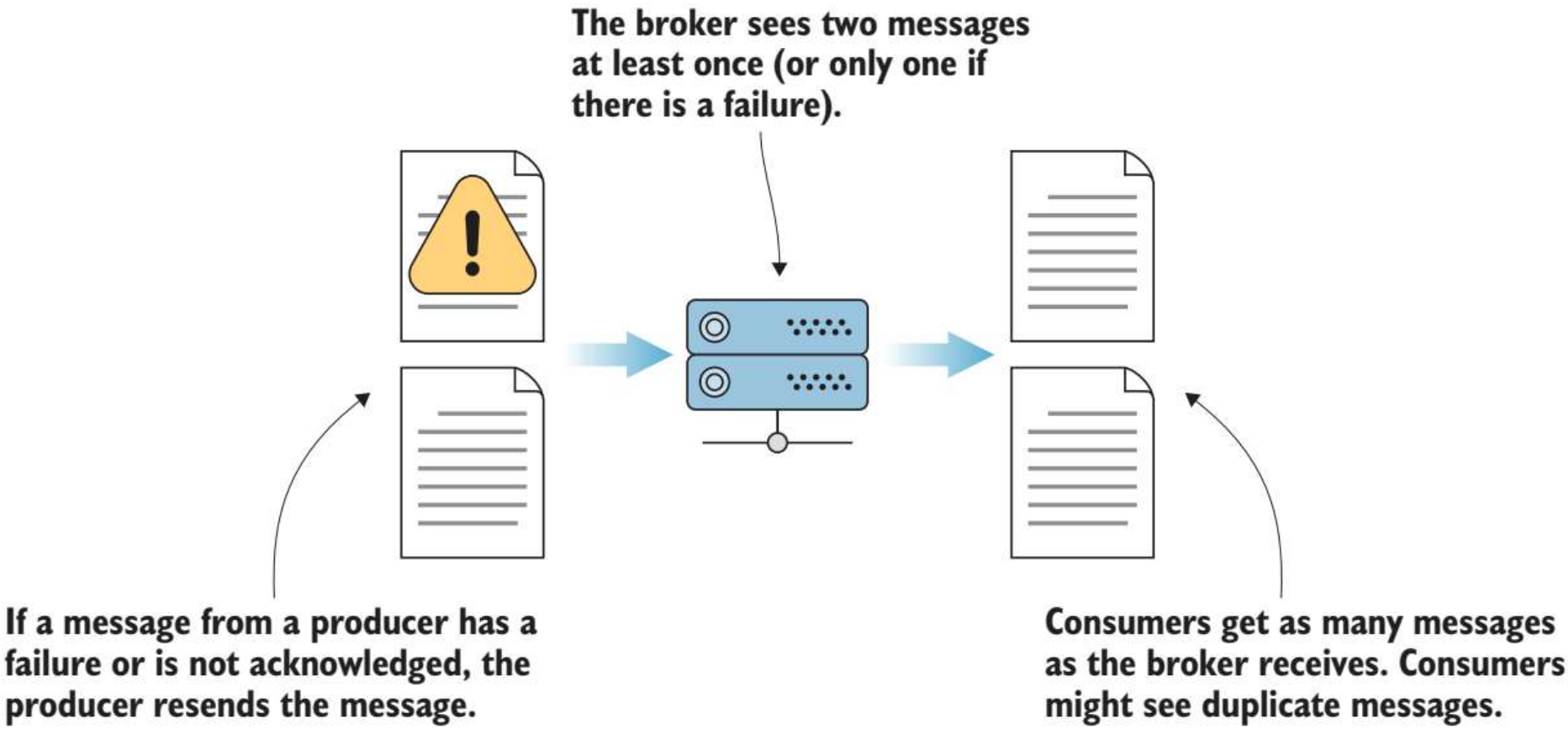
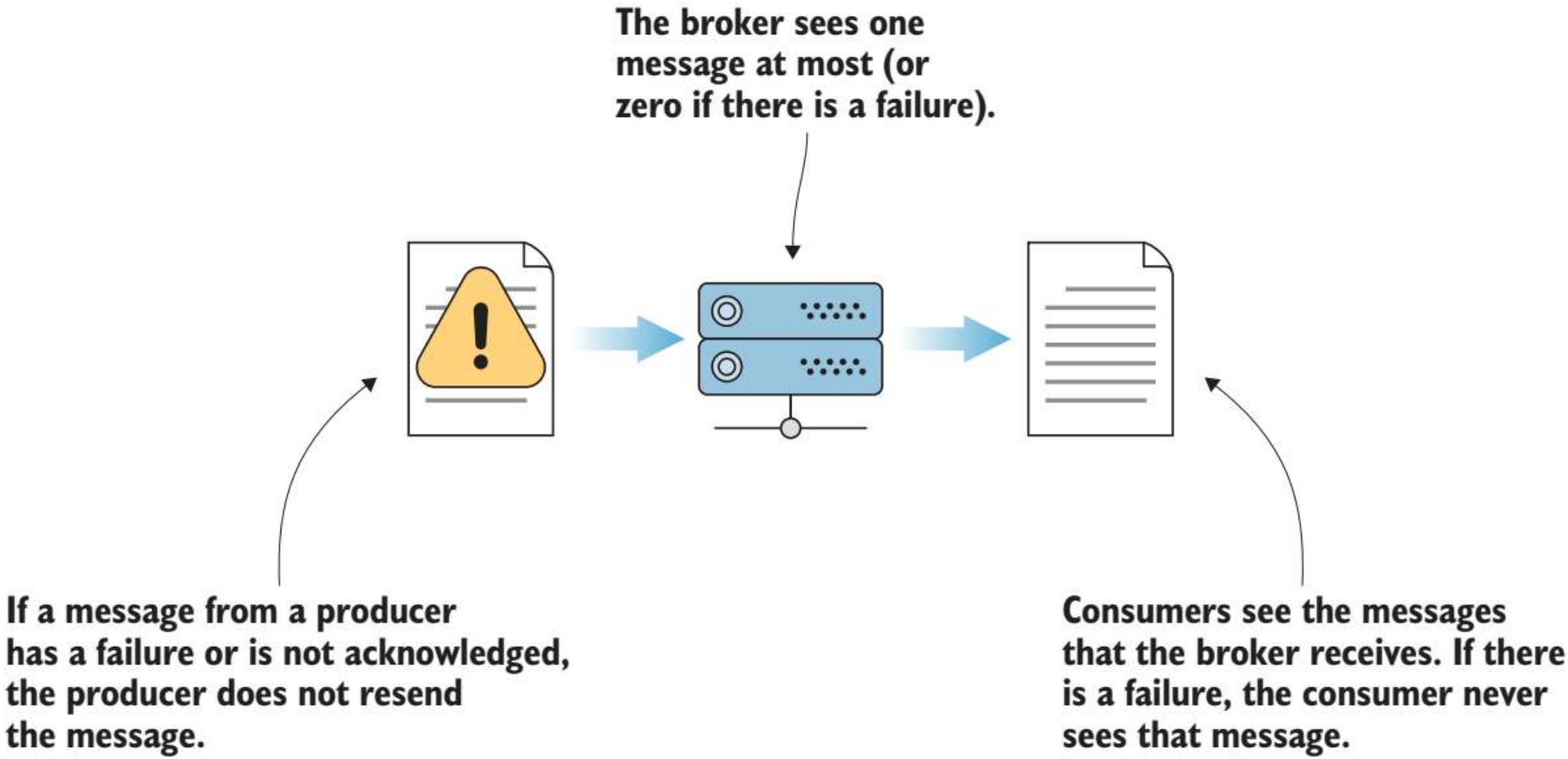
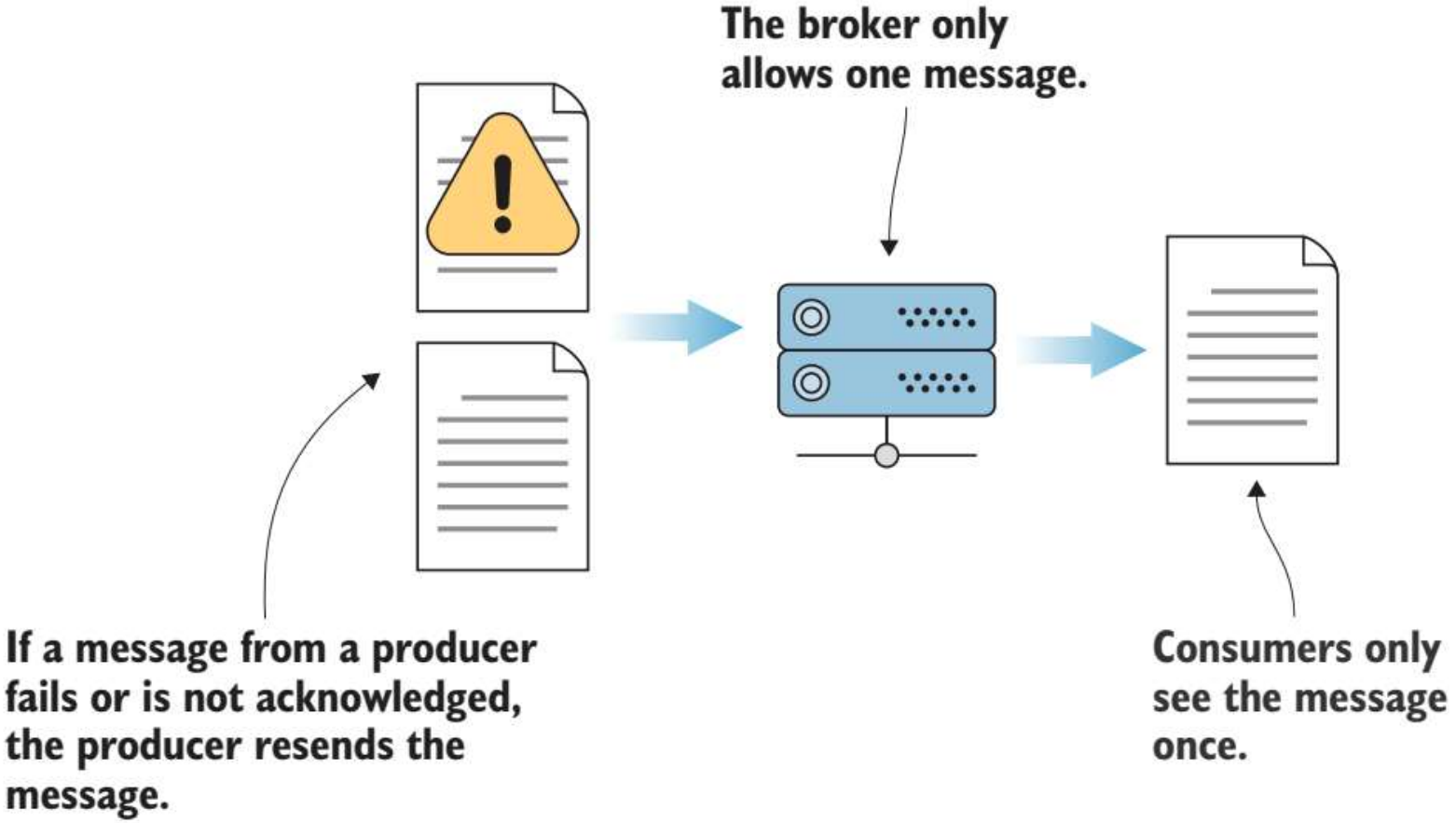
Kafka



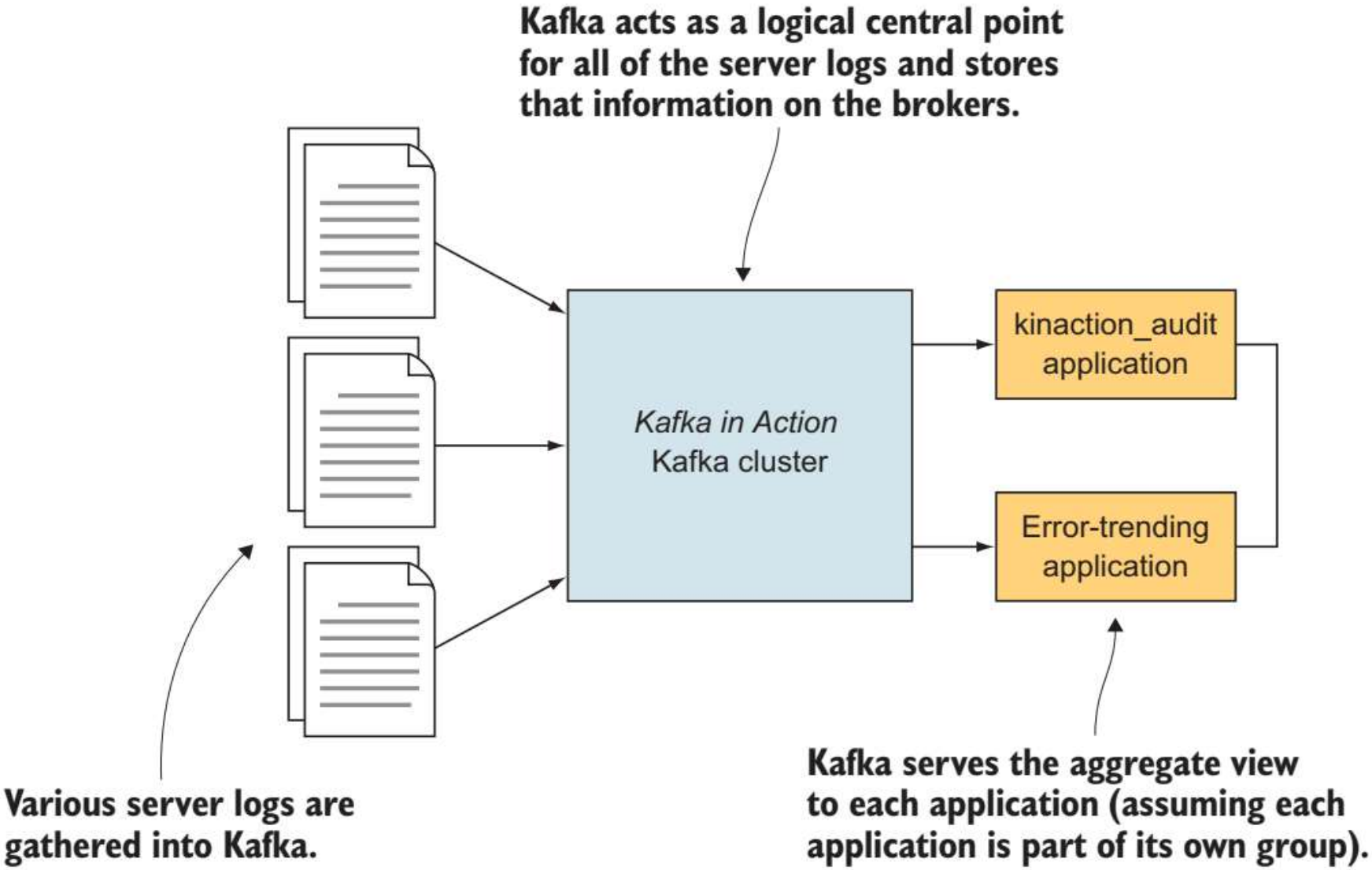
At-least-once message flow



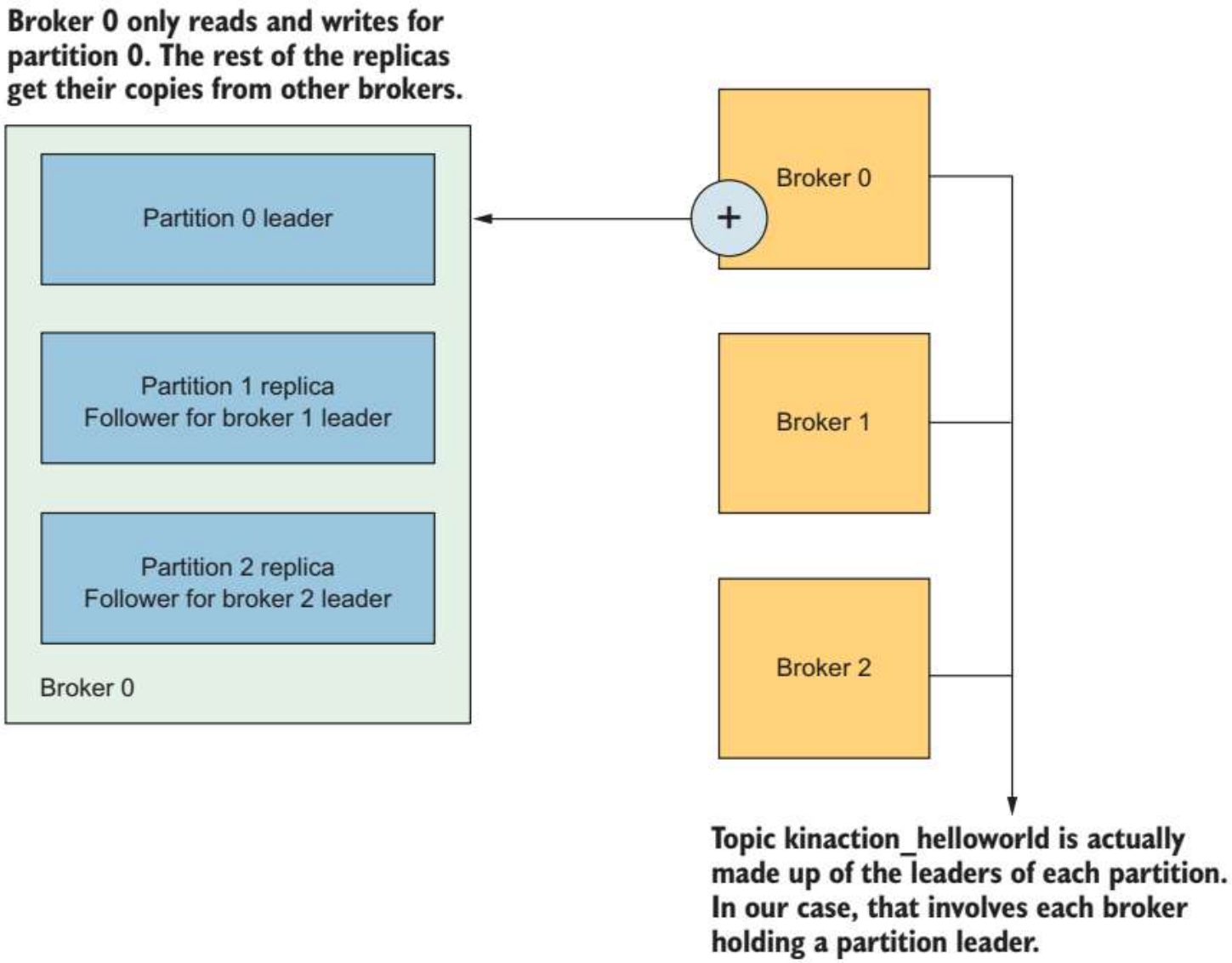
At-most-once message flow



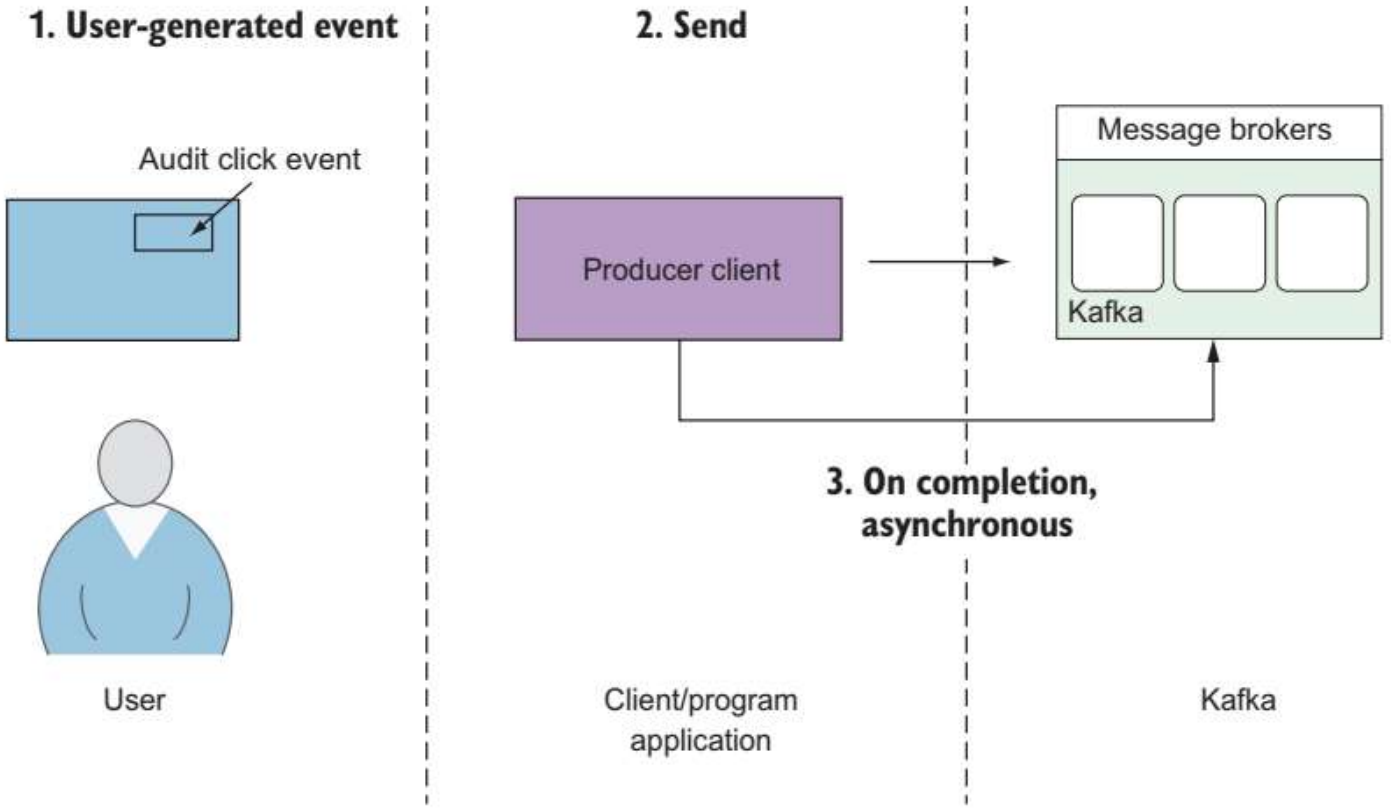
Exactly-once message flow



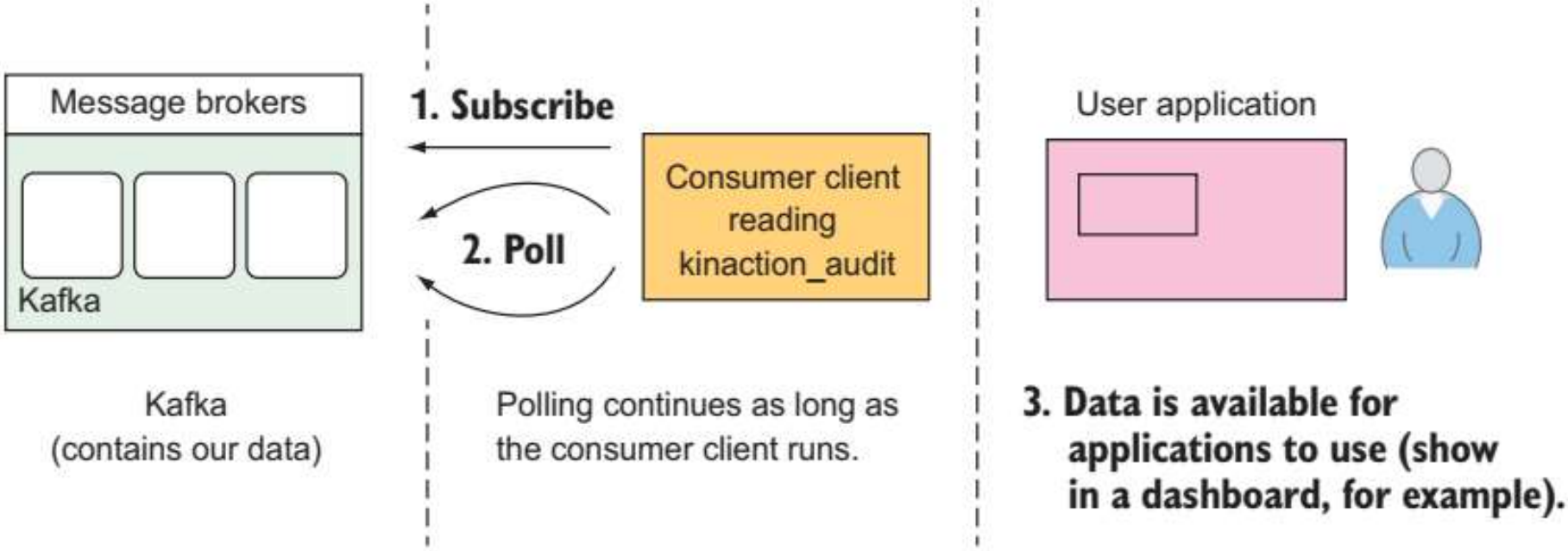
Kafka log aggregation



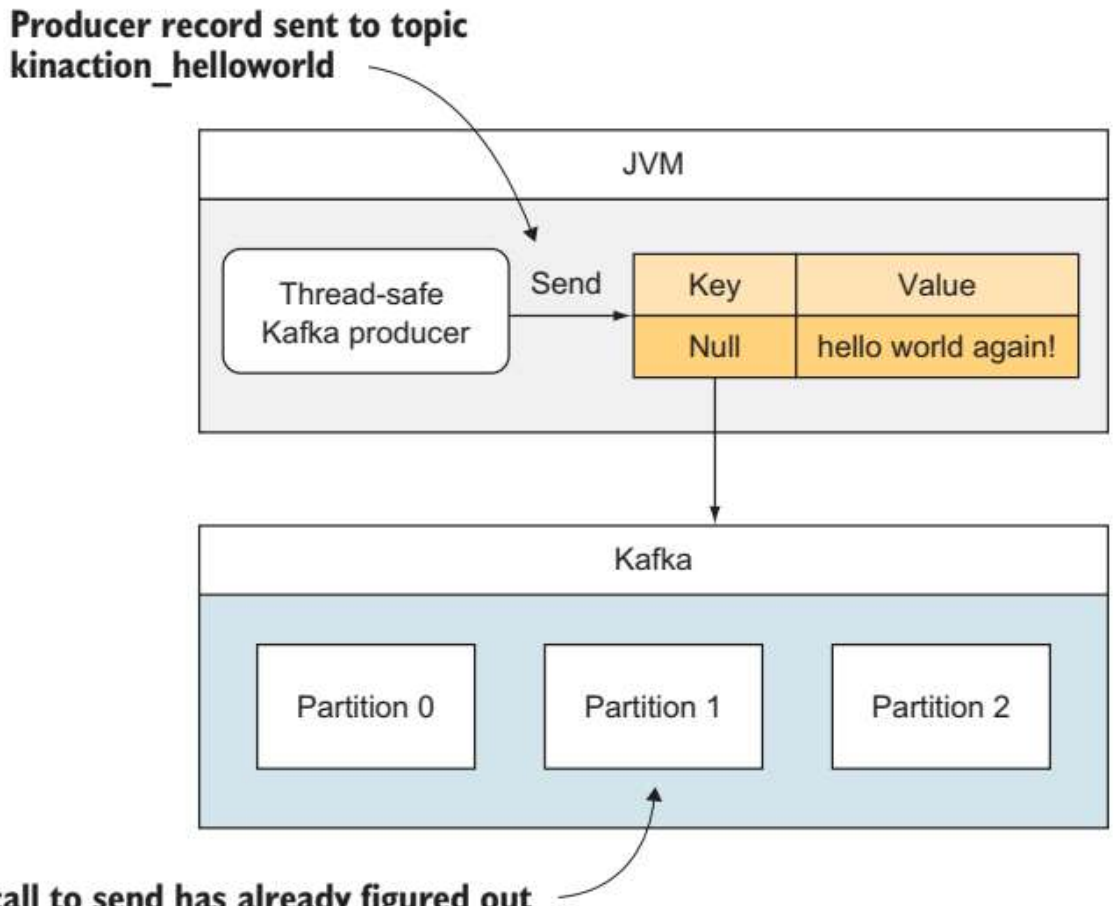
View of one broker



Producer example for user event



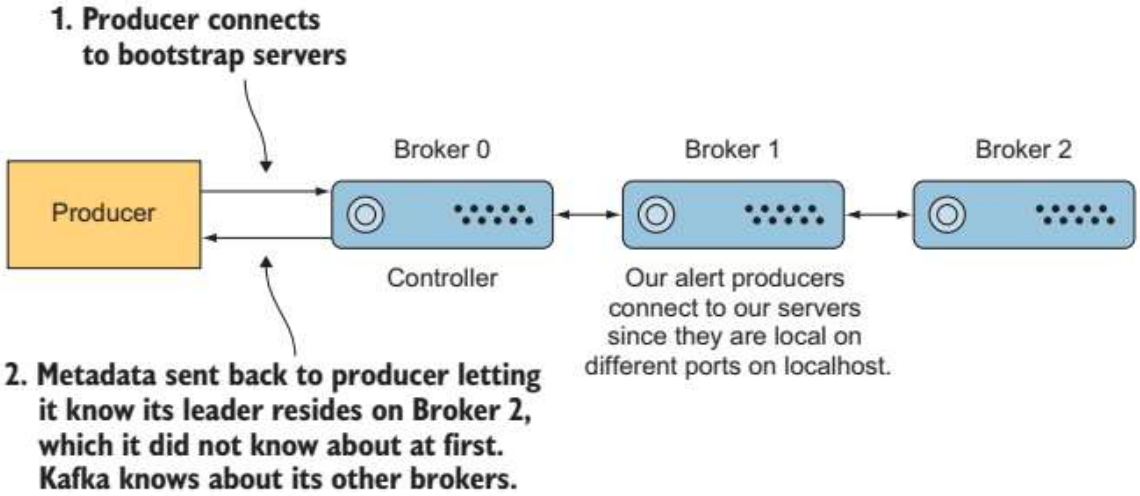
Consumer example flow

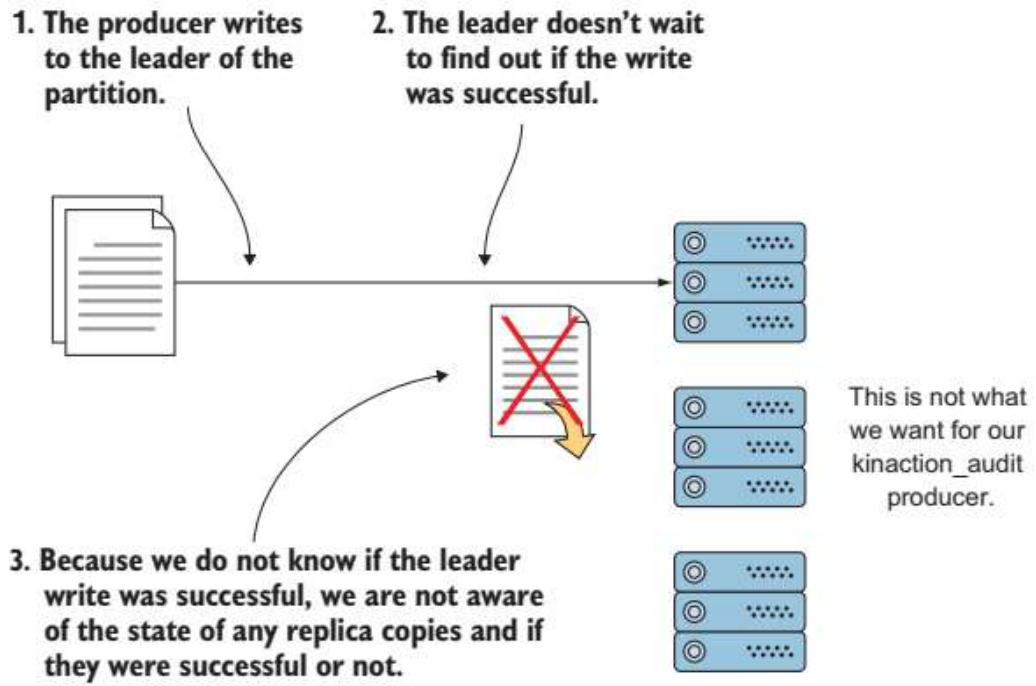


Producer flow

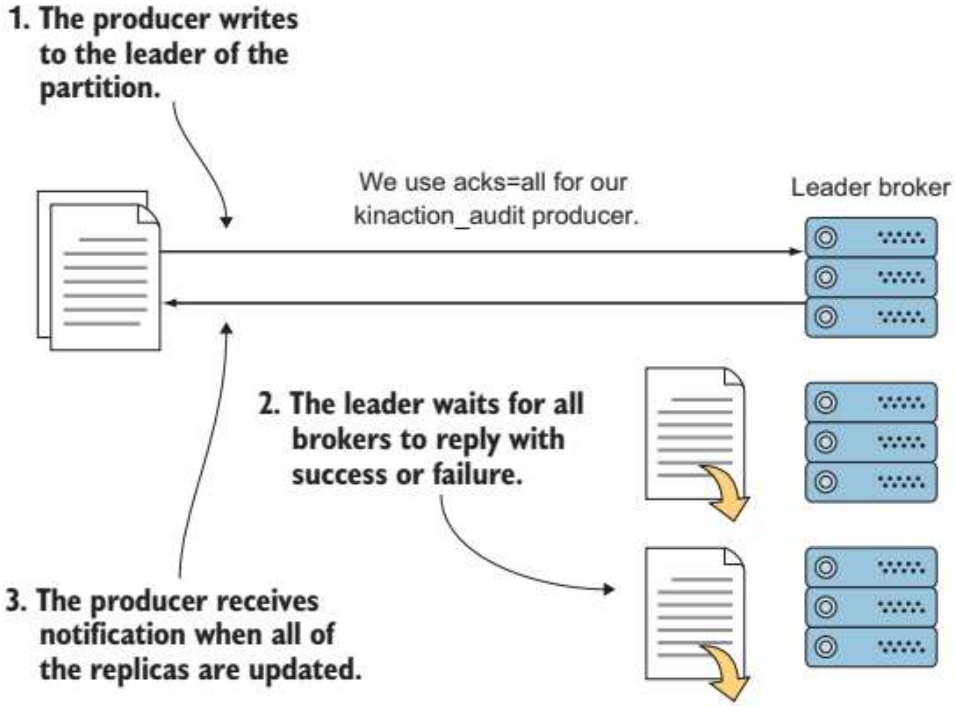
Important producer configurations

Key	Purpose
`acks`	Number of replica acknowledgments that a producer requires before success is established
`bootstrap.servers`	One or more Kafka brokers to connect for startup
`value.serializer`	The class that's used for serialization of the value
`key.serializer`	The class that's used for serialization of the key

