Building Real-Time Data Pipelines with Apache Kafka

Introduction

Chapter 1

Introduction

- **Introduction**

About You

- **Raise your hand if...**

Agenda

- **During this tutorial, we will:**
  - Make sure we're all familiar with Apache Kafka
  - Investigate Kafka Connect for data ingest and export
  - Investigate Kafka Streams, the Kafka API for building stream processing applications
  - Combine the two to create a complete streaming data processing pipeline
- **As we go, we have Hands-On Exercises so you can try this out yourself**

The Motivation for Apache Kafka

Chapter 2

Course Contents

The Motivation for Apache Kafka

- **In this chapter you will learn:**
  - Some of the problems encountered when multiple complex systems must be integrated
  - How processing stream data is preferable to batch processing
  - The key features provided by Apache Kafka

The Motivation for Apache Kafka

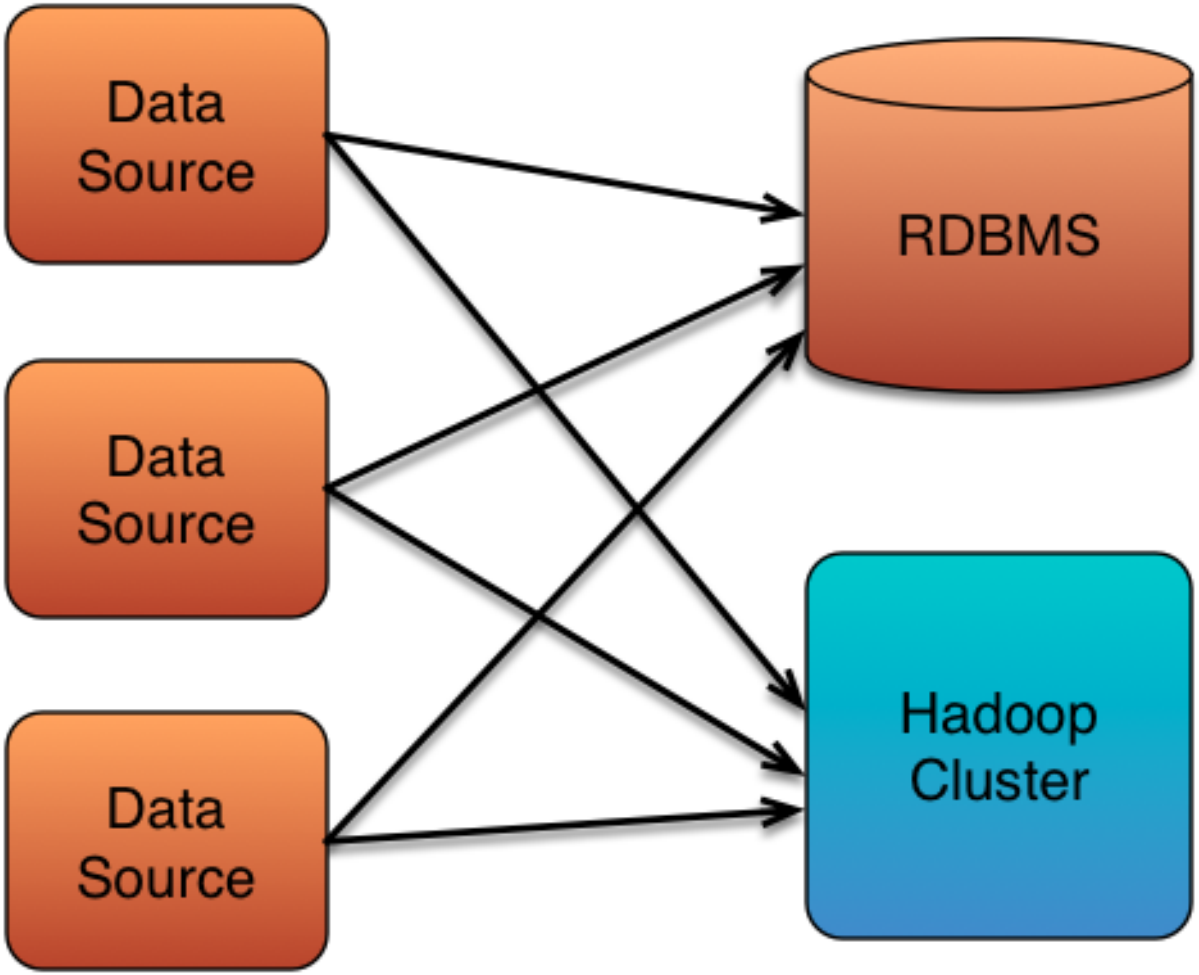
- **Systems Complexity**
- *Real-Time Processing is Becoming Prevalent*
- *Kafka: A Stream Data Platform*

Simple Data Pipelines

- **Data pipelines typically start out simply**
  - A single place where all data resides
  - A single ETL (Extract, Transform, Load) process to move data to that location
- **Data pipelines inevitably grow over time**
  - New systems are added
  - Each new system requires its own ETL procedures
- **Systems and ETL become increasingly hard to manage**
  - Codebase grows
  - Data storage formats diverge

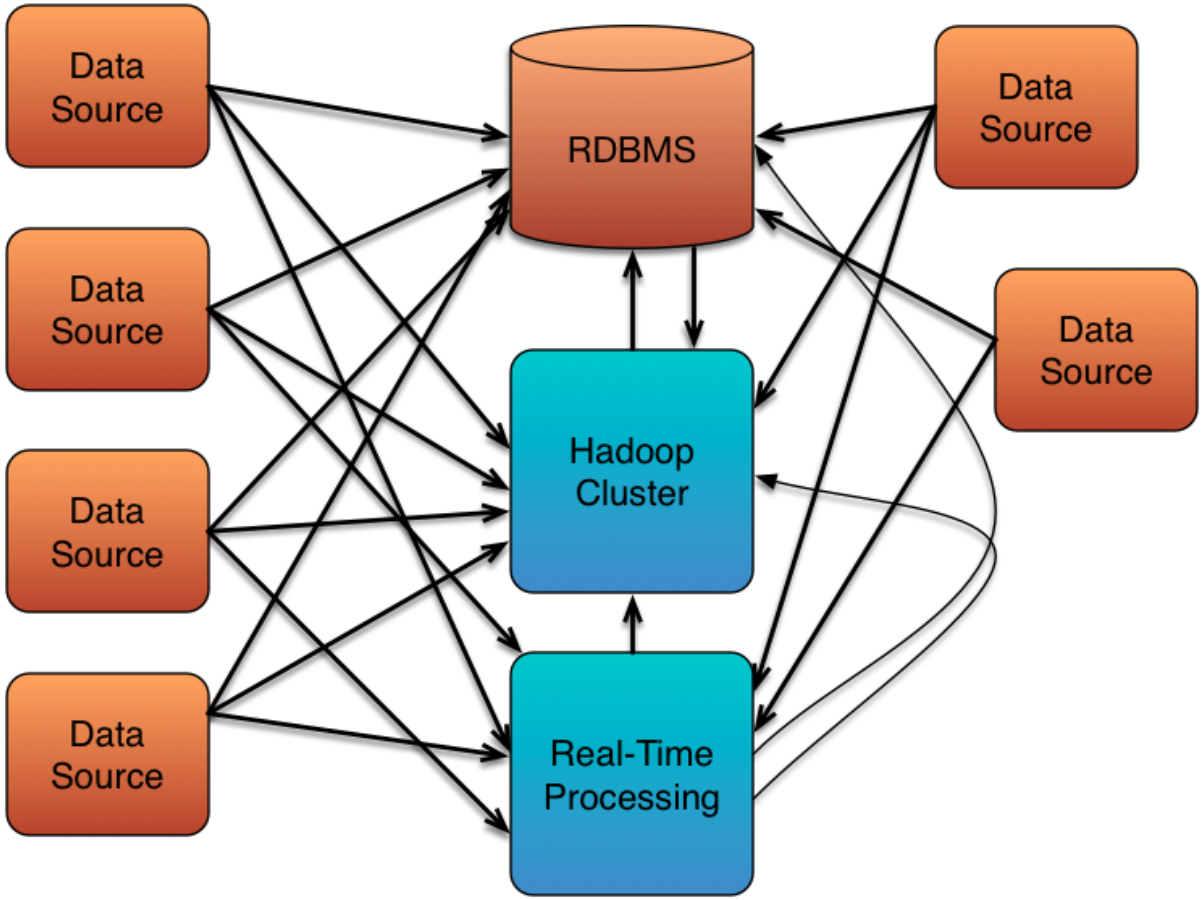
Small Numbers of Systems are Easy to Integrate

It is (relatively) easy to connect just a few systems together



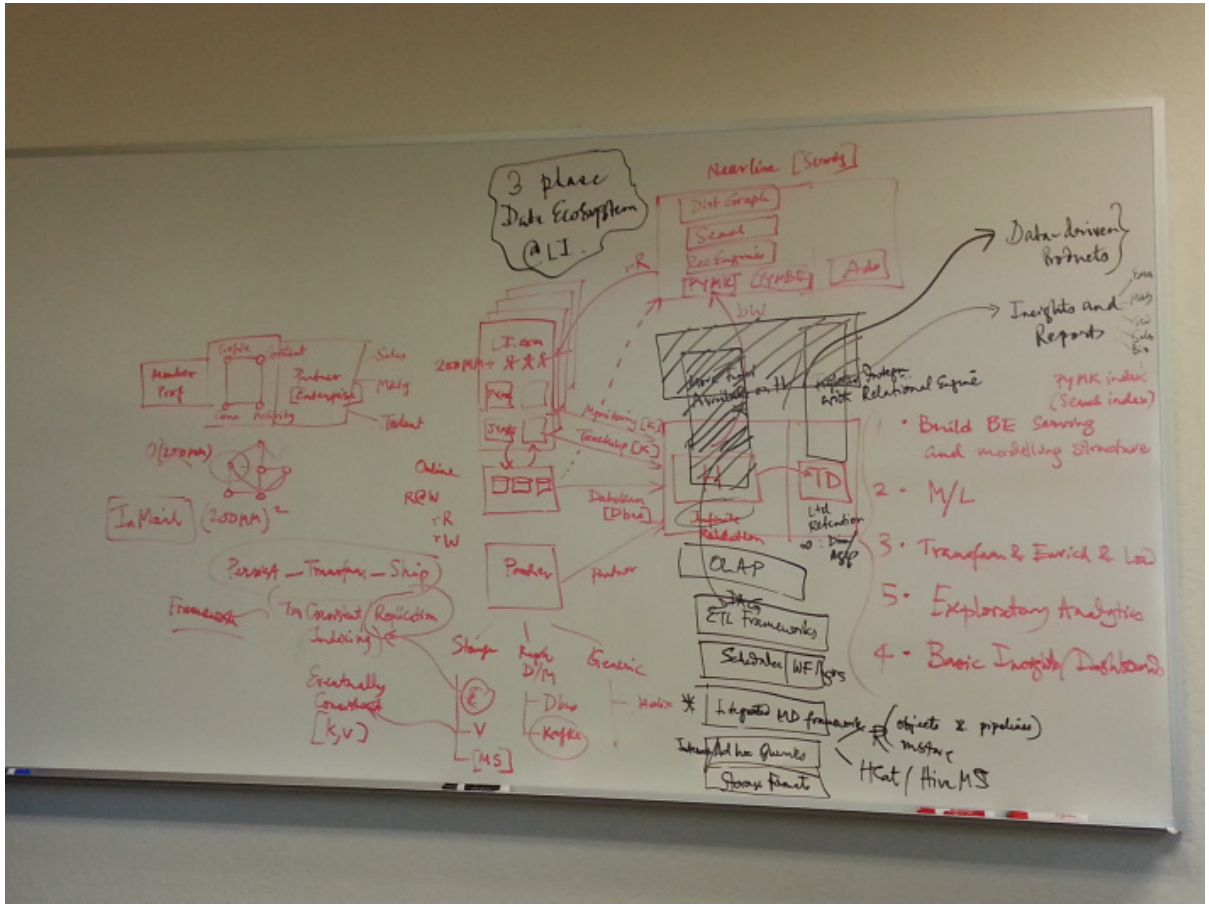
More Systems Rapidly Introduce Complexity (1)

As we add more systems, complexity increases dramatically



More Systems Rapidly Introduce Complexity (2)

...until eventually things become unmanageable



The Motivation for Apache Kafka

- *Systems Complexity*
- **Real-Time Processing is Becoming Prevalent**
- *Kafka: A Stream Data Platform*

Batch Processing: The Traditional Approach

- **Traditionally, almost all data processing was batch-oriented**
  - Daily, weekly, monthly…
- **This is inherently limiting**
  - "I can't start to analyze today's data until the overnight ingest process has run"

Real-Time Processing: Often a Better Approach

- **These days, it is often beneficial to process data as it is being generated**
  - Real-time processing allows real-time decisions
- **Examples:**
  - Fraud detection
  - Recommender systems for e-commerce web sites
  - Log monitoring and fault diagnosis
  - etc.
- **Of course, many legacy systems still rely on batch processing**
  - However, this is changing over time, as more 'stream processing' systems emerge
    - Kafka Streams
    - Apache Spark Streaming
    - Apache Storm
    - Apache Samza
    - etc.

The Motivation for Apache Kafka

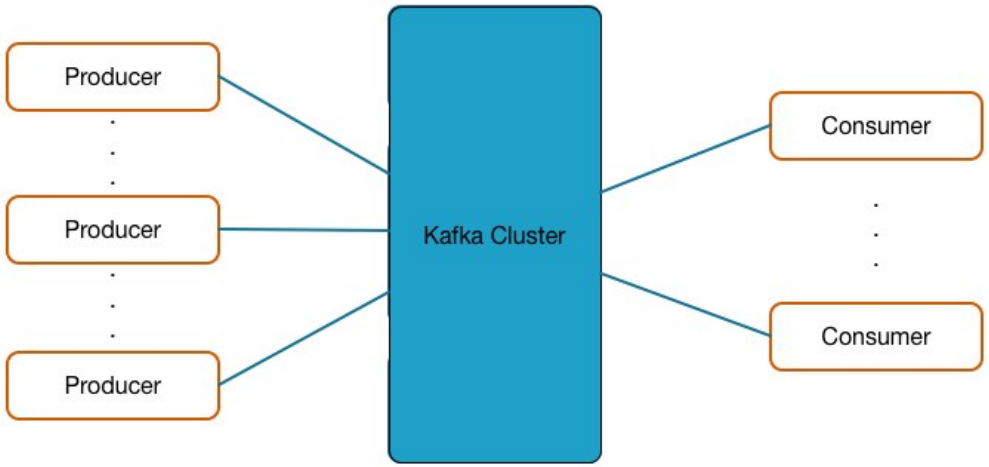
- *Systems Complexity*
- *Real-Time Processing is Becoming Prevalent*
- **Kafka: A Stream Data Platform**

Kafka's Origins

- **Kafka was designed to solve both problems**
  - Simplifying data pipelines
  - Handling streaming data
- **Originally created at LinkedIn in 2010**
  - Designed to support batch and real-time analytics
  - Kafka is now at the core of LinkedIn's architecture
  - Performs extremely well at very large scale
    - Processes over 1.4 *trillion* messages per day
- **An open source, top-level Apache project since 2012**
- **In use at many organizations**
  - Twitter, Netflix, Goldman Sachs, Hotels.com, IBM, Spotify, Uber, Square, Cisco…
- **Confluent was founded by Kafka's original authors to provide commercial support, training, and consulting for Kafka**

A Universal Pipeline for Data

- **Kafka decouples data source and destination systems**
  - Via a *publish/subscribe* architecture
- **All data sources write their data to the Kafka cluster**
- **All systems wishing to use the data read from Kafka**
- **Stream data platform**
  - Data integration: capture streams of events
  - Stream processing: continuous, real-time data processing and transformation



Use Cases

- **Here are some example scenarios where Kafka can be used**
  - Real-time event processing
  - Log aggregation
  - Operational metrics and analytics
  - Stream processing
  - Messaging

Chapter Review

- **Kafka was designed to simplify data pipelines, and to provide a way for systems to process streaming data**
- **Kafka can support batch and real-time analytics**
- **Kafka was started at LinkedIn and is now an open source Apache project with broad deployment**

