DATA TECHNOLOGIES AND SERVICES

LECTURER: Martina Šestak, Ph.D.

martina.sestak@um.si

Office: G2-1N.10

Office hours: Tuesday, 9AM-10AM

(send email beforehand)

Data streaming keywords

- Data-driven systems
- Pub/Sub
- Messaging systems
- Cluster
- Producer
- Consumer
- Broker
- Topic

Batch or stream processing

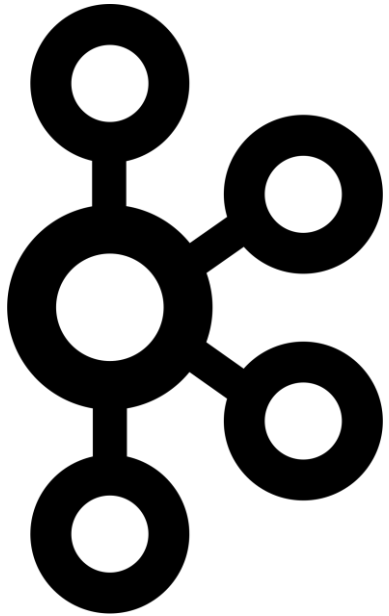
- The distinction between **batch** and **stream** processing is one of the most fundamental principles in the world of big data
- In the batch processing model, data is collected and then transferred to an analytical system for processing. In other words, a (large) batch of information is collected and then sent for processing
- Under the streaming processing model, data is fed into the analytical tools on a piece-by-piece basis. The processing is usually done in real-time

Messaging system

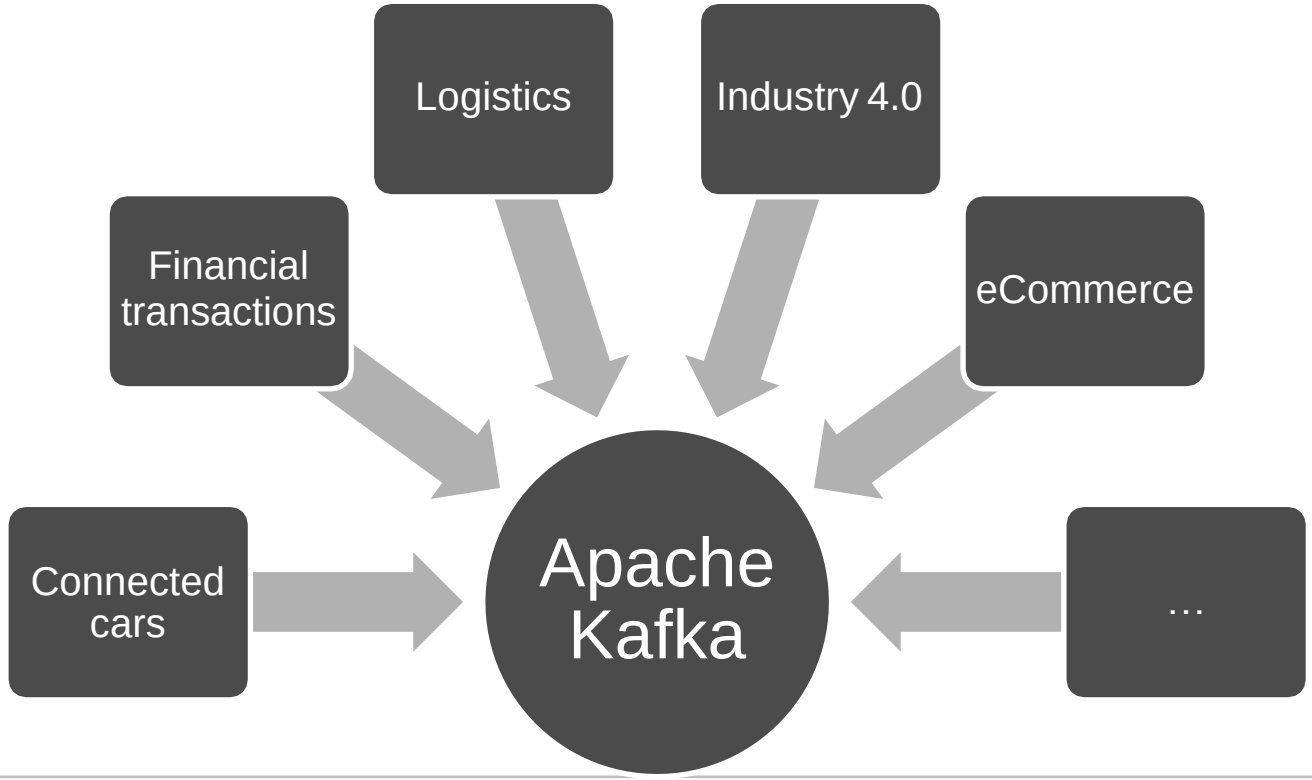
- When it comes to big data, we have two main challenges: (1) how to collect a large amount of data, (2) how to analyse the data collected. To overcome these challenges, we need a **messaging system**
- Messaging systems are responsible for transferring data from one application to another, so applications can focus on the data rather than how to share the data
- Distributed messaging is based on the concept of a reliable message queue. Messages are asynchronously queued between client applications and the messaging system
- Two types of message patterns are available (1) **point-to-point**, (2) **pub-sub**. Most message patterns follow pub-sub

Source of data

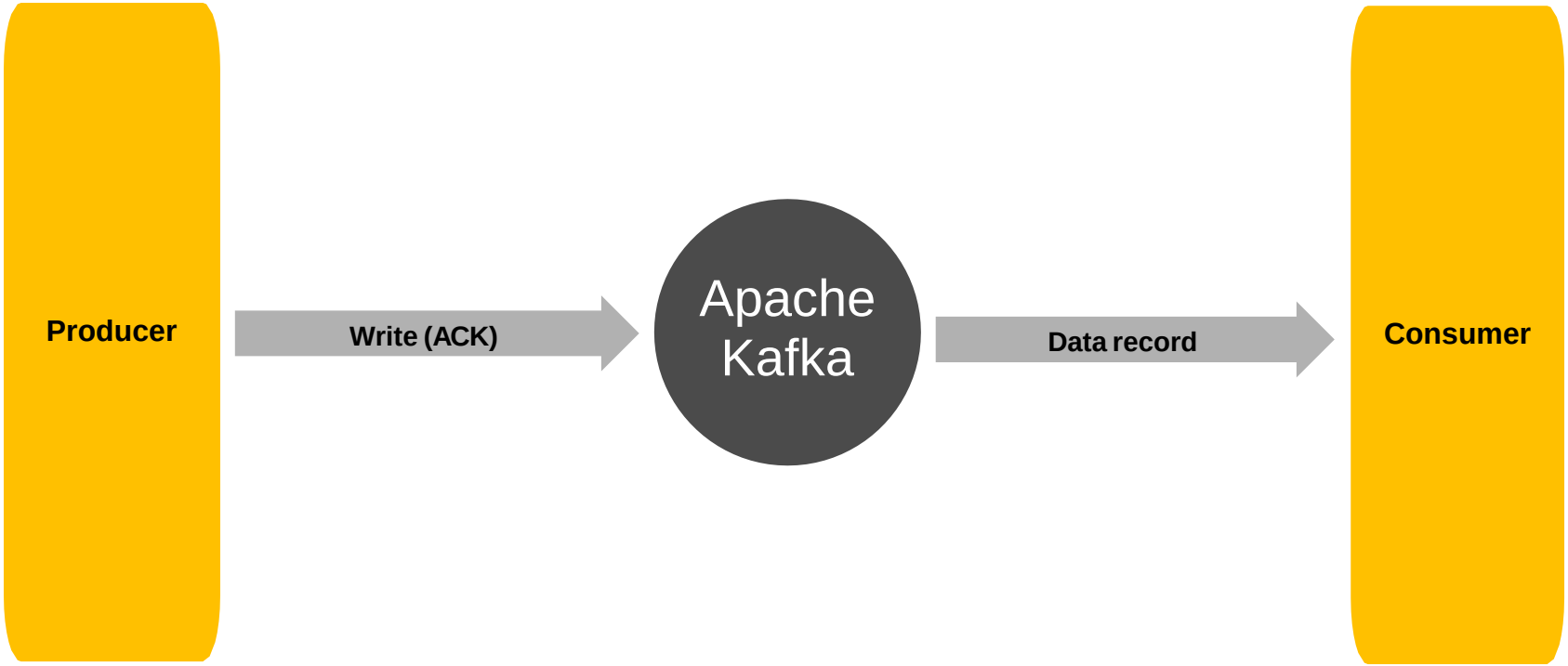
- Apache Kafka is a (distributed) messaging system/ platform / database, that provides real-time data processing at low latency and high throughput
  - Can be used as an enterprise messaging system
  - Can be used for data stream processing