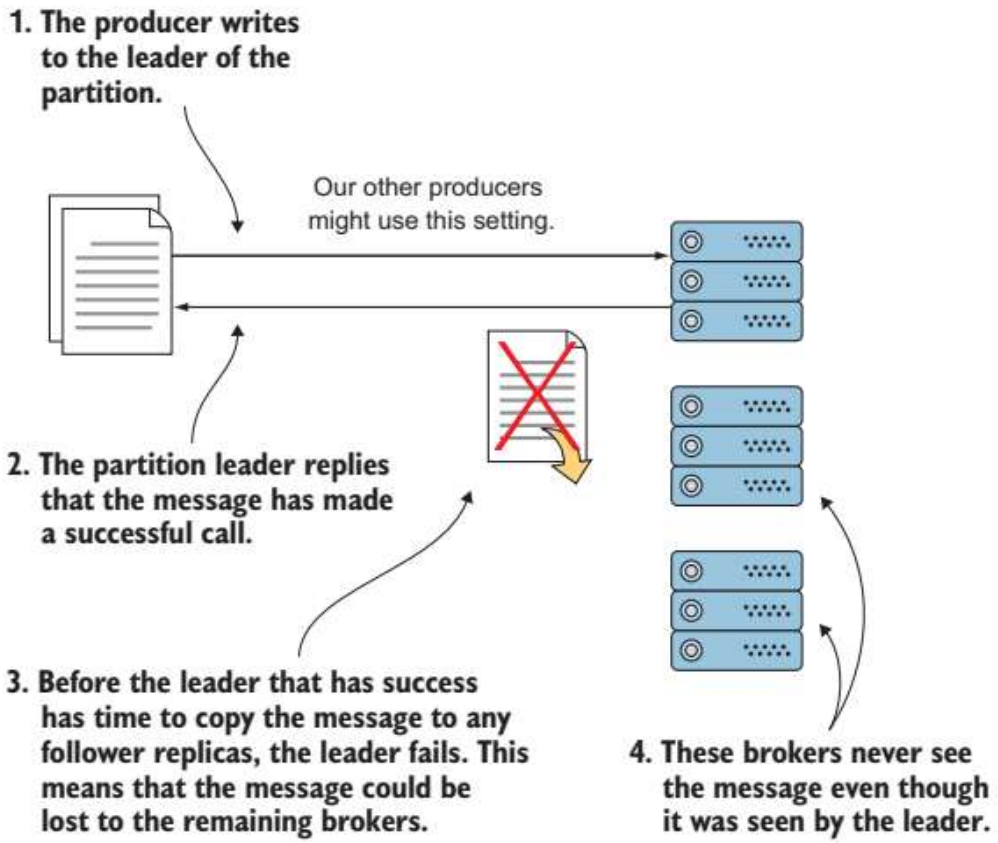

1. Producer connects to bootstrap servers
Broker 0
Broker 1
Broker 2
Producer
Controller
Our alert producers connect to our servers since they are local on different ports on localhost.
2. Metadata sent back to producer letting it know its leader resides on Broker 2, which it did not know about at first. Kafka knows about its other brokers.
Bootstrap servers



The property `acks` equals `0`.



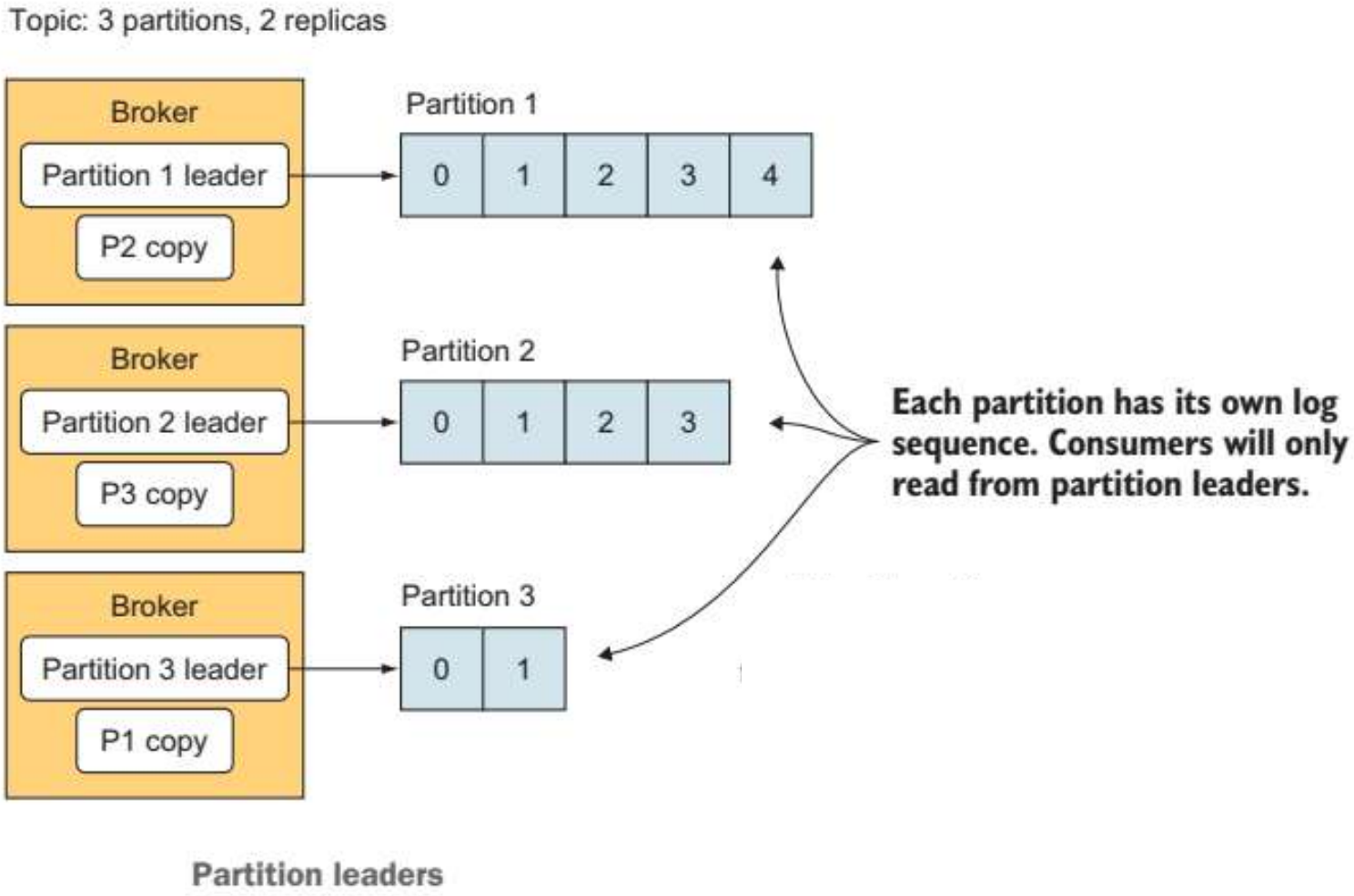
The property `acks` equals `all`.

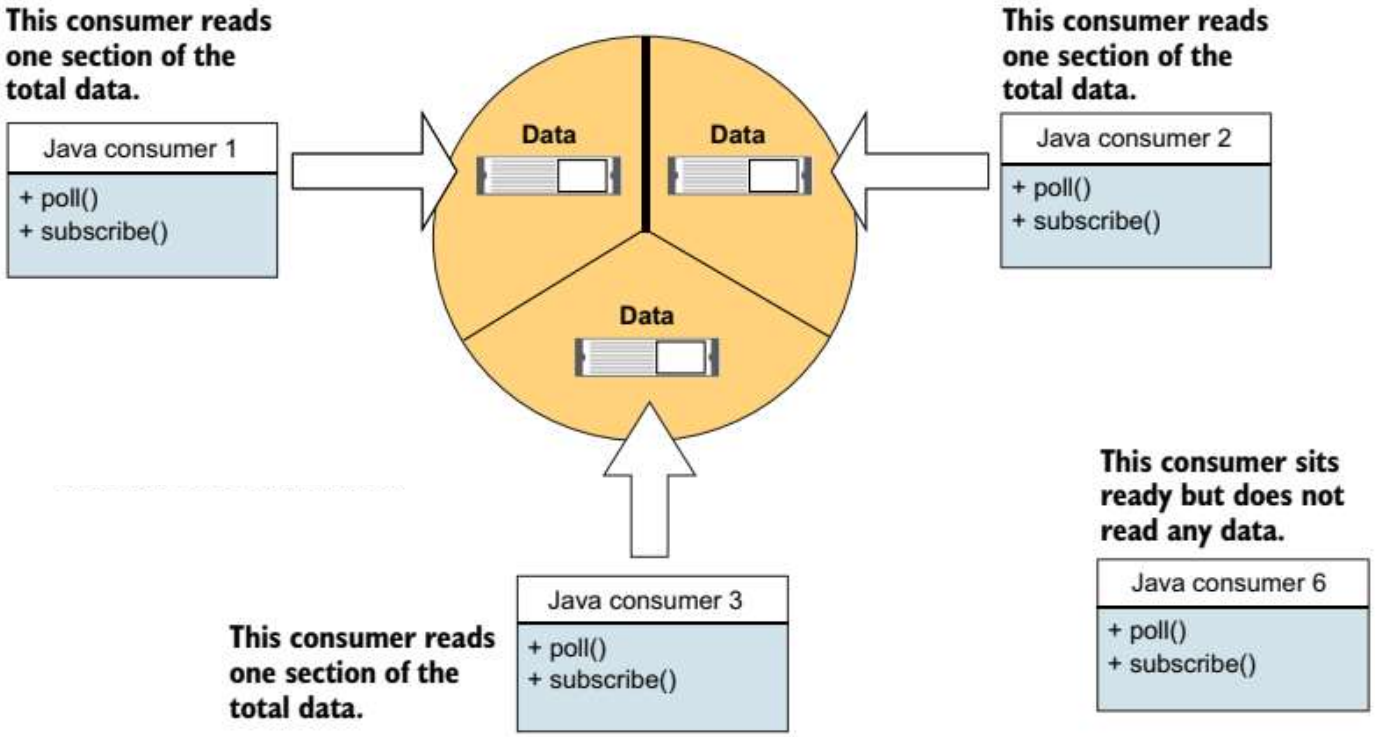


The property `acks` equals `1`.

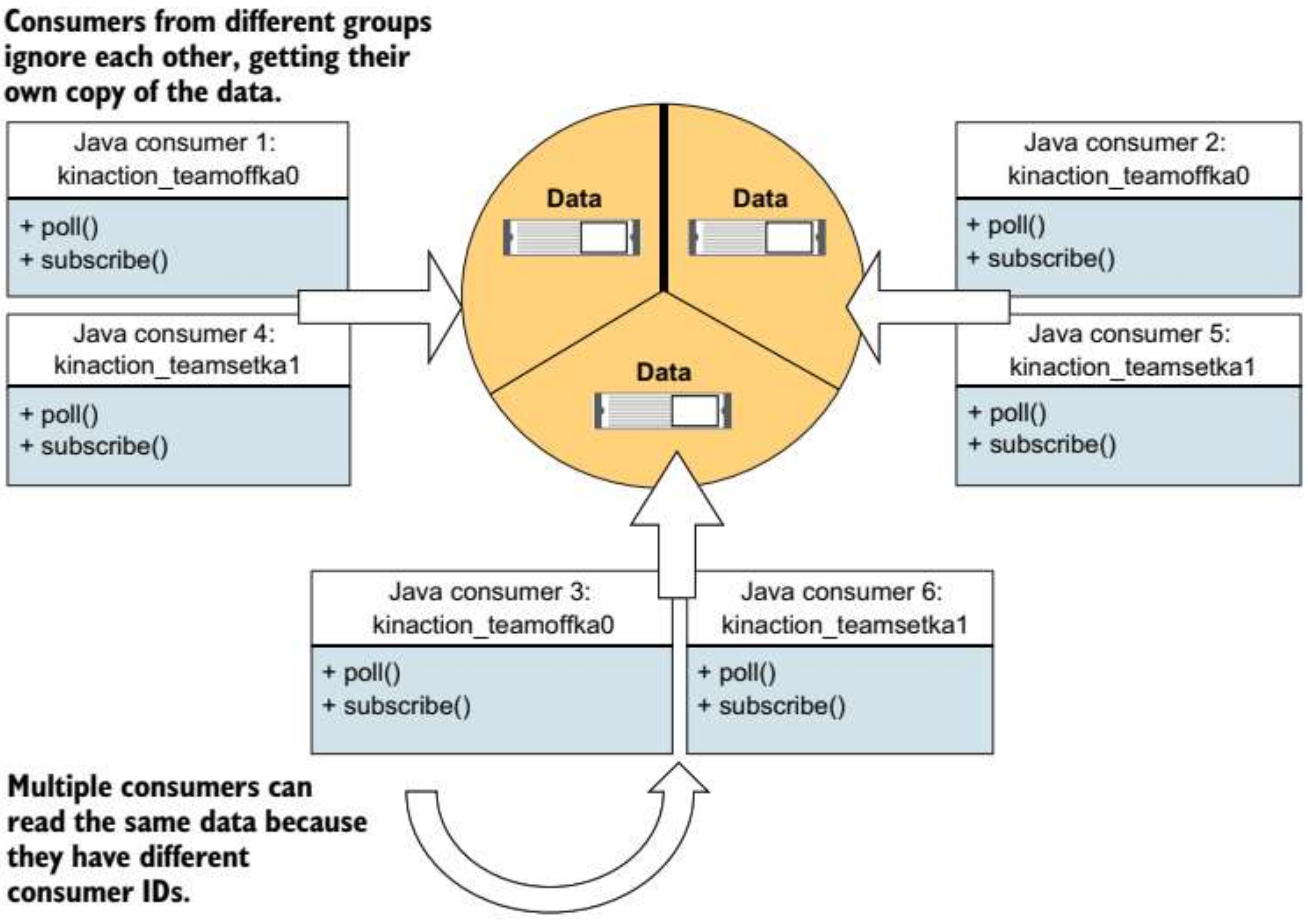
Consumer configuration

Key	Purpose
`bootstrap.servers`	One or more Kafka brokers to connect on startup
`value.deserializer`	Needed for deserialization of the value
`key.deserializer`	Needed for deserialization of the key
`group.id`	A name that's used to join a consumer group
`client.id`	An ID to identify a user
`heartbeat.interval.ms`	Interval for consumer's pings to the group coordinator

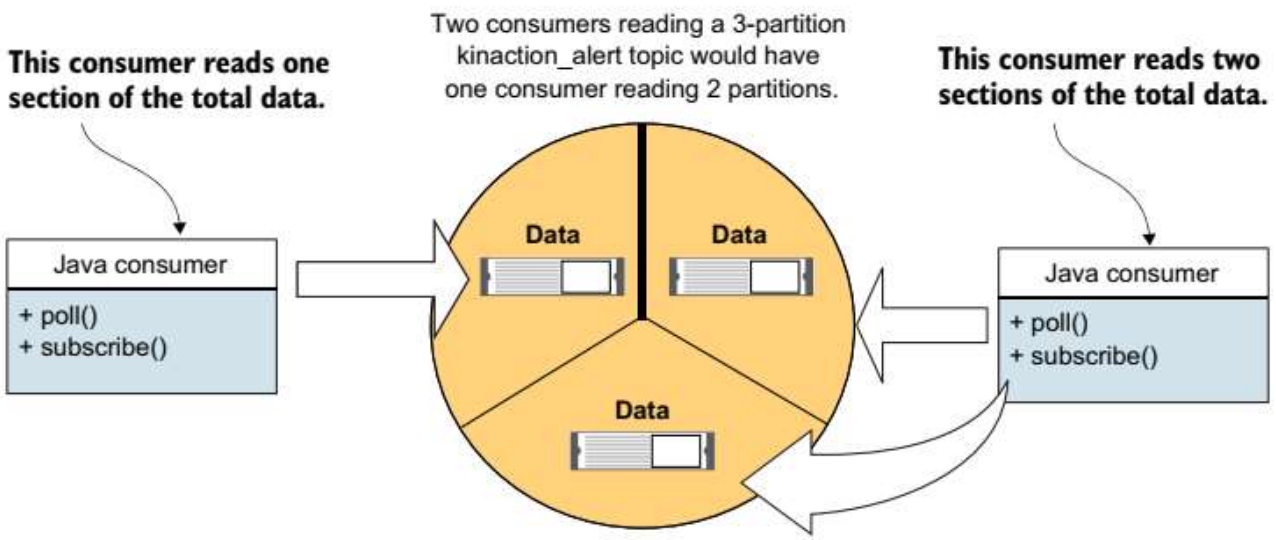

Topic: 3 partitions, 2 replicas
Broker
Partition 1 leader
P2 copy
Partition 1
0
1
2
3
4
Broker
Partition 2 leader
P3 copy
Partition 2
0
1
2
3
Broker
Partition 3 leader
P1 copy
Partition 3
0
1
Each partition has its own log sequence. Consumers will only read from partition leaders.
Partition leaders



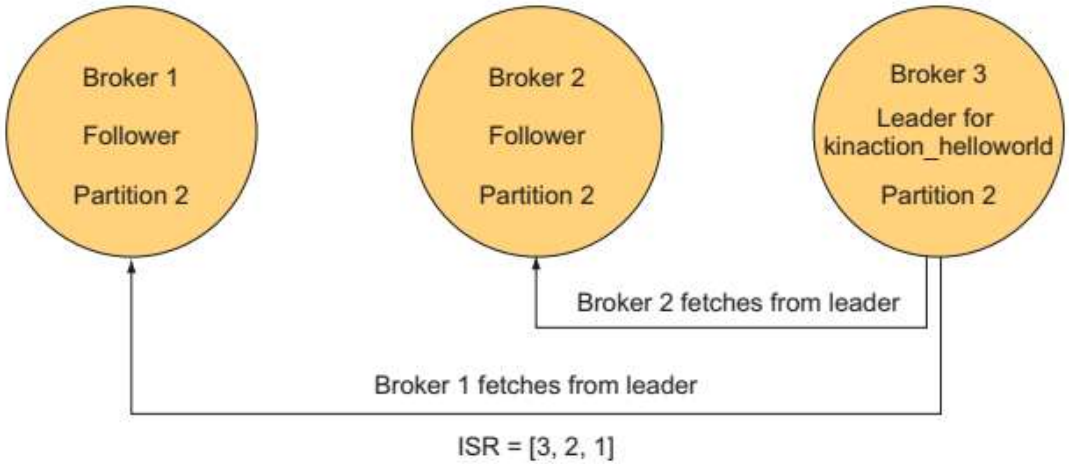
An extra Kafka consumer



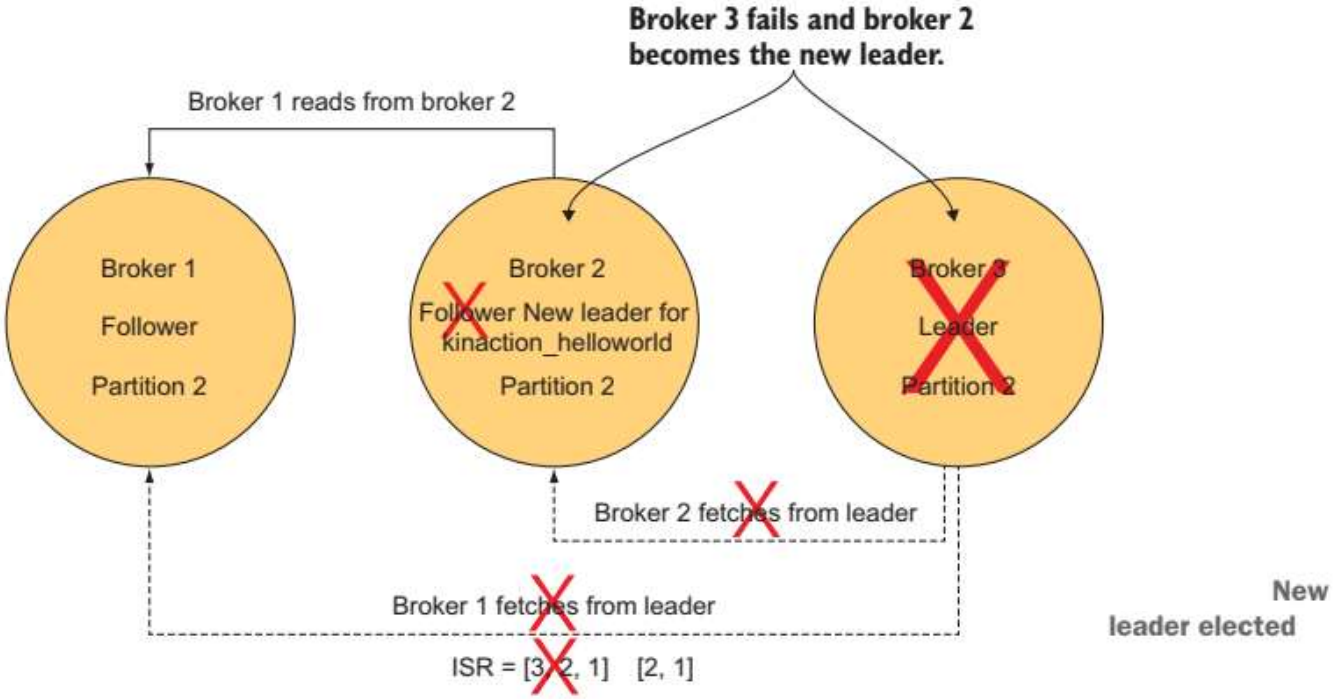
Consumers in separate groups [12]

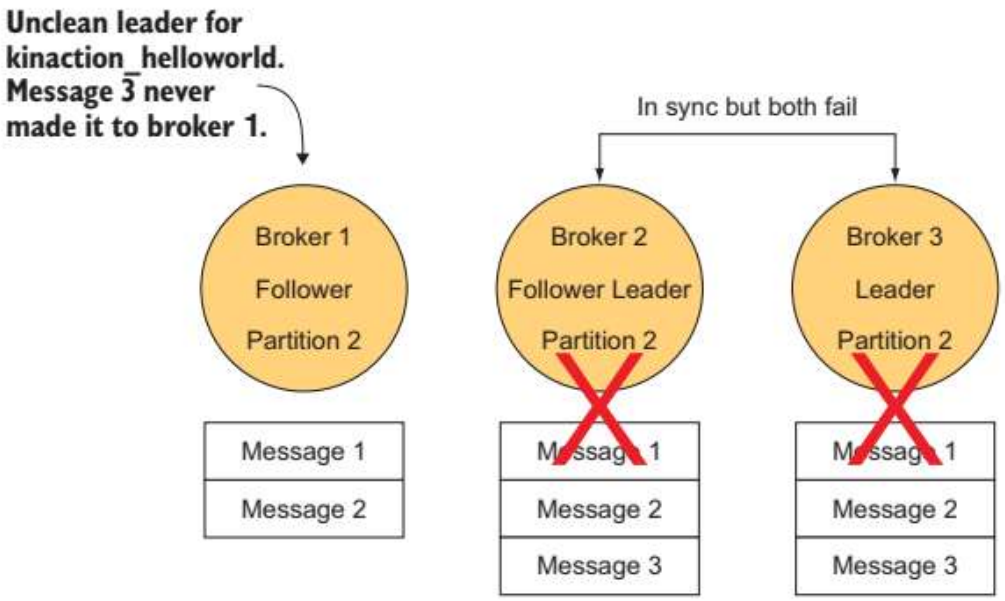


Kafka consumers in a group

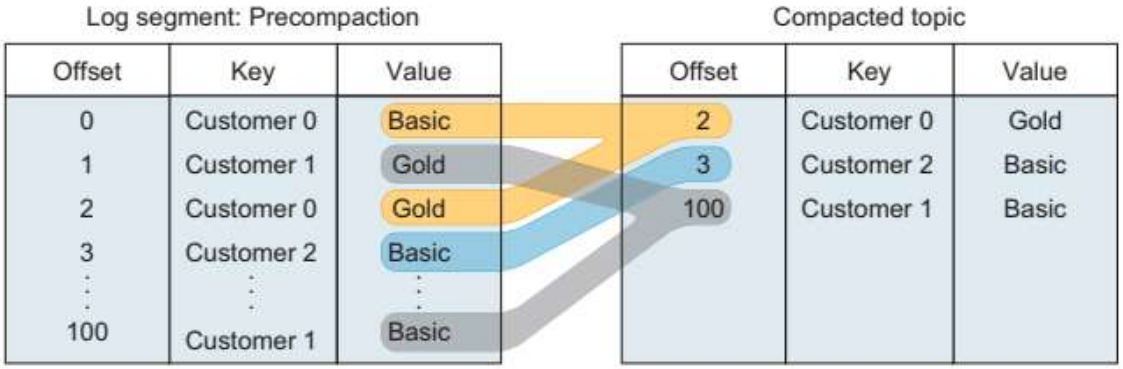

Broker 1
Follower
Partition 2
Broker 2
Follower
Partition 2
Broker 3
Leader for
kinaction_helloworld
Partition 2
Broker 2 fetches from leader
Broker 1 fetches from leader
ISR = [3, 2, 1]

Leader


Broker 3 fails and broker 2 becomes the new leader.
Broker 1 reads from broker 2
Broker 1
Follower
Partition 2
Broker 2
Follower New leader for kinaction_helloworld
Partition 2
Broker 3
Leader
Partition 2
Broker 2 fetches from leader
Broker 1 fetches from leader
ISR = [3, 2, 1] [2, 1]
New leader elected



Unclean leader election

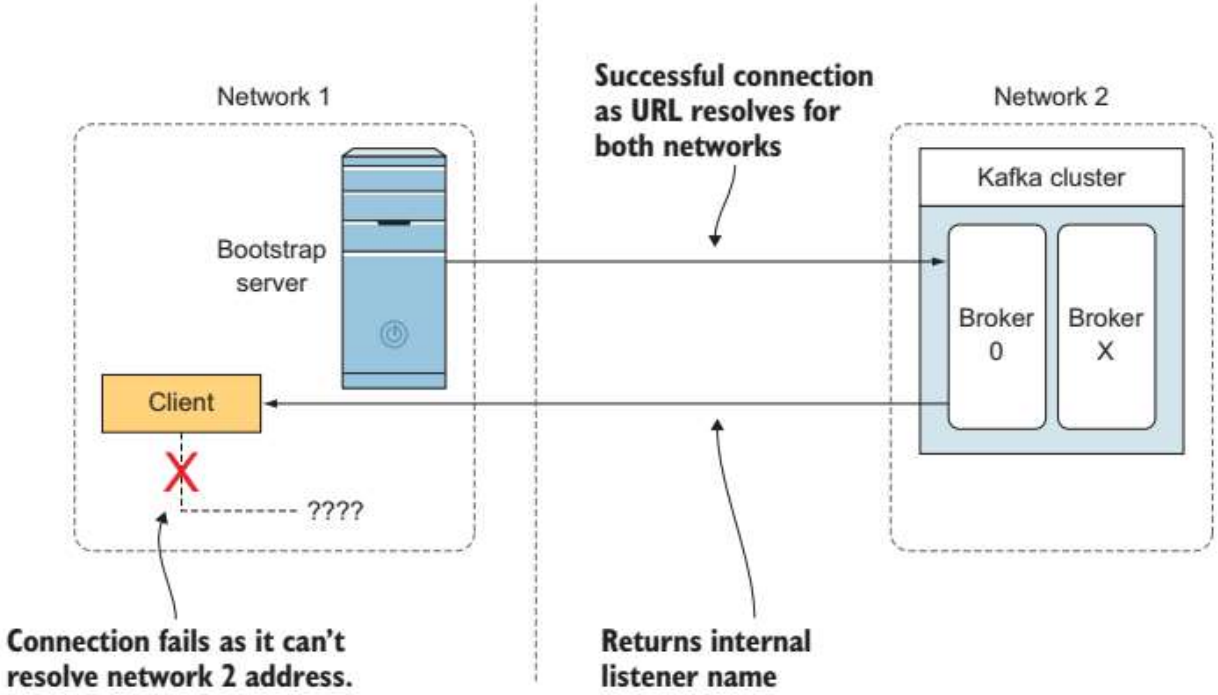


Compaction in general

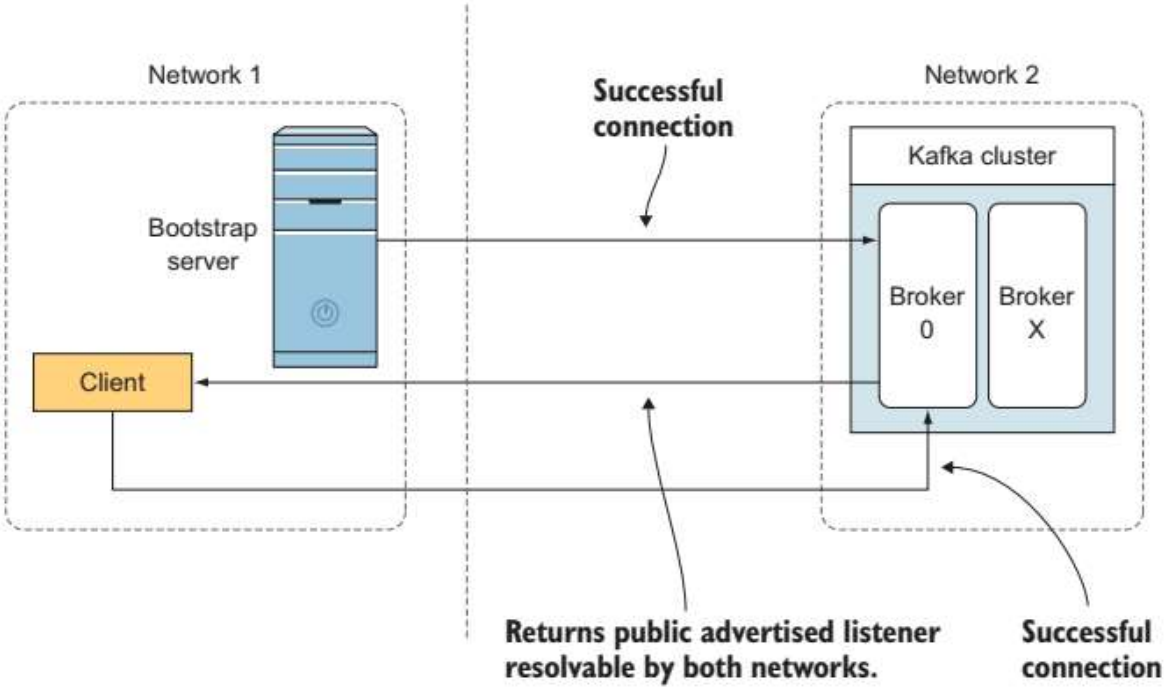
Broker retention configuration

Key	Purpose
`log.retention.bytes`	The largest size threshold in bytes for deleting a log.
`log.retention.ms`	The length in milliseconds a log will be maintained before being deleted.
`log.retention.minutes`	Length before deletion in minutes. `log.retention.ms` is used as well if both are set.
`log.retention.hours`	Length before deletion in hours. `log.retention.ms` and `log.retention.minutes` would be used before this value if either of those are set.

