

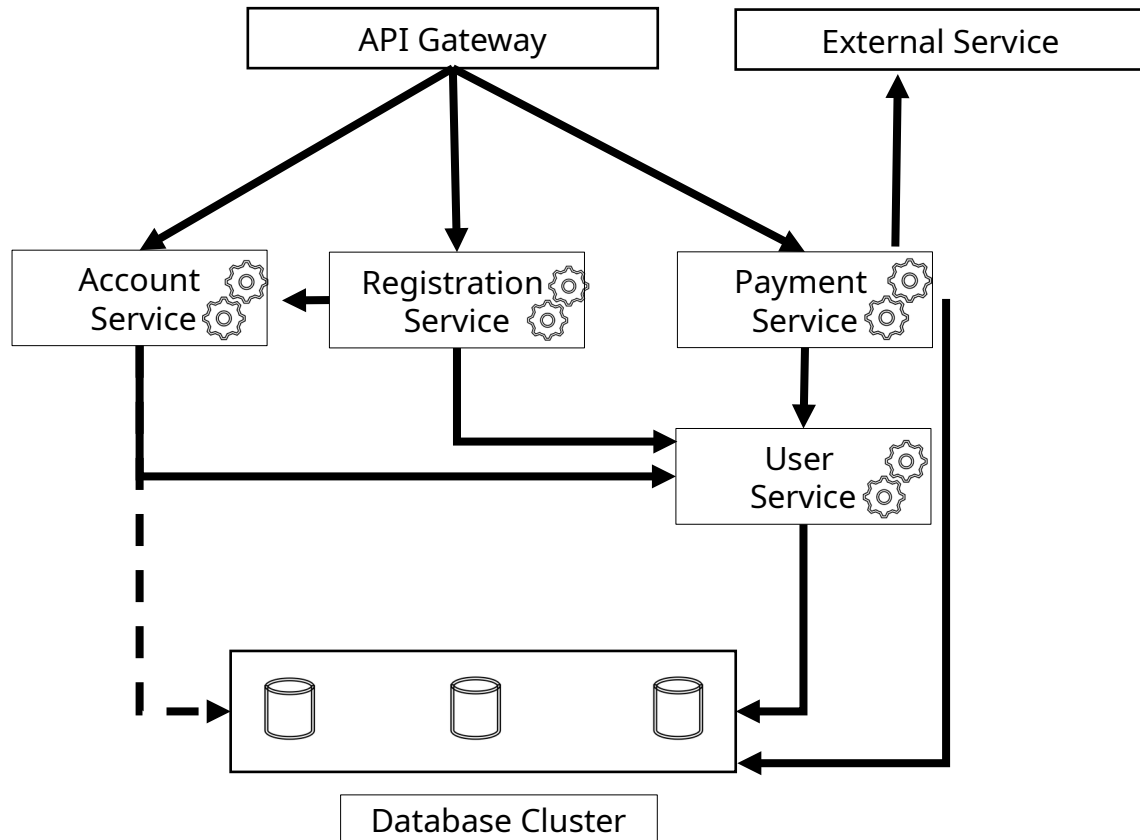


THINKPORT
CLOUD CONSULTING



EVENT DRIVEN ARCHITECTURE

DISTRIBUTED SYSTEMS



Benefits

1. Scalability
2. Reusability
3. Time to Market
4. Flexibility
5. Automation

Challenges

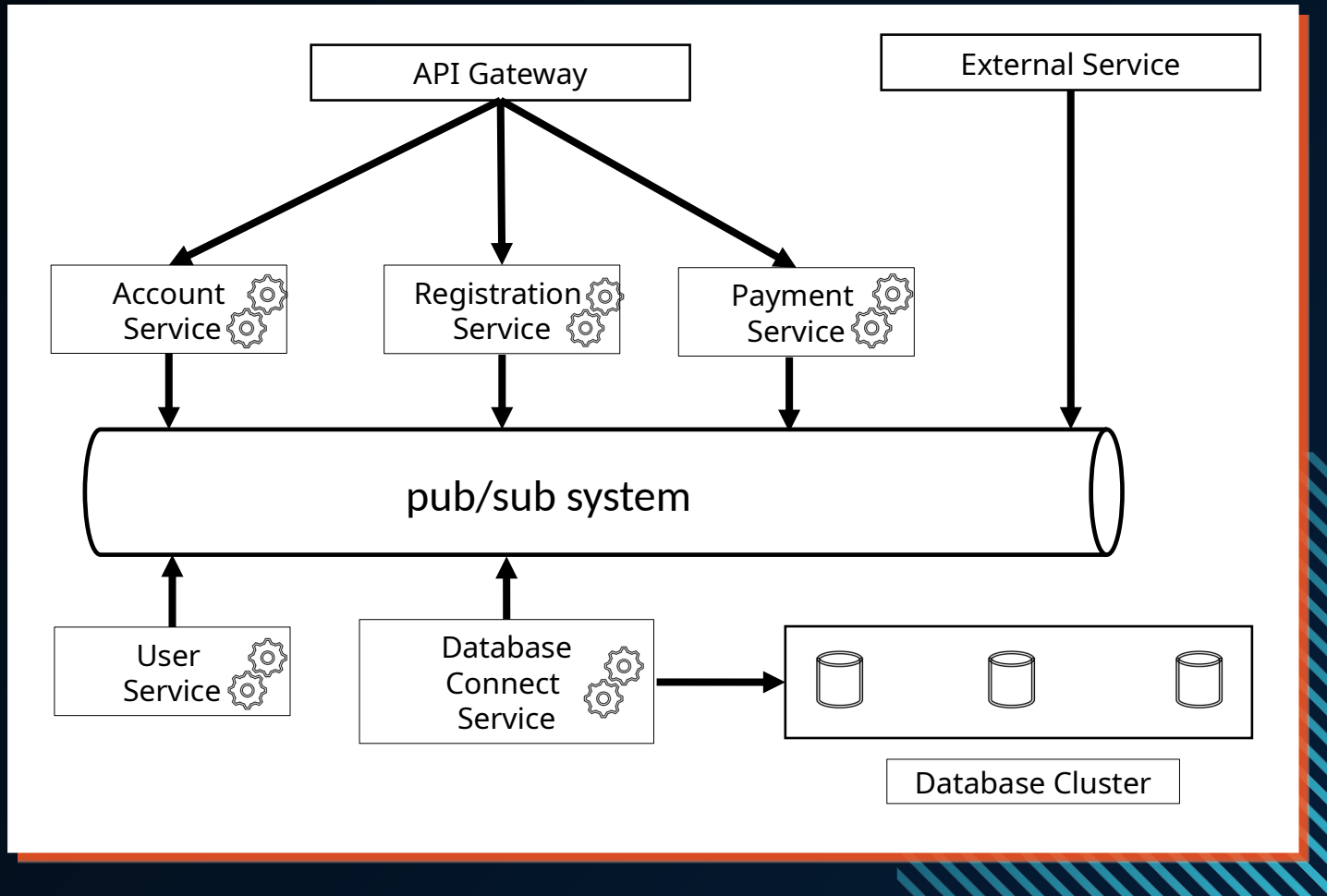
1. Tight Coupling
2. Data Consistency
3. Operation Overhead
4. Data Ownership
5. Blocking
6. Complex Team Communications
7. Complex system tests

Microservice Architecture



Reactive Programming

PUB/SUB systems



ADVANTAGES

1. Loose Coupling
2. No Bottleneck
3. Eventual Consistency
4. Data Ownership
5. Reactive Data Publishing
6. Non-Blocking
7. Processing on-demand
8. Short response times (reaction time)

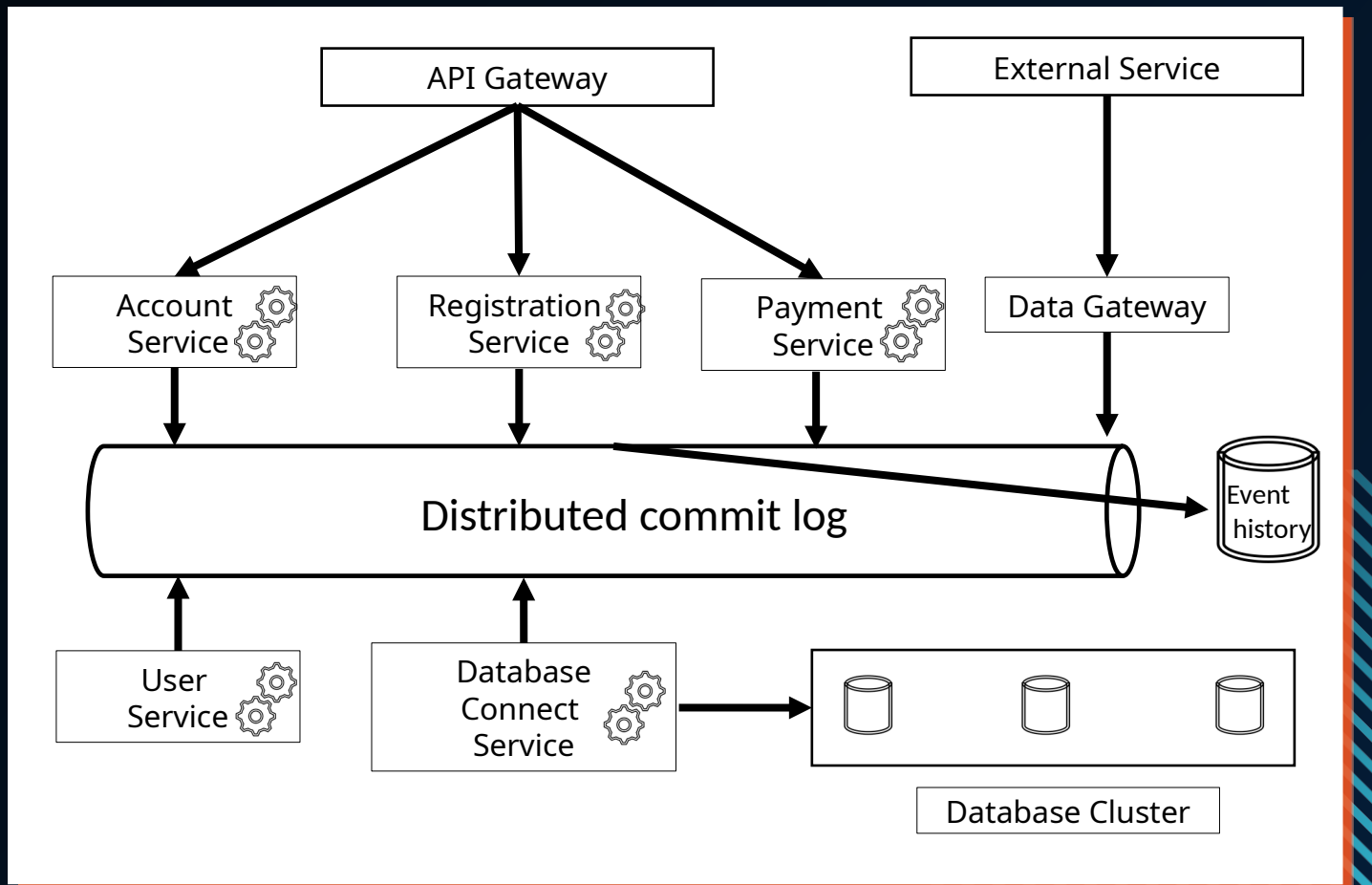
Microservice Architecture



EVENT SOURCING

- Is a pattern for storing data as events in an append-only log
- By storing the events, it also keep the context of the events
- An event represents a fact that took place
- Every change made is represented as an event, and appended to the event log
- An entity's current state can be created by replaying all the events in order of occurrence
- The system information is sourced from the events

EVENT SOURCING WITH A COMMIT LOG



ADVANTAGES

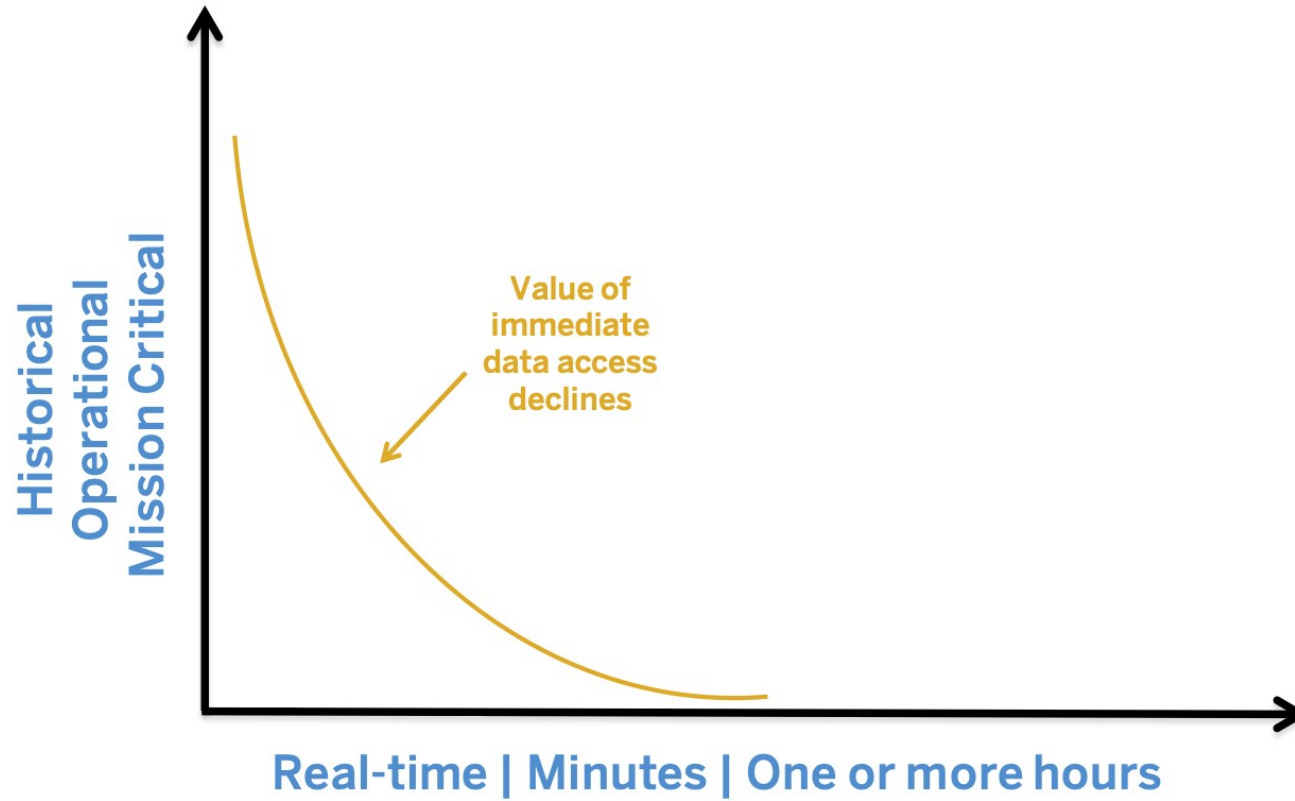
1. Audit
2. Time travel
3. Root cause analysis
4. Fault Tolerance
5. Event-driven architecture
6. Asynchronous first
7. Service Autonomy
8. Replay and Reshape
9. Observability
10. Occasionally connected
11. One way data flow
12. Legacy Migration

