails to poll for too long (5 s by default). If it does, the consumer will also be deemed dead and removed from the group.

This setting is also one that is rarely changed from the default. The reason it might be is that the processing messages after a `poll()` call takes too long and causes a timeout. However, remember that we can tune fetch requests and this might also be an indicator of fetch requests that fetch too much data.

Implications of Rebalance

- Consumption pauses

- A consumer may read a partition some other consumer had been reading

  - Offsets need to be communicated—see next lesson!

- Stateful application may need to rebuild state

- Kafka takes the opportunity to optimize assignments

Rebalancing is a good thing: it allows consumers to fail and the group to bounce back with the least impact possible. It allows us to scale partitions or scale consumers and have the load automatically balanced without any intervention from the developer or the administrator. But...

...it is not free! While the reassignment happens, consumption is paused. This adds some latency.

Another challenge of rebalancing is that consumers may be assigned different partitions than before. But other consumers had been making progress and had offsets and we don't want messages to be reprocessed. But consumer offsets need to be communicated. This is an issue so important of its own lesson, so that's the next lesson.

You could have a consumer application that is stateful, say it's keeping track of information within categories as it processes partitions. That state information also needs to be communicated to the correct consumers, presenting another challenge.
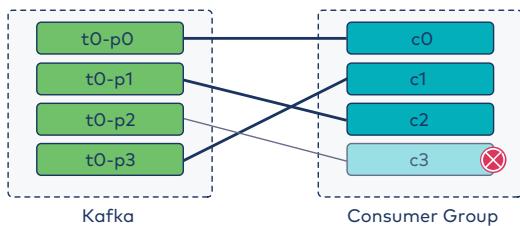
In general, though, rebalance is a good thing and that outweighs the challenges.

Backing up to the first issue: paused consumption and added latency. Two things to note:

- If an administrator is upgrading the Kafka version on a consumer, that would involve taking down the consumer, which would trigger a rebalance. Then that consumer would be brought back up, triggering a second rebalance. Perhaps we want to avoid this problem. While this is more of an administrative concern, the solution is something called Static Group Membership

- Maybe a full rebalance is not necessary. See the next slide...

Do We Always Want to Reassign **Everything**?

Suppose you have this scenario:

Kafka   Consumer Group

So…

- Prerequisite: Using round robin assignment, stateful consumer

- Problem: One partition isn't being read

  ◦ But others are okay

  ◦ Rebalance takes time

- Solution: Use **sticky partitioner**

  ◦ Preserves assignments that don't need to change

  ◦ Selection of new assignment is faster

So, we said on the last page about the first issue: paused consumption and added latency. Maybe a full rebalance is not necessary. Perhaps some assignments can stay in place. The sticky partitioner is an option in this case. It preserves whatever assignments it can in order to reduce rebalancing time.

Here, there are four partitions "in play" and three of them are being consumed by consumers in the group. With Sticky, only partition `t0-p2` is not being consumed (because the now-dead `c3` had been consuming it). The rebalance with Sticky would give `t0-p2` to one of `c0`, `c1`, or `c2`, not thinking about the balance of workload. (In this case, repartitioning with round-robin would still end up with one consumer having two partitions and two consumers having one partition, but it would take longer than with repartitioning with Sticky.)

The setting is:

```
partition.assignment.strategy =
    org.apache.kafka.clients.consumer.*StickyAssignor*
```

Even with the sticky assignor, partition assignments do sometimes change across rebalances because the priority is distributing partitions as evenly as possible. The benefit of attempting to preserve partition assignments across a rebalance is that stateful consumers have less local state to rebuild, but the cost is that reassignment overhead increases with the total number of partitions across the input topics. This would not be a good choice for cross datacenter replication for this reason.

A solution to this is CooperativeSticky. At a high level, it is very similar to Sticky but it uses

consecutive rebalances rather than the single stop-the-world used by Sticky.

The setting is:

```
partition.assignment.strategy =
    org.apache.kafka.clients.consumer.*CooperativeStickyAssignor*
```

More information regarding these assignors is available at the following links:

- [Round Robin documentation](#)

- [Sticky documentation](#)

- [CooperativeSticky documentation](#)

Remember: Not Just Consumers

Group management applies not just to consumers.

Kafka leverages similar logic for groups in other places that you may learn about in other modules.

Just a reminder that group management applies to other areas in Kafka, e.g. Kafka Connect worker groups, Kafka Streams application instances, ksqlDB servers. While there may be specific details to each, as you first learn about each of those concepts, keep the ideas of group management in mind and you'll be off to a very good start in understanding how they work.

Reiterating an example from a few pages back: Kafka Connect has workers that copy data. We can say that every 3 s (by default, or `heartbeat.interval.ms`), each worker sends a heartbeat to Kafka. Workers who fail to send a heartbeat for 10 s (by default, or `session.timeout.ms`) will be deemed dead and removed from the worker group. The logic is exactly the same.

Activity: Reviewing Rebalance Triggers

Suppose we are assuming default settings and using round robin. Further suppose all consumers in question are part of the same group, all are subscribed to one topic and partitions are all part of that one topic.

For each given event, will that event alone trigger a rebalance?

a. Consumer can't access network for 7 seconds

b. Consumer can't access network for 59 seconds

c. Partition count went from 12 to 15

d. Partition count went from 12 to 11

e. Consumer added to group

f. One consumer is reading from 4 partitions and another is reading from 2 partitions

g. Two consumers are each reading from a different partition. First partition receives 200 messages/second, while second receives one message every 3 seconds.

h. Consumer is forcibly shut down.

Apply the knowledge from this lesson to assess each of these scenarios.

05c: How Do Consumer Offsets Work with Groups?

Description

Consumer offsets locally vs. committed to Kafka. Automatic committing. How consumers newly assigned to a partition can recover committed offsets.
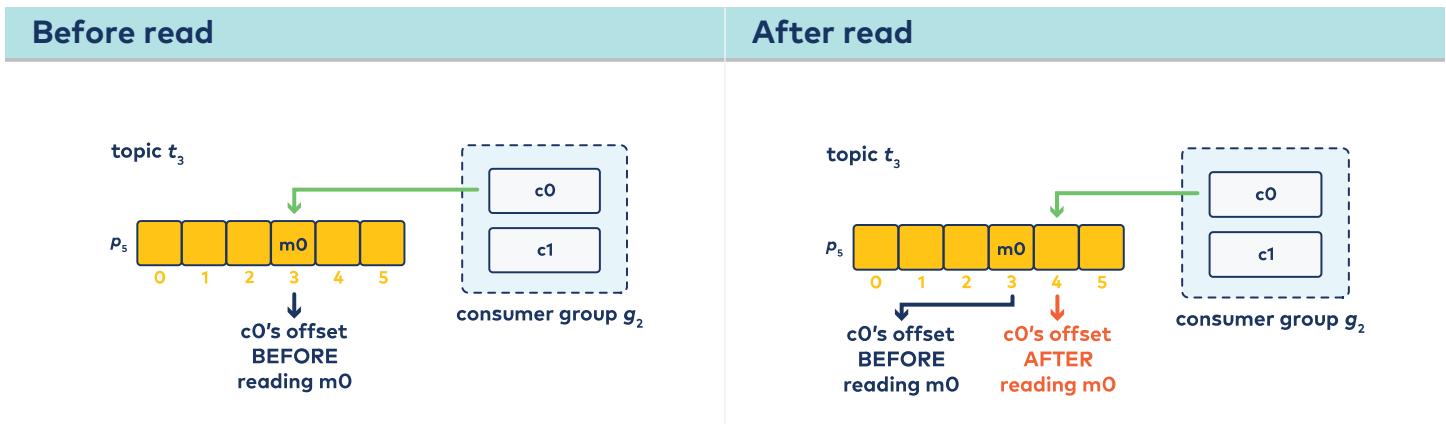
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain how offsets are tracked locally and in Kafka

- Distinguish between three ways consumers could commit offsets

- Describe how a consumer newly assigned a partition knows where to read

- Explain why offsets are tracked group-wide vs. per consumer

Review: What is a Consumer Offset?

Before read	After read
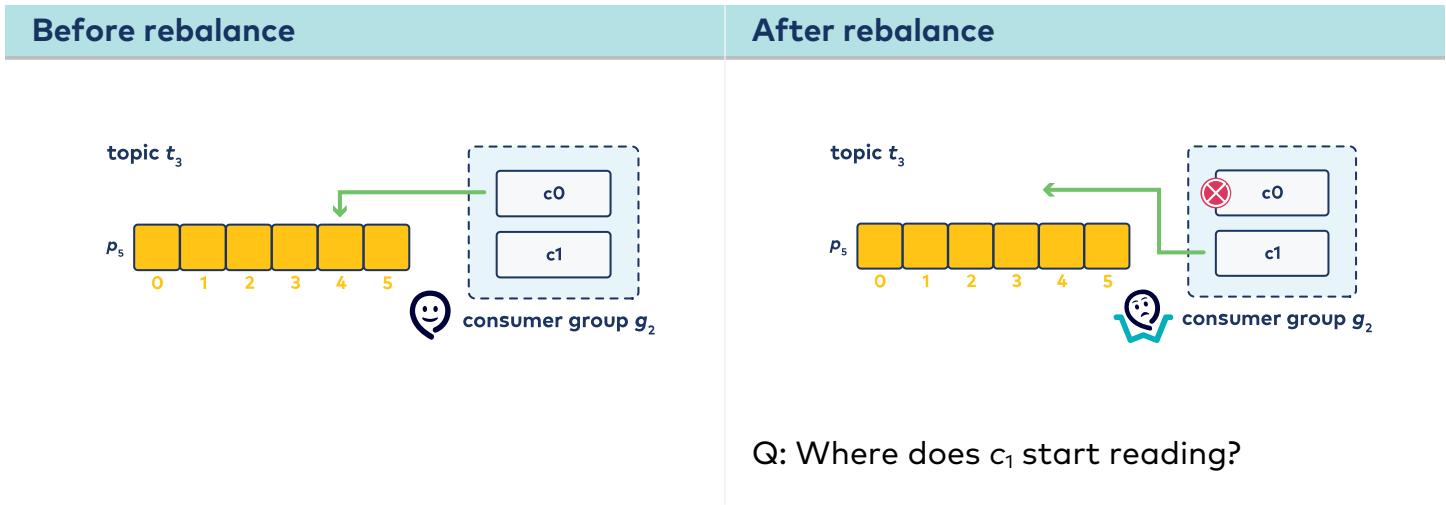
- Per consumer

- Per partition

- Where to read **next**

Recall that each consumer maintains an offset in each partition it is reading. That offset is the location of the message it will read next.

> ⚠️ The language "read next" is important. A consumer offset is **not** what a consumer read last. While this is a common misconception and it is not unreasonable, it is not correct.

What about Rebalancing?

Before rebalance	After rebalance

topic t_3

p_5
0	1	2	3	4	5

c0

c1

consumer group g_2

topic t_3

p_5
0	1	2	3	4	5

c0

c1

consumer group g_2

Q: Where does c_1 start reading?

So, a consumer knows where it is reading in a partition, but what happens when a consumer dies? As we know from the last lesson, a consumer dying triggers a rebalance. This means that some other consumer in the same group will now be assigned to read from the same partition. But if the offset is in the memory of a consumer that is now dead, what does the new consumer do?

Committing Offsets

- Internal topic `__consumer_offsets` tracks consumer offsets. Each entry:
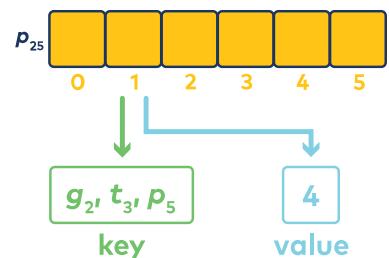
Key	Value
• **group** • topic • partition	• offset

- Consumers back up their offsets to this topic
  - called **committing offsets**
  - every 5 s (or `auto.commit.interval.ms`)
- A consumer newly assigned to a partition checks here to know where to start consuming

(**topic** `__consumer_offsets`)

p_{25} 0 1 2 3 4 5

g_2, t_3, p_5 — **key**

4 — **value**

To deal with the problem on the last slide—consumers back up their offsets to an internal topic in Kafka. This is called committing offsets.

Offsets are committed at the group level, not at the individual consumer level. When a consumer is assigned a new partition, it won't have an offset in its own memory from which to start reading. It will check `__consumer_offsets`. Having the offset stored at the group level makes sense in this regard. Further, since it could not happen that more than one consumer in a group is reading from the same partition, it is not necessary to store more than one offset per group for any partition.

Let's expand on the details of the graphic:

- This graphic illustrates the offset entry committed for the situation on the prior slide - a consumer in g_2 assigned to p_5 of topic t_3 has an offset of 4 to commit.
  - The key of the committed offset entry is (g_2, t_3, p_5).
- The partition in the offsets topic is not the same (in general) as the partition in the topic. So, this graphic uses a different partition number (25) to illustrate that point.
  - Likewise, the offset of the entry in partition 25 of the offsets topic is just the next unused offset in that partition and is independent of the offset in t_3. Here, 1 was chosen just to illustrate the difference.

Note also
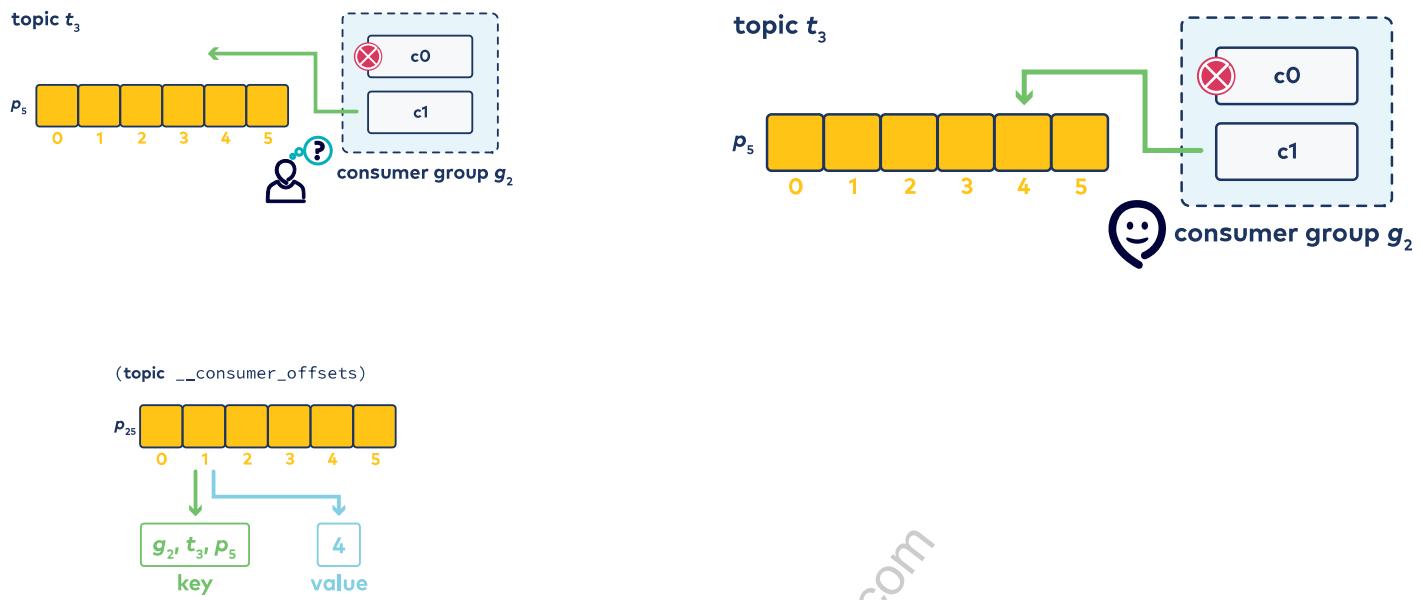
- The topic `__consumer_offsets` is compacted.

- Kafka uses the key tuple (group, topic, partition) to determine which partition of the `__consumer_offsets` topic to write an offset entry—and look up an offset entry.

- These two facts make the lookup of offsets fast; Kafka does **not** have to scan all entries in the offsets topic when trying to determine the offset for a particular (group, topic, partition) tuple.

Committing happens automatically.

Note that `auto.commit.interval.ms` is a consumer property.

If there isn't an entry in `__consumer_offsets` for a given group/topic/partition, the consumer property `auto.offset.reset` governs the behavior. This comes up in the module Challenges with Offsets.

So...

topic t_3

topic t_3

p_5

consumer group g_2

(**topic** __consumer_offsets)

p_{25}

g_2, t_3, p_5 — key

4 — value

For the scenario from a few slides back, consumer *c0* would have committed an entry to
`__consumer_offsets`:

- key: (this consumer group's ID, the topic name, the partition index for the yellow partition in the graphic) = (g_2, t_3, p_5)

- value: 4

So, after the rebalance and *c1* was assigned the yellow parititon, it looks to the
`__consumer_offsets` topic for an entry for the given partition and its consumer group. It finds an entry and sets its consumer offset in memory to 4.

Now that c_1 is assigned to p_5, it can use what it found in the internal topic to pick up where c_0 left off.

More on Committing Offsets

- Automatic committing of offsets is default

  ◦ Can be turned off by setting consumer property `enable.auto.commit` to `false`

- Can manually commit with `commitAsync()`

  ◦ Does not block

  ◦ Should have a callback

  ◦ Or do blocking commit with `commitSync()`

While automatically committing offsets is the default, you may have reasons you need to commit offsets manually.

More will come on this in Challenges with Offsets module.

Activity: Analyzing Offsets and Committing

Problem

Consider this scenario:

- Consumer group g_0 has three consumers - c_0, c_1, c_2.

- All are subscribed to a topic `claims` with four partitions - p_0, p_1, p_2, p_3.

- c_0 is reading from p_0 and p_3, c_1 is reading from p_2, c_2 is reading from p_1.

Suppose these events happen (and no other consumption occurs):

1. Time 7: consumer c_0 receives message at offset 9 from p_0.

2. Time 9: consumer c_0 receives messages at offset 22 and 23 from p_3.

3. Time 10: consumer c_0 commits offsets to `__consumer_offsets`:

   ◦ `(c₀, claims, p₀, _)`

   ◦ `(c₀, claims, p₂, _)`

   ◦ `(c₀, claims, p₃, _)`

4. Time 12, consumer c_0 dies. Rebalance completes successfully but consumption fails.

Your quest: Fill in the missing details. What's wrong with this picture?

Use what you've learned in this lesson and module in general to solve this problem.

Additional Components of Kafka/CP Deployment Overview

CONFLUENT
Global Education

Agenda

This is a branch of our developer content on additional components common in Kafka/Confluent Platform deployment. It is broken down into the following modules:

6. Starting with Schemas

7. Integrating with the Schema Registry

8. Introduction to Streaming and Kafka Streams

9. Introduction to ksqlDB

10. Starting with Kafka Connect

11. Applying Kafka Connect

This branch assumes proficiency in concepts from the Core Kafka Development branch.

Here is an expanded version of the outline, including the lessons that make up each module:

1. Starting with Schemas

   a. Why Should I Care About Schemas?

   b. How Do I Write Schemas in Avro or Protobuf?

   c. How Do I Design Schemas that can Evolve?

2. Integrating with the Schema Registry

   a. How Do I Make Producers and Consumers Use the Schema Registry?

3. Introduction to Streaming and Kafka Streams

   a. What Can I Do with Streaming Applications?

   b. What is Kafka Streams?

   c. A Taste of the Kafka Streams DSL

   d. How Do I Put Together a Kafka Streams App?

4. Introduction to ksqlDB

   a. What Does a Kafka Streams App Look Like in ksqlDB?.

  b. ksqlDB Basics

  c. How Do Windows Work?

  d. How Do I Join Data from Different Topics, Streams, and Tables?

5. Starting with Kafka Connect

  a. What Can I Do with Kafka Connect?

  b. How Do I Configure Workers and Connectors?

6. Applying Kafka Connect

  a. Deep Dive into a Connector & Finding Connectors

  b. Full Solutions Involving Other Systems

This branch assumes proficiency in concepts from the Core Kafka Development branch. In particular, you need some knowledge from the following:

- Starting with Producers

- Starting with Consumers

- Groups and Consumers in Practice

06: Starting with Schemas

CONFLUENT
Global Education

Module Overview

This module contains three lessons:

a. Why Should You Care About Schemas?

b. How Do You Write Schemas in Avro or Protobuf?

c. How Do You Design Schemas that can Evolve?

Where this fits in:

- Hard Prerequisite: Starting with Consumers

- Recommended Follow-Up: Integrating with the Schema Registry

06a: Why Should You Care About Schemas?

Description

What schemas are and why we use them.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe what a schema is

- Given a concept, conceptually describe a schema for it

- List one or two reasons schemas are useful

Activity: Specifying a Writing Utensil in Text
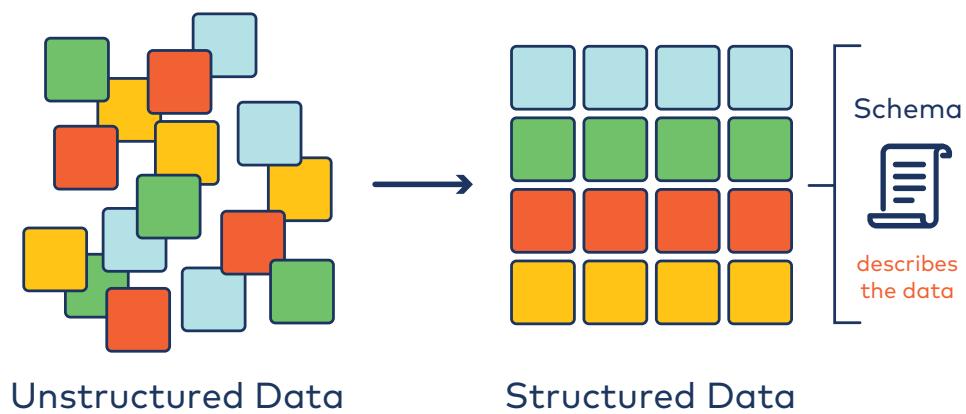
Suppose you want to describe a writing utensil that could be a pen or marker or pencil. You're writing software that will work with plain text files, one for each writing utensil. Your requirements include storing the following for each writing utensil: what type it is, size of the writing tip (e.g., 5 mm), color, brand, model, if it is retractable, and how it can be erased - if it can.

Grab the nearest writing utensil and write a plain text file how you might think to encode this.

Don't think too hard; just write what comes to mind within a minute.

The Need For Schemas

Unstructured Data → Structured Data

Schema
describes the data

While there may be plenty of reasonable ways to represent data, we want to use a **schema** to agree upon how to do so in a structured way.

Schemas Can Change

Imagine `BusinessCustomer` schema:

1980s	1990s	now
• company name	• company name	• company name
• contact person	• contact person	• contact person
• physical address	• physical address	• physical address
• phone	• phone	• phone
• fax number	• fax number	• ~~fax number~~
	• **email address**	• email address

> ℹ️ Schema Evolution will be an important topic we visit in two lessons.

What you initially choose as a schema might change over time. This slide gives an example of changes that would not have been anticipated. You want to keep this in mind from the start.

The notion of schema evolution addresses this; we'll delve deeper in two lessons.

Specifying Schemas

In a subsequent lesson, we'll formalize how we should specify a schema and how we can do it in Confluent Platform.

For now, here's a working conceptual version of a schema for a simple `WritingUtensil`:

• type, a string

• tip size in MM, a number

• color, a string

• brand, a string

This is just a concrete conceptual example of a schema.

Design with Schemas in Mind from the Start

- Think about schemas early

- Plan for schema evolution

- Confluent Schema Registry can help

    ◦ Can leverage with producers and consumers

    ◦ ...and much more

Let's get into the technical details!

So, we now know the basics of schemas, enabling us to go further in the lessons to come!

06b: How Do You Write Schemas in Avro or Protobuf?

Description

Defining why one would use Avro or Protobuf. Specifying basic schemas in both. More involved schemas in Avro via activity.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain the value in using a tool like Avro or Protobuf

- Write a single-level schema compatible with Avro

- Write a schema with a nested record compatible with Avro

- Evaluate between Avro, Protobuf, and JSON schemas

Schema Specification and Serialization Mechanisms (1)

- Can leverage a mechanism to help with

  - Specifying schemas

  - Serialization

- Both Avro and Protobuf support this

- Both are supported by Confluent Schema Registry

- Example schemas on next slide...

Schema Specification and Serialization Mechanisms (2)

Example schema in both:

Avro	Protobuf
```json\n{\n "namespace": "clients.avro",\n "type": "record",\n "name": "PositionValue",\n "fields": [\n {"name": "latitude",\n "type": "double"},\n {"name": "longitude",\n "type": "double"}\n]\n}\n```	```protobuf\nsyntax = "proto3";\n\noption java_package = "clients.proto";\nmessage PositionValue {\n double latitude = 1;\n double longitude = 2;\n}\n```

There is software to help us work with schemas. These mechanisms also help with making serialization easy.

This slide gives an example of the same schema in Avro and Protobuf. While these are syntactically correct, don't worry about the details. Rather, this is to give you a big picture idea what a schema might look like using one of these tools.

Confluent Schema Registry, which we will learn about in a few lessons, supports working with schemas using either of these tools.

Honing in on Avro

- The way we'll use Avro:

  - Specify schema using JSON

  - Use it to generate an equivalent Java class to include in our code

  - Called **specific** mode

- Details:

  - File extension `.avsc`

  - Provide a `namespace` with a schema that becomes Java `package` to import

  - Specify a schema, commonly done with a data type of `record`

We'll focus on Avro for writing schemas. Avro is another Apache project.

What we'll do:

1. Write a schema using JSON in a format to adhere to what Avro expects

2. Use Avro to generate equivalent code to include in our producers
   → You'll experience this in lab.

Avro operates in three different modes:

- Generic: Manually create both the data type (Java class) and the schema (`*.avsc` file)

- Reflection: Manually create the data type and then generate a schema from that code

- Specific: Manually write the schema and then generate code to include in your program

It is the last that we will use here.

The `record` data type in Avro will be used in all of the schemas we use in Kafka. A `record` is comprised of fields of heterogenous data types. We will see examples to come.

Note, though, that `record` is not required and simpler data types could be used for simpler schemas.

Note that while we mention Java specifically on the slide, Avro works with other languages.

[More on why you might want Avro](#)

Sample Schema In Avro

```json
{
    "namespace": "clients.avro",
    "type": "record",
    "name": "WritingUtensil",
    "fields": [
        {"name": "type", "type": "string"},
        {"name": "tipSizeMM", "type": "double"},
        {"name": "color", "type": "string"},
        {"name": "brand", "type": "string"}
    ]
}
```

In the last lesson, we saw a conceptual schema for a writing utensil. This is Avro-formatted JSON for that schema.

Start with this and emulate the syntax in writing your own schemas.

Adding a Few Things

```
{
  "namespace": "clients.avro",
  "type": "record",
  "name": "WritingUtensil",
  "fields": [
      {"name": "type", "type": "string", "default": "pen"},
      {"name": "tipSizeMM", "type": "double", "doc": "size of tip in mm"},
      {"name": "color", "type": {
                          "type": "enum",
                          "name": "color",
                          "symbols": "blue", "black", "red", "green"},
      {"name": "brand", "type": "string"}
  ]
}
```

Observe: 1. `default` value for a field

            2. `doc` string

            3. `enum` data type

This slide shows the prior schema with a few other additions that are important:

1. We can specify a `default` value for any field. In our use, if a consumer is expecting a record to conform to a schema like this and some field is missing, a default value will be used for that field.
   → This will become important in schema evolution in the next lesson.

2. For any field, we can supply a documentation string with `doc`. This is a good practice, like you know in software engineering in general.

3. We are not limited to just simple data types in Avro records. Here we show a complex data type, the `enum`.

Here's an expanded version of the same schema that brings in possible representations of the other fields that were in the original problem statement at the start of the last lesson:

```
{
  "namespace": "clients.avro",
  "type": "record",
  "name": "WritingUtensil",
  "fields": [
      {"name": "type", "type": "string", "default": "pen"},
      {"name": "tipSizeMM", "type": "double", "doc": "size of tip in mm"}
      {"name": "color", "type": {
                                 "type": "enum",
                                 "name": "color",
                                 "symbols": "blue", "black", "red", "green"},
      {"name": "brand", "type": "string"},
      {"name": "isRetractable", "type": "boolean", "default": "true"},
      {"name": "isErasable", "type": "boolean", "default": "false"}
  ]
}
```

What Else Can You Do?

- Nest records within records

- Use `array` data type

- Use `map` data type

  - Keys must be strings

- Allow fields to take on a `null` value

- Use any of these simple data types: `boolean`, `int`, `long`, `float`, `double`, `string`

ℹ	Documentation link and more examples in your handbook.

We give here a few more examples.

See [Avro documentation](#) for all the details.

Example array use with nested record:

```
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  ...
```

Example map use:

```
 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 use of a union and a null value:

```
 1  {
 2    "name" : "experience",
 3    "type": ["null", "int"]
 4  }
```

Two logical data type examples:

Decimal

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

Timestamp (ms precision)

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

- The `decimal` logical type represents an arbitrary-precision signed `decimal` number of the form `unscaled × 10^(-scale)`

  ◦ `scale`: a JSON integer representing the maximum number of decimal places (optional). If not specified the scale is 0.

    ▪ e.g. `1234567.89` has a scale of 2

  ◦ `precision`: a JSON integer representing the maximum number of digits.

    ▪ e.g. `1234567.89` has a precision of 9

- A `timestamp-millis` logical type annotates an Avro `long`, where the long stores the number of milliseconds from the unix epoch, **1 January 1970 00:00:00.000 UTC**.

- Other logical types include `uuid`, `date`, and `duration`. For more information about logical types, see [the Avro documentation](#)

Activity: Interpreting Schema Examples and Writing a Schema

Find this slide in your Student Handbook, and, in particular, find the examples in the handbook text on for the last slide and the examples we saw on the previous slides in the lesson.

1. Review and interpret the example schemas shown

2. Write a schema to represent when a train on a public transit system arrives at a train station that has multiple color-coded lines (like you see at the right). Provide a way of storing:

   a. What train line

   b. What train

   c. What station

   d. When the train arrived

Use the examples on the prior pages of your handbook and/or the documentation link as you work through this problem.

Many students have found it helpful to share screen in a breakout room and edit a schema from a previous example. To help you out, here is the schema from a few slides back, formatted for easy copying and pasting and editing.

Possible Solution to Train Schema

```
 1 {
 2    "namespace": "com.traincompany.examples",
 3    "type": "record",
 4    "name": "TrainArrived",
 5    "fields": [
 6        { "name": "line", "type": {
 7                              "type": "enum",
 8                              "name": "line",
 9                              "symbols": "red", "green", "blue", "orange"},
10        { "name": "trainId", "type": "int" },
11        { "name": "stationId", "type": "int" },
12        { "name": "arrivalTime",
13          "type": {
14             "type": "long",
15             "logicalType": "timestamp-millis"
16          }
17        }
18    ]
19 }
```

Choosing Between Avro and Protobuf

- Both are supported by Confluent Schema Registry

  ◦ Avro support has been around longer

- Both encode data efficiently

  ◦ But differently - see comparison link in guide

- What is your organization otherwise using?

More information on Protobuf:

- [Protocol Buffers defined](#)

- [Google Protocol Buffers Developer Guide](#)

- [Protobuf encoding example](#)

Schemas Written in JSON Format

Avro Schema in JSON Format	JSON Schema
| ```json
{
    "namespace": "clients.avro",
    "type": "record",
    "name": "PositionValue",
    "fields": [
        {"name": "latitude",
         "type": "double"},
        {"name": "longitude",
         "type": "double"}
    ]
}
``` | ```json
{
    "type": "object",
    "title": "driverposition",
    "properties": {
      "latitude": { "type": "number" },
      "longitude": { "type": "number" }
    }
}
``` |

> ⚠️ Messages that use JSON encoding store field names in each individual record...

We showed an example schema in Avro JSON before. We can express it in a form of JSON supported by Confluent Schema Registry that does not use Avro too, as shown here.

Expanding on the warning on the slide, note that messages that use JSON encoding store field names in each individual record, whereas Avro encoding does not. Thus there is a disk space usage tradeoff to take into account when considering the JSON encoding.

For comparison, here's the Protobuf version from earlier:

```proto
syntax = "proto3";

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

More information on JSON Schema:

• JSON Schema serdes

• JSON Schema Organization

• JSON Schema Specification

06c: How Do You Design Schemas that can Evolve?

Description

Defining schema evolution. Comparing the enforcement modes - backward, forward, full, none, transitive modes - with concrete examples via exercises.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Justify why one should consider schema evolution from the start

- Given a pair of adjacent versions of a schema, determine if the second is backward compatible or forward compatible with the first

- Given a schema, create a new version that is backward compatible

- Explain how to set schema evolution to enforce compatibility between non-adjacent numbered versions
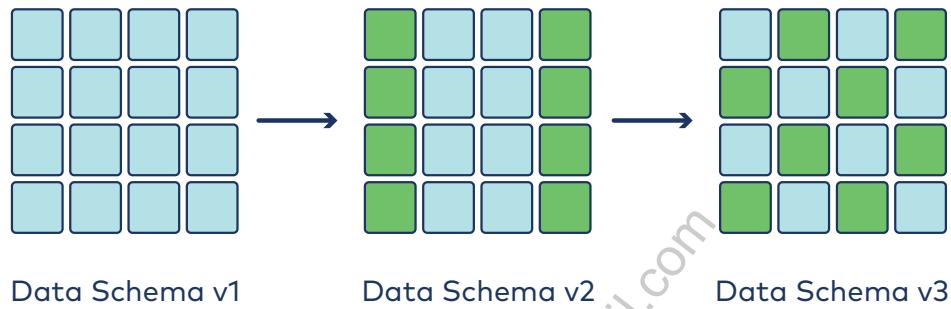
Introduction

Schemas can change!

Recall example from introduction on changes to customer record:

- add email address

- remove fax number

Data Schema v1   Data Schema v2   Data Schema v3

We've looked at writing schemas and we know schemas can change over time, but how do we plan for that change. This lesson gets into that.

"Always have a schema evolution plan in place..." -Chris Smith, VP, Engineering Data Science, Ticketmaster, from a web cast

Schema Compatibility Modes

- There are various modes defining whether one version of a schema is compatible with another

- A system like Confluent Schema Registry can enforce compatibility

- We'll focus on the modes relating two adjacent versions of a schema defined in Schema Registry

Basic Schema Compatibility Modes

Mode	Explanation
BACKWARD	Consumers expecting data using new form of schema can process data using previous version of schema
FORWARD	Consumers expecting data using previous form of schema can process data using next version of schema
FULL	Both BACKWARD and FORWARD
NONE	No compatibility checking

The four schema compatibility modes shown all are supported by Confluent Schema Registry. Your objective in this lesson is to understand the differences, and, perhaps more importantly, understand how to make schemas conform to various compatibility requirements.

Example

Schema V1 fields		Schema V2 fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```

Backward

producer → ? → consumer

Forward

consumer ← ? ← producer

```
"fields":
[
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

Schemas were truncated to fit the slide and focus on the key part.

Full schemas:

Schema V1		Schema V2

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

Backward

producer → ? → consumer

Forward

consumer ← ? ← producer

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

Example Solved

Schema V1 fields		Schema V2 fields

```
"fields":
[
    {

        "name": "firstname",
        "type": "string"
    },
    {

        "name": "lastname",
        "type": "string"
    },
    {

        "name": "age",
        "type": "int",
        "default": -1
    }
]
```

Backward

producer   consumer

Forward

consumer   producer

```
"fields":
[
    {

        "name": "lastname",
        "type": "string"
    },
    {

        "name": "age",
        "type": "int",
        "default": -1
    },
    {

        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

- 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

Activity: Assessing Schema Compatibility

Now evaluate in both directions - Pair 1:

Schema V1 Fields		Schema V2 Fields

```"fields":```
 ```[```
     ```{```
       ```"name": "firstname",```
       ```"type": "string"```
     ```},```
     ```{```
       ```"name": "lastname",```
       ```"type": "string"```
     ```},```
     ```{```
       ```"name": "age",```
       ```"type": "int",```
       ```"default": -1```
     ```}```
 ```]```
	Backward
	 producer → consumer
	
	 Forward
	 consumer ← producer
		```"fields":```
		 ```[```
		     ```{```
		       ```"name": "firstname",```
		       ```"type": "string"```
		     ```},```
		     ```{```
		       ```"name": "lastname",```
		       ```"type": "string"```
		     ```},```
		     ```{```
		       ```"name": "hobby",```
		       ```"type": "string"```
		     ```}```
		 ```]```

Now evaluate in both directions - Pair 2:

Schema V1 Fields		Schema V2 Fields

```"fields":```
 ```[```
     ```{```
       ```"name": "firstname",```
       ```"type": "string"```
     ```},```
     ```{```
       ```"name": "lastname",```
       ```"type": "string"```
     ```},```
     ```{```
       ```"name": "age",```
       ```"type": "int",```
       ```"default": -1```
     ```}```
 ```]```
	Backward
	 producer → consumer
	
	 Forward
	 consumer ← producer
		```"fields":```
		 ```[```
		     ```{```
		       ```"name": "firstname",```
		       ```"type": "string"```
		     ```},```
		     ```{```
		       ```"name": "lastname",```
		       ```"type": "string"```
		     ```},```
		     ```{```
		       ```"name": "hobby",```
		       ```"type": "string",```
		       ```"default": ""```
		     ```}```
		 ```]```

Use what you know from the prior example and definitions to assess schema compatibility in these cases.

Schemas were truncated again.

Full schemas, Pair 1:

Schema V1		Schema V2
| ```json
{
    "namespace": "example.avro",
    "type": "record",
    "name": "user",
    "fields": [
      {
        "name": "firstname",
        "type": "string"
      },
      {
        "name": "lastname",
        "type": "string"
      },
      {
        "name": "age",
        "type": "int",
        "default": -1
      }
    ]
}
``` | **Backward**<br>producer ？ consumer<br><br>**Forward**<br>consumer ？ producer | ```json
{
    "namespace": "example.avro",
    "type": "record",
    "name": "user",
    "fields":
    [
      {
          "name": "firstname",
          "type": "string"
      },
      {
          "name": "lastname",
          "type": "string"
      },
      {
          "name": "hobby",
          "type": "string"
      }
    ]
}
``` |

Full schemas, Pair 2:

Schema V1		Schema V2
| ```json
{
    "namespace": "example.avro",
    "type": "record",
    "name": "user",
    "fields": [
      {
        "name": "firstname",
        "type": "string"
      },
      {
        "name": "lastname",
        "type": "string"
      },
      {
        "name": "age",
        "type": "int",
        "default": -1
      }
    ]
}
``` | **Backward**<br>producer ？ consumer<br><br>**Forward**<br>consumer ？ producer | ```json
{
    "namespace": "example.avro",
    "type": "record",
    "name": "user",
    "fields":
    [
      {
          "name": "firstname",
          "type": "string"
      },
      {
          "name": "lastname",
          "type": "string"
      },
      {
          "name": "hobby",
          "type": "string",
          "default": ""
      }
    ]
}
``` |

Activity Solution, Part 1

Schema V1 Fields		Schema V2 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```

Backward

producer → consumer ✗

Forward

consumer ← producer ✓

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "hobby",
        "type": "string"
    }
]
```

- 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

Activity Solution, Part 2

Schema V1 Fields		Schema V2 Fields
``"fields":
[
{
"name": "firstname",
"type": "string"
},
{
"name": "lastname",
"type": "string"
},
{
"name": "age",
"type": "int",
"default": -1
}
]```	**Backward** producer → consumer ✓ **Forward** consumer ← producer ✓	```"fields":
[
{
"name": "firstname",
"type": "string"
},
{
"name": "lastname",
"type": "string"
},
{
"name": "hobby",
"type": "string",
"default": ""
}
]```

- 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

Takeaways

- If version $n + 1$ of a schema adds a field not present in version n, supply a default value for the new field in the new version for backward compatibility

- If version $n + 1$ of a schema lacks a field not present in version n, a consumer using version $n + 1$ will ignore the extra field and achieve forward compatibility

---

Here is a [documentation page with more tips](#).

Beyond One Version Change

Suppose for some schema,

- Version 3 is backward compatible with Version 2

- Version 2 is backward compatible with Version 1

This does **not** mean Version 3 is backward compatible with Version 1

Want to enforce backward compatibility with **all** prior versions → set `BACKWARD_TRANSITIVE` mode.

Similarly, Schema Registry supports `FORWARD_TRANSITIVE` and `FULL_TRANSITIVE`.

We add a few more schema compatibility modes now. These allow us to enforce compatibility beyond just adjacent versions of schemas.

Here is an example of a set of schemas that are backwards compatible, but not backwards transitive:

- Schema V1: latitude, longitude

- Schema V2: latitude, longitude, altitude (default altitude of 0)

- Schema V3: latitude, longitude, altitude (no default)

Consumers with V3 can read messages produced with V2 because all three of latitude, longitude, and altitude would be provided. Consumers with schema V2 could read messages produced with V1 because they would infer a default value of 0 for altitude. However, consumers with version V3 could not read messages produced with V1 because the consumer is expected a value for altitude to be provided and is not providing a default.

Here's a full table:

Mode	Explnation
`BACKWARD`	Consumers expecting data using new form of schema can process data using previous version of schema
`BACKWARD_TRANSITIVE`	Consumers expecting data using new form of schema can process data using any previous version of schema
`FORWARD`	Consumers expecting data using previous form of schema can process data using next version of schema

Mode	Explnation
FORWARD_TRANSITIVE	Consumers expecting data using previous form of schema can process data using any later version of schema
FULL	Both BACKWARD and FORWARD
FULL_TRANSITIVE	Both BACKWARD_TRANSITIVE and FORWARD_TRANSITIVE
NONE	No compatibility checking

Schema Compatibility Modes in Practice in CP

- Typically, we set a schema per topic and at the level of key or value

- We typically set schema evolution requirements at the same level

- Confluent Schema Registry defines a **subject** as the scope in which schemas evolve

- The default naming strategy appends `-key` or `-value` to a topic name to get subject names.

