



Apache Kafka is a trademark of the Apache Project Foundation

Kafka for Consumers

Kafka Tutorial
What is Kafka?
Why is Kafka important?
Kafka architecture and design
Kafka Universe
Kafka Schemas
Java Consumer examples



Kafka growing

Why Kafka?

Kafka adoption is on the rise
but why

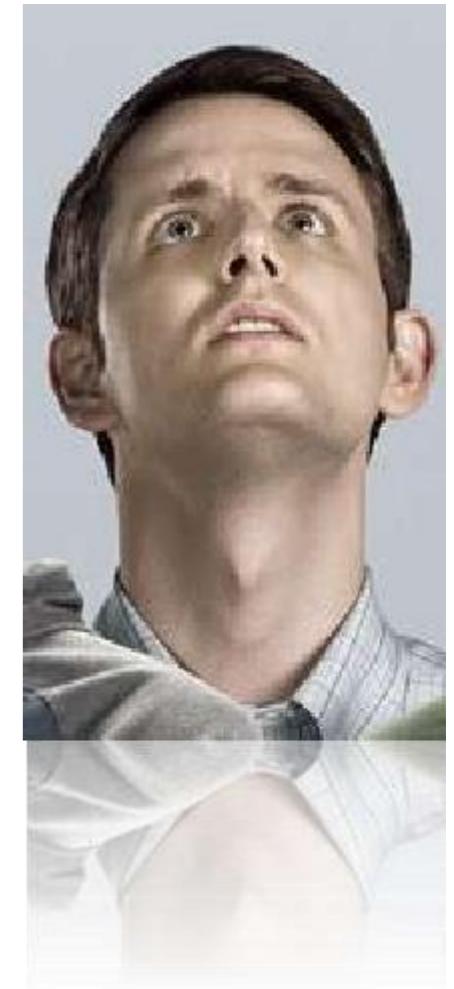
[What is Kafka?](#)

Kafka growth exploding



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

4
commas!



Why Kafka is Needed?



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

Why is Kafka needed? 2



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



Kafka Use Cases

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



Who uses Kafka?

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

Why is Kafka Popular?



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

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

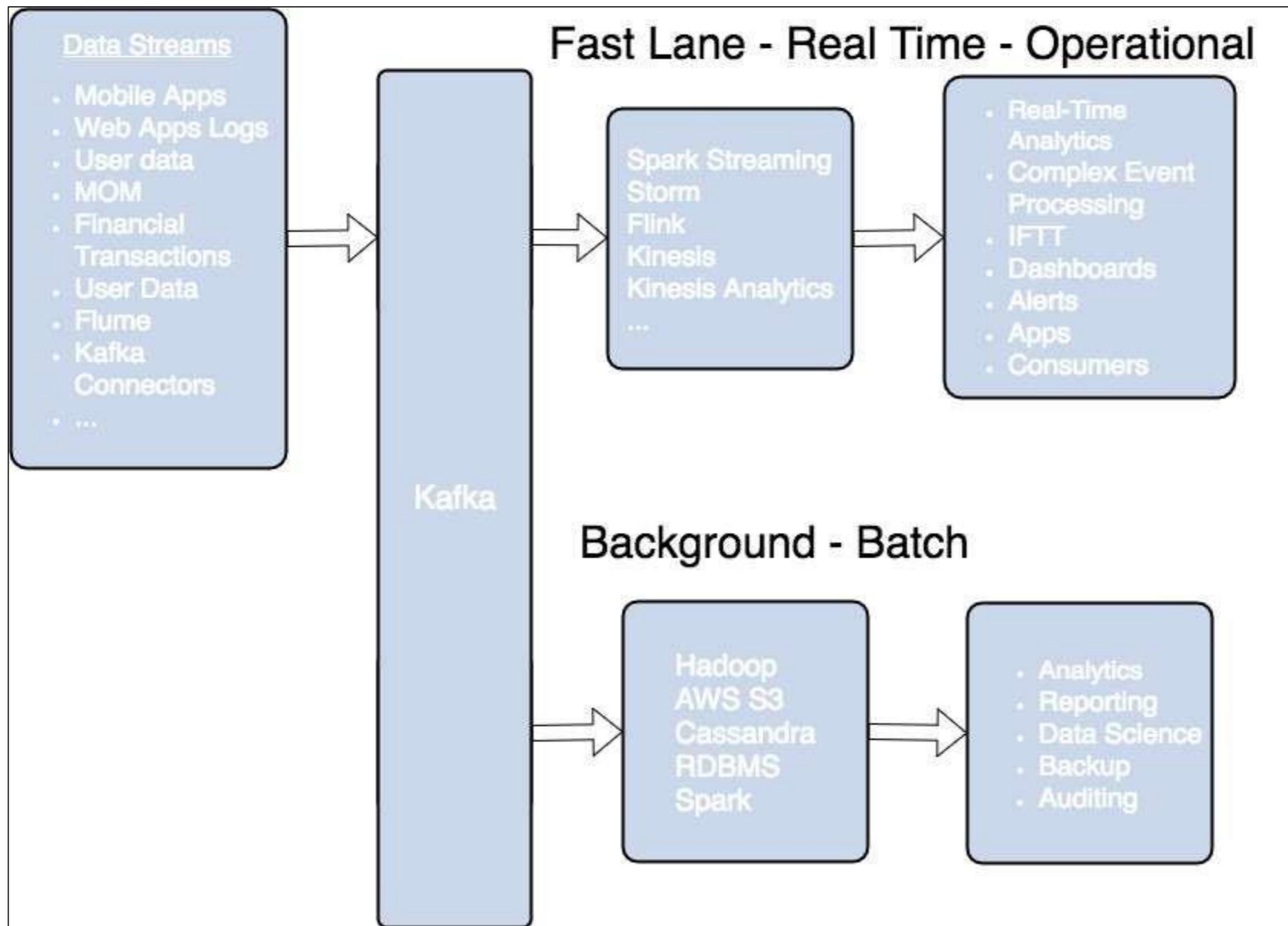
Why is Kafka so fast?



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



Kafka Streaming Architecture





Why Kafka Review



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

Learning Goals

- Summarize the motivation behind Kafka
- Define core components of Kafka
- Summarize the life of a message in Kafka
- Build a simple producer and consumer application

Prerequisites

- Basic understanding of big data concepts
- Basic understanding of the kafka platform
- Basic understanding of application development principles

Learning Goals

- 1.1: Summarize the motivation behind Kafka
- 1.2: Explain what makes Kafka different
- 1.3: Apply Kafka to common use cases

Learning Goals

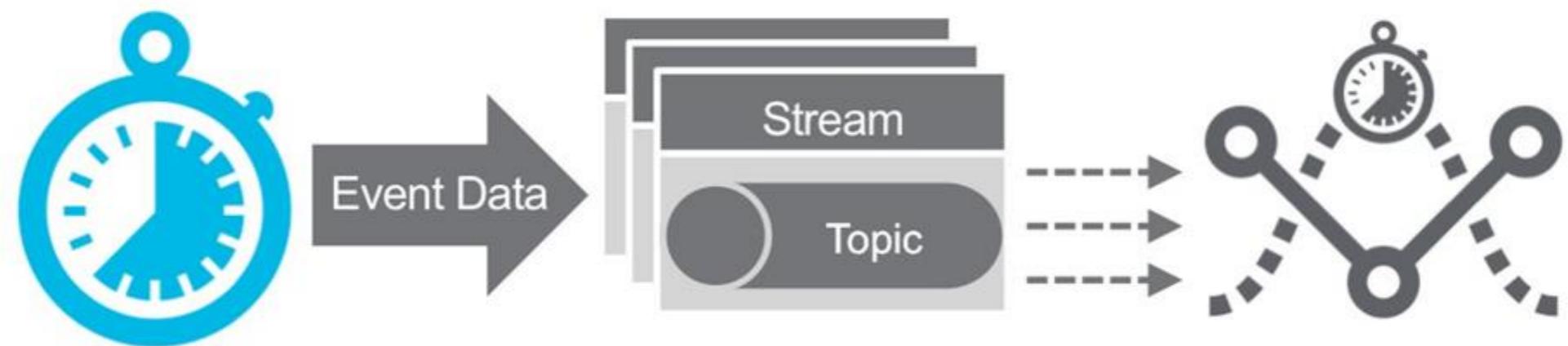
1.1: Summarize the motivation behind Kafka

1.2: Explain what makes Kafka different

1.3: Apply Kafka to common use cases

Why Kafka

Many Big Data sources are Evented Oriented



Trigger Events:

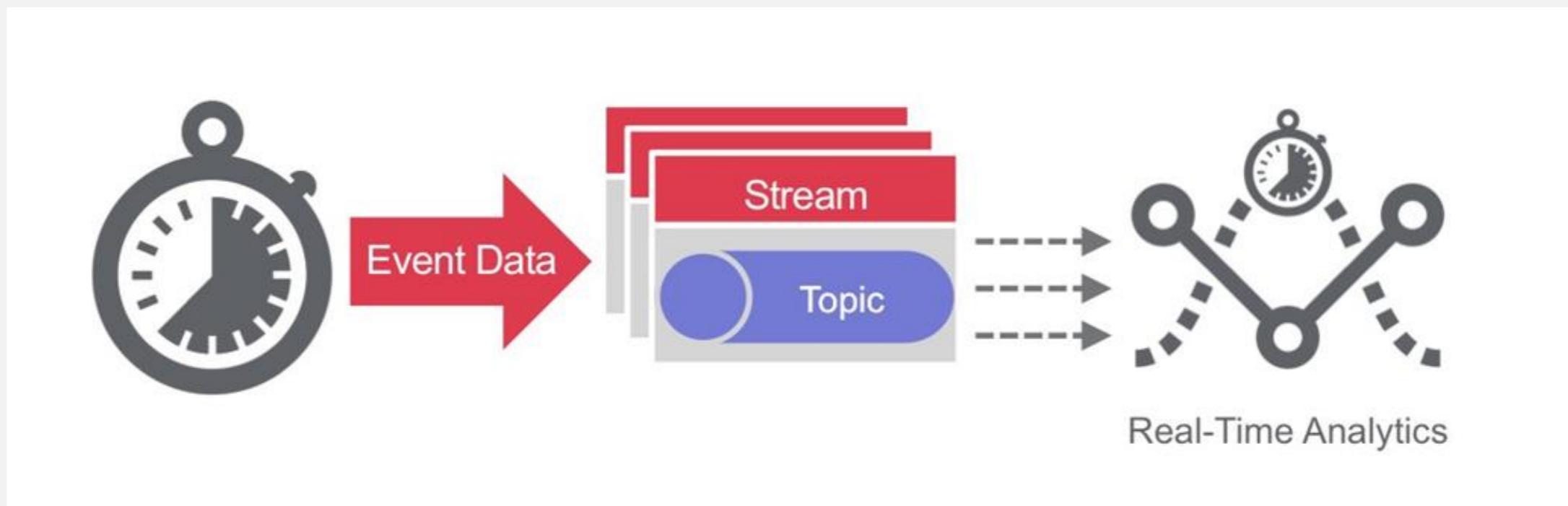
- Stock Prices
- User Activity
- Sensor Data

Real-Time Analytics

Why Kafka

Today's Applications need to:

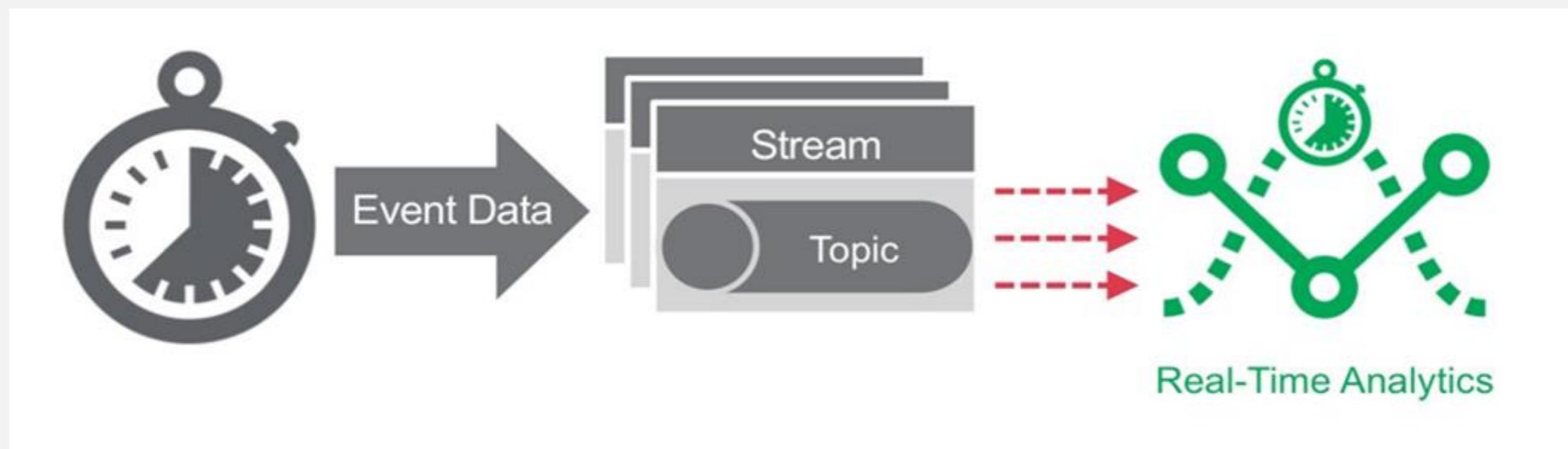
- process high-velocity data as soon as possible
- handle high-volume workloads



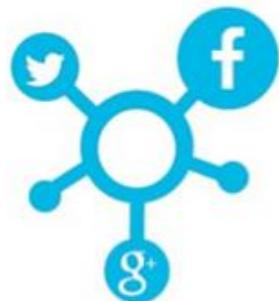
Why Kafka

Today's Applications need to:

- process high-velocity data as soon as possible
- handle high-volume workloads
- provide real-time results with extremely low latency



Diverse Data Sources



Social Media Feeds



Financial Transactions



Geotagging Data



Internet of Things



Web Browser Clickstreams



Application Metrics

Diverse Data Sources



Social Media Feeds



Financial Transactions



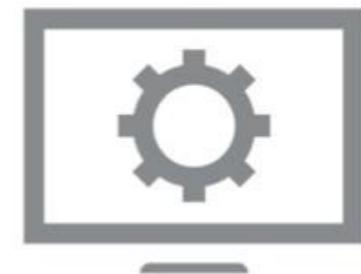
Geotagging Data



Internet of Things



Web Browser Clickstreams



Application Metrics

Diverse Data Sources



Social Media Feeds



Financial Transactions



Geotagging Data



Internet of Things



Web Browser Clickstreams



Application Metrics

Diverse Data Sources



Social Media Feeds



Financial Transactions



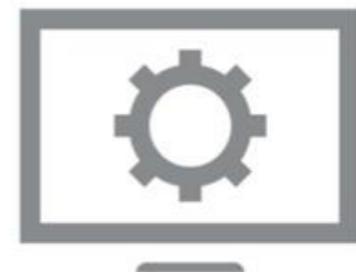
Geotagging Data



Internet of Things



Web Browser Clickstreams



Application Metrics

Diverse Data Sources



Social Media Feeds



Financial Transactions



Geotagging Data



Internet of Things

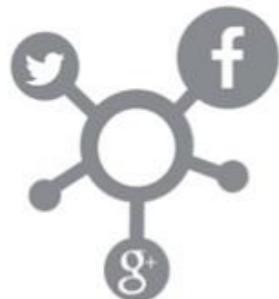


Web Browser Clickstreams



Application Metrics

Diverse Data Sources



Social Media Feeds



Financial Transactions



Geotagging Data



Internet of Things



Web Browser Clickstreams



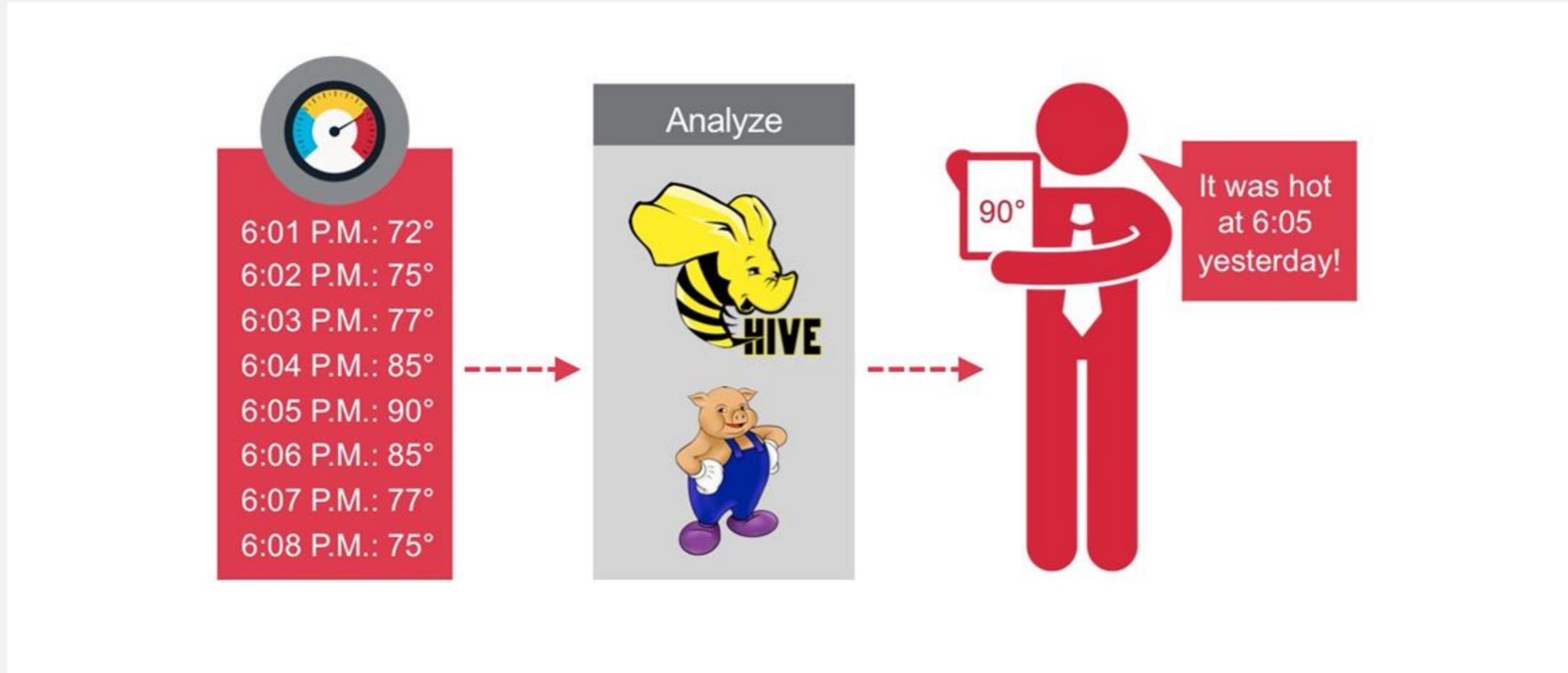
Application Metrics

Analyze Data

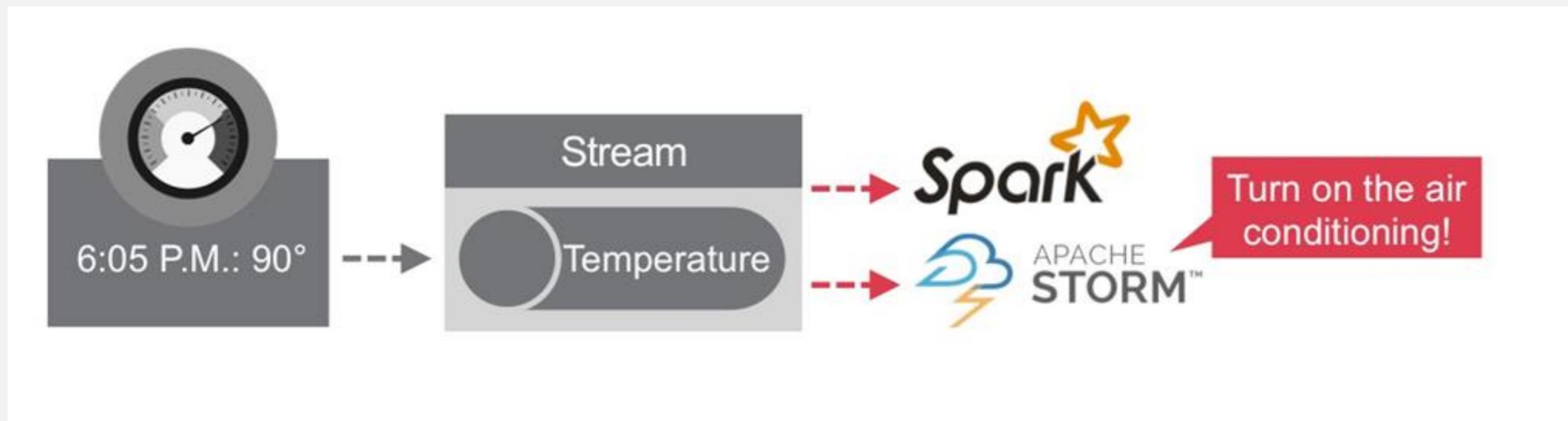
- What if you need to analyze data as it arrives?



Batch Processing with HDFS



Event Processing with Kafka



Organize Data

- What if you need to Organize data as it arrives?

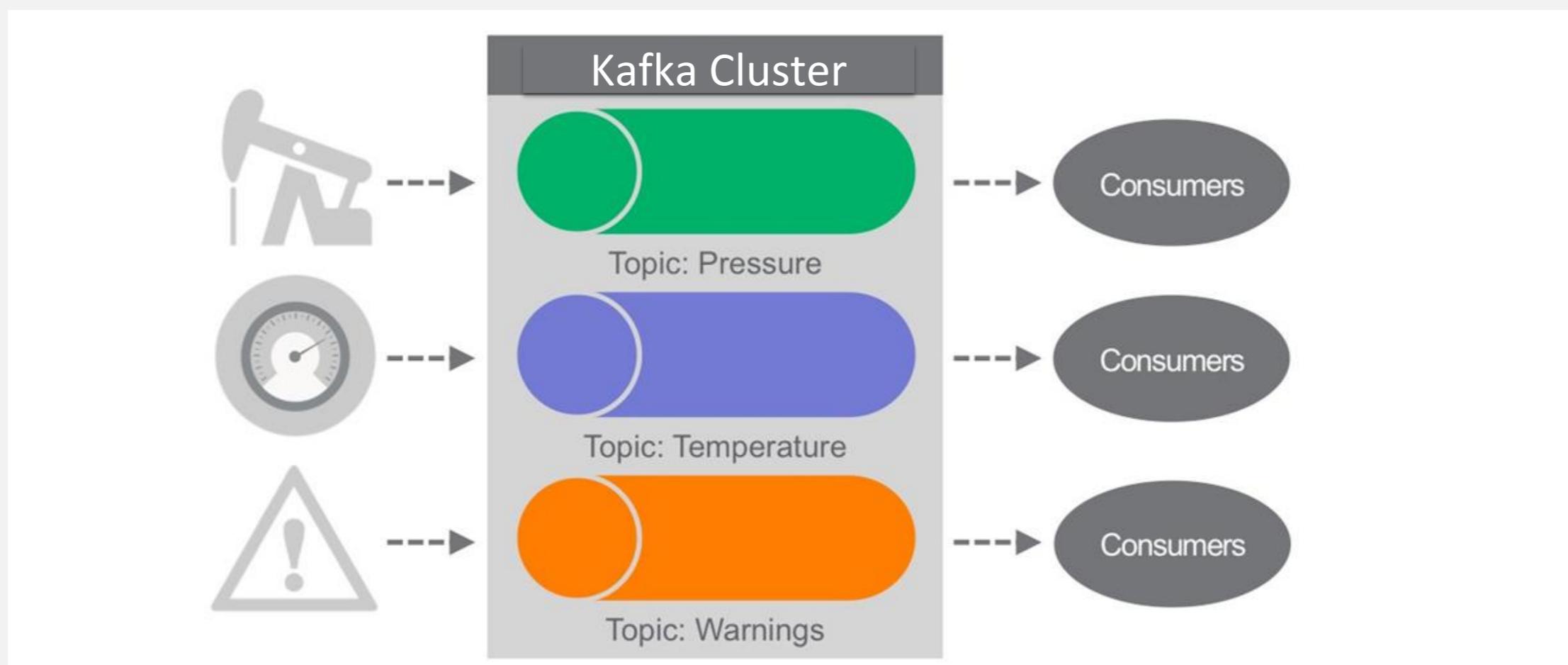


Many Sources of Data



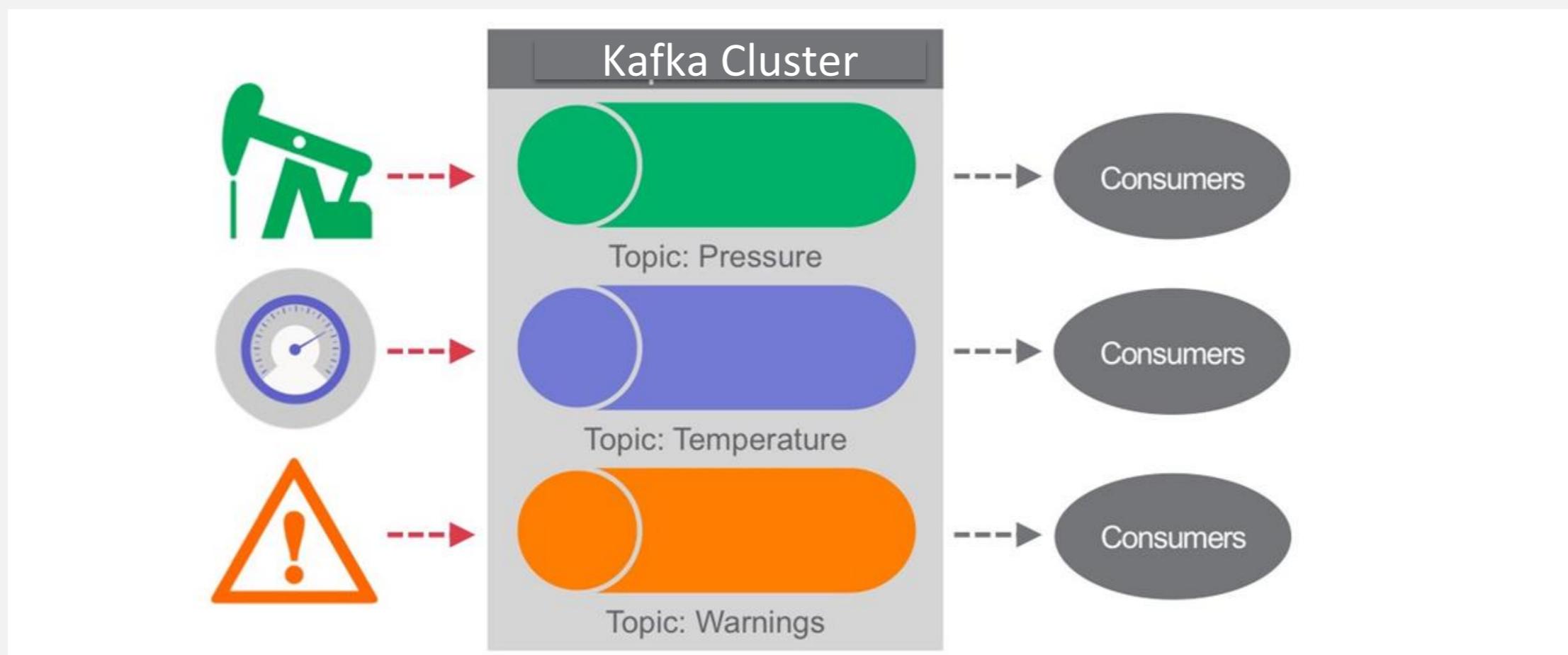
Organize Data into Topics with Kafka

- Topics organize events into categories



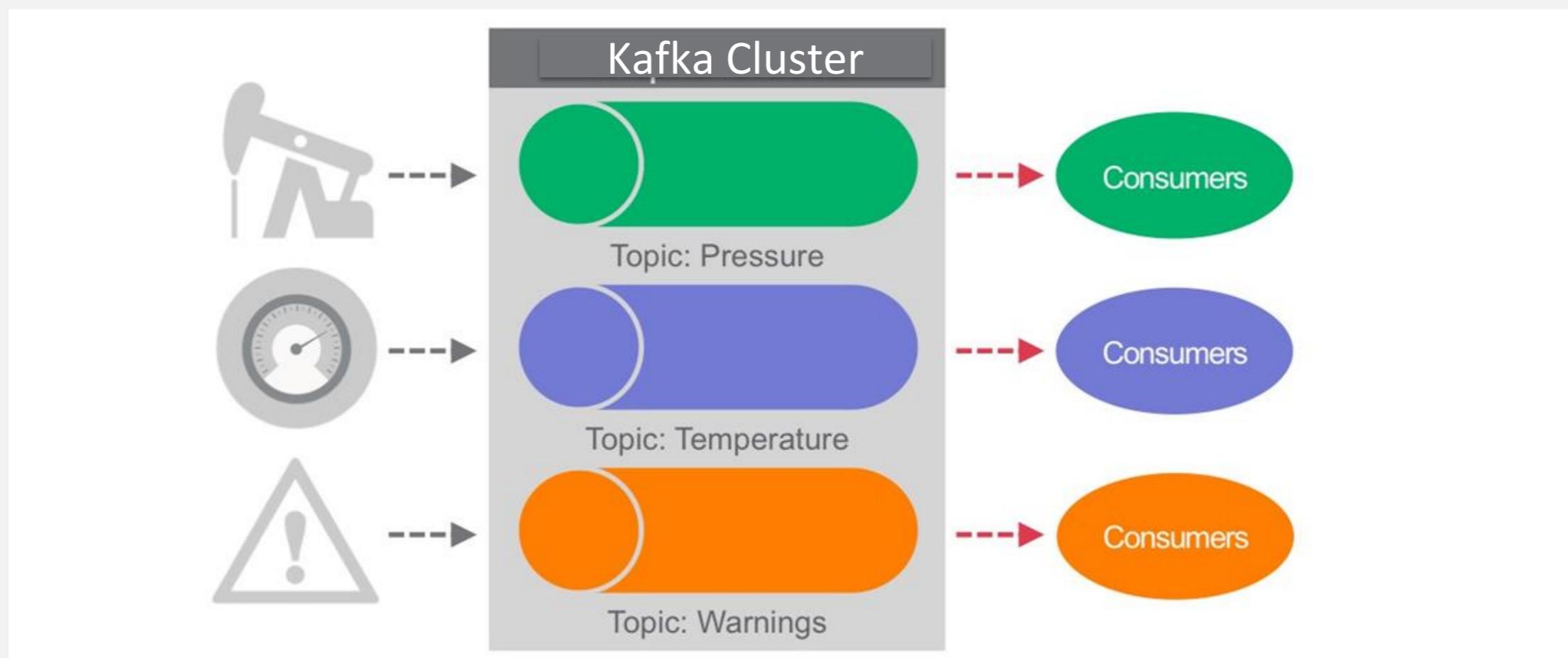
Organize Data into Topics with Kafka

- Producer publish to topics



Organize Data into Topics with Kafka

- Consumer subscribe to topics

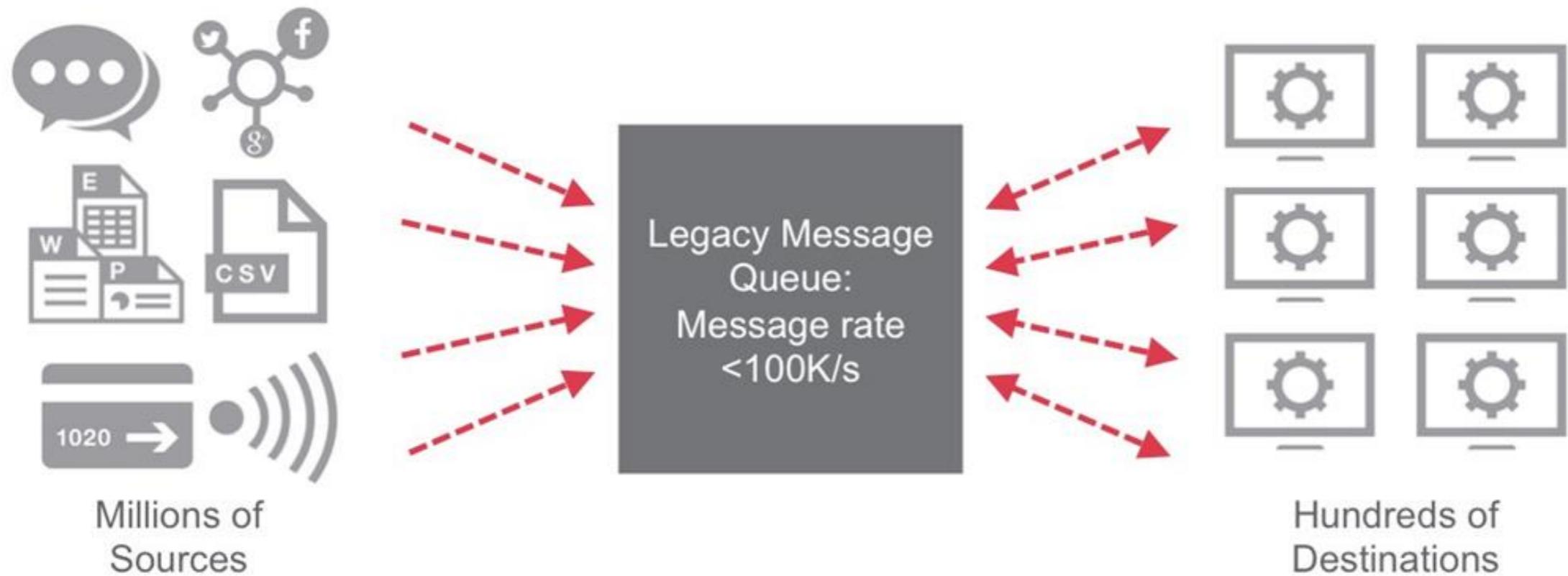


Process High Volume of Data

- What if you need to process a high volume of data as it arrives?