Microservice Architecture



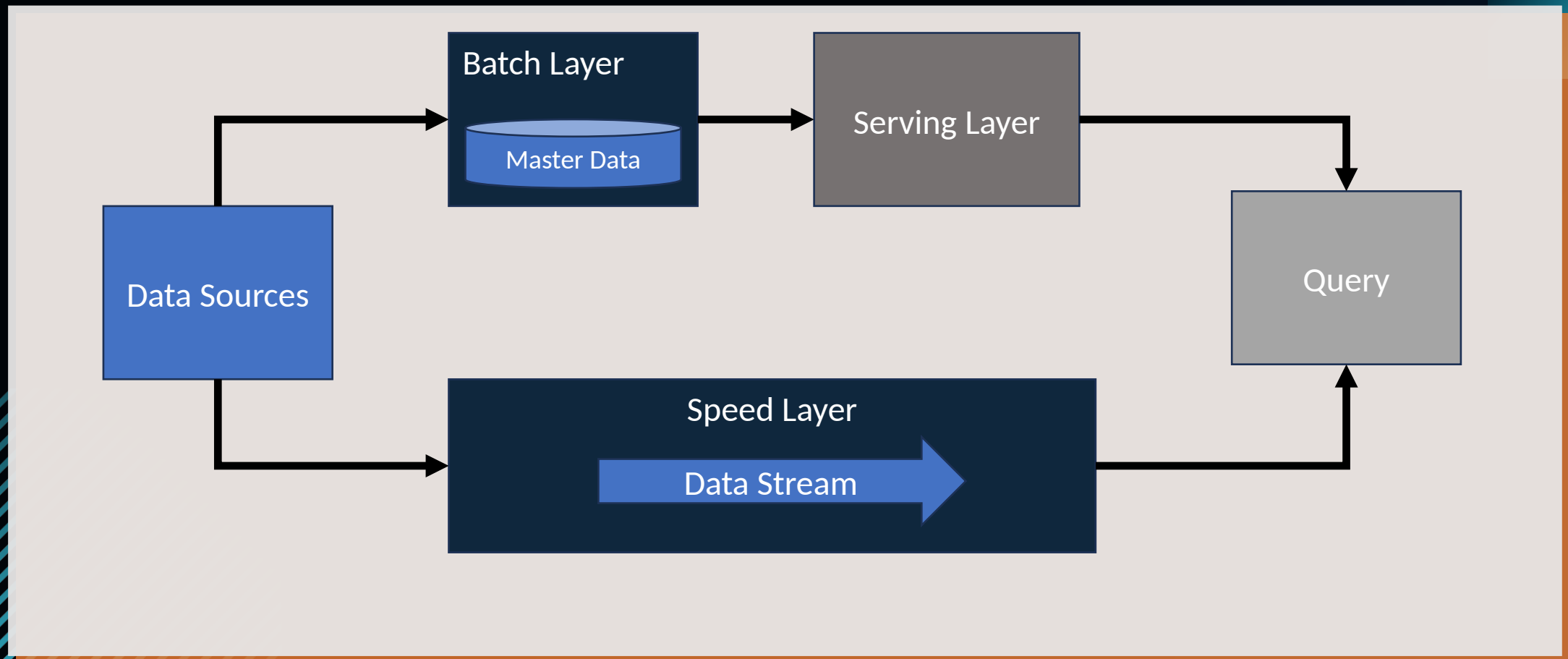
Stream Processing

The time value of data



How do we enable a **real time** platform ?

LAMBDA ARCHITECTURE

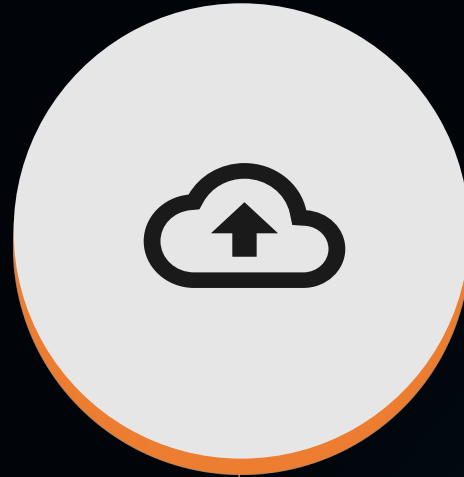


Key Factors



Latency

Fast Data Access if required. Aggregated View on masterdata.



Scalability

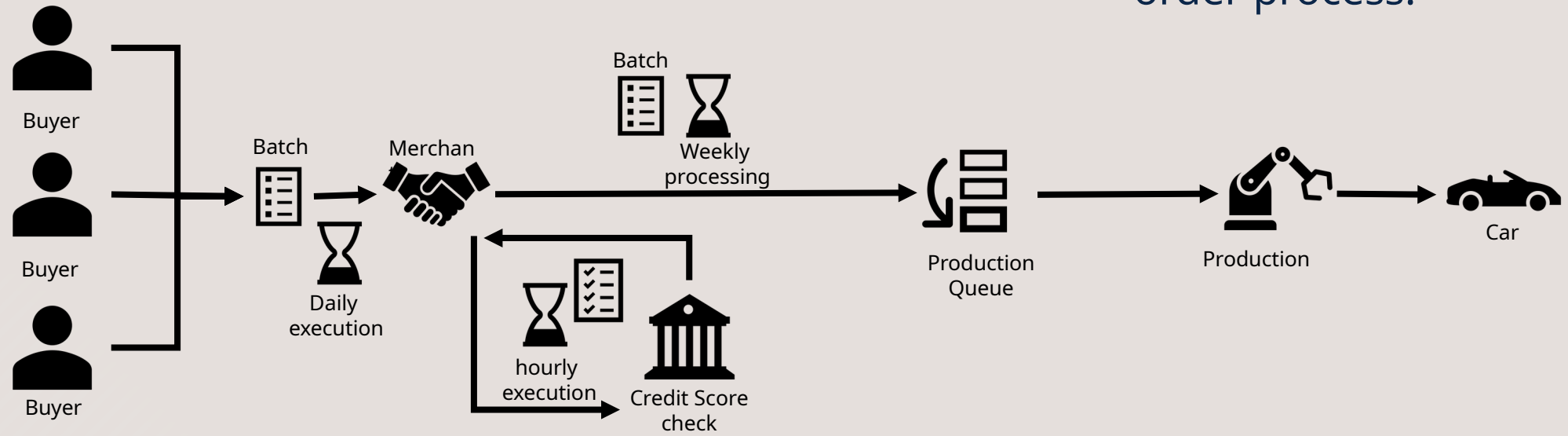
Lambda Pattern allows for clustered technologies that are fault tolerant and horizontally scalable.



Consistency

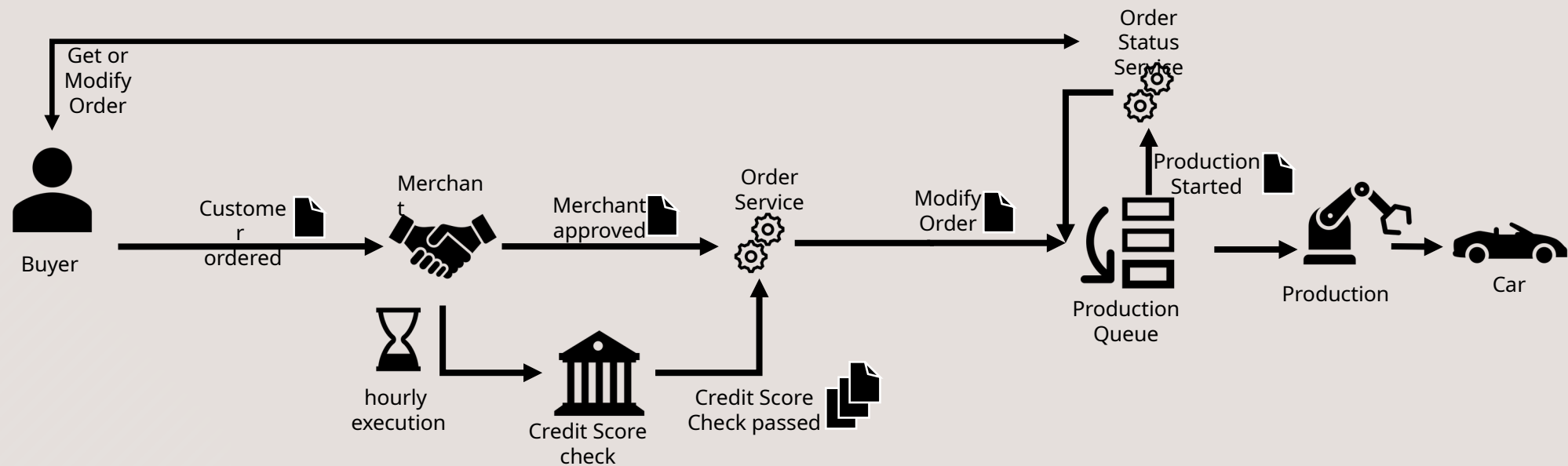
Masterdata will eventually be consistent since it is based on the stream of events.

A customer orders his car and is able to configure it during the order process.



**Disadvantages?
Saving Time?**

A customer orders his car and is able to configure it on demand as long as it is not produced.

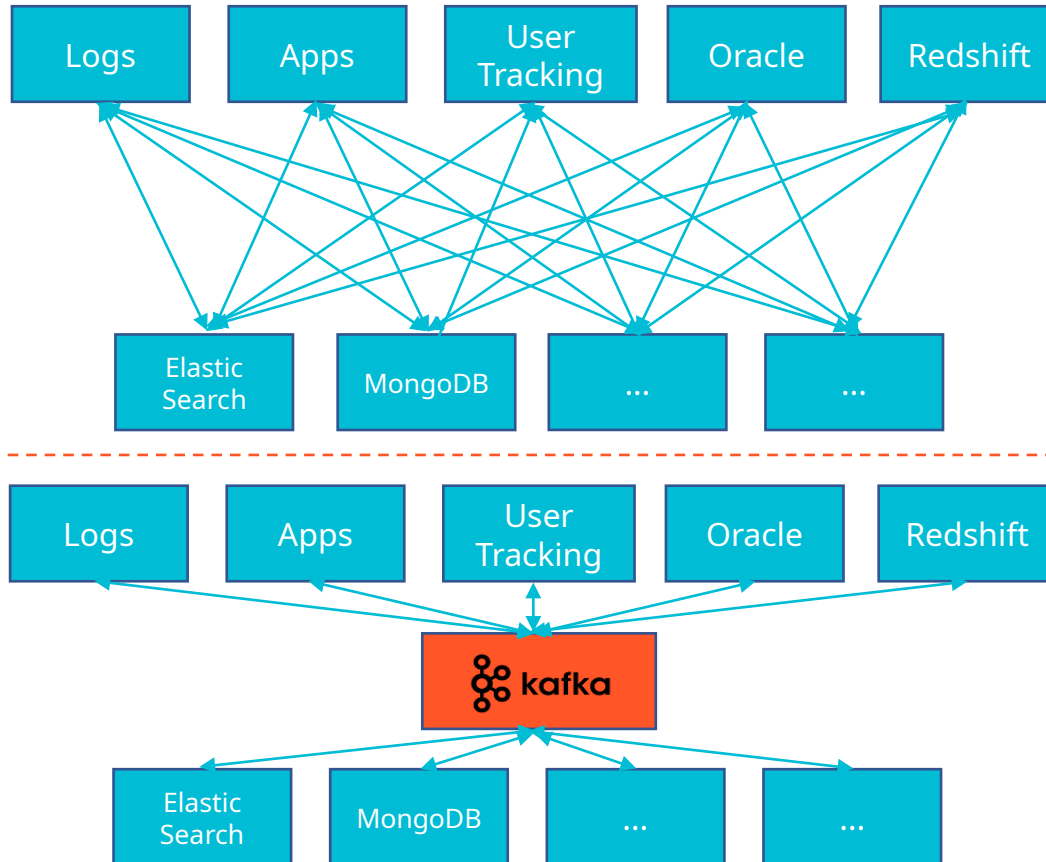




Kafka Basics



WHY KAFKA?



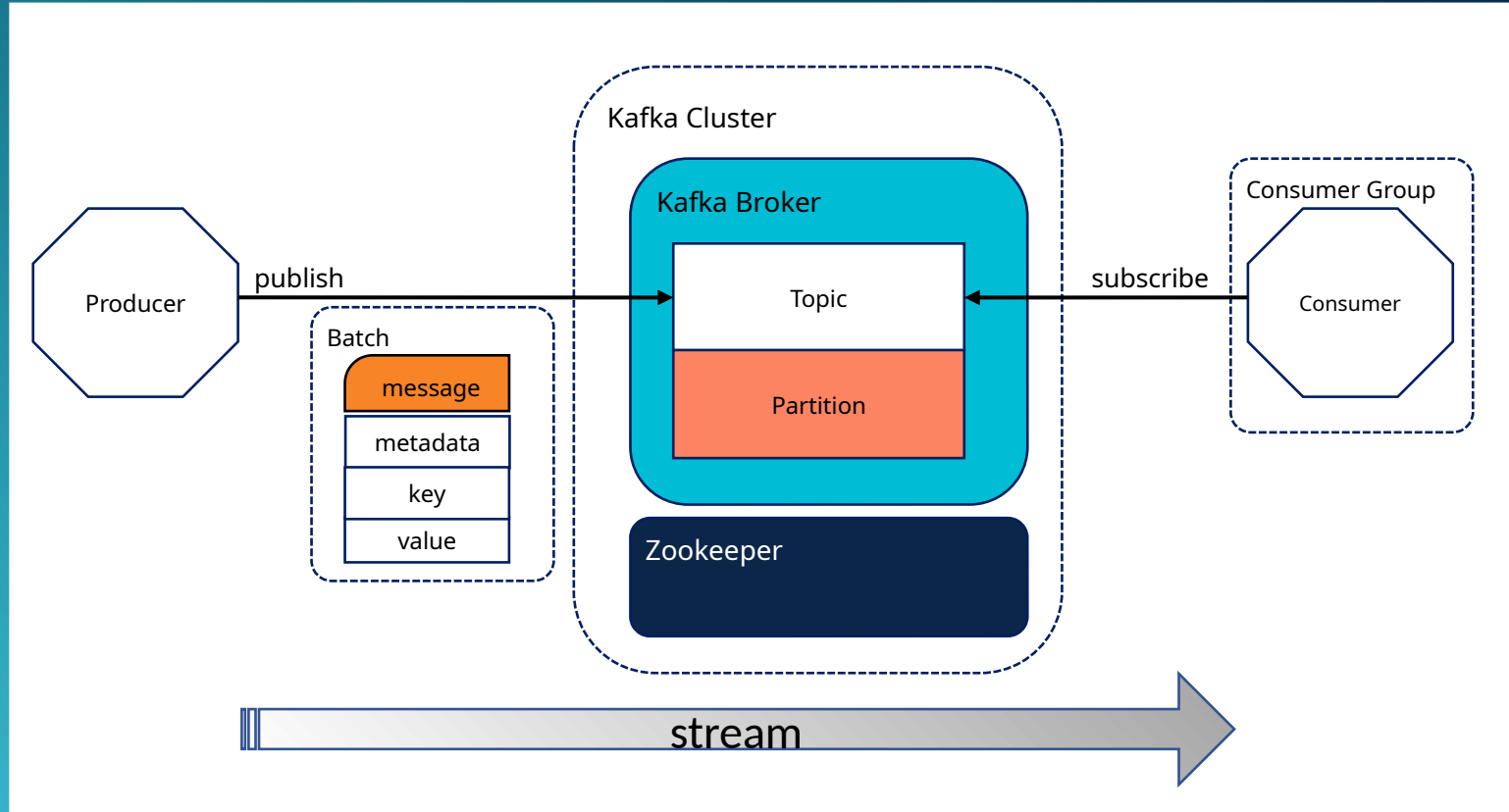
REASON

If you have 5 source systems, and 4 target systems, you need to maintain 20 interfaces!

