SELA

Module 01: Kafka Core

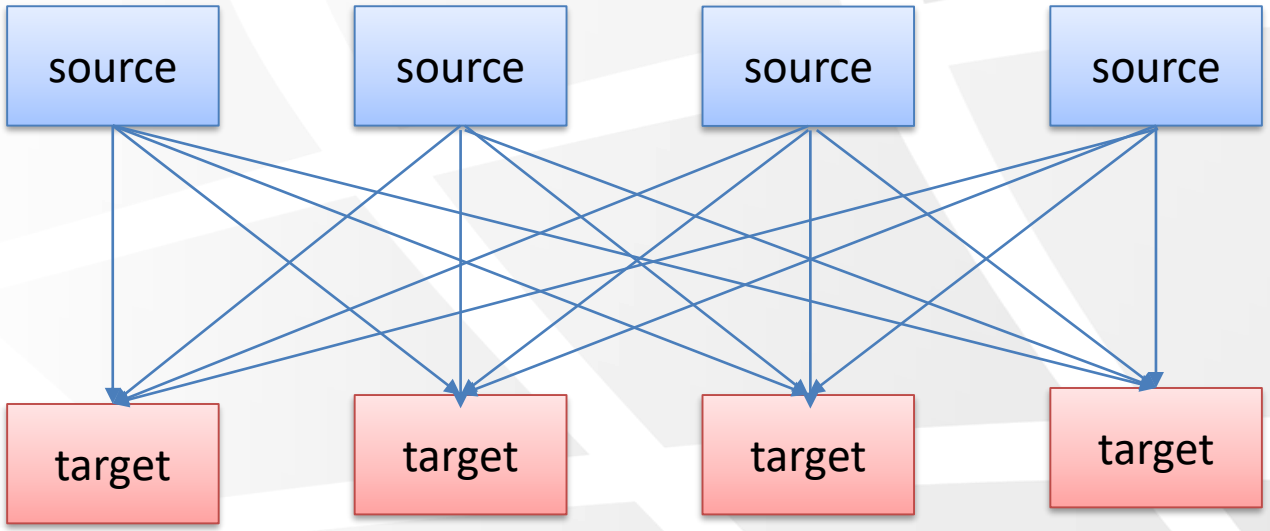
Course Agenda

- Module 01 – Kafka Core
- Module 02 – Installing Kafka
- Module 03 – Kafka CLI
- Module 04 – Programmatic API
- Module 05 – Advanced Programming
- Module 06 – Kafka Streams Overview
- Module 07 – Kafka EchoSystem and Administration

Agenda

- Kafka Brokers
- Topics and Partitions
- Producers
- Consumers and Consumer Groups

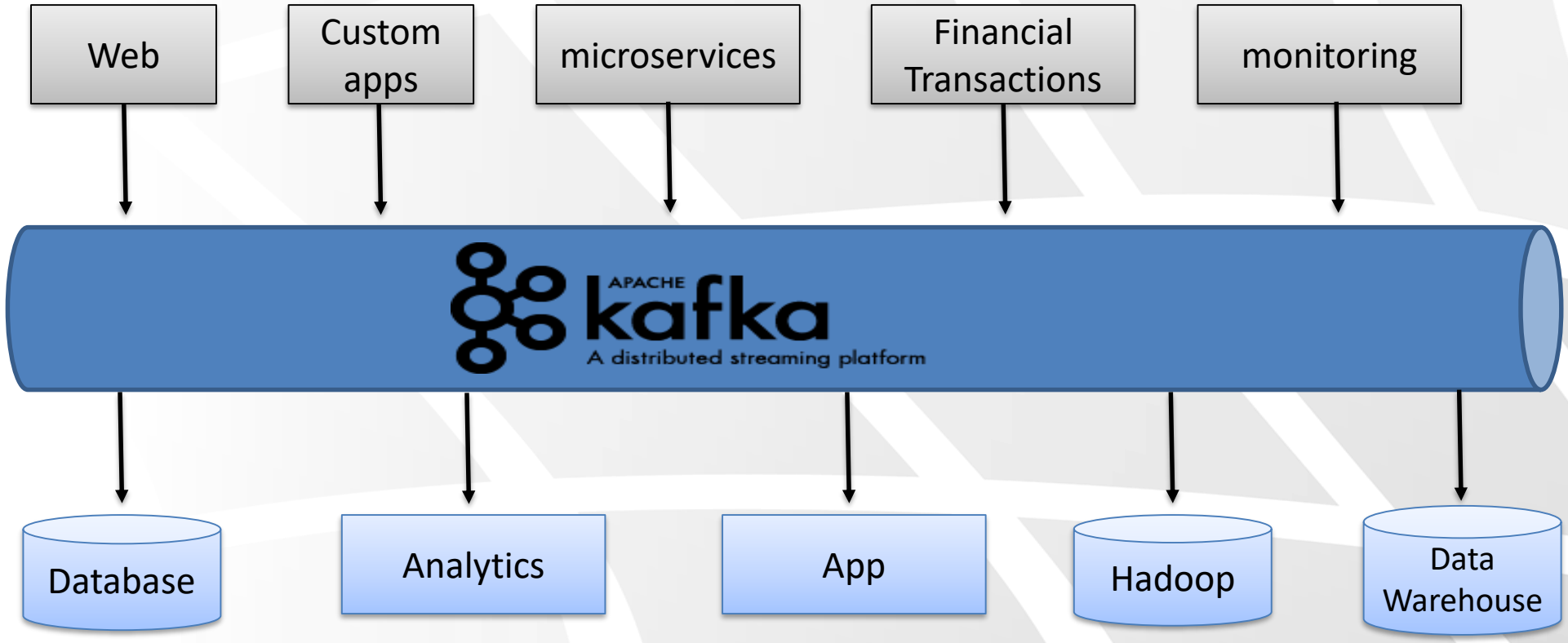
Defining the Decoupling problem



- 4 sources + 4 targets mean 16 integrations
- Protocols (TCP/Http/Rest/FTP)
- Data Format – how the data is parsed (Binary, Json, Avro, CSV)
- Data Schema & evolution – how a data is shaped and may change

Apache Kafka

- A high throughput distributed system



Why Apache Kafka

- Created by LinkedIn, now open source promoted by confluence
- Distributed
- Fault tolerant
- Resilient Architecture
- Scales Horizontally :
  - Can scale to hundreds of brokers
  - Can scale to millions of messages per second
- High Performance
- Low latency (less than 10ms)
- Used by 2000+ firms ,35% of the Fortune 500

Use Cases

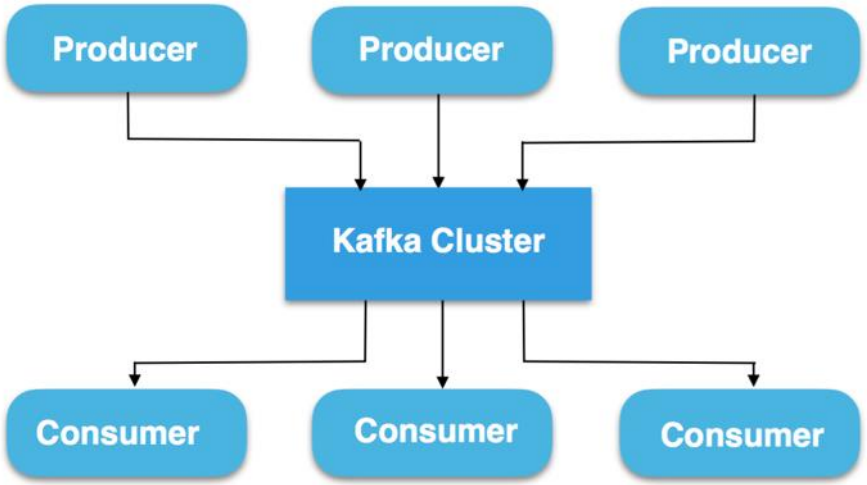
- Messaging System – Kafka has better throuput,build-in partitions, replication and fault-tolerance than most message broker systems.
- Activity Tracking – rebuild user activity pipeline as a pub/sub feeds
- Metrics gathering – operational monitoring data for distributed apps.
- Log Aggregation - cleaner abstractions of log as stream of data.
- Stream Processing – data processing in multiple stages (Kafka streams)
- Event Sourcing – large store log enables time-order sequence analysis

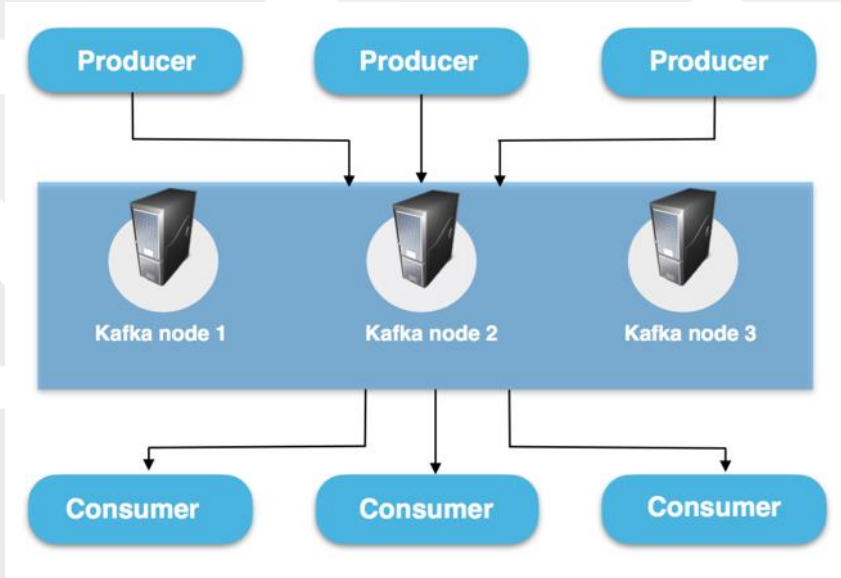
Use Cases

- NETFLIX uses Kafka to apply recommendations in real time while you're watching TV shows
- UBER uses kafka to gather user, taxi and trip data in real time to capture and forecast demand and compute surge pricing
- LinkedIn uses Kafka to prevent spam, collect user interactions to make better connection recommendation in real time.

Brokers

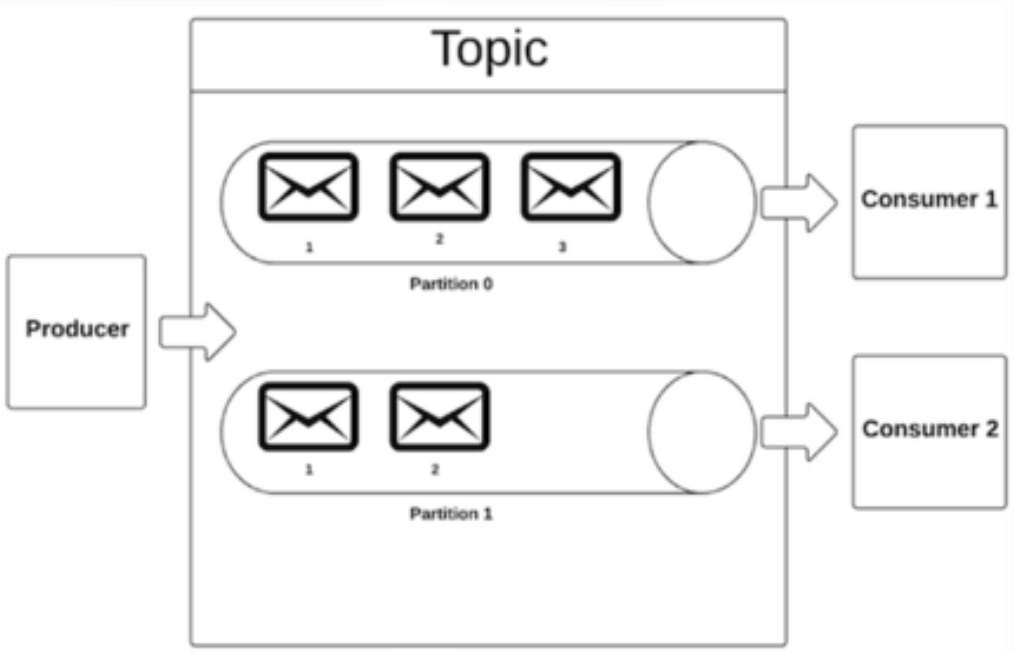
- A Kafka cluster consists one or more servers (a.k.a 'brokers')
- **Producers** are processes that publish data (push messages) into Kafka topic within the broker
- **Consumers** are processes that pulls (read) data off a Kafka topic





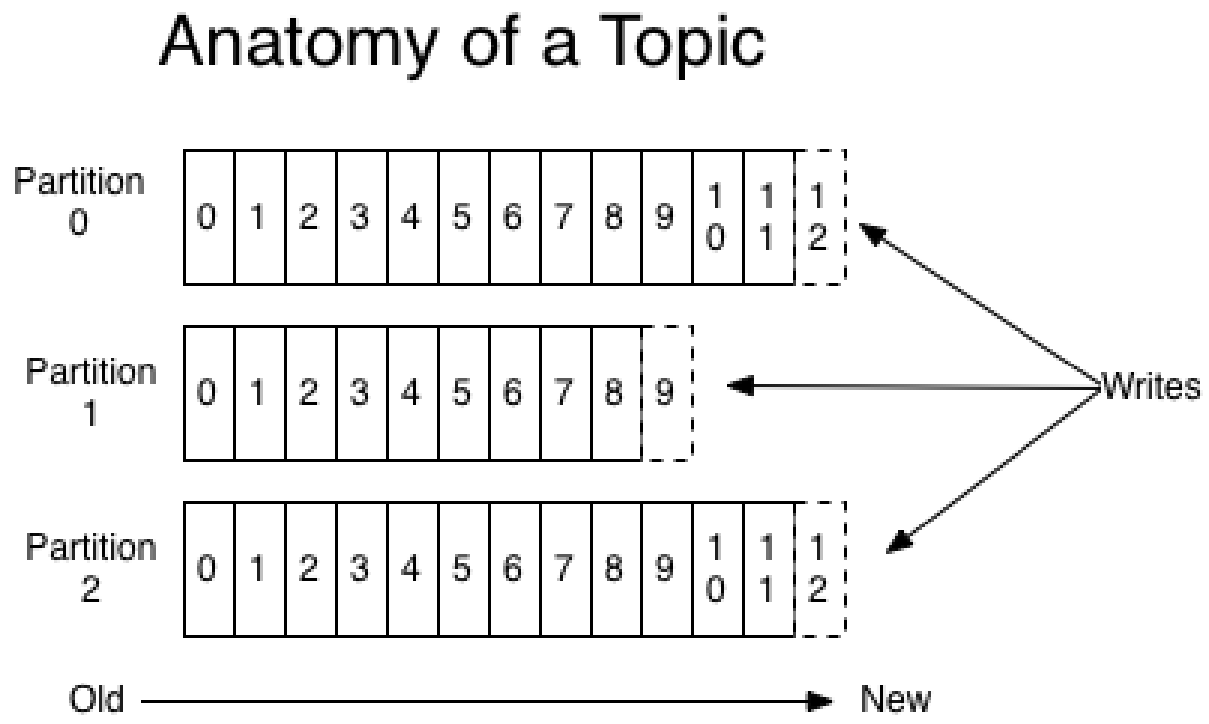
Topic

- A particular stream of Data
- Can be think of as a 'Category' or a table in a Database
  (without all the constraints)
- Each Topic has a unique name
- Producers and Consumer write/read from a specific topic



Anatomy of a Topic

- Topics Are Split into partitions :
- Each Partition is ordered
- Each message within a partition gets an incremental Id called offset
- Data in partitions can be retained for a configurable amount of time (default = One Week)

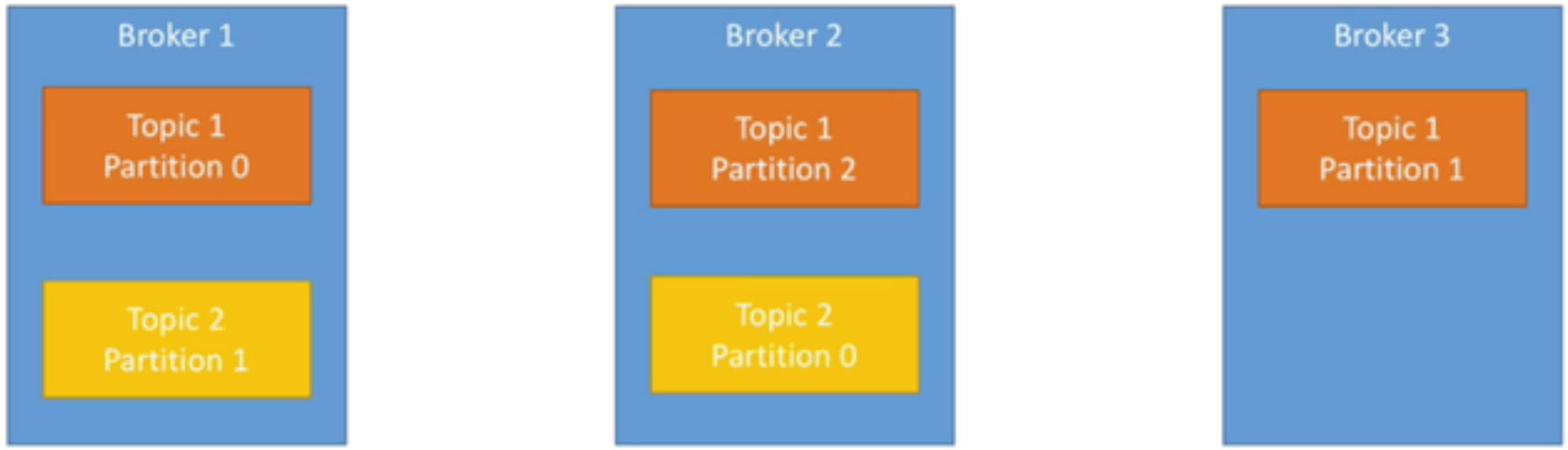


Anatomy of a Topic

- Once data is written to a topic it cannot be changed (immutability)
- Data is assigned randomly to a partition unless a key is specified
- You can have as many partitions per topic as you want
- Order is guaranteed only within a partition