Scenario 1: no advertised listeners. Producer client starts and requests metadata from bootstrap server.



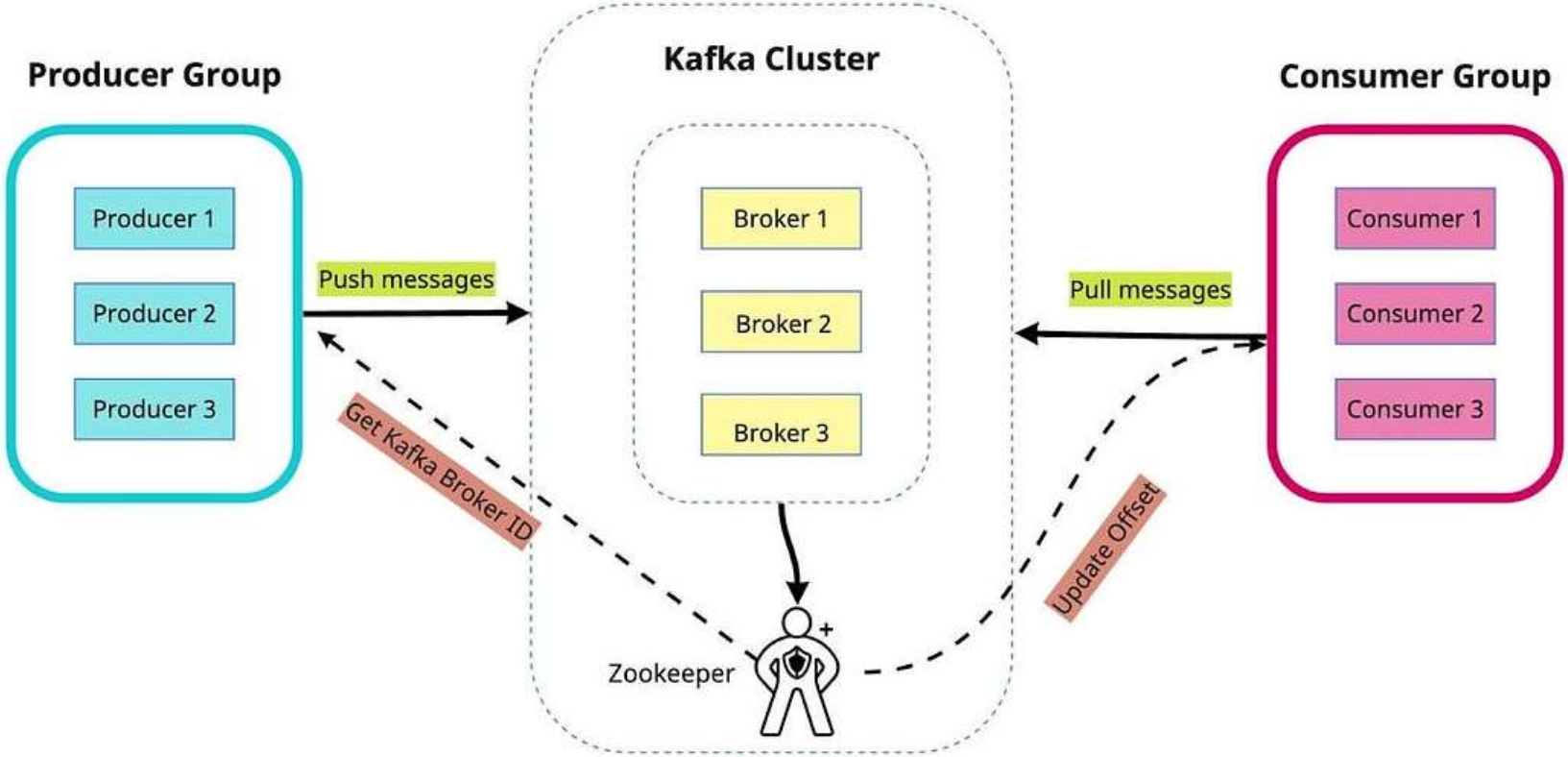
Scenario 2: advertised listeners with URL resolved by both networks. Producer client requests metadata.



Kafka Internals

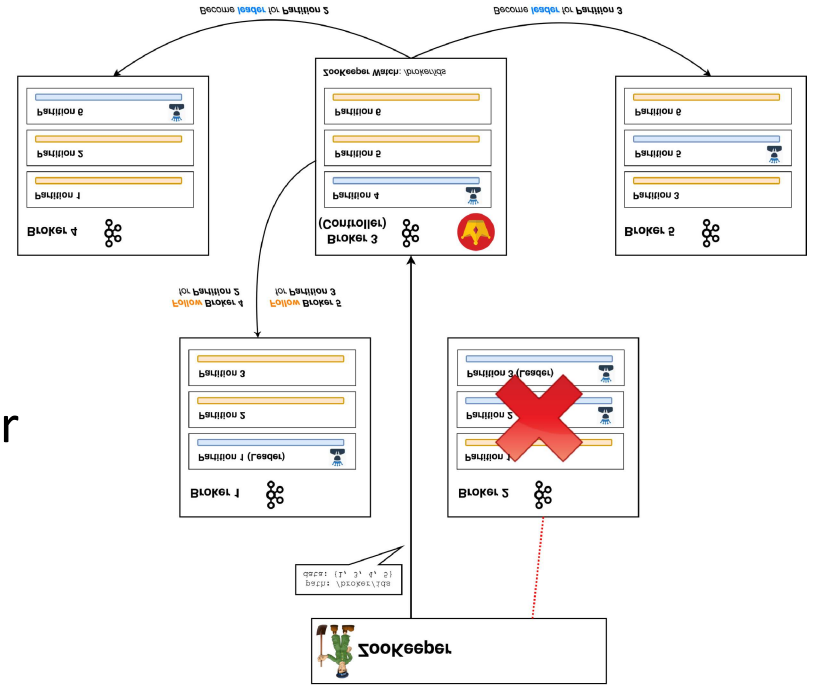
Cluster Membership

- Every Kafka broker gets registered with ZK with their Id.

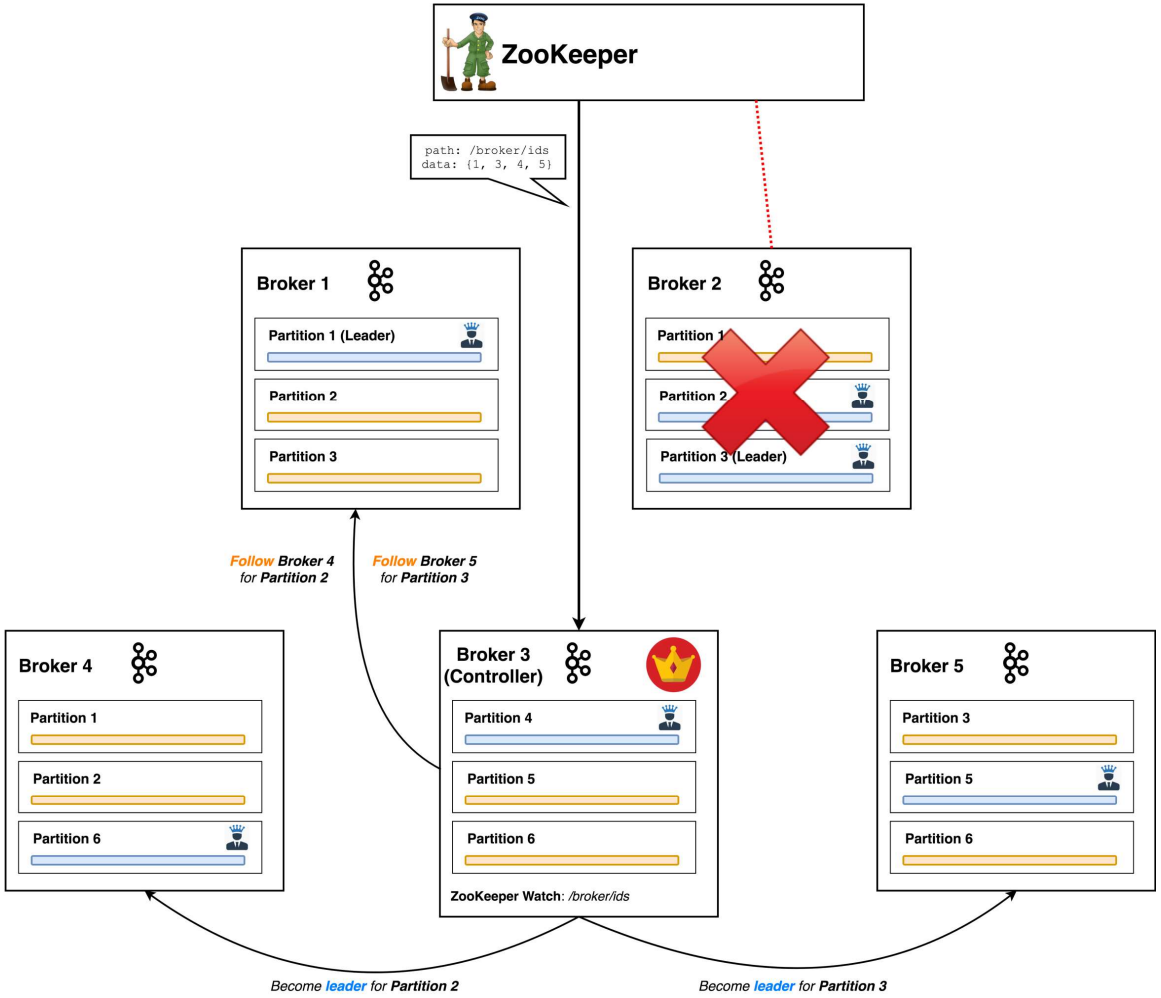


The Controller

- Kafka service that runs on every broker in a Kafka cluster
- Only one can be active (elected) at any point in time.
- The process of promoting a broker to be the active controller is called Kafka Controller Election.
- One of the brokers serves as the controller
  - Responsible for managing
    - The states of partitions and Replicas and
    - For performing administrative tasks like
      - reassigning partitions
  - Responsible for communicating with ZooKeeper and the other brokers in the cluster.

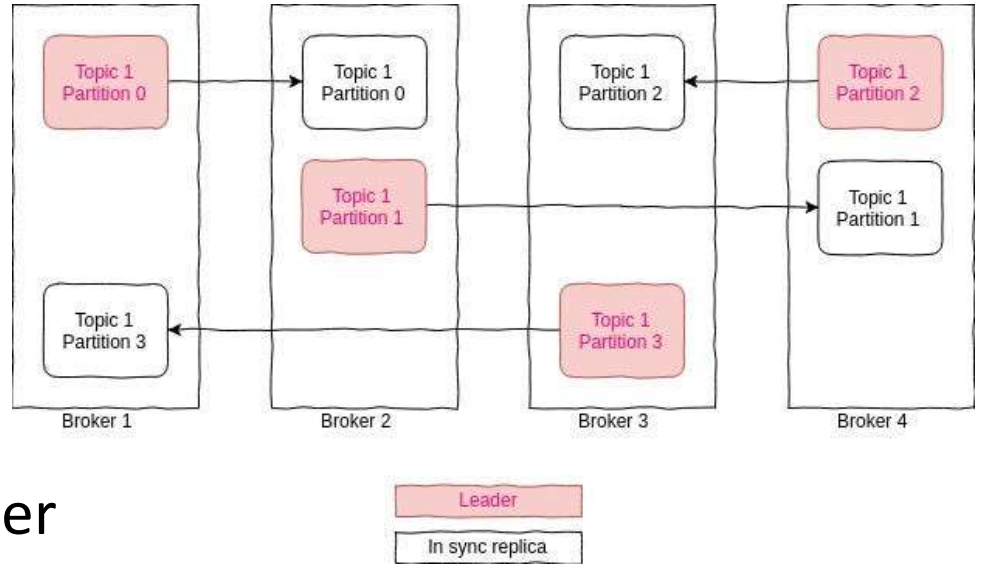


The Controller

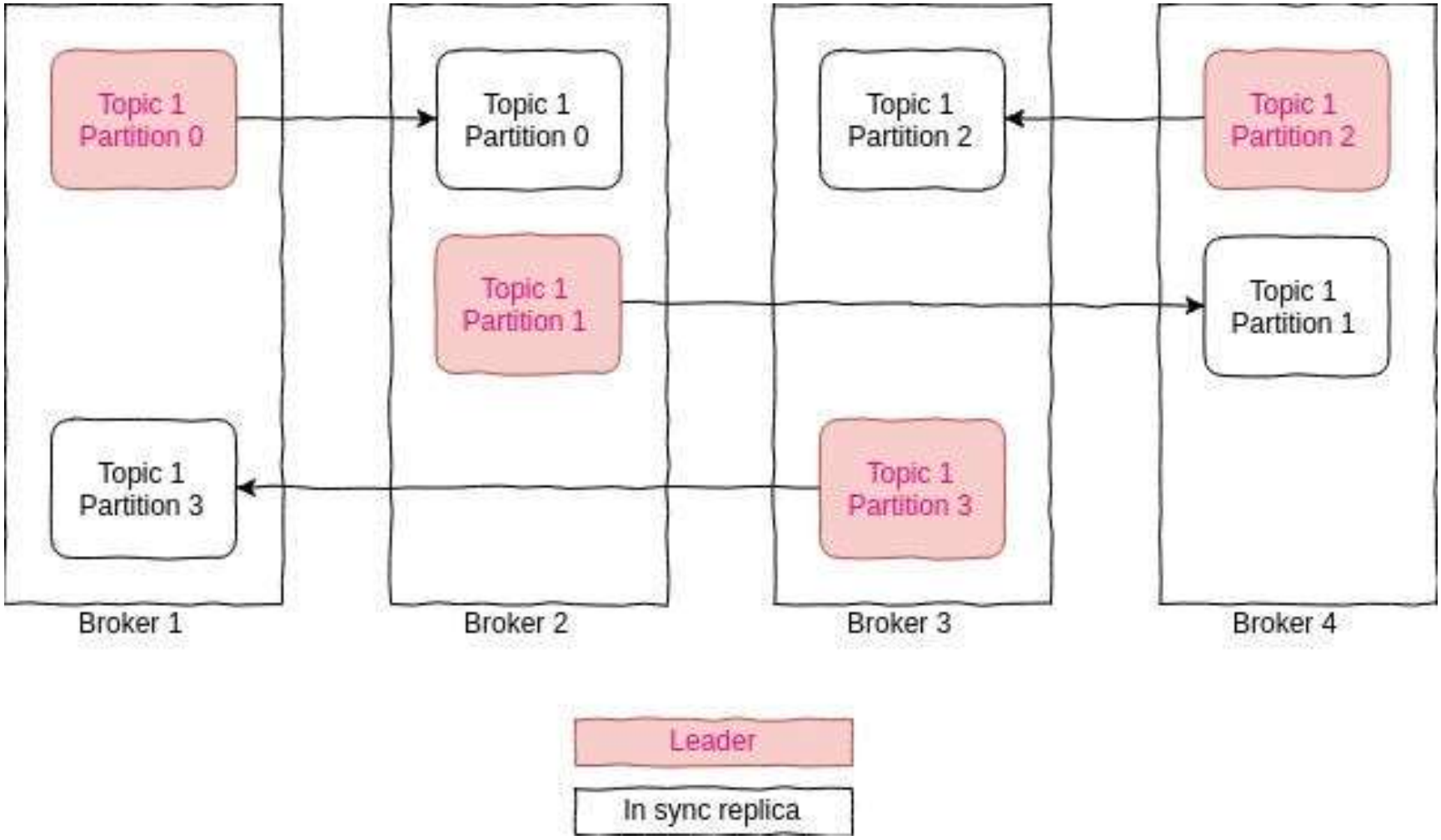


Replication

- leader replica
  - All requests are performed through a leader (ensuring consistency)
- follower replica
  - All replicas that are not leaders
  - Only copies messages from the leader
- To determine whether a replica is ISR,
  the replica makes a request to the leader
- replica.lag.time.max.ms -> 10 seconds

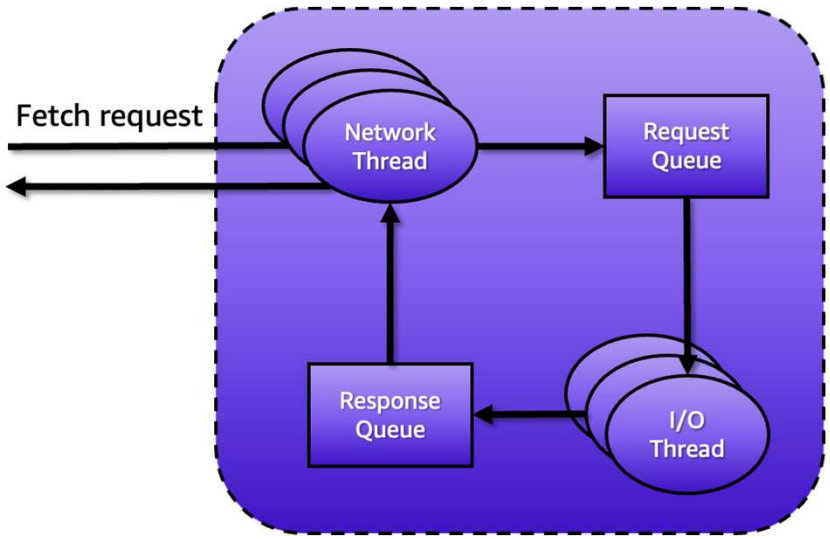


Replication



Request Processing

- A broker has two threads and two queues.
  - acceptor thread
  - processing thread
  - request queue
  - response queue

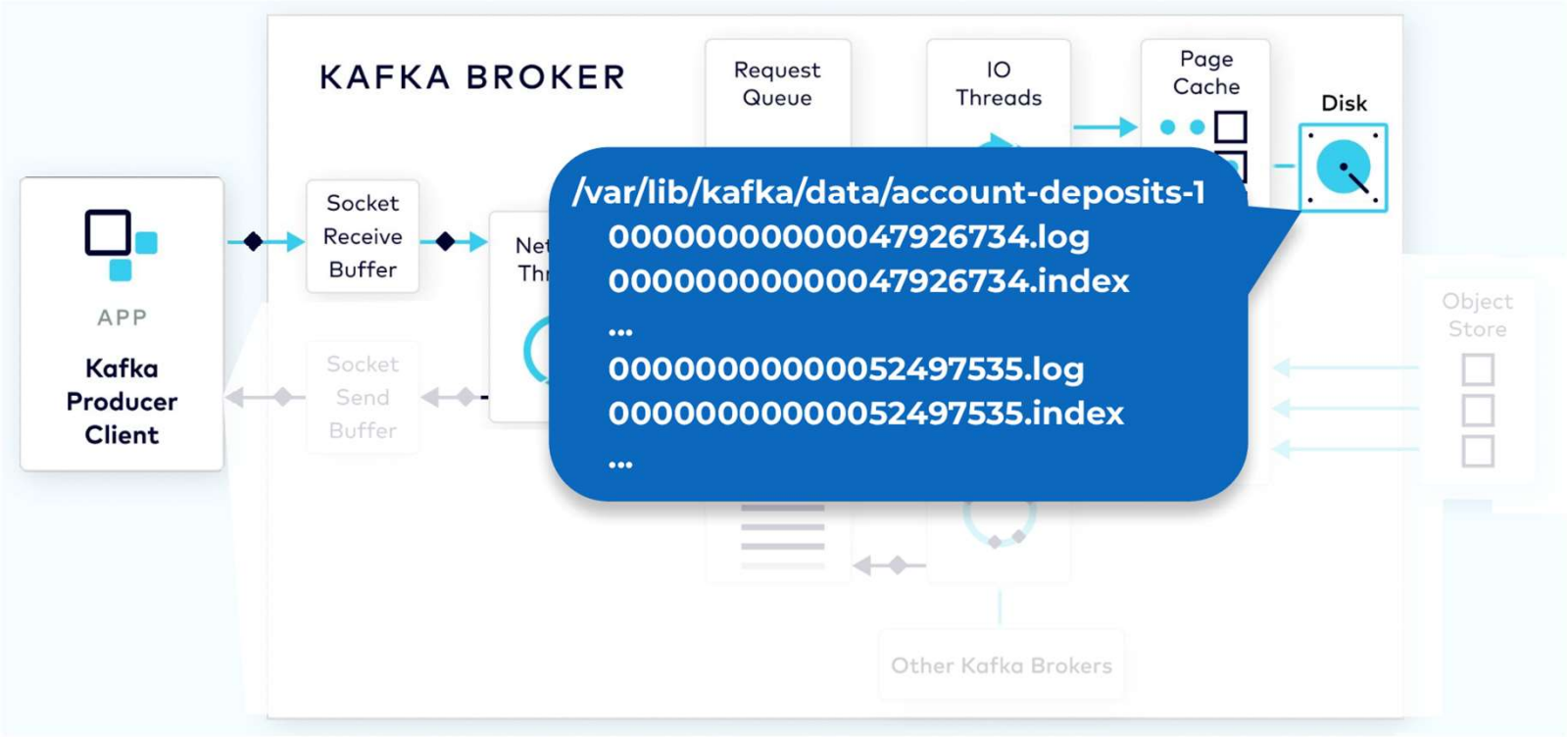


- A connection is established through the client's acceptor thread, and the processor thread stores the request in the Request Queue.

- Additionally, the processor thread receives requests in the request queue, processes them, and stores them in the response queue.

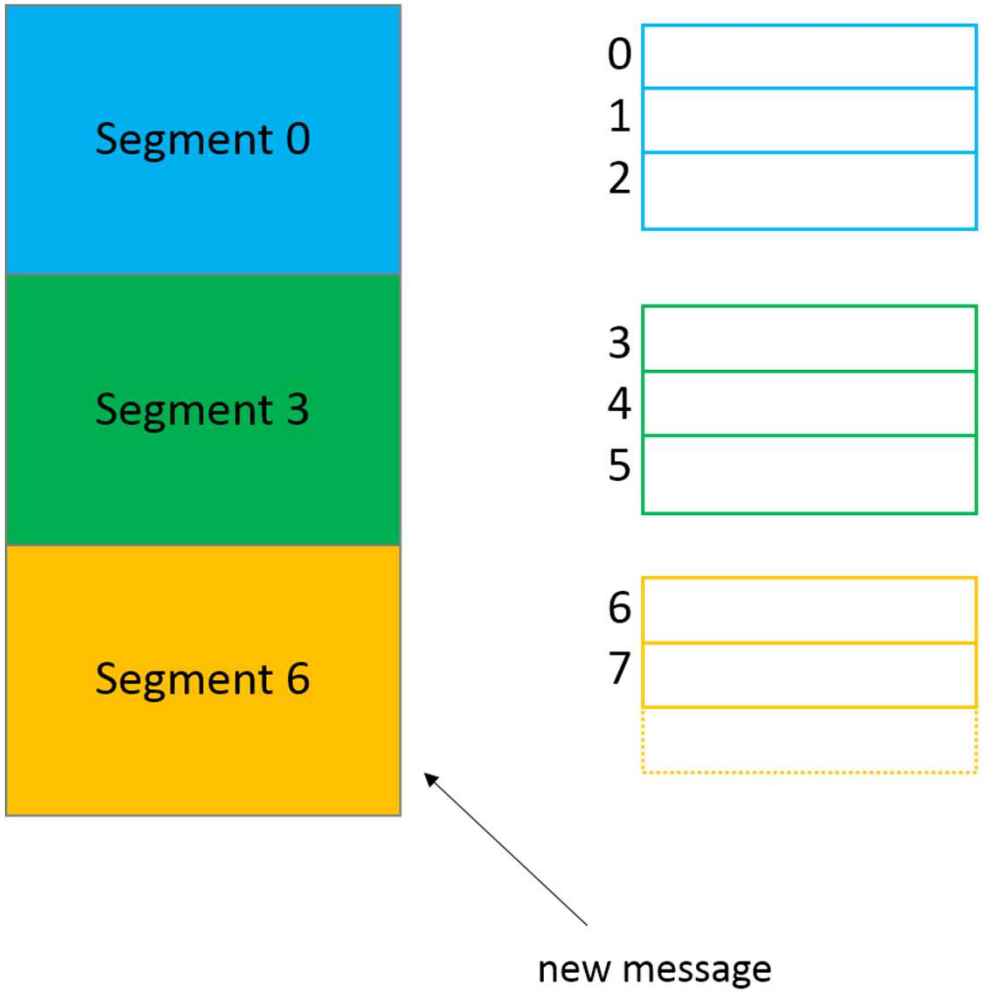
Physical Storage

- Kafka basic storage unit: partition replication
  - One partition cannot be split into multiple brokers.
  - Partitions within one broker cannot be stored across multiple disks.