Kafka Fundamentals

Chapter 3

Course Contents

Kafka Fundamentals

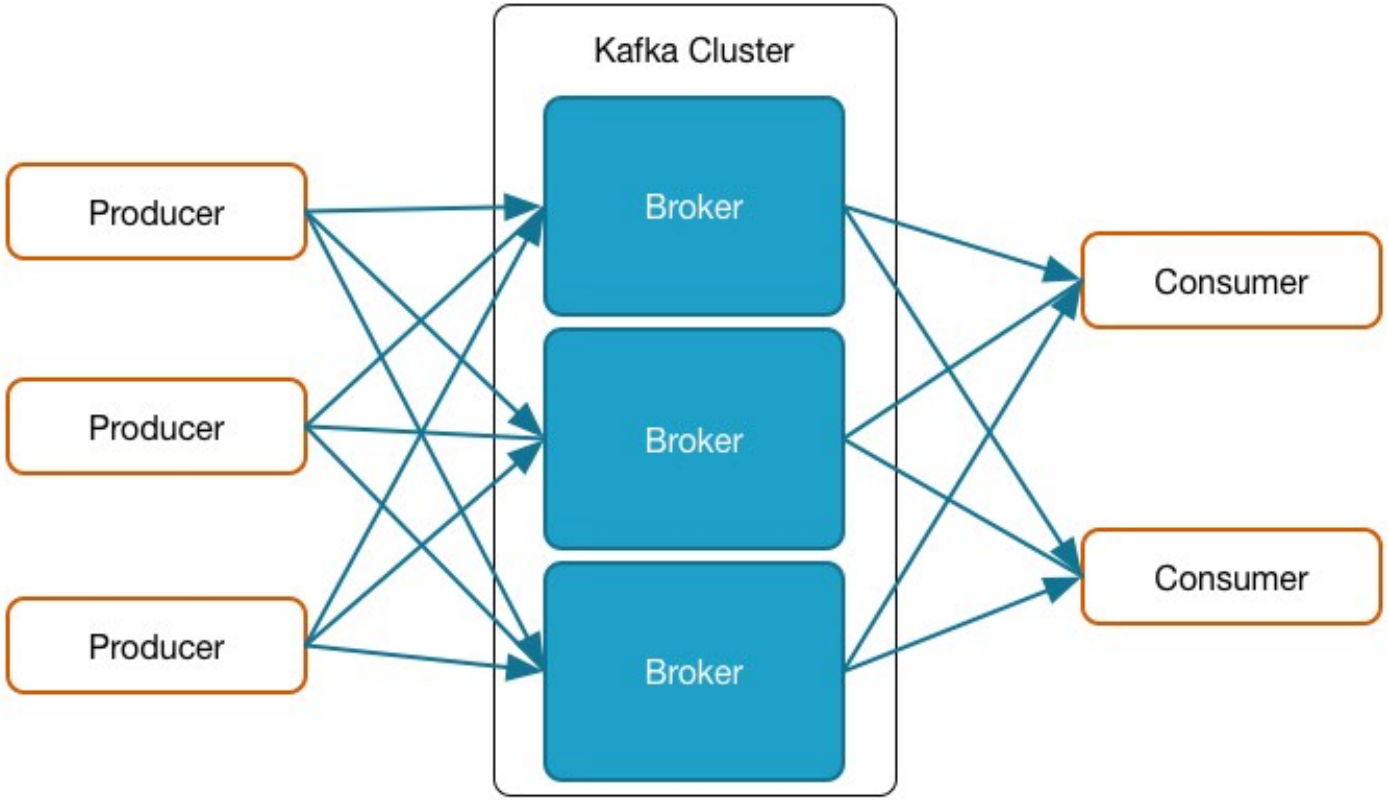
- **In this chapter you will learn:**
  - How Producers write data to a Kafka cluster
  - How data is divided into partitions, and then stored on Brokers
  - How Consumers read data from the cluster

Kafka Fundamentals

- **An Overview of Kafka**
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Chapter Review*

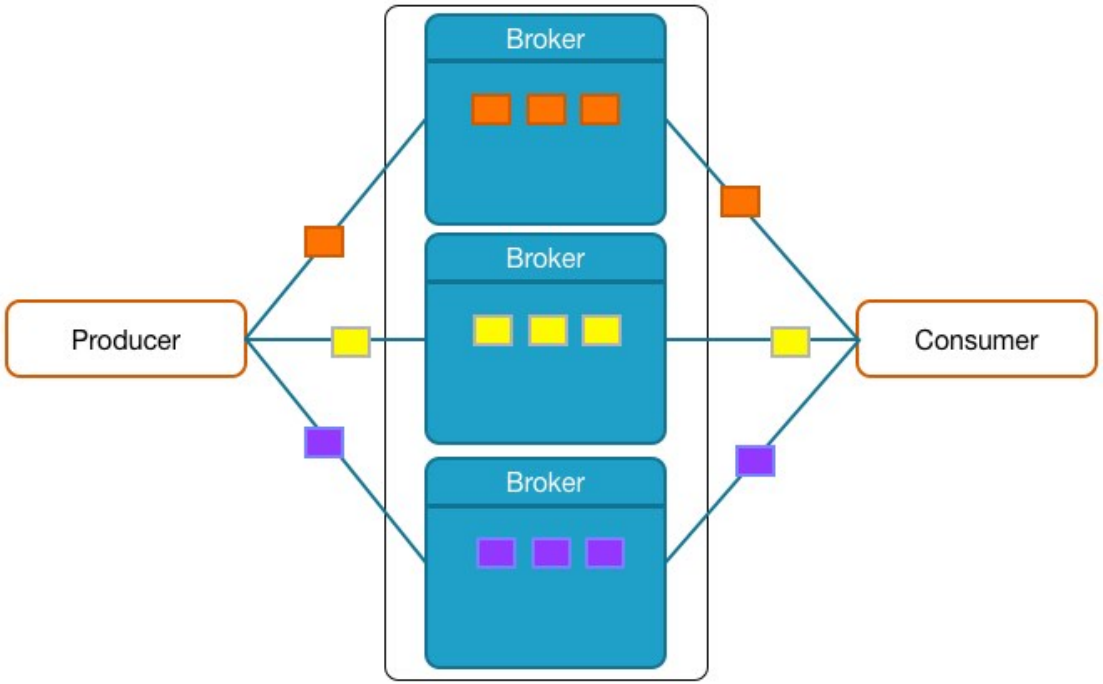
Reprise: A Very High-Level View of Kafka

- ***Producers*** **send data to the Kafka cluster**
- ***Consumers*** **read data from the Kafka cluster**
- ***Brokers*** **are the main storage and messaging components of the Kafka cluster**



Kafka Messages

- **The basic unit of data in Kafka is a *message***
  - *Message* is sometimes used interchangeably with *record*
  - Producers write messages to Brokers
  - Consumers read messages from Brokers

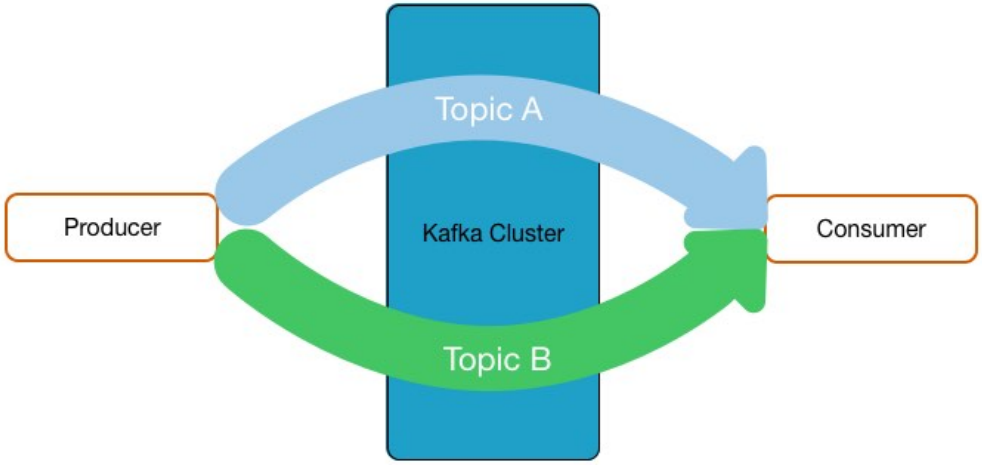


Key-Value Pairs

- **A message is a key-value pair**
  - All data is stored in Kafka as byte arrays
  - Producer provides serializers to convert the key and value to byte arrays
  - Key and value can be any data type

Topics

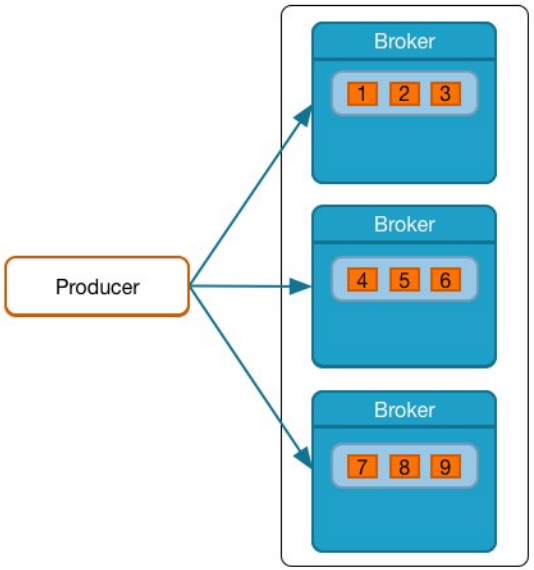
- **Kafka maintains streams of messages called *Topics***
  - Logical representation
  - They categorize messages into groups



- **Developers decide which Topics exist**
  - By default, a Topic is auto-created when it is first used
- **One or more Producers can write to one or more Topics**
- **There is no limit to the number of Topics that can be used**

Partitioned Data Ingestion

- **Producers shard data over a set of *Partitions***
  - Each Partition contains a subset of the Topic's messages
  - Each Partition is an ordered, immutable log of messages
- **Partitions are distributed across the Brokers**
- **Typically, the message key is used to determine which Partition a message is assigned to**
  - This can be overridden by the Producer



Kafka Components

- **There are four key components in a Kafka system**
  - Producers
  - Brokers
  - Consumers
  - ZooKeeper
- **We will now investigate each of these in turn**

Kafka Fundamentals

- *An Overview of Kafka*
- **Kafka Producers**
- *Kafka Brokers*
- *Kafka Consumers*
- *Chapter Review*

Producer Basics

- **Producers write data in the form of *messages* to the Kafka cluster**
- **Producers can be written in any language**
  - Native Java, C, Python, and Go clients are supported by Confluent
  - Clients for many other languages exist
  - Confluent develops and supports a REST (REpresentational State Transfer) server which can be used by clients written in any language
- **A command-line Producer tool exists to send messages to the cluster**
  - Useful for testing, debugging, etc.

Load Balancing and Semantic Partitioning

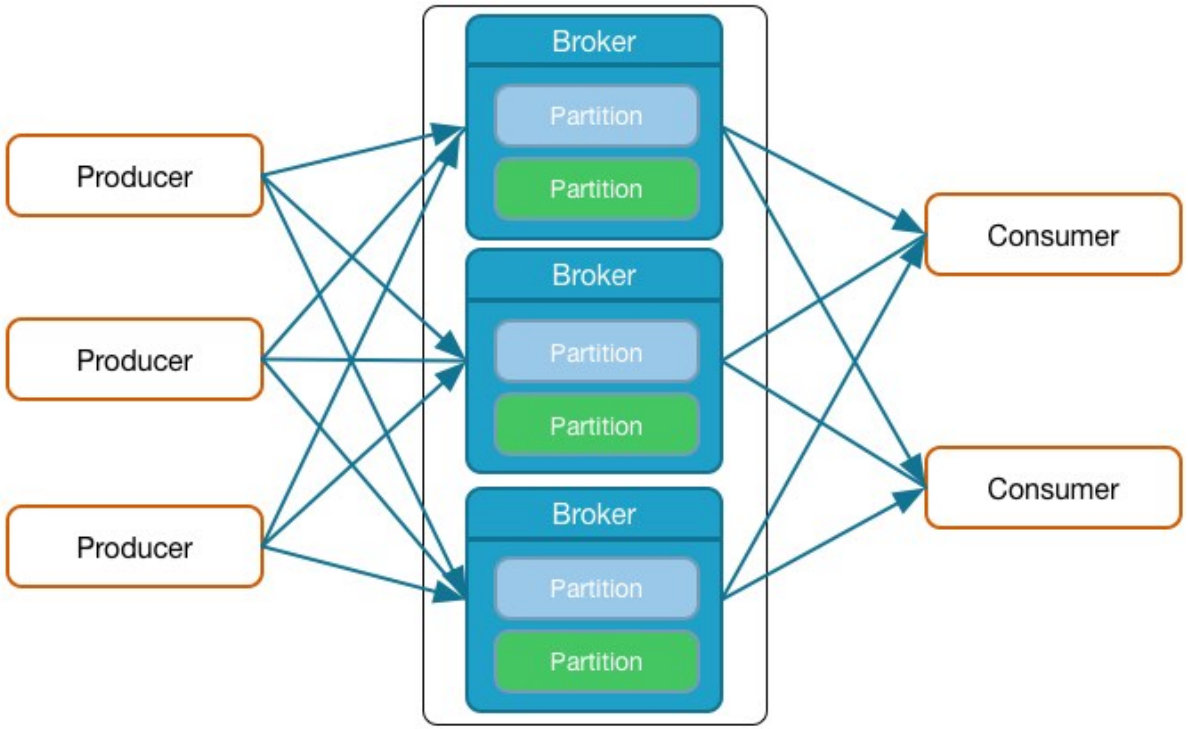
- **Producers use a partitioning strategy to assign each message to a Partition**
- **Having a partitioning strategy serves two purposes**
  - Load balancing: shares the load across the Brokers
  - Semantic partitioning: user-specified key allows locality-sensitive message processing
- **The partitioning strategy is specified by the Producer**
  - Default strategy is a hash of the message key
    - `hash(key) % number_of_partitions`
  - If a key is not specified, messages are sent to Partitions on a round-robin basis
- **Developers can provide a custom partitioner class**

Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- **Kafka Brokers**
- *Kafka Consumers*
- *Chapter Review*

Broker Basics

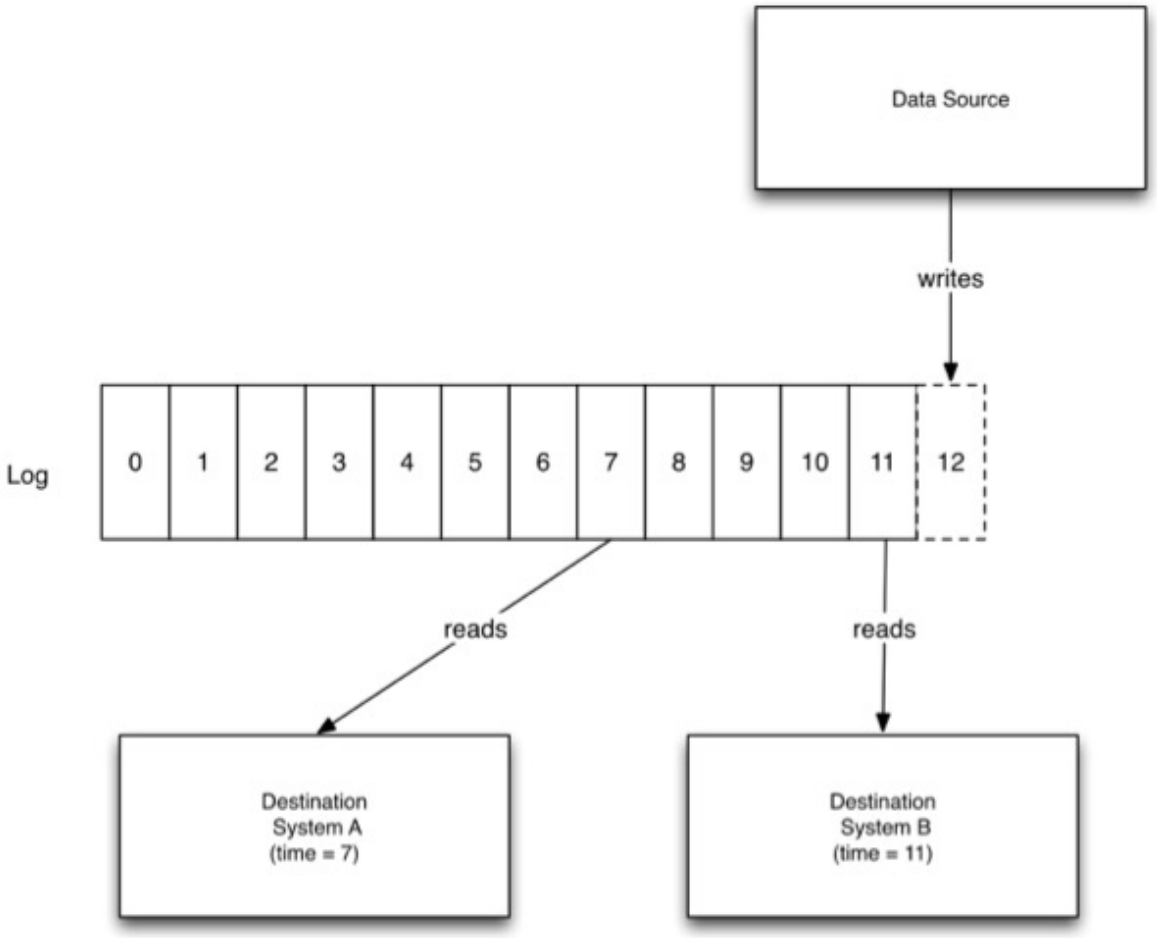
- **Brokers receive and store messages when they are sent by the Producers**
- **A Kafka cluster will typically have multiple Brokers**
  - Each can handle hundreds of thousands, or millions, of messages per second
- **Each Broker manages multiple Partitions**