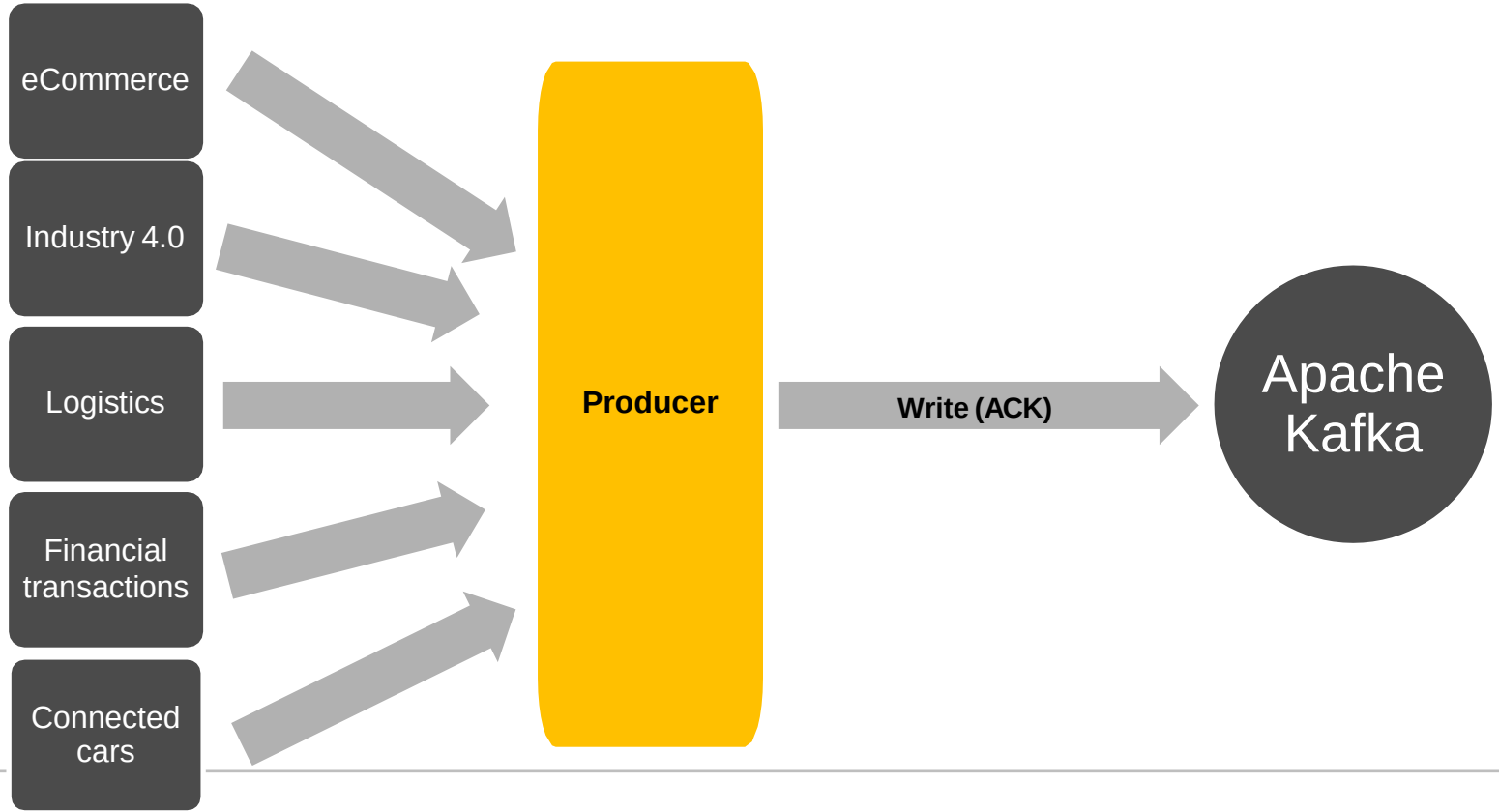
Source of data



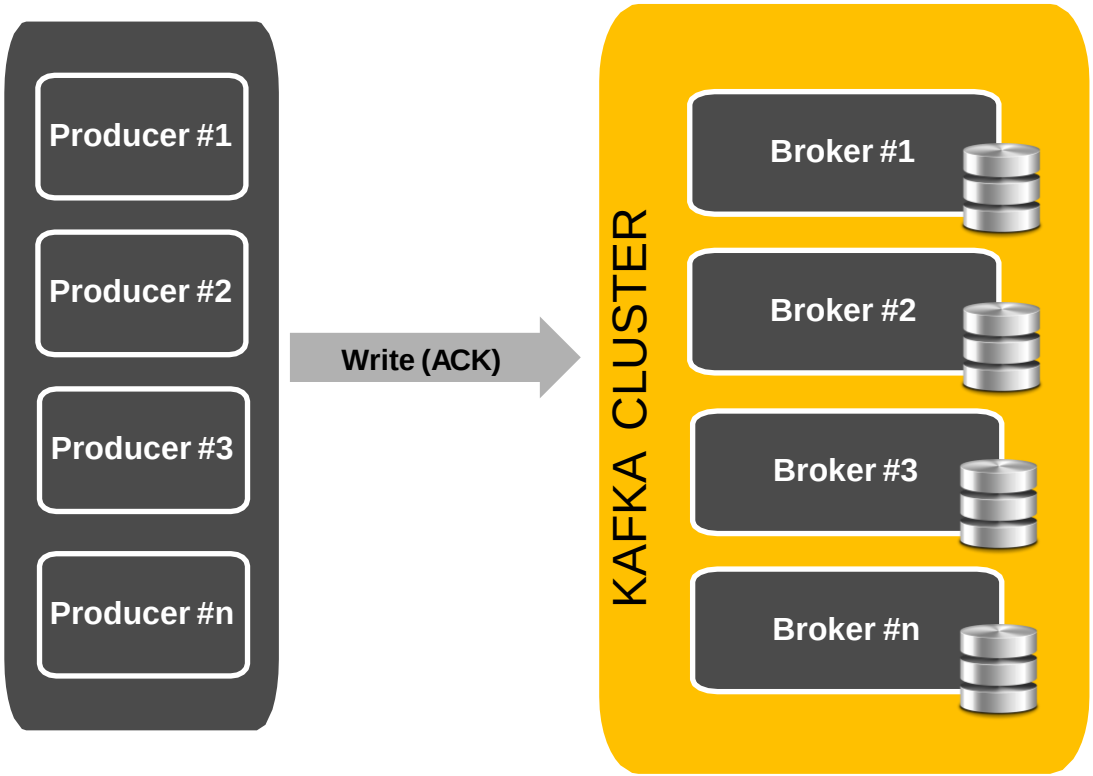
Producer - Consumer



Producer - Consumer

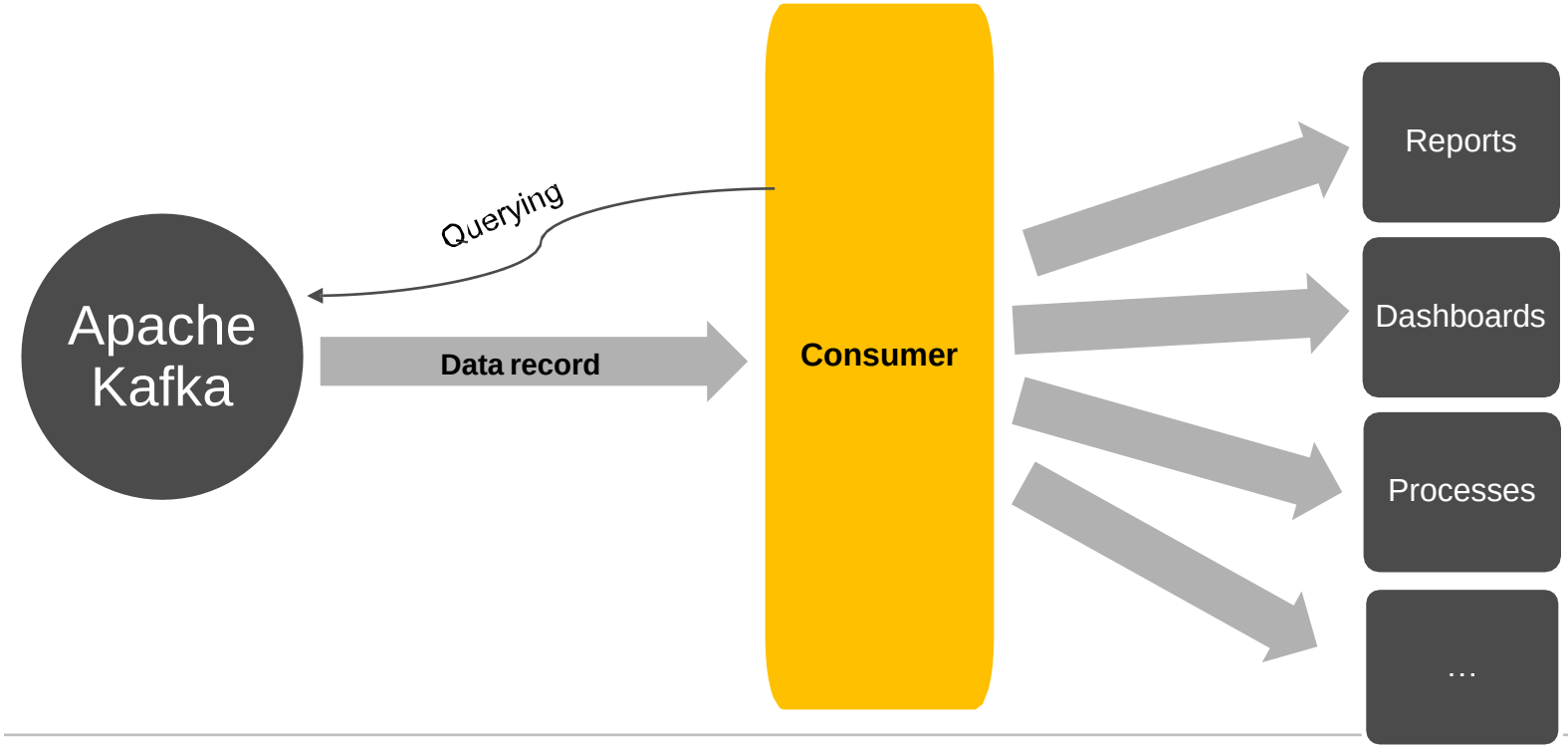


Broker



- Apacke Kafka is by default a cluster of distributed brokers
- It can be a cloud service where we are not "interested" in running the cluster, or we can manage it ourselves
- Each individual broker keeps persistent memory
- The cluster is managed by a "Zookeeper"

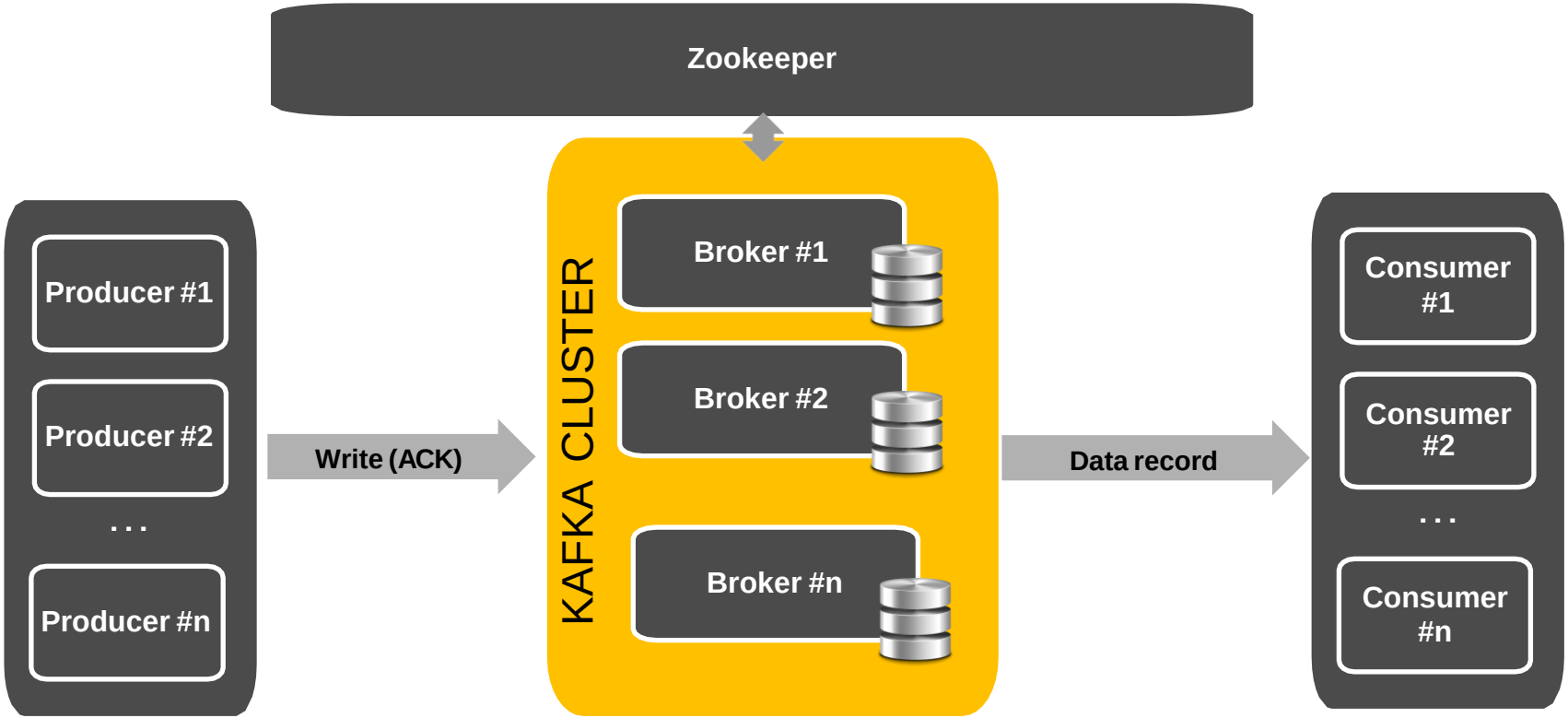
Consumer


Querying
Apache Kafka
Data record
Consumer
Reports
Dashboards
Processes
…

Producer – Consumer – Broker

- In Kafka's ecosystem, this is simplistically represented as infrastructure, while producers and consumers are purposely designed and developed
- Producers and consumers are strictly decoupled
  - Slow consumers do not affect producers
  - Adding consumers does not affect the ecosystem
  - Consumer failures do not affect the ecosystem
  - Consumers can scale without the knowledge of producers

Producer – Consumer – Broker

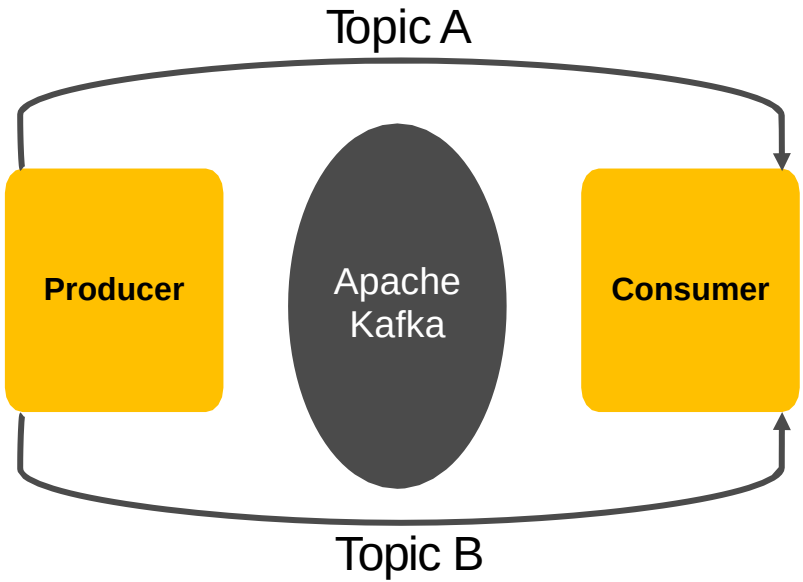


Zookeeper

- Zookeeper takes care of
  - providing distributed agreement/consensus in a distributed Kafka cluster architecture by designating a "leader",
  - storing metadata about the Kafka cluster, as well as details about consumers (e.g., ACLs),
  - error detection,
  - etc.
- We can install a group of Zookeepers called an ensemble. Due to the algorithm used, it is recommended that ensembles contain an odd number of servers (e.g., 3, 5, etc.)
- KIP500 (Kafka Improvement Proposal) attempts to remove Zookeeper and allow it to work without it

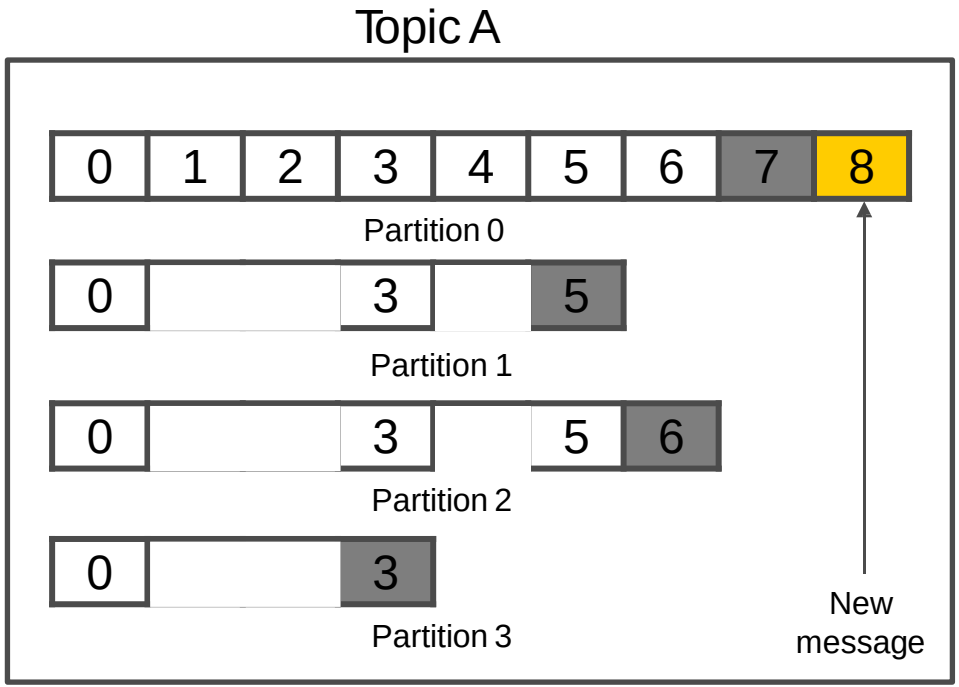
Topic

- The messages in Kafka are grouped into topics
- Topics represent a stream of data records, which are categorised into groups
- Topics are defined by developers
- The closest analogy to a topic is a table in a database
- One producer can generate several different topics
- One topic can be generated by several different producers
- The number of topics in the Kafka ecosystem is unlimited