      Topic: `driver-positions`   Subjects: `driver-positions-key`
                                                                    `driver-positions-value`

> ℹ️ This is called using the subject naming strategy `TopicNameStrategy`. Other modes are available. See the enrichment page in your handbook to learn more.

The subject naming strategy described is called `TopicNameStrategy`.

To learn how to set schema evolution in Schema Registry, see the next lesson.

Subject Naming Strategies

Here are your other subject naming mode choices for the key or value of any topic:

Naming Strategies	Configurations
• `TopicNameStrategy` (default) • `RecordNameStrategy` • `TopicRecordNameStrategy`	• `key.subject.name.strategy` • `value.subject.name.strategy`

Here's why you'd use the other two modes:

• `TopicRecordNameStrategy`: `<subject-name> = <topic>-<type>-key | <topic>-<type>-value`
  This is used in a topic with many event types to allow each type to evolve separately

• `RecordNameStrategy`: `<subject-name> = <type>-key | <type>-value`
  This allows evolution of an event type that is used across many topics

---

Some people call a Topic that has multiple schemas a "fat" Topic. For a detailed discussion of when to use this approach, and the use of custom subject naming strategies, refer to:
https://www.confluent.io/blog/put-several-event-types-kafka-topic/

Example

07: Integrating with the Schema Registry

CONFLUENT
Global Education

Module Overview

This module contains one lesson:

    a. How Do You Make Producers and Consumers Use the Schema Registry?

Where this fits in:

- Hard Prerequisite: Starting with Schemas

- Recommended Follow-Up: continuing with other modules in this branch

07a: How Do You Make Producers and Consumers Use the Schema Registry?

Description

What Confluent Schema Registry is and benefits of using it. Life cycle of a message using SR. Specifying schema evolution restrictions using SR.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- List 3 benefits of Schema Registry

- Describe what a schema ID unique identifies

- Explain how a schema ID gets associated with a message

- Take Java API producer or consumer code and make it compatible with Schema Registry and Avro

- Write a command to set a schema evolution requirement

Scenario

- Say...

  ◦ We have a schema for a bank transaction

  ◦ It contains many details and uses 1 KB of data

  ◦ It is sent with every record for a transaction

  ◦ It is stored with every record for a transaction

  ◦ A bank produces roughly a million transactions per hour

Question: How much disk space does this use? How much bandwidth?

Schema ID

- Maintained by Schema Registry

- Uniquely identifies

  ◦ Schema

  ◦ Version

- Sent with records instead of whole schema → efficient!

- Schema Registry tracks schemas and IDs in internal `_schemas` topic in Kafka

---

Rather than sending the same schema several times and storing the same schema several times, Confluent Schema Registry allows us to send a schema ID to represent a schema. Schema IDs uniquely identify schemas and their versions.

Schemas are stored in a special Kafka topic (default name: `_schemas`, can be reconfigured with the `kafkastore.topic` property). Disk space in not used on the SR, but rather in Kafka.

What Else Can Schema Registry Do For You?

- Check records to make sure they conform to schemas

- Enforce schema evolution

If a record's value or key fails to match the expected schema, Schema Registry will throw an exception.

We'll see how to set schema evolution a bit later in the lesson. Schema Registry will reject new versions of schemas that do not conform to the schema evolution setting chosen.

Schema Registry: Producers

We visualize two scenarios here:

- A producer is about to send a message using a schema that has not been used before. When it sends that schema to SR, SR will register it and send back a new schema ID.

- A producer is about to send a message using a schema that has been used before. When it sends that schema to SR, SR just look up and send back the matching schema ID.

In either event,

- The schema is used along with Avro/Protobuf/JSON to serialize the message's key or value, whichever is appropriate. The picture shows the Avro case.

- The schema ID is serialized too and prepended to the key or value, whichever is appropriate, before the message is sent.

There's a Cache Too!

Producers cache schemas for efficiency. Consider the scenario from the last slide. Using a schema ID saves us from using network traffic between the producer and the Kafka cluster for every schema, but if we had to check the schema registry for every schema, we'd just move the problem. Instead, SR-enabled producers maintain a cache and check there for schema IDs before checking SR, thereby saving network bandwidth.

Whether the producer got the schema ID from its cache or the Schema Registry, once it has it, the process is the same as on the last slide.

Note that it's most likely the value you're using Schema Registry for, so these last two illustrations show that case. There can sometimes be complications using an SR-aware format as the key (e.g., non-deterministic serialization).

However, here's what a version of the graphic on the slide showing both the key and value being SR-aware would look like:

For more details on the serialized message format, see [this documentation page](#).

Schema Registry: The Consumer End

Here we see the consumer end of the process.

Note that when a consumer receives a message, it's serialized, so it's just zeroes and ones. The consumer needs to be configured to be reading in the right mode to be looking for schema IDs. If it is...

...the consumer can pull off the schema ID from the key or value of a message and then find the matching schema. Just like producers, they have a local cache they check first, before going to the schema registry.

Note that this figure is again showing the cases of using Avro for deserialization and using schemas for the value only. There are other cases you could consider - Protobuf, JSON, key using SR along with or instead of value - but they are similar.

Schema-Enabled Java Producer Example

Writing a producer that supports Schema Registry and uses Avro isn't much different from before. Two new steps...

First, we must use the Avro serializer for the key or value, e.g.,

```
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
```

Then, we must also configure the producer to know where our schema registry is:

```
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
          "http://schema-registry1:8081");
```

We otherwise use a Java class generated by Avro. You'll experience how to generate this in lab.

ℹ	Everything else is the same!

It is only in the configuration where our Java code will be different. We need to specify the serializer and SR location. Everything else, we could have done using our knowledge from our earlier producer module.

Here's a more complete code block:

```java
Properties props;
KafkaProducer<String, PositionValue> producer;
String key;
PositionValue value;
ProducerRecord<String, PositionValue> record;

props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
// Configure serializer classes
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
// Configure schema repository server
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://schema-registry1:8081");

// Create the producer, which expects
//     PositionValue object generated from Avro schema
producer = new KafkaProducer<>(props);

// Create the key (String) and
//     value (PositionValue object generated from Avro schema)
key = "driver-1";
value = new PositionValue(47.618580396045445, -122.35454111509547);

// Create the ProducerRecord and send it
record = new ProducerRecord<>("driver-positions-avro", key, value);
producer.send(record)
```

Schema Enabled Java Consumer Example

Three (really two) new things here...

We use the Avro deserializer instead of serializer:

```
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        KafkaAvroDeserializer.class);
```

We still need to identify where SR is:

```
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://schema-registry1:8081");
```

We need to tell our consumer to look for schema IDs:

```
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
```

Note that when a consumer receives a message, it's serialized, so it's just zeroes and ones. The consumer needs to be configured to be reading in the right mode to be looking for schema IDs. That's what
`props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");`
is for.

Here's a more complete code block:

```java
Properties props;
KafkaConsumer<String, PositionValue> consumer;
ConsumerRecord<String, PositionValue> record;
ConsumerRecords<String, PositionValue> records;

props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        KafkaAvroDeserializer.class);
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://schemaregistry1:8081");
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");

consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("driver-positions-avro"));

while (true)
{
    records = consumer.poll(Duration.ofMillis(100));

    for (record : records)
    {
        System.out.printf("Key:%s Latitude:%s Longitude:%s [partition %s]\n",
                    record.key(), record.value().getLatitude(),
                    record.value().getLongitude(), record.partition());
    }
}
```

Setting Schema Evolution: Modes Reference

For reference, Schema Registry supports the following schema evolution modes:

Mode	Explanation
`BACKWARD`	Consumers expecting data using new form of schema can process data using previous version of schema
`BACKWARD_TRANSITIVE`	Consumers expecting data using new form of schema can process data using any previous version of schema
`FORWARD`	Consumers expecting data using previous form of schema can process data using next version of schema
`FORWARD_TRANSITIVE`	Consumers expecting data using previous form of schema can process data using any later version of schema
`FULL`	Both `BACKWARD` and `FORWARD`
`FULL_TRANSITIVE`	Both `BACKWARD_TRANSITIVE` and `FORWARD_TRANSITIVE`
`NONE`	No compatibility checking

See the prior lesson for examples and a deeper understanding of the various modes.

Setting Schema Evolution: Command

We can use a command like the following to set schema evolution:

```
# 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"}
```

Replace NONE with any of the modes from the prior slide.

One can also use the Confluent Control Center UI. See this tutorial for more.

Support for Schema Registry

- Part of Confluent Platform

- Supported by clients

  - Java clients

  - `librdkafka` clients except Go

- Supported by Confluent REST Proxy

- Supported by Kafka Streams

  - …and, in turn, ksqlDB

- Supported by Kafka Connect

We'll learn about other tools in a Confluent Kafka deployment coming up in later modules. We've shown you how to integrate Java clients with SR here; other tools coming up easily integrate with SR too.

Some Notes for the Lab

Troubleshooting

If you do the Java Avro exercises and forget to `gradle build` before starting the VS Code debugger, then the classpath will get into a bad state. If this happens:

1. Open the VS Code command palette with the gear icon at the bottom left of the screen

2. Search for and select "Java: Clean Java Language Server Workspace"

3. Select the "Restart and delete" option

4. VS Code should now be in a happy state! The debugger should now work!

Generated Code

You'll use `gradle build` to generate a Java class from an Avro schema. Look to `build/generated-main-avro-java` to find the generated code.

Question:

```
▲ JAVA-PRODUCER-AVRO
  ▲ build
    ▲ generated-main-avro-java
      ▲ clients
        ▲ avro
          ☕ PositionValue.java
  ▲ src
    ▲ main
      ▲ avro
        ≡ position_value.avsc
      ▲ java
        ▲ clients
          ☕ Producer.java
```

Here is a Kafka project that uses Avro. Why might the developer have chosen **not** to include `PositionValue.java` in source control?

Answer: The Java class compiled from `PositionValue.java` can be generated from `position_value.avsc` using a "specific" Avro plugin (as opposed to generic or reflection). This makes the `.avsc` file the single source of truth for the schema, so that is what should be checked into source control.

Command Line Schema Registry Tools

There are also some command line tools that work with Schema Registry.

Here are examples of working with the Avro console producer and consumer (assuming we have defined a schema `my_schema`):

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

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

Here's the schema definition command:

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

In addition to Avro console commands, there are also Protobuf and JSON versions. Aside from needing to specify the type of schema in defining it, the commands names are the only thing that varies.

Protobuf example:

```
$ 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

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

JSON example:

```
$ 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

$ kafka-json-schema-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --from-beginning \
    --topic driver-positions-json
```
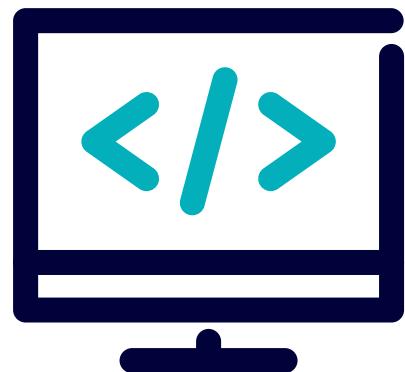
Lab: Schema Registry, Avro Producer and Consumer

Please work on **Lab 7a: Schema Registry, Avro Producer and Consumer**

Refer to the Exercise Guide

08: Introduction to Streaming and Kafka Streams

CONFLUENT
Global Education

Module Overview

This module contains four lessons:

a. What Can You Do with Streaming Applications?

b. What is Kafka Streams?

c. A Taste of the Kafka Streams DSL

d. How Do You Put Together a Kafka Streams App?

Where this fits in:

- Hard Prerequisite: Starting with Consumers

- Recommended Prerequisite: Groups and Consumers in Practice

- Recommended Follow-Up: Introduction to ksqlDB

08a: What Can You Do with Streaming Applications?

Description

Defining what streams are with a concrete conceptual example and relating streams to topics. Comparing streams and tables and which to use when. How time drives streams instead of offsets and the basic idea of windowing.

Learning Objectives
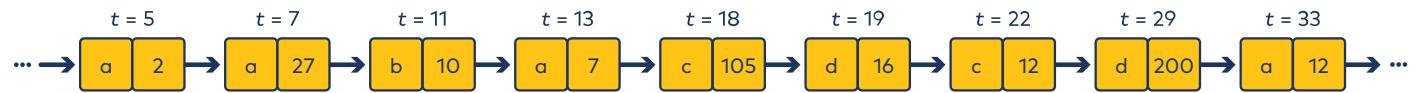
Upon completion of this lesson and associated lab exercises, you will be able to:

- Relate events in streams to records/messages in topics

A Sample Stream

t = 5	t = 7	t = 11	t = 13	t = 18	t = 19	t = 22	t = 29	t = 33
a 2	a 27	b 10	a 7	c 105	d 16	c 12	d 200	a 12

Note:

• **Events** are key/value pairs

• Ours will be sourced from Kafka topics

• **Time** is a big deal

Let's learn some basic ideas of streaming applications.

We start with streams. Streams are comprised of **events**. These are the same as in Kafka, the same thing we might call records or messages.

In turn, events are key-value pairs. Like in Kafka, the keys could be null if we don't care about value.

We'll source streams from Kafka topics. (And we'll later write them to Kafka topics.)
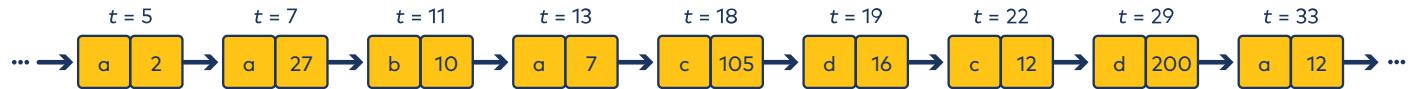
A big difference about streams from Kafka topics is that time is a big factor here. We don't really talk about offsets in the streaming world; we talk about time.

In the figure, each event is labeled with its time.

Operating on the Stream

We can do **stateless operations** on a stream, like filtering.

What would happen if we filtered to keep those records whose value exceeded 50?

$t = 5$	$t = 7$	$t = 11$	$t = 13$	$t = 18$	$t = 19$	$t = 22$	$t = 29$	$t = 33$
a 2	a 27	b 10	a 7	c 105	d 16	c 12	d 200	a 12

Here's a first example of doing an operation on a stream: a filter.

We call it stateless. It produces a new stream and doesn't alter the input
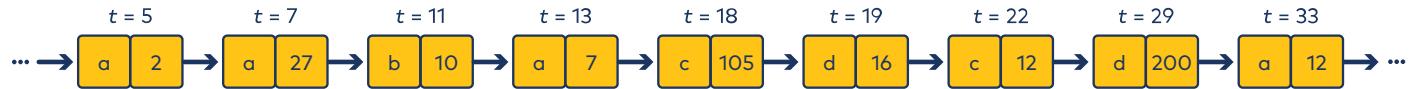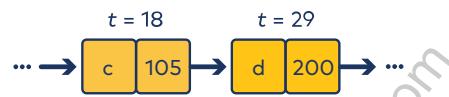
Operating on the Stream

We can do **stateless operations** on a stream, like filtering.

What would happen if we filtered to keep those records whose value exceeded 50?

Input stream:

Output stream:
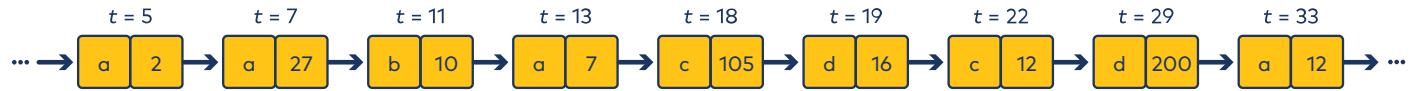
Here we see the solution to the filter example.
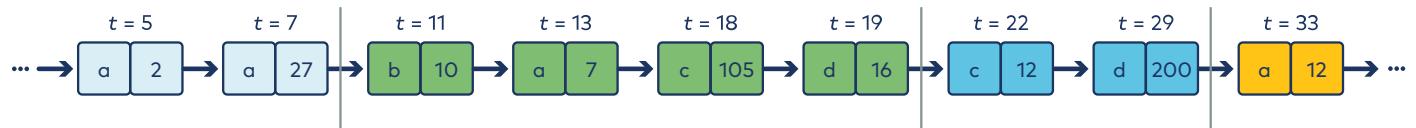
Windowing the Stream

We can split up time into windows.

Here's our input stream:

Here's the same stream, divided into windows of size 10:
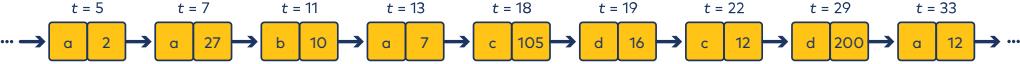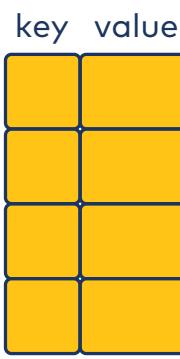
Here we see dividing a stream up into time windows. We can do various operations on the time windows, e.g., looking for counts, extrema, and other aggregations. More on this to come in its own lesson.

Both Kafka Streams and ksqlDB will have the same windowing capabilities. We'll look at windowing once we get into ksqlDB.

Stream-Table Duality

We can view a stream as a table.

Stream	Table
Records are events in time	Records are updates to same-key table entries

Each stream can be treated as a table. You can also source tables directly from Kafka topics. There are two different ways to view the same data.

Streams vs. Tables

	Stream	**Table**
Use for...	Working with events in time/history	Storing status/state
Example 1	Driver positions	Driver profiles
Example 2	Order status changes	Orders, Customers
Example 3	Ledger of sales	Sales totals

As you work through this taste of streaming applications, observe when we use streams and when we use tables. Hopefully, after you've completed this course, you'll see applications come up in your work and naturally think a stream or a table fits them—or does now. This summary is here to start you on that journey.

Activity: Choosing Between Streams and Tables

Quick Check

For each concept described, would it make more sense to represent it as a stream or as a table?

a. number of purchases per customer

b. heartbeat readings per patient coming in every 30 seconds

c. number of high heart rate events per patient observed

d. current addresses of known restaurants

Virtual Classroom Poll:

 use a stream   use a table

08b: What is Kafka Streams?

Description

The consume-process-produce model. How Kafka Streams fits into the Kafka ecosystem, what a streaming topology is, and how data gets back to Kafka. Immutability of streams. Stateless vs. stateful processing conceptually.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe the consume-process-produce model.

- Explain how Kafka Streams applications relate to Producer and Consumer applications.

- List two properties of streams.

- Define stream processor, along with source and sink processors.

- Understand a processor topology conceptually.

The Consume-Process-Produce Model

Idea:

1. **Consume** data from a topic

2. Do something with that data, i.e. **Process**

3. Generate a new record or records based on the processing and **Produce** them to a topic

Kafka Streams

- Separate application

- Tied to a Kafka cluster

- Groups

- Objects `KStream` and `KTable`

- Sourced from a Kafka topic

  - Acts as a Consumer

- Data goes out to a different Kafka topic

  - Acts as a Producer

In the last lesson, we learned about streaming applications in general. Now we turn to Kafka Streams specifically.

Kafka Streams applications act like producers and consumers. They, too, produce data to and read data from a Kafka cluster.

Unbounded, Immutable Streams

- Streams are unbounded

  ◦ Sourced from a topic

  ◦ As the topic receives new records, so does the stream

- Streams are immutable

  ◦ Never change a stream

  ◦ Instead, output a **new** stream from operations

A stream is unbounded, conceivably never ending. You could think of it as subscribing to a topic like a consumer does. A stream is sourced from a topic (or topics) in Kafka and when that underlying topic receives new messages, so does the stream. In turn, receiving new records will trigger processing on those records.

Also note that streams are immutable. Streams receive messages in one way: getting them from the source Kafka topic(s). Operations do not change streams. Go back to our filter example from earlier. The stream containing only those records with values over 50 was a **new** stream. The input stream remained and we could do other operations on it.

Processors
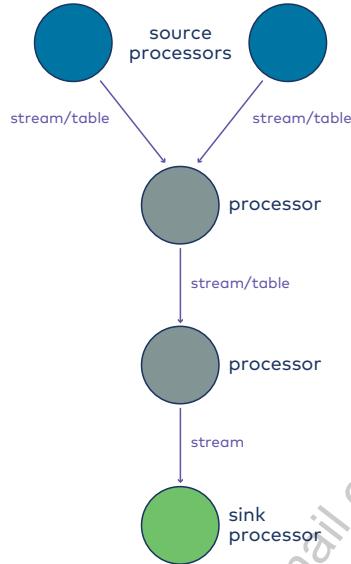
- A **stream processor** operates on stream or table

- One operation

- Examples

  - Filter a stream

  - Get a table where all cost values have tax added

- Special processors:

  - **Source** processor:

    - Read from a Kafka topic

    - Generate a `KStream` or `KTable`

  - **Sink** processor:

    - Take in a `KStream`

    - Produce each event to a Kafka topic

We call the operations on streams and tables in Kafka Streams **stream processors**. We define two special subtypes here.

Processor Topologies

Processors are put together to form a **processor topology**

A processor topology is the heart of a steaming application.

We can represent it with a directed acyclic graph (DAG) as done here:

• vertices or nodes are the processors

• edges (necessarily directed, so drawn with arrows) are the output and input streams or tables

To be even more precise, this is a type of Data Flow Diagram (DFD):

• function → processor

• input/output → source/sink processor

• flow → stream/table

• file/database → state store
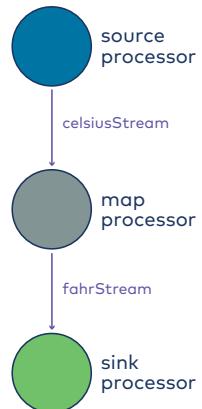
Example Stateless Application Topology

Kafka cluster has topic:

- `temp_readings`

  - keys: postal codes

  - values: temperatures in degrees C

Topology:

- Source processor creates `KStream celsiusStream` from `temp_readings`

- Processor maps each temperature in degrees C to degrees F

  - Creates `KStream fahrStream`

- Sink processor writes `fahrStream` to Kafka topic `temp_readings_fahr`

source processor

celsiusStream

map processor

fahrStream

sink processor

Here's an example of a simple streaming application topology for converting temperatures from one unit system to another.
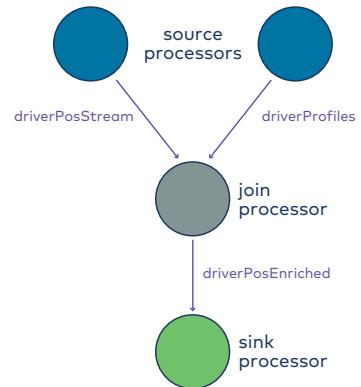
Another Example Application Topology

Kafka cluster has topics:

- `driver_profiles`

- `driver_positions`

Topology:

- Source processors create...

  - `KStream driverPosStream` from `driver_positions`

  - `KTable driverProfiles` from `driver_profiles`

- Processor joins `driverPosStream` with `driver_profiles`
  → Creates stream `driverPosEnriched`

- Sink processor writes `driverPosEnriched` to Kafka topic `driver_positions_enriched`

Here's an example of a simple streaming application topology that goes along with several of our labs. We join driver information with driver profile information.
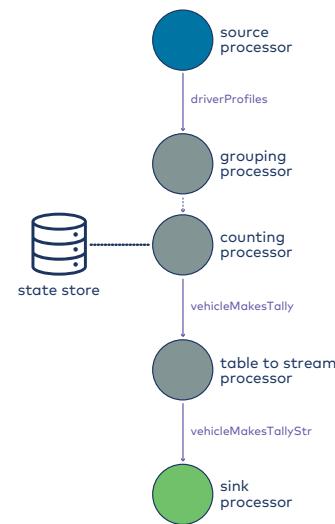
Example Stateful Application Topology

Kafka cluster has topic:

- `driver_profiles`

Topology:

- Source processor creates `KTable driverProfiles` from `driver_profiles`

- Processors group `driverProfiles` by make of vehicle and count number in each group

  - Creates `KTable vehicleMakesTally`

- Processor generates equivalent `KStream vehicleMakesTallyStr` from `vehicle_makes_tally`

- Sink processor writes `vehicleMakesTallyStr` to Kafka topic `vehicle_makes_tally_topic`

Here's another example of a simple streaming application topology that goes along with several of our labs. This time, the processing is stateful. We get the driver profiles as before, but instead, we have a processor that groups vehicles by make and generates a table. The table uses a state store on the machine running the application.

Note that in this picture, you see a grouping processor is shown as setup for the counting processor. Indeed, this is necessary, and the counting processor is the simplest example of an aggregation. While there is an intermediate object output by the counting processor and grouping processor is necessary, neither is particularly useful on its own without the counting processor, hence how they are illustrated and labeled here.

Note that we convert our table to a stream before writing it out to Kafka. This is intentional; the Kafka Streams API allows us to write streams to Kafka topics but not tables. However, it provides a simple function to take in a table and output an equivalent stream.

Activity: Reviewing Kafka Streams Concepts
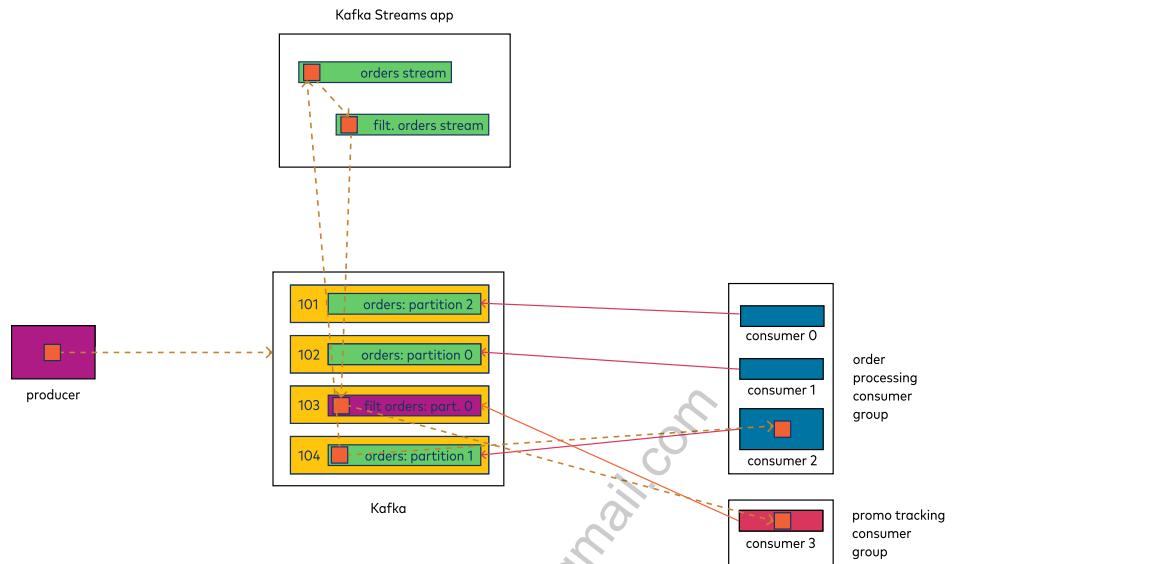
Quick Knowledge Check: Myth or Fact?

1. When we say processor in this context, we refer to all the logic of a Kafka Streams application

2. Kafka Streams applications act as consumers

3. Kafka Streams applications run on Kafka brokers

4. The input data to Kafka Streams can directly come from anywhere

5. Kafka Streams applications act as producers

6. Streams can be changed

7. A processor can take more than one stream as its input

More on Data Flow with Kafka Streams Apps

The above graphic that shows a message moving around in a possible use case:

1. A message originates on a producer, and is tagged as part of the `orders` topic. The producer's partitioner selects partition `1` for this message.

2. The message is sent to the Kafka cluster. It is written to the log for partition `1` of `orders`, which is on broker 104.

3. A Kafka Streams app is running and it has a `KStream` subscribed to the topic `orders` in the cluster. The message makes it to this stream.

4. In the topology of the Kafka Streams app, there is a `filter` processor that chooses only those orders that contain items that fit some promotion and puts them in a new filtered stream. Our message gets put into this stream.

5. The stream topology has a sink processor that writes to a new topic in Kafka for the filtered orders. That topic happens to have one partition, and that partition lives on broker 103. A copy of our message is written there.

6. We see two consumer groups:

   ◦ One for processing orders, subscribed to the original topic. Consumer 2 consumes the message from the original topic.

   ◦ One for processing orders that were filtered into the stream of orders that contain promotional items. Consumer 3 consumes the message from the topic that is the sink topic from our Kafka Streams app.

08c: A Taste of the Kafka Streams DSL

Description

Defining the DSL. Getting streams from Kafka topics in code. Examples of some stateless operations with code: `filter`, `map`(`Values`), `flatMap`(`Values`). Stateful operation examples: `groupBy` and `count`. Writing a stream back to Kafka.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Write KafkaStreams DSL code to source a stream or table from a Kafka topic.

- Write KafkaStreams DSL code to perform a simple stateless operation.

- Choose the best operation given a list of five choices and a task.

- Write KafkaStreams DSL code to perform a simple stateful operation.

- Write KafkaStreams DSL code to send a stream to Kafka.

Interacting with Kafka Streams: the DSL

Two ways to interact with Kafka Streams:

- Domain Specific Language (DSL) ← our focus

- Processor API (PAPI)

We'll look at the DSL for interacting with Kafka Streams here. It is the most common way to write Kafka Streams applications.

The PAPI allows for a bit more granularity. Both are covered in more detail in the 3-day stream processing course.

Getting Streams and Tables from Kafka

This is how we implement a source processor.

Use the `StreamsBuilder` class and operations `stream()` and `table()`.

Objects:

```
StreamsBuilder builder;
KStream<Integer, String> driverPosStream;
KTable<Integer, String>  driverProfiles;
```

Creating entities:

```
builder = new StreamsBuilder();
driverPosStream = builder.stream("driver_positions");
driverProfiles  = builder.table("driver_profiles");
```

Note here:

- Like with `KafkaProducer` and `KafkaConsumer` objects, we provide the data types of the keys and values when creating instances of `KStream` and `KTable`. (It's not a direct analogy; a `KafkaStreams` application can use multiple `KStream` and/or `KTable` objects).

- We use an instance of `StreamsBuilder` to source our streams and tables from Kafka.

Filtering a Stream (Stateless Operation)

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

```
largePurchases = purchases.filter((key,value) -> value.amount > 50.0);
```

Declaration of object for ex:

```
KStream<String, Double> largePurchases;
```

The `filter` method is the equivalent of `WHERE` in a SQL statement.

Mapping (Stateless Operation)

`mapValues`	Creates a new `KStream` by transforming the value of each element in the input stream into a different element in the output stream. Use when only transforming the value.
	
	``java
	fahrStream = celsiusStream.mapValues(value -> value*9.0/5.0 + 32);
	``

`map`	Creates a new `KStream` by transforming each element in the input stream into a different element in the output stream. Use this if you must change the key.
	
	``java
	fahrStreamByCity
	    = celsiusStream.map((key,value)
	                    ->
	                    new KeyValue<>(cityFromZIP(key),
	                                   value*9.0/5.0 + 32));
	``

Declaration of objects for ex:

```java
KStream<Integer, Double> fahrStream;
KStream<String, Double>  fahrStreamByCity;
```

The `map` method is the equivalent of `SELECT` in a SQL statement.

Note that while `map` will work when you only need to compute a new value, you should use it only when you need to compute new keys too.

Mapping, Part 2 (Stateless Operation)

`flatMapValues`	Creates a new `KStream` by transforming the value of each element in the input stream into zero or more elements in the output stream. Only changes value. ```\nfactoredStream\n = numbersStream.flatMapValues(value\n -> getPrimeFactors(value));\n```
`flatMap`	Creates a new `KStream` by transforming each element in the input stream into zero or more elements in the output stream. Use this if you must change the key.

Declaration of object for ex:

```
KStream<Integer, Integer> factoredStream;
```

The difference between these operations and the prior set is that here each input event can yield multiple output events. With `map` and `mapValues`, there is a one-to-one mapping of input to output events, whereas here, it is more general.

Activity: Transforming a Bank Account Stream

Suppose you have a stream where:

- keys of events are bank account numbers

- values of events are delimited strings containing several transactions for the account whose number is the key

You want a corresponding stream where each event:

- has a key that is a bank account number

- has a value that is a *single* transaction

Your quest:

1. Among all the functions we looked at in the DSL, which can solve this? Which can't? Which is best?

2. What would you do if you want the stream to contain only those transactions involving over $100?

Example input record and corresponding output records:

```
input:
key: 3204230930
value: 11/2/2021, Amazon, $22 | 11/3/2021, McDonald's, $205.78 | 11/3/2021, Grubhub,
$122.12


output:
key: 3204230930
value: 11/2/2021, Amazon, $22

key: 3204230930
value: 11/3/2021, McDonald's, $205.78

key: 3204230930
value: 11/3/2021, Grubhub, $122.12
```

Grouping and Counting (Stateful Example)

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

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

- For a comprehensive resource on Kafka Streams, see
https://docs.confluent.io/current/streams/developer-guide/index.html

- For API details, see https://www.javadoc.io/doc/org.apache.kafka/kafka-streams/latest/index.html

Writing A Stream to Kafka

To implement a sink processor, call the `to` method on a `KStream`.

Example:

```
fahrStream.to("temp_readings_fahr");
```

> ℹ️ A `KTable` needs to be converted to a `KStream` first

Convert a `KTable` to a `KStream` using the `toStream()` method, e.g.

```
vehicleMakesTally.toStream().to("vehicle_makes_tally_topic");
```

08d: How Do You Put Together a Kafka Streams App?

Description

The five parts of a Kafka Streams application overall. Code examples of three parts we haven't seen, then a full example of a stateless application and a stateful one.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Write KafkaStreams DSL code to source a stream or table from a Kafka topic.

- Write KafkaStreams DSL code to perform a simple stateless operation.

- Choose the best operation given a list of five choices and a task.

- Write KafkaStreams DSL code to perform a simple stateful operation.

- Write KafkaStreams DSL code to send a stream to Kafka.

Kafka Streams Application Anatomy

Five parts:

1. Imports

2. Configuration

3. Topology

4. Create and start Kafka Streams application

5. Graceful shutdown

Note that everything we looked at in the last lesson was part of the topology.

Running Example

In a prior lesson, we solved this problem...

> Suppose you have a stream where:
>
> - keys of events are bank account numbers
>
> - values of events are delimited strings containing several transactions for the account whose number is the key
>
> You want a corresponding stream where each event:
>
> - has a key that is a bank account number
>
> - has a value that is a *single* transaction

We will be working in the same context for all of our examples in this lesson.

Here we just reiterate the problem we'll solve.

SerDes

One quick note before we dive in...

- With producers, we need to specify serializers

- With consumers, we need to specify deserializers

- Streams apps both consume **and** produce...

- ... need both

- ... hence **SerDes** = **Ser**ializers and **Des**erializers

---

In Kafka Streams apps, we must allow for both serialization and deserialization, so we specify SerDes.

Example: Defining Configuration

```java
public static Properties getConfig()
{
    Properties config = new Properties();

    // Give the application a name, which must be unique in the cluster
    config.put(StreamsConfig.APPLICATION_ID_CONFIG,
      "simple-streams-example");

    // Connect to cluster
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
      "broker-1:9092, broker-2:9092");

    // Specify default (de)serializers for record keys and for record values.
    config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
      Serdes.Integer().getClass());
    config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
      Serdes.String().getClass());

    return config;
}
```

One important step in creating a Streams app is defining its configuration. We do so here in a method that returns the configuration. Note that:

- The application ID, set on Lines 6 and 7, is used to group together multiple instances for high availability and scalability.

- We specify bootstrap servers on Lines 10 and 11 as a way to identify the Kafka cluster from which this Streams app gets data and to which it writes data.

- Lines 14 through 17 show the SerDes for key and value.

  ◦ In this example, the data types will be the same for the inbound and outbound messages.

  ◦ We can specify SerDes for individual actions, but in this case, we don't have to. Hence using DEFAULT SerDes works fine.

- We often want to set it so a Streams app processes all records from the beginning, not only newly-arriving records. To do so, we'd add the config
`config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");`

ℹ	Usually it is not recommend to define default, application-scoped SerDes, unless you're sure they're the only SerDes needed for the whole application!

Example: Defining Topology (Stateless)

```
1    public static Topology getTopology()
2    {
3        StreamsBuilder builder;
4        KStream<Integer, String> delimTransStream;
5        KStream<Integer, String> indTransStream;
6        Topology result;
7
8        builder = new StreamsBuilder();
9
10       // Source processor: get stream from Kafka topic
11       delimTransStream = builder.stream("delim_transactions_topic");
12
13       // Internal processor: break up the stream
14       indTransStream = delimTransStream.flatMapValues(value -> split(value, "|"));
15
16       // Sink processor: new stream back to new Kafka topic
17       indTransStream.to("individual_transactions_topic");
18
19       // Generate and return topology
20       result = builder.build();
21       return result;
22   }
```

In this example, the `textLines KStream` object is created by consuming the topic
`delim_transactions_topic`. The stream `delimTransStream` is transformed by
`flatMapValues` to break up the grouped transaction, resulting in the `KStream` object called
`indTransStream`. That resulting `KStream` is then produced to the topic
`individual_transactions_topic`.

Stateless Example: Main Program

Here we bring together the prior two results and do Steps 4 and 5 of the anatomy:

```
1    public static void main(String[] args) throws Exception
2    {
3        // Create a streams application based on config & topology defined already
4        KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());
5
6        // Run the Streams application via `start()`
7        streams.start();
8
9        // Stop the application gracefully
10       Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
11   }
```

The Streams application as a whole can be launched just like any normal Java application that has a `main()` method.

On the previous slides, we defined the configuration and topology for the application in helper methods. Here, we pass both to the `KafkaStreams` initializer constructor to create our Streams app.

Then we start the app and provide a shutdown hook to allow it to terminate gracefully.

Tweaking the Shut Down Behavior

This is a simple presentation of a shutdown hook. A more robust implementation could do this by introducing a `CountDownLatch` with count 1 to prevent the main application thread from closing before the shutdown hook has had a chance to gracefully clean up. After the Streams app closes, the latch counts down to 0 so that the main application thread can close. Starting the Streams app would come after setting up the shutdown hook in such a case. Here's a function to implement this:

```
1 private static void setupShutdownHook(KafkaStreams streams, CountDownLatch latch)
2 {
3     Runtime.getRuntime()
4             .addShutdownHook(
5                 new Thread(() -> {streams.close();
6                                   latch.countDown();}));
7 }
```

Using this would change our `main` function to look like this:

```java
public static void main(String[] args) throws Exception
{
    KafkaStreams streams;

    Properties config = getConfig();
    Topology topology = getTopology();

    final CountDownLatch latch = new CountDownLatch(1);

    try
    {
        streams = startApp(config, topology);
        setupShutdownHook(streams, latch);
        latch.await();
    }
    catch (final Throwable e)
    {
        System.exit(1);
    }
    System.exit(0);
}
```

Example: Defining Topology (Stateful)

Now let's look at example of stateful processing. We use the same configuration, but a new topology.

```
1    public static Topology getTopology()
2    {
3        StreamsBuilder builder;
4        KStream<Integer, String> delimTransStream;
5        KStream<Integer, String> indTransStream;
6        KTable<Integer, Integer> transByAcctTally;
7        Topology result;
8
9        builder = new StreamsBuilder();
10
11       // Source processor: get stream from Kafka topic
12       delimTransStream = builder.stream("delim_transactions_topic");
13
14       // Internal processor: break up the stream
15       indTransStream = delimTransStream.flatMapValues(value -> split(value, "|"));
16
17       // Group transactions by account number and count
18       transByAcctTally = indTransStream.groupByKey()
19                                        .count();
20
21       // Sink processor: convert table to stream, then write to new Kafka topic
22       transByAcctTally.toStream()
23                       .to("acct_activity_tally_topic",
24                           Produced.with(Serdes.Integer(), Serdes.Integer()));
25
26       // Generate and return topology
27       result = builder.build();
28       return result;
29   }
```

We have a stateful application, so, you might wonder...

1. *What is stateful about this?*

   We are keeping track of a tally of transactions by account, so that state is saved.

2. *How is that state saved?*

   It uses something called a state store. Kafka state stores are implemented using a technology called RocksDB.

Let's hone in on Line 19 for a moment. An improvement might be this:

```
19                                        .count("CountsByAccount");
```

Here we use a string argument to `counts()` to specify the name of the state store. This can prove helpful in more advanced development; one reason is that if we use tools to debug our

topology, it will be more readable this way (otherwise, Kafka Streams names state stores).

Stateful Example: Main Program

Our main program is identical to the prior example; only the topology is different.

```java
public static void main(String[] args) throws Exception
{
    // Create a streams application based on config & topology defined already
    KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());

    // Run the Streams application via `start()`
    streams.start();

    // Stop the application gracefully
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

Going Further

- This is just a taste of Kafka Streams

- Consider more training:

  ◦ Instructor-led course Stream Processing using Apache Kafka® Streams and Confluent ksqlDB

- Consider reading:

  ◦ Documentation on our website

  ◦ *Kafka Streams in Action* book (William Bejeck)

Developer Guide on <u>Writing a Streams Application</u>.

Lab: Kafka Streams

Please work on **Lab 8a: Kafka Streams**

Refer to the Exercise Guide

09: Introduction to ksqlDB

CONFLUENT
Global Education

Module Overview

This module contains four lessons:

a. What Does a Kafka Streams App Look Like in ksqlDB?

b. What are the Basic Ideas You Should Know about ksqlDB?

c. How Do Windows Work?

d. How Do You Join Data from Different Topics, Streams, and Tables?

Where this fits in:

• Hard Prerequisite: Introduction to Streaming and Kafka Streams

• Recommended Follow-Up: Kafka Connect

09a: What Does a Kafka Streams App Look Like in ksqlDB?

Description

Creating a ksqlDB app analogous to an example Kafka streams application.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Read a simple ksqlDB application.

- Compare a simple ksqlDB application to an equivalent Kafka Streams application.

Recall...

