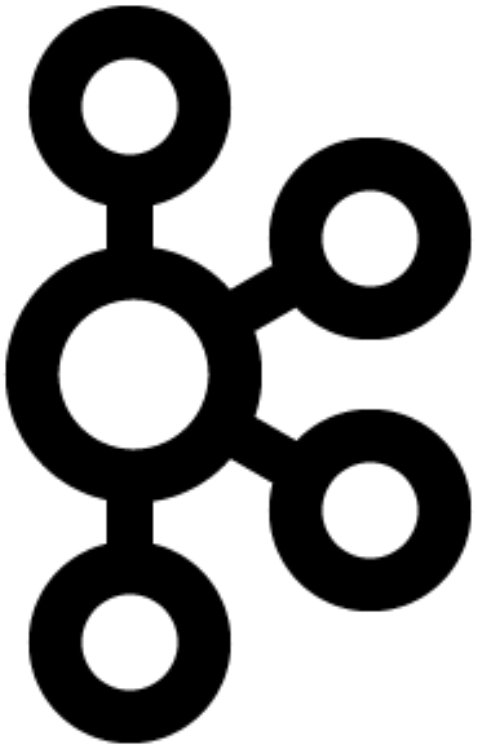


APACHE kafka®

A distributed streaming platform

Need of Messaging Systems

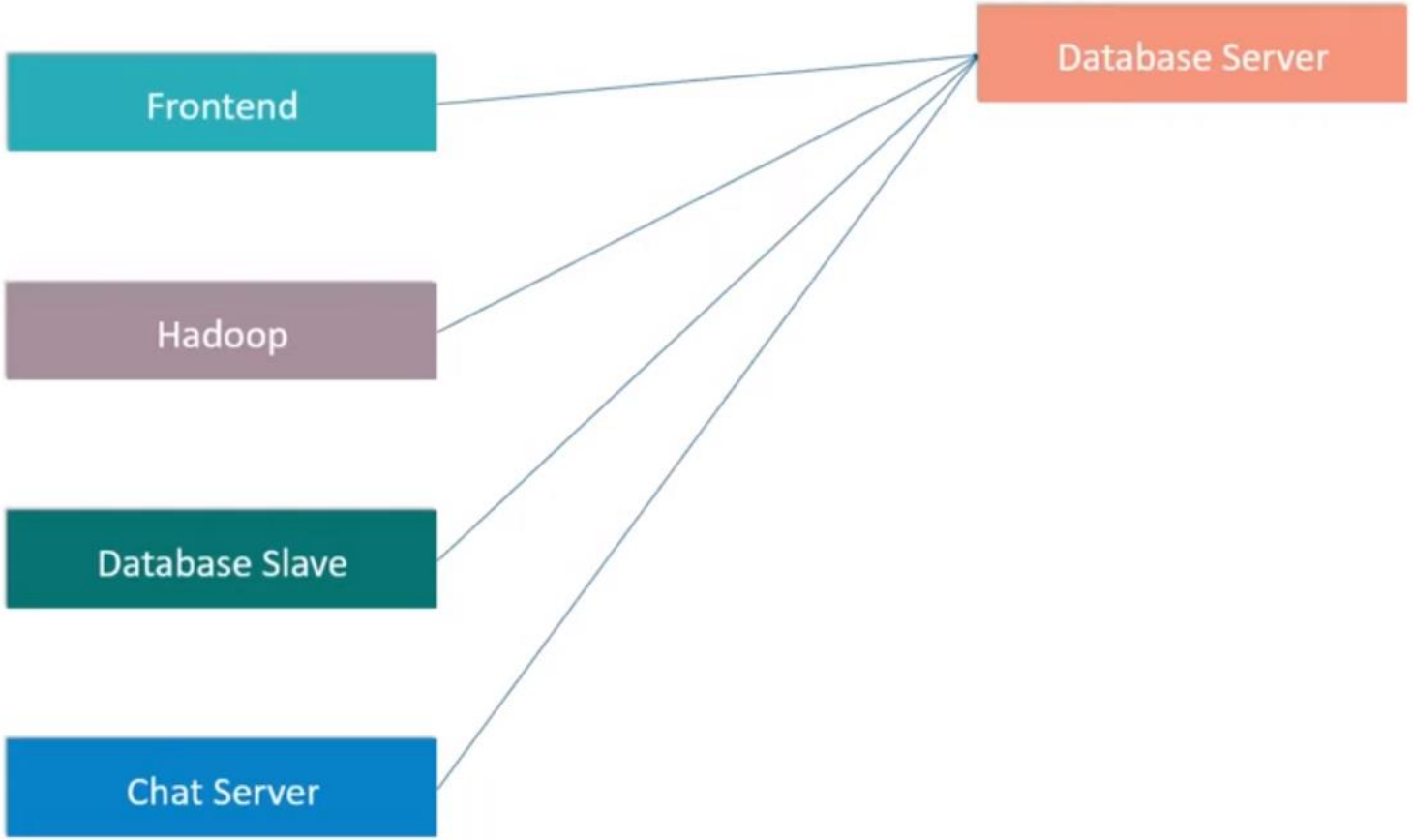
Data Pipelines

Communication is required between different systems in the real-time scenario, which is done by using data pipelines.

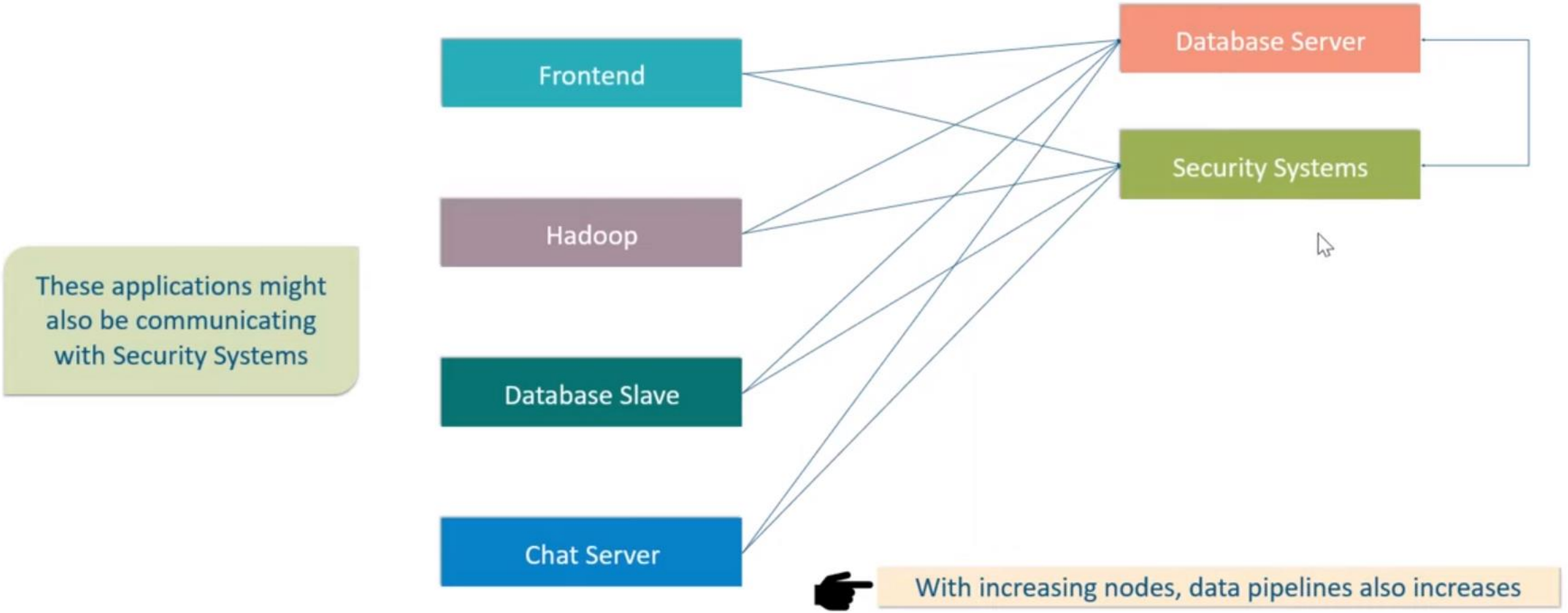
For Example: Chat Server needs to communicate with Database Server for storing messages

Increase in number of Nodes

Similarly, there may be many applications wanting to access the Database Server

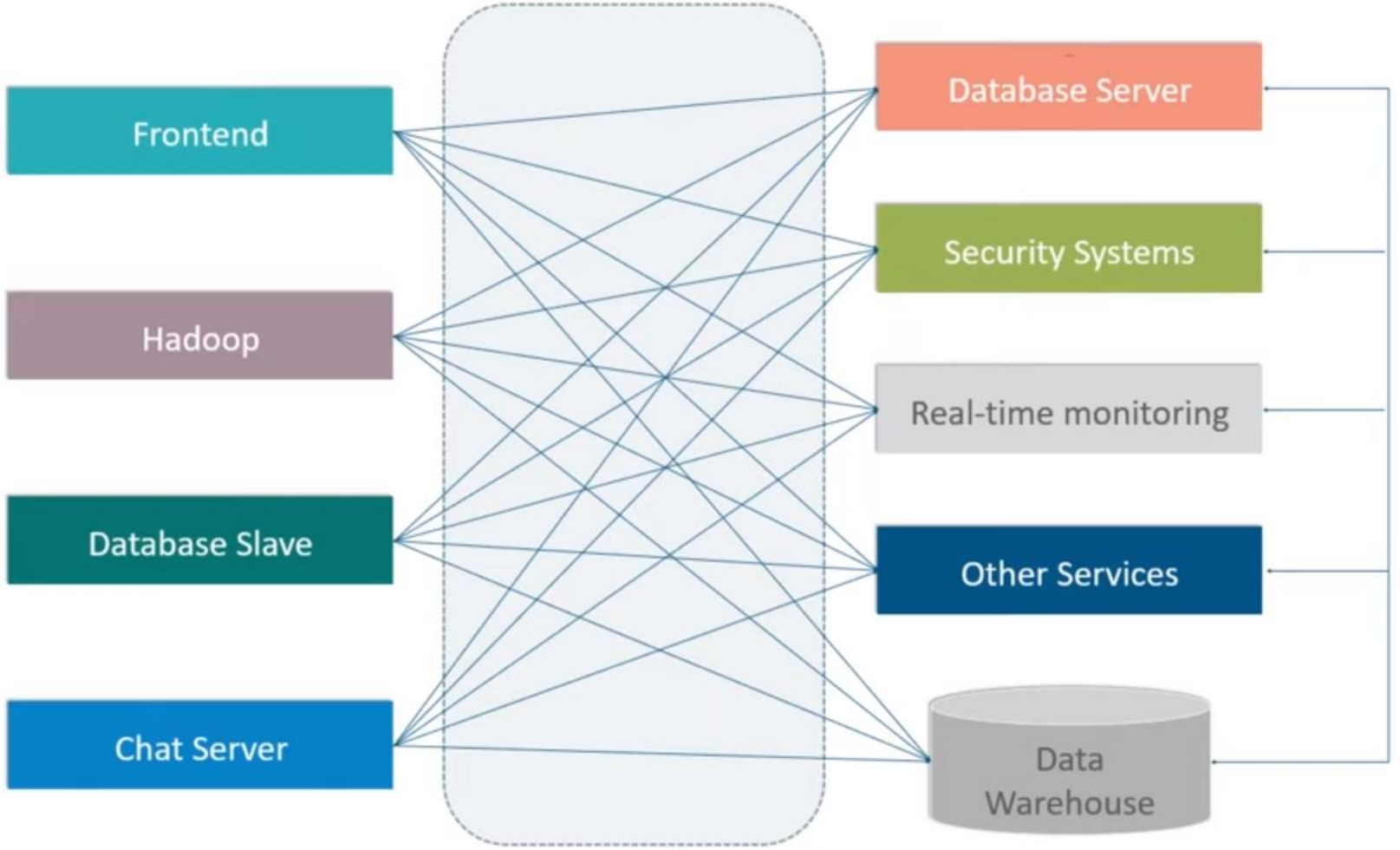


Increase in number of Nodes



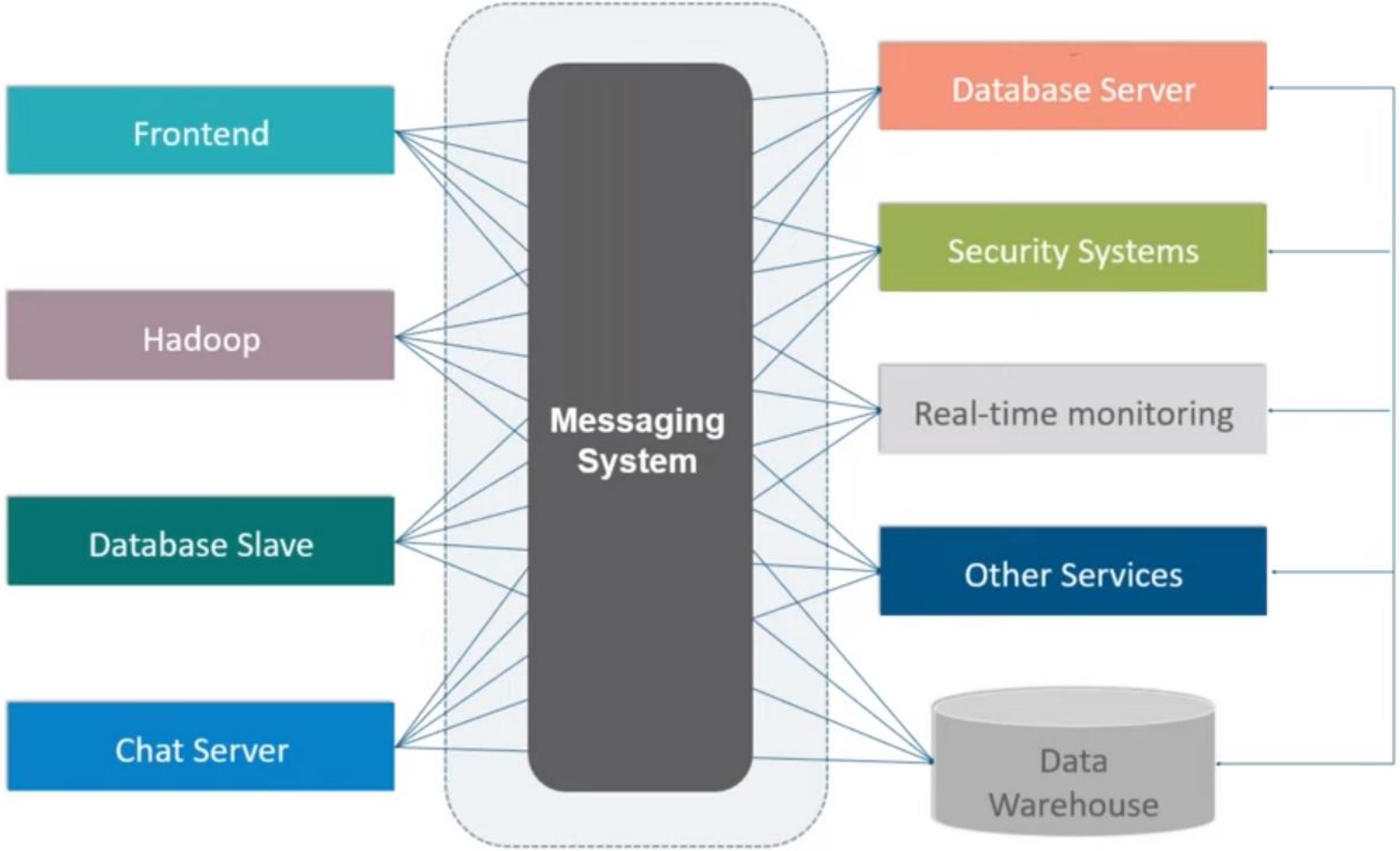
Complex Data Pipelines

Similarly, applications may also be communicating with Real-time monitoring and Other services in real-time scenario



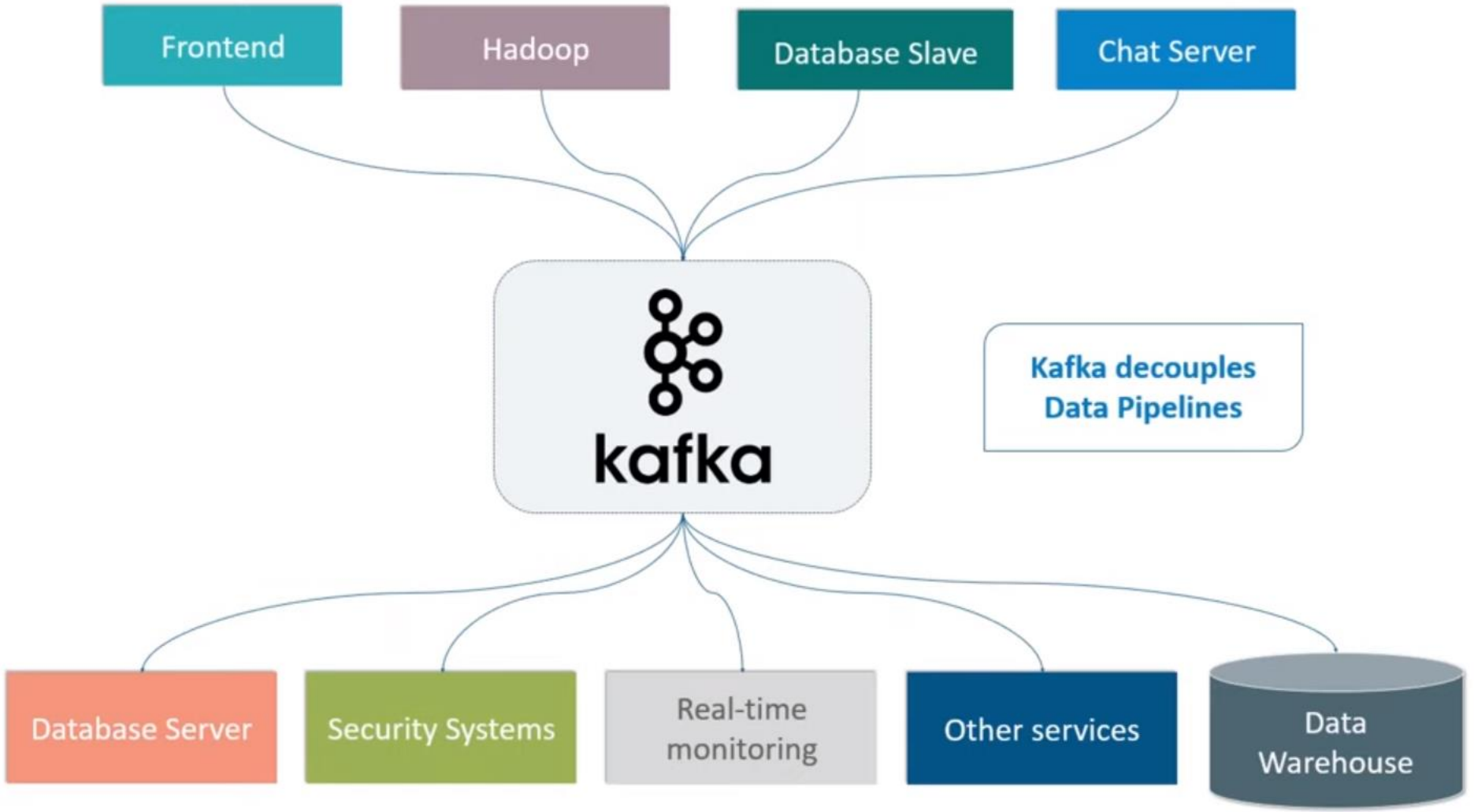
Solution to the Complex Data Pipelines

Messaging Systems helps managing the complexity of the pipelines



Let’s See How Kafka Solves the Problem

Kafka Decouples Data Pipelines



What is Kafka?

- **Apache Kafka** is a distributed *publish-subscribe* messaging system
- It was originally developed at LinkedIn and later on became a part of Apache Project
- Kafka is fast, scalable, durable, fault-tolerant and distributed by design

Kafka @LinkedIn

- *1100+ commodity machines*
- *31,000+ topics*
- *350,000+ partitions*

- *675 billion messages/day*
- *150 TB/day in*
- *580 TB/day out*

Peak Load

- *10.5 million messages/sec*
- *18.5 GB/sec Inbound*
- *70.5 GB/sec Outbound*

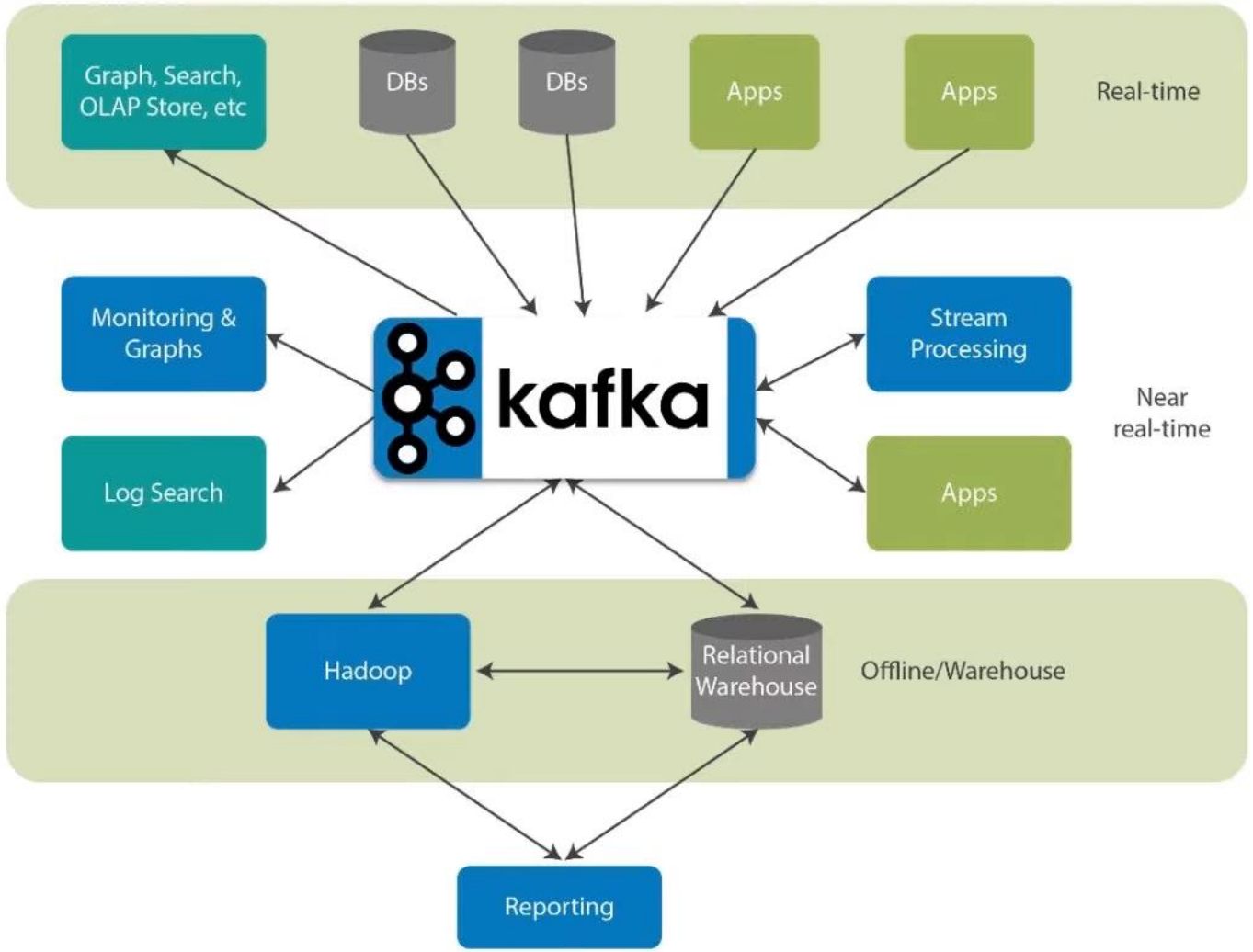


Fig: A modern stream-centric data architecture built around Kafka

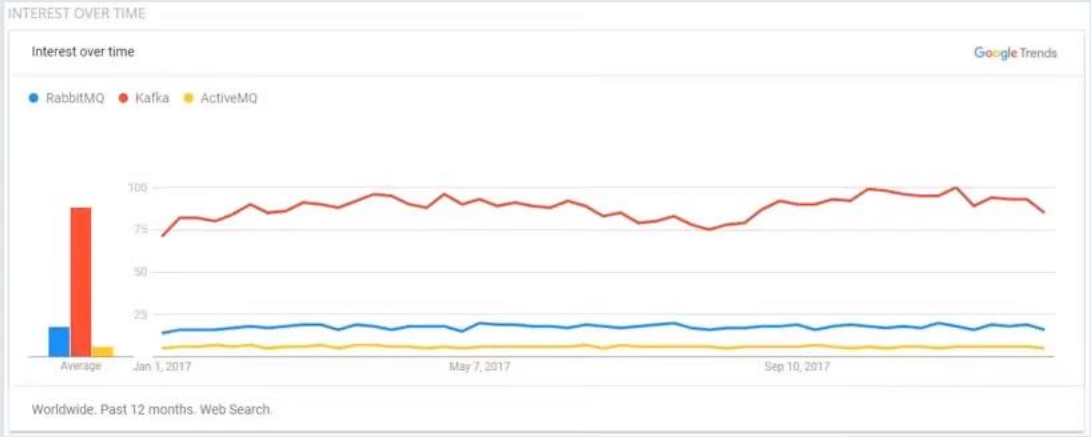
Kafka Growth Exploding

- More than **1/3** of all Fortune **500** companies use **Kafka**.
- These companies includes the top ten travel companies, **7** of top ten banks, **8** of top ten insurance companies, **9** of top ten telecom companies.
- **LinkedIn**, **Microsoft** and **Netflix** process billions of messages a day with Kafka (1,000,000,000,000).
- **Kafka** is used for **real-time streams** of data & used to collect big data for **real time analysis**.

86% of respondents reported that the number of their systems that use Kafka is increasing

20% reported that the number is "growing a lot!"