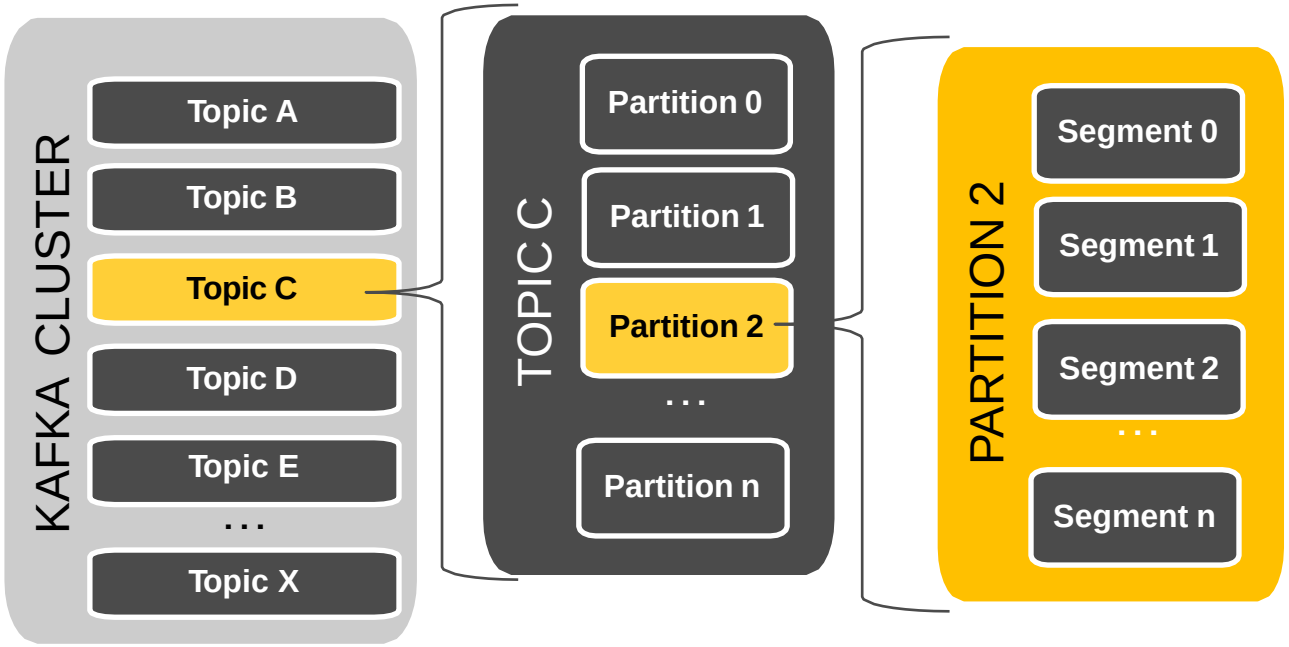


Topic – Partition

- The topic is usually divided into several partitions
- Partitions are Kafka's way of ensuring redundancy and scalability
- Partitions are like journal entries - each new message is just added to the end of the entry
- Order (temporal) is not necessarily guaranteed at the level of the topic, but it is certainly guaranteed at the level of the individual partition of a topic



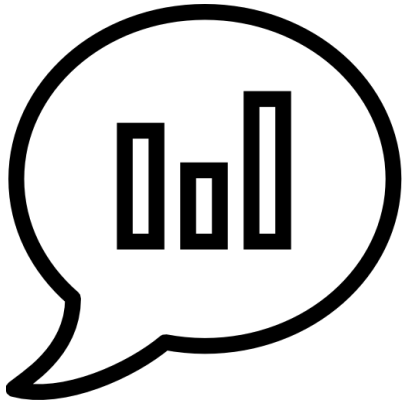
Topic – Partition – Cluster



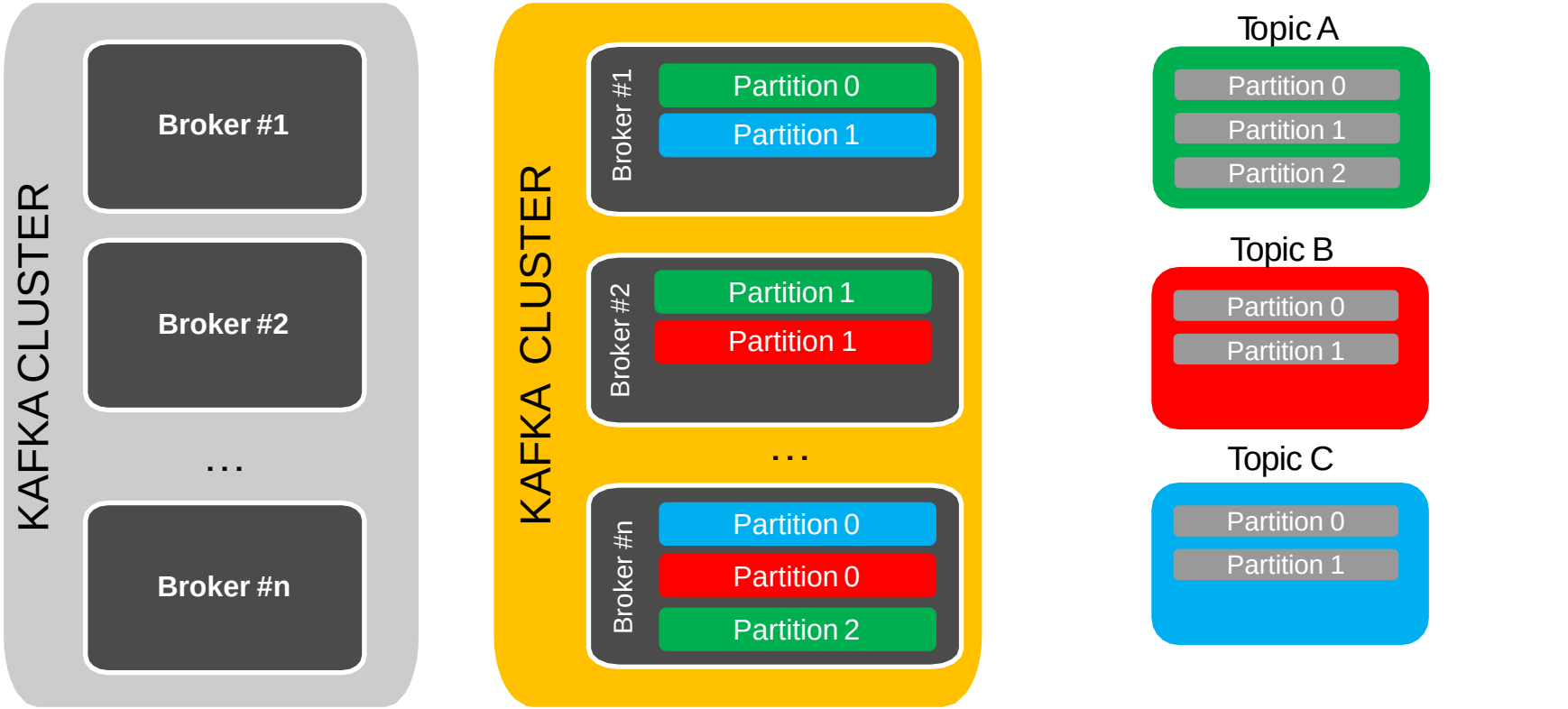
- Each individual partition of a topic can be hosted on different Kafka cluster brokers, which means that we implement horizontal scalability for each topic
- Segments are the physical level of data storage on each broker

Topic

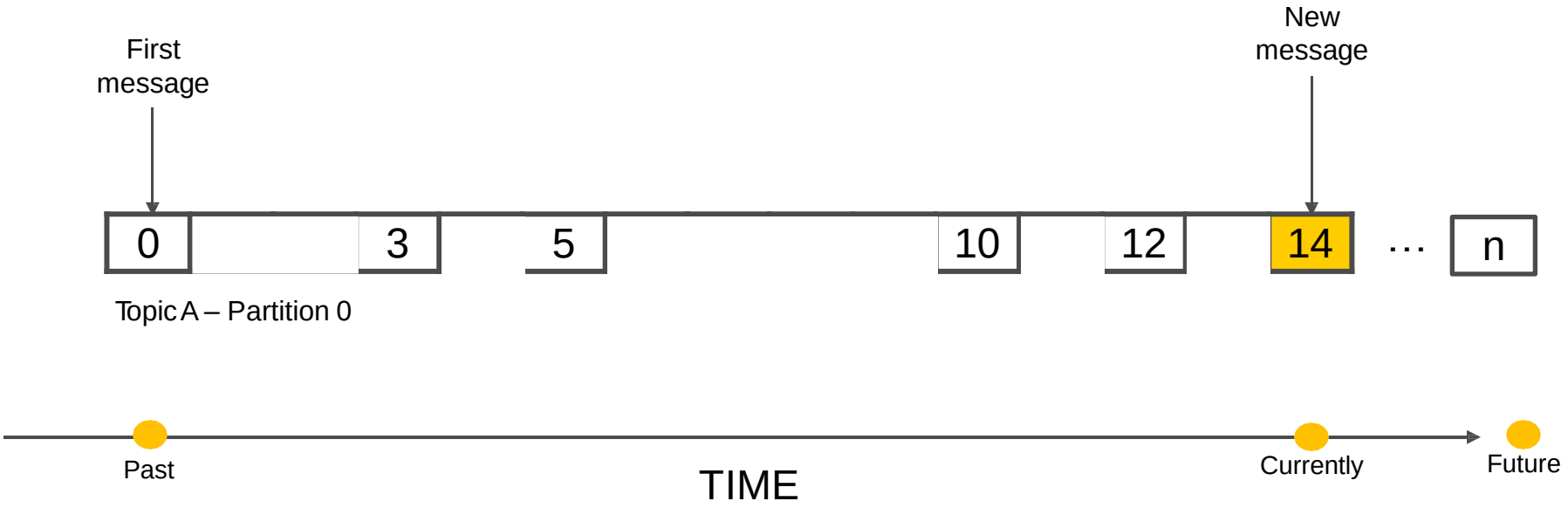
- Producers prepare messages that are part of a topic
- By default, the producer is not interested in which partition a particular message is written to and will balance the messages evenly across all partitions of the topic
- In certain cases, the producer will route messages to specific partitions, ensuring that certain messages are written to the same partition and the same broker
- Consumers subscribe to one or more topics and read the messages in the order in which they were created
- Kafka brokers are configured with default retention settings for topics, either retaining messages for a certain period of time (e.g., 1 hour) or until the topic reaches a certain size in (e.g., 100 MB)



Topic, Partition

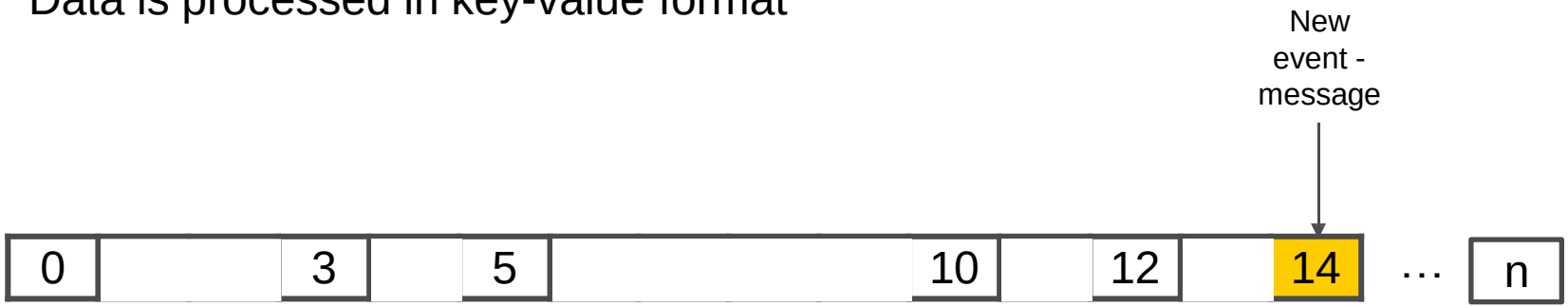


Topic:Partition – Data flow

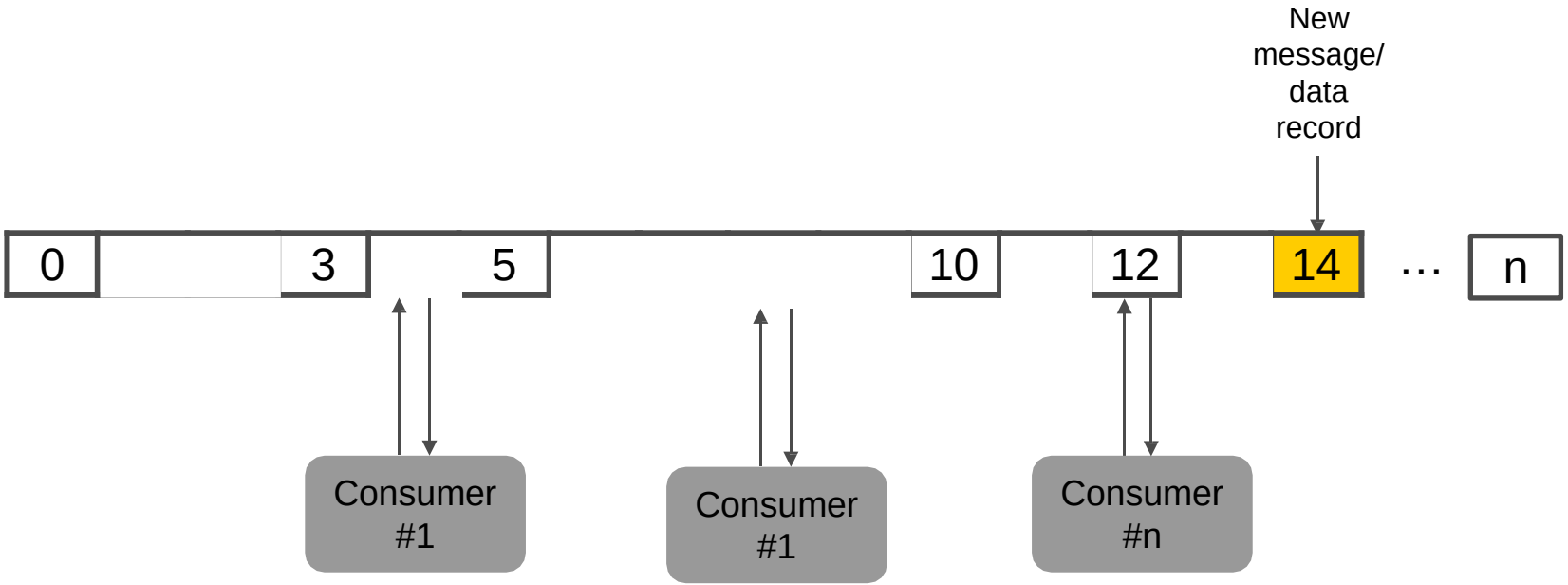


Topic:Partition – Data flow

- Data flow is an unrestricted and continuous flow of data in real-time
- Consumers independently **read** data records from the stream, "wanting" to follow real-time processing as closely as possible
- Data is not deleted by reading data
- Data is processed in key-value format