Brokers Manage Partitions

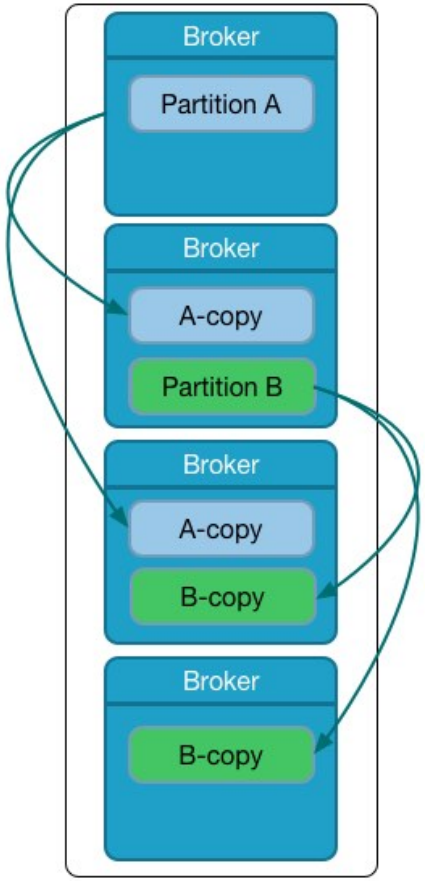
- **Messages in a Topic are spread across Partitions in different Brokers**
- **Typically, a Broker will handle many Partitions**
- **Each Partition is stored on the Broker's disk as one or more `log` files**
  - Not to be confused with `log4j` files used for monitoring
- **Each message in the log is identified by its *offset***
  - A monotonically increasing value
- **Kafka provides a configurable retention policy for messages to manage log file growth**

Messages are Stored in a Persistent Log



Fault Tolerance via a Replicated Log

- **Partitions can be replicated across multiple Brokers**
- **Replication provides fault tolerance in case a Broker goes down**
  - Kafka automatically handles the replication



Kafka Fundamentals

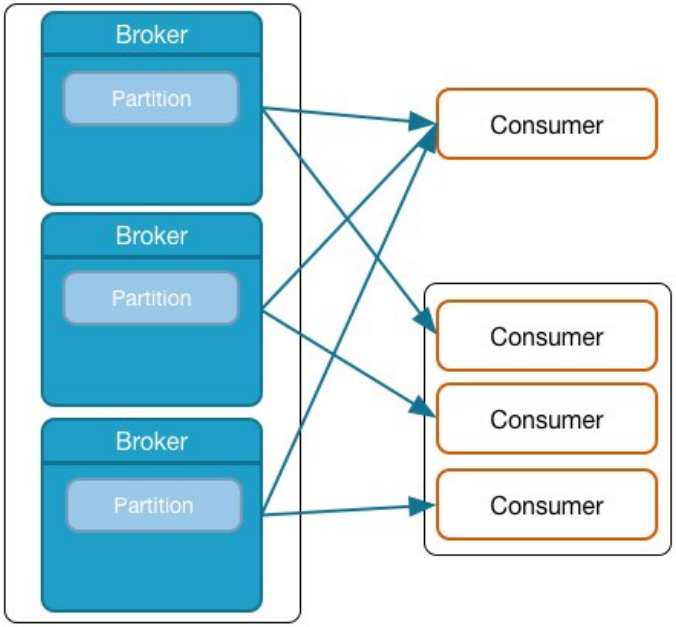
- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- **Kafka Consumers**
- *Chapter Review*

Consumer Basics

- **Consumers pull messages from one or more Topics in the cluster**
  - As messages are written to a Topic, the Consumer will automatically retrieve them
- **The *Consumer Offset* keeps track of the latest message read**
  - If necessary, the Consumer Offset can be changed
    - For example, to reread messages
- **The Consumer Offset is stored in a special Kafka Topic**
- **A command-line Consumer tool exists to read messages from the cluster**
  - Useful for testing, debugging, etc.

Distributed Consumption

- **Different Consumers can read data from the same Topic**
  - By default, each Consumer will receive all the messages in the Topic
- **Multiple Consumers can be combined into a *Consumer Group***
  - Consumer Groups provide scaling capabilities
  - Each Consumer is assigned a subset of Partitions for consumption

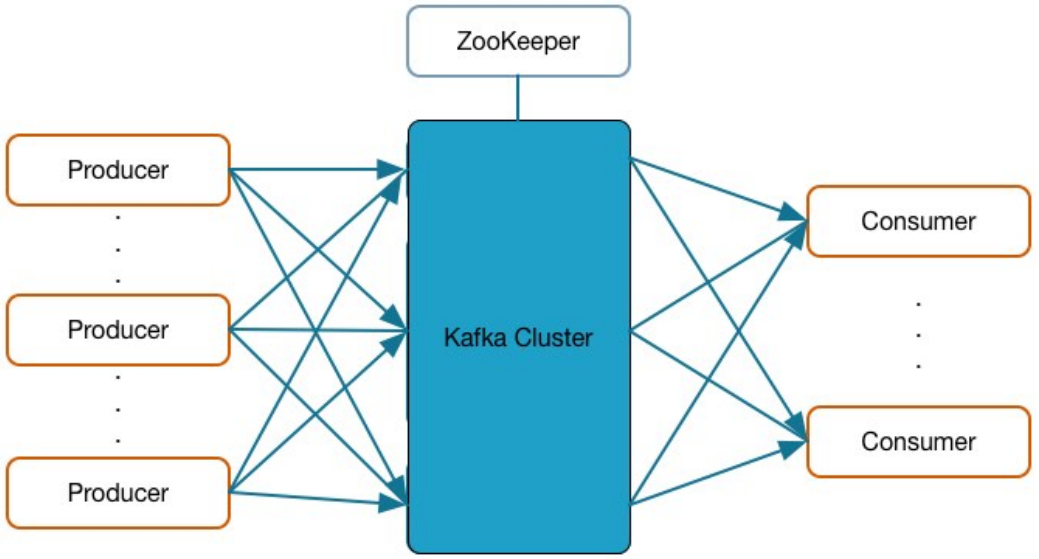


What is ZooKeeper?

- **ZooKeeper is a centralized service that can be used by distributed applications**
  - Open source Apache project
  - Enables highly reliable distributed coordination
  - Maintains configuration information
  - Provides distributed synchronization
- **Used by many projects**
  - Including Kafka and Hadoop
- **Typically consists of three or five servers in a *quorum***
  - This provides resiliency should a machine fail

How Kafka Uses ZooKeeper

- **Kafka Brokers use ZooKeeper for a number of important internal features**
  - Cluster management
  - Failure detection and recovery
  - Access Control List (ACL) storage
- **In earlier versions of Kafka, the Consumer needed access to the ZooKeeper quorum**
  - This is no longer the case



Hands-On Exercise: Using Kafka's Command-Line Tools

- **In this Hands-On Exercise you will use Kafka's command-line tools to Produce and Consume data**
- **Please refer to the Hands-On Exercise Manual**

Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- **Chapter Review**

Chapter Review

- **A Kafka system is made up of Producers, Consumers, and Brokers**
  - ZooKeeper provides co-ordination services for the Brokers
- **Producers write messages to Topics**
  - Topics are broken down into partitions for scalability
- **Consumers read data from one or more Topics**

Ingesting Data with Kafka Connect

Chapter 4

Course Contents

Ingesting Data with Kafka Connect

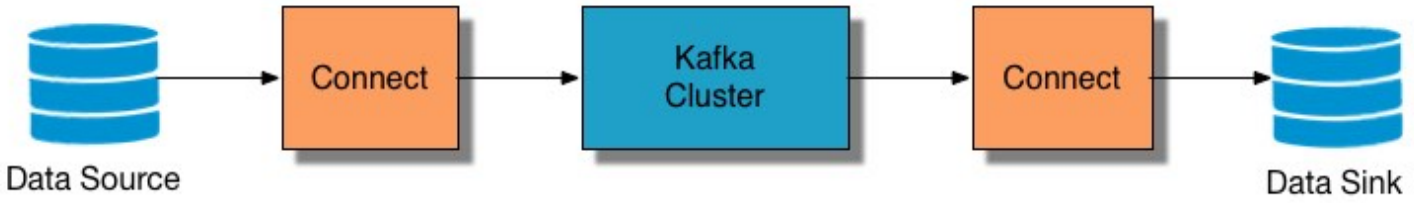
- **In this chapter you will learn:**
  - The motivation for Kafka Connect
  - What standard Connectors are provided
  - The differences between standalone and distributed mode
  - How to configure and use Kafka Connect
  - How Kafka Connect compares to writing your own data transfer system

Ingesting Data with Kafka Connect

- **The Motivation for Kafka Connect**
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

What is Kafka Connect?

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



Example Use Cases

- **Example use cases for Kafka Connect include:**
  - Stream an entire SQL database into Kafka
  - Stream Kafka topics into HDFS for batch processing
  - Stream Kafka topics into Elasticsearch for secondary indexing
  - …

Why Not Just Use Producers and Consumers?

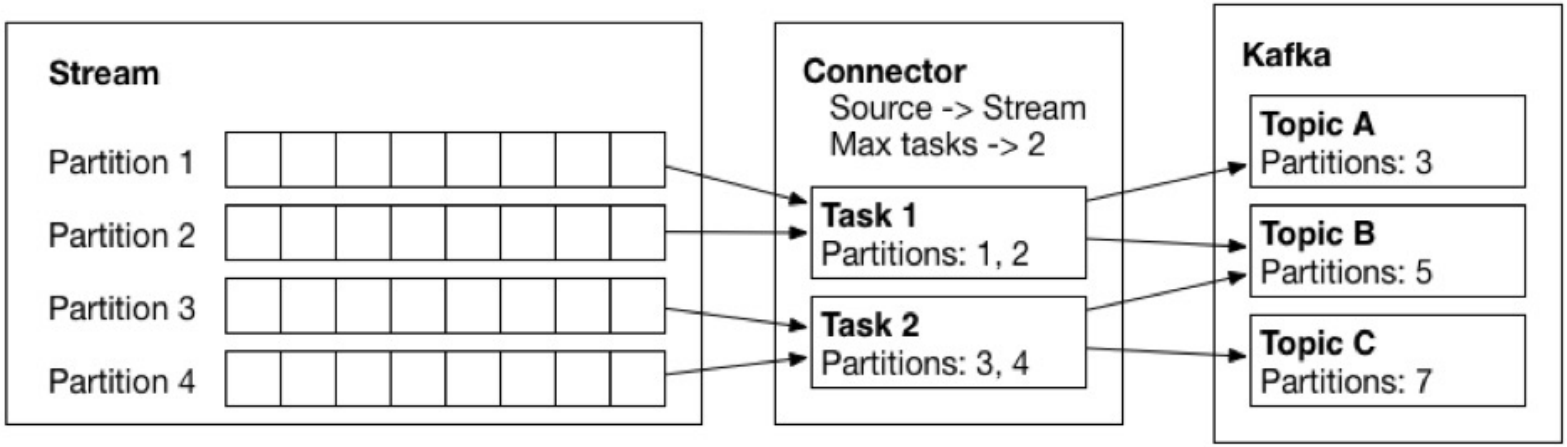
- **Internally, Kafka Connect is a Kafka client using the standard Producer and Consumer APIs**
- **Kafka Connect has benefits over 'do-it-yourself' Producers and Consumers:**
  - Off-the-shelf, tested Connectors for common data sources are available
  - Features fault tolerance and automatic load balancing when running in distributed mode
  - No coding required
    - Just write configuration files for Kafka Connect
  - Pluggable/extendable by developers

Connect Basics

- ***Connectors* are logical jobs responsible for managing the copying of data between Kafka and another system**
- **Connector *Sources* read data *from* an external data system into Kafka**
  - Internally, a connector source is a Kafka Producer client
- **Connector *Sinks* write Kafka data *to* an external data system**
  - Internally, a connector sink is a Kafka Consumer client

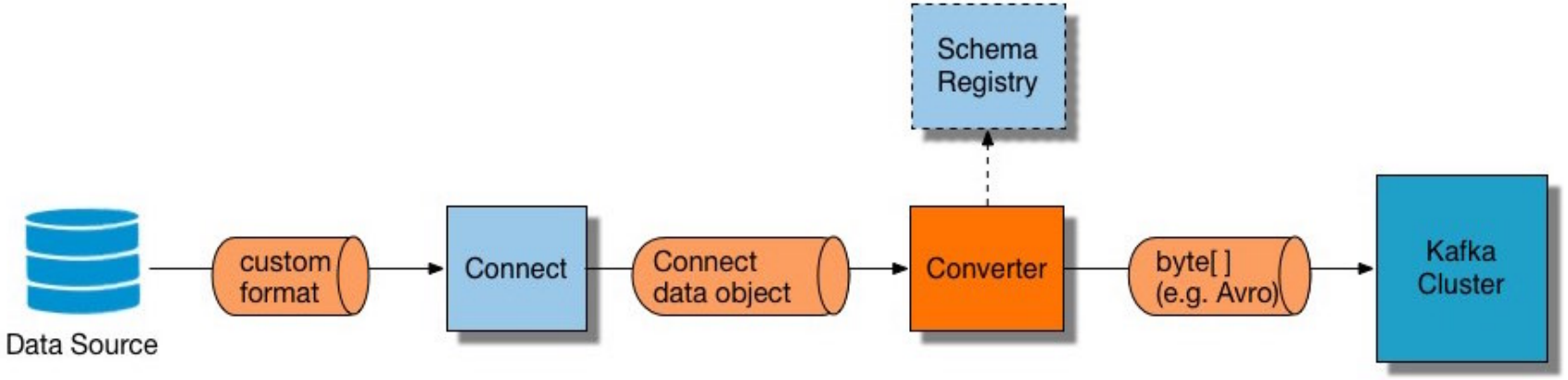
Providing Parallelism and Scalability

- **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 in different threads**
- **Input stream can be partitioned for parallelism, for example:**
  - File input: Partition → file
  - Database input: Partition → table



Converting Data

- ***Converters*** **provide the data format written to or read from Kafka (like Serializers)**



Converter Data Formats

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

Avro Converter as a Best Practice

- **Best Practice is to use an Avro Converter and Schema Registry with the Connectors**

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

- **Benefits**
  - Provides a data structure format
  - Supports code generation of data types
  - Avro data is binary, so stores data efficiently
  - Type checking is performed at write time
- **Avro schemas evolve as updates to code happen**
  - Connectors may support schema evolution and react to schema changes in a configurable way
- **Schemas can be centrally managed in a Schema Registry**

Off-The-Shelf Connectors

- **Confluent Open Source ships with commonly used Connectors**
  - FileStream
  - JDBC
  - HDFS
  - Elasticsearch
  - AWS S3
- **Confluent Enterprise includes additional connectors**
  - Replicator
  - JMS
- **Many other certified Connectors are available**
  - See `https://www.confluent.io/product/connectors/`

JDBC Source Connector: Overview

- **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
- **Records from each table are Produced to their own Kafka topic**
- **New and deleted tables are handled automatically**

JDBC Source Connector: Detecting New and Updated Rows

- **The Connector can detect new and updated rows in several ways:**

Incremental query made	Description
Incrementing column	Check a single column where newer rows have a larger, autoincremented ID. Does not support updated rows
Timestamp column	Checks a single 'last modified' column. Can't guarantee reading all updates
Timestamp and incrementing column	Combination of the two methods above. Guarantees that all updates are read
Custom query	Used in conjunction with the options above for custom filtering

- **Alternative: `bulk` mode for one-time load, not incremental, unfiltered**

JDBC Source Connector: Configuration

Parameter	Description
connection.url	The JDBC connection URL for the database
topic.prefix	The prefix to prepend to table names to generate the Kafka topic name
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
table.blacklist	A list of tables to ignore and not import. If specified, **tables.whitelist** cannot be specified
table.whitelist	A list of tables to import. If specified, **tables.blacklist** cannot be specified

- **Note: This is not a complete list. See http://docs.confluent.io**

HDFS Sink Connector: Overview

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

Ingesting Data with Kafka Connect

- *The Motivation for Kafka Connect*
- **Standalone and Distributed Modes**
- *Configuring the Connectors*
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

Two Modes: Standalone and Distributed

- **Kafka Connect can be run in two modes**
  - Standalone mode
    - Single worker process on a single machine
    - Use case: testing and development, or when a process should not be distributed (*e.g.* `tail` a log file)
  - Distributed mode
    - Multiple worker processes on one or more machines
    - Use Case: requirements for fault tolerance and scalability

