
confluent


APACHE
kafka®
A distributed streaming platform

**Apache Kafka® is a distributed streaming platform.**

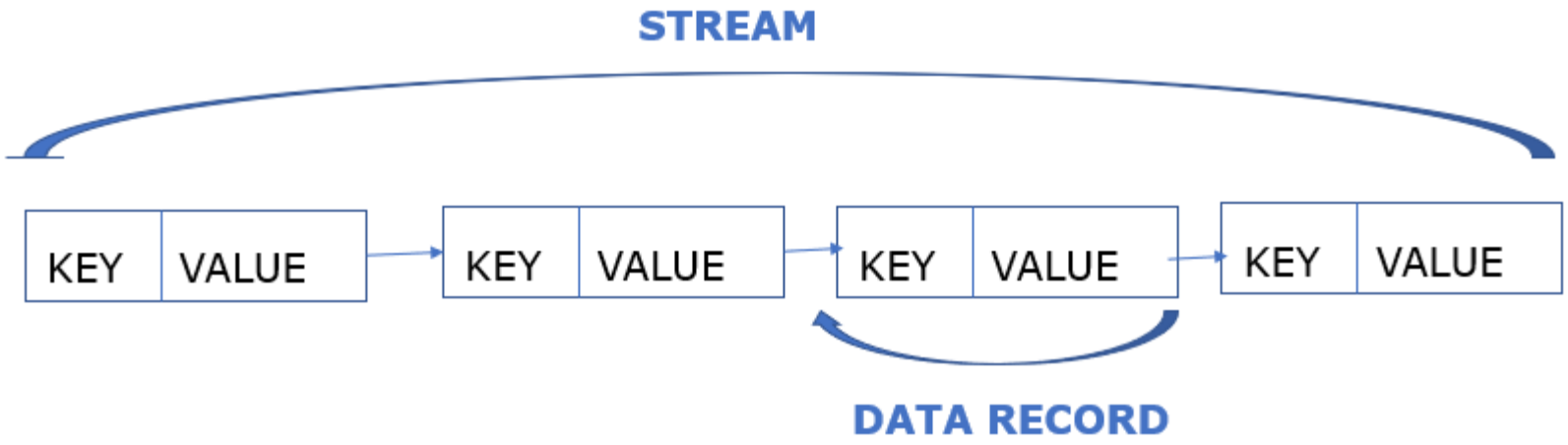
- **Distributed**
- **partitioned,**
- **replicated commit log service.**

**It provides the functionality of a messaging system, but with a unique design.**

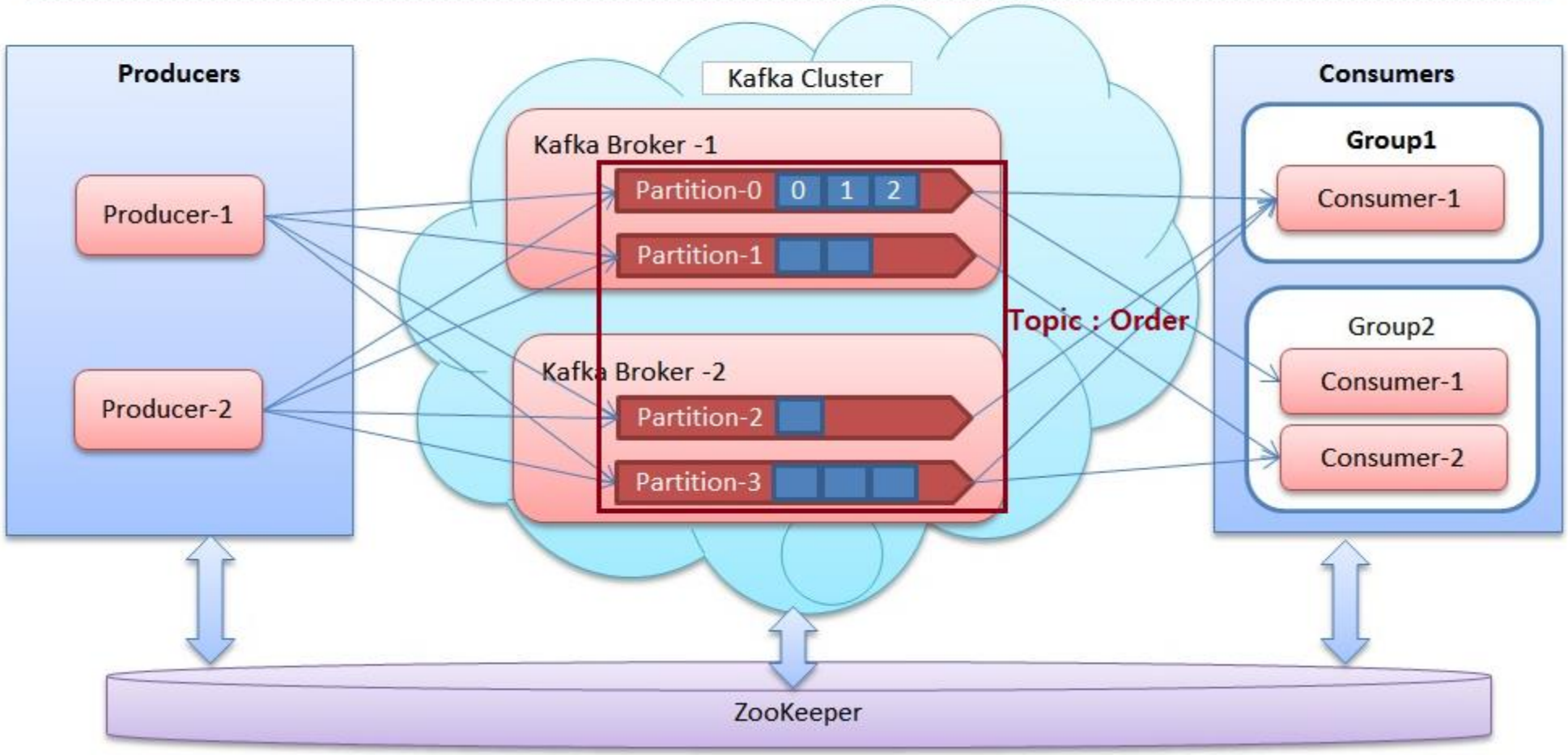
**A stream is :**

**An Ordered,**
**Replayable,**
**And fault-tolerant sequence of immutable data records.**

**Where a data record is defined as a key-value pair.**

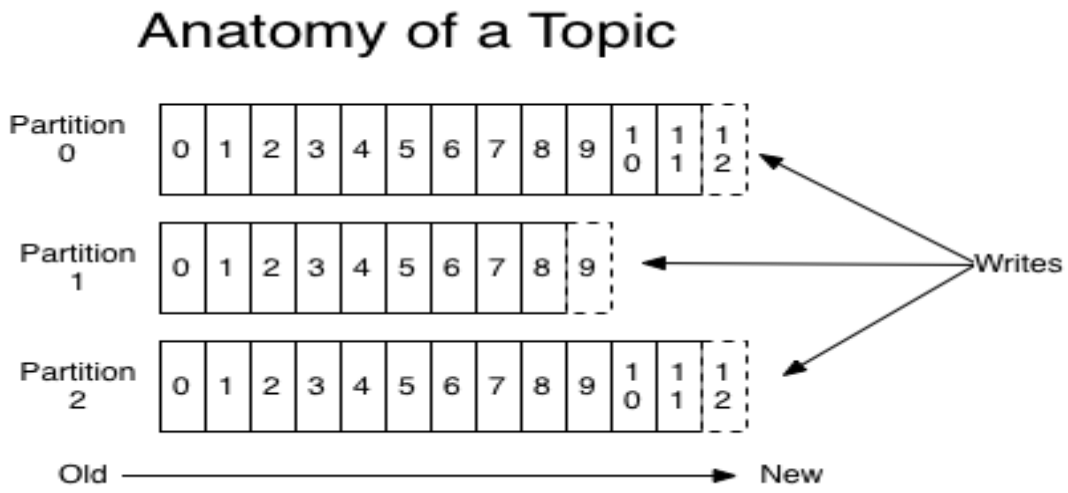


In this diagram, two brokers are shown in a kafka cluster. Two producers are publishing messages to one topic with 4 partitions. Each broker have two partitions. Tow consumer groups are configured, group1 has one consumer and group2 has two consumers. For each consumer group, messages are guaranteed to be consumed at least once.



## Topics and Logs

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



**Kafka Use cases :**

- ✓ **Website Activity Tracking**
- ✓ **Metrics**
- ✓ **Log Aggregation**
- ✓ **Stream Processing**
- ✓ **Event Sourcing**

# Use Cases

Here is a description of a few of the popular use cases for Apache Kafka.

## Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc).

In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

## Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type.

These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view

## Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

## Log Aggregation

Kafka can be used as a replacement for a log aggregation solution.

Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing.

Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption.

In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency

## Stream Processing

Many users end up doing stage-wise processing of data where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption.

For example a processing flow for article recommendation might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage might attempt to match this content to users.

This creates a graph of real-time data flow out of the individual topics.

## Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log

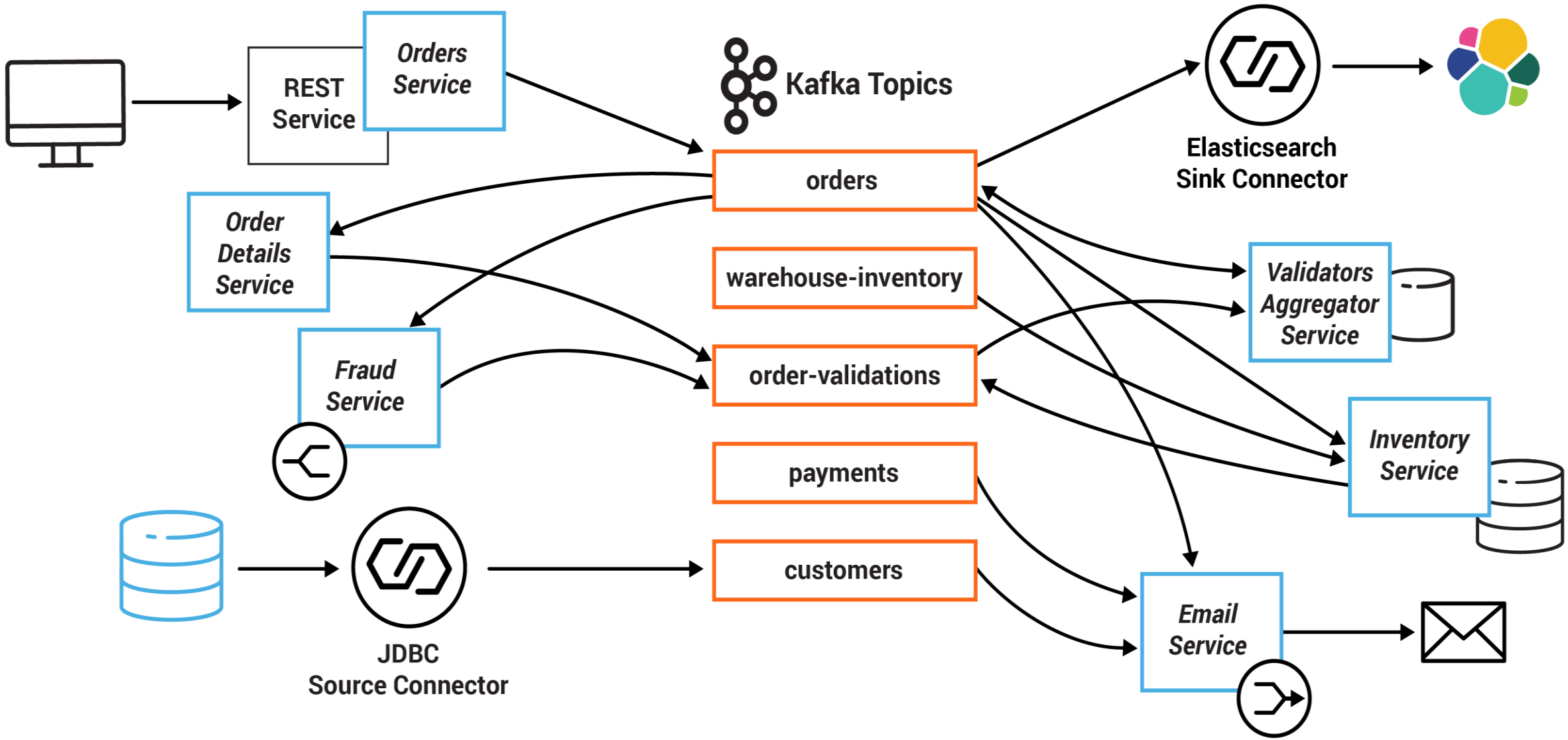
## Event Sourcing

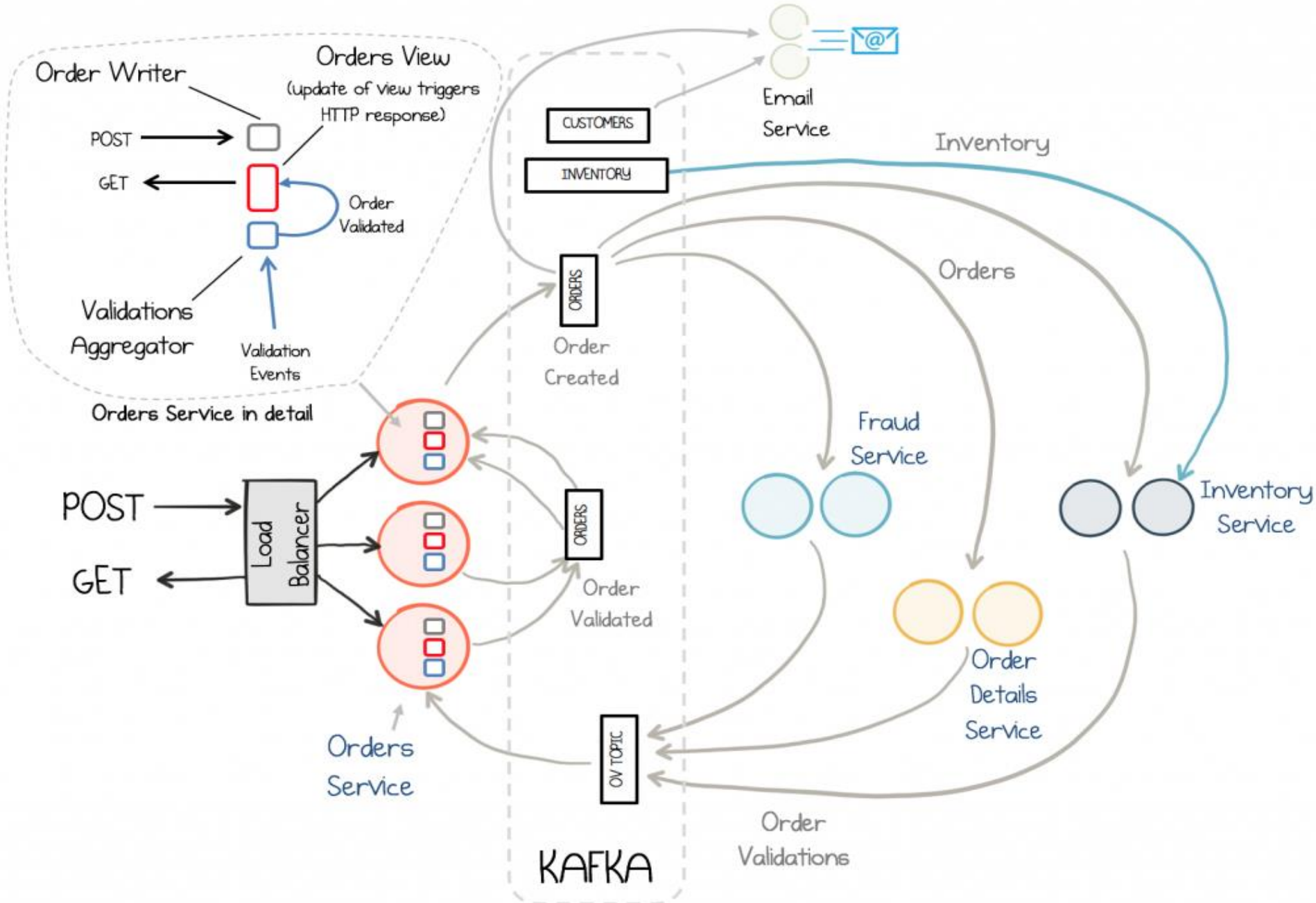
Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events.

This Is used in microservice architecture for distributed transactions.

# Use case:  Ecommerce-order-processing


REST Service
Orders Service
Kafka Topics
Elasticsearch Sink Connector
orders
Order Details Service
warehouse-inventory
Validators Aggregator Service
Fraud Service
order-validations
Inventory Service
payments
customers
Email Service
JDBC Source Connector


Order Writer
Orders View
(update of view triggers HTTP response)
POST
GET
Order Validated
Validations Aggregator
Validation Events
Orders Service in detail
POST
GET
Load Balancer
Orders Service
CUSTOMERS
INVENTORY
ORDERS
ORDERS
OV TOPIC
KAFKA
Email Service
Inventory
Orders
Order Created
Order Validated
Fraud Service
Inventory Service
Order Details Service
Order Validations

In this example, the system centres on an **Orders Service** which exposes a REST interface to POST and GET Orders.

Posting an Order creates an event in Kafka that is recorded in the topic orders.

This is picked up by different validation engines (**Fraud Service, Inventory Service and Order Details Service**), which validate the order in parallel, emitting a PASS or FAIL based on whether each validation succeeds.

The result of each validation is pushed through a separate topic: Order Validations, so that we retain the single writer status of the Orders Service.

The results of the various validation checks are aggregated in the **Validation Aggregator Service**, which then moves the order to a Validated or Failed state, based on the combined result.

There is a simple service that sends emails, and another that collates orders and makes them available in a search index using Elasticsearch.

# Partitions, Segments & Indexes

**Partitions and Segments**

Topics are made of Partitions

Partitions are made of segments (files)

Ex :

Partition-0

segment 0: segment: 1 segment: 2

Offset 0-200 Offset 201-400 Offset 401-? (Active)

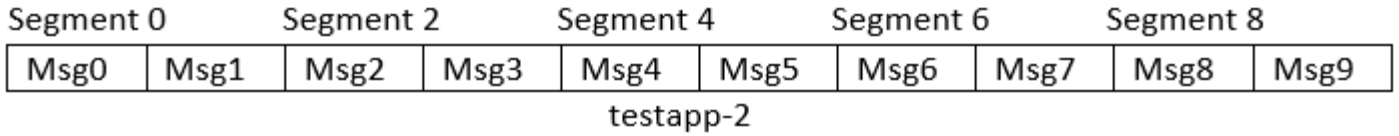
Note : Only one segment is ACTIVE per partition

## Segment and Indexes

Segments come with two indexes (files)

- An offset to position index: allows Kafka where to read to find a message
- A timestamp to offset index: allows Kafka to find messages with a timestamp

This will help Kafka to find the data in a constant time.