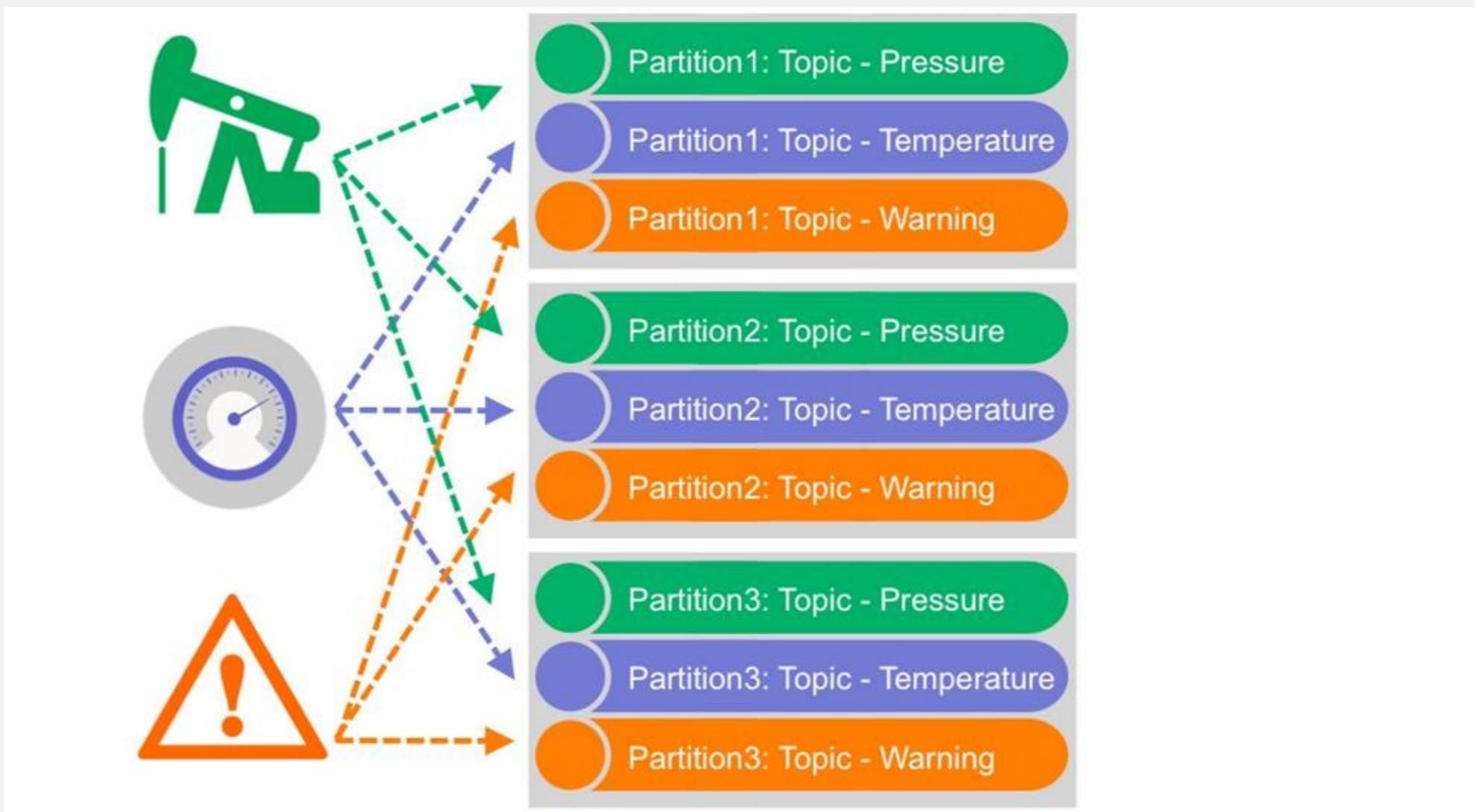
Legacy Messaging



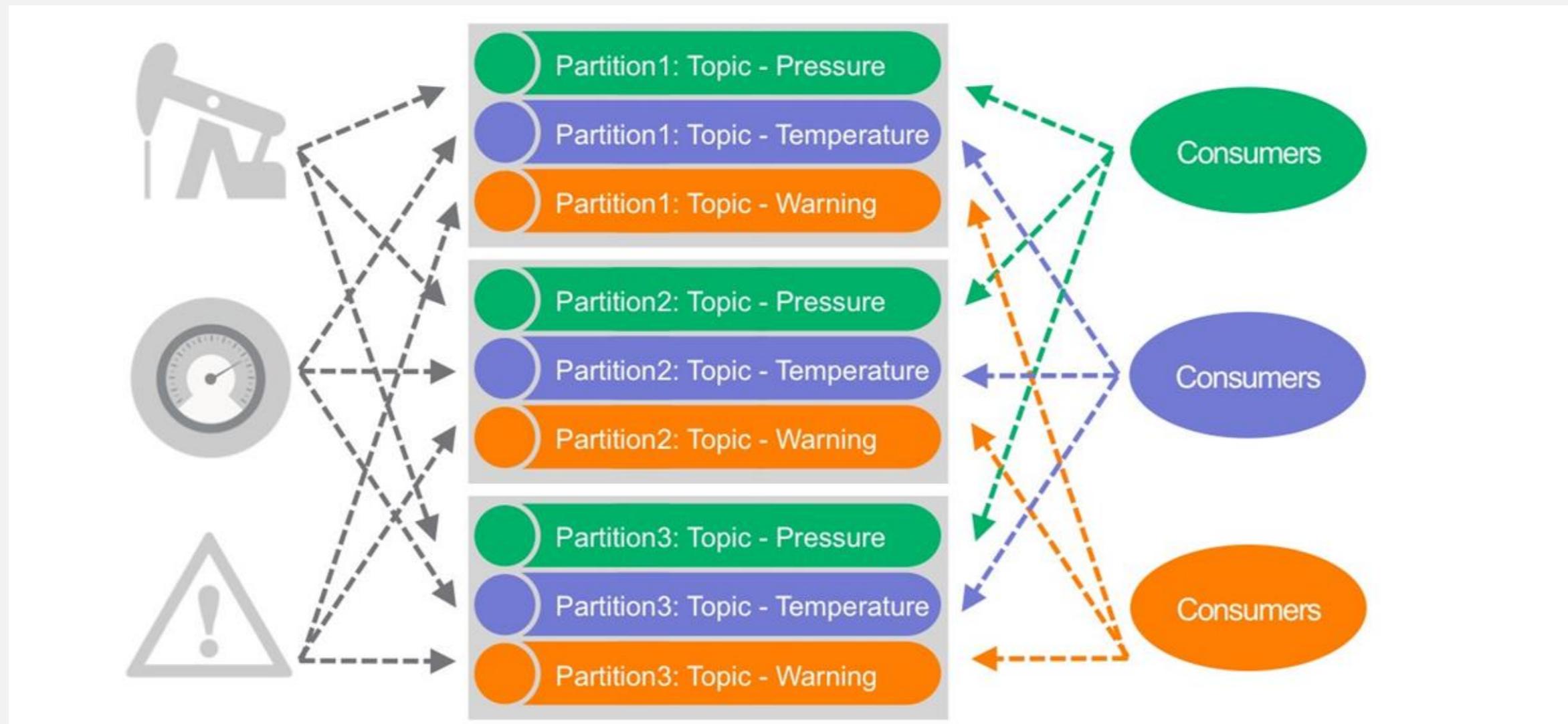
Scalable Messaging with Kafka



Scalable Messaging with Kafka



Scalable Messaging with Kafka

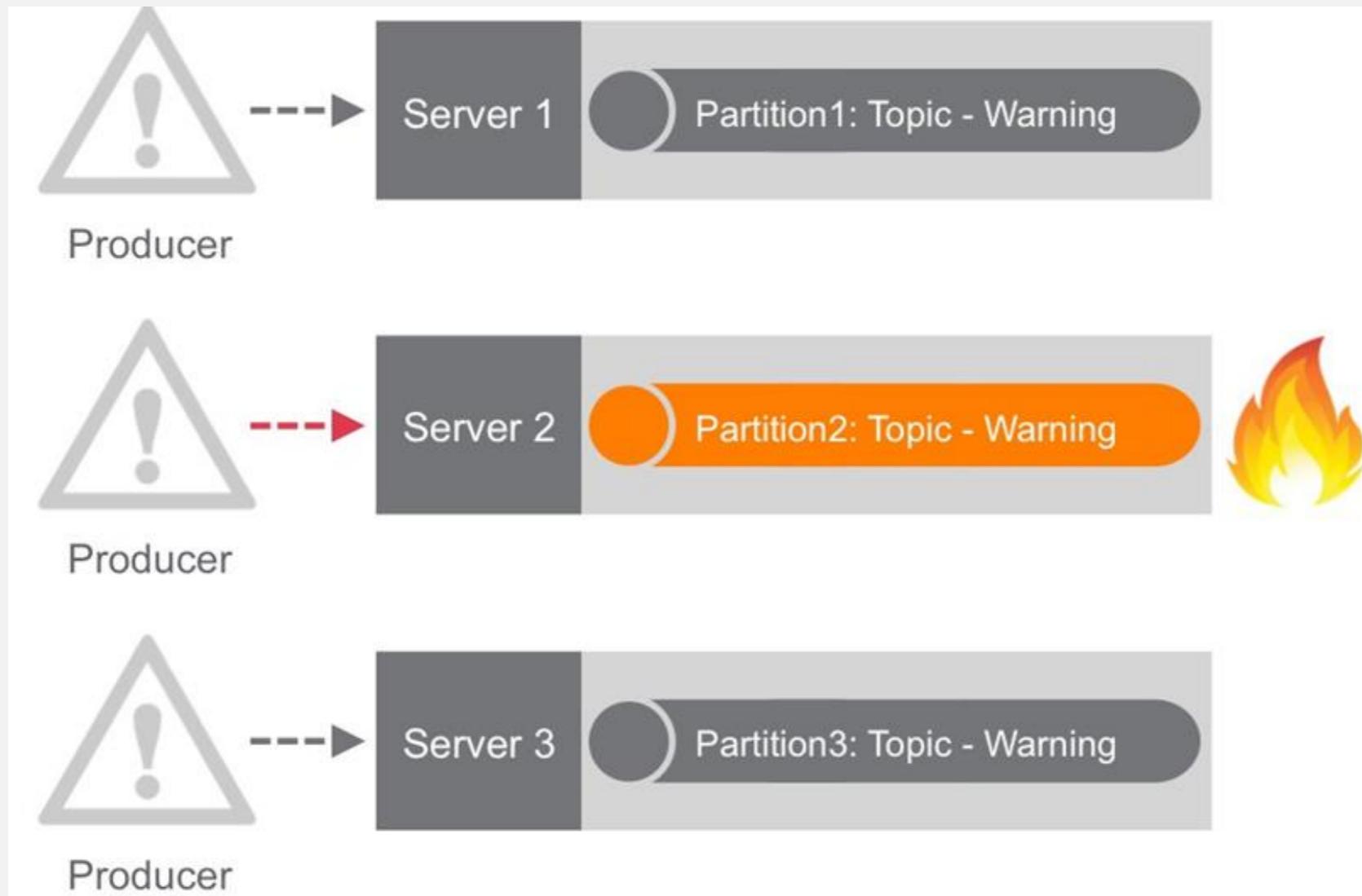


Message Recovery

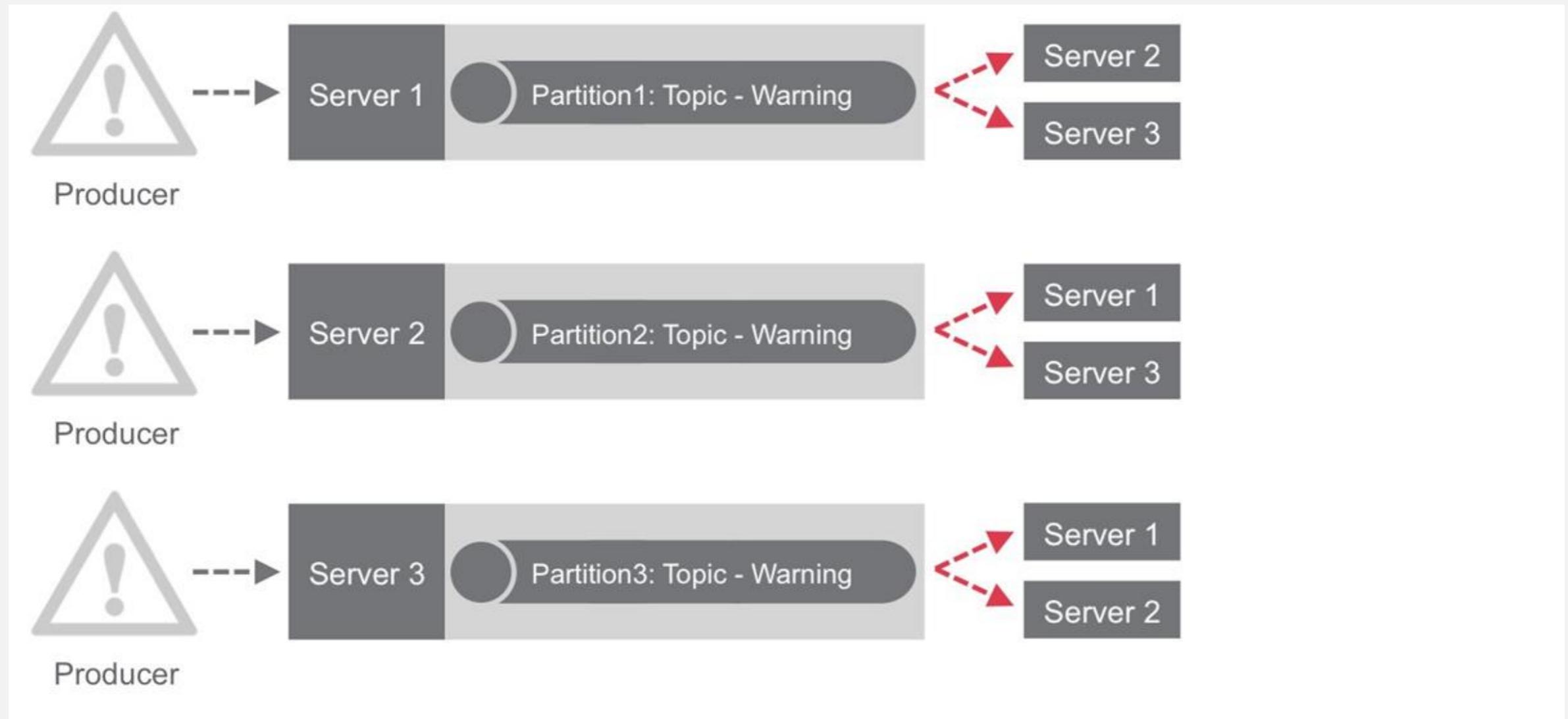
- What if you need to recover messages in case of server failure?



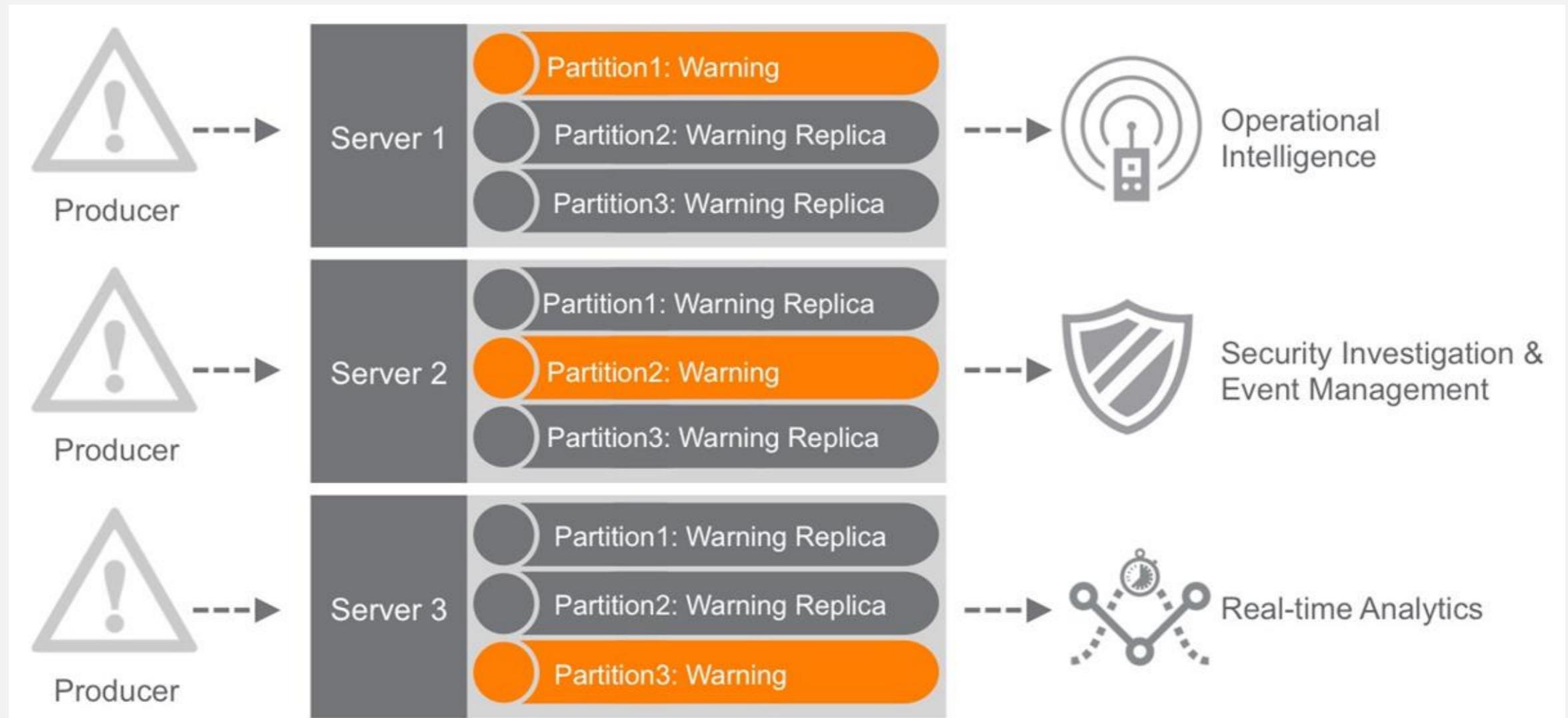
Lack of Replication



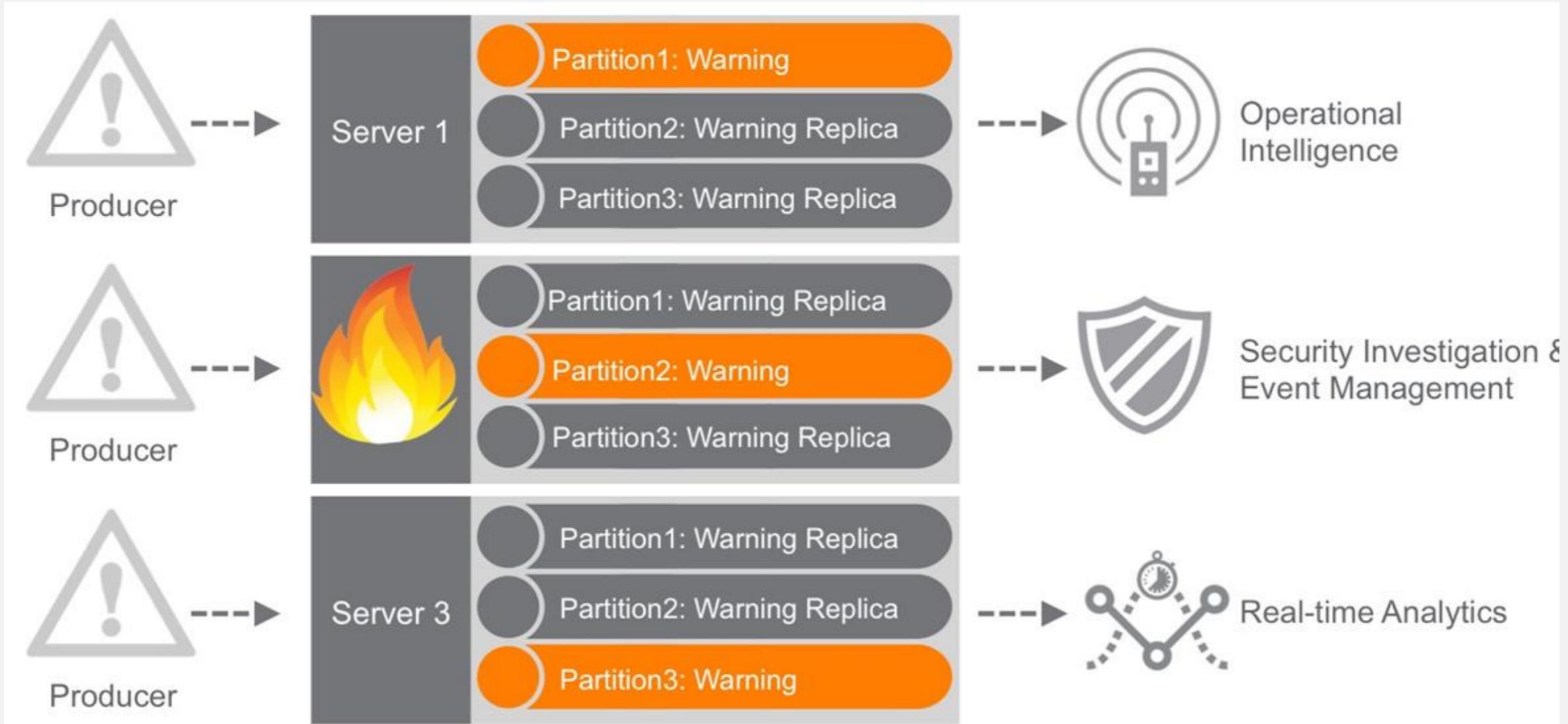
Partitions are Replicated for Faut Tolerance



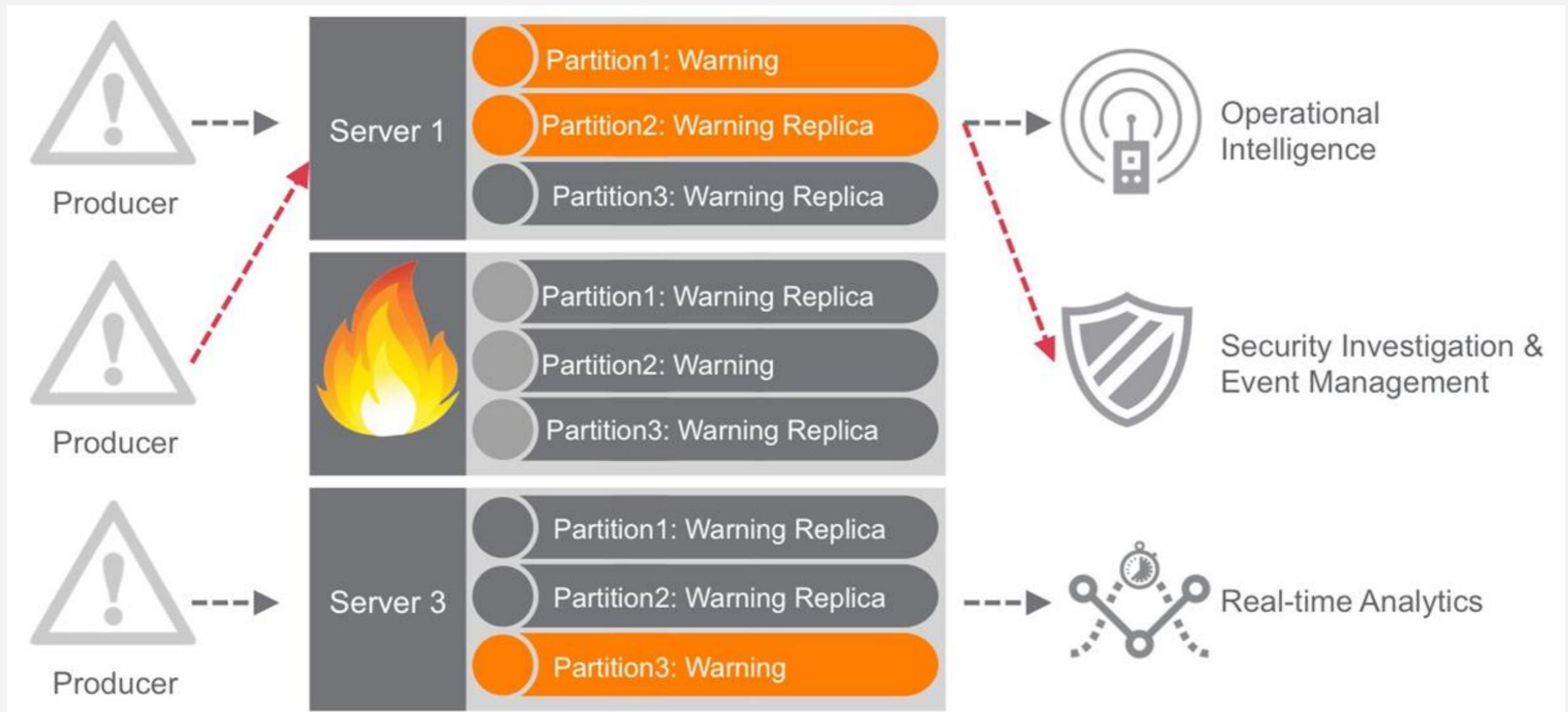
Partitions are Replicated for Fault Tolerance



Partitions are Replicated for Fault Tolerance



Partitions are Replicated for Fault Tolerance



Real-time Access

- What if you need real-time access to live data distributed across multiple clusters and multiple data centers?



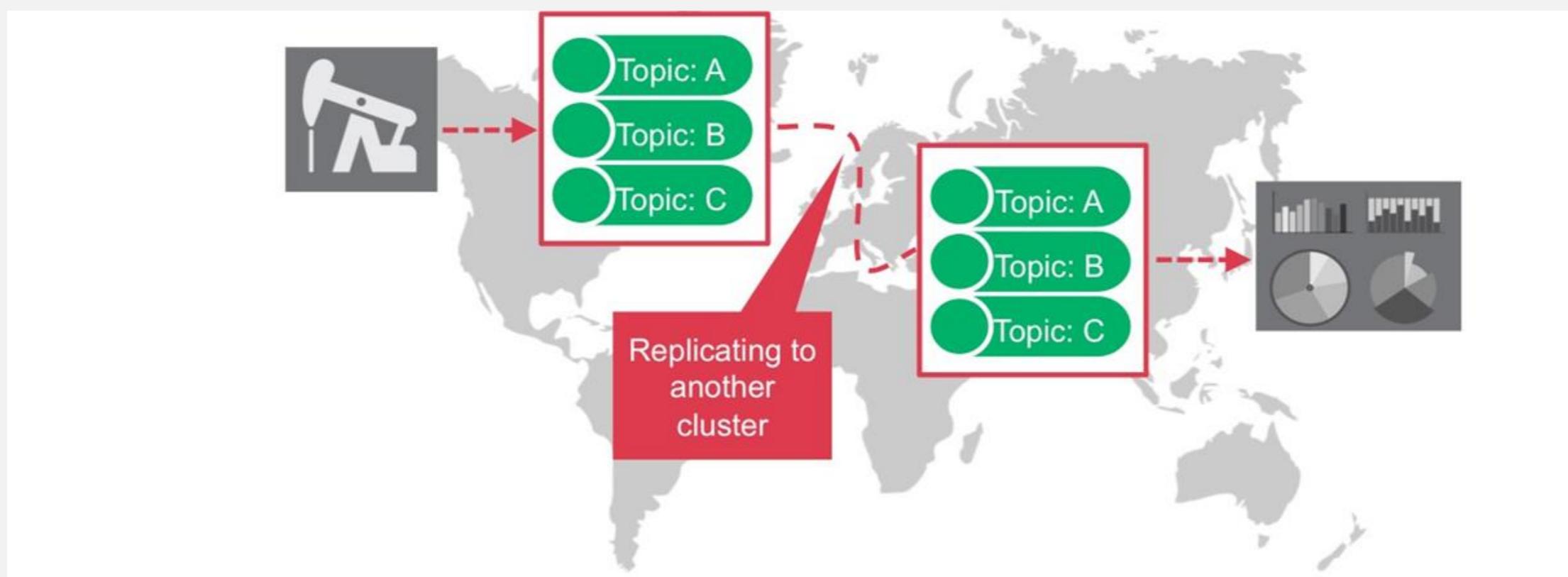
Lack of Global Replication



Streams and Replication

Streams:

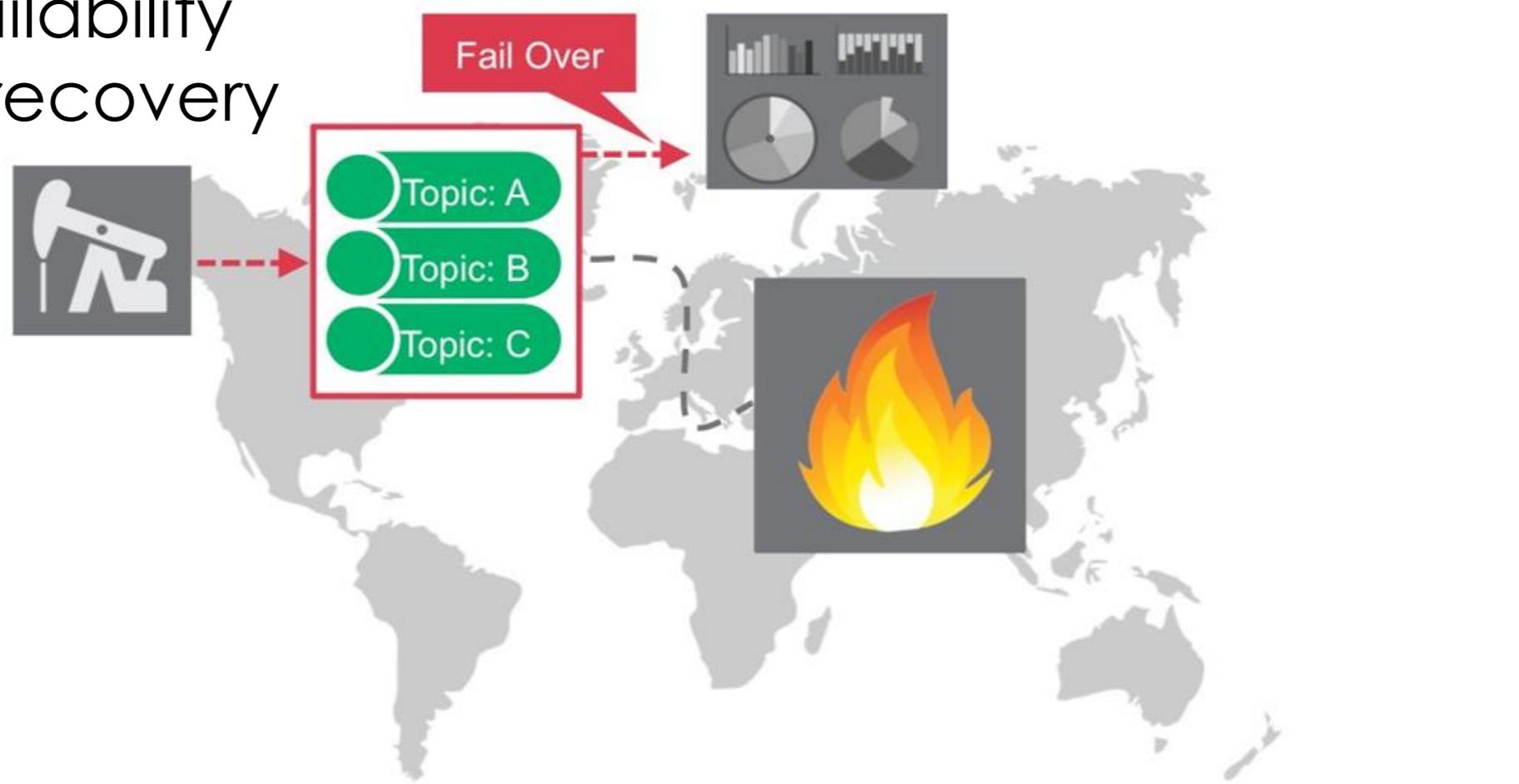
- are a collection of topics
- can be replicated worldwide



Streams and Replication

Streams:

- High availability
- Disaster recovery



Learning Goals

1.1: Summarize the motivation behind Kafka

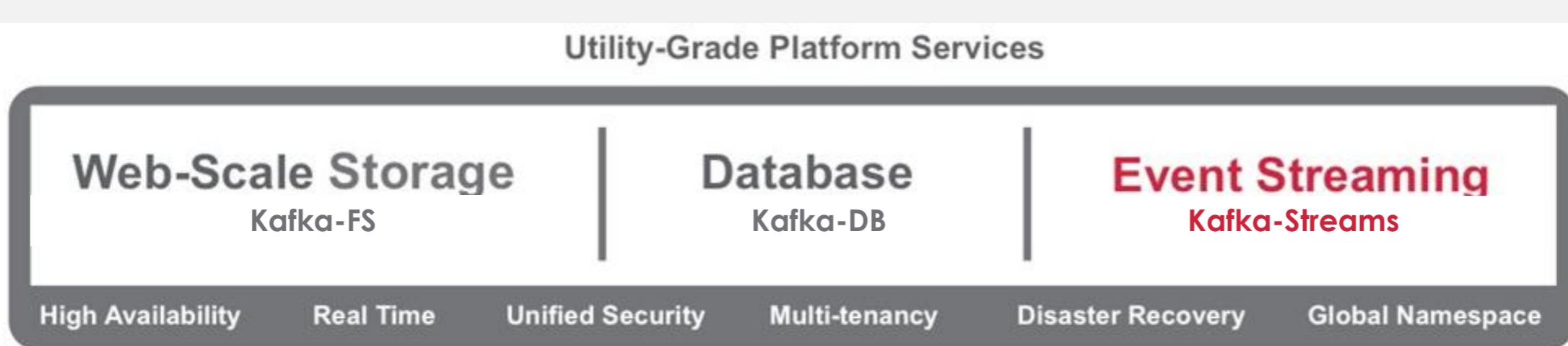
1.2: Explain what makes Kafka different

1.3: Apply Kafka to common use cases

What is Kafka Streams?

- A global publish/subscribe streaming system for big data
- Built on top of the Kafka Platform

kafka Converged Data Platform

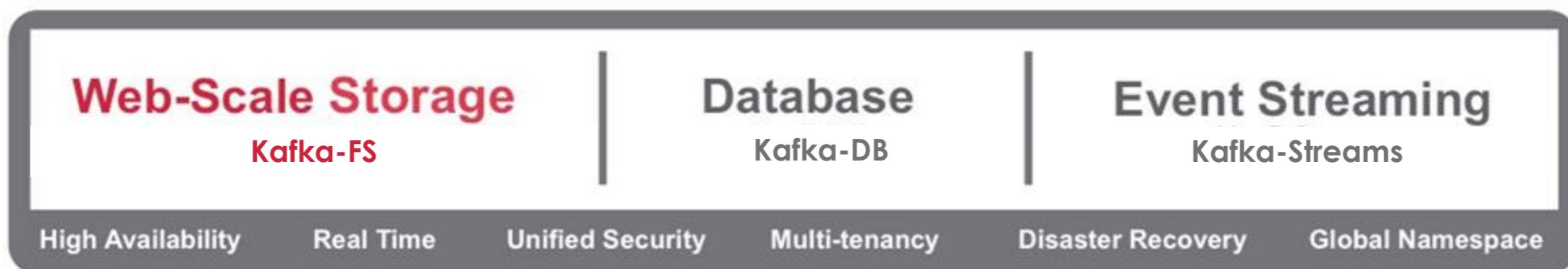


What is Kafka Streams?

- A global publish/subscribe streaming system for big data
- Built on top of the Kafka Platform

kafka Converged Data Platform

Utility-Grade Platform Services



What is Kafka Streams?

- A global publish/subscribe streaming system for big data
- Built on top of the Kafka Platform

kafka Converged Data Platform

Utility-Grade Platform Services



What is Kafka Streams?

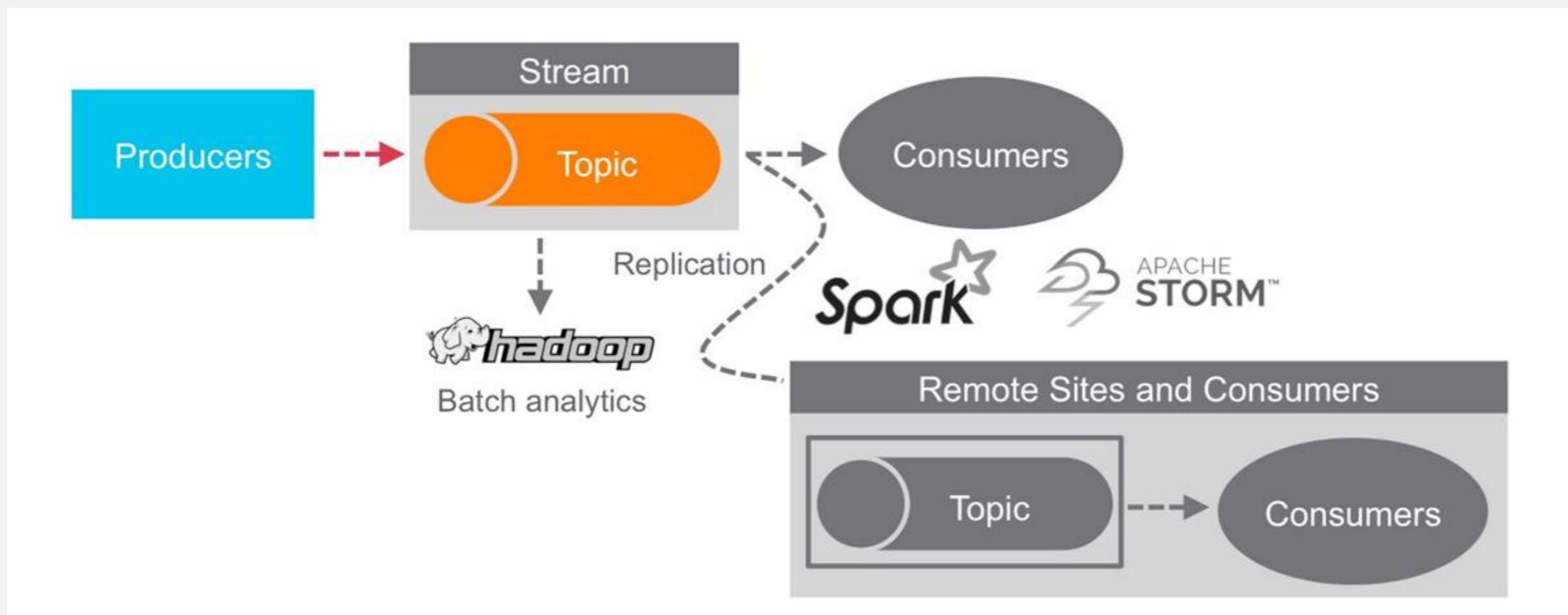
- A global publish/subscribe streaming system for big data
- Built on top of the Kafka Platform

kafka Converged Data Platform



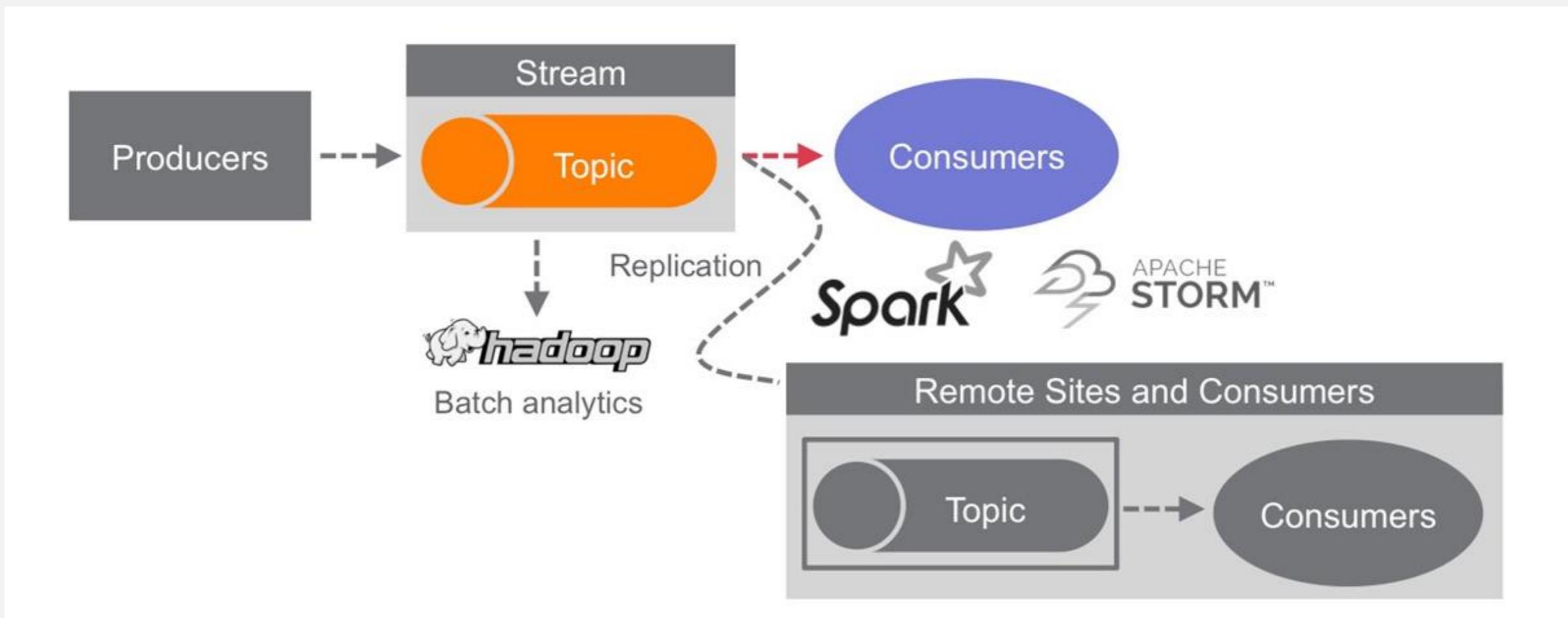
Kafka: A Global Pub-Sub System for Big Data

- Producers publish billions of events/sec



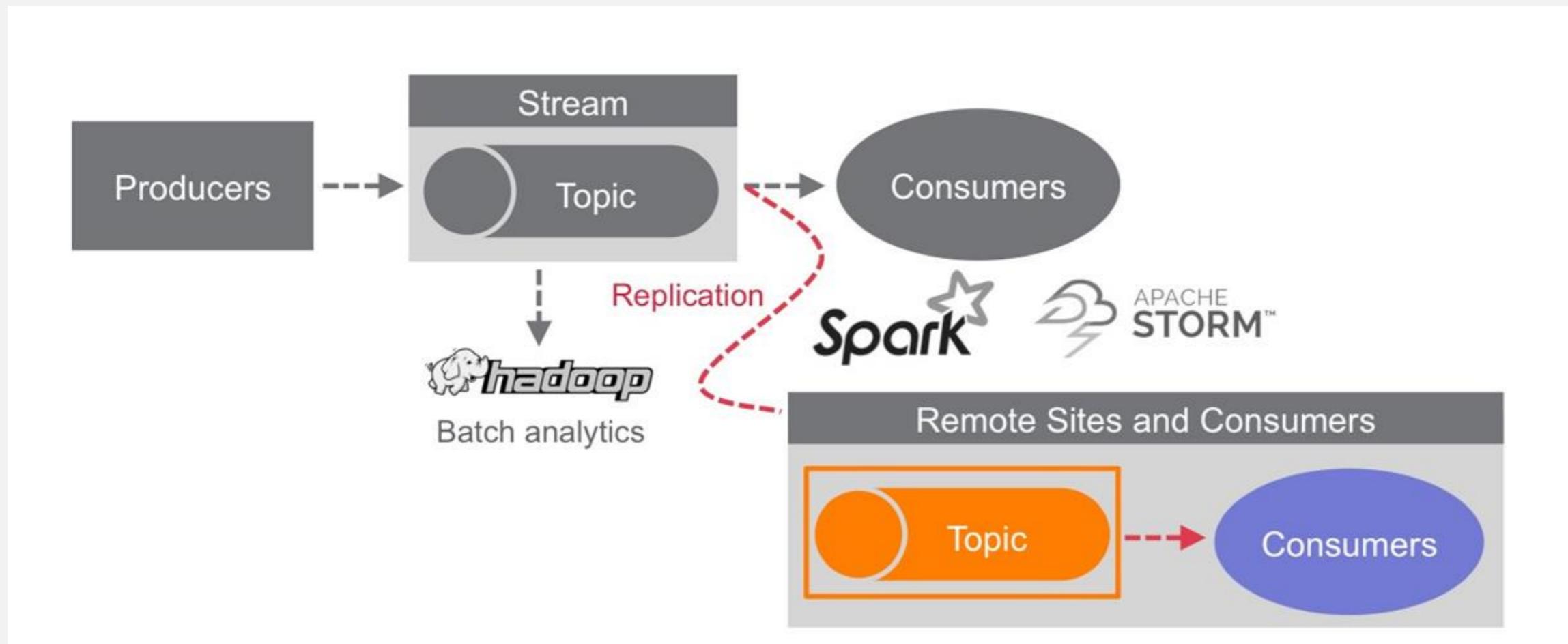
Kafka: A Global Pub-Sub System for Big Data

- Reliable deliveries to all customers



Kafka: A Global Pub-Sub System for Big Data

- Tie together geo-dispersed clusters worldwide

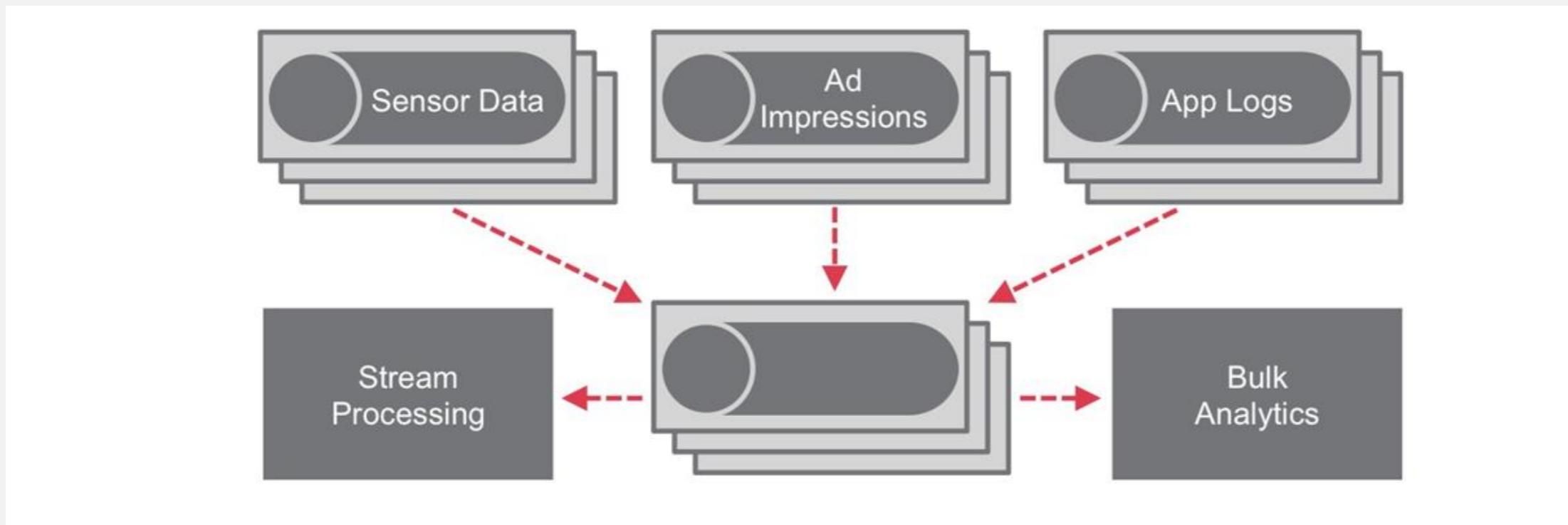


What is Kafka?