52% of organizations have at least 6 systems running Kafka



Source: Google Trends

Kafka Concepts

Kafka Terminologies

Producer

A ***producer*** can be any application who can publish messages to a topic

Consumer

A ***consumer*** can be any application that subscribes to a topic and consume the messages

Partition

Topics are *broken* up into *ordered commit logs* called ***partitions***

Broker

Kafka cluster is a set of servers, each of which is called a ***broker***

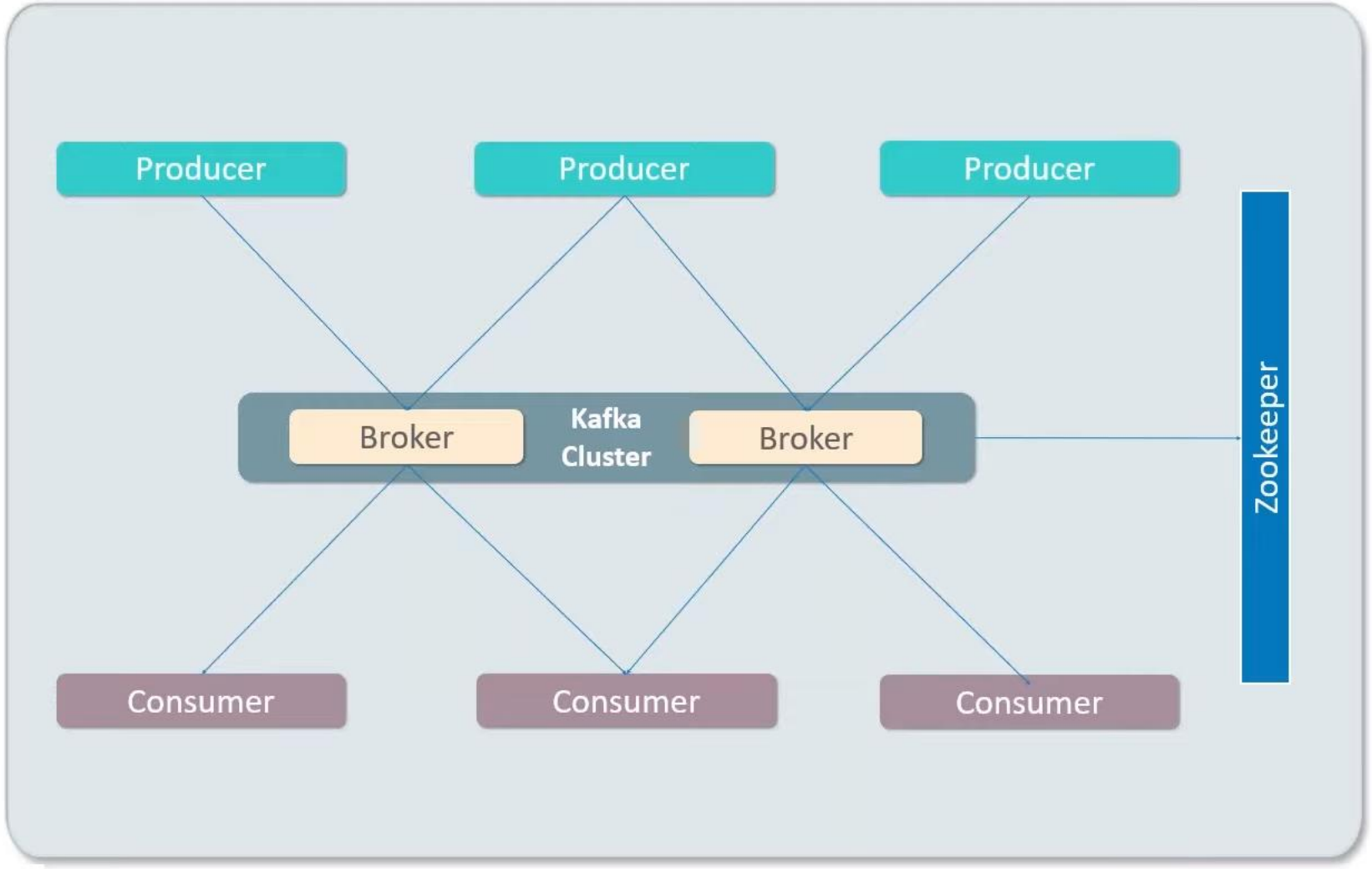
Topic

A ***topic*** is a category or *feed name* to which *records are published*

Zookeeper

ZooKeeper is used for managing and coordinating Kafka broker

Kafka Cluster

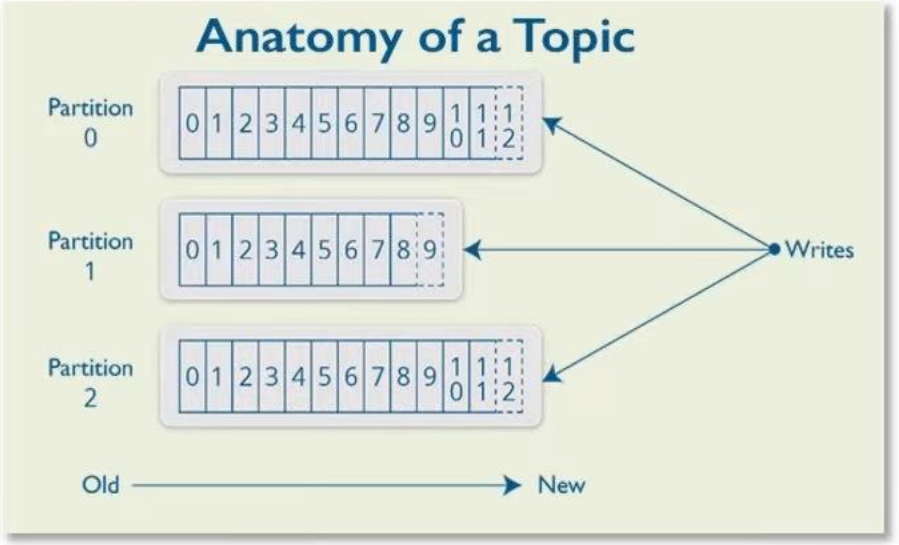


Kafka Features

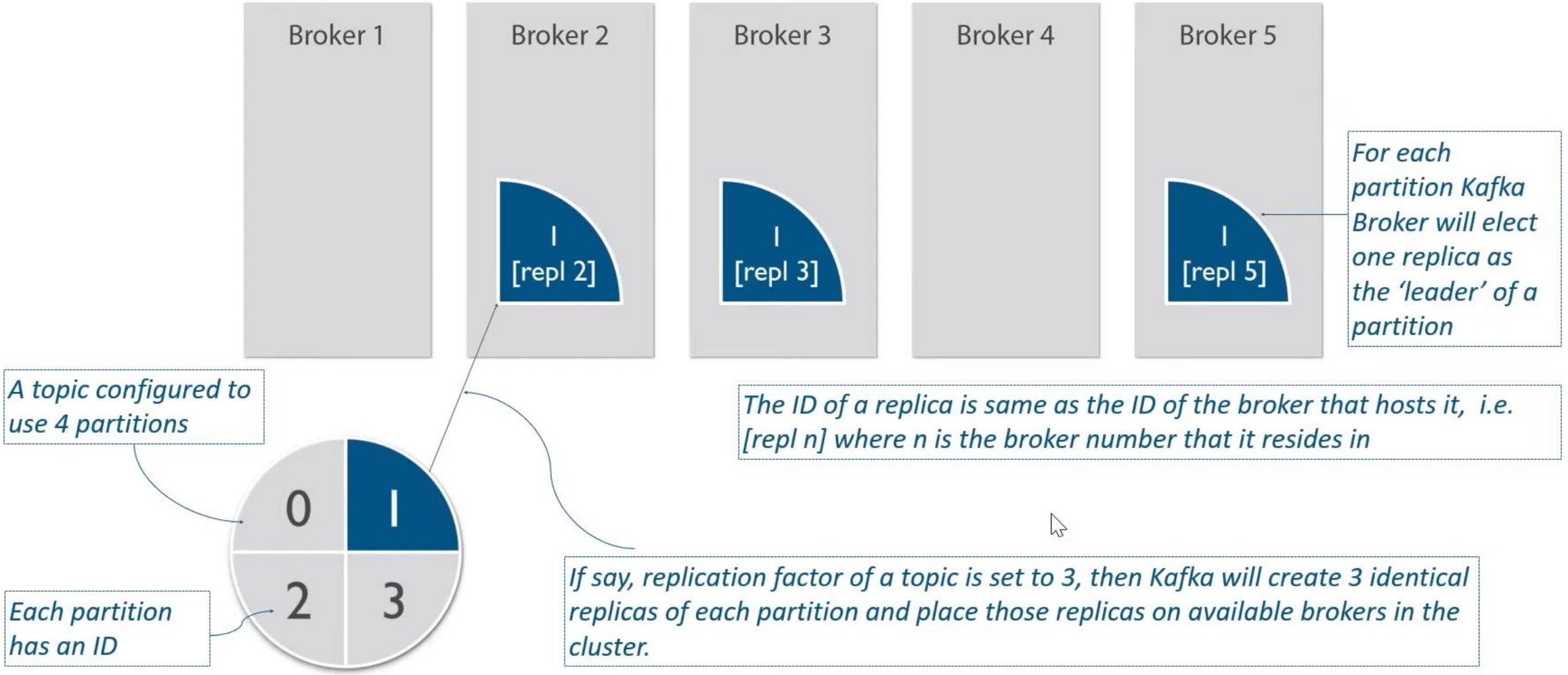
- **High Throughput:** Provides Support for Hundreds of thousands of messages with modest hardware
- **Scalability:** Highly scalable distributed systems with no downtime
- **Data Loss:** Kafka ensures no data loss once configured properly
- **Stream Processing:** Kafka can be used along with real time streaming applications like Spark and Storm
- **Durability:** Provides support to persisting messages on disk
- **Replication:** Messages can be replicated across clusters, which supports multiple subscribers

Kafka Components - Topics and Partitions

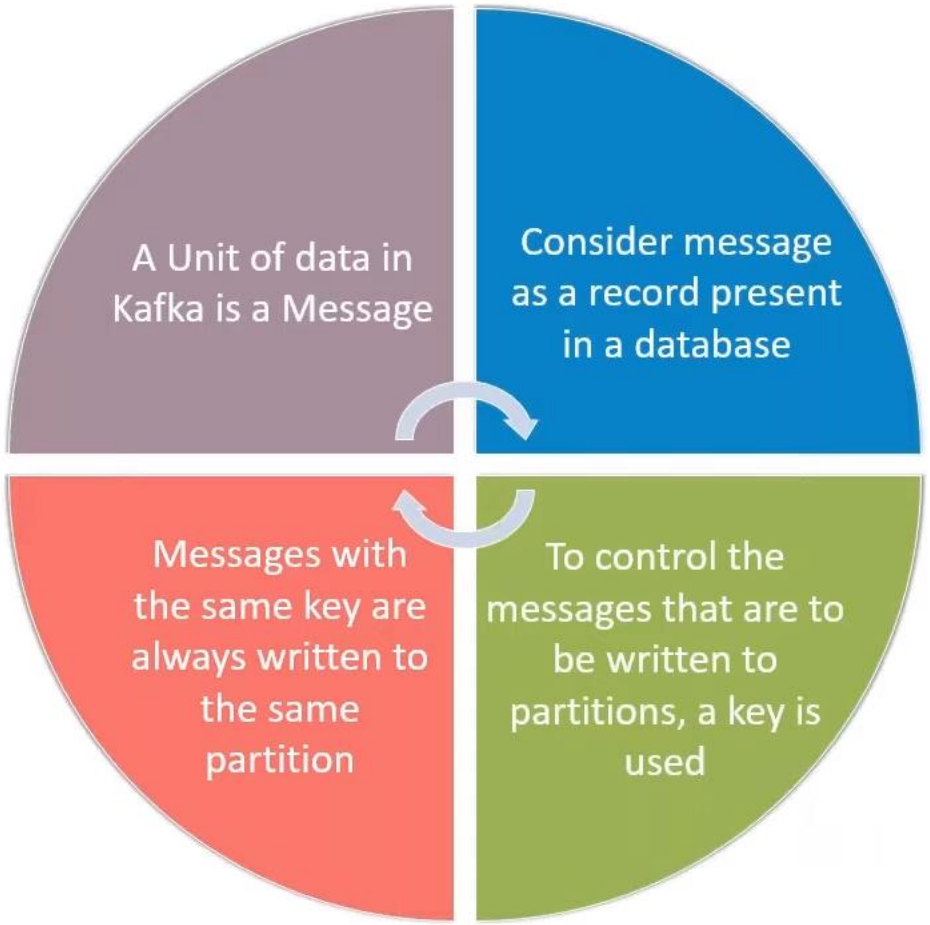
- A *topic* is a category or *feed name* to which *records are published*
- *Topics* are *broken* up into *ordered commit logs* called *partitions*
- Each *message* in a *partition* is assigned a *sequential id* called an *offset*
- Data in a topic is retained for a *configurable period of time*
- *Writes* to a partition *are* generally *sequential* thereby *reducing* the *number of hard disk seeks*
- *Reading messages* can either be from *beginning* & *also* can *rewind* or *skip* to any *point* in partition by *giving* an *offset value*



Kafka Components - Topics, Partitions & Replicas



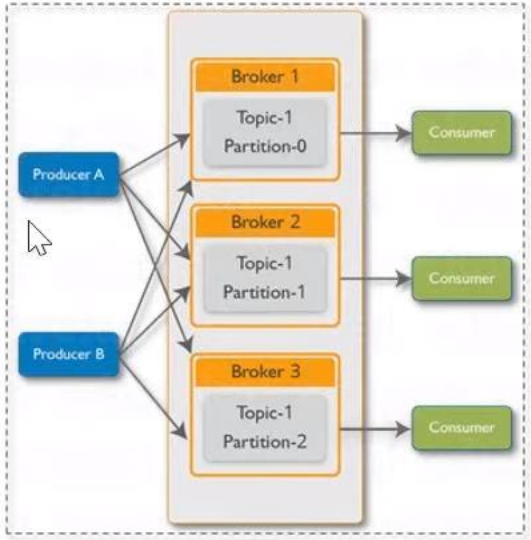
Kafka Components - Messages



Kafka Components - Producer

1

Producer (publisher or writer) publishes a new message to a **specific topic**

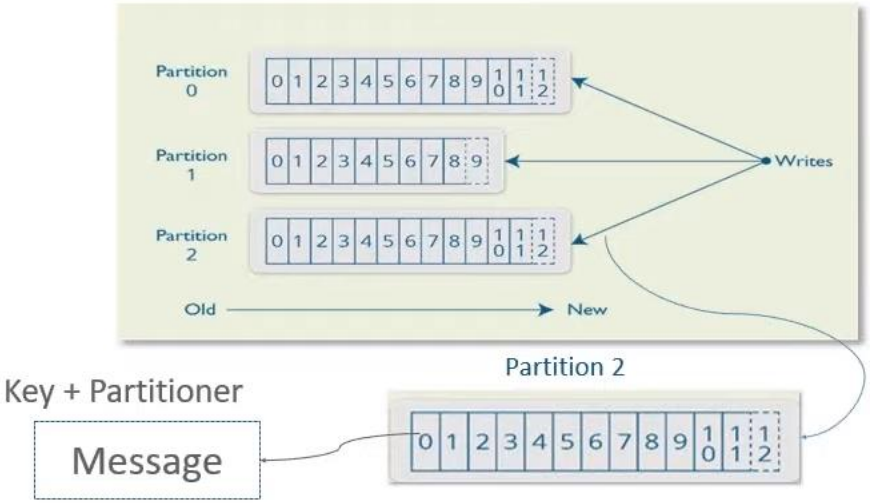


2

The producer does not care what partition a specific message is written to and will balance messages over every partition of a topic evenly

3

Directing messages to a partition is done using the **message key** and a **partitioner**, this will generate a hash of the key and map it to a partition

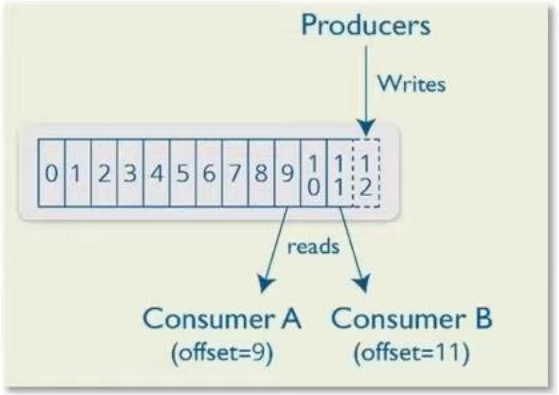


4

Every message a producer publishes in the form of a **key : value** pair

Kafka Components - Consumer

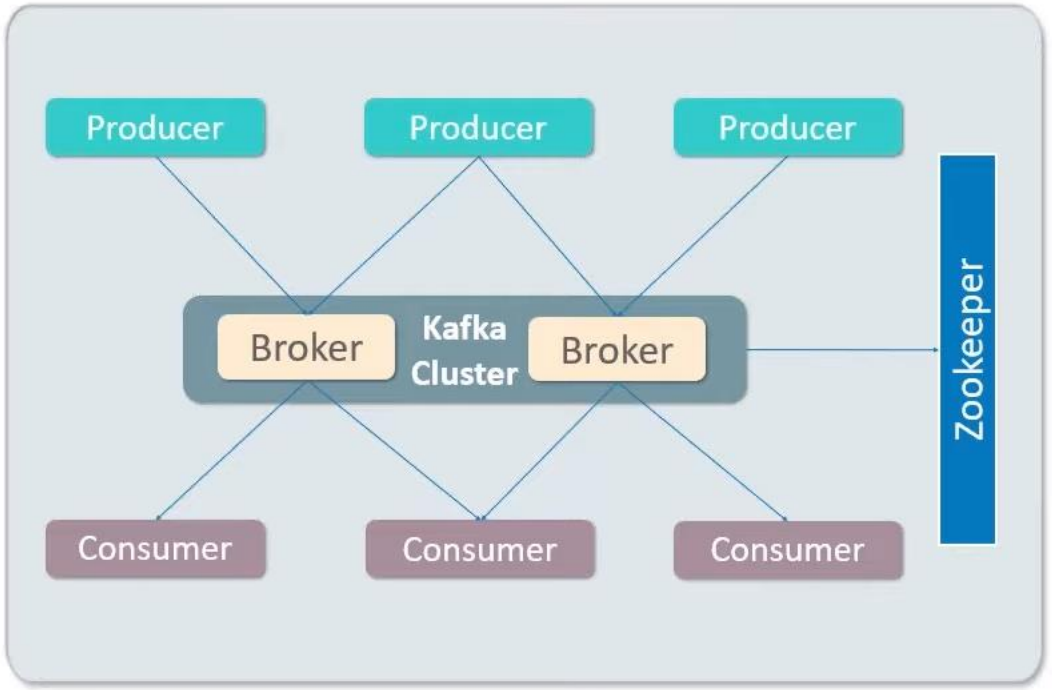
- *Consumers*(subscribers or readers) read messages
- The consumer subscribes to one or more topics and reads the messages sequentially
- The consumer keeps track of the messages it has consumed by keeping track on the offset of messages
- The *offset* is bit of metadata(an integer value that continually increases)that Kafka adds to each message as it is produced
- Each partition has a *unique offset* which is stored
- With the offset of the last consumed message, a consumer can *stop and restart without losing its current state*



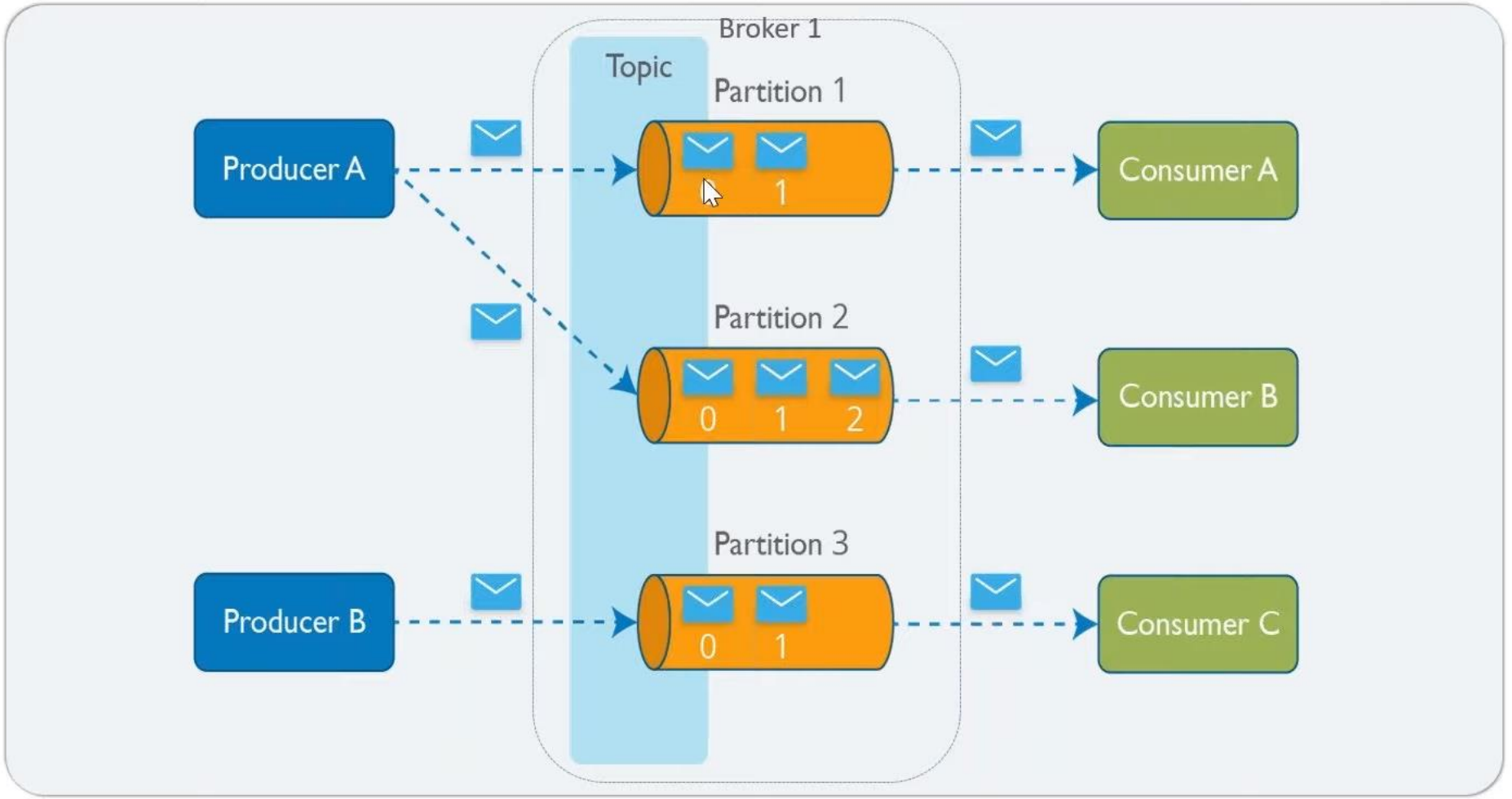
Kafka Components - ZooKeeper

ZooKeeper is used for managing and coordinating Kafka broker

- Zookeeper service is mainly used for co-ordinating between brokers in the Kafka cluster
- Kafka cluster is connected to ZooKeeper to get information about any failure nodes



Kafka Architecture



Let’s see some Use Cases of Kafka

Kafka - Use Cases

- Applications can produce messages using Kafka, without being concerned about the format of the messages
- Messages are sent and handled by a single application that can read all of them consistently, including :
  - A common formatting of messages using a common look
  - Send multiple messages in a single notification
  - Receive messages in a way that meets the users preferences

Kafka - Use Cases

- Originally Kafka was designed at LinkedIn, to track user activity
- When a user interacts with frontend applications, which generates messages regarding actions the user is taking
- Kafka keeps track of simple information like click tracking to complex information like data in a user's profile

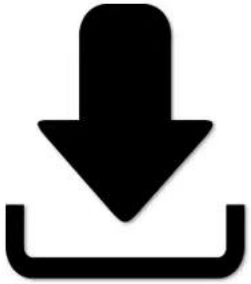
Kafka - Use Cases

- Kafka is also ideal for collecting application's and system metrics and logs
- Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting
- Log messages can be published in the same way and routed to dedicated log search systems like Elasticsearch or security analysis applications

Kafka - Use Cases

- Database changes can be published to Kafka and applications can easily monitor this stream to receive live updates as they happen
- Kafka replicates database updates to a remote system for consolidating changes from multiple applications in a single database view
- Durable retention becomes useful providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the consuming applications
- Log-compacted topics can be used to provide longer retention by only retaining a single change per key

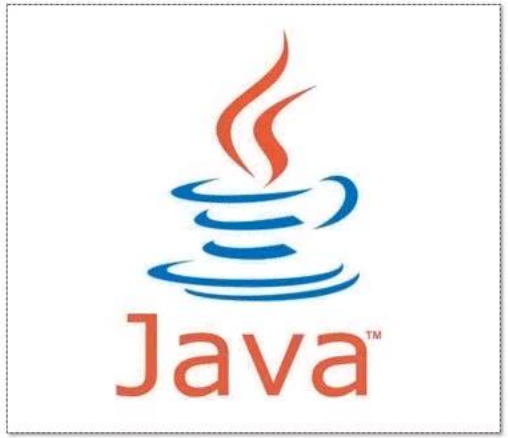
Kafka - Use Cases



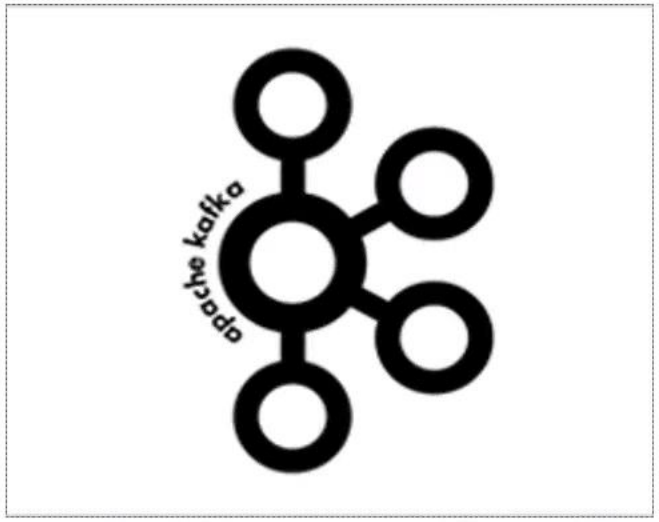
- Stream processing term is typically used to refer applications that provide similar functionality to map/reduce processing in Hadoop
- Stream processing operates on data in real-time, as quickly as messages are produced :
  - Write small applications to operate on Kafka messages,
  - Performing tasks such as counting metrics
  - Partitioning messages for efficient processing by other applications