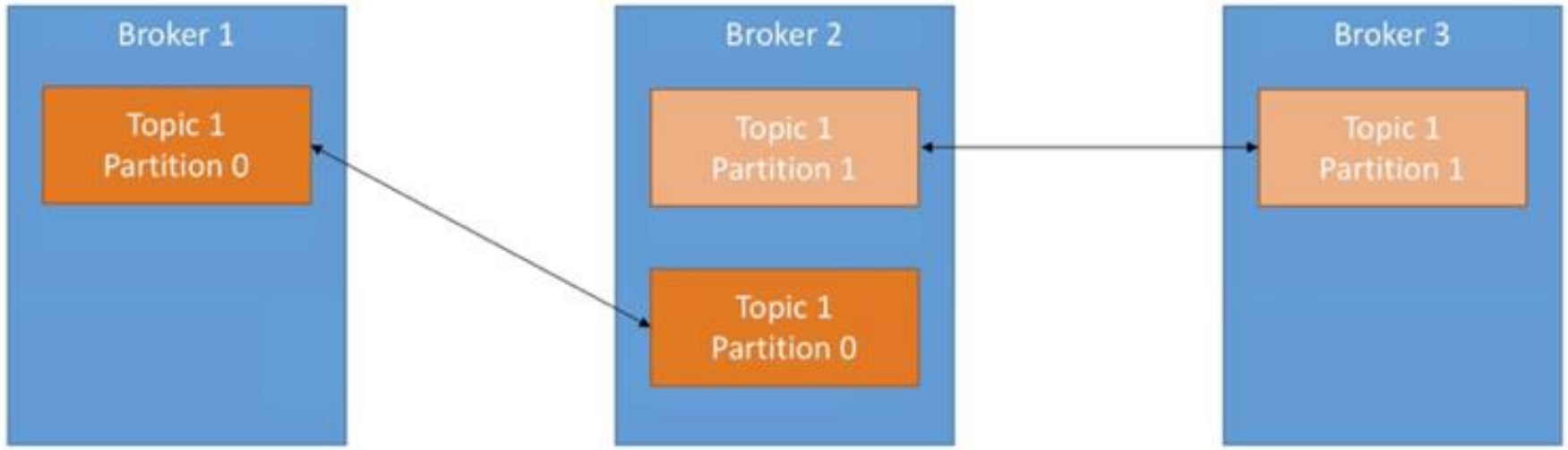
Brokers

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker contains certain topic partitions
- After connecting to any broker (called bootstrap server),you will be connect to the entire cluster



Topic Replication Factor

- Replication factor determines how data is replicated across the nodes
- This allows Kafka to automatically failover to the replica when a server in the cluster fails
- Replica happens at the partition granularity
  - Example : Topic with 2 partitions and replication factor of 2

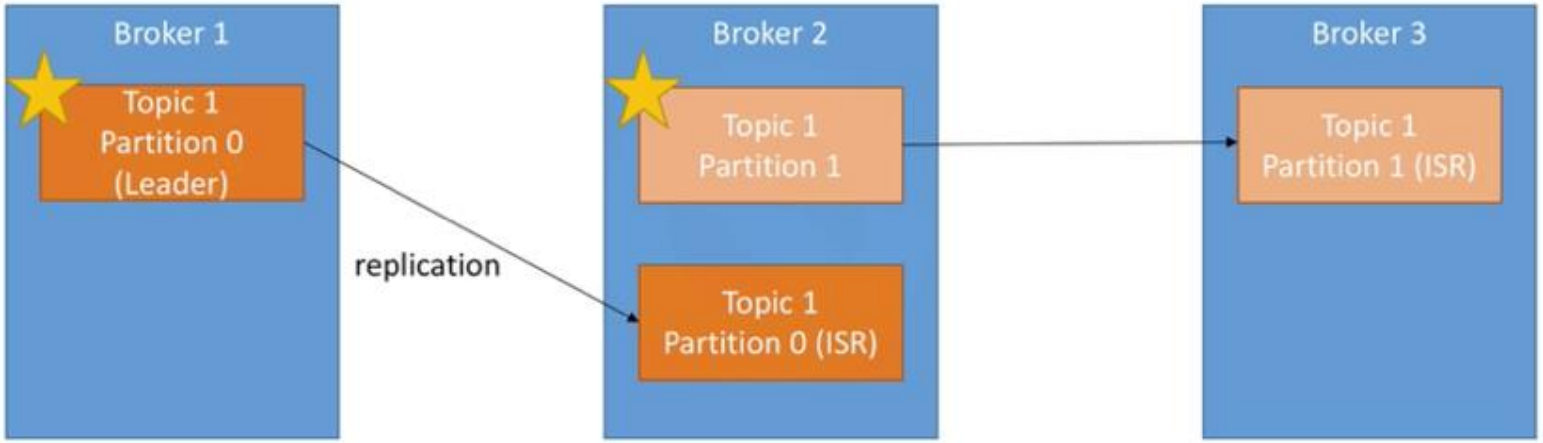


Topic Replication Factor

- Example : Loosing broker 2 can still serve the data

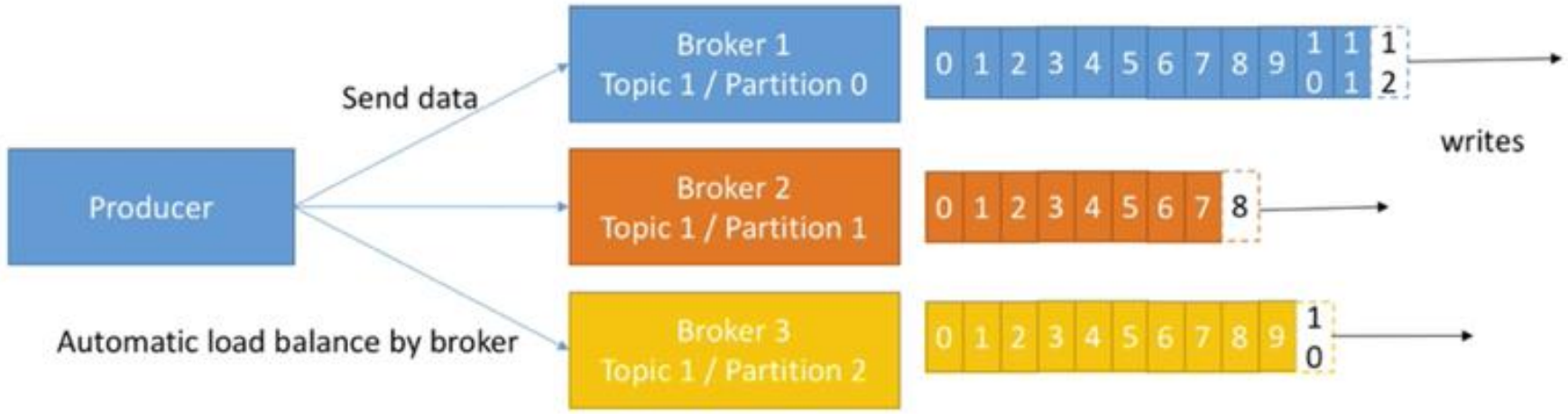
Topic Leader

- One replica is designated as the “leader” while other are followers.
- At a given time only 1 broker can be a leader for a given partition
- Only that leader can send and receive data
- The other partitions will sync to the leader, also called **ISR** - In Sync Replica



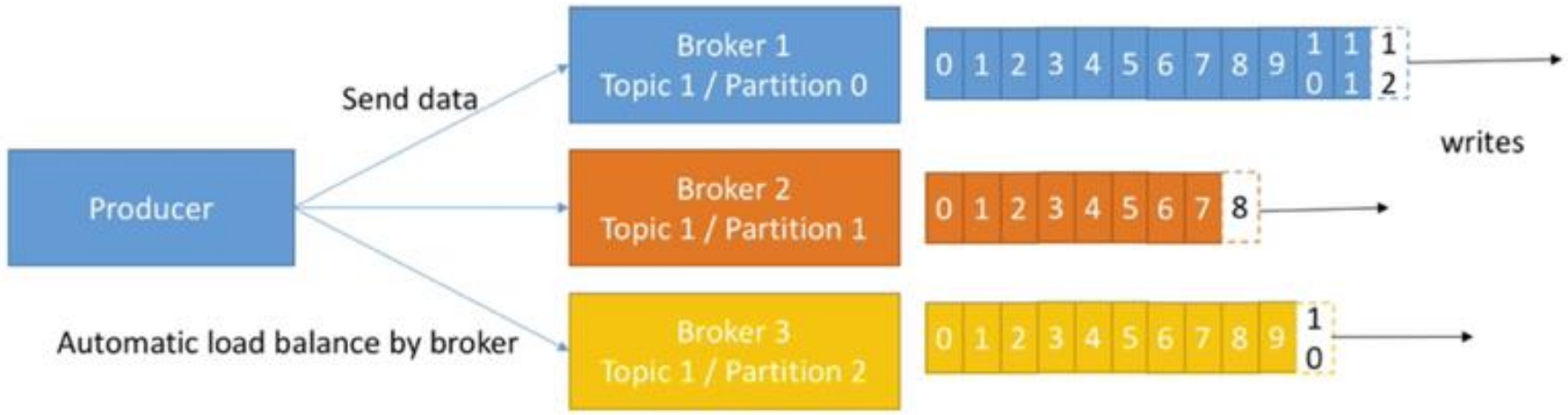
Producers

- Producers write data to the partitions
- In order to send data a producer needs to know:
  - Topic name
  - At least one broker to connect to
  - Kafka will do the routing to the right brokers



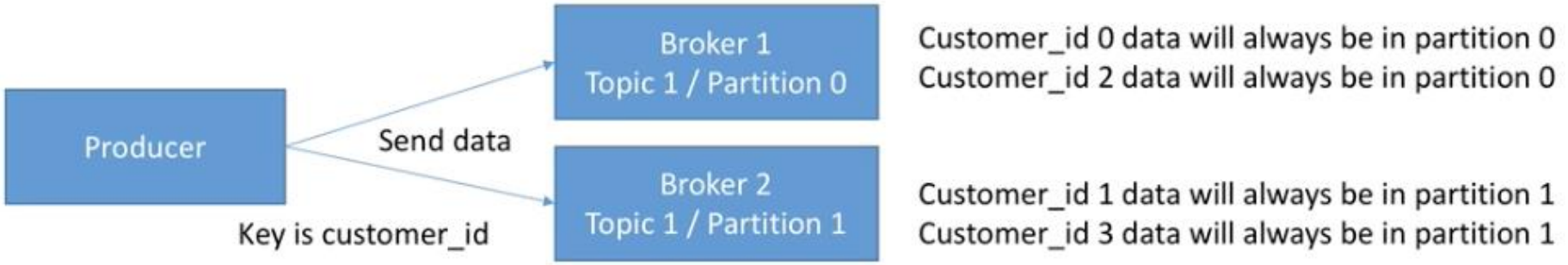
Producers

- Producers can choose to to receive acknowledgment of data writes:
  - Acks = 0 : No waiting for acknowledgment (possible data loss)
  - Acks = 1 : Producer will wait for leader acknowledgment (limited data loss)
  - ,



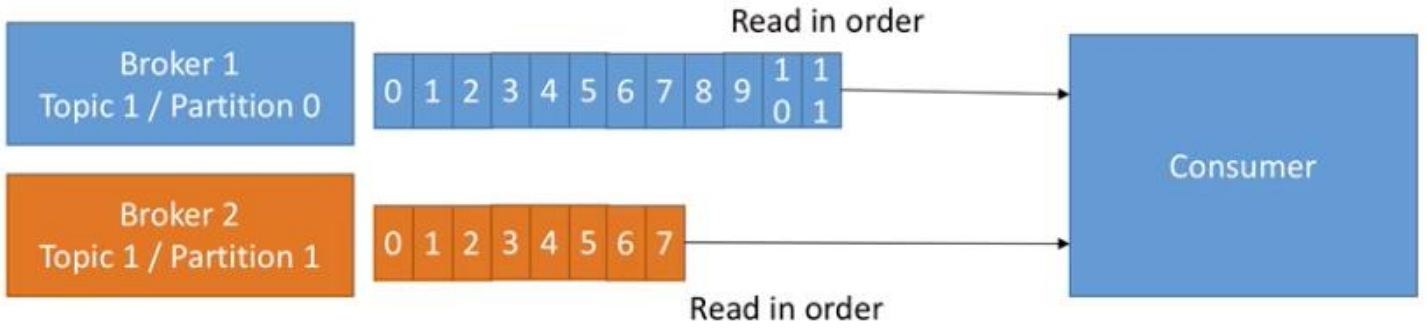
Producers Message Keys

- Producers can choose to send a Key with a message
- When a key is sent , the producer has the guarantee that all message with that key will always go to the same partition.
- This enables Ordering for a specific Key !



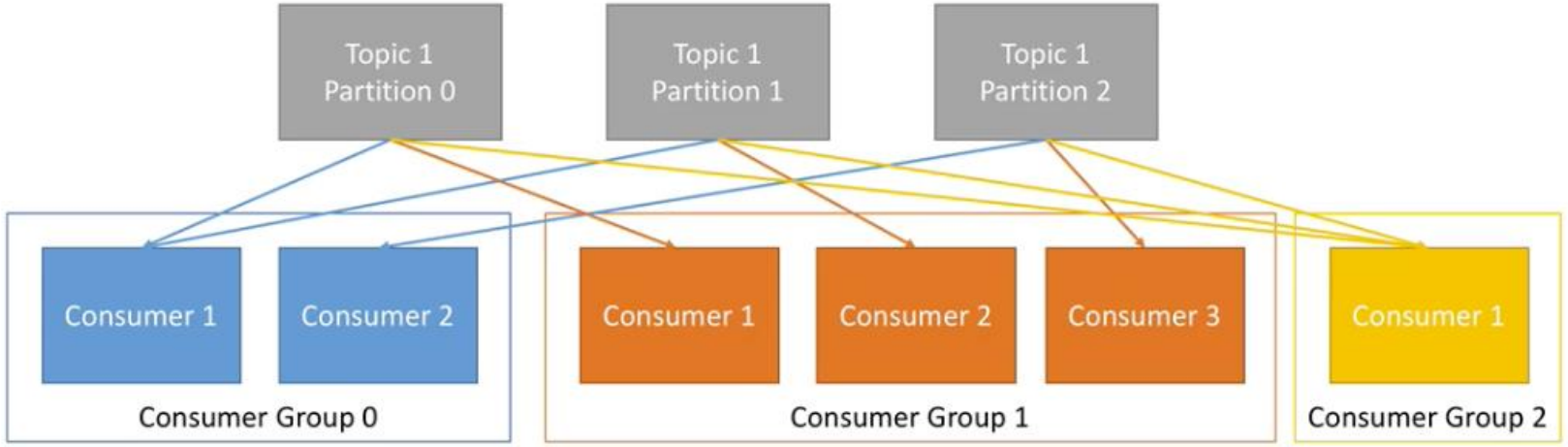
Consumers

- Consumers read data from a partition
- Consumer need to specify :
  - The topic name
  - One broker from the broker list to connect to -> Kafka will take care of pulling the right data from the right broker.
- Data is read in-order for each partition.