- Combines:
 - Massively scalable messaging
 - Consistent, reliable delivery
 - Real-time analytics
 - Security and fault-tolerance
- Standard real-time API (Kafka).
- OJAI API - direct data access from analytics frameworks

Kafka

- Scalable to handle Internet of Things
- Global producers and consumers
- Secure and fault-tolerant
- Bulk and real-time analytics



Learning Goals

- 1.1: Summarize the motivation behind Kafka
- 1.2: Explain what makes Kafka different
- 1.3: Apply Kafka to common use cases**

Kafka & Big Data Analytics



Health Care



Advertising



Credit Card Security



Internet of Things

Use Case: Streaming System of Record for Healthcare

Objective:

- Build a flexible, secure healthcare database

Challenges:

- Many different data models
- Security and privacy issues
- HIPAA compliance



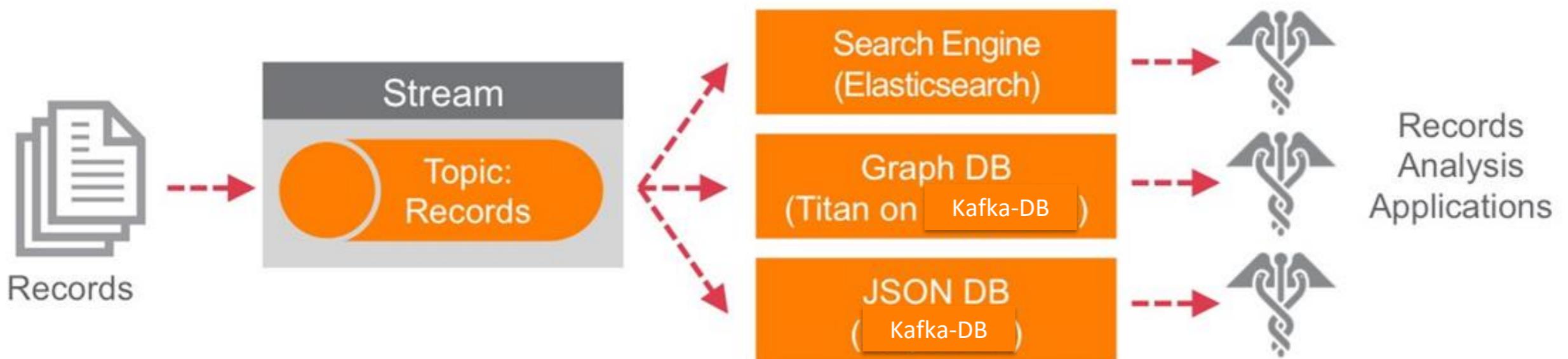
Use Case: Streaming System of Record for Healthcare

Solution:

- Streaming system of record
 - secure
 - immutable
 - rewritable

Auditable:

- Materialized views continuously computed
- Selective cross data center replication



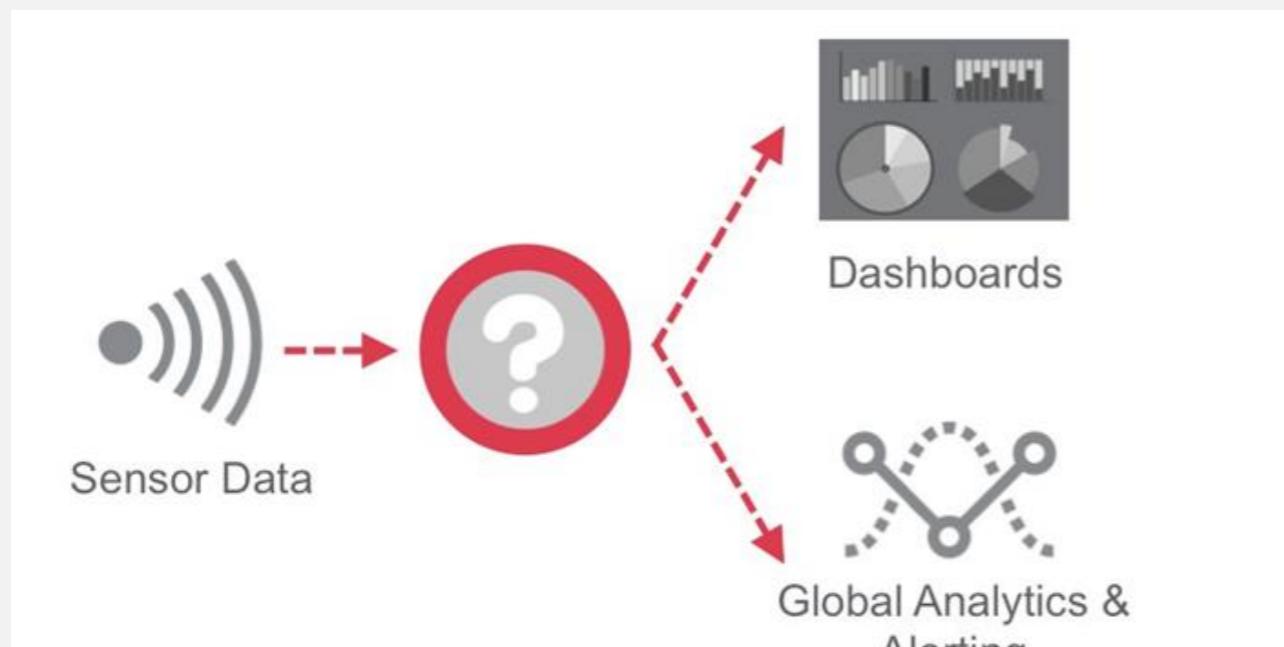
Use Case: Monitoring the Internet of Things in Real-Time

Objective:

- Monitor oil rig sensor data and create real-time alerts

Challenges:

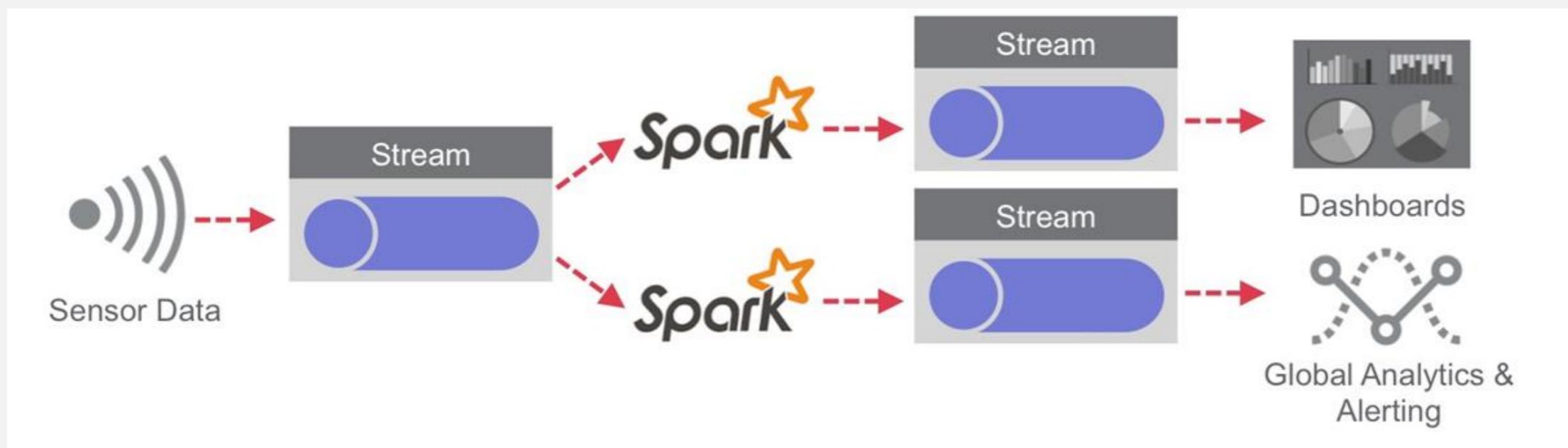
- Disperse, global data sources
- Need alerts in real-time
- Need to audit data



Use Case: Monitoring the Internet of Things in Real-Time

Solution:

- Synchronize global servers
- Streaming analytics
- Replicated streams



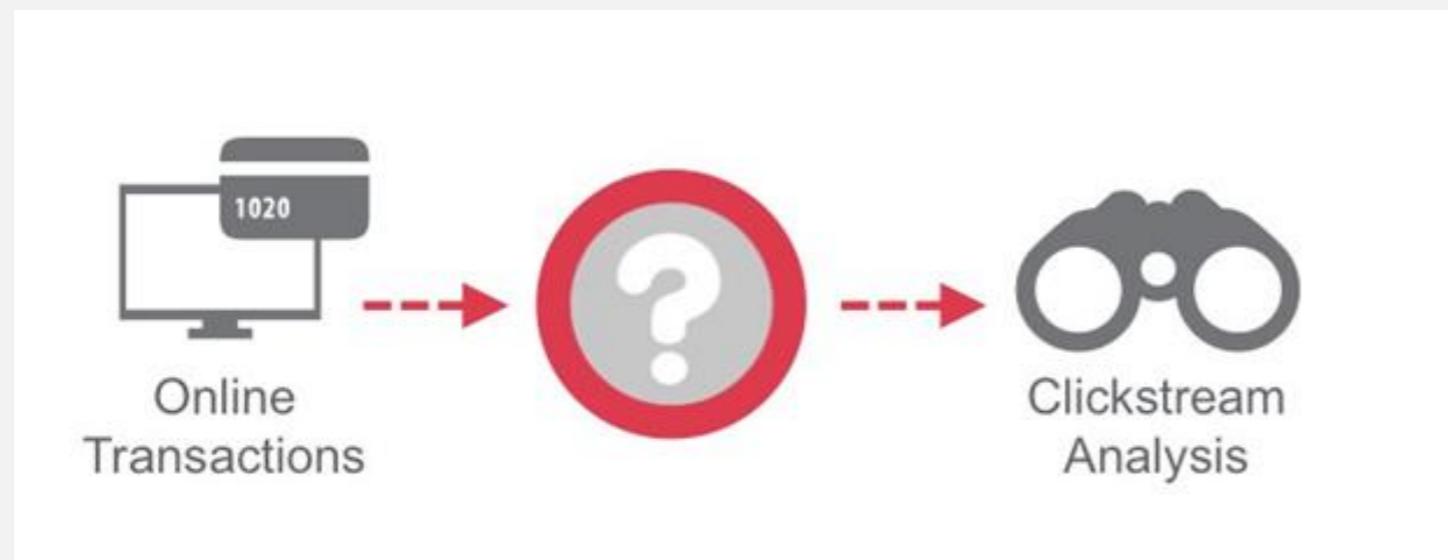
Use Case: Global Analytics in Targeted Advertising

Objective:

- Optimize global ad market with real-time analytics

Challenges:

- Data consistency
- Global insights
- Complex pipeline



Use Case: Global Analytics in Targeted Advertising

Solution:

- Real-time analytics
- Global message streams
- Streaming ETL



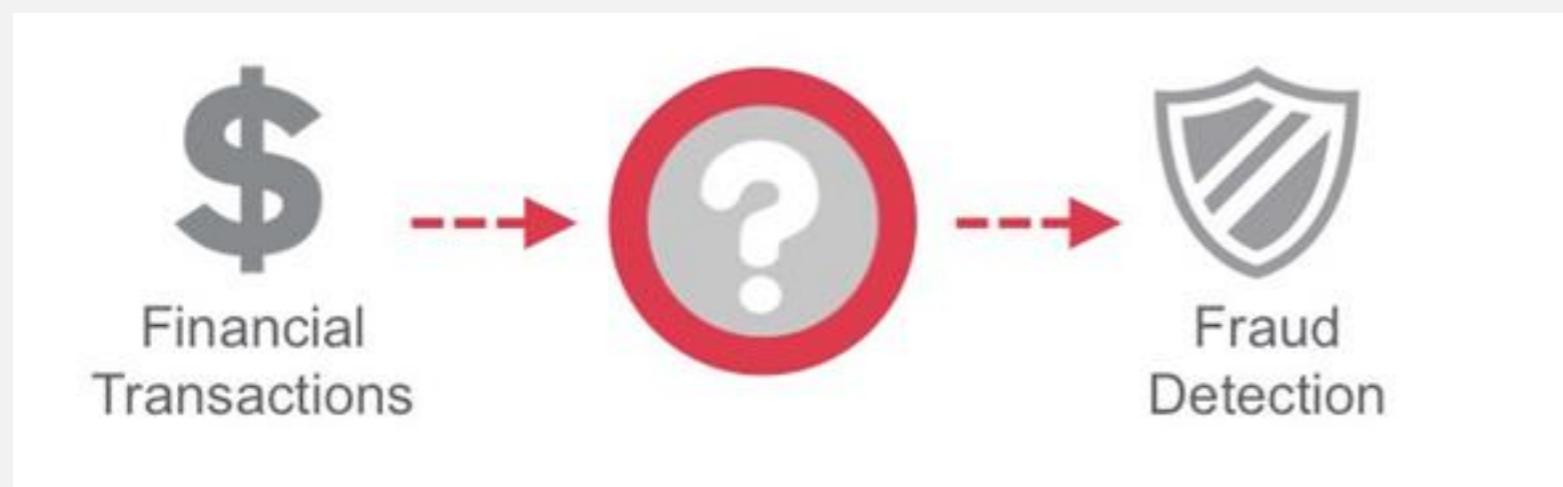
Use Case: Intelligent Credit Card Processing

Objective:

- Build reliable, real-time service bus for financial transactions

Challenges:

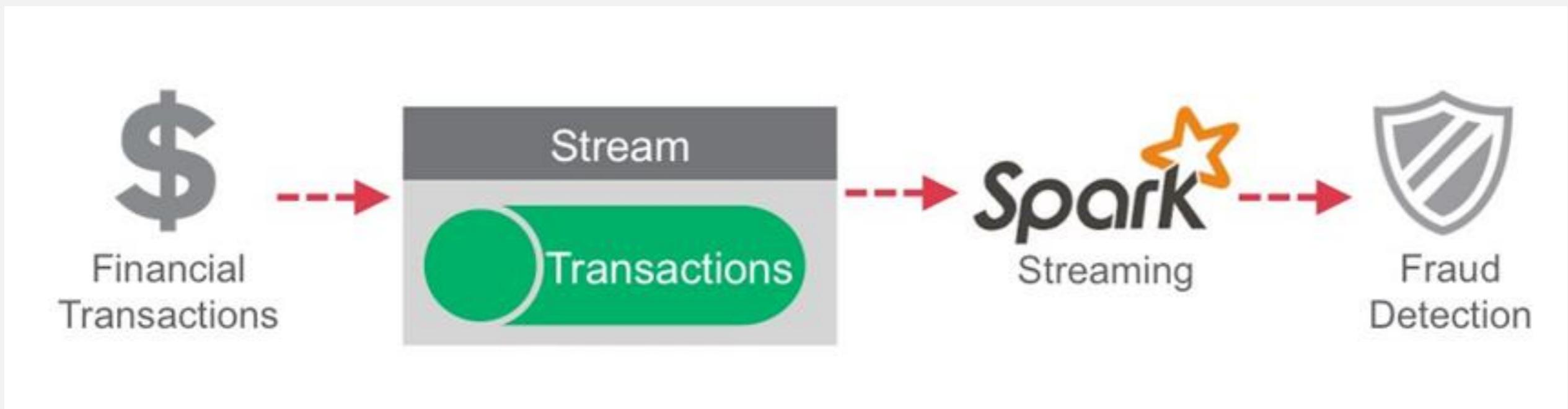
- Accurate data
- Sensitive data
- Ad-hoc querying



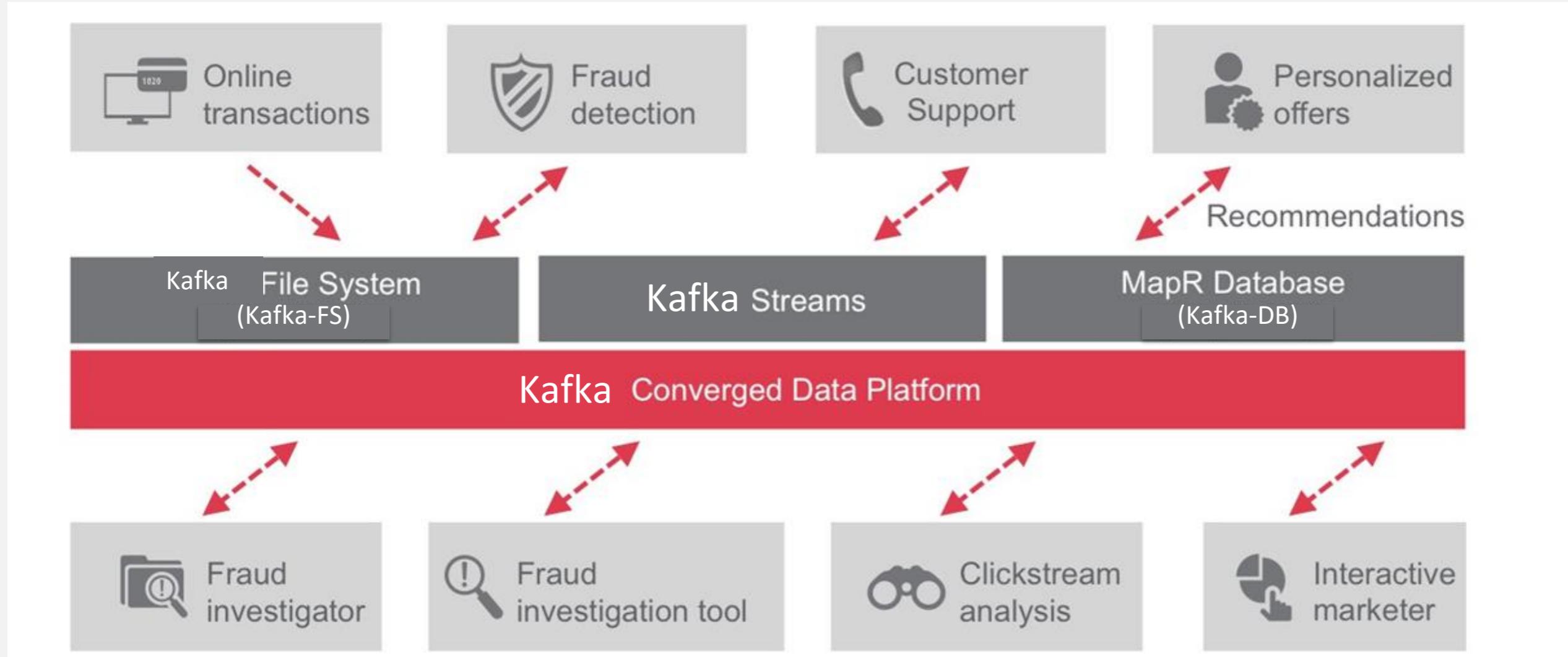
Use Case: Intelligent Credit Card Processing

Solution:

- Reliable, replicated streams
- Security settings



Event time Processing on one platform



Class Discussion

Think about the data you work with, as a developer, administrator, or data analyst

- How can Kafka fit with your data pipeline?
- What features of Kafka will you use?
- What problems can Kafka solve?

Learning Goals

- Define core components of Kafka
- Summarize the life of a message in Kafka
- Explain how Kafka fits in the complete data architecture

Learning Goals

- **Define core components of Kafka**
- Summarize the life of a message in Kafka
- Explain how Kafka fits in the complete data architecture

Core Components : Message/Record

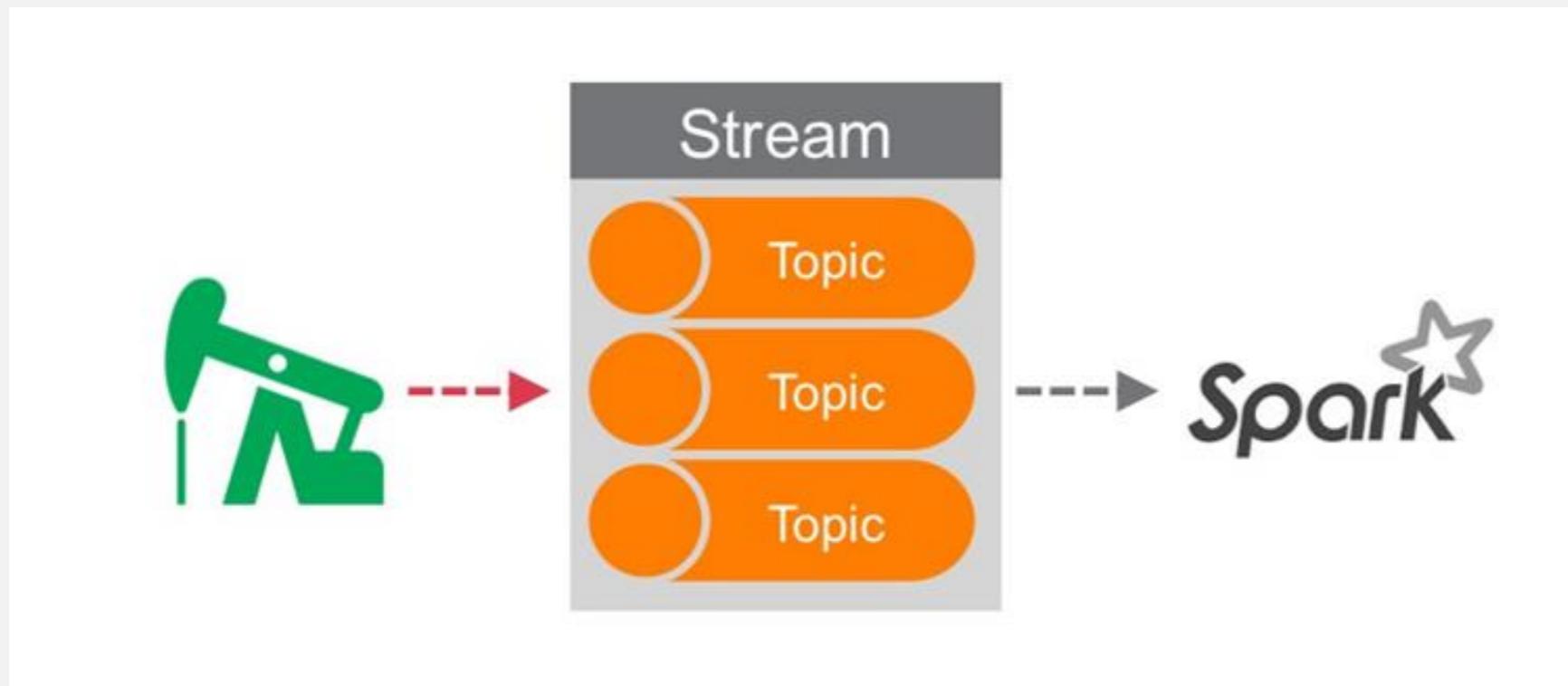
- Messages are key/value pairs
- Keys are optional
- Values can be any type of data

key	value
	{"event_ID":122, "hcid":63, "timestamp": "2015-01-01 00:00:02", "production":47, "sensor_readings": {"pressure":107, "temperature":30, "oil_percentage":18}, "injection_vol": 15367}

Core Components: Producers and Consumers

Producers

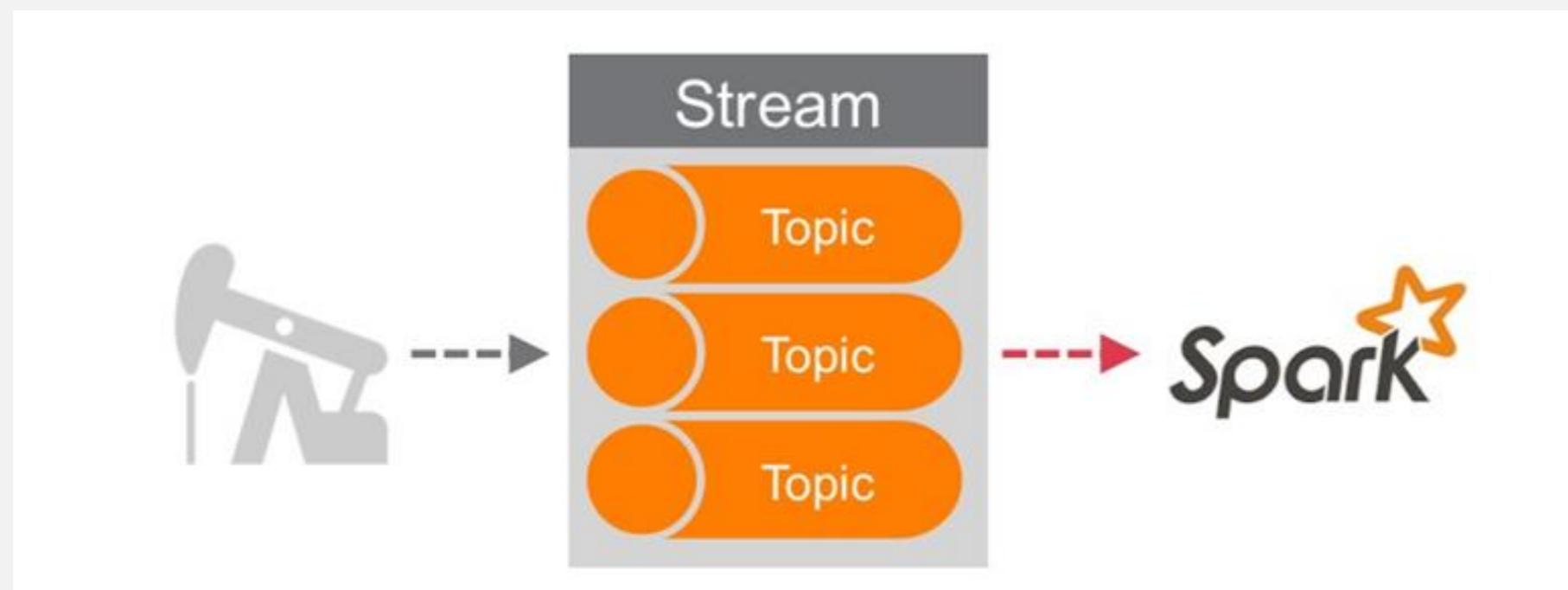
- publish to topics
- example: networked sensors on an oil rig



Core Components: Producers and Consumers

Consumers

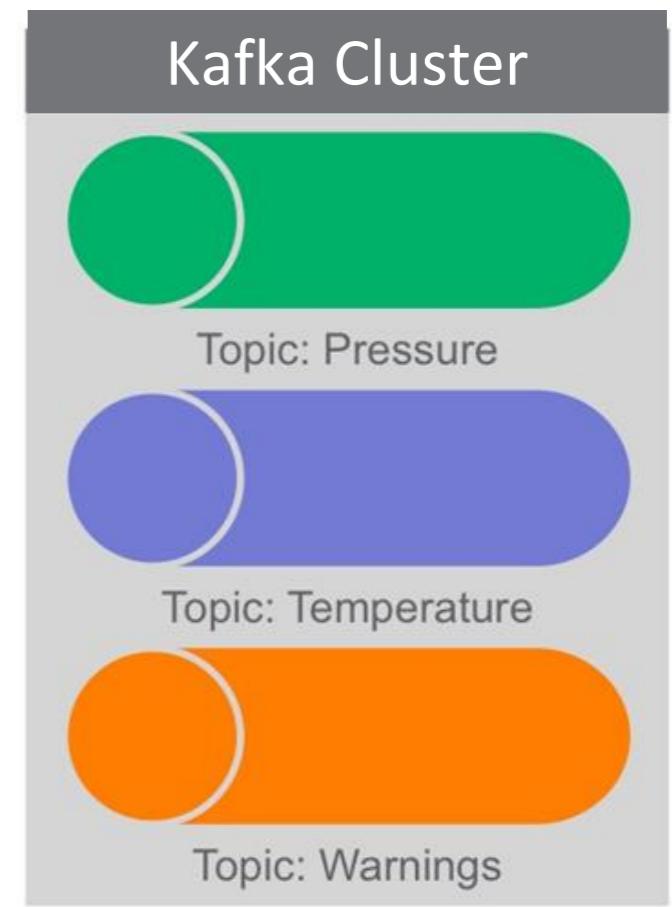
- subscribe to topics
- example: applications using Apache Spark



Core Components: Topics

Topics:

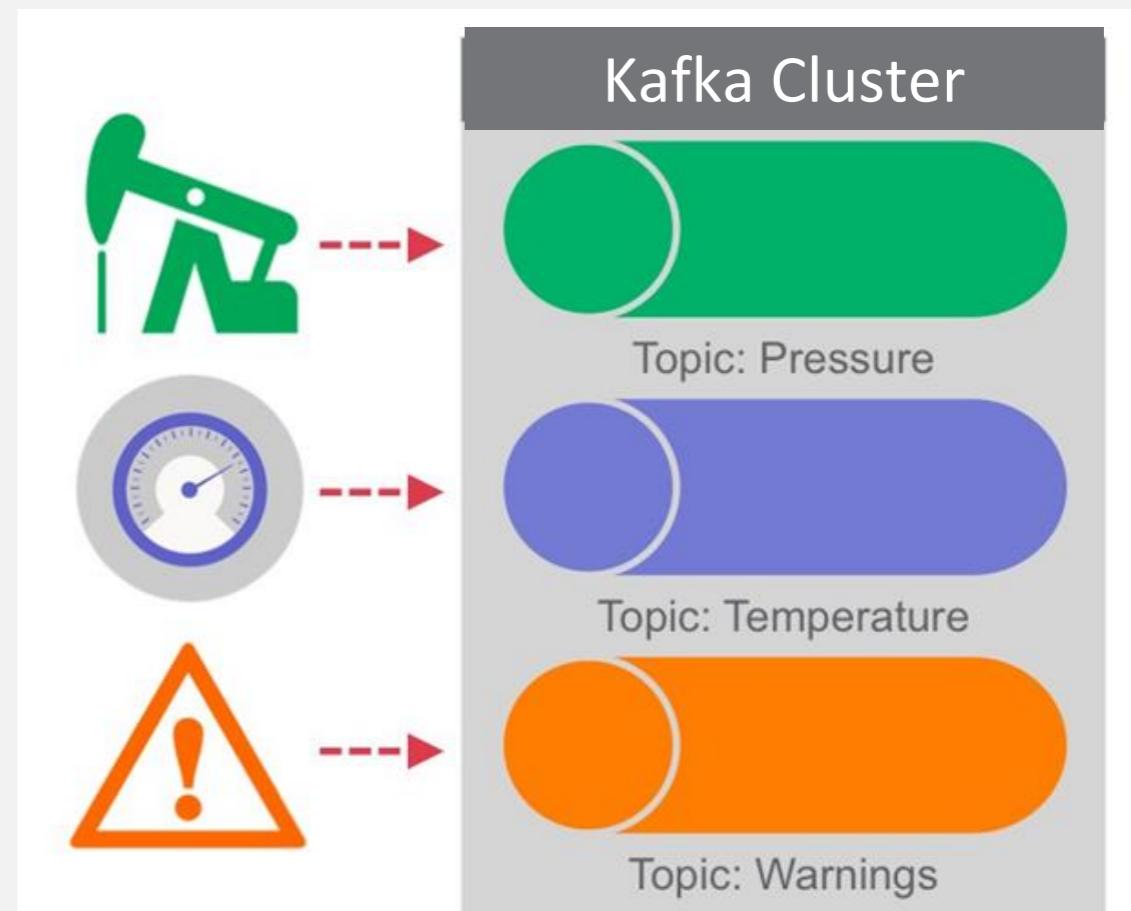
- logical collections of messages managed by Kafka



Core Components: Topics & Producers

Topics:

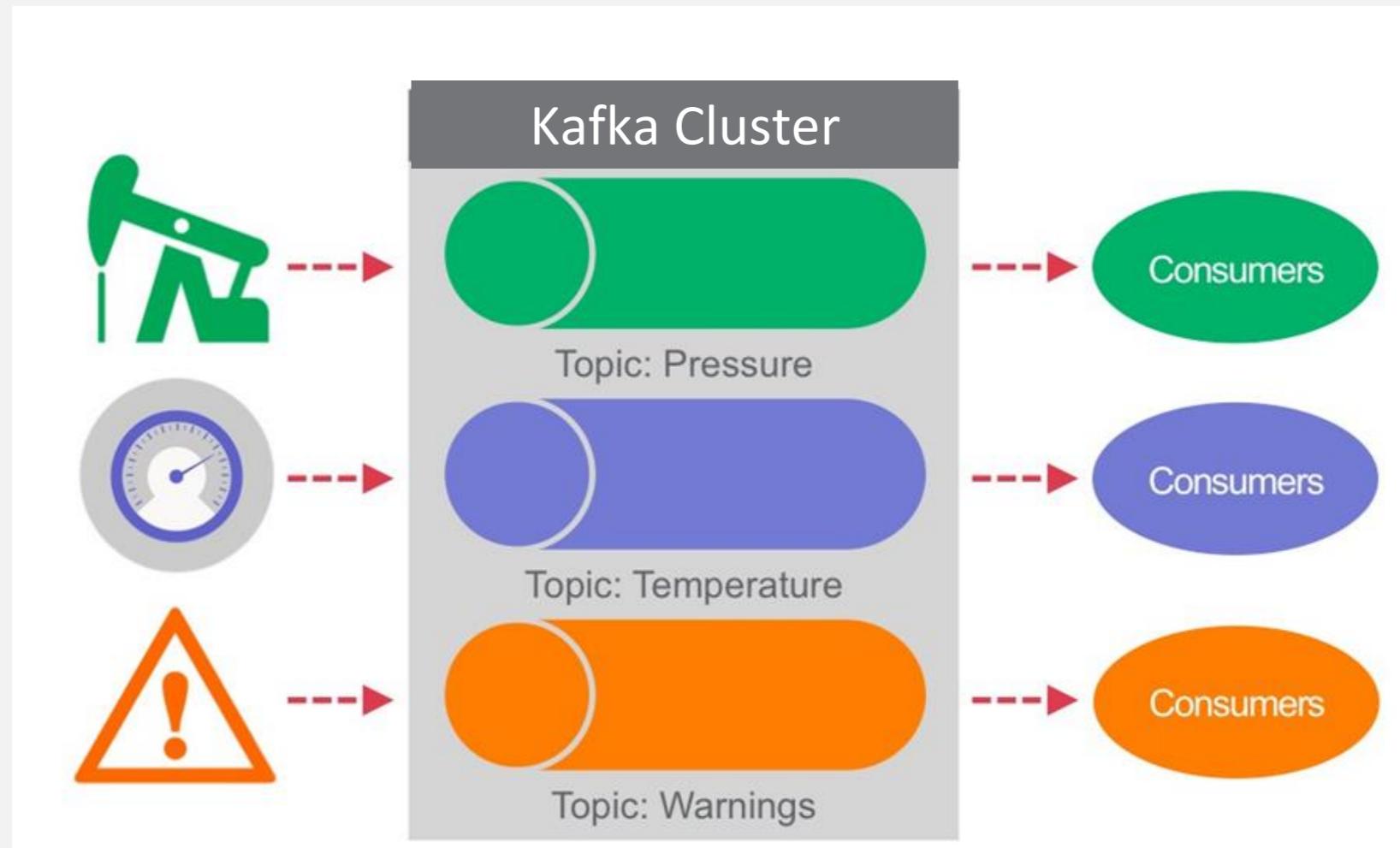
- logical collections of messages managed by Kafka
- Organized events
- Producers publish to relevant topic



Core Components: Topics & Subscriptions

Subscriptions:

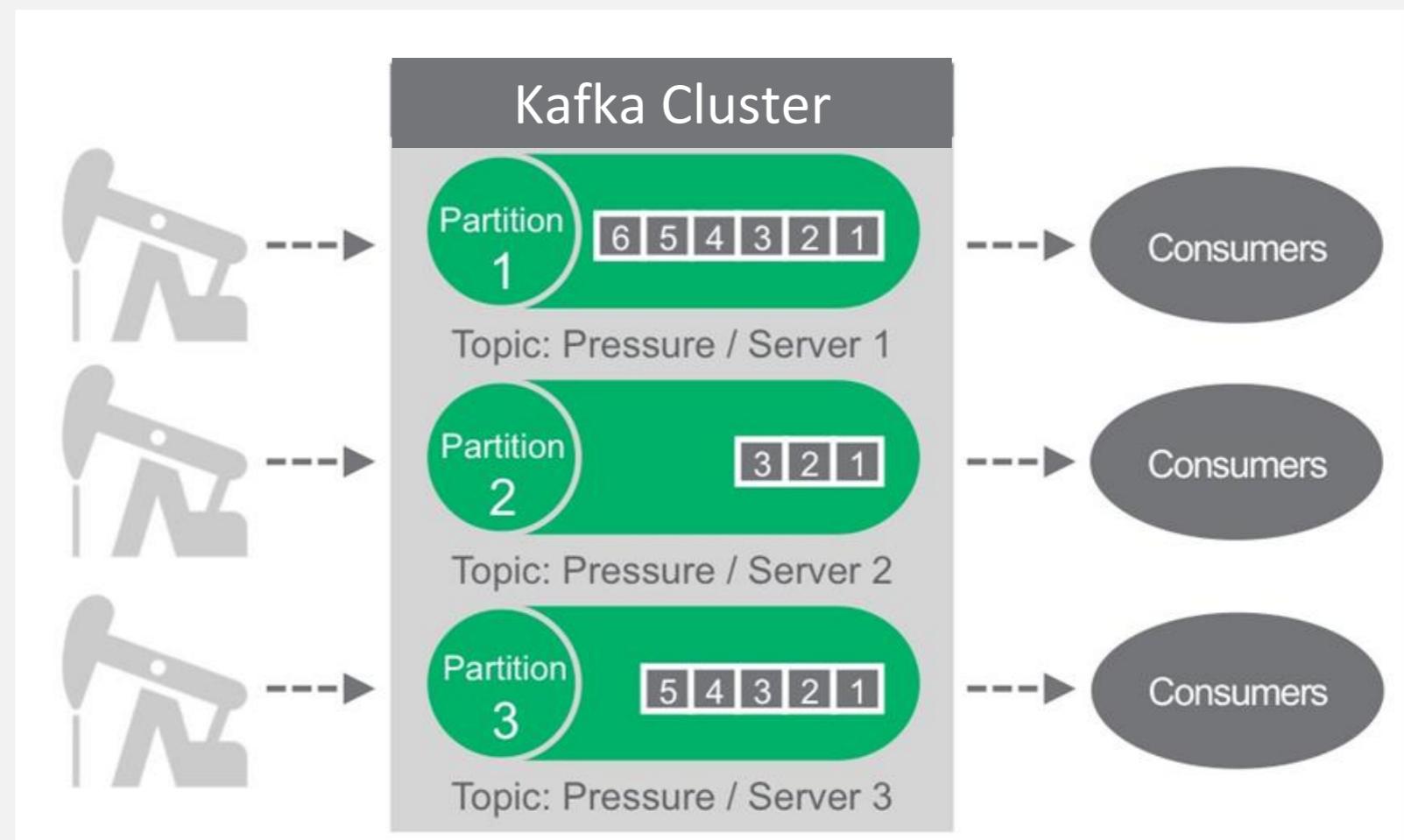
— list of topics a consumer is subscribed to.