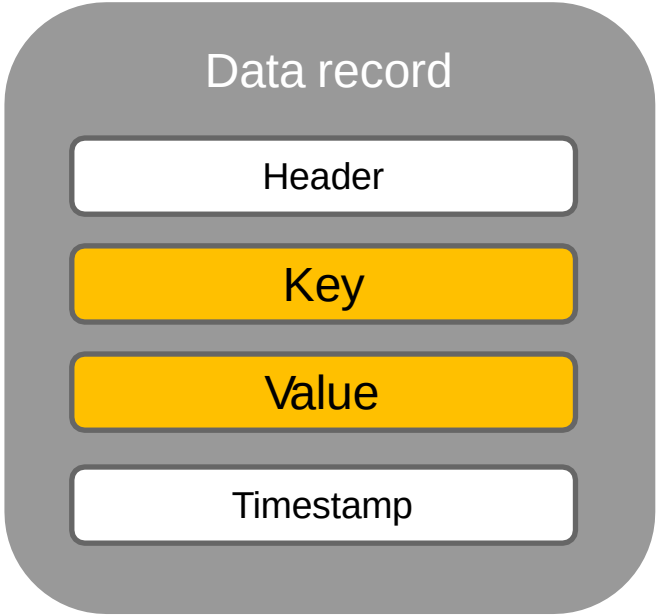


Data flow - Consumer



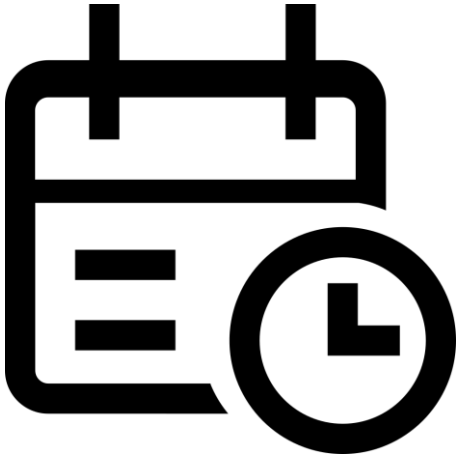
Data record

- Every data record in Kafka is in a KEY-VALUE format from the business system's point of view
- The value itself can be serialised and structured as desired
- A timestamp is automatically assigned to the data record if it is not set manually
  - Manually defined when the actual time of the event described by the record is to be defined
- The header is optional and is structured in key-value format

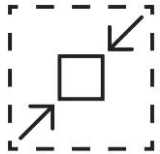


Data retention

- A key feature of Apache Kafka as a messaging system is the provision of (persistent) retention time
- Kafka brokers are configured with a default data retention policy for topics
- Possible retention of messages based on time (e.g., 1h, 7 days, etc.) or size i.e., until the topic reaches a certain size in syllables (e.g., 100 MB)
  - Once these limits are reached the messages are deleted
- The default value is set to **1 week**
- It is configurable either **globally** or at **topic level**
- Fully customisable for business use
  - A matter of judgement



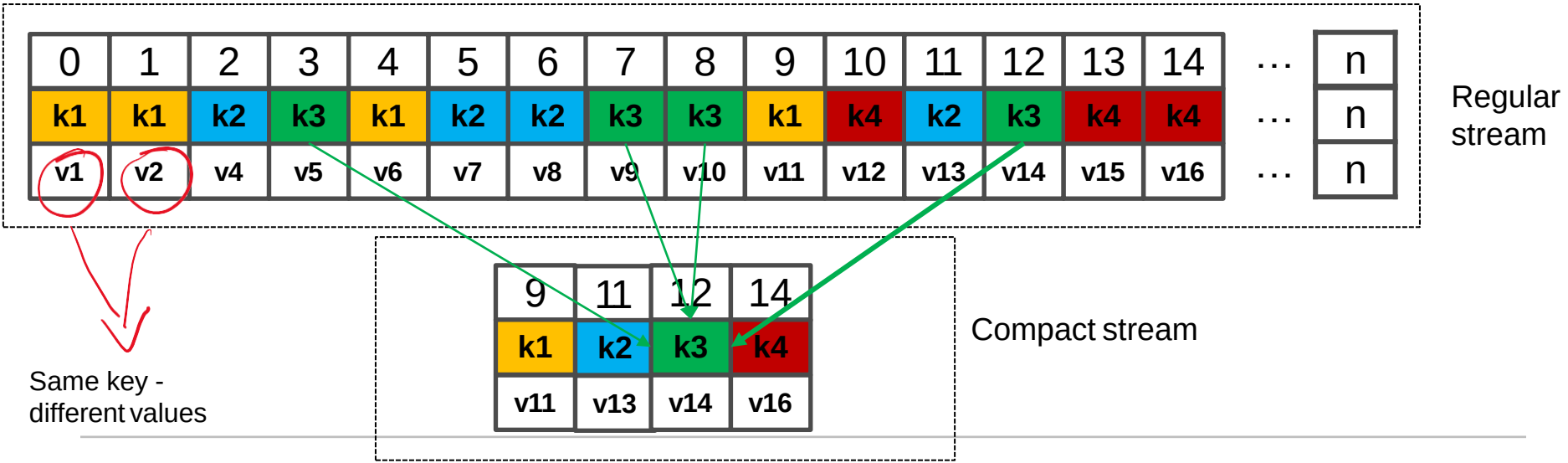
Compacted topic



- In some cases, business logic allows us to ignore older values, and in such cases, we can use so-called **compacted topics**
- Example:
  - If the event is a **user profile update** - a new email, then for a given user we are only interested in the most recent event and not all events in between - in this case we can ignore all events in between
  - The same is not true if the events are values that are necessary for us to process, such as a new share value, because we are also interested in older values for the purpose of plotting the value history
- When setting a compact theme, Kafka **automatically** ensures that the most recent relevant values are preserved at the **level of the individual key**
  - To ensure this feature, Kafka splits the flow of each topic into a head and a tail

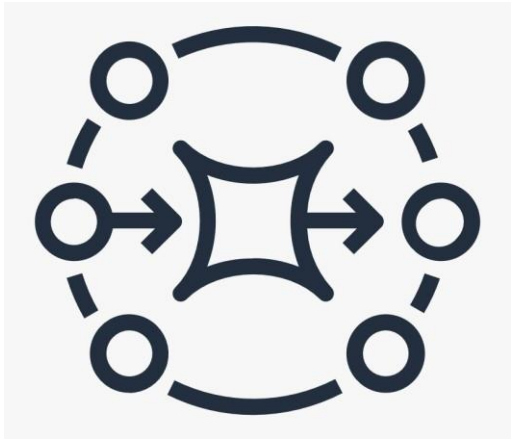
Compacted topic

- In the case of a compact stream of a given topic, Kafka keeps only the last value for each defined topic key

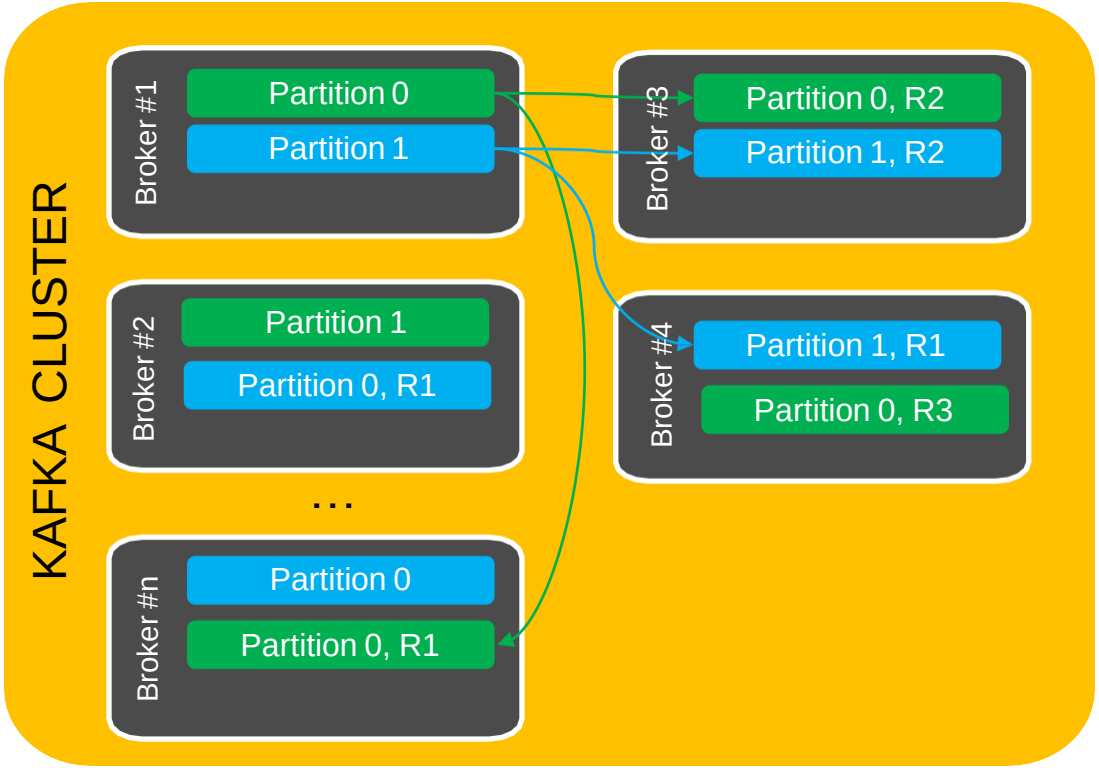


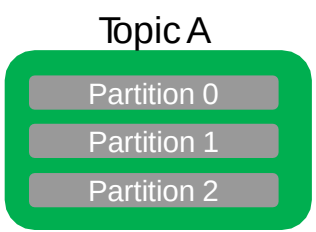
Broker – Partition – Replication

- Brokers manage partitions
- Specific partitions can be distributed among a number of brokers
- Brokers receive messages from producers and store them in appropriate topics, data streams and partitions
- Since partitions are distributed among brokers, if a broker from the cluster were to fail, data would be lost - Kafka implements partition replication to avoid this
- Each partition has a preconfigured number of replicas (replication factor) - so one of the partitions is the LEADER, the other replicas are the FOLLOWERS



Broker – Partition – Replication





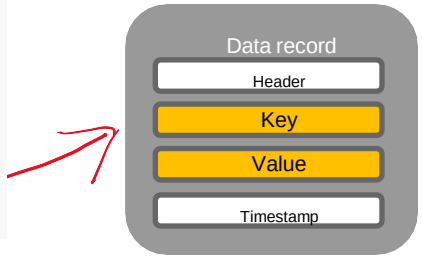
Semantic partitioning

- For the purpose of **load balancing** brokers, producers can control which partition the data will be stored on by specifying the message key (partitioning strategy)
  - Densification (key) % number of partitions
  - A message with the same key always ends up in the same partition
- If the key is not specified, a strategy, which uses the next partition in order, with the partitions following in order of preference, is used

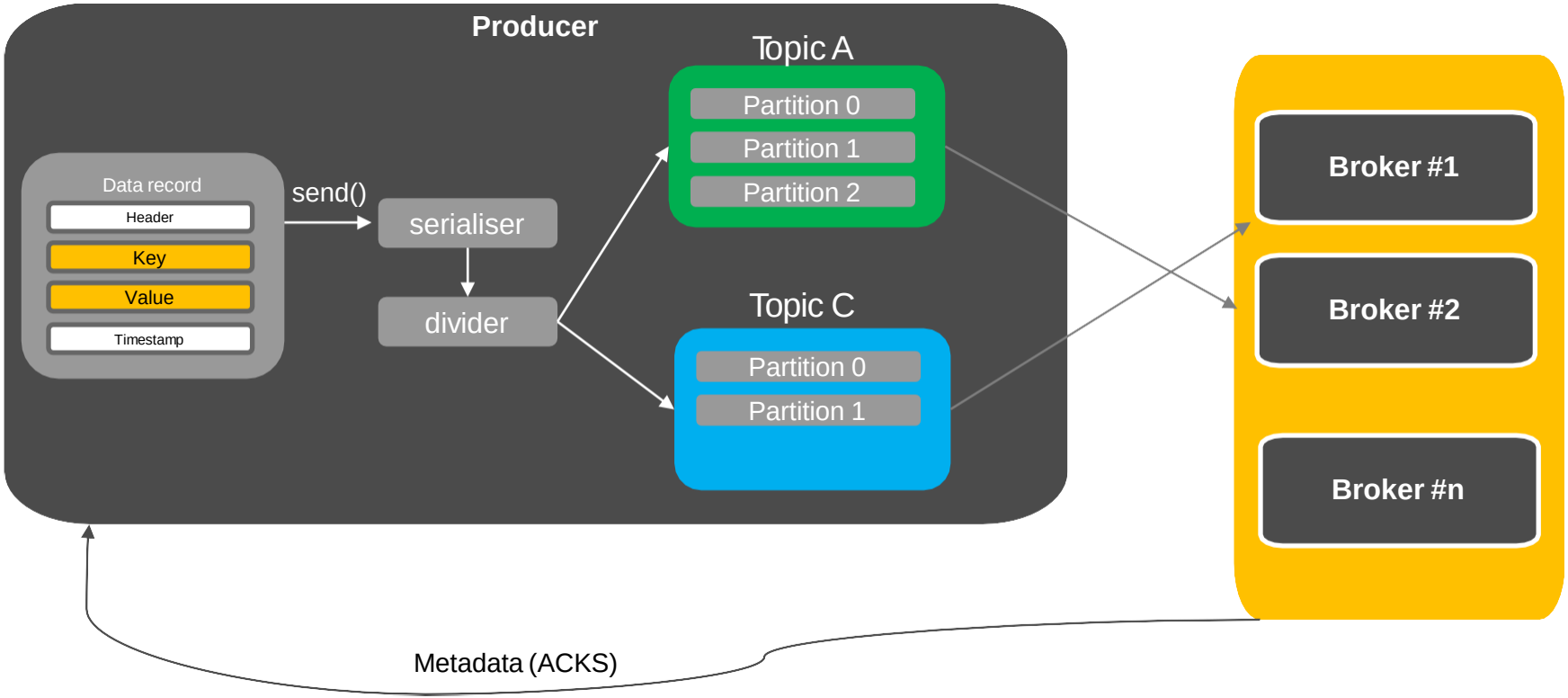
Example of a simple producer

```java
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class Producer {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers", "localhost:9092");
        properties.setProperty("key.serializer", StringSerializer.class.getName())
        properties.setProperty("value.serializer", StringSerializer.class.getName());

        KafkaProducer<String, String> producer = new KafkaProducer<String, String>properties);
        ProducerRecord<String, String> record =
                new ProducerRecord<String, String>("demo_topic", "Hello World");

        // Send data
        producer.send(record);

        // Tell producer to send all data and block until complete - synchronous
        producer.flush();

        // Close the producer
        producer.close();
    }
}
```