Partition Allocation

- Replicas of one partition exist on different brokers
- Equal distribution of replicas between brokers
- Randomly, assign partition leader using round-robin method
- When adding a broker, no auto partition rebalancing
  - Use replica reassignment tool
    - bin/kafka-reassign-partitions.sh



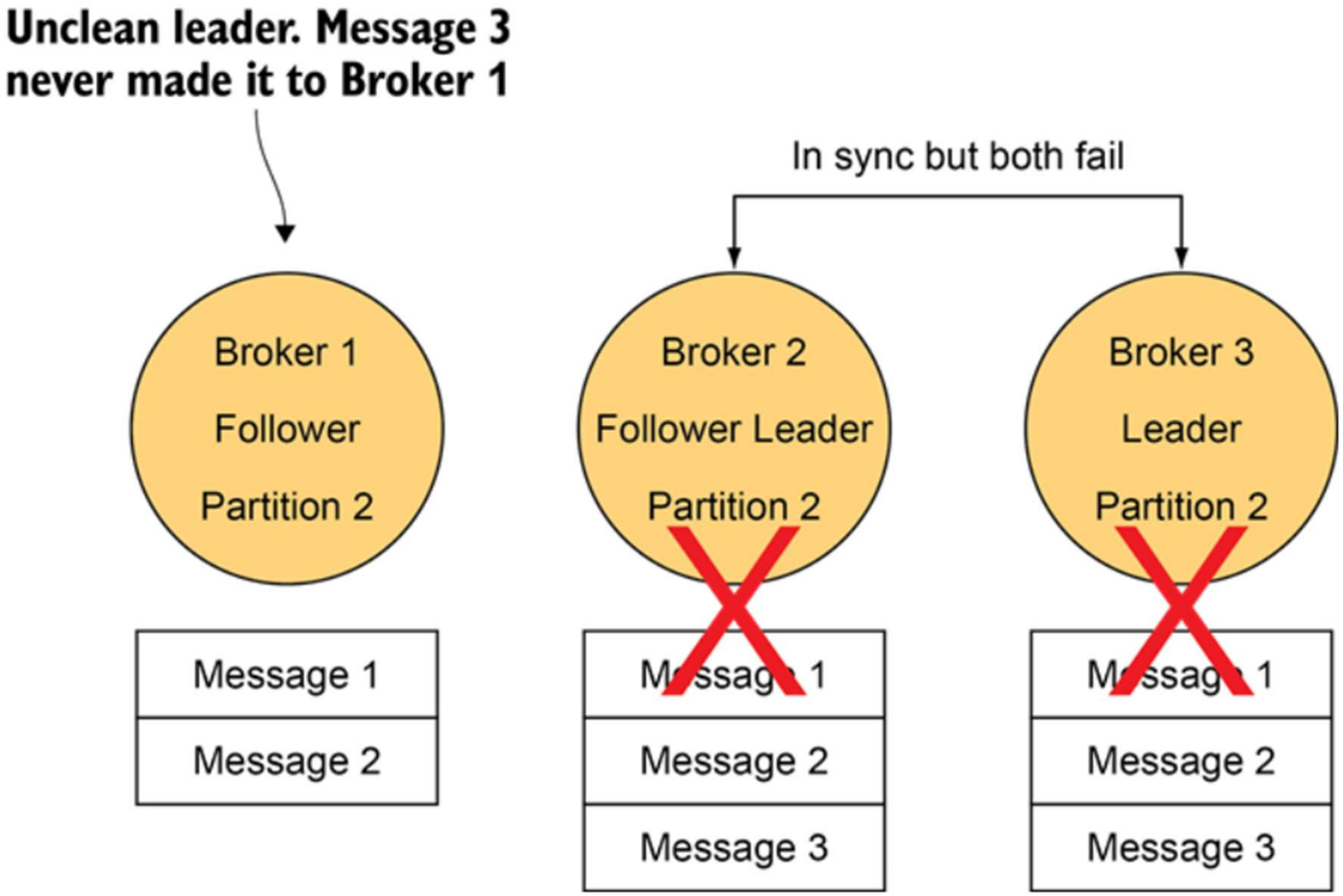
File Management

- Kafka does not store messages permanently
- Does not wait for the consumer to read the message
- Delete messages according to retention period
- Index
  - How can a broker quickly find a message with a valid offset?
  - Maintain/manage an index in each partition to quickly find messages with a given offset
  - Mapping offsets to segment files and positions within the file

Reliability

- Kafka is extremely flexible
- There are two important components to understand reliability
  - Broker
  - Producers

Reliability - **Broker**



Reliability: **Producers**

1. Ack = 0
    1. Highly unreliable option.
2. Ack = 1
    1. Leader broker will send successful ack once data is written to its partition
    2. It carries a risk of data not being replicated to another in-sync replicas
    3. In case leader crashes after sending ack to producer
3. Ack = ALL
    1. It is the safest option
    2. Provides highest reliability along with Min in-sync replica option
    3. But it causes slowness in performance

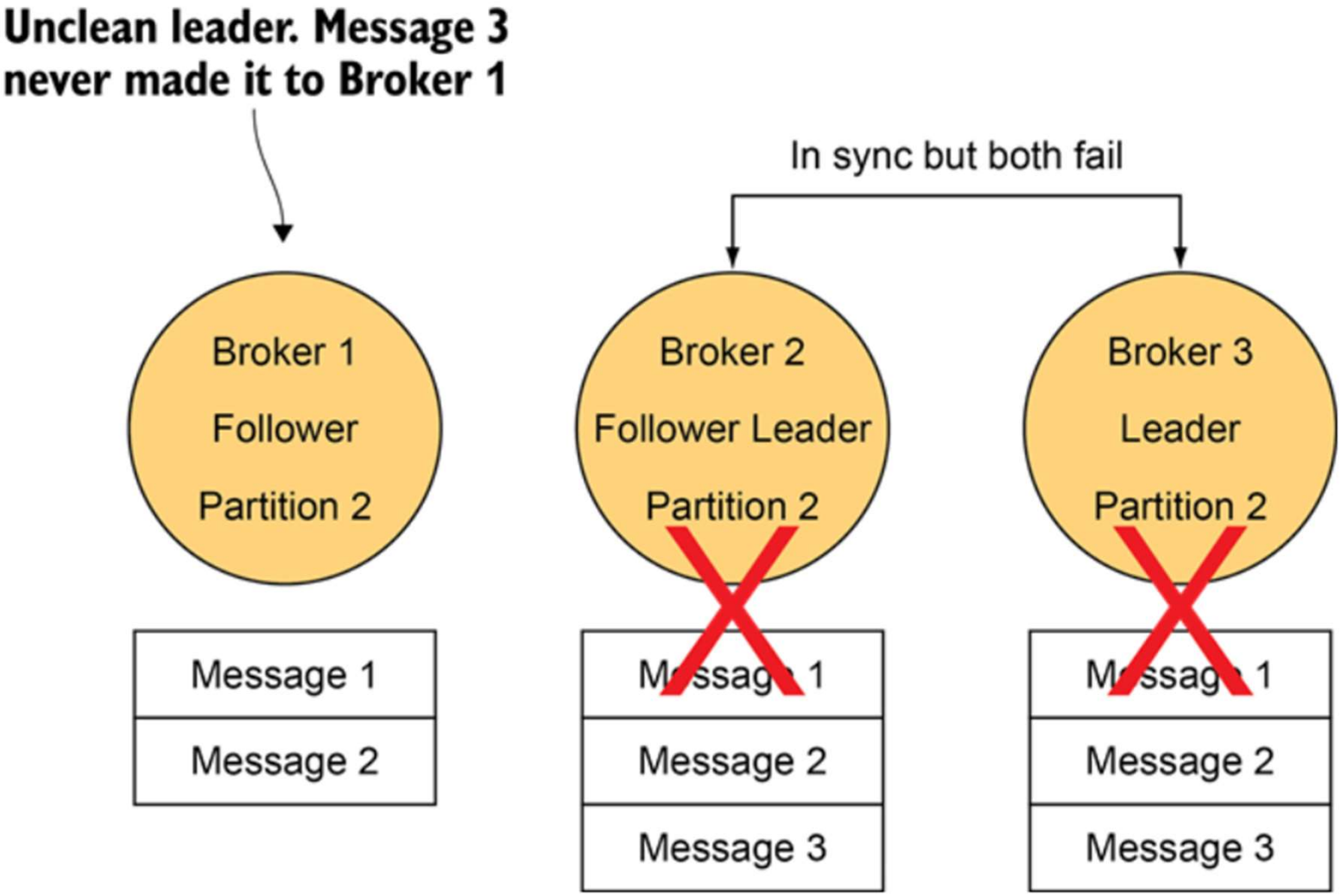
Broker Configuration

- Basic broker configuration
  - num.partitions=1
  - default.replication.factor=3
  - offsets.topic.replication.factor=3
  - log.retention.hours=168
  - log.segment.bytes=1073741824
  - log.retention.check.interval.ms=300000
  - num.network.threads=3
  - num.io.threads=8
  - socket.send.buffer.bytes=102400
  - socket.receive.buffer.bytes=102400
  - socket.request.max.bytes=104857600
  - zookeeper.connection.timeout.ms=6000
  - # ...

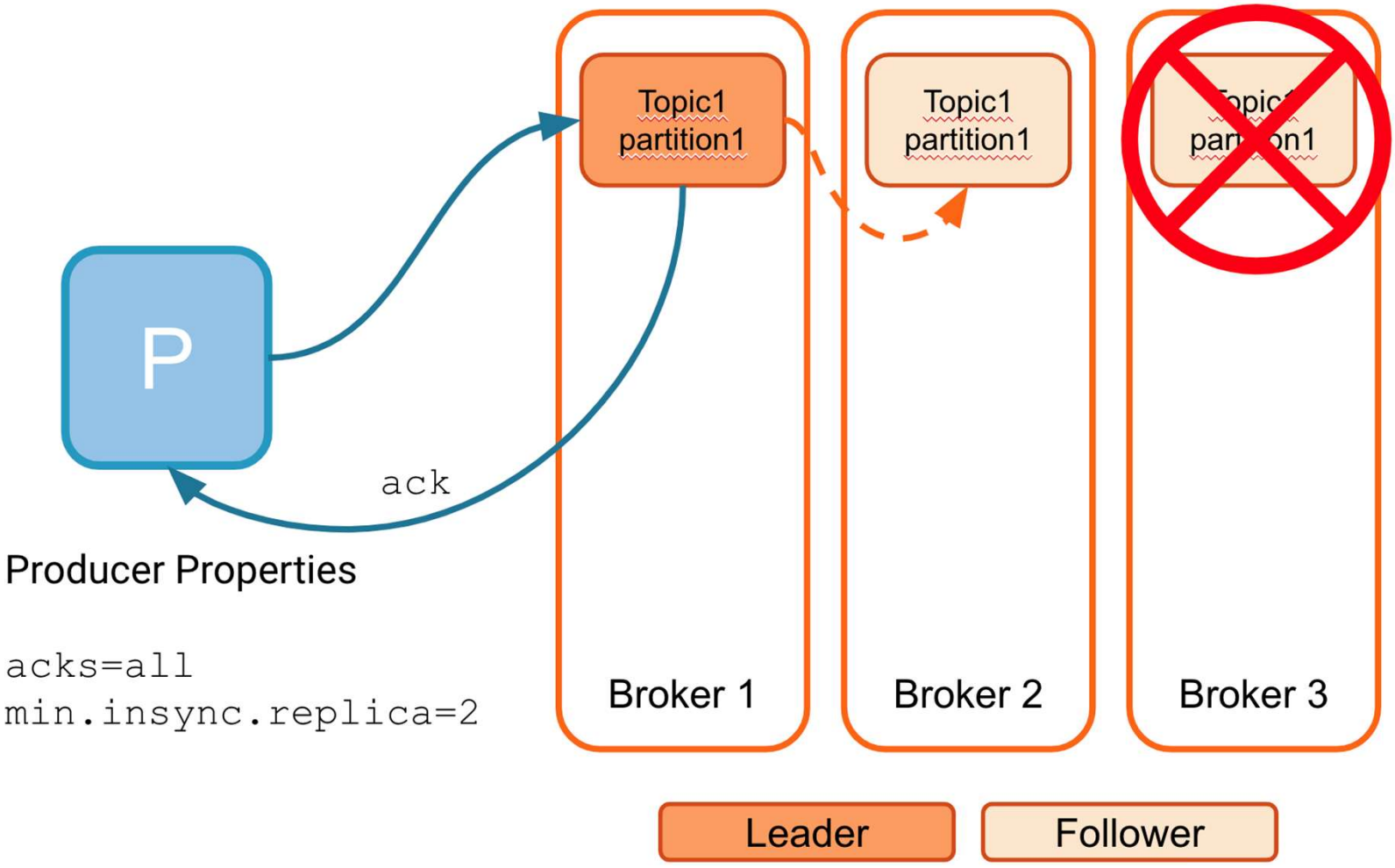
Broker Configuration

- Replicating topics for high availability
  - # ...
  - num.partitions=1
  - default.replication.factor=3
  - min.insync.replicas=2
  - # ...
  - # ...
  - auto.create.topics.enable=false
  - delete.topic.enable=true
  - # ...

Unclean Leader Election



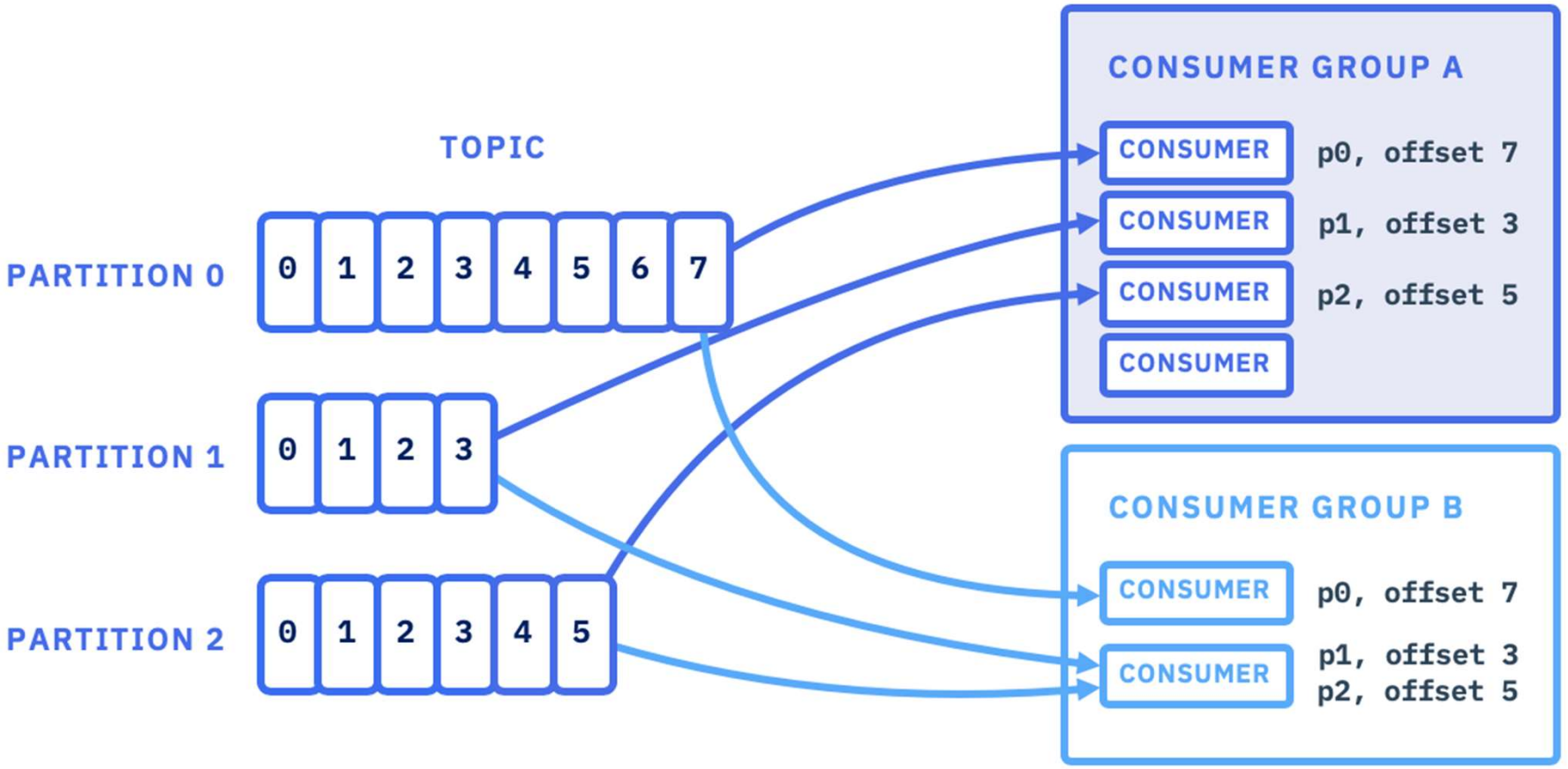
Minimum In-Sync Replicas



Using Producers in a Reliable System

- Use the correct acks configuration to match reliability requirements
  - acks=0
  - acks=1
  - acks=all

Using Consumers in a Reliable System

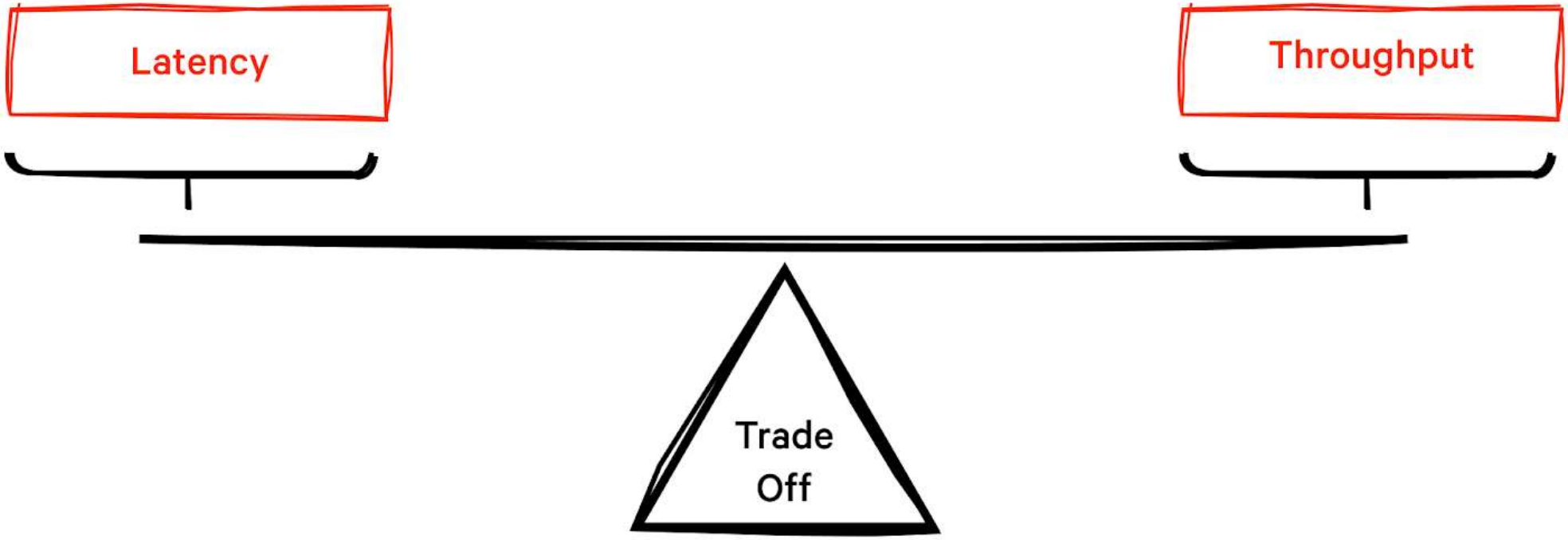


Validating System Reliability

- kafka-verifiable-consumer.sh
- kafka-verifiable-producer.sh
- kafka-replica-verification.sh

Performance Tuning in Kafka

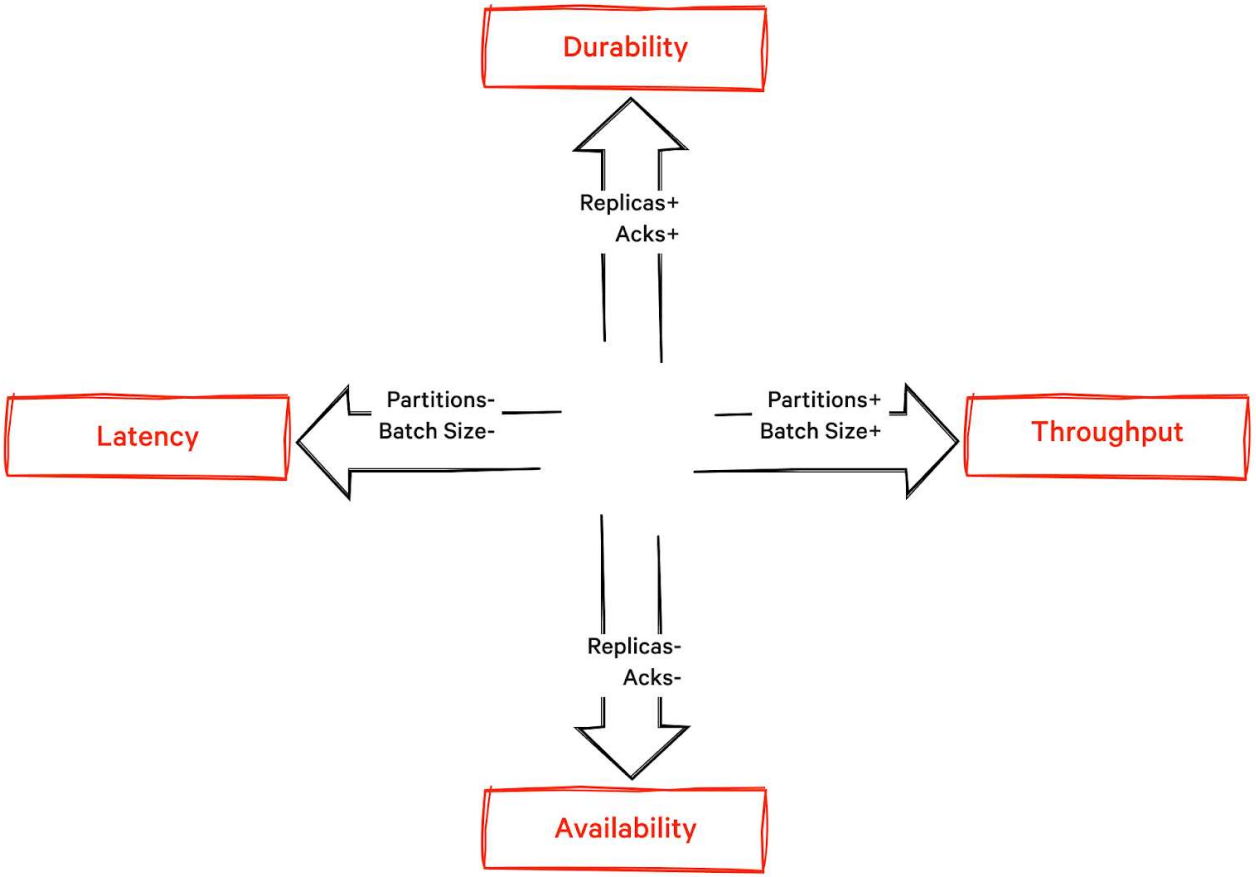
Key Kafka performance metrics



Kafka performance tuning strategies

Tuning Brokers	Balancing partition count and replication factor.
Tuning Producers	Batching data, setting batch size, linger time, and compression.
Tuning Consumers	Setting the consumer number to match partition count, using efficient code and libraries.
Handling Large Messages	Strategies for handling large messages in Kafka, such as compression and message segmentation
Operating System Optimization	Optimizing the operating system for Kafka performance, including file system tuning, network settings, and kernel parameters
Hardware Optimization	Optimizing hardware for Kafka performance, including CPU, memory, storage, and network considerations

Acknowledgment



Kafka Monitoring

Introduction

- The Apache Kafka applications have numerous measurements for their operation
- Kafka runs on the JVM (Java Virtual Machine).
- All of the metrics exposed by Kafka can be accessed via the Java Management Extensions (JMX) interface
- The easiest way to use them in an external monitoring system is to use a collection agent provided by your monitoring system and attach it to the Kafka process.

Metric Basics

- All of the metrics exposed by Kafka can be accessed via the Java Management Extensions (JMX) interface
- Metrics provided via an interface such as JMX are internal metrics
- Common places to host the Kafka metrics are:
  - ELK (ElasticSearch + Kibana)
  - Datadog
  - NewRelic
  - Confluent Control Centre
  - Prometheus

Kafka Metrics to monitor

- There are many metrics exposed by different Kafka components providing information about nearly every function of that component.
  - Under Replicated Partitions
  - OfflinePartitionsCount
  - Request Handlers
  - Request timing
  - Active Controller Count

Kafka Broker Metrics

- Incoming/Outgoing Bytes
  - Indicates the volume of data being ingested or sent out by the brokers
- Request Latency
  - Measures the time taken by brokers to process client requests
- Disk Usage
  - To ensures that brokers have sufficient storage capacity
- CPU and Memory Utilization
  - Provides insights into resource utilization

Client Monitoring

- All applications need monitoring
  - Producer metrics
  - Consumer metrics

Producer Metrics

- Message Throughput
  - Allows administrators to assess the load on the cluster and optimize the performance of producers accordingly.
- Message Error Rate
  - Indicates the rate of failed or rejected messages
- Producer Latency
  - Measures the time taken by producers to send messages to brokers

Consumer Metrics

- Consumer Lag
  - Measures the time difference between the latest produced message and the last consumed message for a consumer group
  - Essential for tracking the real-time progress of consumers
- Message Throughput
  - Helps assess the rate at which consumers are processing messages from Kafka.
- Offset Commit Rate
  - Helps ensure that consumers are committing offsets at an appropriate frequency

End-to-End Monitoring

- Provides a client point of view on the health of the Kafka cluster
- If you are responsible for running the Kafka cluster, and not the clients, you would now have to monitor all of the clients as well
- You would be able to monitor this for every topic individually.
- Kafka Monitor continually produces and consumes data from a topic that is spread across all brokers in a cluster.
- It measures the availability of both produce and consume requests on each broker
  - https://github.com/linkedin/kafka-monitor

Best Practices for Kafka Monitoring

- Understand which metrics to monitor
- Implement proactive monitoring
- Set up alerts and thresholds
- Monitor performance and throughput
- Monitor and manage failures
- Monitor resource utilization
- Utilize monitoring tools and frameworks
- Regularly review and optimize
- Continuous Monitoring
- Periodic Monitoring

Kafka Connect

Considerations When Building Data Pipelines

- Getting data from Kafka to S3 or getting data from MongoDB into Kafka.
- Getting data from Twitter to Elasticsearch by sending the data first from Twitter to Kafka and then from Kafka to Elasticsearch.
- Kafka acts as a giant buffer that decouples the time-sensitivity requirements between producers and consumer
  - Producers can write events in real-time while consumers process batches of events, or vice versa.
- Reliability
- High and Varying Throughput
- Data Formats
- Transformations
- Security

Considerations When Building Data Pipelines

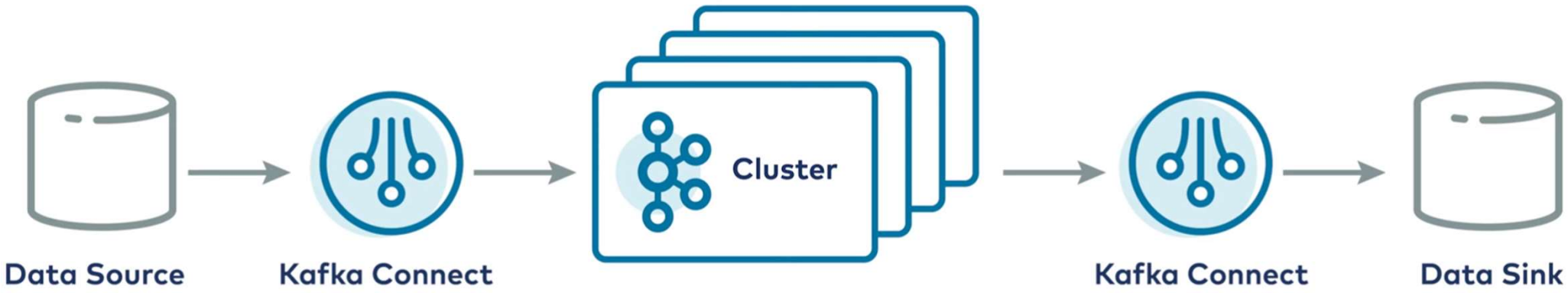
- Failure Handling
- Coupling

When to Use Kafka Connect

- Use Kafka clients when you can modify the code of the application
- Use Connect to
  - Connect Kafka to datastores whose code you cannot modify.
- Where a connector already exists, Connect can be used by nondevelopers, who will only need to configure the connectors.

Kafka Connect

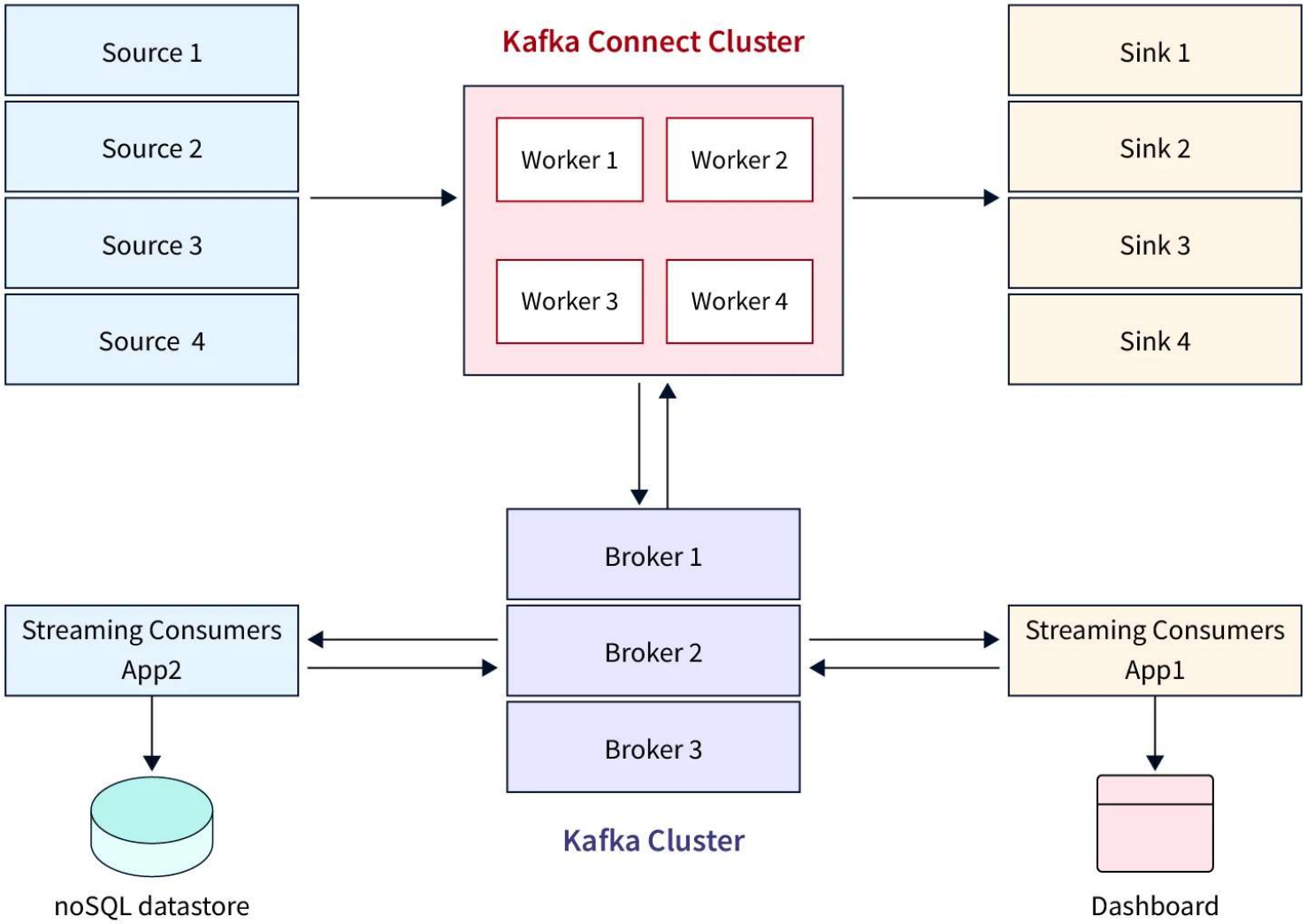
- A framework for connecting Kafka with external systems