Core Components: Partitions

Partitions:

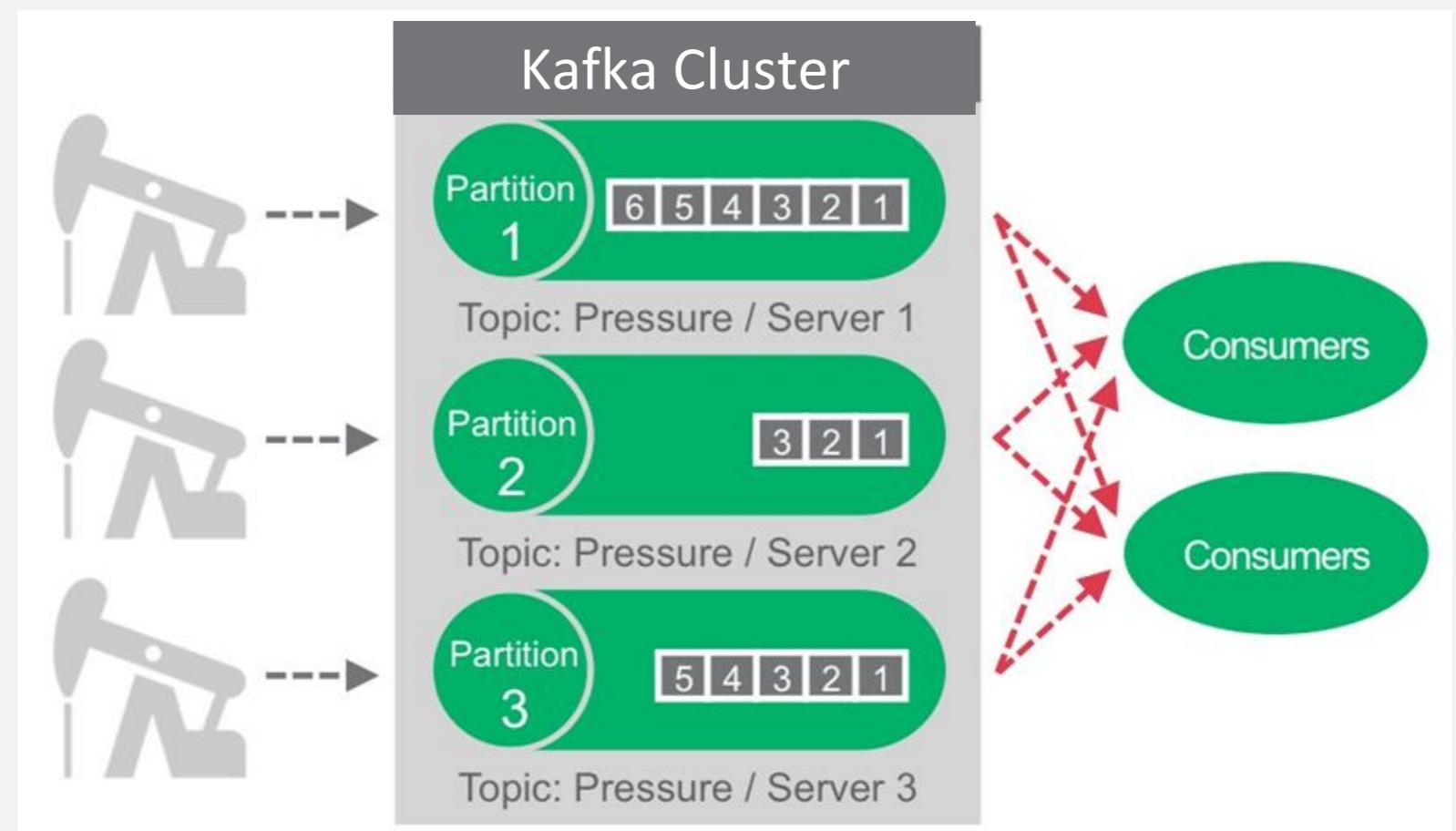
- smaller unit of a topic
- parallel, ordered, sequences of messages continually appended to



Core Components: Partitions

Partitions:

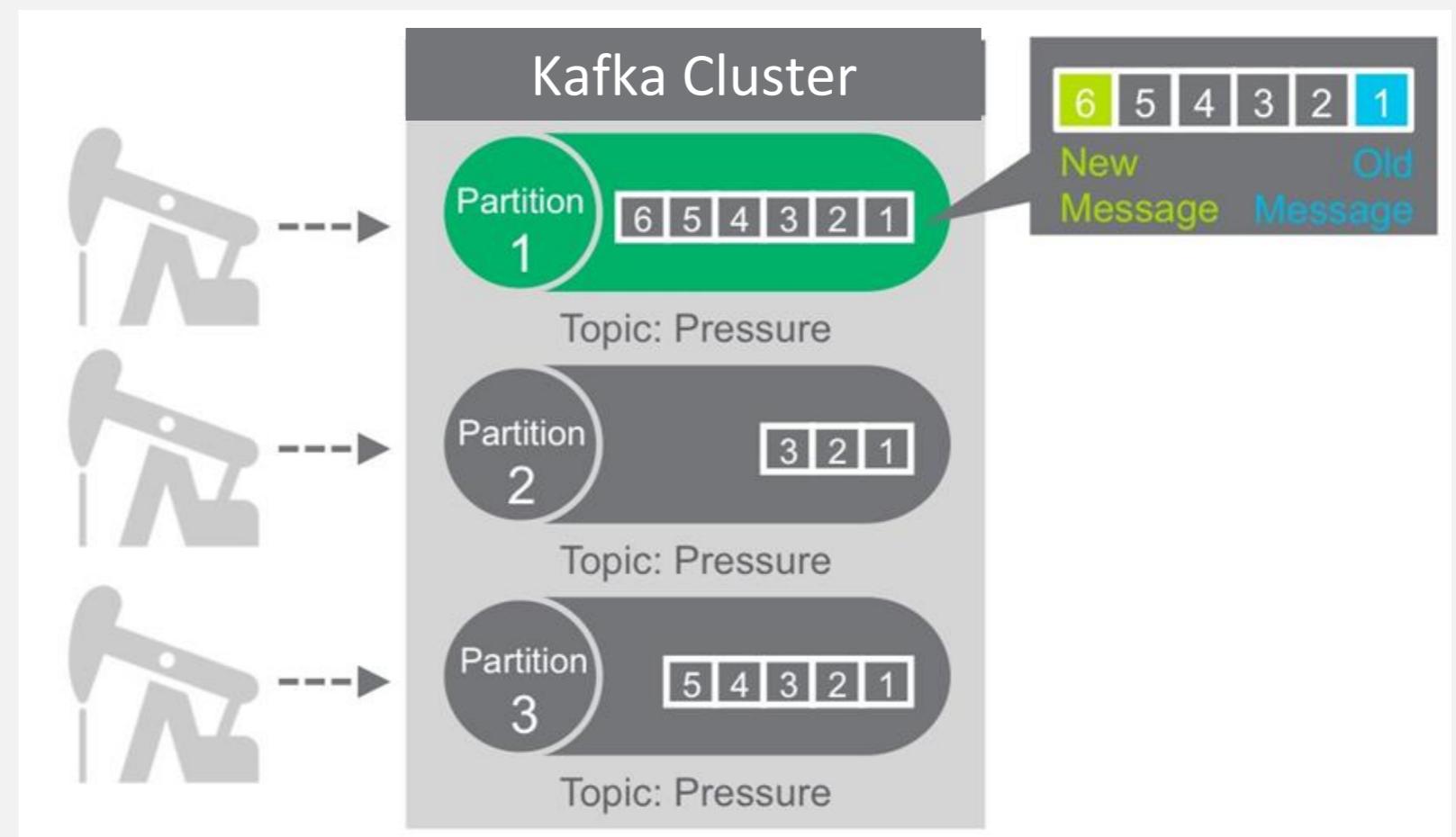
- smaller unit of a topic
- parallel, ordered, sequences of messages continually appended to



Core Components: Partitions

Offset:

- ID of a message within a partition



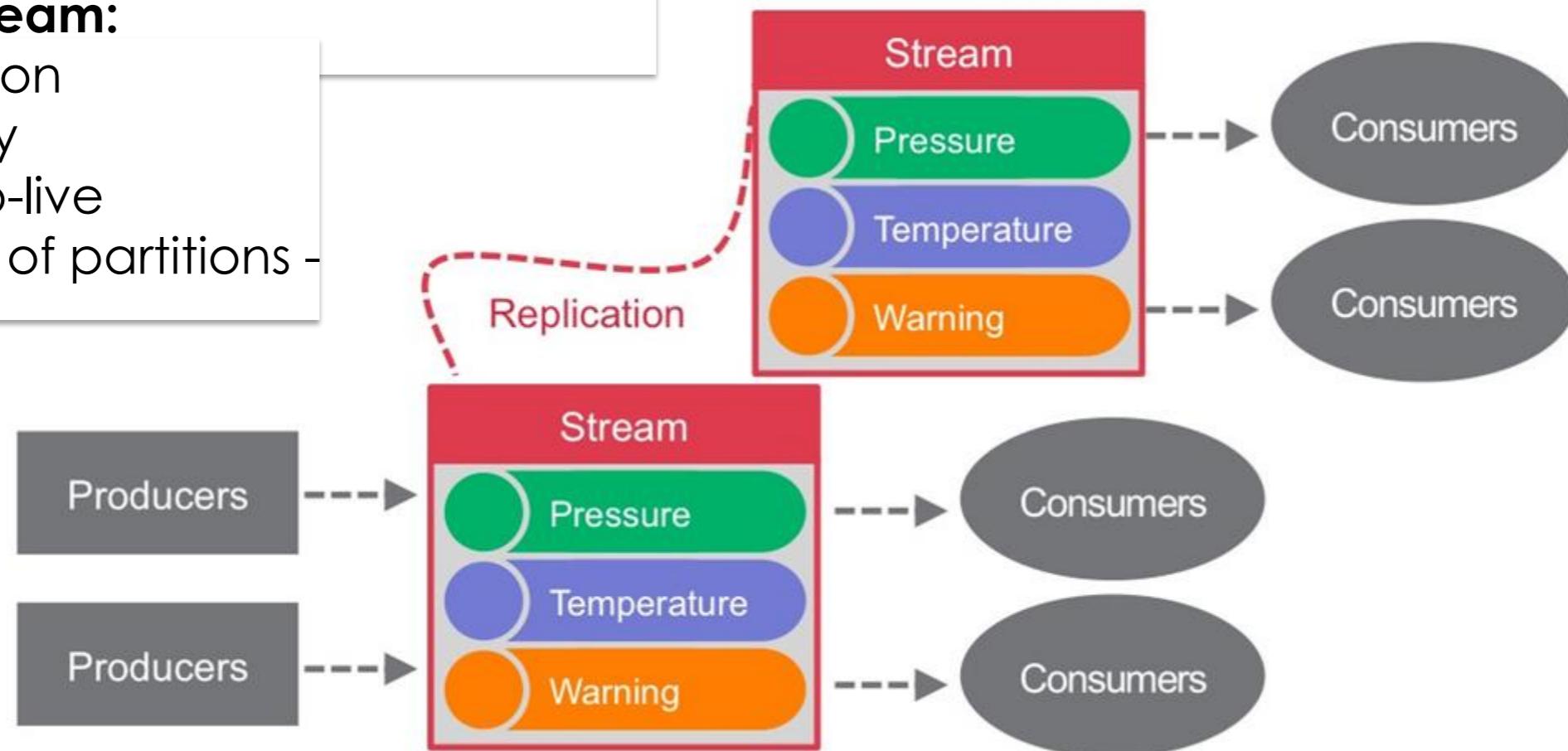
Core Components: Streams

Stream:

- collection of topics managed together

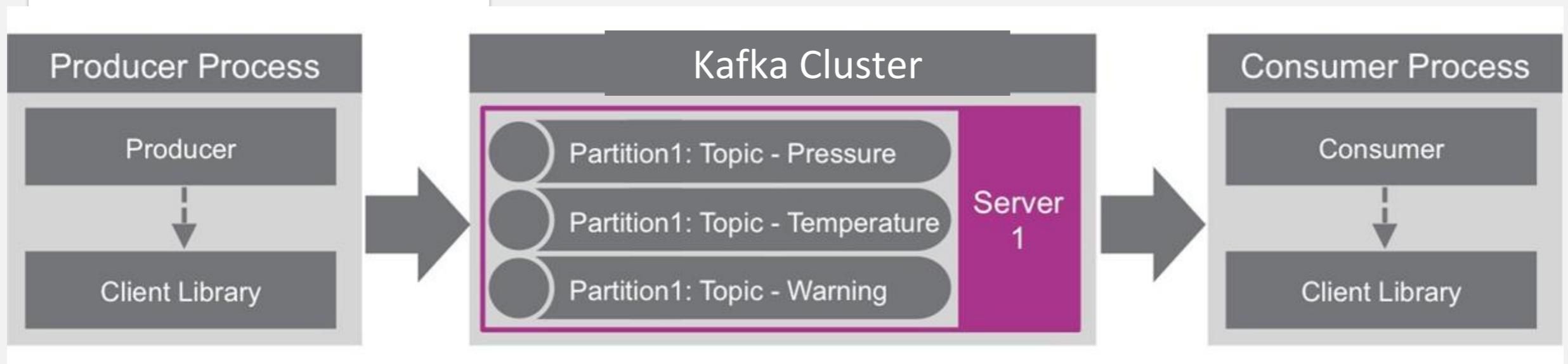
Manage stream:

- replication
- security
- time-to-live
- number of partitions -



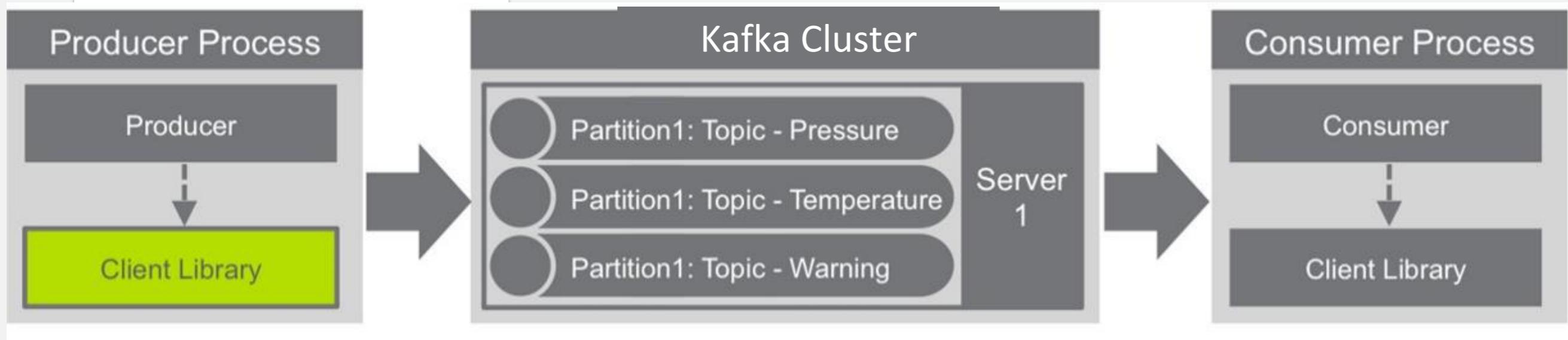
Core Components: Server

- Manages streams, topics, partitions
- Handles requests



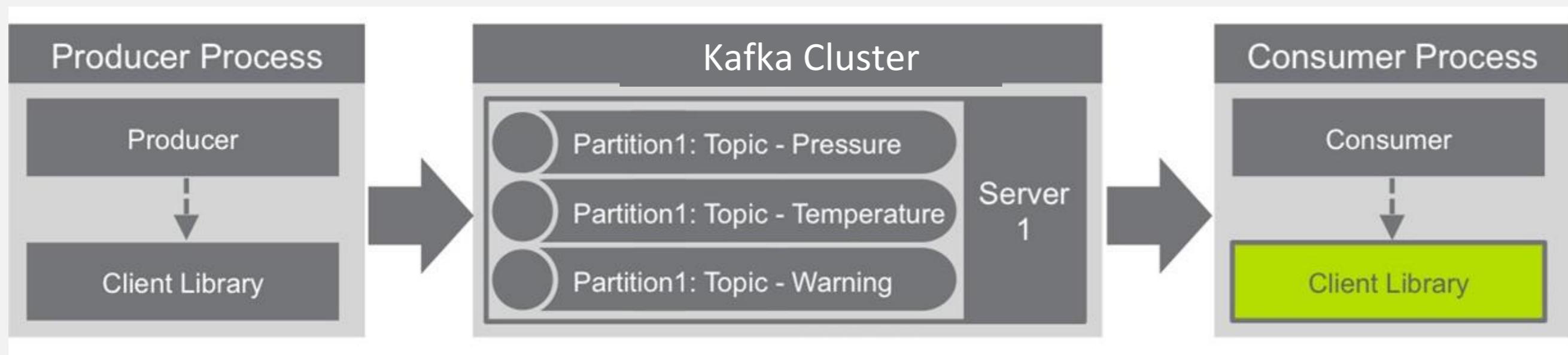
Core Components: Producer Client Library

- Receives producer's messages
- Buffers messages
- Sends messages to the server



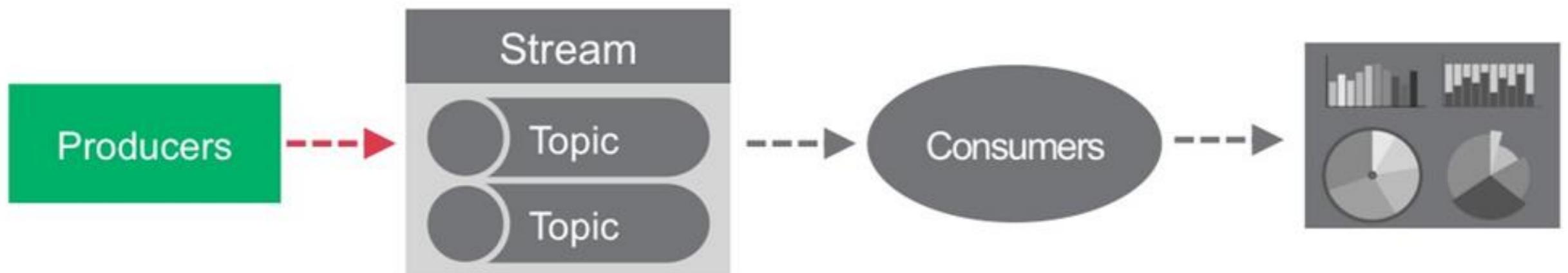
Core Components: Consumer Client Library

- Receives consumer's requests
- Reads messages from topic
- Sends to consumers



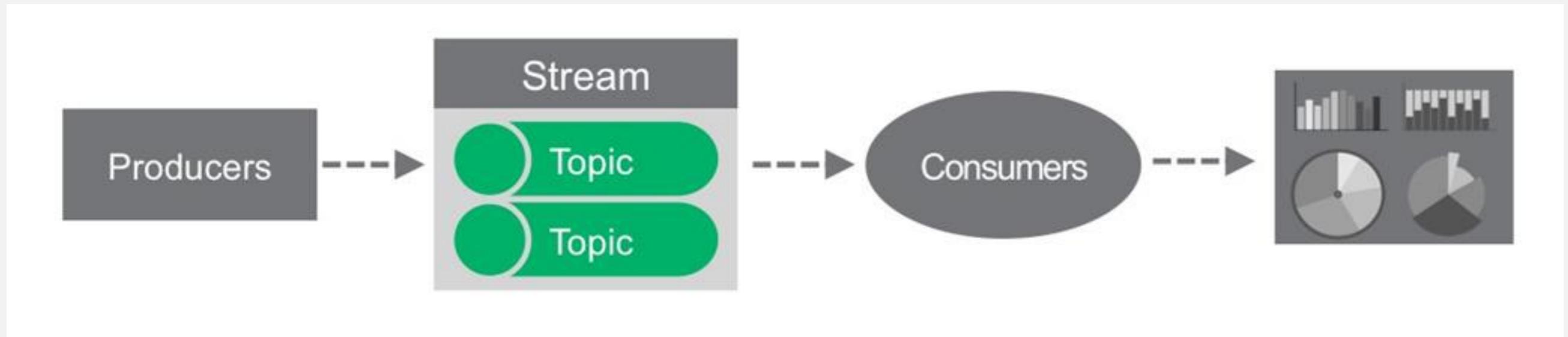
How Does Kafka Work?

- Producers publish messages to a topic



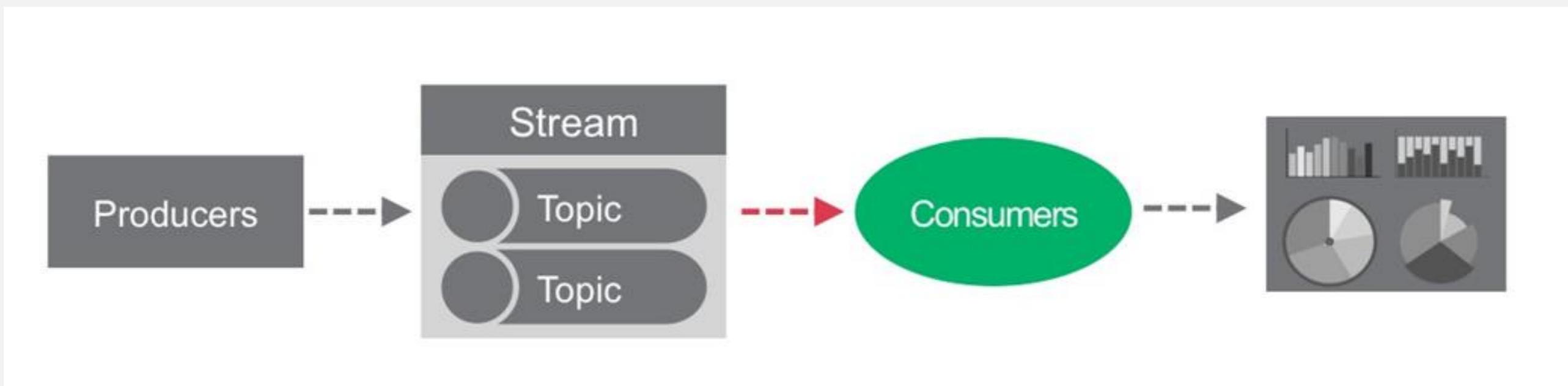
How Does Kafka Work?

- Topics organize messages into categories



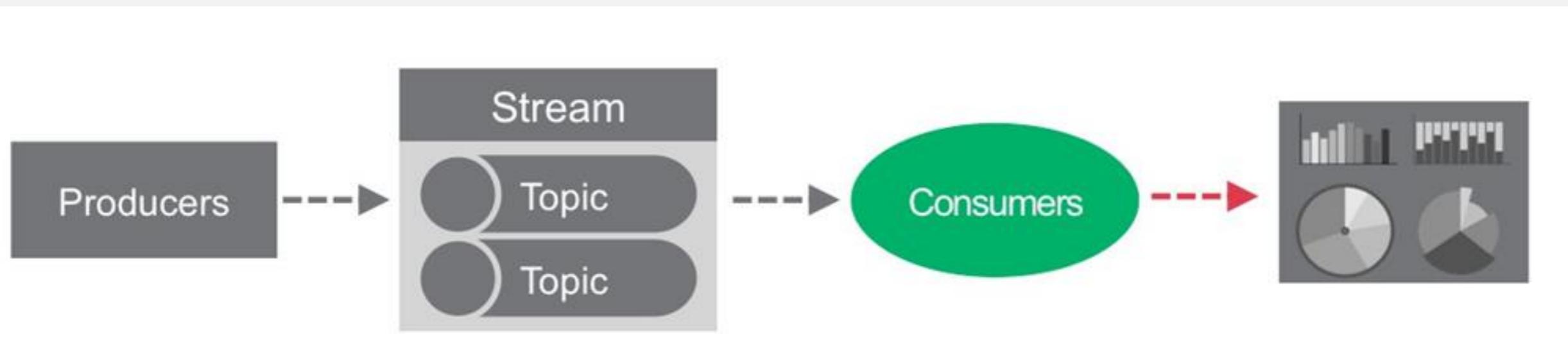
How Does Kafka Work?

- Consumers subscribe to topics with Kafka API



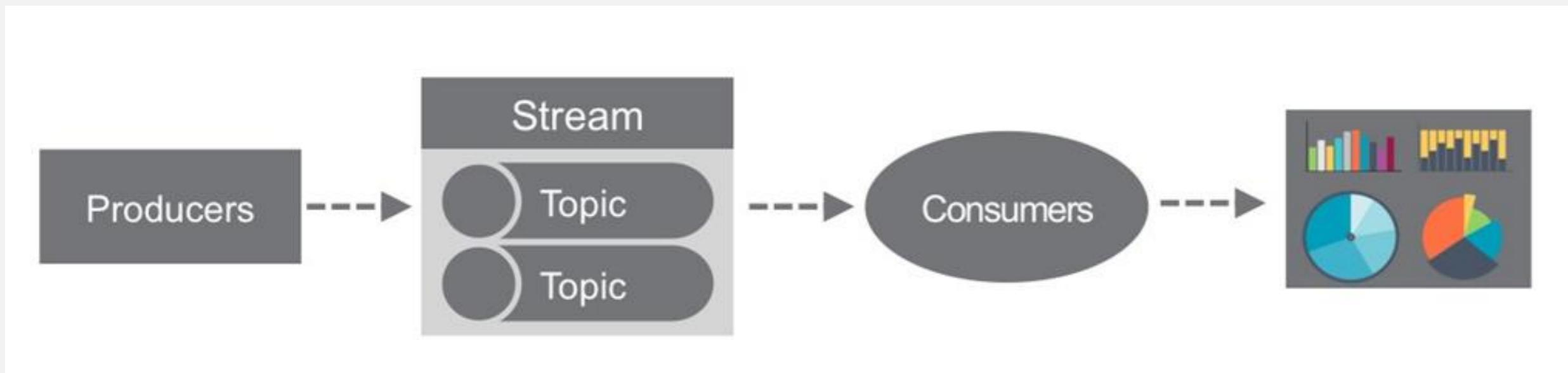
How Does Kafka Work?

- Kafka delivers to consumers in real-time



How Does Kafka Work?

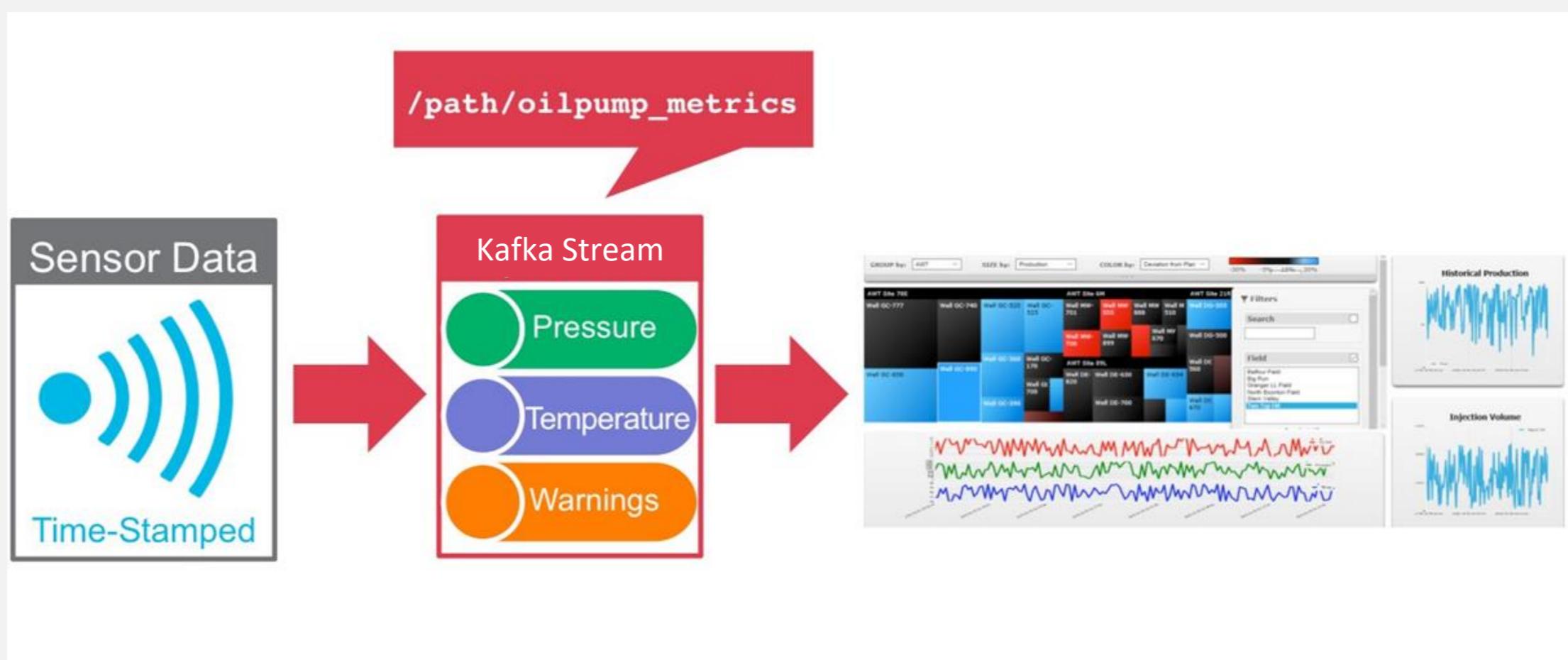
- Consumers process data at their own pace



Learning Goals

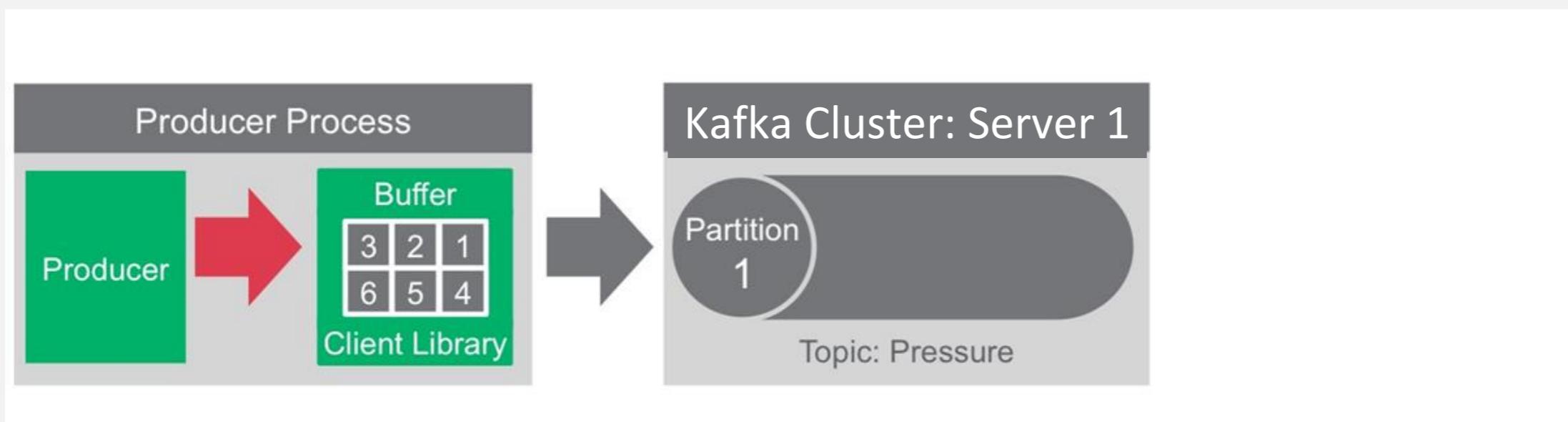
- Define core components of Kafka
- **Summarize the life of a message in Kafka**
- Explain how Kafka fits in the complete data architecture

Life of a Message



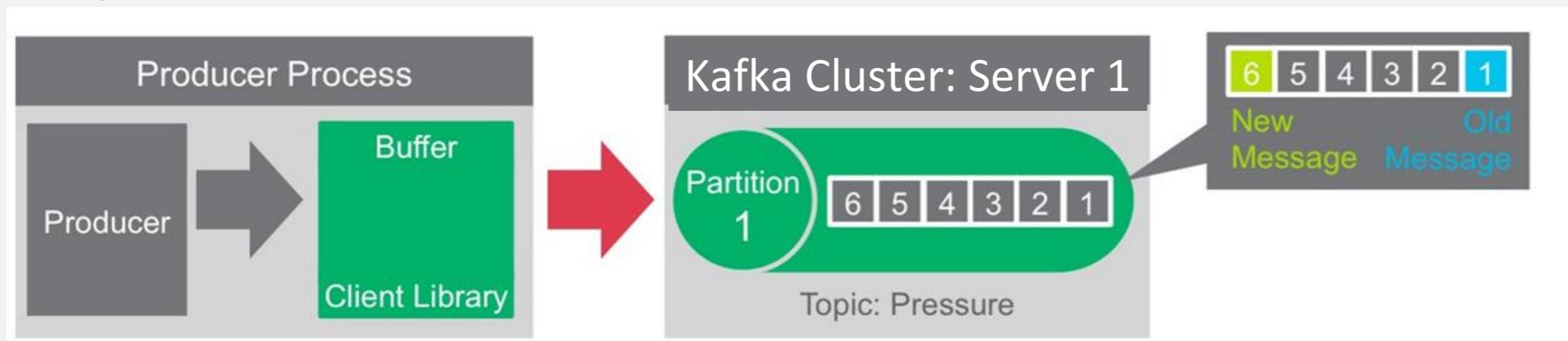
How Are Messages Sent?

- Producers creates and send messages to client library
- Client library buffers the messages



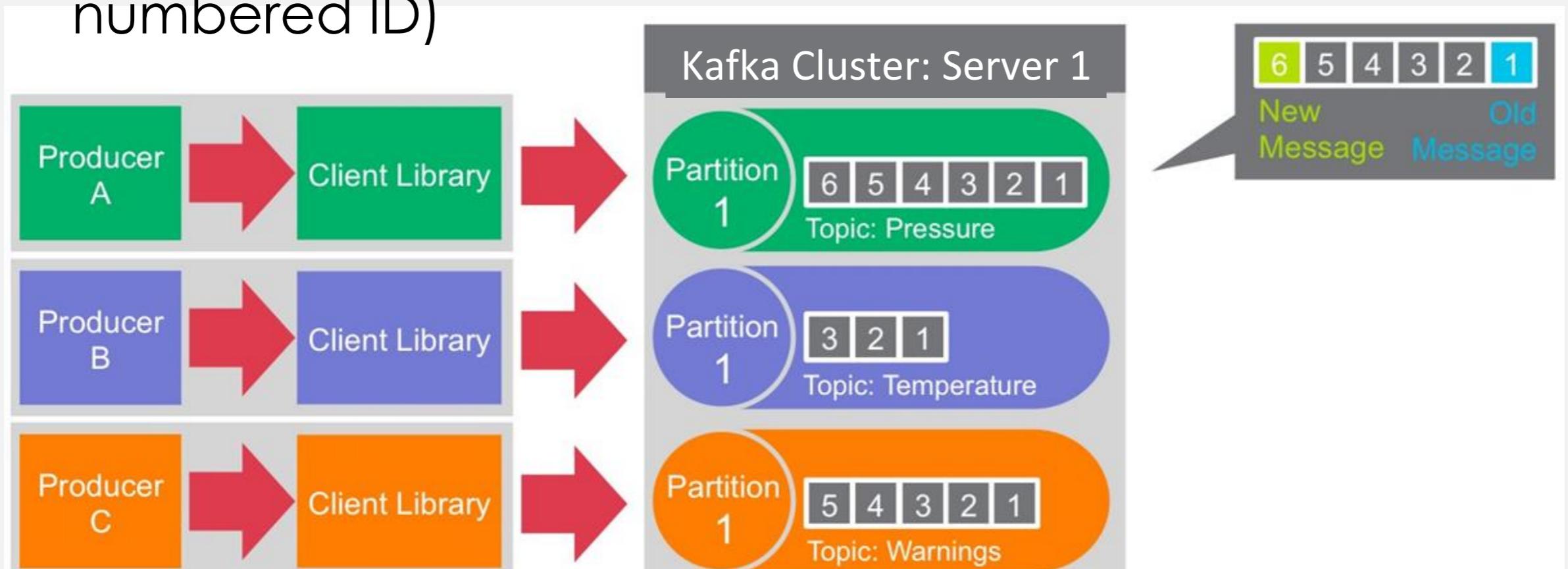
How Are Messages Sent?

- Client batches and sends messages in the buffer
- Server publishes messages to topics specified by the producer



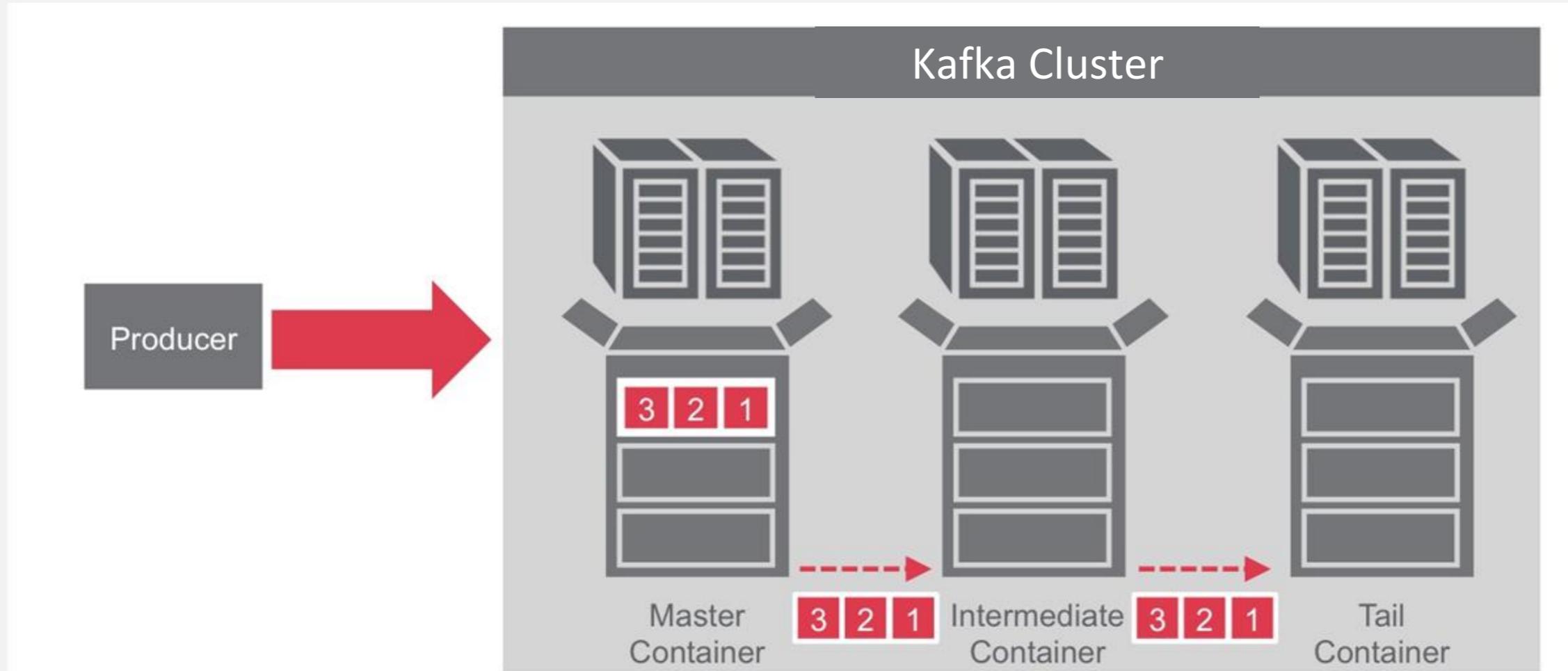
How Are Messages Sent?

- Messages are appended in order
- Each message is given an offset (sequentially numbered ID)



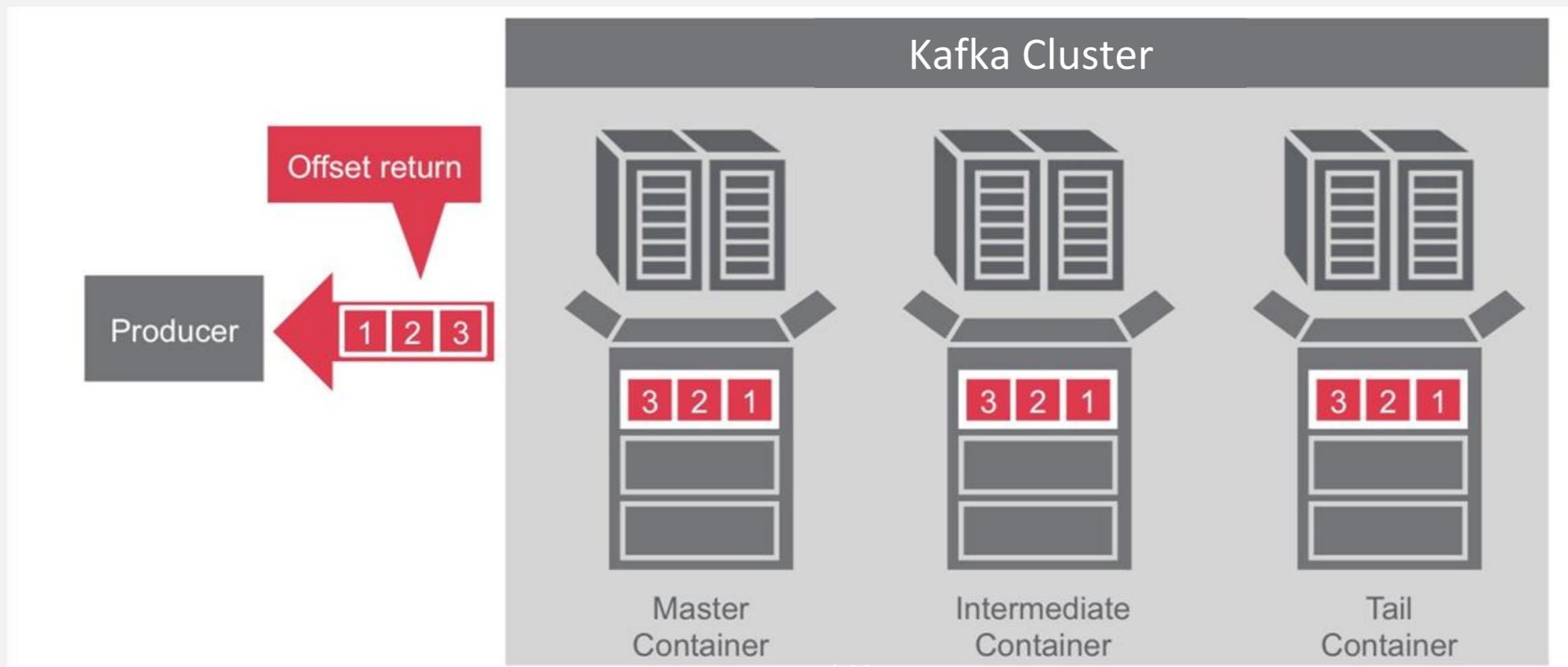
Partitions Are Replicated

- Replication rules are controlled at the volume level



Partitions Are Replicated

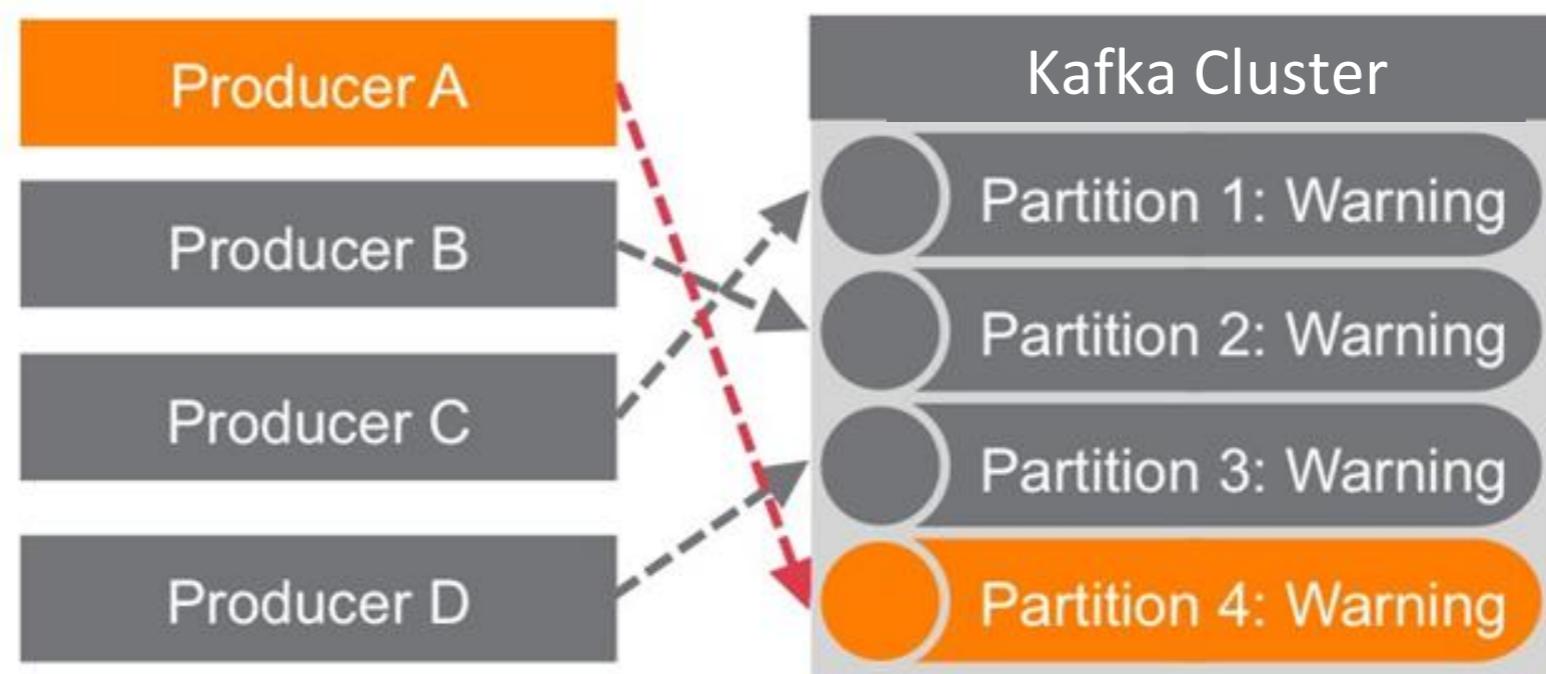
- Server returns message offset IDs



Parallelism When Publishing

- How do servers choose which partition to publish to?