Last module, we arrived at a Kafka Streams application to read in grouped transactions from a topic, split them, tally by account number, and write out to Kafka. More practically, maybe we read in individual transactions, so here's a slightly simplified version of that application:

```java
 1  public class KStreamEx
 2  {
 3      public static Properties getConfig()
 4      {
 5          Properties config = new Properties();
 6
 7          // Give the application a name, which must be unique in the cluster
 8          config.put(StreamsConfig.APPLICATION_ID_CONFIG,
 9            "simple-streams-example");
10
11          // Connect to cluster
12          config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
13            "broker-1:9092, broker-2:9092");
14
15         // Specify default (de)serializers for record keys and for record values.
16          config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
17            Serdes.Integer().getClass());
18          config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
19            Serdes.String().getClass());
20
21          return config;
22      }
23
24      public static Topology getTopology()
25      {
26          StreamsBuilder builder;
27          KStream<Integer, String> indTransStream;
28          KTable<Integer, Integer> transByAcctTally;
29          Topology result;
30
31          builder = new StreamsBuilder();
32
33          // Source processor: get stream from Kafka topic
34          indTransStream = builder.stream("transactions_topic");
35
36          // Group transactions by account number and count
37          transByAcctTally = indTransStream.groupByKey()
38                                           .count();
39
40          // Sink processor: convert table to stream, then write to new Kafka topic
41          transByAcctTally.toStream()
42                          .to("acct_activity_tally_topic",
43                              Produced.with(Serdes.Integer(), Serdes.Integer()));
44
45          // Generate and return topology
46          result = builder.build();
47          return result;
48      }
49
50      public static void main(String[] args) throws Exception
51      {
52          // Create a streams application based on config & topology defined already
53          KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());
54
55          // Run the Streams application via `start()`
56          streams.start();
57
58          // Stop the application gracefully
59          Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
60      }
61  }
```

Let's do the same thing with ksqlDB!

Get a Stream from a Kafka Topic

Here's how we get our stream from our source Kafka topic:

```
CREATE STREAM ind_trans_stream (acct_id INT KEY, amount DOUBLE, details VARCHAR)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'transactions_topic');
```

Notes:

- In ksqlDB, a `STREAM` is the same thing as a Kafka Streams `KStream`

- We'll use the `CREATE STREAM … WITH` construct to do what we did with `StreamsBuilder` and `builder.stream()`

- A `STREAM` has a key and a value, like a `KStream`

- Everything that's not the key is part of the value

  - How the value is stored is determined by `VALUE_FORMAT` in the `WITH` clause. Using `JSON` as the choice is rather common.

We can also use a `DELIMITED VALUE_FORMAT` and supply a `VALUE_DELIMITER`. Here's an example you will see in lab:

```
CREATE TABLE DRIVER (driverkey VARCHAR PRIMARY KEY, firstname VARCHAR,
                     lastname VARCHAR, make VARCHAR, model VARCHAR)
      WITH (KAFKA_TOPIC='driver-profiles-ksql',
            VALUE_FORMAT='delimited', VALUE_DELIMITER='|');
```

Write a SQL Query to Group & Count

In our streaming topology in Kafka Streams, we grouped by key and counted. Here's the equivalent in ksqlDB:

```sql
SELECT    acct_id, count(*)
FROM      ind_trans_stream
GROUP BY acct_id;
```

Indeed, this looks just like SQL. But... we also stored our results in a table. Let's do that too:

```sql
CREATE TABLE trans_by_acct_tally AS
    SELECT    acct_id, count(*)
    FROM      ind_trans_stream
    GROUP BY acct_id;
```

Here again you see a CREATE statement, this time creating as the result of a query.

The construct you see - CREATE TABLE … AS SELECT is often abbreviated "CTAS" and persists the table to memory. If we didn't do this, the results would show up in whatever UI we're executing the query, but that's it.

There exists an equivalent CREATE STREAM … AS SELECT or "CSAS."

Read on...

We're Done! Wait, what?

Yes, we're done!

What about this:

```
39          transByAcctTally.toStream()
40                         .to("acct_activity_tally_topic",
41                              Produced.with(Serdes.String(), Serdes.Long()));
```

It turns out any `TABLE` made with a CTAS is

- known to the ksqlDB server by the given name, `trans_by_acct_tally` in this case

- persisted to a Kafka topic of the same name

So we don't need to do any extra work to write to a Kafka topic!

We can consume from the topic `trans_by_acct_tally` from a Kafka consumer. We could also wire up a Kafka Connect connector to write its contents out to another system (but... stay tuned...).

The equivalent applies to streams, i.e., any `STREAM` made with a CSAS is

- known to the ksqlDB server by the given name

- persisted to a Kafka topic of the same name

Big Observation: Notice how little code we needed in ksqlDB to do the same thing we did in Kafka Streams. Now, before you get too excited, this requires that we have a ksqlDB server running and configured to connect to the correct Kafka cluster, but we don't do the connection on a per-app basis and group management is done at the server level, not at the app level.

Compare...

Our Kafka Streams app read in individual transactions, so here's a slightly simplified version of that application:

```java
 1  public class KStreamEx
 2  {
 3      public static Properties getConfig()
 4      {
 5          Properties config = new Properties();
 6
 7          // Give the application a name, which must be unique in the cluster
 8          config.put(StreamsConfig.APPLICATION_ID_CONFIG,
 9            "simple-streams-example");
10
11          // Connect to cluster
12          config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
13            "broker-1:9092, broker-2:9092");
14
15          // Specify default (de)serializers for record keys and for record values.
16          config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
17            Serdes.Integer().getClass());
18          config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
19            Serdes.String().getClass());
20
21          return config;
22      }
23
24      public static Topology getTopology()
25      {
26          StreamsBuilder builder;
27          KStream<Integer, String> indTransStream;
28          KTable<Integer, Integer> transByAcctTally;
29          Topology result;
30
31          // Source processor: get stream from Kafka topic
32          indTransStream = builder.build("transactions_topic");
33
34          // Group transactions by account number and count
35          transByAcctTally = indTransStream.groupByKey()
36                                          .count();
37
38          // Sink processor: convert table to stream, then write to new Kafka topic
39          transByAcctTally.toStream()
40                      .to("acct_activity_tally_topic",
41                          Produced.with(Serdes.String(), Serdes.Long()));
42
43          // Generate and return topology
44          result = builder.build();
45          return result;
46      }
47
48      public static void main(String[] args) throws Exception
49      {
50          // Create a streams application based on config & topology defined already
51          KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());
52
53          // Run the Streams application via `start()`
54          streams.start();
55
56          // Stop the application gracefully
57          Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
58      }
59  }
```

Here's the ksqlDB app equivalent to the above Kafka Streams app:

```sql
CREATE STREAM ind_trans_stream (acct_id INT KEY, amount DOUBLE, details VARCHAR)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'transactions_topic');

CREATE TABLE trans_by_acct_tally AS
    SELECT   acct_id, count(*)
    FROM     ind_trans_stream
    GROUP BY acct_id;
```

Two More Examples

Let's do just a little more comparison, bring back some examples from the Kafka Streams module.

We talked about filtering, so here's a simple filter:

```
CREATE STREAM big_transaction_amounts AS
    SELECT   acct_id, amount
    FROM     ind_trans_stream
    WHERE    amount > 100
```

And here's filtering *and* aggregating together:

```
CREATE TABLE trans_by_acct_tally AS
    SELECT   acct_id, count(*)
    FROM     ind_trans_stream
    WHERE    amount > 100
    GROUP BY acct_id;
```

Activity: ksqlDB First Impressions

Rapid Response!

Take one minute for this.

Grab a post-it note and write down one thing you learned about ksqlDB from the examples in this lesson.

09b: What are the Basic Ideas You Should Know about ksqlDB?

Description

What ksqlDB is, how it fits in, persistent vs. non-persistent queries, push vs. pull queries, and summarizing basic syntax.
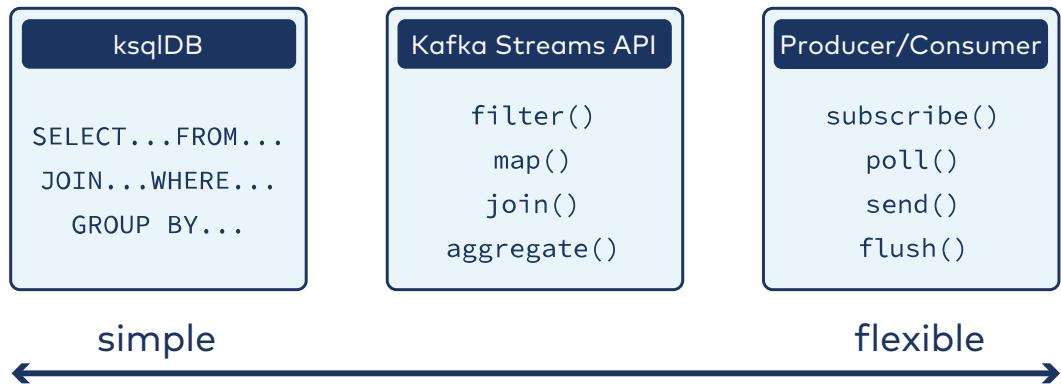
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe where Kafka Streams and ksqlDB fit in with the Producer and Consumer APIs.

- List a few basic things one can do with ksqlDB SQL.

- Make a ksqlDB query persist.

- Differentiate between push and pull queries and write basic of each.

- Interact with ksqlDB via the ksqlDB CLI.

Introducing ksqlDB

So, we've seen we can do some things we can do with Kafka Streams simpler with ksqlDB.

ksqlDB	Kafka Streams API	Producer/Consumer

`SELECT...FROM...`
`JOIN...WHERE...`
`GROUP BY...`
	`filter()`
	`map()`
	`join()`
	`aggregate()`
		`subscribe()`
		`poll()`
		`send()`
		`flush()`

simple   flexible

←——————————————————————————————————————→

We learned about the producer and consumer APIs before to write to and read from Kafka. This is where it all started and the most basic way to interact with Kafka. It is also the way that gives you the most control: with the producer and consumer APIs, you have a lot of flexibility in the kinds of things you can do with your data.

In the last module, we learned about Kafka Streams. Kafka Streams is built on top of the producer and consumer APIs and thus brings along their benefits. It allows you to interact with the data in Kafka just like producers and consumers, but:

- You can achieve an objective with Kafka Streams using less code than core producers and consumers

- What you're trying to do with Kafka Streams has to be achievable with with DSL or PAPI; put differently, you cannot do everything you can do with producers and consumers.

Here, we introduce ksqlDB, which evolved from KSQL. While ksqlDB is current and powerful, we specifically mention KSQL here, as it was initially created as a more accessible way of doing what you can do with Kafka Streams. Honing in on that for now, as you saw in the last lesson where we wrote ksqlDB SQL code to do the same thing as a KafkaStreams application...

- You can achieve an objective in ksqlDB SQL that you can with Kafka Streams

- What you're trying to do with ksqlDB SQL has to be something you can express in a SQL-like syntax; this limits you further [1]

Summarizing, we get a continuum visualized on this slide:

- As you go right, you write more code to do something, but there are more things you can

achieve

- As you go left, you write less code to do something, but there are fewer things you can achieve

We strive to expose you to the three options in this course and equip you to choose the tool that lets you solve the problems you are trying to solve.

ksqlDB SQL

- Designed to be accessible to you if you've worked with most flavors of SQL

- Standard `SELECT`, `FROM` clauses

- Standard aggregation with `GROUP BY`

- Standard filtering with `WHERE` and `HAVING`

- Many popular scalar functions

- More functions specific to ksqlDB, e.g., `EXTRACTJSONFIELD(…)`

- Working with `STREAM` and `TABLE` objects, not just tables

- Saving `STREAM` and `TABLE` objects…

- Windowing capabilities…

- Joining capabilities…

You can view the [full ksqlDB documentation](#) for more details.

You can specifically view [a list of all functions ksqlDB supports](#).

Push vs. Pull Queries

Push Queries a.k.a. Continuous Queries	Pull Queries a.k.a. Point-in-Time Queries
• Does what Kafka Streams can do... • .. but with ksqlDB syntax • As stream gets new results, so does query • Needs `EMIT CHANGES`	• Look up value in *materialized table* • One result set

This the first of two ways of classifying queries.

Push queries are the kind that do what you could do with a Kafka Streams application but with a SQL-like syntax.

Pull queries let you look up data at a given point in time. Note that pull queries are a feature that is new to ksqlDB and was not in KSQL.

Our examples in this module will all be push queries.

Note that even though push queries are called *push* queries, under the hood, they're using Kafka Streams streams and using the standard *pull* architecture of Kafka.

Push Queries: Non-Persistent vs. Persistent Queries

We can further break down **push queries**:

Non-Persistent Queries	Persistent Queries
• Results shown in CLI or Confluent Control Center	• Results are persisted to a named `STREAM` or `TABLE`
• Results don't get saved anywhere	• Created with one of:
• `SELECT`	◦ `CREATE STREAM … AS SELECT`
	◦ `CREATE TABLE … AS SELECT`
	• `STREAM` or `TABLE`…
	◦ …can be queried in another statement
	◦ …backed up to Kafka topic with same name

This the second of two ways of classifying queries. You may recall in our transition lesson on the "Write a SQL Query to Group & Count" slide, we started with a `SELECT` statement that just displayed results and then adapted it to save them too. We went from a non-persistent query to a persistent query.

Note that with persistent queries, the `STREAM` or `TABLE` is backed up to Kafka with a topic of the same name. You don't need a separate step to write out to Kafka like with Kafka Streams. You can query that topic directly.

Going Further

We have scratched the surface in this lesson. Next, we will...

1. Do a lab to play with the basics of ksqlDB interactively

2. Learn about windowing in ksqlDB (...and Kafka Streams)

3. Learn about joins in ksqlDB (...and Kafka Streams and Kafka)
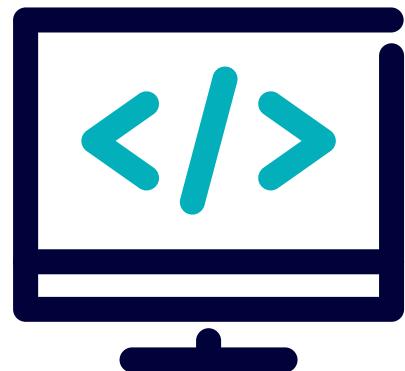
Beyond this course, there is much more depth on ksqlDB and Kafka Streams in the course Stream Processing using Apache Kafka® Streams and Confluent ksqlDB.

Lab: ksqlDB Exploration

Please work on **Lab 9a: ksqlDB Exploration**

Refer to the Exercise Guide

09c: How Do Windows Work?

Description

The three kinds of windows and comparing them. Examples that illustrate tumbling vs. hopping. Code examples in Kafka Streams and ksqlDB.

Learning Objectives
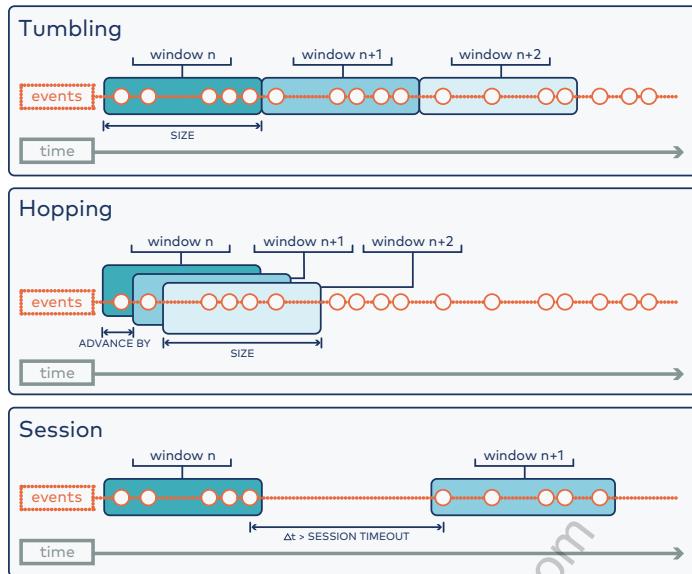
Upon completion of this lesson and associated lab exercises, you will be able to:

- Compare tumbling and hopping windows.

- Give a use case for each.

Windowing Modes in Kafka Streams and ksqlDB

| ℹ️ | ksqlDB is built on Kafka Streams → similar windowing behavior in both |

We see here three kinds of windows supported by Kafka Streams and ksqlDB.

We'll go deeper into the first two on the coming slides.

As for session windows, a typical use case is analyzing user behavior where periods of inactivity indicate different sessions of interacting with a web site or software in general.

Note: Kafka Streams supports a fourth type of window: sliding, but ksqlDB does not (as of 2022-03-23). This is meant to be a brief introduction to windowing, so we leave that for the Stream Processing with Apache Kafka Streams and Confluent ksqlDB course, along with a deeper treatment of session windows.

Tumbling Windows Example

In ksqlDB SQL, to do windowing, we simply add a `WINDOW` clause to a query.

We specify the type of window in the clause, e.g.,

```
CREATE TABLE song_sales_report AS
    SELECT song, artist, count(*)
    FROM song_sales_stream
    WINDOW TUMBLING (SIZE 604800 SECONDS)
    GROUP BY song
    LIMIT 10;
```

Here, `song_sales_stream` tracks sales of songs on a digital music purchasing service.

This query gives a weekly report of songs and their sales.

Weekly top songs lists or "top anything" lists are common; tumbling windows can generate them.

We show here only 10 results. Note that this is not a *top 10*; generating such a report would require use of the `TOPK` function. (We cover this in the course Stream Processing using Apache Kafka® Streams and Confluent ksqlDB.)

One could say tumbling windows has the following semantic: group everything "every x amount of time" / "by x amount of time."

Another Example: Fraud Detection

Consider the following query and example:

Query:

```sql
CREATE TABLE possible_fraud AS
    SELECT     card_number, count(*)
    FROM       authorization_attempts
    WINDOW     TUMBLING (SIZE 5 SECONDS)
    GROUP BY   card_number
    HAVING     count(*) > 3
    EMIT CHANGES;
```

Example timeline:

(Each * represents an authorization attempt.)

Question: Will the query report possible fraud?

Virtual Classroom Poll:

✅ fraud detected   ❌ fraud NOT detected

This slide goes hand-in-hand with the next.

Another Example: Fraud Detection, continued

Let's change the query to use **hopping** windows instead:

```
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
    EMIT CHANGES;
```

Question: Will the query report possible fraud?

Virtual Classroom Poll:

✅ fraud
detected

❌ fraud NOT
detected

Here we see the same query using tumbling windows and hopping windows. From what you learn from the discussion, which is better and why? When do hopping windows apply?

One could say that hopping window has the following semantic: group together what happened "in the last *x* amount of time" (with a new window every *y* time).

Kafka Streams Windowing Example & Beyond

We can do the same thing in Kafka Streams. It requires more details, but, assuming we have a special entity called a `KGroupedStream` called `grouped_auth_attempts`, it might go like this:

```
1  possible_fraud
2      = grouped_auth_attempts.windowedBy(TimeWindows.of(Duration.ofSeconds(5))
3                                                       .advanceBy(Duration.ofSeconds(1))
4                          .aggregate(...);
```

> ℹ️ The details of `aggregate` are covered in our 3-day Stream Processing with Apache Kafka® Streams and Confluent ksqlDB course, along with more details on windowing, like session windows.

As noted before, ksqlDB is built on top of Kafka Streams. So, here were see how to write the credit card fraud example using Kafka Streams.

Activity: Applying Windowing

Think of an example of an application in your work for which either tumbling or hopping windows would make sense over the other and why.

09d: How Do You Join Data from Different Topics, Streams, and Tables?

Description

What a join is, what it looks like in ksqlDB, what it looks like in Kafka Streams, and co-partitioning requirements of underlying topics.

This is an important lesson, even if you don't think you're using streams. While this appears with ksqlDB, this lesson addresses relating data across topics in consumers and Kafka Streams apps as well - and important partitioning considerations you should keep in mind in designing topics.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain requirements of topics for joining data, regardless of where that join is done.

- Join a stream with a table.

- Relate joining between ksqlDB and KafkaStreams and core Kafka.

Joins

- Idea: want to bring together…

  a. related information from a topic and another topic

  b. a stream and another related stream

  c. a stream and a related table

  d. two related tables

- Easy to do in Kafka Streams

- Easier to do in ksqlDB

- Different kinds of joins available

This lesson is designed to tell you about joins and why they're useful, as well as how your data needs to be structured in Kafka topics before you get into streaming platforms. For more on the different kinds of joins, see our Stream Processing with Apache Kafka Streams and Confluent ksqlDB course.

Stream-Table Join Example: Setup

Suppose we have Kafka topics containing information about drivers and where they are. Let's bring them into our streaming world...

Table of driver information - records give *state* of the driver:

```
CREATE TABLE driver_profiles (driver_id INT PRIMARY KEY, name VARCHAR, postal_code
VARCHAR, ...)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'driver_profiles');
```

Stream of driver positions - events are where driver is at each point *in time*:

```
CREATE STREAM driver_positions (driver_id INT KEY, latitude DOUBLE, longitude DOUBLE)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'driver_positions');
```

🛈	Each event implicitly has a time associated with it.

Nothing new here; just two more examples of sourcing streams and tables from Kafka. But, we do this here to set up the example to come.

FYI: TIME info in ksqlDB 0.20.0: https://www.confluent.io/blog/ksqldb-2-0-introduces-date-and-time-data-types/

Stream-Table Join Example: Join

Now let's bring the two together:

```
CREATE STREAM driver_positions_enriched AS
    SELECT    pos.driver_id,
              pos.latitude,
              pos.longitude,
              prof.name,
              prof.postal_code,
              ...
    FROM      driver_positions pos LEFT JOIN driver_profiles prof
              ON pos.driver_id = prof.driver_id
    EMIT CHANGES;
```

ℹ️	Again, each event implicitly has a time associated with it.

We saw this example in our conceptual look at streaming topologies in the Kafka Streams module. Here's the ksqlDB code to join driver position info with driver profile info.

Requirements (1)

Recall the module Groups, Consumers, and Partitions in Practice.

Question: What was the point of the **range** partition assignment strategy?

Question: What needed to be true of the topics whose partitions were involved?

Requirements (2)

Co-partitioning!

To join two entities, they must:

- have the same number of partitions

- use the same partitioning strategy

- use the same set of keys

⚠️	This applies to the underlying Kafka topics used to source our streams or tables.

It's not just joining in streaming; it's design from the start. If you plan on relating two topics with consumers or Kafka Streams or ksqlDB or [insert something else that may not even exist], you should be thinking about how you partition those topics from the start. If you want to bring them together ultimately, how they are structured will need to be the same to do so. Do you have them designed that way from the start? Or do you repartition later (beyond the scope of this course)—at a cost of performance and effort (and is it possible)?

Stream-Stream Inner Join in ksqlDB

We can also join a stream with a stream, e.g.

```
CREATE STREAM need_a_name_here AS
    SELECT *
    FROM   stream_a a JOIN stream_b b
           ON a.key = b.key
           WITHIN 5 seconds
    EMIT CHANGES;
```

Note that when joining a stream with a stream, we need to specify a window. This example shows that.

Why do we need to specify a window? Stream events are processed in real-time, as they appear. Imagine that you have a marker on one hand, and its cap on the other, then you throw them up and try to catch them to "join" them. What's the probability of catching them both at exactly the same time? Chances are that you're going to catch one after the other; if you can catch them in a short period of time then you can join them (this is the join window).

Note again, the takeaway from this lesson is less about the syntax and more about the capabilities of streaming applications and how you should plan your Kafka topic design from the start if you hope to join topics in any way.

Check out the [documentation](#) for more on this.

Stream-Stream Inner Join in Kafka Streams

We can do the same thing in Kafka Streams.

Say we have the following and we build them as normal:

```
1 KStream<...> streamA;
2 KStream<...> streamB;
```

Then the following code shows how to join the streams:

```
15 joinedStream = streamA.join(streamB,
16                             (valA, valB) -> doStuff(valA, valB),
17                             JoinWindows.of(Duration.ofSeconds(5)),
18                             Joined.with(...));
```

Don't worry about all of the details here, but note that:

- We use a lambda to specify what the value of joined records will be.

- Records from the left stream and the right stream will be in the joined output if...

  ◦ They have the same key (implicit)

  ◦ They occur within 5 seconds—as specified in Line 17—of one another

- Joining uses state stores under the hood to do its work, hence we specify serdes in Line 18. In order, we give series for...

  1. The key—the same type for both streams

  2. The value of events in the left stream

  3. The value of events in the right stream

For more on joins in Kafka Streams and ksqlDB, consider our Stream Processing with Apache Kafka® Streams and Confluent ksqlDB course.
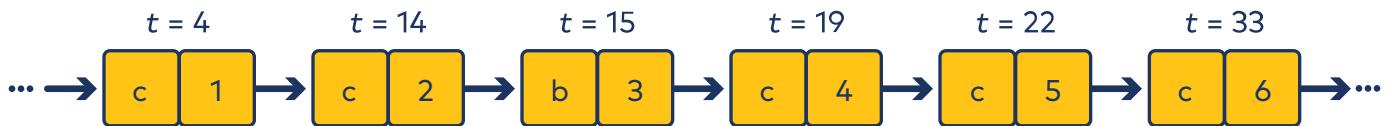
Enrichment: More on Timing

You may be curious about the timing issue. Here's a concrete example.

Suppose this is our left stream:

	t = 11	t = 13	t = 18	t = 19	t = 22	t = 29
···→	b 10 →	a 7 →	c 105 →	d 16 →	c 12 →	d 200 → ···

Suppose this is right stream:

	t = 4	t = 14	t = 15	t = 19	t = 22	t = 33
···→	c 1 →	c 2 →	b 3 →	c 4 →	c 5 →	c 6 → ···

Question: If we consider the record `c/105` at time 18 and have a join window of time 5, which records in the right stream will be joined with it?

Answer: We look backward 5 time units - so the lower bound of the window is 13 - and we look forward 5 time units - so the upper bound of the window is 23. Join windows are inclusive of both endpoints. We're looking in that window for records with the key of `c`, so here are our matches:

```
c/105 joined with c/2
c/105 joined with c/4
c/105 joined with c/5
```

Note that with `JoinWindows.of()`, the join looks for records with the same key both *before* and *after* the time of each event in the left stream for matching events in the right stream. If you wanted to look in one direction only, you could alternatively use `JoinWindows.after()` or `JoinWindows.before()`.

Enrichment: Even More on Timing

Here's another interesting analogy for why time matters in joins…

Let's think about this scenario:

• We all know that washing machines hide socks when you wash them. Now, being developers, we are a bit deranged, so… suppose we had tagged each pair of socks like "left sock" and "right sock." We wash them separately in two washing machines, and we get two separate heaps of socks, namely the left and the right heap.

• What should we do now? Well, we have to recreate all the pairs, hence get a left sock and find the corresponding right sock.

- Because we are hyper-technological, we have two treadmills and two automated robot arms that pick a sock at a time from each of the sock heaps and put them on the corresponding treadmill - the left and the right.

- The treadmills move in such a way that after a while, each sock on them will fall down into a rubbish bin if we don't pick it. Of course, we can pick each of them, if there's a corresponding sock on the other treadmill. If this is the case, we pick the two socks, recreate the pair, and put the pair on a third treadmill - namely, the "emit" treadmill that will eventually bring the pair into the sock drawer.

So this is another way of thinking about how a stream-stream join works: the socks heaps represent the two streams, the robot arms show you that you pick constantly one sock at a time from the stream, and the period of time the sock spends on the treadmill is the window of time it has to find a match. Either the left or the right sock can come first, and each has a "treadmill" life-span, to find a match on the other treadmill.

The only difference is that these socks are kind of "magic": even if picked from the treadmill to create a pair, they keep staying ALSO on the treadmill in case ANOTHER matching sock will appear on the other treadmill during its lifespan, so it can create another pair.

Activity: Designing for and Applying Joins

Scenario: You ultimately want to join information about temperature readings that periodically come in with weather stations. Let's begin with the setup in Kafka and then jump to ksqlDB...

Question 1: Complete these two commands:

```
kafka-topics
    --bootstrap-server kafka1:9092, kafka2:9092, kafka3:9092
    --create
    --partitions ____
    --replication-factor 1
    --topic stations
```

```
kafka-topics
    --bootstrap-server kafka1:9092, kafka2:9092, kafka3:9092
    --create
    --partitions ____
    --replication-factor 1
    --topic readings
```

Question 2: What would you use as keys for `stations`? What would you use as keys for `readings`?

Question 3: Complete these two commands:

```
CREATE ____ station_info (____ INT PRIMARY KEY, city VARCHAR,
country VARCHAR, narrow_loc VARCHAR)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'stations');
```

```
CREATE ____ readings_stream (___ INT KEY, temp INT)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'readings');
```

Question 4: Complete this command:

```
CREATE STREAM temp_readings_enriched AS
    SELECT    s.city,
              s.country,
              s.narrow_loc,
              r.temp
    FROM      readings_stream r ____ station_info s
              ON r.____ = s.____
    EMIT CHANGES;
```

[1] Don't give up on ksqlDB right away if you think something *could* fit a SQL-like style but you don't expect there's a built-in function. ksqlDB has something called user-defined functions (UDFs) that you can add.

10: Starting with Kafka Connect

CONFLUENT

Global Education

Module Overview

This module contains three lessons:

    a. What Can You Do with Kafka Connect?

    b. How Do You Configure Workers and Connectors?

    c. Deep Dive into a Connector & Finding Connectors

Where this fits in:

- Hard Prerequisite: Groups and Consumers in Practice

- Recommended Prerequisite: Introduction to ksqlDB

- Recommended Follow-Up: Applying Kafka Connect

10a: What Can You Do with Kafka Connect?

Description

Motivating what Connect can do and why to use it over self-made solutions. Motivating how it can "factor out" common behavior yet leverage Connectors. Connectors vs. tasks vs. workers. Relating Connect to other components of Kafka and how it works at a high level, e.g., scalability, converters, offsets.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- List two or three tasks one could achieve with Kafka Connect.

- Justify why to use Kafka Connect over Producers or Consumers.

- Explain what a connector is.

Wanted: Copy Database Table into Kafka

Given: Customer information in a table in a MySQL database.

Goal: Get that data into a Kafka topic `customers`.

Question: Could you do this using what you've learned in this course so far and anything else you know about Java?

You can solve this problem using

- JDBC to read from the database table

- The `KafkaProducer` API to produce the rows to the topic.

Wanted: Copy CSV File Info Into Kafka

Given: Weather station information stored in a flat file, where each row describes one station.

Goal: Get that data into a Kafka topic `stations`.

Question: Could you do this using what you've learned in this course so far and anything else you know about Java?

You can solve this problem using

- A file stream in Java to read from the file.

- The `KafkaProducer` API to produce the rows to the topic.

Hmm... this sounds familiar. In fact, it's more or less the same problem as the last. The only thing that's different is where the data came from. Kafka Connect "factors out" the common logic for doing the copying. Plugins called **connectors** handle what is specific to different external systems, sources in this case.

Challenges with Writing Your Own Producer to Copy Data

So, you can write your own producers to copy data from another system to Kafka.

You can write your own consumers to copy data from Kafka to another system.

⚠️	BUT...
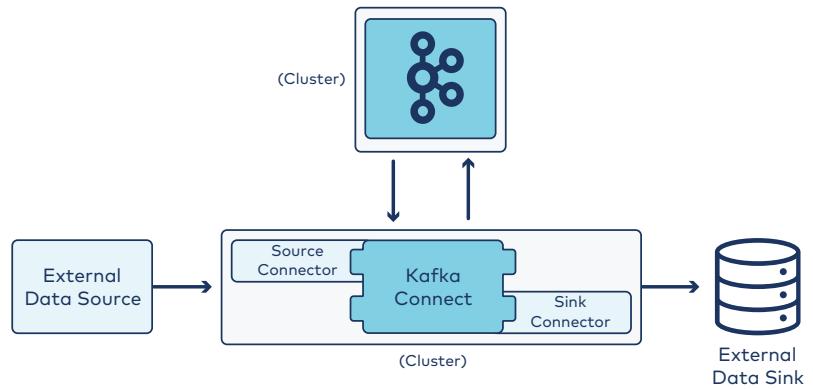
- It takes time
- You might miss an edge case
- You might miss a bug
- You'd write an application for each external system.

Kafka Connect to the Rescue!

Kafka Connect does the work for us!

All copying behavior is in Kafka Connect.
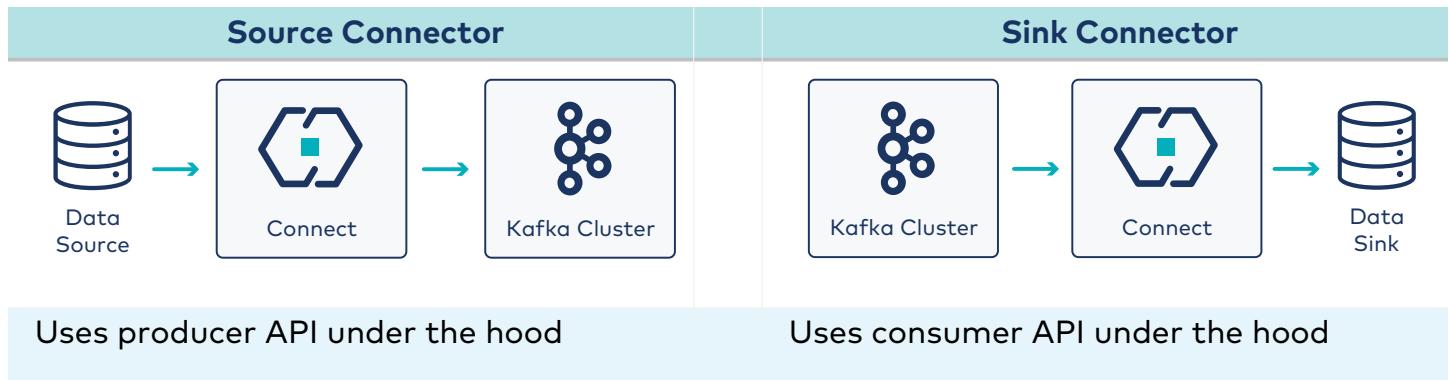
Plugins called **Connectors** contain the logic specific to particular external systems.

The Kafka Connect API is part of core Kafka.

Sources and Sinks

Two kinds of connectors...

Source Connector	Sink Connector
Uses producer API under the hood	Uses consumer API under the hood

We note the two kinds of Connectors here.
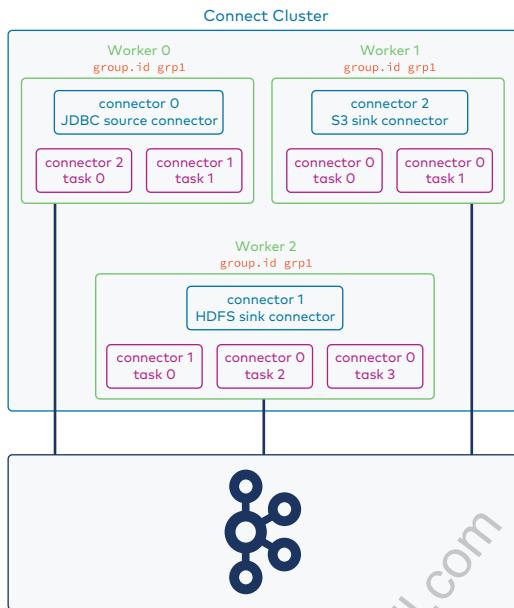
In fact, Kafka Connect is built on top of what we already know - producers and consumers.

Players in the Kafka Connect World

Kafka Connect

logic for copying regardless of the external system

Connector

logic for copying specific to/from a given external system

Task

unit of parallelism into which connector copying logic is broken up

Worker

process that runs connectors and/or tasks

- A connector has
  - one or more tasks
- A worker runs
  - zero or more connectors
  - zero or more tasks
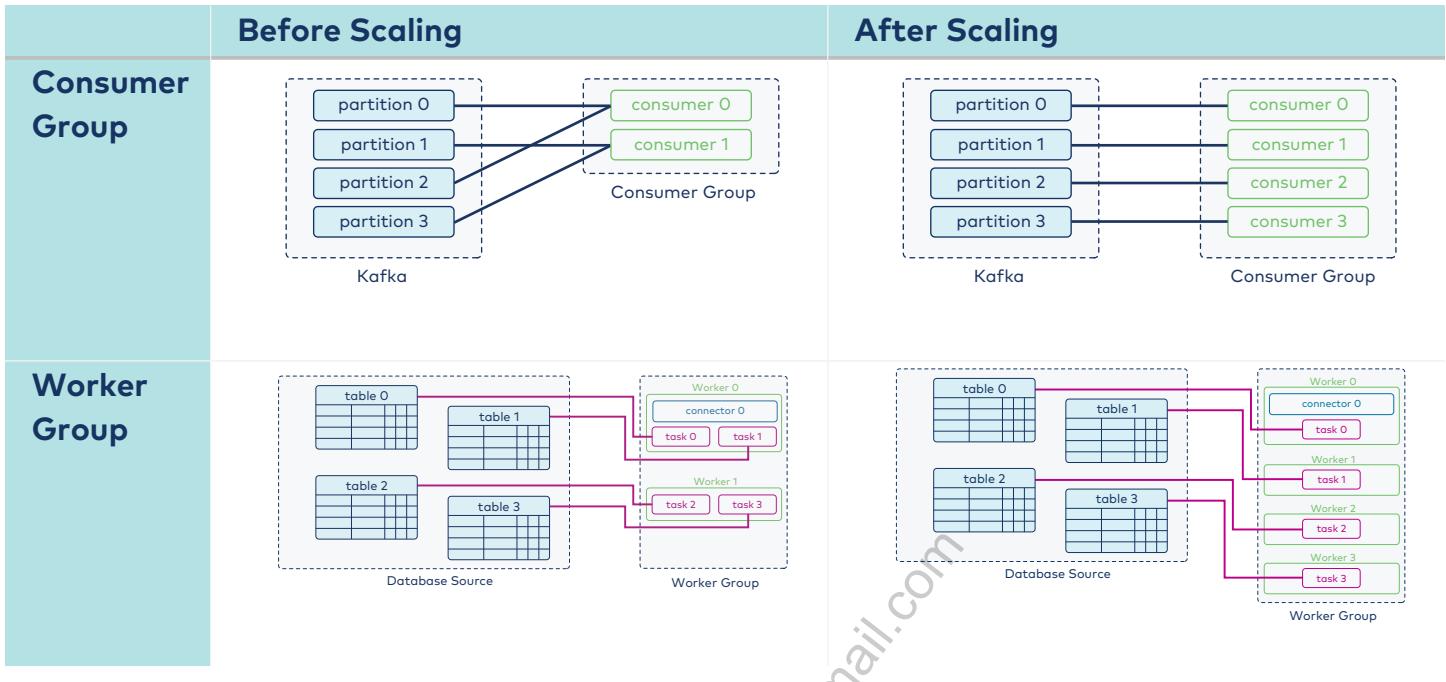
We'll use these terms throughout the module. An objective of this lesson is to learn them and the relationships between them. The activity at the end of the lesson will reinforce this.

Example of a Connect Cluster

Here is an example illustrating a Connect cluster. We see workers running connectors - both source and sink connectors - and tasks. Kafka's group management protocol handles which connector(s) and task(s) are running on each worker.

Groups Again!

	Before Scaling	**After Scaling**
Consumer Group		
Worker Group		
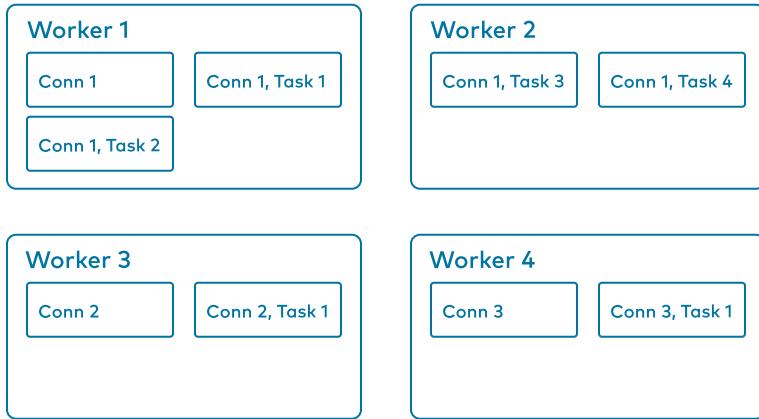
> ℹ️ Like with consumers, we can add workers to groups and get automatic balancing.

Recall the module Groups, Consumers, and Partitions, especially the first two lessons. Workers live in groups just like consumers, and group management happens in the same way. There is automatic rebalancing when we scale up or a worker dies. Workers heartbeat to Kafka in the same way as consumers, governed by the same `heartbeat.interval.ms` and `session.timeout.ms` settings. (Defaults are 3 and 45 seconds, respectively, for consumers, but there is the inconsitency that the default `session.timeout.ms` for workers is 10 s.).
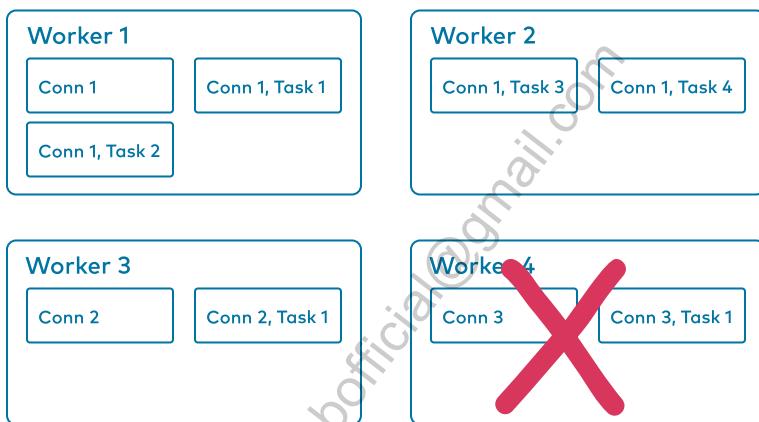
Technically, the workers are running tasks, which in turn are connected to the tables.