Getting Started with Kafka

- **Prerequisites :**



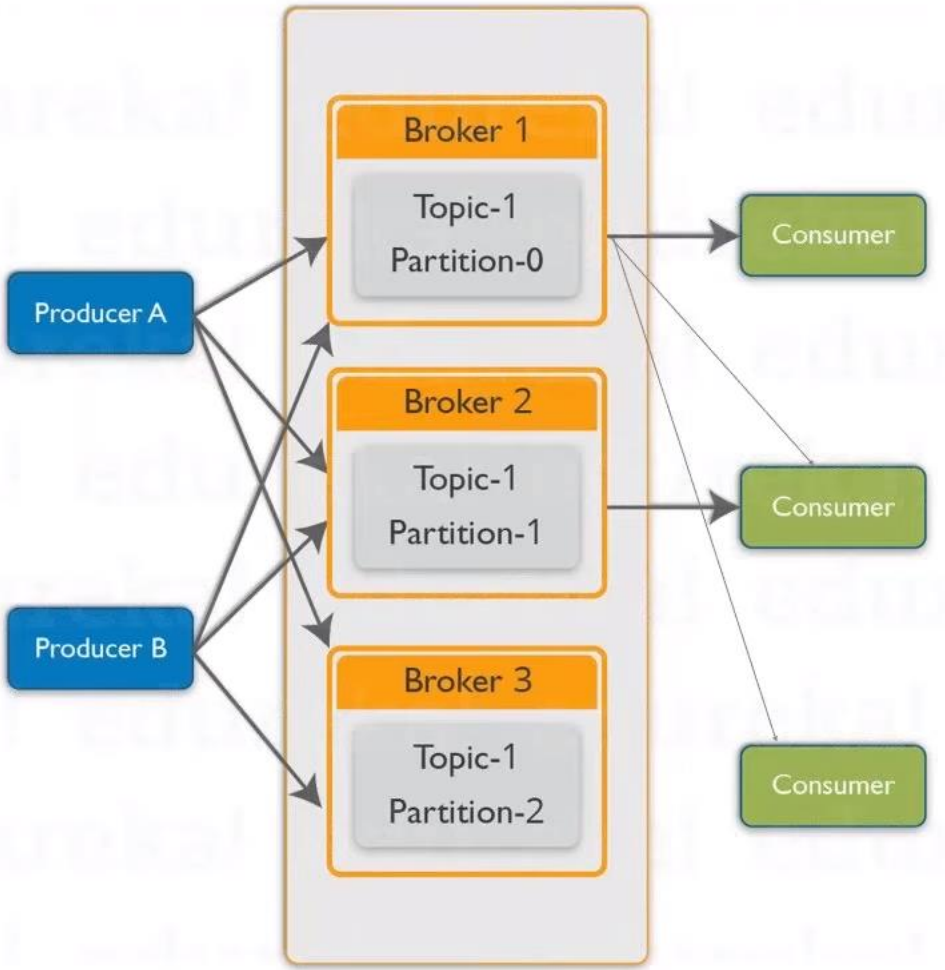
- **Components :**



Let’s Classify Different Types of Clusters in Kafka

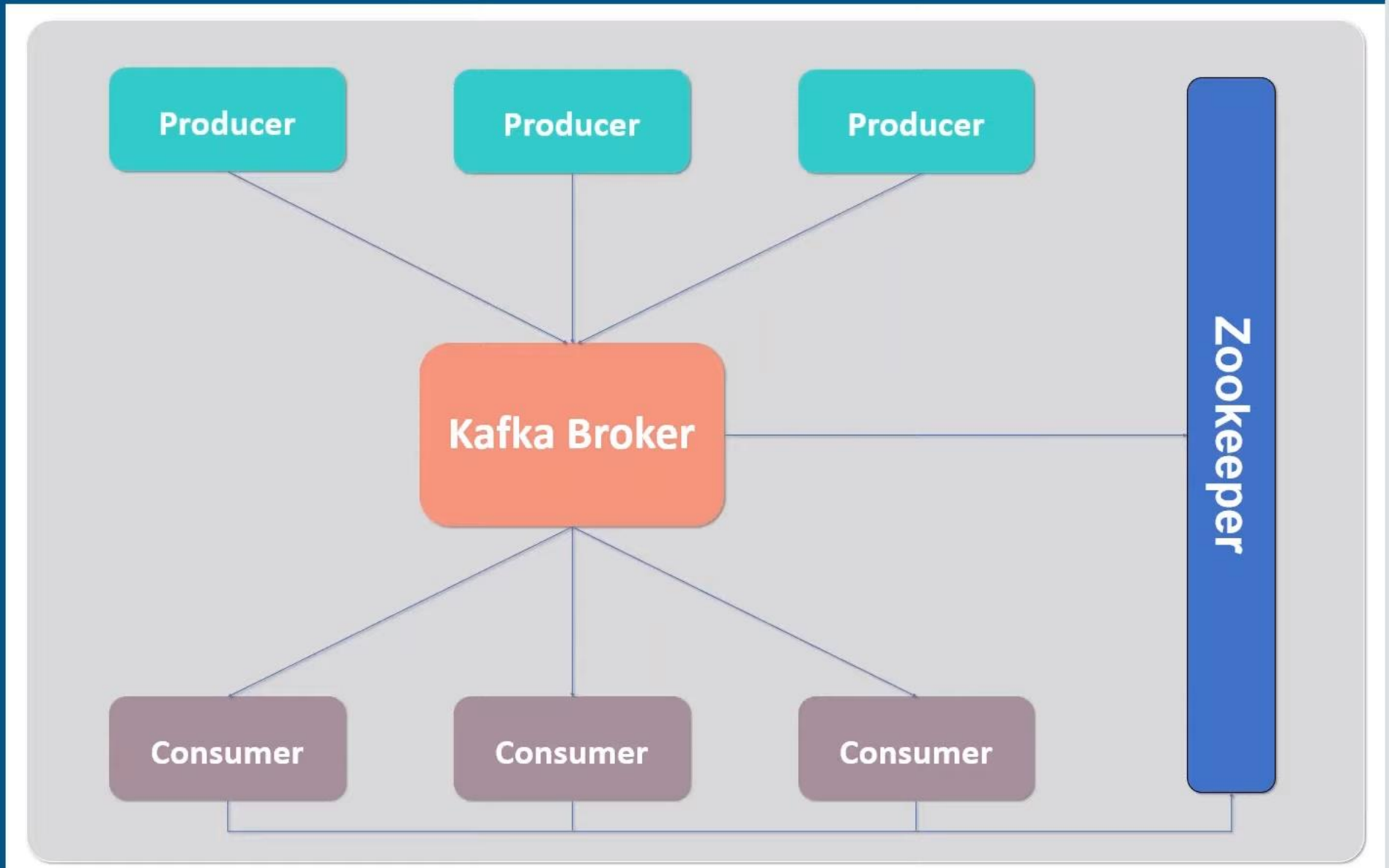
Kafka Cluster

- Kafka brokers are designed to operate as part of a cluster
- One broker will also function as the cluster controller
- Controller is responsible for administrative operations, like
  - Assigning partitions to brokers
  - Monitoring for broker failures in a cluster
- A particular partition is owned by a broker, and that broker is called the leader of the partition
- All consumers and producers operating on that partition must connect to the leader



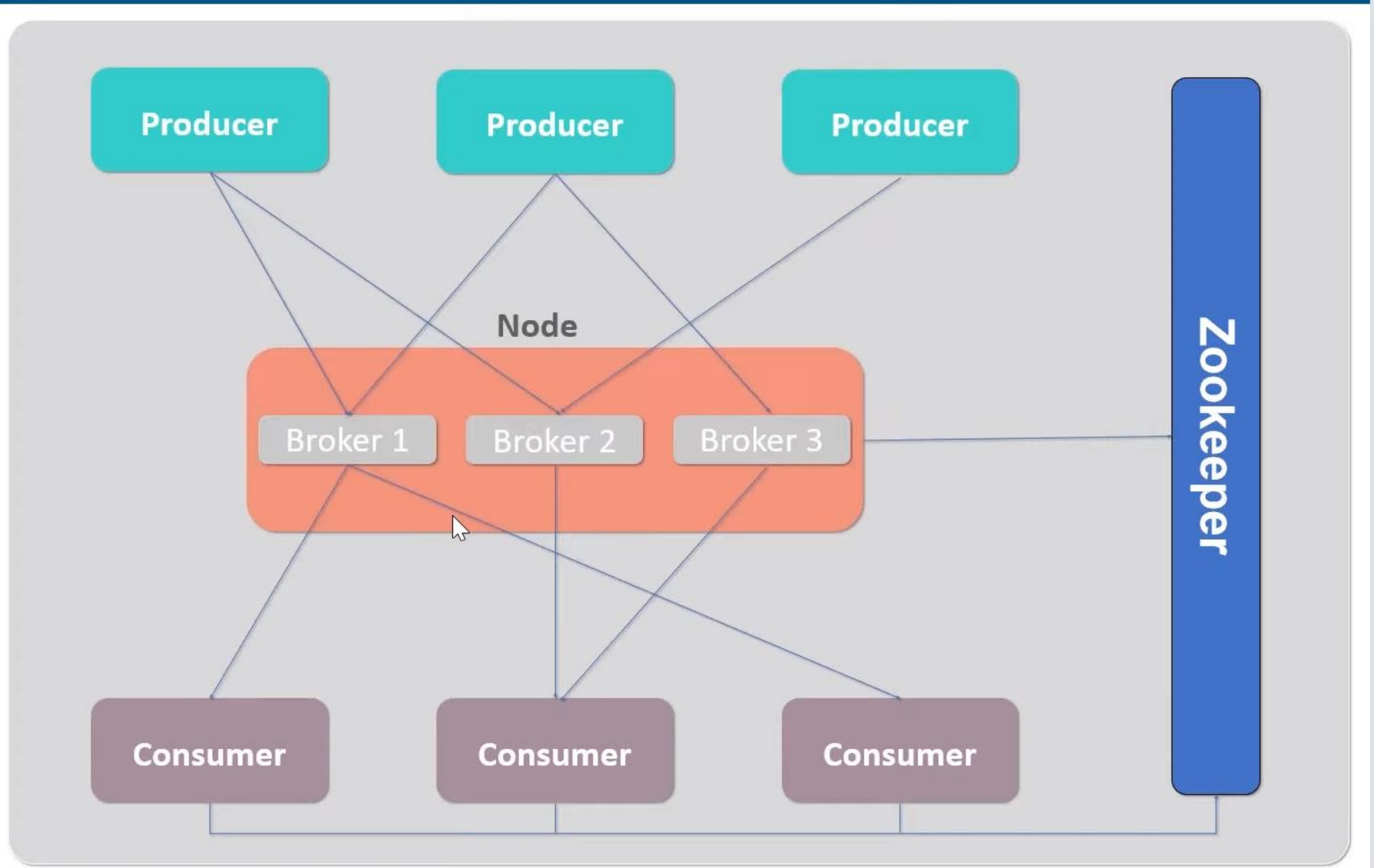
Type of Kafka Clusters

1. **Single Node-Single Broker Cluster**
2. Single Node-Multiple Broker Cluster
3. Multiple Nodes-Multiple Broker Cluster



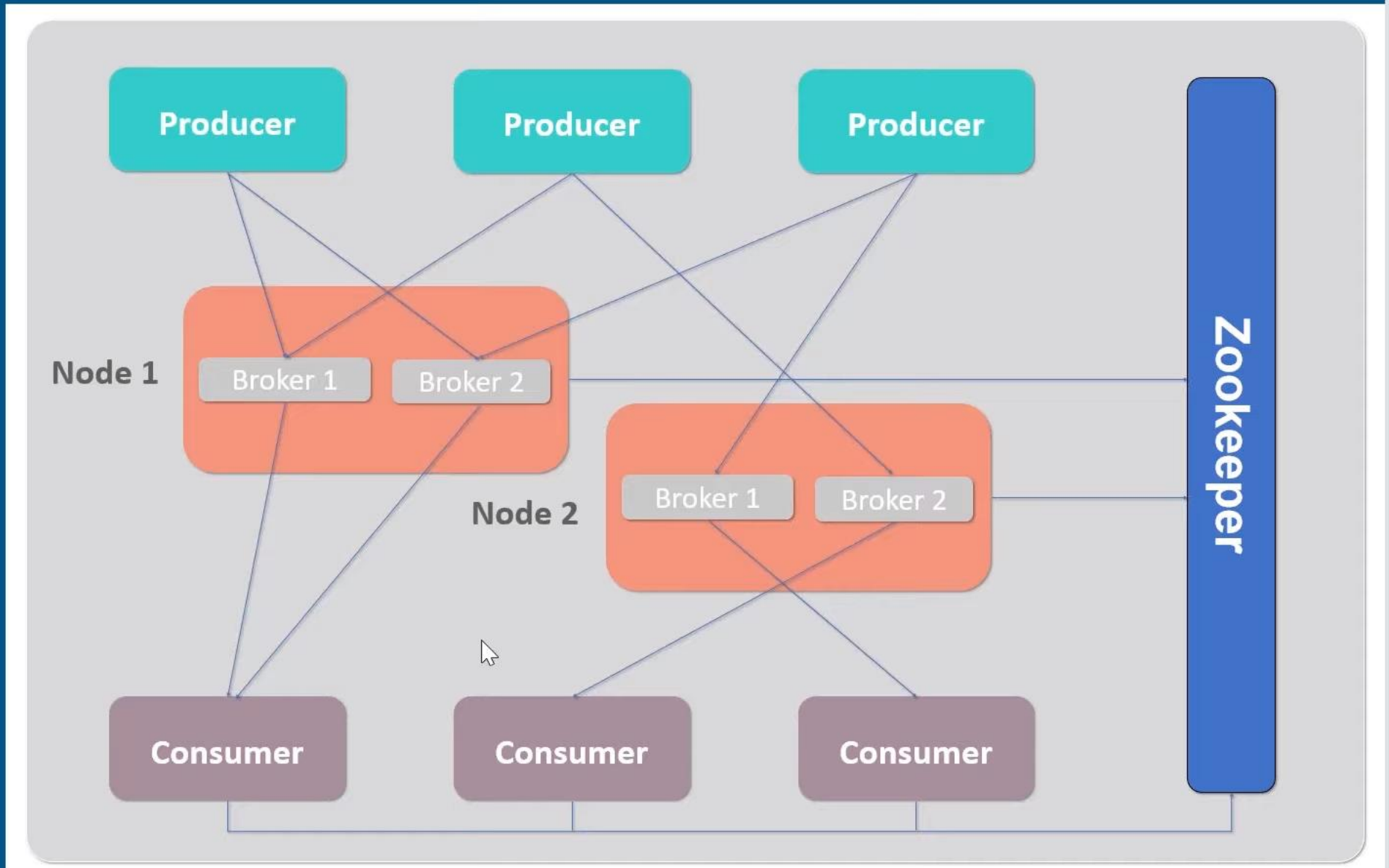
Type of Kafka Clusters

1. Single Node-Single Broker Cluster
2. **Single Node-Multiple Broker Cluster**
3. Multiple Nodes-Multiple Broker Cluster



Type of Kafka Clusters

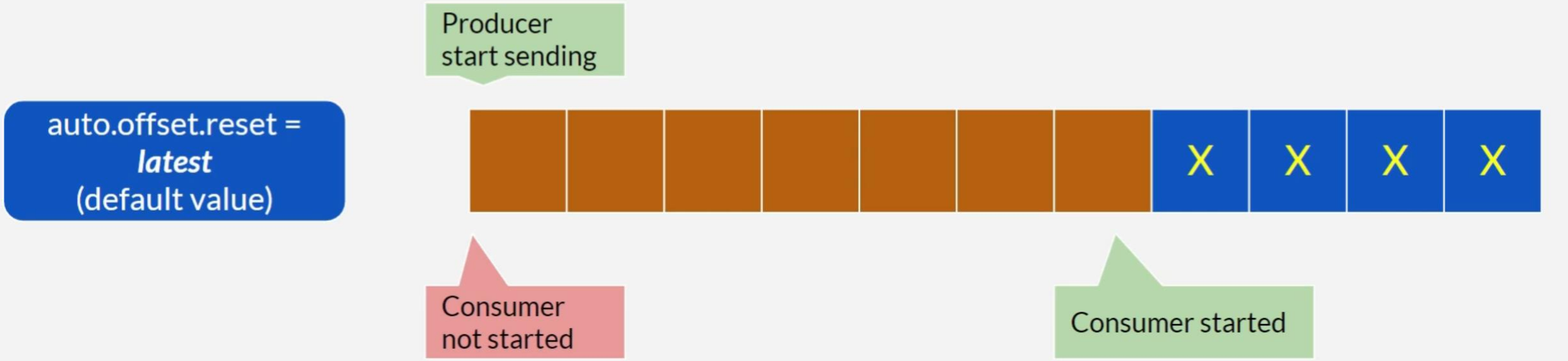
1. Single Node-Single Broker Cluster
2. Single Node-Multiple Broker Cluster
3. **Multiple Nodes-Multiple Broker Cluster**

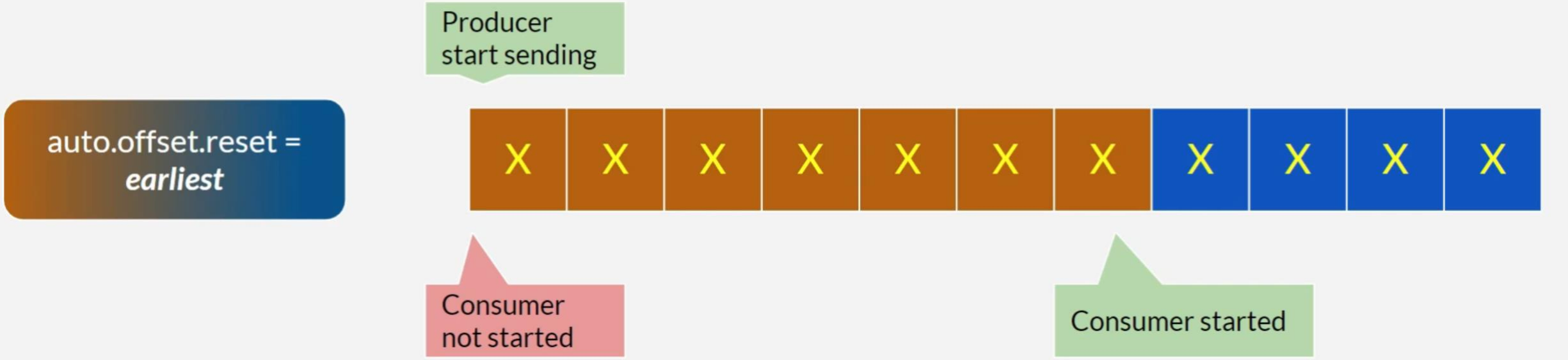


Getting Started with KAFKA & Spring Boot

Producer & Consumer

Consumer Offset on First Run

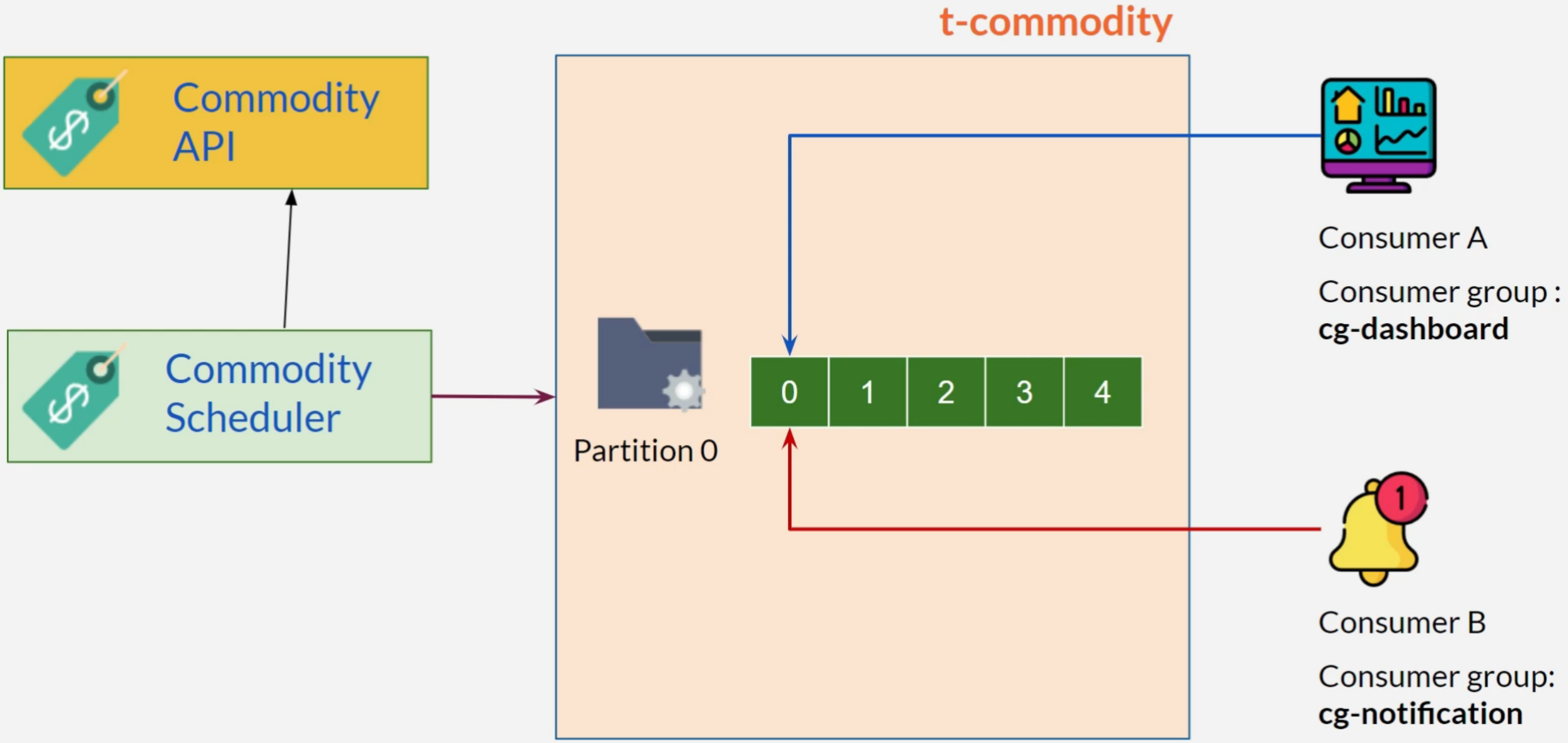


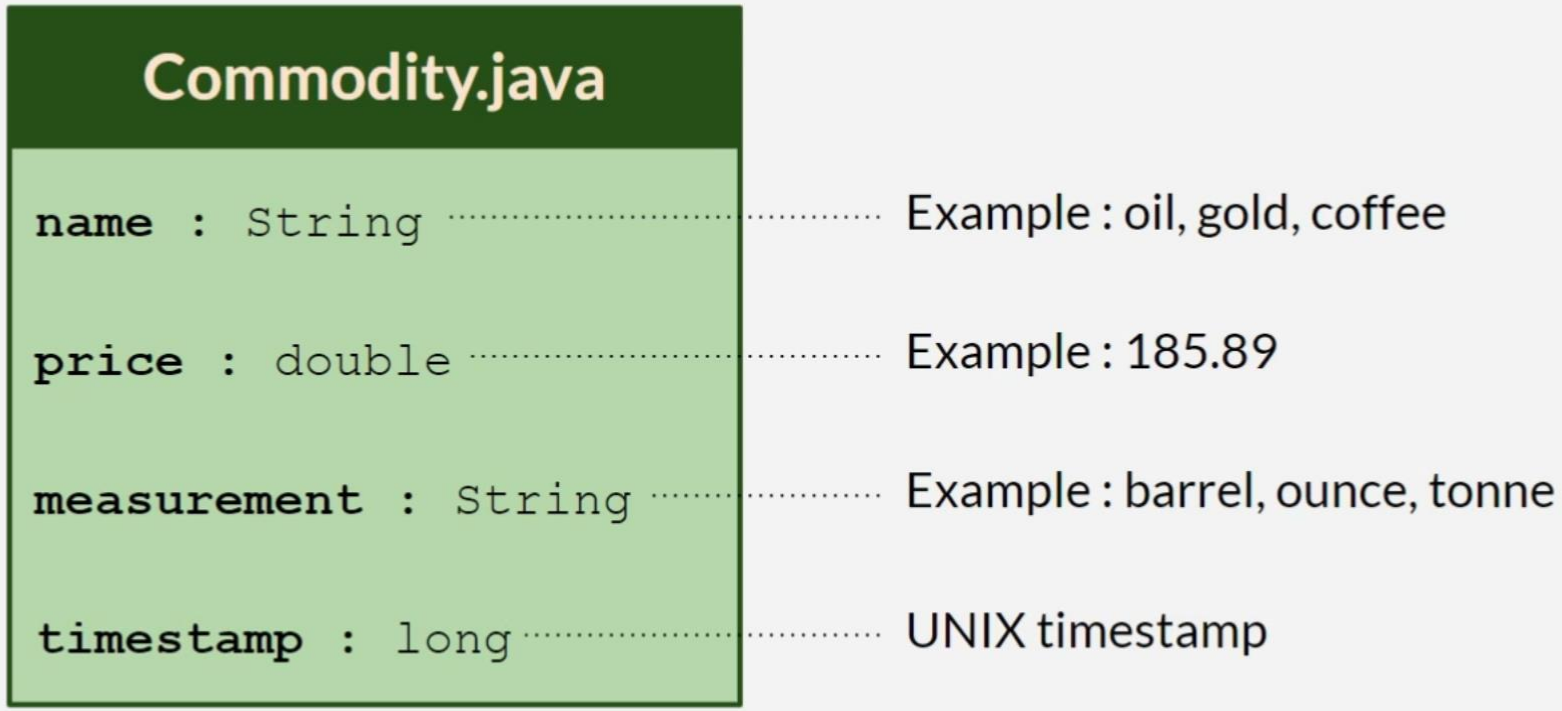


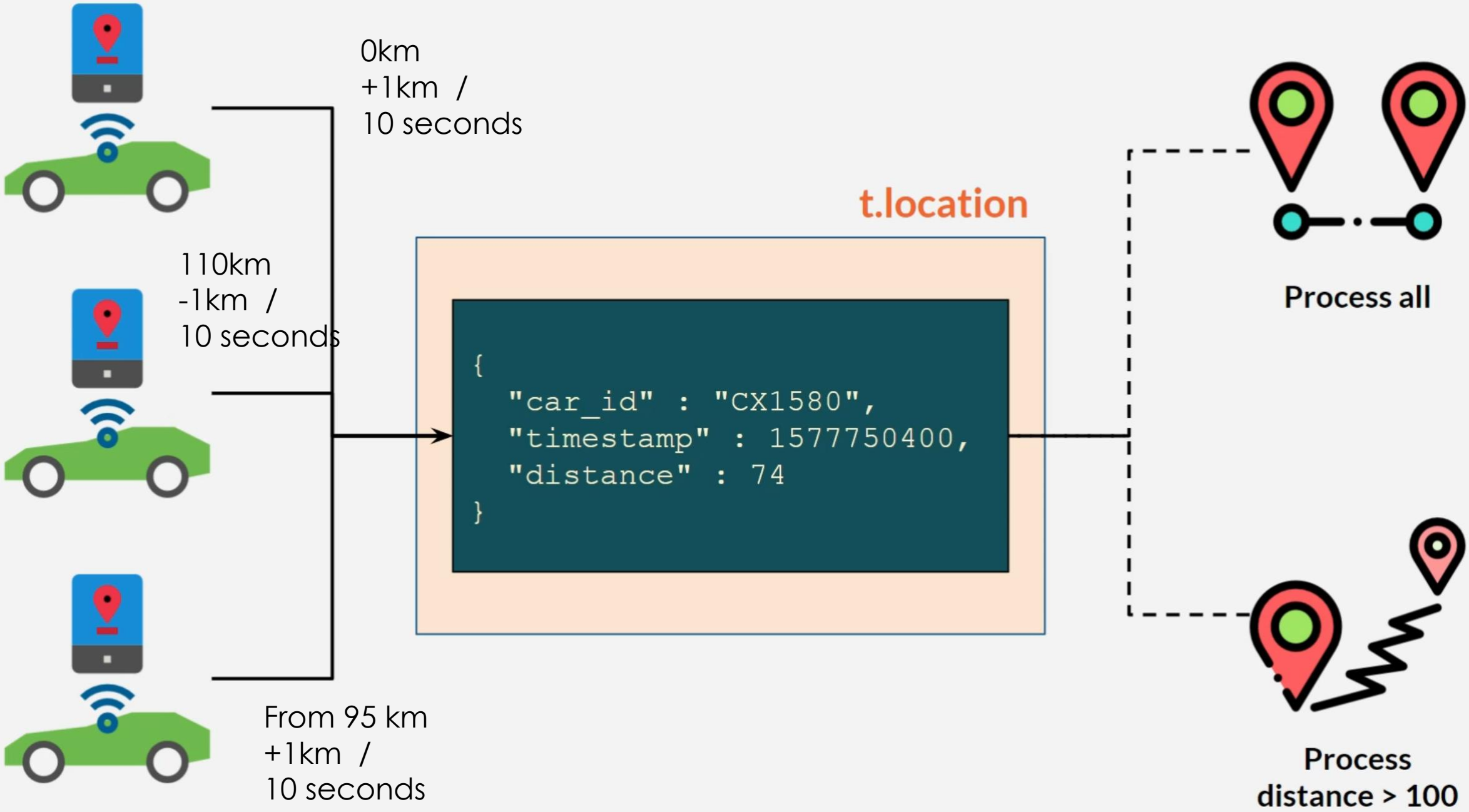
What about deleting partition?