Running in Standalone Mode

- **To run in standalone mode, start a process by providing as arguments**
  - Standalone configuration properties file
  - One or more connector configuration files
    - Each connector instance will be run in its own thread

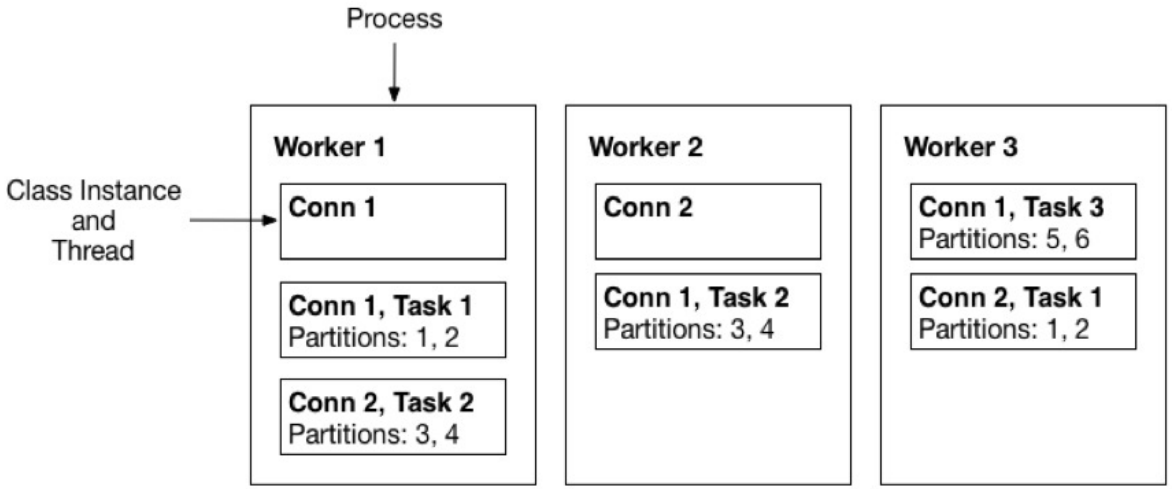
```
$ connect-standalone connect-standalone.properties \
connector1.properties [connector2.properties connector3.properties ...]
```

Running in Distributed Mode

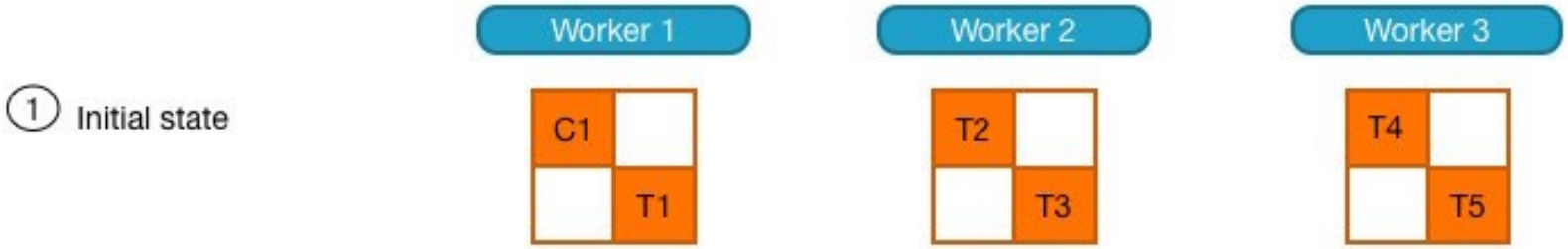
- **To run in distributed mode, start Kafka Connect on each worker node**

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

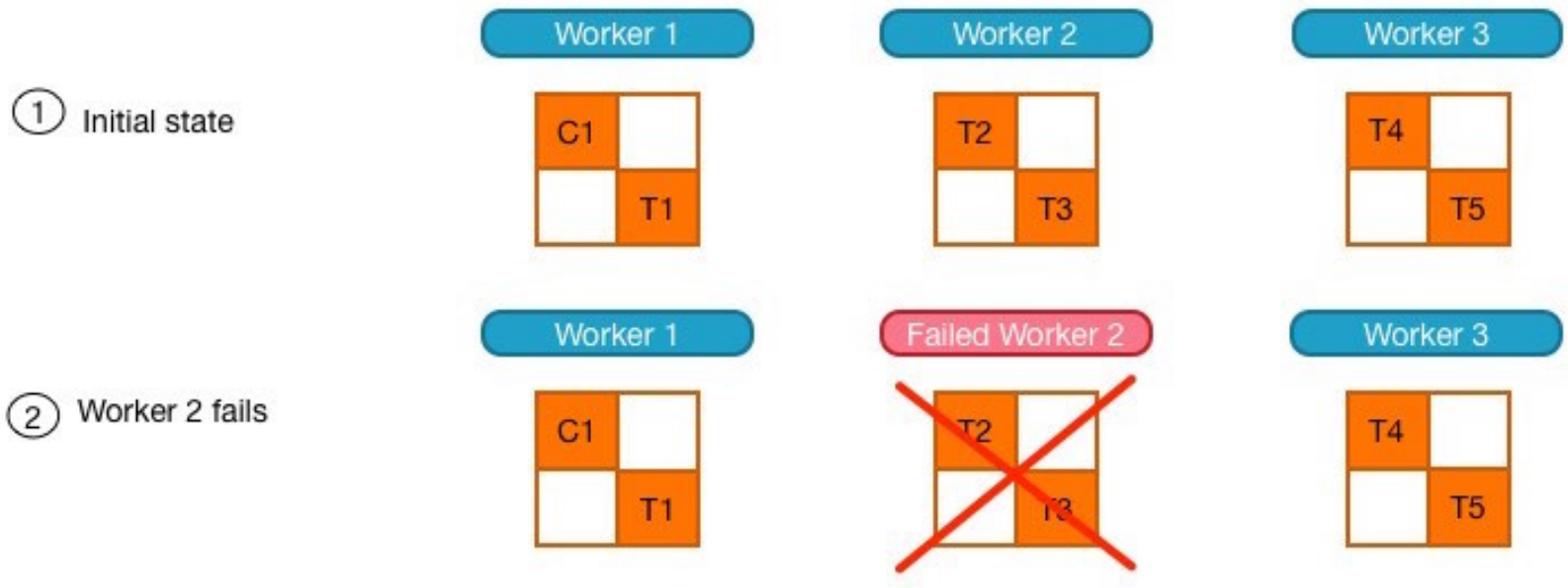
- **Group coordination**
  - Connect leverages Kafka's group membership protocol
    - Configure workers with the same `group.id`
  - Workers distribute load within this Kafka Connect "cluster"



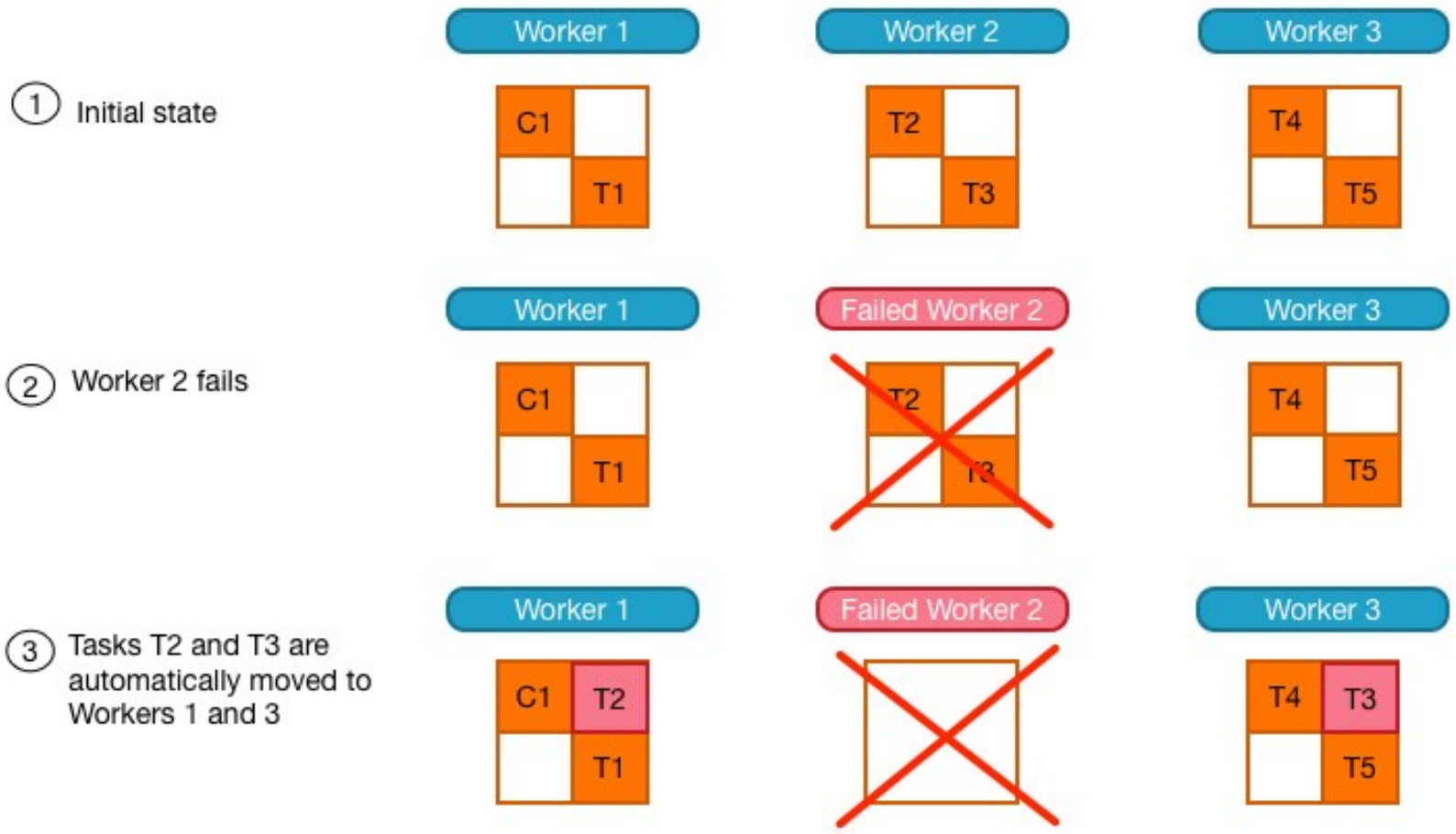
Providing Fault Tolerance in Distributed Mode (1)



Providing Fault Tolerance in Distributed Mode (2)



Providing Fault Tolerance in Distributed Mode (3)



- **Tasks have no state stored within them**
  - Task state is stored in special Topics in Kafka

Ingesting Data with Kafka Connect

- *The Motivation for Kafka Connect*
- *Standalone and Distributed Modes*
- **Configuring the Connectors**
- *Hands-On Exercise: Running Kafka Connect*
- *Chapter Review*

Transforming Data with Connect

- **Kafka 0.10.2.0 provides the ability to transform data a message-at-a-time**
  - Configured at the connector-level
  - Applied to the message key or value
- **A subset of available transformations:**
  - `InsertField`: insert a field using attributes from the message metadata or from a configured static value
  - `ReplaceField`: rename fields, or apply a blacklist or whitelist to filter
  - `ValueToKey`: replace the key with a new key formed from a subset of fields in the value payload
  - `TimestampRouter`: update the topic field as a function of the original topic value and timestamp
- **More information on Connect Transformations can be found at `http://kafka.apache.org/documentation.html#connect_transforms`**

The REST API

- **Connectors can be added, modified, and deleted via a REST API on port `8083`**
  - The REST requests can be made to any worker
- **In standalone mode, configurations can be done via a REST API**
  - This is optional to the other way of modifying the standalone configuration properties file
  - Changes made this way will not persist after worker restart
- **In distributed mode, configurations can be done only via a REST API**
  - Changes made this way will persist after a worker process restart
  - Connector configuration data is stored in a special Kafka topic

Ingesting Data with Kafka Connect

- *The Motivation for Kafka Connect*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- **Hands-On Exercise: Running Kafka Connect**
- *Chapter Review*

Hands-On Exercise: Running Kafka Connect

- **In this Hands-On Exercise, you will run Kafka Connect to pull data from a MySQL database into a Kafka topic**
- **Please refer to the Hands-On Exercise Manual**

Ingesting Data with Kafka Connect

- *The Motivation for Kafka Connect*
- *Standalone and Distributed Modes*
- *Configuring the Connectors*
- *Hands-On Exercise: Running Kafka Connect*
- **Chapter Review**

Chapter Review

- **Kafka Connect provides a scalable, reliable way to transfer data from external systems into Kafka, and vice versa**
- **Many off-the-shelf Connectors are provided by Confluent, and many others are under development by third parties**

Kafka Streams

Chapter 5

Course Contents

confluent

Kafka Streams

- **In this chapter you will learn:**
  - What Kafka Streams is
  - How to create a Kafka Streams Application
  - How the `KStream` and `KTable` abstractions differ

Kafka Streams

- **Kafka Streams Basics**
- *Kafka Streams Basic Concepts*
- *Key Features of Kafka Streams*
- *Kafka Streams APIs*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Chapter Review*

What Is Kafka Streams?

- **Kafka Streams is a lightweight Java library for building distributed stream processing applications using Kafka**
  - Easy to embed in your own applications
- **No external dependencies, other than Kafka**
- **Supports event-at-a-time processing (not microbatching) with millisecond latency**
- **Provides a table-like model for data**
- **Supports windowing operations, and stateful processing including distributed joins and aggregation**
- **Has fault-tolerance and supports distributed processing**
- **Includes both a Domain-Specific Language (DSL) and a low-level API**

A Library, Not a Framework

- **Kafka Streams is an alternative to streaming frameworks such as**
  - Spark Streaming
  - Apache Storm
  - Apache Samza
  - etc.
- **Unlike these, it does not require its own cluster**
  - Can run on a stand-alone machine, or multiple machines

When to Use Kafka Streams

- **Mainstream Application Development**
- **When running a cluster would be too painful**
- **Fast Data apps for small and big data**
- **Reactive and stateful applications**
- **Continuous transformations**
- **Continuous queries**
- **Microservices**
- **Event-driven systems**
- **The “T” in ETL**
- **…and more**

Why Not Just Build Your Own?

- **Many people are currently building their own stream processing applications**
  - Using the Producer and Consumer APIs
- **Using Kafka Streams is much easier than taking the 'do it yourself' approach**
  - Well-designed, well-tested, robust
  - Means you can focus on the application logic, not the low-level plumbing

Installing Kafka Streams

- **There is no 'installation'!**
- **Kafka Streams is a library. Add it to your application like any other library**