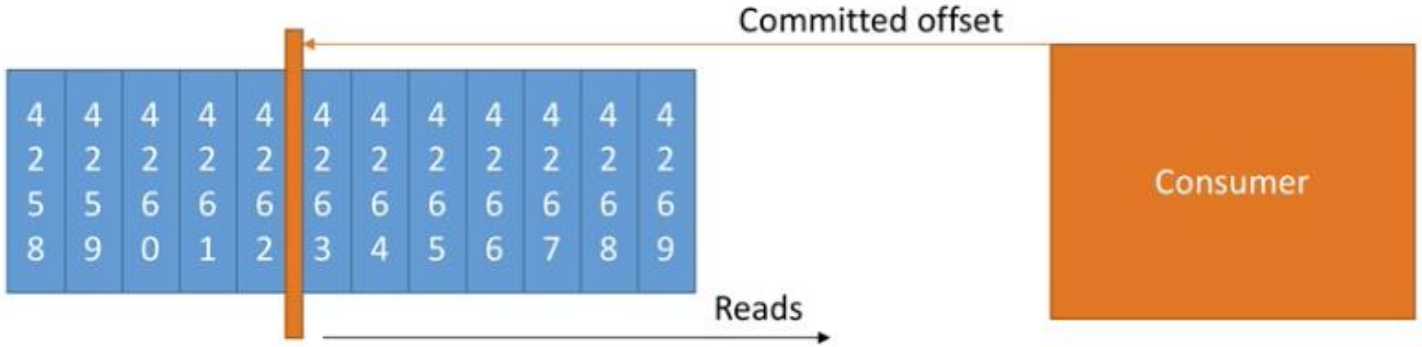
Consumer Groups

- Consumer read data within a specific consumer group
- Each consumer read from exclusive partitions
- You cannot have more consumers than partitions (otherwise some will become inactive)



Consumer Offsets

- Kafka Stores the offsets at which a consumer group has been reading.
- The offset commits are store in a dedicated topic called "__consumer_offsets"
  - When consumer has processed the data successfully It should commit the offset
  - If a consumer process "crashes", it will be able to read back from where it left.

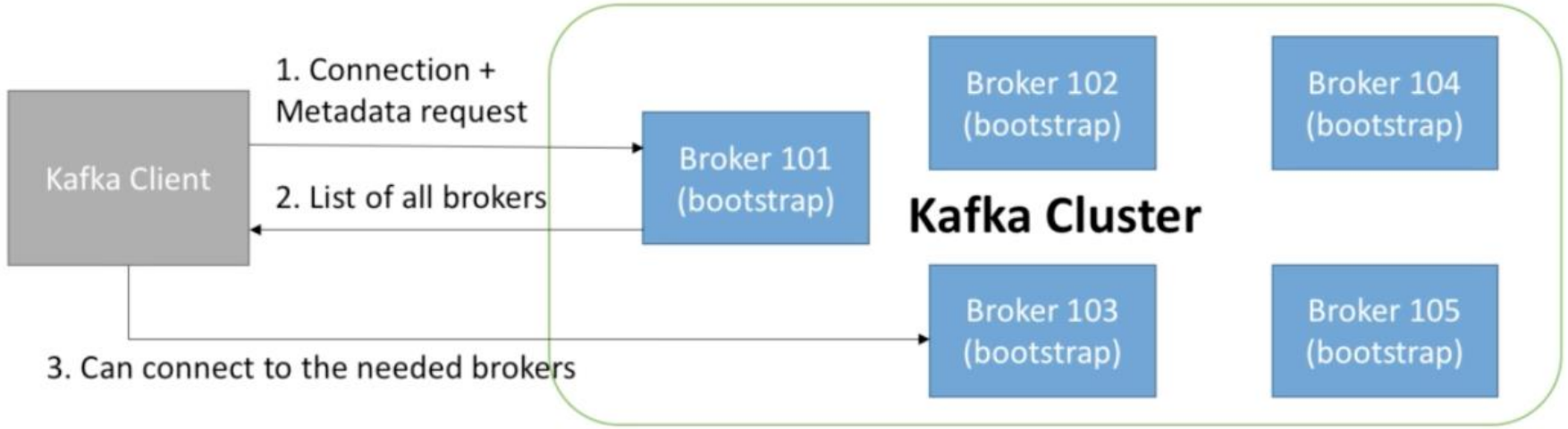


Delivery Semantics for consumers

- Consumers can choose when to commit offsets.
- There are 3 delivery semantics options:
  - At Most Once
    - Offsets are committed as soon as the message is received.
    - If the processing goes wrong, the message will not be read again.
  - At least Once (preferred)
    - Offsets are committed after message is processed.
    - If the processing goes wrong, the message will be read again.
    - Make sure processing the same message twice wont impact your system.
  - Exactly Once
    - Can only be achieved from Kafka to Kafka using K-Streams API.

Kafka Broker Discovery

- Every kafka broker is called a “bootstrap server”
- That mean you only need to connect to one broker and you will be connected to the entire cluster
- Each broker knows about all the brokers, topics and partitions



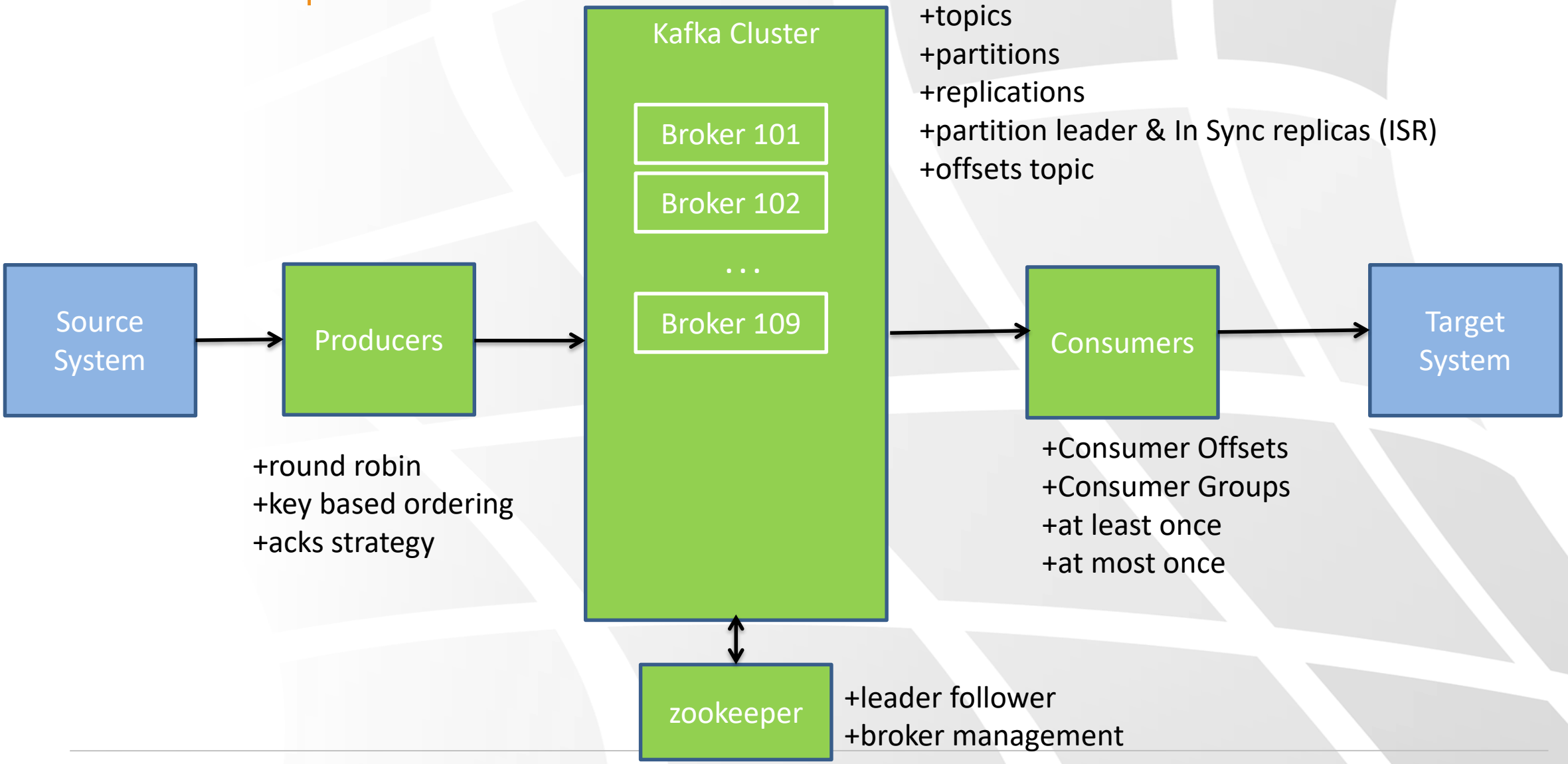
Zookeeper

- Zookeeper manages brokers (keeps a list of them)
- Zookeeper helps in performing leader election for partitions
- Zookeeper sends notifications to kafka in case of changes (e.g. new topic, broker dies, broker recovery, delete topics, etc.)
- **Kafka Can't work without Zookeeper**
- By design works with odd number of servers (3,5,7)
- Zookeeper has a leader (handles writes), the rest are followers
- Zookeeper does not store consumer offset (with Kafka > v0.10)

Kafka Guarantees

- Messages are appended to a topic-partition in the order they are sent
- Consumers read messages in the order stored in a topic-partition
- With replication factor of N, producers and consumers can tolerate up to N-1 brokers being down.
  - Replication factor of 3 :
    - One broker can be down for maintenance
    - Another broker can be down unexpectedly
- As long as the number of paritions does not change, the same key will always go to the same topic.

Roundup


Kafka Cluster
Broker 101
Broker 102
. . .
Broker 109
+topics
+partitions
+replications
+partition leader & In Sync replicas (ISR)
+offsets topic
Source System
Producers
+round robin
+key based ordering
+acks strategy
Consumers
+Consumer Offsets
+Consumer Groups
+at least once
+at most once
Target System
zookeeper
+leader follower
+broker management

Summary

- Kafka Data is organized into streams of data called “Topics”
- Each topic is organized within partitions.
- A producer sends message with a key that guarantees an assignment to a specific partition.
- Consumers are organized into consumer groups
- Each consumer has a exclusive sets of partition he is assigned to.
- Group offset are maintained by committing an offset by the consumer.

SELA

Module 02: Installing Kafka

Install Process

- Typically Kafka Installation will contain the following components :
  - Kafka Brokers
  - Zookeeper
  - Schema Registry (Optional)
- You can install each of those separately but we will install Docker in order to enable fast deployment of all the components in one container

Manual Installation

- Download the Kafka binaries from https://kafka.apache.org/downloads
- The bin folder contains shell scripts for windows and Linux
- You need to run first zookeeper (from the root folder):

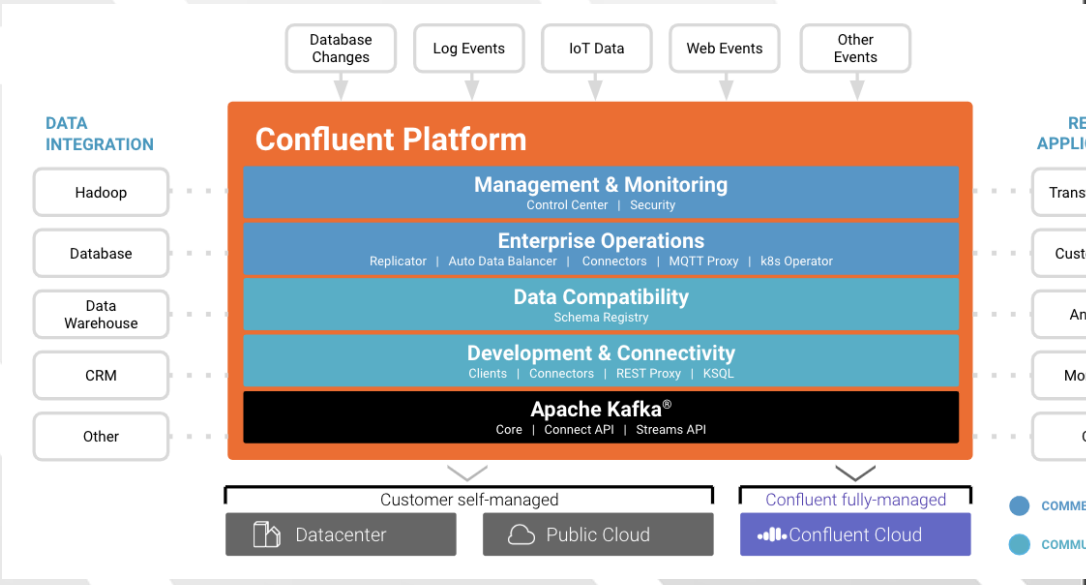
```
$ bin/zookeeper-server.start.sh config/zookeeper.properties
```

- Than start Kafka brokers:

```
$ bin/kafka-server.start.sh config/server.properties
```

Confluent Platform

- Was built from the creators of Apache-kafka to enable enterprise solution for kafka
- Confluent Platform contains all of Kafka Echo system
  - Brokers
  - Kafka Streams + KSQL
  - Data Connectors
  - Client libraries (C++/python/ Go / Java/ .
  - Schema registry and the avro format
  - Monitoring, Control Center



Dev Environment (fast-data-dev image for Docker)

```
# Docker for Mac >= 1.12, Linux, Docker for Windows 10


$docker run --rm -it \
        -p 2181:2181 -p 3030:3030 -p 8081:8081 \
        -p 8082:8082 -p 8083:8083 -p 9092:9092 \
        -e ADV_HOST=127.0.0.1 \
        landoop/fast-data-dev
```

Kafka Endpoints

- A typical Kafka installation will use the following ports for several kafka services :
  - Kafka Brokers : port 9092 -> *
  - Zookeeper : port 2181
  - Schema Registry : 8081
  - Rest Proxy : 8082
- These can be configured in zookeeper.properties/server.properties
- During this course we assume a kafka installation in at ~/kafka
- Our executables will be at ~/kafka/bin

Starting Landoop fast-data-dev environment

Demo

Lab 01: Interacting with Ladoop Kafka Environment

Lab

https://github.com/selagroup/KafkaWorkshop/blob/master/Lab-01.md

Module 03: Kafka CLI

Kafka Workshop

Agenda

- Kafka CLI
- List All Topics
- Create and remove a topic
- Kafka Producer
- Kafka Consumer
- Consumer Groups

The Kafka CLI

- CLI tool can be accessed by downloading the Kafka Binaries:

https://kafka.apache.org/downloads

- Make sure to add the /bin folder to your PATH

- If you Want to work with a predefined docker image :
  - docker run --rm -it --net=host landoop/fast-data-dev bash

$ kafka-topics

```
$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --list
```

```
$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic myTopic --partitions 3  --replication.factor 2
```

```
$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic myTopic --describe
```

- Create/Delete/Describe or Change a Topic
- Need to specify the zookeeper address
- Flags:
  - --list : list all topics
  - --create –topic topic_name
  - --partitions
  - --replication-factor
  - --describe

$ kafka-console-producer

```
$ kafka-console-producer --broker-list 127.0.0.1:9092 --topic myTopic
```

- Publish data to kafka topic using a strings as values and keys
- When no key is specified a random key is generated.
- If The topic does exists, Kafka will produce an error ,but will create the topic.

$ kafka-console-consumer

```
$ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic myTopic
```

```
$ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic myTopic --from-beginning
```

- Reads data from a default Consumer group into the standard output
- Note the --bootstrap-server flag vrs --broker-list flag
- Default behavior will read from the latest committed offset
- Read options includes
  - --from-beginning
  - --group group.id=group1
  - --max-messages 100

Consumer Groups

- What happens when you run the consumer command over and over again on the same consumer group ?

```
$ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic myTopic --group grp1  --from-beginning
```

$ kafka-consumer-groups

```
$ kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --describe --group grp1
```

```
$ kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --list
```

- Lists all consumer groups, describe a consumer group ,delete and reset.
- --reset-offsets allow resetting the consumer group offset
  - --to-date-time
  - --by-period
  - --to-earliest
  - --to-latest
  - --shift-by

```
$ kafka-consumer-groups --bootstrap-server 127.0.0.1:9092 --reset-offsets --shift-by --2 --group grp1 --execute
```