- Example of a simple Java producer



First steps of the producer



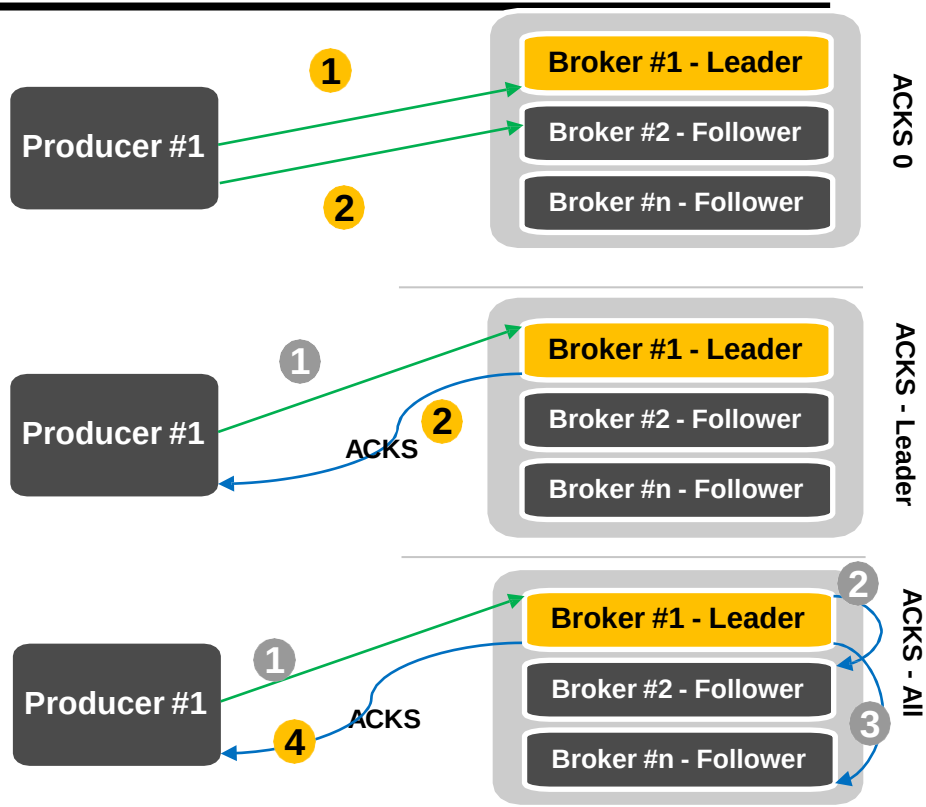
Producers' guarantee

- When sending data records, producers can **potentially wait** for a guarantee that the message has been received by the broker
- Data records are considered as processed when they are written to the appropriate partition on **one or all** synchronised brokers
- The search for a guarantee is configurable and can be chosen according to a business rule
- There are three main ways:
  - ACKS 0 (**none**) - do not wait for confirmation
  - ACKS 1 (**leader**) - wait for confirmation from the lead broker for the selected partition
  - ACKS -1 (**all**) - wait for confirmation from all brokers

Producers' guarantee

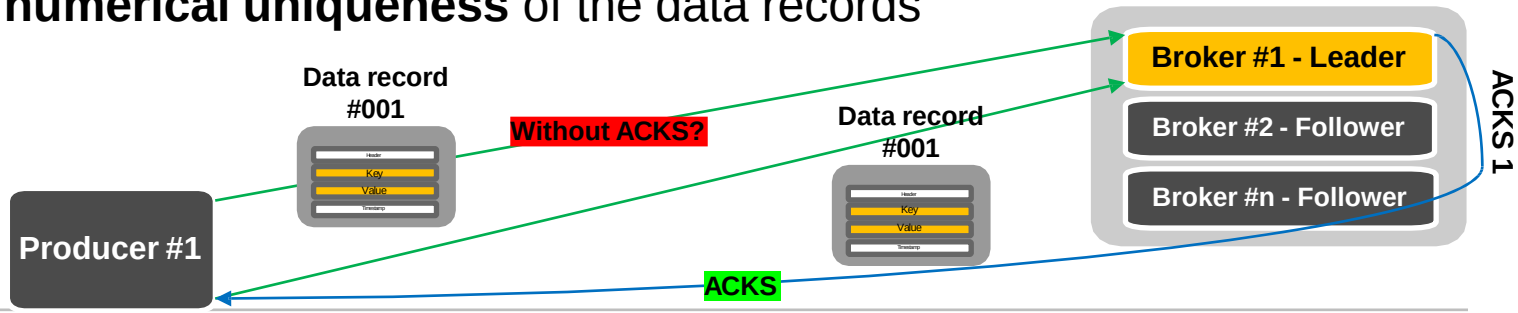
- Producers can choose whether they want to **receive confirmations** for each individual data record when the record is stored in its entirety at least on the leader or on all brokers

- ACKS – 0 - lower guarantee but no latency
- ACKS – ALL - higher guarantee - resulting in higher latency

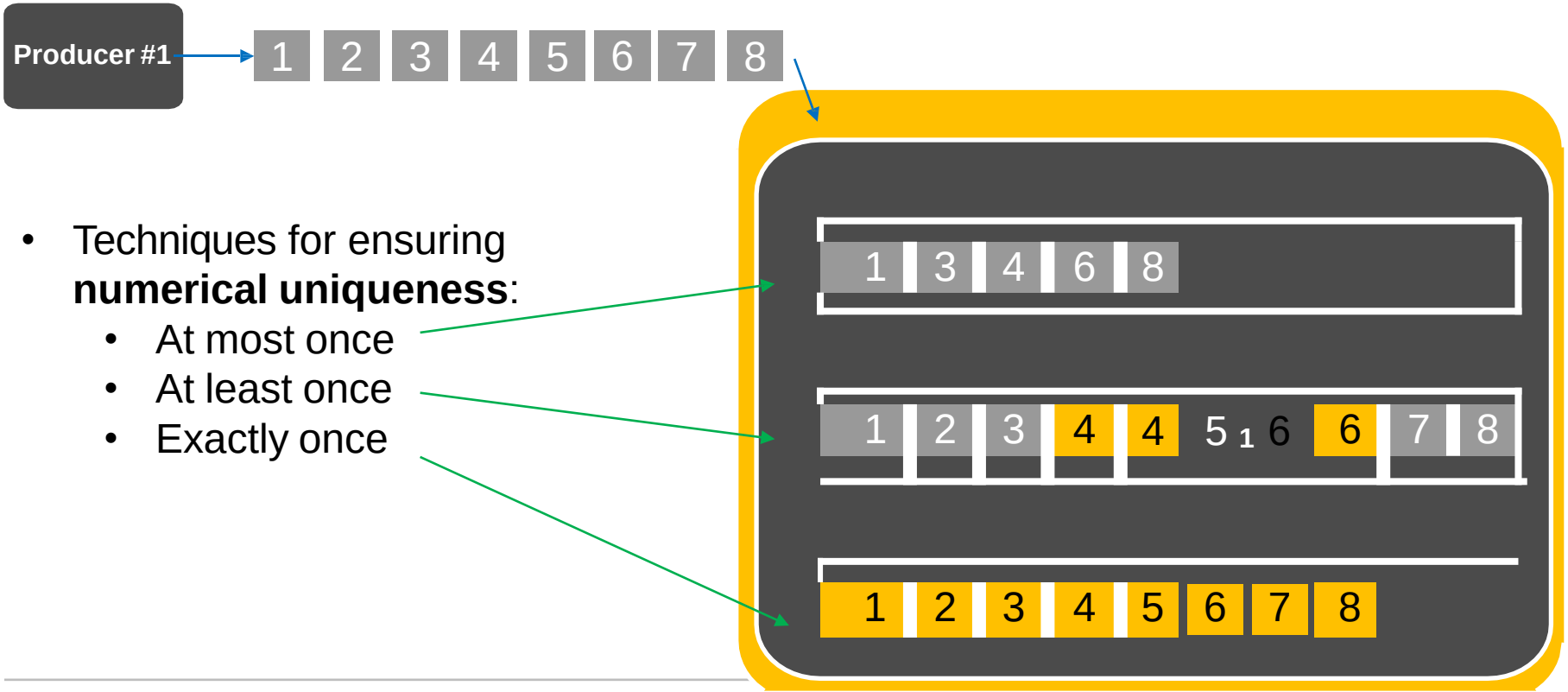


Delivery guarantee

- The delivery guarantee (ACKS 1 or ACKS -1) can lead to so-called duplicate data records on the brokers, which is sometimes not desirable from a business point of view
  - Duplicate records occur when the producer does not get back a guarantee (ACKS) that the data record has been properly stored. In this case, the producer shall retry the transaction
    - This may be caused e.g., by a sudden failure of the broker
- Apache Kafka allows the editing of the delivery guarantee and at the same time the **numerical uniqueness** of the data records



Guarantee of (numerical) uniqueness



- Techniques for ensuring **numerical uniqueness**:
  - At most once
  - At least once
  - Exactly once

Guarantee of (numerical) uniqueness

- Achieving **exactly once** guarantees in distributed systems is very challenging
- Exactly once means that the system will not miss data records and at the same time will not process duplicate data records

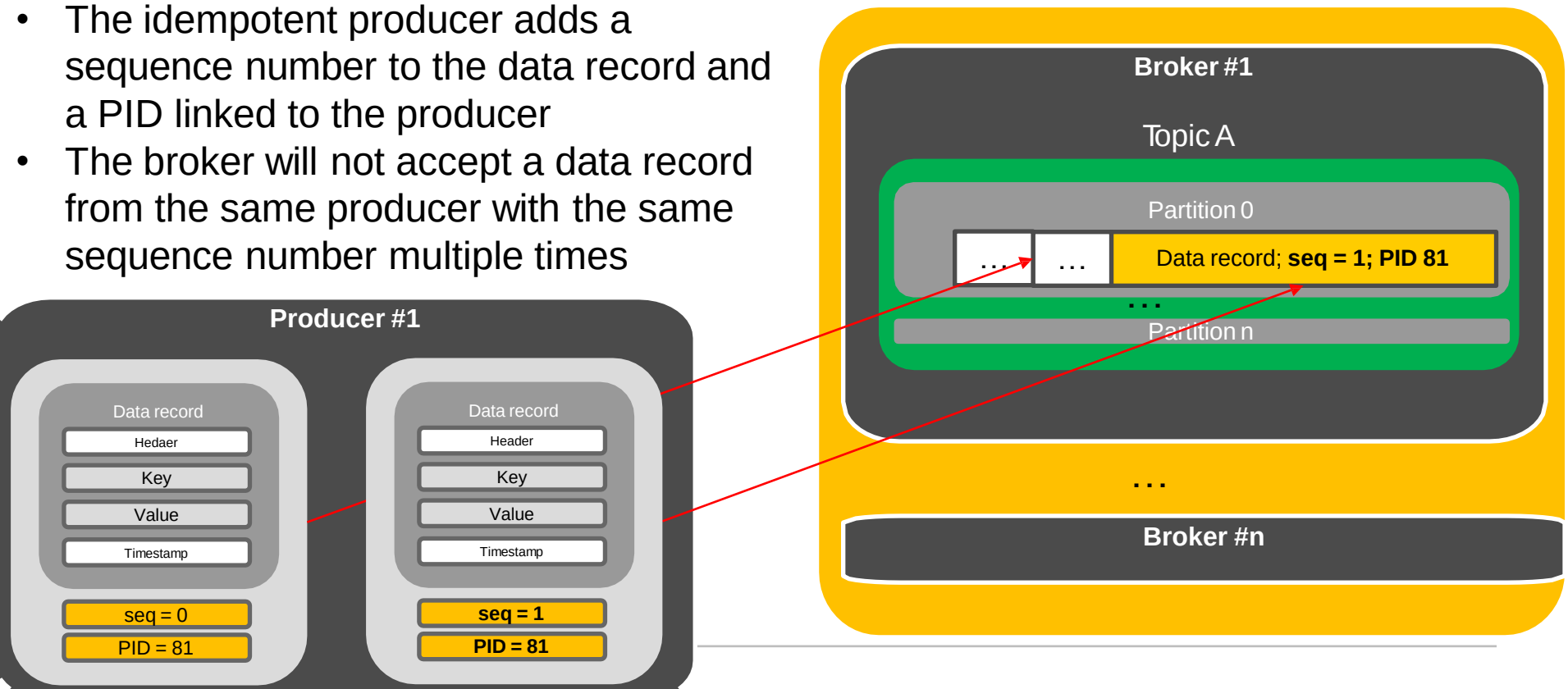
- Kafka can provide the "exactly once" guarantee in three ways:
  - **Idempotent producer**
  - **Transactional Producer/API**
  - **Streams processing exactly once** (Streams API)
    - processing.mode = “exactly_once”

Idempotency

- Idempotency is the **property of certain operations** in computing whereby they can be performed repeatedly without their repeated execution affecting the final result
  - A simplified example: **repeatedly pressing** a button on a pedestrian traffic light **does not affect the initial press**, which itself activated the underlying operation
- We usually consider idempotent functions to be those of the "changeAddress" type of the client, since even if the function were executed three times by mistake, it would not change the final correct state. Otherwise, functions of type "executeOrder" require additional checking logic