Index File

| Offset | Position |
|---|---|
| 0 | 0 |
| 1 | 200 |

Log File

| Offset | Position | Time | Message |
|---|---|---|---|
| 0 | 0 | 1222333333333 | Faith |
| 1 | 200 | 1222233333444 | Trust |

**log.segment.bytes** : the max size of a single segment in bytes (default: 1GB)

A smaller log.segment.bytes means :

- More segments per partitions
- Log compaction happens more often
- But Kafka has to keep more files opened (Error: Too many open files)

**log.segment.ms:** the time kafka will wait before committing if the segment is not full (default:  1 week)

A smaller log.segment.ms means:

You set a max frequency for log compaction (more frequent triggers)

Maybe you want daily compaction instead of weekly?

**Log Cleanup Policies**

Kafka data will get expired, based on the policies:

Policy 1 : log.cleanup.policy=delete  (default for all topics)
> Delete based on age of data (Default is a week)
> Delete based on max size of log (default -1  i.e. infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic_Consumer_offsets)
> Delete based on keys of your messages
> will delete old duplicate keys after that active segment is committed

**log.retention.hours**
number of hours to keep data for (default is 168 hrs / one week)

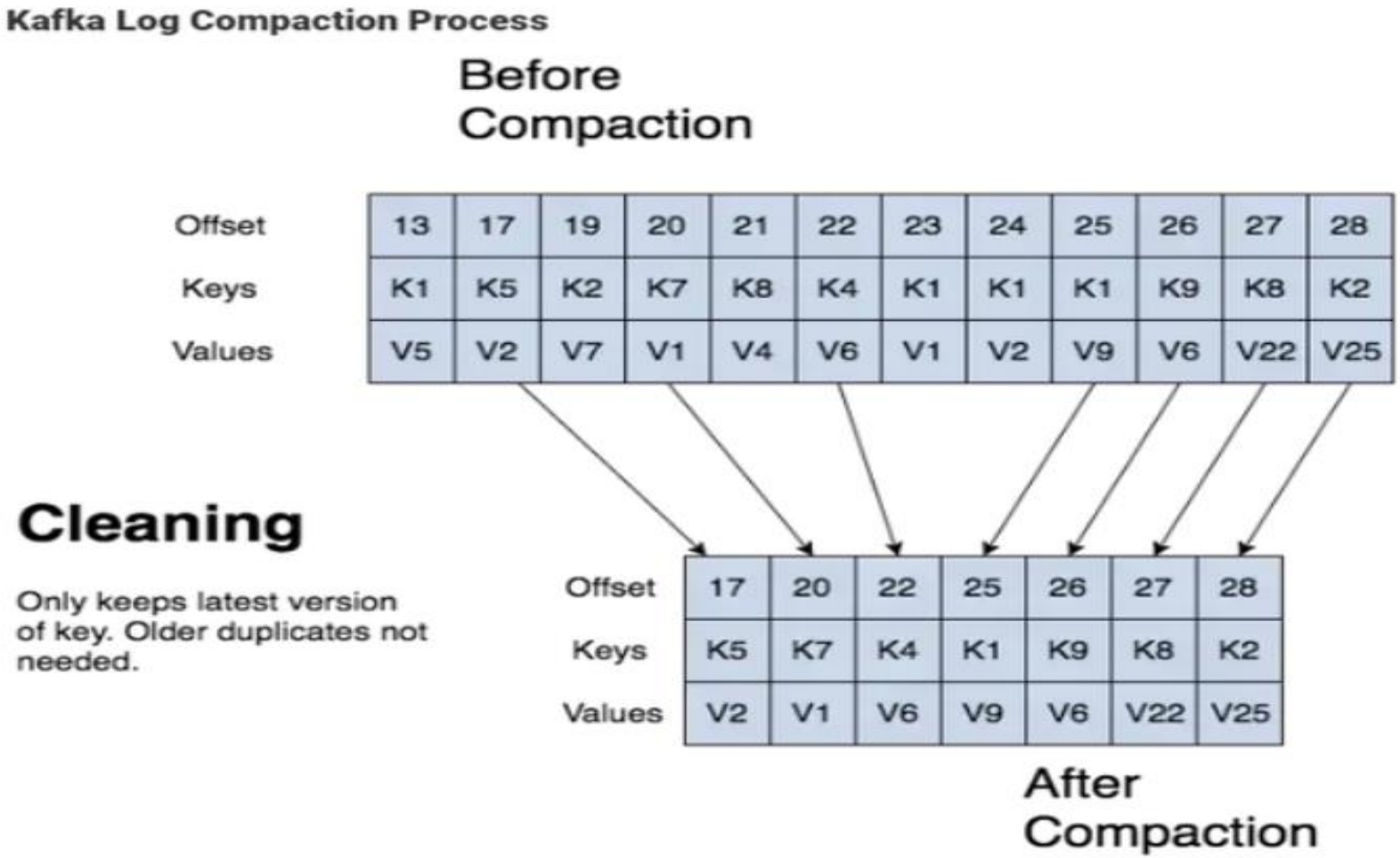
**log.retention.bytes**
max size in bytes for each partition [ defailt is infinite (-1) ]

**Realtime usecases:**

1. One week of retention:
   log.retention.hours=168 and log.retention.bytes=-1

2. Infinite time retention bounded by 500MB
   log.retention.hours=-1  and log.retention.bytes=524288000

Compaction takes the snap shot of the latest value to the key and whenever the failure of happens consumer recovers from the compacted logs as shown below.



**Cleanup policy Compact**

kafka-topics –boot-strap localhost:9092 --create --topic empsalary --partitions 1 --replication-factor 1 --config cleanup.policy=compact --config min.cleanable.dirty.ratio=0.001 --config segment.ms=5000

Note : min.cleanable.dirty.ratio : This configuration controls how frequently the log compactor will attempt to clean the log. Default is 0.5. This ratio bounds the maximum space wasted in the log by duplicates (at 50% at most 50% of the log could be duplicates)

**Start a consumer**

kafka-console-consumer.sh --boot-strap localhost:9092 --topic emp-salary --property print.key=true --property key.separator=,

**Start a producer**

kafka-console-producer.sh --broker-list localhost:9092 --topic empsalary --property parse.key=true --property key.separator=,

ramana,5000
krishna,3000
rushi,9000
ramana,9000
krishna, 9000
now you

./kafka-run-class kafka.tools.DumpLogSegments --deep-iteration --print-data-log --files 00000000000000000000.log

**Get End offset details for each partition**

kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic first-topic

# Producers, Consumers & Consumer Groups

## Producer Overview

There are many reasons an application might need to write messages to Kafka:

- recording user activities for auditing or analysis,
- recording metrics,
- storing log messages,
- recording information from smart appliances,
- communicating asynchronously with other applications,
- buffering information before writing to a database, and much more.

**Those diverse use cases also imply diverse requirements:**

- is every message critical, or can we tolerate loss of messages?
- Are we OK with accidentally duplicating messages?
- Are there any strict latency or throughput requirements we need to support?

- **credit card transaction processing**, it is critical to **never lose** a single message nor duplicate any messages.

  Latency should be low but latencies up to 500ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

- A different use case might be to **store click information from a website**. In that case, **some message loss** or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience.

**Kafka Producer**

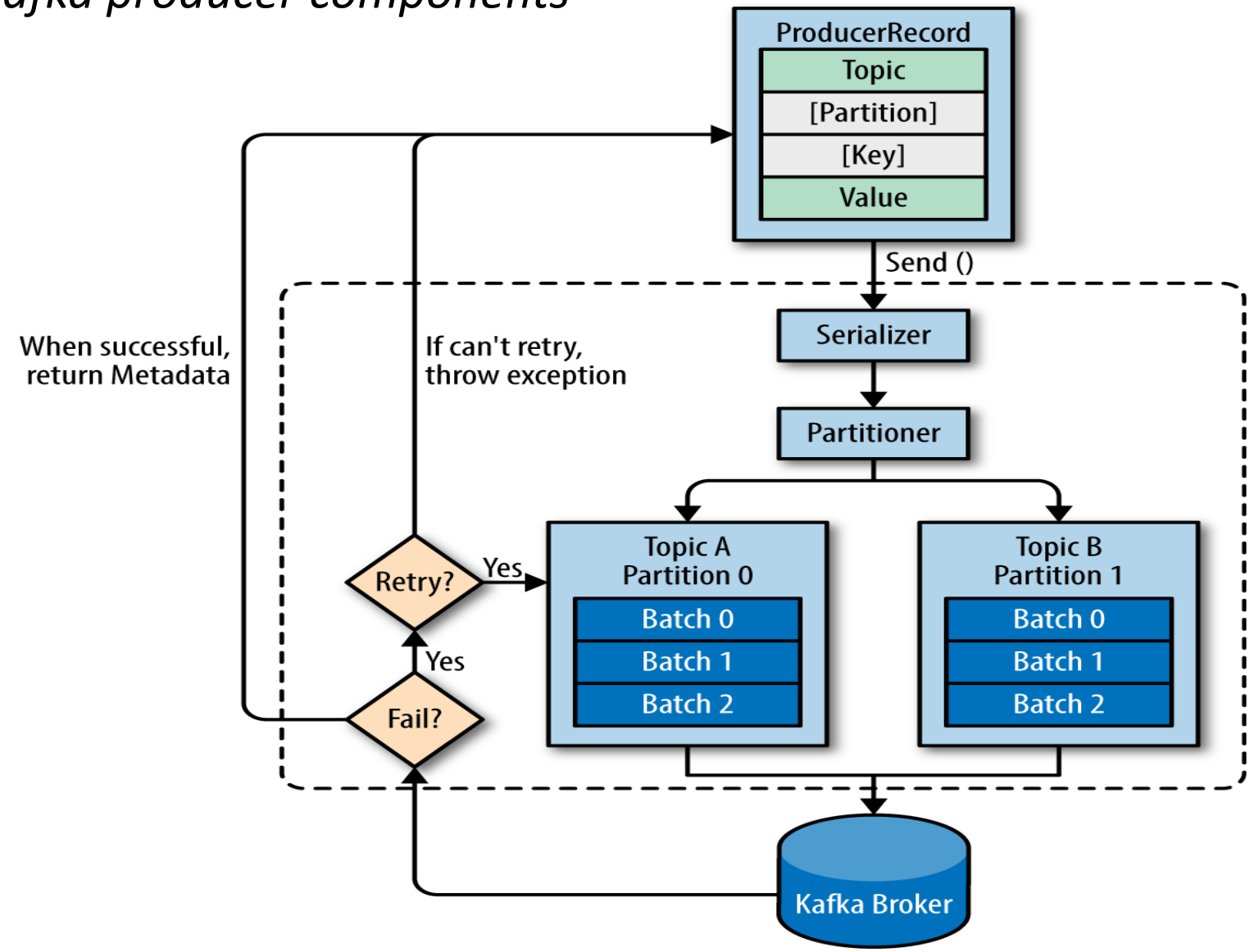
- Kafka client that publishes records to Kafka cluster

- The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances

- Producer has pool of buffer that holds to-be-sent records
  background I/O threads turning records into request bytes and transmit requests to Kafka

- Close producer so producer will not leak resources

*High-level overview of Kafka producer components*



- We start producing messages to Kafka by creating a ProducerRecord, which must include the topic we want to send the record to and a value.

- Optionally, we can also specify a key and/or a partition.

- Once we send the ProducerRecord, the first thing the producer will do is serialize the key and value objects to ByteArrays so they can be sent over the network.

- Next, the data is sent to a partitioner. If we specified a partition in the ProducerRecord, the partitioner doesn't do anything and simply returns the partition.

- If we didn't, the partitioner will choose a partition for us, usually based on the ProducerRecord key[hashCode(key) % noOfPartitions]

- Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition.

- A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

- When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a RecordMetadata object with the  topic, partition, and the offset of the record within the partition.

- If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

**Kafka Producer Send, Acks and Buffers**

- send() method is asynchronous
  - adds the record to output buffer and return right away
  - buffer used to batch records for efficiency IO
  - and compression

- acks config controls Producer record durability.
  - ”all" setting ensures full commit of record, and is most durable and least fast setting.

- Producer can retry failed requests

- Producer has buffers of unsent records per topic partition (sized at batch.size)

**Producer Acks**

- Producer Config property acks  (default 1)

- Write Acknowledgment received count required from partition leader before write request deemed complete

- Controls Producer sent records durability

- Can be all (-1), none (0), or leader (1)

## Acks 0 (NONE)

- Producer does not wait for any ack from broker at all
- Records added to the socket buffer are considered sent
- No guarantees of durability - maybe
- Record Offset returned is set to -1 (unknown)
- Record loss if leader is down
- Use Case: maybe log aggregation

**Acks 1 (LEADER)**

- Partition leader wrote record to its local log but responds without followers confirmed writes

- If leader fails right after sending ack, record could be lost
  Followers might have not replicated the record

- Record loss is rare but possible

- Use Case: log aggregation

**Acks -1 (ALL)**

- Leader gets write confirmation from full set of ISRs before sending ack to producer
- Guarantees record not be lost as long as one ISR remains alive
- Strongest available guarantee
- Even stronger with broker setting min.insync.replicas (specifies the minimum number of ISRs that must acknowledge a write)
- Most Use Cases will use this and set a min.insync.replicas > 1

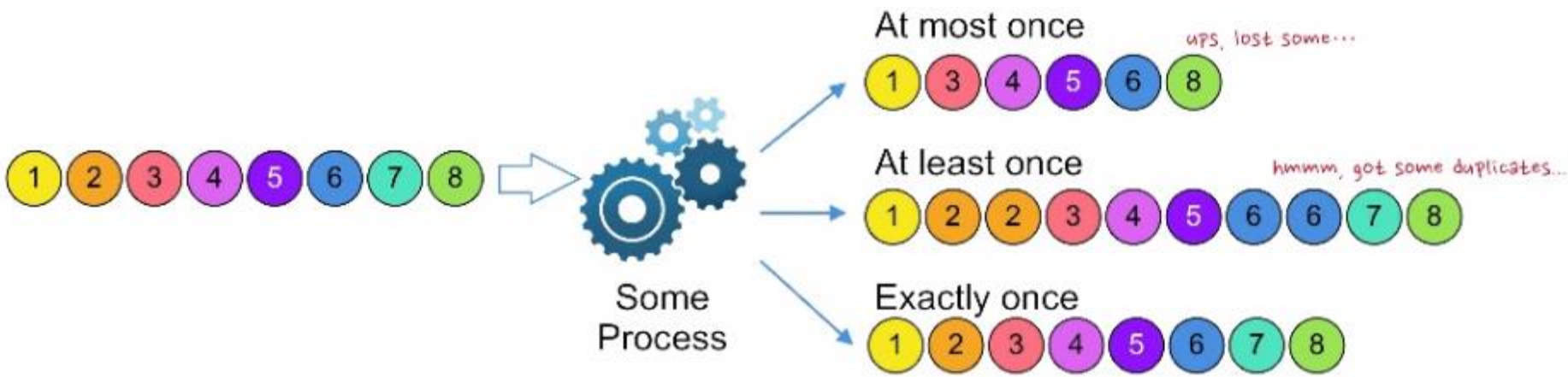
## Message Delivery Semantics

- **At most once**
  Messages may be lost but are never redelivered

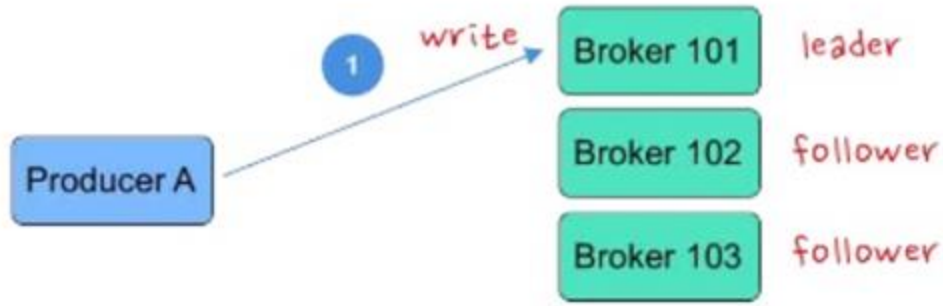
- **At least once**
  Messages are never lost but may be redelivered

- **Exactly once**
  this is what people actually want, each message is delivered once and only once

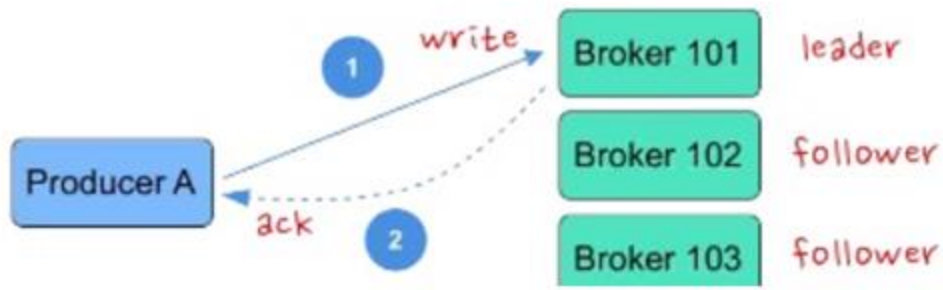
# Delivery Guarantees



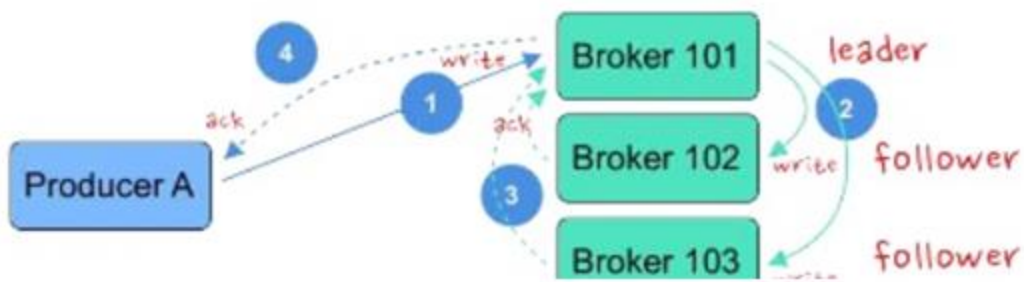
# Producer Guarantees




Acks 0 (NONE)




Acks 1 (LEADER)




Acks -1 (ALL)

# Configurable ISR Commits

| *ACK mode* | *Latency* | *On Failures* |
|---|---|---|
| *“no"* | *no network delay* | *some data loss* |
| *“leader"* | *1 network roundtrip* | *a few data loss* |
| *“all"* | *~2 network roundtrips* | *no data loss* |

# Idempotent & transactional Producer
# [Exactly Once Delivery]

KafkaProducer supports two additional modes:

idempotent producer
transactional producer.

The idempotent producer strengthens Kafka's delivery semantics from at least once to exactly once delivery.

In particular producer retries will no longer introduce duplicates.