- Delete topic is OK
- Can't delete partition
- Can cause data loss
- Wrong key distribution
- Decrease partition > delete & recreate topic
- Real life usually use kafka native linux

Kafka Schema




Commodity.java
name : String
Example : oil, gold, coffee
price : double
Example : 185.89
measurement : String
Example : barrel, ounce, tonne
timestamp : long
UNIX timestamp


0km
+1km /
10 seconds
110km
-1km /
10 seconds
From 95 km
+1km /
10 seconds
t.location
{
"car_id" : "CX1580",
"timestamp" : 1577750400,
"distance" : 74
}
Process all
Process
distance > 100

Idempotency

Handle Duplicate Message

At-Least-Once Delivery Semantic

- × Message guaranteed to be published
- × Message might be published more than once

Idempotent Consumer

- Duplicate message is OK:
  - Outcome of processing message always same even for duplicate messages
  - Example : update search engine index
- Duplicate message is dangerous:
  - Duplicate transaction
  - Example : create (duplicate) payment
  - Filter out duplicate message(s)

Gray Area

- × It might be dangerous, or not
- × Example : send promotion email to user
- × Can either OK, or bad user experience
- × Technically should filter out duplicate messages
- × Double output is not idempotent

How to Deduplicate?

- Use database for permanent unique value
- Use cache for temporary unique value
- Cache
  - Better performance
  - Automatically remove data after certain time
  - Example : Redis
- Database
  - Might publish duplicate after longer period
  - Virtually unlimited storage

Idempotency

Alternative Approach

No Unique Value

- × No unique value on message
- × Alternative : use object as unique value
  - × Bad idea, if the object contains large data
  - × Eat up cache memory / slow database
- × Derive key from combination of fields
  - × Combination must be unique
- × Don’t use java `hashCode()`
  - × Not guaranteed to be unique for different object

Exception Happens!

Handling Exception

- @KafkaListener Error Handler -

Producer Exception

```
public void send(MyMessage myMessage) {
    try {
        // some logic
        // ...

        kafkaTemplate.send("topic", "key", myJSONString);
    } catch (Exception e) {
        // handle exception
    }
}
```

What We Will Do

- Spring default : log exception
- Able to implement our own error handler
- Scenario
  - Publish food order
  - Exception : consume invalid food amount

What We Will Do

- Global error handler : works for all kafka consumers
- Error handler on Spring container
- Scenario
  - Publish random number
  - Consume odd number throws exception
  - Handle using global error handler

Error Handler Flow

FoodOrderConsumer

FoodOrderErrorHandler

re-throw

GlobalErrorHandler

SimpleNumberConsumer

Why Retry Mechanism?

- Spring Kafka will log failed messages
- Our own ErrorHandler
- Case:
  - Service temporarily unavailable
  - Retry hit service without trigger from consumer
  - Retry for n times only

Scenario

- Topic: t-image, 2 partitions
- Publish message represents image
- Consumer
  - Simulate API call to convert image
  - Simulate failed API call
  - Retry after 10 seconds
  - Retry 4 times

Blocking Retry

Message 1	Message 2	Message 3 (blocked)

Encounter error

Retry

Blocking Retry

Message 1	Message 2	Message 3

Encounter error

Retry

or

Blocking Retry

- Good when message must be processed in sequence
- Drawback : process halted (might bottleneck)
- Mitigation : retry "just enough"
- Block only on partition which has error
- Other partition (no error) keep consuming

Dead Letter Topic

- Message process keep failing
  - Non technical issue
  - Permanent technical issue
  - Process message differently
- Send such message to dead letter topic
- Dead letter record

DeadLetterPublishingRecoverer

In this lecture:

- Send to custom topic
- Sendt to custom partition

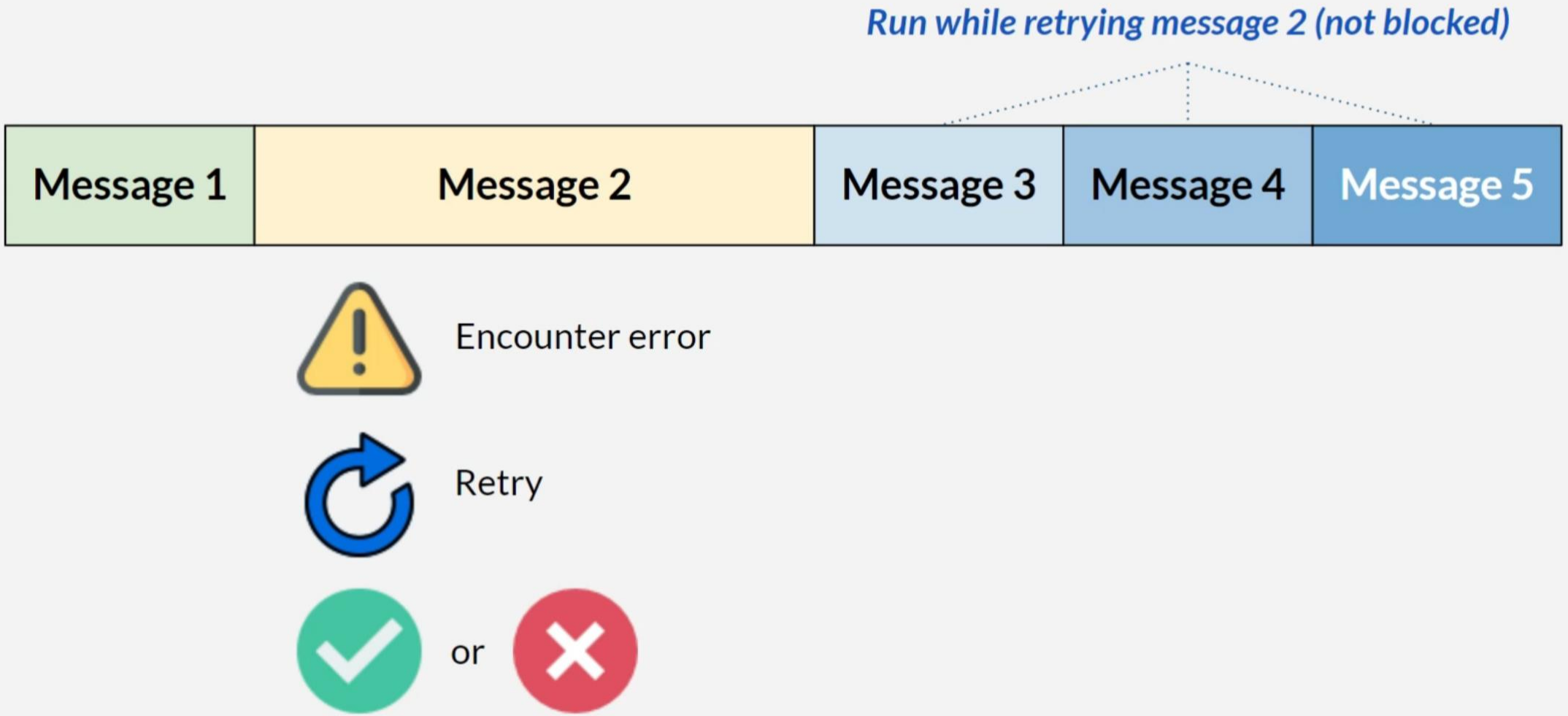
Scenario

- × Publish to t-invoice
- × If amount is less than one, throw exception
- × Retry 5 times
- × After 5 failed retry attempts, publish to t-invoice-dead
- × Another consumer will consume from t-invoice-dead

Non Blocking Retry

- × Message can be processed independently
- × Message retry still needed
- × Spring non blocking retry

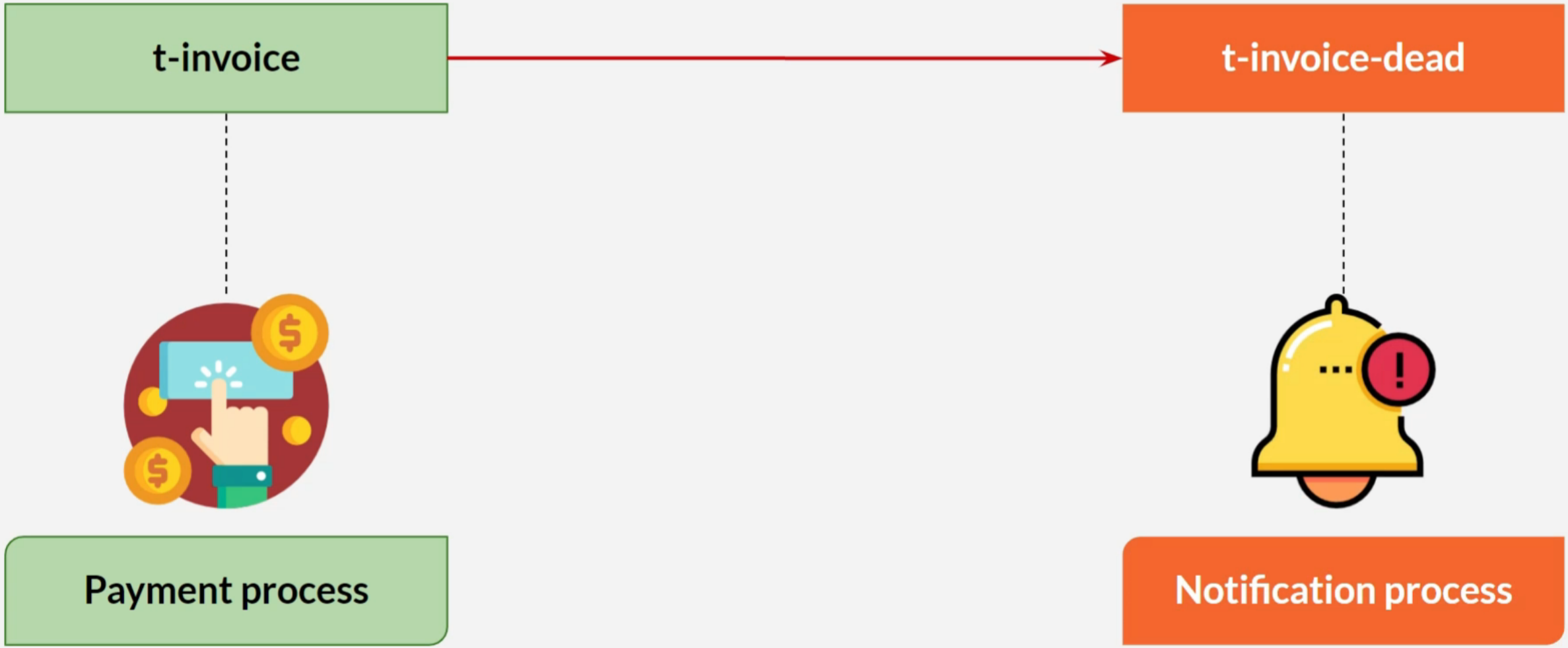
Non Blocking Retry



Dead Letter Records

- Dead letter record : messages sent to DLT
- Just regular topic / message
- Create consumer to listen from DLT

Process Dead Letter Record



Payment app (Android / iOS / web)

REST API

Payment app server

Must live 24/7

Must live 24/7

Publish

kafka

Consume

Must NOT consume during certain time window

General Ledger app server

Scheduler : summarize daily ledger

REST API

Auto off / on

Scheduler / on demand : report

Kafka Consumer : Ledger entry (turn off)

Payment app (Android / iOS / web)

REST API

Payment app server

Must live 24/7

Must live 24/7

Publish

kafka

Consume

Must NOT consume during certain time window

General Ledger app server

Scheduler : summarize daily ledger

REST API

Auto off / on

Scheduler / on demand : report

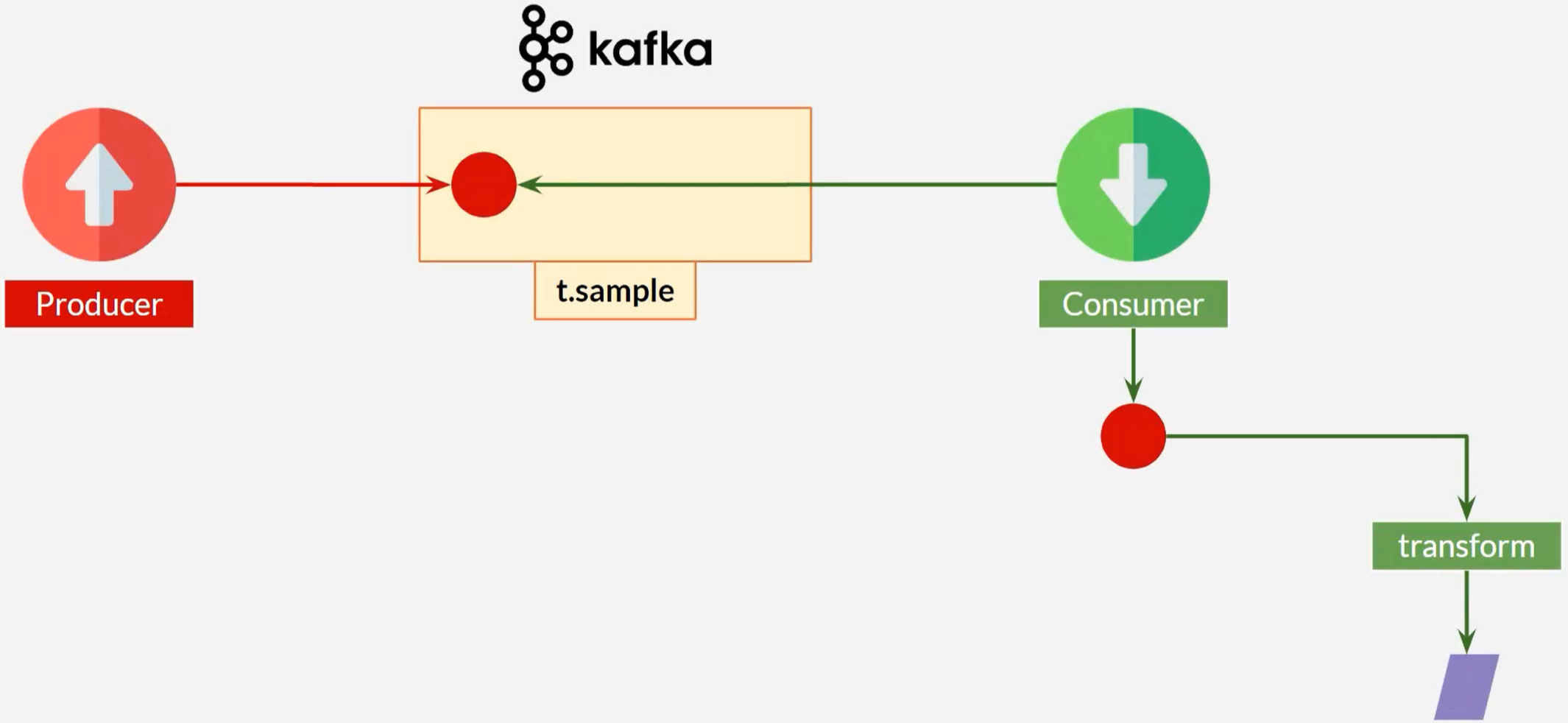
Kafka Consumer : Ledger entry (turn off)

Kafka Stream

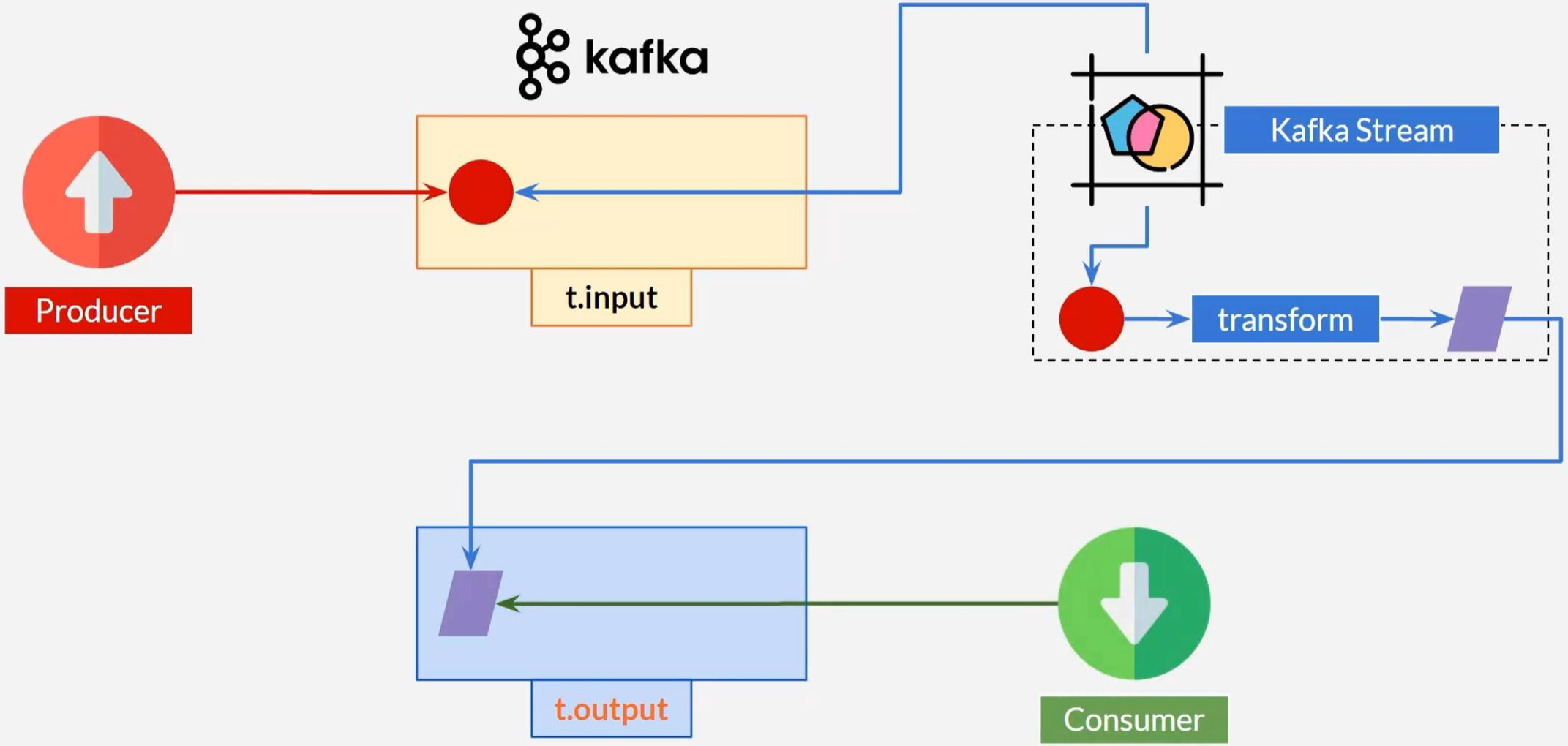
Kafka Stream

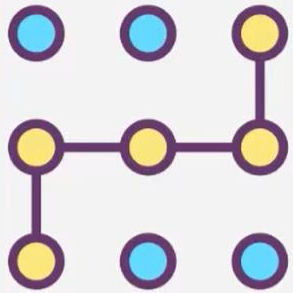
- Stream processing framework
- Released on 2017
- Alternative for Apache Spark, NIFI or Flink
- Stream & stream processing?

Data Transformation



Kafka Stream





Pattern

Spring Kafka

Kafka Stream

Employees

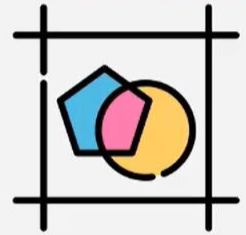
Reward

Spring Kafka

Kafka Stream

Kafka Stream

Spring Kafka



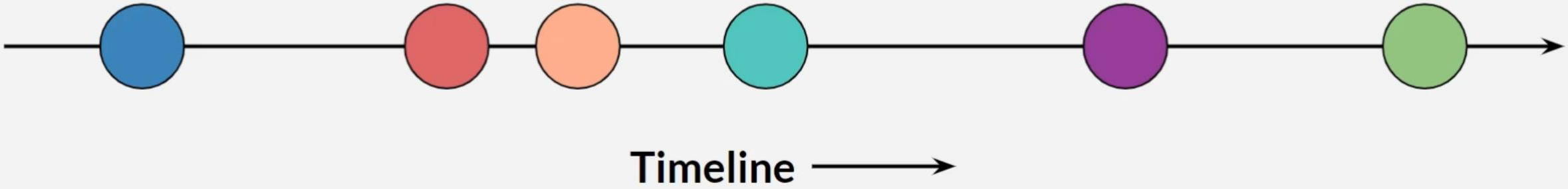
Stream

Storage

Spring Kafka

Kafka Stream

Data Stream



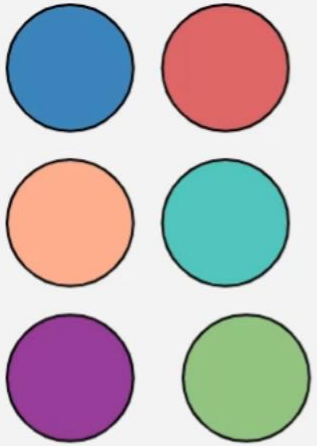
Each circle represents a data

Endless

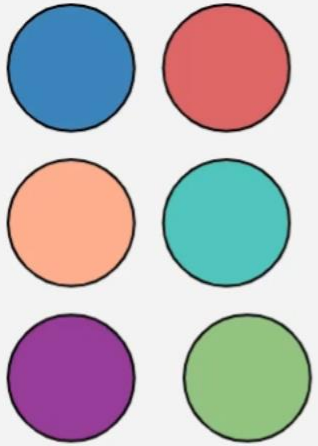
Data (event) is immutable

Can be replayed

Data Processing



Data Processing



- Transformation
- Calculation
- Aggregation
- Combination
- Filter
- etc

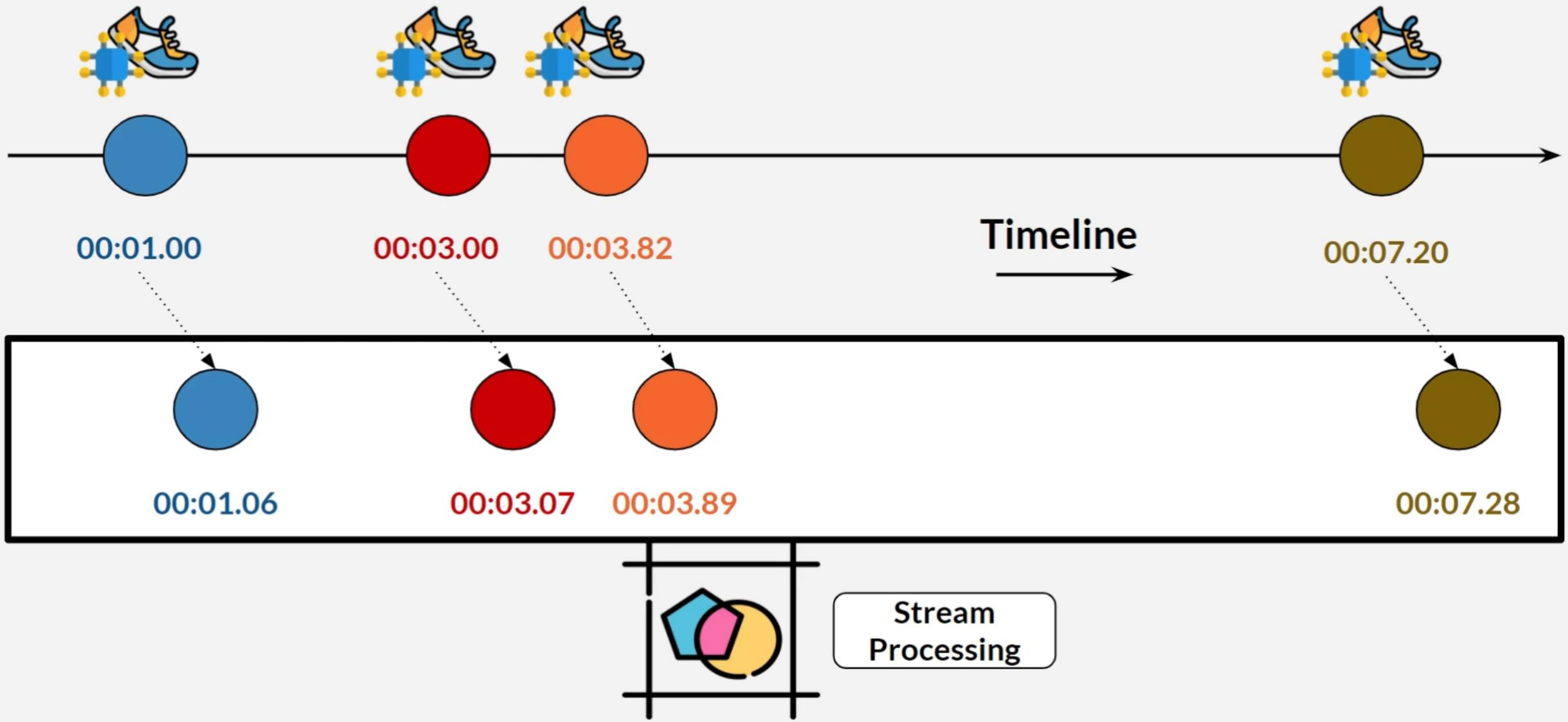
Everyday at 23:00

Every 30 min

Every 1 second

Micro Batching

Stream Processing



Yes To Stream Processing

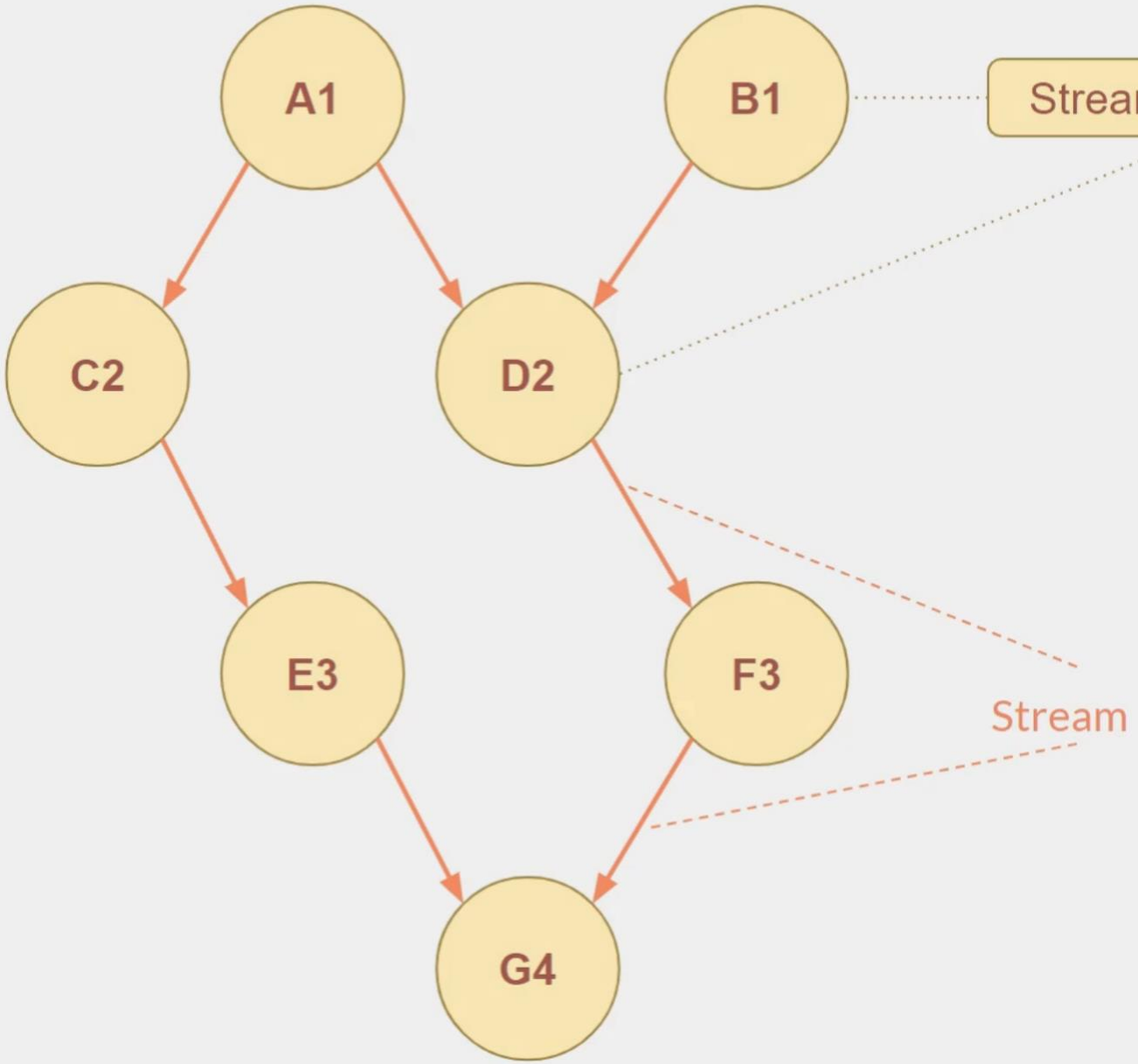
- Yes, When:
  - (relatively) fast data flow
  - Application need to response quick to most recent data