Idempotent producer

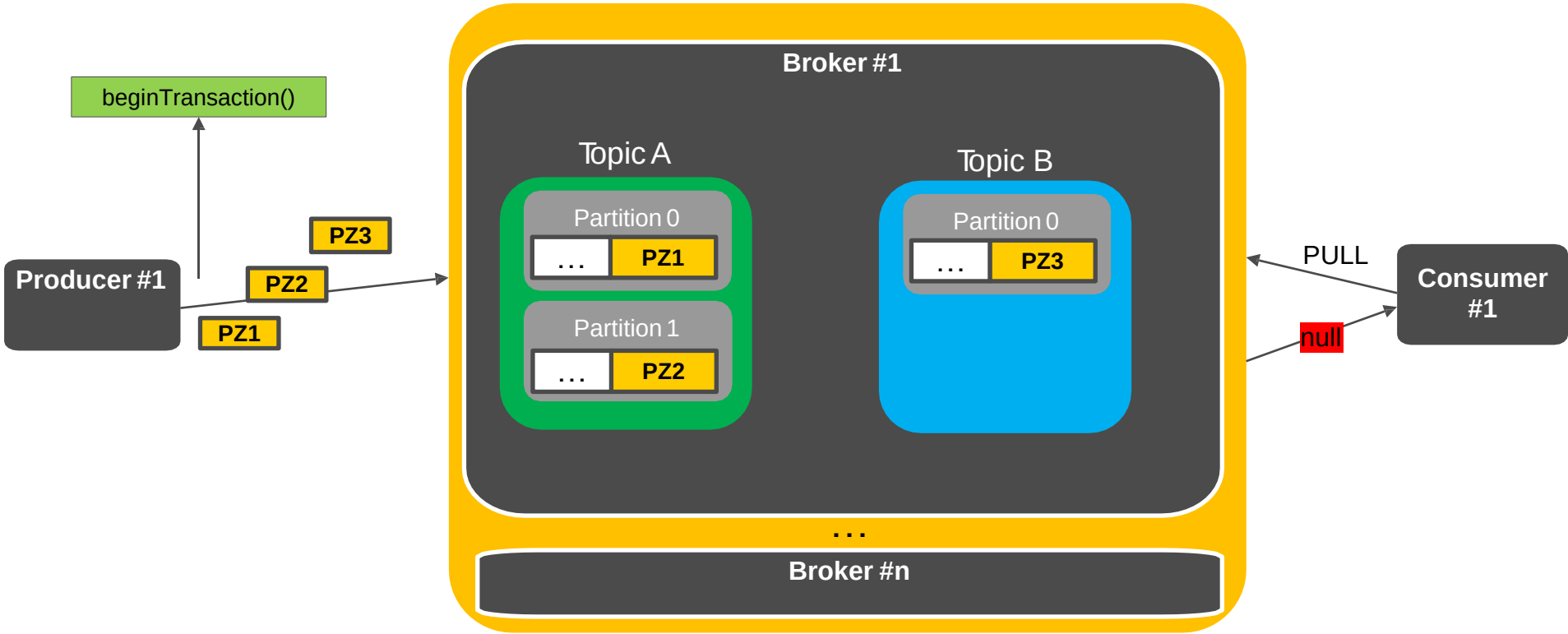
- The idempotent producer adds a sequence number to the data record and a PID linked to the producer
- The broker will not accept a data record from the same producer with the same sequence number multiple times



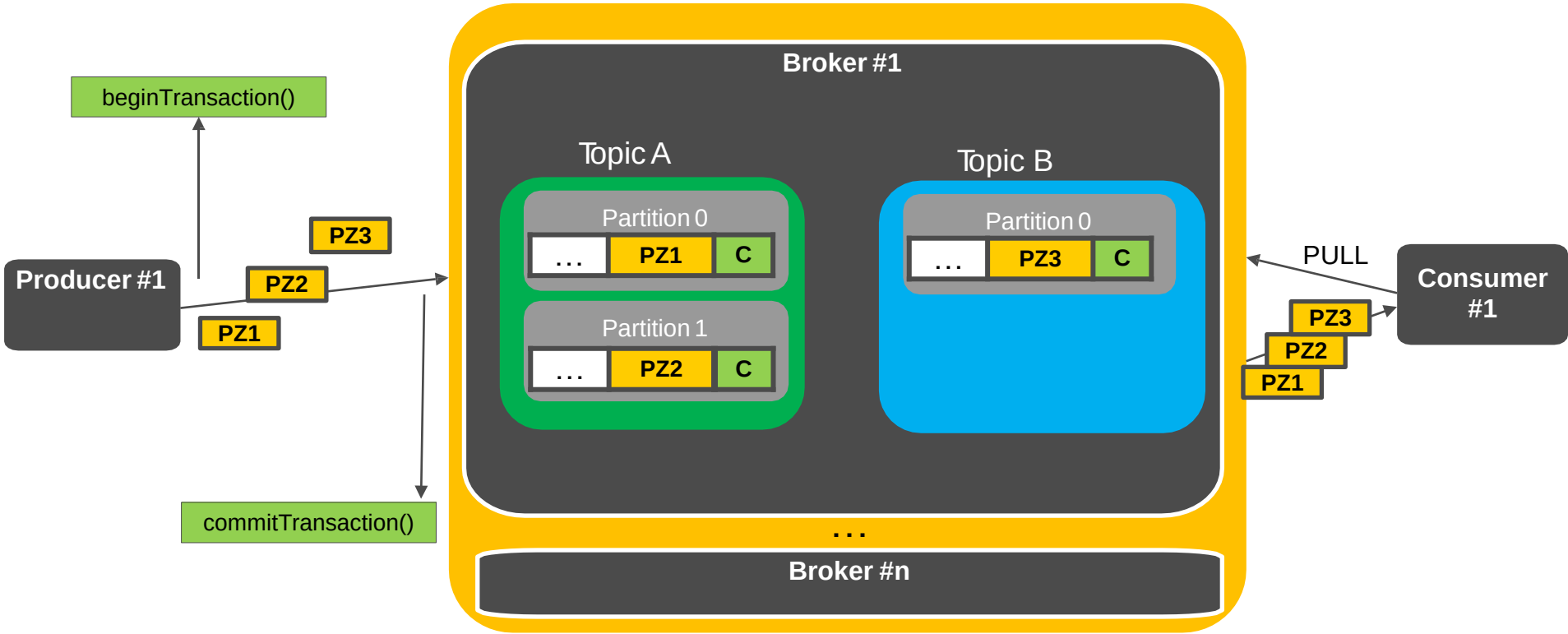
Transactional producer

- Transactional producer or **atomic multi-partition writes**
- Operation similar to TCL (Transaction-Control-Language) in the context of relational databases
- A producer can combine many data records (of the same or different topics) into a single transaction. Once the "COMMIT" has been transmitted, consumers will be able to read them

Transactional producer



Transactional producer



Transactional producer

```java
Properties nastavitve = new Properties();
nastavitve.put("client.id", "basic-producer-v0.1.0");
nastavitve.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092");
nastavitve.put("key.seralizer", "org.apache.kafka.common.serialization.StringSerializer");
nastavitve.put("value.serializer", "org.apache.kafka.common.serialization.StrngSerializer");

final KafkaProducer<String, String> transakcijskiProizvajalec = new KafkaProducer<>(nastavitve);

final String tema = "PMiB";

final ProducerRecord<String> sporocilo1 = new ProducerRecord<>(tema, "VREDNOST1");
final ProducerRecord<String> sporocilo2 = new ProducerRecord<>(tema, "VREDNOST2");

transakcijskiProizvajalec.initTransactions();

try {
    transakcijskiProizvajalec.beginTransaction();

    transakcijskiProizvajalec.send(sporocilo1);
    transakcijskiProizvajalec.send(sporocilo2);

    transakcijskiProizvajalec.commitTransaction();
}
catch(KafkaException e){
    transakcijskiProizvajalec.close();
}
```

Consumers - Reading

- **Consumers** are instances of software code that are developed on demand according to business logic
  - Kafka is consumer (as well as producer) agnostic – independent of platform and programming language
- Consumers read (**PULL**) messages of different **topics** - possibly several at the same time
- For the purpose of reading fresh/recent/new messages, consumers keep the **last offset**
- As there can be several consumers for a given topic, each offset is linked to a consumer - so each consumer has their own offset for each of the topics they read
- Based on the offset, brokers know which new messages the consumer has not yet read/processed

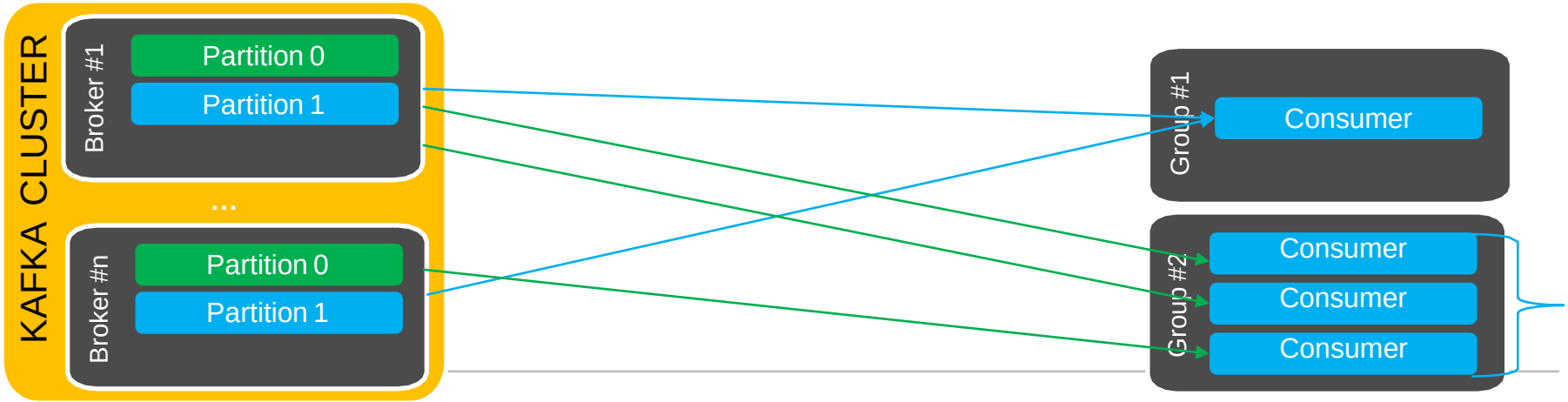
Example of a simple consumer

```java
import java.util.Properties;
import java.util.Arrays;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class SimpleConsumer {
    public static void main(String[] args) throws Exception {

            String tema = args[0].toString();

            Properties nastavitve = new Properties();

            nastavitve.put("bootstrap.servers", "localhost:9092");
            nastavitve.put("group.id", "PMiB-skupina-1");
            nastavitve.put("enable.auto.commit", "true");
            nastavitve.put("auto.commit.interval.ms", "1000");
            nastavitve.put("session.timeout.ms", "30000");
            nastavitve.put("key.deserializer", "org.apache.kafka.common.serializa-tion.StringDeserializer");
            nastavitve.put("value.deserializer"," org.apache.kafka.common.serializa-tion.StringDeserializer");

            KafkaConsumer<String, String> potrosnik = new KafkaConsumer<String, String>(nastavitve);

            potrosnik.subscribe(Arrays.asList(tema));

            while (true) {

                ConsumerRecords<String, String> sporocila = potrosnik.poll(100);

                for (ConsumerRecord<String, String> sporocilo : sporocila)

                    System.out.printf("ODMIK = %d, KLJUČ = %s, VREDNOST = %s\n",
                            sporocilo.offset(), sporocilo.key(), sporocilo.value());
            }
        }
    }
```


- Example of a simple Java consumer

Consumer groups

- Consumers form **consumer groups**
- Represents one final application/system
  - An isolated consumer is a consumer group in itself, even if alone



Consumer groups

- Represents one final application/system
  - Each instance of the same application/system is a consumer for itself (e.g., instantiation using Kubernetes)
  - The consumer group is configured in the consumer settings
  - Allows easier scalability at consumer level

```
Properties nastavitve = new Properties();
nastavitve.put("client.id", "basic-producer-v0.1.0");
nastavitve.put("bootstrap.servers", "kafka-1:9092, kafka-2:9092");
nastavitve.put("key.seralizer", "org.apache.kafka.common.serialization.StringSerializer");
nastavitve.put("value.serializer", "org.apache.kafka.common.serialization.StrngSerializer");
nastavitve.put("group.id","MOJID");
```

Security

- Support for data encryption in transit (SSL)
  - Producer – Broker
  - Broker – Consumer

- Support for authentication of consumers and producers

- Support for authorisation of consumers and producers on the broker's tier using Access Control Lists (ACLs)
  - **kafka-acls** command

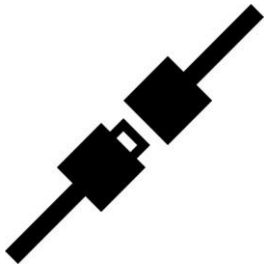
- No security support at the physical level of brokers - data is not encrypted

Main programming interfaces

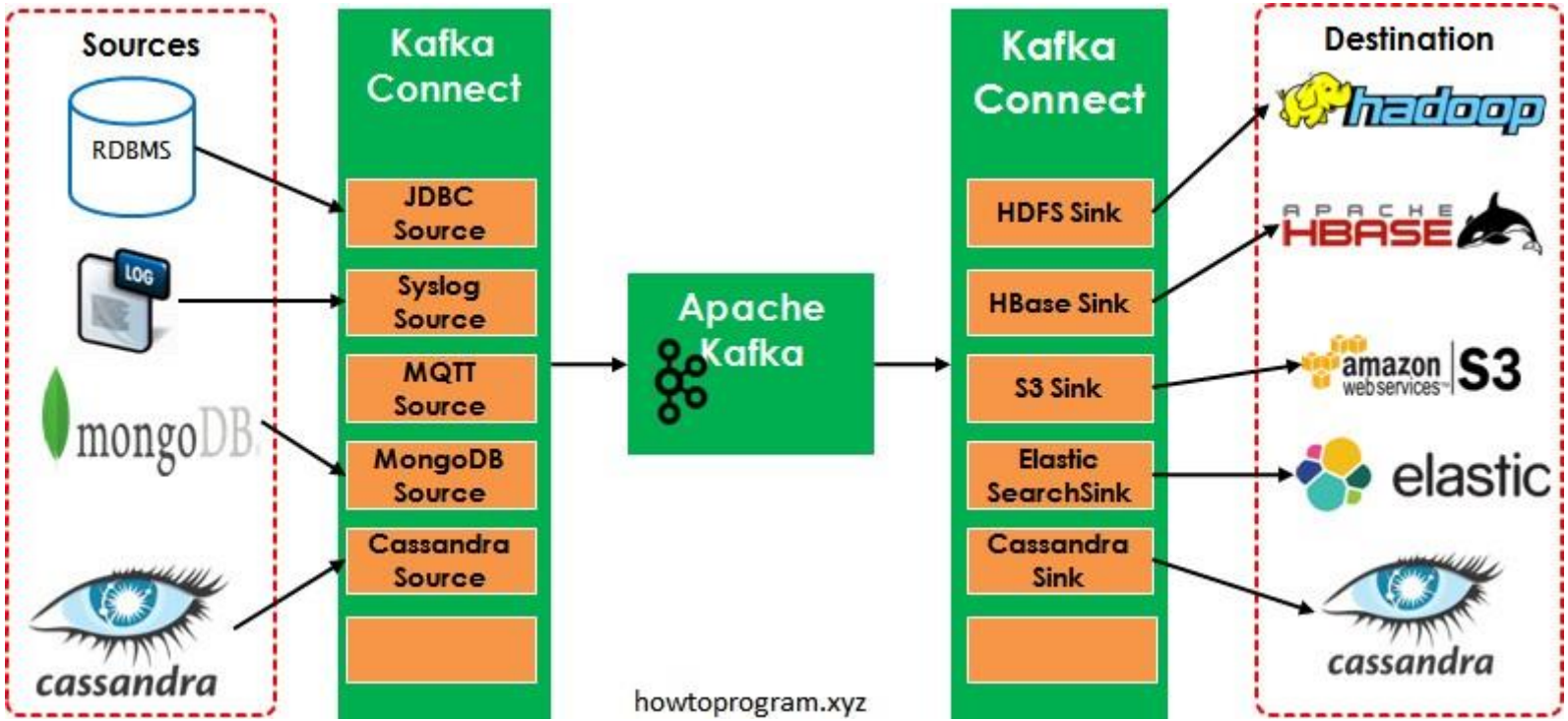
- Kafka includes five main software interfaces that make the Kafka ecosystem easier to manage and use:
  - Producer API (*examples on previous pages*)
  - Consumer API (*examples on previous pages*)
  - Connect API
  - Streams API
  - Admin API