Kafka is a distributed streaming platform. A commit log provides a durable record of all transactions so that they can be replayed to consistently build the state of a system. The data can be distributed reliably within the system.

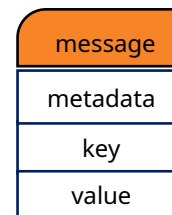
WHAT IS KAFKA?

Kafka is the solution to optimally store, scale and process data streams.

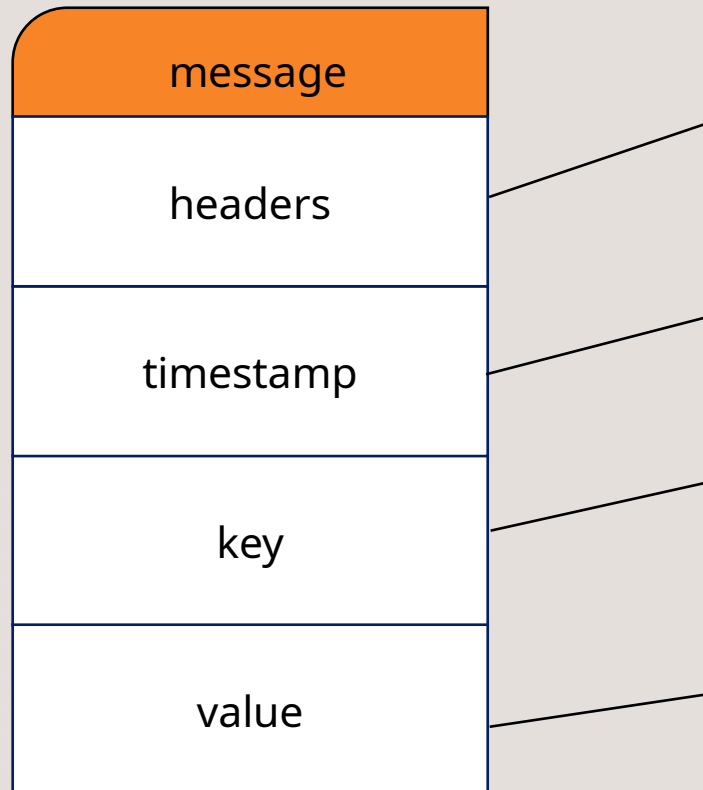


A Message in Kafka is a unit of data. Comparable to a row or a record in a database.

WHAT IS A MESSAGE?



A MESSAGE



The headers of a message contain **user defined key value pairs**. This comes in handy if you want to track metadata inside the system. Those **metadata** might be e.g. the source system the message originates from or the user that produced the message.

The **date time** the message was produced.

A key is an array of bytes that has on its own no meaning to Kafka but might be used by clients to evaluate **partitions** or by Kafka streams applications to identify messages by its keys.

The value is like the key an array of bytes with no meaning to Kafka. The value contains the **user data** that needs to be transferred through Kafka.

A Kafka Partition contains a log of messages.

WHAT IS A PARTITION?



PARTITIONS

Partition 0

Offset 0	Offset 1	Offset 2	Offset 3	Offset ...	Offset N
Message Key: 0	Message Key: 1	Message Key: 3	Message Key: 1	Message Key: ...	Message Key: X

A Kafka topic contains a collection of message logs. It is comparable to a database table.

WHAT IS A TOPIC?

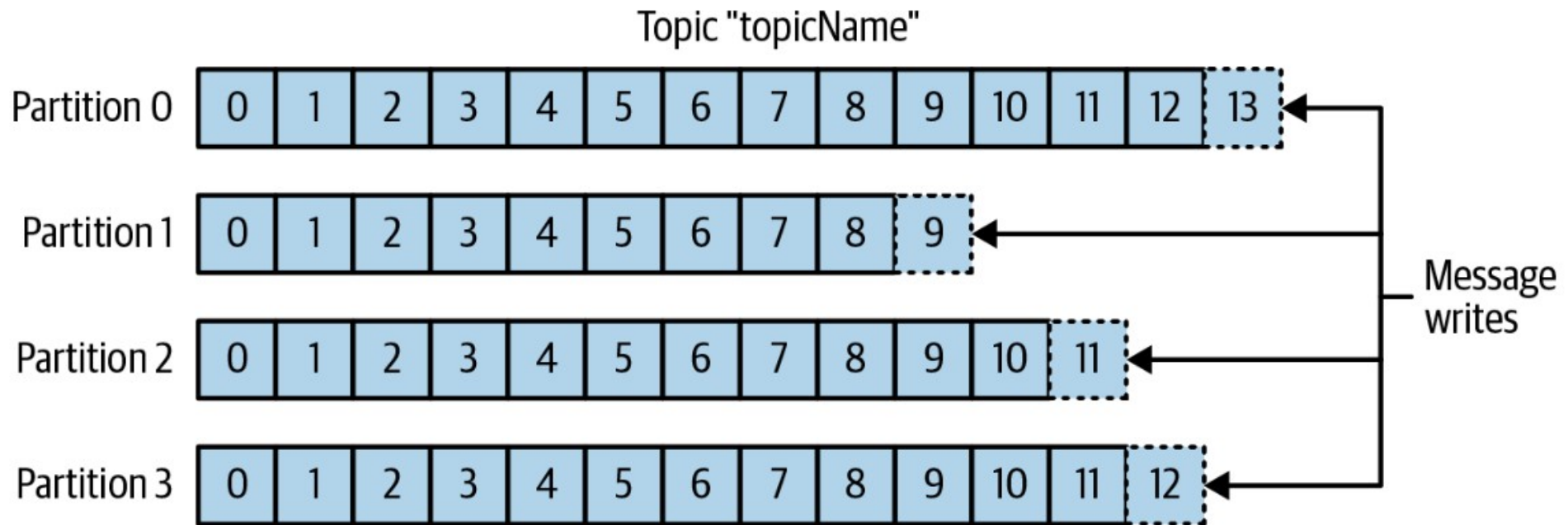


Topic



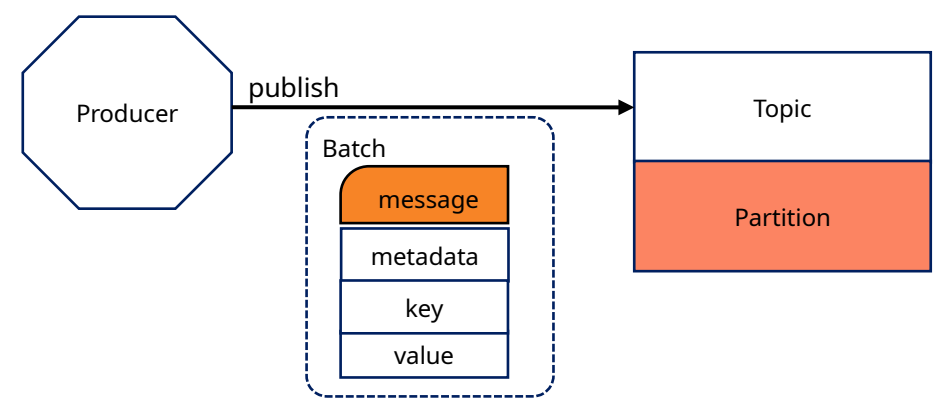
KAFKA TOPICS

- Topics is a particular stream of data.
- Is similar to a table in a database
- A topic is identified by its name
- You can have as many topics as you want
- Topics are split into partitions



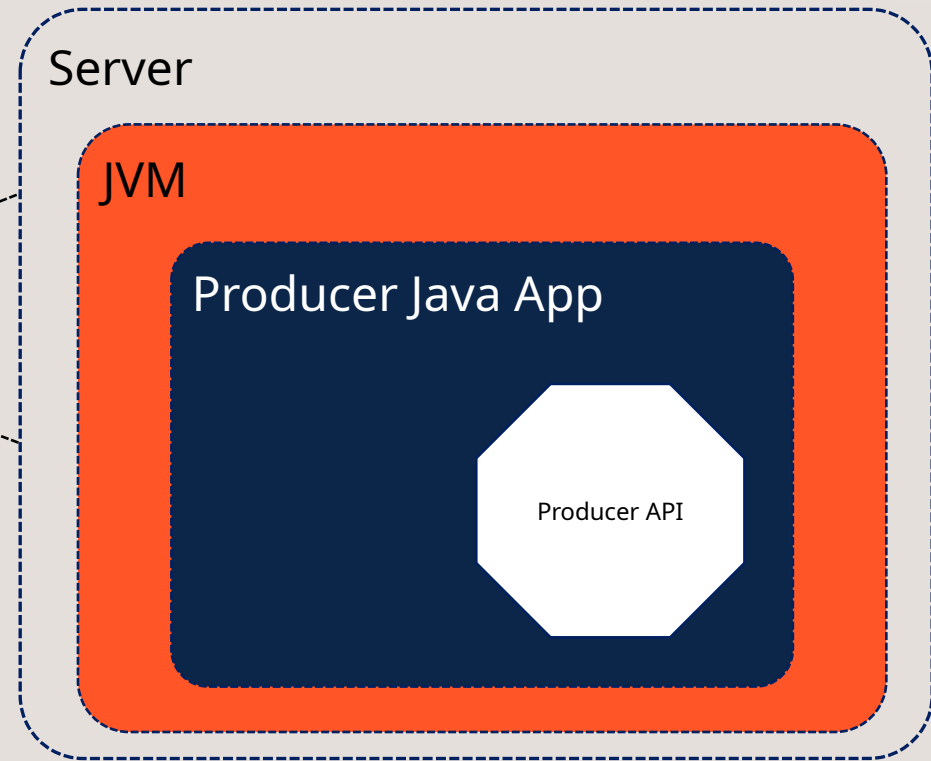
WHAT IS A KAFKA PRODUCER?

A Kafka Producer publishes messages to Kafka.

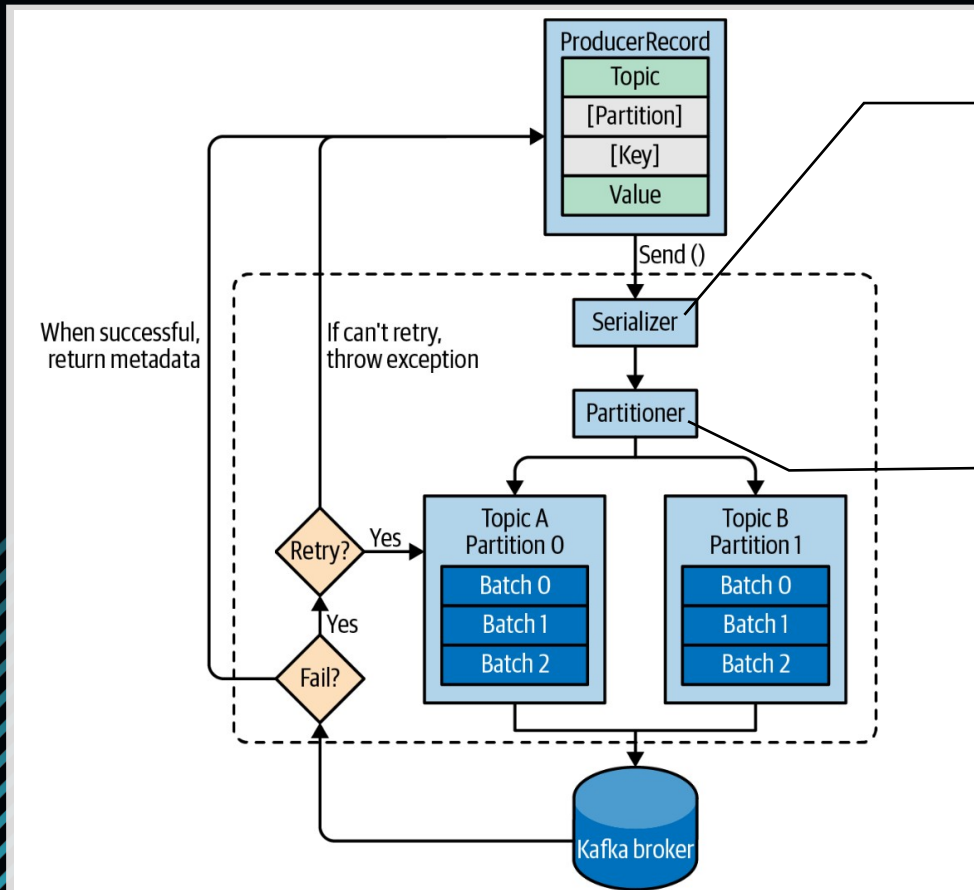


KAFKA PRODUCER – HIGH LEVEL

Producer API Project in different Languages



KAFKA PRODUCER – LOWER LEVEL

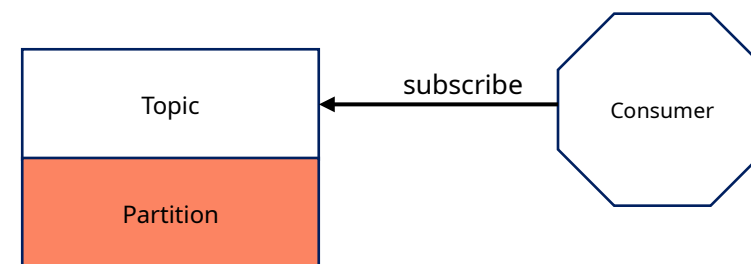


The Serializer transforms messages from objects inside the program to an array of bytes (e.g. the key and value of a message).

The Partitioner determines the partition the message needs to be sent to. Per default it is **the murmur hash of the message key modulo the number of partitions**. This results in the same partition for the same keys and equally distributed partitions with UUIDs.

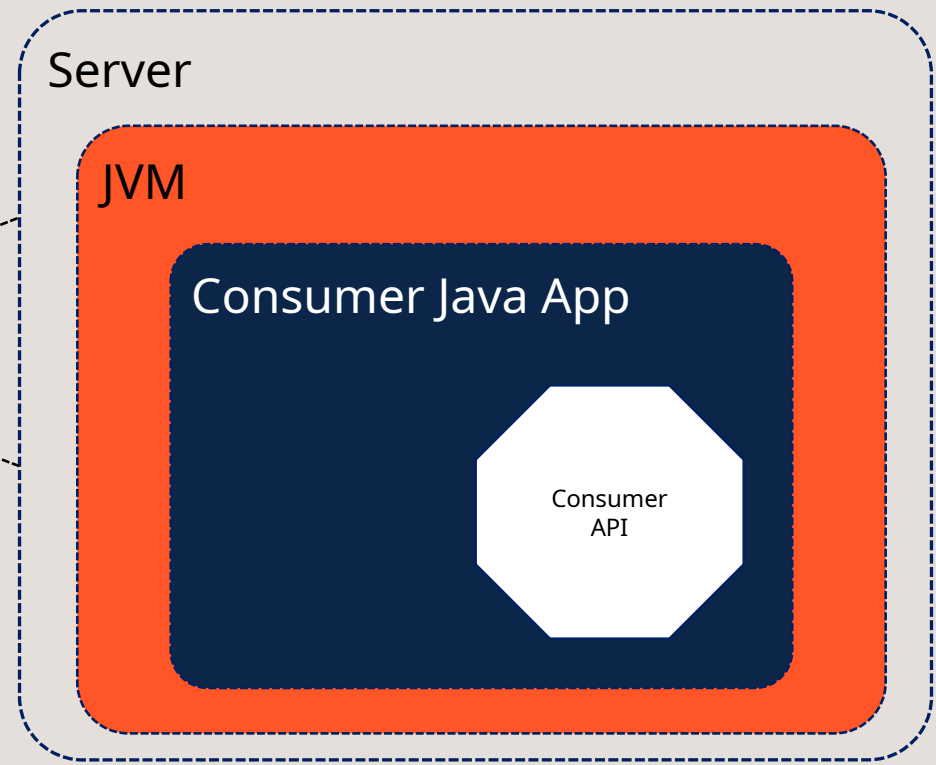
A Kafka Consumer subscribes to topics owns partitions and polls messages from Kafka.