We'll go into configuring how many tasks in the next section.
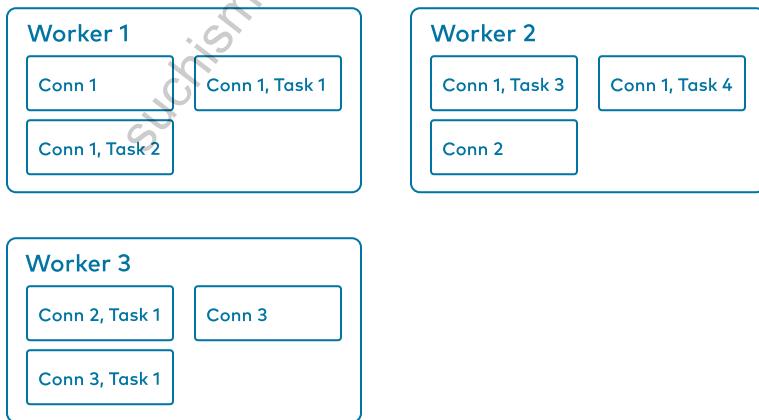
Example: Suppose we start with a situation like this:

Worker 1

| Conn 1 | Conn 1, Task 1 |
| Conn 1, Task 2 | |

Worker 2

| Conn 1, Task 3 | Conn 1, Task 4 |

Worker 3

| Conn 2 | Conn 2, Task 1 |

Worker 4

| Conn 3 | Conn 3, Task 1 |

Then a worker fails:

Worker 1

| Conn 1 | Conn 1, Task 1 |
| Conn 1, Task 2 | |

Worker 2

| Conn 1, Task 3 | Conn 1, Task 4 |

Worker 3

| Conn 2 | Conn 2, Task 1 |

Worker 4

| Conn 3 | Conn 3, Task 1 |

Automatic rebalancing might yield this:

Worker 1

| Conn 1 | Conn 1, Task 1 |
| Conn 1, Task 2 | |

Worker 2

| Conn 1, Task 3 | Conn 1, Task 4 |
| Conn 2 | |

Worker 3

| Conn 2, Task 1 | Conn 3 |
| Conn 3, Task 1 | |
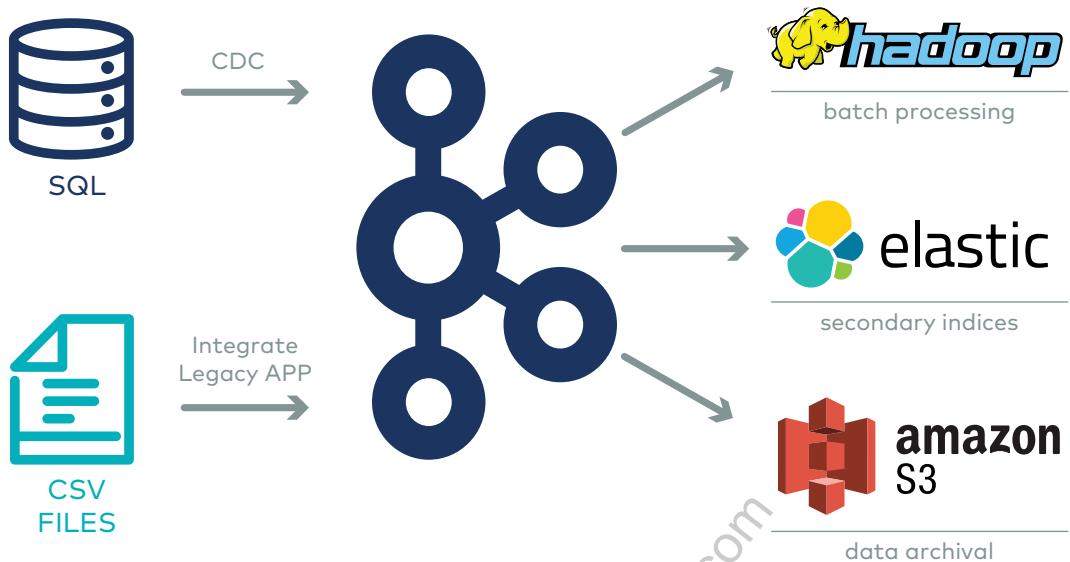
Powered by Kafka, and Behaving Like Kafka

- Same group management protocol

  ◦ Failure detection

  ◦ Scaling up and down

- Producer and consumer under the hood

- Sink connectors maintain consumer offsets → **sink offsets**

- Source connectors track **source offsets**

- Data must be serialized and deserialized → **converters**

Here we see many of the ideas we know from producers and consumers showing up in the Kafka Connect world.

Use Cases

- Example use cases for Kafka Connect include:
  - Stream an entire SQL database into Kafka
    - Bulk - load entire table
    - Change data capture (CDC) - load table changes as they happen
  - Import CSV files generated by legacy app into Kafka
  - Stream Kafka Topics into Hadoop File System (HDFS) for batch processing
  - Stream Kafka Topics into Elasticsearch for secondary indexing
  - Archive older data in low cost object storage
    - e.g., Amazon Simple Storage Service (S3)

Activity: Reviewing Kafka Connect Concepts

Quick Matching Game

For each item on the left, identify which items on the right apply

1. Connector
2. Task
3. Worker
4. Converter

a. unit of parallelism

b. can be part of a group

c. like a serializer

d. relates to one or more tasks

e. like a deserializer

f. specific to an external system

g. could run a connector

ⓘ Not all connectors support multiple tasks and parallelism. For example, the **syslog** source connector only supports one task.

10b: How Do You Configure Workers and Connectors?

Description

Configuration of workers in distributed mode and configuration of connectors in general. Quick overview of standalone mode differences.
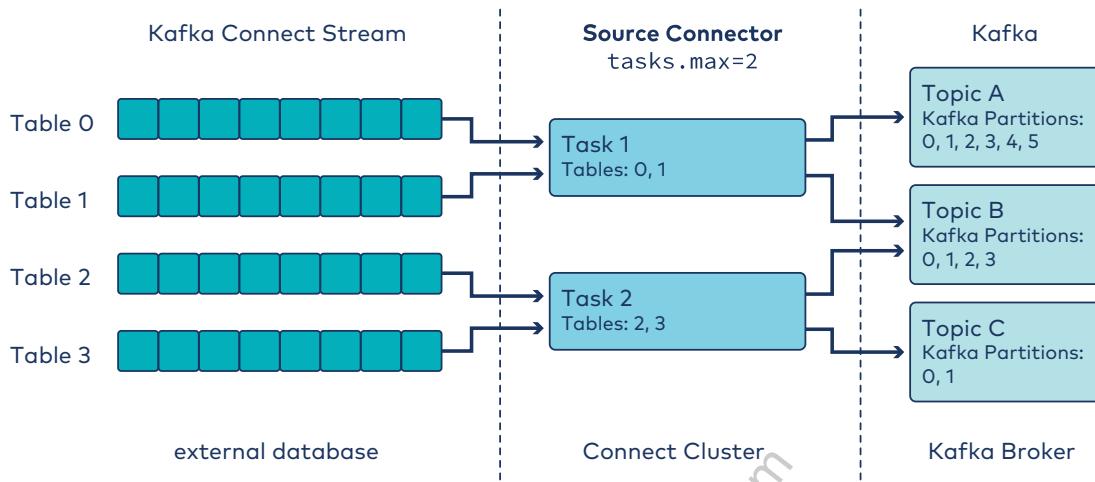
Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain how tasks and workers relate to connectors.

- List some performance benefits of using Kafka Connect.

- Explain how partitioning applies to source connectors.

- List and select some appropriate connector configurations.

- List and select some appropriate worker configurations.

- Contrast standalone mode with distributed mode.

Providing Parallelism & Scalability
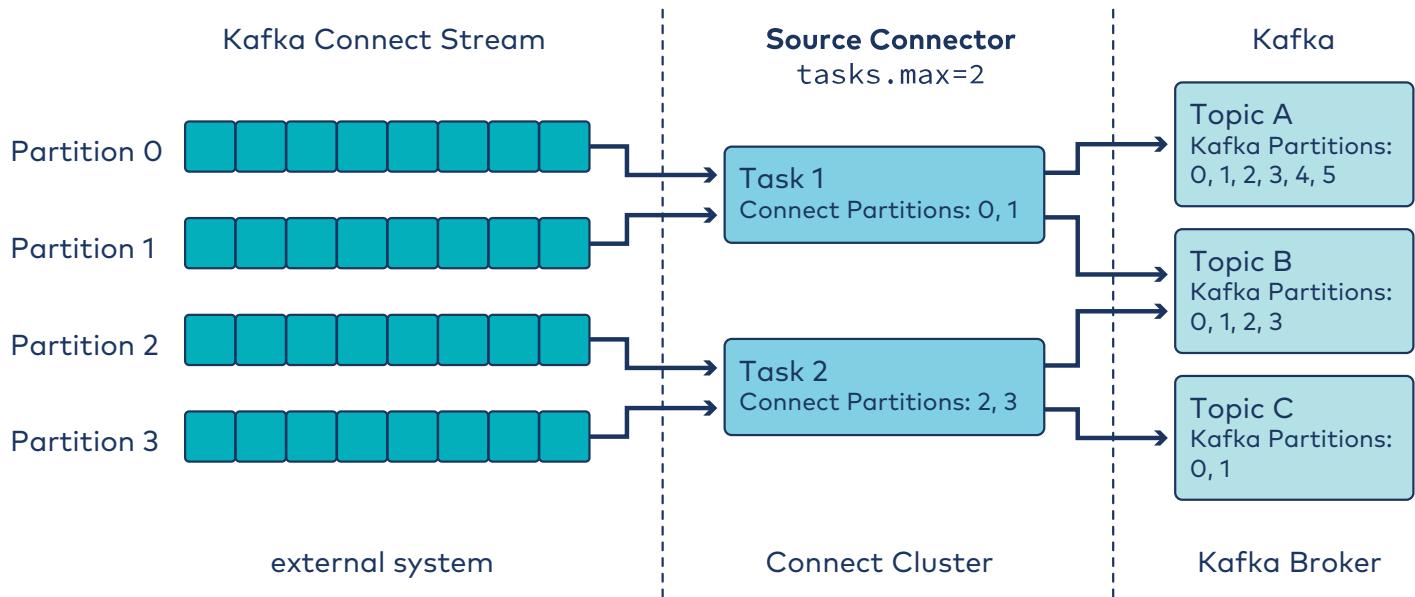
So

- Splitting the workload into smaller pieces provides the parallelism and scalability.

- Connector jobs are broken down into *tasks* that do the actual copying of the data.

- *Workers* are processes running one or more tasks, each in a different thread.

Pictured, we see an external system whose data is imported to Kafka by a source connector. The source connector defines 2 tasks. The tables are assigned to those tasks. The tasks are the threads that actually move the data. In this case, Task 1 produces data from the external system to topics A and B in Kafka. In parallel, Task 2 produces data to topics B and C. Notice that the number of "Connect Partitions" and the number of "Kafka Partitions" are unrelated. Also, notice that the task threads are running in a "connect cluster," not on Kafka brokers.

This image is in terms of a database source connector. We could generalize to "Connect Partitions" from the tables of the database.

We can generalize the above graphic:

Kafka Connect Stream · Source Connector tasks.max=2 · Kafka

Partition 0		Task 1 Connect Partitions: 0, 1		Topic A Kafka Partitions: 0, 1, 2, 3, 4, 5
Partition 1				Topic B Kafka Partitions: 0, 1, 2, 3
Partition 2		Task 2 Connect Partitions: 2, 3		Topic C Kafka Partitions: 0, 1
Partition 3				

external system · Connect Cluster · Kafka Broker

What Do We Need to Configure?

Remember:

- A **connector** applies to a particular external source or sink

- A **connector** may be broken into one or more parallel **tasks**

- A **worker**…

  - … runs zero or more connectors

  - … runs zero or more tasks

  - … is generally part of a group, managed by Kafka's group management protocol

Activity: Brainstorming Connector Configurations

What do you think we need to specify to configure a **connector**? Discuss with a small group for 2 minutes.

Configuring Connectors

Name	Description	Default
`name`	Connector's unique name	
`connector.class`	Name of the Java bytecodes file for the connector	
`tasks.max`	Maximum number of tasks to create - if possible	1
`key.converter`	Converter to (de)serialize keys	(worker setting)
`value.converter`	Converter to (de)serialize values	(worker setting)
`topics`	For **sink connectors only**, comma-separated list of topics to consume from	

Note that `tasks.max` is a limit and is restricted by the shape of the data. If Kafka Connect cannot achieve the desired number of tasks, then it will create as many as possible. For example, if you have a database source connector for a database with 4 tables but set `tasks.max` to 6, you will get 4 tasks, because the copying cannot be parallelized further.

Note that while you can work with connectors in Confluent Cloud, as of August 2022, it is not possible to configure connectors directly through ksqlDB.

It is also possible to define custom topic configurations for the topics that are created by source connectors using the following properties:

Property	Description
`topic.creation.groups`	A list of group aliases that will be used to define per group topic configurations for matching topics. The group `default` always exists and matches all topics.
`topic.creation.$alias.include`	Regular expressions that identify topics to include.
`topic.creation.$alias.exclude`	Regular expressions that identify topics to exclude.
`topic.creation.$alias.replication.factor`	>= 1 for a specific valid value, or -1 to use the broker's default value
`topic.creation.$alias.partitions`	>= 1 for a specific valid value, or -1 to use the broker's default value
`topic.creation.$alias.${kafkaTopicSpecificConfigName}`	List of input topics to consume from

What Do We Need to Configure? (2)

Remember:

- A **connector** applies to a particular external source or sink

- A **connector** may be broken into one or more parallel **tasks**

- A **worker**…

    ◦ … runs zero or more connectors

    ◦ … runs zero or more tasks

    ◦ … is generally part of a group, managed by Kafka's group management protocol

We've established what we need for connectors. So…

Activity: Brainstorming Worker Configurations

What do you think we need to specify to configure a **worker**? Discuss with a small group for 2 minutes.

Configuring a Worker

Name	Description	Default
`bootstrap.servers`	List of host:port pairs to connect	
`group.id`	Identifier of what group this worker is a member of	
`heartbeat.interval.ms`	How frequently heartbeats are sent	3 sec.
`session.timeout.ms`	Time after which a worker that does not heartbeat is deemed dead	10 sec.
`key.converter`	Converter to (de)serialize keys	
`value.converter`	Converter to (de)serialize values	
`topic.creation.enable`	Whether or not source connectors are permitted to create topics	`true`

For more: https://docs.confluent.io/platform/current/installation/configuration/connect/index.html

We see converter configs here and for connectors. Converter configs can be set at the worker level to apply to all connectors running on a worker. They can be overridden at the connector level too.

Another configuration setting to consider is `client.id`; like with producers and consumers, this is a way of naming the client, worker in this case, so it is distinguished in monitoring tools and system logs.

Configuring a Worker: Topics

Name	Stores	Default Num Partitions
`config.storage.topic`	Connector and task configuration	1
`offset.storage.topic`	Source and sink offsets	25
`status.storage.topic`	Current status of connectors and tasks, e.g., running, paused, etc.	5

Kafka Connect uses a few internal topics for configuration settings too. You can configure what those topics are called.

The topics are automatically configured with recommended replication factor and partition count values, and they are compacted. The number of partitions is shown in the table.

If you manually configure these topics, keep the relative partition counts in mind.

Running the Config

- Create a properties file for Connect, e.g., `connect-distributed.properties`

- Run on **each** worker node:

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

- Can configure Connectors via REST API

  ◦ Or, indirectly, via Confluent Control Center

  ◦ You will see this in lab!

- Can configure connectors via ksqlDB as well

In your configuration file, list off properties and their values, e.g.,

```
bootstrap.servers=kafka1:9092, kafka2:9092, kafka3:9092
```

[Connect REST Interface documentation](#)

Standalone Mode

- We've looked at Kafka Connect in **distributed mode**
  → Wanted in production in most cases

- There is a **standalone mode** too

  ◦ Good for development and testing

  ◦ Needed for certain connectors

- Standalone config differences:

  ◦ Offsets are stored in a file rather than in a Kafka topic. Filename is set in
    `offset.storage.file.filename`

Some connectors, e.g., the [Syslog Source Connector](#) require being run in standalone mode.

10c: Deep Dive into a Connector & Finding Connectors

Description

Details of the JDBC Source Connector, configuration details, working through why one would do certain configs with examples. Finding Connectors on Confluent Hub.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe how to configure the JDBC source connector.

- Select an appropriate mode for detecting which rows to copy with the JDBC source connector.

- Determine if the JDBC source connector is suited to an application.

- Tell where to find other Confluent-approved connectors.

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.

The JDBC source connector allows you to import data from any relational database with a JDBC driver into Kafka topics. By using JDBC, this connector can support a wide variety of databases without requiring custom code for each one.

Data is loaded by periodically executing a SQL query and creating an output record for each row in the result set. By default, all tables in a database are copied, each to its own output topic. The database is monitored for new or deleted tables and adapts automatically. When copying data from a table, the connector can load only new or modified rows by specifying which columns should be used to detect new or modified data.

Query Mode (1)

Incremental query mode	Description
Bulk	Load all rows in the table. Does not detect new or updated rows.

The connector can detect new and updated rows in several ways, but let's start simple: for a one-time load, not incremental, unfiltered, we just use `bulk` mode.

Query Mode (2)

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.

The connector can detect new and updated rows in several ways as described on the slide.

For the reasons stated on the slides, many environments will use both the timestamp and the incrementing column to capture all updates.

Because timestamps are not necessarily unique, the timestamp column mode cannot guarantee all updated data will be delivered. If two rows share the same timestamp and are returned by an incremental query, but only one has been processed before the Connect task fails, the second update will be missed when the system recovers.

Query Mode: Custom Query

We can also define a **custom query** to use in conjunction with the previous options for custom filtering.

The custom query option can only be used in conjunction with one of the other incremental modes as long as the necessary WHERE clause can be appended to the query. In some cases, the custom query may handle all filtering itself.

Configuration

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

> ℹ️ See JDBC Connector docs for a complete list

Setting both `table.whitelist` and `table.blacklist` does not fail any upfront configuration validation checks but will fail when starting the connector at runtime.

JDBC Source Connector Config Example

```
 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",
14    }
15  }
```

The goal of this connector is to take the `driver` table of a Postgres database and produce its records to Kafka. We would like each Kafka record to have a string key for the driver ID (`driver-1`, `driver-2`, etc.). We would also like the value of each Kafka record to be an Avro record with id, driverkey, firstname, lastname, make, model, and timestamp.

Unfortunately, the configurations shown will not result in the schema we want. First, the topic name would be `driver` rather than `driver-profiles-avro`. Second, the record keys would be `NULL` and the values would include a field that looks like `{"driverkey": "driver-3"}`. We can modify these minor details using something called SMTs:

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

This lesson is not meant to be your formal introduction to SMTs, but this is provided as an example.

The connector takes the `driver` table of a Postgres database and produces its records to Kafka. We would like each Kafka record to have a string key for the driver ID (`driver-1`, `driver-2`, etc.). The value of each Kafka record will be an Avro record with id, driverkey, firstname, lastname, make, model, and timestamp.

- In line 14, we define three transformations: `suffix`, `createKey`, and `extractKey`. These names can be anything, but it is recommended that they succinctly describe the

transformations.

- In line 15, we define the class that will be used for the `suffix` transformation. In this case, we use the **RegexRouter** class, which is a class that sets the Kafka topic name. Normally, the topic name would be "prefix" + "table." Earlier, we set `topic.prefix` to the empty string. So the topic name should just be the name of the table, which is `driver`. This transformation replaces `driver` with `driver-profiles-avro.`

- In line 18, we define the class used for the `createKey` transformation. In this case, we use the **ValueToKey** class, which is a class that replaces the default Kafka record key with a new key from a field in the table. In this case, we use the `driverkey` field as the key. Without this transformation, the keys would be `null`. With this transformation, an example key would be `{"driverkey": "driver-3"}`.

- In line 20, we further refine the key with the **ExtractField$Key** class. We extract the string associated with `driverkey`. Before this transformation, an example Kafka record key would be `{"driverkey": "driver-3"}`. After this transformation, the Kafka record key is simply the string "driver-3."

Other Connectors

Search Confluent Hub at `confluent.io/hub` for connectors!

You can find many more connectors, along with their documentation, on Confluent Hub.

11: Applying Kafka Connect

CONFLUENT
Global Education

Module Overview

This module contains a hands-on lab for Connect and one lesson:

    a. Full Solutions Involving Other Systems

Where this fits in:
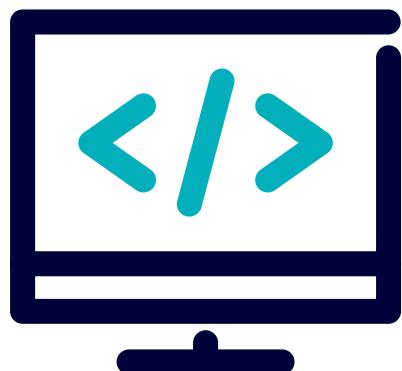
- Hard Prerequisite: Starting with Kafka Connect

Lab: Kafka Connect - Database to Kafka

Please work on **Lab 11a: Kafka Connect - Database to Kafka**

Refer to the Exercise Guide

11a: Full Solutions Involving Other Systems

Description

Case studies of using a Connector to read in data from a source, transform the data, and write it to a sink. SMTs vs. Kafka Streams vs. ksqlDB as options for transforming data.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe how data flows between other systems and Kafka in an ETL use case.

- Describe how data flows between other systems and Kafka in a CDC use case.

- List and evaluate options for the "transform" step of ETL using a Kafka deployment.

Activity: Transforming Temperatures - Part 1

Consider this scenario:

- You have a relational database `db1` with table `temp_readings` with columns for an auto-incremented reading ID, a postal code, a timestamp, and a temperature reading for each row recorded at the given timestamp and in *Celsius*.

- You have a similar relational database `db2` with table `temp_readings`, but temperatures are in *Fahrenheit*.

- You want a Kafka topic with **all** temperature readings in *both* unit systems.

Thinking about everything we've learned so far, how might you go about doing this at a high level? Discuss with a classmate or two.

Activity: Transforming Temperatures - Part 2
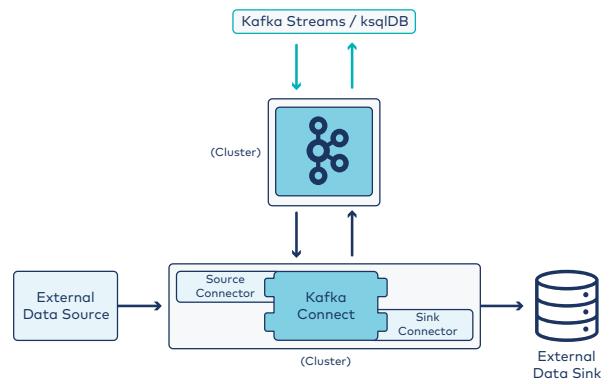
Let's go deeper...

a. Suppose you are using Kafka Connect with the JDBC Source Connector to get this data into Kafka. Locate the "Configuration" slide in the last lesson in your student handbook. What configuration settings are important to set?

b. Thinking back on tools we learned about, what tool would you use to create to generate the desired new topic? (Hint: use something outside of producers and consumers.) Describe at a high level what you would do.

c. Suppose instead of wanting `all` temperature readings, you want this new Kafka topic to contain only the latest temperature reading for each postal code. What would you do differently?

Generalizing & Extending

Here we have

1. Kafka Connect source connectors to read data from external systems, database tables in this case.

2. Kafka topics on the brokers to receive our input.

3. Streaming applications to transform our data.

4. Another Kafka topic on the brokers to receive our transformed data.

We could also send this temperature data to another system that displays temperatures on a dashboard.

This is an example of **ETL**.

A Related Problem
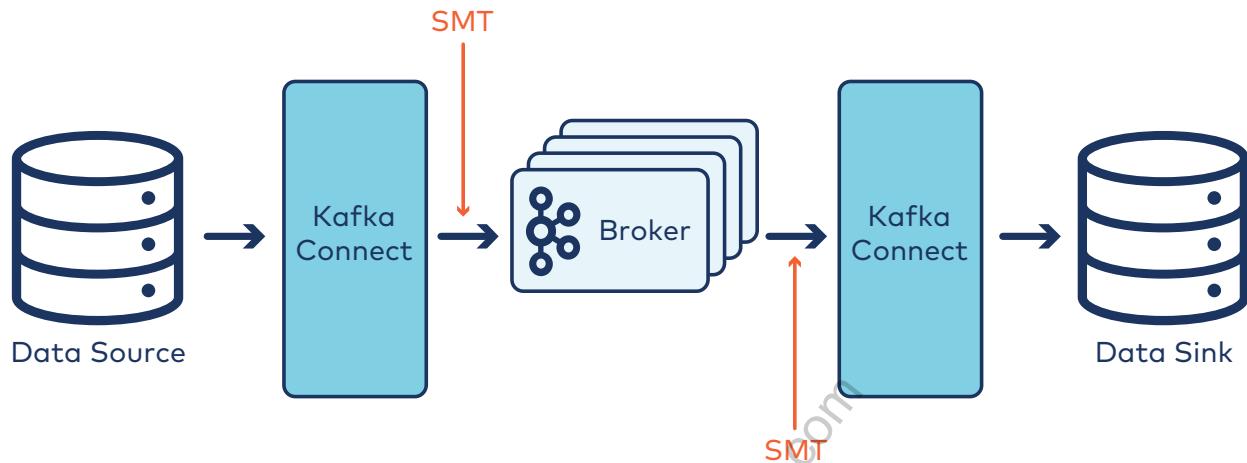
Say we

- Have access to a hospital's database.

- Want to extract information on patients who've been diagnosed with cancer.
  (and compare with those who have not been)

- Want to load this information to CSV files that will be given to medical researchers
  studying correlations between diagnoses and patient traits.
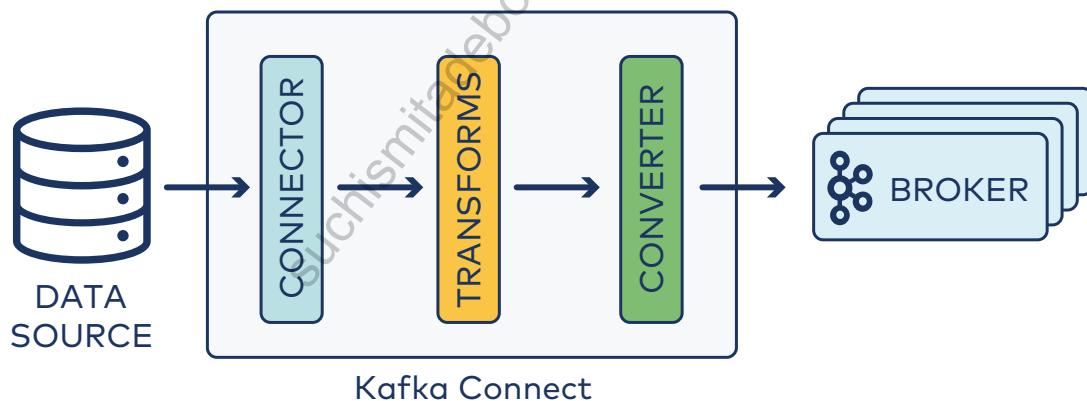
Does this fit what we just did?

Single Message Transforms

Here's a graphic that shows where SMTs live within Connect:

Kafka Connect

Here are several SMTs:

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

Transform	Description
`MaskField`	replace field with valid `null` 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 `Struct` or a `Map`
`ExtractField`	extract a specific field from `Struct` and `Map` 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

For more information on SMTs, see
https://docs.confluent.io/current/connect/transforms/index.html

See your student handbook in the JDBC source connector lesson for an extension of the JDBC source connector example with SMTs added.

One Last Example

- You are maintaining information about car insurance rates for customers.

- Regulations based on where the cars are principally garaged drive a base cost that's part of an insurance rate.

- Information on vehicles and their garage locations is in one external system.

- Insurance rate information is in a different external system.

  ◦ A base rate is one field of a table in such a system.

Questions:

- When a vehicle's location changes, what has to happen?

- What tools would you use to achieve the necessary updates?

This is an example of CDC, or Change Data Capture.

We can use Kafka Connect to read data in and our connector could look for changed records only and read in only changed records to a Kafka topic.

Something could calculate that base cost.

We can use Kafka Connect to write out that base cost to the appropriate place in the other system.

As for the **something** to do the calculation, we probably would need a streaming application. ksqlDB could do the job for us more simply than Kafka Streams, although we might choose to deploy a User Defined Function (UDF). We might think to apply an SMT in Kafka Connect, but there isn't an appropriate SMT to solve this problem in the list.

What if the change we were capturing instead caused us just to write something out where we just needed to convert data to a different format for our destination system? That *could* leverage an SMT, namely `Cast`.

Choose Your Tool

Tool	Good when...	Challenges
Kafka Streams	• Want a lot of control • Want custom logic	• Need to know Java/Scala • Overkill for very simple transformations
ksqlDB	• Logic fits SQL-like syntax • Want to develop quickly	• Not everything fits the syntax • Need to set up ksqlDB server
Kafka Connect SMTs	• Simple transform, e.g., ◦ Remove data for security or performance ◦ Make your data conform to the schema of the output system • Transform exists	• Need to have an SMT • Not for coding business logic • Less control

We know that Kafka Streams, ksqlDB, and Kafka Connect SMTs all allow us to transform data. It's worth thinking about which tool is the best for a task. This slide summarizes some considerations.

More Advanced Kafka Development Matters

Agenda

This is a branch of our developer content on more advanced matters in Kafka development. It is broken down into the following modules:

12. Challenges with Offsets

13. Partitioning Considerations

14. Message Considerations

15. Robust Development

This branch assumes proficiency in the Core Kafka Development branch. The last lesson of the last module assumes having completed the Kafka Connect module of the Additional Components of Kafka/CP Deployment Branch.

Here is an expanded version of the outline, including the lessons that make up each module:

1. Challenges with Offsets

   a. How Does Compaction Affect Consumer Offsets?

   b. What if I Want or Need to Adjust Consumer Offsets Manually?

2. Partitioning Considerations

   a. How Should I Scale Partitions and Consumers?

   b. How Can I Create a Custom Partitioner?

3. Message Considerations

   a. How Do I Guarantee How Messages are Delivered?

   b. How Should I Deal with Kafka's Message Size Limit?

   c. How Do I Send Messages in Transactions?

4. Robust Development

   a. Testing Matters

   b. Exceptions in Kafka

c. Kafka Connect Error Handling

This branch assumes proficiency in the Core Kafka Development branch. The last lesson of the last module assumes having completed the Kafka Connect module of the Additional Components of Kafka/CP Deployment Branch.

12: Challenges with Offsets

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

   a. How Does Compaction Affect Consumer Offsets?

   b. What if You Want or Need to Adjust Consumer Offsets Manually?

Where this fits in:

- Hard Prerequisite: Groups, Consumers, and Partitions in Practice
- Recommended Follow-Up: Other modules in this branch, other courses

12a: How Does Compaction Affect Consumer Offsets?

Description

How consumers deal with missing offsets, what happens when offsets don't make sense, and getting into the details of how compaction works: how it affects offsets and is triggered. Deleting keys.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe how a consumer will read a log when that log has been compacted.

- Describe how to deal with cases when a consumer's offset does not make sense.

- Determine whether or not compaction will run, given information about a log, and know how to tune triggering compaction.

- Describe how to delete all instances of a key from a log.

Getting Started: Log Refresher

Recall: messages are written to logs...

- Per partition

- Messages have offsets

- Divided into segments

  - Active, inactive

  - Clean, dirty

- Log retention policies:

  - `delete`

  - `compact`

(refer back to lesson 1b)

See Lesson 1b for our treatment of retention policies. We go deeper here.

Activity 1: Compacting a Compacted Log

Suppose we have a log like this for a partition:

offset	3	6	8	9	17	26	27	28	29	30	31	32	33	34	35
key	2	1	12	11	10	7	9	1	7	15	7	15	7	12	6

inactive clean segment 1	inactive clean segment 2	inactive dirty segment 3	active segment
size: 10	**size: 10**	**size: 15**	**size: 8**

From this picture, we can see that compaction must have happened. So, let's say a consumer offset is 7. As you can see, the message at consumer offset 7 has been compacted away. Then...

a. What message do you expect the consumer will read next? (This is more gut intuition than anything we've covered in this course.) Why would that make sense to you?

b. Suppose it's time for compaction to happen. What is the resulting log?

Debrief

What have we concluded?

A consumer always advances to the next available offset.

Compaction removes those messages whose key appears later in the log in an inactive segment.

A Taste of Documentation: What Triggers Compaction

Here's a snippet copied from the documentation on a compaction property for you to use in the next activity:

`log.cleaner.min.cleanable.ratio`

- The minimum ratio of dirty log to total log for a log to eligible for cleaning. If the `log.cleaner.max.compaction.lag.ms` or the `log.cleaner.min.compaction.lag.ms` configurations are also specified, then the log compactor considers the log eligible for compaction as soon as either: (i) the dirty ratio threshold has been met and the log has had dirty (uncompacted) records for at least the `log.cleaner.min.compaction.lag.ms` duration, or (ii) if the log has had dirty (uncompacted) records for at most the `log.cleaner.max.compaction.lag.ms` period.

- Default: 0.5

This information is here for your reference for the activity on the next slide.

Documentation link

Note also that

- this is a broker propety, but...

- a topic can override a broker's configured ratio with the property `min.cleanable.dirty.ratio`

Activity 2: Interpreting What Triggers Compaction

Here's that log again:

offset	3	6	8	9	17	26	27	28	29	30	31	32	33	34	35
key	2	1	12	11	10	7	9	1	7	15	7	15	7	12	6

inactive clean segment 1 **size: 10**	inactive clean segment 2 **size: 10**	inactive dirty segment 3 **size: 15**	active segment **size: 8**

Suppose default compaction settings are on. Does log compaction actually happen? If yes, why? If no, what would have to be different to trigger compaction?

Debrief and Gotchas

So...

1. What have we learned about how compaction operates?

2. If you have a development use case where you rely on having exactly one value per key, can you rely on compaction?

We did **not** trigger compaction in the scenario on the last slide due to the ratio of `dirty log size` over `total log size` being `15` over `10 + 10 + 15 = 35`, which is less than 0.5.

Remember:

- Compaction only consider **inactive** segments.

- Even considering all clean segments, there could still be later instances of keys appearing in the active segment or in segments that were active at the time of last compaction and have since become inactive. You must consider this if your development use case depends on it.

One More Problem: Deleting a Key Entirely

Compaction yields a log where there is at most one instance of any key in the inactive segments.

But what if you don't want a key at all anymore?

- Tombstone Messages:
  - delete key K by sending `{K:null}`
  - Consumer has `log.cleaner.delete.retention.ms` time to consume K before it is deleted (default 1 day)

One use case of this: Say a key represents an `order_id` for a retail order. Say values are status, e.g. "order received," "packed," "shipped," "received." After a certain point, storing old orders on the Kafka server is wasteful, especially when their return window has closed. If a business needed to look them up, one could look to archival storage. But compaction would always keep the latest value per key. If we want to get rid of that message after a certain time period, we can achieve that using a tombstone message.

As keys are retired, many systems will send delete messages. The simplest approach would be to retain delete messages forever. But since the purpose of deletes is typically to free up space, this approach would have the problem that the Commit Logs would end up growing forever if the keyspace keeps expanding and the delete markers consume some space. Tombstones (the Kafka implementation of a delete message) are keyed messages with a null value.

However, a delete message should not be removed too quickly or it can result in inconsistency for any Consumer of the data reading the tail of the log. Consider the case where there is a message with key K and a subsequent delete for key K. If log compaction removes delete messages, there is a race condition between a Consumer of the log and the log compaction action. Once the Consumer has seen the original message, we need to ensure it also sees the delete message; this might not happen if the delete message happens too quickly. As a result, the topic can be configured with a configurable SLI for delete retention (`delete.retention.ms`). This SLI is in terms of time from the last cleaning. A consumer that starts from the beginning of the log must consume the tail of the log in this time period to ensure that it sees all delete messages.

If an application needs to be able to send null values that will not be mistaken as tombstones, you need to introduce a null-type – like a `NullInteger` - so it looks like a regular message with non-null value to Kafka.

12b: What if You Want or Need to Adjust Consumer Offsets Manually?

Description

Reprocessing of messages, finding consumer offsets both now and in the past, programmatically changing offsets, and automatic vs. manual committing strategies.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Describe what could happen when a consumer newly assigned an offset for a partition is assigned an out-of-date offset

- Describe how to find a consumer's offset for a partition now or at a given time

- Describe how, programmatically, to change a consumer's offset for a partition

- List some reasons one would want to change from automatic offset management

How Does a Consumer Know Its Offset?

	Consumer is consuming from a partition it was already consuming	Consumer gets assigned a partition that another consumer in group was consuming	Consumer is starting up and is first in group to consume from a partition
Local	Consumer uses offset for this partition from memory	Consumer doesn't have any offset in memory for this partition	Consumer doesn't have any offset in memory for this partition
Kafka	Not needed	Offset comes `__consumer_offsets` topic for this group/partition pair	There won't be an entry in `__consumer_offsets`. Offset determined by `auto.offset.reset` - one of `earliest`, `latest`, `none`

The first two columns are review. See the Consumer Offsets lesson (5c) for the first introduction of the `__consumer_offsets` topic.

The Consumer property `auto.offset.reset` determines what to do if there is no valid offset in Kafka for the Consumer's Consumer Group.

We consider here the case when a particular Consumer Group starts the first time.

- The value of `auto.offset.reset` can be one of:

  ◦ `earliest`: Automatically reset the offset to the earliest available

  ◦ `latest`: Automatically reset the offset to one more than latest offset that has data (in other words, have the consumer ready to read the *next* message that comes in)

  ◦ `none`: Throw an exception if no previous offset can be found for the ConsumerGroup

- The default is `latest`

The next slide will go into other cases where `auto.offset.reset` applies.

What if a Consumer's Offset Makes No Sense?

Here again, `auto.offset.reset` determines how the consumer proceeds.

2 New Scenarios	Values for `auto.offset.reset`	
Consumer offset < smallest offset	`earliest`	Reset offset to earliest available
Consumer offset > last offset + 1	`latest`	Reset offset to latest available
	`none`	Throw exception

- The consumer property `auto.offset.reset` determines what to do if there is no valid offset in Kafka for the Consumer's Consumer Group

  ◦ When a particular consumer group starts the first time

  ◦ If the consumer offset is less than the smallest offset

  ◦ If the consumer offset is greater than the last offset

- The value can be one of:

  ◦ `earliest`: Automatically reset the offset to the earliest available

  ◦ `latest`: Automatically reset the offset to one more than latest offset that has data (in other words, have the consumer ready to read the *next* message that comes in)

  ◦ `none`: Throw an exception if no previous offset can be found for the consumer group

- The default is `latest`
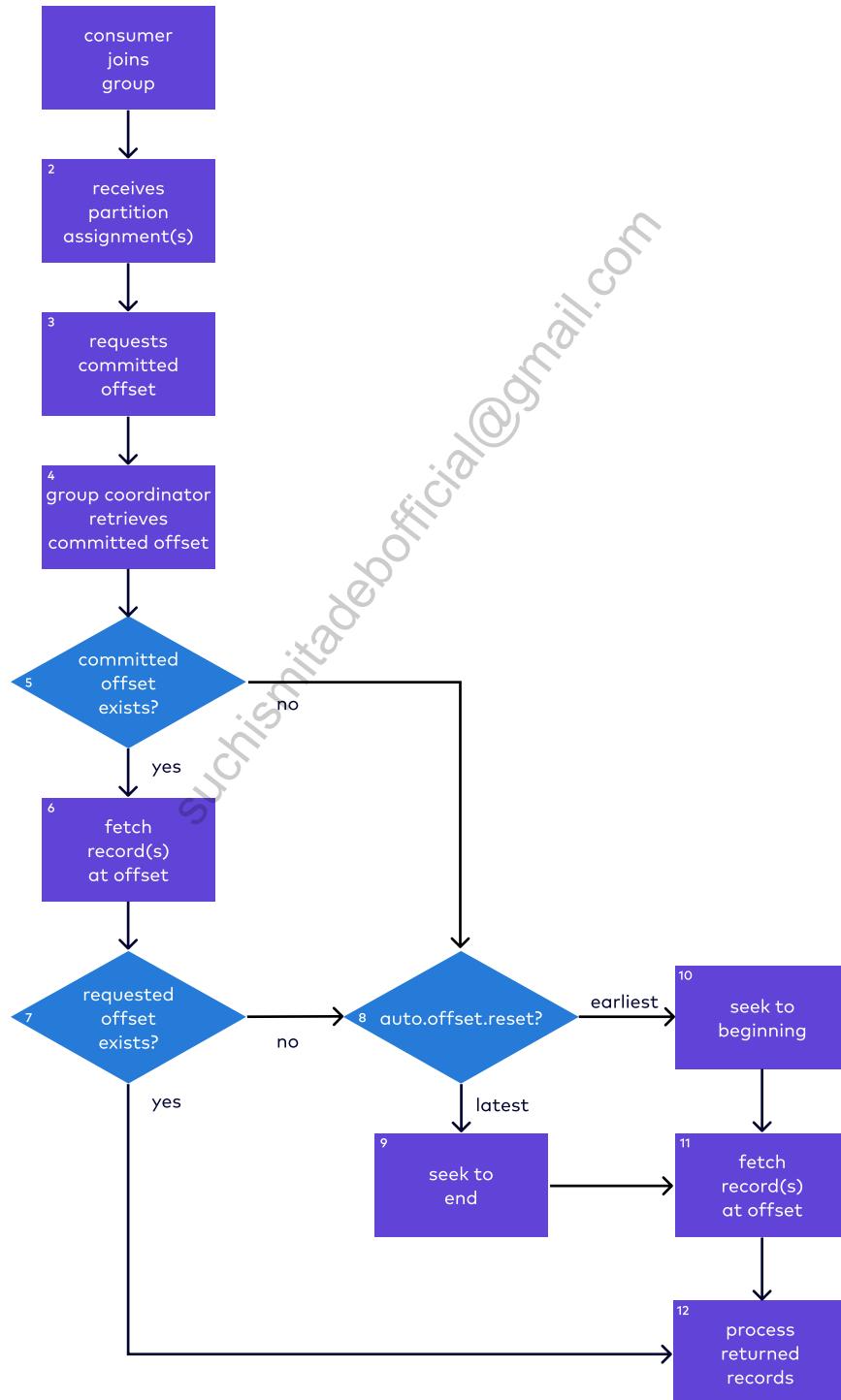
🛈	This setting does not affect offsets that have been deleted or compacted due to log cleanup. In those cases, if the offset specified in the offsets topic does not exist, the broker will advance to the next highest offset and proceed from there.