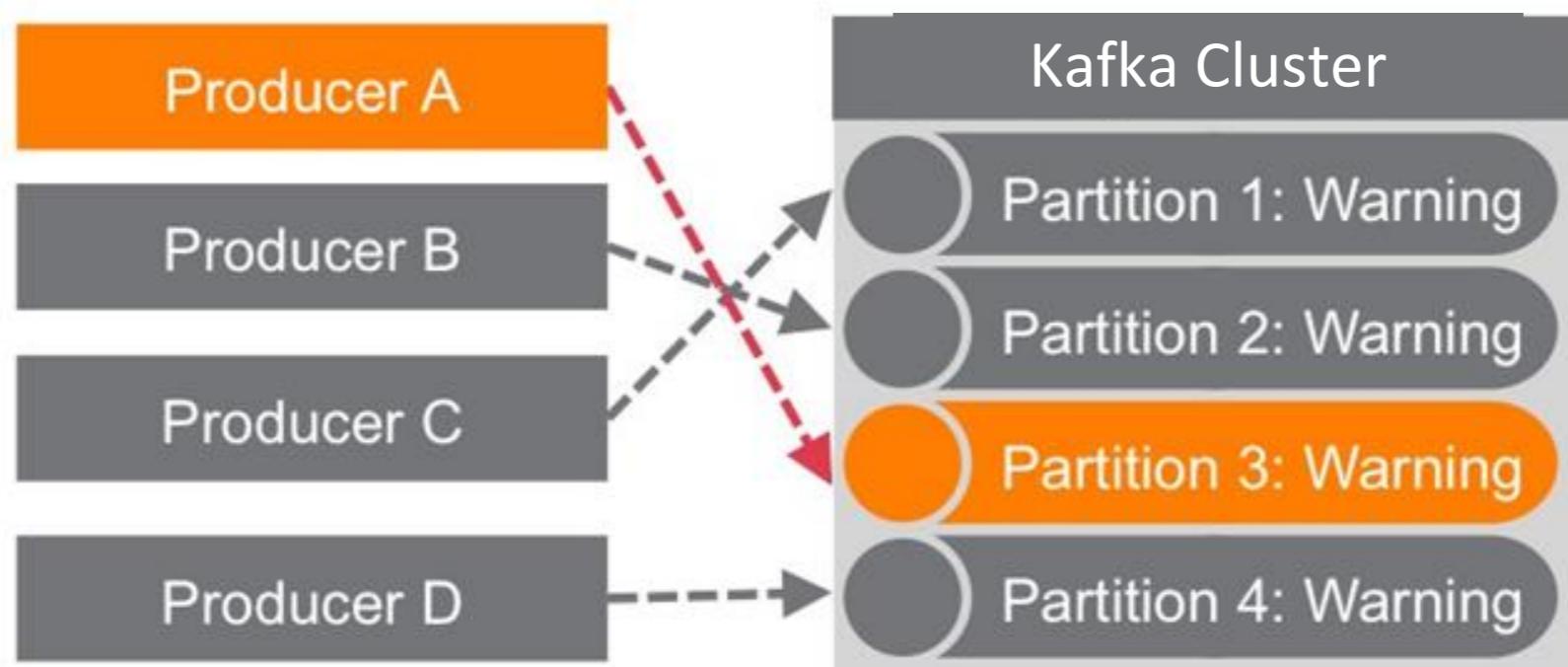
Example: Producer “A” specifies Partition ID=4



Parallelism When Publishing

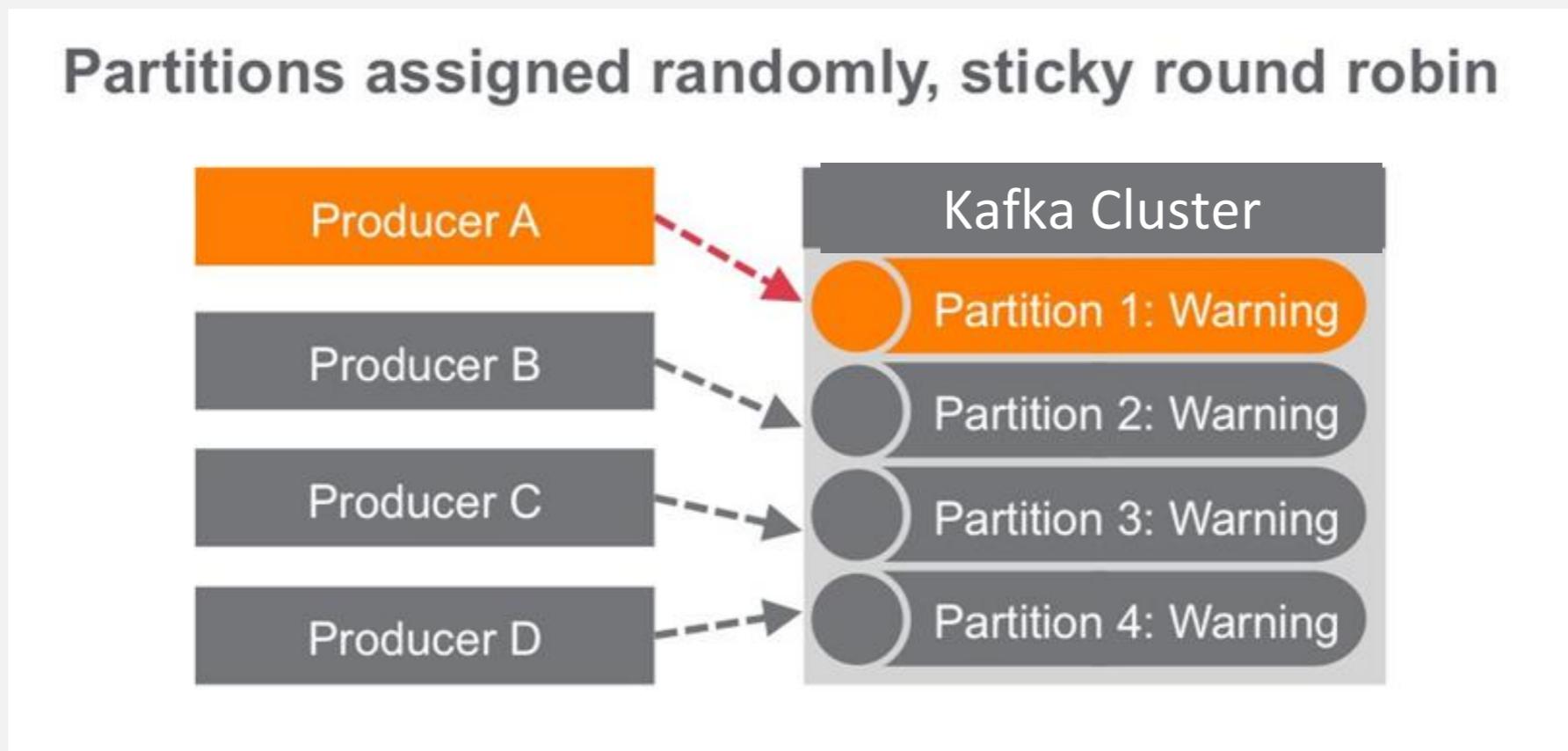
- How do servers choose which partition to publish to?

Example: Producer “A” specifies key



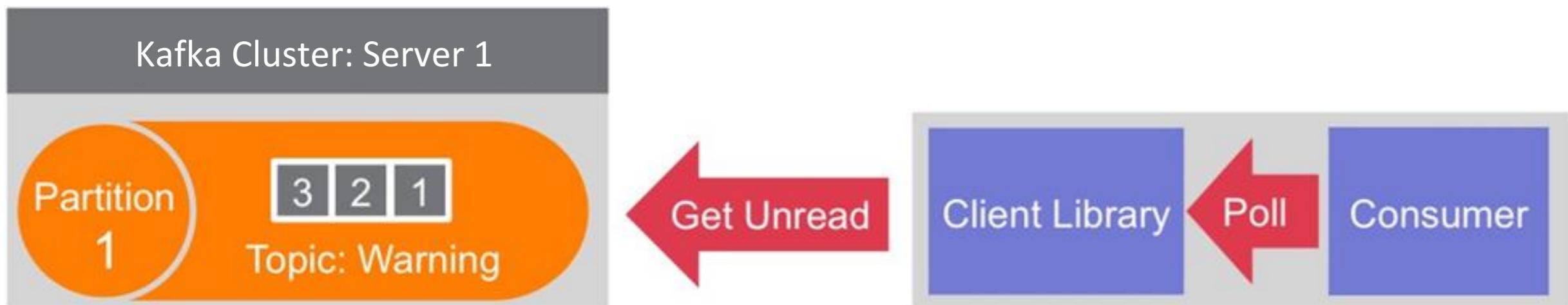
Parallelism When Publishing

- How do servers choose which partition to publish to?



How are Messages Read?

- Consumers read at their own pace
- Consumers polls topics it subscribes to
- Client library requests unread messages



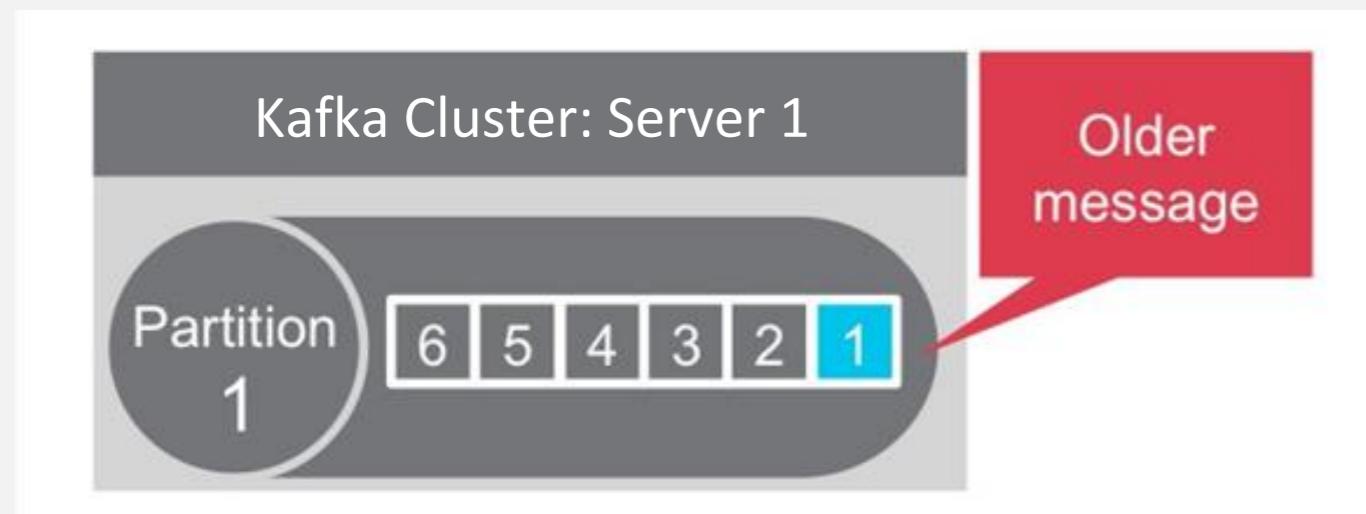
How are Messages Read?

- Primary partition returns unread messages
- Messages remain on the partition, available to other consumers
- Client library passes messages to consumer



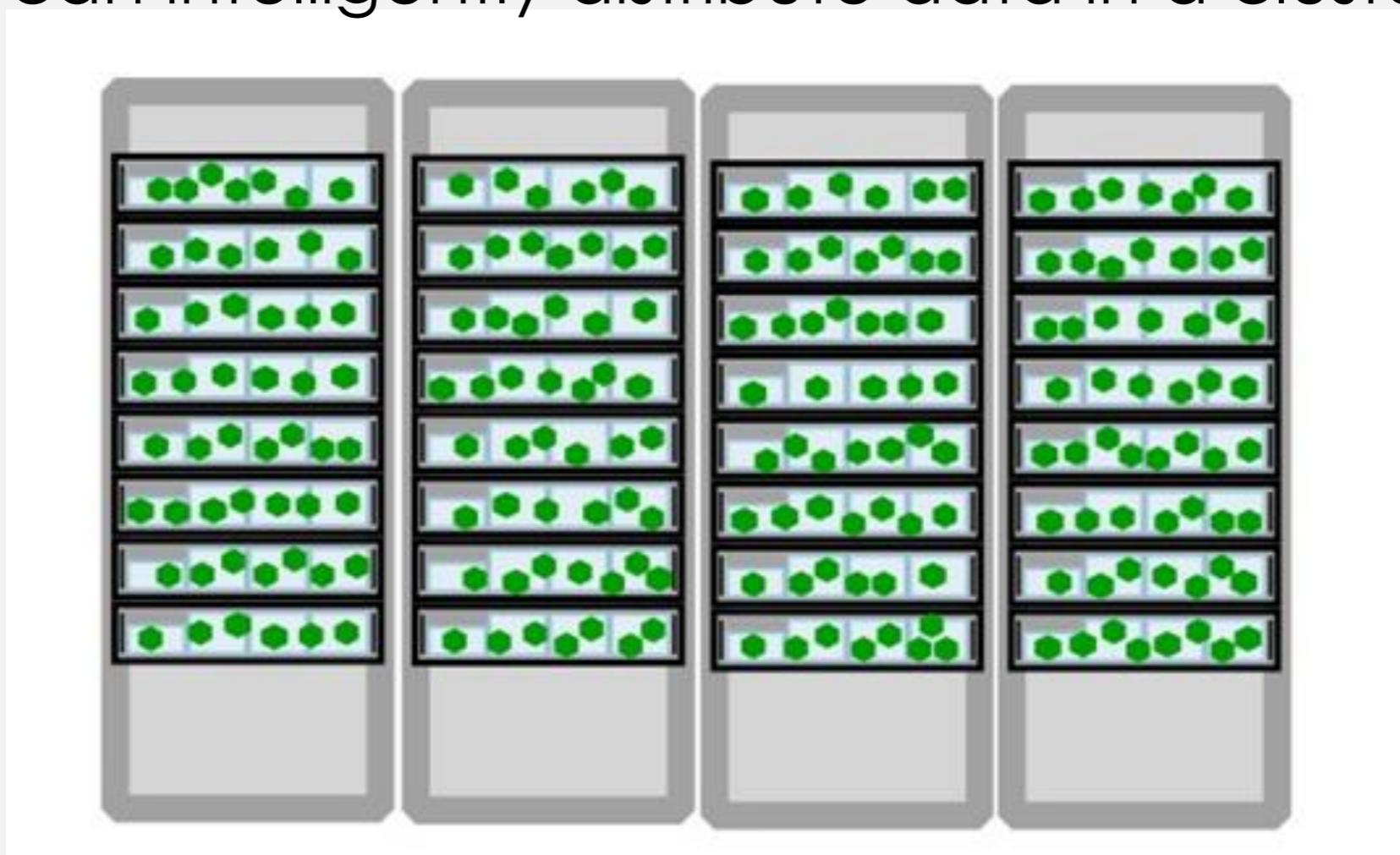
When Are Messages Deleted?

- Older messages deleted based on time to live
- Time-to-live set when stream is created
- Messages don't expire if time-to-live is zero
- Expired messages deleted automatically



Partitions Are Automatically Distributed

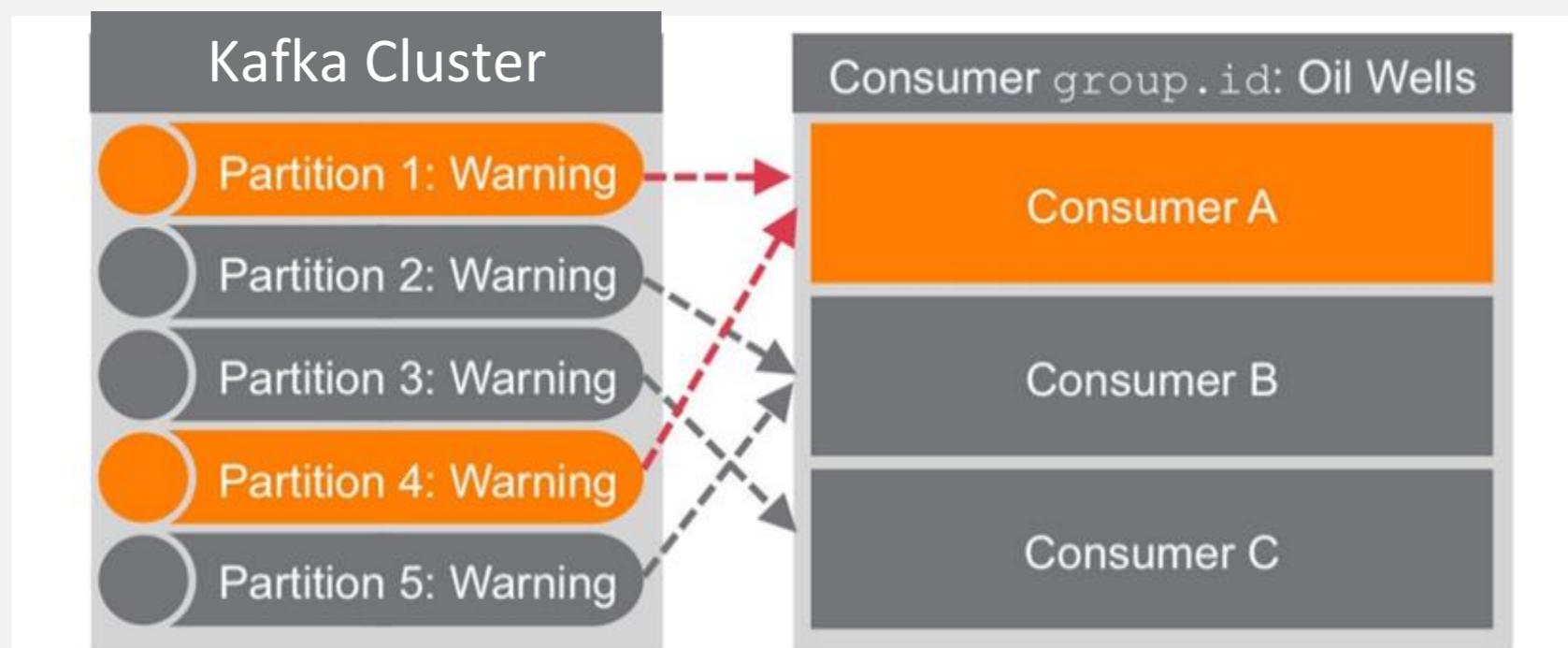
- Kafka can intelligently distribute data in a cluster



Parallelism When Reading

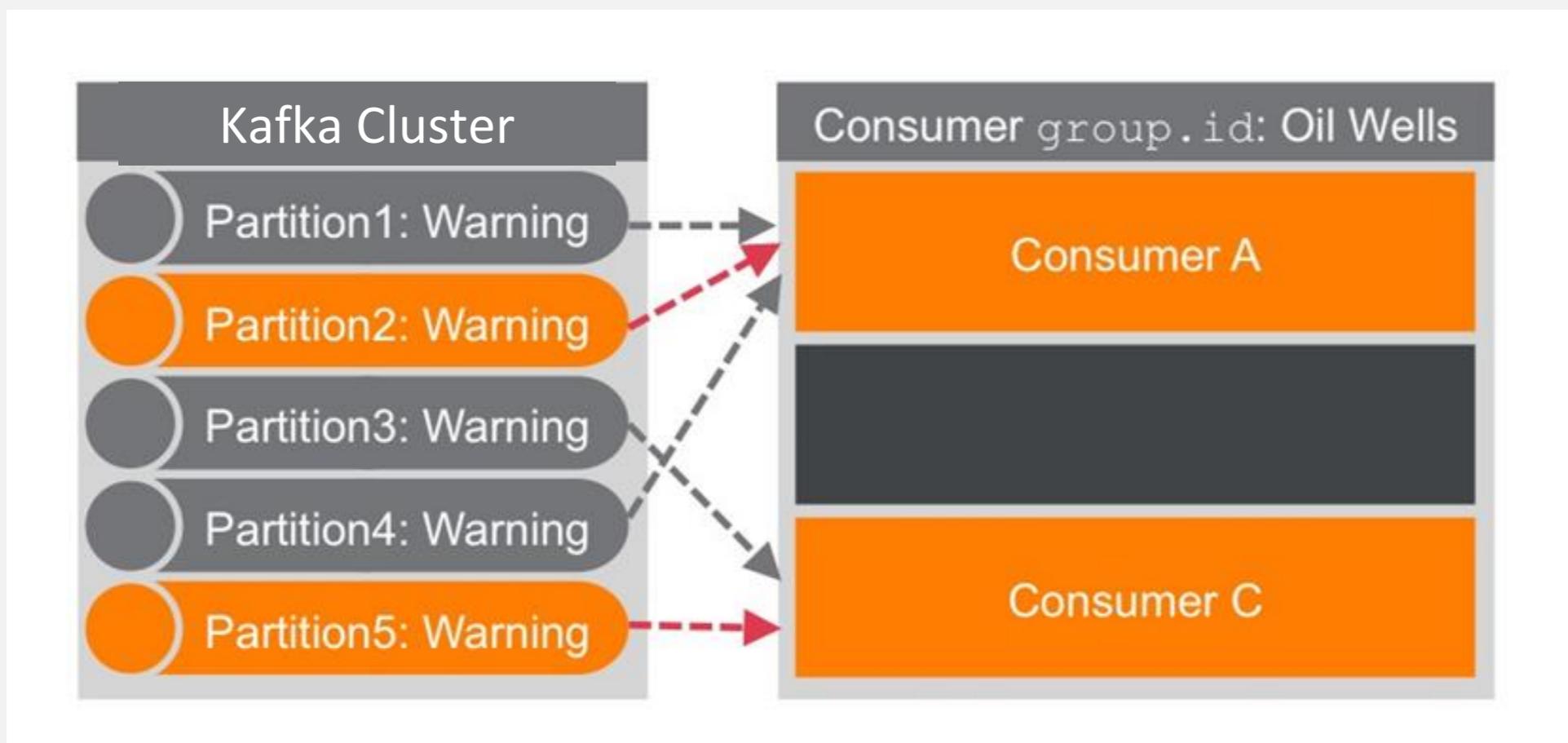
To read messages in parallel:

- create consumer groups
- consumers with same group.id
- partitions assigned dynamically round-robin



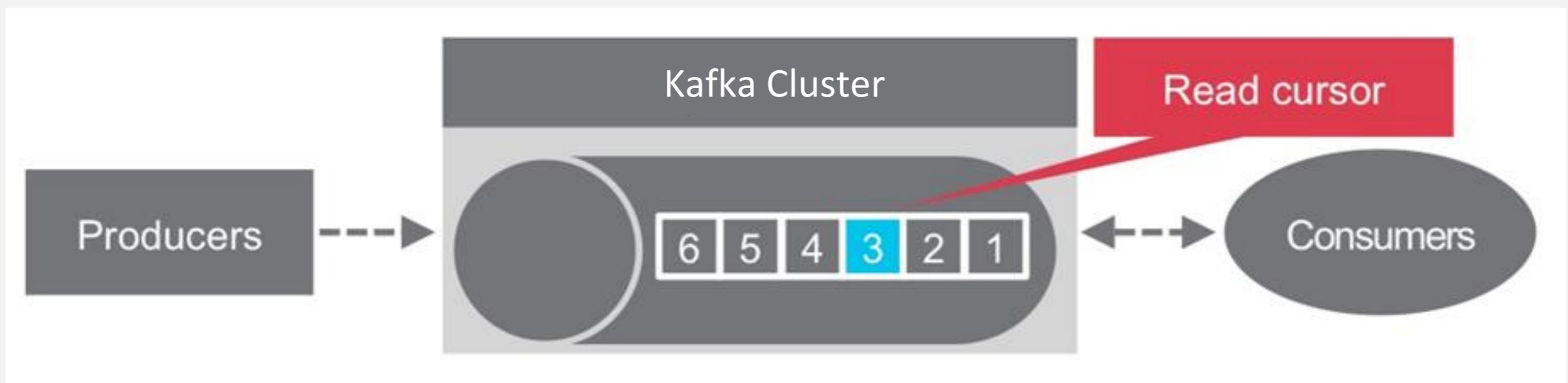
Partitions Re-Assigned Dynamically

- If consumer goes offline, partitions re-assigned



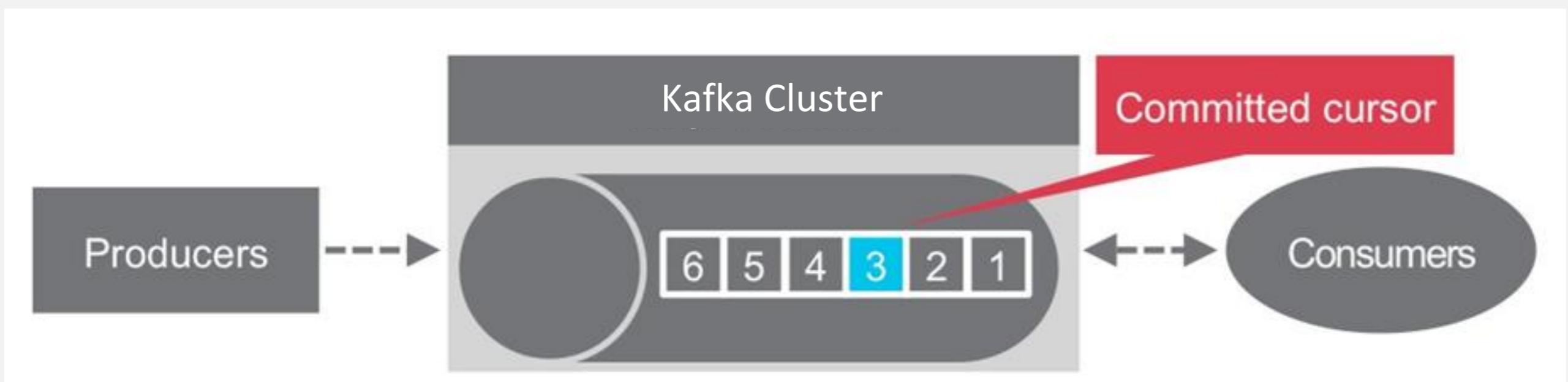
Saving of Cursor Position: Read Cursor

- Cursors keep track of messages read
- Read cursor: offset ID of most recent message sent to consumer

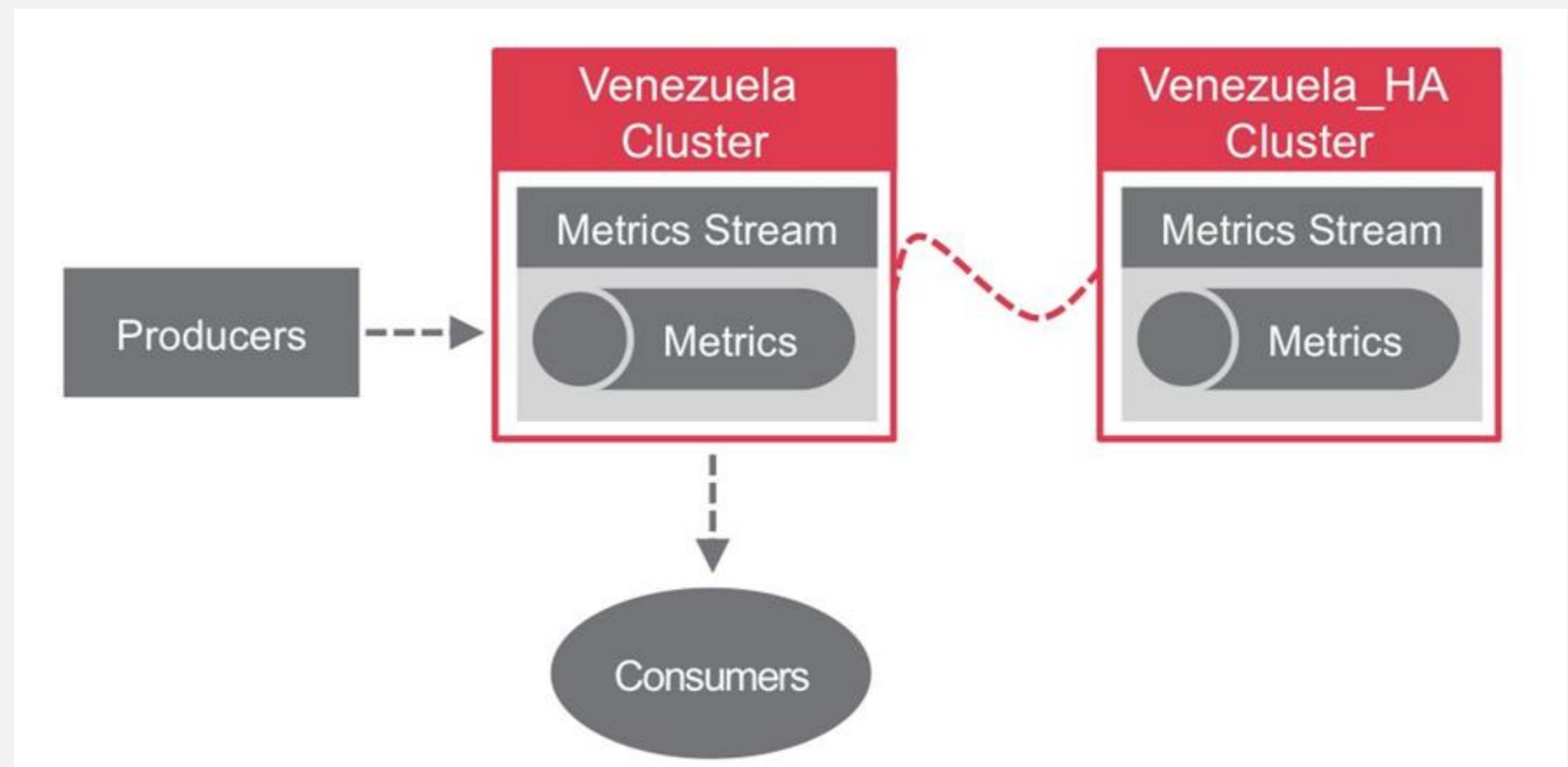


Saving of Cursor Position: Committed Cursor

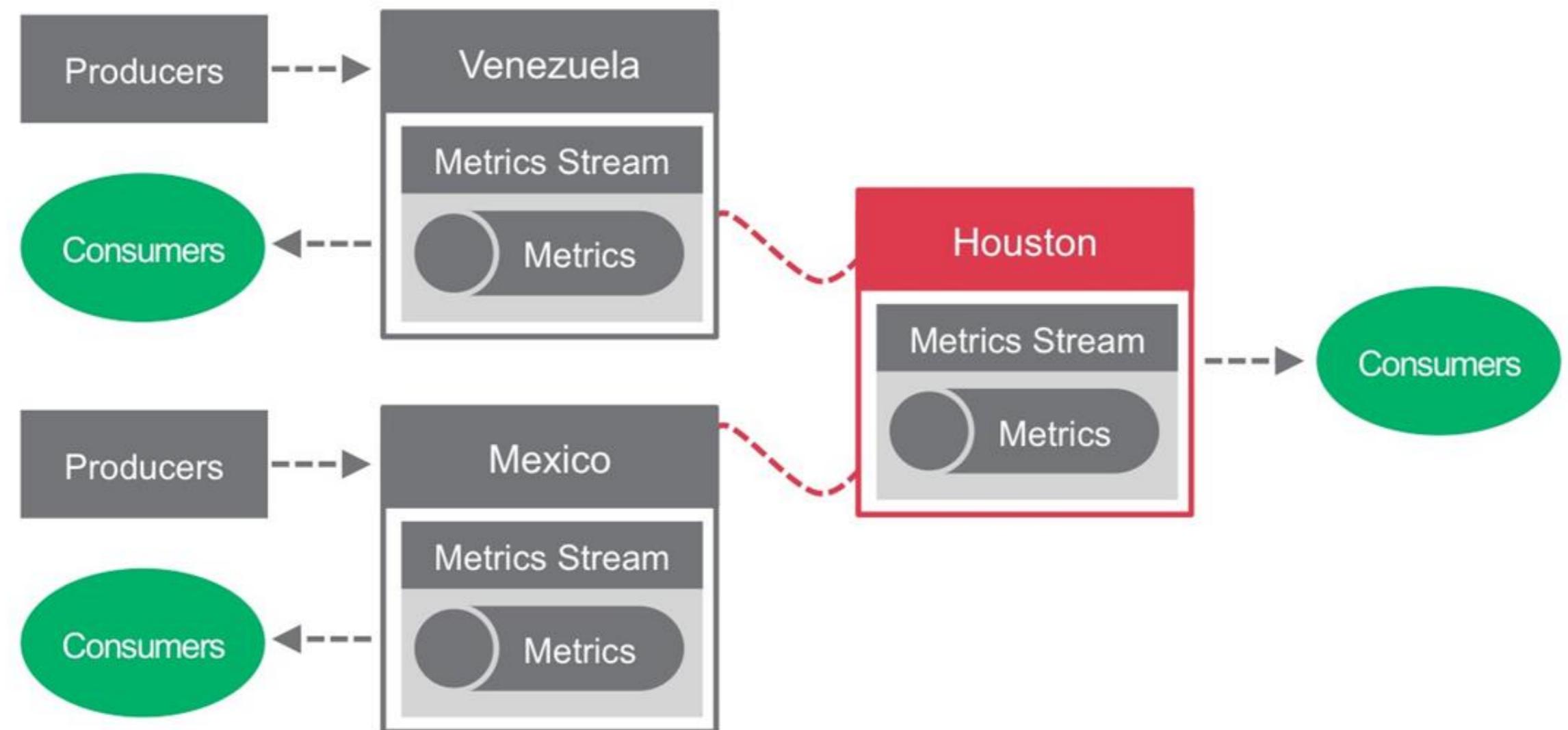
- Committed cursor: saved current position of read cursor



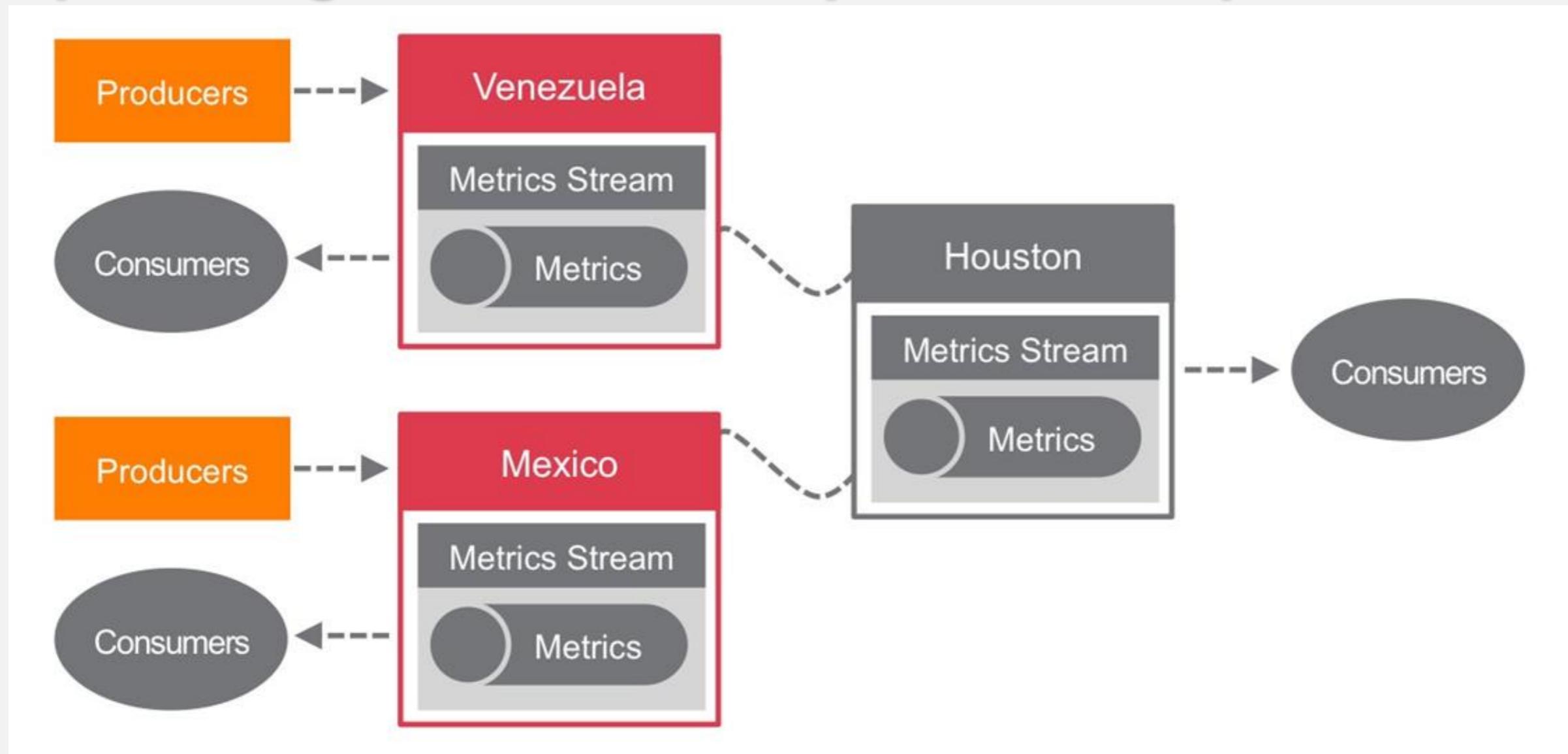
Replicating Streams: Master-Slave Replication



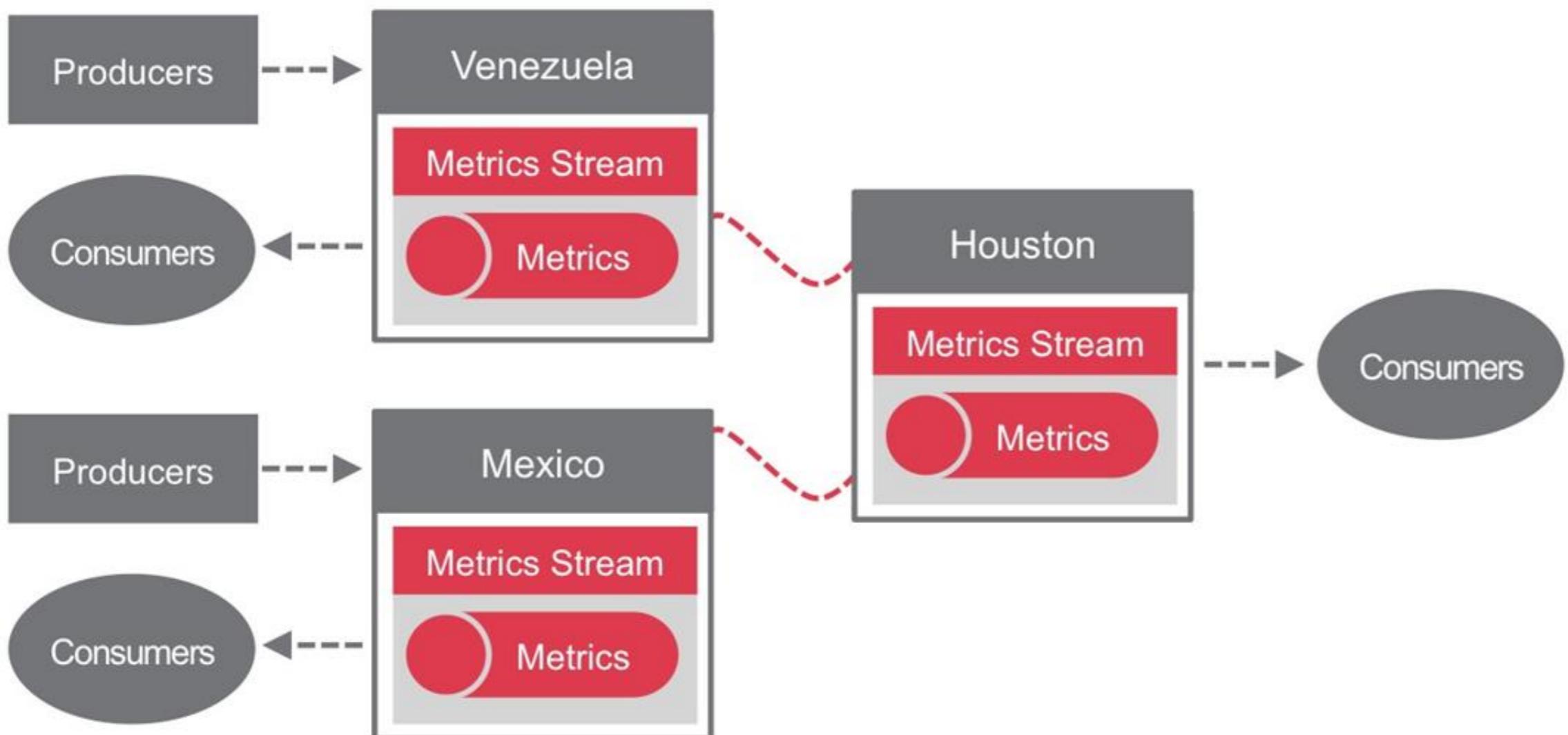
Replicating Streams: Many-to-one Replication