Connect API

- Apache Kafka Connect is a tool for scalable and reliable data streaming between Apache Kafka and other systems (repositories and data sources).
- It allows you to easily define **connectors** that move large datasets into and out of Kafka to external systems (e.g., data lakes)
  - A Kafka connector can ingest entire databases of external sources into entire **topics**
- It is a separate server infrastructure, which can also be distributed (n workers)
- It contains libraries for many connectors (e.g., JMS, mqtt, elastic, AWS)
- Distinguishes:
  - Common framework for Kafka connectors
  - REST interface
  - Automatic offset control
  - Integration for streaming as well as batch processing



Connect API

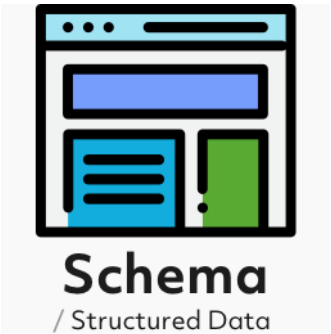

Sources
RDBMS
LOG
mongoDB.
cassandra
Kafka Connect
JDBC Source
Syslog Source
MQTT Source
MongoDB Source
Cassandra Source
Apache Kafka
howtoprogram.xyz
Kafka Connect
HDFS Sink
HBase Sink
S3 Sink
Elastic SearchSink
Cassandra Sink
Destination
hadoop
APACHE HBASE
amazon web services S3
elastic
cassandra

The need for Connect API

- It is necessary to distinguish between the Producer/Consumer API and the Connect API, as they have very similar tasks
- **Producer/Consumer API**
  - Used as developers when we want to connect our service/application to Kafka and have full control over the source or output system;
  - As clients (consumers) we process and use the data we get from Kafka for business purposes and by default do not store it further in data stores
- **Connect API**
  - Used when we want to connect external data stores to Kafka that we do not directly control and cannot modify the source code of, or when we want to transfer/store sub-acts from Kafka to additional systems/stores

Schema registry

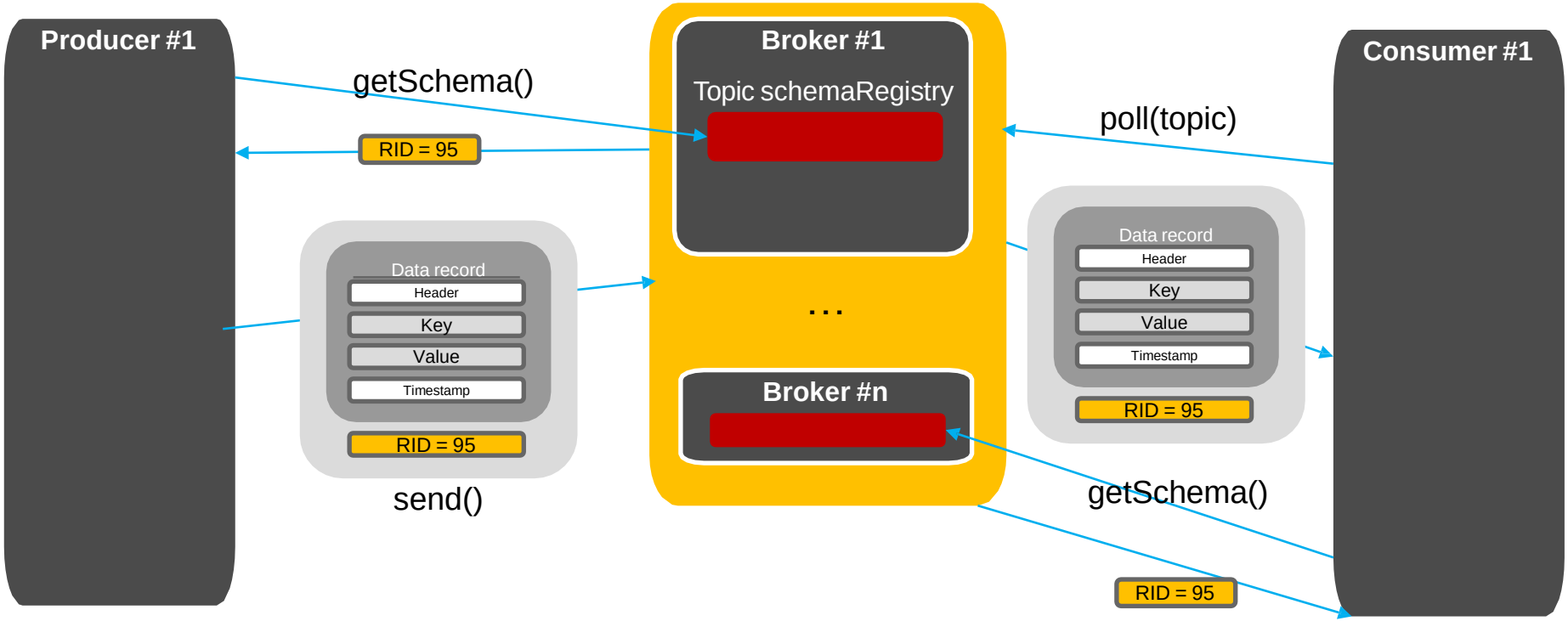
- Within the Kafka ecosystem, we may have **many producers and consumers**, which may represent **different teams of developers** and are usually already **separated** from each other
- We can have **many and different topics** whose structure is either simple or complex
- In such environments it is essential to ensure **compatibility of data records**, which in turn facilitates the evolution of the data schema
- Example:
  - We have 1 producer and 5 different consumers, and all consumers can be informed about the change of the data schema to avoid inconvenience



Schema registry

- Using the Kafka Schema Registry, we define a schema for the data records that will be part of a particular topic
- The schema is defined using an IDL (Interface Description Language), such as AVRO
- Schema registrations are enabled by:
  - defining the expected fields for each Kafka topic
  - automatic management of schema changes
  - backwards compatibility management
  - support multi-data environments from different sources

Schema registry



Real-time processing

- Real-time processing of data, in a continuous, concurrent and per-record (event) manner, is performed using the **Kafka Streams API** or **KSQL**

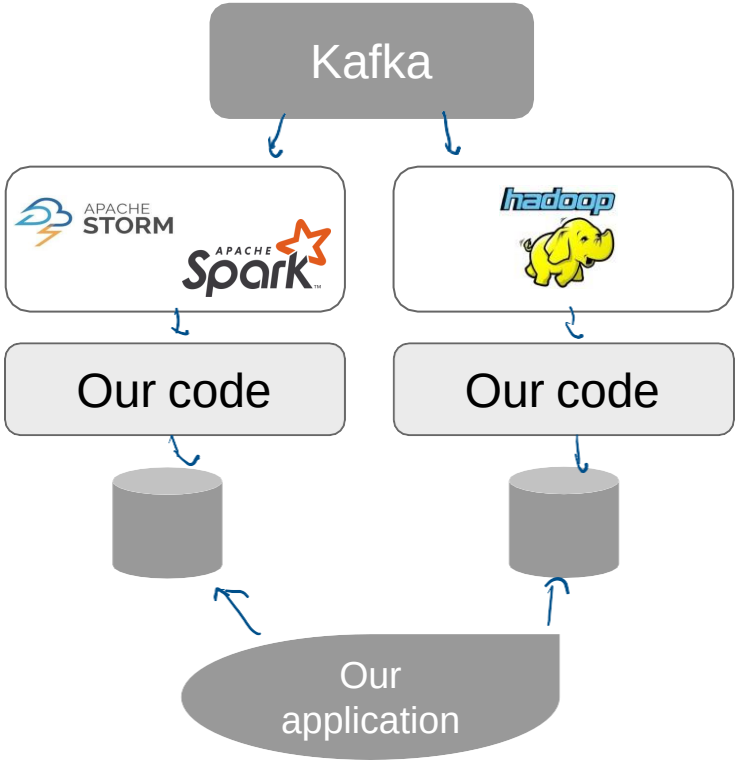
- Apart from being a **stable messaging system**, real-time data processing is one of the central features of Kafka

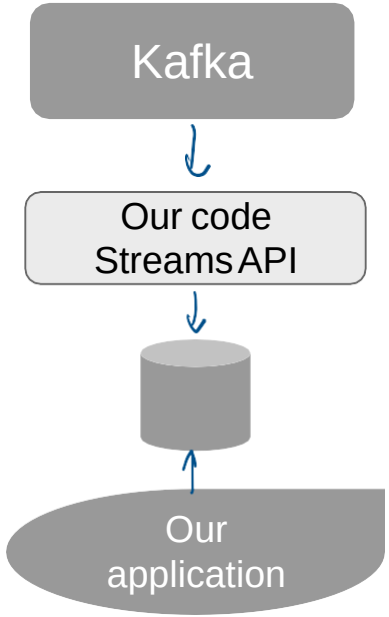
- Kafka basically processes a continuous stream of data. Real-time processing typically involves **reading continuously arriving data records of a particular stream/topic**, **performing** some **analysis** or **transformation** on the data, and **then writing the results** of this **to another or a new stream/topic**

Real-time processing

- There are several ways to implement real-time processing with Kafka:
  - The classical way is to develop a Consumer, which processes the data and then sends it on to a new stream with the help of a Producer. This way is very demanding as a lot of dedicated processing is needed for stable state processing
  - The mode where we use a full-fledged stream processing framework such as **Spark** Streaming, **Flink**, **Storm**, etc. is a very expensive way to process real-time streams.
  - The native way for Kafka. Namely using the **Kafka Streams API** or **KSQL**

Real-time processing



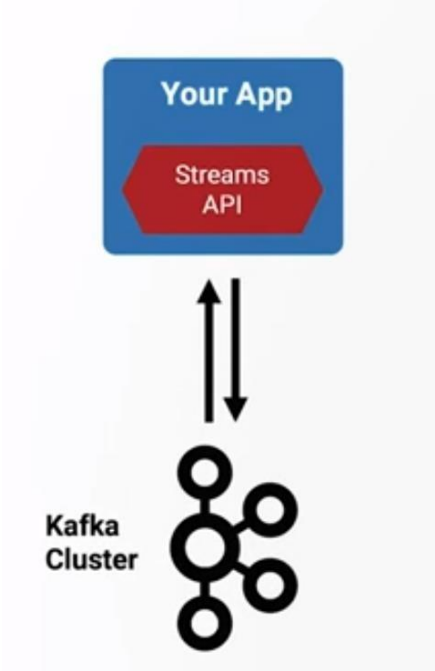


Streams API

- Apache Kafka Streams is the main component that **enables real-time stream data processing**
- It is a client library for building applications and **microservices** where input and output data are stored in Kafka brokers
- It allows a higher level of abstraction and **transformation** and **enrichment** of data **during streaming**
- It features:
  - (1) millisecond obfuscation, (2) elasticity, (3) scalability, (4) fault tolerance, (5) container deployability, (6) integration with Kafka security, (7) support for standard Java and Scala applications, (8) support for exactly-once processing, ...