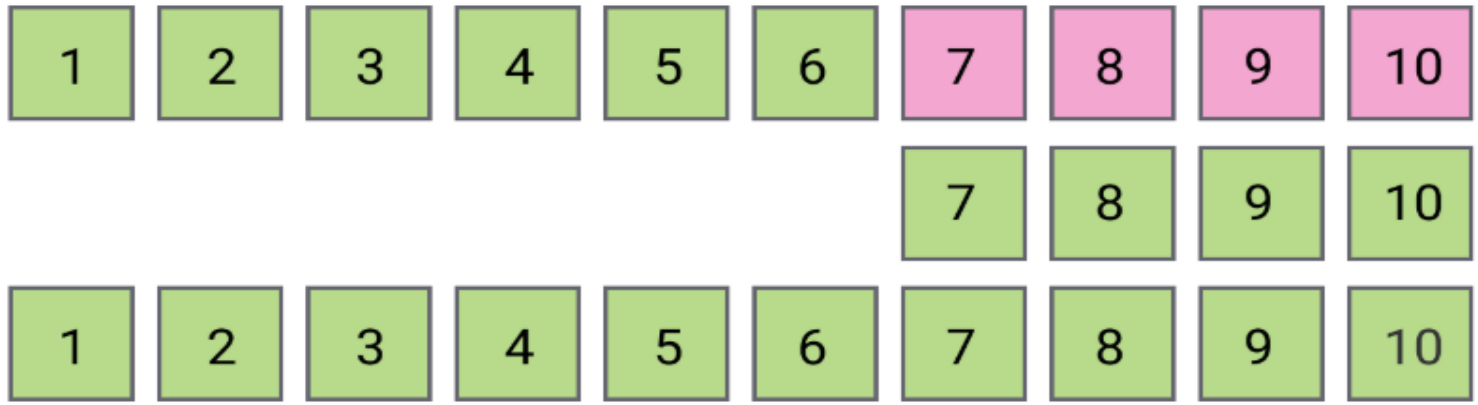
The transactional producer allows an application to send messages to multiple partitions (and topics!) atomically.

When a producer sends messages to a topic, things can go wrong, such as short connection failures.

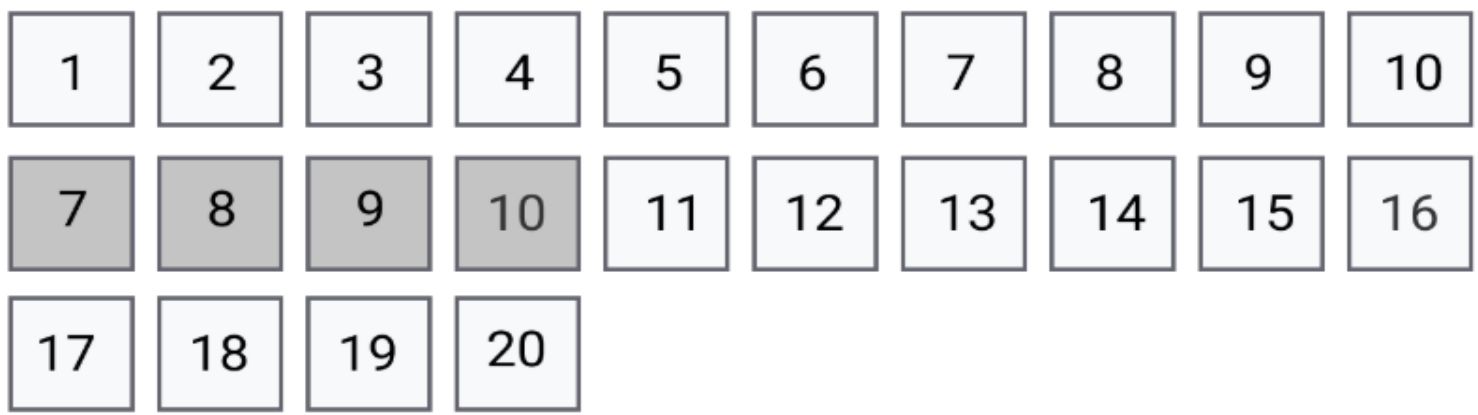
When this happens, any messages that are pending acknowledgements can either be resent or discarded.

The messages may have been successfully written to the topic, or not, there is no way to know. If we resend then we may duplicate the message, but if we don’t resend then the message may essentially be lost.

## Producer Resends on Failure




## Consumer reads duplicates



To enable idempotence, the enable.idempotence configuration must be set to true.

If set, the retries config will default to Integer.MAX_VALUE and the acks config will default to all.

There are no API changes for the idempotent producer, so existing applications will not need to be modified to take advantage of this feature.

Each producer gets assigned a Producer Id (PID) and it includes its PID every time it sends messages to a broker. Additionally, each message gets a an increasing sequence number.

A separate sequence is maintained for each topic partition that a producer sends messages to.

On the broker side, on a per partition basis, it keeps track of the largest PID-Sequence Number combination is has successfully written. When a lower sequence number is received, it is discarded.

But with idempotence enabled, each message comes with a PID and sequence number:

- M1 (PID: 1, SN: 1) - written to partition. For PID 1, Max SN=1
- M2 (PID: 1, SN: 2) - written to partition. For PID 1, Max SN=2
- M3 (PID: 1, SN: 3) - written to partition. For PID 1, Max SN=3
- M4 (PID: 1, SN: 4) - written to partition. For PID 1, Max SN=4
- M5 (PID: 1, SN: 5) - written to partition. For PID 1, Max SN=5
- M6 (PID: 1, SN: 6) - written to partition. For PID 1, Max SN=6
- M4 (PID: 1, SN: 4) - rejected, SN <= Max SN
- M5 (PID: 1, SN: 5) - rejected, SN <= Max SN
- M6 (PID: 1, SN: 6) - rejected, SN <= Max SN
- M7 (PID: 1, SN: 7) - written to partition. For PID 1, Max SN=7
- M8 (PID: 1, SN: 8) - written to partition. For PID 1, Max SN=8
- M9 (PID: 1, SN: 9) - written to partition. For PID 1, Max SN=9
- M10 (PID: 1, SN: 10) - written to partition. For PID 1, Max SN=10

To use the transactional producer, we must set the transactional.id configuration property.

If the transactional.id is set, idempotence is automatically enabled along with the producer configs which idempotence depends on.

Further, topics which are included in transactions should be configured for durability.

In particular, the replication.factor should be at least 3, and the min.insync.replicas for these topics should be set to 2

In order for transactional guarantees to be realized from end-to-end, the consumers must be configured to read only committed messages as well.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());

producer.initTransactions();

try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
} catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
}
producer.close();
```

Consumer – isolation.level (read_committed)

```
KafkaConsumer consumer = createKafkaConsumer(
  “bootstrap.servers”, “localhost:9092”,
  “group.id”, “my-group-id”,
  "isolation.level", "read_committed");
```

**Kafka Producer: Buffering and batching**

- Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight max.in.flight.requests.per.connection)

- The default value is 5

- The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).

- To reduce requests count, set linger.ms > 0
  - -wait up to linger.ms before sending or until batch fills up whichever comes first

  - -Under heavy load linger.ms not met, under light producer load used to increase broker IO throughput and increase compression

- buffer.memory controls total memory available to producer for buffering
  If records sent faster than they can be transmitted to Kafka then this buffer gets exceeded then additional send calls block. If period blocks (max.block.ms) after then Producer throws a TimeoutException

**Batching by Size**

- Producer config property: batch.size
  Default 16KB

- Producer batch records
  fewer requests for multiple records sent to same partition
  Improves IO throughput and performance on both
  producer and server

- If record is larger than the batch size, it will not be batched

- Producer sends requests containing multiple batches
  batch per partition

- Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

**Producer Buffer Memory Size**

- Producer config property: buffer.memory
  default 32MB

- Total memory (bytes) producer can use to buffer records to be sent to broker

- Producer blocks up to max.block.ms if buffer.memory is exceeded, if it is sending faster than the broker can receive, exception is thrown

Note :

batch.size : By default, holds upto 16K per partition

buffer.memory : By default, holds upto 32MB for all partition buffers

The **buffer.memory** controls the total amount of memory available to the producer for buffering. If records are sent faster than they can be transmitted to the server then this buffer space will be exhausted.

When the buffer space is exhausted additional send calls will block. The threshold for time to block is determined by **max.block.ms** after which it throws a TimeoutException.

```
props.put(ProducerConfig.LINGER_MS_CONFIG, 100);
props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);
props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 60000);
props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
```

## request.timeout.ms

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

| | |
|---|---|
| **Type:** | int |
| **Default:** | 30000 (30 seconds) |
| **Valid Values:** | |
| **Importance:** | high |
| **Update Mode:** | read-only |

Problem:

Exception thrown when sending with key and payload{} to topic xyz
TimeoutException: Expiring 1  records for xyz-topic-1:120000ms has
passed since batch creation

The buffer.memory controls the total amount of memory available to the producer for buffering. If records are sent faster than they can be transmitted to the server then this buffer space will be exhausted. When the buffer space is exhausted additional send calls will block. The threshold for time to block is determined by max.block.ms after which it throws a TimeoutException.

**Batching by Time and Size**

- Producer config property: linger.ms
  Default 0

- Producer groups together any records that arrive before they can be sent into a batch
  good if records arrive faster than they can be sent out

- Producer can reduce requests count even under moderate load using linger.ms

- linger.ms adds delay to wait for more records to build up so larger batches are sent
  good brokers throughput at cost of producer latency

- If producer gets records whos size is batch.size or more for a broker's leader partitions, then it is sent right away

- If Producers gets less than batch.size but linger.ms interval has passed, then records for that partition are sent

- Increase to improve throughput of Brokers and reduce broker load (common improvement)

**Compressing Batches**

- Producer config property: compression.type
  Default 0

- Producer compresses request data

- By default producer does not compress

- Can be set to none, gzip, snappy, or lz4

- Compression is by batch
  improves with larger batch sizes

- End to end compression possible if Broker config “compression.type” set to producer. Compressed data from producer sent to log and consumer by broker

**For high-throughput producers, tune buffer sizes**

particularly buffer.memory and batch.size (which is counted in bytes). Because batch.size is a per-partition setting, producer performance and memory usage can be correlated with the number of partitions in the topic.

The values here depend on several factors: producer data rate (both the size and number of messages), the number of partitions you are producing to, and the amount of memory you have available.

Keep in mind that larger buffers are not always better because if the producer stalls for some reason (say, one leader is slower to respond with acknowledgments), having more data buffered on-heap could result in more garbage collection.

**Custom Serializers**

- Serialization is the process of converting an object into a stream of bytes.
- Deserialization is the process of converting bytes into desired data type.
- Kafka provides built in serializers and deserializers for a few data types(String, Long, Double, Integer, Bytes etc.,)
- If our requirement is not fit in  with built in serializers, then we need to  write custom one.
- Just need to be able to convert to/fro a byte[]
- Serializers work for keys and values
- value.serializer and key.serializer

**How DefaultPartitioner works?**

The below explains the default partitioning strategy:

- If a partition is specified in the record, use it - this can be used when we already know the partition.

- If no partition is specified but a key is present, choose a partition based on a hash of the key -
  It can be used when we want to distribute the data based on a key.
  The following formula is used to determine the partition:
  hashCode(key) % noOfPartitions

- If no partition or key is present, choose a partition in a round-robin fashion - If we are not bothered which partition our data is going to, this strategy can be used.

Default strategies work well to start with. The problem comes when we want the data for the same customers to go to  the same partition and we have used a composite key to partition data.

For example, and we have used a key which contains a customer id and a date. Now although the customer id is fixed, the date can change and we will end up with a different hash code.

Which means the data for the same customer will go to a different partition.

**Producer Partitioning**

- Producer config property: partitioner.class
  org.apache.kafka.clients.producer.internals.DefaultPartitioner

- Partitioner class implements Partitioner interface

**Producer Interception**

- Producer config property: interceptor.classes
  empty (we can pass an comma delimited list)
- interceptors implementing ProducerInterceptor interface
- intercept records producer sent to broker and after acks
- Interceptor API will allow mutate the records to support the ability to add metadata to a message for auditing/end-to-end monitoring.

**KafkaProducer send() Method**

- Two forms of send with callback and with no callback both return Future
  - Asynchronously sends a record to a topic
  - Callback gets invoked when send has been acknowledged.

- send is asynchronous and return right away as soon as record has added to send buffer

- Sending many records at once without blocking for response from Kafka broker
- Result of send is a RecordMetadata
  - record partition, record offset, record timestamp

- Callbacks for records sent to same partition are executed in order

**KafkaProducer send() Exceptions**

InterruptException - If the thread is interrupted while blocked

SerializationException - If key or value are not valid objects given configured serializers

TimeoutException - If time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc.

KafkaException - If Kafka error occurs not in public API.

**KafkaProducer flush() method**

- flush() method sends all buffered records now (even if linger.ms > 0)
  blocks until requests complete

- Useful when consuming from some input system and pushing data into Kafka

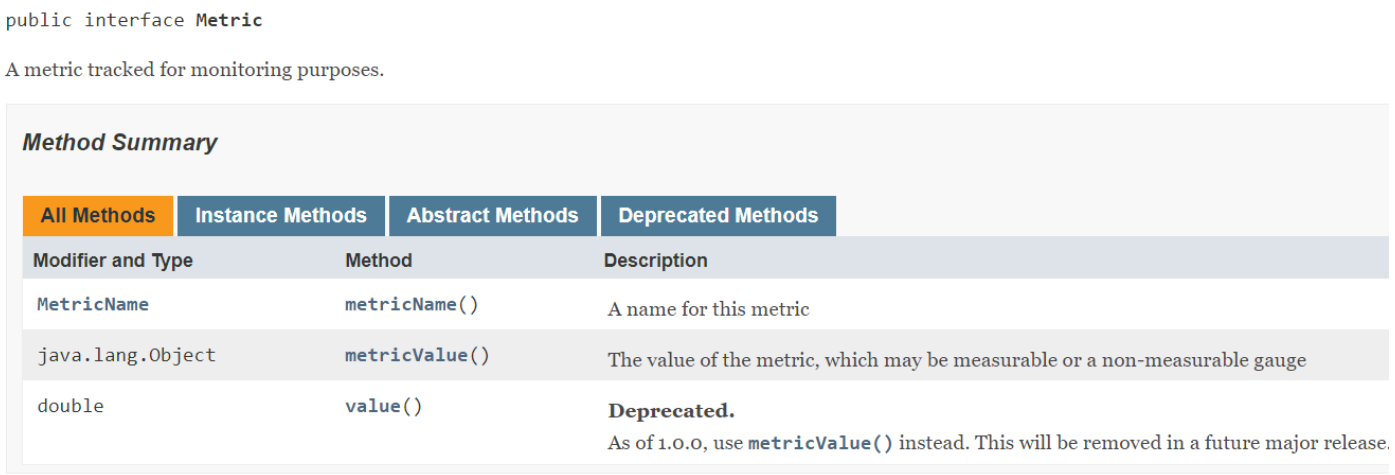
- flush() ensures all previously sent messages have been sent
  we could mark progress as such at completion of flush

## KafkaProducer metrics() method

The metrics() method is used to get a map of metrics:

public Map<MetricName,? extends Metric> metrics().

This method is used to get a full set of producer metrics.



public interface **Metric**

A metric tracked for monitoring purposes.

***Method Summary***

All Methods | Instance Methods | Abstract Methods | Deprecated Methods

| Modifier and Type | Method | Description |
|---|---|---|
| MetricName | metricName() | A name for this metric |
| java.lang.Object | metricValue() | The value of the metric, which may be measurable or a non-measurable gauge |
| double | value() | **Deprecated.** As of 1.0.0, use metricValue() instead. This will be removed in a future major release. |

org.apache.kafka.common

# Class MetricName

java.lang.Object
    org.apache.kafka.common.MetricName

---

```
public final class MetricName
extends java.lang.Object
```

The `MetricName` class encapsulates a metric's name, logical group and its related attributes. It should be constructed using metrics.MetricName(...).

This class captures the following parameters

```
  name The name of the metric
  group logical group name of the metrics to which this metric belongs.
  description A human-readable description to include in the metric. This is optional.
  tags additional key/value attributes of the metric. This is optional.
```

# Consumers

## Consumers

- consumers read data from a topic
- consumers know which broker to read from
- In case of broker failures, consumers know how to recover
- Data is read in order within each partitions

**KafkaConsumer**

- A client that consumes records from a Kafka cluster.
- This client transparently handles the failure of Kafka brokers, and transparently adapts as topic partitions it fetches migrate within the cluster.
- This client also interacts with the broker to allow groups of consumers to load balance consumption using consumer groups.
- Consumer maintains TCP connections to Kafka brokers in cluster
- Use close() method to not leak resources
- Consumer is NOT thread-safe

org.apache.kafka.clients.consumer

## Interface Consumer<K,V>

**All Superinterfaces:**

`java.lang.AutoCloseable, java.io.Closeable`

**All Known Implementing Classes:**

`KafkaConsumer, MockConsumer`

org.apache.kafka.clients.consumer

## Class KafkaConsumer<K,V>

java.lang.Object
    org.apache.kafka.clients.consumer.KafkaConsumer<K,V>

**All Implemented Interfaces:**

`java.io.Closeable, java.lang.AutoCloseable, Consumer<K,V>`

## KafkaConsumer: Offsets and Consumer Position

- Kafka maintains a numerical offset for each record in a partition.
- This offset acts as a unique identifier of a record within that partition, and also denotes the position of the consumer in the partition.
- The position of the consumer gives the offset of the next record that will be given out.

- It automatically advances every time the consumer receives messages in a call to poll(long).

- Consumer *committed position* is last offset that has been stored to broker. If consumer fails, it picks up at last committed position

- Consumer can auto commit offsets (enable.auto.commit) periodically (auto.commit.interval.ms) or do commit explicitly using commitSync() and commitAsync()

**KafkaConsumer: Consumer Groups and Topic Subscriptions**

- Consumers organized into consumer groups (Consumer instances with same group.id).
- Pool of consumers divide work of consuming and processing records  Processes or threads running on same box or distributed for scalability/fault tolerance
- Kafka shares topic partitions among all consumers in a consumer group.

- Each partition is assigned to exactly one consumer in consumer group. Example topic has six partitions, and a consumer group has two consumer processes, each process gets consume three partitions

- Failover and Group rebalancing:
  >> if a consumer fails, Kafka reassigned partitions from failed consumer to other consumers in same consumer group
  >> if new consumer joins, Kafka moves partitions from existing consumers to new one

**Consumer Groups and Topic Subscriptions**

- Consumer group form single logical subscriber made up of multiple consumers
- Kafka is a multi-subscriber system, Kafka supports N number of consumer groups for a given topic without duplicating data
- To get something like a MOM queue all consumers would be in single consumer group : Load balancing like a MOM queue
- To get something like MOM pub-sub each process would have its own consumer group

## KafkaConsumer: Partition Reassignment

- Consumer partition reassignment in a consumer group happens automatically

- Consumers are notified via ConsumerRebalanceListener, this will triggers consumers to finish necessary clean up.

- Consumer can use API to assign specific partitions using assign(Collection). This will disables dynamic partition assignment and consumer group coordination.

- Dead client may see CommitFailedException thrown from a call to commitSync(). Only active members of consumer group can commit offsets.

**Controlling Consumers Position**

- We can control consumer position. Moving consumer forward or backwards - consumer can re-consume older records or skip to most recent records

- Use consumer.seek(TopicPartition, long) to specify new position.
  consumer.seekToBeginning(Collection) and
  consumer.seekToEnd(Collection)

- Use Case

  >> Time-sensitive record processing: Skip to most recent records

  >> Bug Fix: Reset position before bug fix and replay log from there

  >> Restore State for Restart or Recovery: Consumer initialize position on start-up to whatever is contained in local store and replay missed parts (cache warm-up or replacement in case of failure assumes Kafka retains sufficient history or you are using log compaction)

## Storing Offsets : Managing Offsets

- For the consumer to manage its own offset we just need to do the following:
  - >> Set enable.auto.commit=false
  - >> Use offset provided with each ConsumerRecord to save your position (partition/offset)
  - >> On restart restore consumer position using kafkaConsumer.seek(TopicPartition, long).
- Usage like this simplest when the partition assignment is also done manually using assign() instead of subscribe()

## Storing Offsets : Managing Offsets

- If using automatic partition assignment, we must handle cases where partition assignments change

  >> Pass ConsumerRebalanceListener instance in call to kafkaConsumer.subscribe(Collection,ConsumerRebalanceListener)

  >> When partitions taken from consumer, commit its offset for partitions by Implementing ConsumerRebalanceListener.onPartitionsRevoked(Collection)

  >> When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ConsumerRebalanceListener.onPartitionsAssigned(Collection)