WHAT IS A KAFKA CONSUMER?

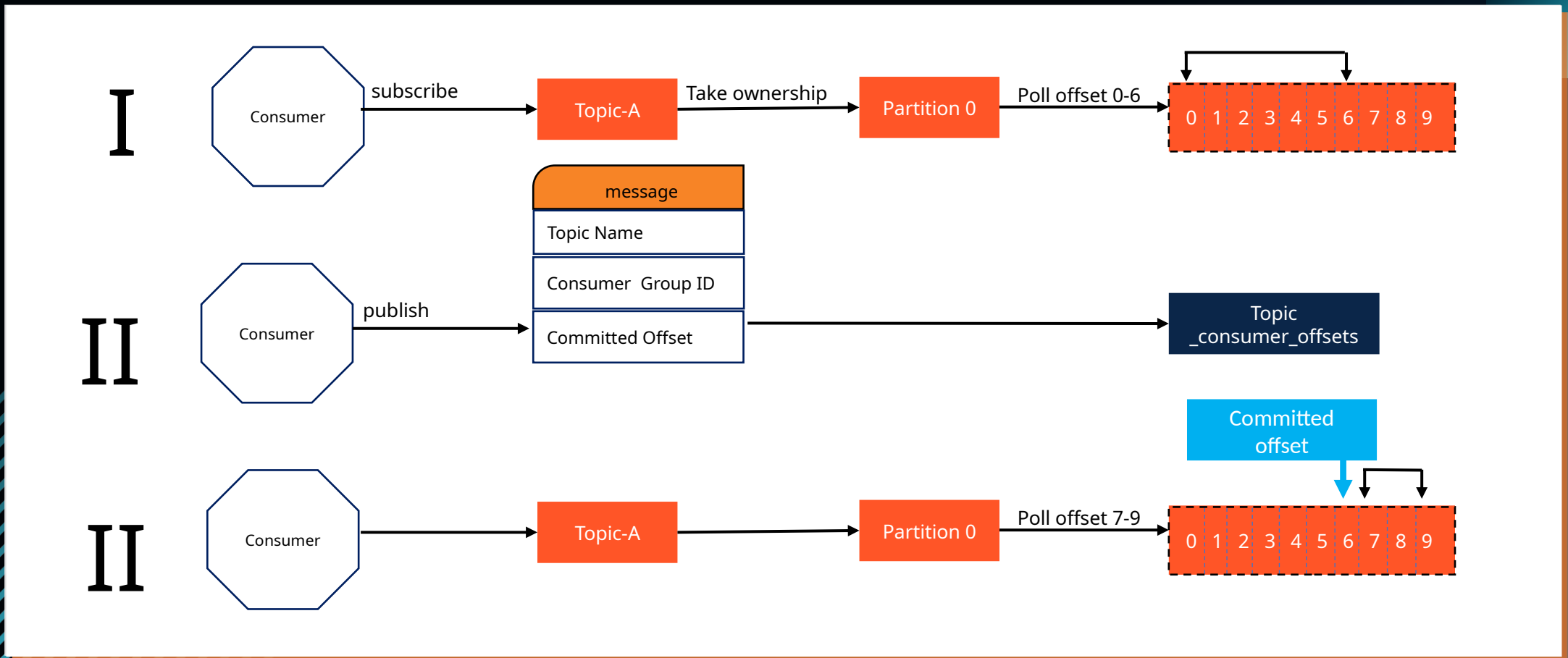


KAFKA CONSUMER – HIGH LEVEL

Producer API Project in different Languages

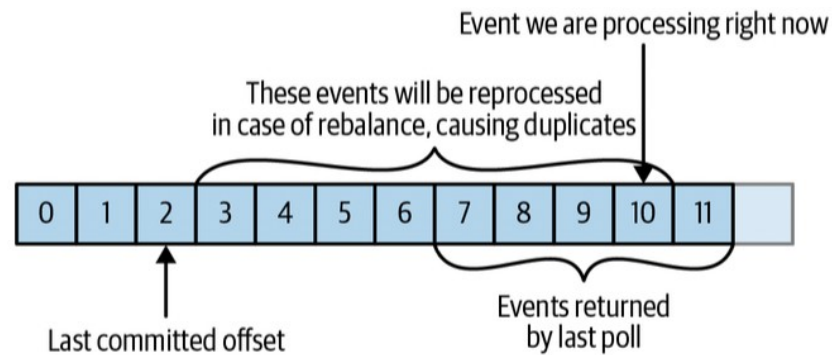


KAFKA CONSUMER – COMMIT OFFSET



KAFKA CONSUMER – COMMIT OFFSET

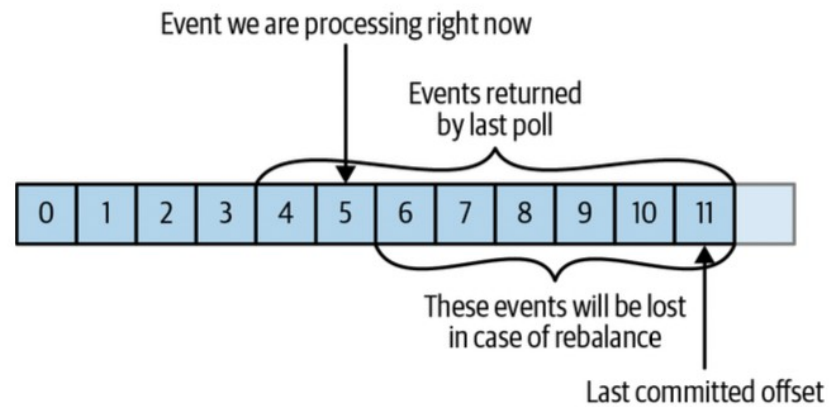
I



duplication

When offset was committed too **late**.

II

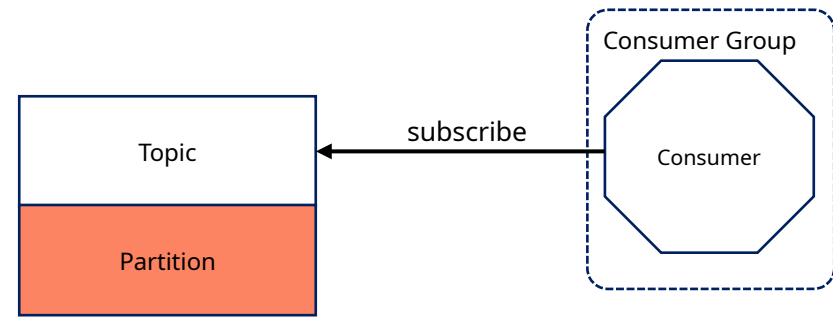


message loss

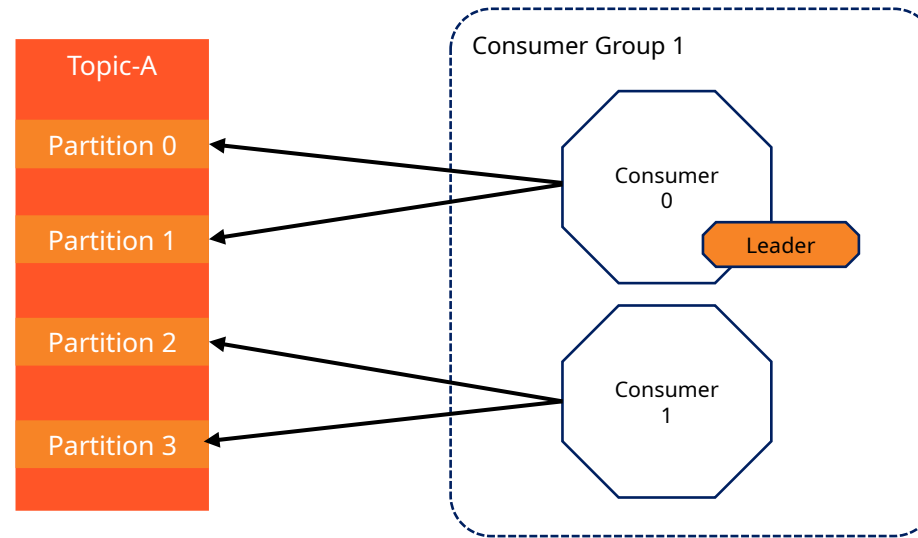
When offset was committed to **early**.

A Consumer Group separates data flows and provides scalability and resilience for consumers.

WHAT IS A KAFKA CONSUMER GROUP?

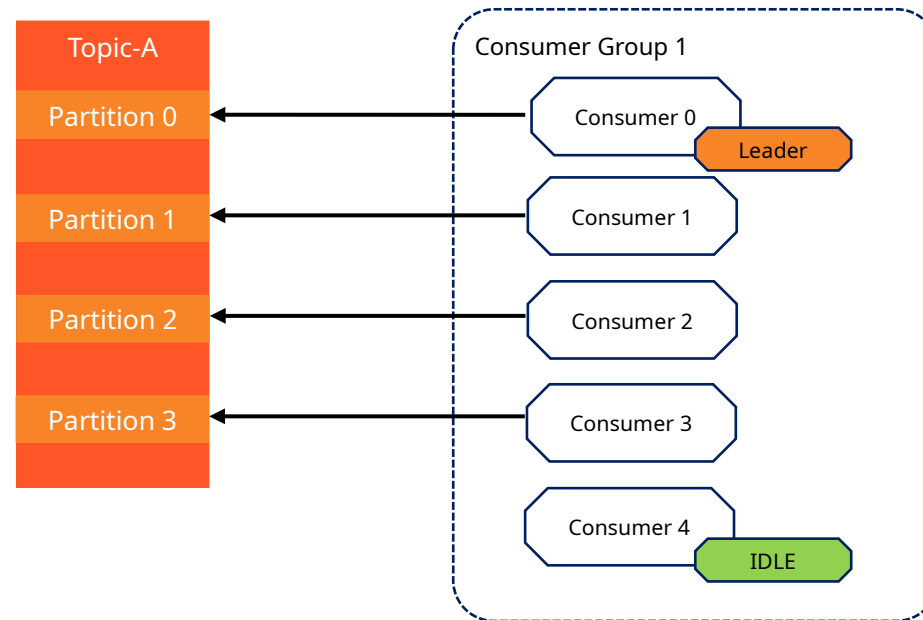


KAFKA CONSUMER GROUP BASICS



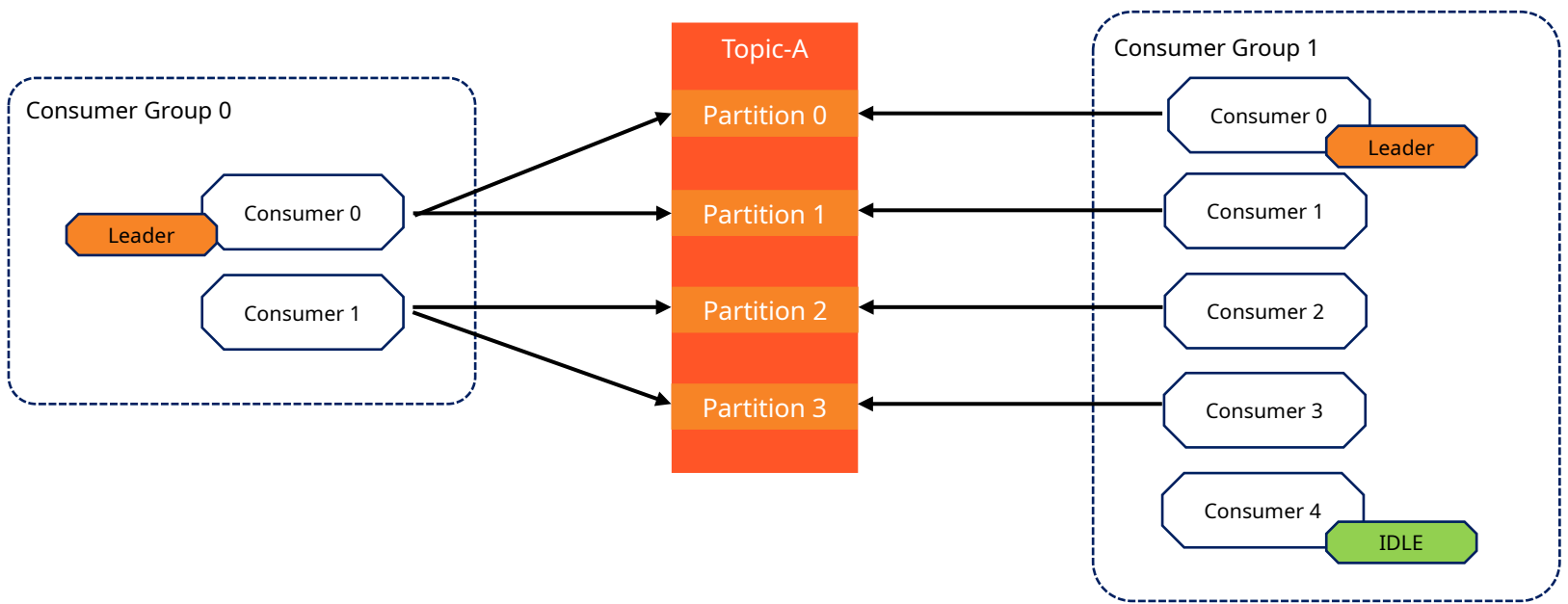
- Consumers take ownership of one or more partitions
- Only one consumer is thought to be owning a partition
- Consumers in a Consumer Group share consumer offsets

KAFKA CONSUMER GROUP IDLE CONSUMERS



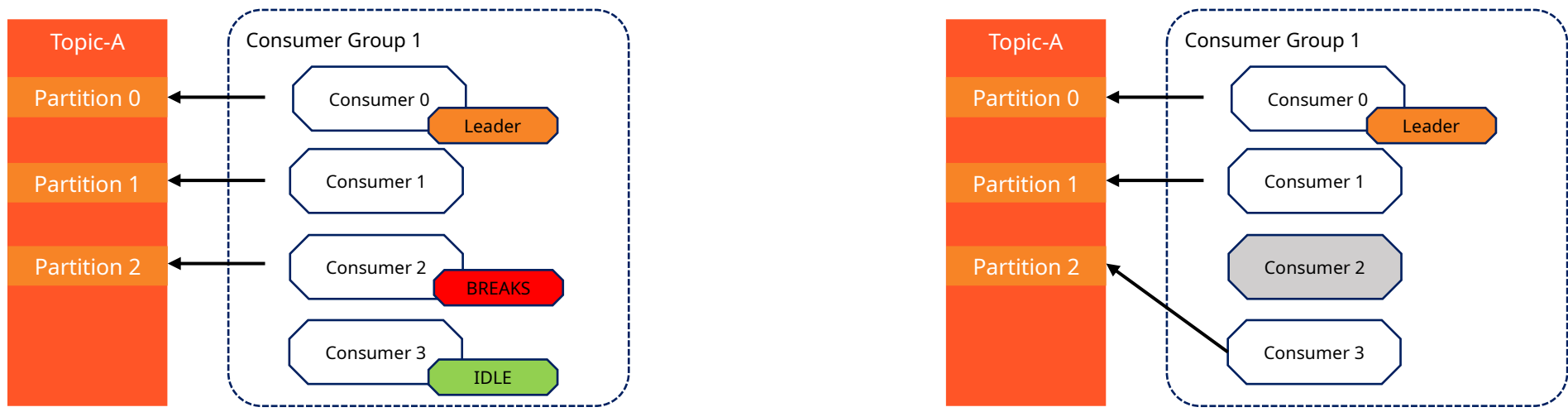
- Partitions are exclusive to a consumer instance
- Additional consumers are idling when assigned to the group
- The Consumer Group Leader takes automatically care of balancing partitions between consumers