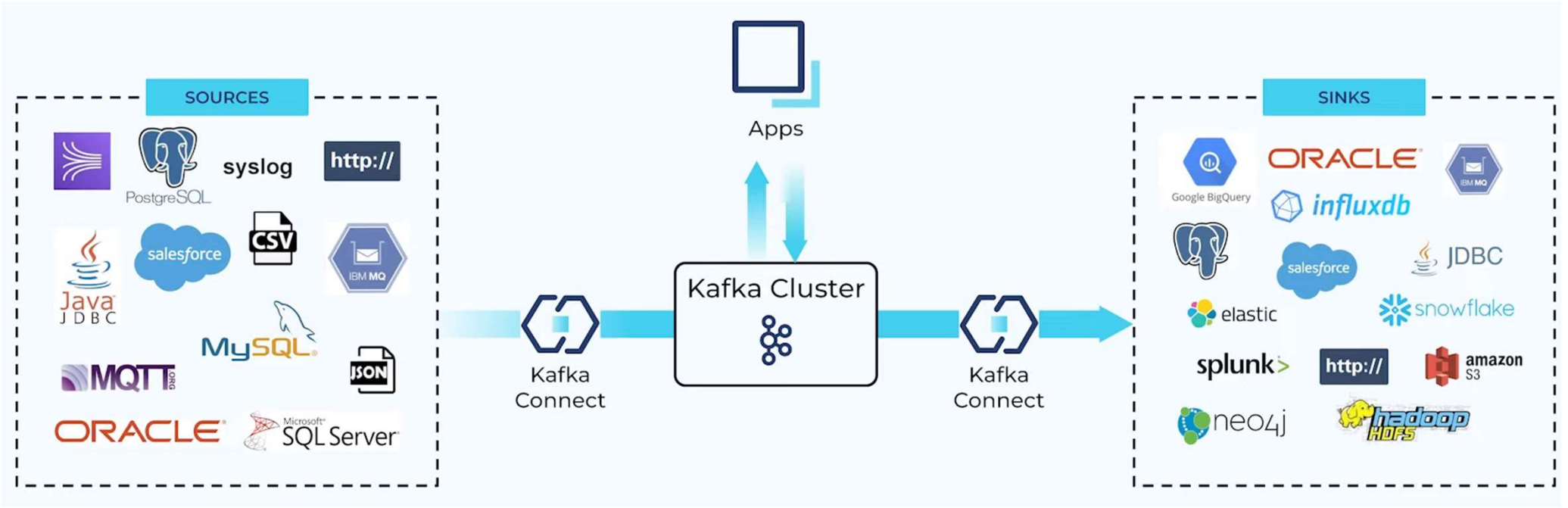
Kafka Connect

- Data integration system and ecosystem
- Because some other systems are not Kafka
- External client process; does not run on Brokers
- Horizontally scalable
- Fault tolerant
- Declarative

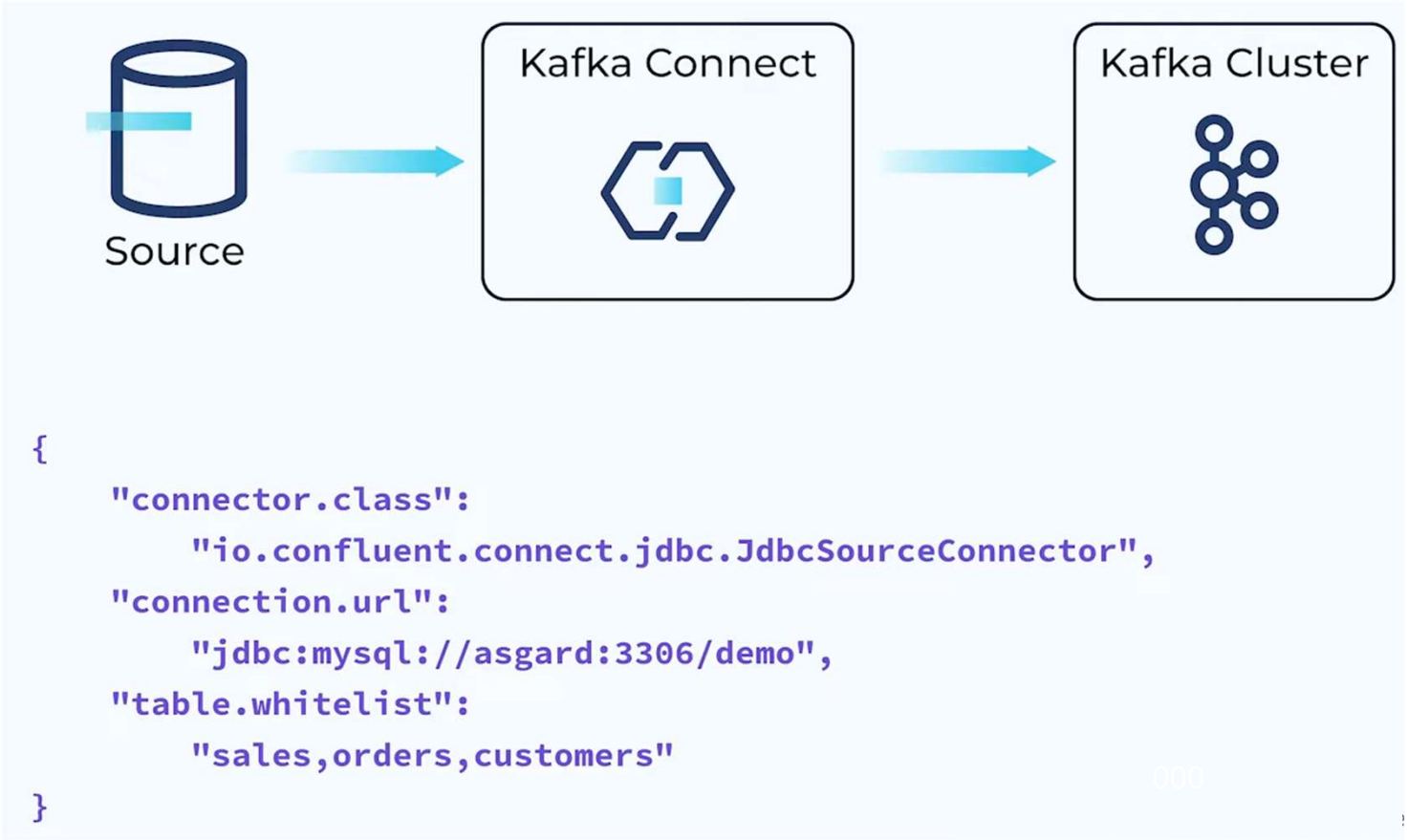
Kafka Connect



Ingest Data from Upstream Systems

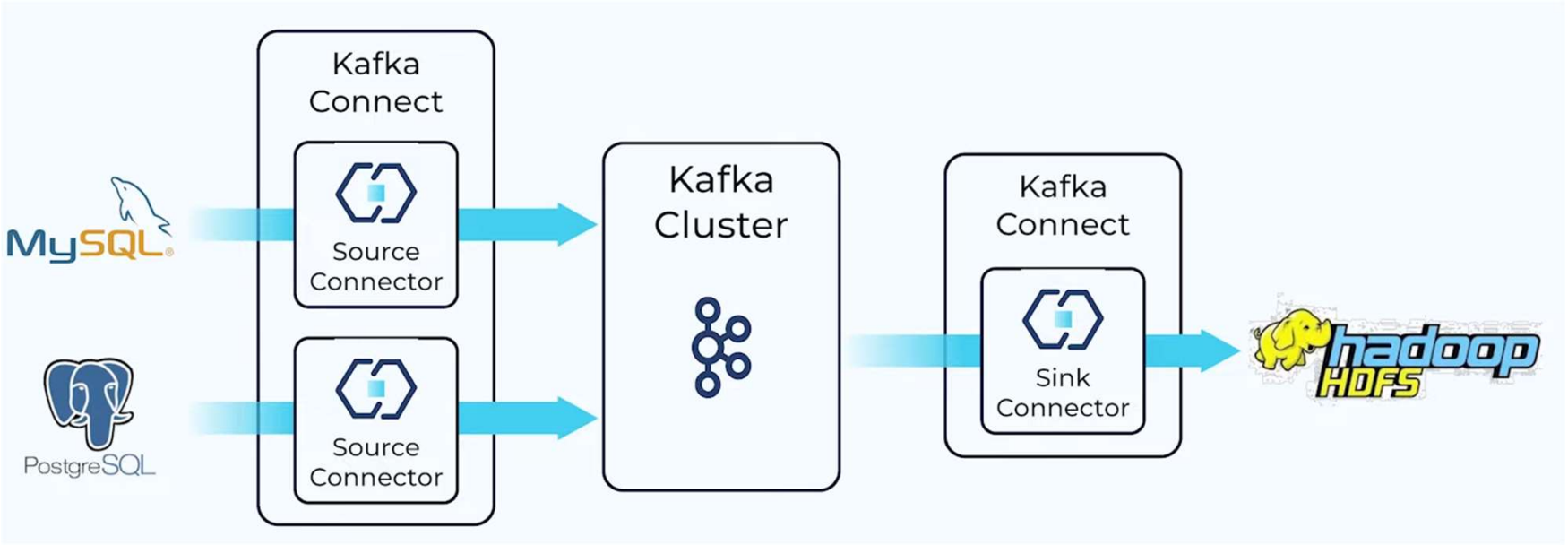


How Kafka Connect Works



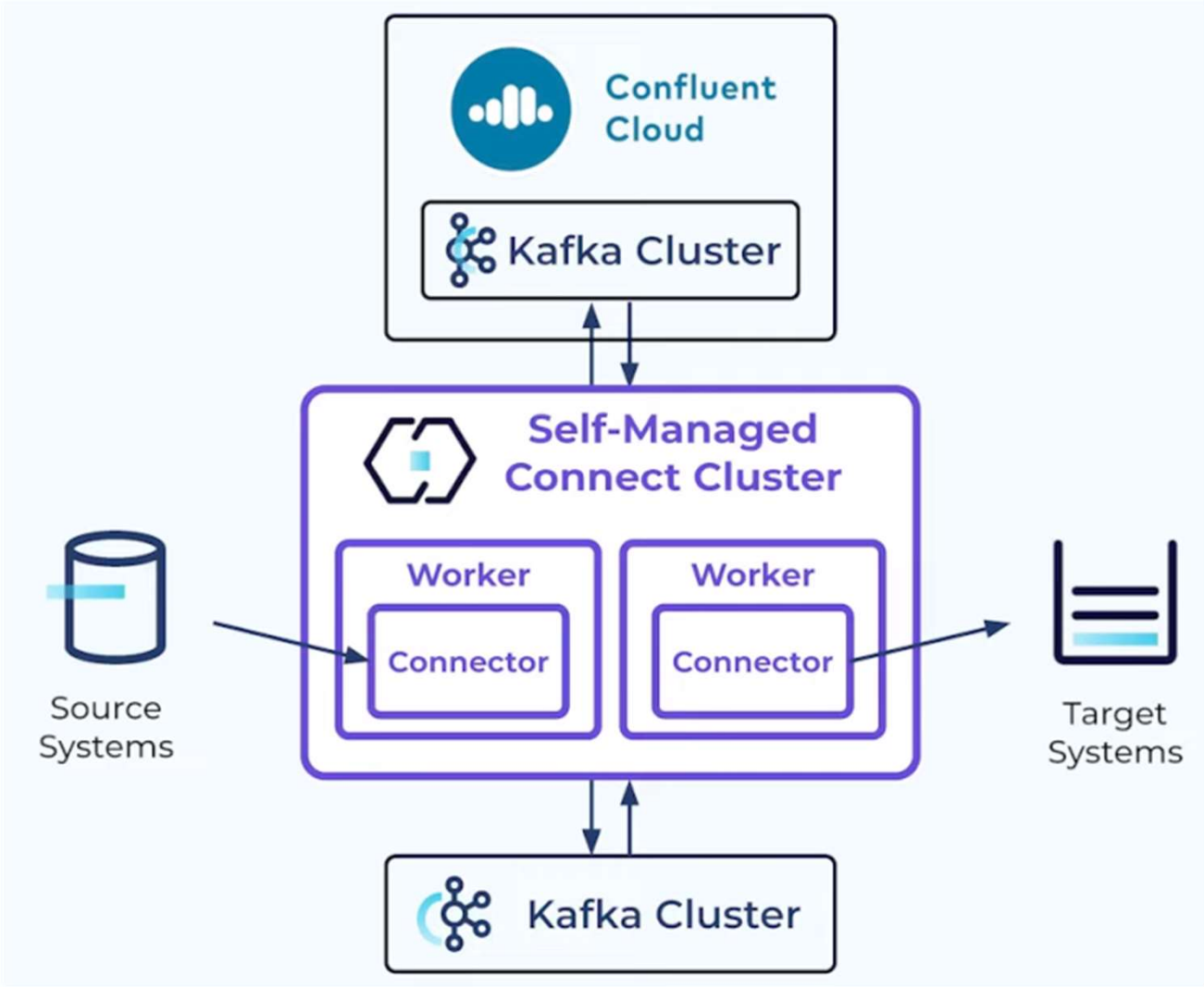

```
{
    "connector.class":
        "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url":
        "jdbc:mysql://asgard:3306/demo",
    "table.whitelist":
        "sales,orders,customers"
}
```

Kafka Connectors

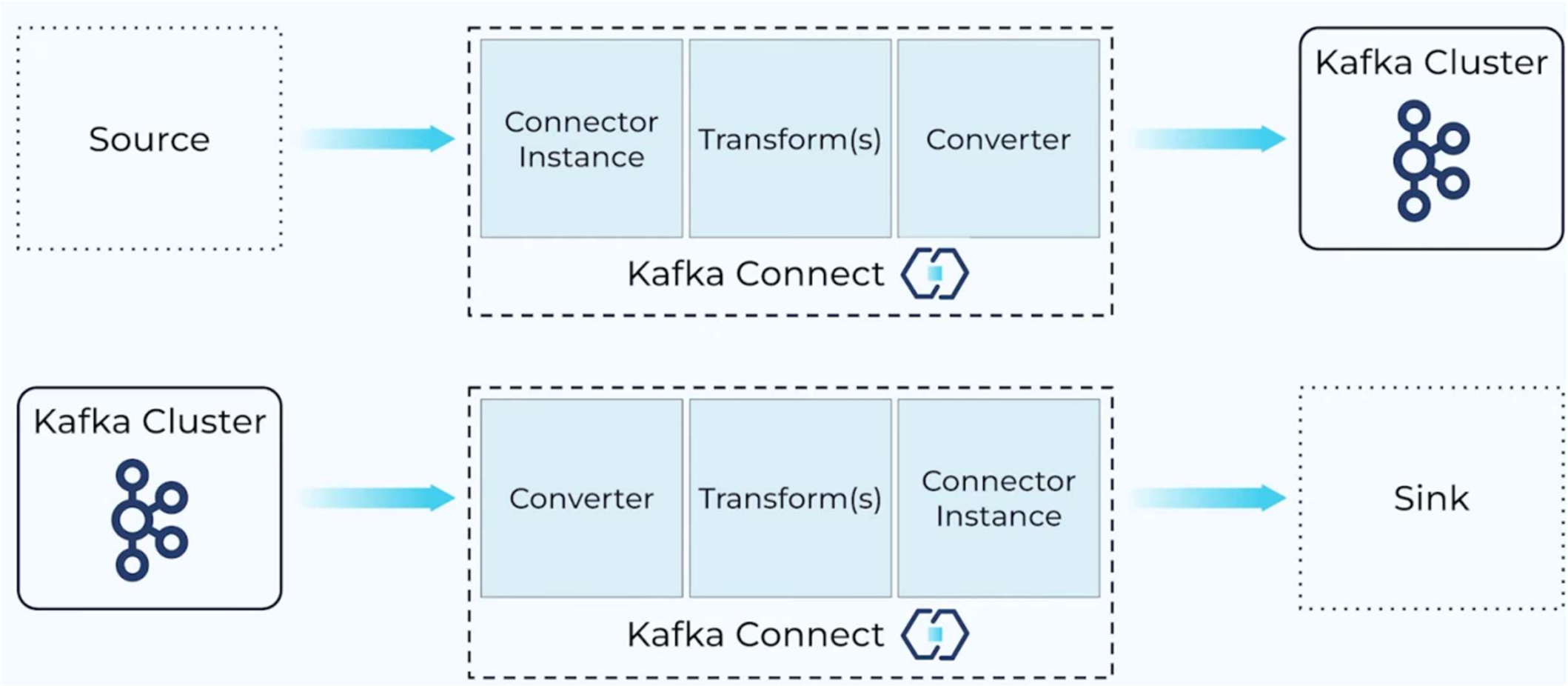


Self-managed Kafka connect

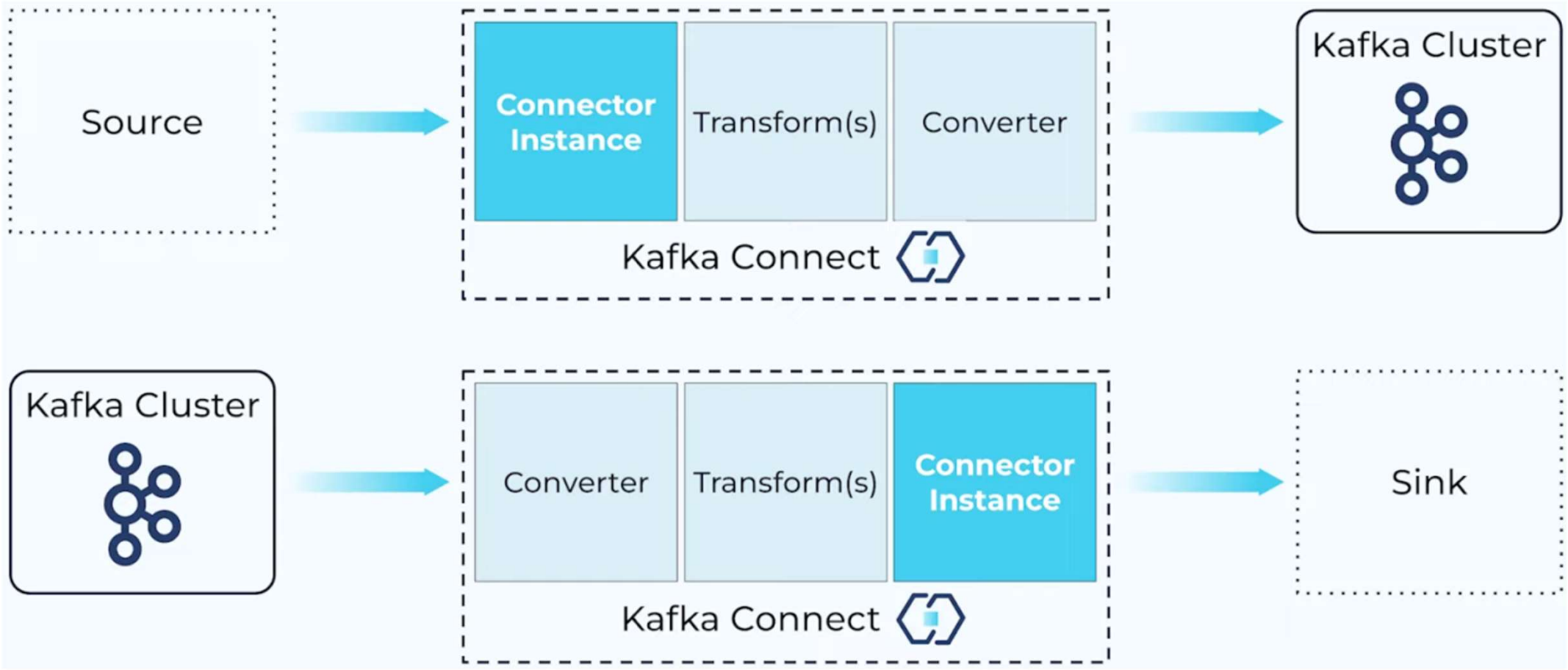
- Kafka connect workers are JVM processes



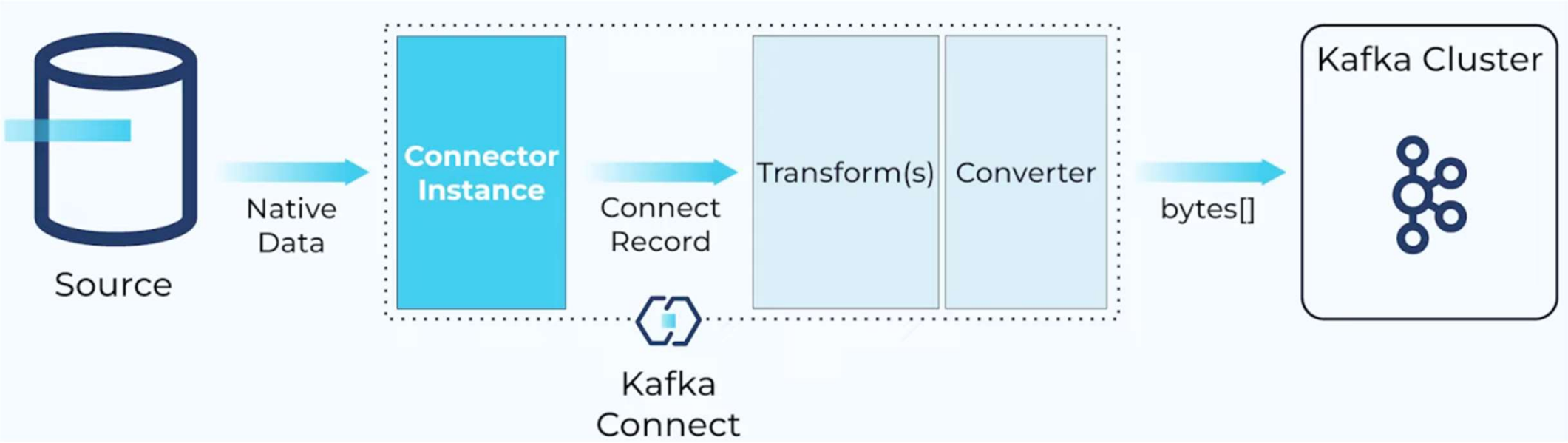
Inside Kafka Connect



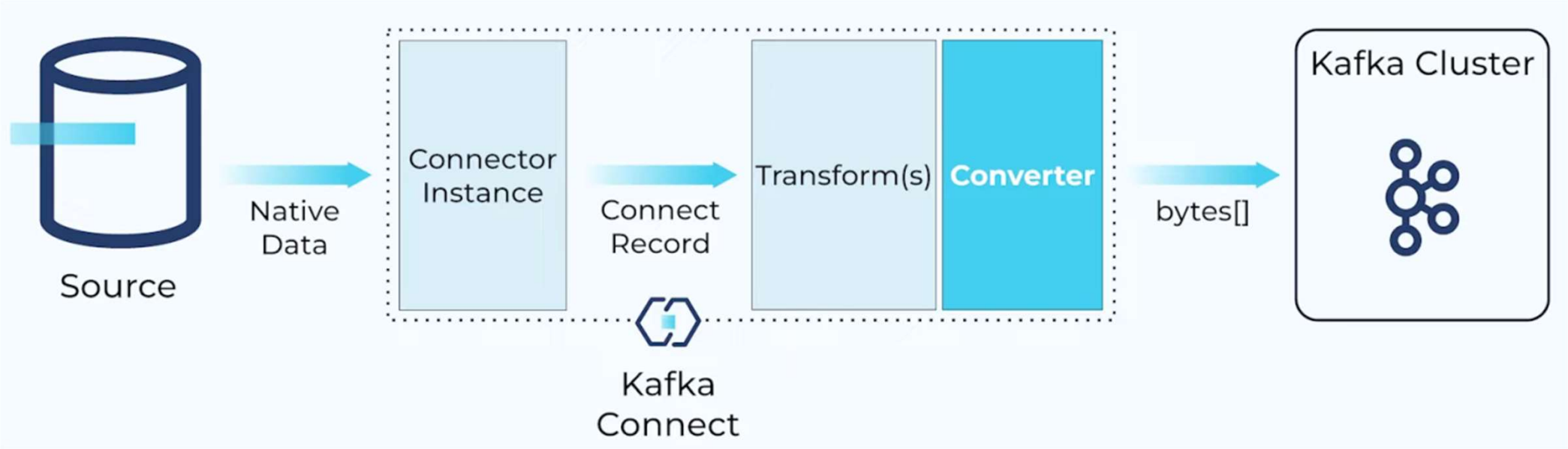
Connectors


Source
Connector Instance
Transform(s)
Converter
Kafka Connect
Kafka Cluster
Kafka Cluster
Converter
Transform(s)
Connector Instance
Kafka Connect
Sink

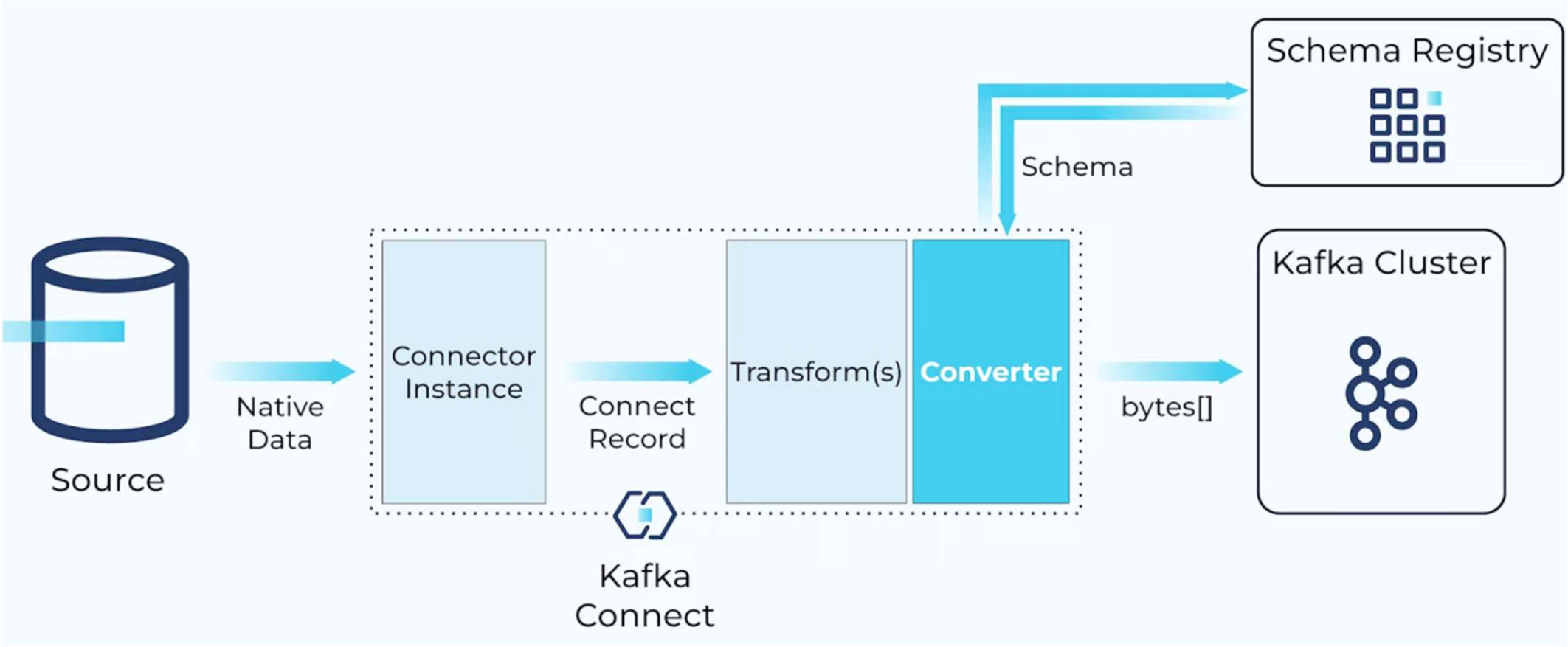
What is the role of a Connector?



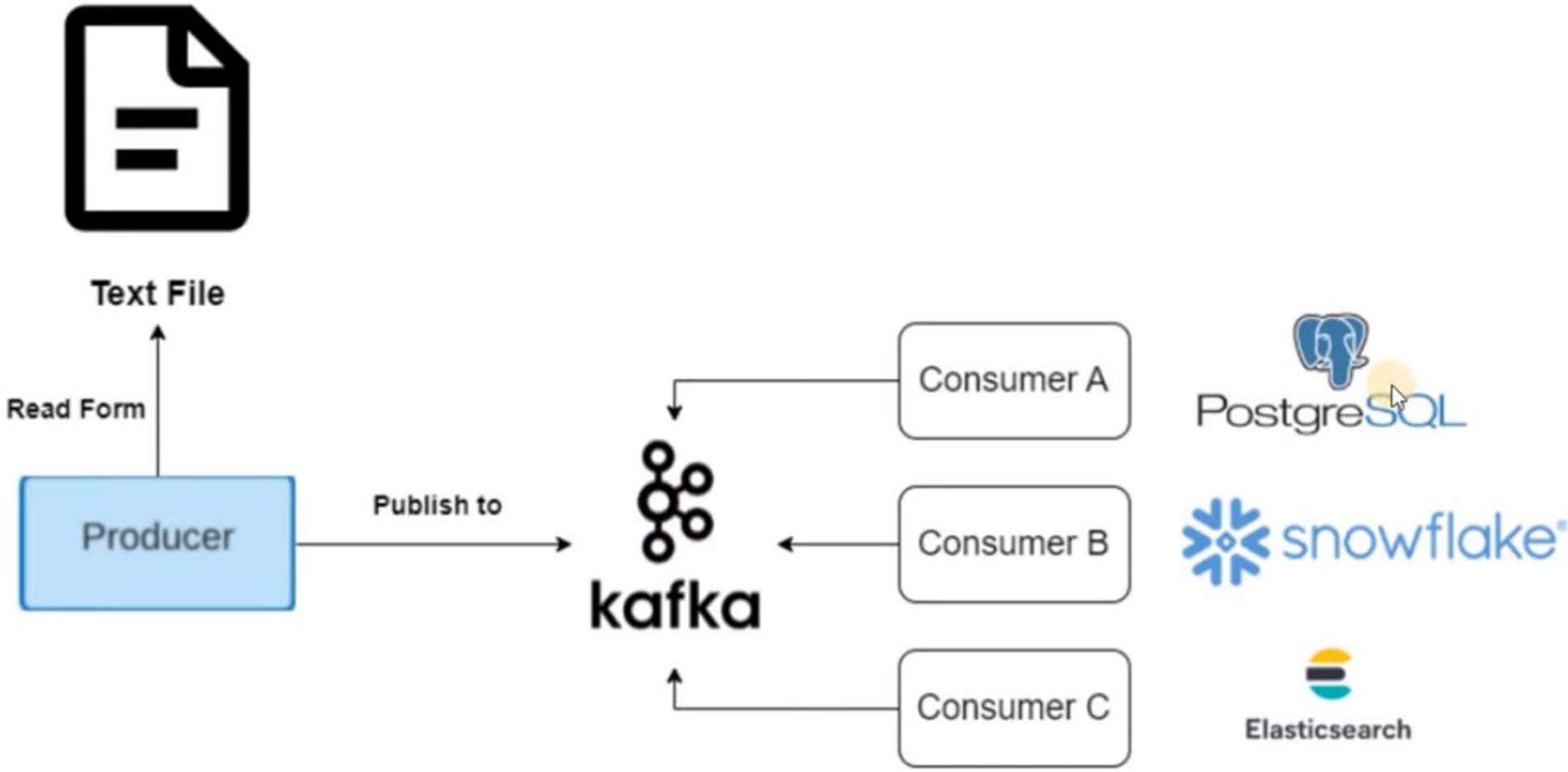
Converters Serialize/Deserialize data



Serialization and Schemas



Without Connect

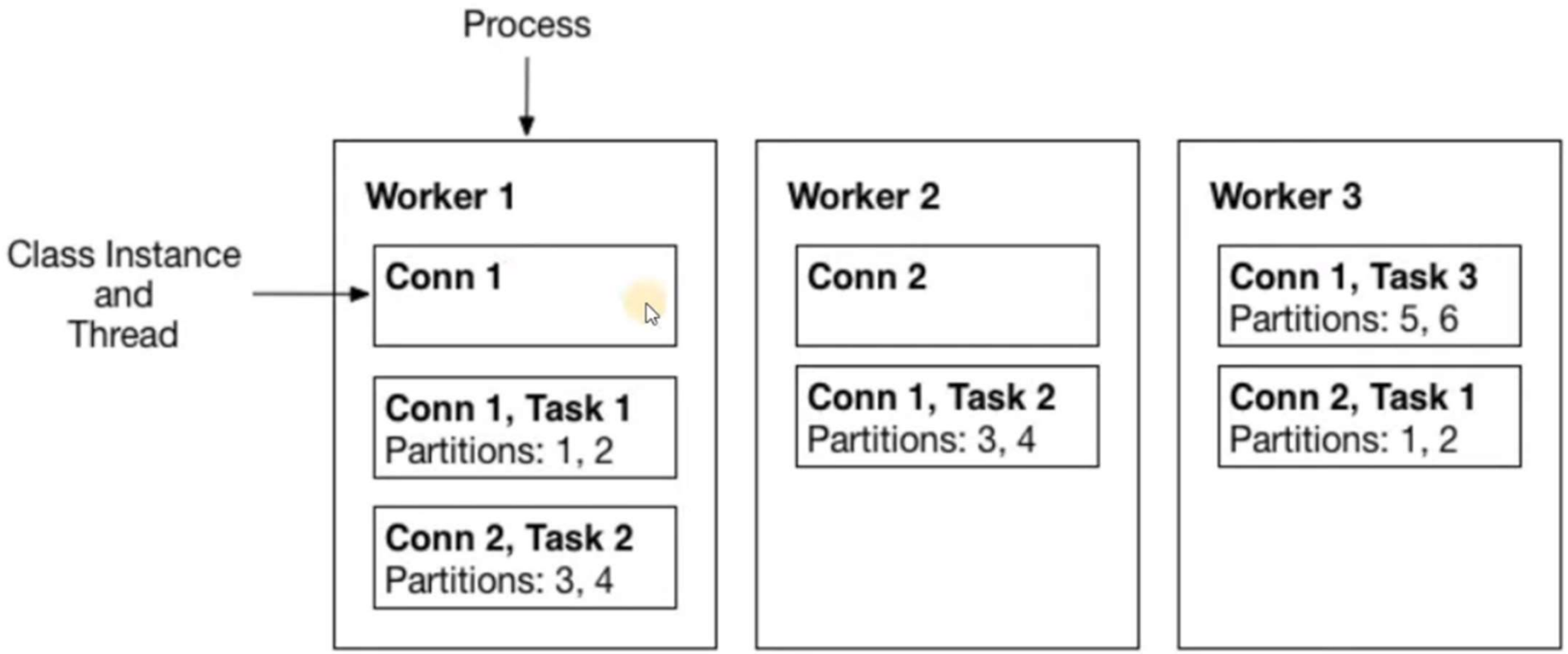


Major concepts

- **Connector**: is a job that manages and coordinates the tasks. It decides how to split the data-copying work between the tasks.
- **Task**: is piece of work that provides service to accomplish actual job.
- Connectors divide the actual job into smaller pieces as tasks in order to have parallelism and scalable data copying with very little configuration.