To get into more detail, note that:

- There is a process called the **group coordinator** running in Kafka for each group. It is involved whenever a consumer joins the group (and it also receives/detects heartbeats).

- When a consumer joins a group, it will receive partition assignments and request committed offsets from Kafka.

This flowchart shows the sequence of events that happen from when a consumer joins a consumer group:

Explaining the details:

1. Consumer joins group:

   ◦ When it starts

   ◦ When informed that a rebalance is in progress

   ◦ When metadata refresh for subscribed topics indicates new matching topic exists, existing topic number of partitions has changed, or existing topic is deleted

2. Group coordinator sends partition assignments to group members

3. For each assigned partition, consumer sends a request for the committed offset to the group coordinator.

4. The group coordinator retrieves the requested committed offset from the `__consumer_offsets` topic.

5. If one exists, i.e., it has been previously consumed by a group member, the committed offset is returned to the consumer. If no offset exists, a corresponding response is returned to the consumer.

6. If the consumer receives the committed offset from the group coordinator, it fetches records from the partition starting at that offset.

7. Depending upon the topic retention and how long it has been since the offset was last committed, the fetched offset may or may not exist.

8. If the offsets topic did not contain a committed offset for the partition or if it did but what an attempt to fetch that offset resulted in an exception indicating the offset does not exist, e.g. its retention may have expired due to the amount of time that has elapsed since the offset was committed to `__consumer_offsets`, the consumer will reset the offset based upon the `auto.offset.reset` property.

9. If `auto.offset.reset` equals `latest`, the consumer will reset to the latest offset.

10. If `auto.offset.reset` equals `earliest`, the consumer will reset to the earliest offset.

11. The consumer will then fetch records using this earliest or latest offset.

12. The consumer will then process records received in the fetch response.

How Do I Find a Consumer's Current Offset?

Function	Return Value
`position(TopicPartition)`	Offset of next record that will be fetched
`offsetsForTimes(Map<TopicPartition, Long> timestampsToSearch)`	Offsets for given partitions by timestamp

- The `KafkaConsumer` API provides ways to view offsets from which the Consumer will read

  - `position(TopicPartition)` provides the offset of the next record that will be fetched

  - `offsetsForTimes(Map<TopicPartition,Long> timestampsToSearch)` looks up the offsets for the given Partitions by timestamp

    - The returned offset for each partition is the earliest offset whose timestamp is greater than or equal to the given timestamp in the corresponding partition. This is useful to rewind offsets if applications need to re-consume messages for a certain period of time, or in a multi-datacenter environment because the offset between two different Kafka clusters are independent and users cannot use the offsets from the failed datacenter to consume from the DR datacenter. In this case, searching by timestamp will help because the messages should have same timestamp if users are using `CreateTime`. Even if users are using `LogAppendTime`, a more granular search based on timestamp can still help reduce the amount of messages to be re-consumed.

Messages written to a topic using replication tools (e.g., MirrorMaker, Confluent Replicator) will retain their original timestamps when copied to the new location.

What if I Need to Reprocess Records?

- 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

How to Reset the Current Offset (1)

Method	Effect
`seek(TopicPartition, offset)`	Seek to specific offset in specified partition
`seekToBeginning(Collection<TopicPartition>)`	Seek to first offset of each specified partition
`seekToEnd(Collection<TopicPartition>)`	Seek to one beyond the last offset of each specified partition

- The `KafkaConsumer` API provides ways to dynamically change the offset from which the consumer will read

  - `seek(TopicPartition, offset)` seeks to a specific offset in the specified Partition

  - `seekToBeginning(Collection<TopicPartition>)` seeks to the first offset of each of the specified partitions

  - `seekToEnd(Collection<TopicPartition>)` seeks the offset to one more than latest offset that has data (in other words, have the consumer ready to read the *next* message that comes in) in each of the specified partitions

Outside of code, the `kafka-consumer-groups` command has the ability to reset the entire consumer group to the offsets corresponding to specific timestamps within the partitions:

```
kafka-consumer-groups \
    --bootstrap-server broker101:9092 \
    --group my-group \
    --topic my_topic \
    --reset-offsets \
    --execute \
    --to-datetime 2017-08-01T17:14:23.933`
```

You can also change the offset for (a) particular partition(s). This alteration would change the offset for partition 2 only:

```
    --topic my_topic:2
```

This alteration would change the offset for three partitions only:

```
    --topic my_topic:2,4,7
```

How to Reset the Current Offset (2)

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

```
 1  for (TopicPartition partition : partitions)
 2  {
 3      timestampsToSearch.put(partition, MY_TIMESTAMP);
 4  }
 5
 6  Map<TopicPartition, OffsetAndTimestamp> result = consumer.offsetsForTimes
    (timestampsToSearch);
 7
 8  for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : result.entrySet())
 9  {
10      consumer.seek(entry.getKey(), entry.getValue().offset());
11  }
```

- Add each partition and timestamp to HashMap

- Get offset for each partition

- Seek to specified offset for each Partition

How to Reset the Current Offset (3)

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

```java
1   ConsumerRebalanceListener listener = new
2       ConsumerRebalanceListener()
3       {
4           @Override
5           public void onPartitionsRevoked(Collection<TopicPartition> partitions)
6           {
7               // nothing to do...
8           }
9
10          @Override
11          public void onPartitionsAssigned(Collection<TopicPartition> partitions)
12          {
13              consumer.seekToBeginning(partitions);
14          }
15      };
16  consumer.subscribe(Arrays.asList("my_topic"), listener);
17  consumer.poll(Duration.ofMillis(0));
```

Summary of code:
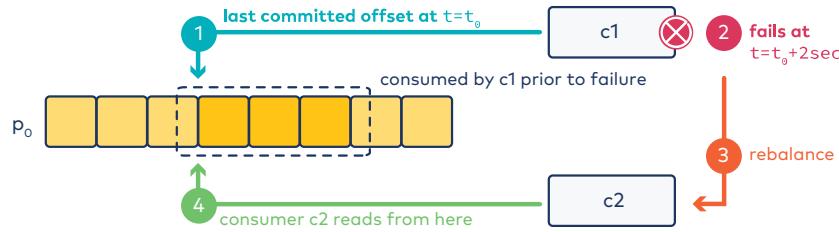
- Subscribe to topics & register `ConsumerRebalanceListener`

- `poll(Duration.ofMillis(0))` retrieves metadata from broker

  - **Example:** to seek to the beginning of all partitions that are being read by a consumer for a particular topic, you might do something like:

The `poll` method has several functions:

- Fetch messages from assigned partitions

- Trigger partition assignment (if necessary)

- Commit offsets if auto offset commit is enabled

In this example, the data returned by `poll` (i.e., messages fetched from the partitions) is not important since we are resetting the place that it is reading from so the data is not assigned to an object. Calling `poll` immediately after the `subscribe` triggers the partition assignment.

What if Messages Get Reprocessed After a Consumer Fails?

In a rebalance scenario, any records that were processed **but not committed** by the previous consumer of the partition would be reprocessed by the new consumer. In many use cases, this is completely fine; consumers process one record at a time without side effects, so reprocessing a couple of records during a rebalance is acceptable.

However, there are certain use cases where this is not ideal. For example, if the consumer is sending emails, then reprocessing means resending those emails. Customers could get annoyed by the duplicate emails.

One Solution: Handle Data and Offsets Atomically

We could employ the help of an external transactional database for storing offsets.

pseudocode

```
1  SELECT partition, offset FROM database_offset_table
2  consumer.seek(partition, offset)
3
4  while true:
5    input_data = consumer.poll()
6    for purchase_input in input_data:
7      purchase = purchase_input.process_purchase()
8      BEGIN TRANSACTION
9      INSERT purchase.id, purchase.amount INTO purchase_table
10     INSERT partition, consumer.position(partition) INTO database_offset_table
11       ON DUPLICATE KEY UPDATE
12     COMMIT TRANSACTION
```

This pseudocode describes the general idea of handling a Kafka message and its offset together atomically. In this case, we use databases that supports transactions. We also show another application of the consume-process-produce model.

We first obtain information about the partition and offset from one database, seek to that offset on the partition in Kafka, and then enter the poll loop. Then:

1. The application acts as a consumer and polls for records (Line 5, the "consume")

2. For each record consumed:

   a. The application processes the records (Line 7, the "process")

   b. The application enters that data **along with the partition and offset** into databases as part of an atomic transaction (Lines 8-12, the "produce")

Since the Kafka message and its offset are handled together atomically, a consumer rebalance won't lead to reprocessing. The tradeoff is that the throughput is limited to the external system's ability to process these transactions.

The offsets would almost certainly be stored in a different table that the business-relevant data, so we intentionally use two different database tables (seen on Lines 9 and 10).
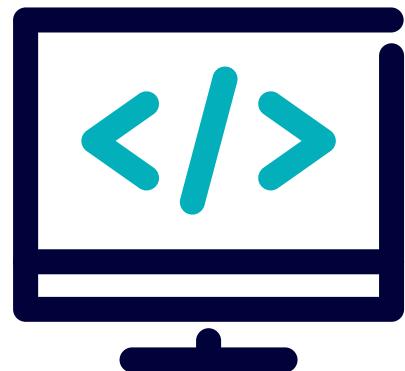
Additionally, note that the consumer will know its offset, but the polling/processing loop will otherwise not, hence the explicit use of `consumer.position()` on Line 10. The insert is more likely an "upsert," as shown on Line 11.

We could also employ a Kafka Connect sink connector that is idempotent to help with this.

Lab: Kafka Consumer - offsetsForTimes

Please work on **Lab 12a: Kafka Consumer - offsetsForTimes**

Refer to the Exercise Guide

13: Partitioning Considerations

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

a. How Should You Scale Partitions and Consumers?

b. How Can You Create a Custom Partitioner?

Where this fits in:

- Hard Prerequisite: Groups, Consumers, and Partitions in Practice

- Recommended Follow-Up: Other modules in this branch, other courses

13a: How Should You Scale Partitions and Consumers?

Description

Discussions about choosing a number of partitions, a number of consumers, and the effects of changing the number of partitions.

Learning Objectives
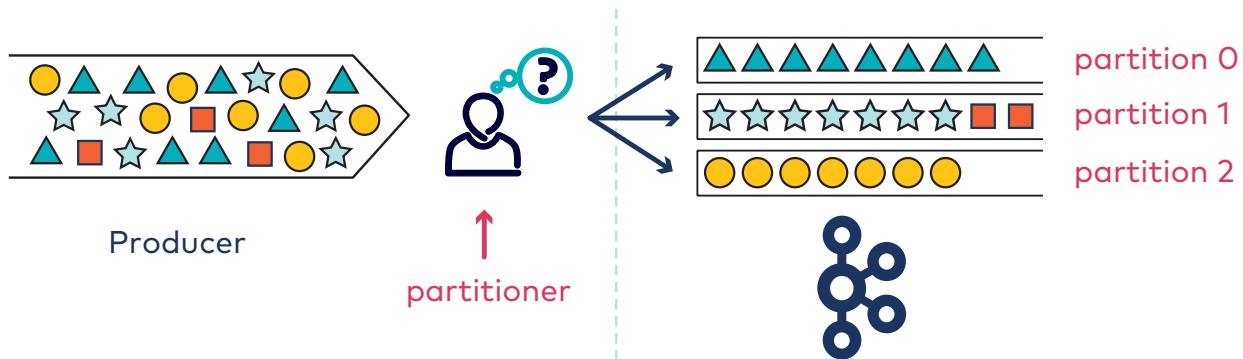
Upon completion of this lesson and associated lab exercises, you will be able to:

- List some considerations in choosing the number of partitions for a topic.

- Evaluate some considerations in choosing the number of consumers in a group, given partitioning information.

- Evaluate and work around implications of changing the number of partitions for a topic.

Discussion: Deciding Number of Partitions

Producer

partitioner

partition 0

partition 1

partition 2

- How many partitions should you create for your topic?

A 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

This slide is a guideline for establishing the number of partitions for a topic if the design requires a specific target throughput, but it is by no means a complete answer. An important takeaway is that performance testing is needed to make a good plan for the number of partitions.

The limiting factor is likely to be the consumers, so the number of partitions will likely be `t/c`. Topics should be sized so that Consumers can keep up with the throughput from a physical (NIC speed) and computational (processing time per poll) standpoint.

For what it's worth, Solutions Architects at Confluent have anecdotally observed that 30 partitions is enough to scale most applications. The number 30 is divisible by 2, 3, 5, 6, 10, 15, and 30, so there are lots of options for scaling the number of consumers as well (next discussion!). Overhead for partitions is negligible until there are more than 4000 per broker or 200,000 per cluster, so "over-partitioning" when a topic is created is the recommended course of action.

- You might also 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`) a|* Vary Consumer properties:

  - Fetch size (`fetch.min.bytes`)

  - Fetch wait time (`fetch.max.wait.ms`)

- You might also 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 (range), what would happen if a consumer group of 10 consumers subscribed to 10 topics, each topic with 1 partition?

Refer back to the "Groups, Consumers, and Partitions" module in your handbook (specifically lesson 5a) if you need to review what range means.

Discussion: Deciding the Number of Consumers, notes

A consumer group's scalability is limited by the number of partitions, and with the default partition assignment strategy (RangeAssignor), subscribing to multiple topics has other considerations as well.

Consider 1 topic for a moment. Because of the way partitions are assigned, it is ideal for the topic to have a number of partitions that is **highly divisible**, so that the consumer group can scale effectively to many consumers while maintaining a balanced load. With 30 partitions, for example, consumer groups can scale to 1, 2, 3, 5, 6, 10, 15, or 30 consumers with balanced load.

If a consumer group is subscribed to multiple topics with the default partition assignment strategy, then these topics may have different numbers of partitions, so it becomes even more important that all topics have partition counts that are highly divisible. If one topic has 30 partitions, and another has 24, then the consumer group could consume from both topics with balanced load for 1, 2, 3, or 6 consumers.

If a consumer group of 10 consumers is subscribed to 10 topics, each with 1 partition, then all 10 partitions will be assigned to the first consumer and there will be 9 idle consumers. The RangeAssignor assigns partitions topic-by-topic, each in exactly the same way.

Discussion: What If I Need More Partitions?

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

What happens if a topic is created with too few partitions to accommodate client applications as they scale?

The next few slides offer the most recommended solutions and workarounds. Other solutions and workarounds are:

Custom partitioner: Build a custom partitioner that keeps keys assigned to the same partitions they're currently on. The downside to this solution is the newly added partitions will only be used when new keys are produced. Another downside is now you must maintain a custom partitioner.
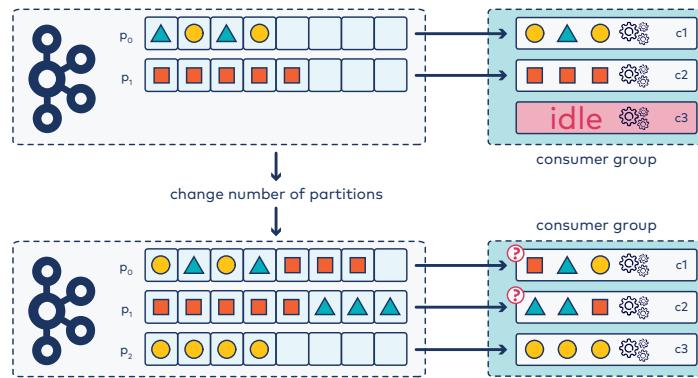
ℹ	We will discuss custom partitioners more in an upcoming lesson.

External, shared state: For the time before and after the partition count changes, change consumers to store state in an external, shared system, rather than in local memory or disk. For example, consider using HBase, Cassandra, etc. Although keep in mind that introducing a write or lookup to these systems will introduce blocking IO in the consumer, and lower throughput.

"Close" the upstream system before migrating: If the upstream system can be "closed" such that all in-flight "transactions" in the Kafka log are finished, "close" the upstream system, let consumers finish consuming all messages, then increase the partition count, restart consumers, and "reopen" the upstream system.

Increasing Partitions: Accept Your Fate

Accept the change in the guarantee: Some applications can just accept that keys will change partitions and hence will happily continue consuming. This can be done with the command:

```
$ kafka-topics \
    --bootstrap-server kafka-1:9092 \
    --alter \
    --topic grow-topic \
    --partitions 12
```

On this slide, icons with same shapes represent records with same key. The topic originally had 2 partitions and 3 consumers, which left one consumer idle. When we increase the number of partitions to 3, the consumer group now utilizes all 3 consumers. However, keys are now being delivered to different partitions than they were before. Squares are now delivered to p_0, stars stay on p_1, and circles now go to p_2.

Recall that semantic partitioning works on the idea that a message will be sent to the partition determined by the formula `hash(key) % n`, where `n` is the number of partitions. Increasing the `n` number could change the output of the formula. Once the number of partitions is increased, a given key may move to a new partition.

Increasing Partitions: Migrate to a New Topic

ksqlDB makes it relatively simple to populate the new topic:

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

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

Question: Is it easier to change the number of consumers or number of partitions?
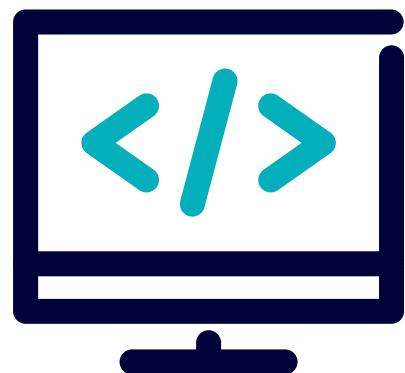
Create a new topic: Create a copy of the original topic, where the new topic has the desired, larger number of partitions. Then, switch the consumers to the new topic, ensuring that the consumers start at the right offset in the new topic. Offsets will be different in the new topic, so a time-based approached will be required. The easiest way to do this would be with ksqlDB.

For more on ksqlDB, consider our 3-day Stream Processing with Kafka Streams and Confluent ksqlDB course.

Lab: Increasing Topic Partition Count

Please work on **Lab 13a: Increasing Topic Partition Count**

Refer to the Exercise Guide

13b: How Can You Create a Custom Partitioner?

Description

The motivation for and how to write a custom partitioner.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:
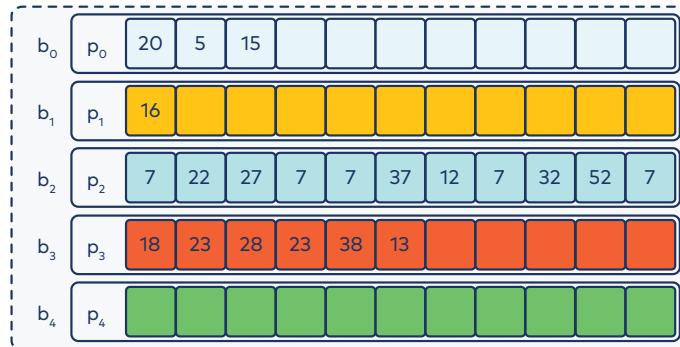
- Create a custom partitioner.

- Configure a producer to use a custom partitioner.

- Compare and choose appropriate partitioning strategies.

Partitioning with Keys

Default: `hash(key) % numPartitions`

But what if there is skew?

b_0	p_0	20	5	15									
b_1	p_1	16											
b_2	p_2	7	22	27	7	7	37	12	7	32	52	7	
b_3	p_3	18	23	28	23	38	13						
b_4	p_4												

In the picture, only the keys of messages are shown. Here, we have one partition ignored completely, while another has significantly more messages. In this example, we might chose to handle the messages with key 7 differently and put them all on the last partition.

Custom Producer Partitioner

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

```
 1  public interface Partitioner extends Configurable, ...
 2  {
 3      void configure(java.util.Map<java.lang.String,?> configs);
 4      void close();
 5      void onNewBatch(String topic, Cluster cluster, int prePartition);
 6
 7      int partition(java.lang.String topic,
 8                    java.lang.Object key,
 9                    byte[] keyBytes,
10                    java.lang.Object value,
11                    byte[] valueBytes,
12                    Cluster cluster);
13  }
```

- Set producer property `partitioner.class`

Sometimes it makes sense to implement a custom partitioner. For example, a stock trading company may trade many stocks and set the key for trade events as a share ID. If the company trades 1000 stocks, but only 5 stocks make up 90% of their trades, then it makes sense to write a partitioner that assigns each of those "hot keys" its own dedicated partition.

- To create a custom Partitioner, you should implement the `Partitioner` interface:
  - This interface includes `configure()`, `close()`, `onNewBatch()`, and `partition()` methods, although often you will only implement `partition()`

- `partition()` is given the Topic, key, serialized key, value, serialized value, and cluster metadata
  - It should return the number of the Partition this particular message should be sent to (0-based)

- `onNewBatch()` was added in AK 2.4 to allow for "batch aware" partitioning of messages with null keys. This allows a new partition to be chosen when a batch fills and a new batch is opened. For more information, see KIP-480

You then need to register the custom partitioner with the `partitioner.class` producer config property. Also, `configure` and `close` are optional lifecycle methods called once by the producer client library when the producer starts and is closed, and `partition` is called for every message. So whatever the `partition` method does, it should ideally be fast, otherwise it can slow down the entire producer.

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:

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

Custom Partitioner: Example (2)

- And the remaining code contains the decision logic:

```
13              // This key will always go to Partition 0
14              if (((String) key).equals("OurBigKey"))
15                  return 0;
16
17              // Other records will go to remaining partitions using a hashing function
18              return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;
19          }
20  }
```

This sample partitioner verifies that messages have a non-null String-type key. Then it returns 0 for messages with the specified key and a non-0 partition number for any other key, based on the standard hashing function.

This custom partitioner `MyPartitioner` would then be passed into the producer Java code using the `partitioner.class` property and is used by the `KafkaProducer` as part of the `send()` call. The developer code does not call the partitioner directly.

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: When might this be useful?

• It is also possible to specify the partition to which a message should be written when creating the `ProducerRecord`

• **Answer**:

  ◦ Writing the partitioning logic directly into the producer code is simpler than creating a new class so it may be useful in very simple cases…

  ◦ It is convenient to use for testing.

  ◦ But… see the activity!

Activity: Analyzing a Partitioning Strategy

Say a producer for `driver_profiles` includes code like this:

```
1  hashedKey = hash(driverID);
2  zipPopulation = getCityPopulationByZIPCode(driverZip);
3
4  if zipPop < MID_CITY_THRESHOLD              //small towns: use 0-4
5      part = hashedKey%5;
6  else if zipPop < LARGE_CITY_THRESHOLD       //medium cities: use 5-14
7      part = hashedKey%10 + 10;
8  else                                        //big cities: use 15-29
9      part = hashedKey%15 + 15;
10
11 newRec = new ProducerRecord<>("driver_profiles", part, key, val);
```

Then:

a. Evaluate this way of partitioning.

b. Say we ultimately wanted to join `driver_profiles` with `driver_positions` downstream. Re-evaluate.

14: Message Considerations

CONFLUENT
Global Education

Module Overview

This module contains three lessons:

a. How Do You Guarantee How Messages are Delivered?

b. How Should You Deal with Kafka's Message Size Limit?

c. How Do You Send Messages in Transactions?

Where this fits in:

- Hard Prerequisite: Preparing Producers for Practical Uses

- Recommended Prerequisite: Groups, Consumers, and Partitions in Practice

- Recommended Follow-Up: Other modules in this branch, other courses

14a: How Do You Guarantee How Messages are Delivered?

Description

How to deal with guaranteeing ordered delivery of messages and non-duplicated writes with idempotence, along with how it works.

Learning Objectives
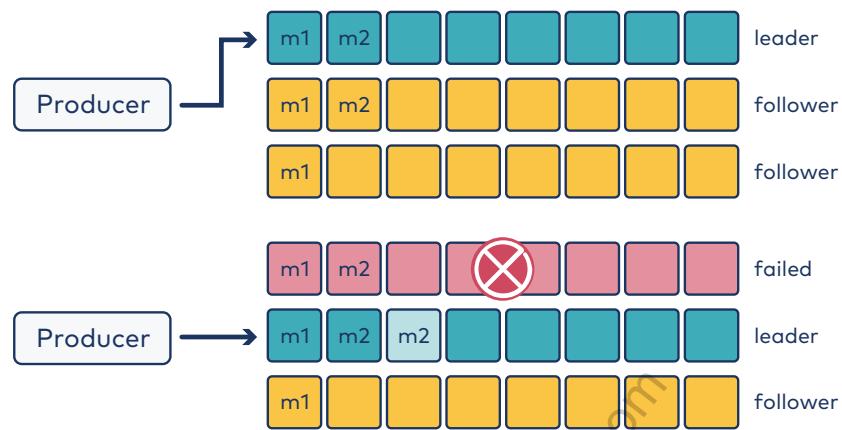
Upon completion of this lesson and associated lab exercises, you will be able to:

- Solve the problem of out-of-order message delivery with idempotence

- Guarantee non duplicate delivery of messages

- Explain how Kafka implements idempotence

Problem: Producing Duplicates to the Log
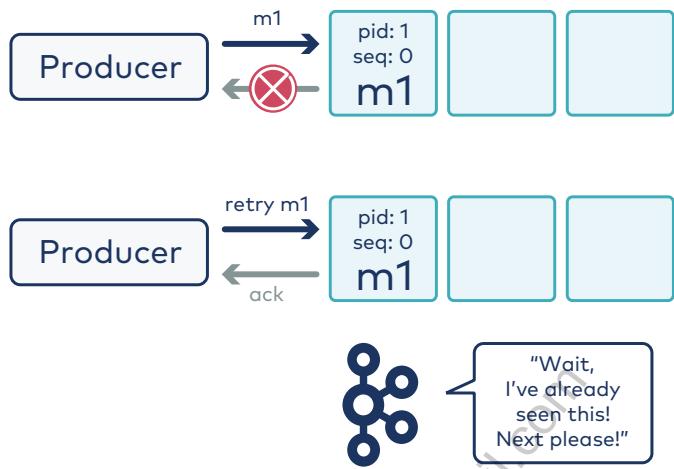
- `acks = all`

- `retries > 0`

Here, the producer has settings `acks=all` with `retries` enabled. The leader fails before record batch m2 is committed to all the brokers. The new leader already received m2, but the retry means that m2 is duplicated. Now all the followers will have a duplicate for m2.

Solution: Idempotent Producers

- `enable.idempotence = true`

- `acks = all`

The `enable.idempotence = true` setting in the producer ensures messages aren't duplicated, even in the case of producer retries or broker failure. This is possible due to headers in the message format for producer ID and sequence number.

- **Producer ID:** A unique identifier for a producer session

- **Sequence number:** Each message a producer sends is given a sequence number that increments with each message.

In this example, we see that the broker recognizes the sequence number from this producer, so it acknowledges the producer without appending a duplicate of m1 to the log.

The broker will retain a map `{ PID : sequence number }` in memory that is occasionally snapshotted to the log in a `.snapshot` file. If the broker recovers from failure, it could read through the log and catch up to the current mapping of PID → Sequence number, but this could take a while. The `.snapshot` file speeds up this process.
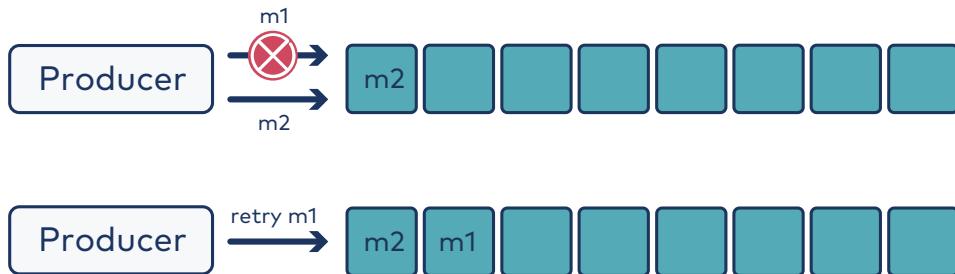
> ⓘ Enabling idempotent producers usually has negligible performance impact, thus making it useful in many situations. The caveats to enabling idempotence are that `max.in.flight.requests.per.connection` must be less than or equal to 5, `retries` must be greater than 0, and `acks` must be "all." If these values are not explicitly set by the user, suitable values will be chosen. If incompatible values are set, a `ConfigException` will be thrown.

Kafka Streams also supports Exactly Once Semantics. In a Kafka Streams app, we set `processing.guarantee=exactly_once` to get exactly once processing. (Default: `at_least_once`.)

Problem: Producing Messages Out of Order
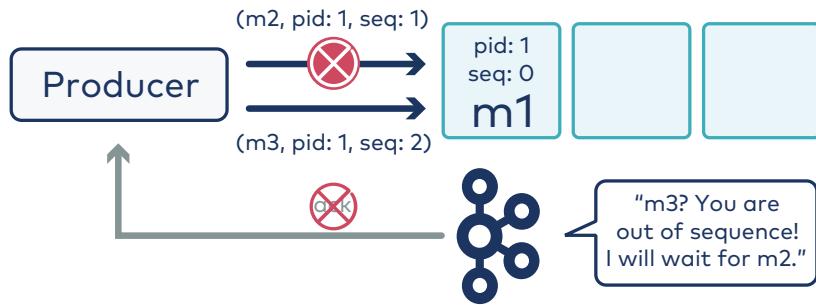
- `max.in.flight.requests.per.connection` > 1

- `retries` > 0

We've not addressed `max.in.flight.requests.per.connection` before, so... it allows for pipelining. Remember from the module "Preparing Producers for Practical Uses" that calling `send()` puts a message into the producer's local buffer and doesn't simply write it to the producer; a lot happens before the message is successfully written. Generally we **want** to allow multiple requests to be going between the producer and the cluster, and `max.in.flight.requests.per.connection` has a default value of 5 in that spirit. But...

With `max.in.flight.requests.per.connection` and `retries` producer properties set to greater than 1, it is possible for messages to be appended to the log out of order. In this example, we see record batch m1 fails for some reason, but m2 succeeds. The producer retries m1, and thus the log has the records from m1 and m2 out of order.

Solution: Idempotent Producers

Here we see record batch m1 has been appended to the log with sequence number 0 and there are two in-flight requests to the broker for m2 and m3. For some reason, m2 fails to reach the broker, but m3 doesn't. The broker notices that the sequence number for m3 is not the next number in the sequence for this particular producer, so it sends an `OutOfOrderSequenceException`.

Remember, the record batches will sit in the producer's memory buffer for up to the time configured by `delivery.timeout.ms` (Default: 2 minutes). After this time, the producer would decrement the sequence number of batches it is sending. This allows future batches to be accepted by Kafka and not rejected because of missing sequence numbers due to expired batches.

Activity: Justifying a Configuration PR

Quick review:

You made a pull request wherein you made one change to your producer code, adding this:

```java
props.put("enable.idempotence", "true");
```

A colleague is hesitant to approve this and asks you why you did this. Tell your colleague two problems this could prevent (better if you can phrase them in the context of what your company/application does).

Manual Committing and Weaker Message Delivery Guarantees

If you want to commit offsets manually, set `enable.auto.commit=false`

Synchronous

- `commitSync()`

- **blocks** until success or exception

Asynchronous

- `commitAsync()`

- **non-blocking**, should have **callback**

But be careful! Consider these two code examples:

```
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"?

- A consumer can manually commit offsets to control the committed position

  - Disable automatic commits: set `enable.auto.commit` to `false`

- `commitSync()`

  - Blocks until it succeeds, retrying as long as it does not receive a fatal error

  - For "at most once" delivery, call `commitSync()` immediately after `poll()` and then process the messages

  - Consumer should ensure it has processed all the records returned by `poll()` or it may miss messages

- `commitAsync()`

  - Returns immediately

  - Optionally takes a callback that will be triggered when the Broker responds

  - Has higher throughput since the consumer can process next message batch before commit returns

- Tradeoff: Consumer may find out later the commit failed

- Tradeoff: Less control over commits

- `commitAsync` is also useful for slow-performing consumers (e.g. consumers who need to call an external service API), as an alternative to increasing the session timeout

This slide doesn't explicitly show that `commitSync/commitAsync` can be called with parameters to specify topic/partition/offset to commit, allowing more fine-grained offset updates, not just committing the whole batch returned in the last poll call.

- A consumer can combine both `commitSync` and `commitAsync`:

  - `commitAsync()` during normal processing of messages and

  - `commitSync()` just before exiting the Consumer or before a rebalance (in the "finally" clause)

Developers should make sure to commit offsets before a partition rebalance by using the sync methods as part of the `onParititionsRevoked` method.

Regarding the examples...

The code block on the left represents "at most once" delivery. The consumer commits its offset before actually processing the records, so if there is a failure during processing, the next consumer will pick up the committed offset and skip all of the records from the poll in line 1. This ensures no record will be reprocessed, but at the cost of potentially skipping records.

The code block on the right represents "at least once" delivery. The records are processed before the consumer commits its offset, so if there is a failure during processing, the records from the poll in line 1 will be reprocessed. This ensures all records are processed at least once, but at the cost of potentially reprocessing records.

14b: How Should You Deal with Kafka's Message Size Limit?

Description

Understanding message size limits, best practices, and strategies for dealing with natural demands for large messages.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- State the default message size max

- Explain why not to change the message size limit

- State strategies for dealing with applications needing larger messages

Kafka's Message Size Requirements

- Broker default for `message.max.bytes` is 1 MB

- Producer default for `max.request.size` is also 1 MB

ℹ	Confluent Cloud has larger defaults.

- 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

Question: So how can we deal with this?
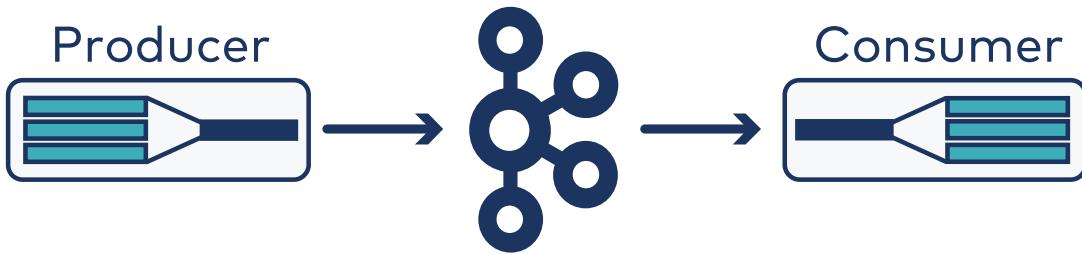
Kafka is not optimized for large messages. When a Broker receives a large record batch, a byte buffer is allocated to receive the entire record batch. The larger the record batch, the more likely that performance issues might occur (e.g., fragmentation in the Java heap). The default maximum is 1 MB, and it is **strongly recommended** not to change this default. The vast majority of use cases can be attained with record batch sizes of less than 1 MB.

- The occasional large record won't degrade broker performance, so if you absolutely must change the maximum size for a batch of messages that the broker can receive from a producer:

  - globally, adjust `message.max.bytes` on broker

  - per topic, adjust `max.message.bytes`

  - on the producer, adjust `max.request.size`

ℹ	Confluent Cloud has a max message size of 8 MB for the standard package and 20 MB for a dedicated cluster. See [this page](#) for more details and the most up-to-date information on Confluent Cloud sizing and options. If large message sizes are business critical, then Confluent Cloud is an excellent option.
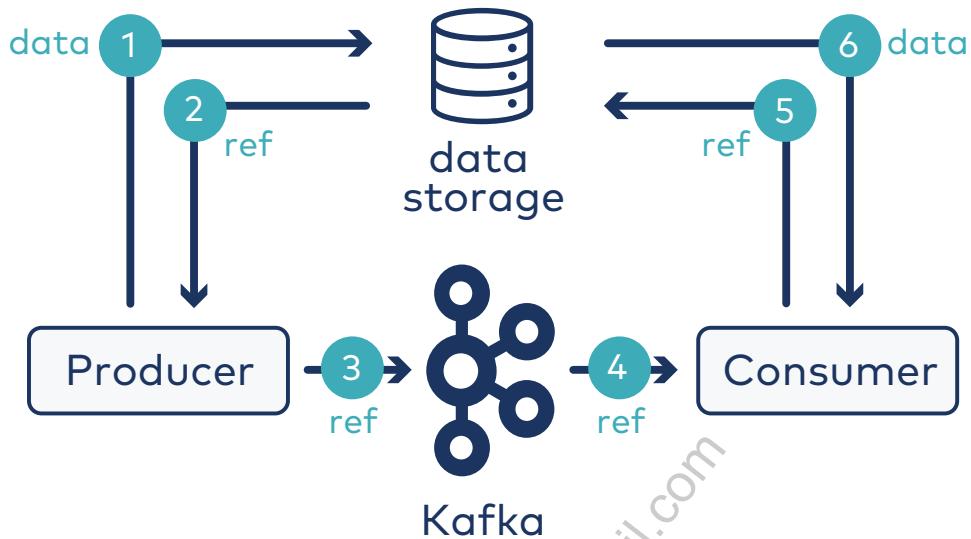
Solution 1: Compression

1. Producer batches and then compresses the record batch

2. Compressed record batch stored in Kafka

3. Consumer decompresses

Change the producer's `compression.type` property from `none` to `gzip`, `snappy`, `lz4`, or `zstd`. Compression applies to full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression). The compression type used by a producer is noted as a header of the record batch. This allows multiple producers writing to the same topic to use different compression types. Consumers will decompress batches of messages according to the compression type denoted in the header.

Solution 2: Pass Reference to External Store

If compression alone isn't enough, then another strategy is to store a large message in object storage like S3, a document database like MongoDB, or a cache like Redis, and then produce a **reference** to the data's true location to Kafka. The consumer will read that reference and retrieve the data from the external system. This design pattern is sometimes called "claim check," since it behaves like a luggage claim at the airport.

This strategy brings new complexity. Teams must manage a new external system and account for more failure scenarios. For example, what happens if the producer fails after sending the large message but before sending the reference to Kafka? There are a few ways to handle this, like living with the fact that some messages won't be claimed and settings time-to-live (TTL) for the message in the external system to clean up unclaimed messages.
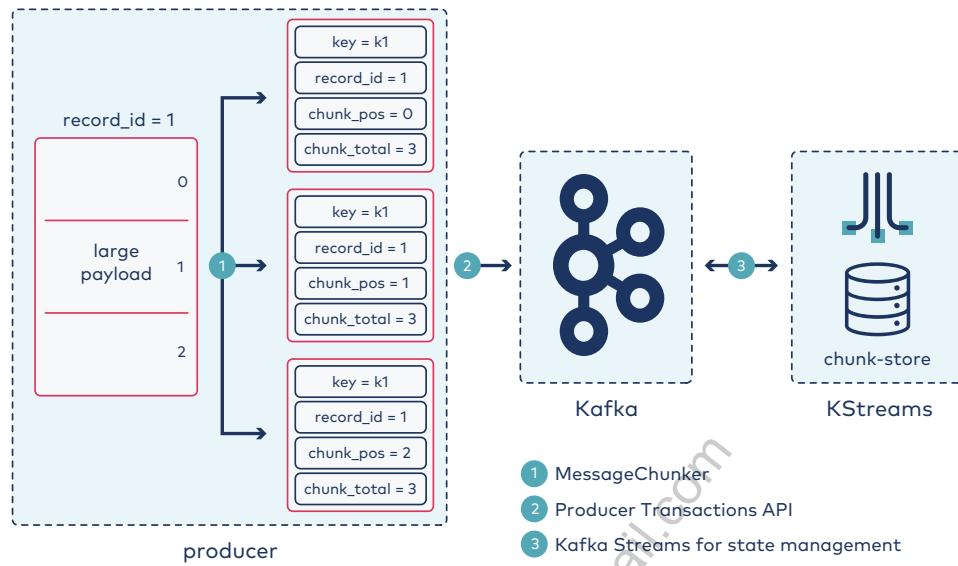
A more complex solution would be:

1. Set unique keys for each large message.

2. Produce a "preparing reference" record to Kafka on a compacted topic.

3. Write the data to the data store and receive the reference.

4. Produce the reference record to Kafka, which deletes the "preparing reference" record due to compaction.

5. When a consumer receives a "preparing reference" message, it sets a timeout. When the timeout is reached, throw an exception to record the fact that the reference was not received in that time.

The benefit to this complex solution is that there would be a trace of which large messages were unclaimed, which allows for more flexibility in how to recover.

Solution 3: Record Chunking

producer

1. MessageChunker
2. Producer Transactions API
3. Kafka Streams for state management

If compression isn't enough and reference passing doesn't suffice, then consider breaking down large payloads into "chunks" and sending those chunks as individual messages. This is quite advanced, so we only discuss a brief overview of the strategy. Here are the steps:

1. The first step is to implement some sort of `MessageChunker` class to break a large payload into a list of Kafka producer records.

2. Once the records are made, they must be sent to Kafka atomically and in order. This requires the `enable.idempotence=true` property and the producer's transactional API. Transactions are usually done in the context of Exactly Once Semantics, but this is a use case where transactions are appropriate outside of this pattern. Transactions will be discussed in greater detail in an upcoming lesson.

3. The difficulty of consuming these messages is state management. The consumer must keep track of the message chunks and assemble them into the original payload when all chunks are present. You could implement your own "chunk-store" in the consumer, but if there is a rebalance, that state would have to be shuffled to the new consumer reading from that partition. As we have seen, Kafka Streams is designed to handle state shuffling elegantly, so the approach would be to use the Kafka Streams API (the Processor API in particular) to create a state store to track the chunks. Note that since the chunks were produced using a transaction, the Kafka Streams application's consumers must be configured with `isolation.level=read_committed`. This will be explained in more detail in an upcoming lesson.

14c: How Do You Send Messages in Transactions?

Description

What a transaction is, committed vs. aborted transactions, how Kafka marks them, and what a single-partition log looks like with transactions involved. Consumer side of transactions. Code for a transactional producer. How the above would change if the producer first consumed or if it produced to more than one partition.

Learning Objectives

Upon completion of this lesson and associated lab exercises, you will be able to:

- Explain what a transaction is

- Distinguish between a transaction that should be committed vs. aborted

- Explain how Kafka marked committed vs. aborted messages in the logs—and why not other ways

- Illustrate a log for a partition that contains messages from committed transactions, aborted transactions, and non-transactional messages

- Specify what configuration change must happen on the consumer end to make a consumer transaction aware

- Explain—or edit—code for a simple application that is only producing and producing one transaction

- Illustrate what is different if messages form a transaction

- Explain what one must do differently in code if an application is consuming and processing before producing transactional messages and why

Overview

The idea: Group messages together as a **transaction**. Process them only if all can be processed.

Kafka has a Transactions API. Support for this is part of core Kafka.
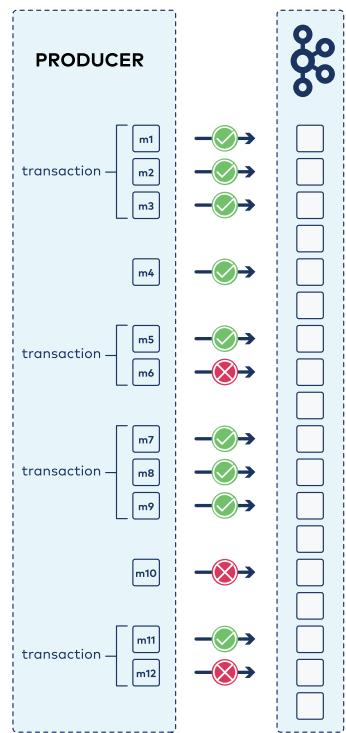
Interactive Example Overview

A producer is going to send some messages in transactions and some other messages
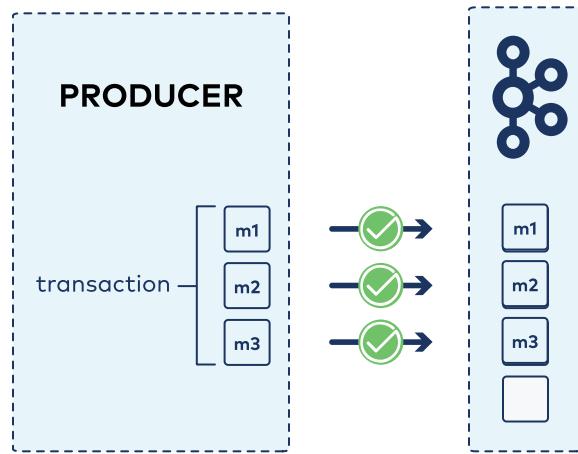
We see here which ultimately succeed and fail to make it to Kafka

We'll assess sequentially which ones consumers should process

Interactive Example - Scenario 1

Messages 1, 2, and 3 are in a transaction. All are successfully written to Kafka.

Q: For each message, should a consumer be allowed to read it and process it?

Virtual Classroom Poll:

consumer should be allowed to process message
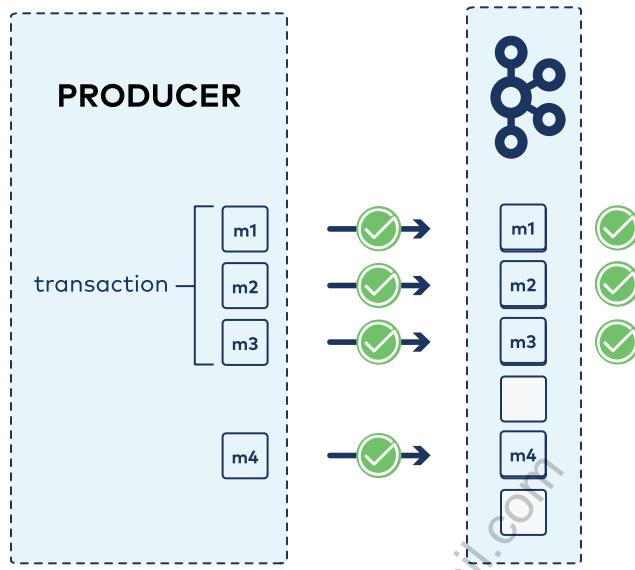
consumer should not be allowed to process message

We won't know the fate of transactional messages until all have been sent. There are a few cases. In this case, once we know all messages in the transaction have made it to Kafka, we can assess whether consumers should be able to read those messages. Since the whole transaction was successful in this case, consumers should be able to read all of these messages.

Interactive Example - Scenario 2

Message 4 is not in a transaction. It is successfully written to Kafka.

Q: For each message, should a consumer be allowed to read it and process it?

Virtual Classroom Poll:

consumer should be allowed to process message
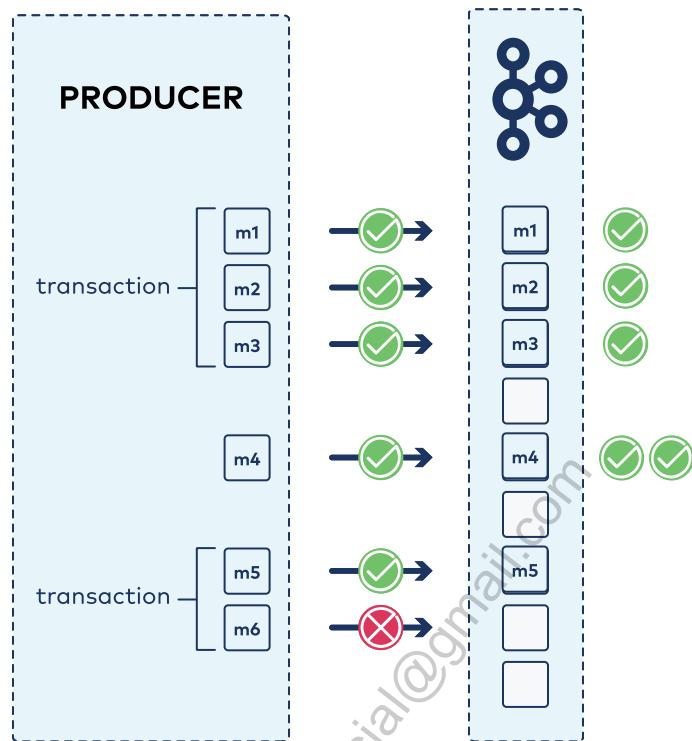
consumer should not be allowed to process message

Just because we have a transactional producer does not rule out the ability to send messages that are not part of a transaction. Those messages will always be okay to consume.

Interactive Example - Scenario 3

Messages 5 and 6 are in a transaction. One is successfully written to Kafka; one is not.

Q: For each message, should a consumer be allowed to read it and process it?

Virtual Classroom Poll:

✅ consumer should
be allowed to
process message
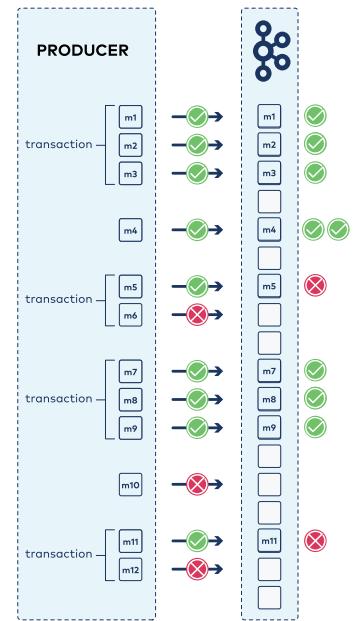
❌ consumer should
not be allowed to
process message

Once again, we can't assess whether consumers should consume messages in a transaction until all have been attempted (and retries and timeouts have run out). In this case, one message in our transaction made it to Kafka and one did not. Consumers should **not** process the message that made it to Kafka, so we can call it "bad."

Putting it all Together and Going Further
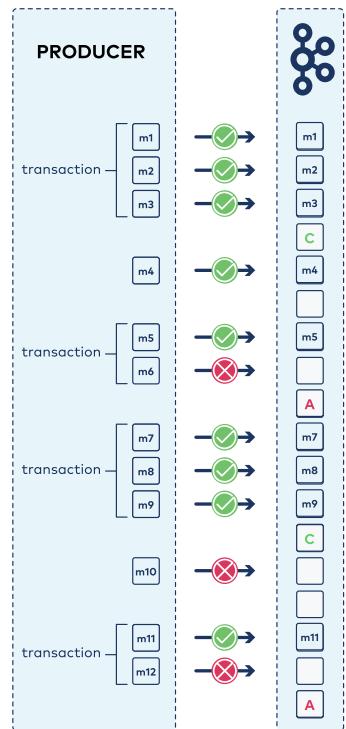
Here's everything from the last three slides and more:

Question: So how can Kafka logs denote committed transactions?

This slide shows everything we've seen, along with a few more message send attempts.

Commit and Abort Markers

- Remember, messages are **immutable**.

- `commitTransaction()` causes a C marker to be written to the log

- `abortTransaction()` causes an A marker to be written to the log

- Consumers use these markers

We can**not** put anything in the metadata to tag messages as good or bad, nor can we delete individual messages. In Kafka, we use commit markers and abort markers—special messages added to the logs after successful or unsuccessful transactions. Consumers can use these markers, along with metadata added to messages that are transactional, to "filter out" messages that are part of failed transactions. How do you make that happen? See the slides to come!

You may notice we say each of the above methods "causes a ... marker to be written to the log" rather than "writes..." It is a process called the Transaction Coordinator that handles this. See the "A Step Beyond" page at the end of the lesson to learn more.

What about what happens if a producer dies before we get to call `abortTransacation()`? Well, the transaction will timeout based upon the `transaction.timeout.ms` setting (default: 1 minute). When that happens, the transaction coordinator will write abort markers for all messages that are part of the incomplete transaction. Details.

Setting up a Transactional Producer

1. Need EOS → turn on idempotence

   ◦ Set `enable.idempotence = true`

   ◦ Causes messages to have metadata:

      ▪ Producer ID

      ▪ Sequence number

2. Start transactions

   ◦ Call `producer.initTransactions()`

   ◦ Metadata effects:

      ▪ Messages will have transactional ID

      ▪ All messages in a transaction share that transactional ID

For transactions to work, we need three things added to the metadata. The two steps here make that happen.

Transactional Producer Code Example