- Example
  - Marathon
  - Credit card fraud
  - Stock trading
  - Log analysis

No To Stream Processing

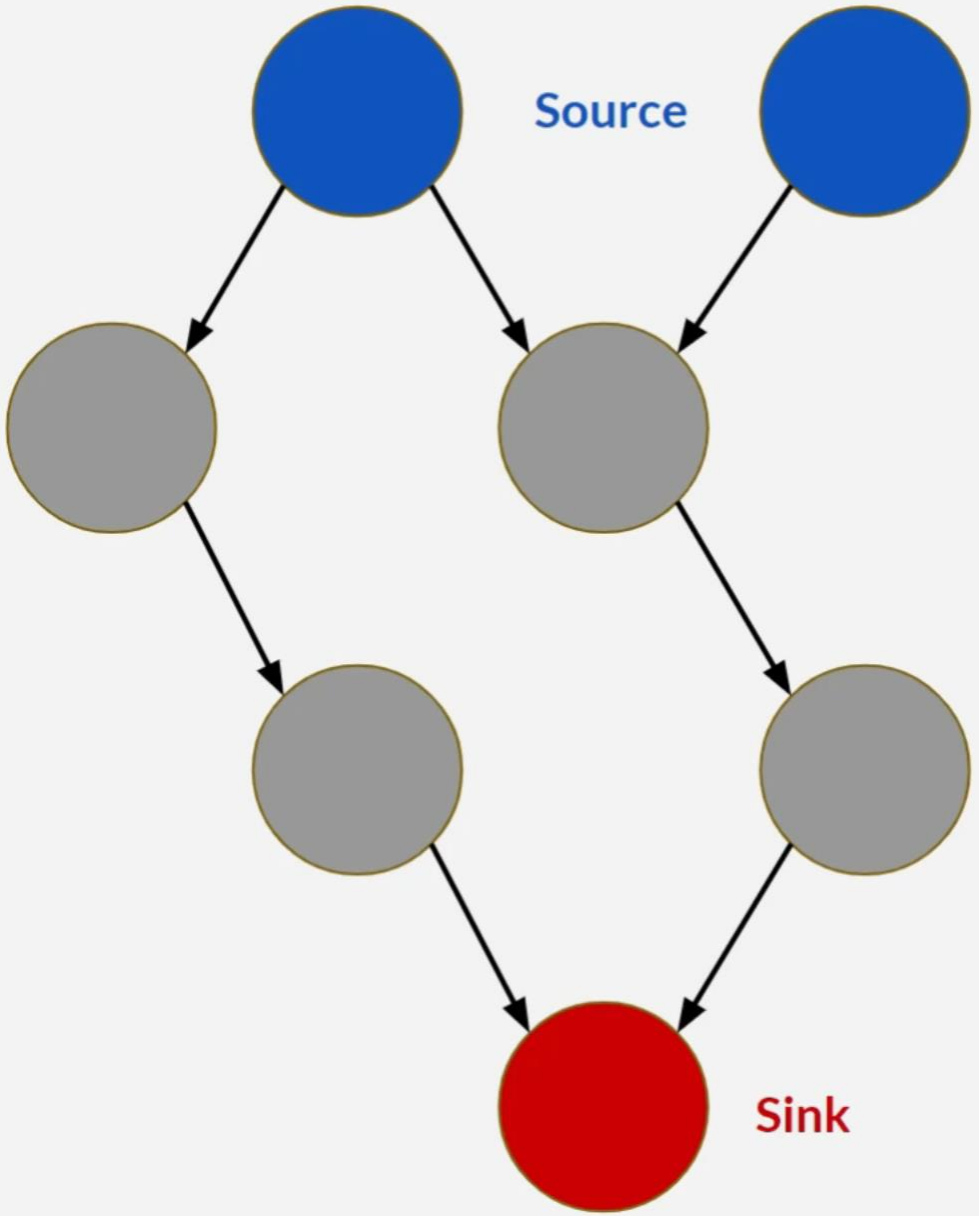
- Example
  - Daily Interest
  - Forecasting

Topology / DAG



- Stream processor process incoming data stream
- Stream processor can create new output stream
- Data flows from parent to child, not vice versa
- Parent = upstream
- Child = downstream
- Child stream processor can define another child(ren)

Kafka Stream Topology



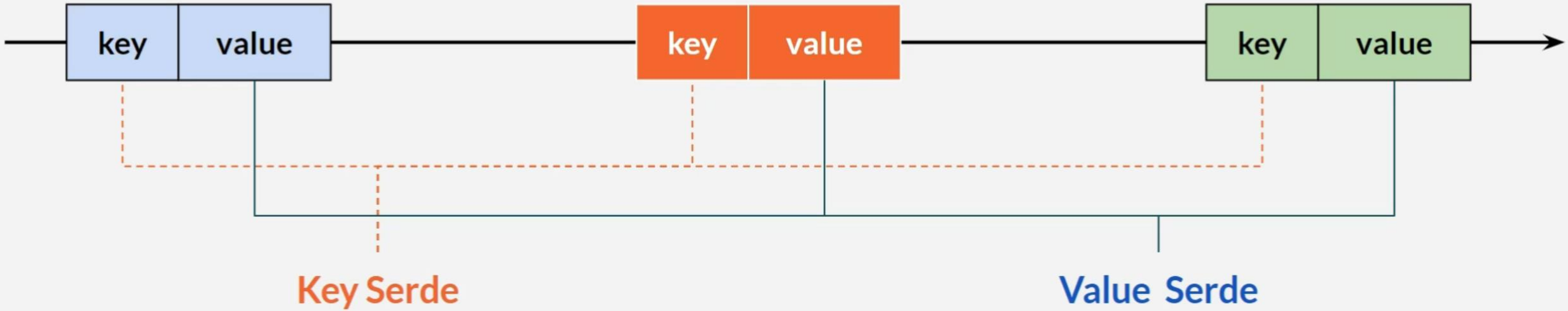
Source Processor

- Does not have upstream
- Consumes from one or more kafka topics
- Forwarding data to downstream

Sink Processor

- Does not have downstream
- Receive data from upstream
- Send data to specified kafka topic

Serde (Serializer / Deserializer)




```
Serdes.String()
Serdes.Long()
Serdes.ByteArray()
...
```

```
new JSONSerde<T>()
```

```
class CsvSerde<T> implements Serde<T>
```

Stream & Table

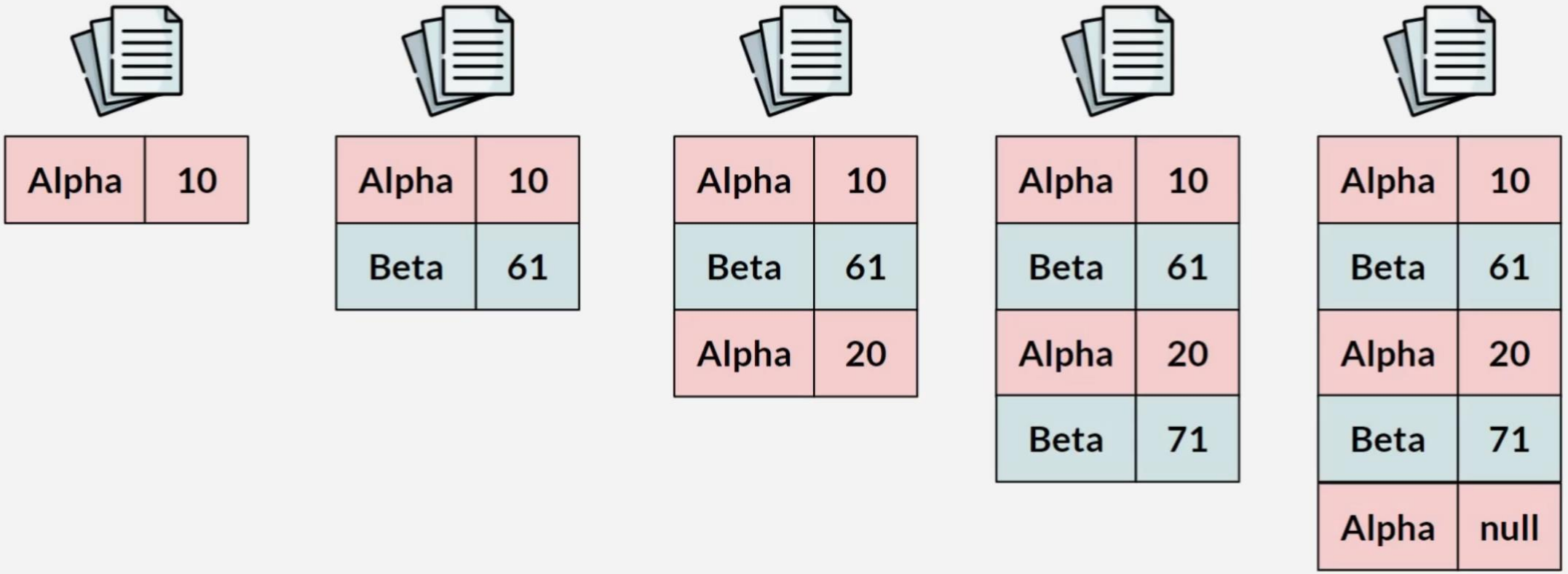
`KStream`
Ordered sequence of messages
Unbounded
Inserts data

Stream

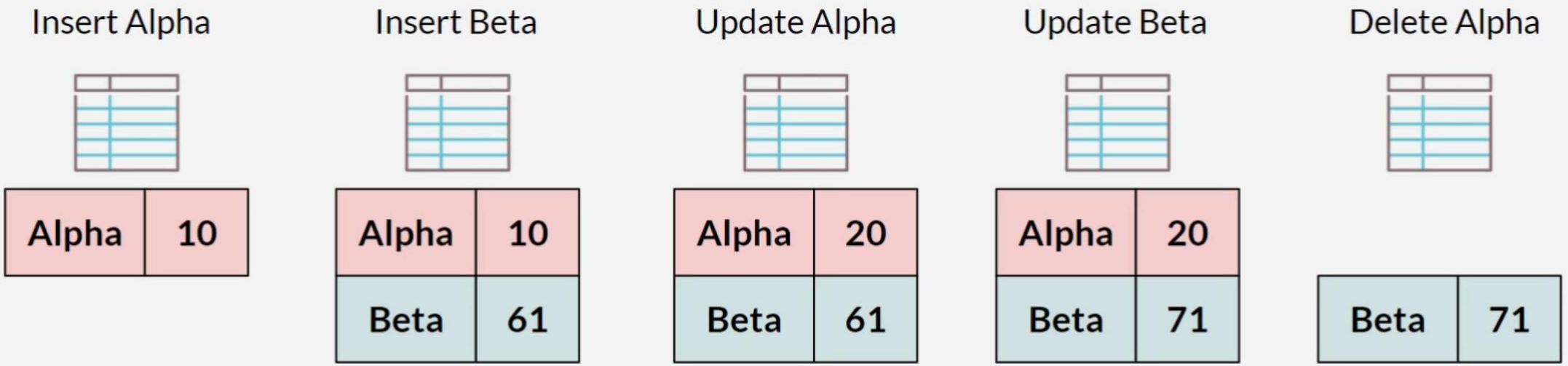
Table

`KTable`
Unbounded
Upserts data : insert or update based on key
Delete on null value
Analogy : database table

KStream : Inserts Data



KTable : Upserts / Delete Data



When to Use KStream / KTable?

- KStream
  - Topic not log-compacted
  - Data is partial information
- KTable
  - Topic is log-compacted
  - Data is self sufficient

Log Compaction

- Kafka admin process
- Keep at least latest value & delete the older
- Based on record key
- Useful if we need latest snapshot
- Configure when creating topic

Log Compaction

Topic : T, partition 0

Offset	0	1	2	3	4	5	6	7	8	9	10
Key	Alpha	Sigma	Beta	Alpha	Omega	Alpha	Delta	Delta	Beta	Omega	Omega
Value	10	180	20	11	240	12	40	40	21	241	242

Log compaction

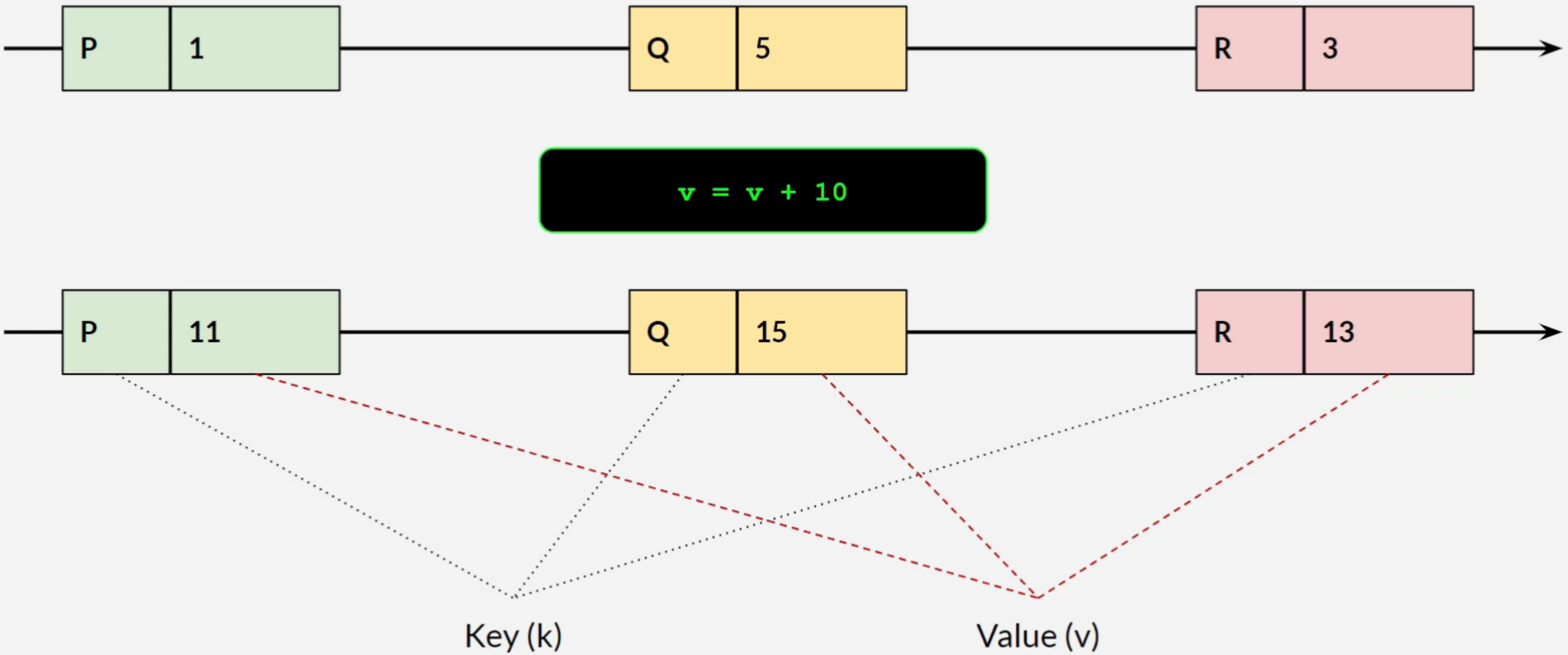
Topic : T, partition 0

Offset	1	5	7	8	10
Key	Sigma	Alpha	Delta	Beta	Omega
Value	180	12	41	21	242

Log Compaction

- Keep the order
- Not change offset
- Not duplication validator
- Can fail

Diagram



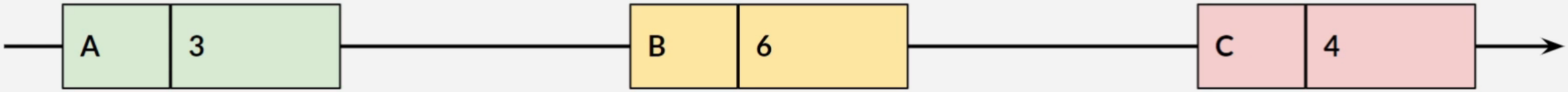
Intermediate & Terminal Operation

- Intermediate
  - KStream -> KStream
  - KTable -> Ktable
- Terminal
  - KStream -> void
  - KTable -> void
  - “Final” operation

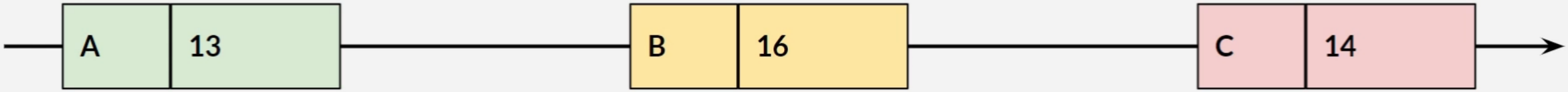
Reminder : Key & Partition

- Key & partition is related
- Partition according to key
- Repartition
  - From partition A to partition X

mapValues

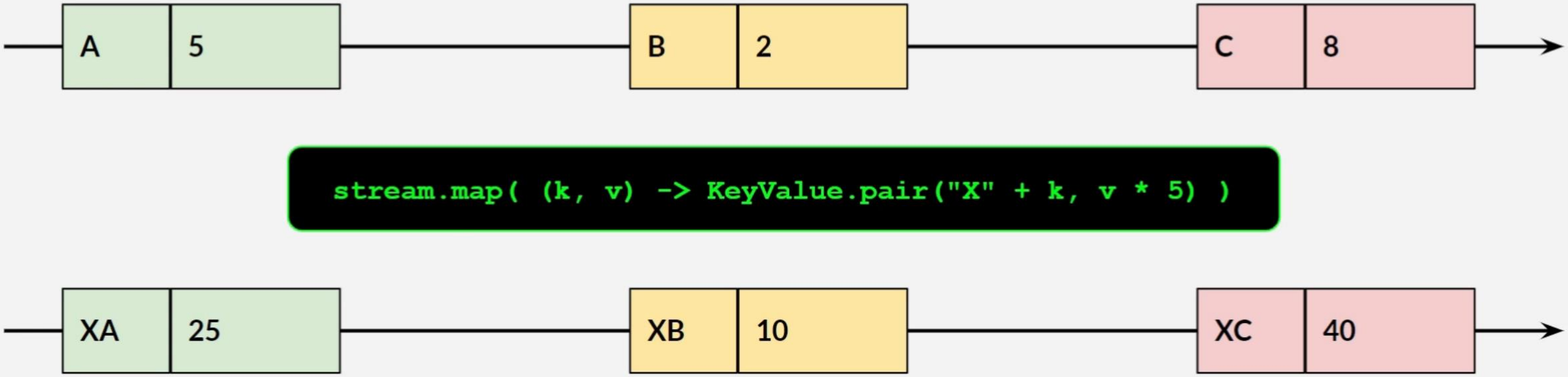



```
stream.mapValues(v -> v + 10)
```



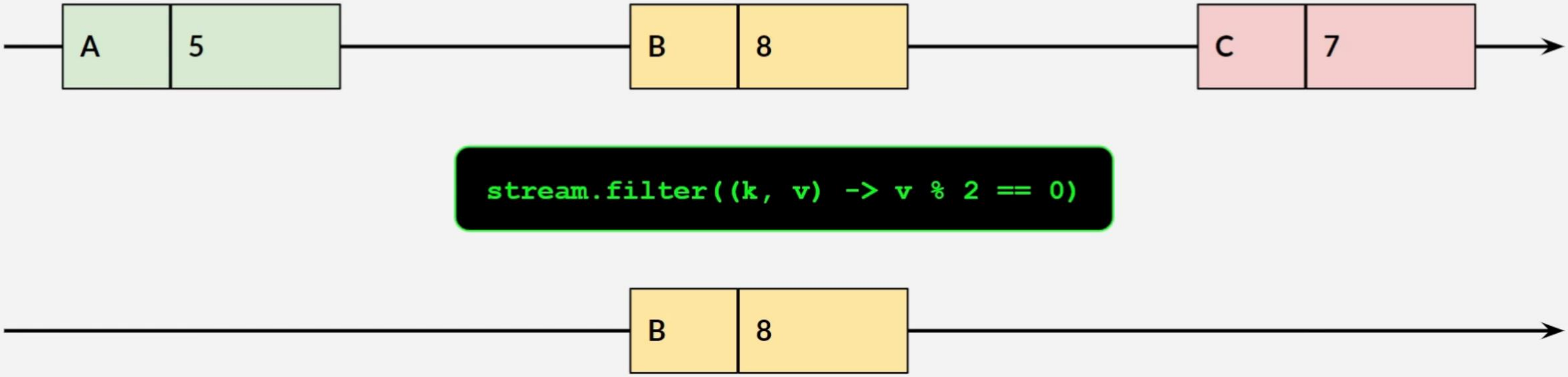
- Takes one record, produces one record
- Does not change key
- Affect only value
- Not trigger repartition
- Intermediate operation
- KStream & Ktable

map



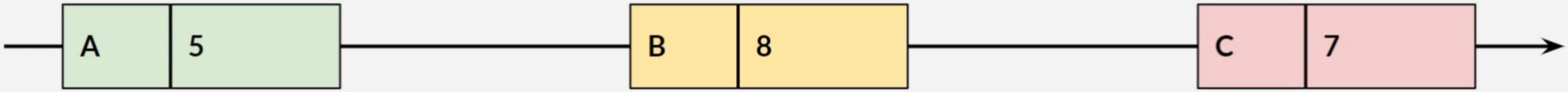
- Takes one record, produces one record
- Change key
- Change value
- Trigger repartition
- Intermediate operation
- KStream

filter

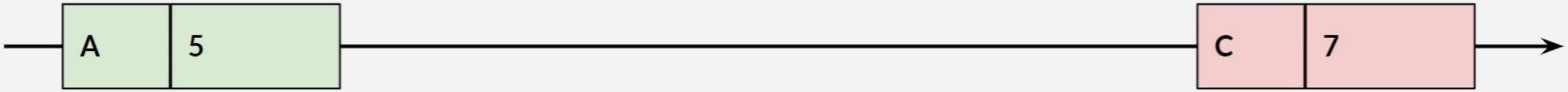


- Takes one record, produces one or zero record
- Produce record that match condition
- Does not change key or value
- Not trigger repartition
- Intermediate operation
- KStream & Ktable

filterNot

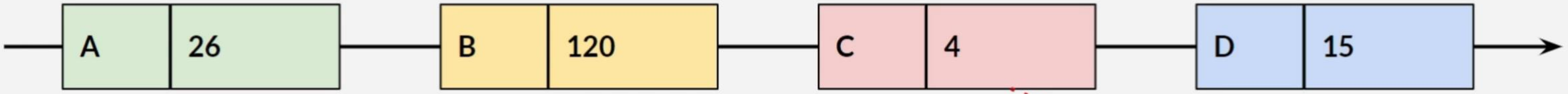



```
stream.filterNot((k, v) -> v % 2 == 0)
```



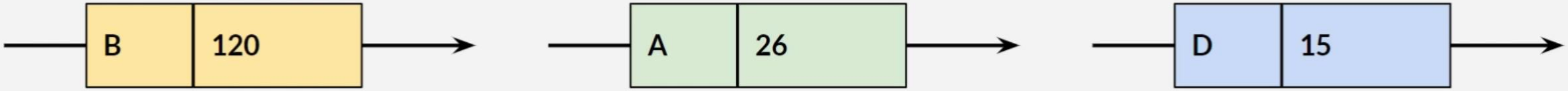
- Takes one record, produces one or zero record
- Produce record that NOT match condition
- Does not change key or value
- Not trigger repartition
- Intermediate operation
- KStream & KTable

branch




```
var arrayStream = stream.branch(
    (k, v) -> v > 100,
    (k, v) -> v > 20,
    (k, v) -> v > 10
)
```

Dropped, no match



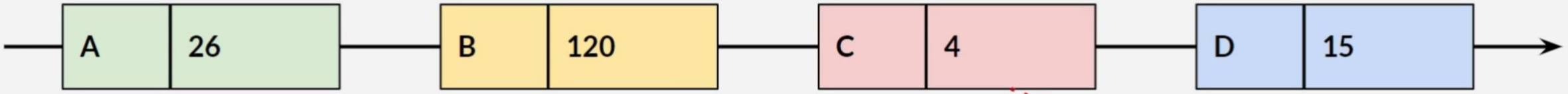
arrayStream[0] (> 100)

arrayStream[1] (> 20)

arrayStream[2] (> 10)

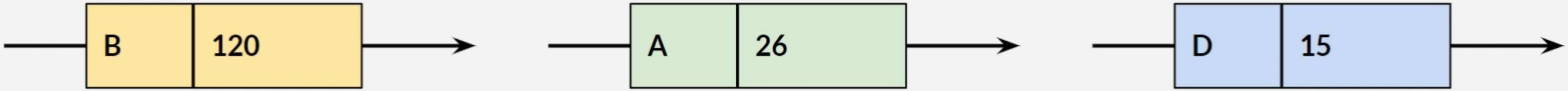
- Split stream based on predicates
- Evaluate predicate in order
- Record only placed once on first match, drop unmatched record
- Returns array of stream
- Intermediate operation
- KStream

split & branch




```
stream.split().
    branch((k, v) -> v > 100, Branched.withConsumer(ks -> ks.to("t-x"))
    .branch((k, v) -> v > 20, Branched.withConsumer(ks -> ks.to("t-y"))
    .branch((k, v) -> v > 10, Branched.withConsumer(ks -> ks.to("t-z"))
)
```