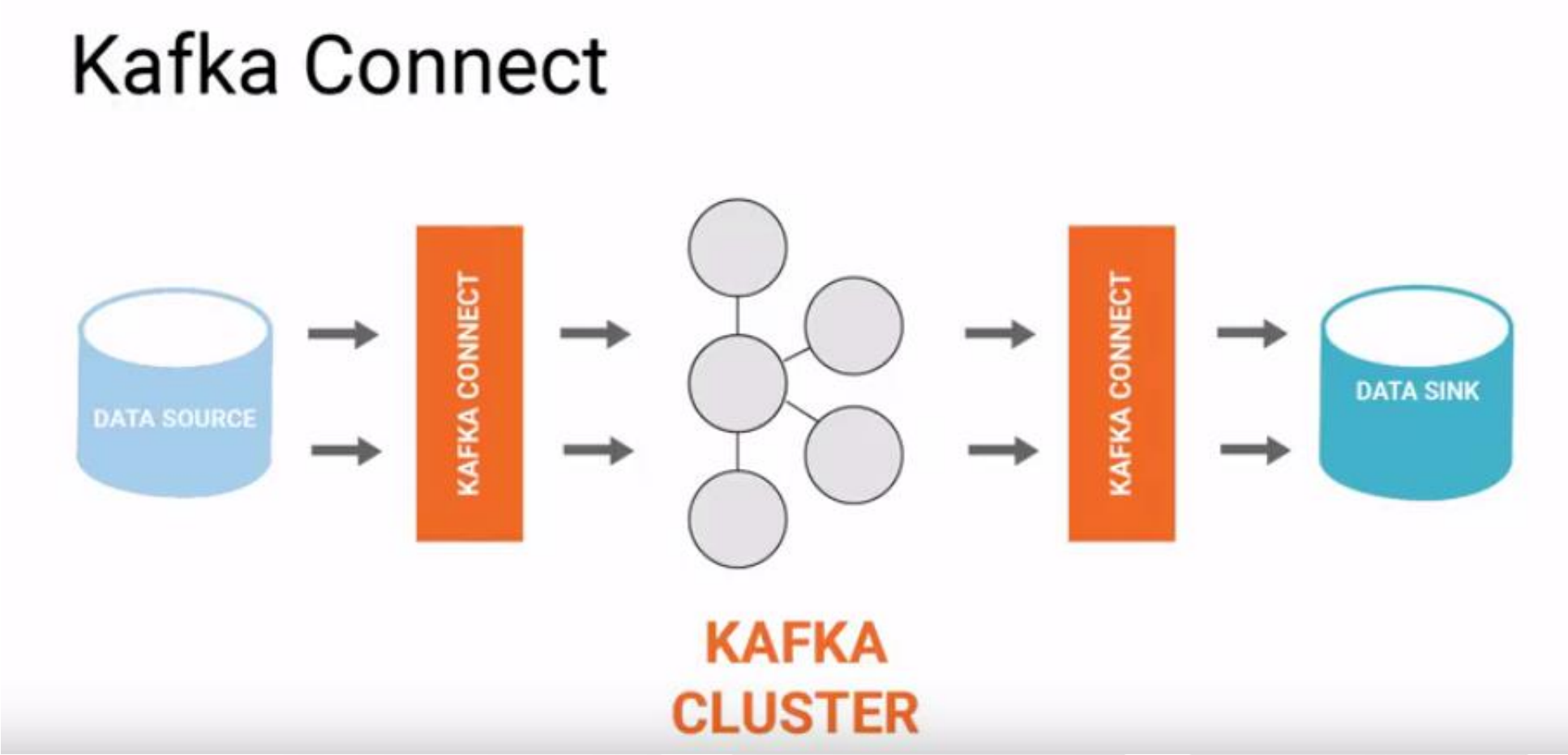
Producer/Consumer with a custom ‘Key’

```
$ kafka-console-producer --broker-list 127.0.0.1:9092 --topic myTopic \
 --property "parse.key=true" \
  --property "key.separator=:"
```

- It is possible to specify a dedicated key when publishing a message
- Kafka will hash all keys into the same partition Id

```
$ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic myTopic    \
 --property ”print.key=true" \
  --property "key.separator=:"
```

Kafka Connect with the CLI Tool (Module 7)



Lab 02: Basic commands

Lab

https://github.com/selagroup/KafkaWorkshop/blob/master/Lab-02.md

Questions

Module 04: Programmatic API

Kafka Workshop

Agenda

- The Kafka SDK For Java
- Producer API
- Consumer API
- Consumer Groups and rebalance
- Using Configuration files
- Using Custom Serializers

Producer API

- The Producer API allows applications to send streams of data to Kafka

https://kafka.apache.org/documentation/#producerapi

- Maven Config :

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.1.0</version>
</dependency>
```

Producer Config

- bootstrap.servers : a list of one or more of borkers
- key.serializer/value.serializer : the serialization classes for the keys and values
- acks (0/1/all) : request for either the leader or its replicants to send ack after sending.
- retries : In case of failure this field controls the retry count. Effects Ordering !
- linger.ms : batches all the request that arrive within the time frame and send them as single batch. This may effect latency, but reduces number of

Building and using a Producer

```
public static KafkaProducer<String,String> BuildProducer(String brokers)
{
    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers",brokers);
    properties.setProperty("key.serializer", StringSerializer.class.getName());
    properties.setProperty("value.serializer", StringSerializer.class.getName());

    properties.setProperty("acks", "1");
    properties.setProperty("retries","3");
    properties.setProperty("linger.ms","1");

    return new KafkaProducer<>(properties);

}
```

```
producer.send(new ProducerRecord<String, String>(topic   , sentence));
```

Consumer Config

- bootstrap.servers : a list of one or more of borkers
- key.deserializer/value.deserializer : the deserialization classes for the keys and values
- enable.auto.commit :consumer will commit the offset automatically in the background.
- group.id : the consumer group id.
- auto.offset.reset : behavior when there is no offset (earliest/latest/none)

Building and using a Consumer

```
public static KafkaConsumer<String,String> BuildConsumer(String brokers, String groupId)
{
    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers",brokers);
    properties.setProperty("bootstrap.servers",brokers);
    properties.setProperty("key.deserializer", StringDeserializer.class.getName());
    properties.setProperty("value.deserializer", StringDeserializer.class.getName());

    properties.setProperty("enable.auto.commit","true");
    properties.setProperty("group.id",groupId);
    properties.setProperty("auto.offset.reset","earliest");

    KafkaConsumer<String,String> consumer = new KafkaConsumer<~>(properties);
    return consumer;
}
```

```
consumer.subscribe(Arrays.asList(topic));
ConsumerRecords<String,String> records = consumer.poll(Duration.ofMillis(500));
```

Simple Producer and Consumer

Demo

Working with config files

- It is advices to work with configuration files to describe the various kafka configuration.
- A recommended pattern is to use the HCON file format with the *TypeSafe* maven repository to describe the configuration

```
<dependency>
    <groupId>com.typesafe</groupId>
    <artifactId>config</artifactId>
    <version>1.3.3</version>
</dependency>
```

```
{
  kafka {
    topic-name: "my-topic",
    producer {
      bootstrap.servers = "localhost:9092",
      key.serializer = org.apache.kafka.common.serialization.StringSerializer,
      value.serializer = org.apache.kafka.common.serialization.StringSerializer,
      acks = 1,
      retries = 3,
      linger.ms = 1
    },
    consumer {
      bootstrap.servers = "localhost:9092",
      key.deserializer = org.apache.kafka.common.serialization.StringDeserializer,
      value.deserializer = org.apache.kafka.common.serialization.StringDeserializer,
      enable.auto.commit = true,
      group.id = grp1,
      auto.offset.reset = "earliest"
    }
  }
```

Working with consumer groups and re-balance

- When we ran the several consumer processes on the same consumer group a rebalance phase will occur.

- Look for the “AbstractCoordinator” logs that describe the re-assignments of partitions across consumers , e.g.
  - Revoking previously assigned partitions[my-topic-0,my-topic-1]
  - Setting newly assigned partitions[my-topic-0]

```
s.AbstractCoordinator - [Consumer clientId=consumer-1, groupId=grp12] (Re-)joining group
s.AbstractCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Successfully joined group with generation 36
s.ConsumerCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Setting newly assigned partitions [my-topic-0, my-topic-1]
s.AbstractCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Attempt to heartbeat failed since group is rebalancing
s.ConsumerCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Revoking previously assigned partitions [my-topic-0, my-topic-1]
s.AbstractCoordinator - [Consumer clientId=consumer-1, groupId=grp12] (Re-)joining group
s.AbstractCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Successfully joined group with generation 37
s.ConsumerCoordinator - [Consumer clientId=consumer-1, groupId=grp12] Setting newly assigned partitions [my-topic-0]
umer - got record #1 - partition:0 - offset : 1683
```

Interacting with Zookeeper

- In case you want to create/delete and manage topics, you will need to interact with zookeeper itself .
- The AdminZkClient is a utility class that can assist in those tasks :
  - Create, delete and modify a topic
  - Add partitions to a topic
  - List all topics and their configuration
  - Change the broker configuration

```
private static void listTopics() {
    KafkaZkClient zkClient = KafkaZkClient.apply(zookeeperHost, isSucre, sessionTimeoutMs,
            connectionTimeoutMs, maxInFlightRequests, time, metricGroup, metricType);

    AdminZkClient adminZkClient = new AdminZkClient(zkClient);
    Map<String,Properties> topics = adminZkClient.getAllTopicConfigs();
    scala.collection.Iterator iter = topics.keysIterator();
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }

}
```

Assign and Seek

- Sometimes it is desirable to read from a specific topic or even a certain offset
  - Playback “bad” input stream that was streamed incorrectly
  - Logging and tracing
  - Performance testing
- The class “*TopicPartition*” determines the partition from which to read from a specific topic
- The Assign and Seek method assist in seeking
- No consumer group is involved in this process.

Assign and Seek

Demo

Lab 03: Java API and Re-balance

Lab

https://github.com/selagroup/KafkaWorkshop/blob/master/Lab-03.md

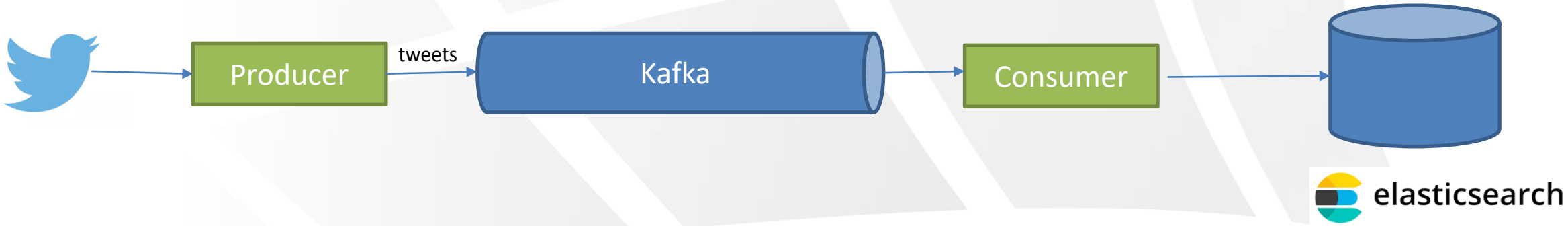
Questions

Module 05: Advanced Programming & Delivery Semantics

Agenda

- Advance sample app with Twitter API integration
- High Throughput and Safe Producer
- Message Compression
- Kafka Elastic Search Consumer + Advance Config
- Idempotent Producer and Consumer
- Delivery Sematcis

Twitter to Elastic Search Demo



- We will build a highly scalable twitter collector using kafka API Concepts:
- Producer will be reliable (retries mechanism, compression, batch)
- Consumer delivery semantics (“at-most-once” / “at-least-once”/ etc)
- Offset groups reset in case we want to “replay”

Producers Acks = 0 (no acks)

- No response is requested
- If the broker goes offline or an exception is raised, we don't know and we loose data

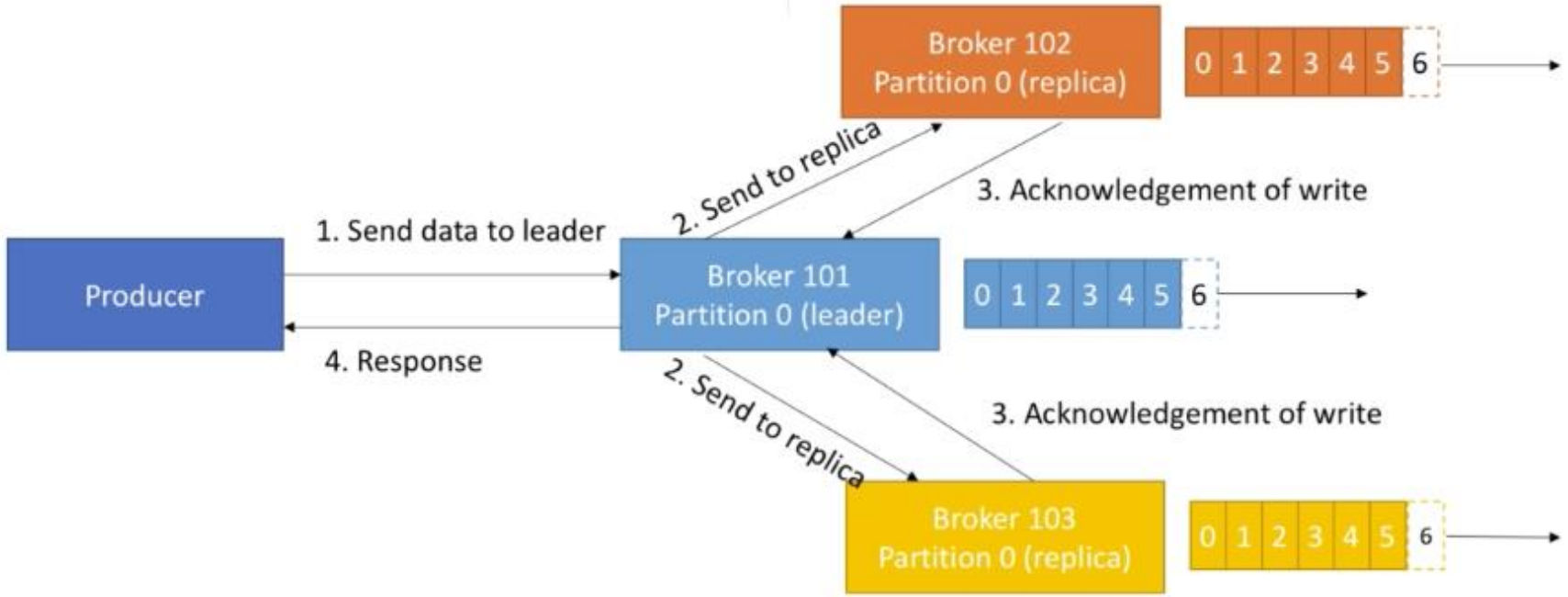
- Useful for cases where data loss is acceptable :
  - Metrics collection
  - Log collection

Producers Acks =1 (leader acks)

- Leader response is requested but replication is not guaranteed (background)
- If the leader brokers goes offline before replication was completed we have data loss.

Producers Acks =2 (replicas acks)

- Leader +Replicas ack Requested



- Added latency and safety
- Necessary if you don't want to loose data

Producers Acks =2 (replicas acks)

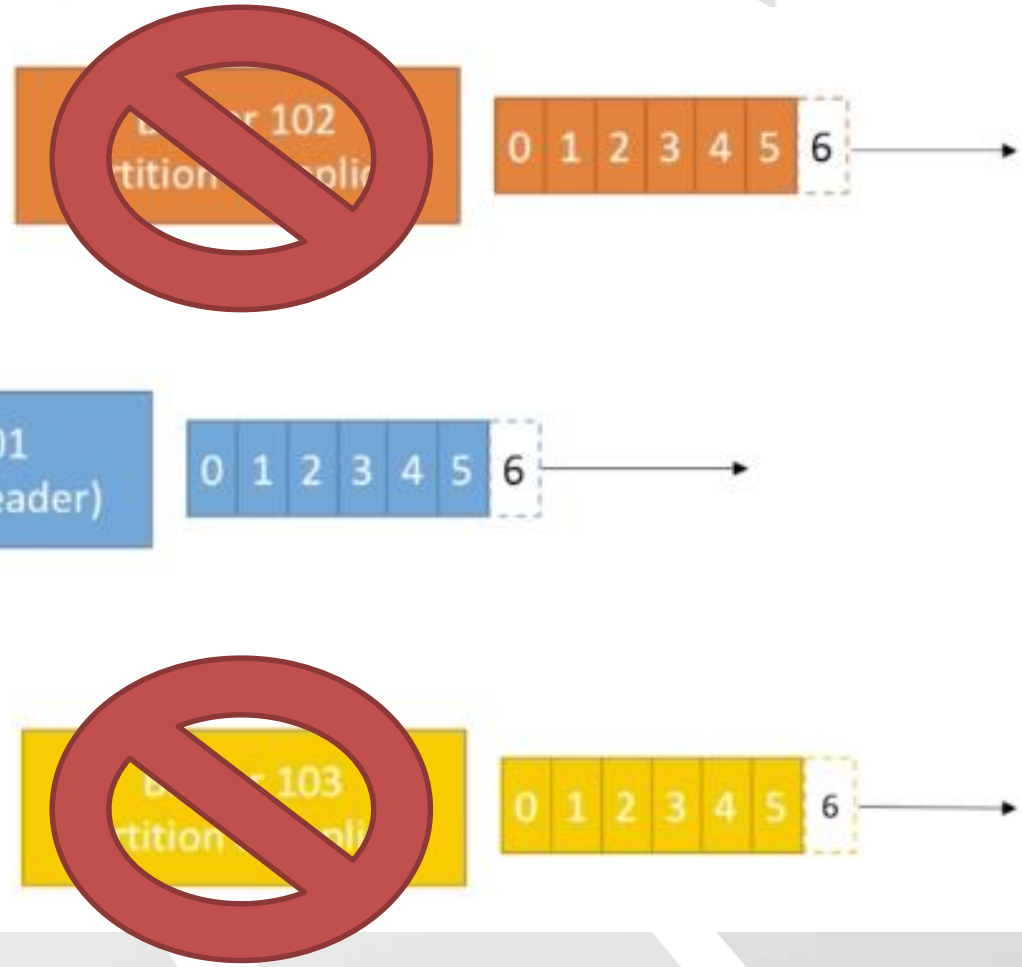
- Acks=all must be used in conjunction with min.insync.replicas=2
- min.insync.replicas=2 can be set at the broker or topic level (override)

- min.insync.replicas=2 implies that at least 2 brokers that are ISR (including the leader) must respond that they have the data.

- That means that if you use replication.factor=3,min.insync.replicas=2 , acks=all, you can only tolerate 1 broker going down, otherwise the producer will receive an Exception on send.

Producers Acks =2 (replicas acks)

- min.insync.replicas=2

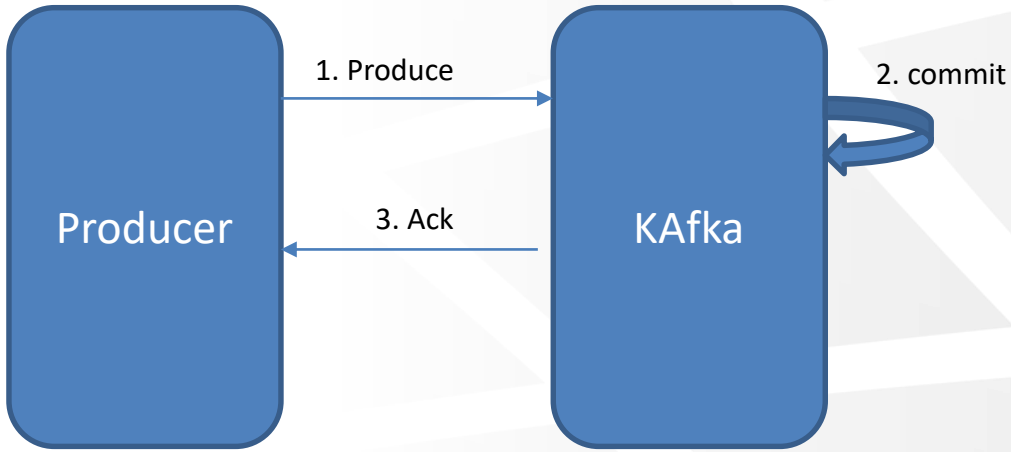


Producer retries

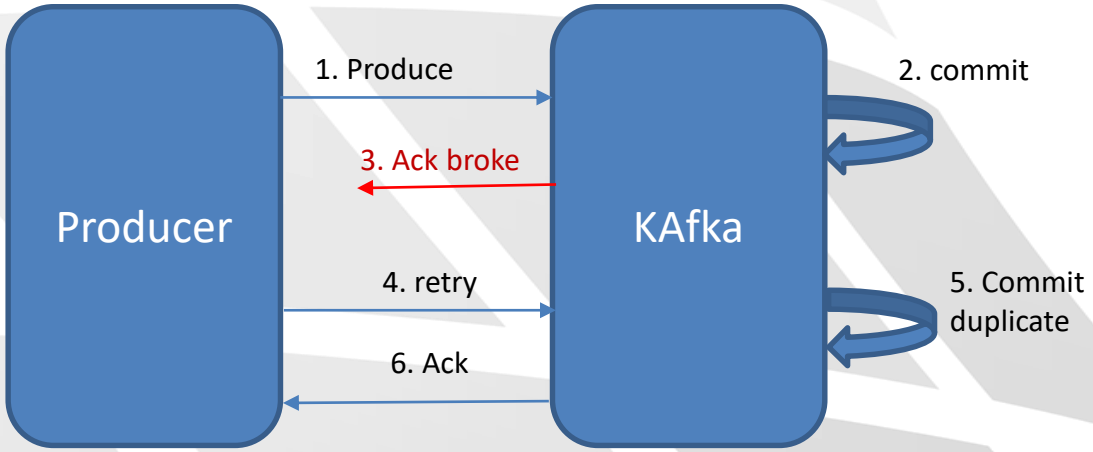
- In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.
- Example of transient exceptions :
  - NotEnoughReplicasException
- There is a "retries" setting:
  - Default is 0
  - Can increase to a high number e.g. Integer.MAX_VALUE
  - In case of retries there is a chance a message will be sent out of order
  - We can control parallelism with max.in.flight.requests.per.connection (default is 5)
    - Set it to 1 ensures ordering.
    - There is a better solution in Kafka > 1.0.0 !

Idempotent Producer – The problem

"Good" Request

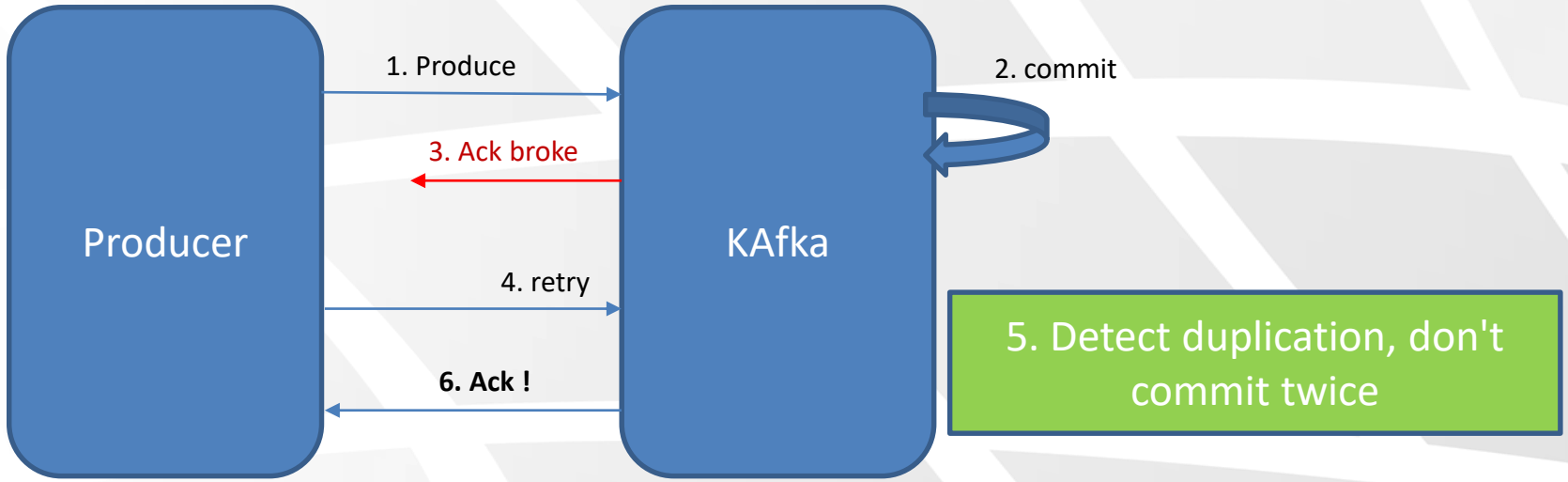


"Duplicate message" Request



Idempotent Producer – Solution (Kafka >=0.11)

- In Kafka >=0.11 , you can define an “idempotent producer” which won’t introduce duplicate on network failure



Idempotent Producer

- Idempotent producer are great to guarantee a safe and stable pipeline
- They must come with :
  - retries = Integer.MAX_VALUE
  - max.in.flight.requests =1 (Kafka < 1.1)
  - max.in.flight.requests =5 (Kafka > 1.1 – higher performance)
  - acks = all
- We need to set
  - **producerProperties.put("enable.idempotence" , true)**

Safe Producer – Summary & Demo

- For Kafka < 0.11
  - acks = all (producer level)
  - min.insync.replicas=2 (topic/broker level)
  - retries = MAX_INT (producer level)
  - max.inflight.requests.per.connection=1 (producer level)
- For Kafka >= 0.11
  - enable.idemopotence=true
  - min.insync.replicas=2
  - Implies:
    - Acks = all , retries= MAX_INT,max.inflight.requests.per.connection=5
    - Order is maintained and performance is high !

Running a safe producer might impact throughput, always test for your case !

Using Safe / Idempotent Producer

Demo

Message Compression

- Producer usually sends data that is text based, for example Json
- In such case It is important to enable compression to the producer.
- This does not effect the consumer and/or the broker
- compression.type can be:
  - none
  - gzip
  - lz4
  - snappy
- The bigger the batch is , the better the compression is .
- Benchmark : https://blog.cloudflare.com/squeezing-the-firehose/

Compression benefits

- Advantage of using compression:
  - Much smaller producer request size (compression ration up to x4 !)
  - Faster to transfer the data over the network
  - Better throughput
  - Better Disk utilization
- Disadvantages:
  - Producers commit some CPU cycles is used for the compression
  - Consumers commit some CPU cycles to decompress data
- Overall :
  - Consider testing for compression against snappy and lz4 for speed/ratio
  - Always use compression in production
  - Consider tweaking linger.ms and batch.size to have bigger batches

linger.ms and batch size

- By default Kafka tries to send messages as soon as possible
  - It will have up to 5 queues in-flight, meaning up to 5 messages are sent concurrently.
  - In the background other messages are already being batched while they wait.
  - This allows kafka to increase throughput with low latency !
  - Batches have higher compression rate.
- linger.ms
  - The number of milliseconds a producer is willing to wait before sending a batch
    - default : 0
    - By providing some lag (e.g linger.ms = 5) ,we increase the chance of batching
    - If batch is full (see batch.size) before the end of linger.ms ,it is being sent immediately.
- batch.size :
  - Maximum number of bytes that will be included in a batch. default is 16KB
  - Increasing to 32K or 64K can improve compression, hence throughput
  - Any message that is bigger than the batch size will not be batched
  - A batch is allocated per partition ! Setting it to a large number may drain your memory !

High Throughput Producer

Demo

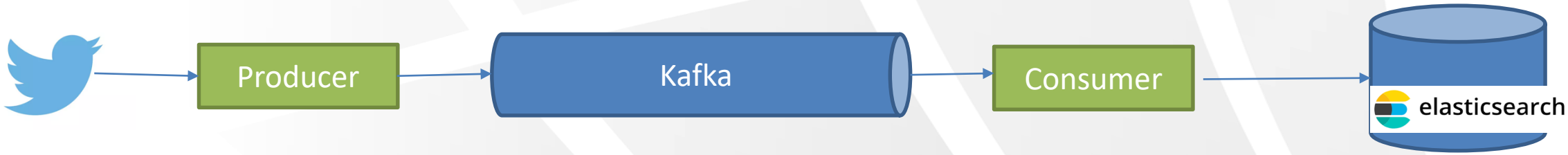
- Adding snappy
- Increase batch size to 32K
- Add delay of linger.ms = 20ms

How Keys are hashed

- By default, your keys are hashed using the “murmur2” algorithm
- It is preferred not to override the behavior of the partitioner, but it is possible to do so (partitioner class)
- The formula is

  targetPartitioner = Utils.abs(Utils.murmur2(record.key()))%numPartitions;
- Same key will always go to the same partition, but adding partition will break keying.

Consumer And Elastic Search



Create a new Elastic Search Cluster (For Dev)

- The leader search engine platform for distributed ,multitenant full text-search.
- Json-based docs with backed schema
- Create a free Elastic Search cluster at :

https://bonsai.io/

- Start with exploring the cluster :

https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started-explore.html

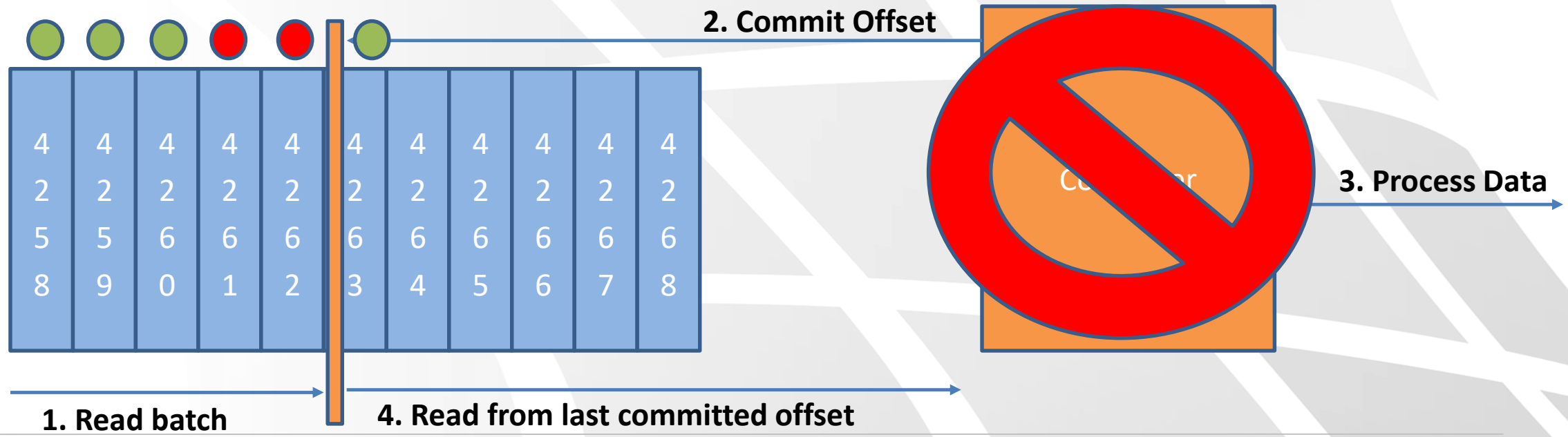
- Create and list Indexes
  - PUT /twitter
  - GET /_cat/indices

Using an ElasticSearch Cluster and adding docs

Demo

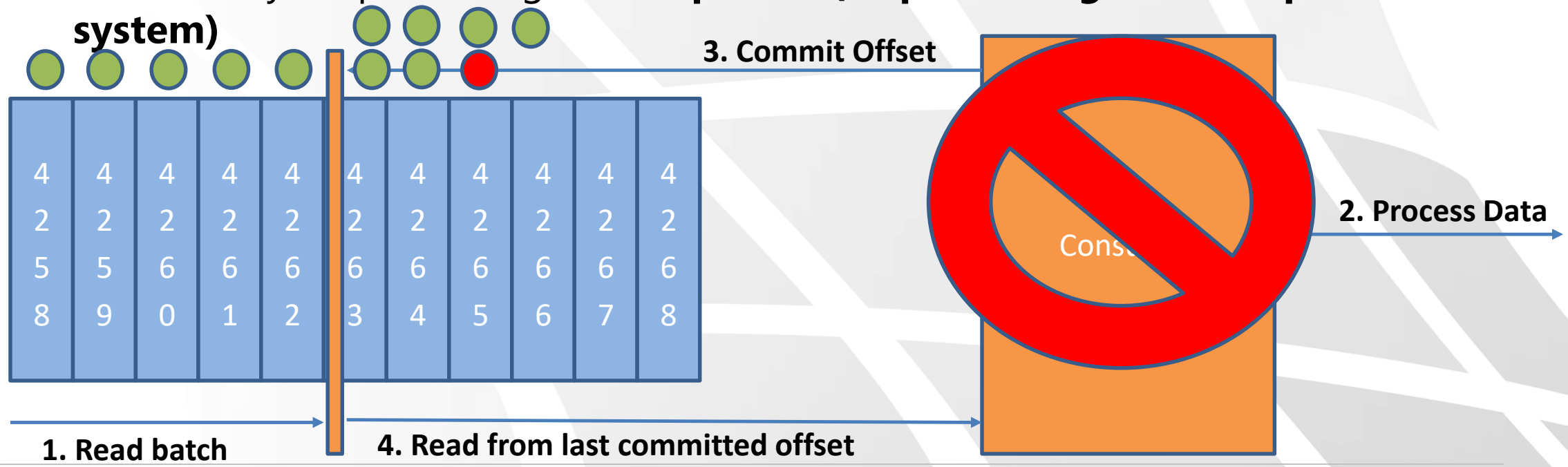
Delivery Semantics for Consumers – At most Once

- At Most Once
  - Offsets are committed as soon as the message batch is received
  - If the processing goes wrong (crashes) the message is lost .



Delivery Semantics- At Least Once

- At Least Once
  - Offsets are committed after the message is processed
  - If the processing goes wrong (crashes) the message will be read again.
  - Make sure your processing is **Idempotent (re-processing won't impact the system)**



Delivery Semantics - Summary

- At most once
  - Offsets are committed as soon as the message is received . Possible data loss.
- At least Once (default)
  - Offset are committed after the message is processed. Something goes wrong same message can be read twice and processed twice. Make sure your processing is **Idempotent** (e.g.: Processing again the message won't impact your system).
- Exactly Once
  - Only works between Kafka to Kafka Workflows using Kafka Streams API

Conclusion : for most application use at-least-once and make sure your processing is idempotent

Make the Consumer Idempotent

Demo

- Generating unique Id's to make producer idempotent

Consumer Poll Behavior

- Kafka Consumers have a “poll” model, while other messaging bus systems has a “push” model
- This allows Kafka consumers to control where in the log they want to consume, how fast, and gives them the ability to reply.
- Fetch.min.bytes (default 1):
  - How much data you want to poll at least on each request
  - Helps increase throughput in the cost of latency
- Max.poll.records (default 500)
  - Controls how many records received per poll request
  - Increase If message are very small and you have lots of RAM
  - Good to monitor how many records are pulled per request.

Consumer Poll Behavior

- Max.partitinos.fetch.bytes(default 1 MB):
  - Maximum data returned by the broker per partition
  - If you read from 100 partitions you will need a lot of memory
- Fetch.max.bytes(default 50MB):
  - Maximum data returned for each fetch request (covers multiple partitions)
  - The consumer performs multiple fetches in parallel

Consumer Offset Commit Strategies

- There are two most common patterns for committing offsets in a consumer application
- (easy) enable.auto.commit = true + synchronous processing of bat