```
// Subscribe to the topic.
consumer.subscribe(Collections.singletonList(
        StockAppConstants.TOPIC),
        new SeekToConsumerRebalanceListener(consumer, seekTo, location));
```

**KafkaConsumer: Consumer Alive Detection**

- Consumers join consumer group after subscribe and then poll() is called

- Automatically, consumer sends periodic heartbeats to Kafka brokers server

- If consumer crashes or unable to send heartbeats for a duration of session.timeout.ms(default: 3s), then consumer is deemed dead and its partitions are reassigned

## KafkaConsumer: Manual Partition Assignment

- Instead of subscribing to the topic using subscribe, you can call assign(Collection) with the full topic partition list

```
String topic = “log-replication“;
TopicPartition part0 = new TopicPartition(topic, 0);
TopicPartition part1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(part0, part1));
```

- Manual partition assignment negates use of group coordination, and auto consumer fail over - Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion)

- We have to use assign() or subscribe() but not both

## KafkaConsumer: Consumer Alive if Polling

- Calling poll() marks consumer as alive
  - >> If consumer continues to call poll(), then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every max.poll.interval.ms interval) default : max.poll.interval.ms (5 minutes)
  - >> Not calling poll(), even if consumer is sending heartbeats, consumer is still considered dead

- Processing of records from poll has to be faster than max.poll.interval.ms interval

- max.poll.records is used to limit total records returned from a poll call - easier to predict max time to process records on each poll interval

## Message Delivery Semantics

- **At most once**
  Messages may be lost but are never redelivered

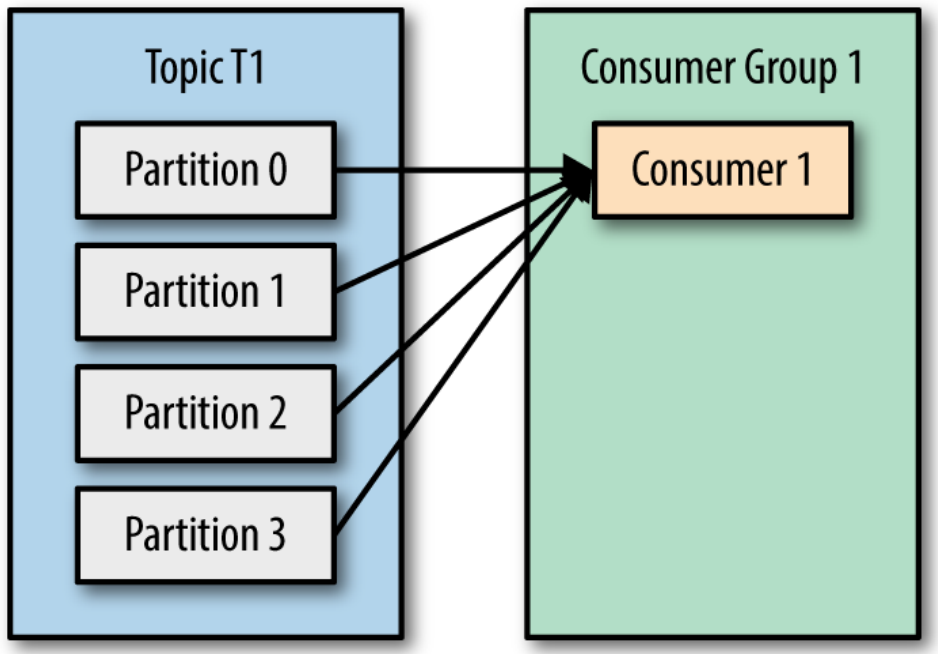
- **At least once**
  Messages are never lost but may be redelivered

- **Exactly once**
  this is what people actually want, each message is delivered once and only once

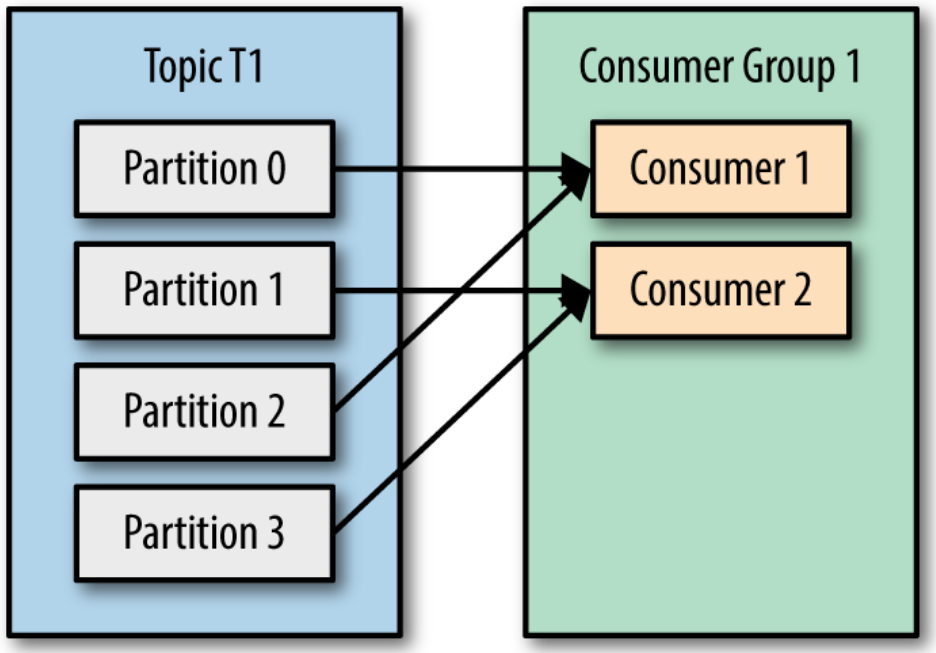
## Consumer Groups

- consumers read data in consumer groups
- each consumer within a group reads from exclusive partitions
- if you have more consumers than partitions, some consumers will be inactive

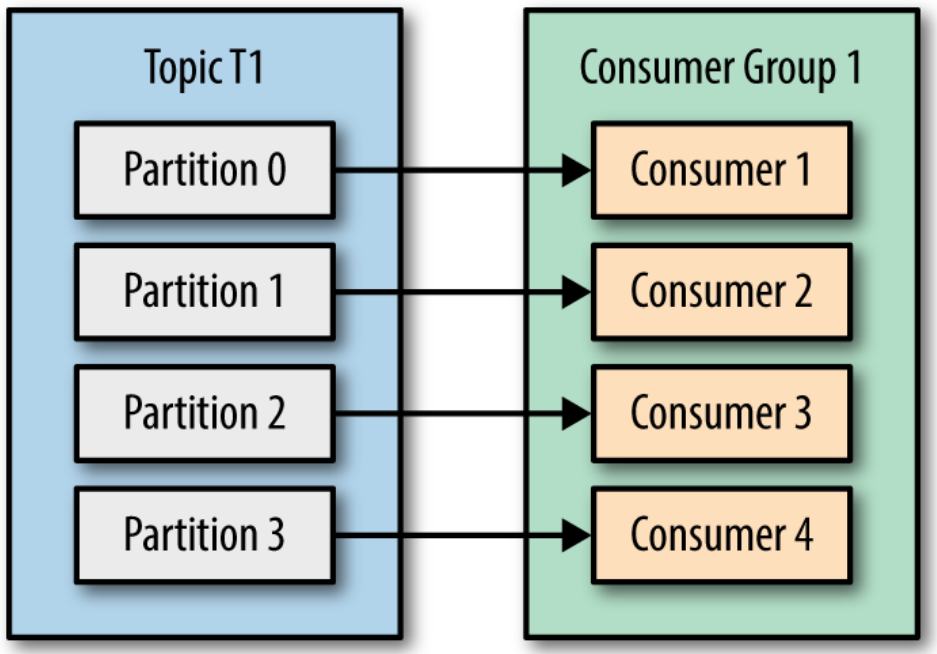
*One Consumer group with four partitions*

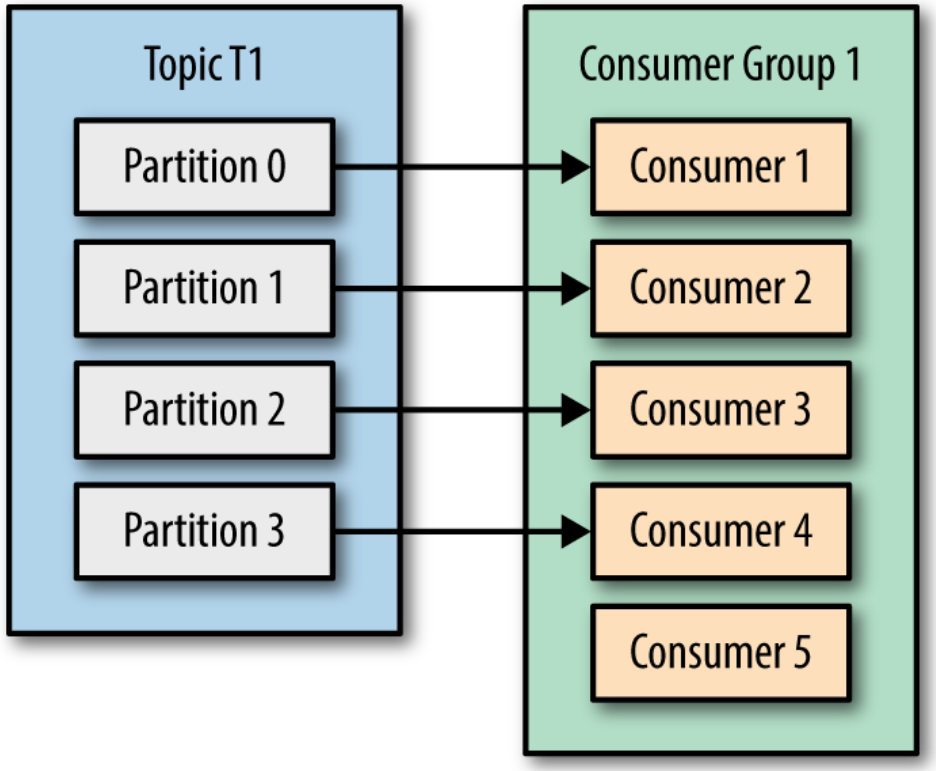


*Four partitions split to two consumer groups*

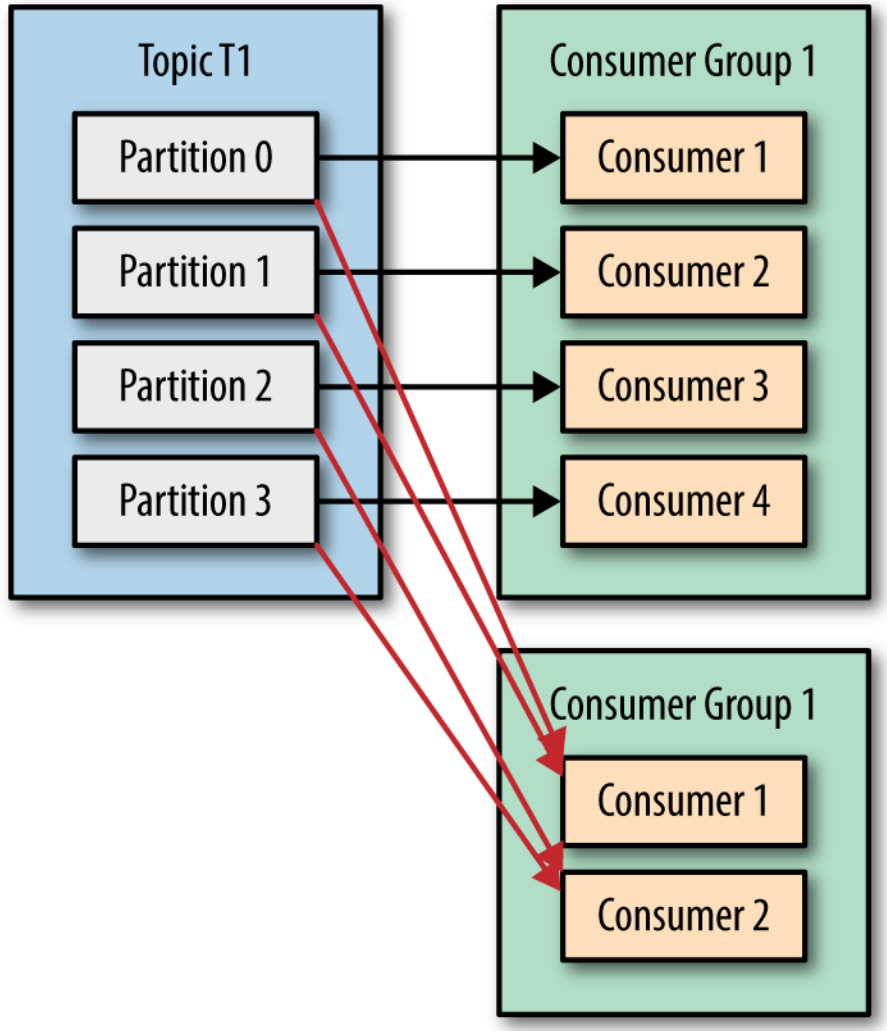


*Four consumer groups to one partition each*

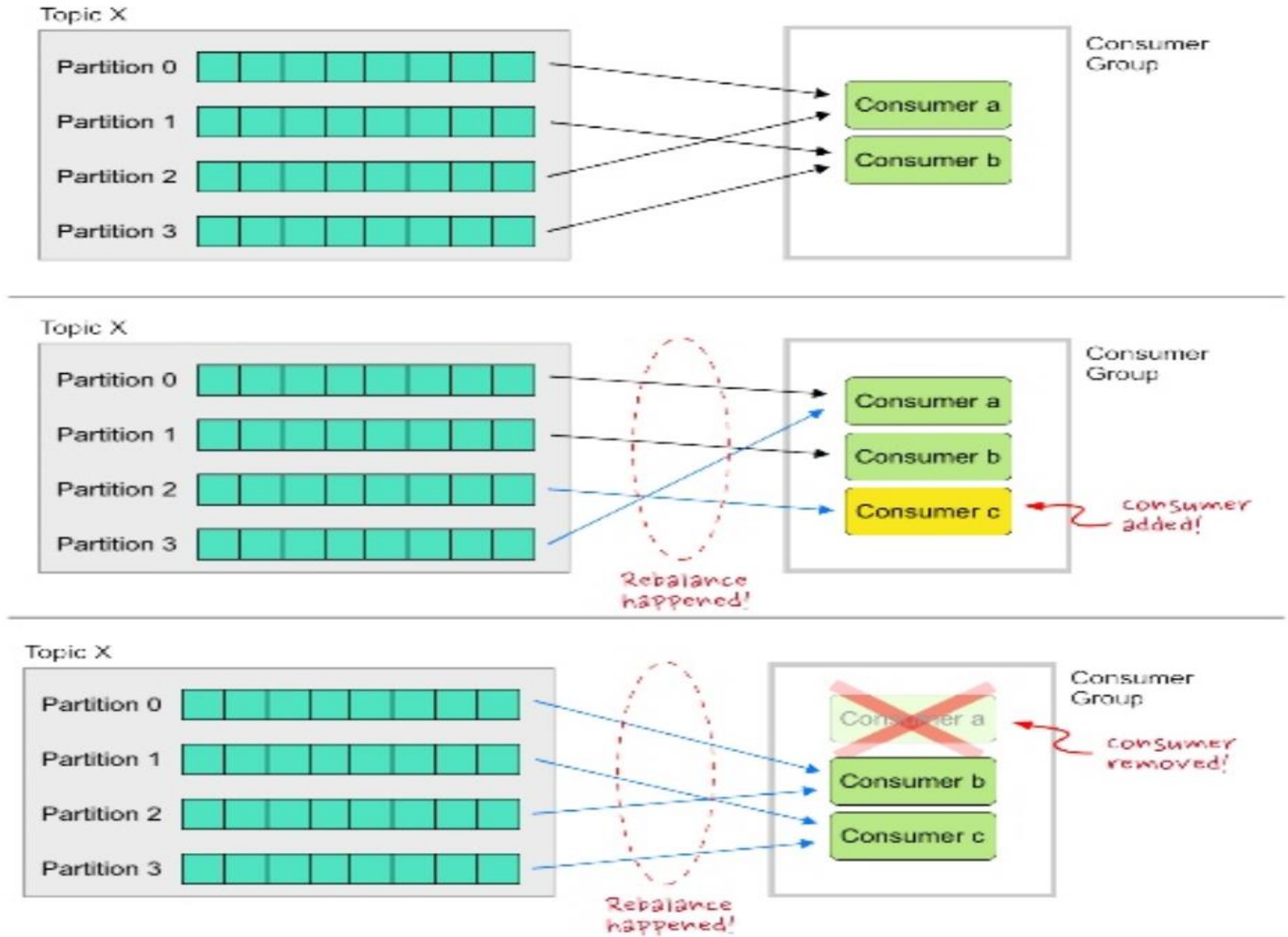





If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all.


Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1
Consumer 2
Consumer 3
Consumer 4
Consumer Group 1
Consumer 1
Consumer 2

# Consumer Rebalances



**Consumer can consume  data from Kafka:**

- ✓ Reading messages from the beginning of the topic (latest), from the end (earliest),
- ✓ from where we stopped before (committed offsets).
- ✓ Look for offsets based on timestamps

**Delivery semantics for consumers**

Consumers choose when to commit offsets

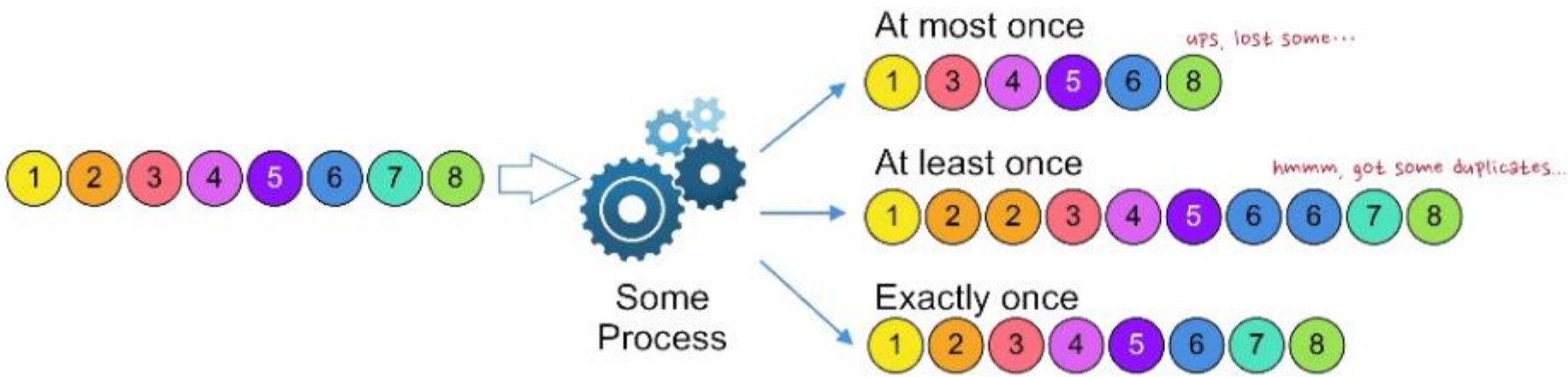
- At most once
- At least once (usually preferred)
- Exactly once (transactional)

**At most once:**
offsets are committed as soon as the message is received
if the processing goes wrong, the message will be lost

**At least once:**
offsets are committed after the message is processed if the processing goes wrong, the message will be read again this can result in duplicate processing of messages.
use idempotent consumers.

**Exactly once:**
Not supported in Consumer API
However, we can be achieved for Kafka => kafka workflows using kafka streams api

# Delivery Guarantees



**At-most-once Kafka Consumer (Zero or one deliveries)**
At-most-once consumer is the default behavior of a KAFKA consumer.

To configure this type of consumer,

- ❑ Set ‘enable.auto.commit’ to true
- ❑ Set ‘auto.commit.interval.ms’ to a lower timeframe.
- ❑ And do not make call to consumer.commitSync(); from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

**At-Least-Once Kafka Consumer** (One or more message deliveries, duplicate possible)

To configure this type of consumer:

- Set ‘enable.auto.commit’ to false

- Consumer should then take control of the message offset commits to Kafka by making the following call consumer.commitSync()

auto.offset.reset

**earliest**: automatically reset the offset to the earliest offset

**latest**: automatically reset the offset to the latest offset

**none**: throw exception to the consumer if no previous offset is found for the consumer's group

## Consumer Groups and Topic Subscriptions

Kafka uses the concept of consumer groups to allow a pool of processes to divide the work of consuming and processing records.

These processes can either be running on the same machine or they can be distributed over many machines to provide scalability and fault tolerance for processing.

All consumer instances sharing the same group.id will be part of the same consumer group.

It is also possible that the consumer could encounter a "livelock" situation where it is continuing to send heartbeats, but no progress is being made.

To prevent the consumer from holding onto its partitions indefinitely in this case, we provide a liveness detection mechanism using the max.poll.interval.ms setting.

The consumer provides two configuration settings to control the behavior of the poll loop:

**max.poll.interval.ms:**

By increasing the interval between expected polls, we can give the consumer more time to handle a batch of records returned from poll(long). The drawback is that increasing this value may delay a group rebalance since the consumer will only join the rebalance inside the call to poll. You can use this setting to bound the time to finish a rebalance, but you risk slower progress if the consumer cannot actually call poll often enough.

**max.poll.records:**

Use this setting to limit the total records returned from a single call to poll. This can make it easier to predict the maximum that must be handled within each poll interval. By tuning this value, you may be able to reduce the poll interval, which will reduce the impact of group rebalancing.

**heartbeat.interval.ms**

The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group.

Default : 3000

**Manual Partition Assignment**

In some cases we may need finer control over the specific partitions that are assigned. For example:

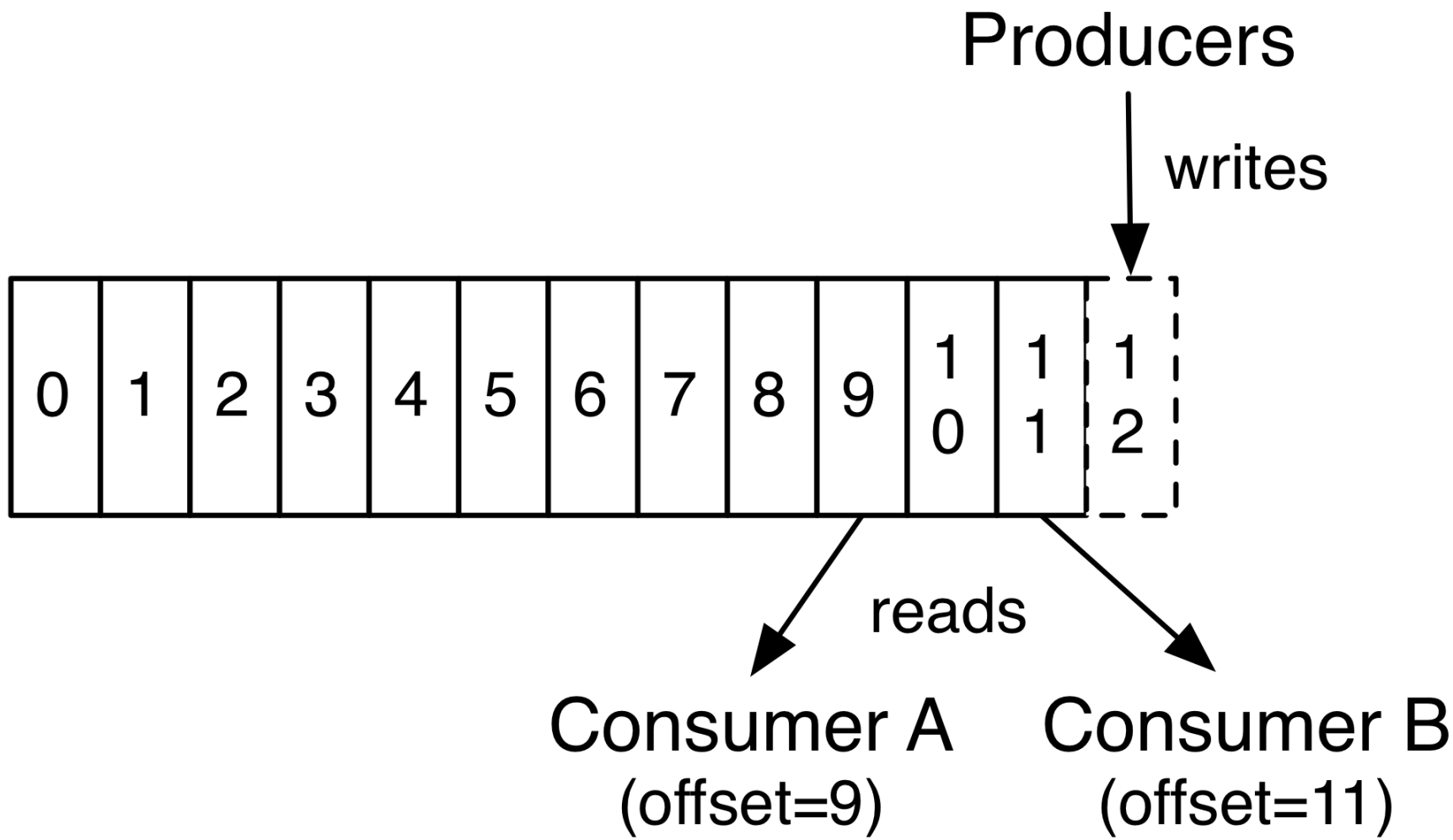
- If the process is maintaining some kind of local state associated with that partition (like a local on-disk key-value store), then it should only get records for the partition it is maintaining on disk.

- If the process itself is **highly available** and will be restarted if it fails (perhaps using a cluster management framework like YARN, Mesos, or AWS facilities, or as part of a stream processing framework).

- In this case there is no need for Kafka to detect the failure and reassign the partition since the consuming process will be restarted on another machine.