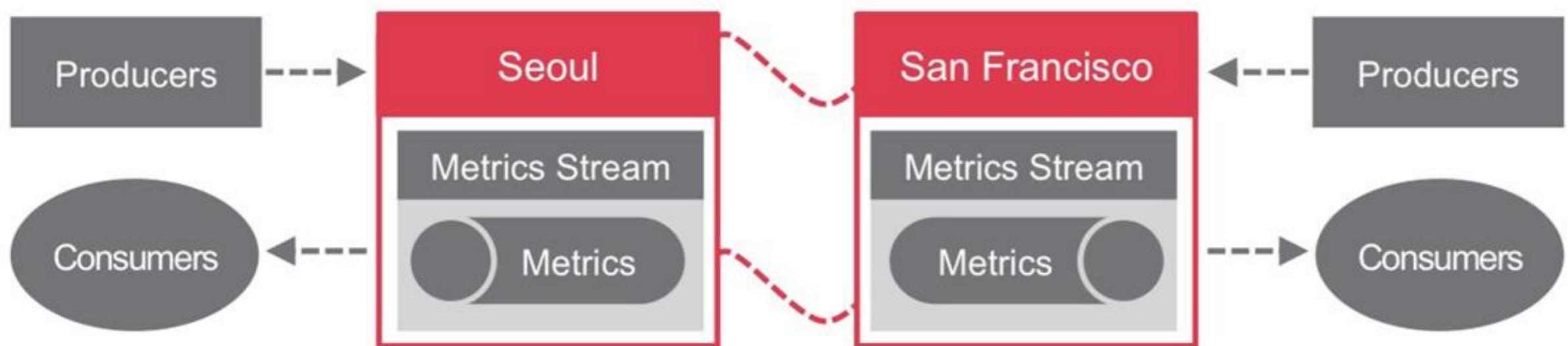
Replicating Streams: Many-to-one Replication



Replicating Streams: Many-to-one Replication



Replicating Streams: Multi-Master Replication



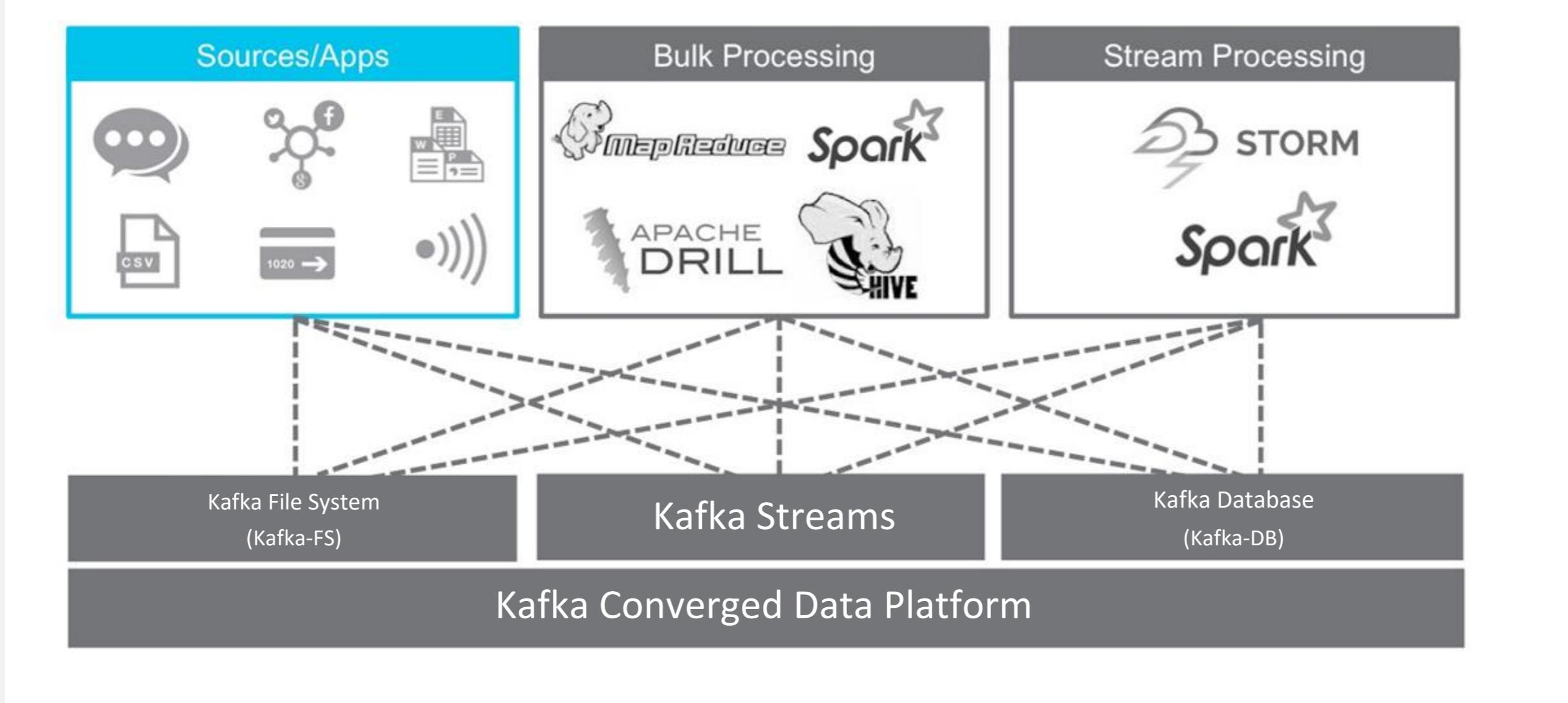
Streams Replication



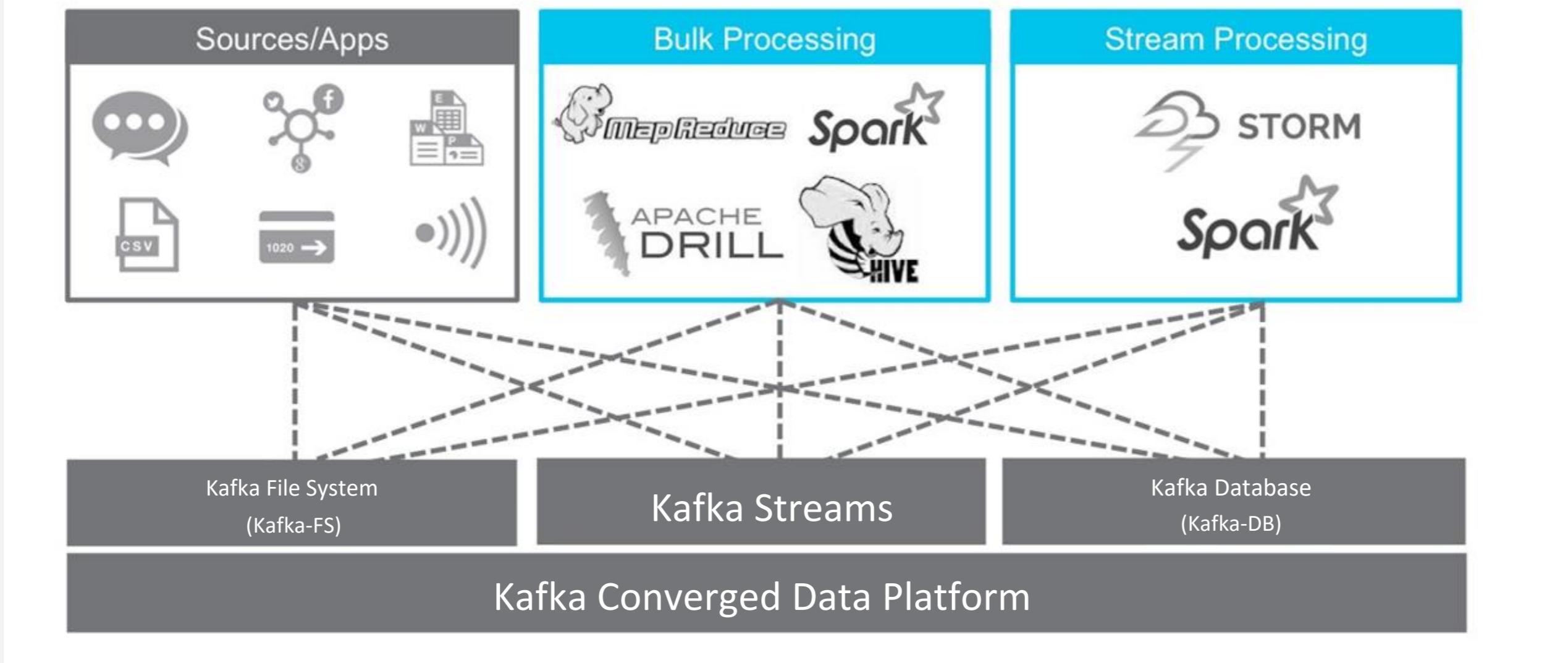
Learning Goals

- Define core components of Kafka
- Summarize the life of a message in Kafka
- **Explain how Kafka fits in the complete data architecture**

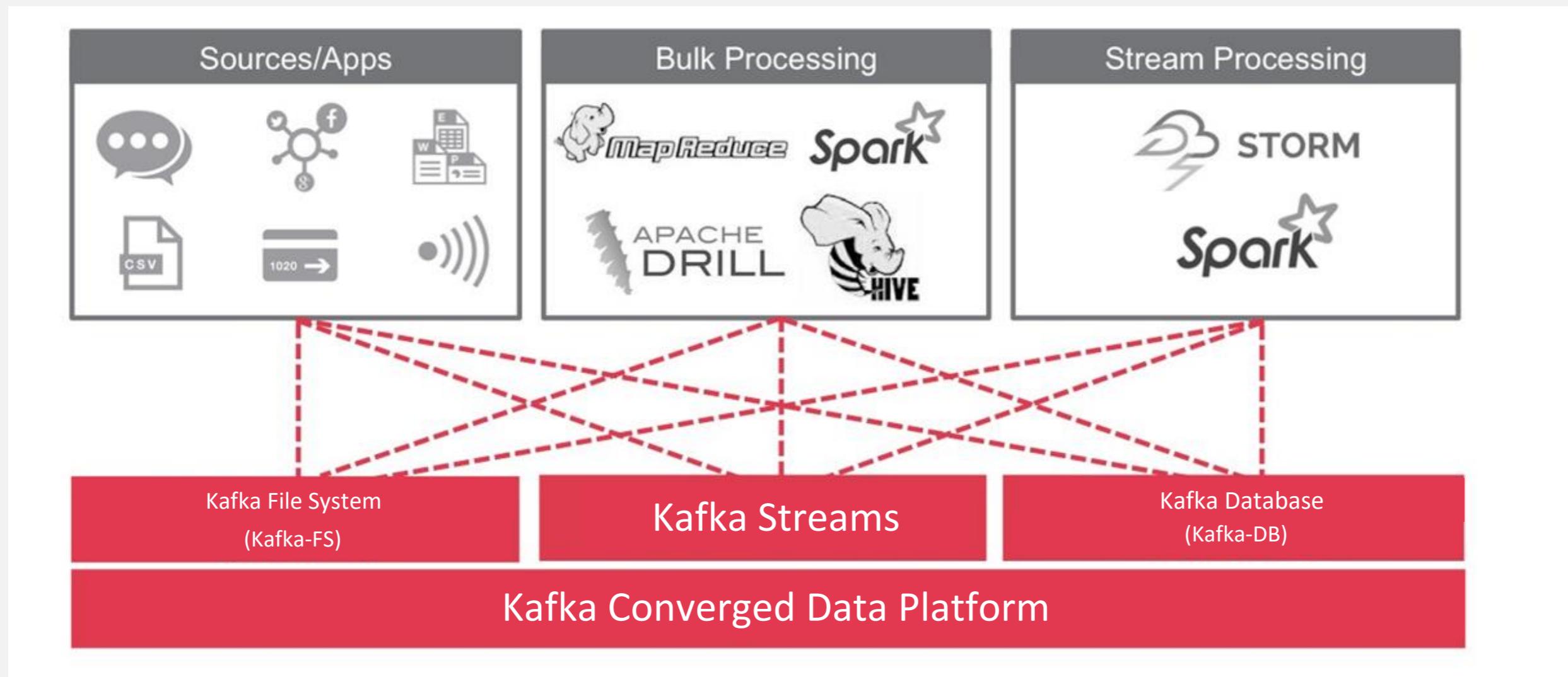
Build a complete Data Architecture



Build a complete Data Architecture



Build a complete Data Architecture

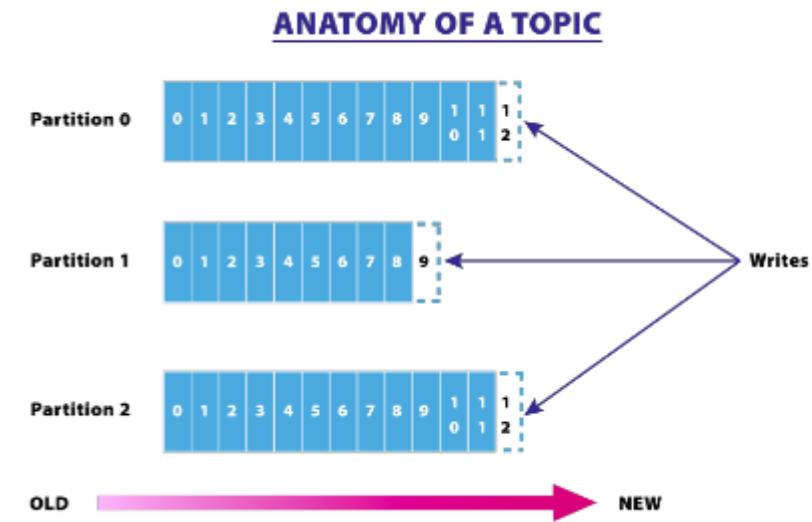


**Congratulations! You have completed
Lesson : Kafka Architecture.**

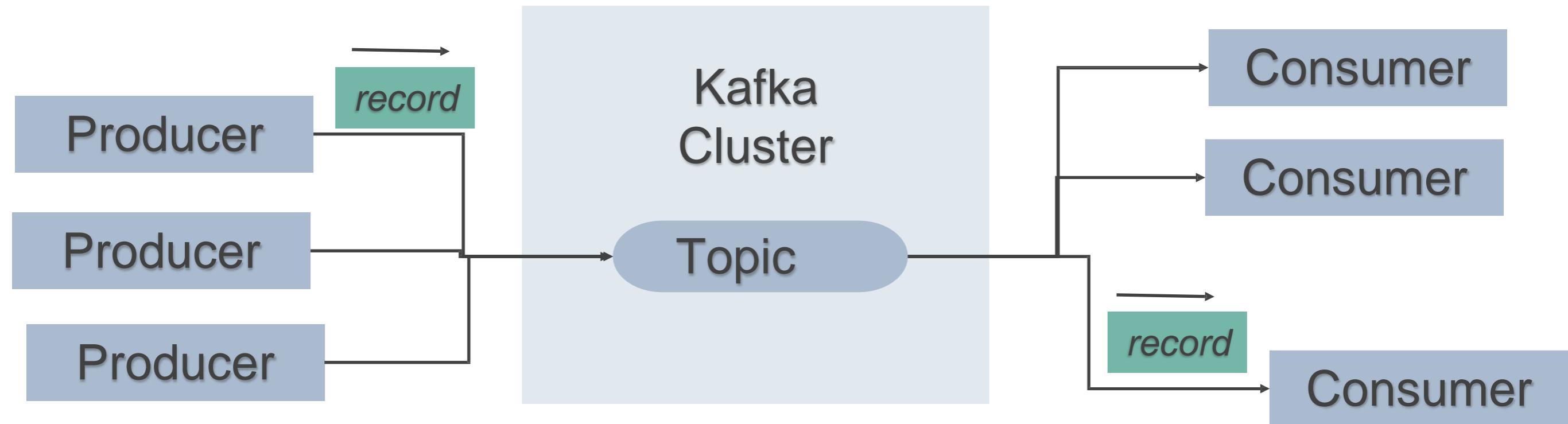
Kafka Fundamentals



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



Kafka: Topics, Producers, and Consumers



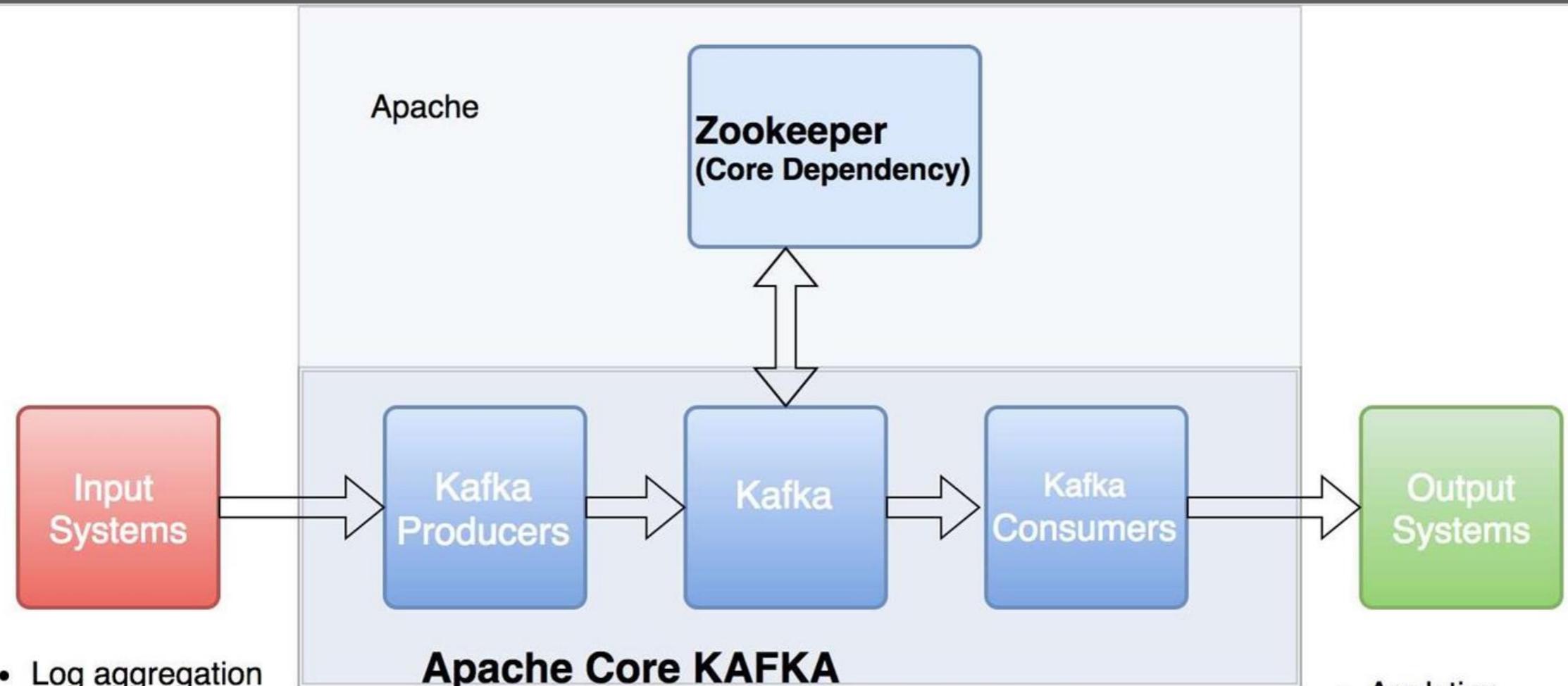
Apache Kafka - Core Kafka



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



Apache Kafka



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

Not part of core

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

Apache Kafka Core

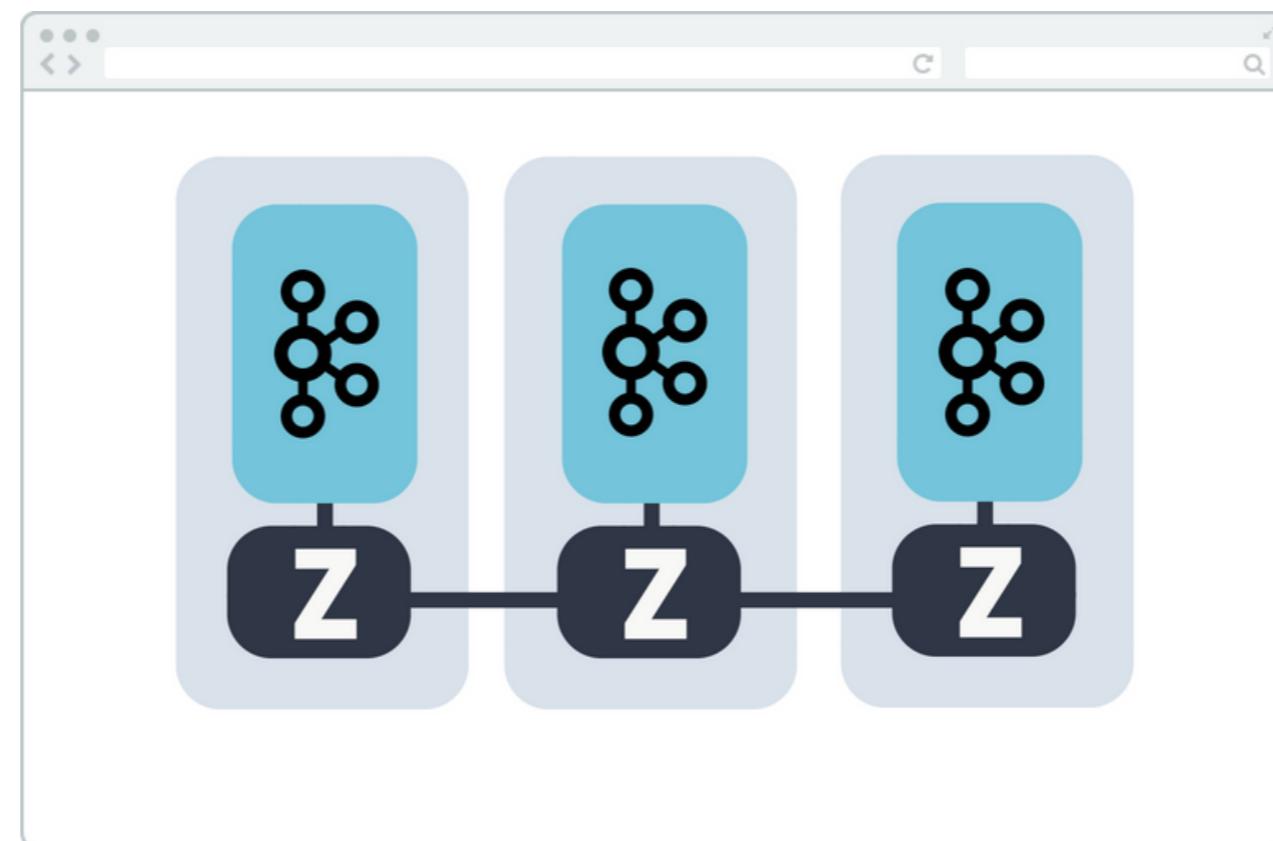
- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

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

Kafka needs Zookeeper



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

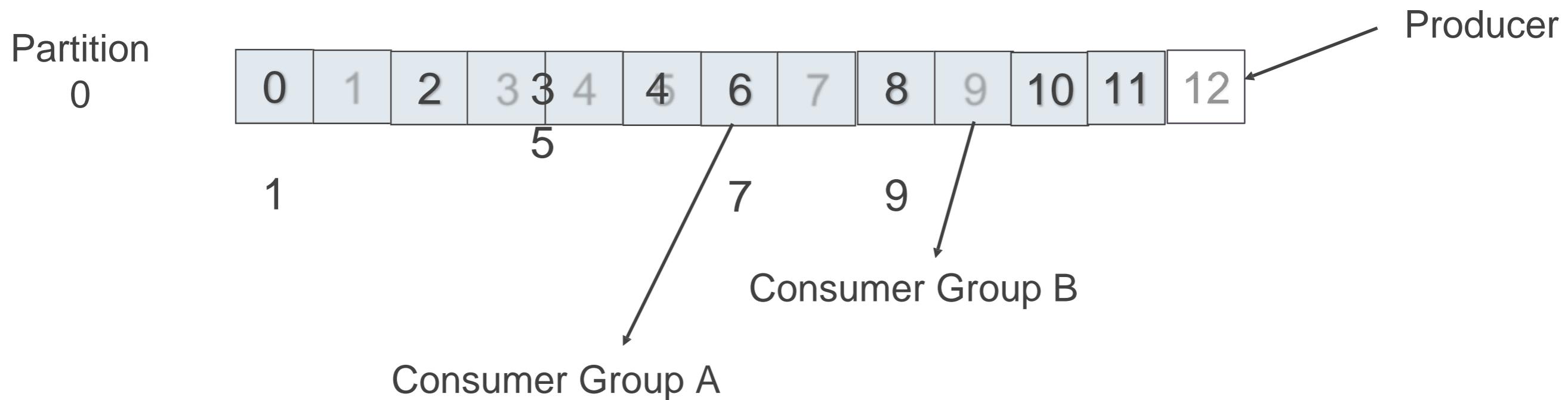


Kafka Producer/Consumer Details



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

Kafka Topic Partition, Consumers, Producers



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

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

Kafka Scale and Speed



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

Kafka Brokers



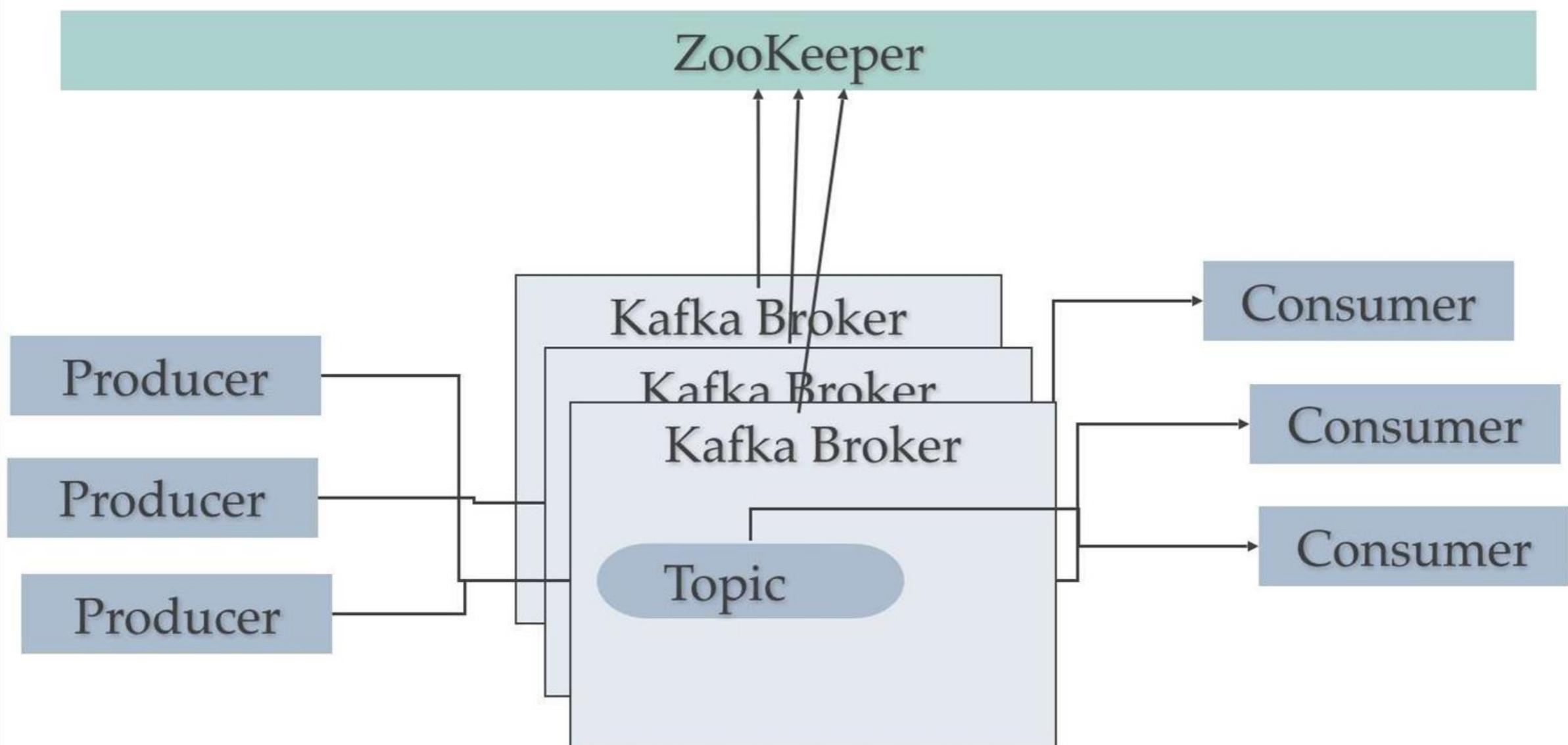
- ❖ Kafka Cluster is made up of multiple Kafka Brokers
- ❖ Each Broker has an ID (number)
- ❖ Brokers contain topic log partitions
- ❖ Connecting to one broker bootstraps client to entire cluster
- ❖ Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

Kafka Cluster, Failover, ISRs



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

ZooKeeper does coordination for Kafka Cluster



Failover vs. Disaster Recovery



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



Kafka Review



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



Kafka Consumers

Load balancing
consumers Failover for
consumers
Offset management per consumer
group

[Kafka Consumer Architecture](#)

Kafka Consumer Groups



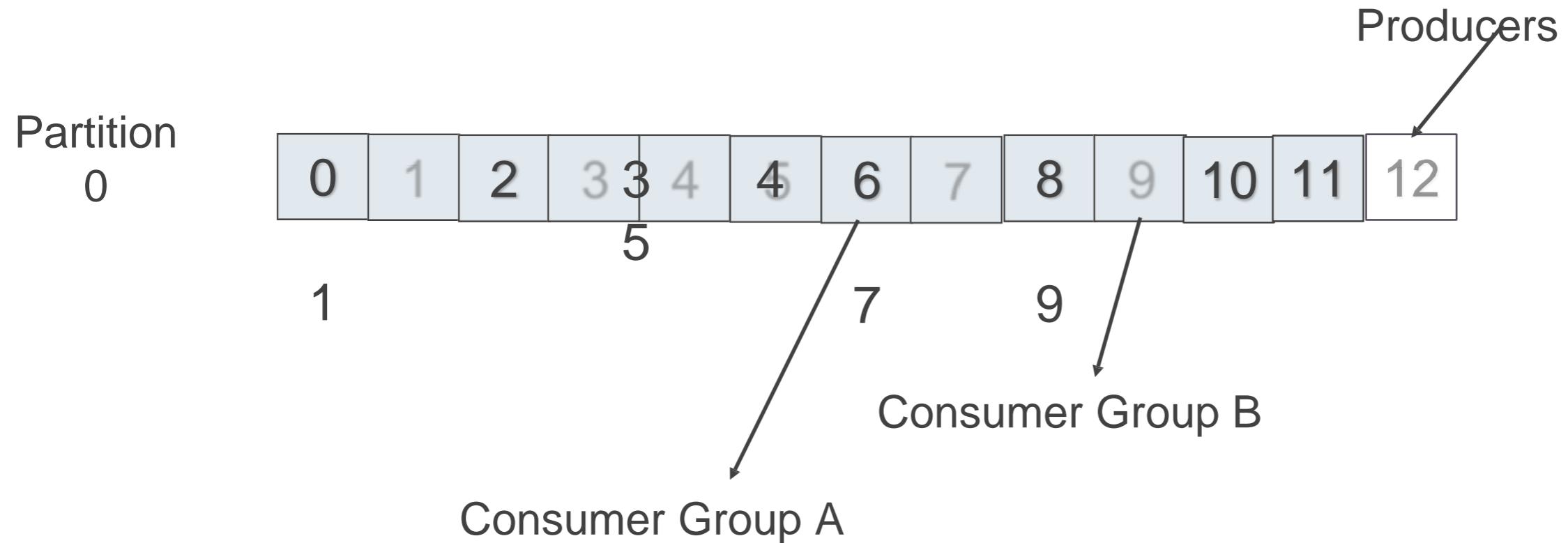
- ❖ Consumers are grouped into a ***Consumer Group***
 - ❖ ***Consumer group*** has a unique id
 - ❖ Each ***consumer group*** is a subscriber
 - ❖ Each ***consumer group*** maintains its own offset
 - ❖ Multiple subscribers = multiple consumer groups
 - ❖ Each has different function: one might delivering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

Kafka Consumer Load Share



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

Kafka Consumer Groups



Consumers remember offset where they left off.

Consumers groups each have their own offset per partition.

Kafka Consumer Groups Processing



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

Kafka Consumer Failover



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

Kafka Consumer Offsets and Recovery

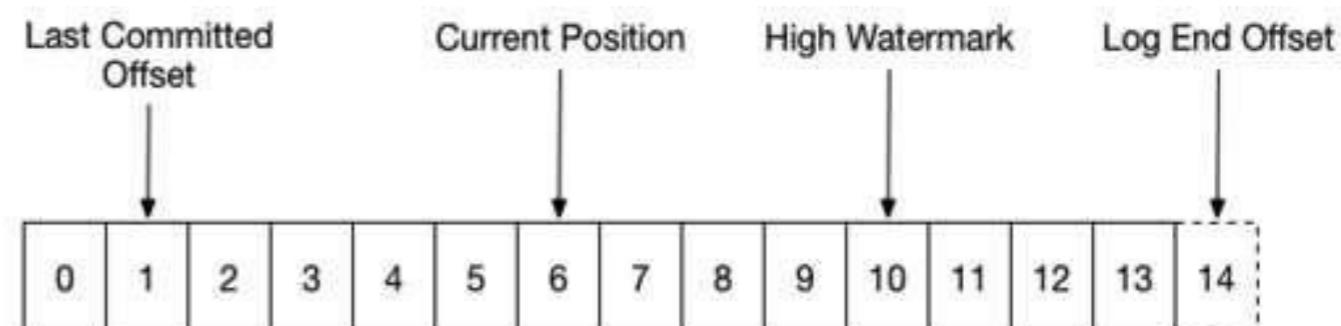


- ❖ Kafka stores offsets in topic called “`__consumer_offset`”
 - ❖ Uses Topic Log Compaction
- ❖ When a consumer has processed data, it should commit offsets
- ❖ If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in “`__consumer_offset`”

Kafka Consumer: What can be consumed?



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

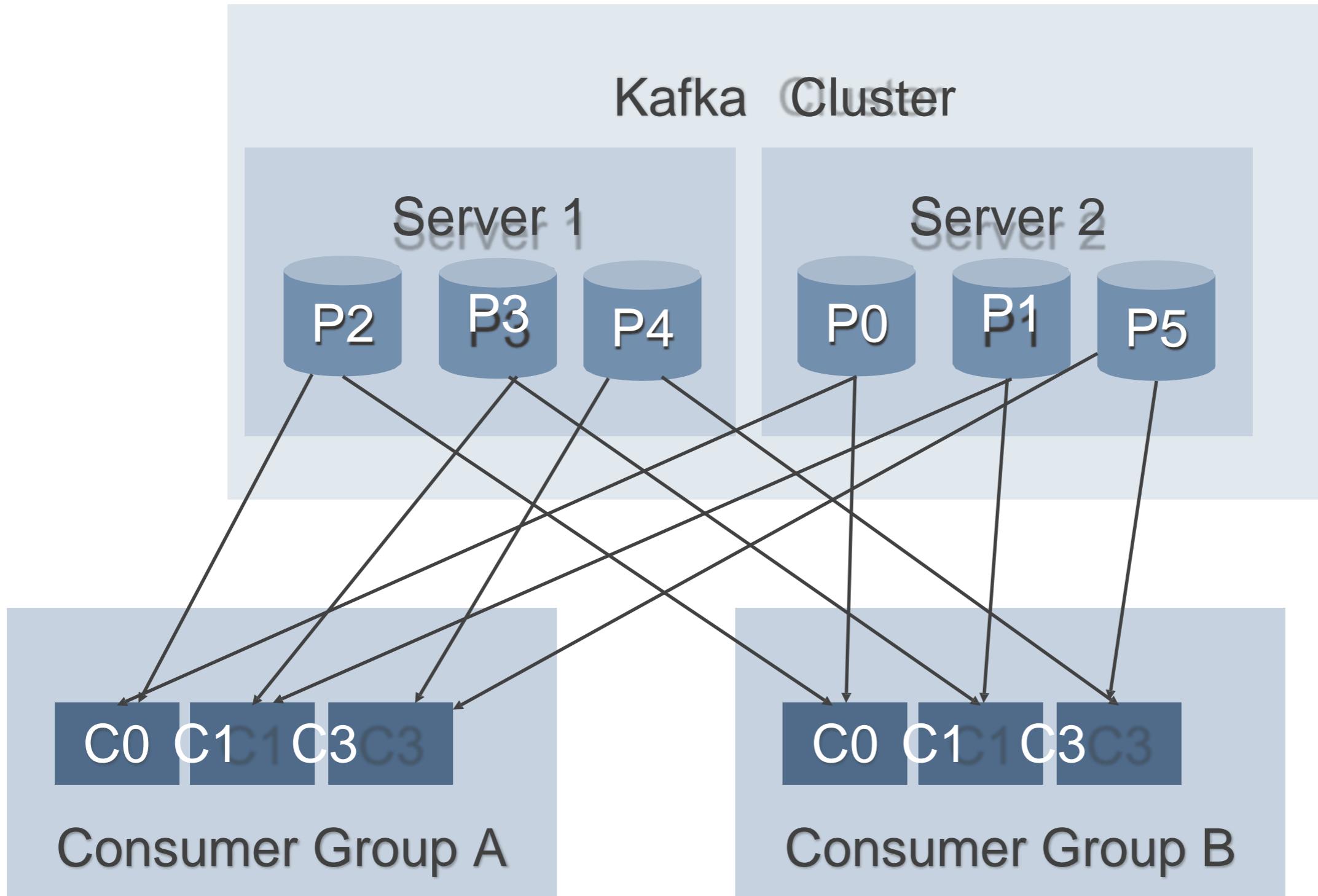


Consumer to Partition Cardinality



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

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



Multi-threaded Consumers



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



Consumer Review



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



Using Kafka Single Node

Using Kafka Single Node

Run ZooKeeper, Kafka
Create a topic
Send messages from command line
Read messages from command line

[Tutorial Using Kafka Single Node](#)



Run Kafka

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

Run ZooKeeper



run-zookeeper.sh x

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

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



Run Kafka Server

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

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

Create Kafka Topic



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

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



List Topics

```
> list-topics.sh ×
```

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

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



Run Kafka Producer

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



Run Kafka Consumer

start-consumer-console.sh x

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

Running Kafka Producer and Consumer



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

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

? Kafka Single Node Review



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



Use Kafka to send and receive messages

Lab Use Kafka

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

Complete Lab 1



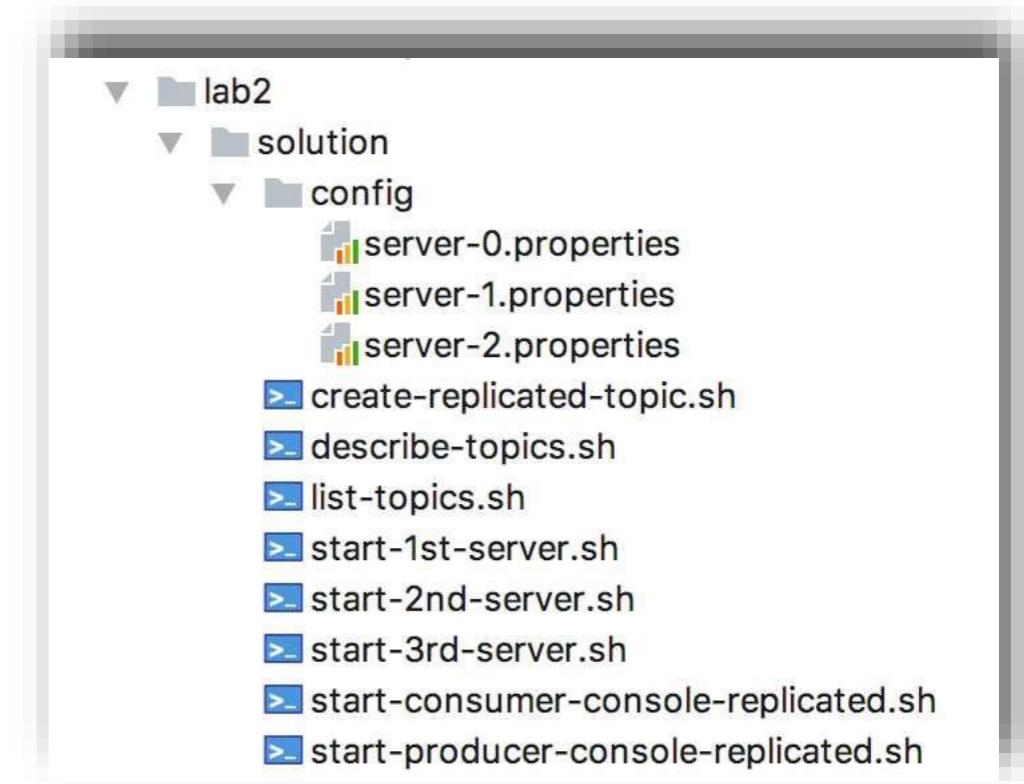
Using Kafka Cluster and Failover

Demonstrate Kafka Cluster
Create topic with replication
Show consumer failover
Show broker failover
[Kafka Tutorial Cluster and
Failover](#)



Objectives

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



Running many nodes



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

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

Create three new server- n.properties files



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

Modify server-x.properties



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

server-1.properties
1 broker.id=1
2 port=9093
3 log.dirs=./logs/kafka-1

server-2.properties
1 broker.id=2
2 port=9094
3 log.dirs=./logs/kafka-2
```

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

Create Startup scripts for three Kafka servers



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

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

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



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

- ❖ Passing properties files from last step



Run Servers

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

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

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

Create Kafka replicated topic my-failsafe-topic



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

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

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

Start Kafka Consumer



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

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

Start Kafka Producer



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

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

- ❖ Start producer
- ❖ Pass list of Kafka Brokers

Kafka 1 consumer and 1 producer running



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

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

Start a second and third consumer



```
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
message 1
message 2
message 3
Last login: Mon May 15 11:28:21 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are thin
How are you?
Hi mom
message 1
message 2
message 3
Last login: Mon May 15 11:35:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
Last login: Mon May 15 11:35:35 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

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

Running consumers in same group



start-consumer-console-replicated.sh ×

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

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

Start up three consumers again

A screenshot of a terminal window showing four separate terminal sessions. Each session is running a script to start a Kafka consumer. The messages are being distributed across three consumers (m3, m5, m6) in the first two sessions, and across three consumers (m1, m4, m7) in the last two sessions, demonstrating load balancing.