```
while(true){
    List<Records> batch = consumer.poll(Duration.ofMillis(100))
    doSomethingSynchronous(batch)
}
```

  - With auto-commit, offsets will be committed automatically for you at regular interval (auto.commit.interval.ms = 5000 by default) whenever you call **.poll**
  - If you don't use synchronous processing you will be in "at-most-once" behavior

Consumer Offset Commit Strategies – Cont.

- enable.auto.commit = false + synchronous processing of batches

```
while(true){
    batch += consumer.poll(Duration.ofMillis(100))
    if isReady(batch) {
        doSomethingSynchronous(batch)
        consumer.commitSync();
    }
}
```

- You control when you commit offsets and what's the condition for committing them
- Example : accumulating records to a buffer and then flushing the buffer to a database + commiting offset then

Commit strategies + max poll size

Demo

- Controlling the commit strategy by allowing manual commit
- Configuring max poll size
- Use Elastic Search Batches

Consumer Offset Reset Behavior

- When our app has a bug and cannot read from kafka since the consumer are down , we get by default 7 days of retention
- The behavior for the consumer then should be :
  - auto.offset.reset = latest - will read from the end of the log
  - auto.offset.reset = earliest - will read from the start of the log
  - auto.offset.reset = none - will throw exception if no offset is found
- Consumer offsets can be lost :
  - If a consumer hasn't read new data in 1 day (kafka < 2.0)
  - If a consumer hasn't read new data in 7 day (kafka >= 2.0)
  - This can be controlled by the broker settings **offset.retention.minutes**

Querying Consumer Groups offsets

- We can query what is the current offset of each consumer group:

```
$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --group grp --describe
```

- We can reset a consumer group :

```
$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --group grp --reset-offsets
--execute --to-earliest --topic tweets
```

Replaying data for Consumers

- To reply data for a consumer group :
  - Take all the consumer from a specific group down
  - Use ‘Kafka-Consume-groups’ command to set the offset
  - Restart Consumers

Summary :

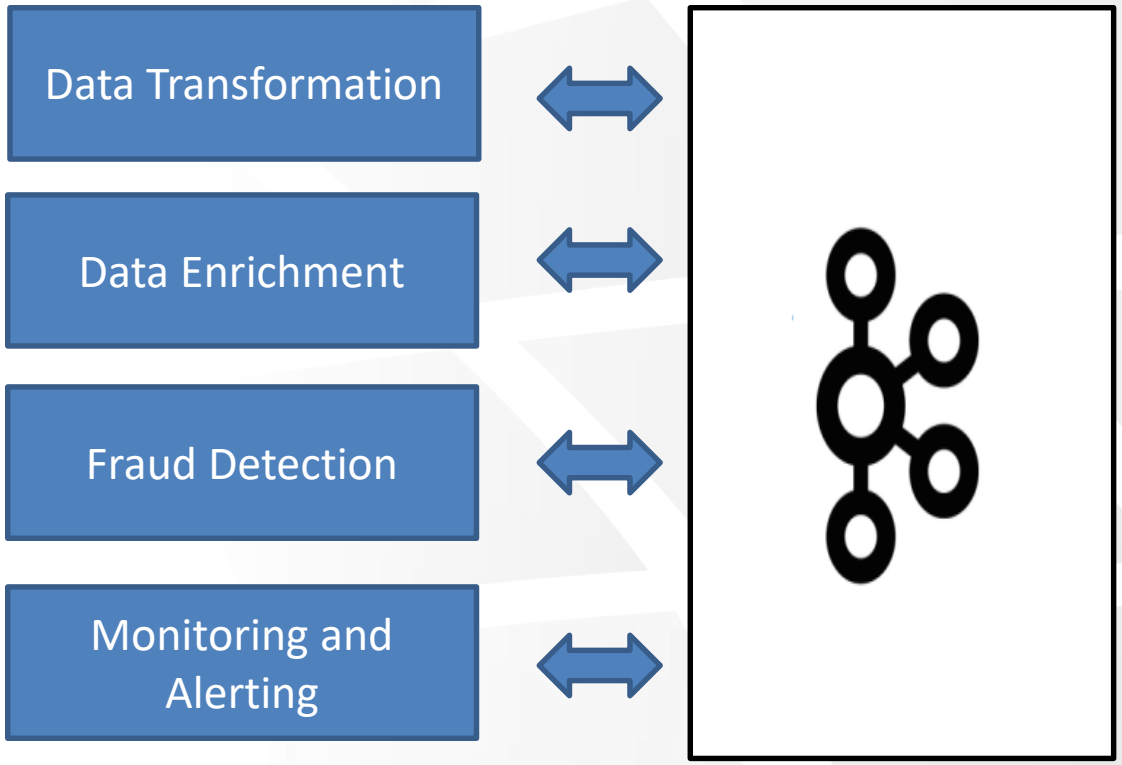
- Set proper data.retention.period and offset.retention.period
- Make sure the auto.reset.behavior is what you want (auto/manual)
- Use reply capability in case of unexpected behavior

Questions

Module 06: Kafka Streams (Intro)

What is Kafka Streams

- **A data processing and transformation library** within Kafka

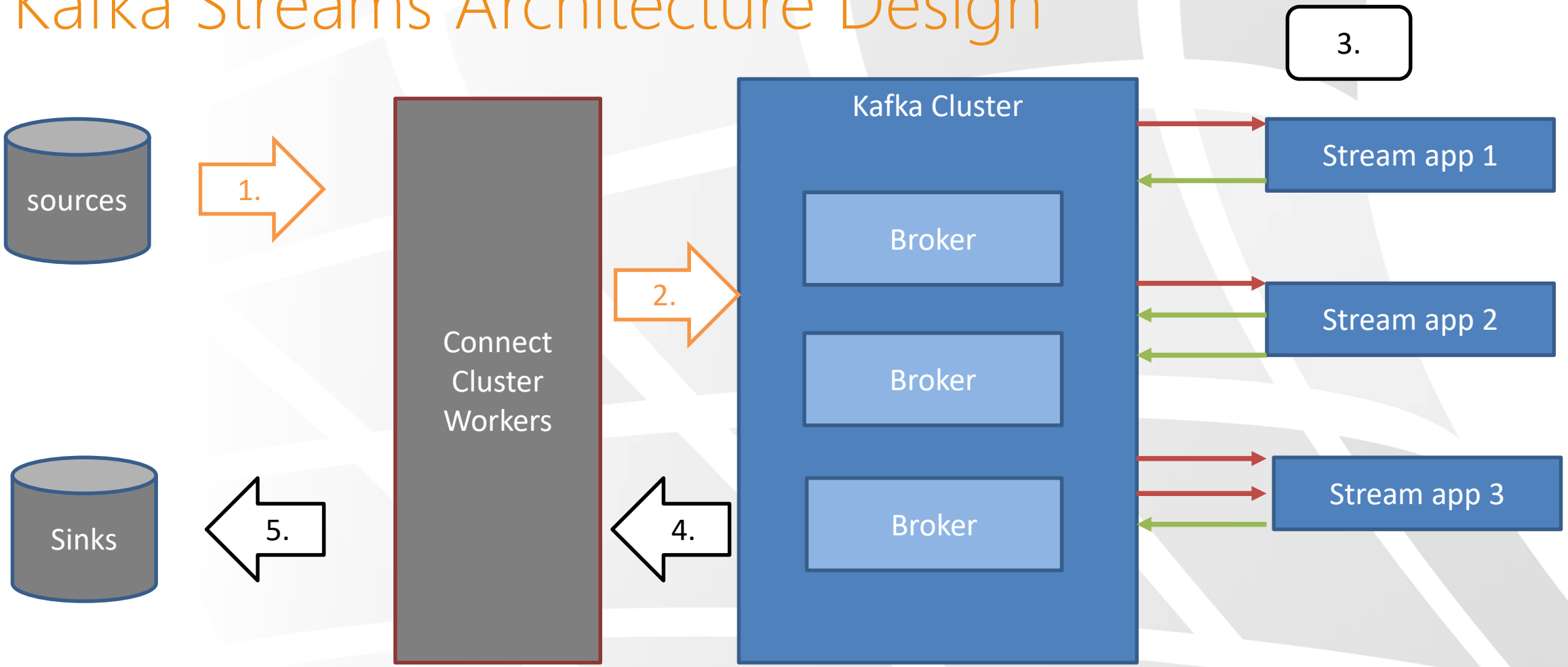


- Standard Java Application
- No need to create separate cluster
- Highly scalable, elastic and fault-tolerant
- Exactly Once capabilities

What is stream processing ?

- Stream processing is really all about **Transformation on a** continuous stream of data
- Transformation are in forms of filters, maps, joins and aggregations
- We can divide stream processing into two categories:
  - **Real time Map reduce :**
    Storm , Spark, Flink
    - They ran on dedicated clusters
    - Long running analytics, machine learning ,etc.
  - **Even Driven micro-services :**
    - The streaming platform is the central nervous system
    - Micro-services units acts as stream processing units
    - Input and output are always streams
    - Kafka Streams, Akka-Streams