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>0.10.1.0</version>
</dependency>
```

But… Where's THE CLUSTER to Process the Data?

- **There is no cluster!**
- **Unlearn your bad habits: 'do cool stuff with data' != 'must have a cluster'**

OK

OK

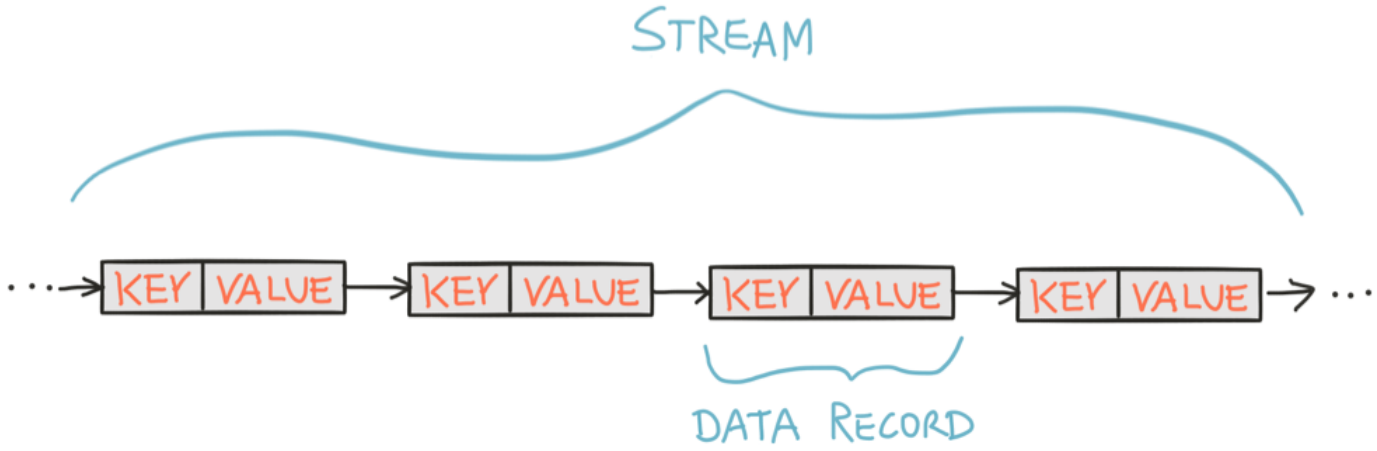
OK

Kafka Streams

- *Kafka Streams Basics*
- **Kafka Streams Basic Concepts**
- *Key Features of Kafka Streams*
- *Kafka Streams APIs*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Chapter Review*

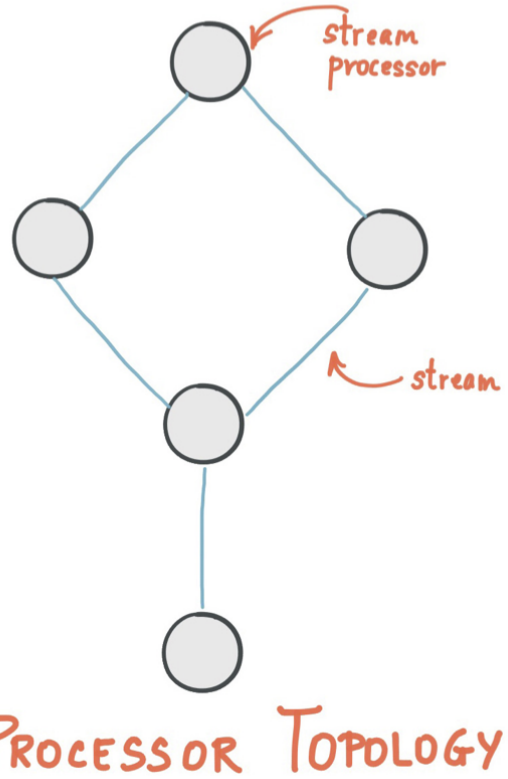
Kafka Streams Basic Concepts (1)

- **A *stream* is an unbounded, continuously updating data set**



Kafka Streams Basic Concepts (2)

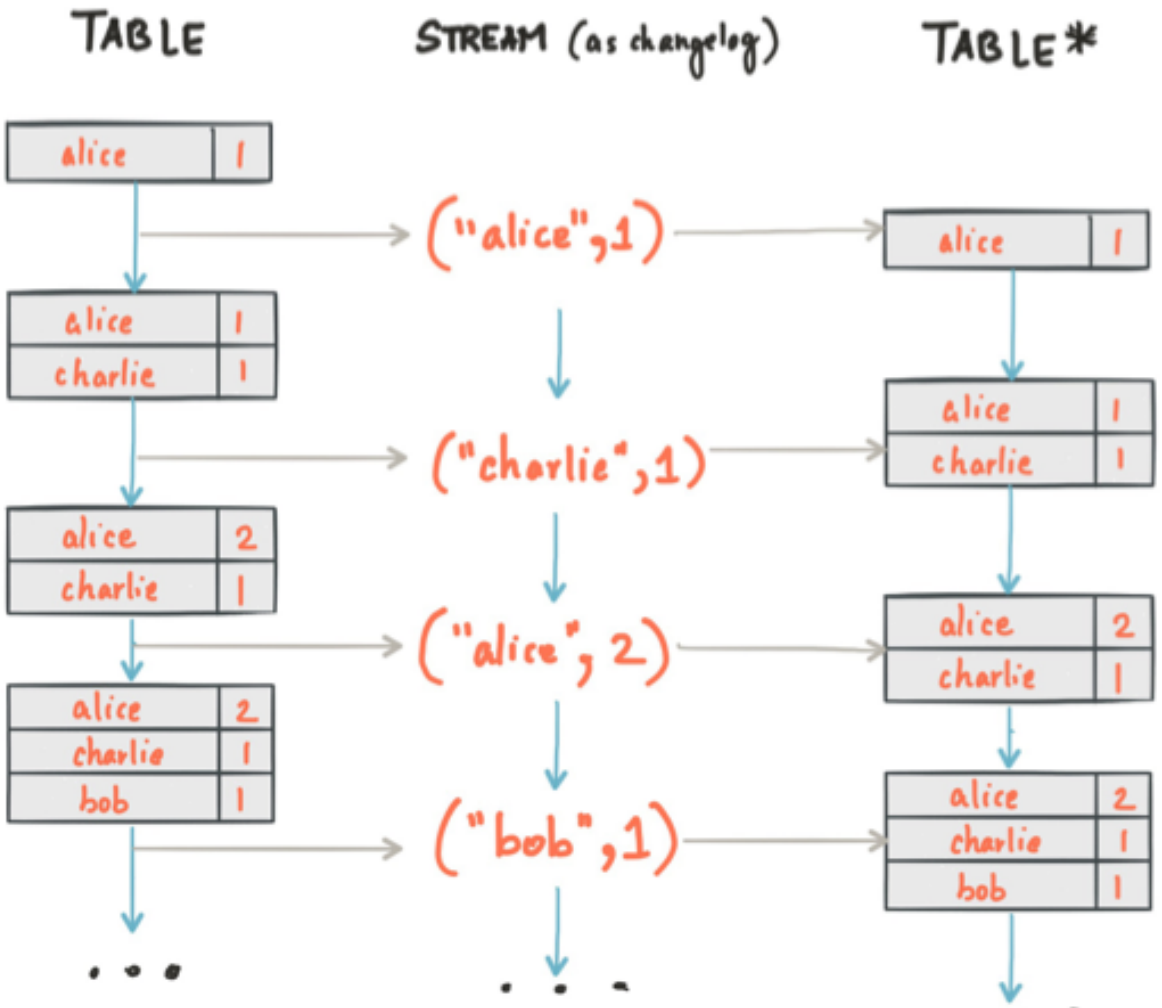
- A *stream processor* transforms data in a stream
- A *processor topology* defines the data flow through the stream processors



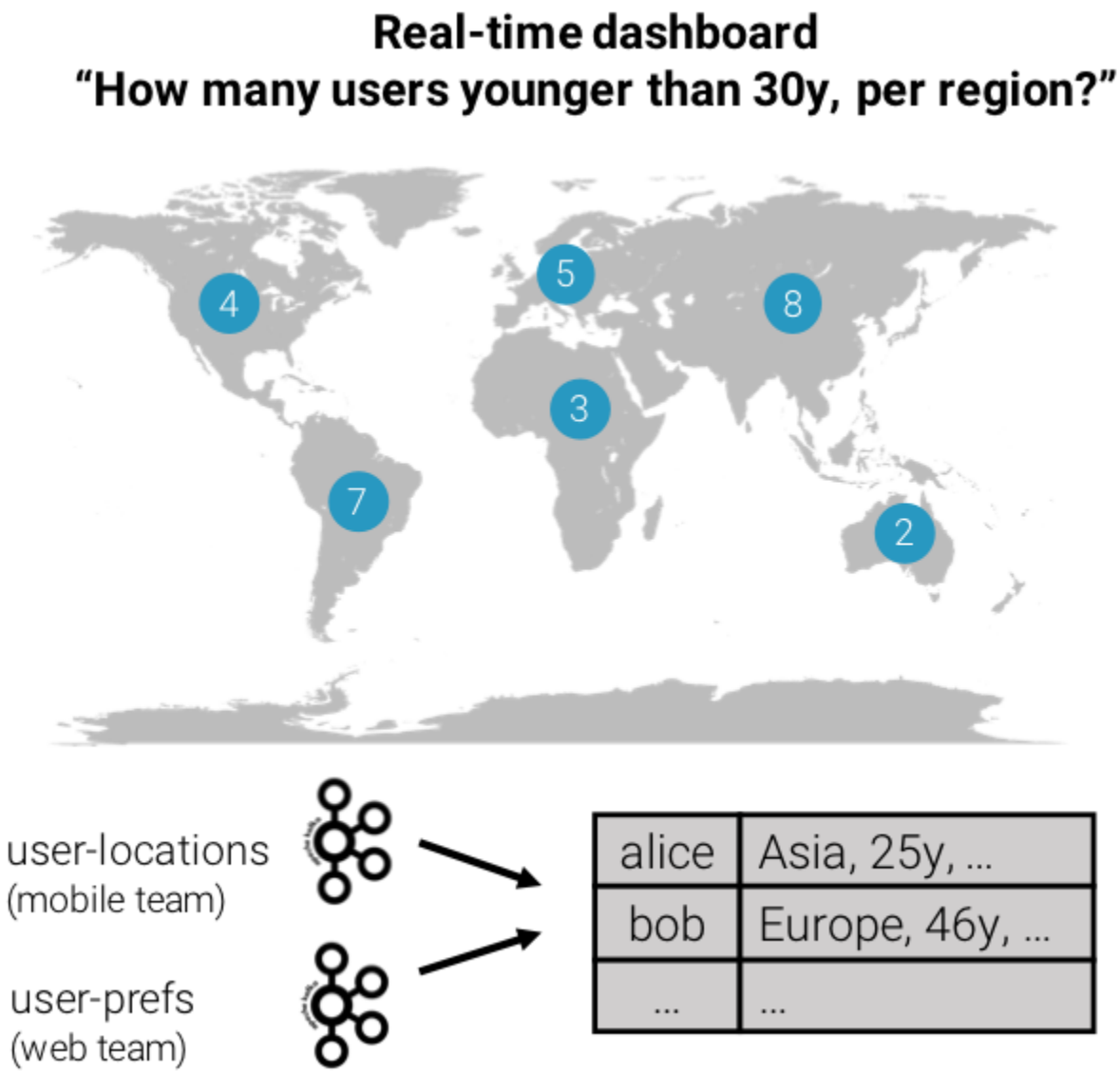
KStreams and KTables

- **A KStream is an abstraction of a record stream**
  - Each record represents a self-contained piece of data in the unbounded data set
- **A KTable is an abstraction of a changelog stream**
  - Each record represents an update
- **Example: We send two records to the stream**
  - `('apple', 1)`, and `('apple', 5)`
- **If we were to treat the stream as a KStream and sum up the values for apple, the result would be 6**
- **If we were to treat the stream as a KTable and sum up the values for apple, the result would be 5**
  - The second record is treated as an update to the first, because they have the same key