KAFKA CONSUMER GROUP SEPARATE USE CASES



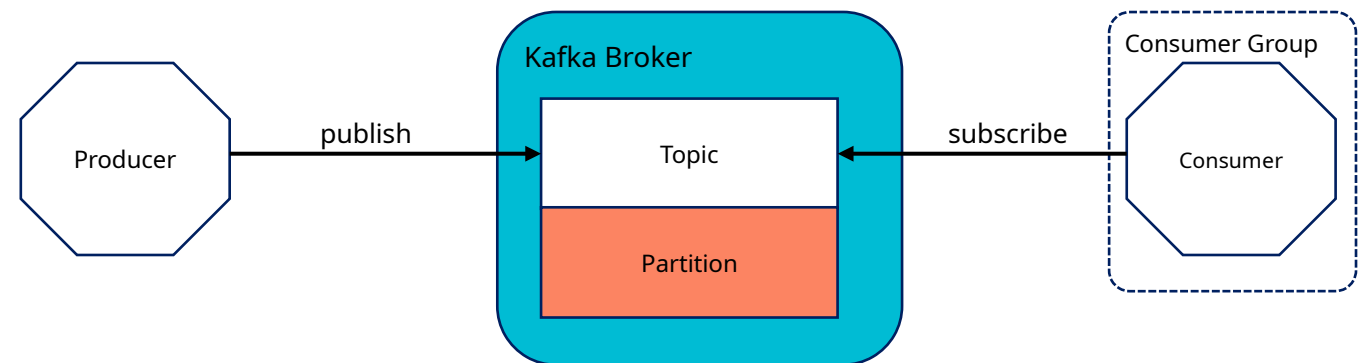
- Additional Groups consume the full set of data
- Additional Groups have their own consumer offset
- Separates technical or business use cases with this technique from each other

KAFKA CONSUMER GROUP PARTITION REBALANCE



- Rebalancing is needed if the number of partition changes
- Rebalancing is needed if the number of non idle consumer in a consumer group changes
- The leader performs the rebalancing

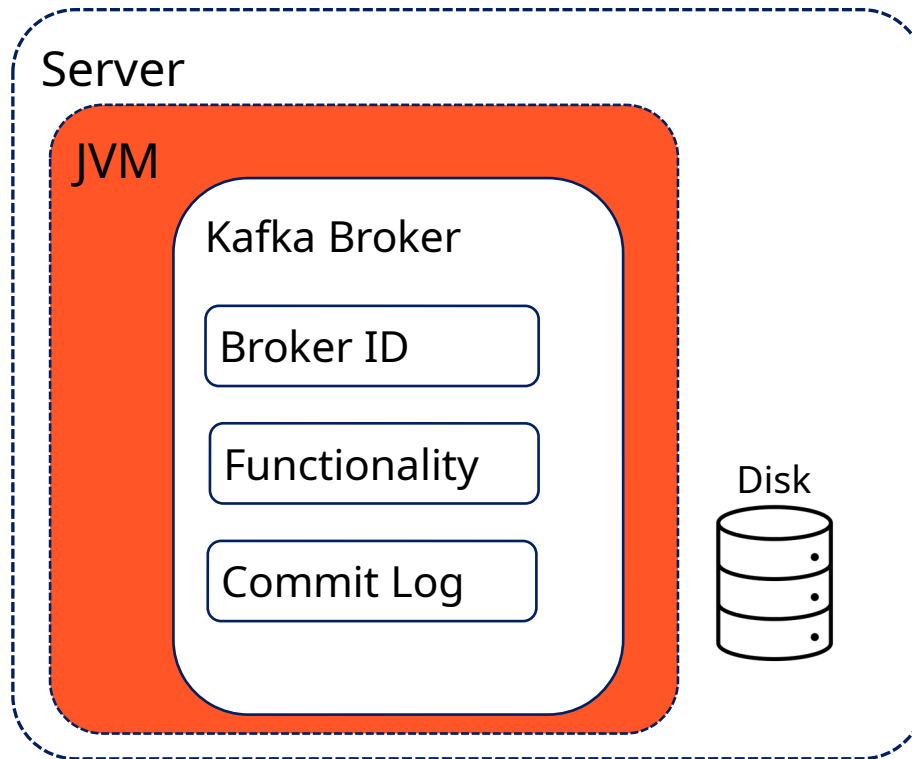
A single Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them and stores them on disk. It also receives fetch requests from consumers and responds with the messages.



WHAT IS A KAFKA BROKER?

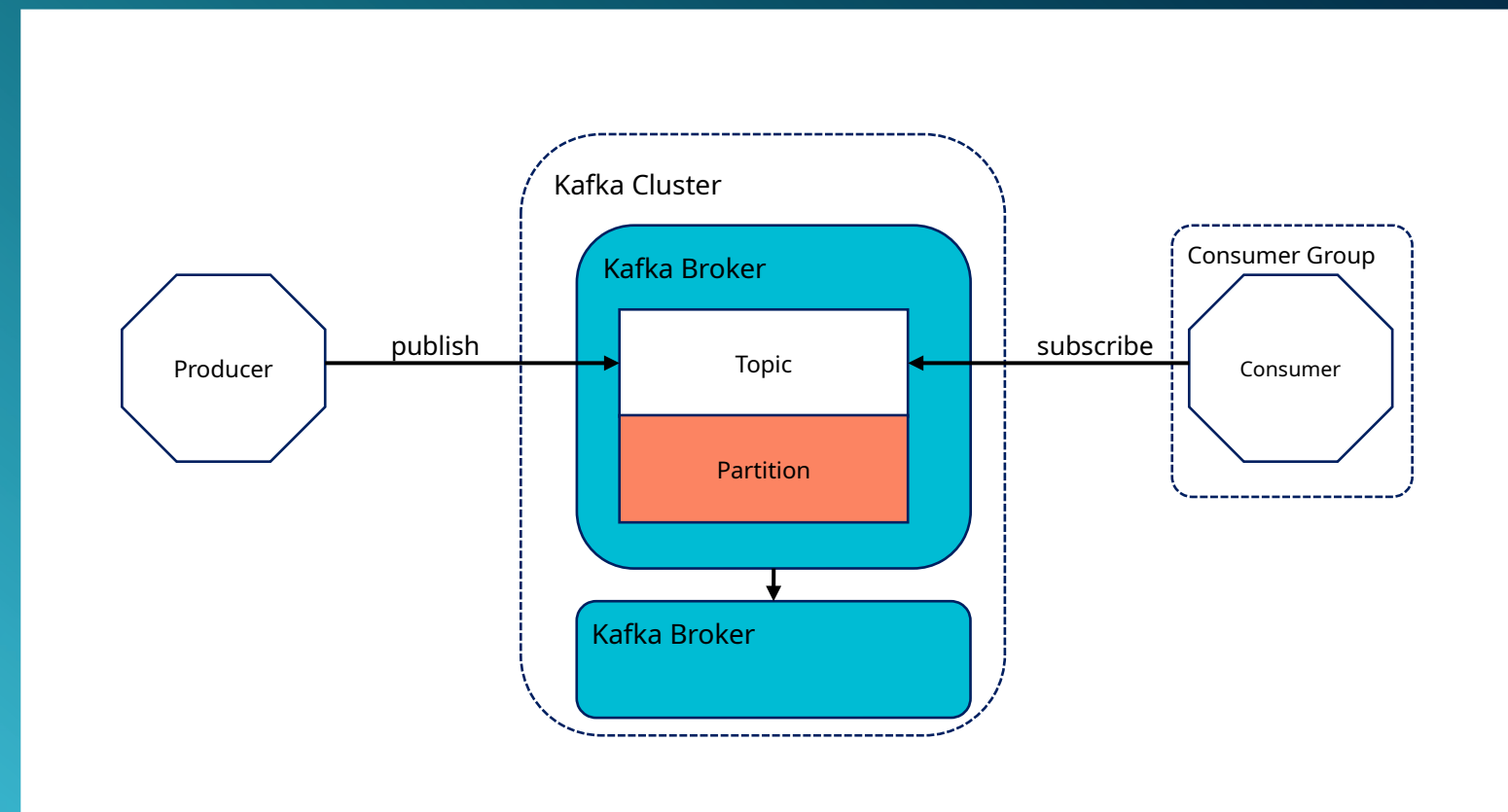
KAFKA BROKER

THE ARCHITECTURE

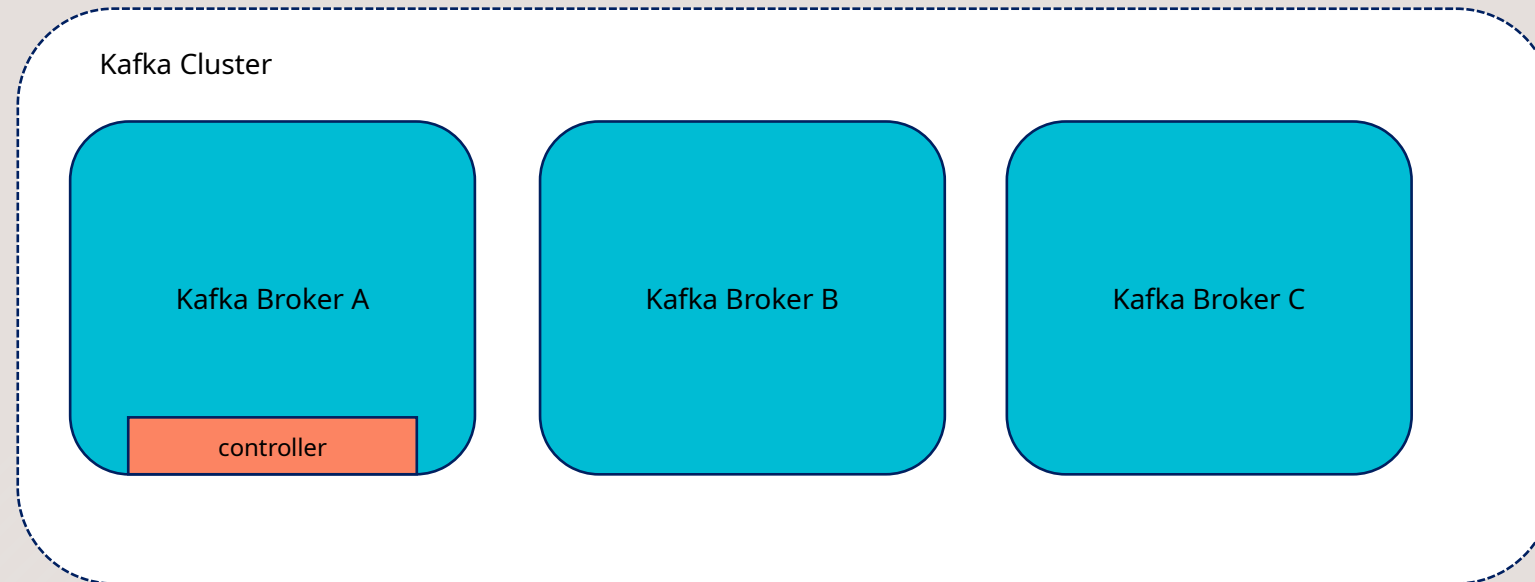


WHAT IS A KAFKA CLUSTER?

Kafka brokers are designed to work as a cluster. A cluster provides scalability and resilience.

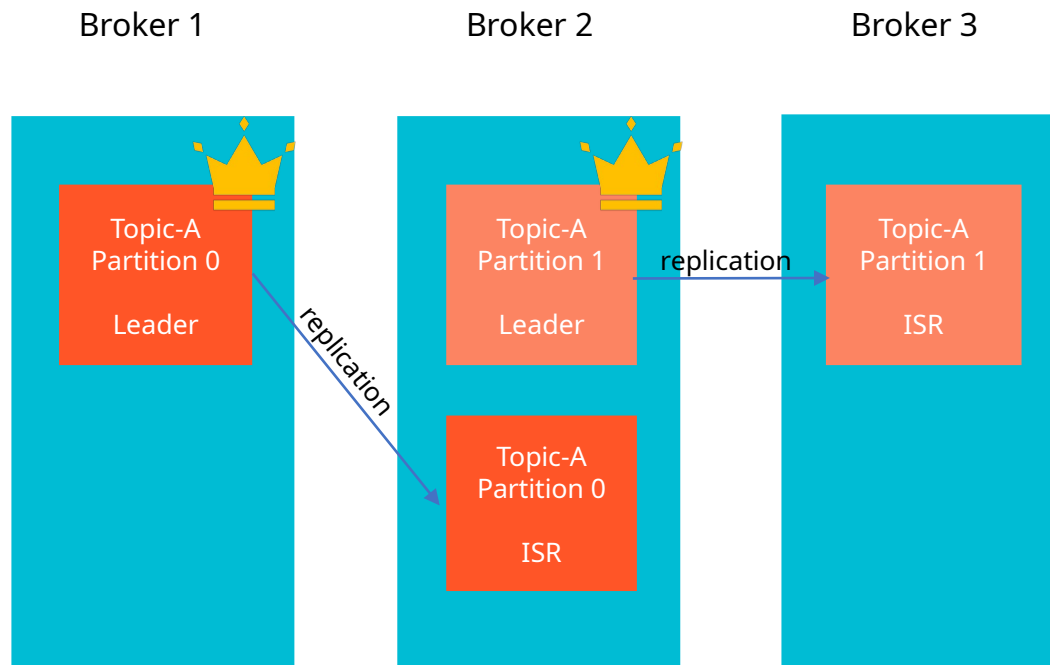


KAFKA CLUSTER – MULTIPLE BROKERS



- A Kafka cluster consists of one or more brokers
- Cluster consensus is provided using the RAFT protocol (KRaft)
- One broker is elected as a controller and monitors broker failures and performs leader elections