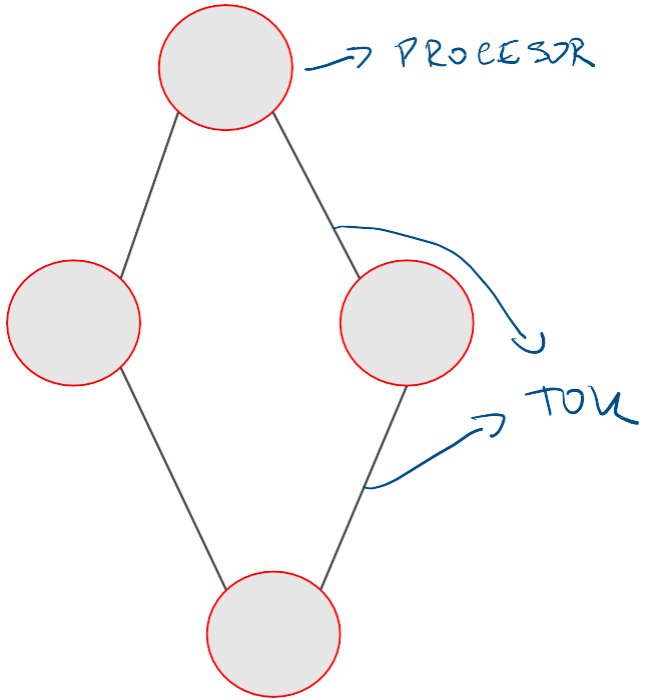
Streams API

- Apache Kafka Streams as a library is implemented within our application and not within brokers
- Apache Kafka Streams allows:
  - Process each individual data record during streaming with a millisecond latency
  - Grouping the stream by key
  - Aggregation of multiple streams
  - Continuous transformation
  - Stateless processing
    - filtering, mapping
  - Stateful processing
    - aggregations
  - Temporal windowing
    - Processing according to a fixed time interval

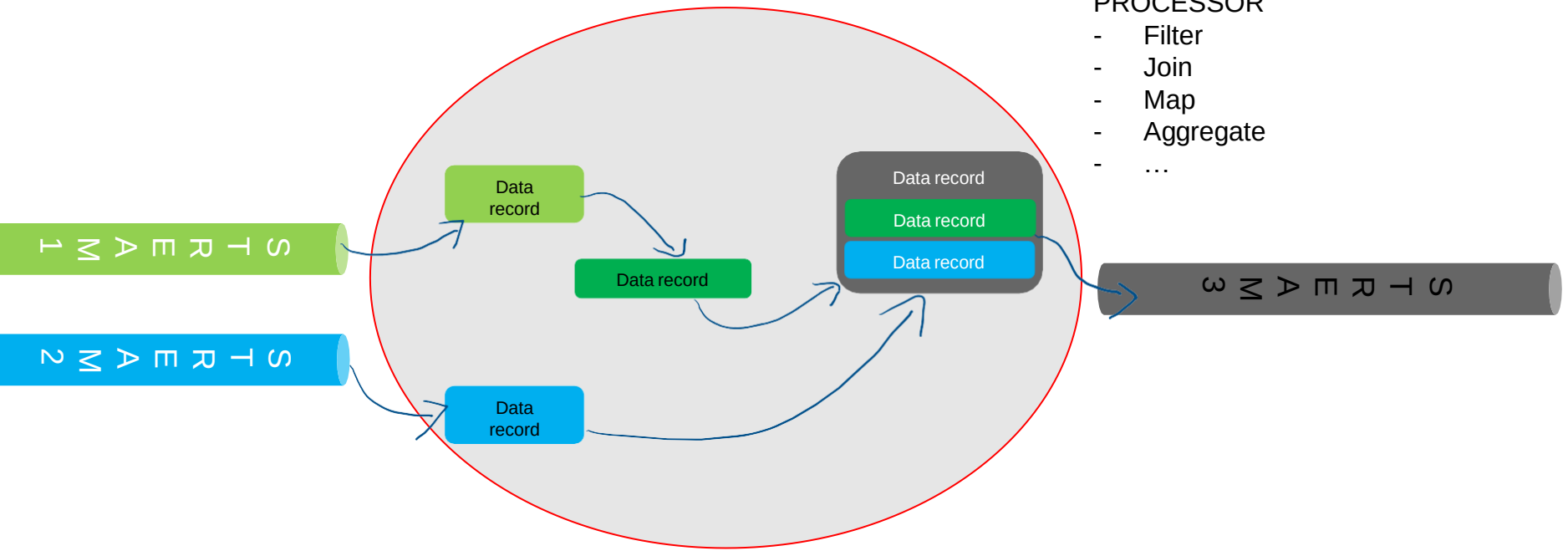


Process topology

- The processor topology **defines the data processing logic** to be executed by the streams application
- In a stream processor topology, there is a node called the stream processor. It represents the processing step to convert data into streams by accepting one input sub-record from its upstream processors in the topology. It also subsequently produces one or more output records for its downstream processors



Streams API

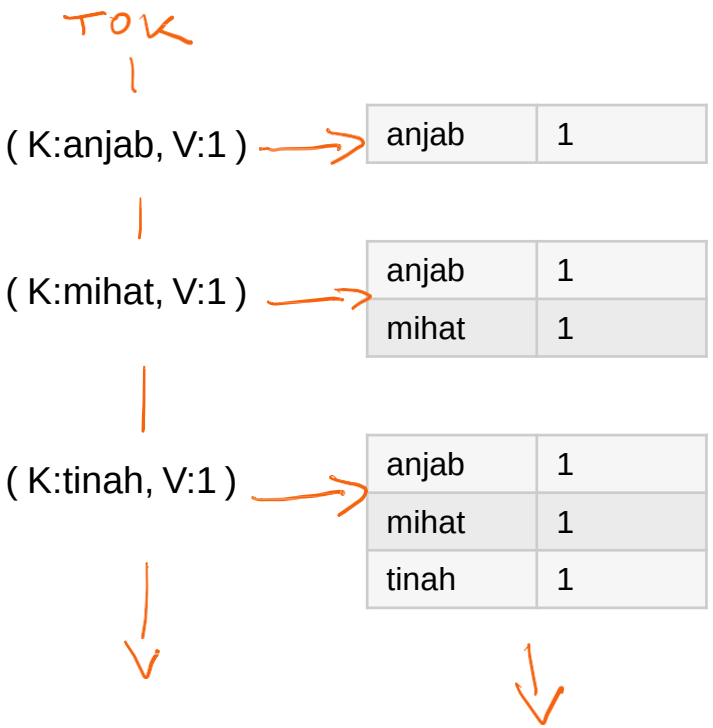

PROCESSOR
- Filter
- Join
- Map
- Aggregate
- …
STREAM 1
STREAM 2
STREAM 3
Data record
Data record
Data record
Data record
Data record
Data record

Duality of flows and tables

- In practice, we usually need both streams and databases when processing streams
- An example of an application that is very common in practice is an e-commerce application where the incoming customer transaction stream is enriched with the latest customer information from a database table
- The Kafka Streams API provides support for processing streams as tables using its basic abstractions **KStream** and **KTable**
- There is a close relationship between streams and tables, the so-called stream-table duality
- Kafka exploits this feature in several ways: (1) to improve application elasticity, (2) to support fault-tolerant processing, (3) to perform interactive queries, (4) etc.

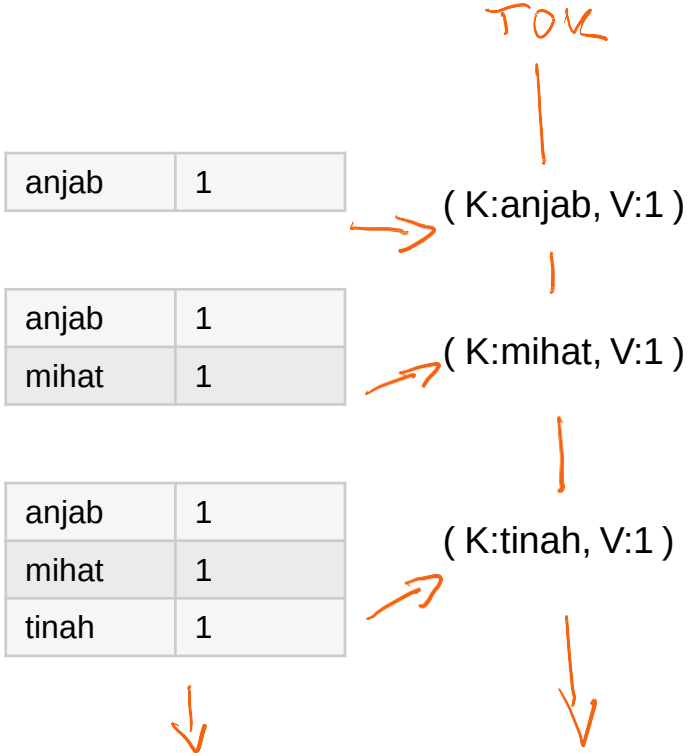
Duality of flows and tables

- The duality of flows and tables means that a flow can be seen as a table and a table as a flow
- **Flow as a table**
  - A stream can be viewed as a table change log, where each data record in the stream captures a change in the state of the table. A stream is therefore a table in disguise, which can easily be transformed into a "real" table by playing the change log from start to finish the reconstruction of the table. Similarly, merging data records in a stream will return a table, e.g., the total number of page views per user can be calculated from the input page view event stream, and the result would be a table with the user as the table key and the corresponding number of page views as the value



Duality of flows and tables

- **Table as a flow**
  - The table can be considered as a snapshot of the most recent value of each key in the stream at a given time (stream data records are key-value pairs). The table is therefore a transformed stream and can easily be turned into a "real" stream by iterating over each key-value entry in the table



KStream

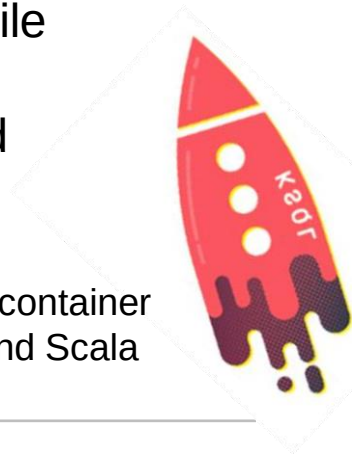
- **KStream** is an object used to abstract a stream (topic), which can be used to perform certain operations such as filtering and mapping (non-volatile operations)
- KStream is used when events in a stream are viewed as independent immutable facts that are constantly updated
    - Example: an event in a sports match (e.g., goal, pass, corner). Events are constantly occurring, and a new event does not overwrite the previous one

KTable

- **KTable** is an abstraction of a change log stream, where each data record represents an update. More specifically, a value in a data record is interpreted as an "UPDATE" of the last value for the same record key. If the corresponding key does not yet exist, the update is considered as an "INSERT" entry
- Compared to an event stream, a table represents the state of the world at a given moment in time, usually the freshest state
  - Example: Statistics of a football match (e.g., Tavares: shots on target:3; assists:5). The table is a view of the flow of events and this view is constantly updated when a new event is captured
- We perform certain operations, such as join and aggregate, to create so-called enriched streams

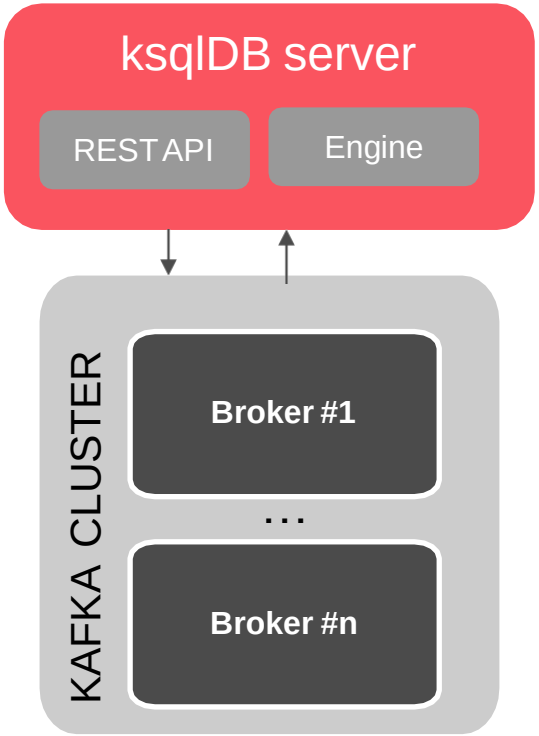
KSQL - ksqlDB

- ksqlDB is built on top of Kafka Streams and is a robust framework for processing streams that are part of Kafka. It offers a query layer to build applications for streaming events on Kafka topics
- While Kafka Streams allows writing some complex topologies, requiring some essential programming knowledge, KSQL aims to abstract this complexity by providing us with the familiar semantics of SQL, keeping in mind that we are not talking about batch SQL but **streaming SQL**, i.e., querying while streaming data
- Similarly to Streams API, it allows a higher level of abstraction and **transformation** and **enrichment** of data **during streaming**
- Distinguishing features:
  - (1) millisecond obfuscation, (2) elasticity, (3) extensibility, (4) fault tolerance, (5) container deployability, (6) integration with Kafka security, (7) support for standard Java and Scala applications, (8) support for exactly-once processing, ...



ksqlDB

- Unlike the Streams API, which runs as part of our application, ksqlDB runs as a database that is installed in parallel alongside and in close connection with the Kafka cluster
- Since ksqlDB is automatically executed on Apache Kafka, it is therefore necessary to install Kafka. Kafka must be configured to use ksqlDB



KSQL – ksqlDB – Use cases

- **KSQL** can be used for a number of operations on streams that make our lives easier, namely:
  - **Data exploration** (events, data records)
    - Example: overview of all topics, extraction of data of a specific topic

```
SHOW TOPICS;
PRINT 'topic' FROM BEGINNING;
```

  - **Extract-Transform-Load (ETL) streaming**
    - Example: analysis and transformation of events in real time

```
CREATE STREAM newstream AS
SELECT userid, page, action FROM clickstream c
LEFT JOIN users ON c.userid = u.user_id
WHERE u.type = 1;
```

KSQL – ksqlDB – Use cases

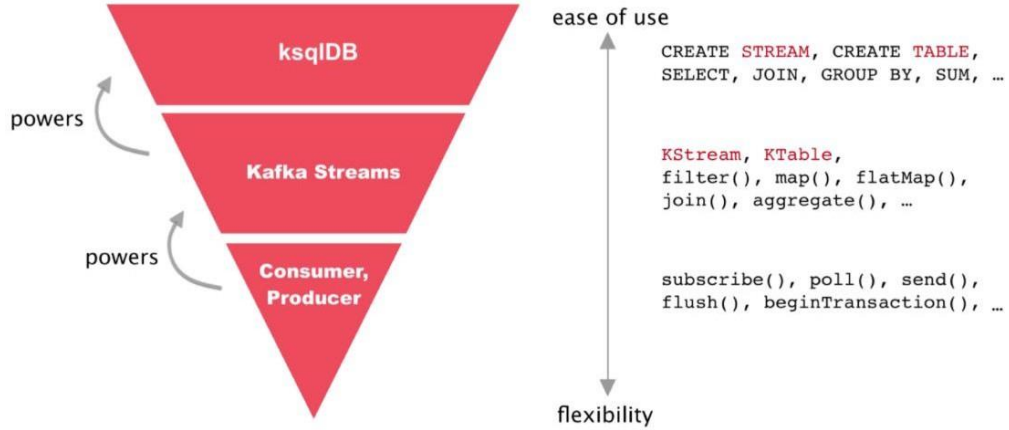
- **KSQL** can be used for a number of operations on streams that make our lives easier, namely:
  - **Anomaly detection** (identifying patterns or anomalies)

```
CREATE TABLE potentialFraud AS
SELECT card_number, COUNT(*)
FROM verification_attempts
WINDOW TUMBLING (SIZE 5 SECONDS)
GROUP BY card_number
HAVING COUNT(*) > 3;
```

  - **Real-time monitoring**
  - **Data transformation**
  - **…**

KSQL – Streams API

- Depending on the business needs of our application, we can either use simple Kafka software elements such as producers and consumers, or we can switch to using Streams API or KSQL, which help us process data in real-time at different levels of abstraction



ksqldb.io

Kafka ecosystem