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

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



Consumer Failover

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

Create Kafka Describe Topic



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

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

Use Describe Topics



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

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

Test Broker Failover: Kill 1st server



Kill the first server

```
~/kafka-training/lab2/solution
[$ kill `ps aux | grep java | grep server-0.properties | tr -s " " " | cut -d " " -f2`
```

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

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

Show Broker Failover Worked



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

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



Kafka Cluster Review



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



Use Kafka to send and receive messages

Complete Lab 1.2: Kafka Multiple Nodes

Use a Kafka Cluster to
replicate a Kafka topic log



Kafka Ecosystem

Kafka Connect
Kafka Streaming
Kafka Schema Registry
Kafka REST

Kafka Universe



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

What comes in Apache Kafka Core?



Apache Kafka Core Includes:

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

What comes in Kafka Extensions?



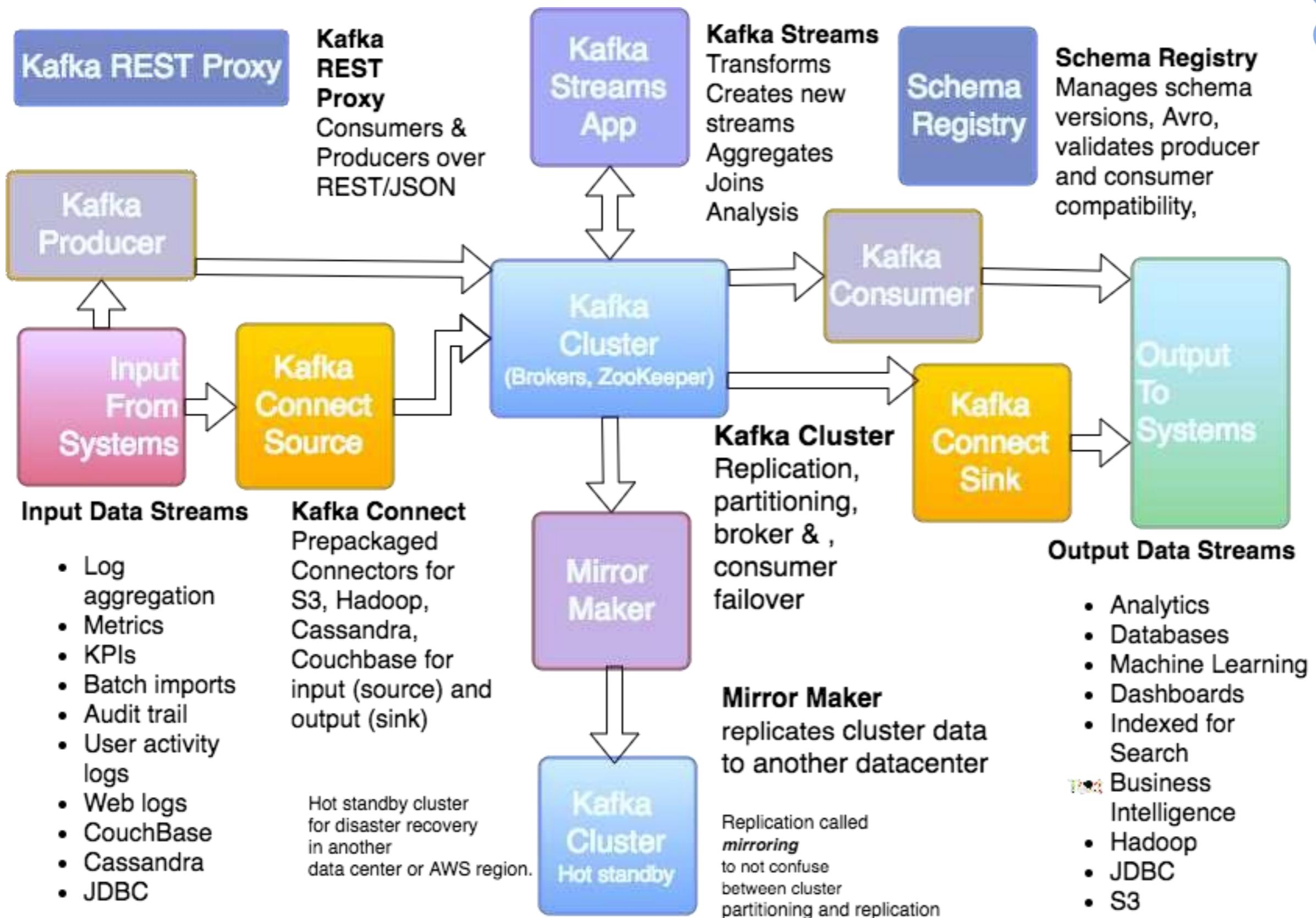
Confluent.io:

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

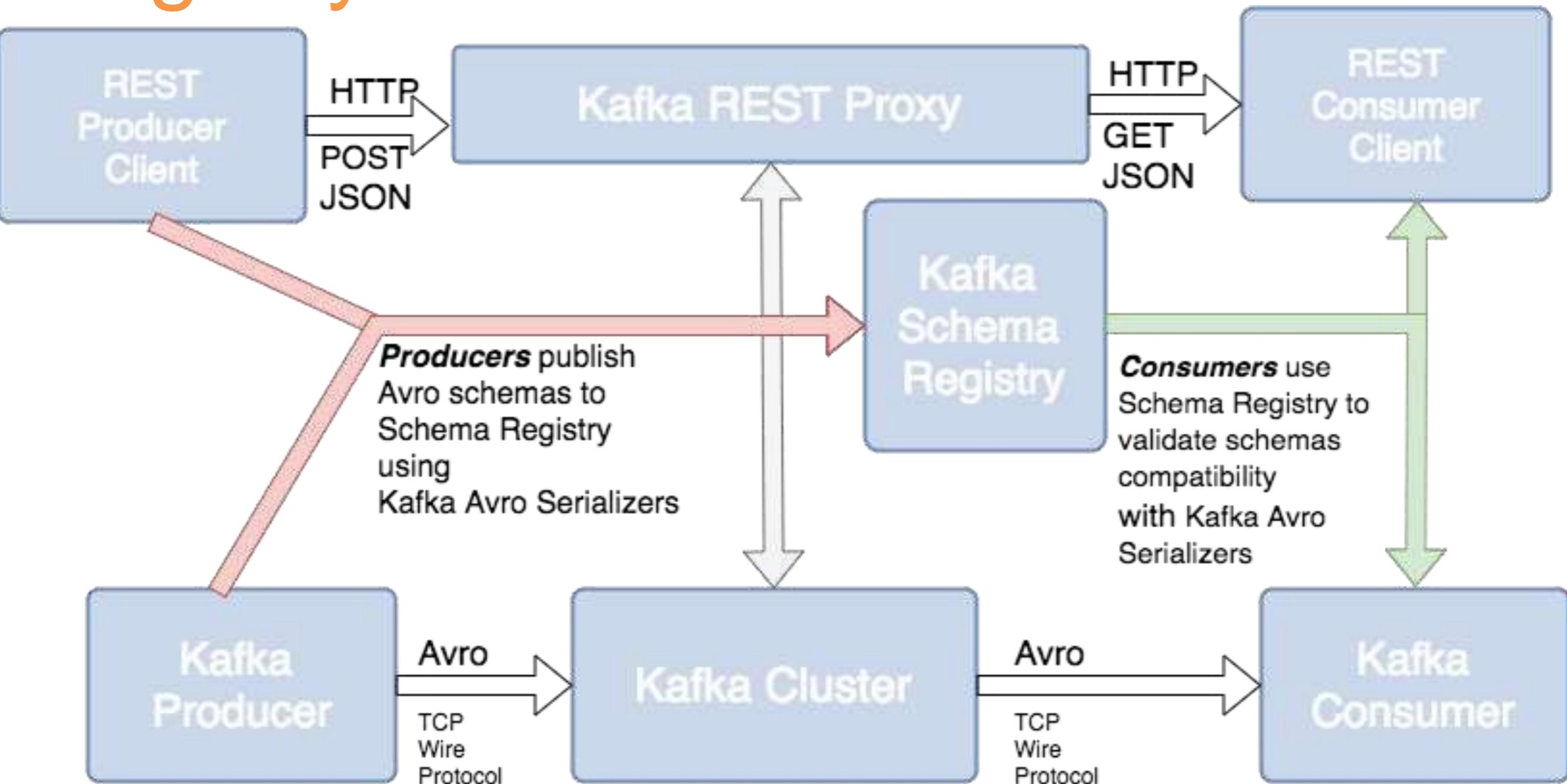
Community of Kafka Connectors from 3rd parties and [Confluent](#)



Kafka Universe



Kafka REST Proxy and Schema Registry

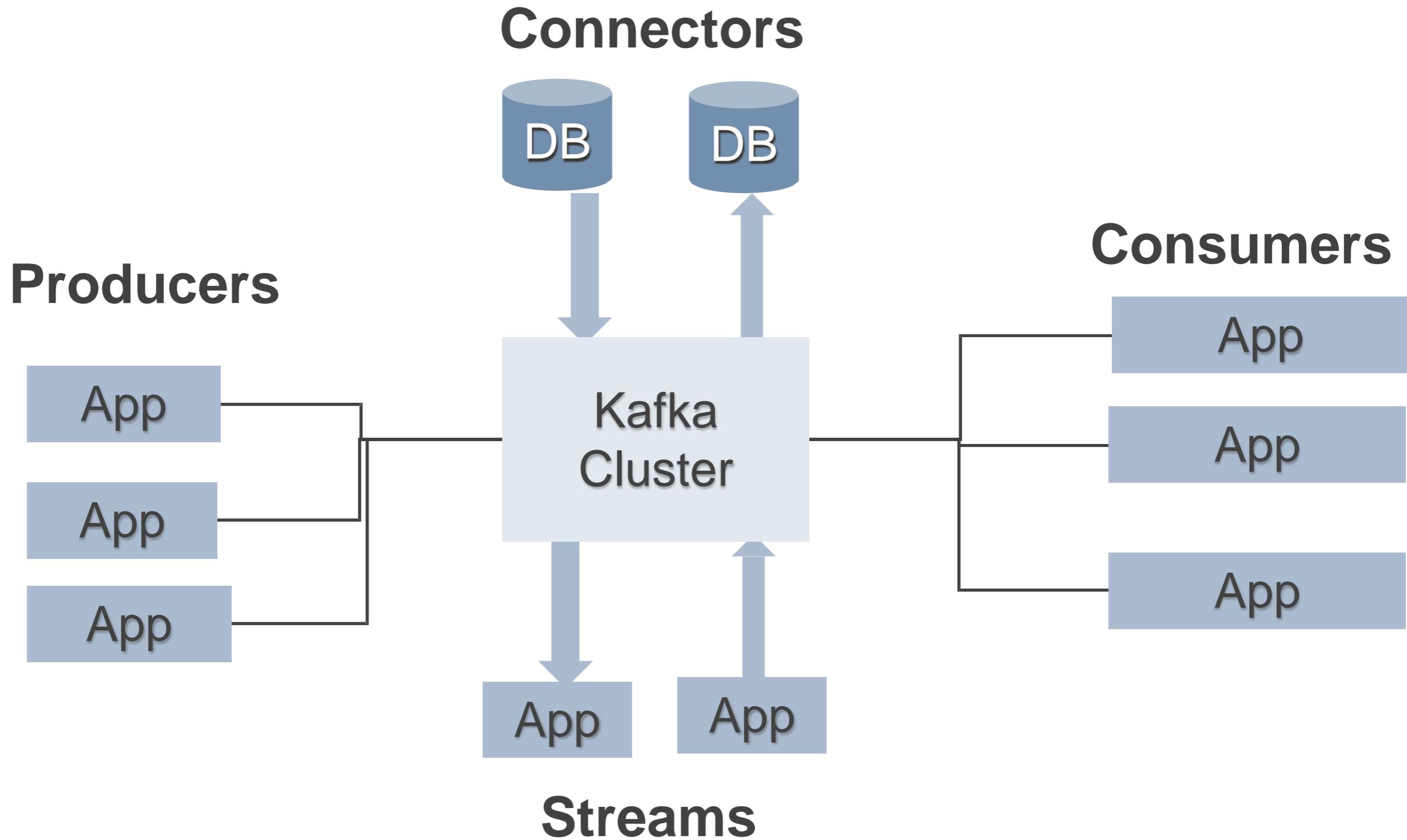


Kafka Stream : Stream Processing



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

Kafka Connectors and Streams



? Kafka Ecosystem review



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



Working with Kafka Consumer – Java Example

3. Kafka Consumer Introduction Java Examples

Working with consumers

Step by step first example

[Creating a Kafka Java Consumer](#)

Objectives Create a Consumer



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

Create Consumer using Topic to Receive Records



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

Common Kafka imports and constants



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

Create Consumer using Topic to Receive Records



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

Process messages from Kafka with Consumer



KafkaConsumerExample.java x

KafkaConsumerExample

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



Consumer poll

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

Running both Consumer then Producer



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

Run KafkaConsumerExample

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

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



Logging

```
logback.xml x

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

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

Try this: Consumers in Same Group



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

Outcome 3 Consumers Load Share



Consumer 0 (key, value, partition, offset)

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

Consumer 1 (key, value, partition, offset)

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

Consumer 2 (key, value, partition, offset)

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

Producer

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

Try this: Consumers in Different Groups



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

Pass Unique Group Id



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



Outcome 3 Subscribers

Consumer 0 (key, value, partition, offset)

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

Producer

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

Consumer 1 (key, value, partition, offset)

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

Which consumer(s) owns partition 10?

How many ConsumerRecords objects did Consumer 0 get?

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

Consumer 2 (key, value, partition, offset)

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

Why does each consumer get the same messages?

Try this: Consumers in Different Groups



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



Modify Consumer

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

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

Add a 100 ms delay to Consumer after poll



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

Modify Producer: Run 10 times, add 30 second delay



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

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

Notice one or more partitions per ConsumerRecords



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



Now run it again but..

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

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

Output from batch size 500



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

Java Kafka Simple Consumer Example Recap



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

? Kafka Consumer Review



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

Complete Lab 2 & 3



Kafka Low Level Architecture

4. Kafka Low-Level Design

Design discussion of Kafka low level design

[Kafka Architecture: Low-Level Design](#)

Kafka Design Motivation Goals



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

Kafka Design Motivation Scale



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

Kafka Design Motivation Use Cases



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

Persistence: Embrace filesystem



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

Big fast HDDs and long sequential access



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

Kafka Record Retention Redux



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



Broker Log Config

Kafka Broker Config for Logs

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

Broker Log Retention Config



Kafka Broker Config for Logs

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

Broker Log Segment File Config



Kafka Broker Config - Log Segments

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

Kafka Producer Load Balancing



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

Kafka Producer Record Batching



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

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

More producer settings for performance



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

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

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

Speeds up throughput drastically.



Kafka Compression

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

Read more at [Kafka documents on end to end compression](#)

Kafka Compression Config



Kafka Broker Compression Config

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

Pull vs. Push/Streams: Pull



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

Pull vs. Push/Streams: Push



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

MOM Consumer Message State



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

Kafka Consumer Message State



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

Message Delivery Semantics



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

Consumer: Message Delivery Semantics



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

Kafka Producer Acknowledgement



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

Kafka Replication



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

Kafka Broker Failover



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

Replicated Log Partitions

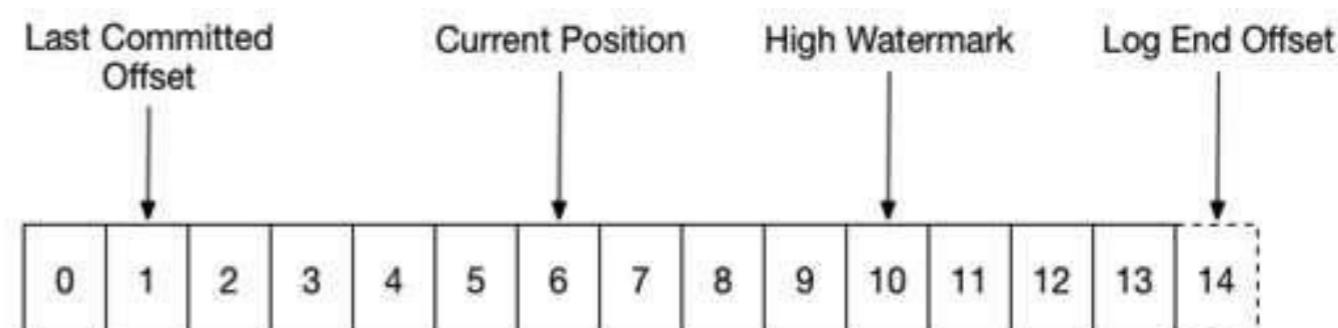


- ❖ A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- ❖ Replicated log useful for building distributed systems using state-machines
- ❖ A replicated log models “coming into consensus” on ordered series of values
 - ❖ While leader stays alive, all followers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

Kafka Consumer Replication Redux



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



Kafka Broker ReplicationConfig



Kafka Broker Config

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

Kafka Replication Broker Config 2



Kafka Broker Config

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

Kafka and Quorum



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

Kafka and Quorum 2



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

Kafka Quorum Majority of ISRs



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

Kafka Quorum Majority of ISRs 2



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

All nodes die at same time. Now what?



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

Producer Durability Acks



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



Quotas

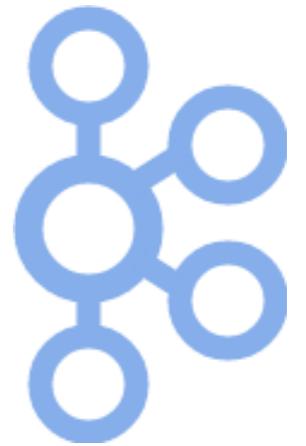
- ❖ Kafka has quotas for Consumers and Producers
- ❖ Limits bandwidth they are allowed to consume
- ❖ Prevents Consumer or Producer from hogging up all Broker resources
- ❖ Quota is by client id or user
- ❖ Data is stored in ZooKeeper; changes do not necessitate restarting Kafka

? Kafka Low-Level Review



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

Complete Lab 5.1



Working with Kafka Advanced Consumers

6. Kafka Consumers Advanced

Working with consumers in
Java

Details and advanced topics

Objectives Advanced Kafka Consumers



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

Java Consumer Examples Overview



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



KafkaConsumer

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

StockPrice App to demo Advanced Producer



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

Kafka Consumer Example



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

Consumer: createConsumer / Consumer Config



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

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

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

SimpleStockPriceConsumer.runConsumer



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

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

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



Using ConsumerRecords : SimpleStockPriceConsumer.display

SimpleStockPriceConsumer.java x

SimpleStockPriceConsumer displayRecordsStatsAndStocks()

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

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

Consumer Deserializer: StockDeserializer



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

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

Kafka Consumer: Offsets and Consumer Position

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

KafkaConsumer: Consumer Groups



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

Consumer Groups and Topic Subscriptions



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

KafkaConsumer: Partition Reassignment



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

Controlling Consumers Position



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

Storing Offsets Outside: Managing Offsets



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

Storing Offsets Outside: Managing Offsets



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

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



Controlling Consumers Position Example

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

Kafka Consumer: Consumer Alive Detection

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



Kafka Consumer: Manual Partition Assignment

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

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

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

KafkaConsumer: Consumer Alive if Polling



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

Message Delivery Semantics



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

“At-Least-Once” - Delivery Semantics



```
SimpleStockPriceConsumer pollRecordsAndProcess()
```

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

“At-Most-Once” - Delivery Semantics



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



Fine Grained “At-Most-Once”

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



Fine Grained “At-Least-Once”

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

Consumer: Exactly Once, Saving Offset



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

Saving Topic, Offset, Partition in DB



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

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

“Exactly-Once” - Delivery Semantics



SimpleStockPriceConsumer pollRecordsAndProcess()

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

Move Offsets past saved Records



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

Exactly Once - Move Offsets past saved Records



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

Kafka Consumer: Consumption Flow Control

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

Kafka Consumer: MultiThreaded Process

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

Kafka Consumer: One Consumer Per Thread

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

One Consumer Per Thread: Runnable



StockPriceConsumerRunnable

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

One Consumer Per Thread: runConsumer



c StockPriceConsumerRunnable.java x

StockPriceConsumerRunnable pollRecordsAndProcess()

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

One Consumer Per Thread: runConsumer



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

One Consumer Per Thread: Thread Pool



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

KafkaConsumer: One Consumer with Worker Threads



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

One Consumer with Worker Threads



StockPriceConsumerRunnable

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



Worker

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

processCommits

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

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

Using partitionsFor() for Priority Queue



ConsumerMain main()

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

Using assign() for Priority Queue



```
StockPriceConsumerRunnable runConsumer()
```

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

Complete Labs 6.1-6.7



Avro

7. Kafka & Avro: Confluent Schema Registry

Managing Record Schema in
Kafka

Confluent Schema Registry

- ❖ Confluent Schema Registry stores Avro Schemas for Kafka clients
- ❖ Provides REST interface for putting and getting Avro schemas
- ❖ Stores a history of schemas
 - ❖ versioned
 - ❖ allows you to configure compatibility setting
 - ❖ supports evolution of schemas
- ❖ Provides serializers used by Kafka clients which handles schema storage and serialization of records using Avro

Why Schema Registry?

- ❖ Producer creates a record/message, which is an Avro record
- ❖ Record contains the schema and data
- ❖ Schema Registry Avro Serializer serializes the data and schema id (just id)
 - ❖ Keeps a cache of registered schemas from Schema Registry to ids
- ❖ Consumer receives payload and deserializes it with Schema Registry Avro Deserializers
- ❖ Deserializer looks up the full schema from cache or Schema Registry based on id
- ❖ Consumer has its schema, one it is expecting record/message to conform to
 - ❖ Compatibility check is performed or two schemas
 - ❖ if no match, but are compatible, then payload transformation happens aka Schema Evolution
 - ❖ if not failure
- ❖ Kafka records have Key and Value and schema can be done on both

Schema Compatibility

- ❖ Backward Compatibility (default)
 - ❖ New, backward compatible schema, will not break consumers
 - ❖ Producers could be using older schema that is backwards compatible with Consumer
- ❖ Forward compatibility
 - ❖ Records sent with new forward compatible schema can be deserialized with older schemas
 - ❖ Consumers can use an older schema and never be updated (maybe never needs new fields)
- ❖ Full compatibility
 - ❖ New version of a schema is backward and forward compatible
- ❖ None
 - ❖ Schema will not be validated for compatibility at all

Schema Registry Config

- ❖ Compatibility can be configured globally or per schema
- ❖ Options are:
 - ❖ NONE - don't check for schema compatibility
 - ❖ FORWARD - check to make sure last schema version is forward compatible with new schemas
 - ❖ BACKWARDS (default) - make sure new schema is backwards compatible with latest
 - ❖ FULL - make sure new schema is forwards and backwards compatible from latest to new and from new to latest

Schema Registry Actions

- ❖ Register schemas for key and values of Kafka records
- ❖ List schemas (subjects)
- ❖ List all versions of a subject (schema)
- ❖ Retrieve a schema by version or id
 - ❖ get latest version of schema
- ❖ Check to see if schema is compatible with a certain version
- ❖ Get the compatibility level setting of the Schema Registry
 - ❖ BACKWARDS, NONE
- ❖ Add compatibility settings to a subject/schema

Schema Evolution

- ❖ Avro schema is changed after data has been written to store using an older version of that schema, then Avro might do a Schema Evolution
- ❖ Schema evolution is automatic transformation of Avro schema
 - ❖ transformation is between version of consumer schema and what the producer put into the Kafka log
 - ❖ When Consumer schema is not identical to the Producer schema used to serialize the Kafka Record then a data transformation is performed on the Kafka record (key or value)
 - ❖ If the schemas match then no need to do a transformation
 - ❖ Schema evolution is happens only during deserialization at the Consumer
 - ❖ If Consumer's schema is different from Producer's schema, then value or key is automatically modified during deserialization to conform to consumers reader schema

Allowed Schema Modifications

- ❖ Add a field with a default
- ❖ Remove a field that had a default value
- ❖ Change a fields order attribute
- ❖ Change a fields default value
- ❖ Remove or add a field alias
- ❖ Remove or add a type alias
- ❖ Change a type to a union that contains original type

Rules of the Road for modifying Schema

- ❖ Provide a default value for fields in your schema
 - ❖ Allows you to delete the field later
- ❖ Don't change a field's data type
- ❖ When adding a new field to your schema, you have to provide a default value for the field
- ❖ Don't rename an existing field
 - ❖ You can add an alias

Remember our Employee example



The screenshot shows an IDE interface with a project structure on the left and a code editor on the right. The project structure under the 'avro' directory includes '.gradle', '.idea', 'build' (containing 'classes' and 'dependency-cache'), 'generated-main-avro-java [main]' (containing 'com.cloudurable.phonebook' with an 'Employee' file), 'libs', 'tmp', 'gradle', 'src' (containing 'main' with 'avro' and 'com.cloudurable.phonebook' subfolders), and 'Employee.avsc'. The code editor displays the 'Employee.avsc' file content:

```
1 {"namespace": "com.cloudurable.phonebook",
2  "type": "record",
3  "name": "Employee",
4  "fields": [
5      {"name": "firstName", "type": "string"},
6      {"name": "nickName", "type": ["string", "null"]},
7      {"name": "lastName", "type": "string"},
8      {"name": "age", "type": "int"},
9      {"name": "phoneNumber", "type": "string"}]
```

Avro covered in [Avro/Kafka Tutorial](#)

Let's say

- ❖ Employee did not have an age in version 1 of the schema
- ❖ Later we decided to add an age field with a default value of -1
- ❖ Now let's say we have a Producer using version 2, and a Consumer using version 1

Scenario adding a new field age with default value

- ❖ Producer uses version 2 of the Employee schema and creates a com.cloudurable.Employee record, and sets age field to 42, then sends it to Kafka topic new-employees
- ❖ Consumer consumes records from new-employees using version 1 of the Employee Schema
 - ❖ Since Consumer is using version 1 of schema, age field is removed during deserialization
- ❖ Same consumer modifies name field and then writes the record back to a NoSQL store
 - ❖ When it does this, the age field is missing from value that it writes to the store
- ❖ Another client using version 2 reads the record from the NoSQL store
 - ❖ Age field is missing from the record (because the Consumer wrote it with version 1), age is set to default value of -1

Schema Registry Actions

- ❖ Register schemas for key and values of Kafka records
- ❖ List schemas (subjects)
- ❖ List all versions of a subject (schema)
- ❖ Retrieve a schema by version or id
 - ❖ get latest version of schema
- ❖ Check to see if schema is compatible with a certain version
- ❖ Get the compatibility level setting of the Schema Registry
 - ❖ BACKWARDS, FORWARD, FULL, NONE
- ❖ Add compatibility settings to a subject/schema

Register a Schema



```
private final static MediaType SCHEMA_CONTENT =
    MediaType.parse("application/vnd.schemaregistry.v1+json");

private final static String EMPLOYEE_SCHEMA = "{\n" +
    "  \"schema\": \"\" +\n    "{" +\n      "\"namespace\": \"com.cloudurable.phonebook\", " +\n      "\"type\": \"record\", " +\n      "\"name\": \"Employee\", " +\n      "\"fields\": [ " +\n        {"name\": \"firstName\", \"type\": \"string\"}, " +\n        {"name\": \"lastName\", \"type\": \"string\"}, " +\n        {"name\": \"age\", \"type\": \"int\"}, " +\n        {"name\": \"phoneNumber\", \"type\": \"string\"} " +\n      "] " +\n    }\" +\n  \"\";\n};
```



Register a Schema

```
final OkHttpClient client = new OkHttpClient();

//POST A NEW SCHEMA
Request request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee/versions")
    .build();

String output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
curl -X POST -H "Content-Type: application/vnd.schema.registry.v1+json"
--data '{"schema": "{\"type\": ...}"' \
http://localhost:8081/subjects/Employee/versions
```

{"id":2}

List All Schema



```
//LIST ALL SCHEMAS
request = new Request.Builder()
    .url("http://localhost:8081/subjects")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
curl -X GET http://localhost:8081/subjects
```

["Employee","Employee2","FooBar"]

Working with versions

```
//SHOW ALL VERSIONS OF EMPLOYEE
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/")
    .build(); [1,2,3,4,5]

output = client.newCall(request).execute().body().string();
System.out.println(output);

//SHOW VERSION 2 OF EMPLOYEE
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/2")
    .build(); {"subject":"Employee","version":2,"id":4,"schema":
    {"type":"record","name":"Employee",
    "namespace":"com.cloudurable.phonebook",...}

output = client.newCall(request).execute().body().string();
System.out.println(output);

//SHOW THE SCHEMA WITH ID 3
request = new Request.Builder()
    .url("http://localhost:8081/schemas/ids/3")
    .build(); {"subject":"Employee","version":1,"id":3,"schema":
    {"type":"record","name":"Employee",
    "namespace":"com.cloudurable.phonebook",...}

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

Working with Schemas



```
//SHOW THE LATEST VERSION OF EMPLOYEE 2
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/latest")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
//CHECK IF SCHEMA IS REGISTERED
request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee")
    .build();
```

```
output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
//TEST COMPATIBILITY
request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/compatibility/subjects/Employee/versions/latest")
    .build();
```

Changing Compatibility Checks



```
// SET TOP LEVEL CONFIG
// VALUES are none, backward, forward and full
request = new Request.Builder()
    .put(RequestBody.create(SCHEMA_CONTENT, "{\"compatibility\": \"none\"}"))
    .url("http://localhost:8081/config")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);

// SET CONFIG FOR EMPLOYEE
// VALUES are none, backward, forward and full
request = new Request.Builder()
    .put(RequestBody.create(SCHEMA_CONTENT, "{\"compatibility\": \"backward\"}"))
    .url("http://localhost:8081/config/Employee")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);
```



Incompatible

Changes

```
//POST A NEW SCHEMA
Request request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee/versions")
    .build();

String output = client.newCall(request).execute().body().string();
System.out.println(output);

private final static String EMPLOYEE_SCHEMA = "{\n" +
    "  \"schema\": \"\" +\n    "{" +\n      "  \"namespace\": \"com.cloudurable.phonebook\", " +\n      "  \"type\": \"record\", " +\n      "  \"name\": \"Employee\", " +\n      "  \"fields\": [ " +\n        "    {\"name\": \"fName\", \"type\": \"string\"}, " +\n        "    {\"name\": \"lName\", \"type\": \"string\"}, " +\n        "    {\"name\": \"age\", \"type\": \"int\"}, " +\n        "    {\"name\": \"phoneNumber\", \"type\": \"string\"} " +\n      "  ]" +\n    "}" +\n  "}";
}
```

{"error_code":409,
message:"Schema being registered is incompatible with an e

Incompatible Change



```
private final static String EMPLOYEE_SCHEMA = "{\n" +  
    "  \"schema\": \"\" +  
    "  {" +  
    "    \"namespace\": \"com.cloudurable.phonebook\", " +  
    "    \"type\": \"record\", " +  
    "    \"name\": \"Employee\", " +  
    "    \"fields\": [ " +  
    "      {" +  
    "        \"name\": \"fName\", \"type\": \"string\"}, " +  
    "        {" +  
    "          \"name\": \"lName\", \"type\": \"string\"}, " +  
    "          {" +  
    "            \"name\": \"age\", \"type\": \"int\"}, " +  
    "            {" +  
    "              \"name\": \"phoneNumber\", \"type\": \"string\"} " +  
    "        ]" +  
    "    }\" +  
    "};
```

```
//TEST COMPATIBILITY  
request = new Request.Builder()  
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))  
    .url("http://localhost:8081/compatibility/subjects/Employee/versions/latest")  
    .build();
```

{"is_compatible":false}

Use Schema Registry

- ❖ Start up Schema Registry server pointing to Zookeeper cluster
- ❖ Import Kafka Avro Serializer and Avro Jars
- ❖ Configure Producer to use Schema Registry
- ❖ Use KafkaAvroSerializer from Producer
- ❖ Configure Consumer to use Schema Registry
- ❖ Use KafkaAvroDeserializer from Consumer

Start up Schema Registry Server

```
cat ~/tools/confluent-3.2.1/etc/schema-registry/schema-registry.properties
```

```
listeners=http://0.0.0.0:8081
kafkastore.connection.url=localhost:218
1 kafkastore.topic=_schemas
debug=false
```

```
$ bin/schema-registry-start ~/tools/confluent-3.2.1/etc/schema-registry/schema-registry.properties
[2017-05-09 15:45:34,055] INFO SchemaRegistryConfig values:
  metric.reporters = []
  kafkastore.sasl.kerberos.kinit.cmd = /usr/bin/kinit
  response.mediatype.default = application/vnd.schemaregistry.v1+json
  kafkastore.ssl.trustmanager.algorithm = PKIX
  authentication.realm =
  ssl.keystore.type = JKS
  kafkastore.topic = _schemas
  metrics.jmx.prefix = kafka.schema.registry
```

Import Kafka Avro Serializer & Avro Jars



The screenshot shows a Java project named "schema-registry" in an IDE. The project structure on the left includes a "generated" folder, a "com.clo" package containing "Er", "Pr", and "St" files, and a "src" folder with "main" and "avro" subfolders. The "avro" folder contains "emp" (JSON) and "java" subfolders, which further contain "com" subfolders. The "build.gradle" file is open in the editor, displaying the following code:

```
group 'cludurable'
version '1.0-SNAPSHOT'
apply plugin: 'java'
sourceCompatibility = 1.8

dependencies {
    compile "org.apache.avro:avro:1.8.1"
    compile 'com.squareup.okhttp3:okhttp:3.7.0'
    testCompile 'junit:junit:4.11'
    compile 'org.apache.kafka:kafka-clients:0.10.2.0'
    compile 'io.confluent:kafka-avro-serializer:3.2.1'
}

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "http://packages.confluent.io/maven/"
    }
}

avro {
    createSetters = false
    fieldVisibility = "PRIVATE"
}
```



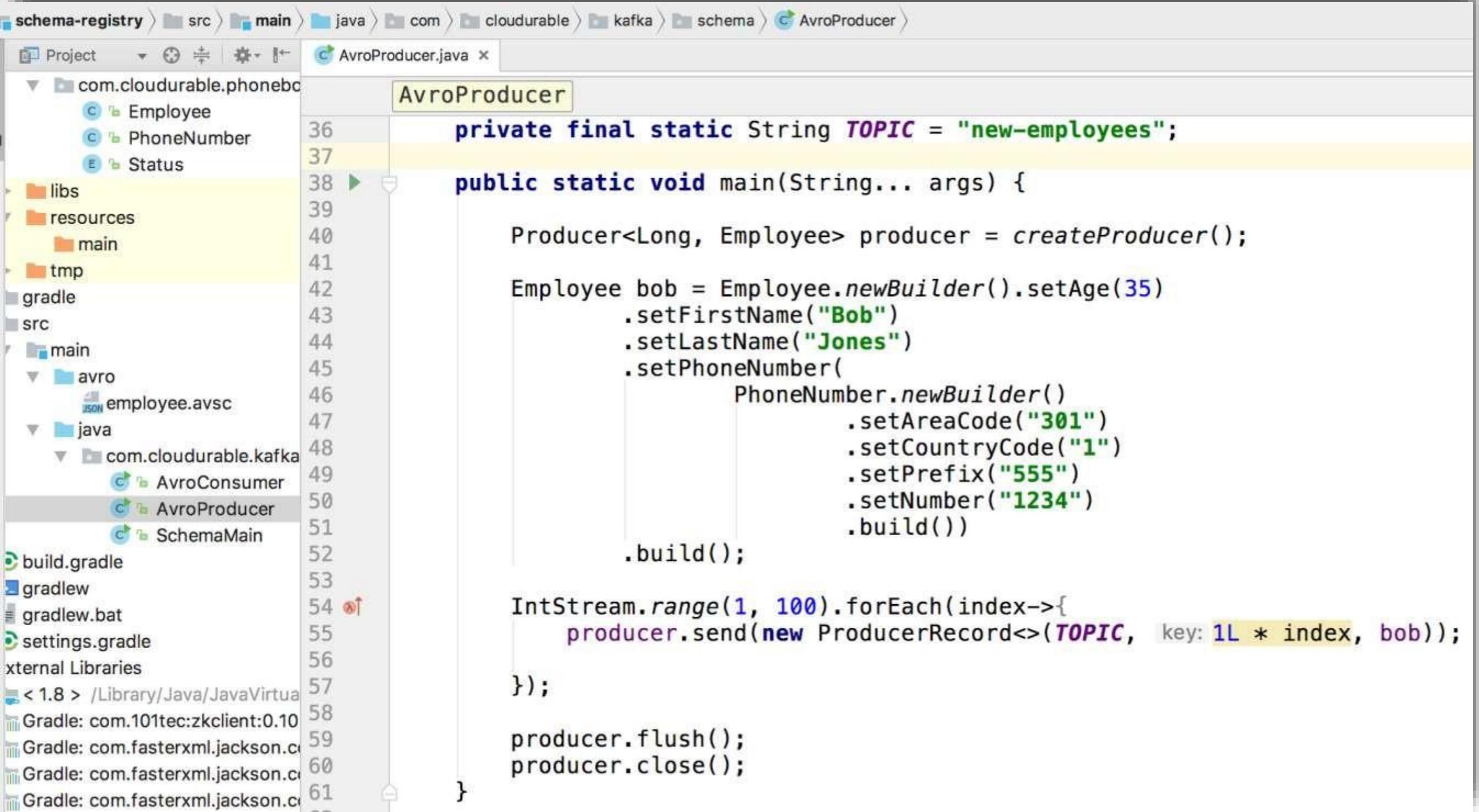
Configure Producer to use Schema Registry

schema-registry > src > main > java > com > cludurable > kafka > schema > AvroProducer

Project com.cloudurable.phonebook AvroProducer.java AbstractKafkaAvroSerDeConfig.class

```
AvroProducer
8 import org.apache.kafka.clients.producer.ProducerConfig;
9 import org.apache.kafka.clients.producer.ProducerRecord;
10 import org.apache.kafka.common.serialization.LongSerializer;
11 import io.confluent.kafka.serializers.KafkaAvroSerializer;
12
13 import java.util.Properties;
14 import java.util.stream.IntStream;
15
16 public class AvroProducer {
17
18     private static Producer<Long, Employee> createProducer() {
19         Properties props = new Properties();
20         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
21         props.put(ProducerConfig.CLIENT_ID_CONFIG, "AvroProducer");
22         props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
23                  LongSerializer.class.getName());
24
25         // Configure the KafkaAvroSerializer.
26         props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
27                  KafkaAvroSerializer.class.getName());
28
29
30
31
32
33
34 }
```

Use KafkaAvroSerializer from Producer



The screenshot shows a Java code editor with the file `AvroProducer.java` open. The code is part of a project named `schema-registry`. The code itself is as follows:

```
private final static String TOPIC = "new-employees";

public static void main(String... args) {
    Producer<Long, Employee> producer = createProducer();

    Employee bob = Employee.newBuilder().setAge(35)
        .setFirstName("Bob")
        .setLastName("Jones")
        .setPhoneNumber(
            PhoneNumber.newBuilder()
                .setAreaCode("301")
                .setCountryCode("1")
                .setPrefix("555")
                .setNumber("1234")
                .build())
        .build();

    IntStream.range(1, 100).forEach(index->{
        producer.send(new ProducerRecord<>(TOPIC, key: 1L * index, bob));
    });

    producer.flush();
    producer.close();
}
```

The code defines a `TOPIC` constant and a `main` method. Inside the `main` method, it creates a `producer` using `createProducer()`. It then creates an `Employee` object named `bob` with various fields set. It uses `PhoneNumber.newBuilder()` to build a `PhoneNumber` object with area code 301, country code 1, prefix 555, and number 1234. Finally, it sends 100 records to the `TOPIC` with keys ranging from 1 to 100 and the value being the `bob` employee object.