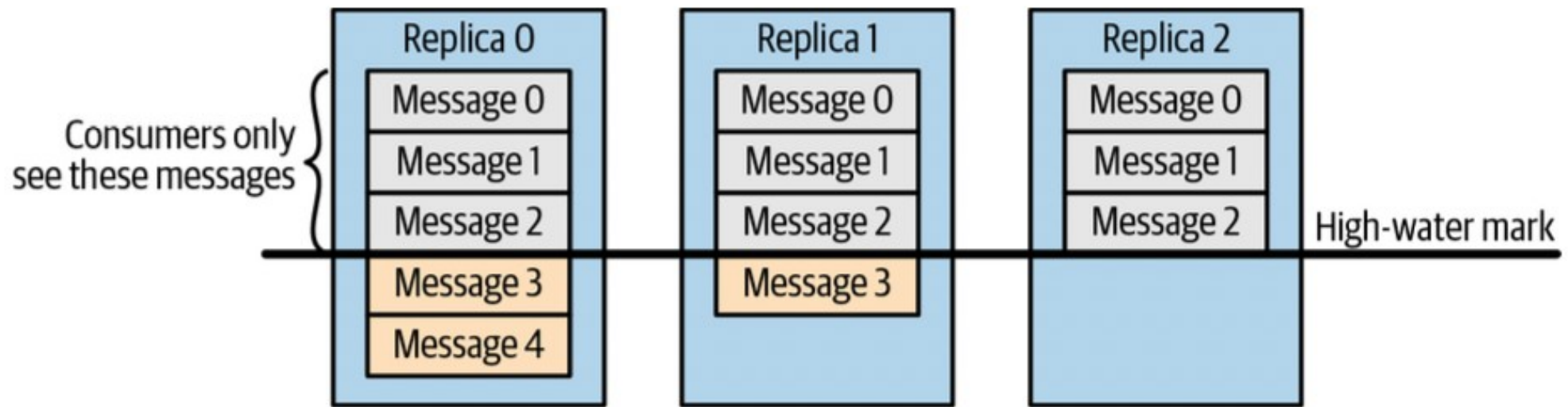
KAFKA REPLICATION FACTOR



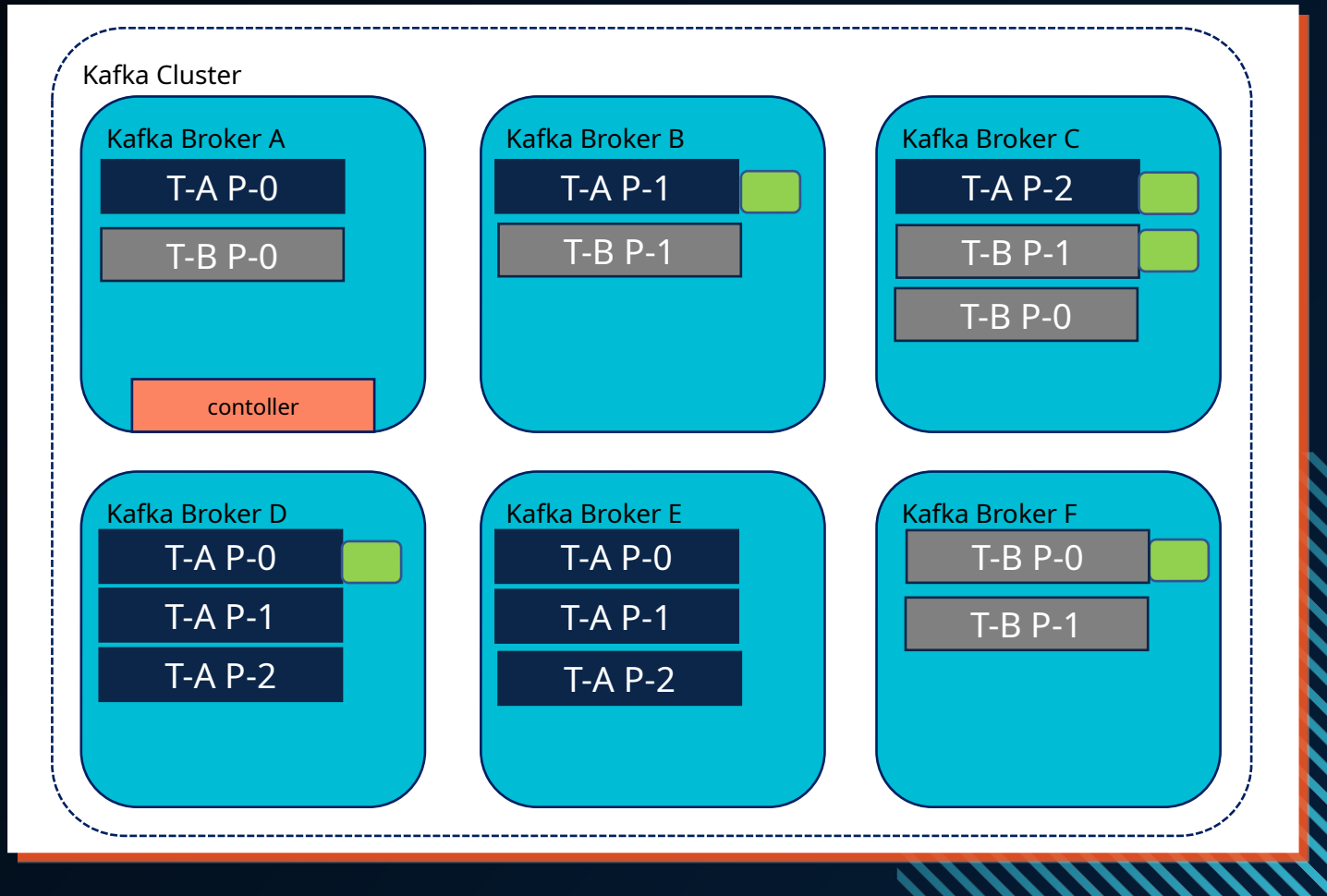
ABOUT

- Necessary, if a broker is down, another broker can serve the data
- Topics should have a replication factor > 1 (usually between 2 and 3)
- Only one broker can be a leader for a given partition
- The leader can receive and serve data for a partition (--> the other brokers will synchronize the data)
- Example:
 - Topic-A with 2 partitions and replication factor 2


KAFKA CLUSTER REPLICATION HIGH WATERMARK



TOPIC REPLICATION - EXAMPLE

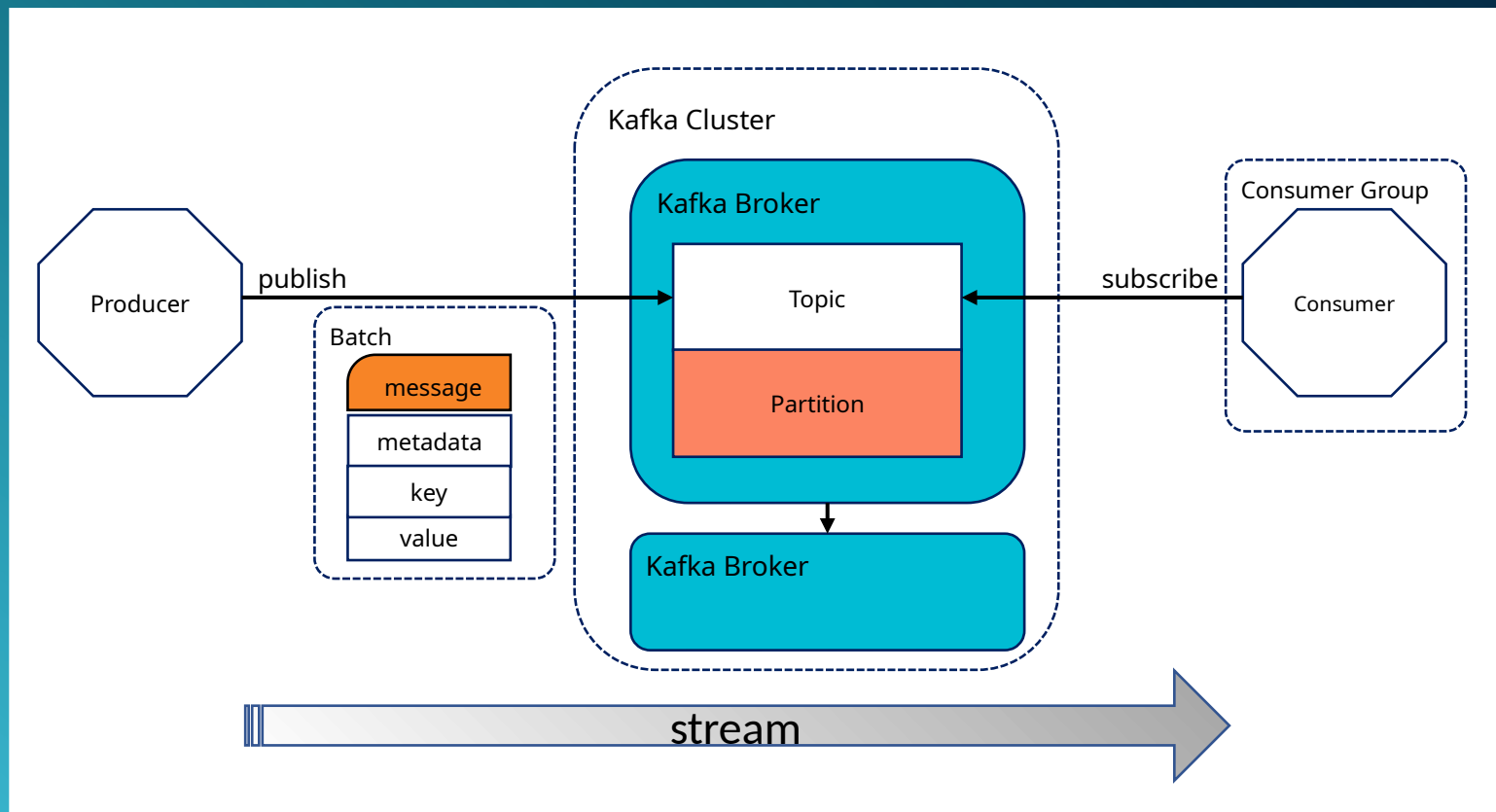


SETUP

- 6 Kafka Brokers
- Replication factor per Topic set to 3
- Topic A with 3 Partitions
- Topic B with 2 Partitions
- Leader 

THIS IS KAFKA

Kafka is the solution to optimally store, scale and process data streams.

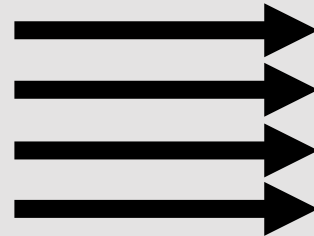




KAFKA SCHEMA REGISTRY

WHAT IS A SCHEMA?

Data set (record)
Alice; 42; Frankfurt; Burger with vegan Patty; ;



Schema		
Name of the value	Data typ	Mandatory field
Name of service-staff	Text	No
Odering number	Number	Yes
Location	Text	Yes
Ordered product	Text	Yes
Additionally Ingredients	Text List	No

Note:
Data without a description of the content cannot be interpreted technically.

An abstract description of data enables a defined exchange of technical information.

COMMON SERIALIZATION PROTOCOLS

	Avro	Protobuf	JSON	Parquet
Format	Binary	Binary	Non-Binary Human-readable	Binary
Use case	Transaction systems with high performance	Transaction systems with high performance	Transaction systems	Bulk data storage
Supported in schema registry	Yes	Yes	Yes	No

ABOUT

- Extensive and complex structured data sets can be mapped
- These data sets can be converted into an efficient binary format to optimize the exchange between components of a system

SCHEMA – EXAMPLE AVRO



SCHEMA

```
{ type : record,  
  namespace : digital.thinkport.kafkaworkshop,  
  name : order,  
  fields : [  
    { name : bedienung, type : [null,string], default:null},  
    { name : order-number, type : int },  
    { name : location , type : int },  
    { name : product , type : int },  
    { name : additional_ingredient,  
      type :[null,  
        {type: array,  
          items:{ name : ingredient, type : string}  
        }  
      ]},  
    { name : additional_ingredient,  
      type : [null, string],  
      default : null  
    }  
  ]  
}
```

ABOUT

- **RECORD** describes a collection of attributes
- **NAMESPACE** in conjunction with **NAME** can be used to map a domain model to ensure uniqueness in schema administrations
- **FIELDS** are the attributes
- optional fields are defined by [null, string], default: null
- Complex data types can be nested

OPTIMIZED RESOURCES THROUGH AVRO

Schemaless JSON
<pre>{ "service-staff":"Alice", "order-number":"42" , "location":"Frankfurt" , "product":"Burger with vegan Patty"}</pre>

Size:100 Bytes

Schemaless JSON-Files contain the information about the subject matter in the data set itself.

Avro-File with Schema
<pre>{ type : record, namespace : digital.thinkport.kafkaworkshop, name : Order, fields : [{ name : service-staff, type : [null,string], default:null}, { "name" : "order-number" , "type" : "int" }, { "name" : "location" , "type" : "int" }, { "name" : "product" , "type" : "int" }, { "name" : "additionally_ingredients", "type":[null, {"type": "array", "items":{"name":"ingredient", "type" : "string" } }]}, "default" : "null"] }</pre>
Data set in binary format

Size:300 Bytes

Avro files are saved together with the schema. The data set is optimized in binary format. Programs can use this information to deserialize the data.

Avro-File with Schema ID
Schema ID
Data set in binary format

Size: 30 Bytes

If the schema is stored externally, a reference to which schema was used is sufficient. The data itself can be stored and or sent efficiently.

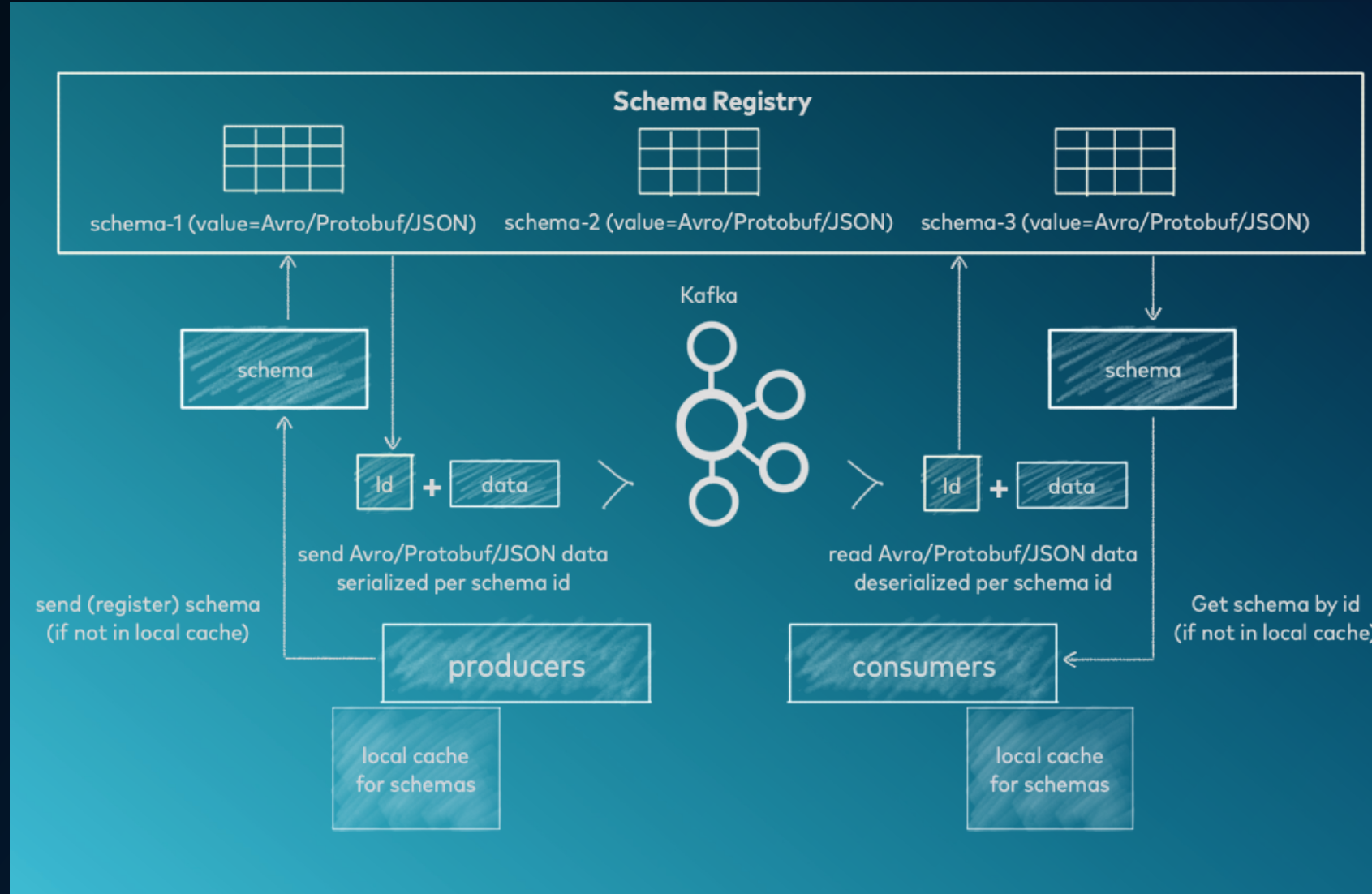
SCHEMA REGISTRY Overview

Benefits

1. Dealing with organizational challenges of data management
2. Stable data pipelines
3. Safe scheme evolution
4. Memory and calculation efficiency
5. Data discovery
6. Cost-efficient ecosystem
7. Data policy enforcement