```
 1  producer.initTransactions();
 2
 3  try
 4  {
 5      producer.beginTransaction();
 6      producer.send(...);
 7      producer.send(...);
 8      producer.send(...);
 9      producer.commitTransaction();
10  }
11  catch(ProducerFencedException pfe)
12  {
13      producer.close();
14  }
15  catch(KafkaException ke)
16  {
17      producer.abortTransaction();
18  }
```

Notes:

- We must tell our producer to be transactional. That's Line 1.

- A transaction might succeed, but it might fail, so we put our transaction in a `try` block, seen on Lines 3-10.

- In a happy case, we do our sending between starting our transaction (Line 5) and committing in (Line 9).

  - As seen in Lines 6 to 8, we can have two or more `send(…)` calls.

  - Line 9 will cause the C marker(s) to be written.

- If one or more messages in our transaction fails, it will cause an exception, leaving the `try` block. This triggers the exception caught in the `catch` block in Lines 15-18.

  - Line 17 will cause the A marker(s) to be written.

  - There is no special exception for a message in a transaction failing, so a standard `KafkaException` will do, but...

- The `ProducerFencedException catch` block beginning on Line 11 is necessary in the case of any "zombie producers" that would have transactional metadata that would cause consumers not to be able to handle this transaction successfully.

  - As it is a kind of `KafkaException`, we must catch it first.

Consuming with Transactions

- Consumer property `isolation_level` has two options:

  - `read_uncommitted` (default) - process all messages, whether they are part of transactions or not

  - `read_committed` - process transactional messages from **committed** transactions only, as well as all non-transactional messages
    → **Set this if using transactions**

⚠️	This still does not protect against failures on the consumer end.

In addition to the setup on the producer end from two slides back and using the proper transactions API on the producer end, we must also configure consumers to think transactionally. This setting is how to do that.

Back in Kafka, the leader maintains a marker called the last stable offset (LSO), the smallest offset of any open transaction. When a consumer has `read_committed` set, the response returned by `consumer.poll()` only includes records up to the last stable offset.

What if Messages in a Transaction Land on Different Partitions?

Producers could send messages from the same transaction to different partitions. This is okay:

- Need to write C or A markers to all affected partitions

- Kafka's Transactions API handles remembering affected partitions for this

balance-update topic
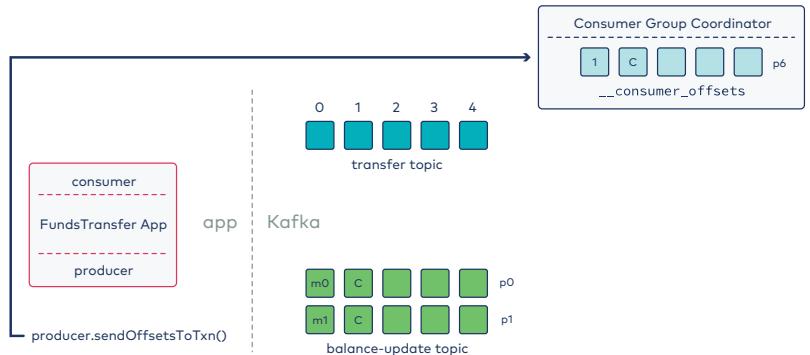
You don't need to do anything special here—Kafka takes care of it for you—but you should be aware of this matter. The Producer API handles partitioning and might send messages in a transaction to different partitions. The Transactions API tracks affected partitions and writes the C and A markers to **all** affected partitions.

What if I Consume, Process, and THEN Produce Transactionally?

- Want to advance relevant committed consumer offset only if transaction was successful

- We must also call sendOffsetsToTxn()

    - Causes C to be written in __consumer_offsets too

Recall the **consume-process-produce** model of processing. The consumption step might result in producing *multiple* messages that all need to be successful, so transactions may be needed.

Consider transferring money from one bank account to another. One might

1. **Consume** from a topic containing transfer requests.

2. **Process** those transfer requests, i.e. set up the need to:

    a. withdraw money from one account.

    b. deposit money to another account

3. **Produce** the messages to change the balances of the two affected accounts. Because **both** things must happen, we need a transaction.

Should our transaction succeed, we would want the consumer offset committed for the transfer topic to advance, like in normal consumption. But should the transaction fail, we do not. The additional line of code makes this happen properly.

Here's a revised copy of the code from a few slides back showing the additional line of code, now Line 9:

```java
producer.initTransactions();

try
{
    producer.beginTransaction();
    producer.send(...);
    producer.send(...);
    producer.send(...);
    producer.sendOffsetsToTxn();
    producer.commitTransaction();
}
catch(ProducerFencedException pfe)
{
    producer.close();
}
catch(KafkaException ke)
{
    producer.abortTransaction();
}
```
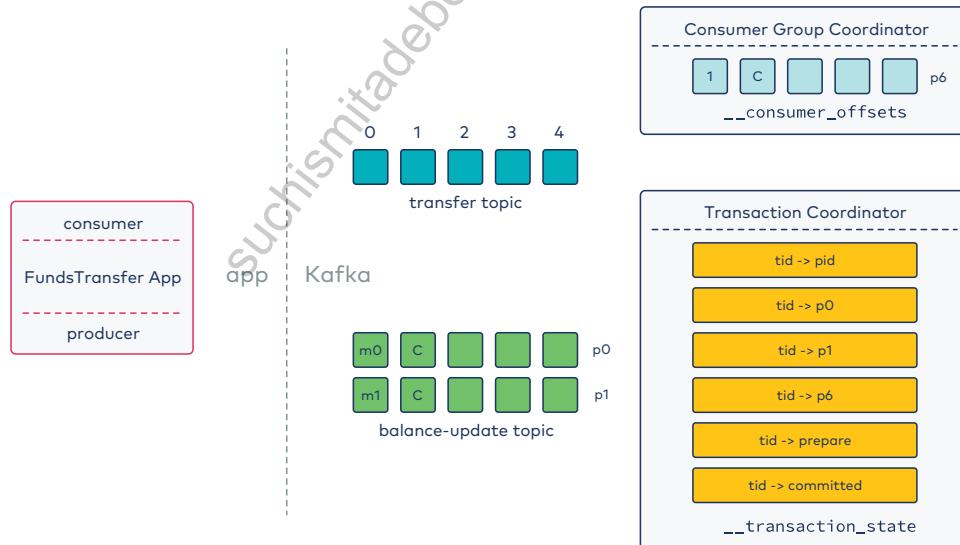
The Transaction Coordinator

Behind the scenes,

- There is a **Transaction Coordinator** making everything happen.

- It maintains a **transaction log**

  ◦ Using internal topic `__transaction_state`

  ◦ Tracks the status of the transaction

  ◦ In the event of failure, is used to prevent the need for redoing work

  ◦ All affected partitions are recorded in the log

    ▪ So Transaction Coordinator knows where to write markers

    ▪ This is both in data partitions and partitions of `__consumer_offsets` if using the consume-process-produce model

Here's a visual:

See the Appendix for a 14-step demo with a great deal of detail of a consume-process-produce application that uses transactions. The image here is the concluding step of that demo.

Here's a [blog post with a lot more detail](#)

CONFLUENT
Global Education

Module Overview

This module contains two lessons:

    a. What Should You Think About When Testing Kafka Applications?

    b. How Can You Leverage Error Handling Best in Kafka Connect?

Where this fits in:

- Hard Prerequisite: Starting with Consumers, Kafka Connect

- Recommended Prerequisite: Groups, Consumers, and Partitions in Practice

- Recommended Follow-Up: Other modules in this branch, other courses

15a: What Should You Think About When Testing Kafka Applications?

Description

Testing considerations as they apply to Kafka development.

Learning Objectives
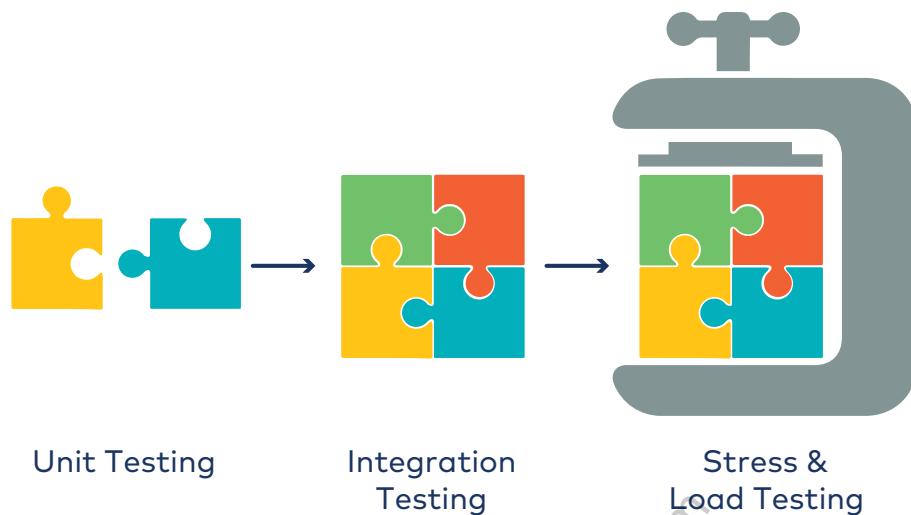
Completing this lesson and associated exercises will enable you to:

- Create a plan for testing producers and consumers in your Kafka deployment.

- Distinguish between three categories of testing.

- List some considerations to stress and load test a Kafka system.

Kinds of Testing

| Unit Testing | Integration Testing | Stress & Load Testing |

Here we note four different classes of testing:

- Unit Testing - testing individual components of your system - i.e., producers by themseles, consumers by themselves - to see that they work on their own

- Integration Testing - testing what happens when the inidividual components are put together

- Stress and Load Testing - testing what happens when we put the system under more stress than normal use cases
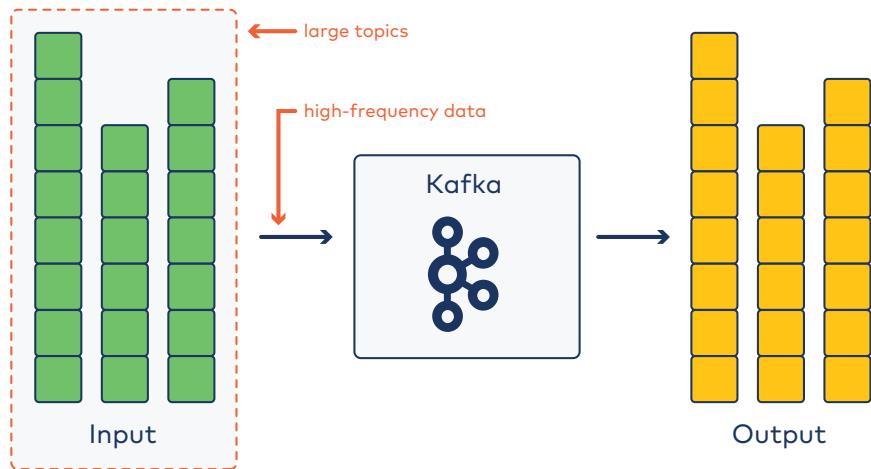
We'll go into some more detail on the slides to come.

Considerations

- Test inidiviudal producers and consumers before testing how they integrate with Kafka.

- Mock or stub dependencies for unit testing.

- Consider the "Anatomy of a Kafka Streams Application" lesson

  ◦ Having `getTopology()` and `getProperties()` as individual methods facilitates better testing.

Stress & Load Testing Considerations

Input → Kafka → Output

large topics, high-frequency data

Some things to consider testing:

- You should have an expectation for how much data you expect to be produced under normal use cases. Stress the system by trying to produce more. How does it react?

- You should have an expectation for how much data you expect to be consumed under normal use cases. Stress the system by trying to produce more. How does it react? How would your consumers hold up on the day of a new product release of a sale that had 10 times as many purchases as a normal day?

- Consider the "key space," i.e., how many distinct keys your data has. What is the effect of having many more keys than normal?

Activity

Class discussion:

- What are some things you found out about a system via testing that you wouldn't have thought of otherwise?

- What are some experiences you've had testing a Kafka system that you can share with us?

15b: How Can You Leverage Error Handling Best in Kafka Connect?

Description

Kafka Connect error handling framework options. Dead letter queue.

Learning Objectives

Completing this lesson and associated exercises will enable you to:

- Compare and contrast error handling options provided by the Kafka Connect framework.

- Describe how a dead letter queue applies in Kafka Connect.

Handling Errors in Connect

Copying to/from external systems can fail...

- source system unavailable

- destination system unavailable

- messages that cannot be processed

- transient failure

What to do?

- Give up completely?

- Allow some errors?

- Retry on failure?

- Log problem messages to handle separately?

A common situation with any software that ingests data into Kafka, or other systems such as a database, is error management. How to handle situations where the target or source is unavailable, overloaded or simply what to do when the record is stored and failed, for example for deserialization.

Since Apache Kafka 2.0, KIP-298 was introduced to have a common error management for Kafka Connect, so errors don't have to be handled on a per-connector basis.

Fail Fast Scenario

Why?	How?

• **Poisoned messages**
 i.e., cannot be processed
 	
• Source/target system unavailable
	Confirguration settings:
	
	``
	# disable retries on failure (default 0)
	errors.retry.timeout=0
	
	# do not log the error and their
	contexts
	errors.log.enable=false
	
	# do not record errors in a
	# dead letter queue topic
	errors.deadletterqueue.topic.name=""
	
	# Fail on first error
	errors.tolerance=none
	``

The fail-fast (i.e. non-managed errors) behavior can be achieved by disabling the error management framework, this can be done with the configuration shown on the right side of the slide. This is certainly not a good practice; if we are running an Apache Kafka version newer or equal than 2.0.0, it is always recommended to enable error management.

A situation that we may find is if a connector shows a status of `FAILED`, but we are not able to bring it back to a `RUNNING` state, even after multiple restart attempts.

We should work to remove the impediment that is causing this fast failure. Usual problems that could cause this situation are poisoned messages as well when the source or target systems are not available, or underperforming.

What are **Poisoned messages**?
These are messages that can not be processed and so are rejected. As the message was not successfully consumed, the connector will retry again and again.

The second situation is **source or target systems are unavailable**:
A situation that could arise from time to time in our deployment is when one of the systems the connectors are pulling data from or pushing data to becomes unavailable. This could be for multiple reasons, e.g., this target system is being taken offline for maintenance, or an upgrade, or simply because the target system is currently overloaded.
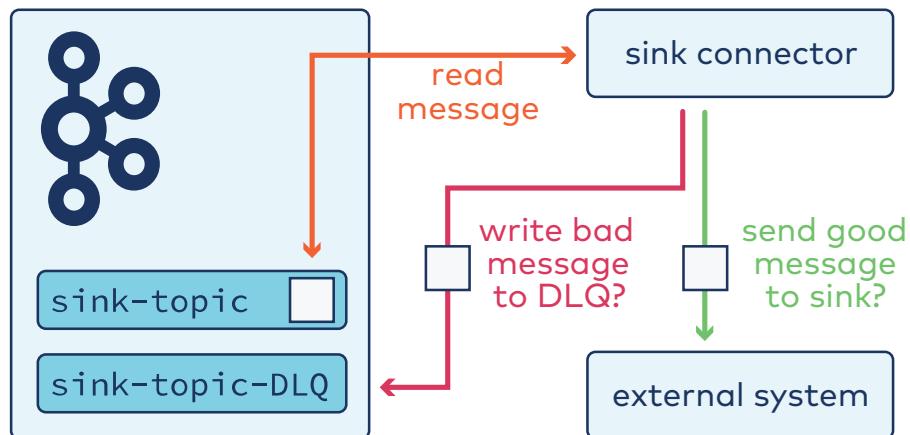
Each connector handles this slightly different. The JDBC connector handles connection attempts or operation retries, both with exponential back off for easy recovery. Other popular connectors such as the HDFS sink have an option to retry delivery of messages with exponential back off.

To plan for such scenarios, it is very important to check the different options present in the configuration of the connector and search for how each one is handling timeouts and reconnection attempts.

Dead Letter Queue

- **Problem:** Writing message to external system fails

- **Solution:**

  - Rather than giving up, produce this message to a special Kafka topic
  → Called the **dead letter queue (DLQ)**

  - Can inspect those messages separately and decide how to handle

⚠️	The DLQ is for sink connectors only.

More on DLQ:

- [Deep Dive on Connect Error Handling](#)

- [DLQ in Confluent Cloud](#)

- Retries, configurable via the settings on the slides to come, happen before a message that could not be written to the sink system is written to the DLQ.

Error Management Options

To configure error management, configure the **connector** settings:

Name	Default	Source Connectors?	Sink Connectors?
errors.retry.timeout	0	yes	yes
errors.retry.delay.max.ms	1 min	yes	yes
errors.tolerance	-	yes	yes
errors.deadletterqueue.topic.name	""	**no**	yes
errors.log.enable	false	yes	yes
errors.log.include.messages	false	yes	yes

The Connect error framework handles:

- **Retry on failure:** Which handles how an operation is retried after failing.

- **Task error tolerance:** How many errors to tolerate per task.

- **Dead letter queue:** For sink connectors, the original record (from the Kafka topic the sink connector is consuming from) which caused the failure will be written to another topic used as queue.

See next slide for an example configuration.

Recommended Error Management Config

Here's an example of configuring error management:

```
# retry for at most 10 minutes waiting up
# to 30 seconds between consecutive failures
errors.retry.timeout=600000
errors.retry.delay.max.ms=30000

# log error context along with application logs
# but do not include configs and messages
errors.log.enable=true
errors.log.include.messages=false

# produce error context into the Kafka topic
errors.deadletterqueue.topic.name=my-connector-errors

# Tolerate all errors.
errors.tolerance=all
```

Conclusion

Course Contents

Now that you have completed this course, you should have the skills 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 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 polices

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®

- Managing Data in Motion with Confluent Cloud

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

- **Confluent Stream Processing Using Apache Kafka® Streams & ksqlDB** covers:

    ◦ Identify common patterns and use cases for real-time stream processing

    ◦ Understand the high level architecture of Kafka Streams

    ◦ Write real-time applications with the Kafka Streams API to filter, transform, enrich, aggregate, and join data streams

    ◦ Describe how ksqlDB combines the elastic, fault-tolerant, high-performance stream processing capabilities of Kafka Streams with the simplicity of a SQL-like syntax

    ◦ Author ksqlDB queries that showcase its balance of power and simplicity

    ◦ Test, secure, deploy, and monitor Kafka Streams applications and ksqlDB queries

- **Apache Kafka® Administration by Confluent** covers:

    ◦ Data Durability in Kafka

    ◦ Replication and log management

    ◦ How to optimize Kafka performance

    ◦ How to secure the Kafka cluster

    ◦ Basic cluster management

    ◦ Design principles for high availability

    ◦ Inter-cluster design

- **Confluent Advanced Skills for Optimizing Apache Kafka**

- Formulate the Apache Kafka® Confluent Platform specific needs of your organization

- Monitor all essential aspects of your Confluent Platform

- Tune the Confluent Platform according to your specific needs

- Provide first level production support for your Confluent Platform

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

Benefits:

- Recognition for your Confluent skills with an official credential

- Digital certificate and use of the official Confluent Certified Developer Associate logo
  Exam Details:

- The exam is linked to the current Confluent Platform version

- Multiple choice questions

- 90 minutes

- Designed to validate professionals with a minimum of 6-to-9 months hands-on experience

- Remotely proctored on your computer

- Available globally in English

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

This course prepares you to manage a production-level Kafka environment, but does not guarantee success on the Confluent Certified Administrator Certification exam. We recommend running Kafka in Production for a few months and studying these materials thoroughly before attempting the exam.

Benefits:

- Recognition for your Confluent skills with an official credential

- Digital certificate and use of the official Confluent Certified Administrator Associate logo

Exam Details:

- The exam is linked to the current Confluent Platform version

- Multiple choice and multiple select questions

- 90 minutes

- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience

- Remotely proctored on your computer

- Available globally in English

We Appreciate Your Feedback!

Please complete the course survey now.

Your instructor will give you details on how to access the survey

Thank You!

Appendix: Additional Problems to Solve

Overview

This section contains a few additional problems to be solved that will reinforce the concepts in this course.

These problems were originally written as warm-up problems for instructor-led training for this course. Your instructor may or may not choose to incorporate some or all of these problems in class; you may find them to provide additional enrichment in any case. Some other problems originally created as warm-up problems have been adapted into activities in the content of this version of this course.

Problem A: Comparing Producers and Consumers

We looked at Java code for writing custom producers and consumers yesterday...

a. To create producers, we instantiated objects of three different classes. What were they? How are they related?

b. What are the analogous classes for consumers?

c. What additional step must we do for consumers in setting them up? Why not for producers?

d. What is the main operation a producer does? A consumer?

Prerequisite Modules:

• 2: Starting with Producers

• 4: Starting with Consumers

Problem B: Partitioning with Keys
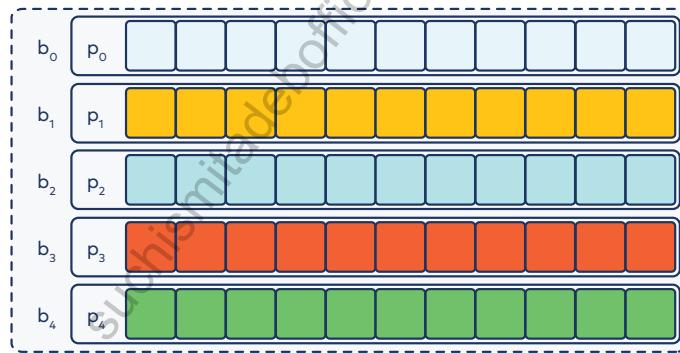
Suppose we have 5 brokers and 5 partitions. The five partitions are p_0, p_1, p_2, p_3, and p_4; and they are stored on brokers b_0, b_1, b_2, b_3, and b_4, respectively. Suppose we are using default Kafka settings and `hash(n) = n`. Then...

Suppose we send these messages:

- m_0 with key 1

- m_1 with key 7

- m_2 with key 12

- m_3 with key 18

- m_4 with key 27

- m_5 with key 10

Which partitions contain which messages after the Kafka cluster has received them all?

Here's an illustration of the situation:
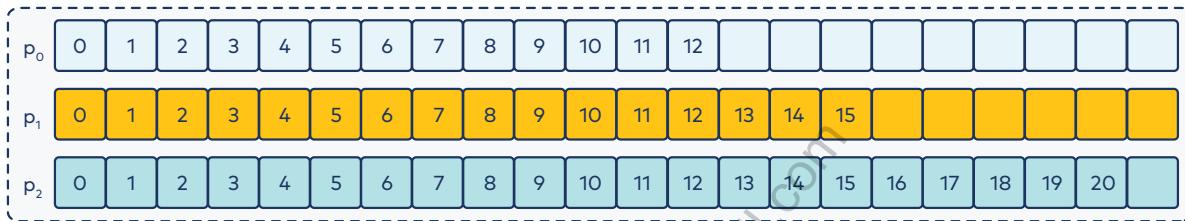
Prerequisite Modules:

- Fundamentals Class

- 2: Starting with Producers

Problem C: Groups, Consumers, and Partitions

Suppose we have partitions p_0, p_1, p_2 each with first offset 0 and with most recent offsets of 12, 15, and 20, respectively (where every offset between contains data). Suppose we have consumer group g_0 with consumers c_0, c_1, and c_2 and consumer group g_1 with consumers c_3, and c_4. (Some details may be missing - you interpret!)

The setup might look like this:

Your quest:

a. Give a valid assignment of consumers to partitions that could result.

b. How many different consumer offsets are stored in this scenario? Explain/list them.

c. For each consumer offset in your list, give a valid value it could have

d. Suppose c_1, goes down. What happens? Concretely illustrate the scenario now.

e. For any one consumer, tell the offsets of the messages read in the case that it gets 2 messages in the next batch. Repeat for some other consumer in the case that this other consumer gets 3 messages in the next batch.

f. Give two examples of things that could happen that would trigger a consumer/partition assignment rebalance. One must not involve anything with consumers changing (before the rebalance).
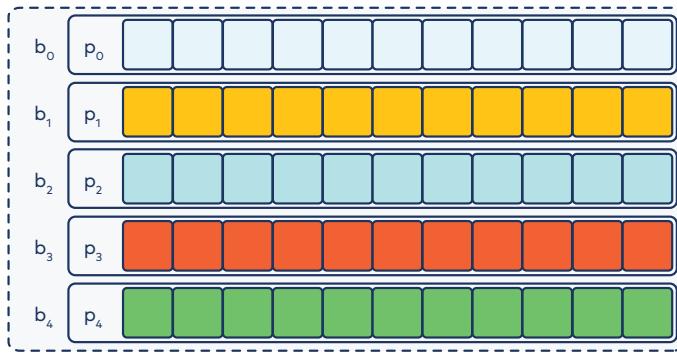
Prerequisite Modules:

• 5: Groups, Consumers, and Partitions

Problem D: Partitioning without Keys

We will again work with this scenario of five brokers and five partitions:

Suppose instead of the keyed messages in another problem, we have messages with null keys.

Skim [KIP 480](#) for more on how Kafka handles partitioning by default when messages do not have keys.

Suppose we have messages that get assigned to partitions in the buffer and these were the batches before being flushed:

batch 0:	a, b, c
batch 1:	d, e
batch 2:	f, g, h, i
batch 3:	j
batch 4:	k, l, m, n
batch 5:	o, p
batch 6:	q, r
batch 7:	s, t, u
batch 8:	v, w, x, y, z

Give an illustration of which messages would land on which partitions...

- ...if you are in a breakout room with an even number: assume we had relatively decent luck, or

- …if you are in a breakout room with an odd number: assume we had not such great luck

---

Prerequisite Modules:

- None, really, other than Fundamentals course.

Appendix: Additional Content

Overview

This appendix contains a few additional lessons. These lessons are for additional information for you, but are not designed the same as the rest; namely, they do not have activities or labs to reinforce the content like the rest.
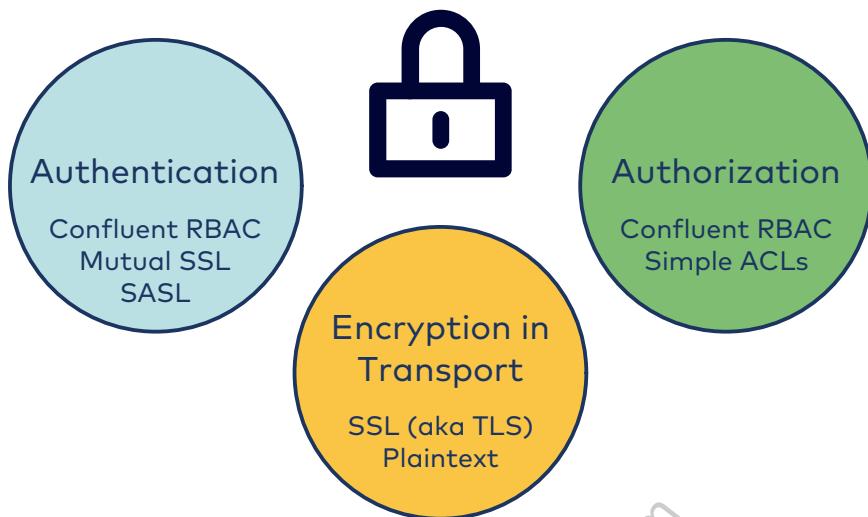
Appendix A: A Taste of Kafka Security for Developers

Description

What Kafka security supports vs. does not for self-managed vs. Confluent Cloud. High level overview of security options in Kafka/CP. Simple, basic config examples.

Security Overview (1)

- **Encryption in Transport:** Can people spy on your messages?

  ◦ Apache Kafka offers encryption over SSL (aka TLS) when the broker's keystore is trusted by the client's truststore. Transport encryption is a requirement in most secure contexts.

    TLS 1.3 is the default TLS protocol when using Java 11 or higher, and TLS 1.2 is the default for earlier Java versions. TLS 1.0 and 1.1 are disabled by default due to known security vulnerabilities, though users can still enable them if required.

- **Authentication:** Are you who you say you are?

  ◦ With mutual SSL, not only does the client trust the broker certificate, but the broker trusts the client certificate as well, which means the client's identity is established.

  ◦ Open source Apache Kafka comes with its own authentication features via SASL (Simple Authentication and Security Layer)

- **Authorization:** Are you allowed to do what you're trying to do?

  ◦ Open source Apache Kafka comes with the ability to manage Access Control Lists

> ℹ Confluent RBAC is a commercial feature of Confluent Platform that provides a unified, flexible, and scalable security posture with respect to authentication and authorization. Details of RBAC are beyond the scope of this course.

Security Overview (2)

Security is mostly out of your hands as a developer, but be aware:

- For all deployments of Kafka, authorization and authentication are supported

- So is encryption. But... what is suppored varies:

  - For self-managed deployments of Kafka, only encryption of data in transport

  - For Confluent Cloud deployments, encryption at rest also (and standard)

- Authorization rights can be set to allow clients to

  - Describe topics

  - Read from topics

  - Write to topics

⚠️	If your application is failing but you suspect your logic is correct, make sure you have the appropriate permissions.

Security falls in the administration track of our training, so we don't go into it in detail here. However, this slide summarizes a few things worth knowing as a developer.

Here's one additional tidbit: You may recall in our code examples where we set `bootstrap.servers` to connect to a cluster, we specified `host:port` pairs. Administrators can configure different ports with different security settings. So, you may end up choosing a port number based on security requirements/configurations.

For more information on all security topics:

- See the security documentation: https://docs.confluent.io/current/security/index.html

- Consider attending **Apache Kafka® Administration by Confluent**

Appendix B: Confluent Cloud vs. Self-Managed Kafka
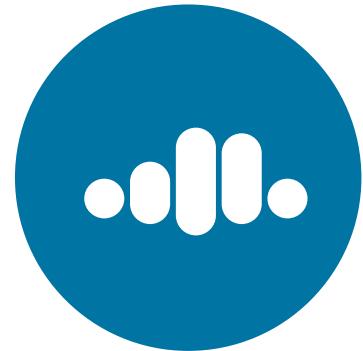
Description

Defining what Confluent Cloud is and comparing CCloud deployments to self-managed deployments. Key considerations for developing clients for CCloud vs. for self-managed deployments.

What is Confluent Cloud?

- Can deploy CP as self-managed software but...

- Confluent Cloud = **fully-managed** deployment of CP

  ◦ Many administrative tasks done for you

- Confluent Cloud available on

  ◦ AWS

  ◦ Google Cloud Platform

  ◦ Microsoft Azure

Confluent Cloud removes some administrative tasks from your organization's responsibilities.

Currently there are two ways to deploy the Confluent Streaming Platform:

1. Fully managed service with **Confluent Cloud** (shown on right)

2. Self-managed **Confluent Platform** (shown on left)

Confluent Cloud™ is a fully-managed event streaming platform service built upon Apache Kafka. Confluent Cloud comes in two flavors:

- **Standard:**

  ◦ Consumption based pricing

- **Dedicated:**

  ◦ Custom pricing

  ◦ Unlimited data throughput and retention

  ◦ Higher SLA

  ◦ Enterprise level security & compliance

  ◦ VPC peering
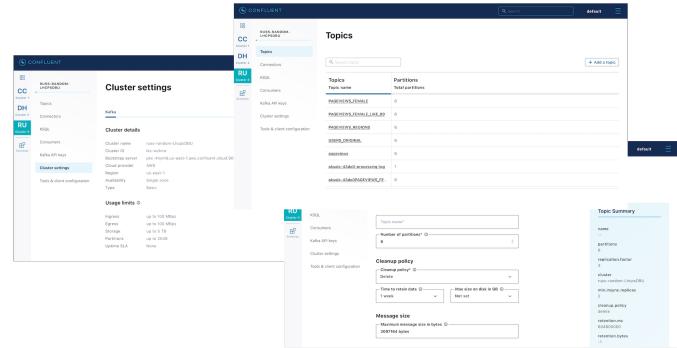
For more details, see https://www.confluent.io/confluent-cloud/compare/

Quick Look at the UI

CLI	GUI

```$ ccloud kafka topic list```
```demo-topic```
```other-topic```
```my-topic```
 	
```$ ccloud kafka topic create product-```
```topic \```
```   --replication-factor 3 \```
``` --partitions 6```	

One more example CLI command:

```
$ 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]
...
```

What's Different?

Topics, partitions, replication, producing, consuming, etc. all behave the same whether you're developing for Confluent Cloud or developing for a self-managed deployment of Confluent Platform.

The biggest differences are administrative:

• With self-managed Kafka, there is no security setup out of the box. With CCloud, there is.

• With self-managed Kafka security enabled, there is no support for encryption of data at rest, only data in transit. With CCloud, data is encrypted at rest.

How Can I Learn More?

- Managing Data in Motion with Confluent Cloud instructor-led training class - one-day

- Free self-paced course "Introduction to Confluent Cloud" available via Content Raven - sign up at http://training.confluent.io

- Introductory exercise using Confluent Cloud in Fundamentals course: Getting Started in Confluent Cloud
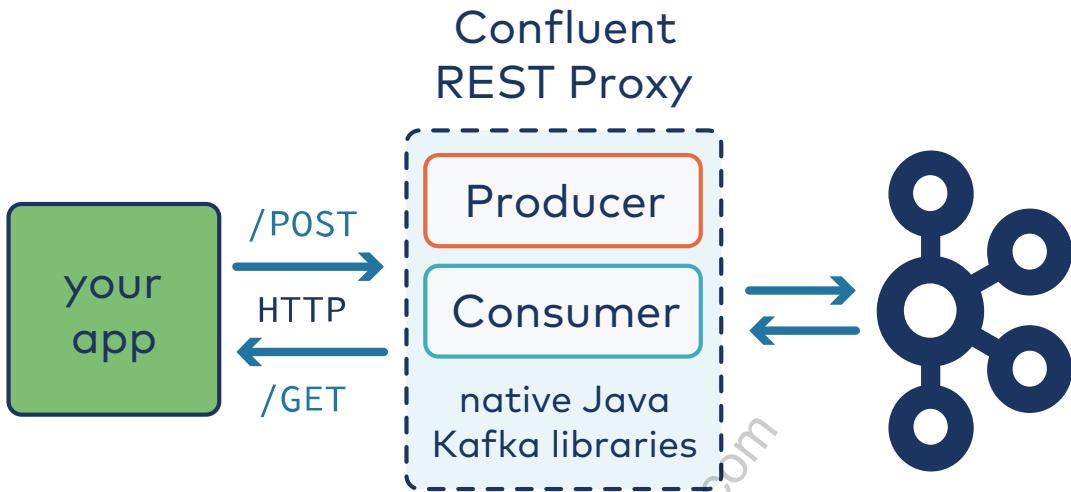
Appendix C: Developing with the REST Proxy

Description

What is the REST Proxy, why one would use it, how a REST producer might look, how a REST consumer might look, examples of other places in the DEV modules where REST fits in.

Confluent REST Proxy (1)

Confluent Community includes a REST Proxy for Kafka. This allows **any language to access Kafka** via a REST interface over HTTP.

REST Proxy API reference: https://docs.confluent.io/current/kafka-rest/api.html#crest-long-api-reference

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

The Confluent REST Proxy is an open source Confluent Community component. The proxy provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some example use cases are:

- Reporting data to Kafka from any frontend app built in any language not supported by official Kafka clients

- Ingesting messages into a stream processing framework that doesn't yet support Kafka

- Scripting administrative actions

> ℹ️ Configuring the REST proxy is not part of this course. Refer to the documentation for more details: https://docs.confluent.io/platform/current/kafka-rest/production-deployment/index.html

Creating a Producer with REST Proxy

```python
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
```

> ℹ️ This is just a Confluent REST Proxy example. This is not the Python producer client.

The target topic is specified as part of the URL, after the REST proxy.

This example shows how to use the JSON format with `vnd.kafka.json`. Refer to the REST Proxy documentation for examples using the other formats (e.g., Avro-encoded JSON).

A question that often comes up: How are all the producer and consumer properties configured through the REST Proxy? In almost all cases, the configs have to be defined on the REST server itself, and cannot be specified as part of the client request.

Creating a Consumer with REST Proxy (1)

- Main Logic

```python
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)
```

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

Using the REST Proxy as a consumer has a few more steps than the producer.

First we define a few global variables:

1. `FORMAT`: defines Kafka's specific JSON format in version 2, used in HTTP request

2. `POST_HEADERS`: headers used for `POST` requests

3. `GET_HEADERS`: headers used for `GET` requests

The actual application logic:

1. Create a consumer instance called `my_consumer` in consumer group `group1`

2. Subscribe the consumer to the topic `hello_world_topic`

3. Now consume messages

4. When done, delete the consumer instance to avoid resource leaks

> ℹ️ `base_uri` is used to identify the consumer instance

Creating a Consumer with REST Proxy (2)

- Creating the Consumer Instance

```python
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"]
```

In this part of the example, we are creating the instance of the consumer. Notice that the url is referencing the consumers rather than a topic name as we saw in the producer example.

1. We define the url to the desired consumer group

2. The payload defines among other things the instance name we want to use

3. Now we do an HTTP POST request to the url

4. In case of an error we stop the application with sys.exit(…)

5. As a last step, we return base_uri, which is used to identify the consumer instance

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")
```

In the body of the POST request we send the list of topics (here only a single one) that we want this consumer to subscribe to.