```
String topic = "foo";
TopicPartition partition0 = new TopicPartition(topic, 0);
TopicPartition partition1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(partition0, partition1));
```

# Reprocess of the Failed Message & Message Offset management.


Producers
writes
0
1
2
3
4
5
6
7
8
9
10
11
12
reads
Consumer A
(offset=9)
Consumer B
(offset=11)

This offset is controlled by the consumer:

Normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes.

For example a consumer can reset to an older offset to **reprocess data** from the past or skip ahead to the most recent record and start consuming from "now".

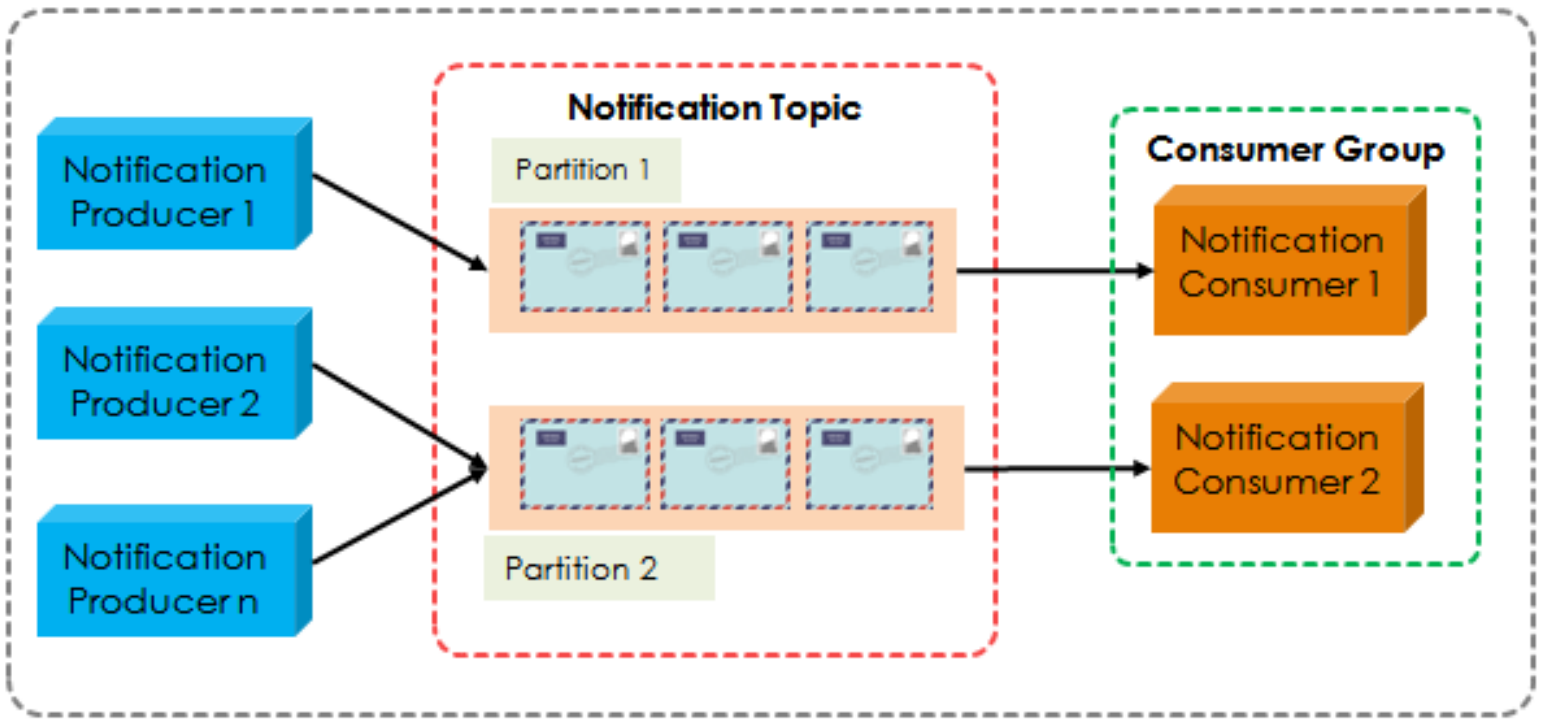
**Controlling The Consumer's Position**

Kafka allows specifying the position using

seek(TopicPartition, long) to specify the new position.

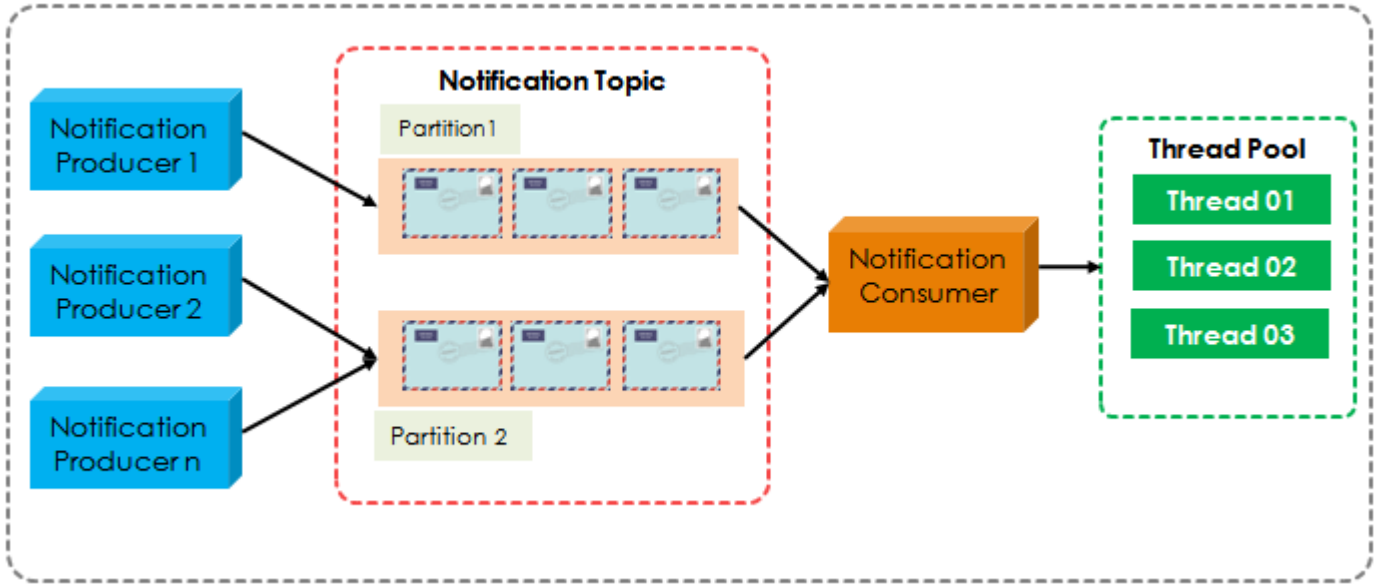
Special methods for seeking to the earliest and latest offset the server maintains are also available ( seekToBeginning(Collection) and seekToEnd(Collection) respectively).

Multiple consumers with their own threads



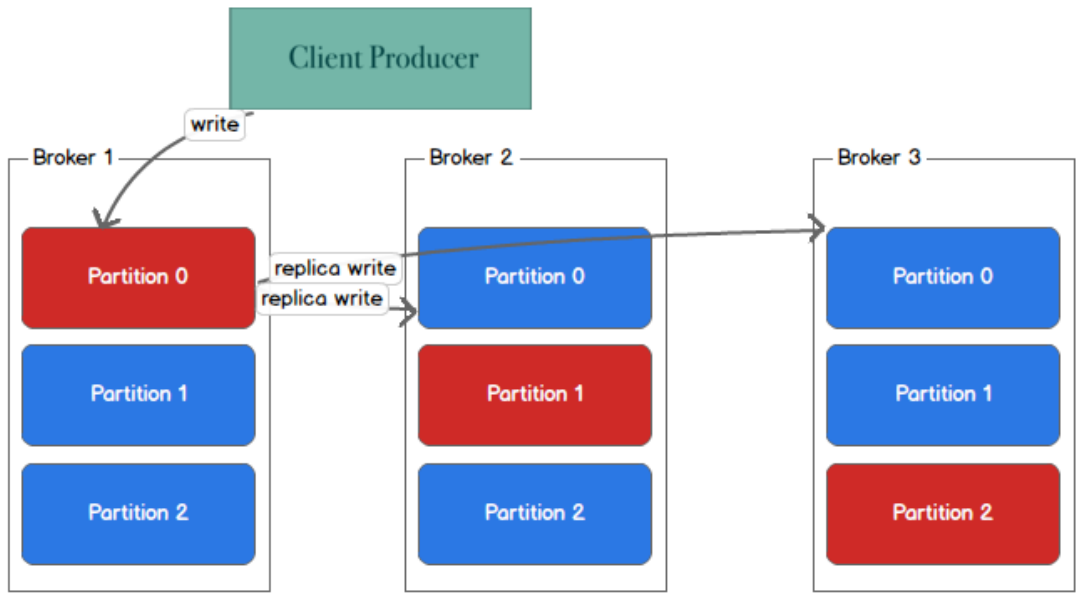
| Pros | Cons |
| --- | --- |
| Easy to implement | The total of consumers is limited the total partitions of the topic. |
| Implementing in-order processing on per-partition is easier. | More TCP connections to the brokers |

Single consumer, multiple worker processing threads



| Pros | Cons |
| --- | --- |
| Be flexible in scale out the number of processing thread | It’s not easy to implement in-order processing on per partition. Let’s say there are 2 messages on the same partitions being processed by 2 different threads. To guarantee the order, those 2 threads must be coordinated somehow. |

# Kafka Topic Architecture - Replication, Failover, and Parallel Processing




Leader (red)
replicas (blue)

Record is considered "committed" when all ISRs for partition wrote to their log.

**Only committed records are readable from consumer**

- The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions.

- Each partition is replicated across a configurable number of servers for fault tolerance.

- Each partition has one server which acts as the "leader" and zero or more servers which act as "followers".

- The leader handles all read and write requests for the partition while the followers passively replicate the leader.
- If the leader fails, one of the followers will automatically become the new leader.
- Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

**auto.leader.rebalance.enable (default: true)**

Enables auto leader balancing. A background thread checks the distribution of partition leaders at regular intervals, configurable by `leader.imbalance.check.interval.seconds`.

If the leader imbalance exceeds `leader.imbalance.per.broker.percentage`(default: 10), leader rebalance to the preferred leader for partitions is triggered.


confluent

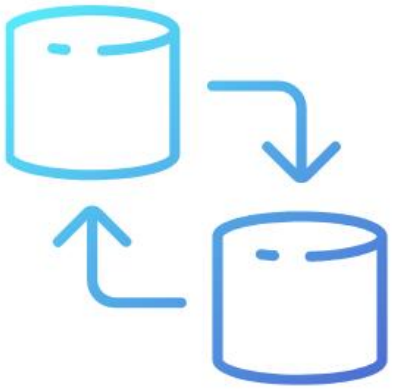
- Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day.

- Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log.

- Since being created and open sourced by LinkedIn in 2011, Kafka has quickly evolved from messaging queue to a full-fledged event streaming platform.

**A streaming platform has three key capabilities:**

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.



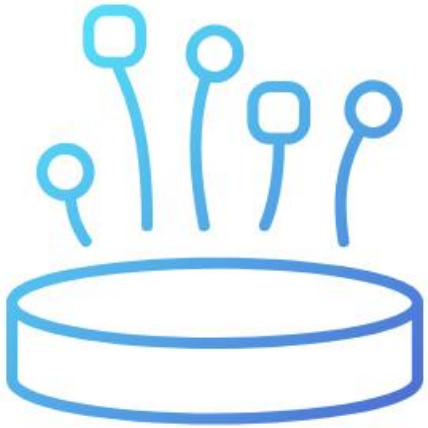
**Publish + Subscribe**

It is immutable commit log, and from there we can subscribe to it, and publish data to any number of systems or real-time applications.

Unlike messaging queues, Kafka is a highly scalable, fault tolerant distributed system, allowing it to be deployed for applications like:

- ❑ managing passenger and driver matching at Uber,
- ❑ providing real-time analytics and predictive maintenance for British Gas' smart home,
- ❑ and performing numerous real-time services across all of LinkedIn.

This unique performance makes it perfect to scale from one app to company-wide use.



**Store**

An abstraction of a distributed commit log commonly found in distributed databases(Oracle SGA), Apache Kafka provides durable storage.

Kafka can act as a ‘source of truth’, being able to distribute data across multiple nodes for a highly available deployment within a single data center or across multiple availability zones.

* single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is stored exactly once.

## Process

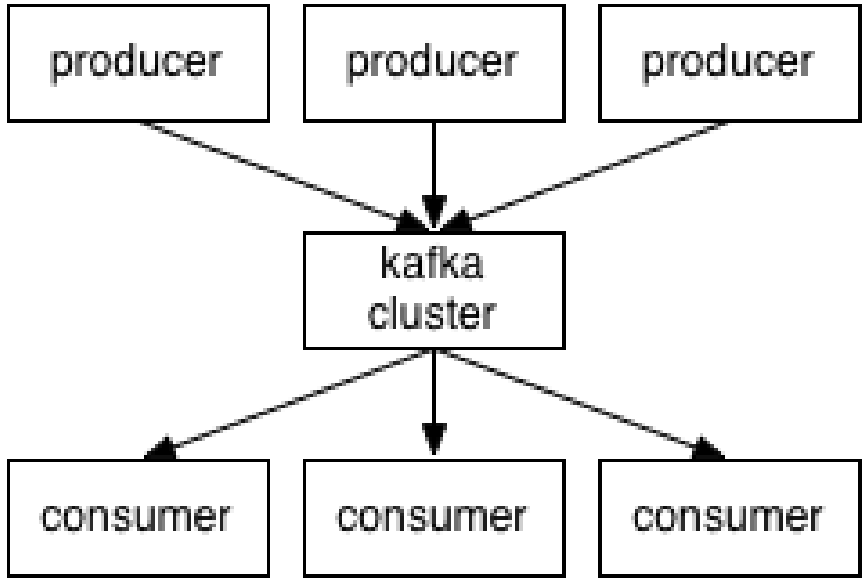
An event streaming platform would not be complete without the ability to manipulate that data as it arrives.

The Streams API within Apache Kafka is a powerful, lightweight library that allows for on-the-fly processing, letting you aggregate, create windowing parameters, perform joins of data within a stream, and more.

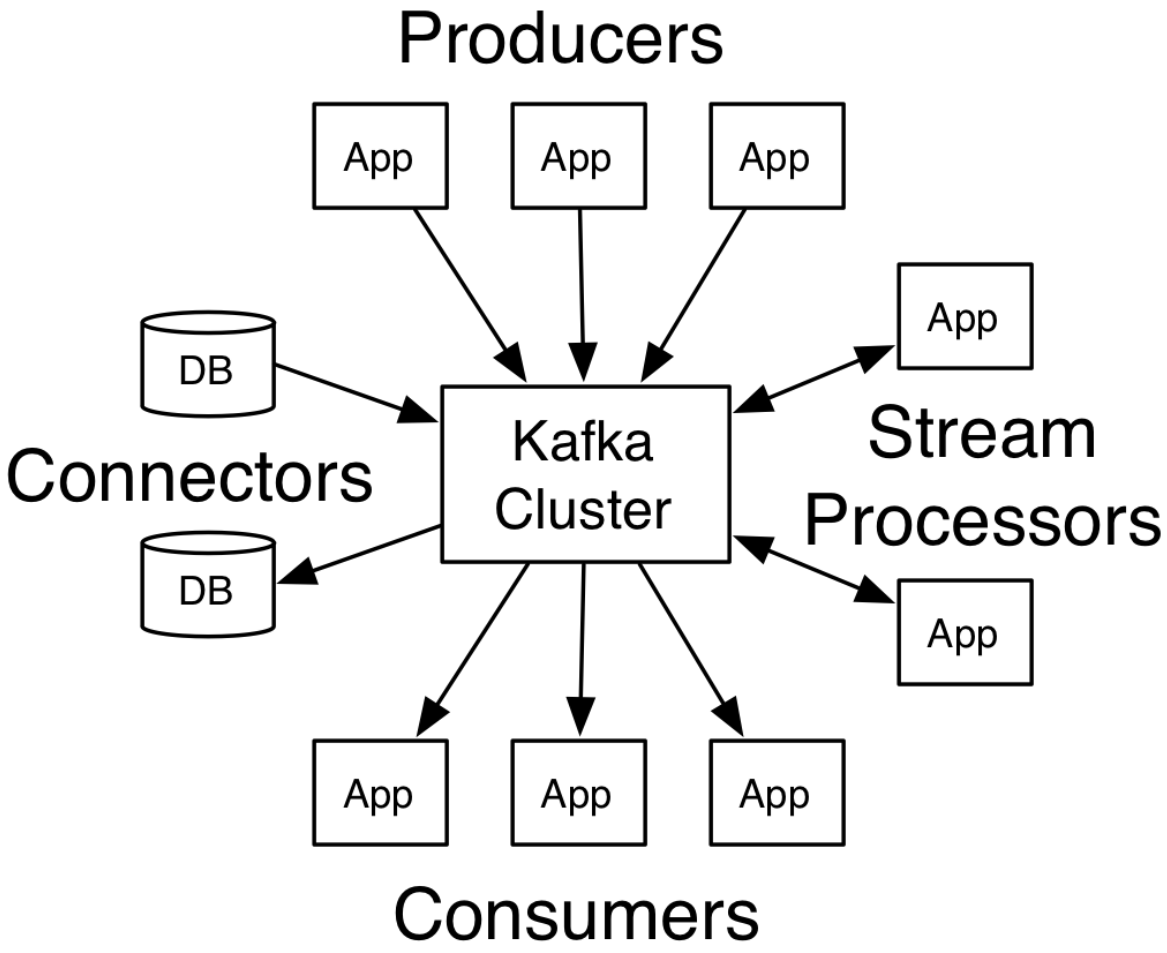
Perhaps best of all, it is built as a Java application on top of Kafka, keeping your workflow intact with no extra clusters to maintain.

**Some basic messaging terminology:**

- Kafka maintains feeds of messages in categories called **topics**.
- **Producers** publish the messages to a Kafka topic.
- **Consumers** will subscribe to topics and process the messages.
- **Kafka cluster** consists of one or more servers (Kafka brokers).



**Kafka has four core APIs:**



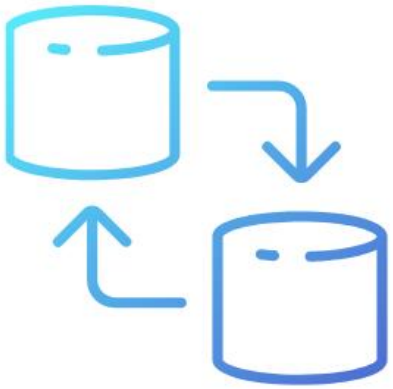
- Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day.

- Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log.

- Since being created and open sourced by LinkedIn in 2011, Kafka has quickly evolved from messaging queue to a full-fledged event streaming platform.

**A streaming platform has three key capabilities:**

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.



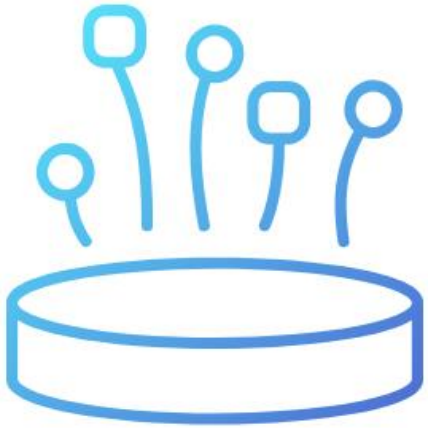
**Publish + Subscribe**

It is immutable commit log, and from there we can subscribe to it, and publish data to any number of systems or real-time applications.

Unlike messaging queues, Kafka is a highly scalable, fault tolerant distributed system, allowing it to be deployed for applications like:

- ❑ managing passenger and driver matching at Uber,
- ❑ providing real-time analytics and predictive maintenance for British Gas' smart home,
- ❑ and performing numerous real-time services across all of LinkedIn.

This unique performance makes it perfect to scale from one app to company-wide use.



**Store**

An abstraction of a distributed commit log commonly found in distributed databases(Oracle SGA), Apache Kafka provides durable storage.

Kafka can act as a ‘source of truth’, being able to distribute data across multiple nodes for a highly available deployment within a single data center or across multiple availability zones.

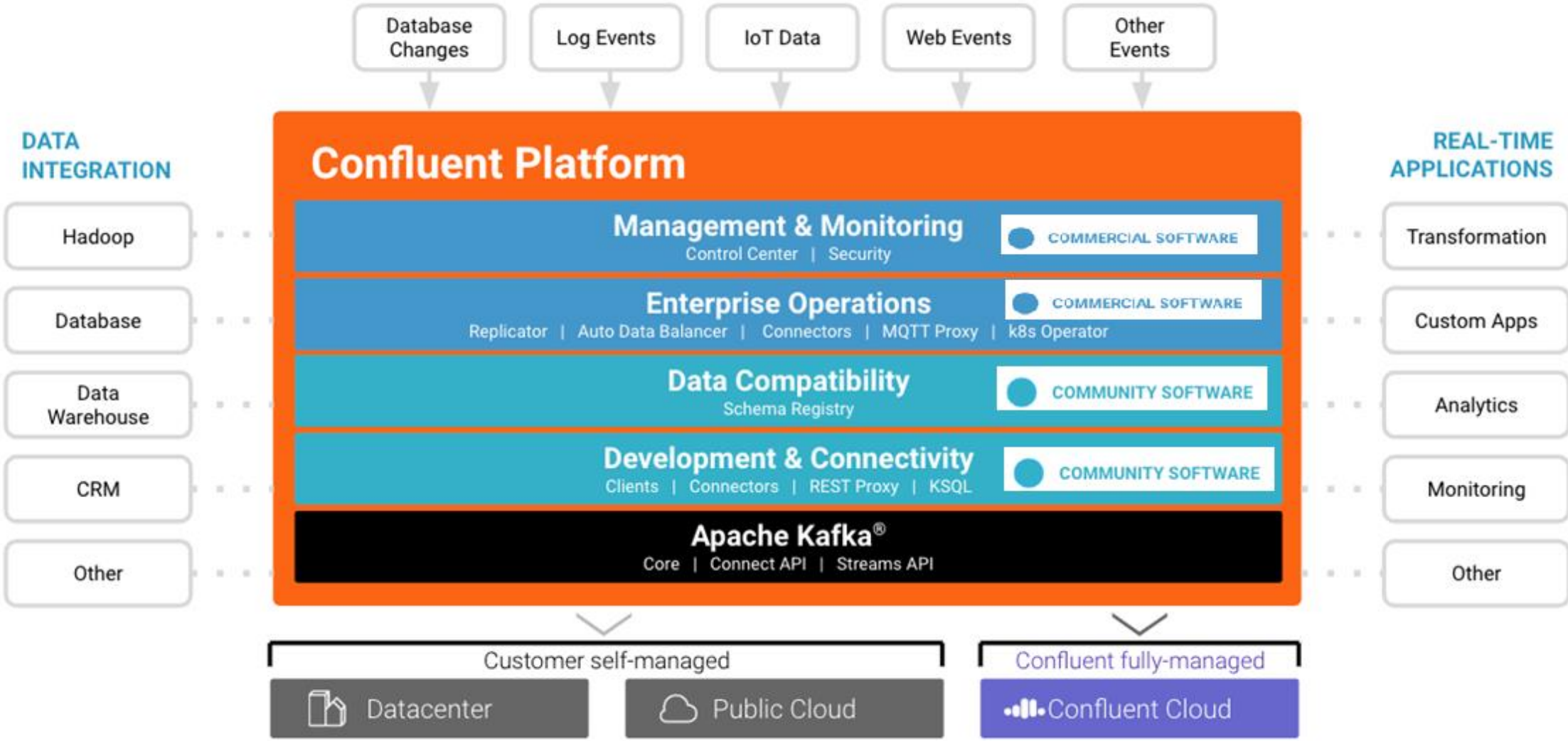
* single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is stored exactly once.

## Process

An event streaming platform would not be complete without the ability to manipulate that data as it arrives.

The Streams API within Apache Kafka is a powerful, lightweight library that allows for on-the-fly processing, letting you aggregate, create windowing parameters, perform joins of data within a stream, and more.

Perhaps best of all, it is built as a Java application on top of Kafka, keeping your workflow intact with no extra clusters to maintain.


Database Changes
Log Events
IoT Data
Web Events
Other Events
DATA INTEGRATION
Hadoop
Database
Data Warehouse
CRM
Other
Confluent Platform
Management & Monitoring
Control Center | Security