SCHEMA REGISTRY

- Confluent provides a RESTful interface for schema registries
- Avro, JSON Schema and Protobuf are available
- Schema Registry lives outside of and separately from your Kafka brokers
- No need to select a schema registry format





Kafka Streams

Kafka Ecosystem

Kafka Streams

- Is a Java library
- Optimized for processing unbounded datasets quickly and efficiently,
- A great solution for problems in low-latency, time-critical domains
- A great choice for building micro services on top of real-time event streams

Before Kafka Streams existed:

- lack of library support for processing data in Kafka topics

Two main options for building Kafka-based stream processing applications:

- Use the Consumer and Producer APIs directly
- Use another stream processing framework (e.g., Apache Spark Streaming, Apache Flink)

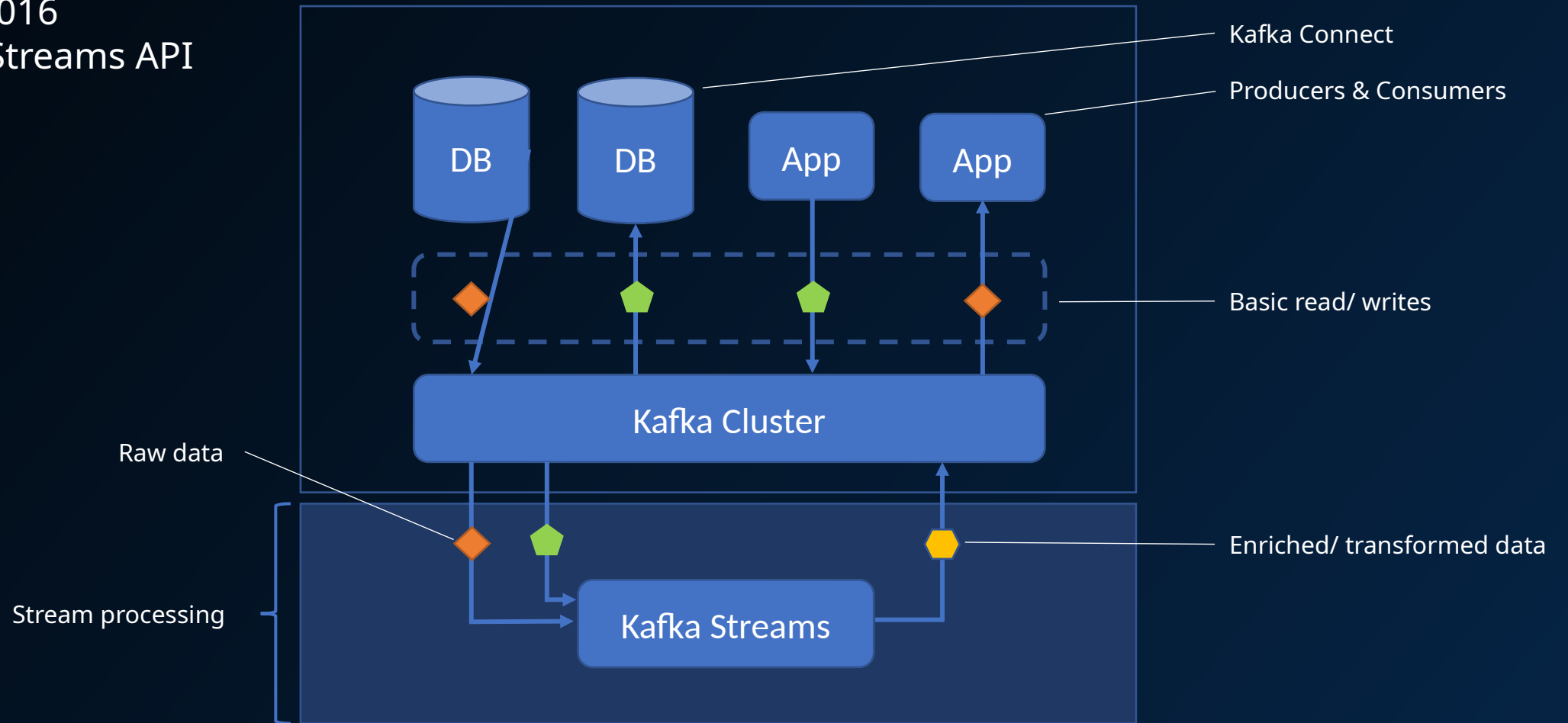
These APIs are very basic and lack many of the primitives that would qualify them as a stream processing API, including:

- Local and fault-tolerant state
- A rich set of operators for transforming streams of data
- More advanced representations of streams
- Sophisticated handling of time

The second option, which involves adopting a full-blown streaming platform like Apache Spark or Apache Flink, introduces a lot of unneeded complexity.

KAFKA STREAMS

After 2016
Kafka Streams API



Kafka Streams features:

- A high-level DSL that looks and feels like Java's streaming API
- A low-level Processor API
- Convenient abstractions for modeling data
- The ability to join streams and tables
- Operators and utilities for building stateless & stateful stream processing applications
- Support for time-based operations, (incl. windowing, periodic functions)
- Easy installation (just a library)
- Scalability, reliability, maintainability

Scalability

- By increasing the number of partitions on the source topics
- By leveraging consumer groups

Kafka Streams is also elastic, allowing you to seamlessly (albeit manually) scale the number of application instances in or out.

Reliability

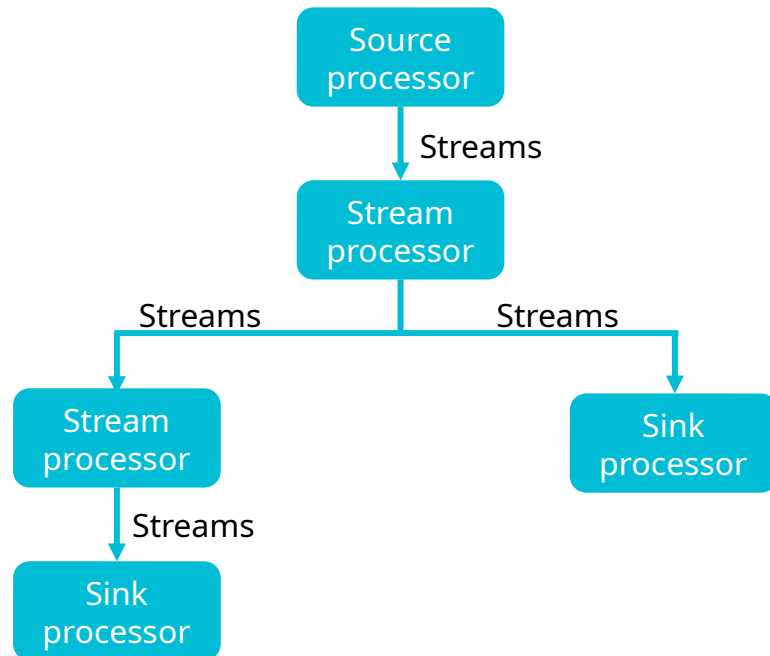
- Fault-tolerant features --> automatic failovers and partition rebalancing via consumer groups

Maintainability

- Troubleshooting and fixing bugs is relatively straightforward (Java library)
- KStreams API is succinct & intuitive, code-level maintenance is less time-consuming

- Financial data processing (Flipkart), purchase monitoring, fraud detection
- Algorithmic trading
- Stock market/crypto exchange monitoring
- Real-time inventory tracking and replenishment (Walmart)
- Event booking, seat selection (Ticketmaster)
- Email delivery tracking and monitoring (Mailchimp)
- Video game telemetry processing (Activision, the publisher of Call of Duty)
- Search indexing (Yelp)
- Geospatial tracking/calculations (e.g., distance comparison, arrival projections)
- Smart Home/IoT sensor processing (sometimes called AIOT, or the Artificial Intelligence of Things)
- Change data capture (Redhat)
- Sports broadcasting/real-time widgets (Gracenote)
- Real-time ad platforms (Pinterest)
- Predictive healthcare, vitals monitoring (Children's Healthcare of Atlanta)
- Chat infrastructure (Slack), chat bots, virtual assistants
- Machine learning pipelines (Twitter) and platforms (Kafka Graphs)

PROCESSOR TOPOLOGIES



ABOUT

- Kafka Streams leverages a programming paradigm called dataflow programming (DFP), which is a data-centric method of representing programs as a series of inputs, outputs, and processing stages
- very natural and intuitive way
- stream processing logic in a Kafka Streams application is structured as a directed acyclic graph (DAG)

3 basic kinds of processors in Kafka Streams

Source processors

Sources are where information flows into the Kafka Streams application. Data is read from a Kafka topic and sent to one or more stream processors.

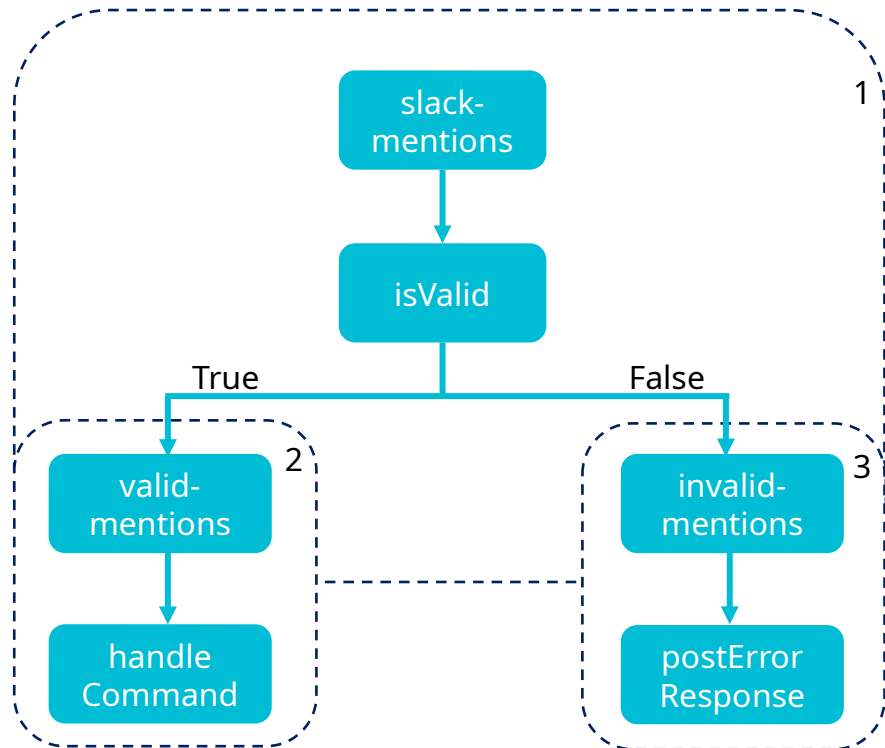
Stream processors

These processors are responsible for applying data processing/transformation logic on the input stream. In the high-level DSL, these processors are defined using a set of built-in operators that are exposed by the Kafka Streams library. Some example operators are filter, map, flatMap, and join.

Sink processors

Sinks are where enriched, transformed, filtered, or otherwise processed records are written back to Kafka, either to be handled by another stream processing application or to be sent to a downstream data store via something like KafkaConnect. Like source processors, sink processors are connected to a Kafka topic.

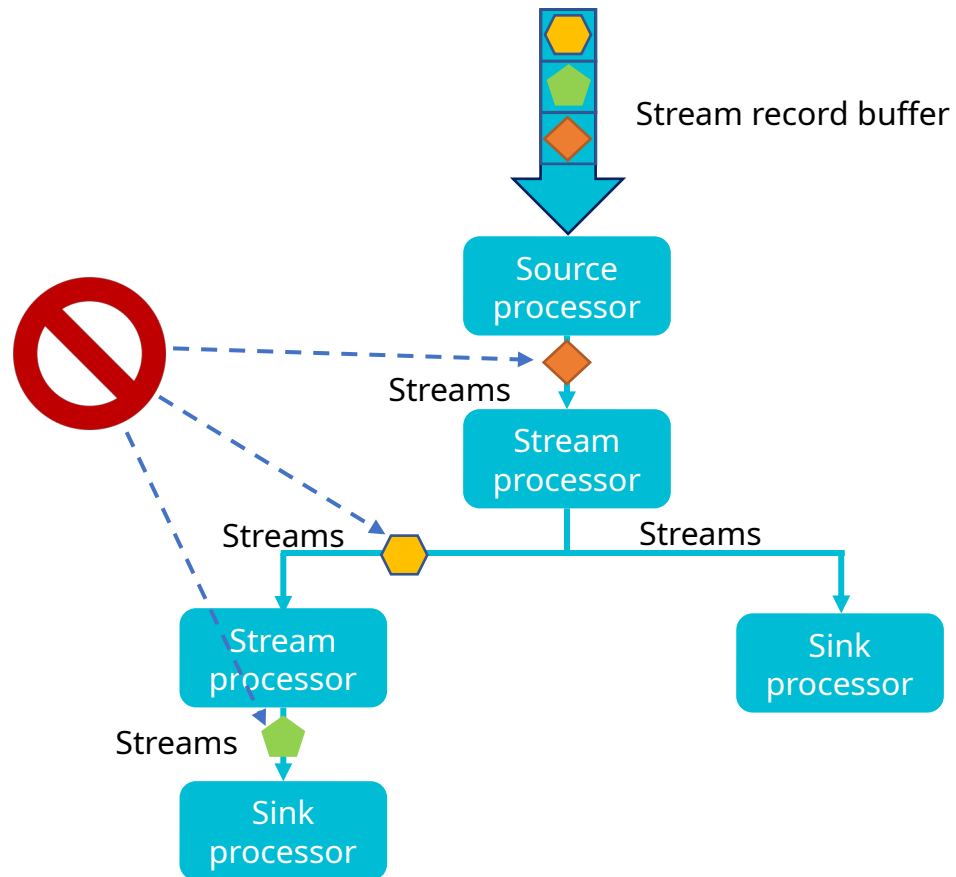
SUB-TOPOLOGIES



ABOUT

- Divide topology into smaller sub-topologies to parallelize the work
- Exception --> when topics are joined --> a single topology will read from each source topic involved in the join without further dividing the step into sub-topologies

DEPTH-FIRST STRATEGY PROCESSING



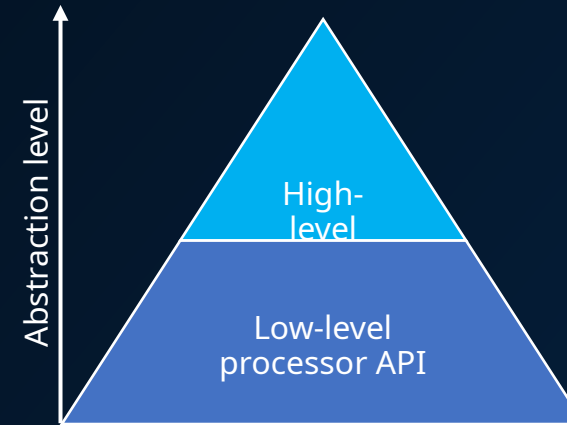
ABOUT

- A single record moves through the entire topology before another record is processed
- Makes the dataflow much easier to reason about,
- **Slow stream processing operations can block other records from being processed in the same thread**
- Note: When multiple sub-topologies are in play, the single-event rule does not apply to the entire topology, but to each sub-topology