Dropped, no match



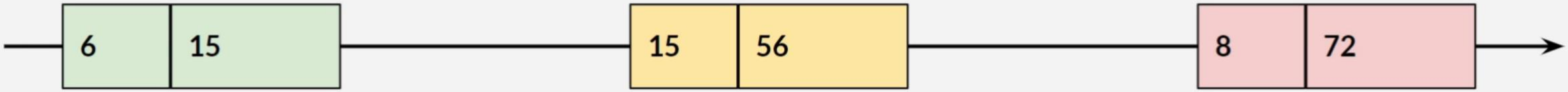
KStream 1 (> 100)

KStream 2 (> 20)

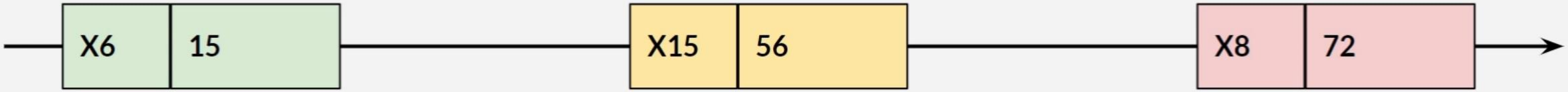
KStream 3 (> 10)

- Split stream based on predicates
- Evaluate predicate in order
- Record only placed once on first match, drop unmatched record
- Get KStream for each branch
- split() returns final BranchedKStream
- Each branch returns KStream to be processed further

selectKey

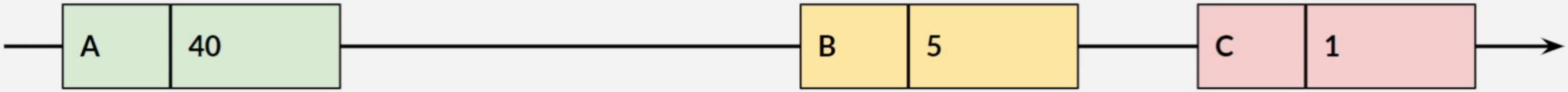



```
stream.selectKey((k, v) -> "X" + k)
```

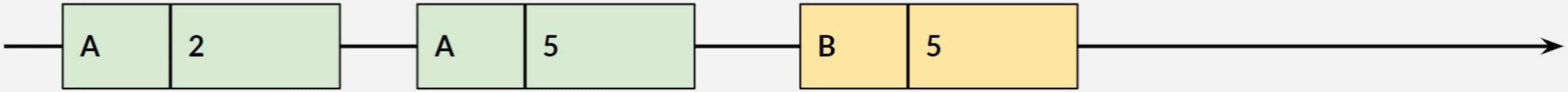


- Takes one record, produces one record
- Set / replace record key
- Possible to change key data type
- Trigger repartitioning
- Value not change
- Intermediate operation
- KStream

flatMapValues

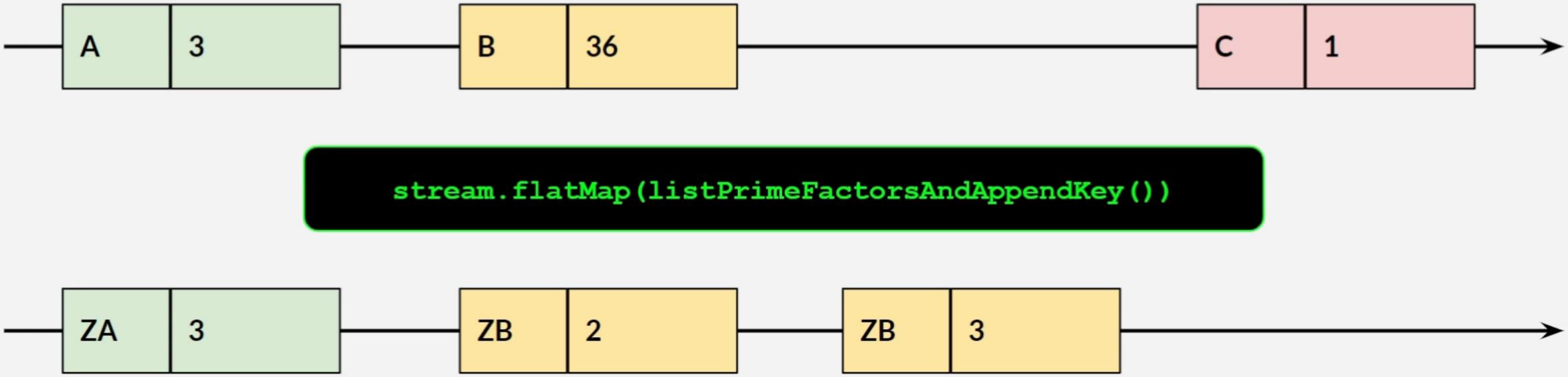



```
stream.flatMapValues(listPrimeFactors())
```



- Takes one record, produces zero or more record
- Does not change key
- Affect only value
- Not trigger repartition
- Intermediate operation
- KStream

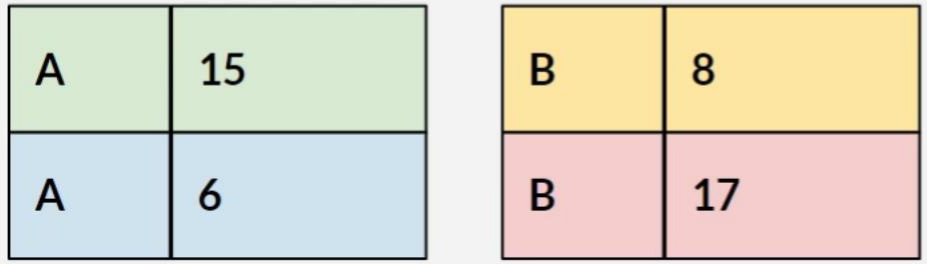
flatMap



- Takes one record, produces zero or more record
- Change key
- Change value
- Trigger repartition
- Intermediate operation
- KStream

groupByKey

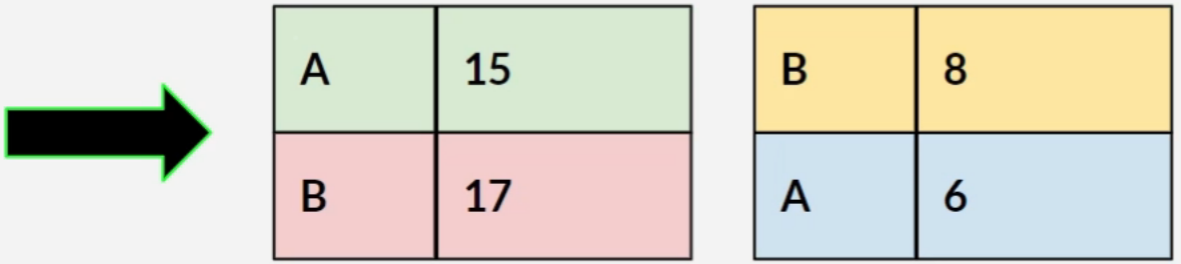

```
stream.groupByKey()
```



- Intermediate operation
- Group records by existing key
- KStream

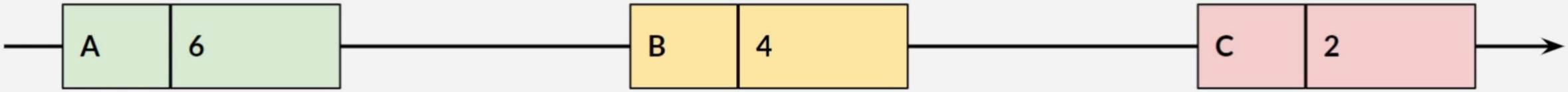
groupBy


```
stream.groupBy((k, v) -> v % 2)
```



- Intermediate operation
- Group records by new key
- KStream & KTable

forEach




```
stream.forEach((k,v) -> insertToDatabase(v))
```

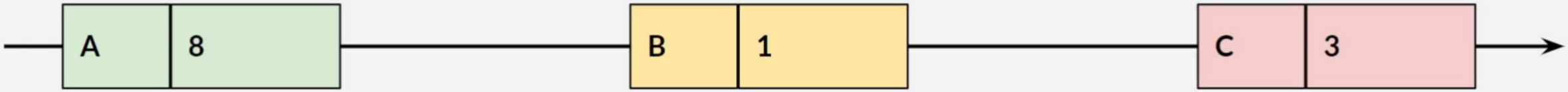

6


6
4


6
4
2

- Terminal operation
- Takes one record, produces none
- Produces side effect
- Side effect not tracked by kafka
- KStream & KTable

peek




```
stream.peek((k,v) -> insertToDatabase(v)).[nextProcessor]
```

Next processor

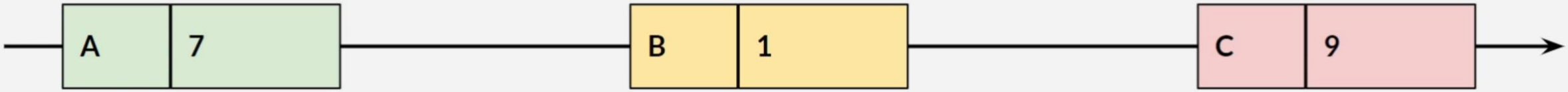
8

8
1

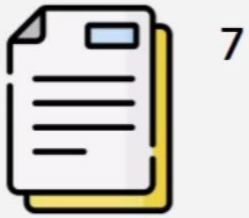
8
1
3

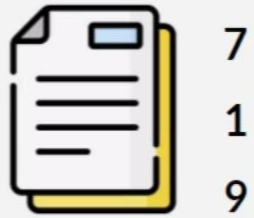
- Produces unchanged stream
- Produces side effect
- Side effect not tracked by kafka
- Result stream can be processed further
- Intermediate operation
- KStream

print



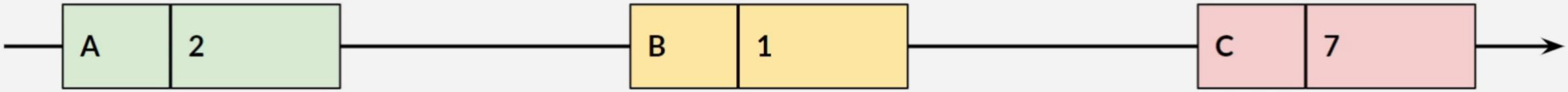

```
stream.print(Printed.toSysout())
```





- Terminal operation
- Print each record
- Something like kafka console consumer
- Print to file or console
- KStream

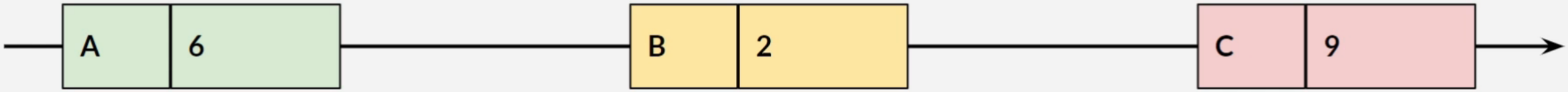
to




```
stream.to("output-topic")
```

- Terminal operation
- Write stream to destination topic
- KStream

through

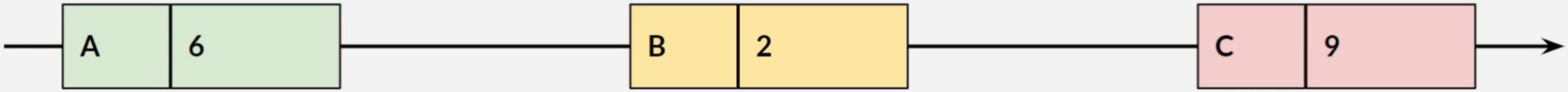



```
stream.through("output-topic").[nextProcessor]
```

Next processor

- Intermediate operation
- Write stream to destination topic
- Continue record processing
- KStream

repartition



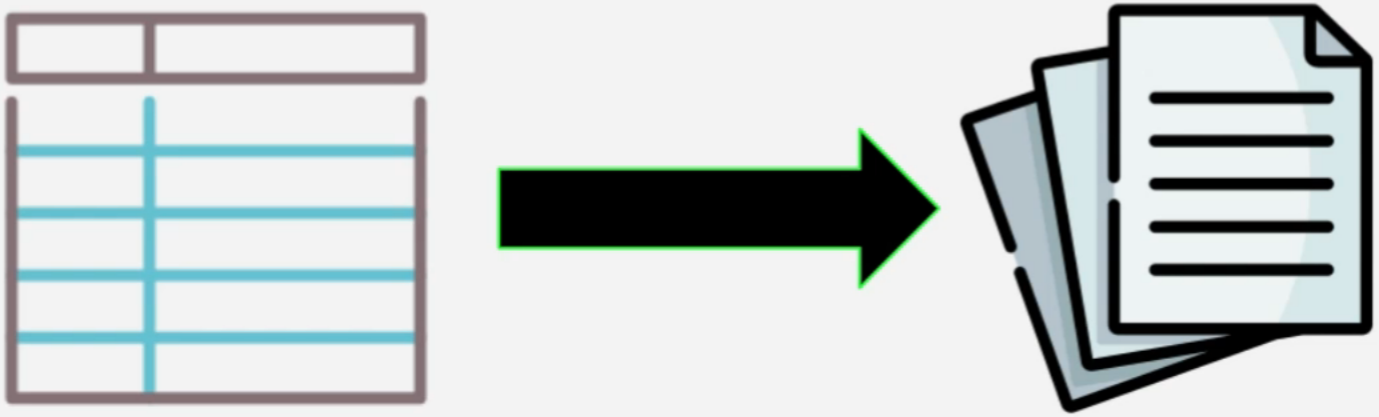

```
stream.repartition().[nextProcessor]
stream.repartition(Repartitioned.as("output-topic")).[nextProcessor]
```

Next processor

- Intermediate operation
- Write stream to destination topic
- Continue record processing
- repartition() output-topic name is fixed
- Topic only for kafka internal use

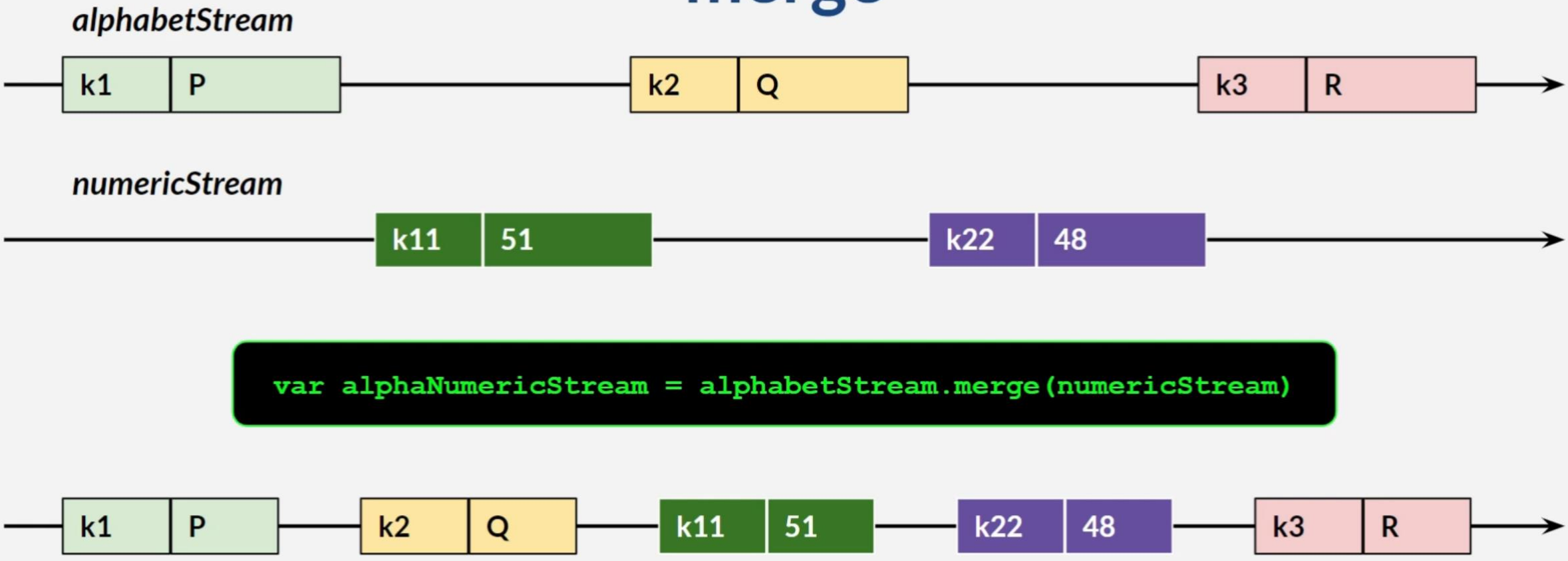
toStream

```
table.toStream()
```



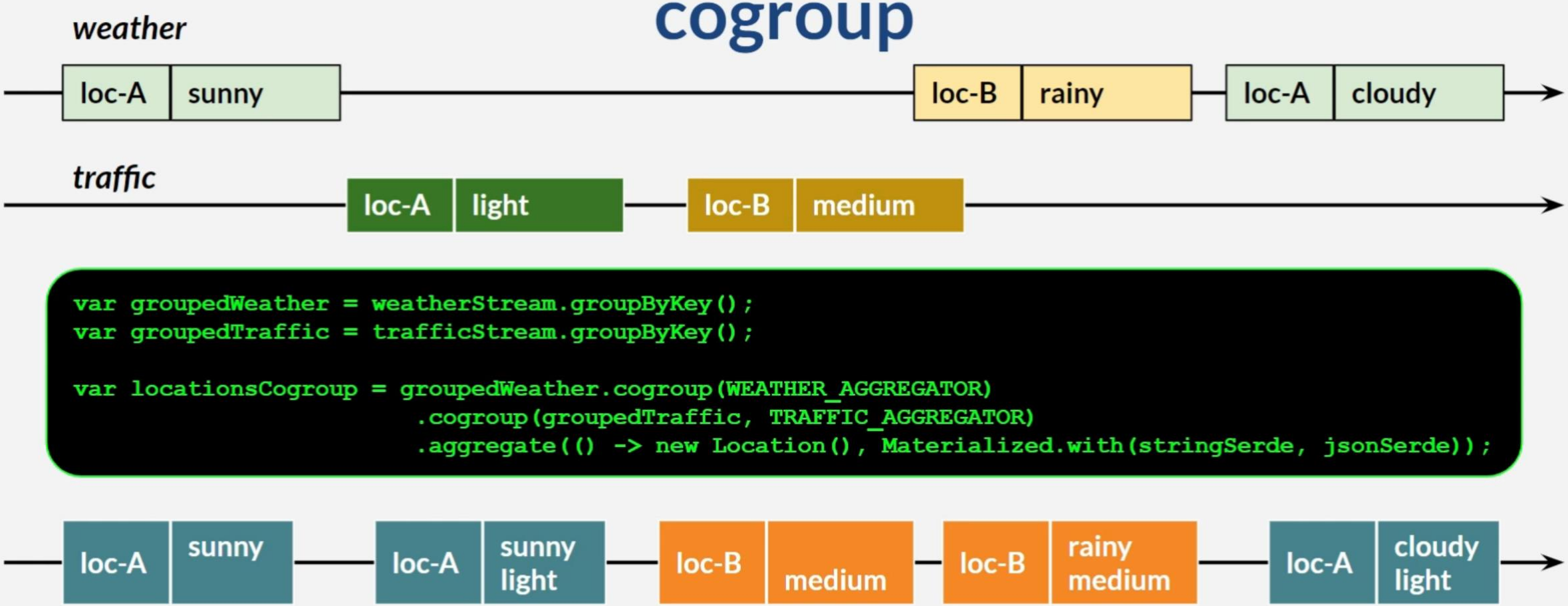
- KTable
- Intermediate operation
- Convert KTable to KStream

merge



- Merge two streams into one new stream
- No ordering guarantee on resulting stream
- Intermediate operation
- KStream

cogroup



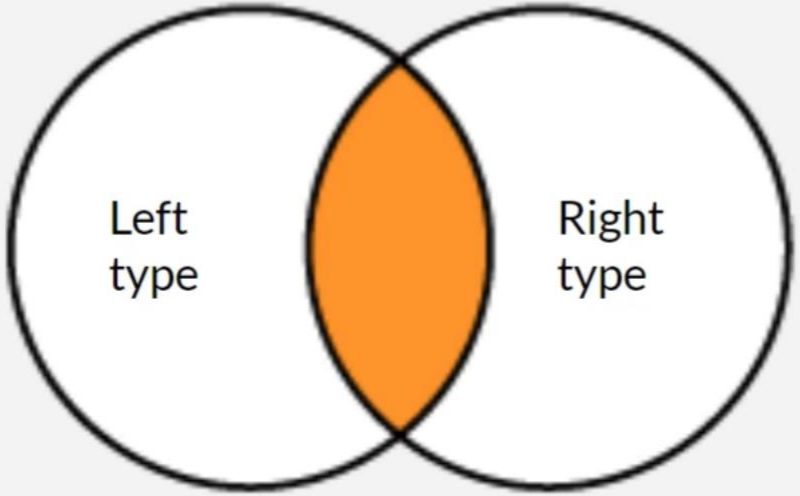

```
var groupedWeather = weatherStream.groupByKey();
var groupedTraffic = trafficStream.groupByKey();

var locationsCogroup = groupedWeather.cogroup(WEATHER_AGGREGATOR)
                      .cogroup(groupedTraffic, TRAFFIC_AGGREGATOR)
                      .aggregate(() -> new Location(), Materialized.with(stringSerde, jsonSerde));
```


- Intermediate operation
- KStream
- Need aggregator for each cogroup
- This sample : String key, JSON value
- Difference with merge()

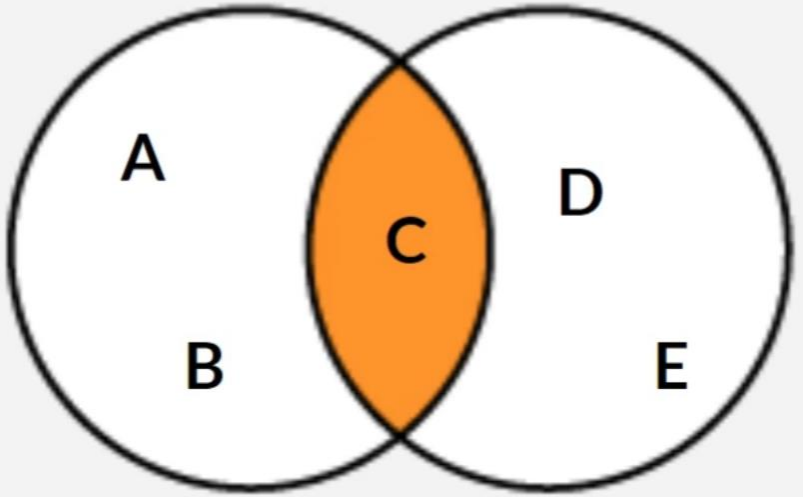
Windowed KStream-KStream

- KStream : unbounded, unlimited data
- KStream / KStream join need constraints
- Otherwise both KStream will be scanned when new data arrives -> huge performance cost
- Use window for constraint

Inner Join



example

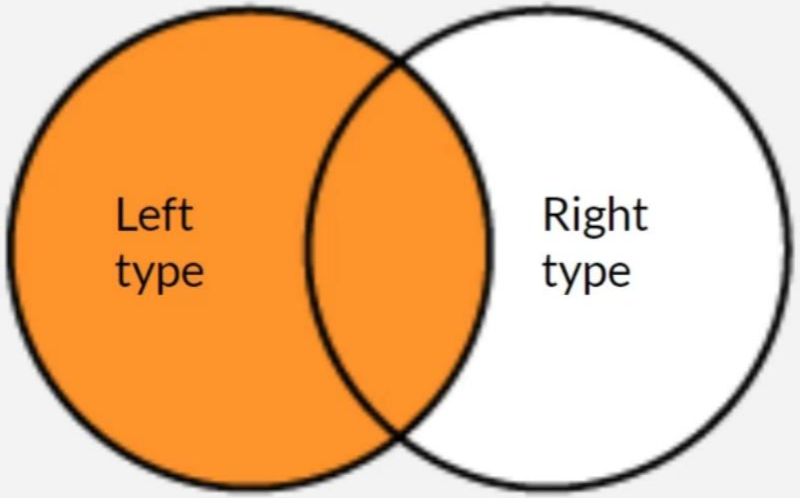


Left (Primary)	Right (Secondary)
KStream	KStream
KTable	KTable
KStream	KStream
KStream	GlobalKTable

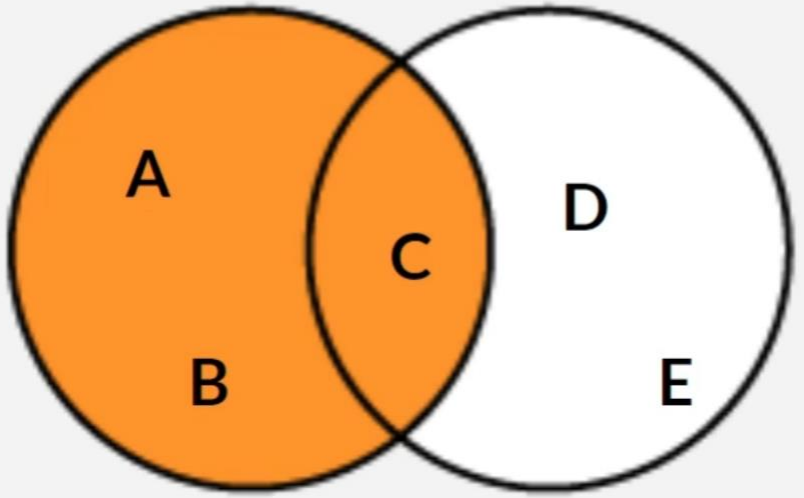
(left value, right value)

Result is:
(C, C)

Left Join



example

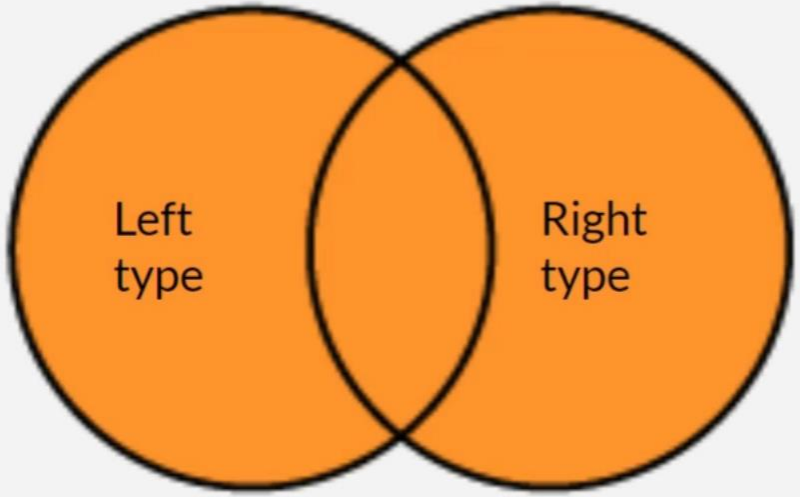


Left (Primary)	Right (Secondary)
KStream	KStream
KTable	KTable
KStream	KStream
KStream	GlobalKTable