COMMERCIAL SOFTWARE
Enterprise Operations
Replicator | Auto Data Balancer | Connectors | MQTT Proxy | k8s Operator
COMMERCIAL SOFTWARE
Data Compatibility
Schema Registry
COMMUNITY SOFTWARE
Development & Connectivity
Clients | Connectors | REST Proxy | KSQL
COMMUNITY SOFTWARE
Apache Kafka®
Core | Connect API | Streams API
REAL-TIME APPLICATIONS
Transformation
Custom Apps
Analytics
Monitoring
Other
Customer self-managed
Datacenter
Public Cloud
Confluent fully-managed
Confluent Cloud

**Confluent Platform** includes Apache Kafka, and also few things that can make Apache Kafka easier to use:

**Clients** in Python, C, C++ and Go. Apache Kafka includes Java client. If you use a different language, Confluent Platform may include a client you can use.

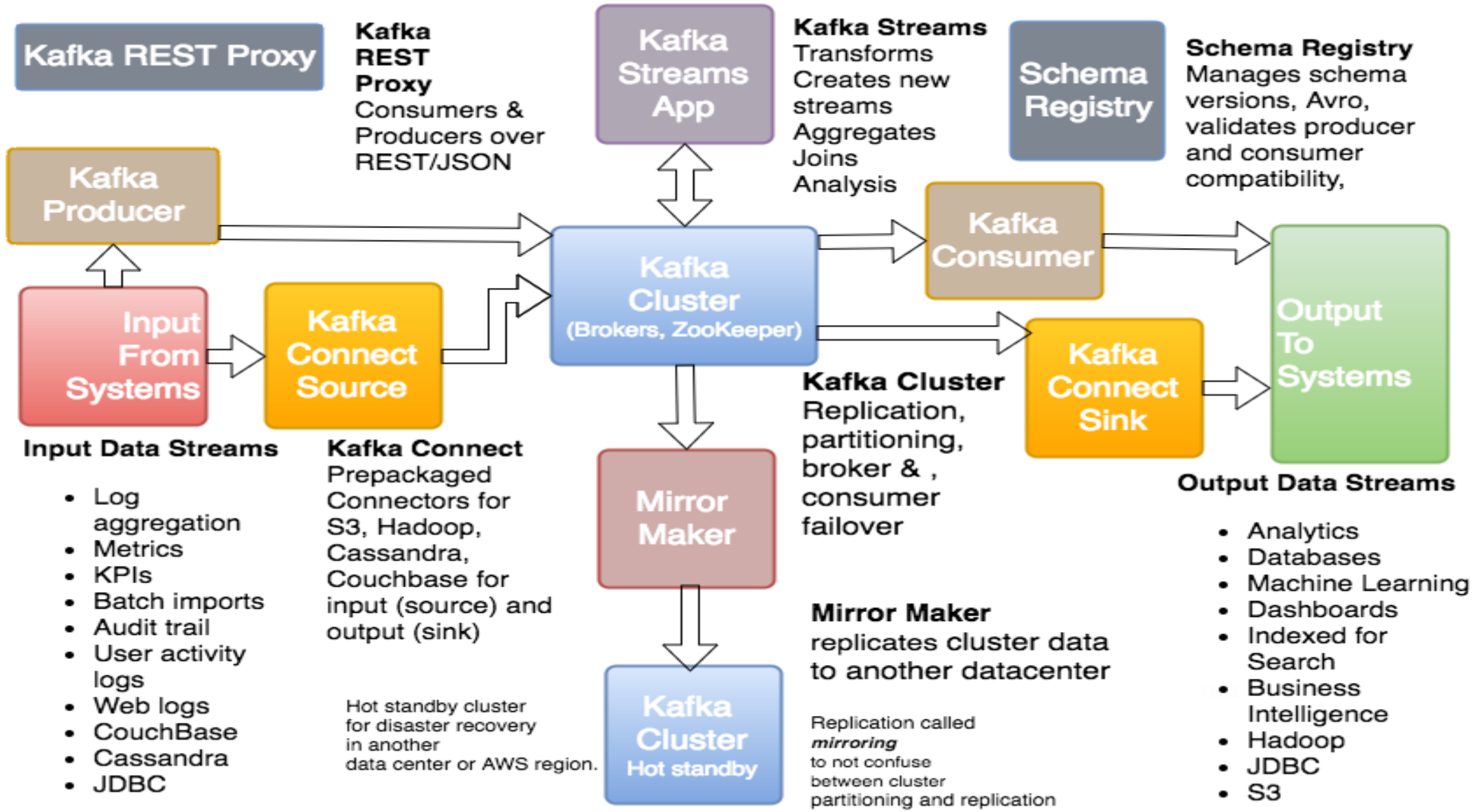
**Connectors** - Apache Kafka include a file connector. Confluent adds HDFS, JDBC and Elastic Search connectors.

**REST Proxy** - adds a REST API to Apache Kafka, so you can use it in any language or even from your browser

**Schema Registry** - if you use Avro, the schema registry will store the Avro schemas for each topic for you and help with schema evolution and compatibility.

**Support** - Confluent Platform is supported by Confluent. Apache Kafka on its own is not really supported by anyone (other vendors package it with their own platforms and support their own platforms, just like Confluent supports Kafka in the Confluent Platform

| Component | Port |
| --- | --- |
| Kafka brokers (plain text) | 9092 |
| Kafka inter-broker communication | 9091 |
| Confluent Control Center | 9021 |
| Kafka Connect REST API | 8083 |
| ksqlDB Server REST API | 8088 |
| Metadata Service (MDS) | 8090 |
| REST Proxy | 8082 |
| Schema Registry REST API | 8081 |
| ZooKeeper | 2181, 2888, 3888 |


Kafka REST Proxy
Kafka REST Proxy
Consumers & Producers over REST/JSON
Kafka Streams App
Kafka Streams
Transforms
Creates new streams
Aggregates
Joins
Analysis
Schema Registry
Schema Registry
Manages schema versions, Avro, validates producer and consumer compatibility,
Kafka Producer
Kafka Cluster
(Brokers, ZooKeeper)
Kafka Consumer
Input From Systems
Kafka Connect Source
Kafka Connect Sink
Output To Systems
Input Data Streams
Log aggregation
Metrics
KPIs
Batch imports
Audit trail
User activity logs
Web logs
CouchBase
Cassandra
JDBC
Kafka Connect
Prepackaged Connectors for S3, Hadoop, Cassandra, Couchbase for input (source) and output (sink)
Kafka Cluster
Replication, partitioning, broker & , consumer failover
Mirror Maker
Mirror Maker
replicates cluster data to another datacenter
Kafka Cluster
Hot standby
Hot standby cluster for disaster recovery in another data center or AWS region.
Replication called mirroring to not confuse between cluster partitioning and replication
Output Data Streams
Analytics
Databases
Machine Learning
Dashboards
Indexed for Search
Business Intelligence
Hadoop
JDBC
S3

At the **core of Confluent Platform** is Apache Kafka, the most popular open source distributed streaming platform

The key components of the Kafka open source project are Kafka Brokers and Kafka Java Client APIs.

Kafka Java Client APIs includes:

**Producer API** is a Java Client that allows an application to publish a stream records to one or more Kafka topics.

**Consumer API** is a Java Client that allows an application to subscribe to one or more topics and process the stream of records produced to them.

**Streams API** allows applications to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

**Connect API** is a component that you can use to stream data between Kafka and other data systems in a scalable and reliable way. It makes it simple to configure connectors to move data into and out of Kafka.

Example : ETL based applications

# Use Case Examples

The Kafka Streams API is applicable to a wide range of use cases and industries.

- Travel companies can build applications with the Kafka Streams API to make real-time decisions to find best suitable pricing for individual customers, to cross-sell additional services, and to process bookings and reservations.
- The finance industry can build applications to aggregate data sources for real-time views of potential exposures and for detecting and minimizing fraudulent transactions.
- Logistics companies can build applications to track their shipments fast, reliably, and in real-time.
- Retailers can build applications to decide in real-time on next best offers, personalized promotions, pricing, and inventory management.
- Automotive and manufacturing companies can build applications to ensure their production lines perform optimally, to gain real-time insights into their supply chains, and to monitor telemetry data from connected cars to decide if an inspection is needed.
- And many more.

## Commercial Features:

### Confluent Control Center

Confluent Control Center is a GUI-based system for managing and monitoring Kafka.

### Confluent Replicator

Confluent Platform makes it easier than ever to maintain multiple Kafka clusters in multiple data centers.

### Confluent Auto Data Balancer

As clusters grow, topics and partitions grow at different rates, brokers are added and removed and over time this leads to unbalanced workload across datacenter resources.

**Confluent JMS Client**

Confluent Platform includes a JMS-compatible client for Kafka. This Kafka client implements the JMS 1.1 standard API, using Kafka brokers as the backend.

**Confluent MQTT Proxy**

Message Queuing Telemetry Transport

Provides bidirectional access to Kafka from MQTT devices and gateways without the need for a MQTT Broker in the mind

**Confluent Security Plugins**

Confluent Security Plugins are used to add security capabilities to various Confluent Platform tools and products

## Community Features

## Confluent KSQL

Confluent KSQL is the streaming SQL engine for Kafka.

It provides an easy-to-use yet powerful interactive SQL interface for stream processing on Kafka, without the need to write code in a programming language such as Java or Python

## Confluent Connectors

Connectors leverage the Kafka Connect API to connect Kafka to other data systems such as Apache Hadoop.

Confluent Platform provides connectors for the most popular data sources and sinks, and include fully tested and supported versions of these connectors with Confluent Platform.

The available connectors include JDBC, HDFS, Elasticsearch, and S3.

## Confluent Clients

C/C++ Client Library, Python Client Library, Go Client Library, .Net Client Library

## Confluent Schema Registry

One of the most difficult challenges with loosely coupled systems is ensuring compatibility of data and code as the system grows and evolves.

With a messaging service like Kafka, services that interact with each other must agree on a common format, called a schema, for messages.

## Confluent REST Proxy

Kafka and Confluent provide native clients for Java, C, C++, and Python that make it fast and easy to produce and consume messages through Kafka.

But sometimes, it isn't practical to write and maintain an application that uses the native clients.

The Confluent REST Proxy makes it easy to work with Kafka from any language by providing a RESTful HTTP service for interacting with Kafka clusters.

# Confluent Platform – Quick Start

Java 1.8 is supported in this version of Confluent Platform (Java 1.9 and 1.10 are currently not supported).

We should run with the Garbage-First (G1) garbage collector.