- **Typically, if you are going to treat a topic as a KTable it makes sense to configure log compaction on the topic**

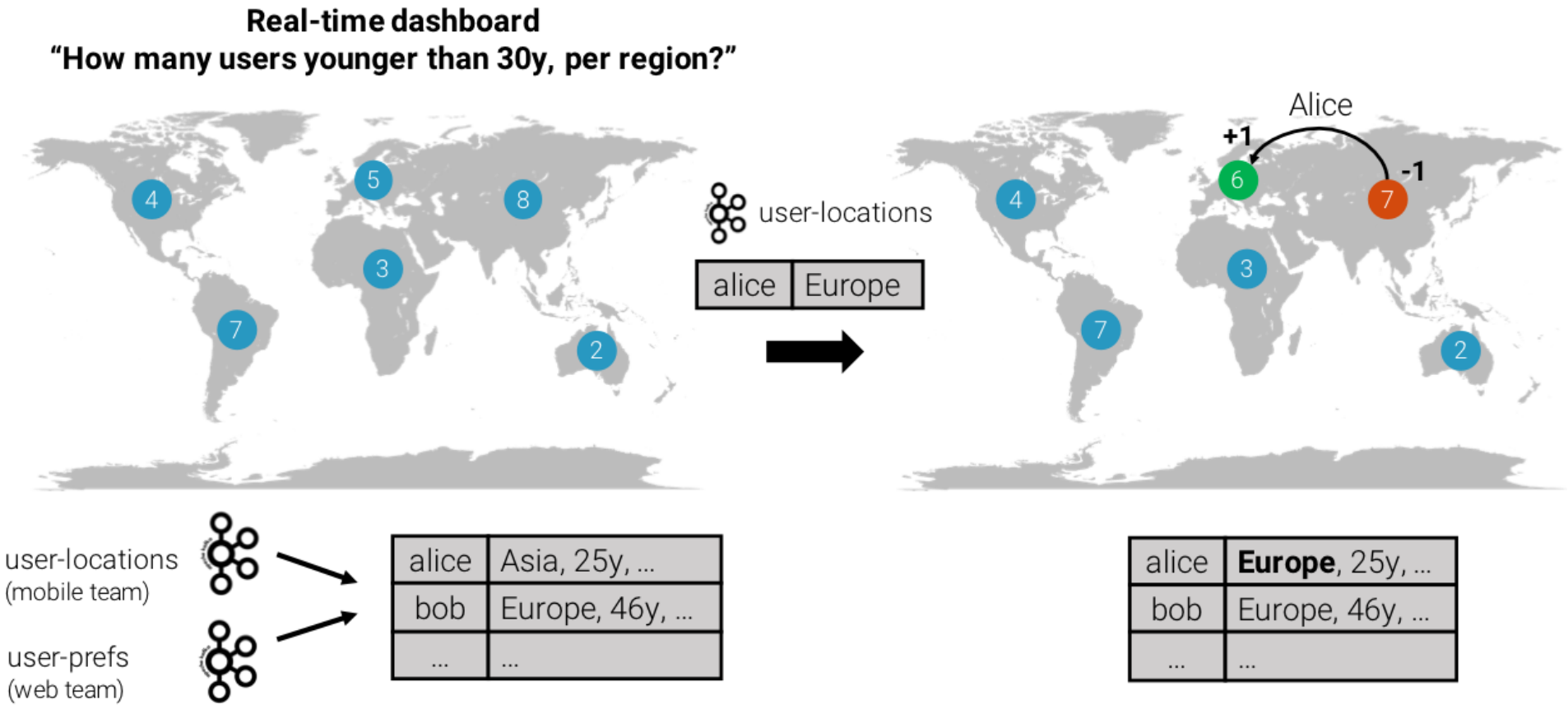
KStreams and KTables: Example



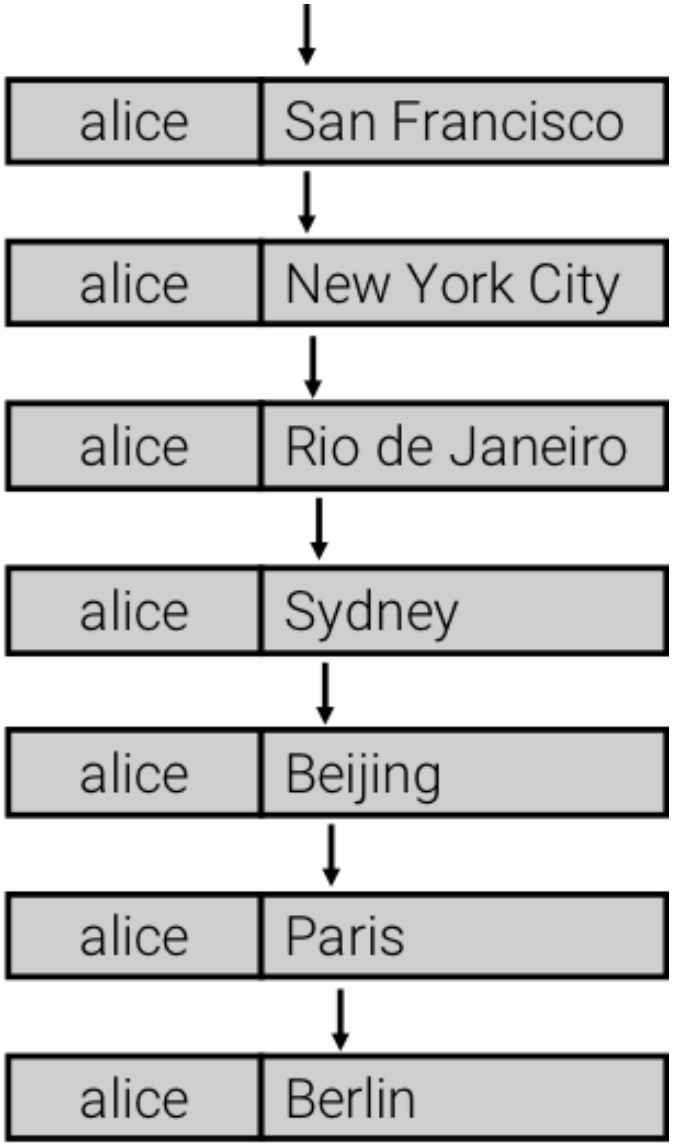
Motivating Example: Users Per Region



Motivating Example: Users Per Region



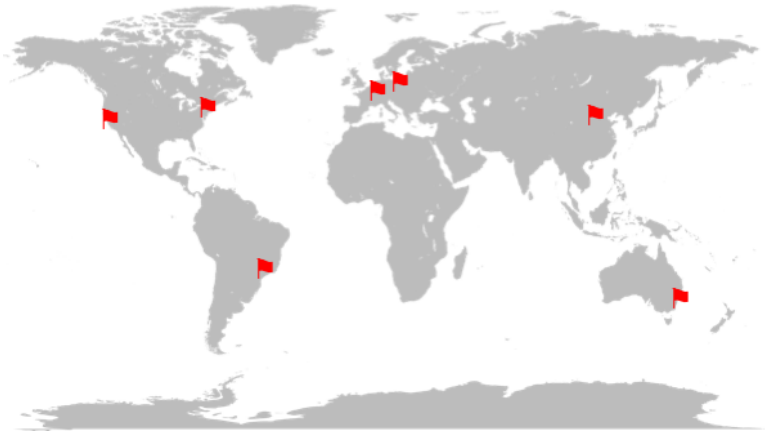
Different Use Cases, Different Interpretations (1)



Different Use Cases, Different Interpretations (2)

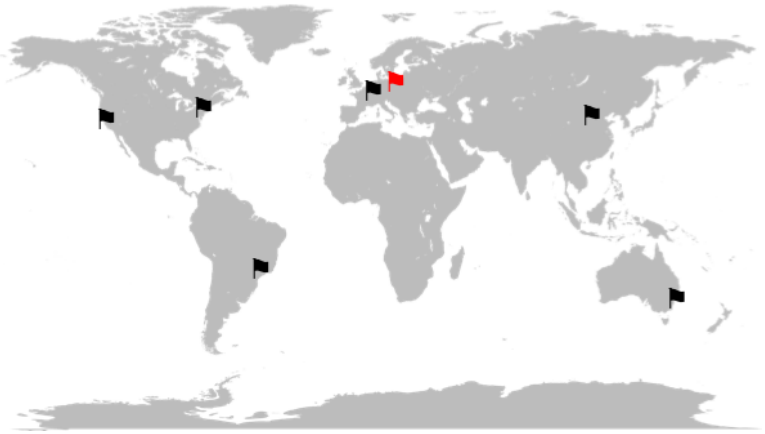
Use case 1: Frequent traveler status?

"Alice has been to SFO, NYC, Rio, Sydney, Beijing, Paris, and finally Berlin."



Use case 2: Current location?

"Alice is in ~~SFO~~, ~~NYC~~, ~~Rio~~, ~~Sydney~~, ~~Beijing~~, ~~Paris~~, Berlin right now."



Streams, Meet Tables

Example	When you need…	Read the Topic into	Data interpreted as	Messages interpreted as
All the places Alice has ever been	All the values of a key	`KStream`	***record*** stream	INSERT (append)
Where Alice is right now	Latest value of a key	`KTable`	***changelog*** stream	UPSERT (overwrite existing)

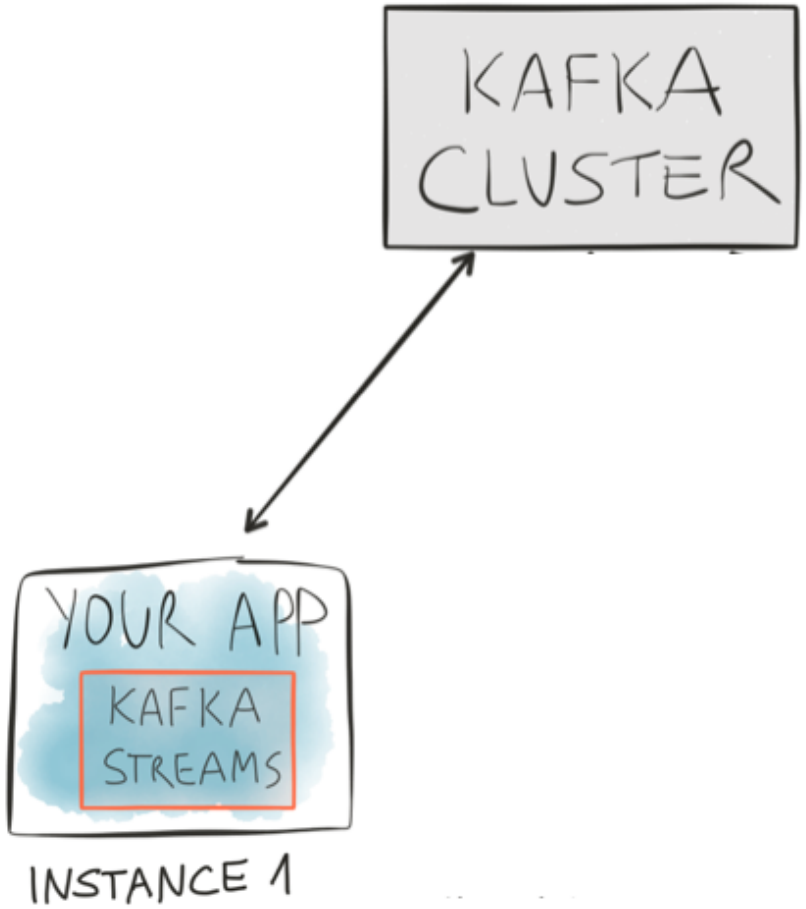
Kafka Streams

- *Kafka Streams Basics*
- *Kafka Streams Basic Concepts*
- **Key Features of Kafka Streams**
- *Kafka Streams APIs*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Chapter Review*

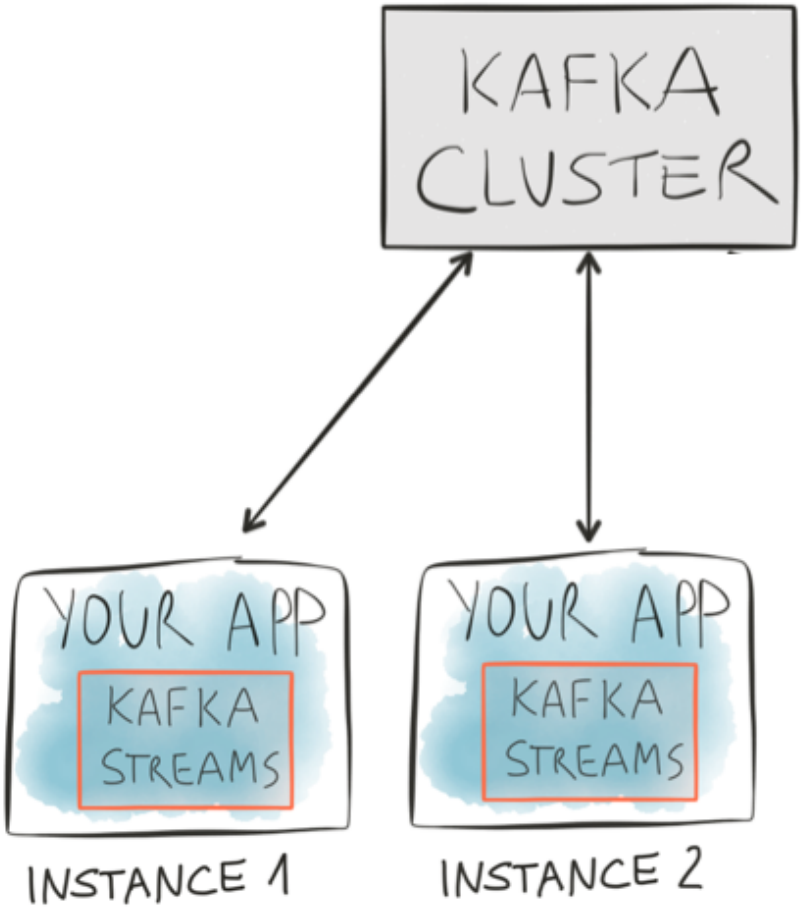
Key Kafka Streams Features

- **Key Kafka Streams features**
  - Reliable
  - Fault-tolerant
  - Scalable
  - Stateful and Stateless Computations
  - Interactive Queries