High-Level DSL vs Low-Level Processor API

The two APIs:

- The high-level DSL
- The low-level Processor API



The high-level DSL

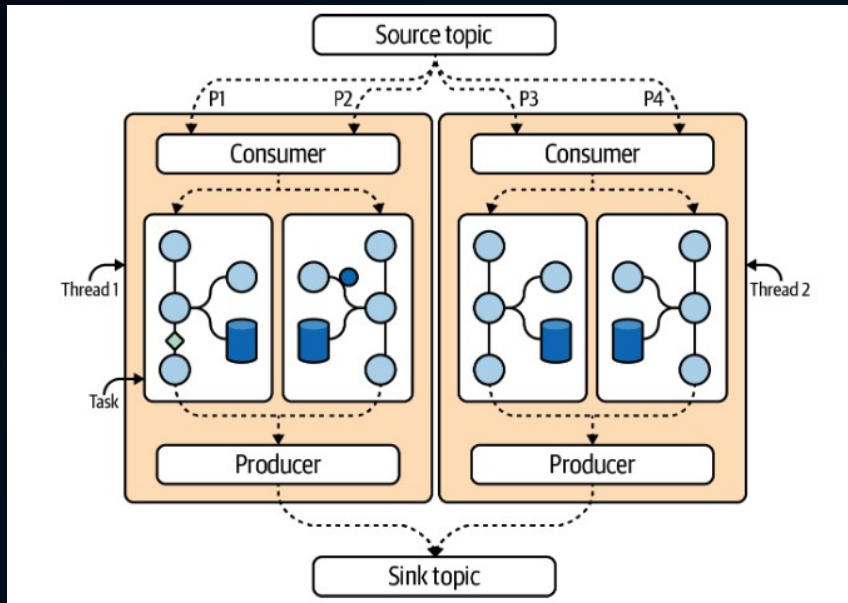
- Using a functional style of programming, and would also like to leverage some higher-level abstractions for working with your data (streams and tables)

The low-level Processor API

- Lower-level access to your data (e.g., access to record metadata),
- the ability to schedule periodic functions,
- more granular access to your application state,
- more fine-grained control over the timing of certain operations

Task and threads

num.stream.threads = 2

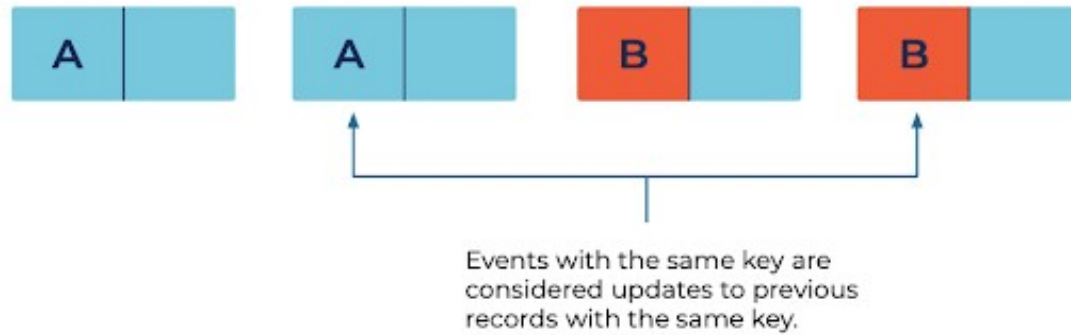


num.stream.threads = 4



UPDATE STREAMS

- Opposite of basic operations
- Note: If a new record comes in with the same key as an existing record, the existing record will be overwritten!



There are two ways to model the data in your Kafka topics:

- as a stream (also called a record stream) or
- a table (also known as a change log stream).

Streams

- thought of as inserts in database
- Each distinct record remains in this view of the log

Table 2-3. Stream view of ssh logs

Key	Value	Offset
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

Tables

- thought of as updates to a database
- logs only the current state (either the latest record for a given key or some kind of aggregation) for each key is retained
- usually built from compacted topics (cleanup.policy of compact)

Table 2-4. Table view of ssh logs

Key	Value	Offset
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

KStream

- A KStream is an abstraction of a partitioned record stream, in which data is represented using insert semantics (i.e., each event is considered to be independent of other events)

KTable

- an abstraction of a partitioned table (i.e., change log stream), in which data is represented using update semantics (the latest representation of a given key is tracked by the application)
- Since KTables are partitioned, each KafkaStreams task contains only a subset of the full table

GlobalKTable

- similar to a KTable,
- except each GlobalKTable contains a complete (i.e., unpartitioned) copy of the underlying data

State in Applications



Stateless applications

- each event handled by your Kafka Streams application is processed independently of other events, and only stream views are needed by your application
- treats each event as a self-contained insert and requires no memory of previously seen events
- operators, like filter, are considered stateless

VS

Stateful applications

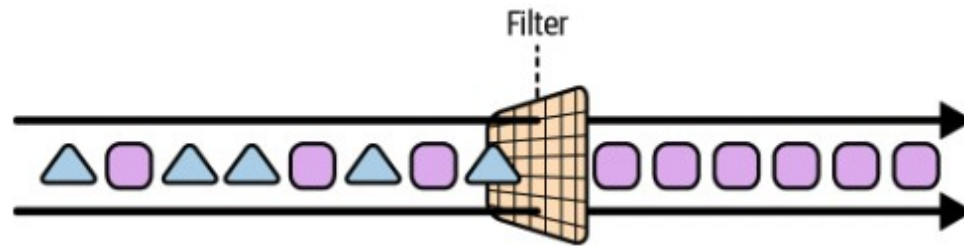
- need to remember information about previously seen events in one or more steps of your processor topology, usually for the purpose of aggregating, windowing, or joining event streams. These applications are more complex under the hood since they need to track additional data, or state
- operators, like count, are stateful

Definition:

- The simplest form of stream processing
- Requires no memory of previously seen events
- Each event is consumed, processed, and subsequently forgotten

Stateless operators:

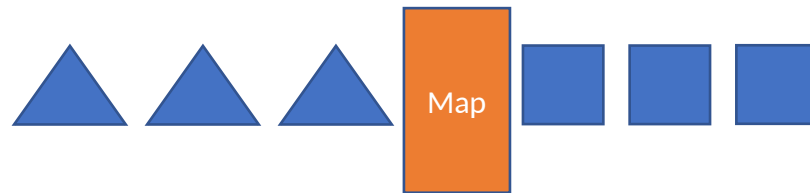
- Filtering records
- Adding and removing fields
- Rekeying records
- Branching streams
- Merging streams
- Transforming records into one or more outputs
- Enriching records



```
KStream<String, Long> stream = ...;  
// A filter that selects (keeps) only positive numbers  
KStream<String, Long> onlyPositives =  
    stream.filter(  
        (key, value) -> value > 0  
    );
```

ABOUT

- Two primary operators
 - filter
 - filterNot
- filter requires to pass in a Boolean expression (Predicate)
 - predicate returns **true**, the event will be forwarded to downstream processors
 - returns **false**, then the record will be excluded from further processing



```
KStream<byte[], String> stream = ...;
```

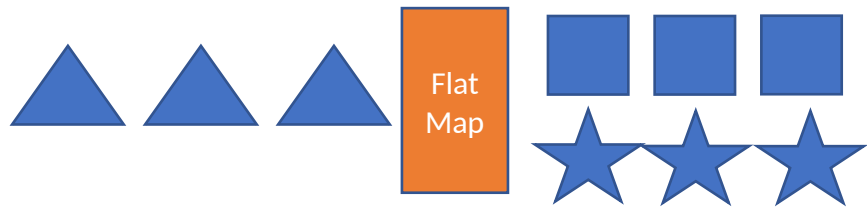
```
//Convert value's lowercase to key and the values length to the new value  
KStream<String, Integer> transformed =  
    stream.map(  
        (key, value) ->  
            KeyValue.pair(value.toLowerCase(), value.length())  
    );
```

ABOUT

- transforming one input record into exactly one new output record (whose key or value may or may not be the same type as the input record)

Two operators:

- map
 - mapValues
-
- a 1:1 mapping between input and output records
 - map ✉ requires to specify a new record value and record key
 - mapValues ✉ requires us to just set a new value



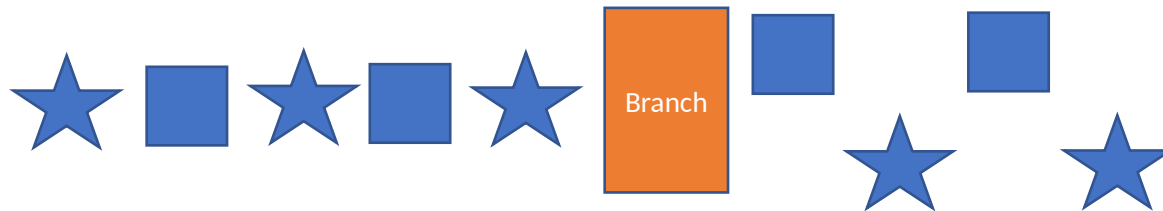
```
KStream<Long, String> stream = ...;
KStream<String, Integer> transformed = stream.flatMap(
    // Here, we generate two output records for each input record.
    // We also change the key and value types.
    (key, value) -> {
        List<KeyValue<String, Integer>> result = new LinkedList<>();
        result.add(KeyValue.pair(value.toLowerCase(), value.length()));
        result.add(KeyValue.pair(value.toUpperCase(), 42));
        return result;
    }
);
```

ABOUT

- Takes one record and produces zero, one, or more records. You can modify the record keys and values, including their types.

Two operators:

- flatMap
 - flatMapValues
-
- a 1:1 mapping between input and output records
 - map ✉ requires to specify a new record value and record key
 - mapValues ✉ requires us to just set a new value



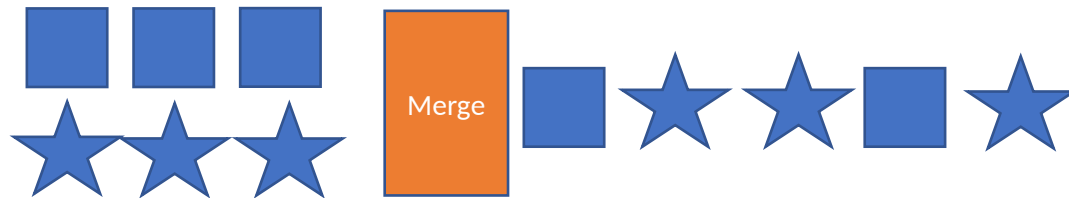
```
KStream<String, Long> stream = ...;
Map<String, KStream<String, Long>> branches =
    stream.split(Named.as("Branch-"))
        .branch((key, value) -> key.startsWith("A"), /* first predicate */
                Branched.as("A"))
        .branch((key, value) -> key.startsWith("B"), /* second predicate */
                Branched.as("B"))
        .defaultBranch(Branched.as("C")) /* default branch */
);

// KStream branches.get("Branch-A") contains all records whose keys start with "A"
// KStream branches.get("Branch-B") contains all records whose keys start with "B"
// KStream branches.get("Branch-C") contains all other records
```

ABOUT

Required when events need to be routed to different stream processing steps or output topics based on some attribute of the event itself

Note: Using the default branch as a errorcheck for e.g. unexpected values is a common setup.



```
KStream<byte[], String> stream1 = ...;
```

```
KStream<byte[], String> stream2 = ...;
```

```
KStream<byte[], String> merged = stream1.merge(stream2);
```

ABOUT

- two separate streams merging to one
- have the same stream/sink processor

There is no ordering guarantee between records from different streams in the merged stream. Relative order is preserved within each input stream though (ie, records within the same input stream are processed in order)



```
KStream<String, Long> stream = ...;
```

```
// Print the contents of the KStream to the local console.  
stream.foreach(  
    (key, value) -> System.out.println(key + " => " + value)  
);
```

ABOUT

You would use `foreach` to cause side effects based on the input data

Any side effects of an action (such as writing to external systems) are not trackable by Kafka, which means they will typically not benefit from Kafka's processing guarantees.

Definition:

- Ability to capture and remember information about the events we consume
- The captured information, or state, allows to perform more advanced stream processing operations,

Stateful operators:

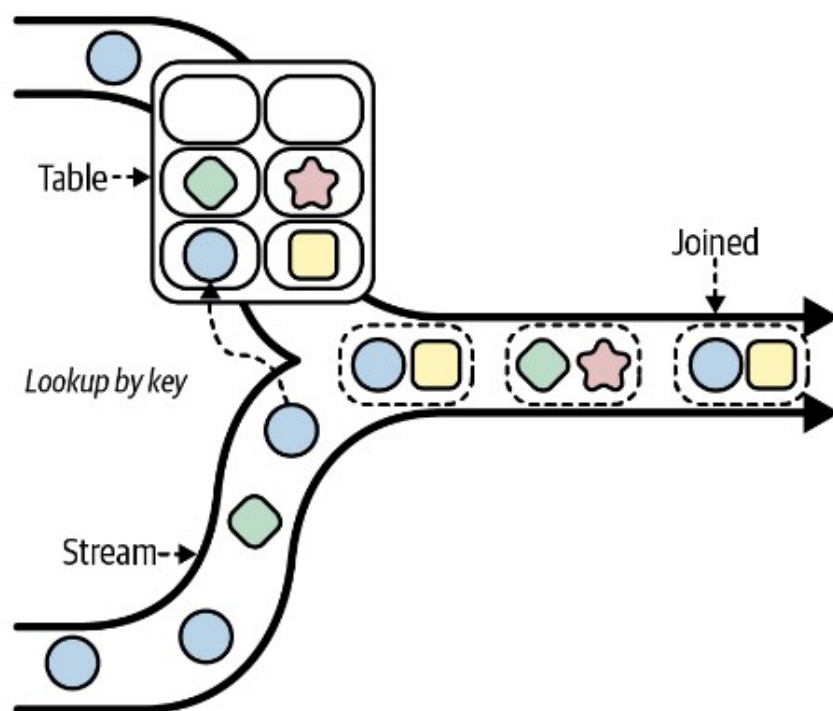
- Joining
- Aggregating
- Windowing

Benefits:

- helps to understand the relationships between events
- Recognize patterns and behaviors in our event streams
- Perform aggregations
- Enrich data in more sophisticated ways using joins
- gives an additional abstraction for representing data
- build a point-in-time representation of continuous and unbounded record streams
- allows to understand our data using more sophisticated mental models



JOINS



ABOUT

- Special kind of conditional merge that cares about the relationship between events, and where the records are not copied verbatim into the output stream but rather combined
- these relationships must be captured, stored, and referenced at merge time to facilitate joining, which makes joining a stateful operation

Type	Windowed	Operators	Co-partitioning required
KStream-Kstream	Yes	<ul style="list-style-type: none">• join• leftJoin• outerJoin	Yes
KTable-Ktable	No	<ul style="list-style-type: none">• leftJoin• outerJoin	Yes
KStream-Ktable	No	<ul style="list-style-type: none">• join• leftJoin	Yes
KStream-GlobalKTable	No	<ul style="list-style-type: none">• join• leftJoin	No

ABOUT

Kafka Streams supports many different types of joins,



Windows allow us to group events into explicit time buckets
can be used for creating more advanced joins and aggregations

WINDOWS & TIMES – WINDOWING