Kafka Streams Architecture Design



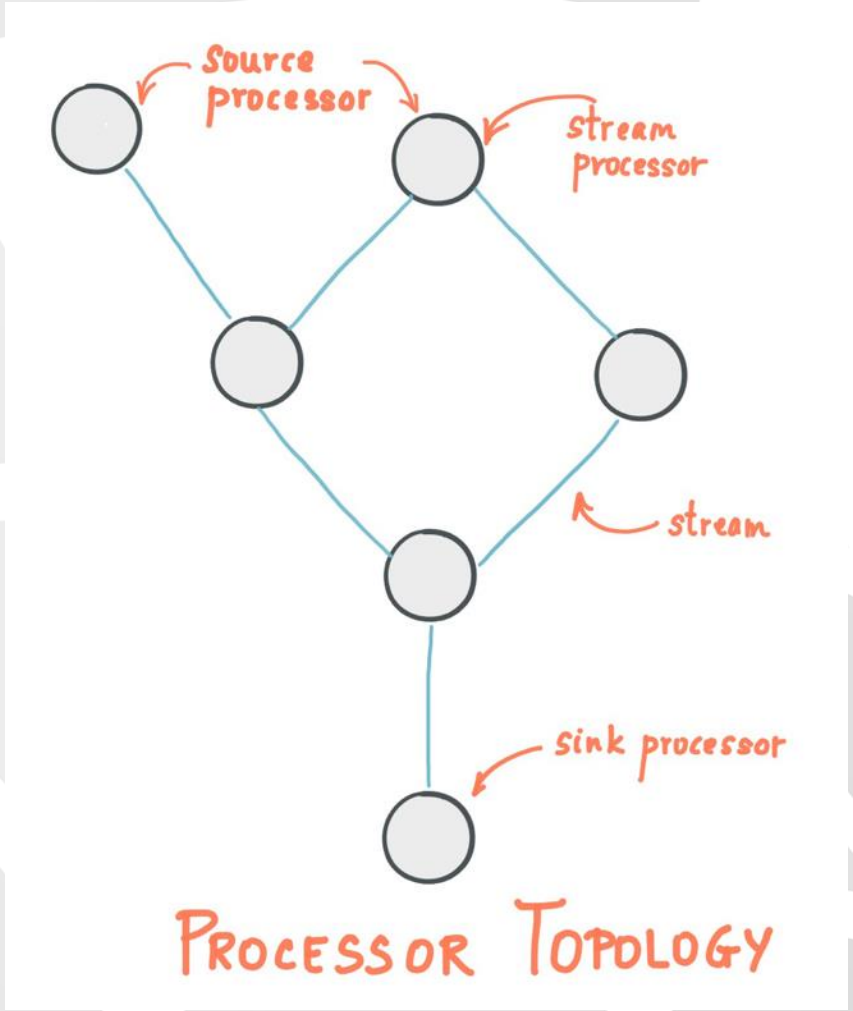
Example usage

- We want to detect a fraud of credit cards which are detected by using the same Credit –card number **in a window of 5 minutes** within 2 different IP addresses

- We want to filter all tweets according to a key word or a topic and put the result back to kafka

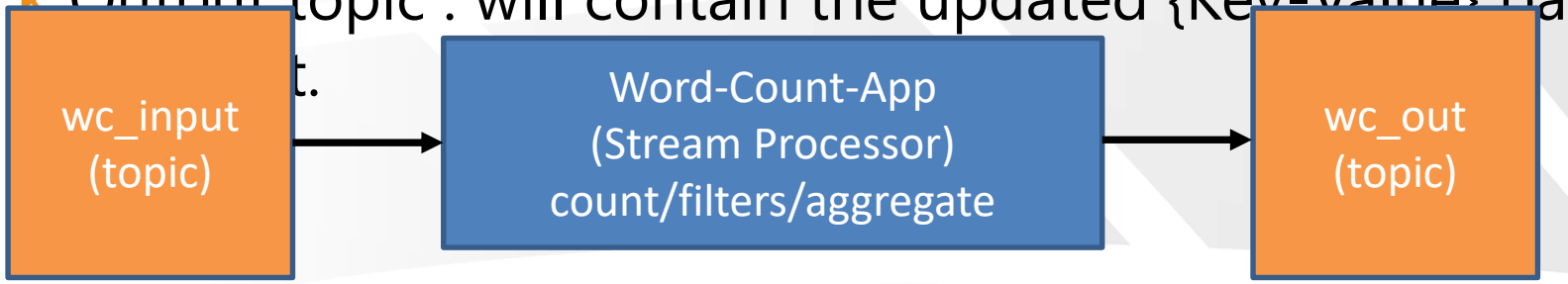
Stream Topology

- A stream processor is the **node** in the topology, it represent the processing steps tot transform data
- Arch represent the **stream**
- **Source Processor –** sends the data from one or more kafka topics.
- **Sink Processor –** does not have any more downstream processors.



Building a word-count Pipeline

- We are going to build a simple pipeline of Kstreams that counts words
- Our java app will act as the Stream Processor and will interact with :
  - Input Topic : will contains sentances that will be break down to words
  - Output topic : will contain the updated {Key-Value} pair of each word and t.

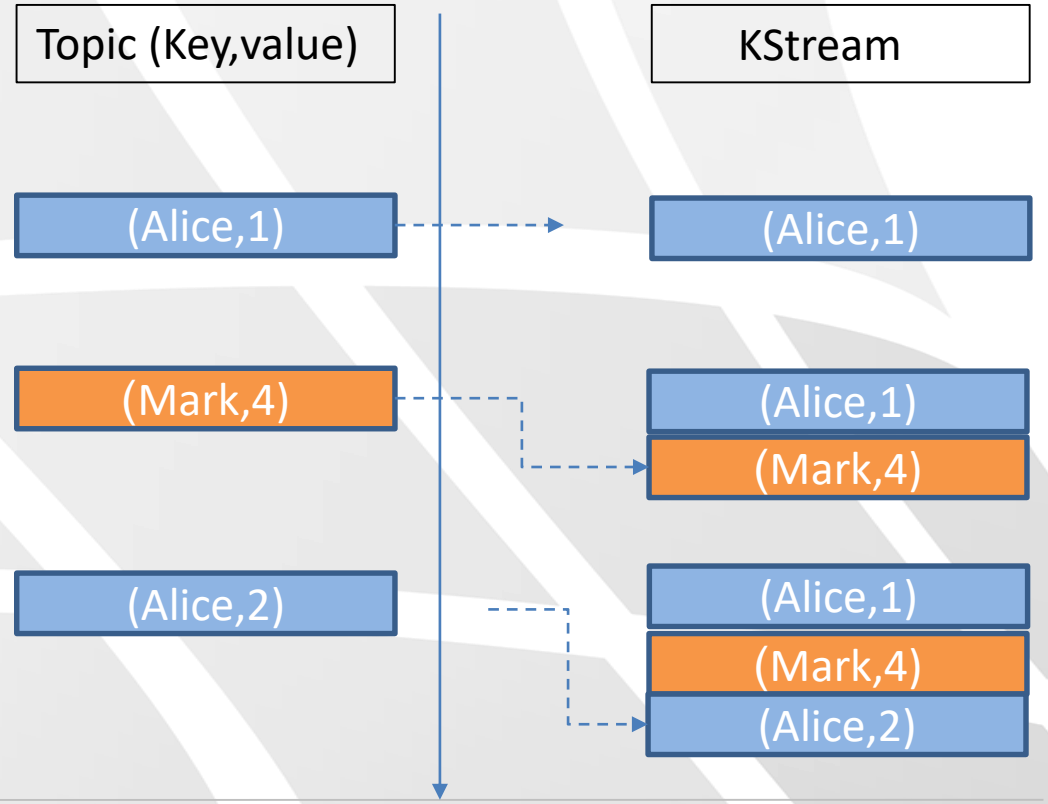


Building a Word-Count Kafka Stream Pipeline

Demo

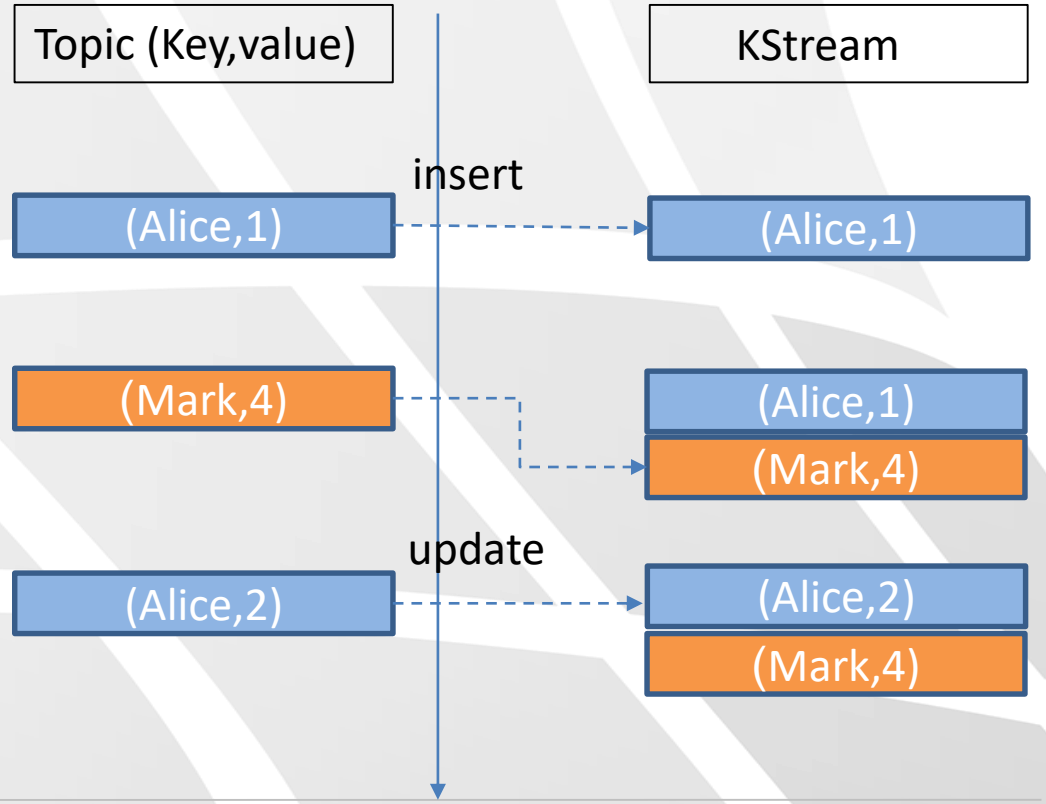
Kstream

- An abstraction of data pipeline from Topics that supports :
  - All **inserts**
  - Similar to a log
  - Infinite
  - Unbounded data streams

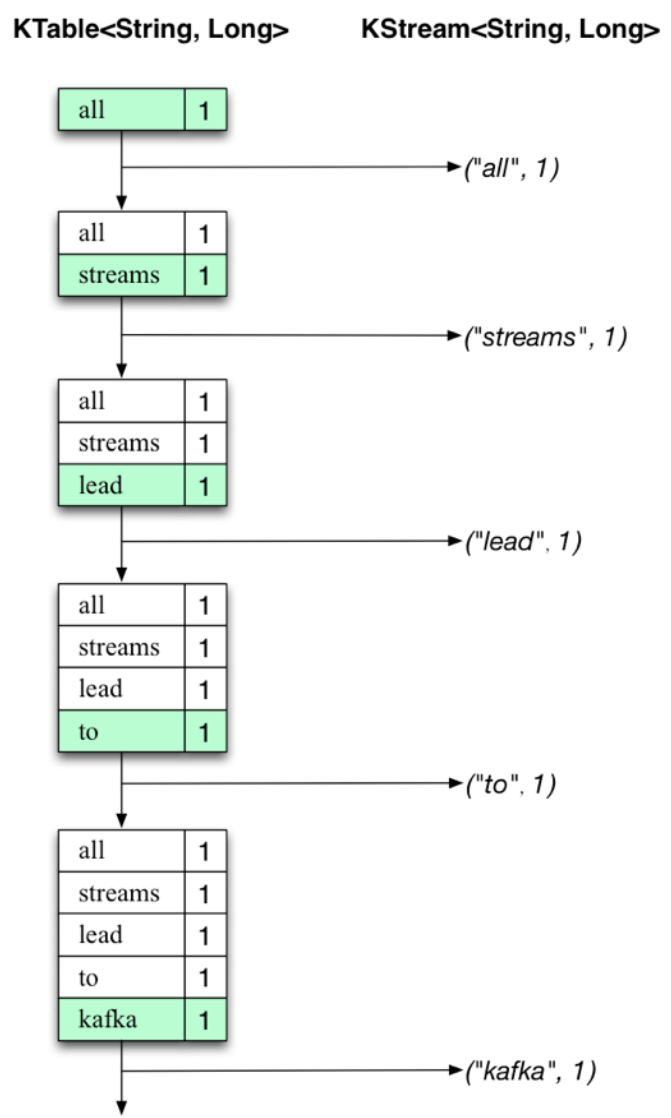


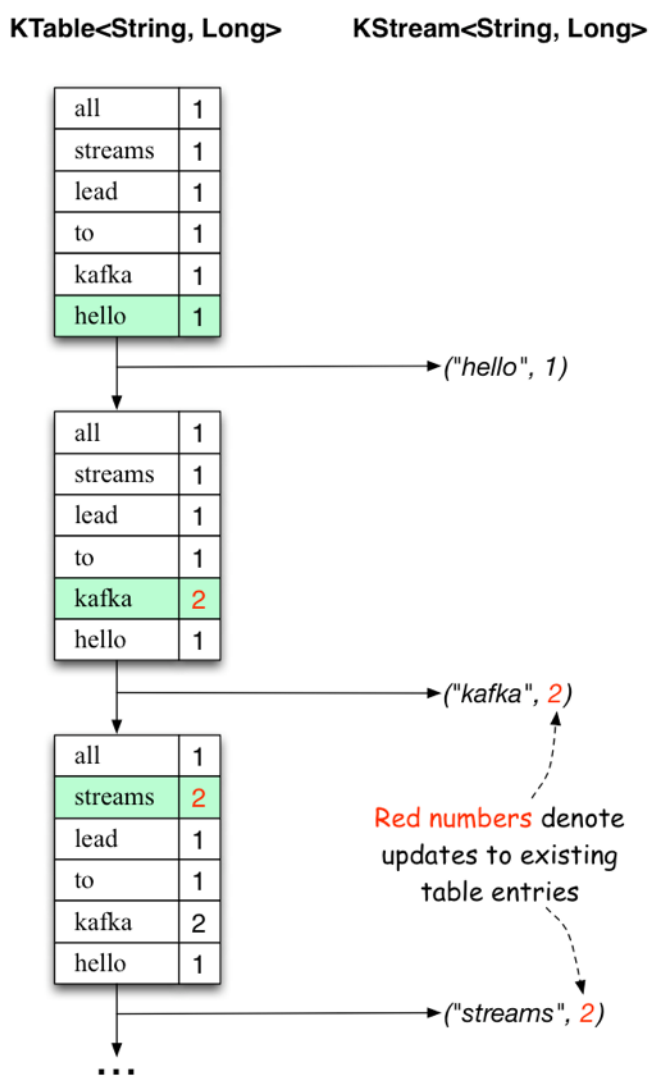
KTable

- All **Upserts** on non-null values
- Deletes on null value
- Similar to a a table
- Parallel with log compact topics



Demo Explained



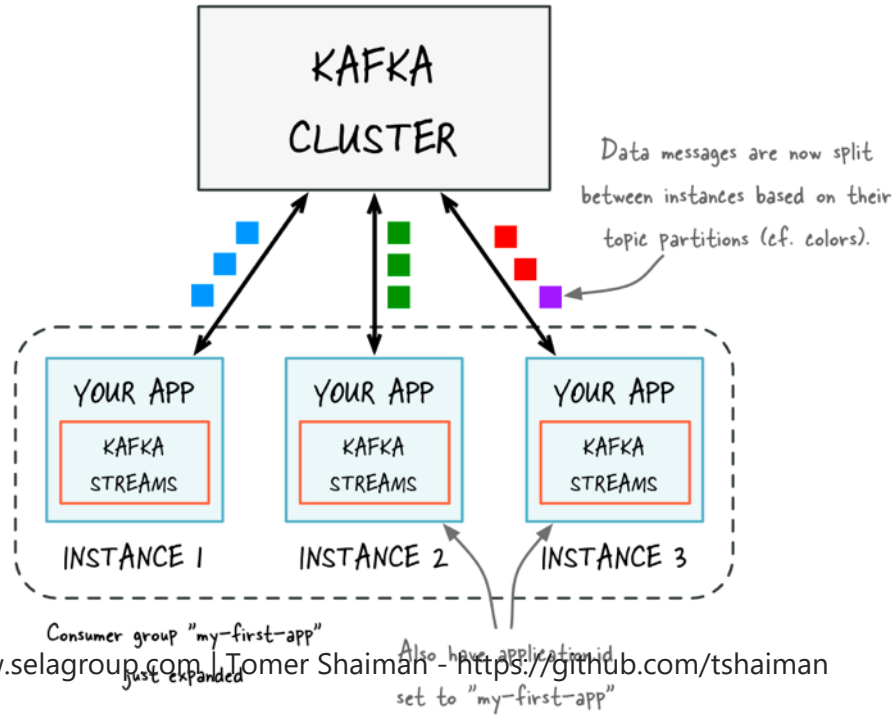


Word Count Internal Topics

- Kafka Streams application will eventually create internal topics:
  - Repartition Topics – In case you transform your key in the stream, a repartition will happen
  - Changelog Topics- In case you create aggregation, Kafka streams will save compact data in these topics
- Internal Topics
  - are managed by Kafka Streams
  - are prefix by application.id parameter
  - **You should never delete, altered or publish to ,or change your app.id**

Scaling our application

- Since our topic has 2 partitions, we can launch up to 2 instances of the same app in parallel
- This is due to the fact that Kafka Streams application relies on KafkaConsumer which allows us to add more consumers to consumer group.
- Conclusion **: Scaling in Kafka Stream application does not require cluster**



Running WordCount pipeline with filters

Lab

Module 07: Kafka Eco-System and Architecture

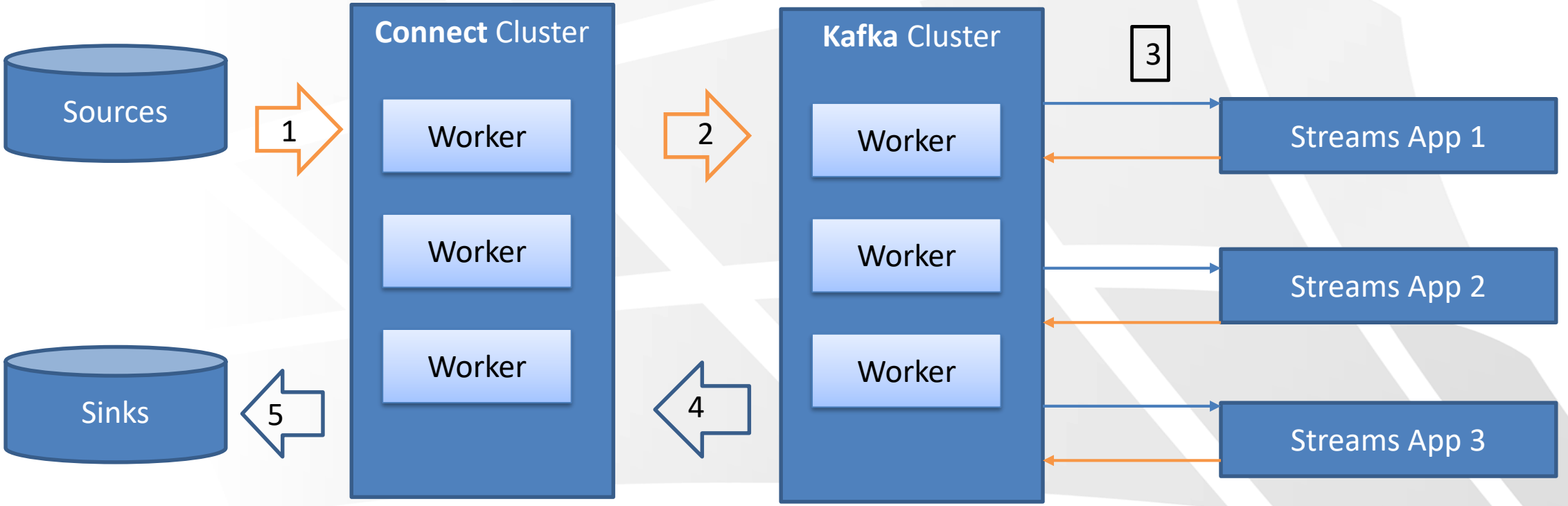
Agenda

- Kafka Connect
- Kafka Schema Registry
- Architecture case Study
- Admins – Cluster Set-up
- Admins – Monitoring & Operations
- Start Kafka Differently

Kafka Connect

- Simplify and improve getting data in and out of kafka.
- You're not the only one who
  read data from a DB/Stream and needs to send it to Kafka
- You're not the only one who needs
  to send data to Db/stream(Elasticsearch/
  MongoDb/Sql/ PostgreSQL)
- Its hard to achieve fault -Tolerance,
  Idempotence, Distribution ordering

Kafka Connect and Streams Architecture Design



Scenario: Output Kafka Topic to a File (Kafka-Connect)

- It is common to output the content of a topic into a file
- Kafka Connect is a framework that provides scalable and reliable streaming of data to and from Apache Kafka.
- We will use a “File Sink” connector with configuration file as out “Sink”