If there is an error, we delete the consumer and then exit the application

Creating a Consumer with REST Proxy (4)

- Consuming and processing messages

```python
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"])
```

In a more realistic example you would loop indefinitely, since a topic is an open ended stream of data.

Creating a Consumer with REST Proxy (5)

- Deleting the consumer

```python
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)
```

For more information about the REST Proxy API, see
https://docs.confluent.io/current/kafka-rest/api.html#consumers

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

- Connectors can be added, modified, and deleted via a REST API on port 8083

- In distributed mode, configuration can be done only via this REST API

  - Changes made this way will persist after a worker process restart

  - Connector configuration data is stored in a special Kafka Topic

  - The REST requests can be made to any worker

- In standalone mode, configuration can also be done via a REST API

  - However, typically configuration is done via a properties file

    - Changes made via the REST API when running in standalone mode will not persist after worker restart

- Confluent Control Center leverages this REST API to let users configure and manage connectors through the GUI

To make changes to connectors, the Worker will also run a REST API as part of its code. This allows HTTP calls to be sent to any of the Workers that will then configure the connectors. Changes made via the REST API take effect immediately and without a reboot.

> 🛈 The REST API included with a Connect Worker is different from the Confluent REST Proxy.

Using the REST API with Connect

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

At times a requirement exists for the Connect REST API to return certain HTTP headers. The `response.http.headers.config` setting can be used to customize HTTP response headers.

Example:

```
response.http.headers.config="add Cache-Control: no-cache, no-store, must-revalidate",
add X-XSS-Protection: 1; mode=block, add Strict-Transport-Security: max-age=31536000;
includeSubDomains, add X-Content-Type-Options: nosniff
```

Output of Response Header:

```
< HTTP/1.1 200 OK
< Date: Sat, 07 Mar 2020 17:33:39 GMT
< Strict-Transport-Security: max-age=31536000;includeSubDomains
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, must-revalidate
< Content-Type: application/json
< Vary: Accept-Encoding, User-Agent
< Content-Length: 136
```

More information on the REST API can be found at
https://docs.confluent.io/current/connect/references/restapi.html

REST API and CCC

Note that many features of Confluent Control Center use the REST API under the hood...

- ksqlDB interactive mode uses the REST API
  - Can access REST API directly
  - Or indirectly via CCC
- Rather than configuring Kafka Connect connectors via the REST API as just shown, you can use CCC
  - ...but this uses the REST API indirectly

Appendix D: Comparing the Java and .NET Consumer API

Description

A comparison of a basic consumer written in Java with a basic consumer written in C#/.NET.

Comparing a Java Consumer with a C#/.NET Consumer

Here we will look at a partial solution to an old basic Consumer lab in two languages. Remember that the C# client is based on the librdkafka library, so this is, on a different level, a comparison of a JVM-based client and a librdkafka-based client.

Top matter, e.g., `import` / `using` statements, is omitted.

That said, here's the full Java consumer code:

```java
 1 public class ConsumerEx
 2 {
 3     public static void main(String[] args)
 4     {
 5         Properties settings;
 6         KafkaConsumer<String, String> consumer;
 7         ConsumerRecords<String, String> records;
 8
 9         System.out.println("*** Starting VP Consumer ***");
10
11         settings = new Properties();
12         settings.put(ConsumerConfig.GROUP_ID_CONFIG, "vp-consumer");
13         settings.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
14         settings.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
15         settings.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
16                     StringDeserializer.class);
17         settings.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
18                     StringDeserializer.class);
19
20         consumer = new KafkaConsumer<>(settings);
21
22         try
23         {
24             consumer.subscribe(Arrays.asList("vehicle-positions"));
25
26             while (true)
27             {
28                 records = consumer.poll(Duration.ofMillis(100));
29
30                 for (ConsumerRecord<String, String> record : records)
31                 {
32                     System.out.printf("offset = %d, key = %s, value = %s\n",
33                                     record.offset(), record.key(),
34                                     record.value());
35                 }
36             }
37         }
38         finally
39         {
40             System.out.println("*** Ending VP Consumer ***");
41             consumer.close();
42         }
43     }
44 }
```

Here's the full C#/.NET consumer code:

```csharp
 1  class Program
 2  {
 3      public static void Main(string[] args)
 4      {
 5          Console.WriteLine("Starting consumer (.NET)");
 6
 7          var conf = new ConsumerConfig
 8          {
 9              GroupId = "vp-consumer-group-net",
10              BootstrapServers = "kafka:9092",
11              AutoOffsetReset = AutoOffsetReset.Earliest
12          };
13
14          using (var c = new ConsumerBuilder<Ignore, string>(conf).Build())
15          {
16              c.Subscribe("vehicle-positions");
17
18              CancellationTokenSource cts = new CancellationTokenSource();
19              Console.CancelKeyPress += (_, e) => {
20                  e.Cancel = true; // prevent the process from terminating.
21                  cts.Cancel();
22              };
23              try
24              {
25                  while (true)
26                  {
27                      try
28                      {
29                          var cr = c.Consume(cts.Token);
30                          Console.WriteLine($"Consumed message '{cr.Value}' at:" +
31                              " '{cr.TopicPartitionOffset}'.");
32                      }
33                      catch (ConsumeException e)
34                      {
35                          Console.WriteLine($"Error occured: {e.Error.Reason}");
36                      }
37                  }
38              }
39              finally
40              {
41                  c.Close();
42              }
43          }
44      }
45  }
```

The overall structure has some differences to be aware of. But we can pull out seven different things that happen in a consumer:

1. Setting configuration

2. Initializing the consumer

3. Subscribing to topic(s)

4. Polling for records

5. Breaking up the batch received and processing it

6. Looping to continue polling and processing

7. Closing the consumer

We'll look at these in turn in each consumer.

Setting Configuration

Java code:

```java
11        settings = new Properties();
12        settings.put(ConsumerConfig.GROUP_ID_CONFIG, "vp-consumer");
13        settings.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
14        settings.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
15        settings.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
16                    StringDeserializer.class);
17        settings.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
18                    StringDeserializer.class);
```

C#/.NET code:

```csharp
7        var conf = new ConsumerConfig
8        {
9            GroupId = "vp-consumer-group-net",
10           BootstrapServers = "kafka:9092",
11           AutoOffsetReset = AutoOffsetReset.Earliest
12       };
```

The analog to a `Properties` object is populating an instance of `ConsumerConfig`, which then gets parsed automatically to set the properties. ([More](#))

In C#, the common standard serializers can be inferred automatically from the generic types. Serializers can be explicitly configured if needed, though. To set explicit deserializers, could do something like this, replacing line 14 (as addressed below):

```csharp
14       using (var c =
15               new ConsumerBuilder<Ignore, string>(conf)
16                   .SetKeyDeserializer(new AvroDeserializer<string>(schemaRegistry))
17                   .SetValueDeserializer(new AvroDeserializer<User>(schemaRegistry))
18                   .Build())
```

Initializing the Consumer

Java code:

```java
20       consumer = new KafkaConsumer<>(settings);
```

C#/.NET code:

```csharp
14       using (var c = new ConsumerBuilder<Ignore, string>(conf).Build())
```

The setup is similar, but

- The class name is different
- The instantion happens in C# in the `using` block header

Subscribing to Topic(s)

Java code:

```
24              consumer.subscribe(Arrays.asList("vehicle-positions"));
```

C#/.NET code:

```
16              c.Subscribe("vehicle-positions");
```

The only significant difference is the placement of the statement

Polling for Records

Java code:

```
12                records = consumer.poll(Duration.ofMillis(100));
```

C#/.NET code:

```
29                    var cr = c.Consume(cts.Token);
```

The idea is the same, but the syntax is, of course, different.

Breaking up the Batch Received and Processing It

Java code:

```
28              for (ConsumerRecord<String, String> record : records)
29              {
30                  // ...
```

C#/.NET code:

```
// N/A
```

In the Java consumer, we use `ConsumerRecords` and consume zero or more messages at once. The C# consumer - really the `librdkafka` consumer of which it is an instance - is

multithreaded internally and internally manages message buffers. So we're still, potentially, fetching multiple records at a time. There is no need to process the collection.

Looping to Continue Polling and Processing

Java code:

```
26              while (true)
```

C#/.NET code:

```
25                while (true)
```

The idea is exactly the same, and so is the syntax. See the full code blocks for the overall placement and what's inside.

Closing the Consumer

Java code:

```
41              consumer.close();
```

C#/.NET code:

```
41                c.Close();
```

Here again, the idea is the same and the difference is only in casing.
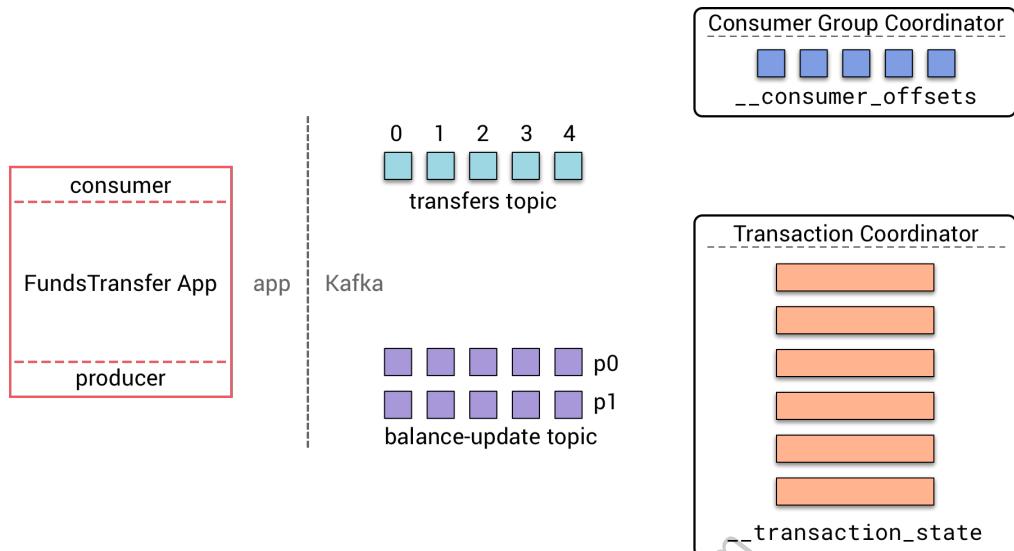
Appendix E: Detailed Transactions Demo

Description

This section presents a more detailed demo of a consume-process-produce application that uses transactions.
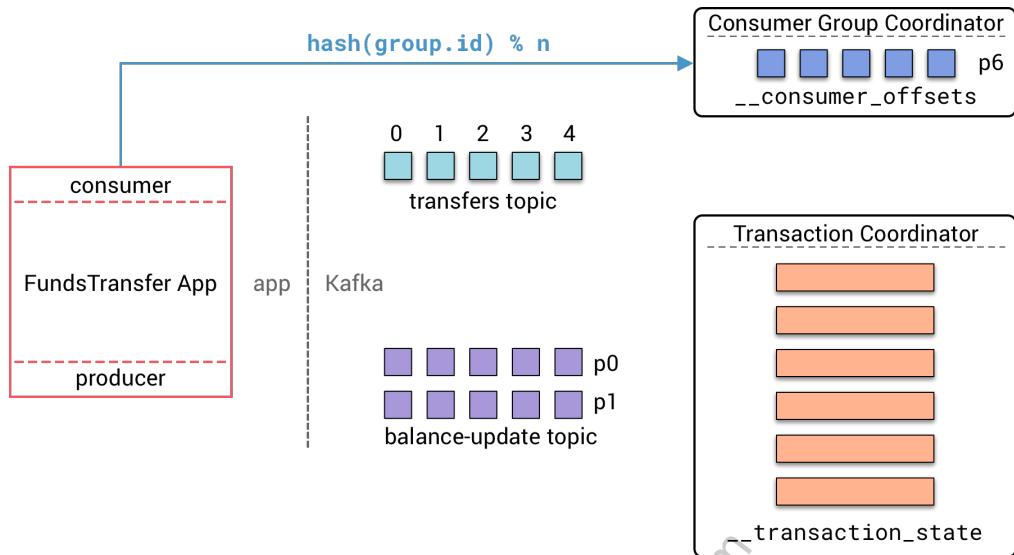
Pictured is a stream processing application called FundsTransfer that follows the consume-process-produce paradigm. The idea is to read a financial transaction from the "transfers" topic and produce balance updates to the "balance-update" topic.
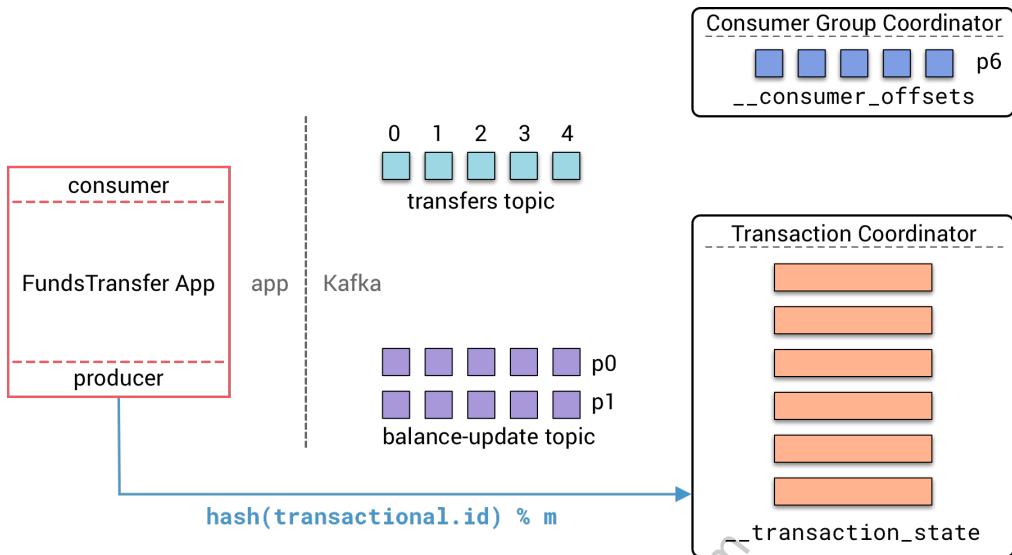
A Transaction Coordinator is a module that is available on any Broker. The Transaction Coordinator is responsible for managing the lifecycle of a transaction in the "Transaction Log"—the internal Kafka Topic `__transaction_state` partitioned by `transactional.id`. The Broker that acts as the Transaction Coordinator is not necessarily a Broker that the Producer is sending messages to. For a given Producer (identified by `transactional.id`), the Transaction Coordinator is the leader of the Partition of the Transaction Log where `transactional.id` resides. Because the Transaction Log is a Kafka Topic, it has durability guarantees.
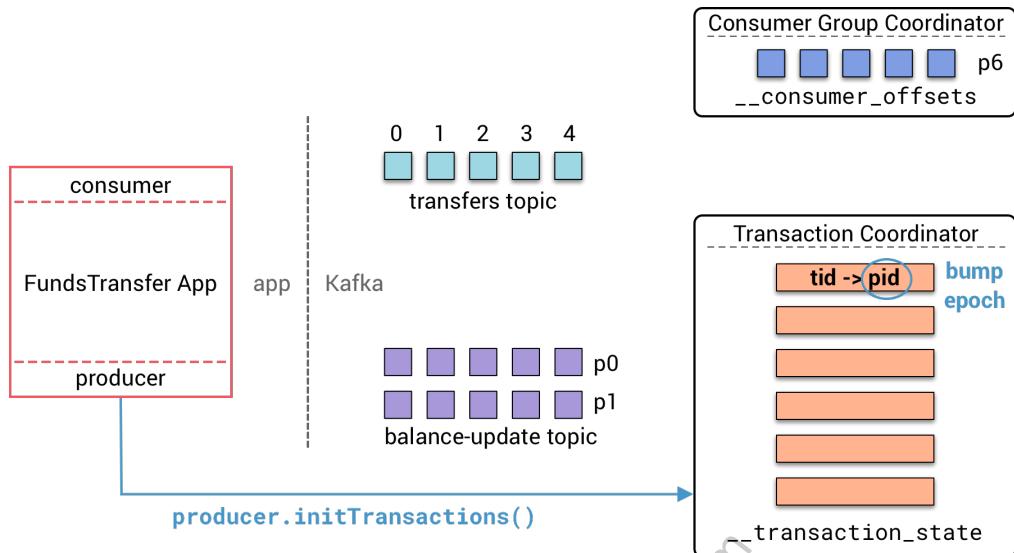
The consumer and producer are initialized before stream processing is started. Here we see the consumer subscribe to the "transfers" topic and identify its Consumer Group Coordinator using `hash(group.id) % n`, where n is the number of partitions of the consumer offsets topic (Default: 50). Here, the p6 indicates that this Consumer Group Coordinator is the broker that holds the lead replica for partition 6 of the consumer offsets topic.

Here we see the producer initiating the transaction. The producer identifies the Transaction Coordinator using `hash(transactional.id) % m`, where `m` is the number of partitions of the `__transaction_state` topic (Default: 50).
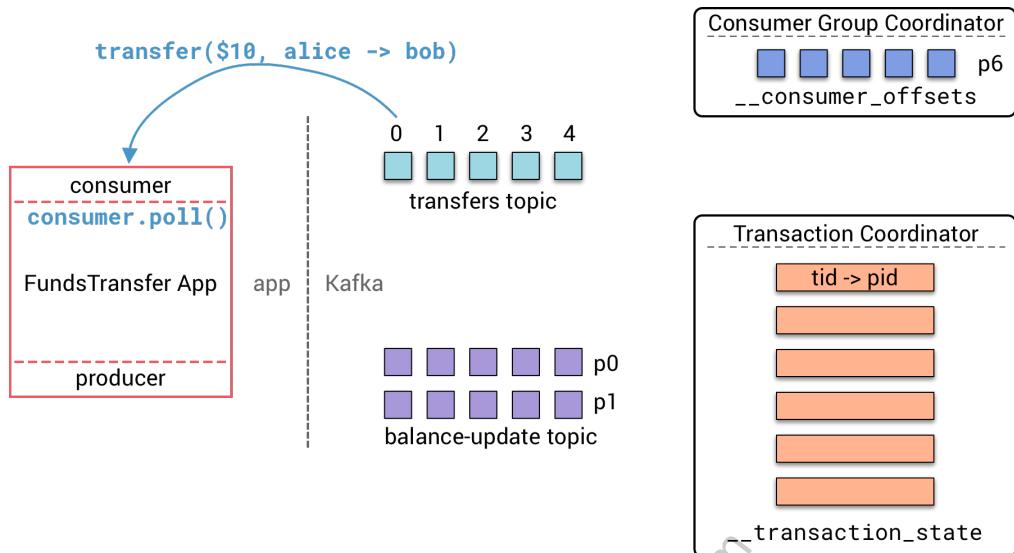
During the initiation, the Producer registers itself to the Transaction Coordinator with its `transactional.id`. The Transaction Coordinator records a mapping `{ Transactional ID : Producer ID }`. The Transaction Coordinator also increments an **epoch** associated with the `transactional.id`.
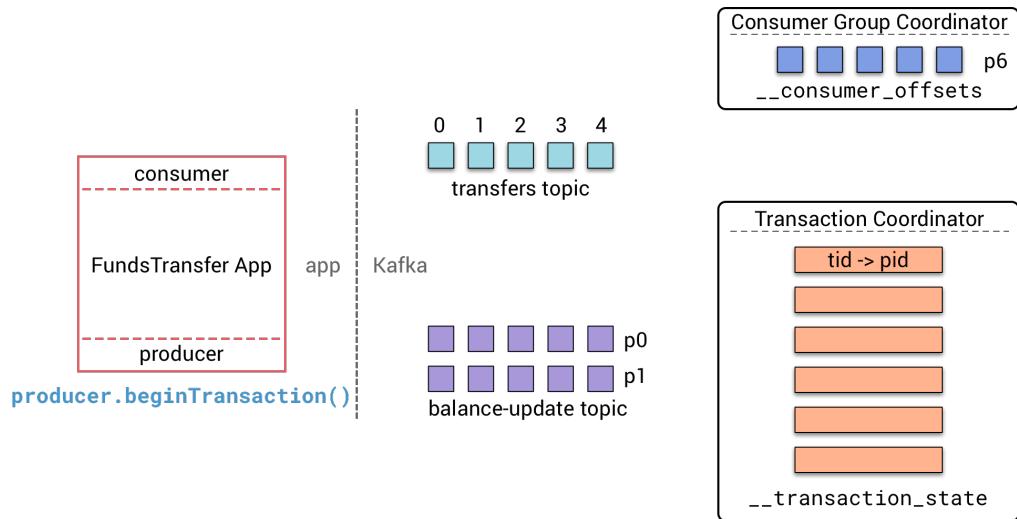
The epoch is an internal piece of metadata stored for every `transactional.id`. Once the epoch is bumped, any producers with same `transactional.id` and an older epoch are considered zombies and are fenced off and future transactional writes from those producers are rejected. This enables reliability semantics which span multiple producer sessions since it allows the client to guarantee that transactions using the same TransactionalId have been completed prior to starting any new transactions.

transfer($10, alice -> bob)

Consumer Group Coordinator

p6

__consumer_offsets

0 1 2 3 4

transfers topic

consumer

consumer.poll()

FundsTransfer App

app | Kafka

producer

Transaction Coordinator

tid -> pid

__transaction_state

p0

p1

balance-update topic
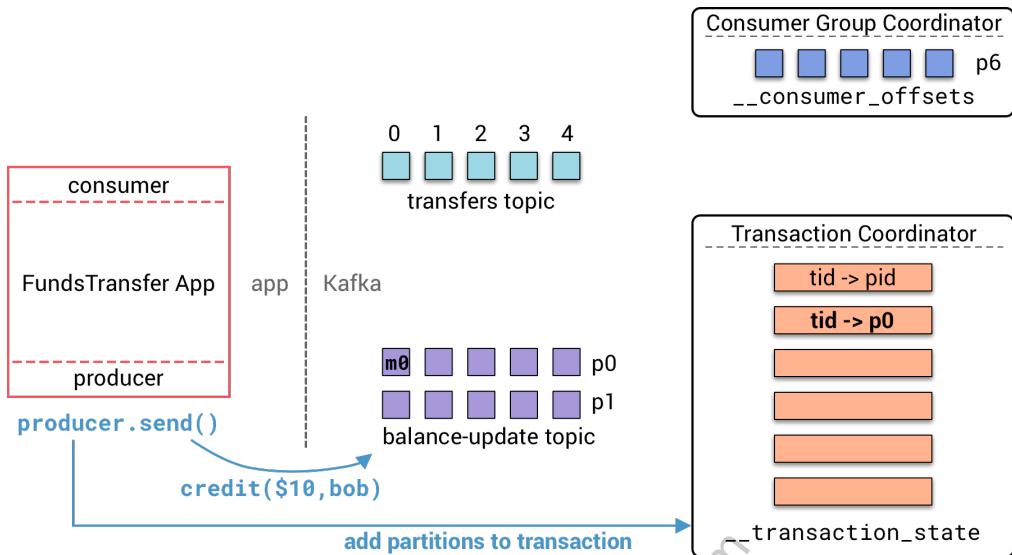
The Consumer polls for messages from the input Topic. Here, the consumer reads an event that transfers $10 from Alice to Bob. The goal of the FundsTransfer app is to transactionally write events to the "balance-update" topic that credits Bob with $10 and debits $10 from Alice.

The Producer begins the transaction.
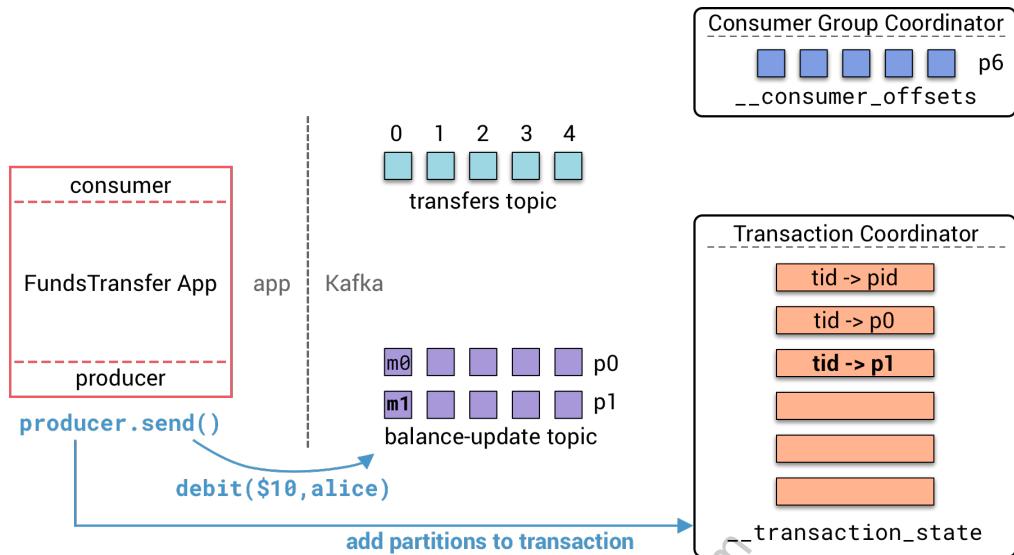
The Producer sends a message to a partition. Here, the message is to credit Bob with $10.
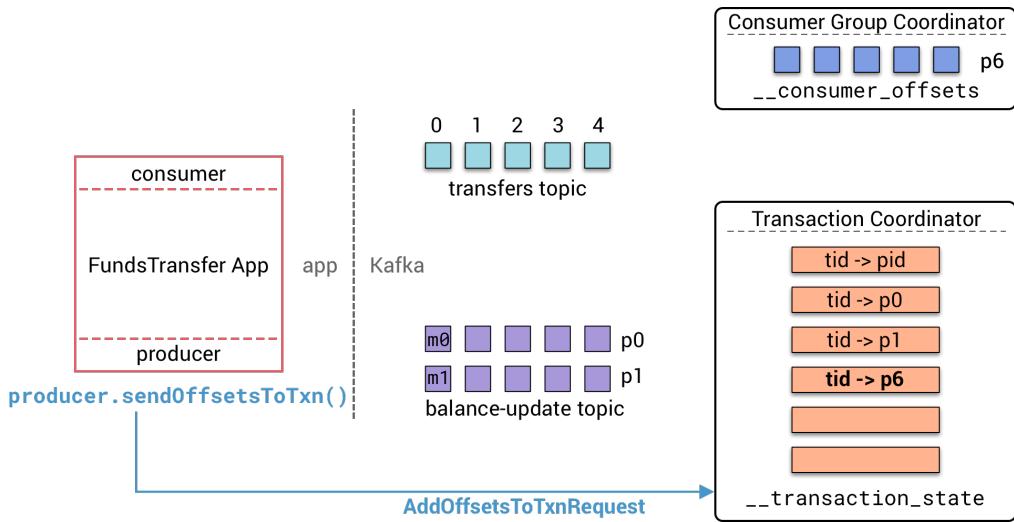
ℹ️	The first time a new TopicPartition is written to as part of a transaction, the producer sends a "Register Partitions" request to the transaction coordinator and this TopicPartition is logged. The transaction coordinator needs this information so that it can write the commit or abort markers to each TopicPartition. If this is the first partition added to the transaction, the coordinator will also start the transaction timer.

The Producer sends a message to a second partition. Here, the message is to debit $10 from Alice. This message happens to land on a different partition from the previous message, so this partition is also added to the transaction log.
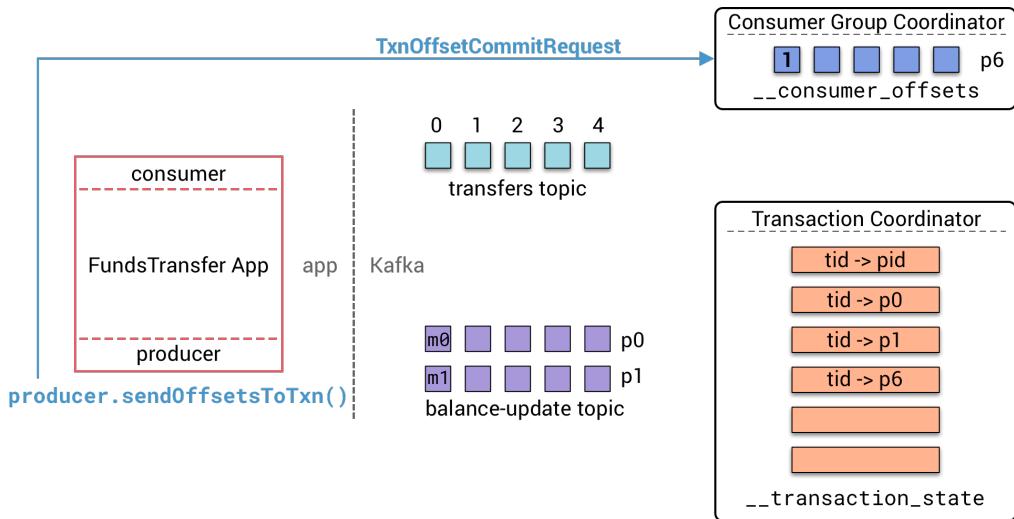
The `sendOffsetsToTxn()` method sends the consumer's offset and consumer group information to the transaction coordinator via an `AddOffsetCommitsToTxnRequest`. This makes the consumer's offset become a part of the transaction. If the transaction fails, the consumer's offset doesn't move forward and the transaction can start over.

Of course, the consumer may be subscribed to multiple partitions across multiple topics, in which case all relevant consumer offsets are included in the transaction state log. In this simple example, there is only one partition's offset to track.

ℹ	To take advantage of the `sendOffsetsToTxn()` method, the consumer should have `enable.auto.commit=false` and should also not commit offsets manually. See the Java API documentation
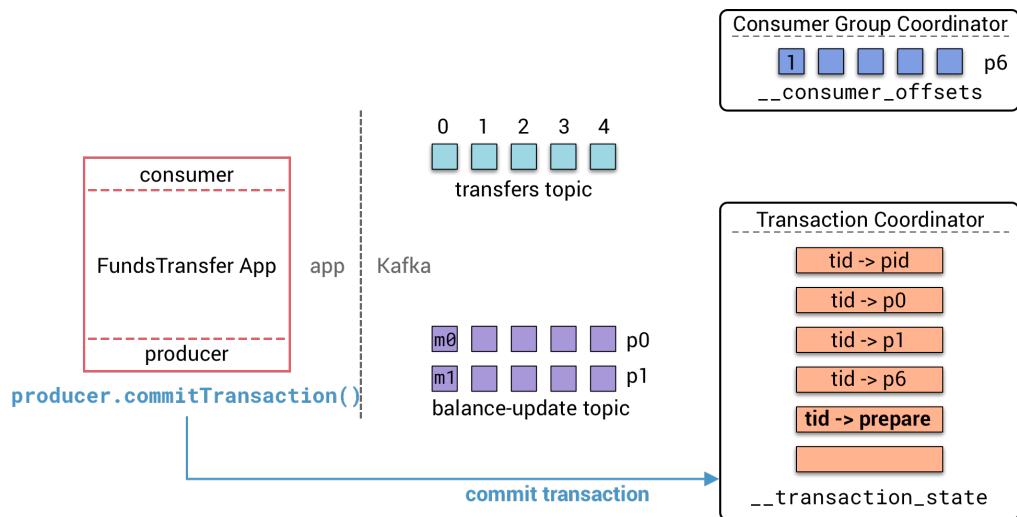
Transactions - Commit Consumer Offset (10/14)

Also as part of `sendOffsetsToTxn()`, the producer will send a TxnOffsetCommitRequest to the consumer coordinator to persist the offsets in the `__consumer_offsets` topic.
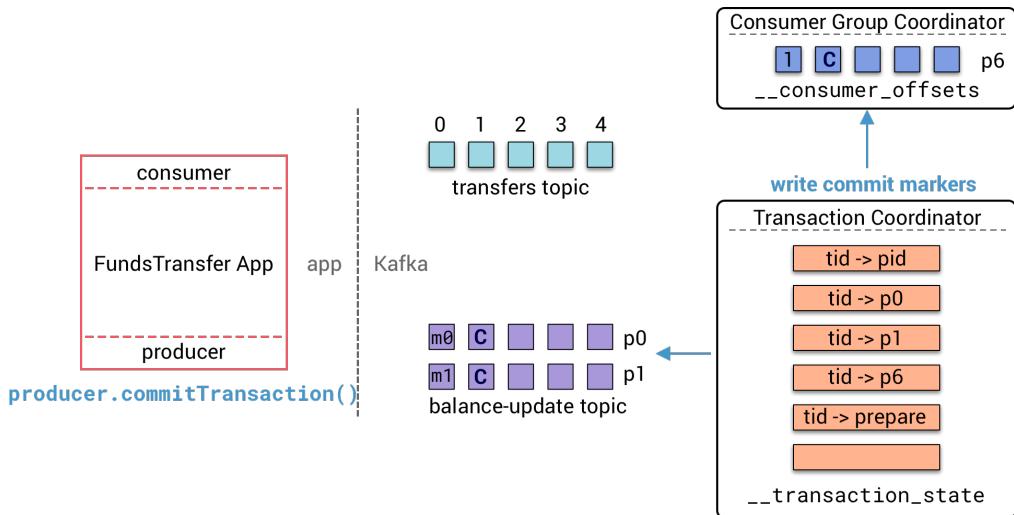
This guarantees the offsets and the output records will be committed as an atomic unit.
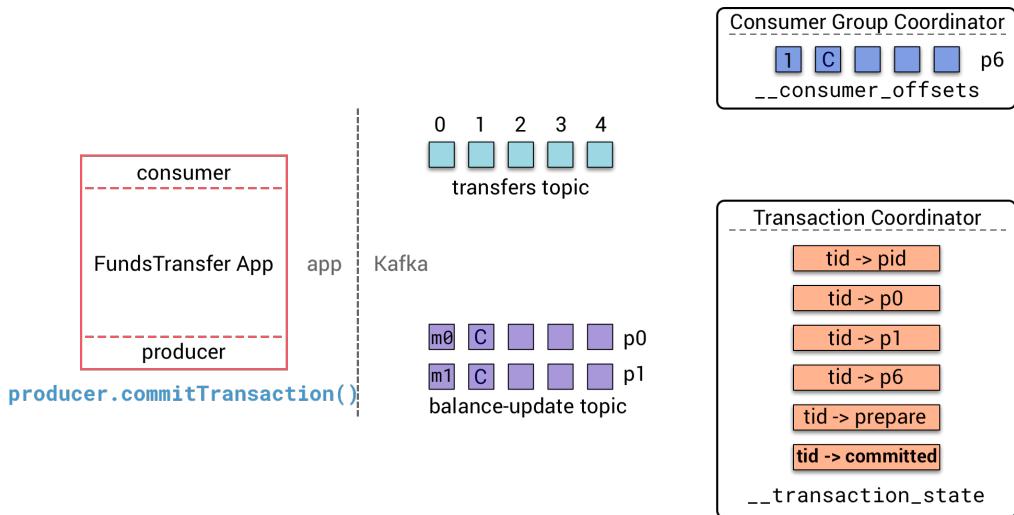
Producer commits the transaction. The transaction coordinator marks the transaction as in status of "preparing."
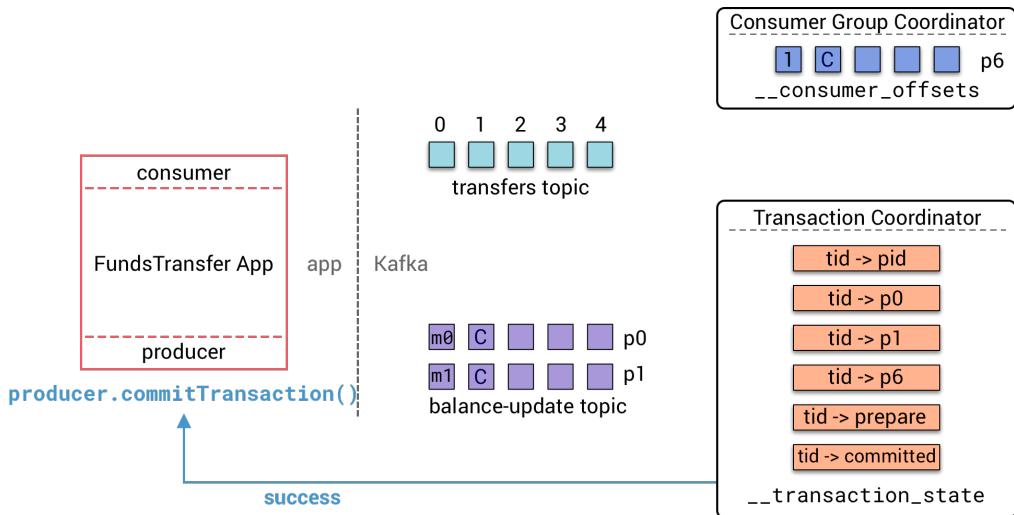
The Transaction Coordinator writes commit markers to the Partitions the Producer writes to as well as to the `__consumer_offsets` Partition. Commit markers are special messages which log the producer id and the result of the transaction (committed or aborted). These messages are internal only and are not exposed by standard consumer operations.