Worker: is the node that is running the connector and its tasks.

How Connect works?

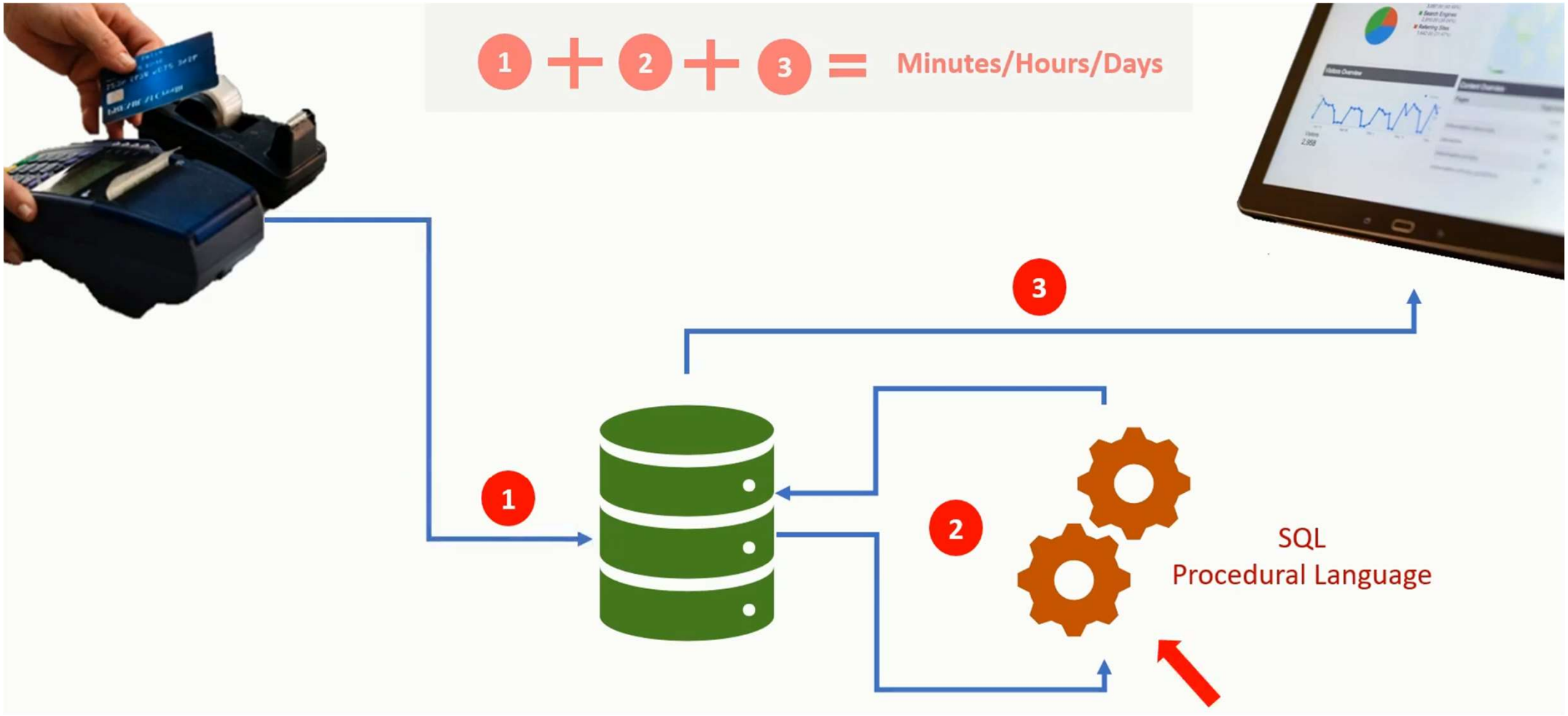


Working few famous connectors

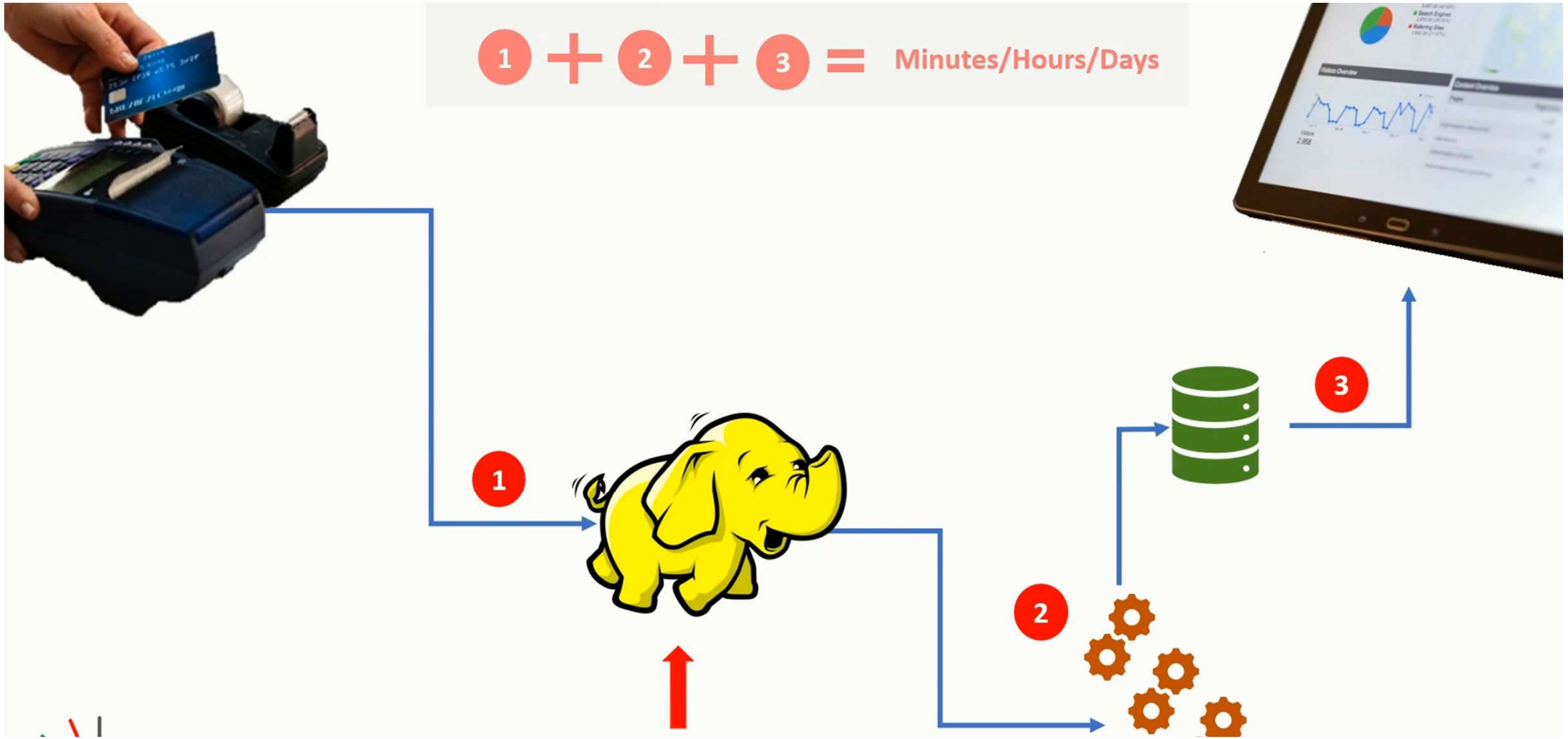
- JDBC Source and Sink Connector
  - Enables a Java application to interact with a Database
- Google BigQuery Sink Connector
  - To stream data into BigQuery Tables
- JMS Source Connector
  - For moving messages from any JMS-compliant broker into a Kafka Topic
- Elasticsearch Service Sink Connector
  - For moving data from a Kafka to Elasticsearch
- Amazon S3 Sink Connector
  - Exports data from Kafka Topics to Amazon S3
- HDFS 2 Sink Connector
  - For exporting data from any Kafka Topic to HDFS 2.x files in a variety of formats
- Replicator
  - Replicate Topics from one Kafka Cluster to another

Kafka Stream Processing

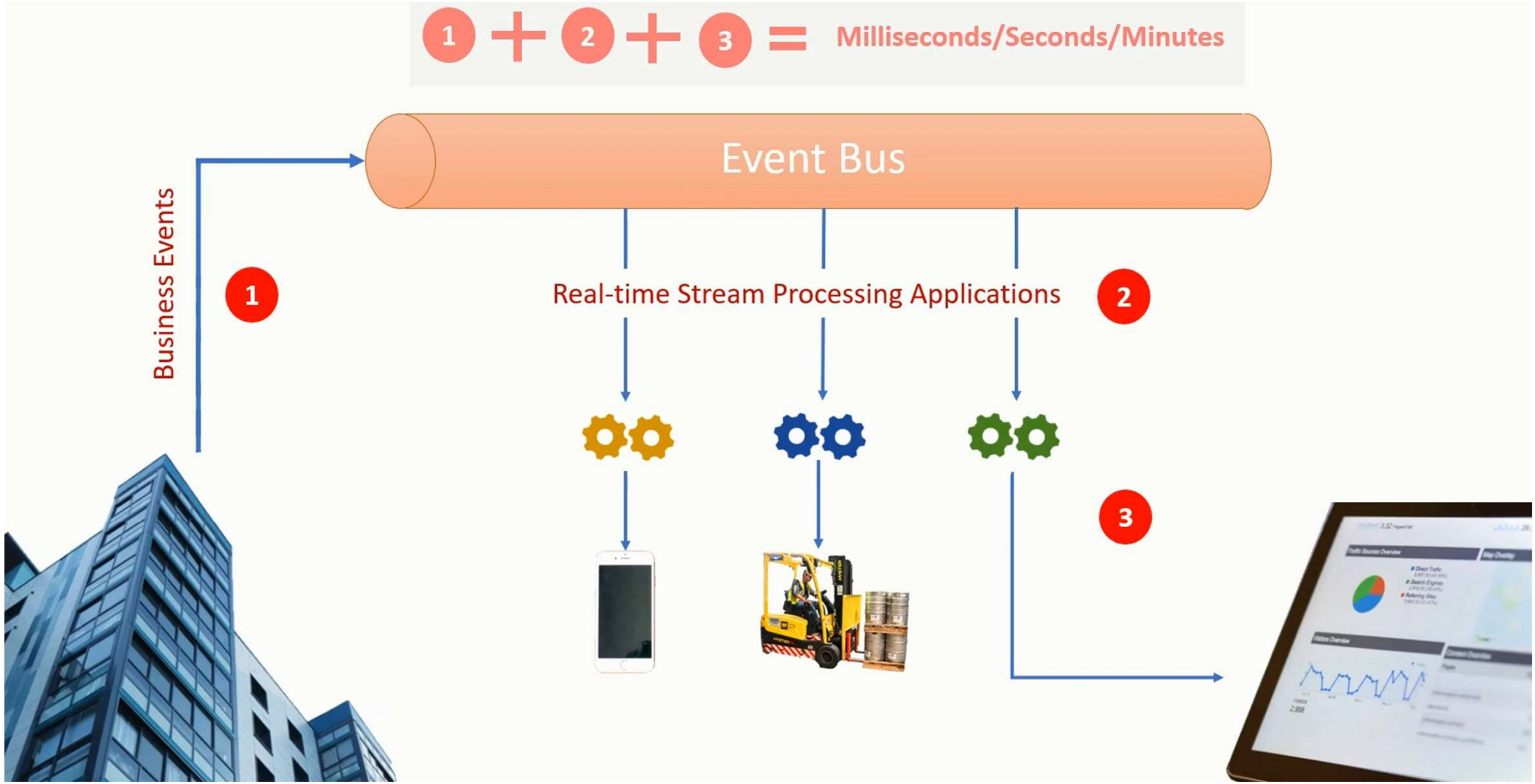
Traditional Data Processing



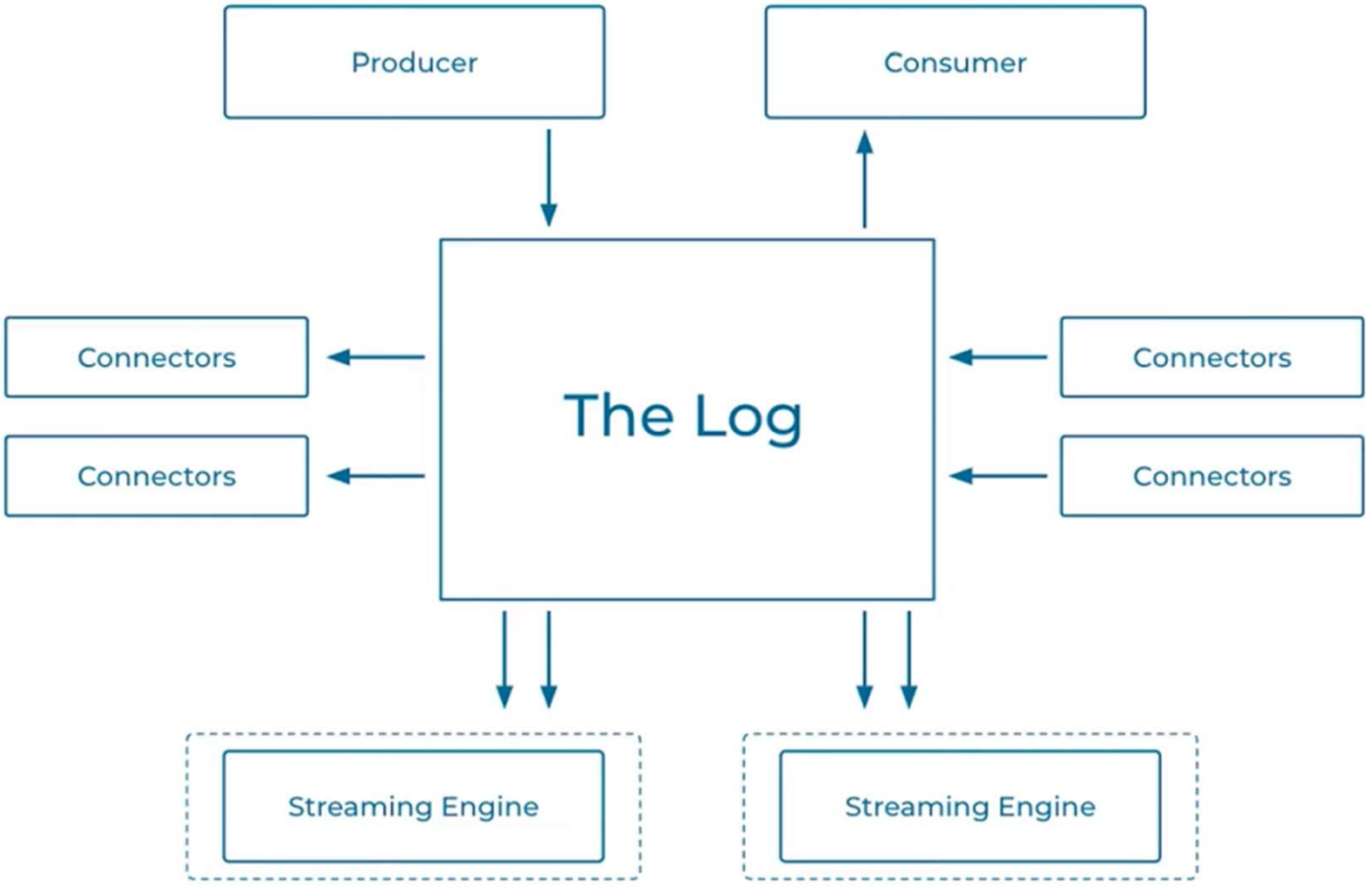
Hadoop for Bigdata processing



Stream Processing



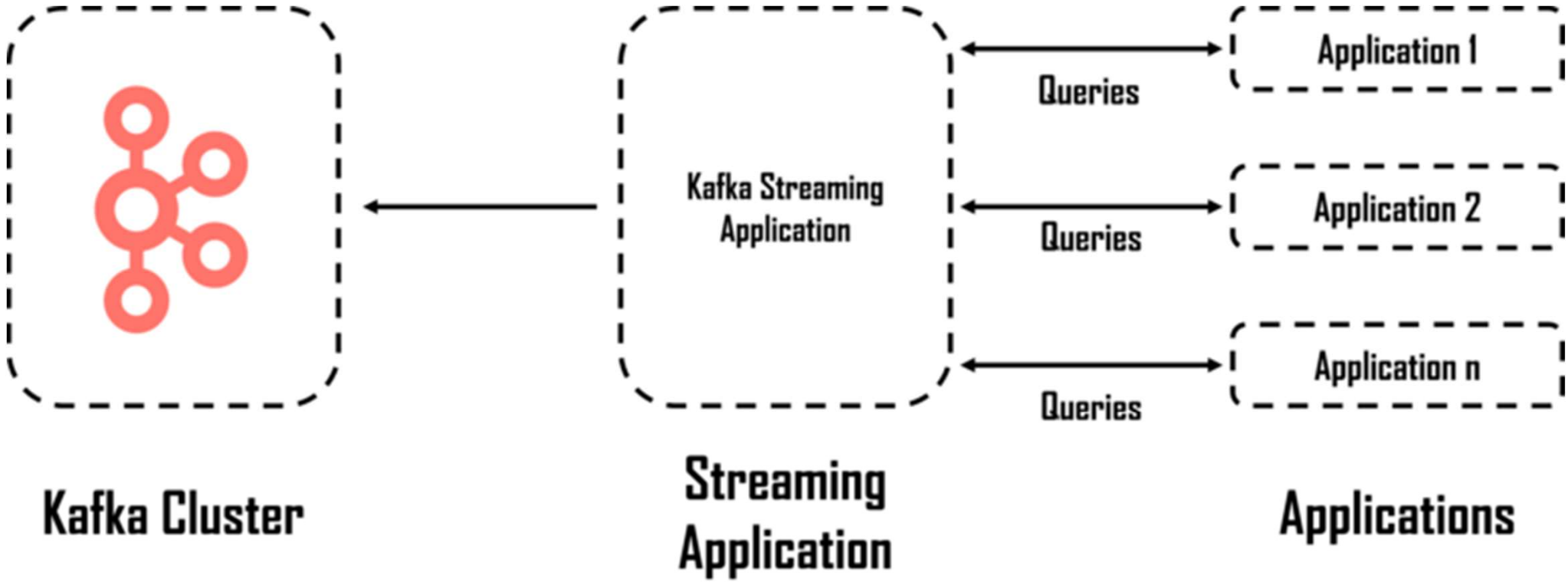
Kafka


Producer
Consumer
Connectors
Connectors
The Log
Connectors
Connectors
Streaming Engine
Streaming Engine

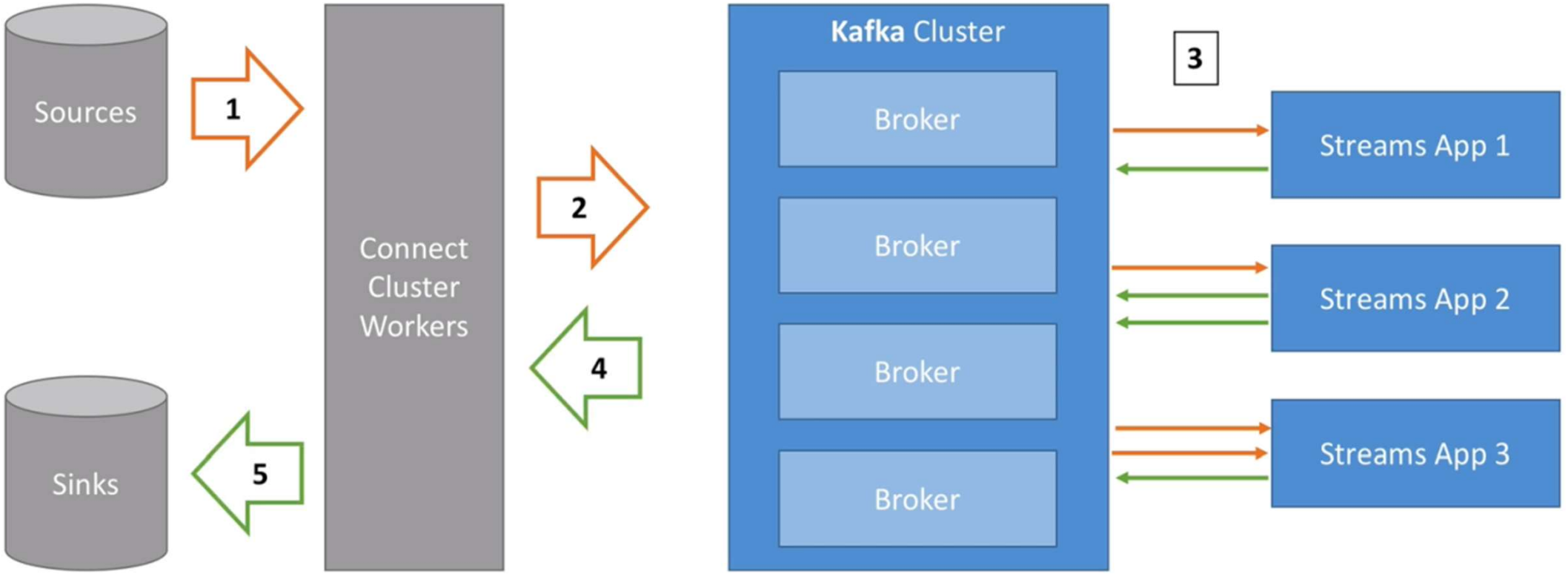
Kafka Streams

- Functional Java API
- Filtering, grouping, aggregating, joining, and more
- Scalable, fault-tolerant state management

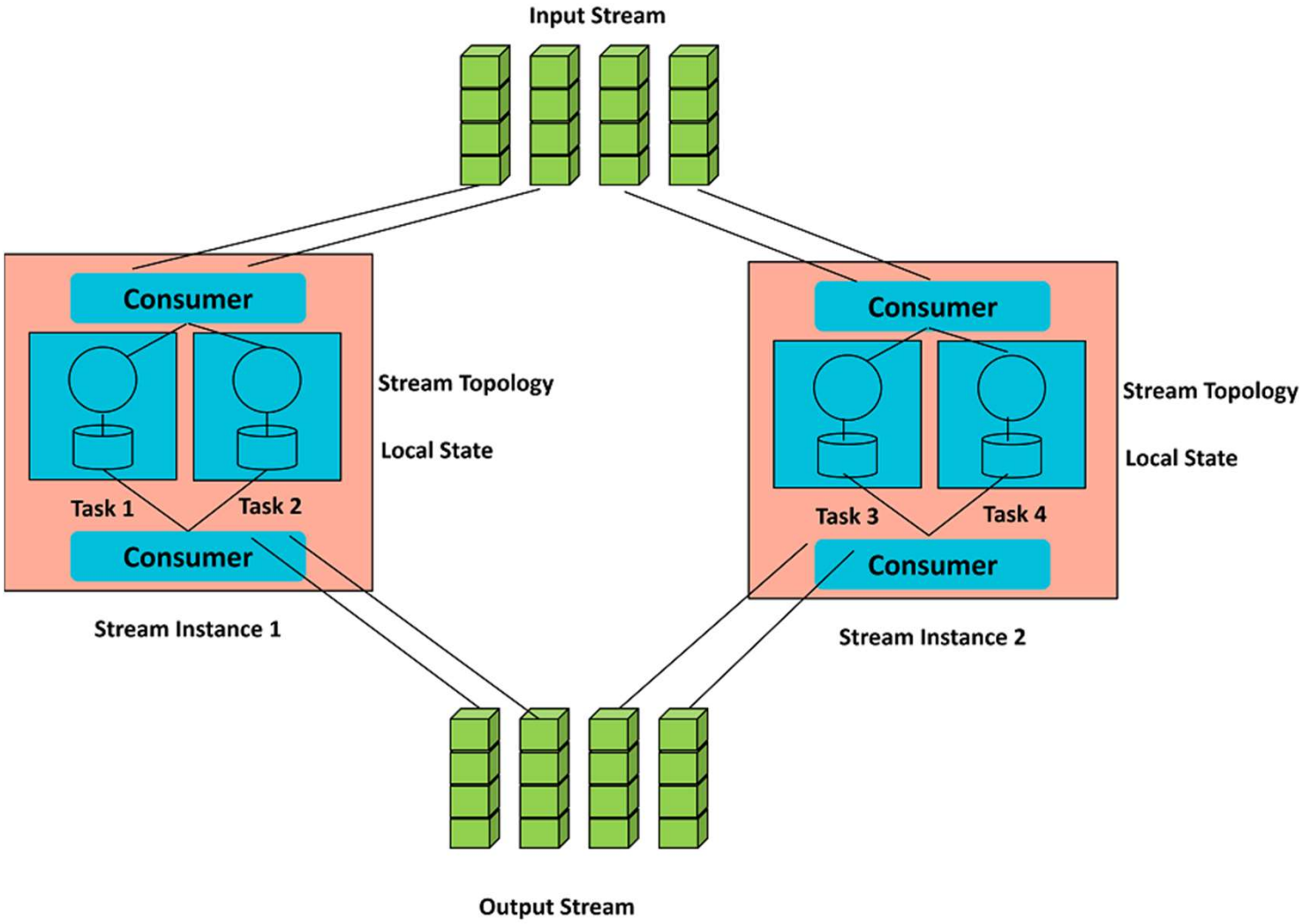
Kafka Stream



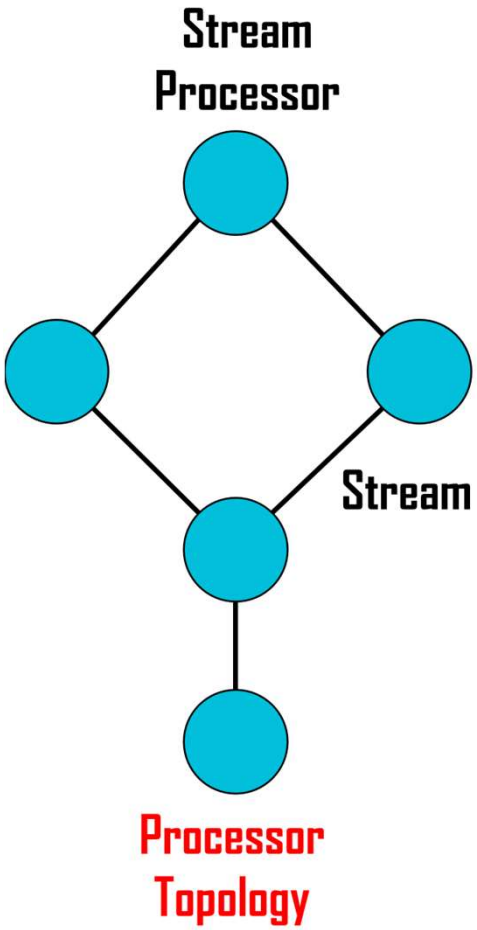
Kafka Stream Architecture Design



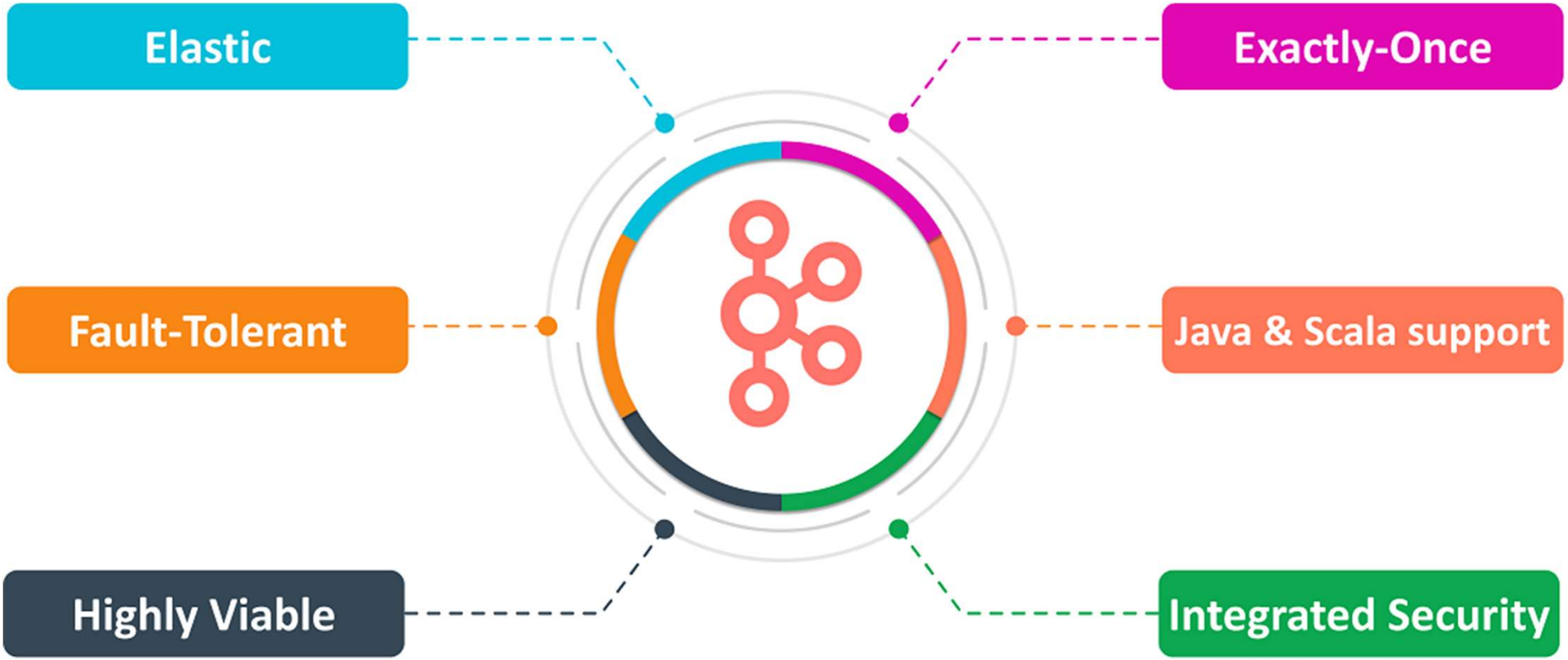
Kafka Stream



Stream Topology



Kafka Stream Features



Use cases of Apache Kafka Streams API

- **The New York Times** store and distribute the real-time news through various applications and systems to their readers.