```
Confluent
  |_kafka-server-start
       |_exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@“
            |_# JVM performance options
                    if [ -z "$KAFKA_JVM_PERFORMANCE_OPTS" ]; then
                    KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseG1GC
                    -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
                    -XX:+ExplicitGCInvokesConcurrent -Djava.awt.headless=true“
                    fi
```

Step 1: Download and Start Confluent Platform
Go to the downloads page and choose Confluent Platform.

Step 2: Decompress the file. You should have these directories:

| Folder | Description |
| --- | --- |
| /bin/ | Driver scripts for starting and stopping services |
| /etc/ | Configuration files |
| /lib/ | Systemd services |
| /logs/ | Log files |
| /share/ | Jars and licenses |
| /src/ | Source files that require a platform-dependent build |

3. Add the install location of the Confluent bin directory to your PATH:

export PATH=<path-to-confluent>/bin:$PATH

4. Install the Kafka Connect Datagen source connector using the Confluent Hub client. This connector generates mock data for demonstration purposes and is not suitable for production. Confluent Hub is an online library of pre-packaged and ready-to-install extensions or add-ons for Confluent Platform and Apache Kafka.

<path-to-confluent>/bin/confluent-hub install \
--no-prompt confluentinc/kafka-connect-datagen:0.1.0

5. Start Confluent Platform using the Confluent CLI start command.

<path-to-confluent>/bin/confluent start

6. Navigate to the Control Center web interface at http://localhost:9021/.

# Kafka Connect

**Kafka Connect** is part of Apache Kafka, providing streaming integration between data stores and Kafka.

For data engineers, it just requires JSON configuration files to use. There are connectors for common  data stores like JDBC, Elasticsearch, Neo4j etc.,

For developers, Kafka Connect has a rich API in which additional connectors can be developed if required. In addition to this, it also has a REST API for configuration and management of connectors.

**Kafka Connect  provides** a very powerful way of handling integration requirements. Some key components include:

**Connectors –** the JAR files that define how to integrate with the data store itself.

**Converters –** handling serialization and deserialization of data.

**Transforms –** optional in-flight manipulation of messages.

**Common Kafka Use Cases:**

Source => Kafka [Producer API] Kafka Connect Source

Kafka => Kafka [Consumer, Producer API] Kafka Streams

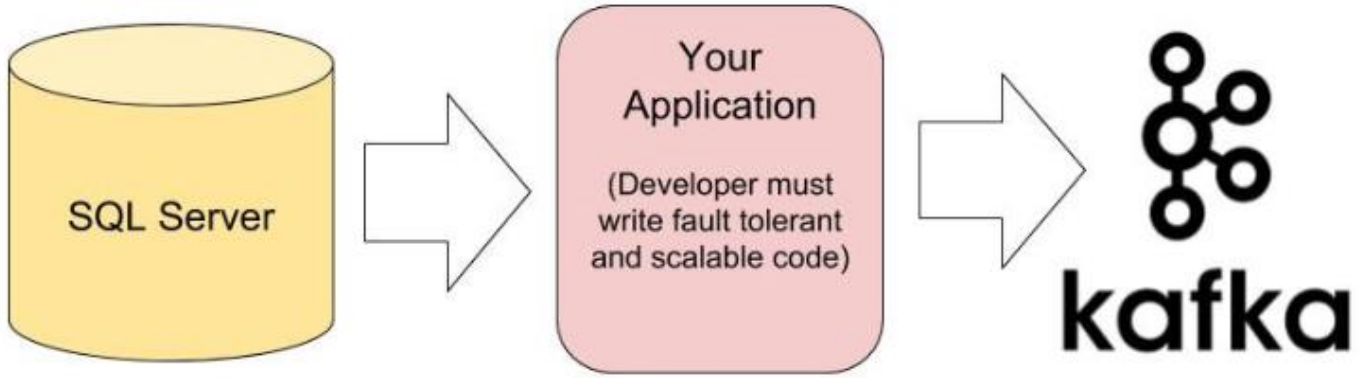
Kafka => Sink [Consumer API] Kafka Connect Sink

Kafka => App [Consumer API]

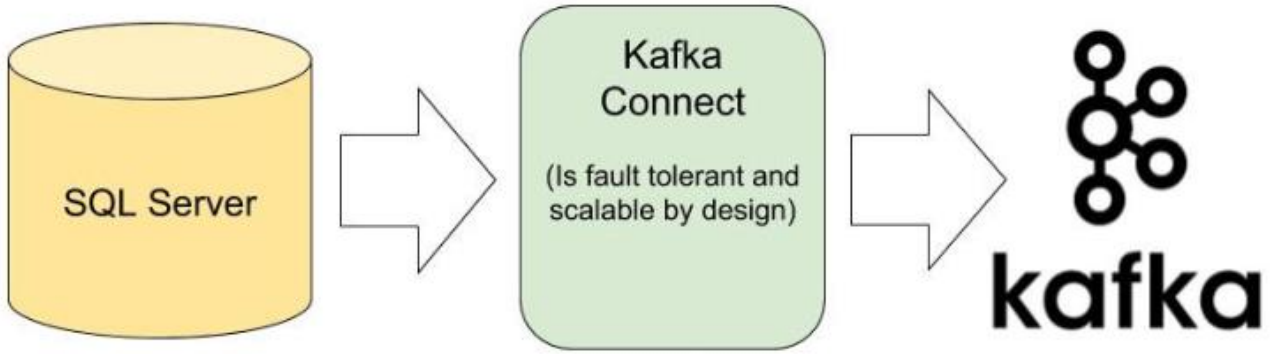
App => Kafka [Producer API]

## Kafka Connect Simplify and improve getting data in and out of Kafka

write codes that move your data to the Kafka cluster




move data without writing any code



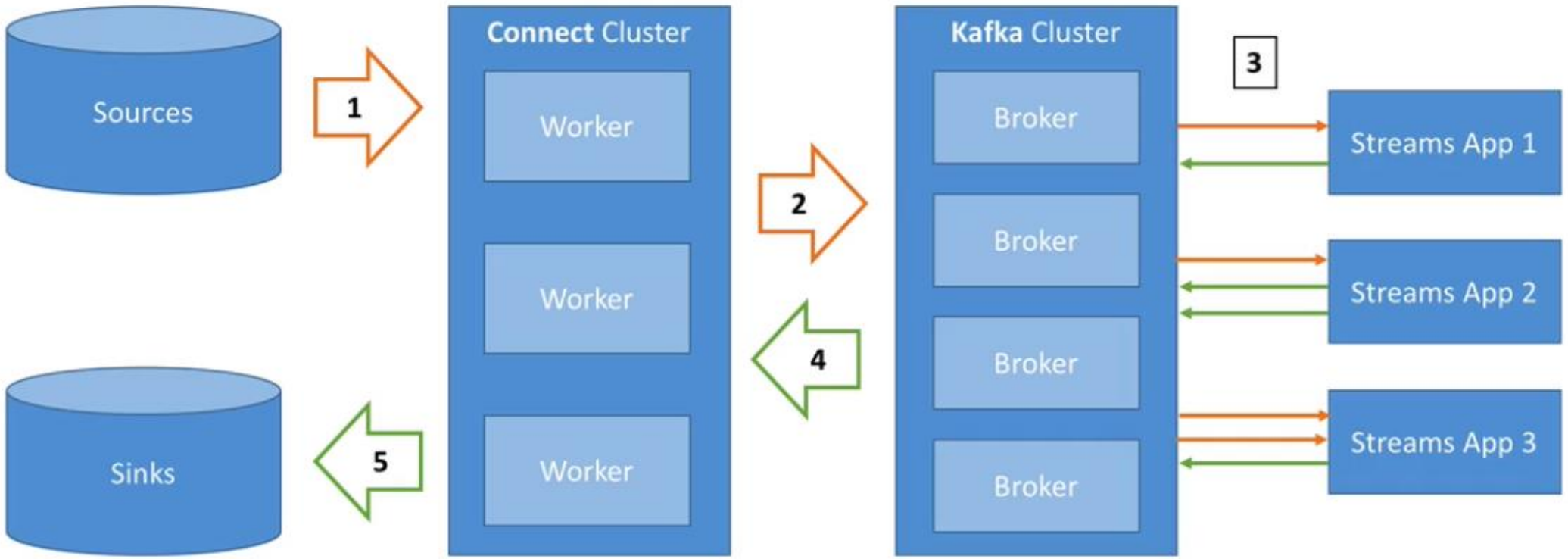
## Kafka Connect?

We want to import data from the source location:
JDBC, Cassandra, MongoDB, Twitter, FTP etc.,

We want to store data in to the sink location:
JDBC, Cassandra, MongoDB, Twitter, FTP, Splunk, Redis etc.,

It is difficult to achieve Fault Tolerance, Exactly Once Delivery, Distribution, Ordering.

This is where Kafka Connect helps use with pre-defined connectors for source and sink locations.




Kafka Connect And Streams Architecture

**Kafka Connect** is a framework to stream data into and out of Apache Kafka.

The **Confluent Platform** ships with several built-in connectors that can be used to stream data to or from commonly used systems such as relational databases or HDFS.

## Kafka Connect

Kafka Connect has multiple loaded **Connectors**
- Each connector is a re-usable piece of code (java jars)
- Many connectors exist in the open souce

Connectors + User configuration => **Tasks**
- A task is linked to a connector configuration
- A job configuration may spawn multiple tasks

Tasks are executed by Kafka Connect **Workers** (servers)
- A worker is a single java process
- A worker can be standalone or in a cluster

**Connectors** -- the high level abstraction that coordinates data streaming by managing tasks.

**Tasks** -- the implementation of how data is copied to or from Kafka

**Workers** -- the running processes that execute connectors and tasks

**Converters** -- the code used to translate data between Connect and the system sending or receiving data

**Transforms** -- simple logic to alter each message produced by or sent to a connector

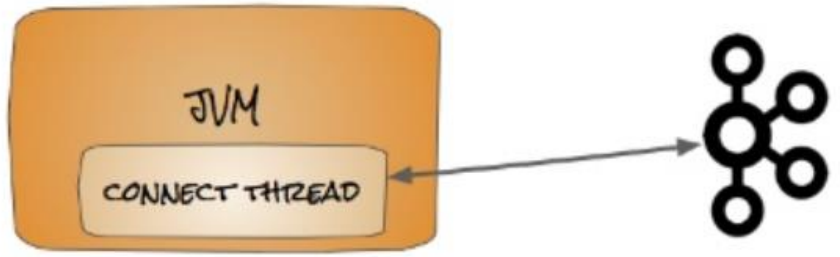
**Workers**

Connectors and tasks are logical units of work and must be scheduled to execute in a process.

Kafka Connect calls these processes workers and has two types of workers: standalone and distributed.

**Standalone**:

- A single process runs the connectors and tasks.
- Configuration is bundled with the process.
- Useful for development and testing
- Not fault tolerant, no scalability, hard to monitor.



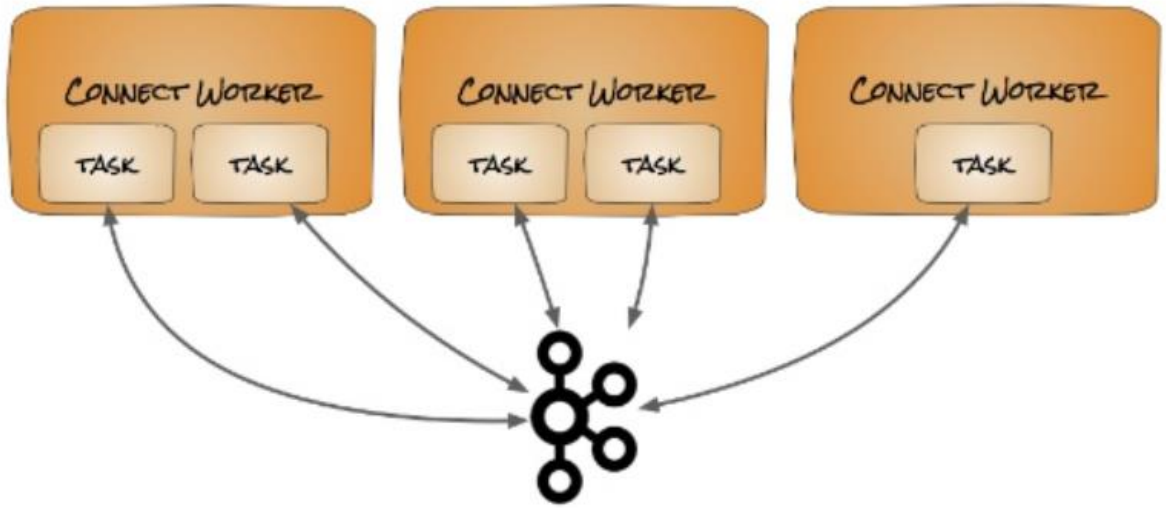
## Standalone Mode

Standalone mode is typically used for development and testing, or for lightweight, single-agent environments

./connect-standalone worker.properties connector1.properties [connector2.properties connector3.properties ...]

**Distributed**:

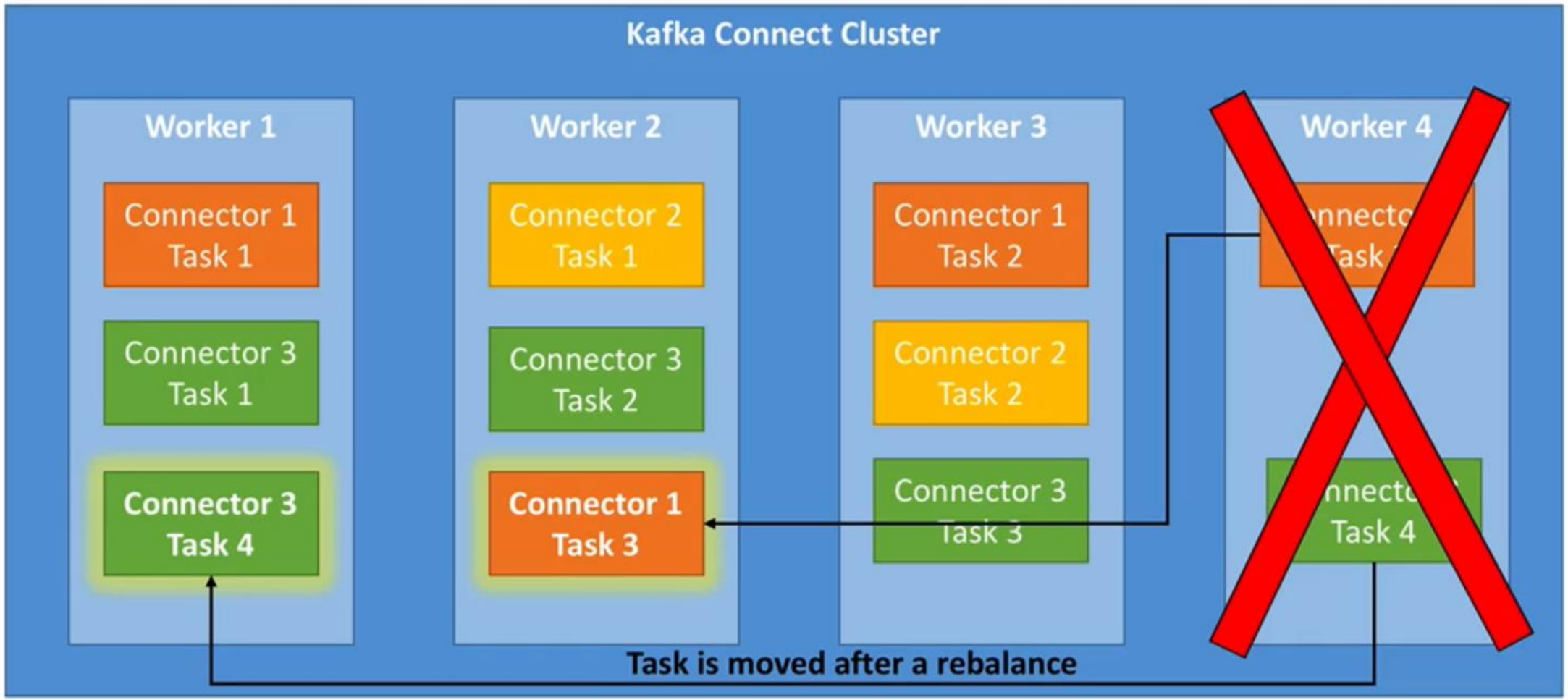
- Multiple workers run the connectors and tasks
- Configuration is submitted using a REST API
- Easy to scale and fault tolerant (rebalancing in case a worker dies)
- Useful for production deployments

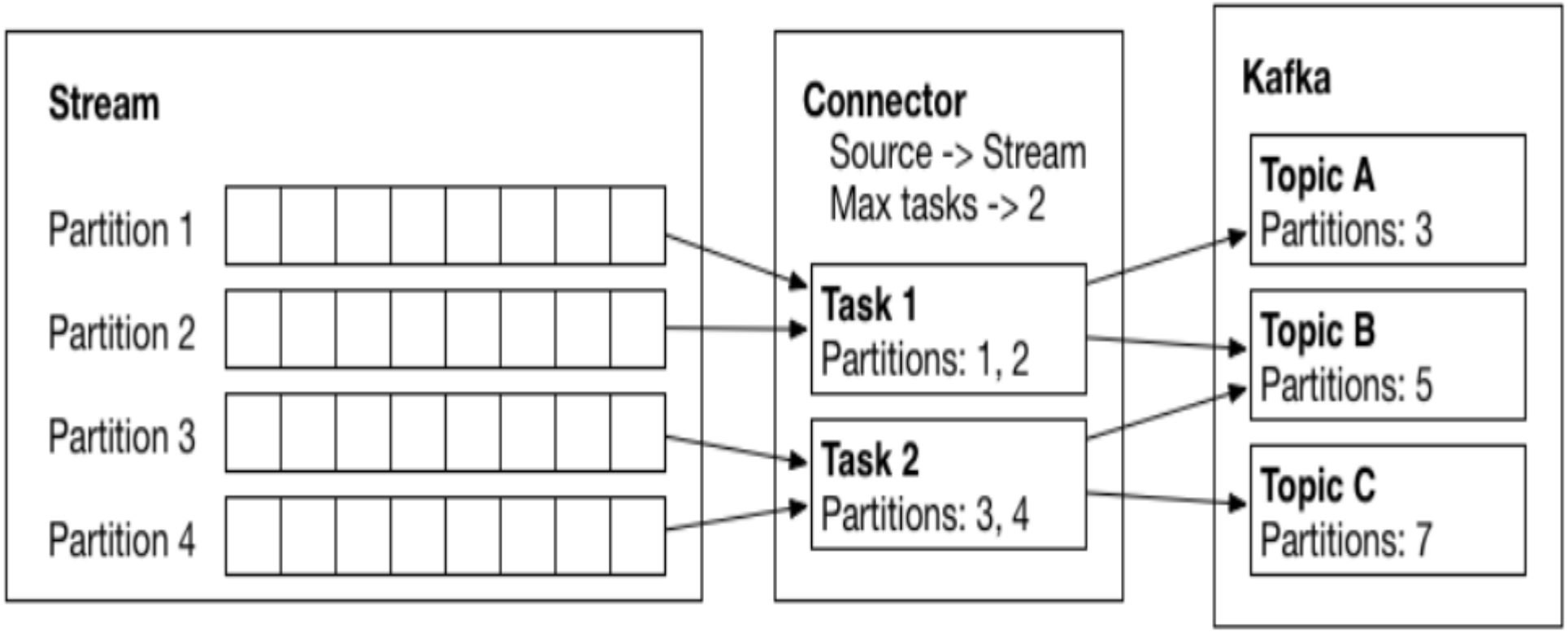


## Distributed Mode

Connect stores connector and task configurations, offsets, and status in several Kafka topics.

bin/connect-distributed worker.properties


Kafka Connect Cluster
Worker 1
Connector 1 Task 1
Connector 3 Task 1
Connector 3 Task 4
Worker 2
Connector 2 Task 1
Connector 3 Task 2
Connector 1 Task 3
Worker 3
Connector 1 Task 2
Connector 2 Task 2
Connector 3 Task 3
Worker 4
Task 4
Task is moved after a rebalance



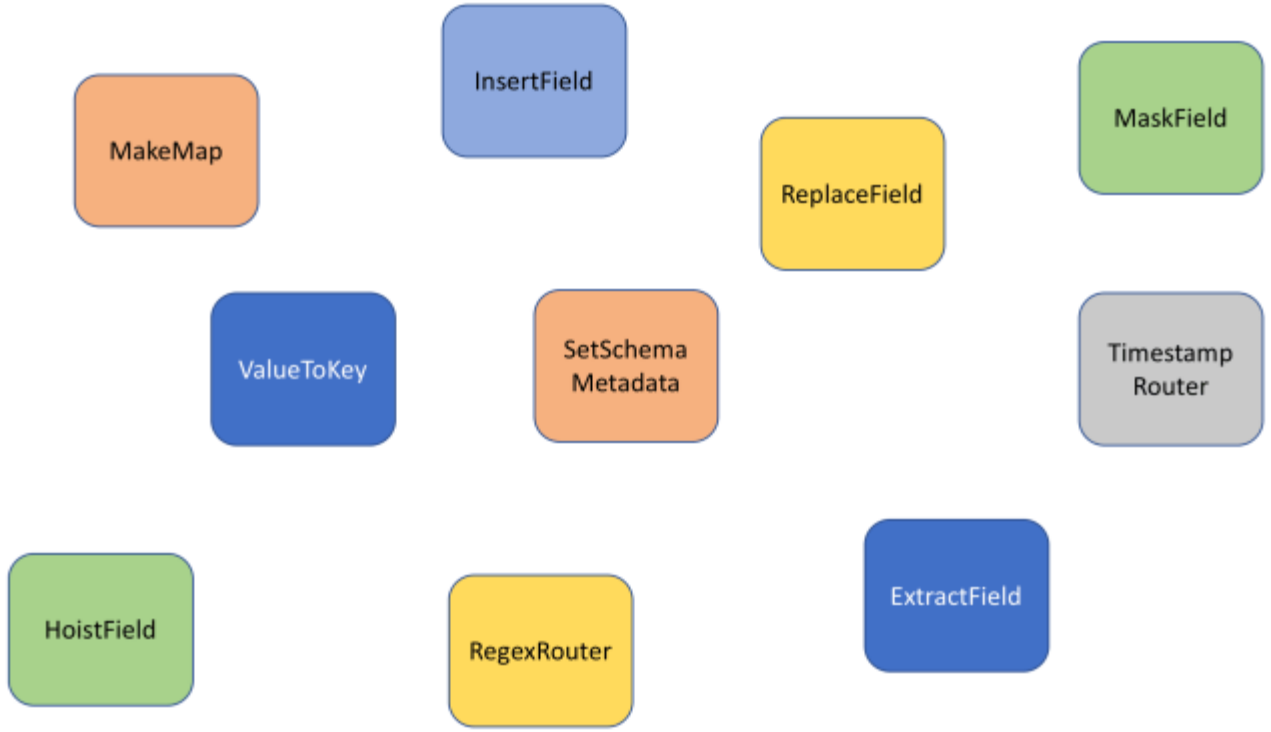

*Example of a source connector which has created two tasks, which copy data from input partitions and write records to Kafka.*

# Kafka Connect – Single Message Transforms (SMTs)

Single Message Transforms (SMT) is a functionality within Kafka Connect that enables the transformation … of single messages

Anything that's more complex, such as aggregating or joins streams of data should be done with Kafka Streams

Single Message Transforms provides us the ability to transform a message before they get in or out of a connector using Kafka Connect. Kafka Connect is installed with a number of connectors. Here is the list of available Transforms:



## Drop a Field

This configuration snippet shows how to use ReplaceField to blacklist a field and remove it.

"transforms": "ReplaceField",
"transforms.ReplaceField.type":
"org.apache.kafka.connect.transforms.ReplaceField$Value"
,
"transforms.ReplaceField.blacklist": "c2“

This removes c2, transforming the original message as seen here:

{"c1":22,"c2":"foo"}  to this result:  {"c1":22}

**Rename a Field**

This configuration snippet shows how to use ReplaceField with the renames property.

"transforms": "RenameField",
"transforms.RenameField.type":
"org.apache.kafka.connect.transforms.ReplaceField$Value",
"transforms.RenameField.renames": "foo:c1,bar:c2"
This replaces the foo and bar field names with c1 and c2.

Before: {"foo":22,"bar":"baz"}

After: {"c1":22,"c2":"baz"}

This configuration snippet shows how to use `MaskField` to mask the value of a field.