Configure Consumer to use Schema Registry



```
AvroConsumer.java x
AvroConsumer
16 > public class AvroConsumer {
17
18     private final static String BOOTSTRAP_SERVERS = "localhost:9092";
19     private final static String TOPIC = "new-employees";
20
21     private static Consumer<Long, Employee> createConsumer() {
22         Properties props = new Properties();
23         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
24         props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleAvroConsumer");
25         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
26                   LongDeserializer.class.getName());
27
28
29
30
31
32     //Use Specific Record or else you get Avro GenericRecord.
33     props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
34
35
36     //Schema registry location.
37     props.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG,
38               "http://localhost:8081"); //----- Run Schema Registry on 8081
39
40
41     return new KafkaConsumer<>(props);
42 }
43
```

Use KafkaAvroDeserializer from Consumer



```
public static void main(String... args) {  
  
    final Consumer<Long, Employee> consumer = createConsumer();  
    consumer.subscribe(Collections.singletonList(TOPIC));  
  
    IntStream.range(1, 100).forEach(index -> {  
  
        final ConsumerRecords<Long, Employee> records =  
            consumer.poll( timeout: 100);  
  
        if (records.count() == 0) {  
            System.out.println("None found");  
        } else records.forEach(record -> {  
  
            Employee employeeRecord = record.value();  
  
            System.out.printf("%s %d %d %s \n", record.topic(),  
                record.partition(), record.offset(), employeeRecord);  
        });  
    });  
}
```

Schema Registry

- ❖ Confluent provides Schema Registry to manage Avro Schemas for Kafka Consumers and Producers
- ❖ Avro provides Schema Migration
- ❖ Confluent uses Schema compatibility checks to see if Producer schema and Consumer schemas are compatible and to do Schema evolution if needed
- ❖ Use KafkaAvroSerializer from Producer
- ❖ Use KafkaAvroDeserializer from Consumer



Kafka Log Compaction Architecture

Kafka Log Compaction

Design discussion of Kafka Log
Compaction

[Kafka Architecture: Log
Compaction](#)

Kafka Log Compaction Overview



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

Log Compaction



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

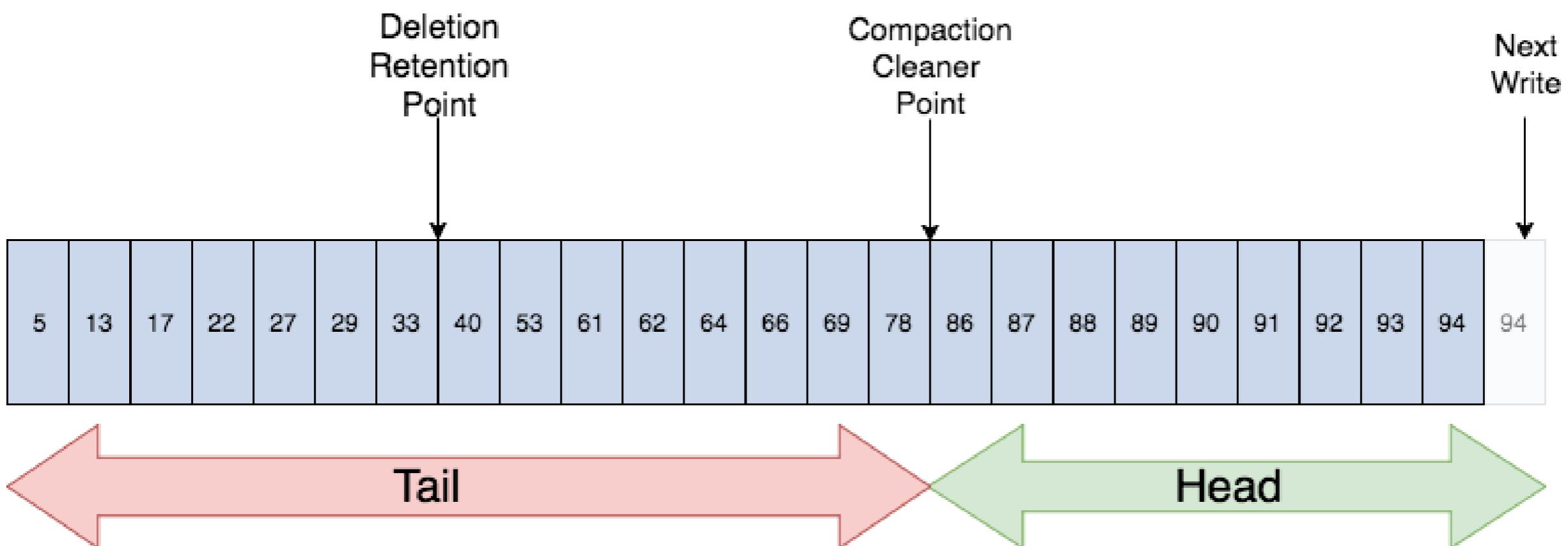
Log Compaction Structure



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



Compaction Tail/Head



Log Compaction Basics



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

Log Compaction Cleaning



Before Compaction

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

Cleaning

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

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

After Compaction

Log Compaction Guarantees



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



Log Cleaner

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

Topic Config for Log Compaction



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

Broker Config for Log Compaction

Kafka Broker Log Compaction Config

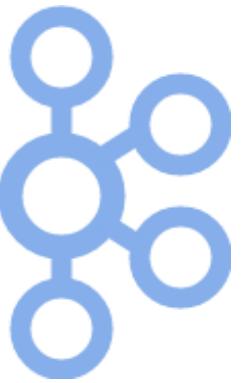


NAME	DESCRIPTION	TYPE	DEFAULT
log.cleaner.backoff.ms	Sleep period when no logs need cleaning	long	15,000
log.cleaner.dedupe.buffer.size	The total memory for log dedupe process for all cleaner threads	long	134,217,728
log.cleaner.delete.retention.ms	How long record delete markers (tombstones) are retained.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

? Log Compaction Review

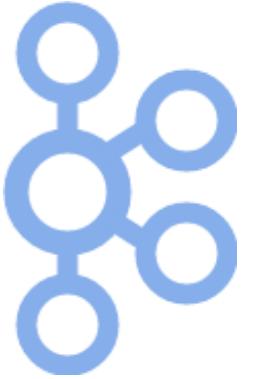


- ❖ What are three ways Kafka can delete records?
- ❖ What is log compaction good for?
- ❖ What is the structure of a compacted log? Describe the structure.
- ❖ After compaction, do log record offsets change?
- ❖ What is a partition segment?



9. Kafka MirrorMaker

MirrorMaker for Disaster Recovery and Scaling Reads



Kafka MirrorMaker

Kafka Disaster Recovery

MirrorMaker

MirrorMaker
Disaster Recovery
Scaling Reads
Isolate Mission Critical Kafka
Clusters



MirrorMaker and Mirroring

- ❖ Mirroring is replication between clusters - called ***mirroring*** to not confuse with ***replication***
 - ❖ replication uses cluster involving brokers, partition leaders, partition followers, ISRs and ZooKeeper
 - ❖ mirroring is just a consumer/producer pair in two clusters
- ❖ ***MirrorMaker*** is used for replicating one clusters data to another cluster



Mirror Maker

- ❖ ***MirrorMaker*** acts like a ***consumer*** to a ***source cluster***
- ❖ ***MirrorMaker*** acts like a ***producer*** to a ***destination cluster***
- ❖ ***Data read from source topics in source cluster and written to same named topics in destination cluster***
- ❖ Source and destination clusters are independent and not coupled
 - ❖ Topics can be configured differently, have different offsets
 - ❖ e.g., different partition count and different replication factors

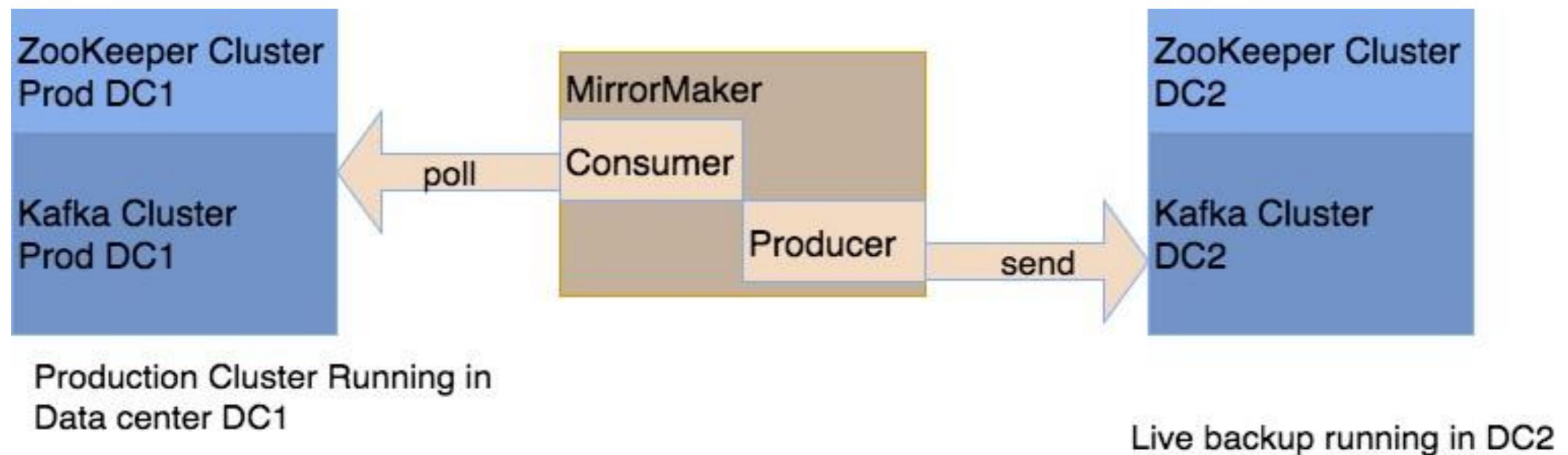


MirrorMaker Use Cases

- ❖ Provide a replica to another datacenter or AWS region
- ❖ ***Mirroring*** used for ***disaster recovery***
 - ❖ datacenter or region goes down
 - ❖ cluster is used for normal fault-tolerance
- ❖ ***Mirroring*** can also be used for ***increased throughput***
 - ❖ scale consumers
 - ❖ scales reads



Disaster Recovery

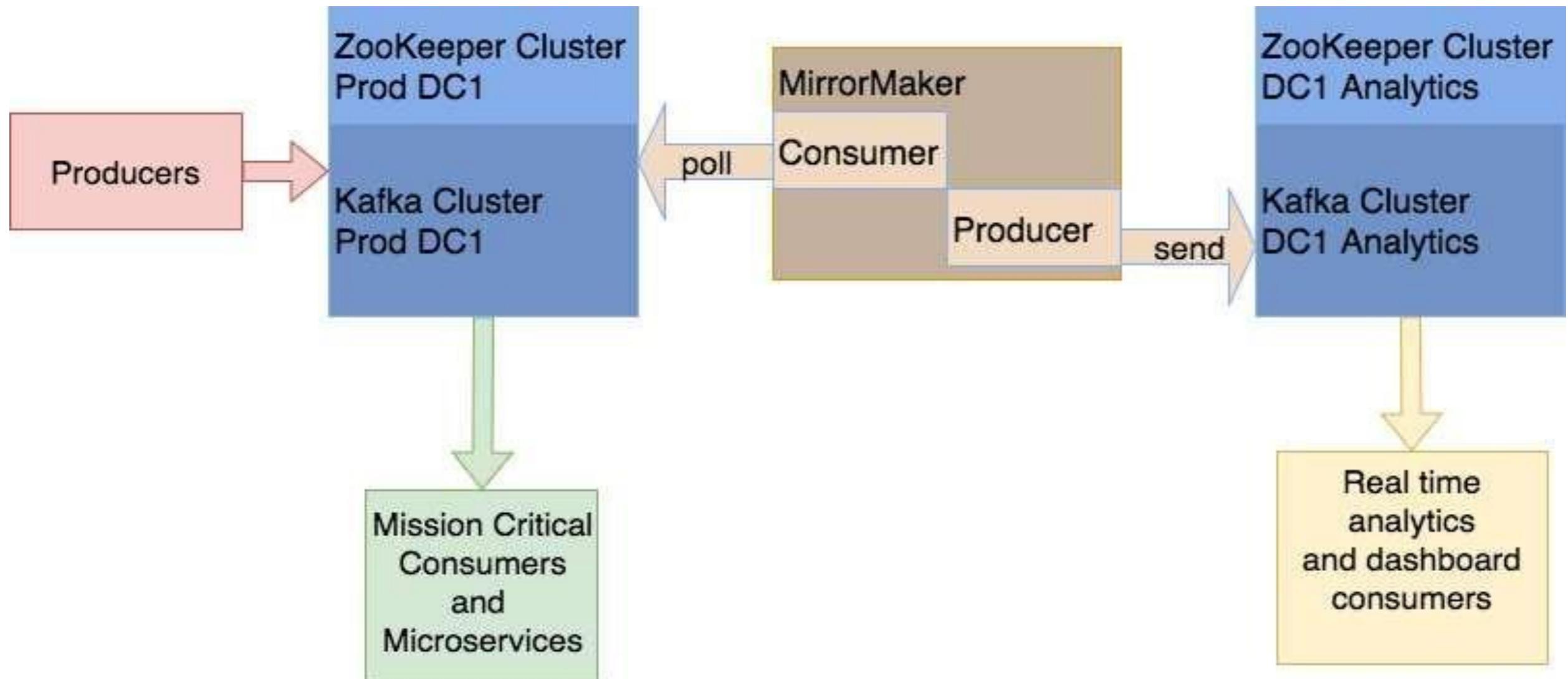


Scale Reads / Scale Consumers

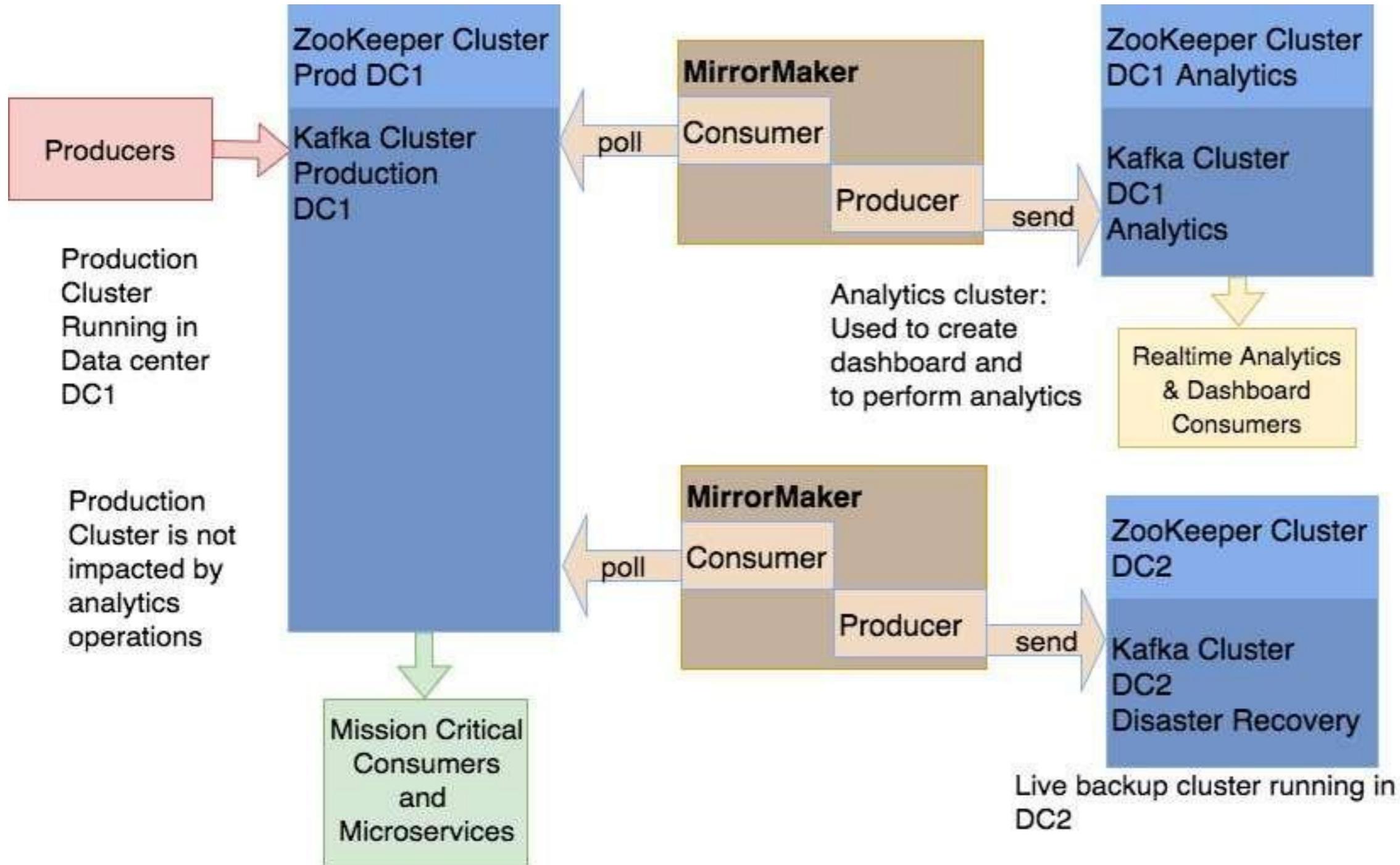


- ❖ You can use MirrorMaker to scale reads
- ❖ You could move non-mission critical consumers to another cluster and replicate to this other cluster
- ❖ Other cluster can replay log or do read intensive log operations and analytics **w/o impacting Production**
- ❖ Production cluster **to serve mission critical services**
- ❖ Analytics cluster could be doing real time dash boards and analytics

Scale Write, Avoid Impacting Mission Critical Services



Many MirrorMakers for different Purposes





Mirror Maker Command Line

- ❖ ***kafka-mirror-maker.sh***
- ❖ **--whitelist** specifies regex for topics to mirror
 - ❖ ‘stock-prices|stocks’ selects two topics
 - ❖ ‘*’ selects all topics
- ❖ **--blacklist** —whitelist regex for topics to exclude
- ❖ Using mirroring with broker config
auto.create.topics.enable=true on destination cluster makes auto replication with no config possible (—whitelist “*”)



Mirror Maker Command Line

start-mirror-maker-1st-to-2nd.sh ×

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka Mirror Maker
6 kafka/bin/kafka-mirror-maker.sh \
7   --consumer.config "$CONFIG/mm-consumer-1st.properties" \
8   --producer.config "$CONFIG/mm-producer-2nd.properties" \
9   --whitelist "./*"
10
```

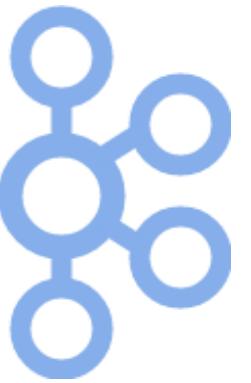
- ❖ Pass consumer properties to read from 1st cluster
- ❖ Pass producer properties to write to 2nd cluster
- ❖ Specify that you want to replicate all topics via whitelist regex



MirrorMaker Review



- ❖ What is the difference between failover and disaster recovery?
- ❖ What are two use cases where you would use MirrorMaker?
- ❖ Why might you want to separate a production microservice messages from a more ad hoc analytics system?
- ❖ If you had to run a nightly job that tallied analytics to all of the calls to a 24/7 production microservice for the last month would you run that in the production cluster?



Lab Running Mirror Maker



Lab Objectives

- ❖ Show running MirrorMaker to mirror a Kafka Cluster to another Kafka Cluster
- ❖ Show how topics can be configured differently per Cluster
- ❖ Demonstrate how to run MirrorMaker
- ❖ Demonstrate how to configure MirrorMakers Consumer
- ❖ Demonstrate how to configure MirrorMakers Producer



Example Details Java

- ❖ Uses a version of StockPriceProducer example
- ❖ Modifies Consumer to consume from one of three Clusters
- ❖ Three clusters in example
- ❖ Producer sends to 1st Cluster
- ❖ Consumer to consume from 1st Cluster
- ❖ Consumer to consume from 2nd Cluster
- ❖ Consumer to consume from 3rd Cluster



Example Details Scripts

- ❖ Script to startup first, second and 3rd cluster
 - ❖ start up Broker and ZooKeeper
- ❖ Script to mirror records between 1st and 2nd cluster
- ❖ Script to mirror records between 2nd and 3rd cluster
- ❖ Scripts to describe topic for each cluster



Scripts

Scripts

Script Name	Description
start-1st-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the first cluster.
start-2nd-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the second cluster.
start-3rd-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the third cluster.
start-mirror-maker-1st-to-2nd.sh	Starts up MirrorMaker to mirror records from the 1st cluster to the second cluster.
start-mirror-maker-2nd-to-3rd.sh	Starts up MirrorMaker to mirror records from the 2nd cluster to the 3rd cluster.
create-topic.sh	Creates stock-prices topic.
describe-topic-1st.sh	Describes stock-prices on first cluster.
describe-topic-2nd.sh	Describes stock-prices on second cluster.
describe-topic-3rd.sh	Describes stock-prices on third cluster.

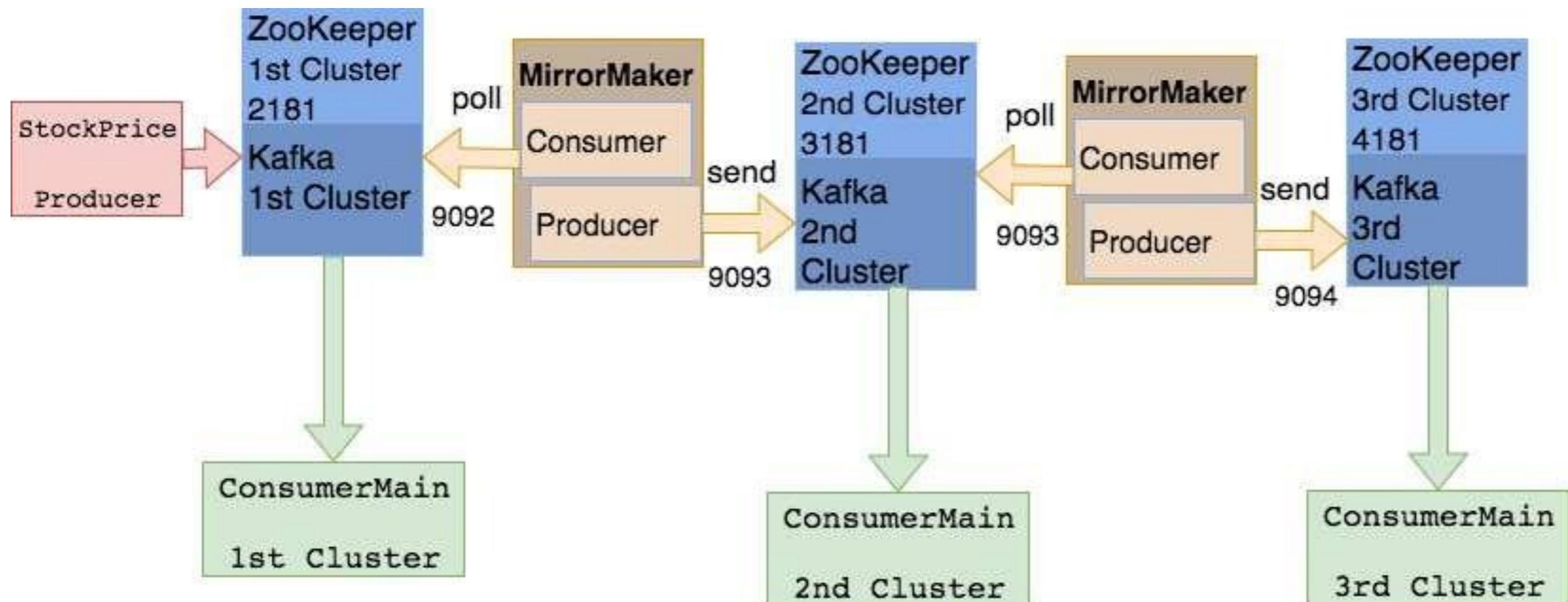


Cluster Ports

Cluster Config

Cluster	Kafka Client Port	ZooKeeper Port
1st Cluster	9092	2181
2nd Cluster	9093	3181
3rd Cluster	9094	4181

Example MirrorMaker Cluster Producer Consumer





Start Cluster Scripts

start-1st-cluster.sh ×

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

start-2nd-cluster.sh ×

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

start-3rd-cluster.sh ×

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run ZooKeeper for 3rd Cluster
6 kafka/bin/zookeeper-server-start.sh \
7 "$CONFIG/zookeeper-2.properties" &
8
9
10 ## Run Kafka for 3rd Cluster
11 kafka/bin/kafka-server-start.sh \
12 "$CONFIG/server-2.properties"
```

Server and MirrorMaker Config

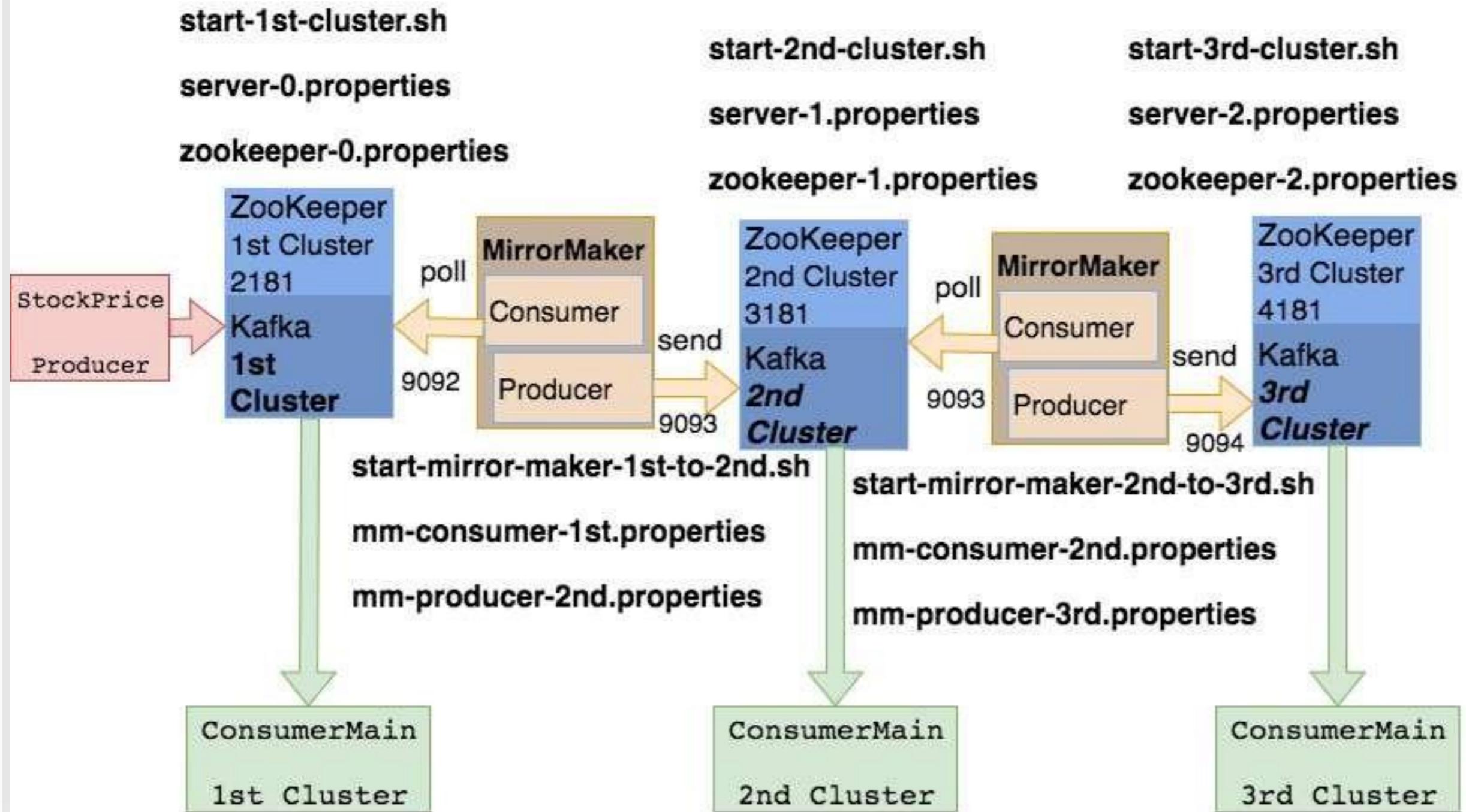


Config

Config File	Description	Config Type
server-0.properties	Kafka Server 0 for 1st Cluster	Kafka Broker
server-0.properties	Kafka Server 1 for 2nd Cluster	Kafka Broker
server-0.properties	Kafka Server 2 for 3rd Cluster	Kafka Broker
zookeeper-0.properties	ZooKeeper Server Config for 1st Cluster	ZooKeeper
zookeeper-1.properties	ZooKeeper Server Config for 2nd Cluster	ZooKeeper
zookeeper-2.properties	ZooKeeper Server Config for 3rd Cluster	ZooKeeper
mm-consumer-1st.properties	MirrorMaker Consumer Config for 1st Cluster	Kafka Consumer
mm-consumer-2nd.properties	MirrorMaker Consumer Config for 2nd Cluster	Kafka Consumer
mm-producer-2nd.properties	MirrorMaker Producer Config for 2nd Cluster	Kafka Producer
mm-producer-3rd.properties	MirrorMaker Producer Config for 3rd Cluster	Kafka Producer



Diagram to Script/Config





Create Topic

```
>_ create-topic.sh <

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

- ❖ Create a topic but only in the ***first*** cluster (**2181**)
- ❖ Only one broker so only one replication factor too



1st Cluster Config

```
server-0.properties x  
1 broker.id=0  
2 port=9092  
3 log.dirs=./logs/kafka-0  
4 ## Require three replicas to respond  
5 ## before acknowledging send from producer.  
6 min.insync.replicas=1  
7 auto.create.topics.enable=false  
8 zookeeper.connect=localhost:2181  
9  
10
```

- ❖ Broker ID 0
- ❖ Kafka Broker Port **9092**
- ❖ Connects to ZooKeeper at port **2181**
- ❖ **No Topic Auto Create**



2nd Cluster Config

```
server-1.properties x  
1 broker.id=1  
2 port=9093  
3 log.dirs=./logs/kafka-1  
4 min.insync.replicas=1  
5 auto.create.topics.enable=true  
6 zookeeper.connect=localhost:3181  
7 num.partitions=13
```

- ❖ Broker ID 1
- ❖ Kafka Broker Port **9093**
- ❖ Connects to ZooKeeper at port **3181**
- ❖ **Topic Auto Create Enabled!**
- ❖ Default Num Partitions **13**



3rd Cluster Config

```
server-2.properties x  
1 broker.id=2  
2 port=9094  
3 log.dirs=./logs/kafka-2  
4 min.insync.replicas=1  
5 auto.create.topics.enable=true  
6 zookeeper.connect=localhost:4181  
7 num.partitions=33
```

- ❖ Broker ID 2
- ❖ Kafka Broker Port **9094**
- ❖ Connects to ZooKeeper at port **4181**
- ❖ **Topic Auto Create Enabled!**
- ❖ Default Num Partitions **33**



ZooKeeper Config

```
zookeeper-0.properties x
1 dataDir=/tmp/zookeeper1
2 clientPort=2181
3 maxClientCnxns=0
4

zookeeper-1.properties x
1 dataDir=/tmp/zookeeper2
2 clientPort=3181
3 maxClientCnxns=0
4

zookeeper-2.properties x
1 dataDir=/tmp/zookeeper3
2 clientPort=4181
3 maxClientCnxns=0
4
```

- ❖ Each ZooKeeper instance runs on its own port and is independent of the others

Mirror Maker Consumer Config



```
mm-consumer-2nd.properties x
1 bootstrap.servers=localhost:9093
2 client.id=mm2.Consumer
3 key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
4 value.deserializer=org.apache.kafka.common.serialization.ByteArrayDeserializer
5 group.id=mm2.Consumer
6 partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor

mm-consumer-1st.properties x
1 bootstrap.servers=localhost:9092
2 client.id=mm1.Consumer
3 key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
4 value.deserializer=org.apache.kafka.common.serialization.ByteArrayDeserializer
5 group.id=mm1.Consumer
6 partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor
```

- ❖ Two Consumer for 2 different MirrorMakers
- ❖ One consumes 2nd Cluster (9093)
- ❖ One consumes 1st Cluster (9092)
- ❖ Notice we use ByteArrayDeserializer because we want MirrorMaker treating payload as opaque

Mirror Maker Producer Config



```
mm-consumer-2nd.properties x mm-producer-3rd.properties x
1 bootstrap.servers=localhost:9094
2 client.id=mml1.Producer
3 key.serializer=org.apache.kafka.common.serialization.StringSerializer
4 value.serializer=org.apache.kafka.common.serialization.BytesSerializer
5 compression.type=lz4
6 linger.ms=100
7 batch.size=65536
8

mm-consumer-1st.properties x mm-producer-2nd.properties x
1 bootstrap.servers=localhost:9093
2 client.id=mml1.Producer
3 key.serializer=org.apache.kafka.common.serialization.StringSerializer
4 value.serializer=org.apache.kafka.common.serialization.BytesSerializer
5 compression.type=lz4
6 linger.ms=100
7 batch.size=65536
```

- ❖ Two Consumer for 2 different MirrorMakers
- ❖ One produces to 3rd Cluster
- ❖ One produces to 2nd Cluster
- ❖ Notice we use BytesSerializer because we want MirrorMaker treating payload as opaque



Mirror Maker Start Scripts

```
mm-consumer-2nd.properties x mm-producer-3rd.properties x start-mirror-maker-2nd-to-3rd.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka Mirror Maker: Mirror 2nd Cluster to 3rd Cluster
5 kafka/bin/kafka-mirror-maker.sh \
6   --consumer.config "$CONFIG/mm-consumer-2nd.properties" \
7   --producer.config "$CONFIG/mm-producer-3rd.properties" \
8   --whitelist ".*"

mm-consumer-1st.properties x mm-producer-2nd.properties x start-mirror-maker-1st-to-2nd.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka Mirror Maker: Mirror 1st Cluster to 2nd Cluster
5 kafka/bin/kafka-mirror-maker.sh \
6   --consumer.config "$CONFIG/mm-consumer-1st.properties" \
7   --producer.config "$CONFIG/mm-producer-2nd.properties" \
8   --whitelist ".*"
```

- ❖ Mirror 2nd Cluster to 3rd using Producer and Consumer config
- ❖ Mirror 1st Cluster to 2nd using Producer and Consumer config



Run Scripts

- ❖ Start up ***first*** cluster: ***bin/start-1st-cluster.sh***
- ❖ In a new terminal, Start up ***2nd*** cluster: ***bin/start-2nd-cluster.sh***
- ❖ In a new terminal, Start up ***3rd*** cluster: ***bin/start-3rd-cluster.sh***
- ❖ Create Topic: ***bin/create-topic.sh***
- ❖ Wait 30 seconds
- ❖ Startup Mirror Maker for 1st to 2nd mirroring:
 - ❖ ***start-mirror-maker-1st-to-2nd.sh***
- ❖ Startup Mirror Maker for 2nd to 3rd mirroring
 - ❖ ***start-mirror-maker-2nd-to-3rd.sh***

```
Terminal
+ Cluster1 Cluster2 Cluster3 mirrorMaker1to2 mirrorMaker2to3
x ~/kafka-training/lab9/solution
$ bin/start-mirror-maker-2nd-to-3rd.sh
```

Run Java Consumer(s) and Producer



- ❖ Run ***ConsumerMain1stCluster*** in IDE
- ❖ Run ***ConsumerMain2ndCluster*** in IDE
- ❖ Run ***ConsumerMain3rdCluster*** in IDE
- ❖ Run ***StockPriceProducer*** in IDE
- ❖ After 30 Seconds stop StockPriceProducer
- ❖ Wait 30 seconds and ensure consumers have same stock prices

Output, Stocks should have same value



```
Run: ConsumerMain1stCluster
New ConsumerRecords par
ticker BBB price 80.14
ticker FFF price 80.14

New ConsumerRecords par
ticker AAA price 80.14
ticker EEE price 80.14
ticker IBM price 88.34

Run: ConsumerMain1stCluster ConsumerMain2ndCluster ConsumerMain3rdCluster
New ConsumerRecords par count 0 count 0, max offset 0
ticker IBM price 88.34 saved true Thread 9

New ConsumerRecords par count 0 count 0, max offset 0
ticker AAA price 80.14 saved true Thread 7
ticker GOOG price 482.11 saved true Thread 5

Run: ConsumerMain1stCluster ConsumerMain2ndCluster ConsumerMain3rdCluster
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
ticker AAA price 80.14 saved true Thread 13
```



ConsumerMain1stCluster

```
c ConsumerMain1stCluster.java x
1 package com.cloudurable.kafka.consumer;
2
3 import static com.cloudurable.kafka.consumer.ConsumerUtil.FIRST_CLUSTER;
4 import static com.cloudurable.kafka.consumer.ConsumerUtil.startConsumers;
5
6 public class ConsumerMain1stCluster {
7     public static void main(String... args) throws Exception {
8         startConsumers(FIRST_CLUSTER);
9     }
10 }
```

```
c ConsumerUtil.java x
ConsumerUtil
17 import java.util.stream.IntStream;
18
19 import static com.cloudurable.kafka.StockAppConstants.TOPIC;
20 import static java.util.concurrent.Executors.newFixedThreadPool;
21
22 public class ConsumerUtil {
23
24     private static final Logger logger =
25             LoggerFactory.getLogger(ConsumerUtil.class);
26
27     public static final String FIRST_CLUSTER = "localhost:9092";
28     public static final String SECOND_CLUSTER = "localhost:9093";
29     public static final String THIRD_CLUSTER = "localhost:9094";
```

ConsumerUtils startConsumers



```
C ConsumerUtil.java x
ConsumerUtil startConsumers() -> Runnable

62
63     public static void startConsumers(final String cluster) {
64         final Consumer<String, StockPrice> consumer = createConsumer(cluster);
65         final int threadCount = getPartitionCount(consumer);
66         final ExecutorService executorService = newFixedThreadPool(threadCount);
67
68         final List<StockPriceConsumerRunnable> workers = new ArrayList<>(threadCount);
69
70         IntStream.range(0, threadCount).forEach(index -> {
71             final StockPriceConsumerRunnable stockPriceConsumer =
72                 new StockPriceConsumerRunnable(createConsumer(cluster),
73                     readCountStatusUpdate: 5, index);
74             workers.add(stockPriceConsumer);
75             executorService.submit(stockPriceConsumer);
76         });
    }
```

- ❖ Start Consumers takes cluster which is broker list
- ❖ Same code as earlier examples runs Consumer per Thread
- ❖ thread count determined by partition count for topic



Describe Topic per Cluster

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

> describe-topic-2nd.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic stock-prices \
8   --zookeeper localhost:3181
9

> describe-topic-3rd.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic stock-prices \
8   --zookeeper localhost:4181
```

- ❖ Script per cluster to describe **stock-prices**



stock-prices Topic on 1st Cluster

Terminal

	Cluster1	Cluster2	Cluster3	mirrorMaker1to2	mirrorMaker2to3	Util	describe1	describe2	describe3
+									
✖									

```
~/kafka-training/lab9/solution
$ bin/describe-topic-1st.sh
Topic:stock-prices      PartitionCount:3      ReplicationFactor:1      Configs:min.insync.replicas=1
    Topic: stock-prices      Partition: 0      Leader: 0      Replicas: 0      Isr: 0
    Topic: stock-prices      Partition: 1      Leader: 0      Replicas: 0      Isr: 0
    Topic: stock-prices      Partition: 2      Leader: 0      Replicas: 0      Isr: 0
~/kafka-training/lab9/solution
```

- ❖ ***bin/describe-topic-1st.sh***
- ❖ Why does this only have three partitions?
- ❖ Where did we configure three partitions for **stock-prices**?



stock-prices Topic on 2nd Cluster

Terminal

+	Cluster1	Cluster2	Cluster3	mirrorMaker1to2	mirrorMaker2to3	Util	describe1	describe2	describe3
✖	~/kafka-training/lab9/solution								
	\$ bin/describe-topic-2nd.sh								
	Topic:stock-prices	PartitionCount:13		ReplicationFactor:1	Configs:				
	Topic: stock-prices	Partition: 0	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 1	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 2	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 3	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 4	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 5	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 6	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 7	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 8	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 9	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 10	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 11	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 12	Leader: 1	Replicas: 1	Isr: 1				

- ❖ ***bin/describe-topic-2nd.sh***
- ❖ Why does this only have 13 partitions?
- ❖ Where did we configure 13 partitions for ***stock-prices***?

stock-prices Topic on 3rd Cluster



Terminal

+ Cluster1 Cluster2 Cluster3 mirrorMaker1to2 mirrorMaker2to3 Util describe1 describe2 describe3

~/kafka-training/lab9/solution

```
$ bin/describe-topic-3rd.sh
```

Topic: stock-prices	PartitionCount: 33	ReplicationFactor: 1	Configs:
Topic: stock-prices	Partition: 0	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 1	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 2	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 3	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 4	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 5	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 6	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 7	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 8	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 9	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 10	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 11	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 12	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 13	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 14	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 15	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 16	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 17	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 18	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 19	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 20	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 21	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 22	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 23	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 24	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 25	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 26	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 27	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 28	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 29	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 30	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 31	Leader: 2	Replicas: 2 Isr: 2
Topic: stock-prices	Partition: 32	Leader: 2	Replicas: 2 Isr: 2

- ❖ ***bin/describe-topic-3rd.sh***
- ❖ Why does this only have 33 partitions?
- ❖ Where did we configure 33 partitions for ***stock-prices***?

Lab Conclusion



- ❖ Used MirrorMaker to mirror a Kafka Cluster to another Kafka Cluster
 - ❖ Mirrored 1st Cluster records to 2nd Cluster
 - ❖ Mirrored 2nd Cluster records to 3rd Cluster
- ❖ Ran 1 producer that produced to 1st Cluster
- ❖ Ran 1 consumer group that read from 1st Cluster
- ❖ Ran 1 consumer group that read from 2nd Cluster
- ❖ Ran 1 consumer group that read from 3rd Cluster
- ❖ Configured topics differently per cluster (3, 13 and 33 partitions)
- ❖ Used describe topics to show topics were configured differently on different clusters