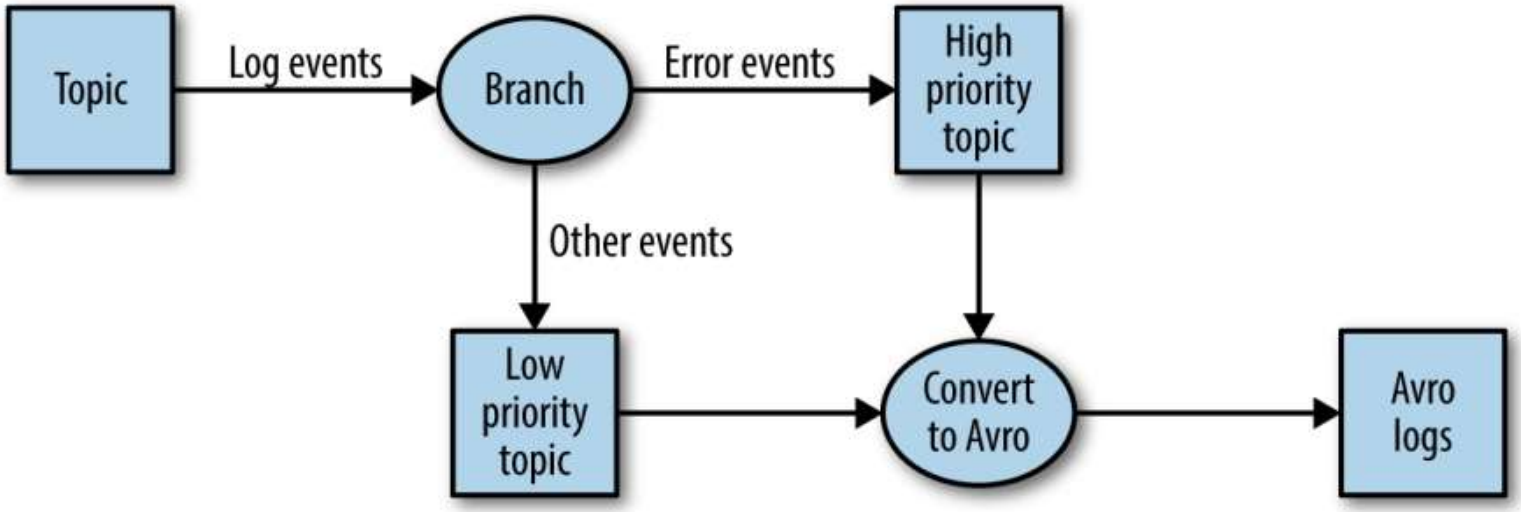
- **Trivago** use Kafka, Kafka Connect and Kafka Streams to enable their developers to access details of various hotels

- **Pinterest** uses Kafka to power the real-time predictive budgeting system of their advertising system

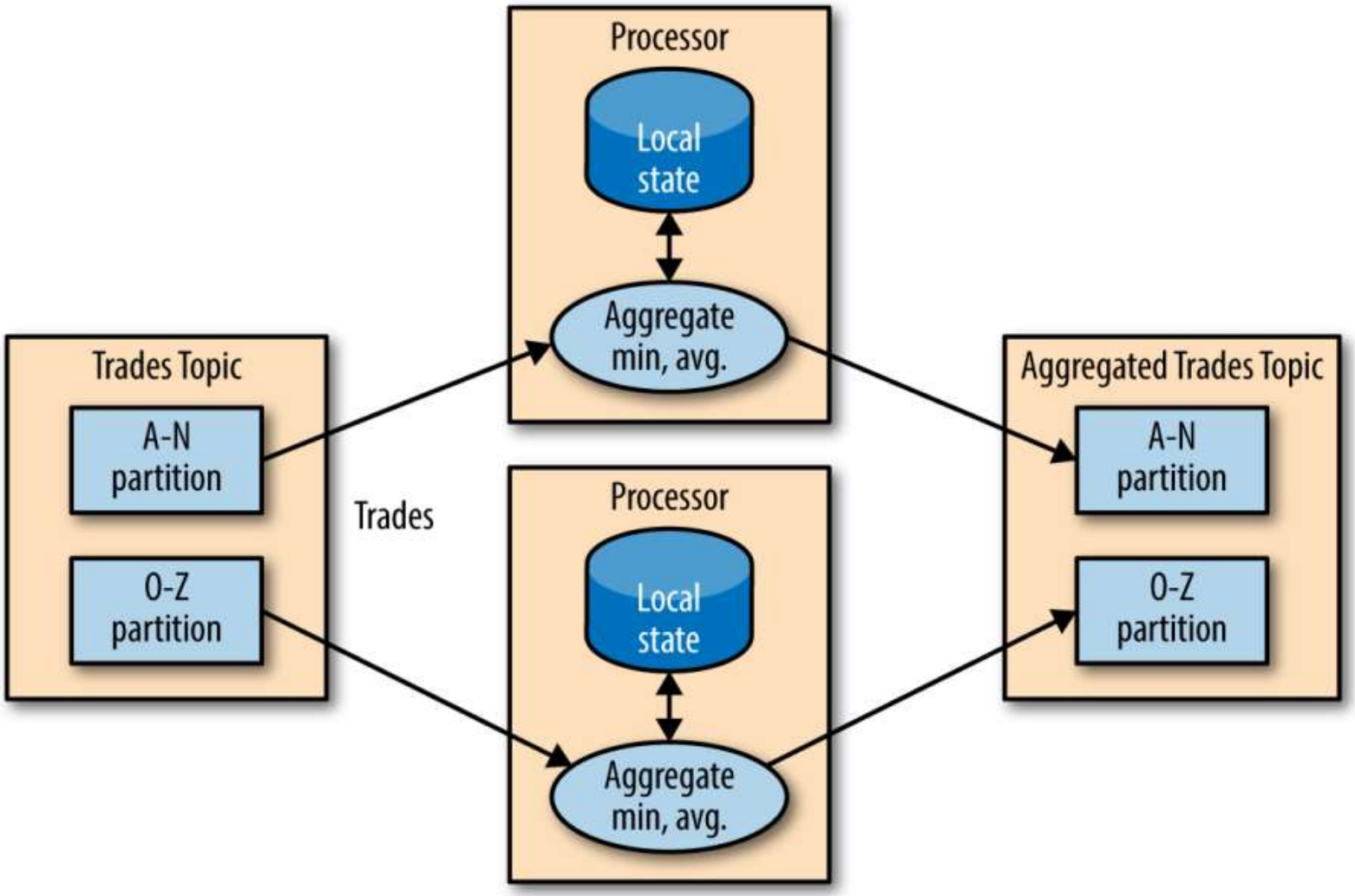
Stream-Processing Design Patterns

- Single-Event Processing
- Processing with Local State
- Multiphase Processing/Repartitioning
- Processing with External Lookup: Stream-Table Join
- Streaming Join

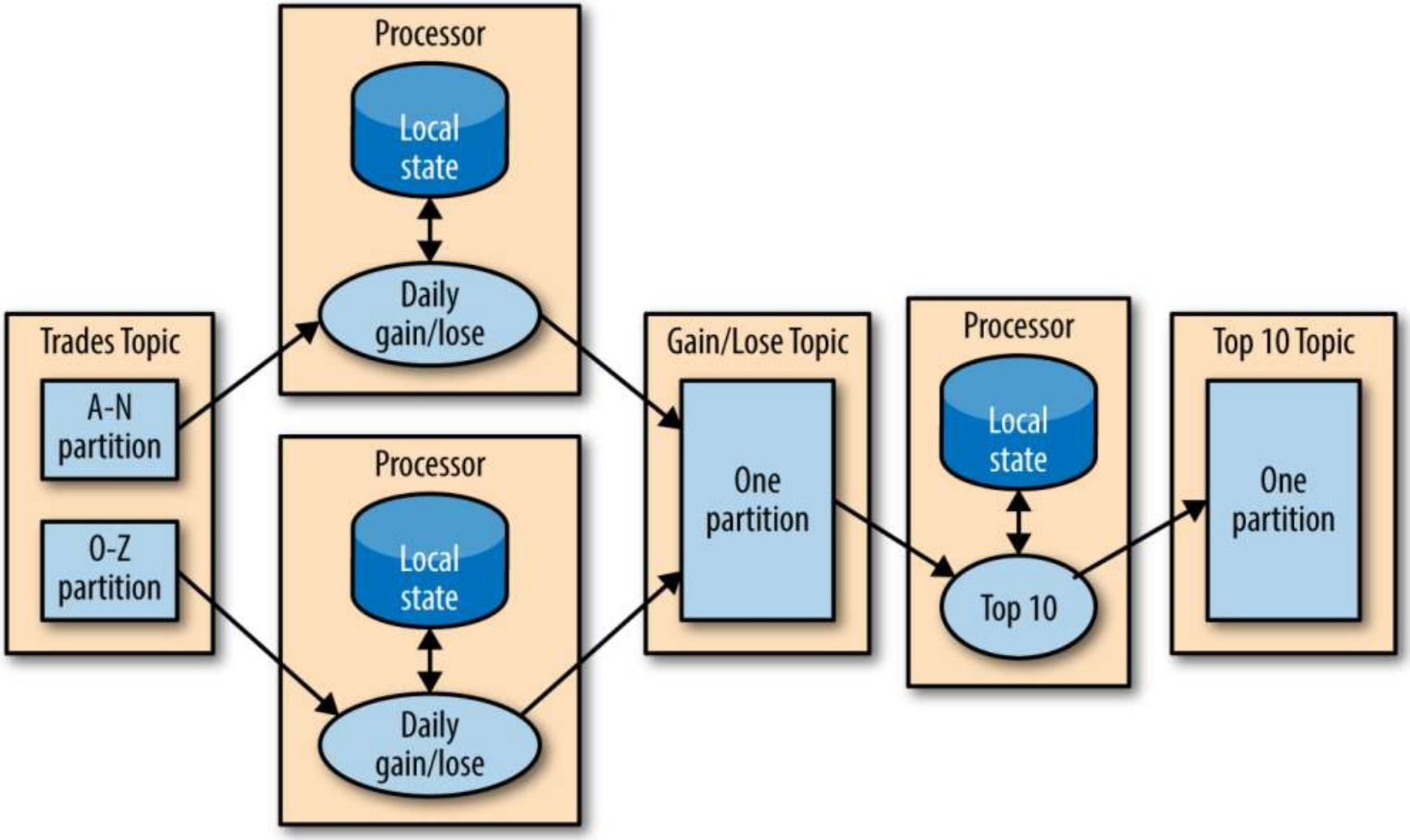
Single-Event Processing



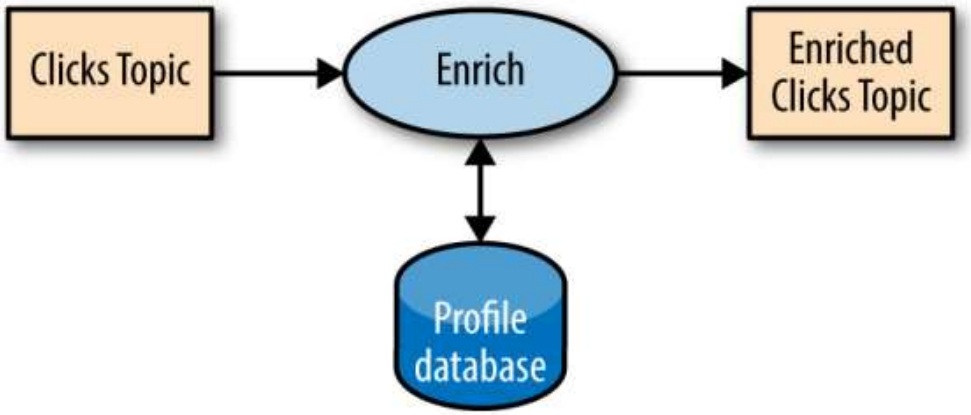
Processing with Local State



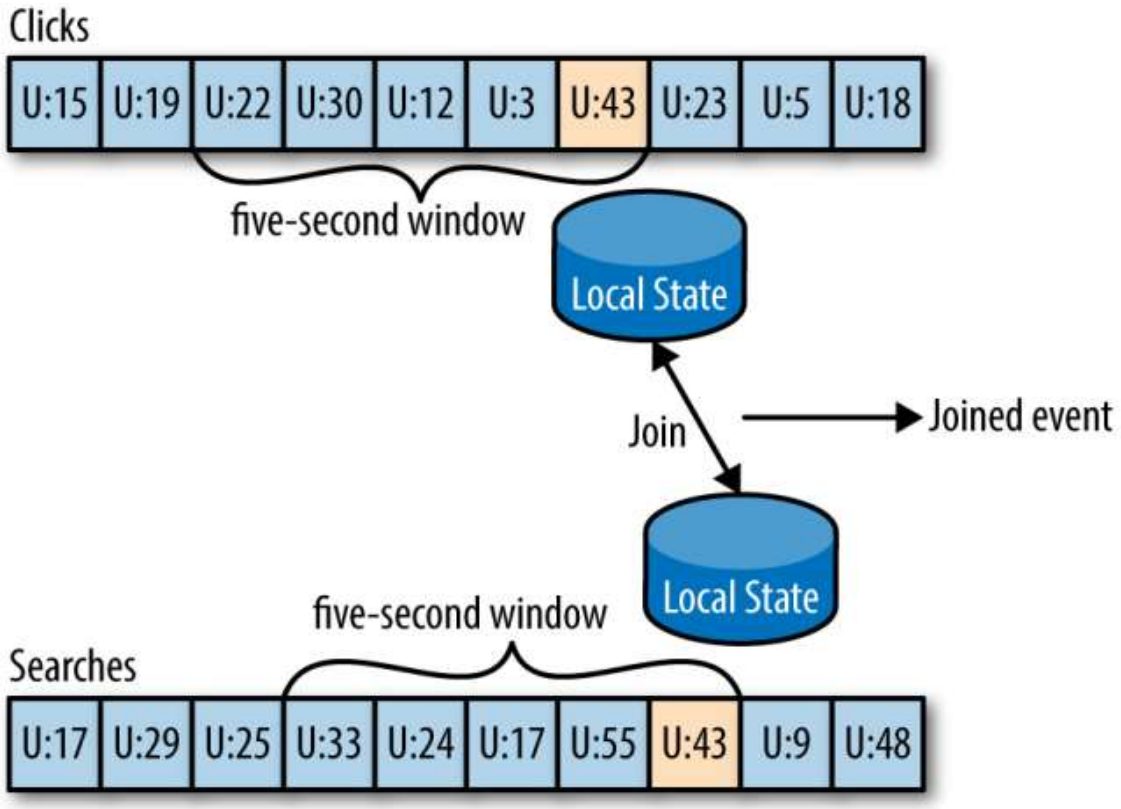
Multiphase Processing/Repartitioning



Processing with External Lookup: Stream-Table Join



Streaming Join



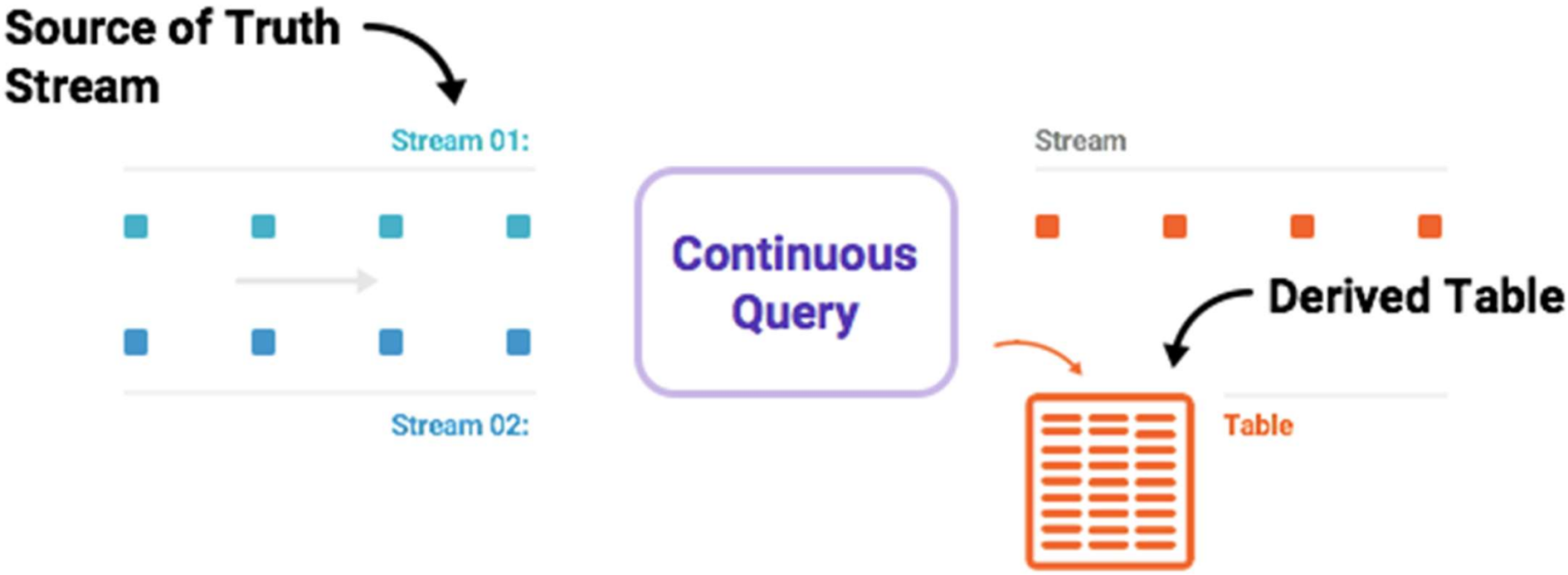
Kafka Structured Streaming

Confluent – KSQL

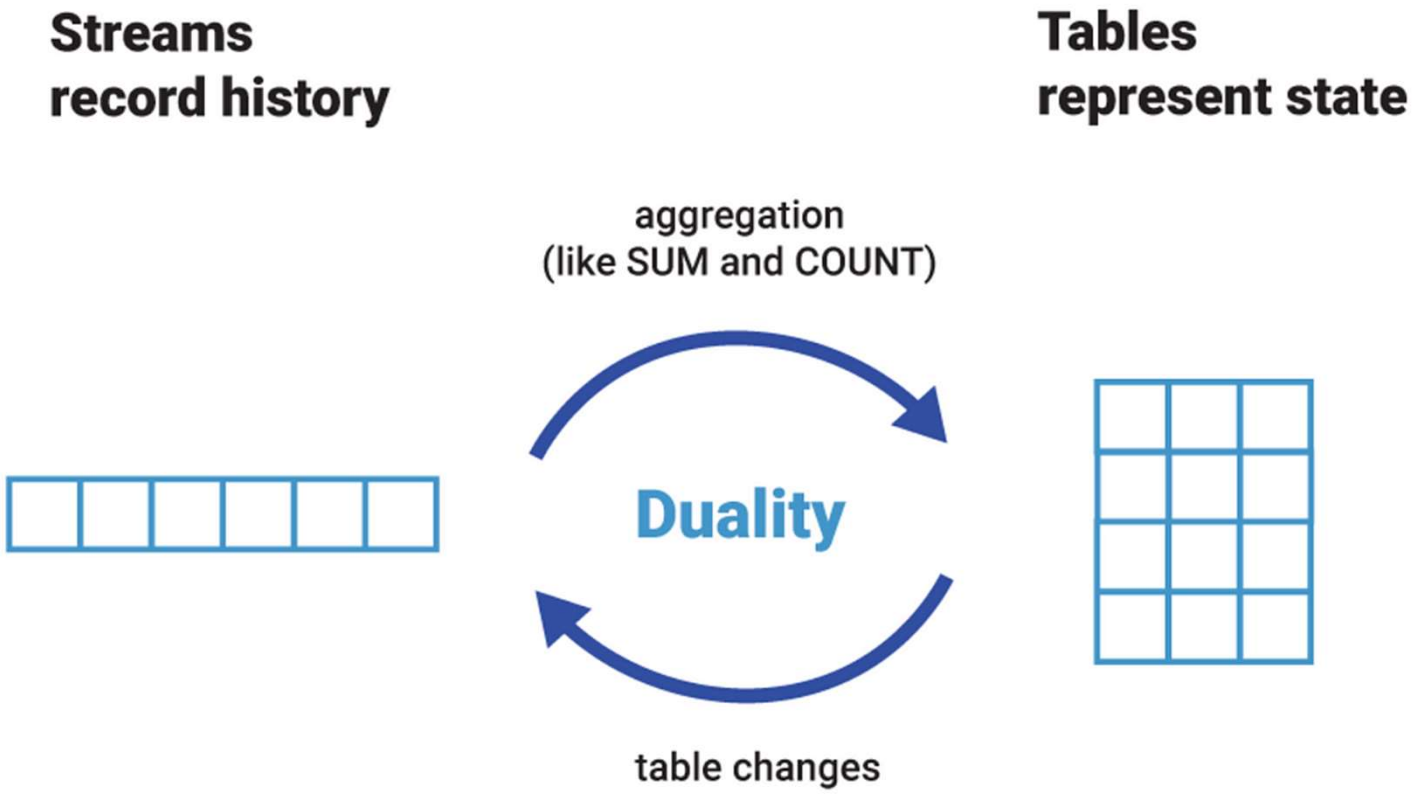
- Streaming SQL for Apache Kafka
- Provides a simple and completely interactive SQL interface for processing data in Kafka
- No longer need to write code
- Supports aggregations, joins, windowing, sessionization, and much more

- Example:
  - CREATE TABLE error_counts AS
  - SELECT error_code, count(*)FROM monitoring_stream
  - WINDOW TUMBLING (SIZE 1 MINUTE)
  - WHERE type = 'ERROR'

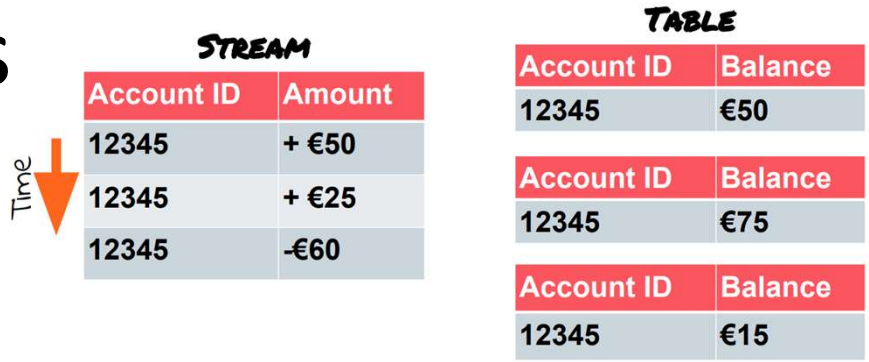
Confluent – KSQL



Structured Streaming – Kstreams and KTable



Kstreams and Ktable Use Cases



- Kstream
  - Commonly used for real-time analytics
  - Event-driven architectures
  - Stream transformations
  - Ideal to process an unbounded stream of records and continuously update the stream
- Ktable
  - Commonly used for building materialized views
  - Performing table lookups
  - Maintaining state
  - Performing joins with other KTables or Kstreams
  - Suitable for a snapshot view of the stream’s latest values for each key

KSQL Joins

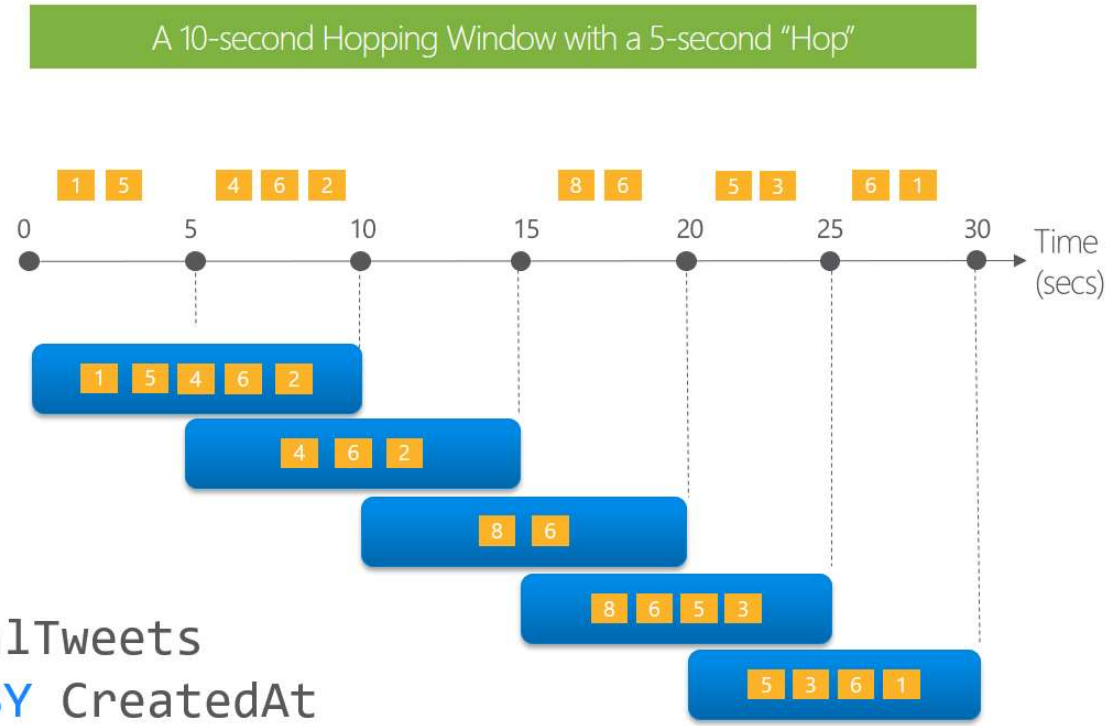
```
ksql> CREATE STREAM RATINGS_WITH_CUSTOMER_DATA WITH (PARTITIONS=1) AS \
SELECT R.RATING_ID, R.CHANNEL, R.STARS, R.MESSAGE, \
    C.ID, C.CLUB_STATUS, C.EMAIL, \
    C.FIRST_NAME, C.LAST_NAME \
FROM RATINGS R \
    INNER JOIN CUSTOMERS C \
      ON R.USER_ID = C.ID;
```

Window Operations

- Hopping Window
- Tumbling Window
- Session Window

Window Operations - Hopping Window

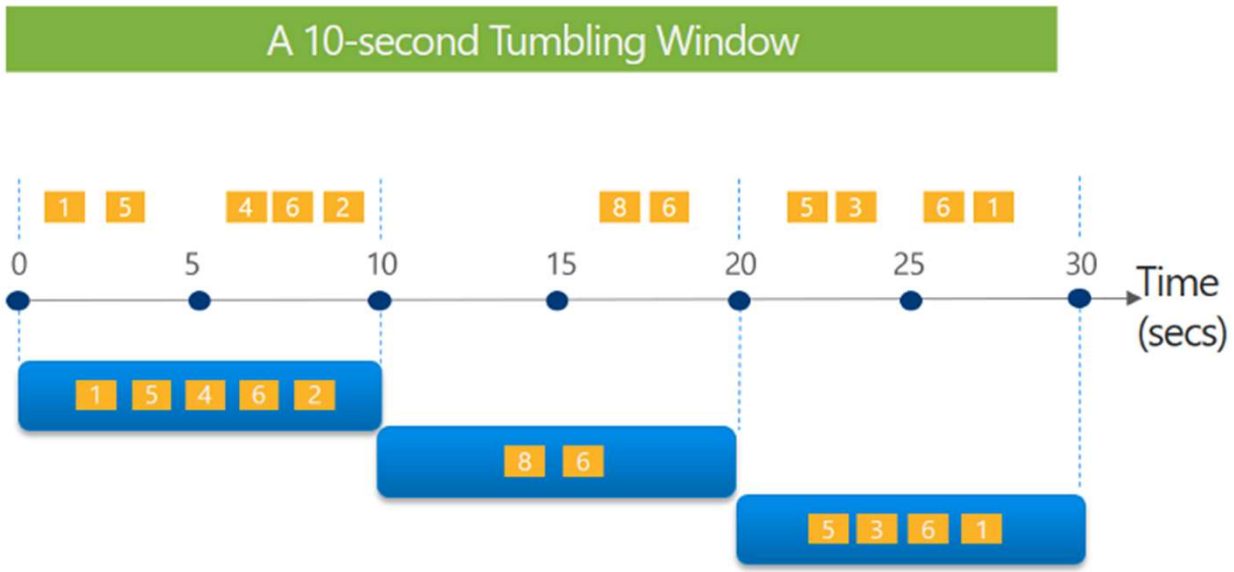
Every 5 seconds give me the count of tweets over the last 10 seconds




```
SELECT Topic, COUNT(*) AS TotalTweets
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, HoppingWindow(second, 10 , 5)
```