```
#my-file-sink.properties config file
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=/tmp/my-file-sink.txt
topics=my-connect-test
```

Kafka Connect –Cont. : Workers

- Processors that execute Kafka Connect connectors are called Workers

```
#bootstrap kafka servers
bootstrap.servers=localhost:9092

# specify input data format
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter

# The internal converter used for offsets, most will always want to use the built-in default
internal.key.converter=org.apache.kafka.connect.json.JsonConverter
internal.value.converter=org.apache.kafka.connect.json.JsonConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false

# local file storing offsets and config data
offset.storage.file.filename=/tmp/connect.offsets
```

Kafka-Connect – Running Workers

```
$ connect-standalone.sh my-standalone.properties my-file-sink.properties
```

- Launches the Kafka Connect Job On the sink properties file
- Worker manages its own offset into a file system without effecting and of the offset for other Consumer Groups
- Can be done for both ways : Producing a Data from File into Kafka Topic
- Extremely powerful for Porting /Testing / Automation tasks !

Kafka Connect – Topic To File Sink

Demo

The need for Schema registry

- Kafka takes bytes as input and publishes them
- No data verification !
  - What if the producer produces bad data ?
  - What if a field gets rename ?
  - What if the format changes over time ?

The Consumer might break !

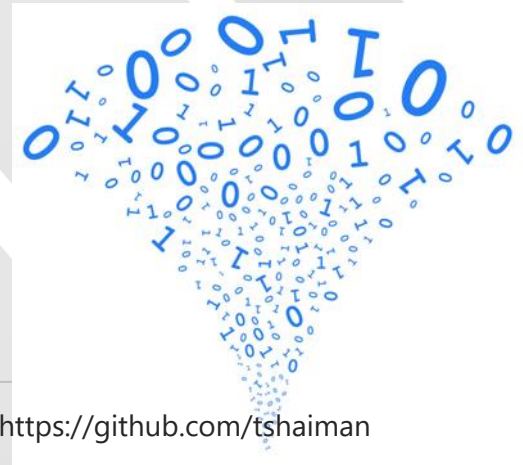
- We need the data to be self – describable
- We need to evolve data without breaking downstream consumers
- We Need Schema + Schema registry

The need for Schema Registry

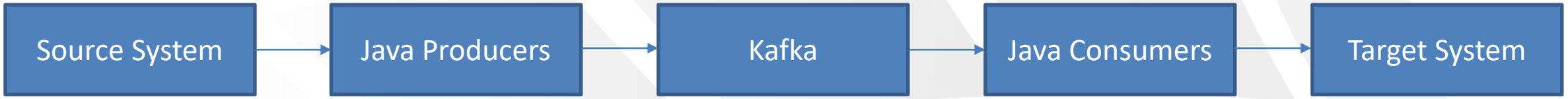
- What if the kafka brokers were verifying the messages ?

- It would break what Kafka so good at :
  - Kafka doesn't parse or even read your data (no CPU usage)
  - Kafka takes bytes as input without loading them to memory
  - Kafka doesn't know if you wrote string, integer ,Boolean and does not parse the data.

- The Schema registry has to be separate component !
- Producers and Consumers needs to talk to it
- A commod data format must be agreed upon

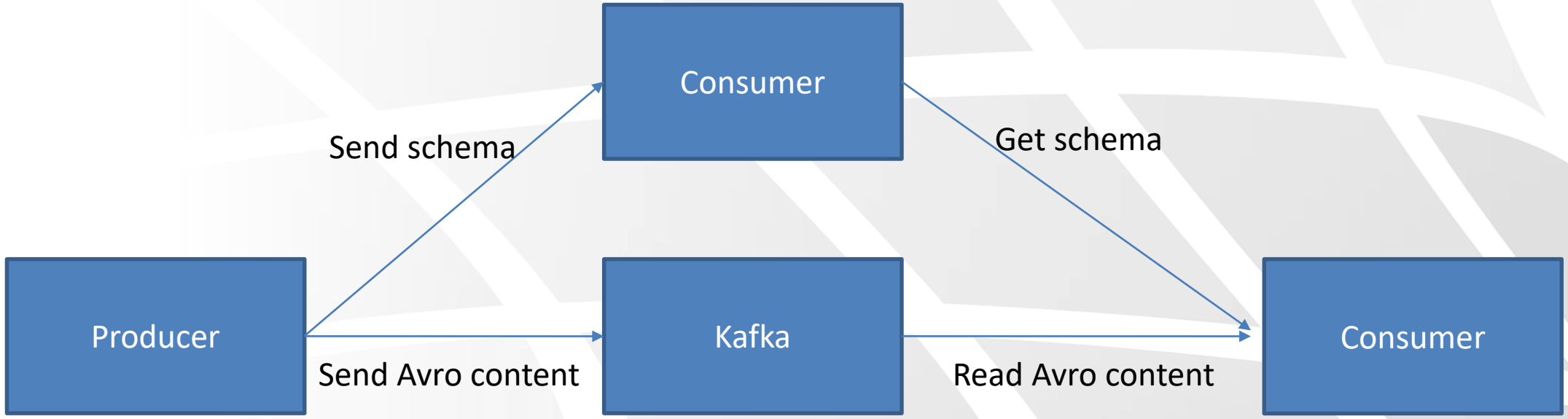


Pipeline without Schema Registry



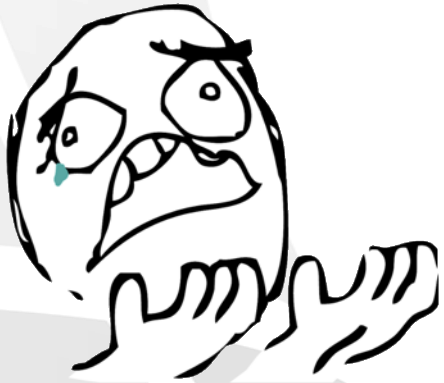
Confluent Schema Registry Purpose

- Store and retrieve schemas for Producers / Consumers
- Enforce Backward / Forward /Full compatibility on topics
- Decrease the size of the payload of data sent to Kafka



Schema Registry : gotchas

- There are many benefits for using Schema registry
- But it implies :
  - Set it up well
  - Make it highly available !
  - Partially change the producer and consumer code
- Apache **avro** is great but has a learning curve

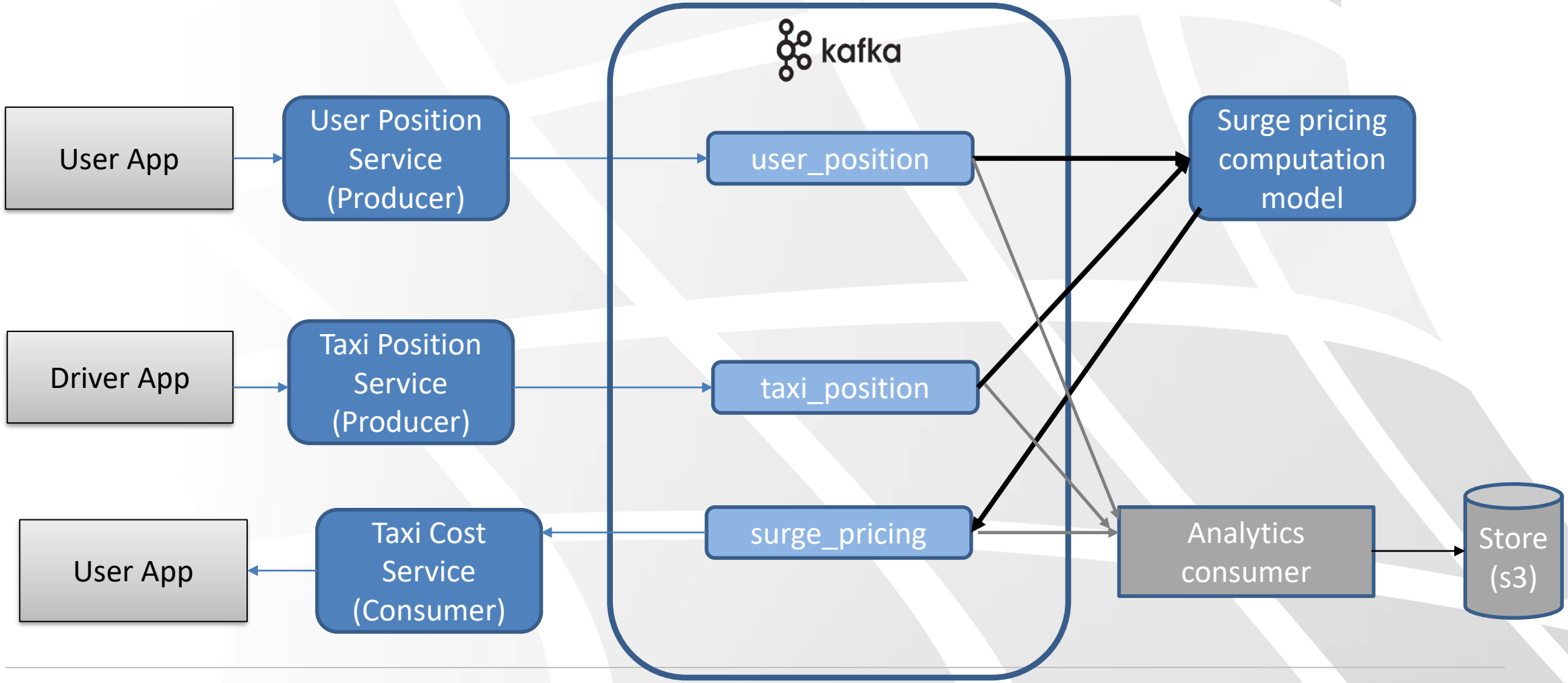


- The Schema registry is free and open sourced ,created by Confluent

Case Study – GetTaxy

- GetTaxy is a (fake) company that matches customers with taxi drivers on demand
- The Business wants the following:
  - Users should match with the closest driver
  - The pricing should “surge” if the numbers of drivers are low/high
  - All the position data before and during the ride should be stored in an analytics store so that the cost is computed correctly
  - Take few minutes to come up with solution

Case Study – GetTaxy



Case Study – GetTaxy

- taxi_position, user_position topics :
  - Can have multiple producers
  - Should be highly distributed if high volume, topics > 30
  - Key : could be "user_id" / "taxi_id"
  - Data should not be stored for a long time

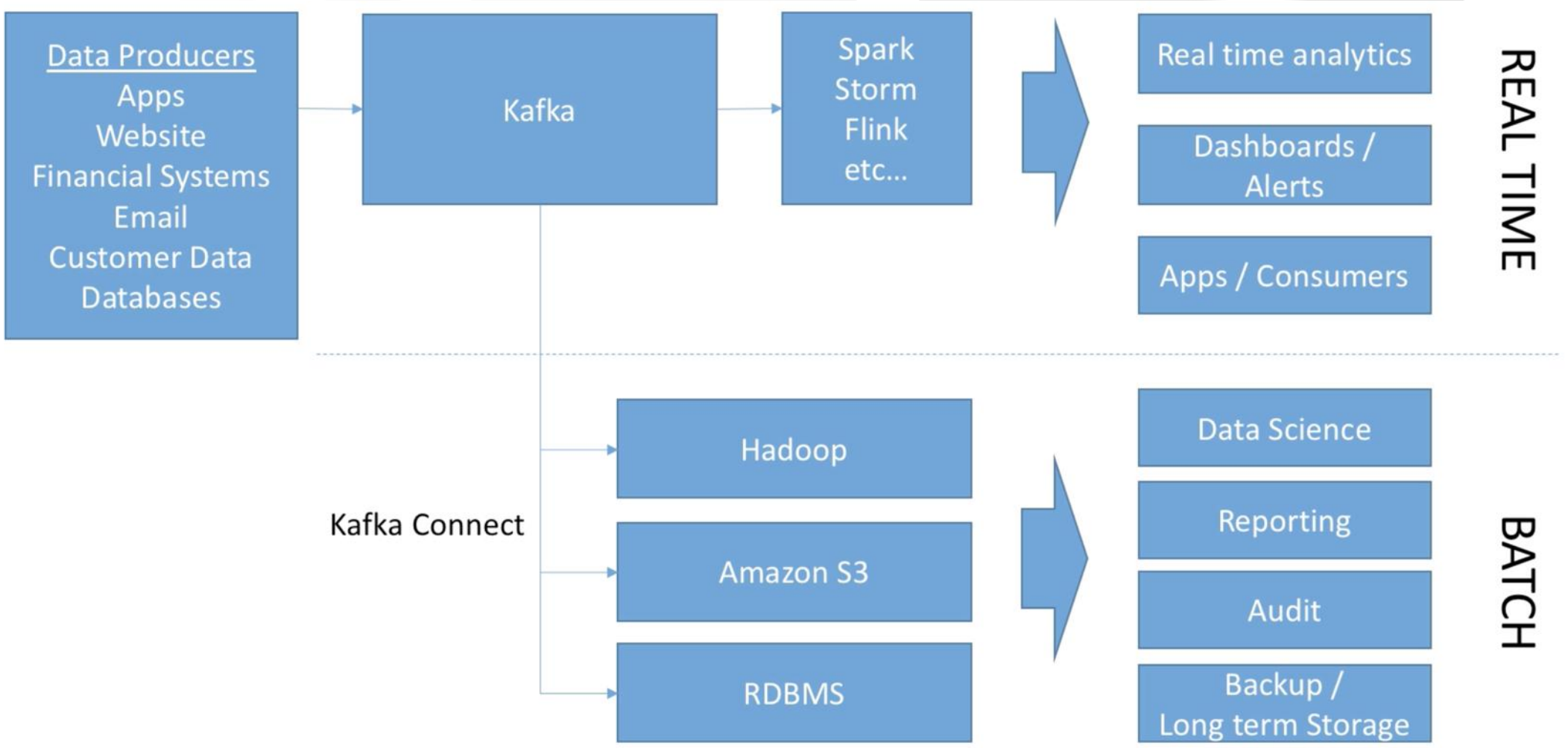
- surge_pricing topic :
  - The computation of it comes from Kafka streams app
  - Surge pricing may be regional therefore the topic has high vo
  - Maybe use "weather" or "events" for the Kstream App !

Case Study – Big Data Ingestion

- Sometimes it is desired to have a “batch” pipeline for big-data analysis that occurs once in a while (machine learning/ analytics). This pipeline is considered “slow” (S3, HDFS, ElasticSearch etc)
- On the other hand , kafka can serve as a “near-real-time” /”fast” layer for processing large amount of data and react immediately (e.g : fraud detection, social media, brand-safety, ad-tech)

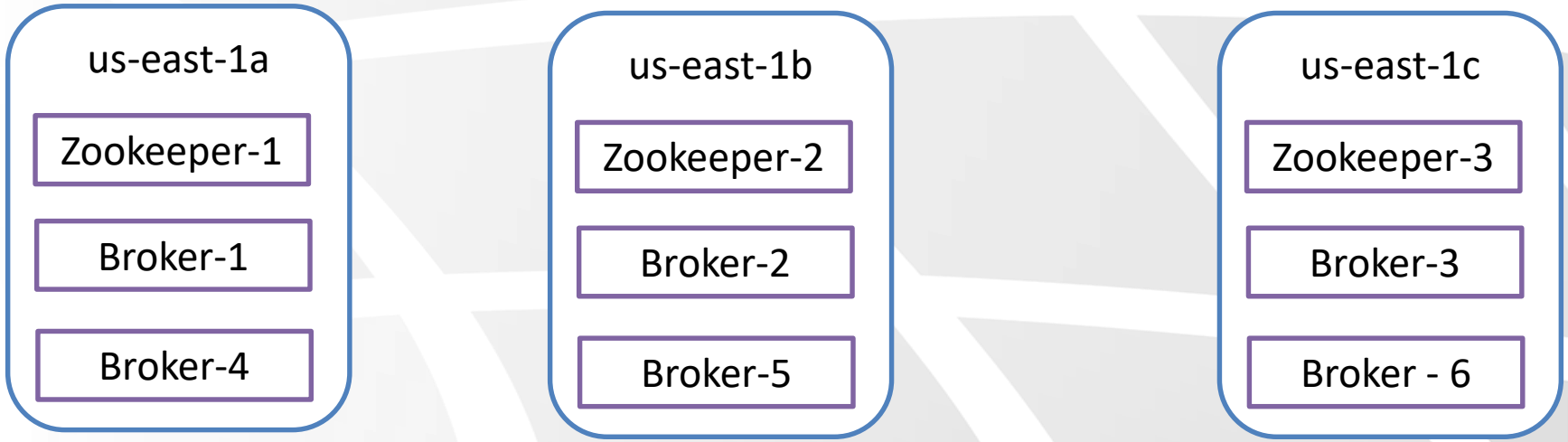
- ”Kafka-As-A-Front” to Big-Data ingestion is a common pattern, in such case kafka is like buffer before the store.

Big Data Ingestion



Kafka Cluster Setup – High Level Architecture

- You want multiple brokers in different data centers to distribute your load.
- You want a cluster of at least 3 zookeeper
- In AWS :



Kafka Cluster

- Its not easy to setup an HA cluster
- You want to isolate the Zookeeper & Brokers on separate servers
- Monitoring needs to be implemented
- Operations have to be mastered
- You need a really good Kafka Admin !

- Alternative : Kafka-As-A-Service (on the web / Azure Kafka Service)

Kafka Monitoring & Operations

- Kafka Exposes Metrics via JMX
- These metrics are highly important for monitoring kafka, and ensures the system behaves correctly under load
- Common Places to host the Kafka Metrics :
  - ELK (ElasticSearch + Kibana)
  - Datadog
  - NewRelic
  - Promotheous
  - Confluent Control Center
  - Influx Db

Most Important Metrics

- Under Replicated Partitions : Number of Partitions that have problems with the ISR (In-Sync-Replica). May indicate a high load on the system
- Request Handlers : Utilization of threads for I/O, network,etc..overall utilization of Apache Kafka broker.
- Request timing : how long it takes to reply to requests. Lower is better
- Lag : the Lag offset of consumer group
- JVM Metrics : CPU / Memory /etc

Monitoring Resources

- Excellent blog on how to configure InfluxDB to work with Kafka:

https://softwaremill.com/monitoring-apache-kafka-with-influxdb-grafana/

- Confluent Docs :

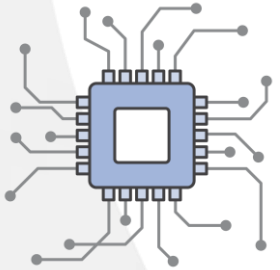
https://docs.confluent.io/current/kafka/monitoring.html

- Kafka Docs :

https://kafka.apache.org/documentation/#monitoring

Operations

- Kafka Operations team must be able to perform the following tasks:
  - Rolling Restart of brokers
  - Updating Configurations
  - Rebalancing partitions
  - Increase replication factor
  - Adding a broker
  - Replace a broker
  - Removing a broker
  - Upgrading kafka cluster with zero downtime

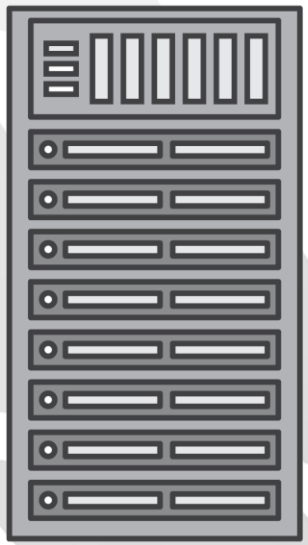


Start Kafka Differently

- Confluent CLI Tools :

https://github.com/confluentinc/confluent-cli

- Create a multi node cluster
  - Clone the server.properties to be server1.properties
  - Change ports
  - Change log directory
  - replicate data directory
  - start 3 Instances of kafka-server-start.sh
- Using Docker-Compose command with the provided file:
  - **$ docker-compose up**



Wrap Up

- Stream processing is Huge topic and we have covered many concepts
- You should try and build a simple pipeline and advance gradually
- Sys Admin and Security were not covered but are key factor of kafka cluster
- Think of How Kafka Connect can Assist you in your ETL process
- Kafka Streams can leverage your transformation and aggregation tasks from the Batch Processing into real-time streaming apps !
- Use Schema registry to protect your data and enable evolution and versioning !

We Are here for you...

- Our Architects And Consultants has vast experience in :
  - Demolish Monoliths
  - Building Serverless Architecture on AWS
  - PaaS and IaaS project on the cloud
  - Reactive systems on open source technologies with variety of tech-stack and languages
  - Streaming processing and Big Data Projects
  - Docker Kubernetes and Service Mesh (Istio/OpenShift)

Questions