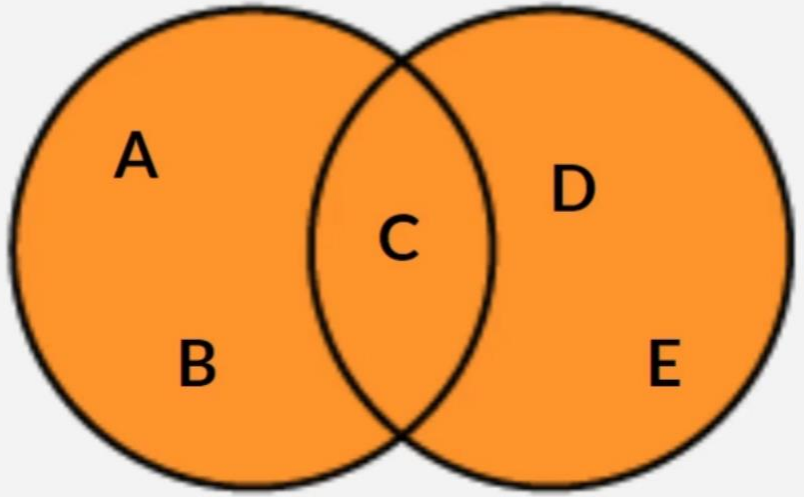
(left value, right value)

Result are:
(A, null)
(B, null)
(C, C)

Outer Join



example



Left (Primary)	Right (Secondary)
KStream	KStream
KTable	KTable

(right value, left value)

Result are:
(A, null)
(B, null)
(C, C)
(null , D)
(null, E)

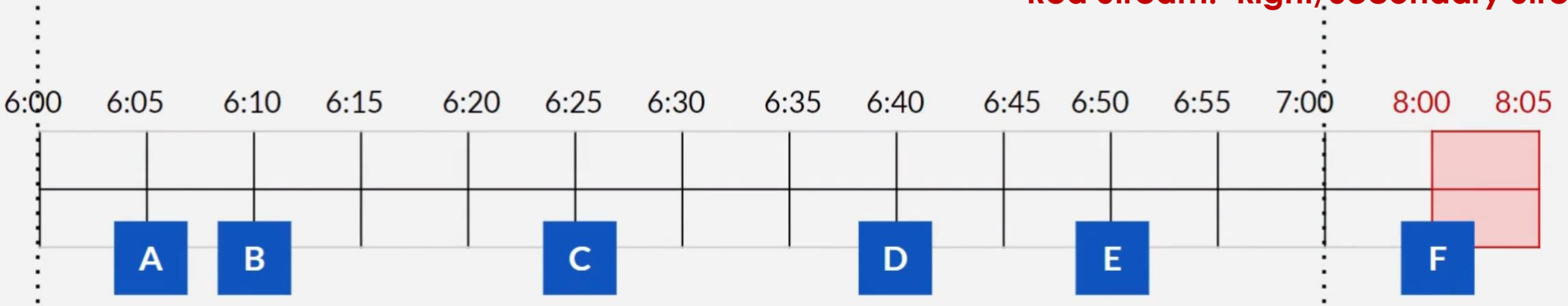
Join Window

- Class `JoinWindows`
- Define maximum time difference
- Record key K arrives on left stream at 7:15
  - Matching record on right stream between 7:15 - 8:15
  - 7:16 : match
  - 7:50 : match
  - 8:12 : match
  - 8:17 : not match

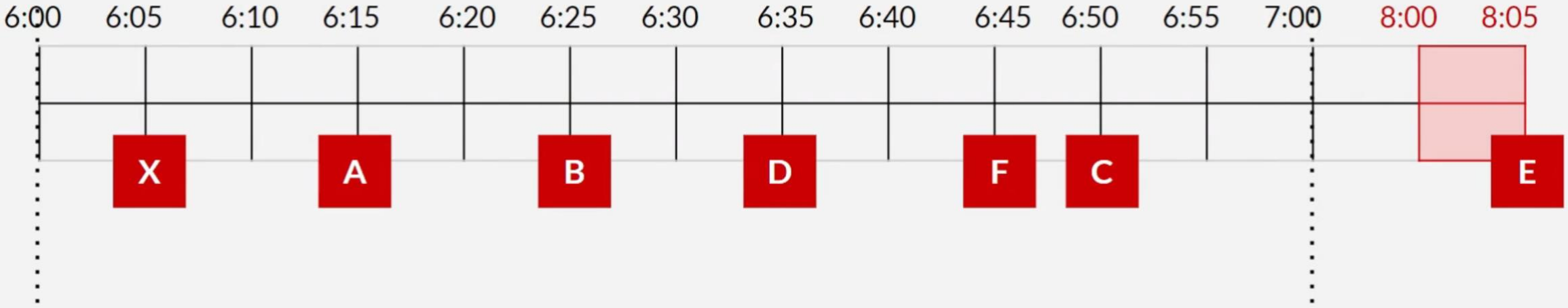
Input Streams

Blue Stream: Left/Primary Stream

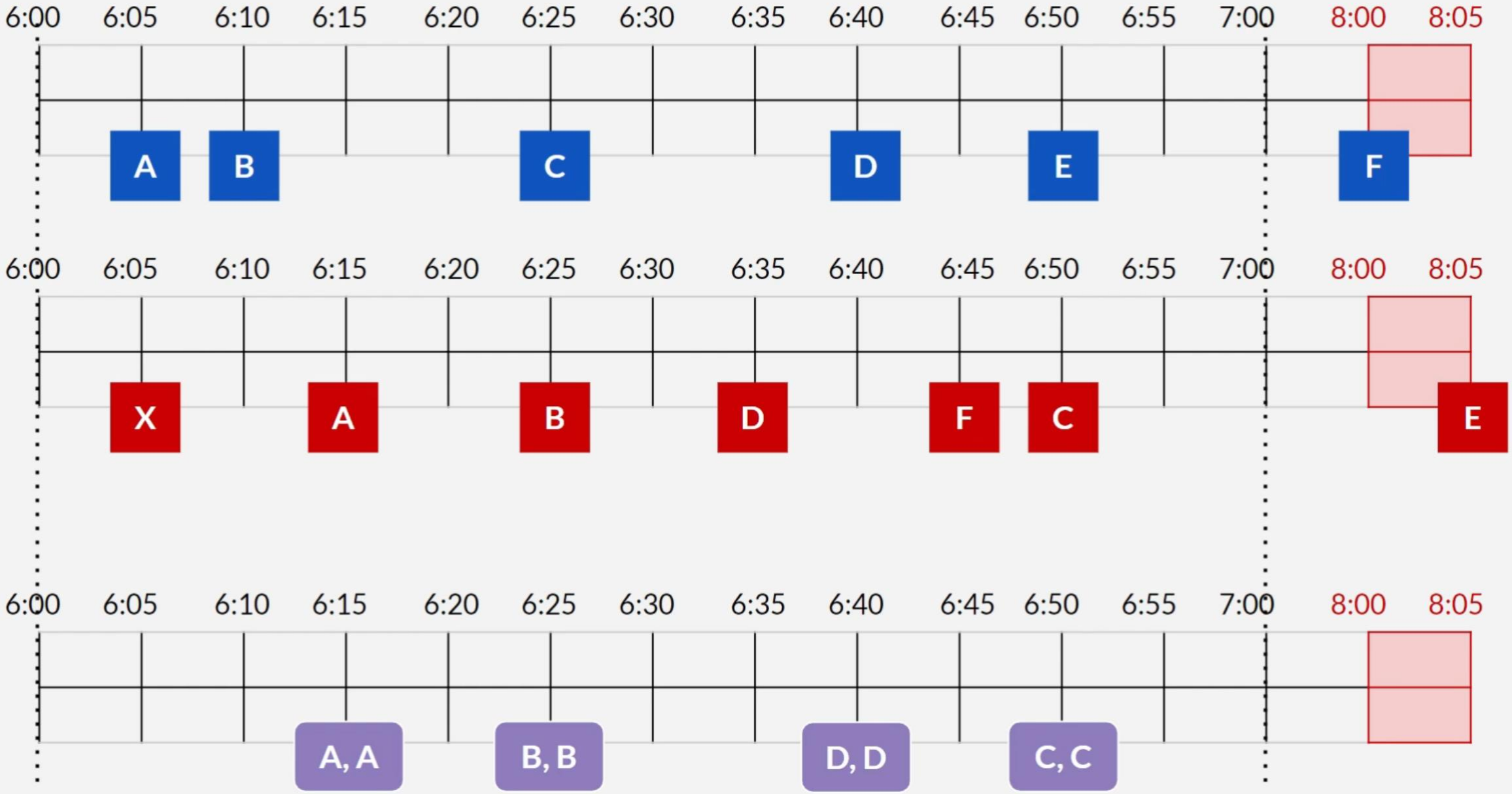
Red Stream: Right/Secondary Stream



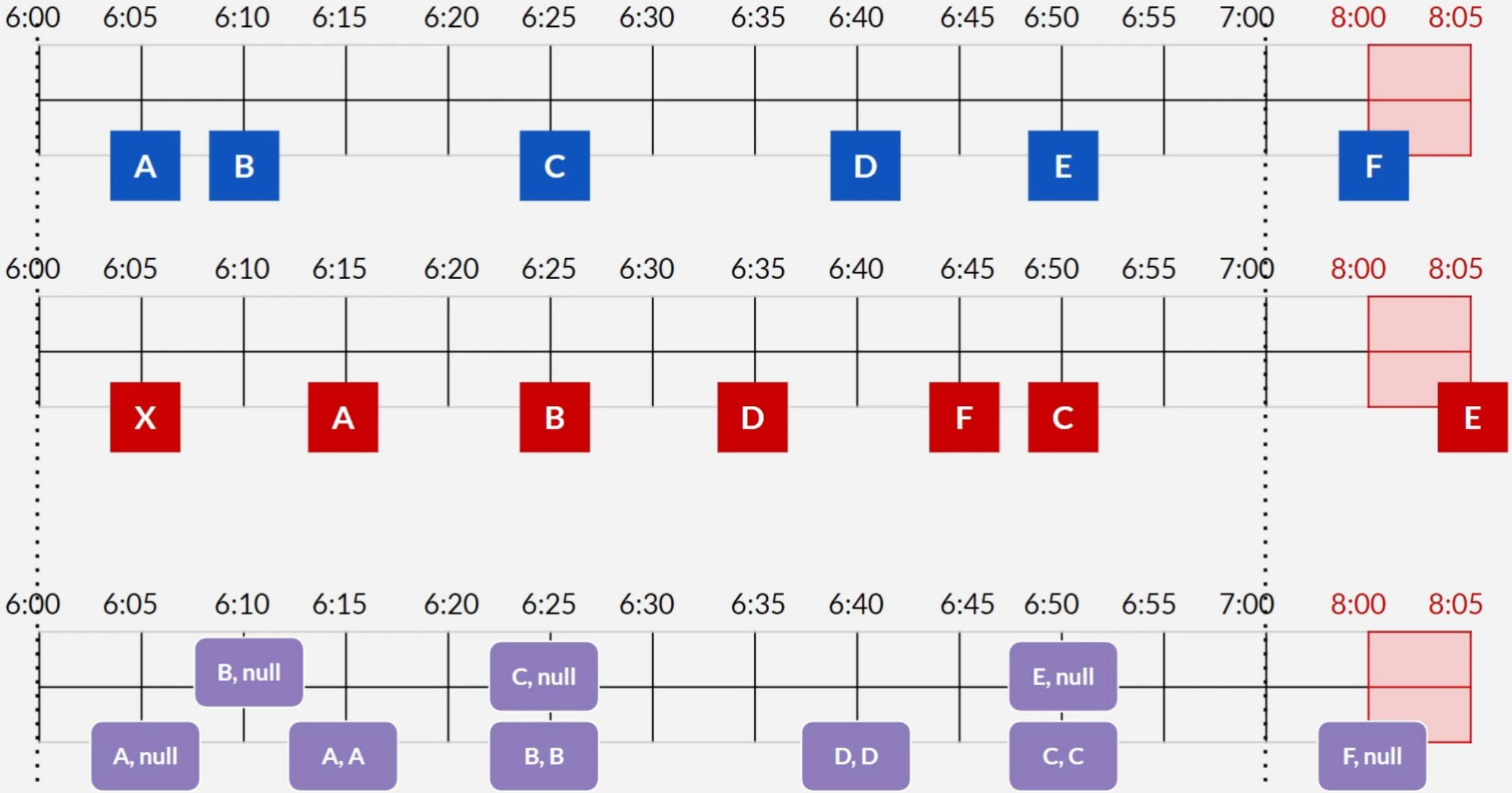
Time window : 1 hour



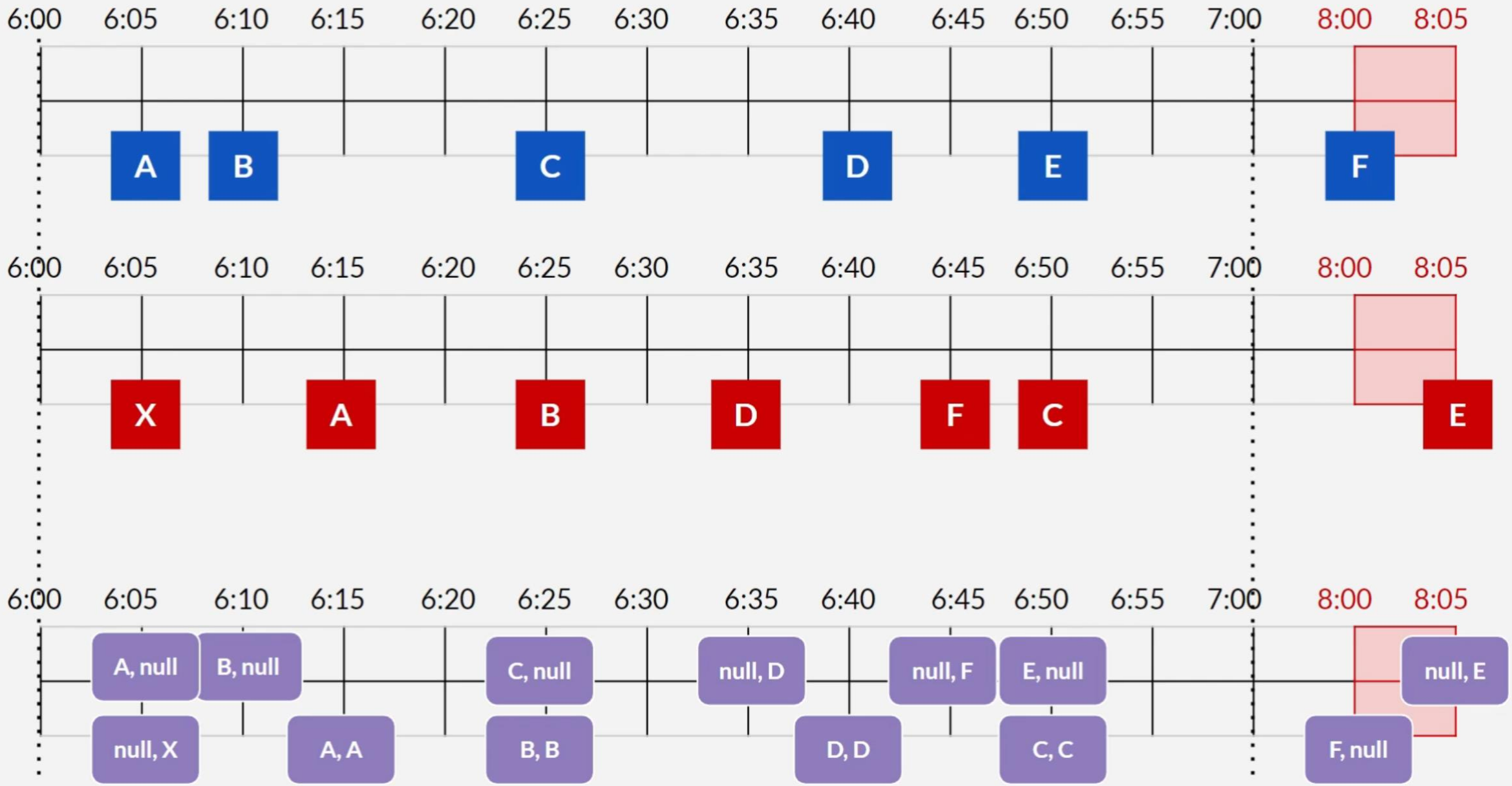
Inner Join Stream-Stream



Left Join Stream-Stream

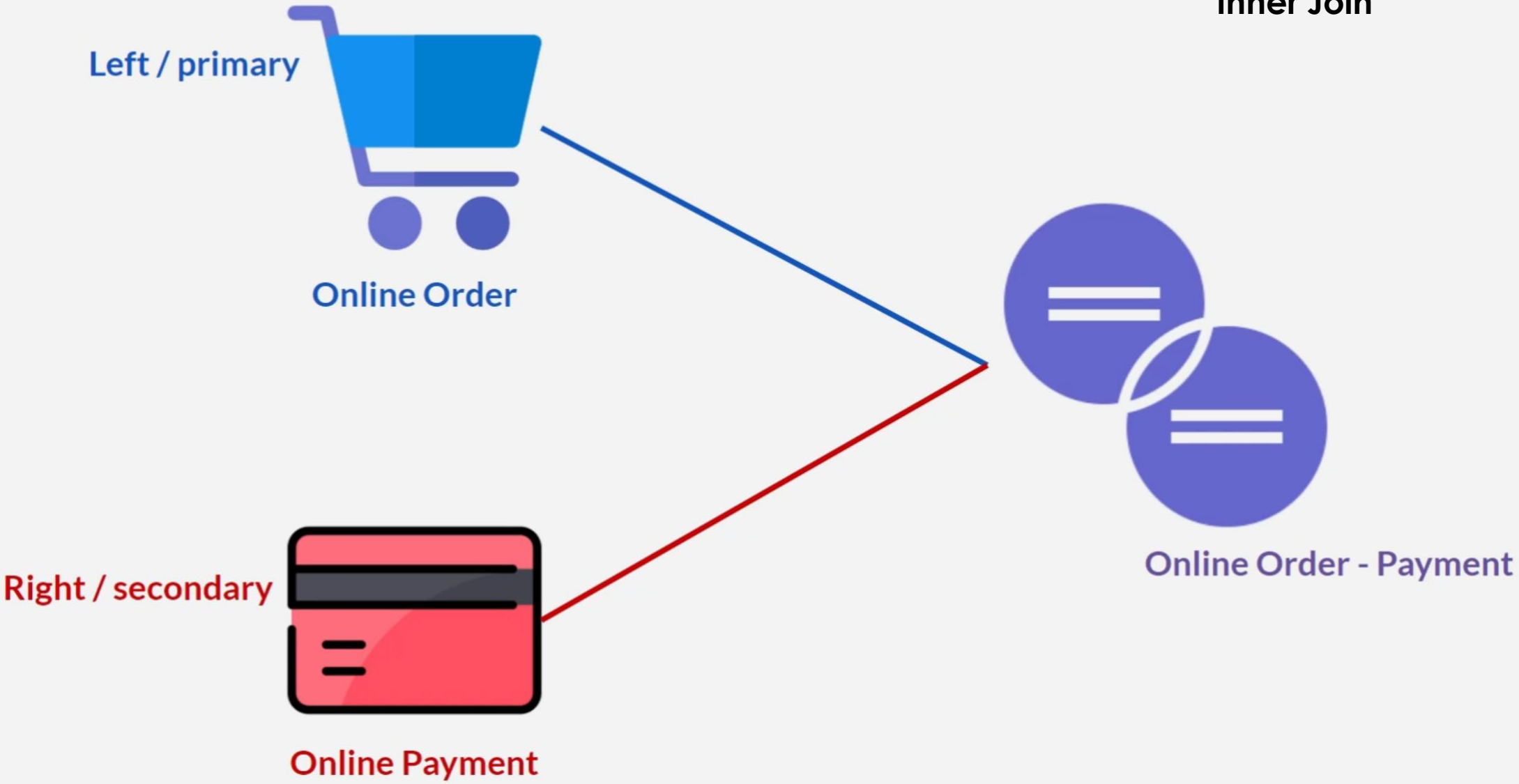


Outer Join Stream-Stream



Online Order & Payment

Inner Join



Join Class

OnlineOrderMessage.java

1st class : left source stream
int a Int b

OnlinePaymentMessage.java

2nd class : right source stream
double x

OnlineOrderPaymentMessage.java

3rd class : join class
int a int b double x

join class example 2
int a double x

join class example 3
int a String someField double aPlusX()

Join Syntax

```
leftStream.join(rightStream, joiner, windows, joined)
leftStream.leftJoin(rightStream, joiner, windows, joined)
leftStream.outerJoin(rightStream, joiner, windows, joined)
```

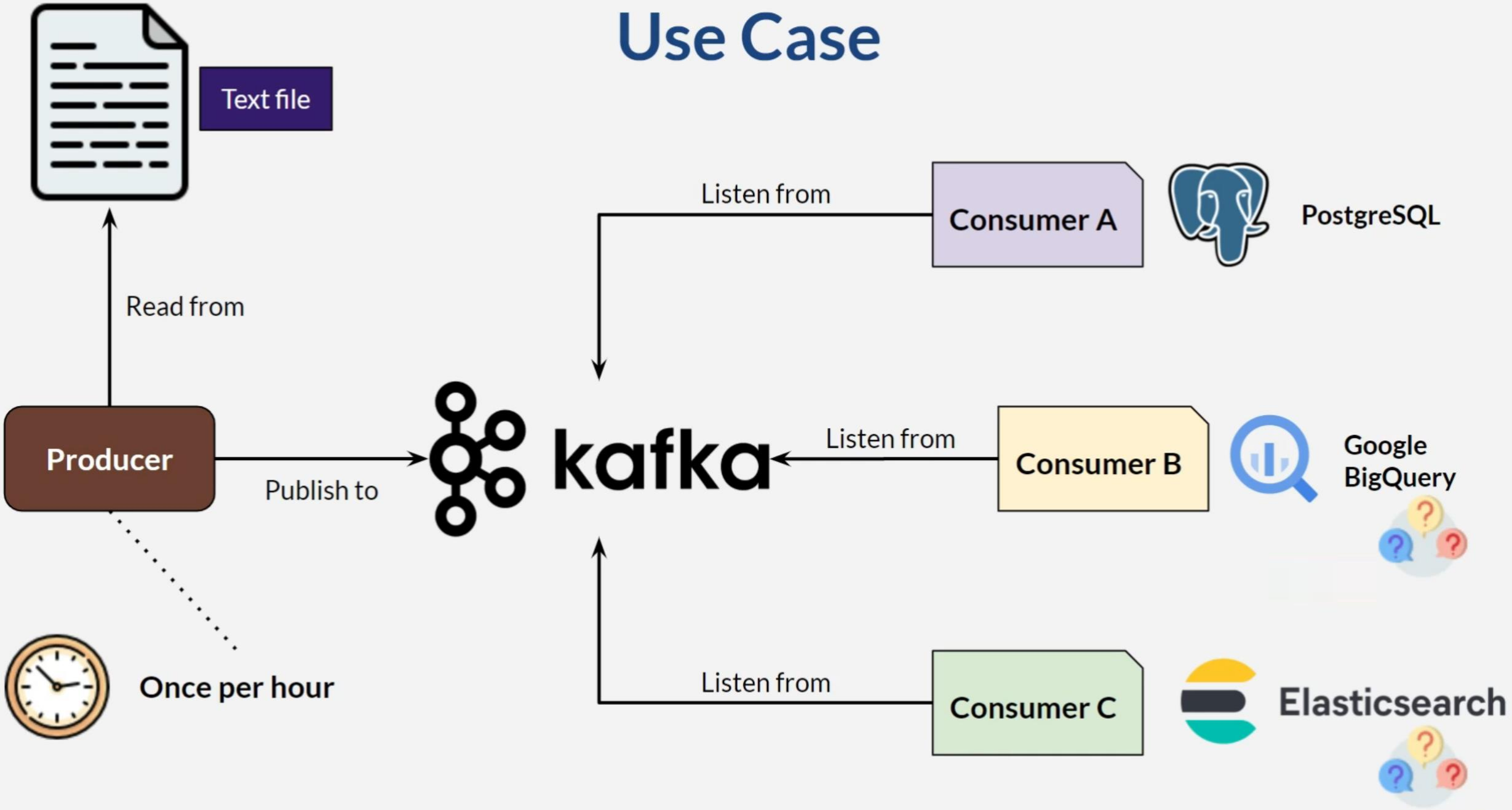
```
// joiner

private JoinClass joiner(LeftClass left, RightClass right) { ... }
```

```
// windows

JoinWindows.of(...)
```

```
// joined

Joined.with(keySerde, leftSerde, rightSerde)
```

Kafka Connect

Use Case


Text file
Read from
Producer
Publish to
kafka
Once per hour
Listen from
Consumer A
PostgreSQL
Listen from
Consumer B
Google
BigQuery
Listen from
Consumer C
Elasticsearch

Another Use Case

PostgreSQL

Read from

Producer

Publish to

kafka

Listen from

Consumer A

Google Cloud Storage

Listen from

Consumer B

RabbitMQ

Listen from

Consumer C

CockroachDB

Near real time

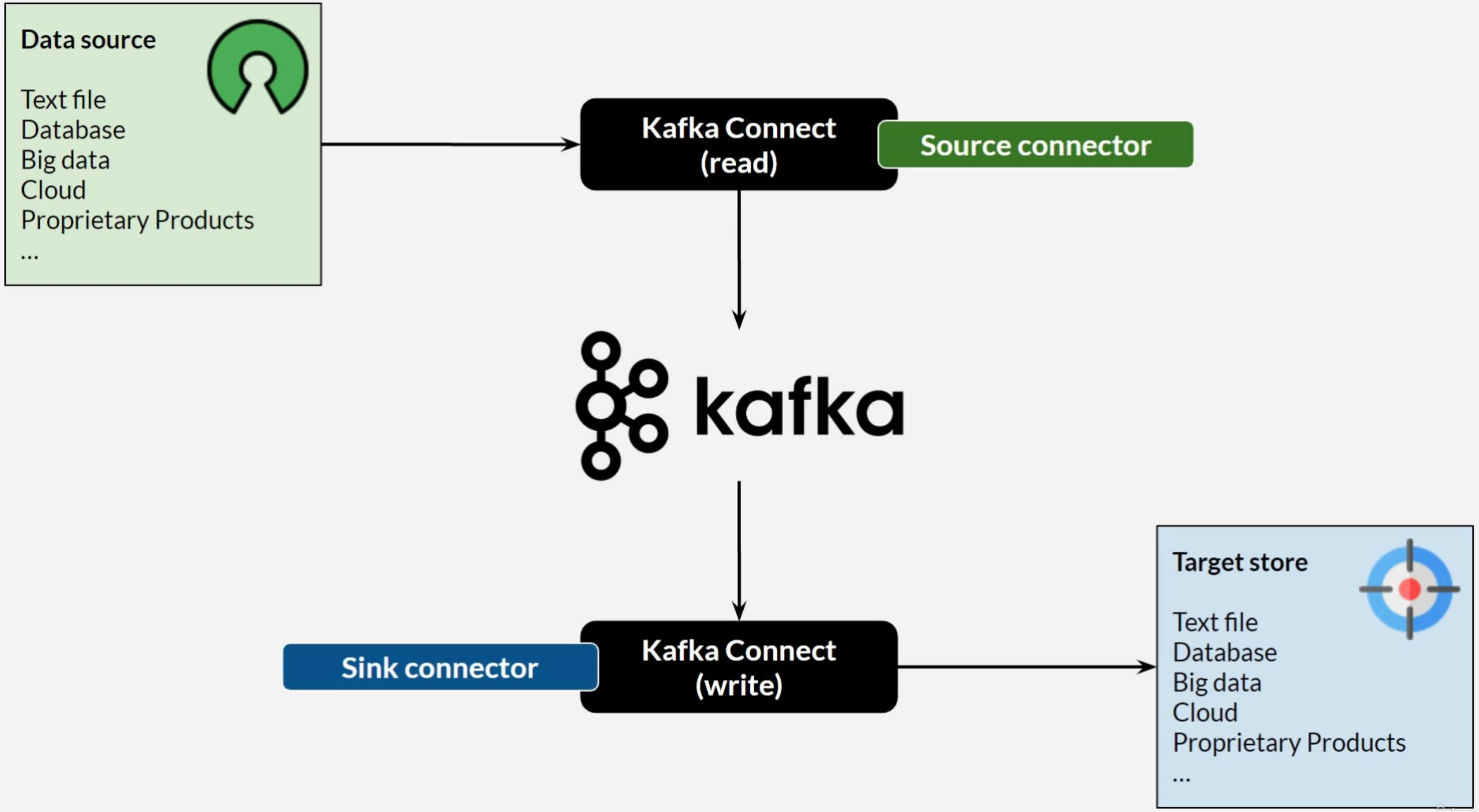
Hard to implement this which is fast & reliable