Window Operations - Tumbling Window

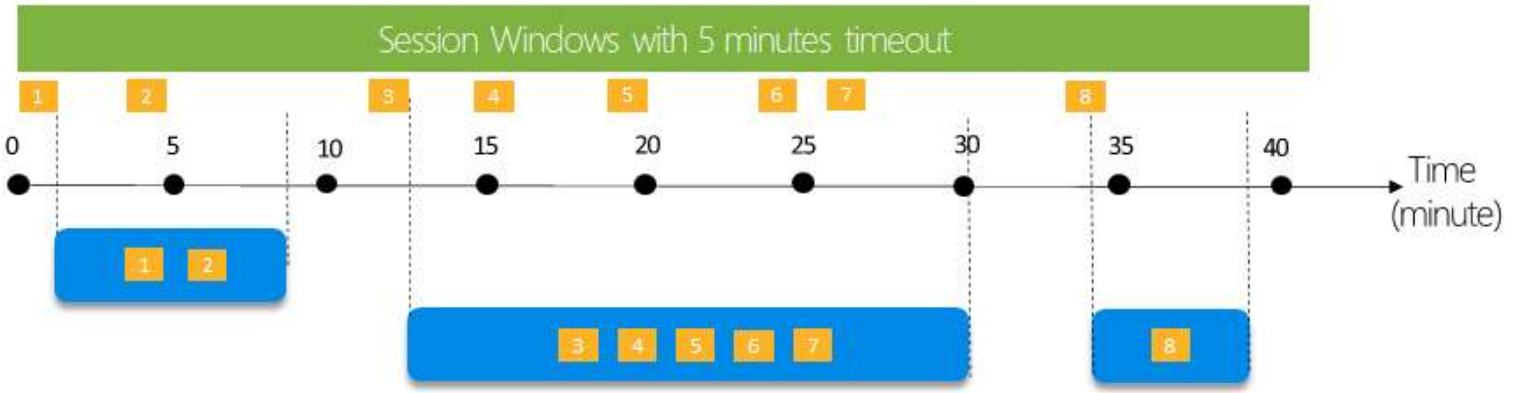
Tell me the count of tweets per time zone every 10 seconds



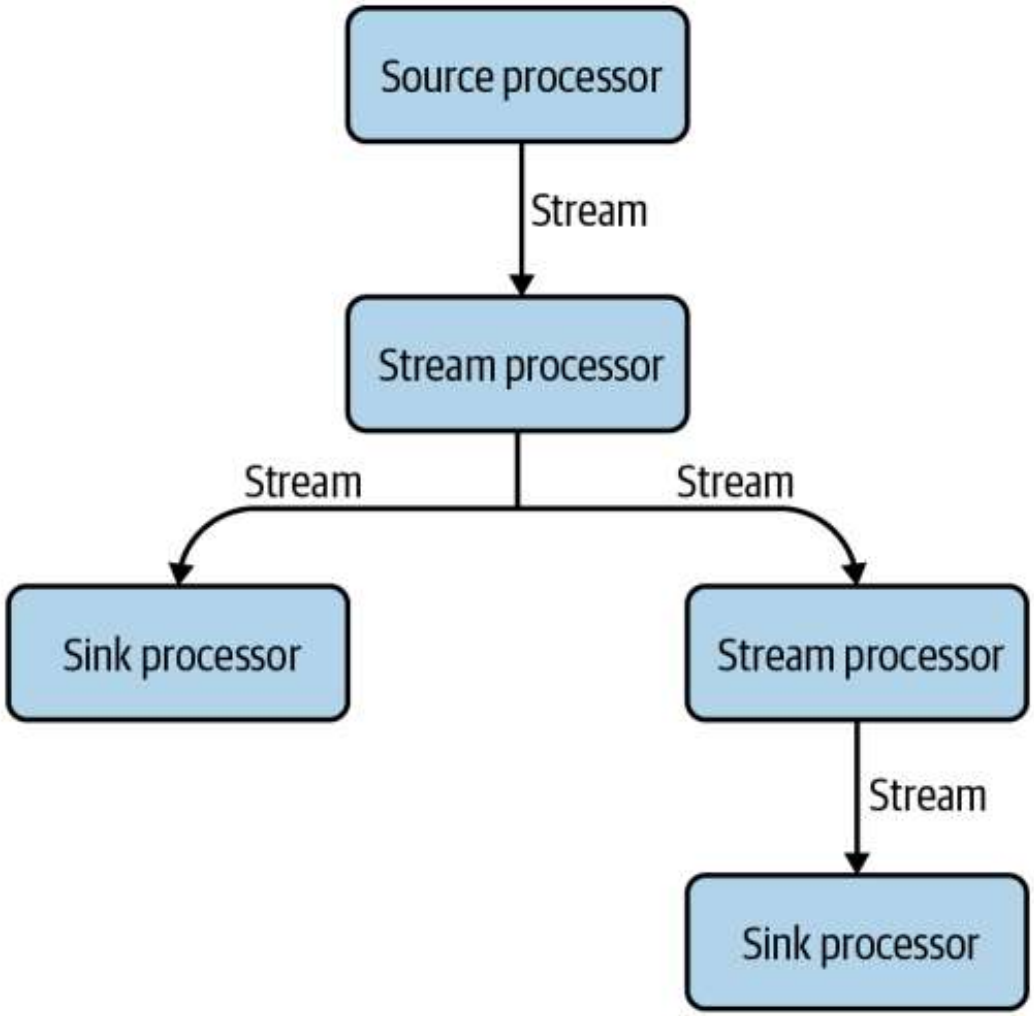

```
SELECT TimeZone, COUNT(*) AS Count
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY TimeZone, TumblingWindow(second,10)
```

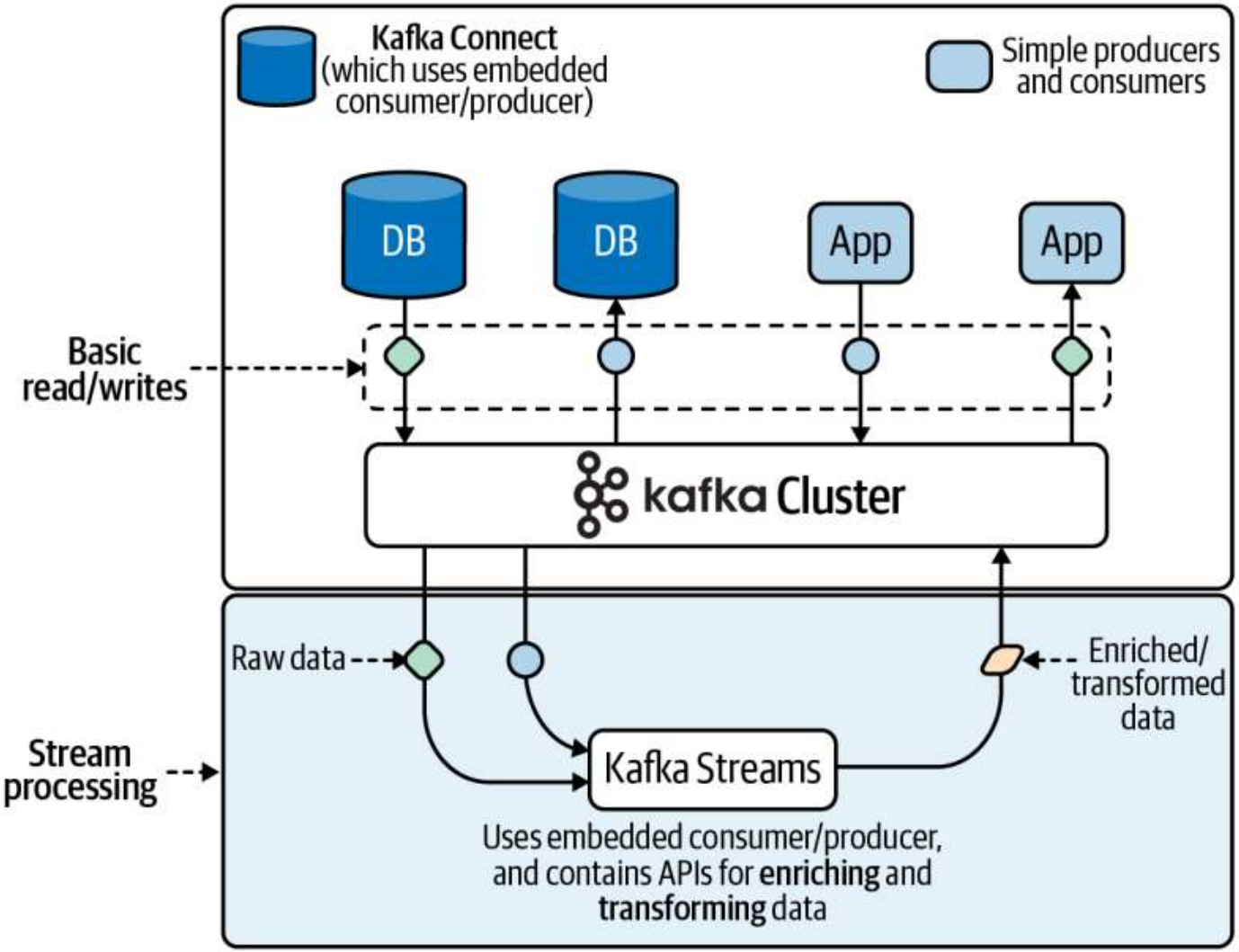
Window Operations - Session Window

Tell me the count of Tweets that occur within 5 minutes to each other

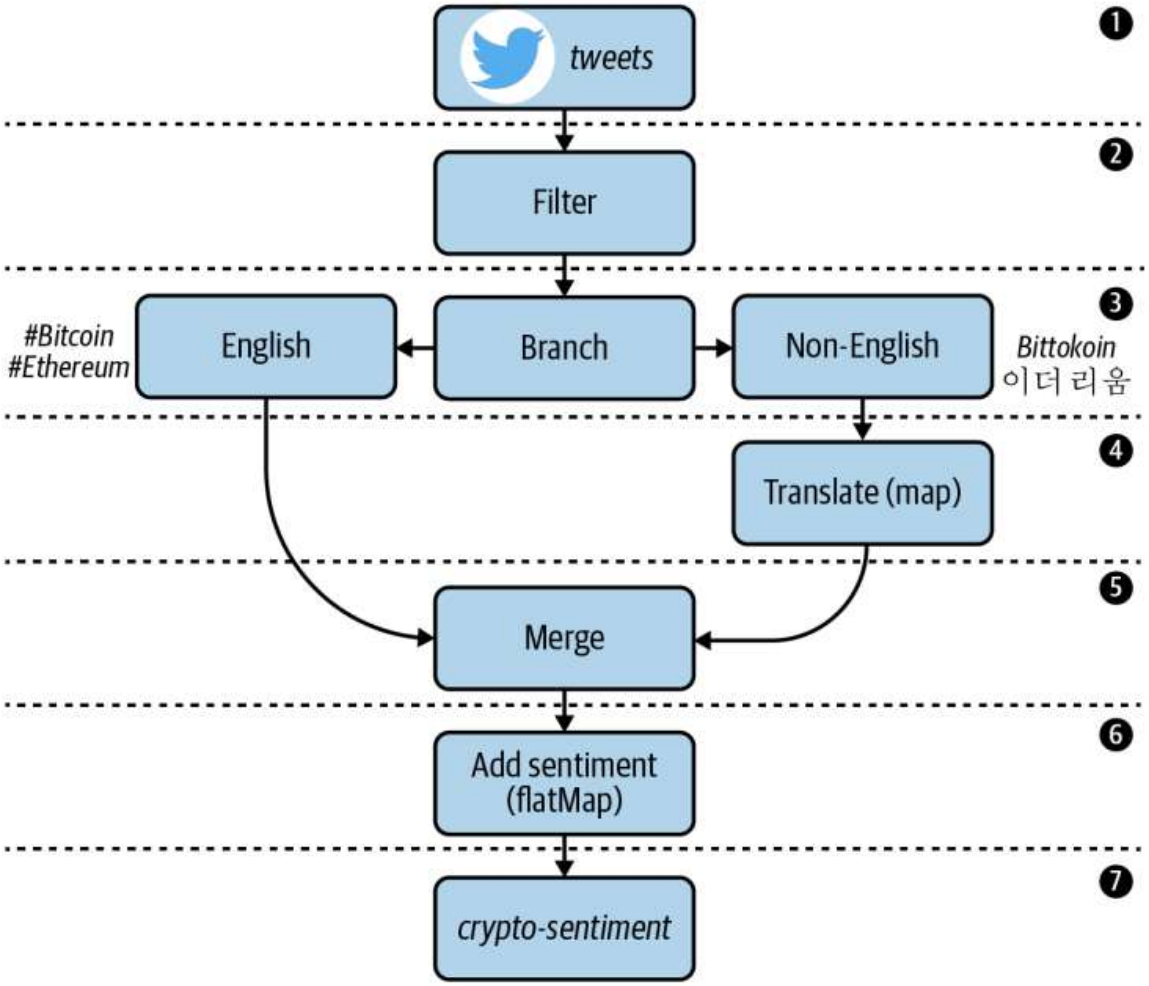


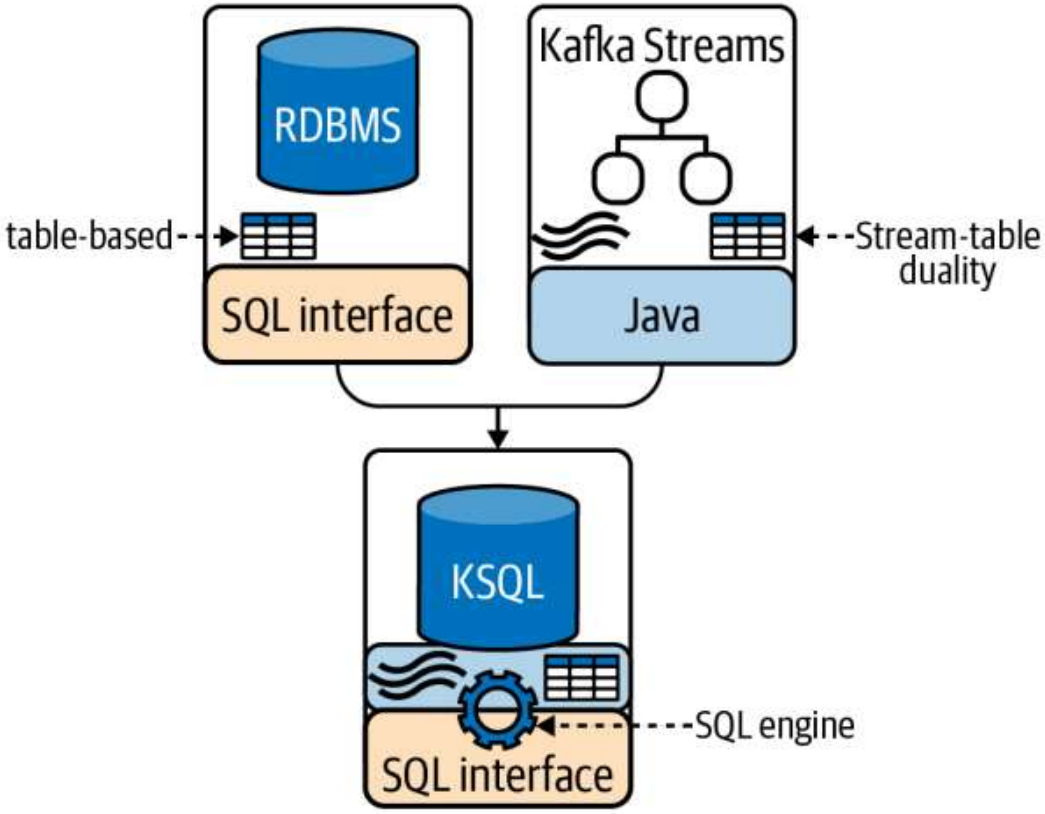

```
SELECT Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, SessionWindow(minute, 5, 10)
```

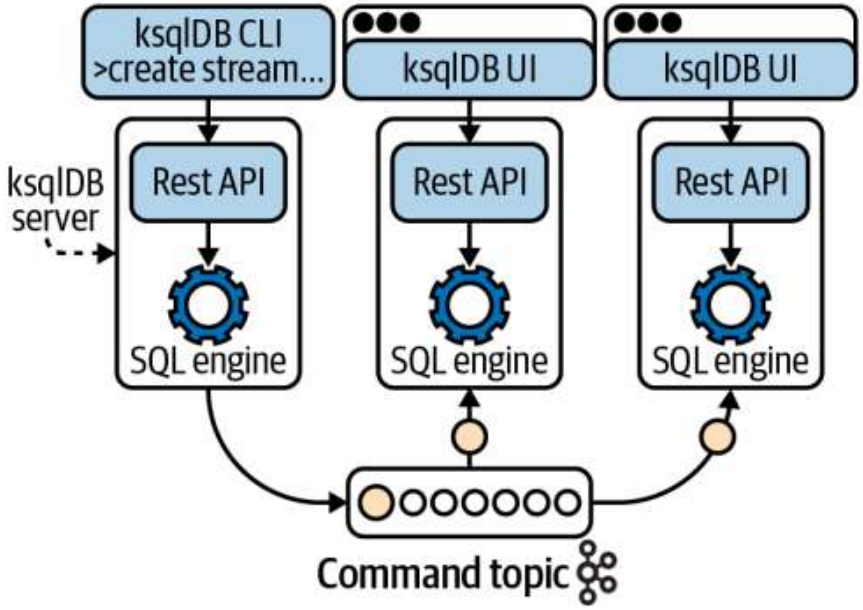

Source processor
Stream
Stream processor
Stream
Stream
Sink processor
Stream processor
Stream
Sink processor


Kafka Connect
(which uses embedded
consumer/producer)
Simple producers
and consumers
DB
DB
App
App
Basic
read/writes
kafka Cluster
Raw data
Enriched/
transformed
data
Stream
processing
Kafka Streams
Uses embedded consumer/producer,
and contains APIs for enriching and
transforming data

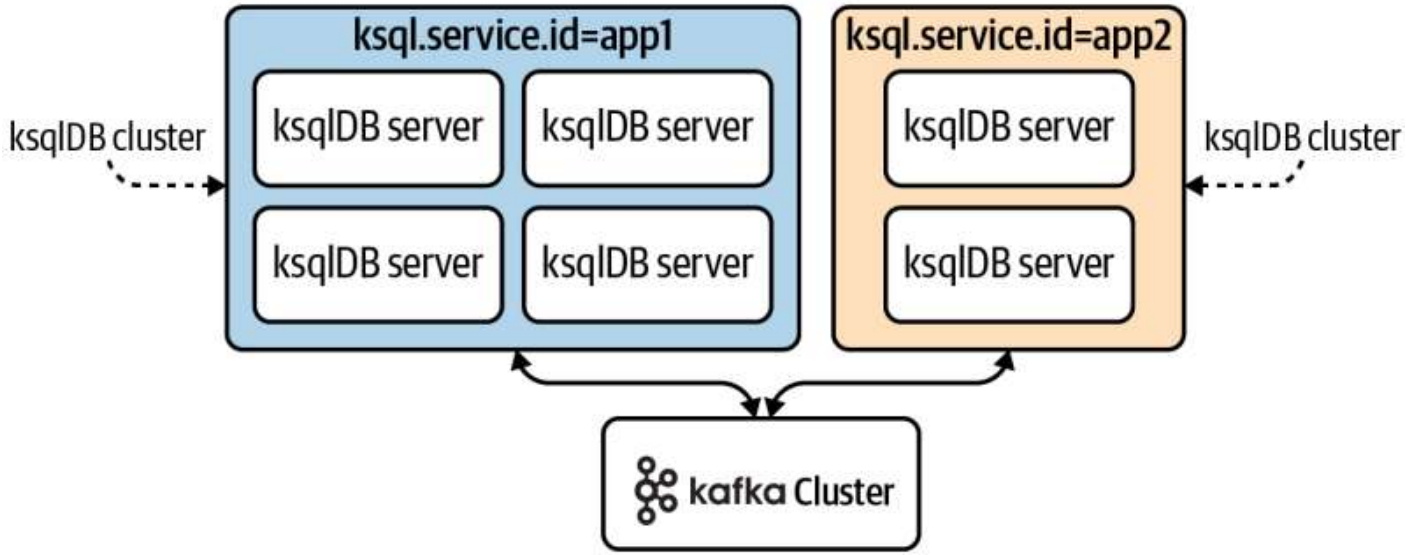
Tweet enrichment application



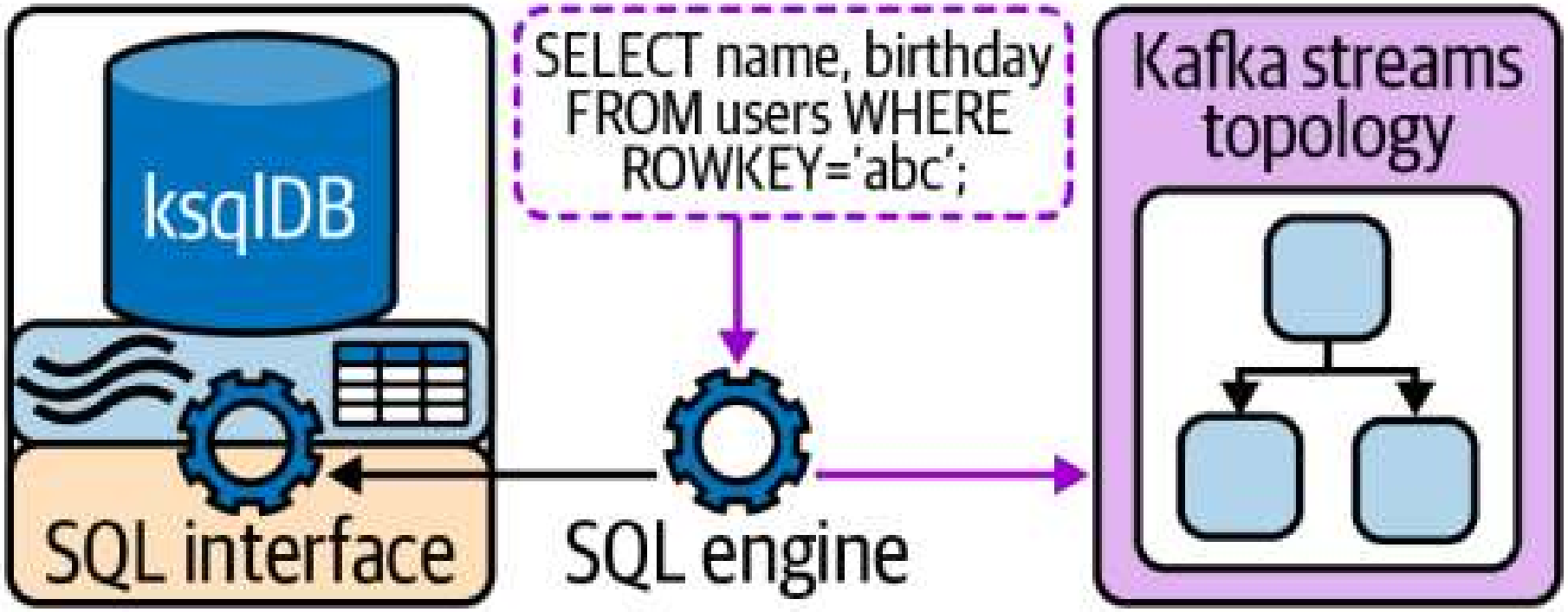

RDBMS
Kafka Streams
table-based
Stream-table duality
SQL interface
Java
KSQL
SQL engine
SQL interface


ksqlDB CLI
>create stream...
ksqlDB UI
ksqlDB UI
ksqlDB
server
Rest API
Rest API
Rest API
SQL engine
SQL engine
SQL engine
Command topic

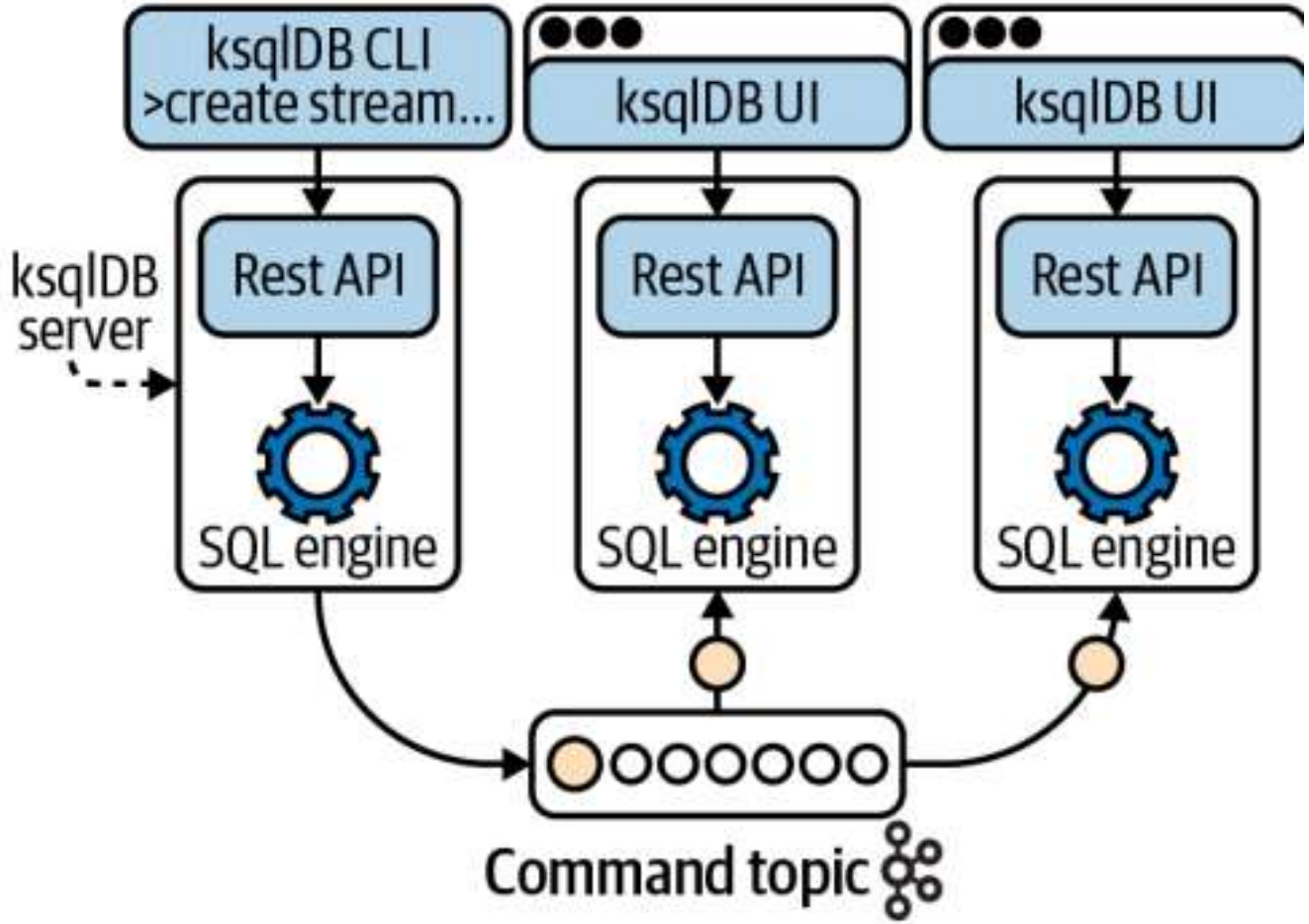
Two kqlDB Clusters



SQL engine converts SQL statements into Kafka Streams



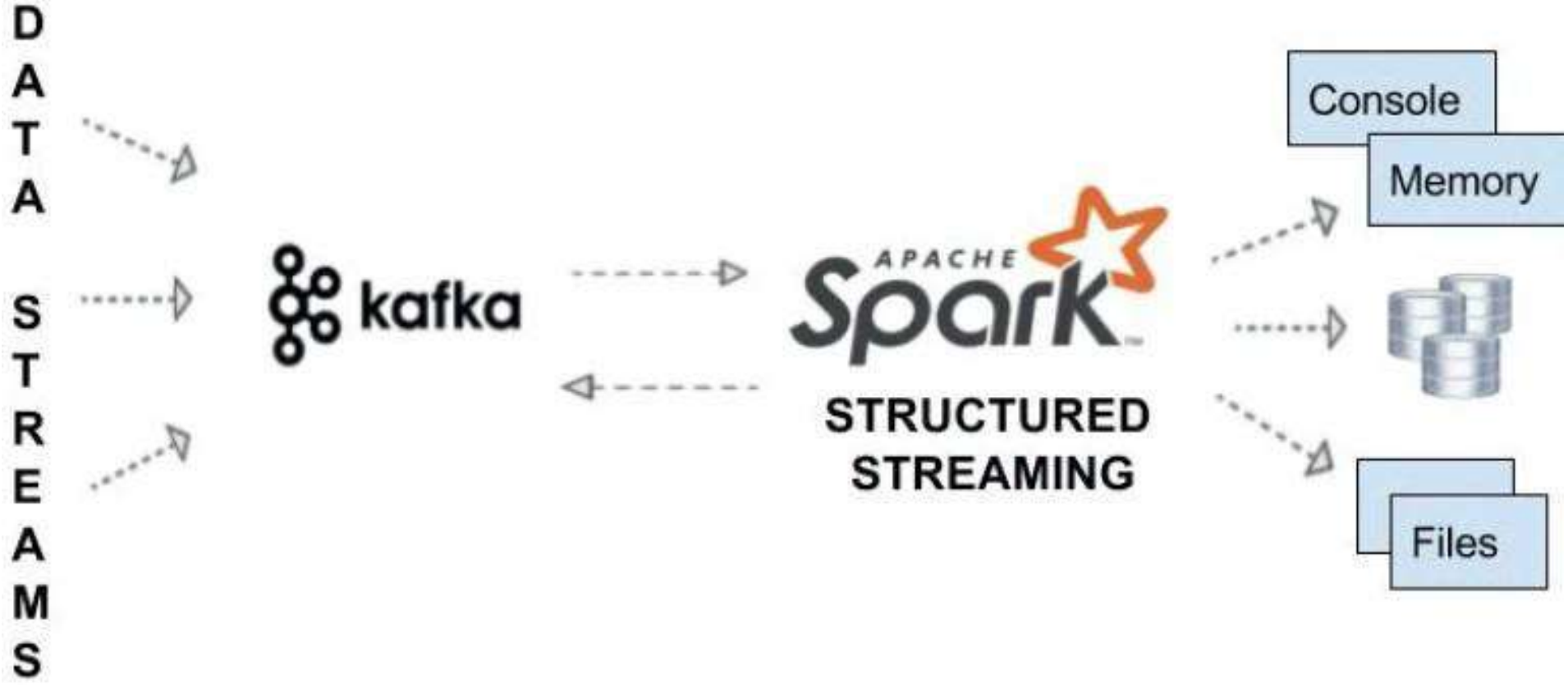
ksqlDB running in interactive mode



Spark & Kafka Integration

Integration of Spark with Kafka

- Allows a parallelism between partitions of Kafka and Spark



Integration of Spark with Kafka

- Example of real-time fraud detection system
  - Data from various transaction sources, such as credit card transactions and online purchases, is ingested into Kafka topics
  - Spark streams
    - Consume the data
    - Apply machine learning algorithms
    - Apply anomaly detection techniques
    - Generate real-time alerts for potential fraudulent activities

Thanks