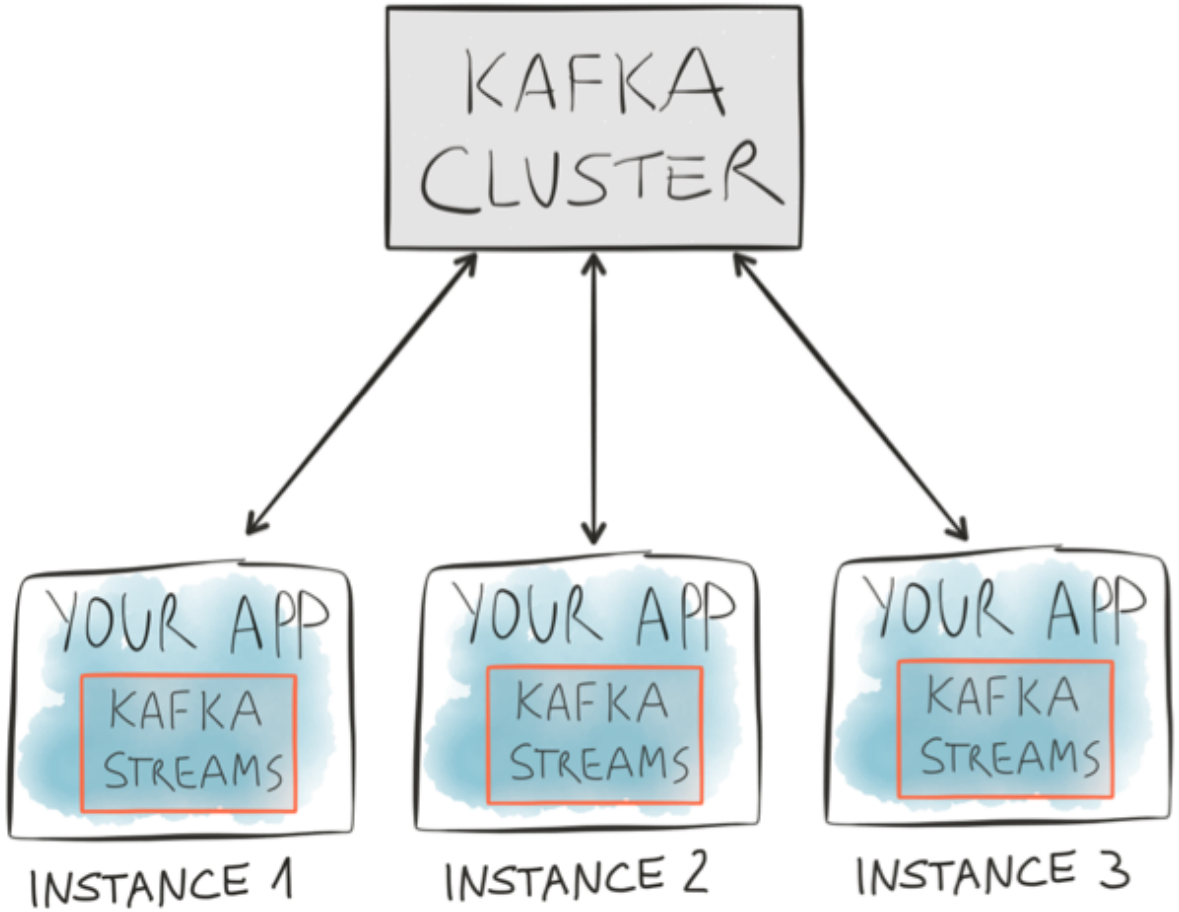
Fault-Tolerance and Scalability (1)



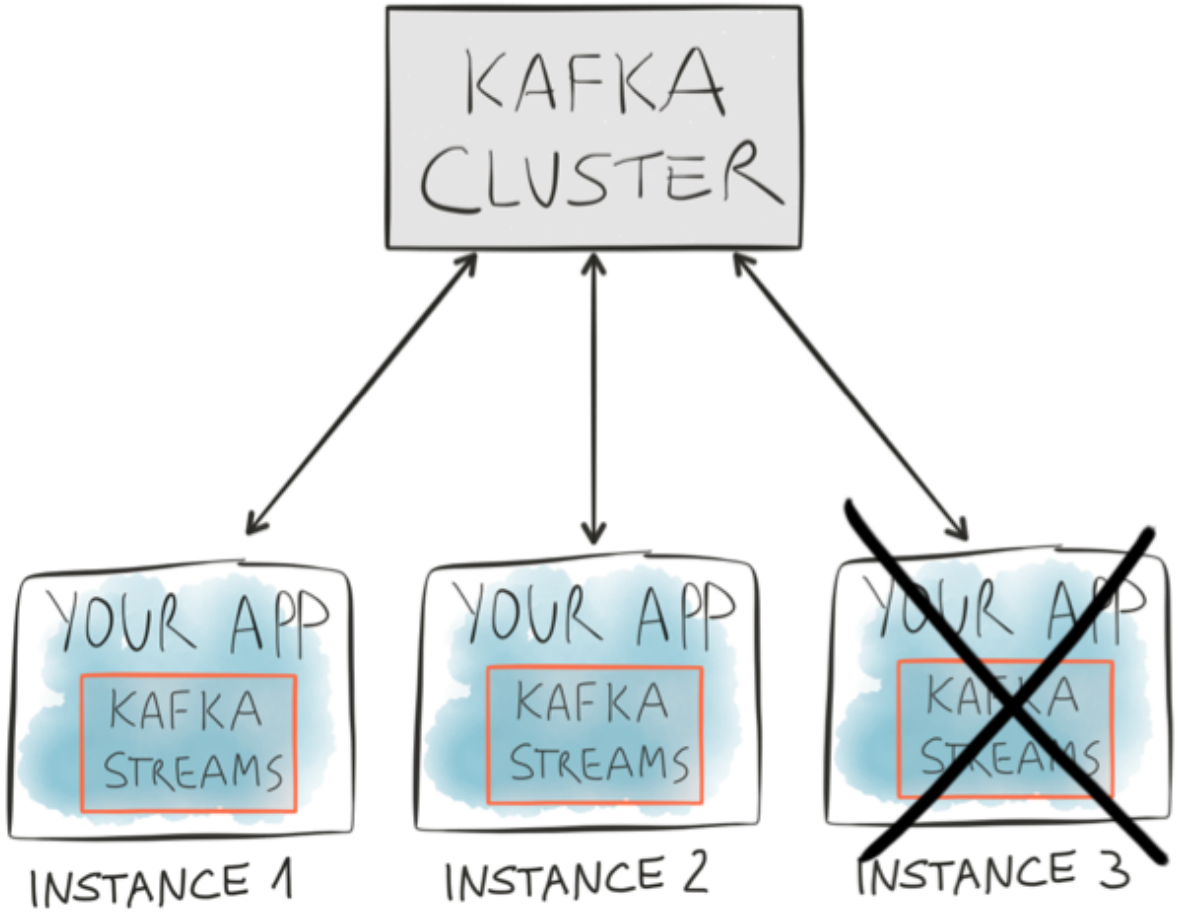
Fault-Tolerance and Scalability (2)



Fault-Tolerance and Scalability (3)



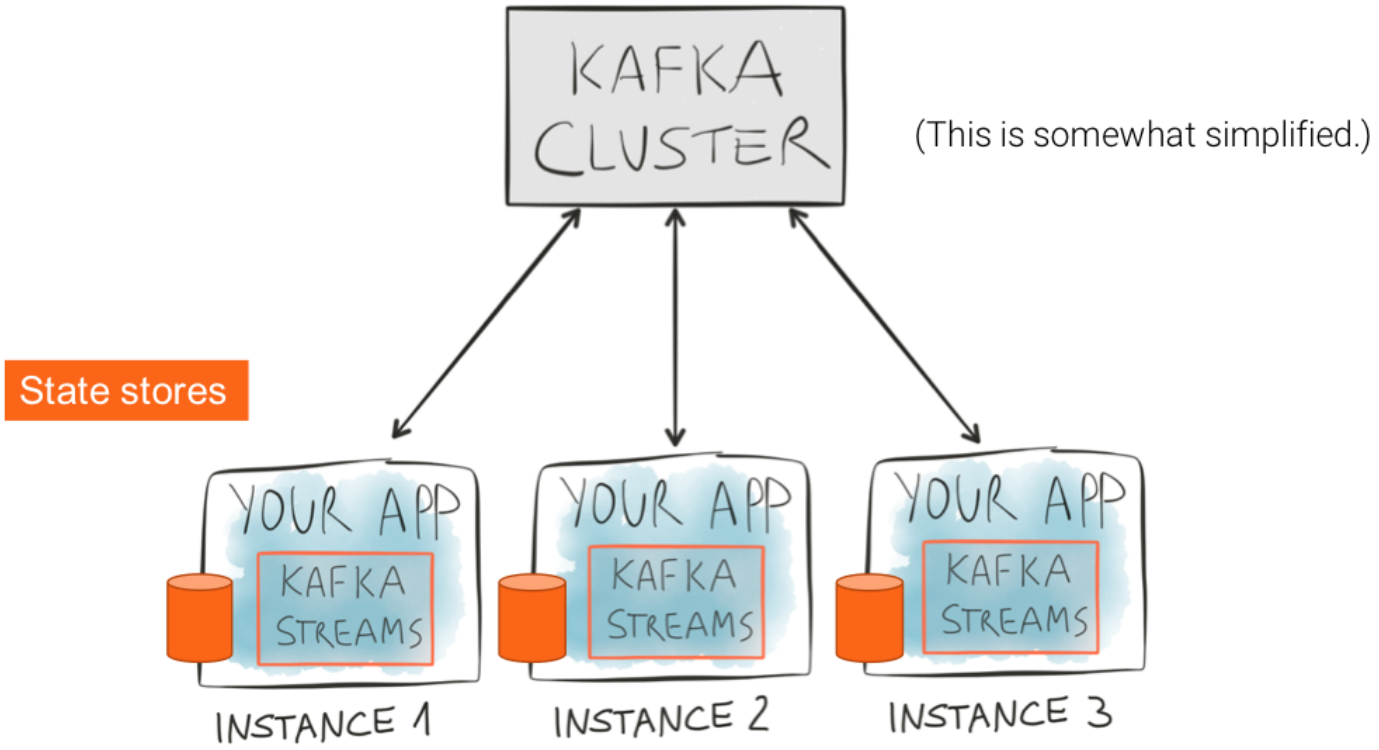
Fault-Tolerance and Scalability (4)



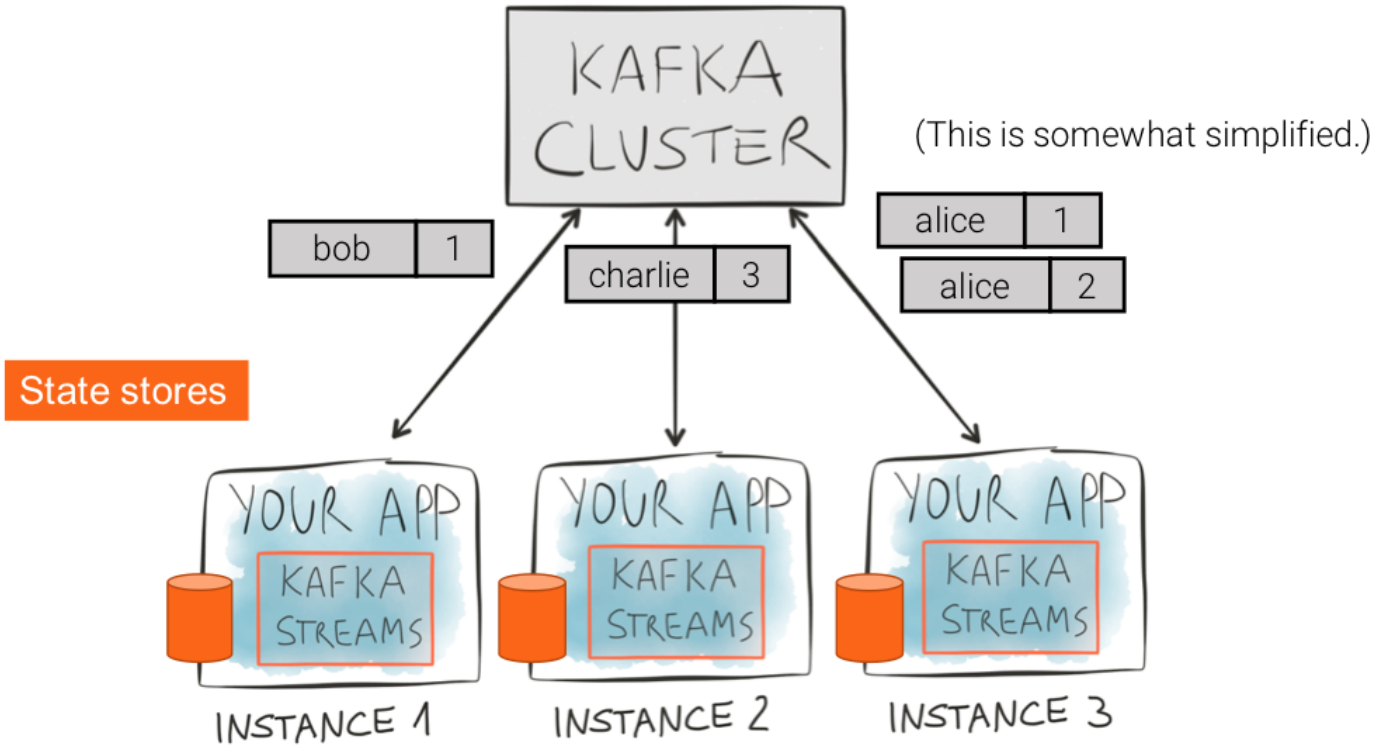
Stateful Computations

- **Stateful computations like aggregations or joins require state**
- **Example: `count()` will cause the creation of a state store to keep track of counts**
- **State stores in Kafka Streams...**
  - are local for best performance
  - are replicated to Kafka for elasticity and for fault-tolerance
- **Pluggable storage engines**
  - Default: RocksDB (key-value store) to allow for local state that is larger than available RAM
  - Further built-in options available: in-memory store
  - You can also use your own, custom storage engine

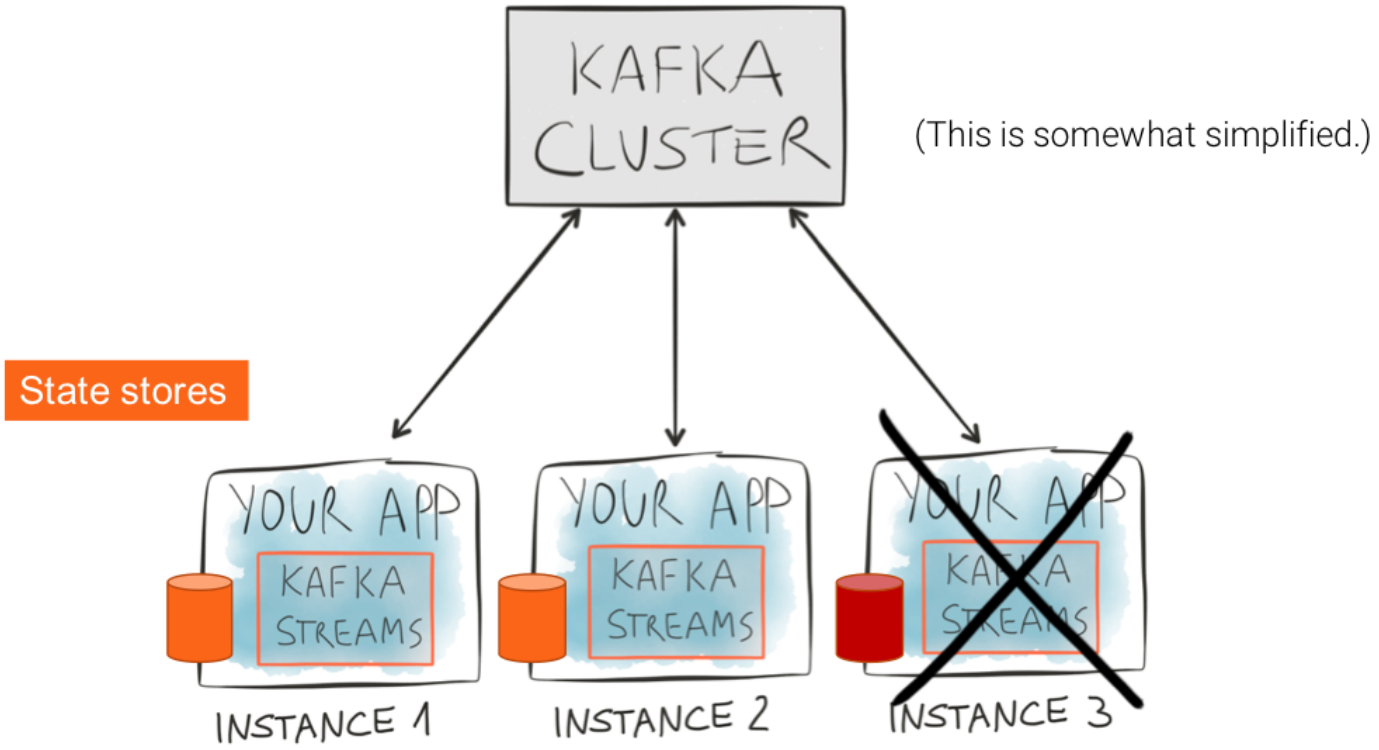
State Management (1)



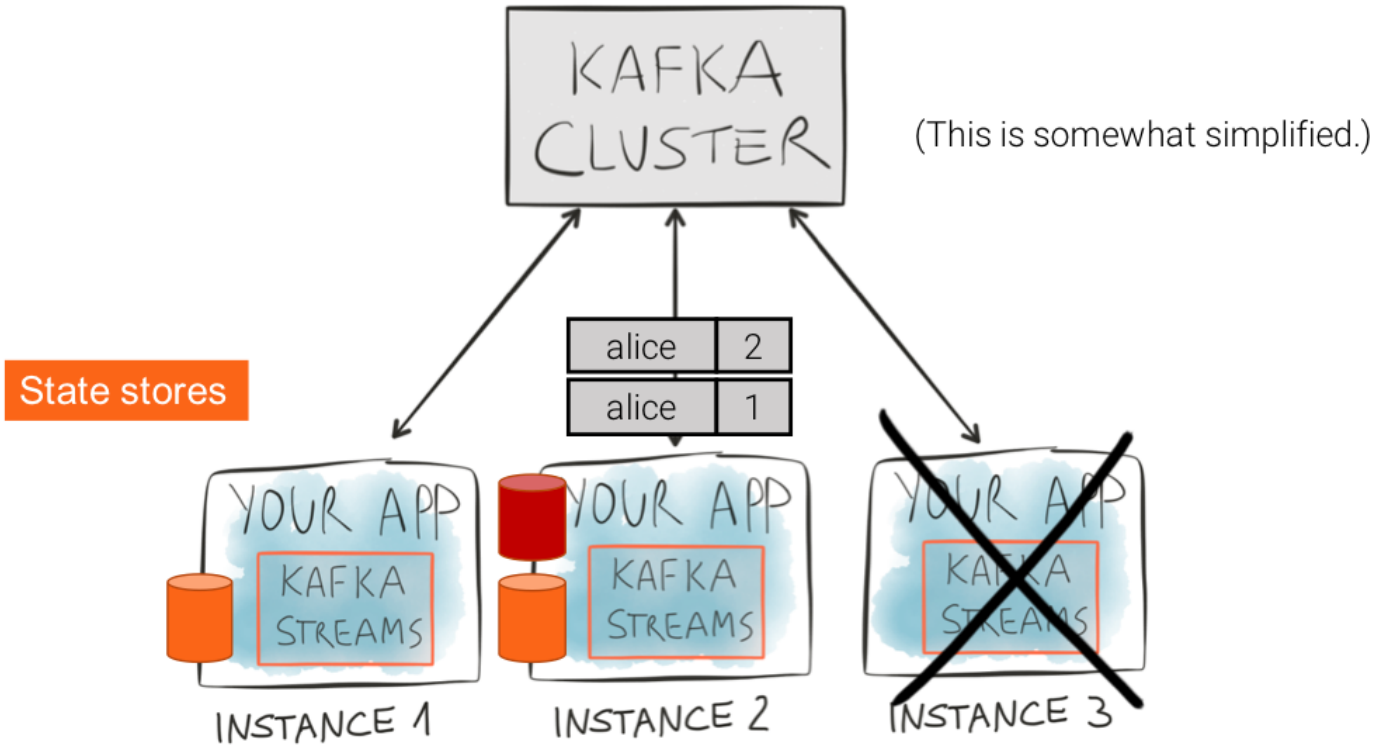
State Management (2)



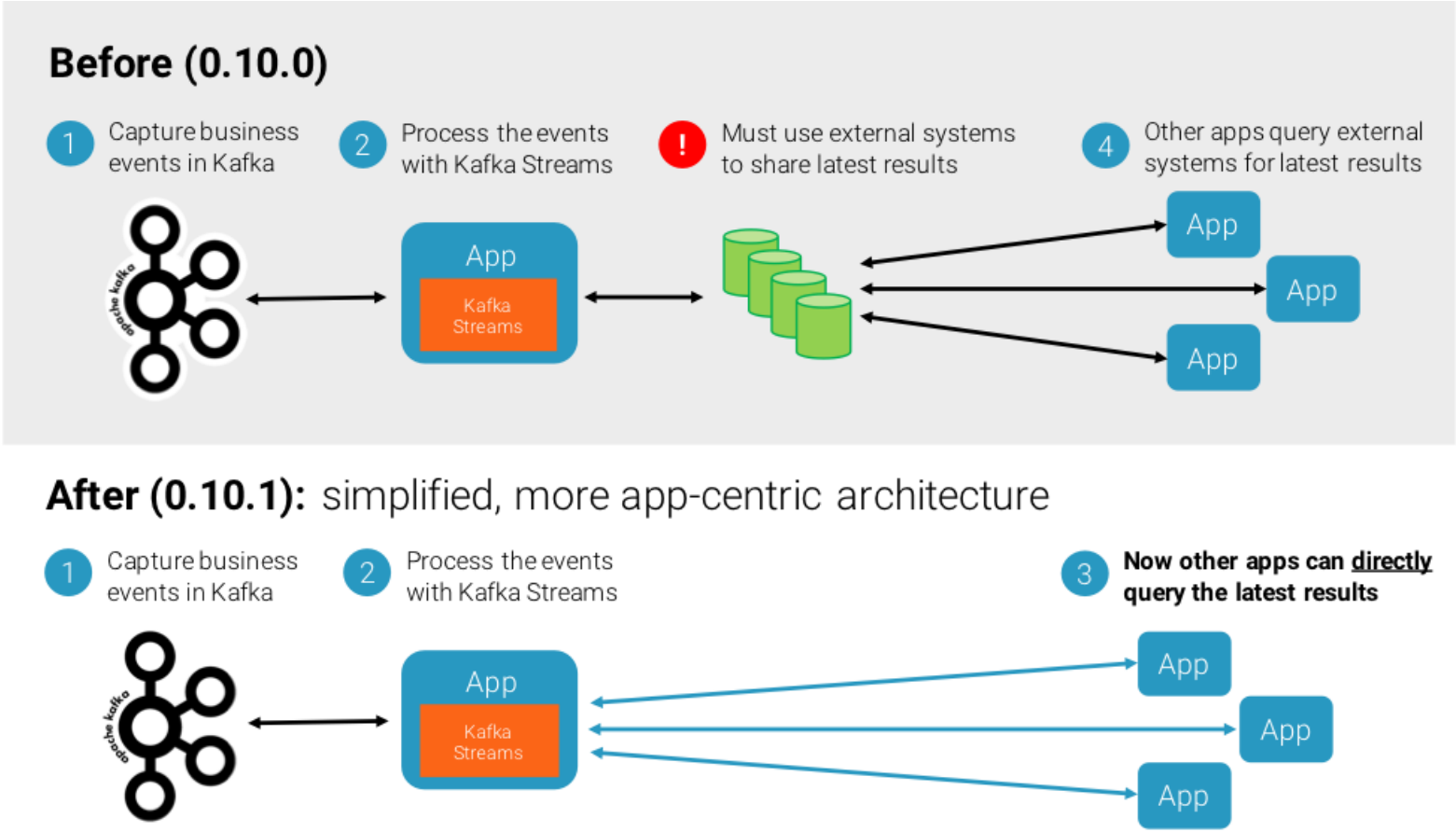
State Management (3)



State Management (4)



Interactive Queries



Kafka Streams

- *Kafka Streams Basics*
- *Kafka Streams Basic Concepts*
- *Key Features of Kafka Streams*
- **Kafka Streams APIs**
- *Hands-On Exercise: Writing a Kafka Streams Application*
- *Chapter Review*

API Option 1: Processor API

- **Flexible, but requires more manual work**
- **May appeal to people who are familiar with Storm, Samza etc**
  - Or for people who require functionality that is not yet available in the DSL