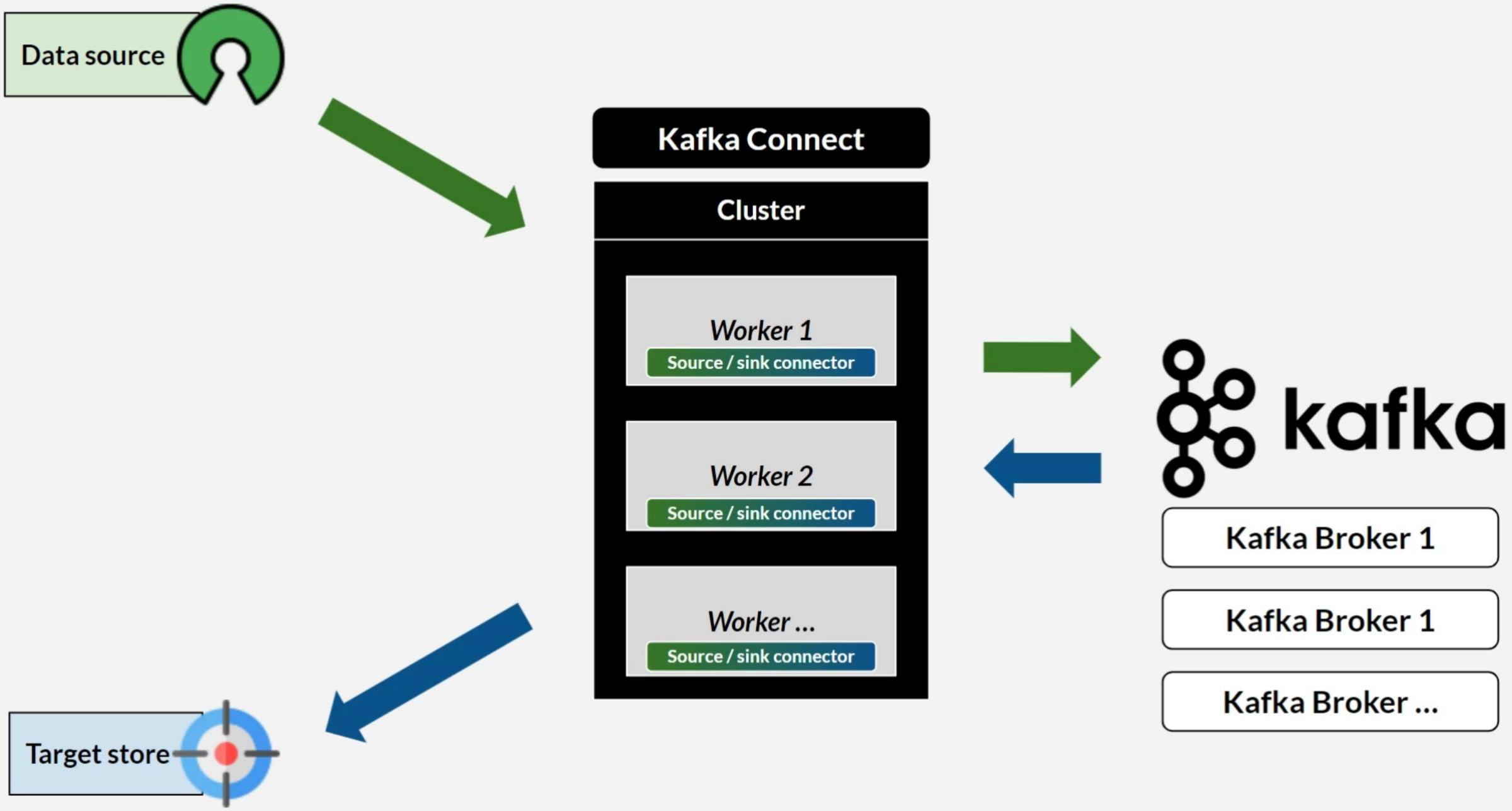
Message In, Message Out

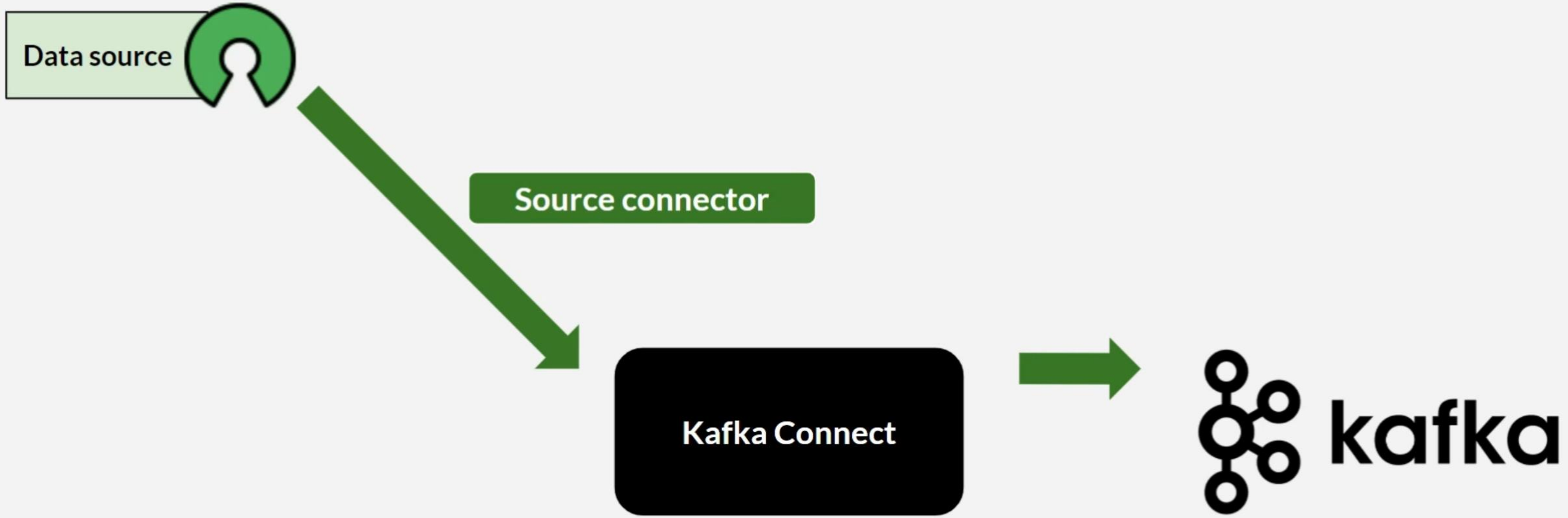
- Happens in real life
- Today we have multiple data sources (**in**)
  - text file, relational databases
  - Non relational database, email, cloud, social media, specific software, etc
  - Can be more than 100
- Multiple target data store (**out**)
  - local text file, FTP, relational / non-relational databases, big data, cloud, etc

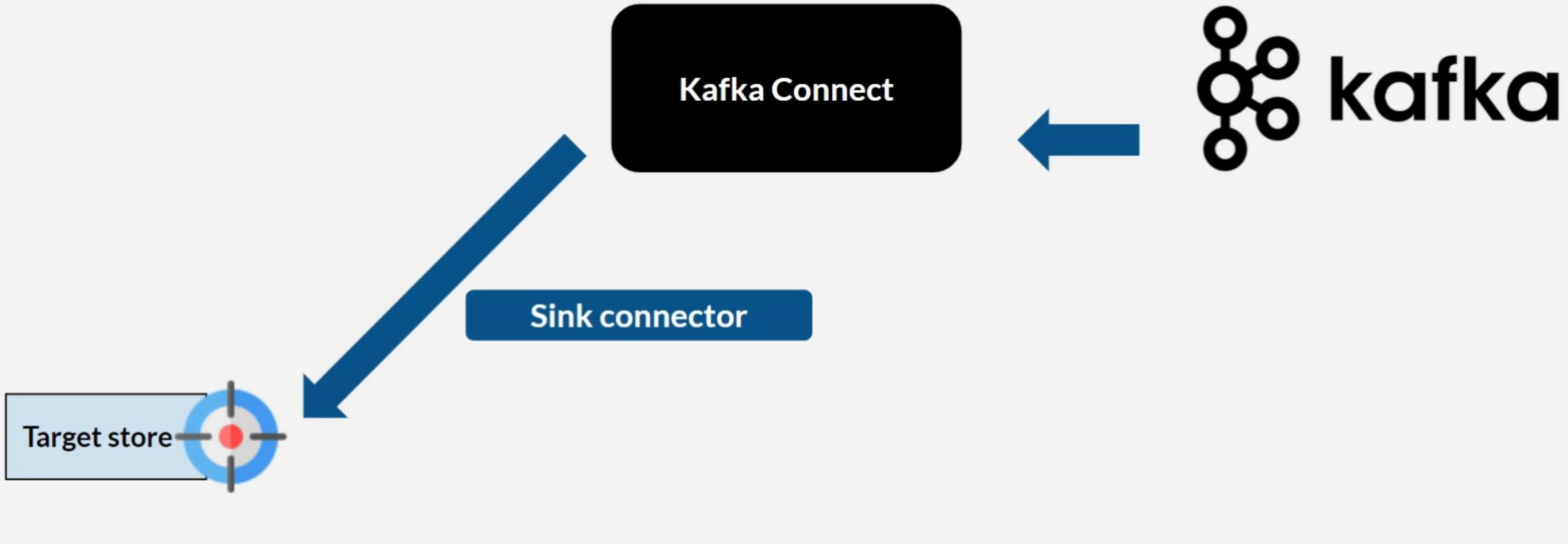
On Kafka

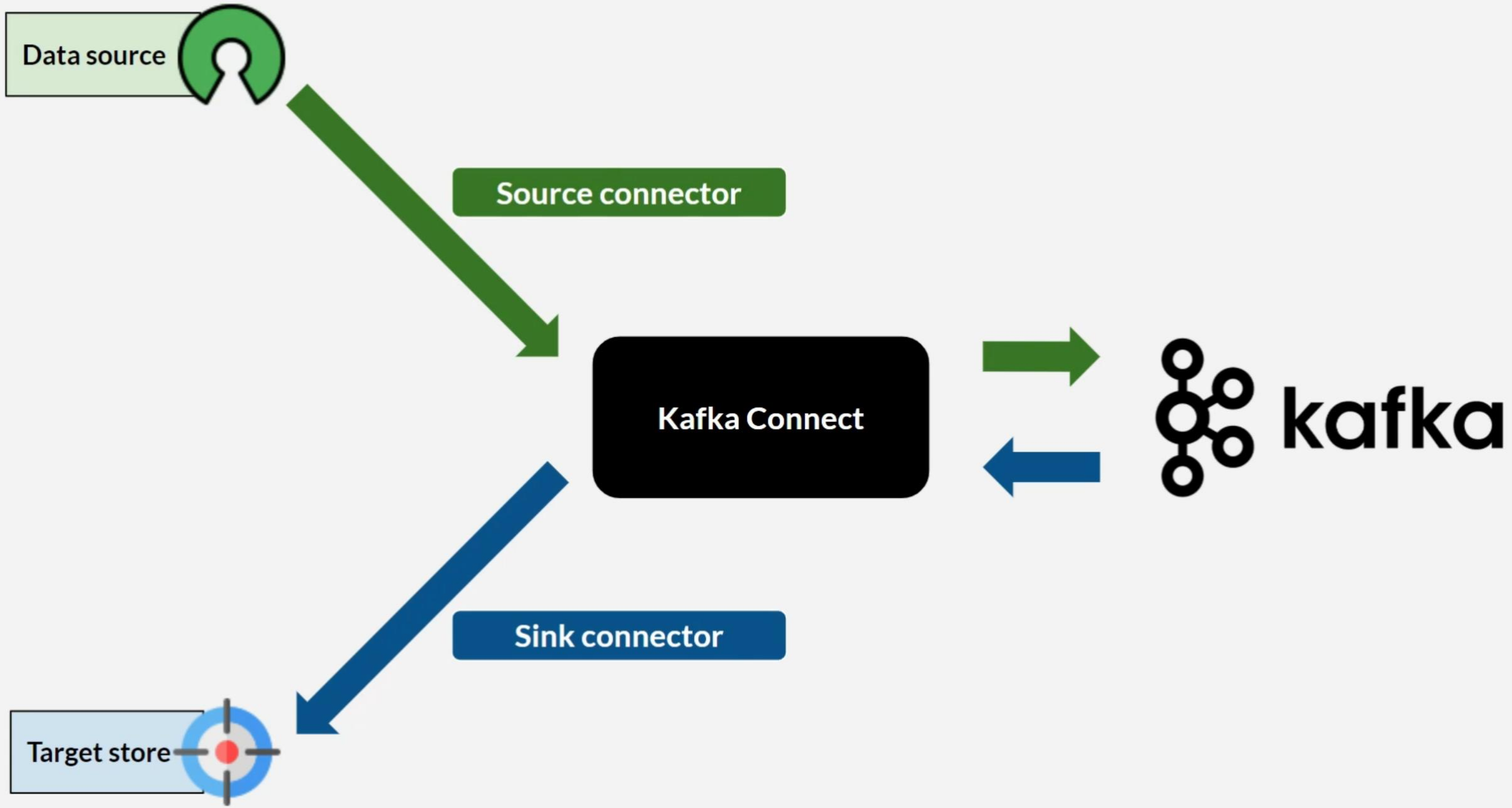
- Write a lot of producers and consumers
- Extra time & effort for performance & reliability
- Good news : ***you don't have to write your own!***
- People / companies already wrote them
- Read plugin : from non-kafka into kafka
- Write plugin : from kafka into non-kafka
- **Kafka Connect**


Data source
Text file
Database
Big data
Cloud
Proprietary Products
...
Kafka Connect (read)
Source connector
kafka
Sink connector
Kafka Connect (write)
Target store
Text file
Database
Big data
Cloud
Proprietary Products
...


Data source
Kafka Connect
Cluster
Worker 1
Source / sink connector
Worker 2
Source / sink connector
Worker ...
Source / sink connector
kafka
Kafka Broker 1
Kafka Broker 1
Kafka Broker ...
Target store


Data source
Source connector
Kafka Connect
kafka


Kafka Connect
kafka
Sink connector
Target store


Data source
Source connector
Kafka Connect
kafka
Sink connector
Target store

Kafka Connect

- Additional platform for kafka
- Data integration
- Transfer data between Kafka - non kafka
- Horizontally scalable & fault tolerant
- Uses connectors for interact with kafka server

Connectors

- Java jar file
- Plugin for kafka connect
- Interface between kafka and non-kafka
- **Source** connector : read (ingest) into kafka (**producer**)
- **Sink** connector : write from kafka to non-kafka (**consumer**)
- Install connector for specific need
- Write configuration (json)
- Declarative configuration
- Fast & reliable

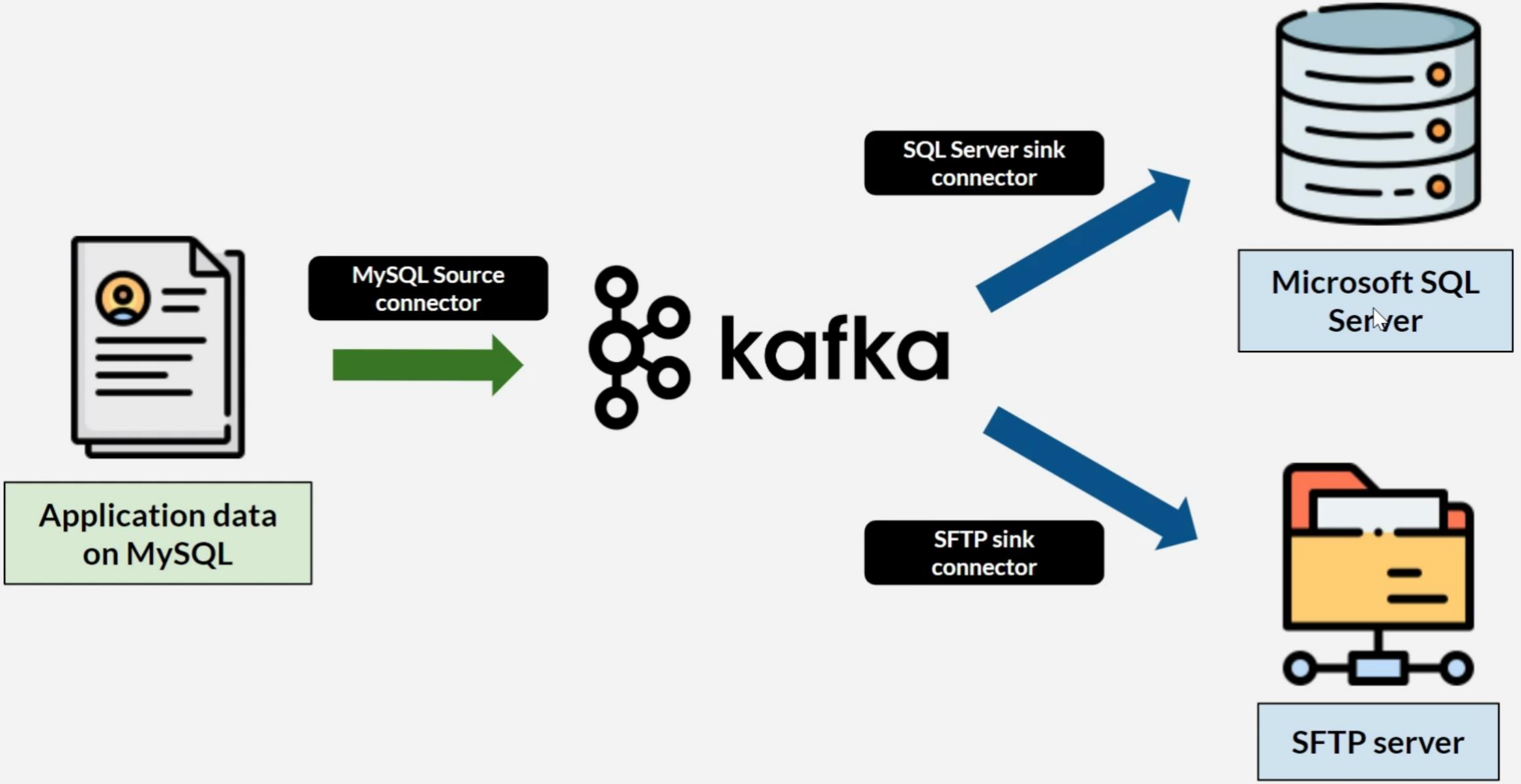
Connectors

- A lot of source / sink connectors
- Shorten time and effort

How to Get Connectors?

- Curated list on ***confluent.io/hub***
- Google : *kafka source / sink connector for xxx*
- Build your own

Use Case : Write to Data Stores



Use Case : Modernize Legacy System

- Modernize legacy monolith into microservices
- Modernization is hard and long
- Legacy system still needs to run during modernization
- Modernize functionalities part by part
- Use kafka connect CDC (Change Data Capture) connectors

Use Case : Modernize Legacy System

Monolith app

Monolith DB (PostgreSQL)

CDC Connector 1 (finance)

CDC Connector 2 (marketing))

CDC Connector 3 (accounting)

kafka

PostgreSQL sink connector

Finance new database (PostgreSQL)

Listen from

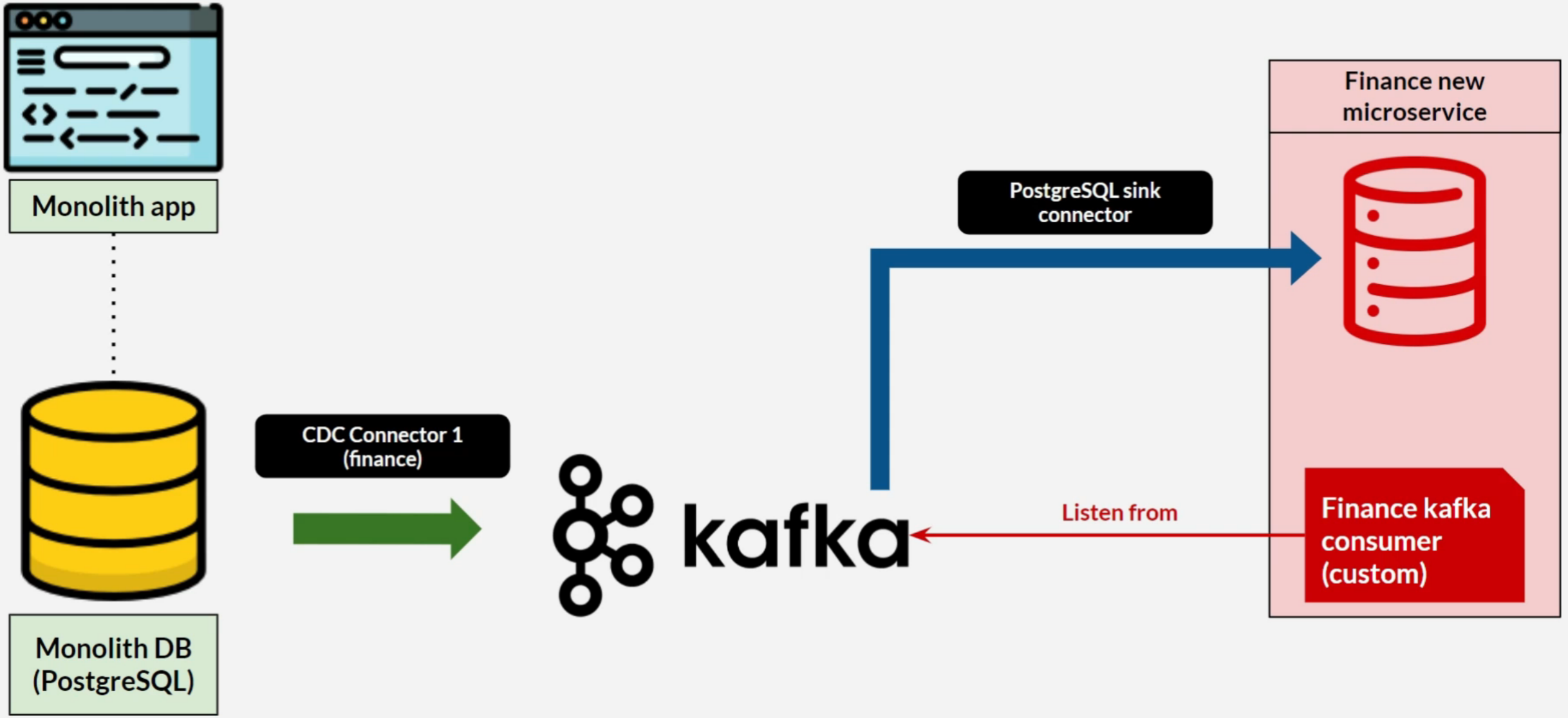
Marketing kafka consumers

Listen from

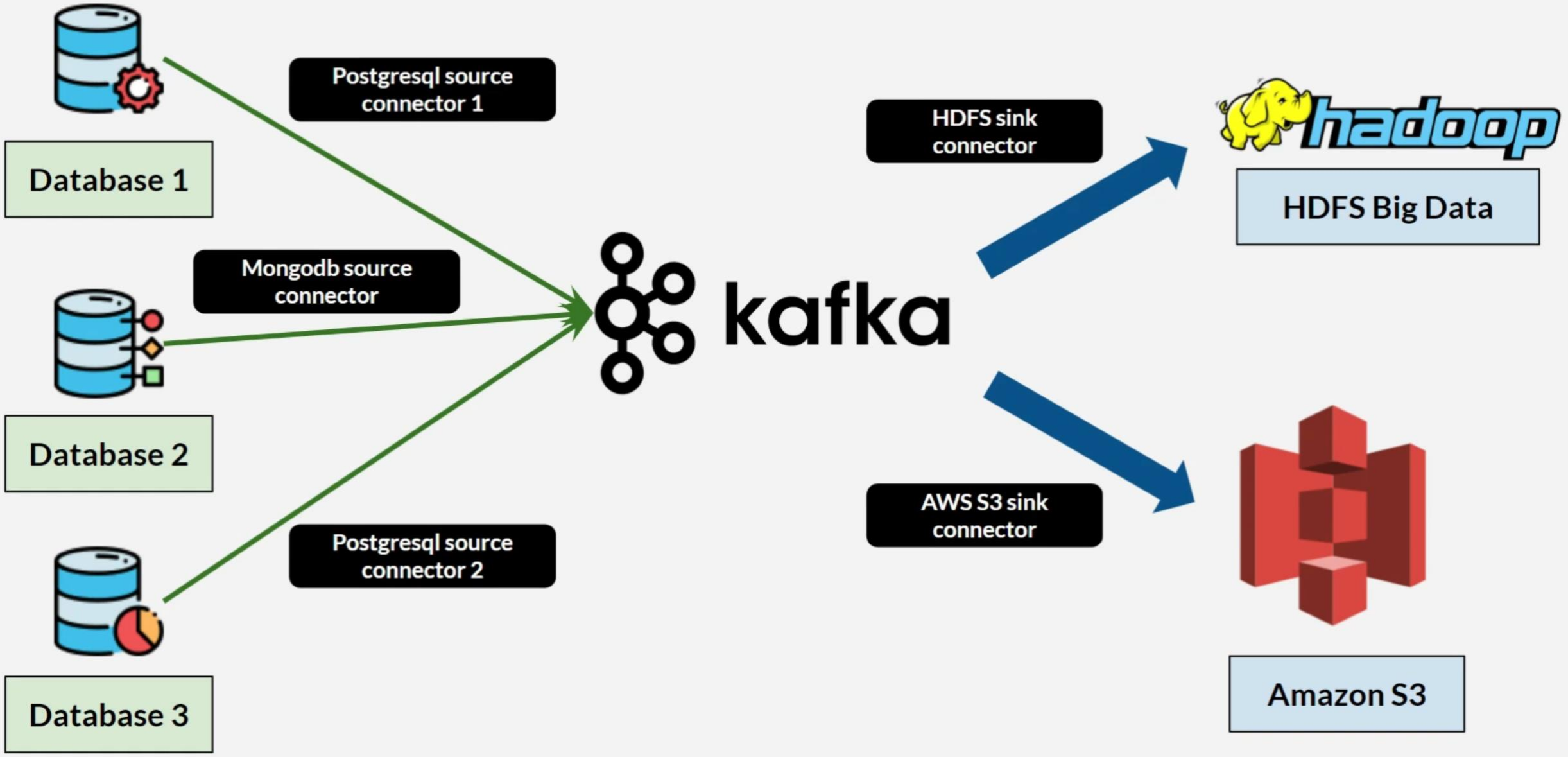
Accounting kafka consumers

Changed data : insert / update / delete

Use Case : Modernize Legacy System



Use Case : Data Engineering ETL Pipeline



Venkat
Corporate Trainer & Motivational Speaker