The Transaction Coordinator marks the transaction as committed.

Transactions - Success (14/14)

Consumer Group Coordinator

| 1 | C | | | | p6 |

`__consumer_offsets`

```
  0   1   2   3   4
```
transfers topic

consumer

FundsTransfer App | app | Kafka

producer

producer.commitTransaction()

| m0 | C | | | | p0 |
| m1 | C | | | | p1 |

balance-update topic

success

Transaction Coordinator

tid -> pid
tid -> p0
tid -> p1
tid -> p6
tid -> prepare
tid -> committed

`__transaction_state`

As a final step the transaction coordinator sends an acknowledgment to the producer.

Appendix: Confluent Technical Fundamentals of Apache Kafka® Content

CONFLUENT
Global Education

Module Overview

This section contains 5 lessons - the content lessons from the Fundamentals prerequisite:

Lessons of Presentation:

1. Getting Started

2. How are Messages Organized?

3. How Do I Scale and Do More Things With My Data?

4. What's Going On Inside Kafka?

5. Recapping and Going Further

CONFLUENT
Global Education

Why Kafka?

In a nutshell...

Kafka is good for	Kafka is not meant for
• data in motion	• batch processing
• real-time processing	• archiving data

One Example Use Case: Ordering Food

Suppose we are building a system for a restaurant chain:
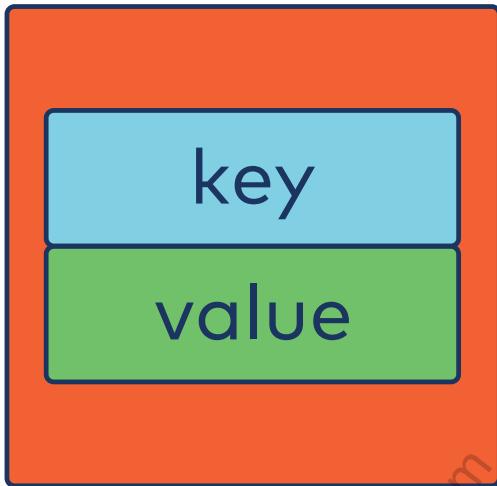
- customers order food via an app - mobile or kiosk

- staff receive orders to fulfill in real-time

- management tracks inventory based on orders

We will build up some of the fundamental details of Kafka and use this example.

Messages

The atomic unit of Kafka is a **message** or **record** or **event**

This treatment is deliberately simple: just key and value. You can specify other components in specifying a message; there's more in the Developer and Administrator training.

It is possible to have messages without keys; this course will assume all messages have keys specified.

Topics

Messages are organized in logical groups called **topics**.

Example topics:

• **orders**

• menu items

• customers

• restaurants

The topics listed go along with the example use case from a few slides back.

The topic **orders** is bold because we'll focus on that one specifically.

Three Basic Components

Let's start simple - with an **orders** topic in place in Kafka, a producer, and a consumer:

Here, we just show three components at a high level. The next three slides go into each in more detail.

We'll add even more detail in the lessons to come and wrap up with more detailed views of these illustrations.
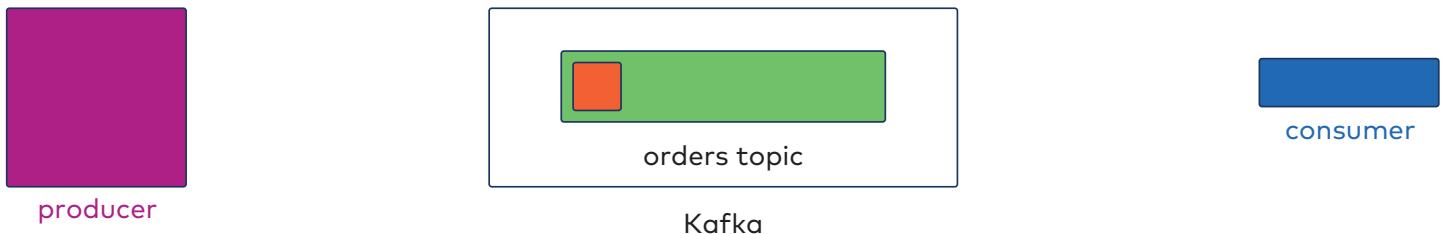
Life Cycle of a Message: Producing

A **producer** prepares messages and publishes them to Kafka.
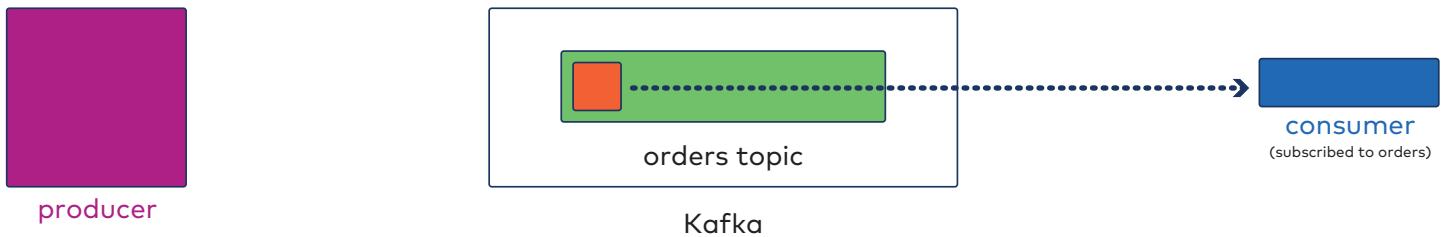
Life Cycle of a Message: Kafka

Produced messages live in Kafka, organized by topic.

producer

orders topic

Kafka

consumer

Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.

producer   Kafka

When we set up consumers, we subscribe to topics to read from.

Note that the message is still shown in the topic in Kafka even though it is shown in the consumer. This is correct. Consuming a message does not remove it from Kafka.
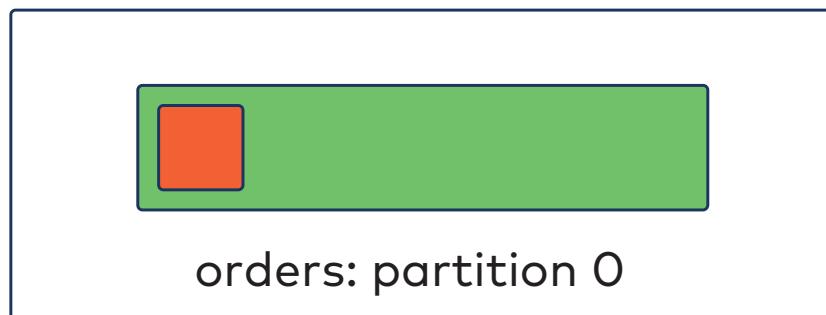
CONFLUENT
Global Education

Topics and Partitions

Topics are broken down into **partitions**

Simplest case: Topic with one partition

orders: partition 0
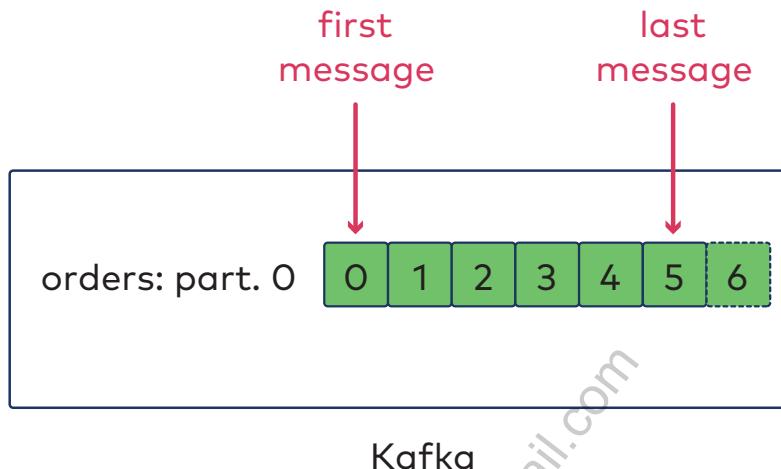
Kafka

The presentation is meant to be only about logical organization at this point, and the messages in a partition are a (logical) subset of the messages in a topic. A partition is also a *physical* grouping of messages; more on the physical later.

Offsets—in Kafka

- Each message in a partition has an **offset**

- Starting from 0

first message   last message

orders: part. 0   0 | 1 | 2 | 3 | 4 | 5 | 6

Kafka
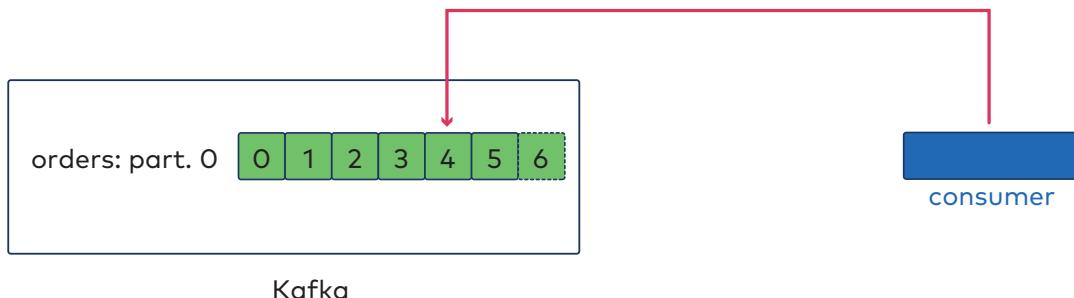
A different way of thinking of a message's offset is "how many messages were written to this partition before this message?"

Here we point out the offset of the last message. The offset where the next message will be written is illustrated too; the hands-on activity will show this offset being reported a Kafka command line tool.

Offsets—Consumer Offsets

- Consumers track where they will read next via a **consumer offset**

Kafka

> ℹ️ In this picture, the consumer has last read the message at offset 3.

For now, let's say our consumer is assigned to the **orders** topic and it has one partition, and in turn our consumer must be assigned to that one partition. A consumer offset is:

- per consumer

- per partition

But there is only one of each so far, so there is one offset. Multiple consumers and multiple consumers will come in the next lesson.

Check Your Knowledge!

Try a [quick quiz on Lessons 1 and 2](#).

3: How Do I Scale and Do More Things With My Data?

CONFLUENT
Global Education

Scaling Up...

So far, we have seen...

- one partition

- one consumer

In practice...

- multiple consumers in a consumer group

- multiple consumer groups

- multiple partitions in a topic
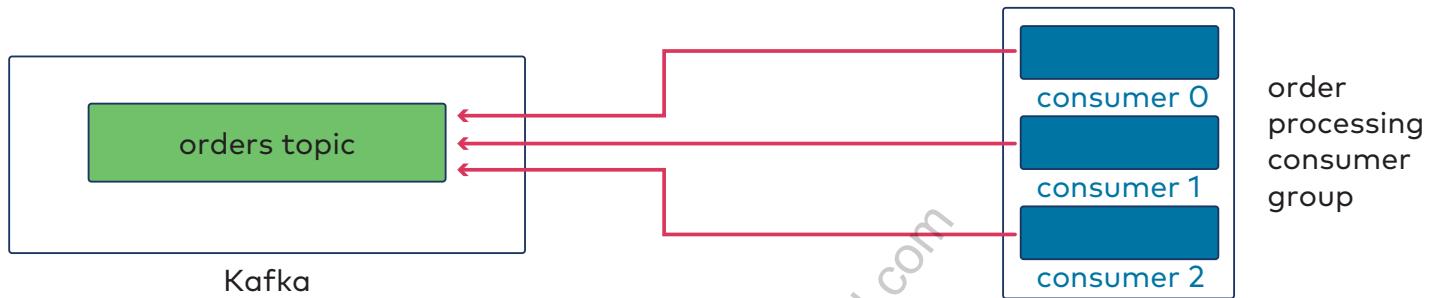
In this lesson, we'll expand on the last and look into having multiple consumers or partitions.

Consumer Groups

Consumers exist in **consumer groups**

Consumers in a group:
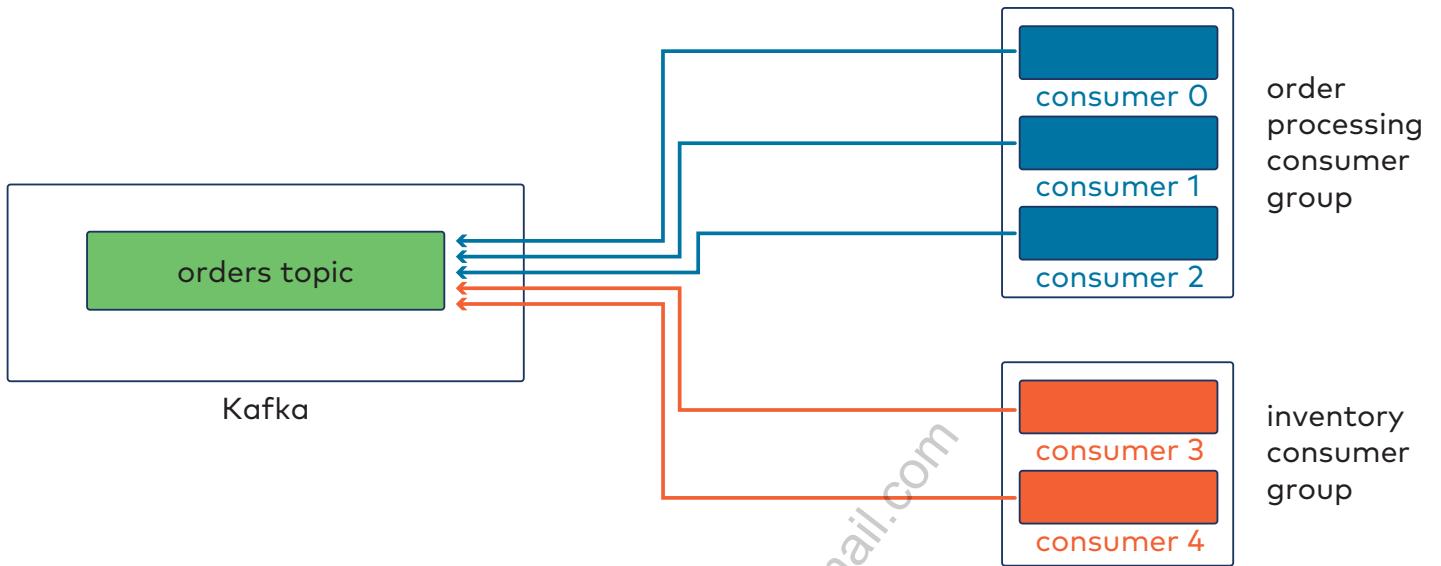
* **same** application

* **different** data

Technically, it is recommended but not required consumers be in groups. We'll always model consumers in groups.

Multiple Consumption

Could have multiple groups using the same data...
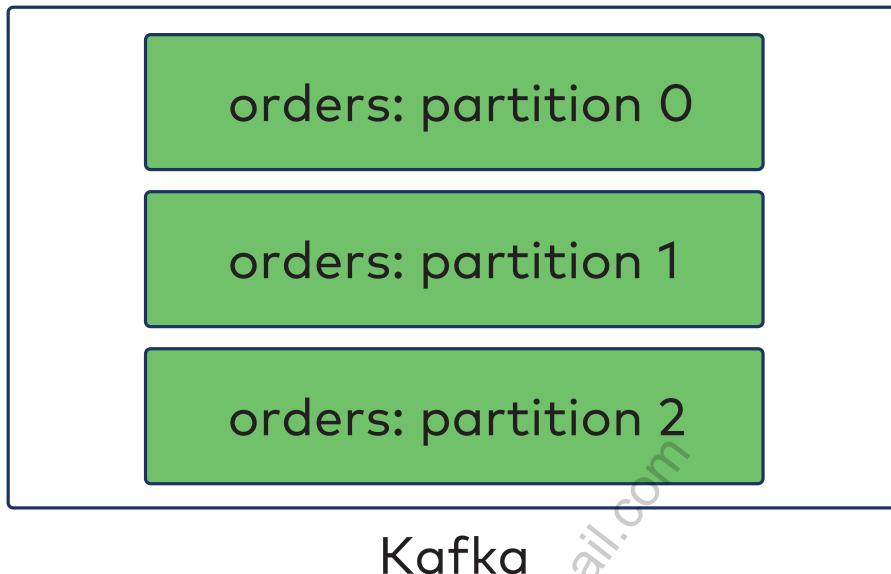
When a consumer reads a message, that does not remove the message. Different consumers could read the same message.

All of the consumers in a group are doing the same task. We see here our order processing group as before, but we add a second group of consumers: they go through orders (with less urgency) to tally what has been ordered and help alert a restaurant which ingredients or supplies should be restocked.

Multiple Partitions

In practice, we want topics to have multiple partitions.

| orders: partition 0 |
| orders: partition 1 |
| orders: partition 2 |

Kafka
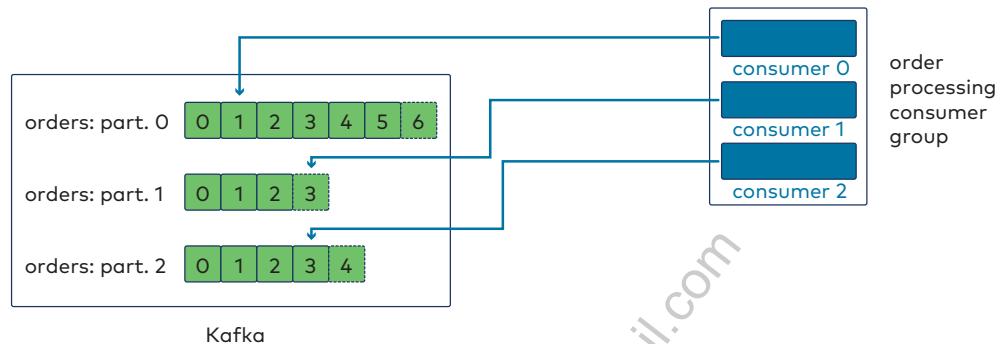
Before, we said a partition was a subset of the messages in a topic, but we only saw one partition. Here is where we first see a topic broken up into multiple partitions.

Consuming from Multiple Partitions

Now our consumers can consume in parallel:

- Consumers subscribed to a topic are assigned partitions

- Group covers all partitions

- Each consumer has an offset for each partition
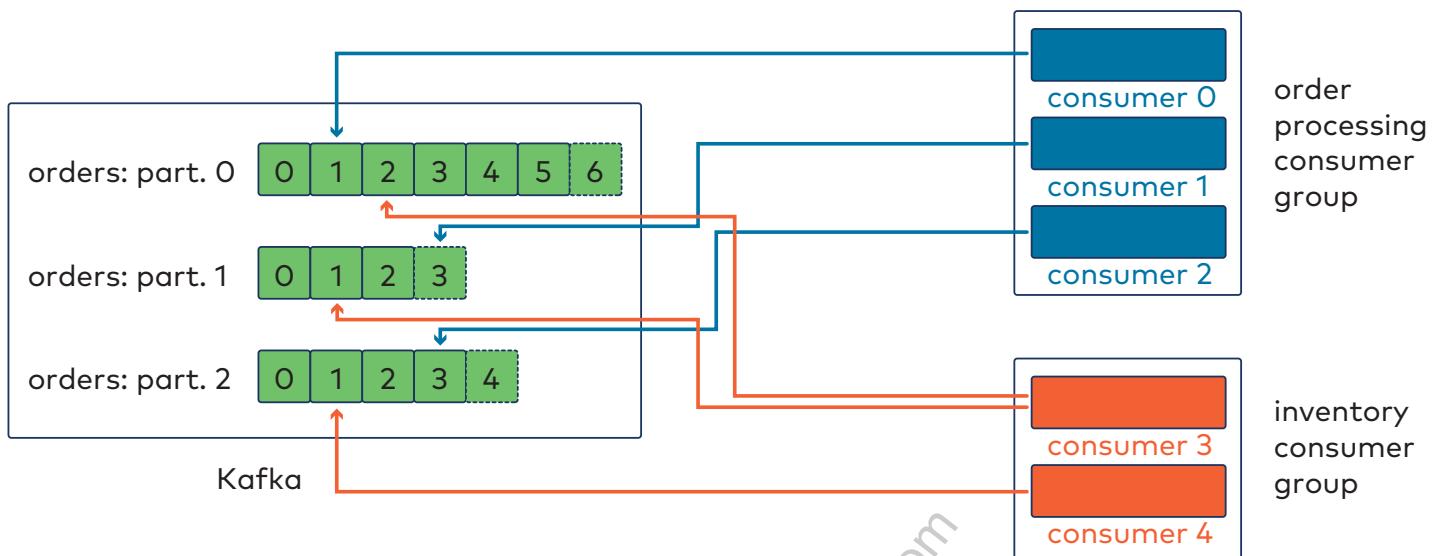
Now that we have multiple partitions, different consumers could read from different partitions all at the same time.

Kafka handles the assignment of consumers to partitions; all users need to do is subscribe to topics. There are configuration settings and details; more on this in the Administrator and Developer classes.

Each consumer could be reading from each partition at a different location, so consumer offsets are per consumer and per partition. The picture illustrates three consumers all at different locations in their assigned partitions.

Expanding the Last Picture

Here we bring back the second consumer group from a few slides ago. We can see these consumers assigned to partitions. The group as a whole needs to read all messages in all partitions and we have two consumers but three partitions, so one consumer has to handle two partitions this time.

Note, also, that different consumers reading from the same partition don't have to be at the same place, as shown in this picture.

How Do Messages Get Partitioned?

- **Producers** decide which messages go to which partition

- Partitions are indexed from `0` to `numberOfPartitions - 1`

- Default partitioner: `partitionIndex = hash(key) % numberOfPartitions`

It is the producer that decided which messages go to which partition of a topic. The slide gives the formula used by default - and which is used by the console producer you'll use in the hands-on exercise.

Developers may specify a partitioner in producer code.

Scaling is Easy!

Say you want to

- Increase the number of consumers in a group during a busy season

- Decrease the number of consumers in a group when things are slow

- Increase the number of partitions for a topic

When you do, Kafka *automatically* redistributes the assignments of consumers to partitions!

ℹ	More on how all of this works in both our Developer and Administrator training!

You can change some factors about your Kafka deployment after you've started using it. When you change what's on the slide, it will cause Kafka to change which consumers work with which partitions in what is called a *rebalance*. The details of how this work are beyond the scope of this course but get significant attention in the Developer and Administrator classes.

CONFLUENT
Global Education
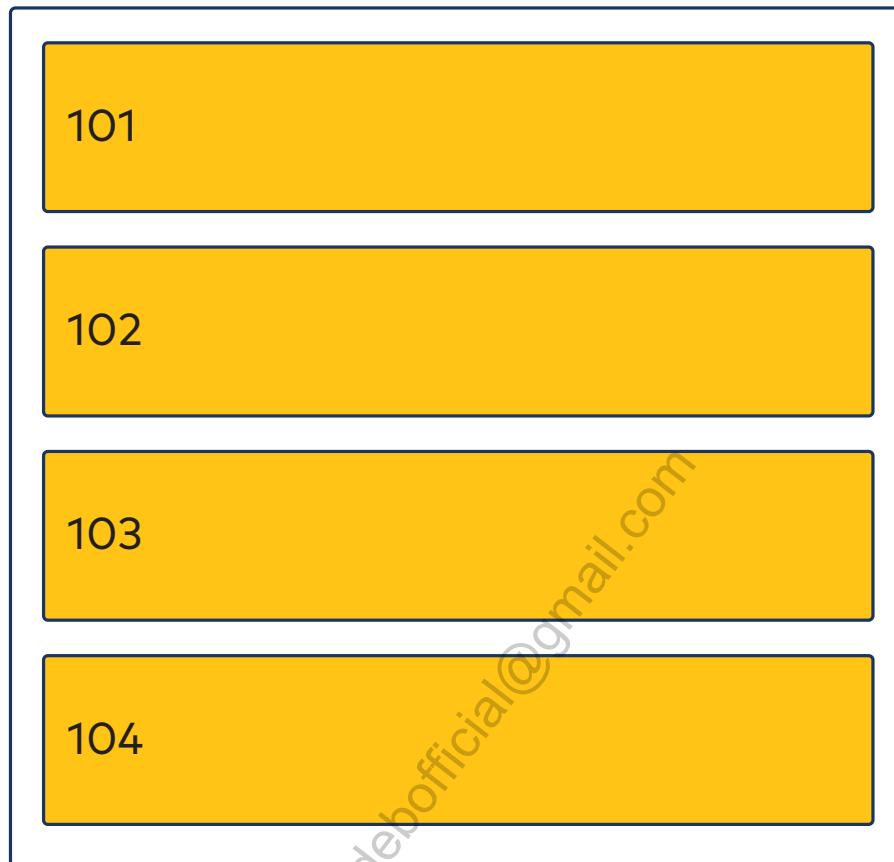
Going Deeper…

Now let's learn about some more details about a Kafka cluster, especially *physical* things...

Brokers

A Kafka cluster consists of multiple **brokers**

Kafka
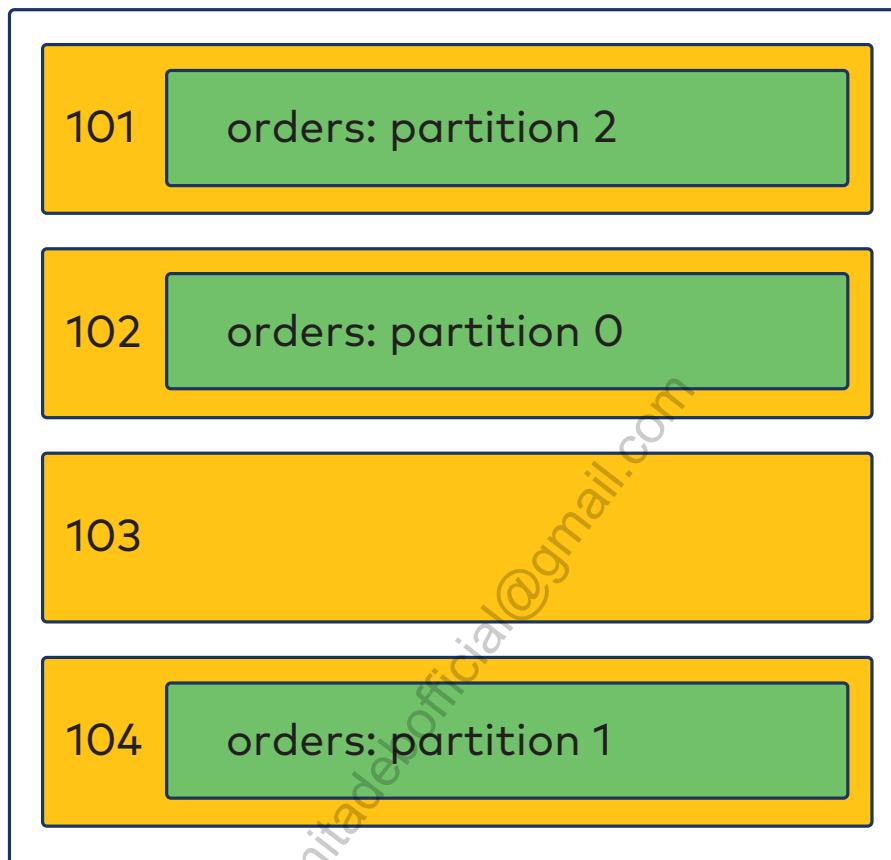
Brokers could be physical servers, but could also be VMs, Docker containers, etc.

The orange boxes represent brokers and the numbers are broker IDs.

Partitions & Brokers

Partitions are really **physical** groupings of the messages in topics.
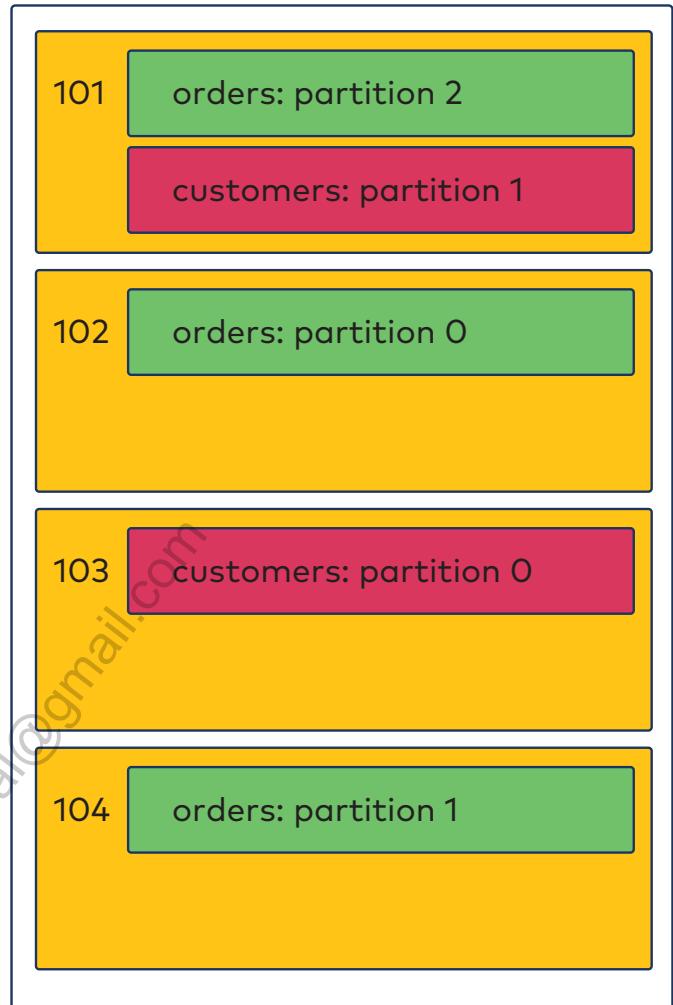
Partitions are stored on brokers.

101	orders: partition 2
102	orders: partition 0
103	
104	orders: partition 1

Kafka

Partitions & Brokers (2)

The number of partitions is a topic setting.

Kafka decides how partitions get distributed across brokers.

101	orders: partition 2
	customers: partition 1
102	orders: partition 0
103	customers: partition 0
104	orders: partition 1
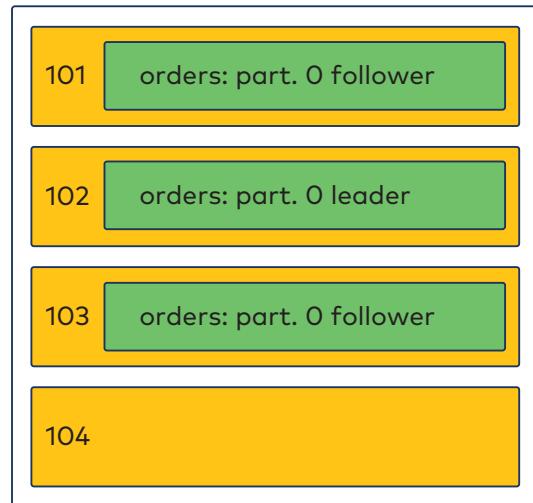
Kafka

Here we see a second topic with a different number of partitions added to the illustration.

What if a Broker Goes Down?

- Want *high availability* of data in partitions

- Achieved via **replication**

- Writes are reads go to **leader** replica

- **Follower** replicas keep backup copies of the leader

- If leader dies, a follower becomes the leader

101	orders: part. 0 follower
102	orders: part. 0 leader
103	orders: part. 0 follower
104	

Kafka

Note that this is a very simple treatment and replication is covered in much more detail in the Developer and Administrator courses.

Serialization and Deserialization

- Kafka stores messages as byte arrays

- Producers must **serialize** messages

- Consumers must **deserialize** messages

In developing custom producers and consumers, developers must specify the serializers and deserializers. There are also tools like Avro and Protobuf that can be used for complex data types.

Immutable Messages

- Messages are **immutable**

- Once written, we cannot change anything about them

...But We Don't Keep Messages Forever...

Control which messages stay in Kafka via a **retention policy**:

Policy	Deletion	Compaction
Idea	Remove messages older than a certain age (default 7 days)	Keep only the latest value for each key
Before	offset: 0 1 2 3 4 / key: a b b a a / age in days: 12 10 6 5 2	offset: 0 1 2 3 4 / key: a b b a a / age in days: 12 10 6 5 2
After	offset: 2 3 4 / age in days: 6 5 2	offset: 2 4 / key: b a

> ℹ️ Partitions are divided into segments, which affect both retention policies.

Note that this is a very simple treatment and these policies are covered in much more detail in the Developer and Administrator courses. In particular, the impact of segments matters. Deletion is per segment - but the details of segments comes in the other courses.

Maybe, for a deletion use case, we might keep orders around for up to a week to track trends on what people ordered in the last week.

Maybe, for a compaction use case, we might have some sort of greeting for a returning customer like "last time, you ordered... would you like to order this again?"

But, above all, these policies are about smart use of storage.

Check Your Knowledge!

Try a [quick quiz on Lessons 3 and 4](.).
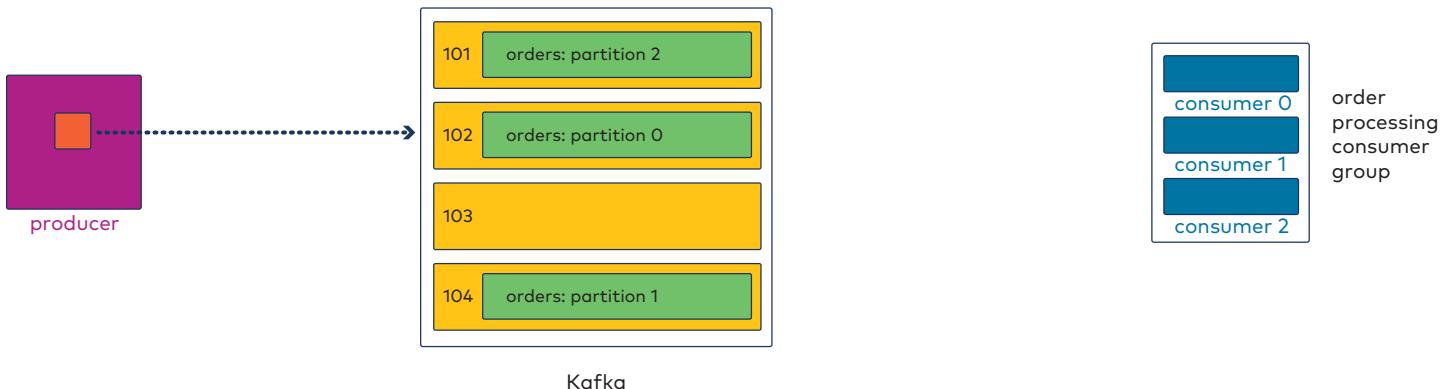
CONFLUENT
Global Education

Life Cycle of a Message: Producing


Kafka

- Producers serialize and partition messages

- Producers send messages

  ◦ ...in batches - can be configured for throughput and latency desires

---
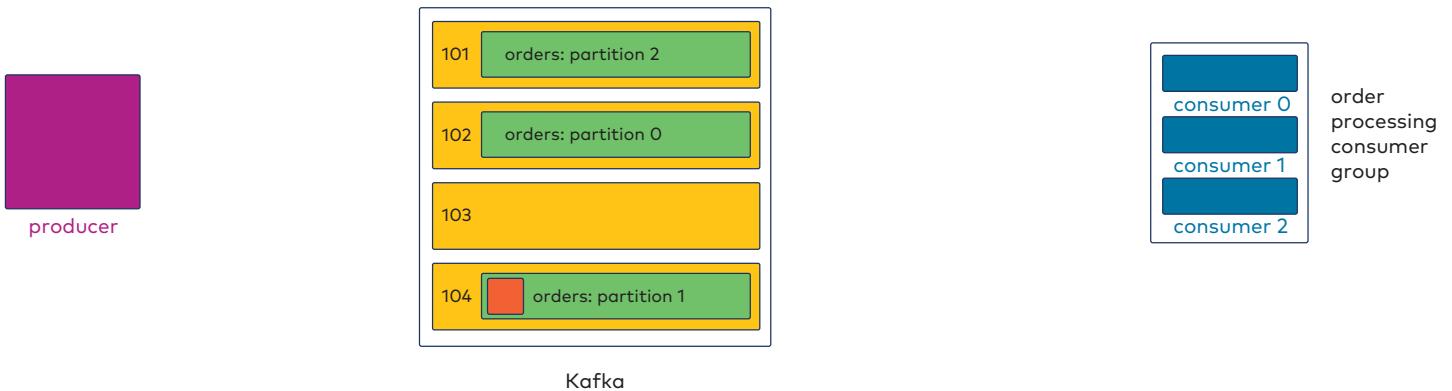
Let's summarize what we've learned about the life cycle of a message... in the context of our running example. We work with a setup similar to what we had in the first lesson, with some of the details from later lessons added.

We start here with a message on a producer being sent...

Life Cycle of a Message: Kafka

Produced messages live in Kafka, organized by topic.
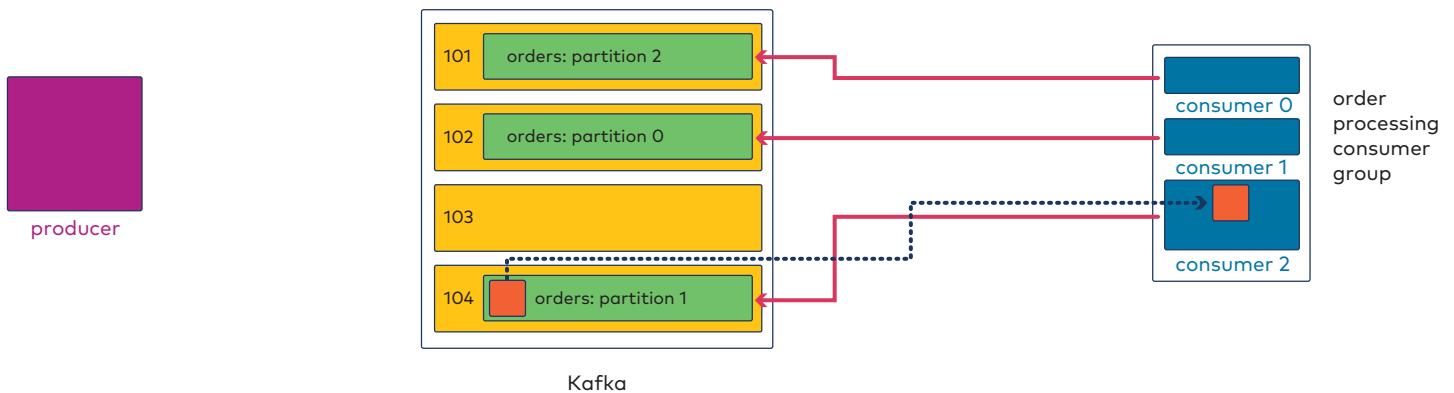
Kafka

- Kafka consists of brokers

- Brokers contain partitions, which contain messages

- Brokers handle retention and replication

Now our message makes it to Kafka, specifically to a partition of a topic, and that partition lives on a particular broker.

Life Cycle of a Message: Consumption

Consumers subscribe to topics in Kafka and poll for new messages.

Kafka

- Consumers operate in groups

- Consumers subscribe to topics, are assigned partitions of those topics

- Consumers poll for messages in partitions at consumer offsets

  ◦ ...and fetch in batches - can be configured for throughput and latency desires

---

Now we see a group of consumers subscribed to the orders topic and see a message being read by one of those consumers.

A Step Beyond Fundamentals: Other Components

We've addressed some aspects of Core Kafka in this course. Some other topics you may want to learn about include:

- **Kafka Connect** - a tool that helps you copy data to Kafka from other systems and vice-versa

- **Kafka Streams** - a layer on top of the Producer and Consumer APIs that allows for stream processing

- **Confluent ksqlDB** - a tool for stream processing using a more-accessible SQL-like syntax, among other things

- **Confluent Schema Registry** - a tool for managing schemas, guiding schema evolution, and enforcing data integrity

You can learn more about these topics in our Confluent Developer Skills for Building Apache Kafka® and Apache Kafka® Administration by Confluent courses.

What Does Confluent Platform Add to Kafka?

CONFLUENT PLATFORM

SECURITY & RESILIENCY
RBAC | Audit Logs | Schema Validation | Multi-Region Clusters | Replicator | Cluster Linking

PERFORMANCE & SCALABILITY
Tiered Storage | Self-Balancing Clusters | K8s Operator

MANAGEMENT & MONITORING
Control Center | Proactive Support

DEVELOPMENT & CONNECTIVITY
Connectors | Non-Java Clients | REST Proxy | Schema Registry | ksqlDB

APACHE KAFKA®
Core | Connect API | Streams API
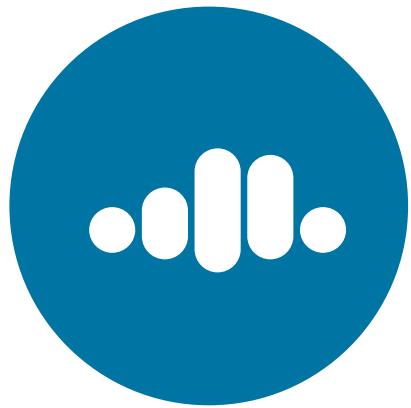
Everything we've discussed in this course is part of core Apache Kafka. Confluent Platform adds additional features beyond the core. The top two boxes in the medium shade of blue are paid features; the next two in the teal shade of blue are free features.

Confluent Cloud

- Can deploy CP as self-managed software but...

- Confluent Cloud = **fully-managed** deployment of CP

  ◦ Many administrative tasks done for you

- Confluent Cloud available on

  ◦ AWS

  ◦ Google Cloud Platform

  ◦ Microsoft Azure

Everything we've learned about in this course is independent of platform, but CP may be deployed in a self-managed way or via Confluent Cloud, where our team handles many management tasks for you.

Your Next Steps

1. Complete interactive lab on seeing console producers and consumers in action.
   → Short Confluent Cloud version
   → Gitpod version: More involved version using Gitpod

2. Work though other Critical Thinking Challenge Exercises.
   → On the web
   → Solutions on the web too!

3. Enroll in and complete one of these courses, as suits your role:

   ◦ Apache Kafka® Administration by Confluent

   ◦ Confluent Developer Skills for Building Apache Kafka®

Thank You

Thank you for attending the course!