```
class PrintToConsoleProcessor
    implements Processor<K, V> {

  @Override
  public void init(ProcessorContext context) {}

  @Override
  void process(K key, V value) {
    System.out.println("Received record with " +
      "key=" + key + " and value=" + value);
  }

  @Override
  void punctuate(long timestamp) {}

  @Override
  void close() {}
}
```

API Option 2: Kafka Streams DSL

- **The preferred API for most use-cases**
- **Will appeal to people who are used to Spark, Flink,...**
- **This is the API we'll concentrate on in the tutorial**

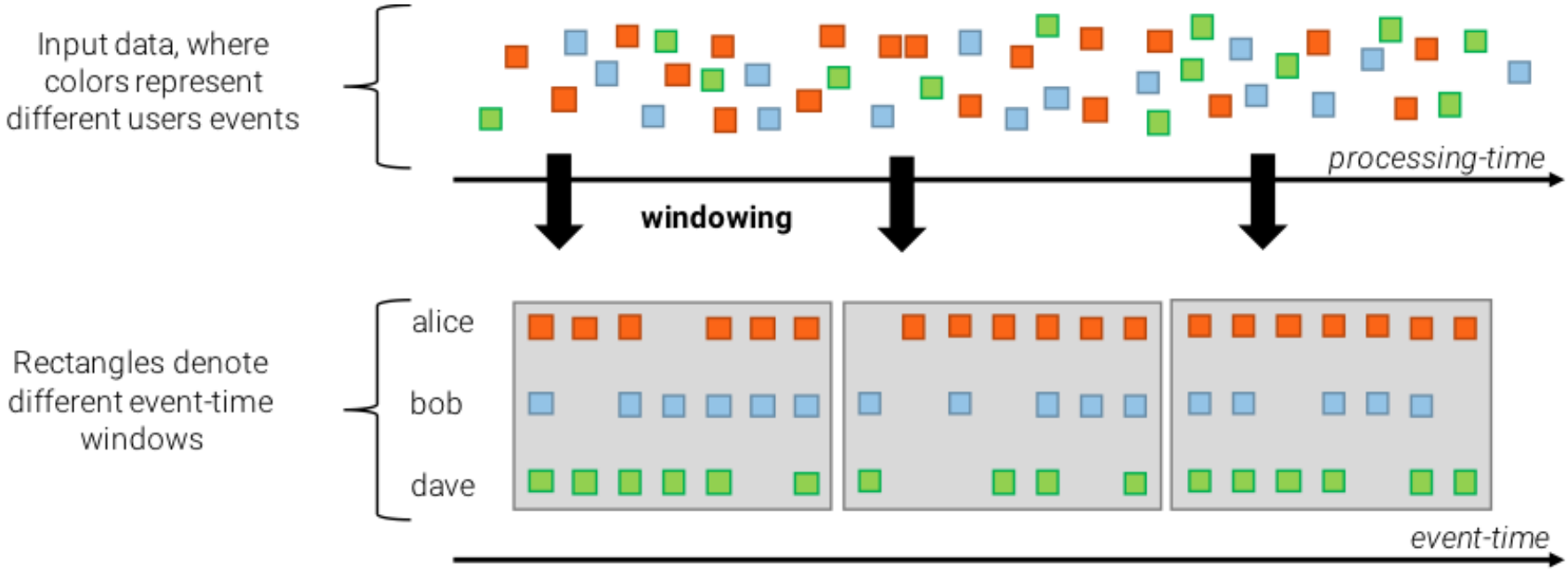
```
KStream<Integer, Integer> input =
    builder.stream("numbers-topic");

// Stateless computation
KStream<Integer, Integer> doubled =
    input.mapValues(v -> v * 2);

// Stateful computation
KTable<Integer, Integer> sumOfOdds = input
    .filter((k,v) -> v % 2 != 0)
    .selectKey((k, v) -> 1)
    .reduce((v1, v2) -> v1 + v2, "sum-of-odds");
```

Windowing, Joins, and Aggregations

- **Kafka Streams allows us to *window* the stream of data by time**
  - To divide it up into ‘time buckets’
- **We can join, or merge, two streams together based on their keys**
  - Typically we do this on windows of data
- **We can aggregate records**
  - Combine multiple input records together in some way into a single output record
  - Examples: sum, count
  - Again, this is usually done on a windowed basis

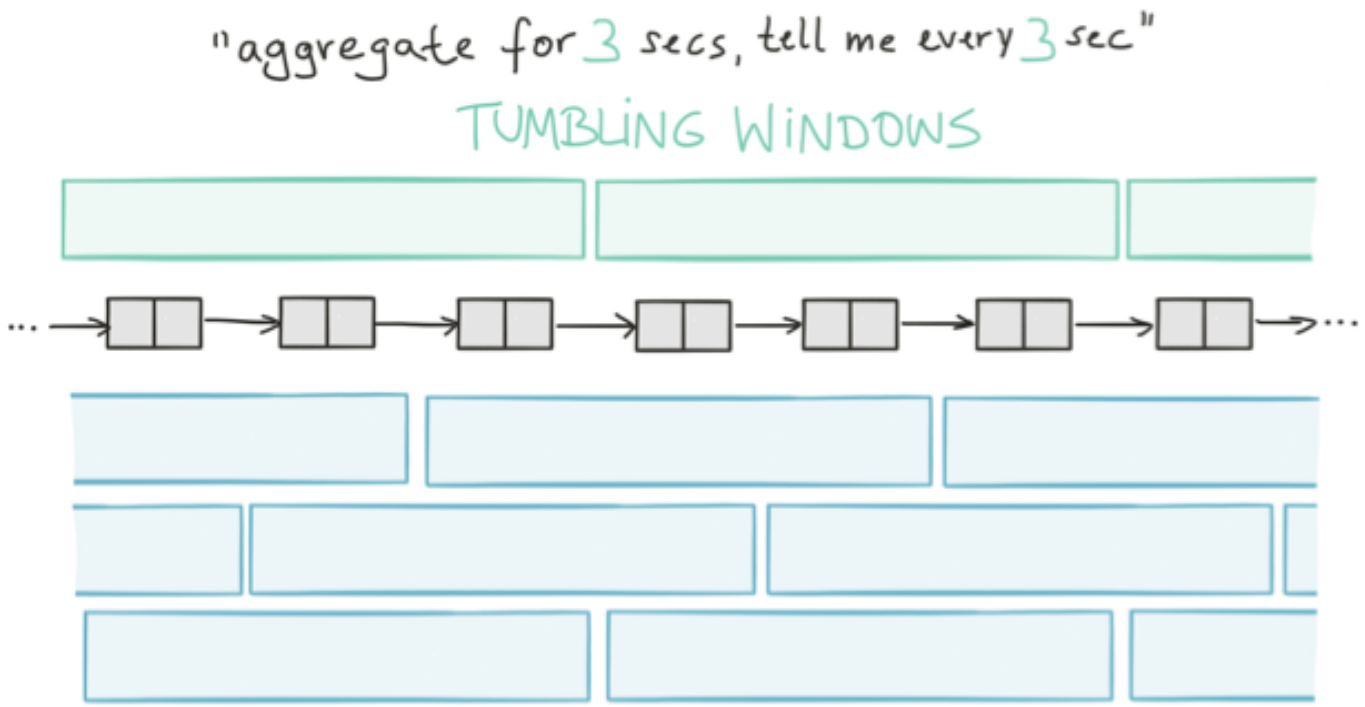
Windowing: Example

- **Windowing: Group events in a stream using time-based windows**
- **Use case examples:**
  - Time-based analysis of ad impressions (“number of ads clicked in the past hour”)
  - Monitoring statistics of telemetry data (“1min/5min/15min averages”)



Tumbling and Hopping Windows

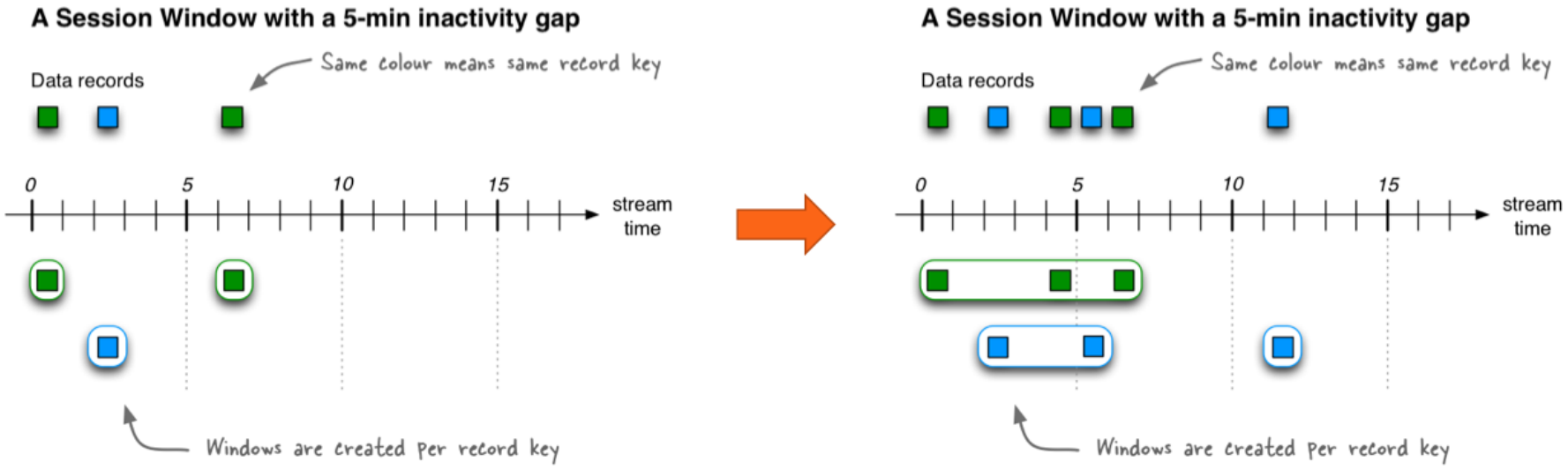
```
TimeWindows.of(3000)
```




```
TimeWindows.of(3000).advanceBy(1000)
```

New in Kafka 10.2.0: Session Windows

- **Session: periods of activity separated by a defined gap of inactivity**
- **Use for behavior analysis**



Configuring the Application

- **Kafka Streams configuration is specified with a `StreamsConfig` instance**
- **This is typically created from a `java.util.Properties` instance**
- **Example:**

```
Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-example");
streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");

// Continue to specify configuration options
streamsConfiguration.put(..., ...);
```

Serializers and Deserializers (Serdes)

- **Each Kafka Streams application must provide serdes (serializers and deserializers) for the data types of the keys and values of the records**
  - These will be used by some of the operations which transform the data
- **Set default serdes in the configuration of your application**
  - These can be overridden in individual methods by specifying explicit serdes
- **Configuration example:**

```
streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().
getName());
streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass().
getName());
```

Available Serdes

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

Data type	Serde
byte[]	**`Serdes.ByteArray()`**, **`Serdes.Bytes()`** (**`Bytes`** wraps Java's **`byte[]`** and supports equality and ordering semantics)
ByteBuffer	**`Serdes.ByteBuffer()`**
Double	**`Serdes.Double()`**
Integer	**`Serdes.Integer()`**
Long	**`Serdes.Long()`**
String	**`Serdes.String()`**

- **Confluent also provides `GenericAvroSerde` and `SpecificAvroSerde` as examples**
  - See `https://github.com/confluentinc/examples/tree/kafka-0.10.0.0-cp-3.0.0/kafka-streams/src/main/java/io/confluent/examples/streams/utils`

Creating the Processing Topology

- **Create a `KStream` or `KTable` object from one or more Kafka Topics using `KStreamBuilder` or `KTableBuilder`**
  - For `KTableBuilder` only a single Topic can be specified
- **Example:**

```
KStreamBuilder builder = new KStreamBuilder();

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

Transforming a Stream

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

Some Stateless Transformation Operations (1)

- **Examples of stateless transformation operations:**
  - `filter`
    - Creates a new `KStream` containing only records from the previous `KStream` which meet some specified criteria
  - `map`
    - Creates a new `KStream` by transforming each element in the current stream into a different element in the new stream
  - `mapValues`
    - Creates a new `KStream` by transforming the value of each element in the current stream into a different element in the new stream

Some Stateless Transformation Operations (2)

- **Examples of stateless transformation operations (cont'd):**

- `flatMap`
  - Creates a new `KStream` by transforming each element in the current stream into zero or more different elements in the new stream
- `flatMapValues`
  - Creates a new `KStream` by transforming the value of each element in the current stream into zero or more different elements in the new stream

Some Stateful Transformation Operations

- **Examples of stateful transformation operations:**
  - `countByKey`
    - Counts the number of instances of each key in the stream; results in a new, ever-updating `KTable`
  - `reduceByKey`
    - Combines values of the stream using a supplied Reducer into a new, ever-updating `KTable`
- **For a full list of operations, see the JavaDocs at `http://http://docs.confluent.io/3.0.0/streams/javadocs/index.html`**

Writing Streams Back to Kafka

- **Streams can be written to Kafka topics using the `to` method**
  - Example:

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

- **We often want to write to a topic but then continue to process the data**
  - Do this using the `through` method

```
myNewStream.through("NewTopic").flatMap(...)...;
```

Printing A Stream's Contents

- **It is sometimes useful to be able to see what a stream contains**
  - Especially when testing and debugging
- **`print()` writes the contents of the stream to `System.out`**

Running the Application

- **To start processing the stream, create a `KafkaStreams` object**
  - Configure it with your `KStreamBuilder` or `KTableBuilder` and your configuration
- **Then call the `start()` method**
- **Example:**

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

- **If you need to stop the application, call `streams.close()`**

A Simple Kafka Streams Example (1)

```
1 public class SimpleStreamsExample {
2
3
4   public static void main(String[] args) throws Exception {
5     Properties streamsConfiguration = new Properties();
6     // Give the Streams application a unique name.  The name must be unique in the Kafka cluster
7     streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-streams-example");
8     streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
9     streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");
10    streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
11    // Specify default (de)serializers for record keys and for record values.
12    streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.ByteArray().getClass
().getName());
13    streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass
().getName());
```

A Simple Kafka Streams Example (2)

```
    KStreamBuilder builder = new KStreamBuilder();

    // Read the input Kafka topic into a KStream instance.
    KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

    // Convert to upper case (:: is Java 8 syntax)
    KStream<byte[], String> uppercasedWithMapValues = textLines.mapValues(String::toUpperCase);

    // Write the results to a new Kafka topic called "UppercasedTextLinesTopic".
    uppercasedWithMapValues.to("UppercasedTextLinesTopic");

    KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
    streams.start();
  }
}
```

A More Complex Kafka Streams Application (1)

```
1 public class WordCountLambdaExample {
2   public static void main(String[] args) throws Exception {
3     Properties streamsConfiguration = new Properties();
4     // Application name must be unique in the Kafka cluster
5     streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-lambda-example");
6
7     streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
8     streamsConfiguration.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "zookeeper1:2181");
9     // Specify default (de)serializers for record keys and for record values.
10    streamsConfiguration.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass
().getName());
11    streamsConfiguration.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass
().getName());
12
13    KStreamBuilder builder = new KStreamBuilder();
```

A More Complex Kafka Streams Application (2)

```
    // Construct a KStream from the input topic "TextLinesTopic"
    KStream<String, String> textLines = builder.stream("TextLinesTopic");

    KStream<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    .map((key, word) -> new KeyValue<>(word, word))
      // Count the occurrences of each word (record key).
      // This will change the stream type from KStream<String, String> to
      // KTable<String, Long> (word -> count).
    .countByKey("Counts")
      // Convert the KTable<String, Long> into a `KStream<String, Long>`.
    .toStream();

    // Write the `KStream<String, Long>` to the output topic.
    wordCounts.to(Serdes.String(), Serdes.Long(), "WordsWithCountsTopic");

    // Now that we have finished the definition of the processing topology we can actually run
    // it via `start()`.  The Streams application as a whole can be launched just like any
    // normal Java application that has a `main()` method.
    KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
    streams.start();
  }
}
```

Processing Guarantees

- **Kafka streams supports *at-least-once* processing**
- **With no failures, it will process data exactly once**
- **If a machine fails, it is possible that some records may be processed more than once**
  - Whether this is acceptable or not depends on the use-case
- **Exactly-once processing semantics will be supported in the next major release of Kafka**

More Kafka Streams Features

- **We have only scratched the surface of Kafka Streams**
- **Check the documentation to learn about processing windowed tables, joining tables, joining streams and tables...**

Kafka Streams

- *Kafka Streams Basics*
- *Kafka Streams Basic Concepts*
- *Key Features of Kafka Streams*
- *Kafka Streams APIs*
- **Hands-On Exercise: Writing a Kafka Streams Application**
- *Chapter Review*

Hands-On Exercise: Writing a Kafka Streams Application

- **In this Hands-On Exercise, you will write a Kafka Streams application**
- **Please refer to the Hands-On Exercise Manual**

Kafka Streams

- *Kafka Streams Basics*
- *Kafka Streams Basic Concepts*
- *Key Features of Kafka Streams*
- *Kafka Streams APIs*
- *Hands-On Exercise: Writing a Kafka Streams Application*
- **Chapter Review**

Chapter Review

- **Kafka Streams provides a way of writing streaming data applications that is**
  - Simple
  - Powerful
  - Flexible
  - Scalable

Conclusion

Chapter 6

Course Contents

Kafka: A Complete Streaming Data Platform

- **Kafka Connect ingests and exports data, with no code required**
- **Kafka Streams provides powerful, flexible, scalable stream data processing**
- **Together, they give us a way to import data from external systems, manipulate or modify that data *in real time*, and then export it to destination systems for storage or further processing**

Thanks!

- **Thank you for coming!**
- **If you have feedback, suggestions, or questions, please e-mail `ian@confluent.io`**
- **Obligatory commercial messages:**
  - Confluent offers Developer and Operations training for Kafka. See `http://confluent.io/training`
  - We're hiring! See `http://confluent.io/careers`
    - Especially: We're hiring instructors!


kafka summit
Discount code: kafstrata
Special Strata Attendee discount code = 25% off
www.kafka-summit.org
Kafka Summit New York: May 8
Kafka Summit San Francisco: August 28
Presented by confluent