```
"transforms": "MaskField",
"transforms.MaskField.type": "org.apache.kafka.connect.transforms.MaskField$Value",
"transforms.MaskField.fields": "string_field"
```

This masks `string_field`, transforming the original message as seen here:

```
{"integer_field":22, "string_field":"foo"}
```

into the result here:

```
{"integer_field":22, "string_field":""}
```

## Transform Fields to Message Key

This configuration snippet shows how to use `ValueToKey` to transform the `UserId`, `city`, and `state` fields into a message key.

```
"transforms": "ValueToKey",
"transforms.ValueToKey.type":"org.apache.kafka.connect.transforms.ValueToKey",
"transforms.ValueToKey.fields":"userId,city,state"
```

Before: `{"userId": 12, "address": "1942 Wilhelm Boulevard", "city": "Topeka", "state": "KS", "country": "US"}`

After: `{"userId": 12, "city": "Topeka", "state": "KS"}`

## Extract Field Name

The configuration snippet below shows how to use `ExtractField` to extract the field name `"id"`.

```
"transforms": "ExtractField",
"transforms.ExtractField.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
"transforms.ExtractField.field":"id"
```

Before: `{"id": 42, "cost": 4000}`

After: `42`

## Add a Topic Prefix

This configuration snippet shows how to add the prefix acme_ to the beginning of a topic.

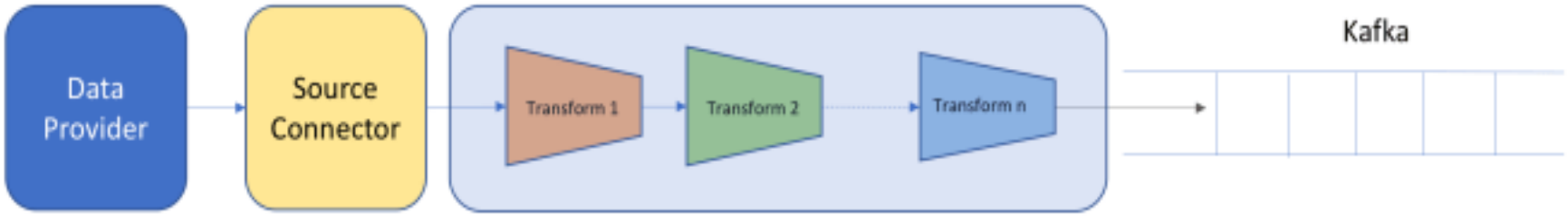
"transforms=AddPrefix"
"transforms.AddPrefix.type"="org.apache.kafka.connect.transforms.RegexRouter"
"transforms.AddPrefix.regex=.*"
"transforms.AddPrefix.replacement=acme_$0“

Before: Order

After: acme_Order

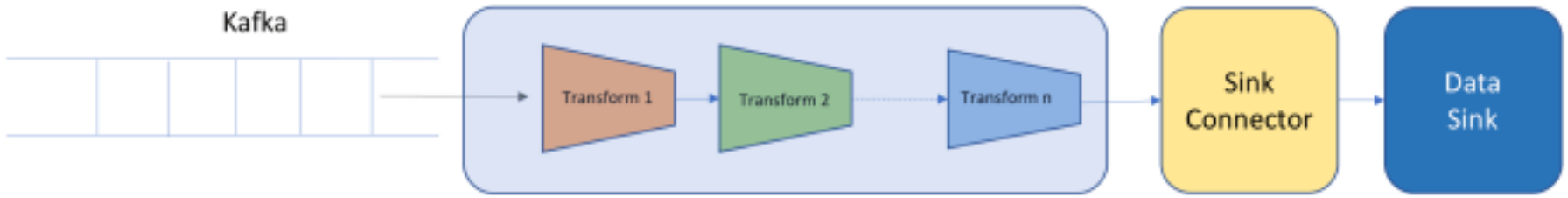
## Source Transforms

Source connector can be configured with a list of transforms. These transforms are applied operate on the data produced by a source connector. They work in a pipeline fashion where the output of one transform is provided as the input for the next transform. In the end, the data from the last transform is published to Kafka.



## Sink Transforms

Sink connectors can also be configured with a list of transforms. As consumer API pulls data from Kafka, it also goes through a list of transforms in a pipeline fashion. The data is then provided to the Sink connector, which can operate and push it to an actual sink e.g. database.



# Kafka Streams

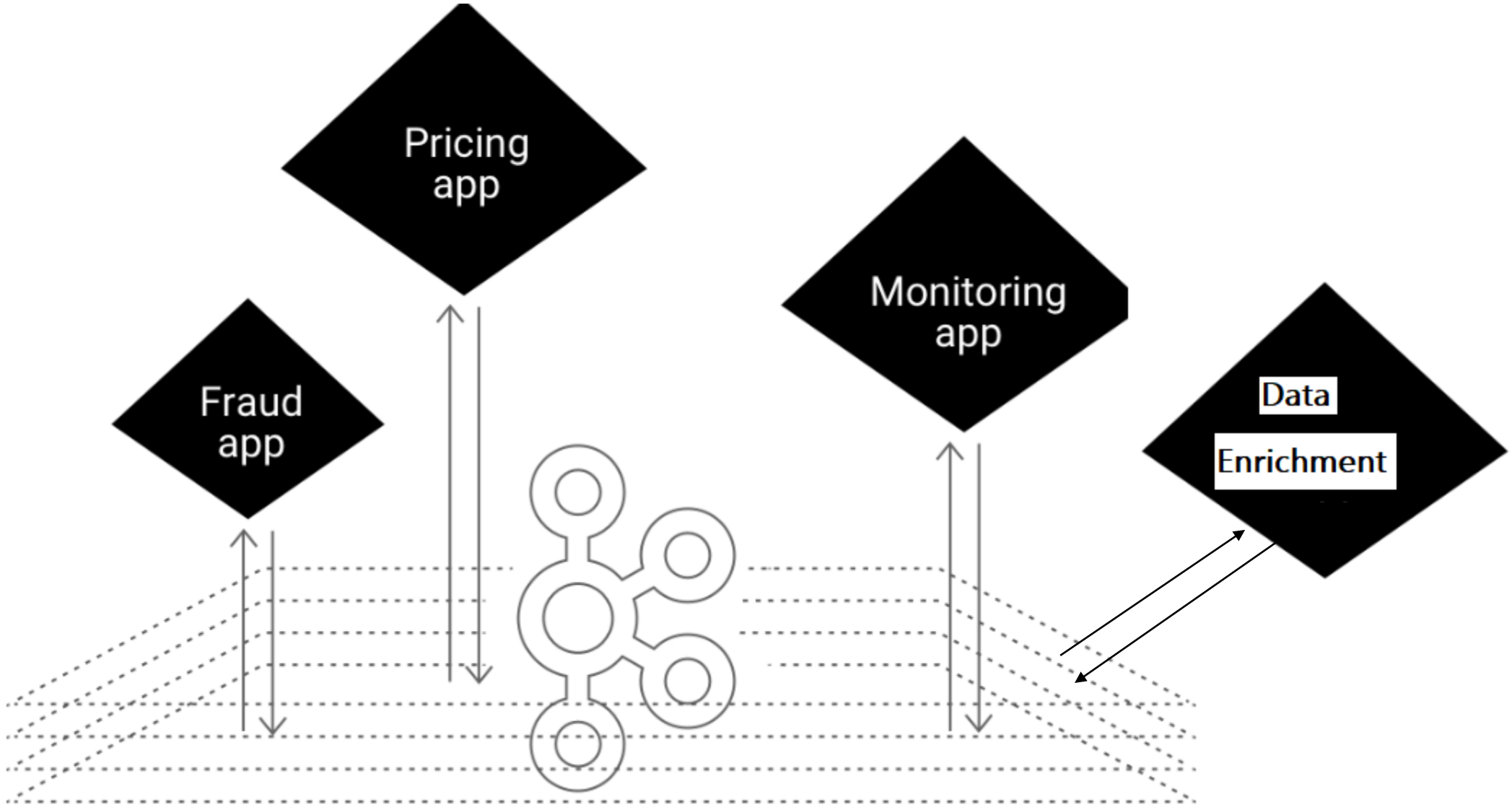
**Kafka Streams** is a java library used for analyzing and processing data stored in Apache Kafka.

It is  capable of doing stateful and/or stateless processing on real-time data.

It's built on top of native Kafka consumer/producer protocols

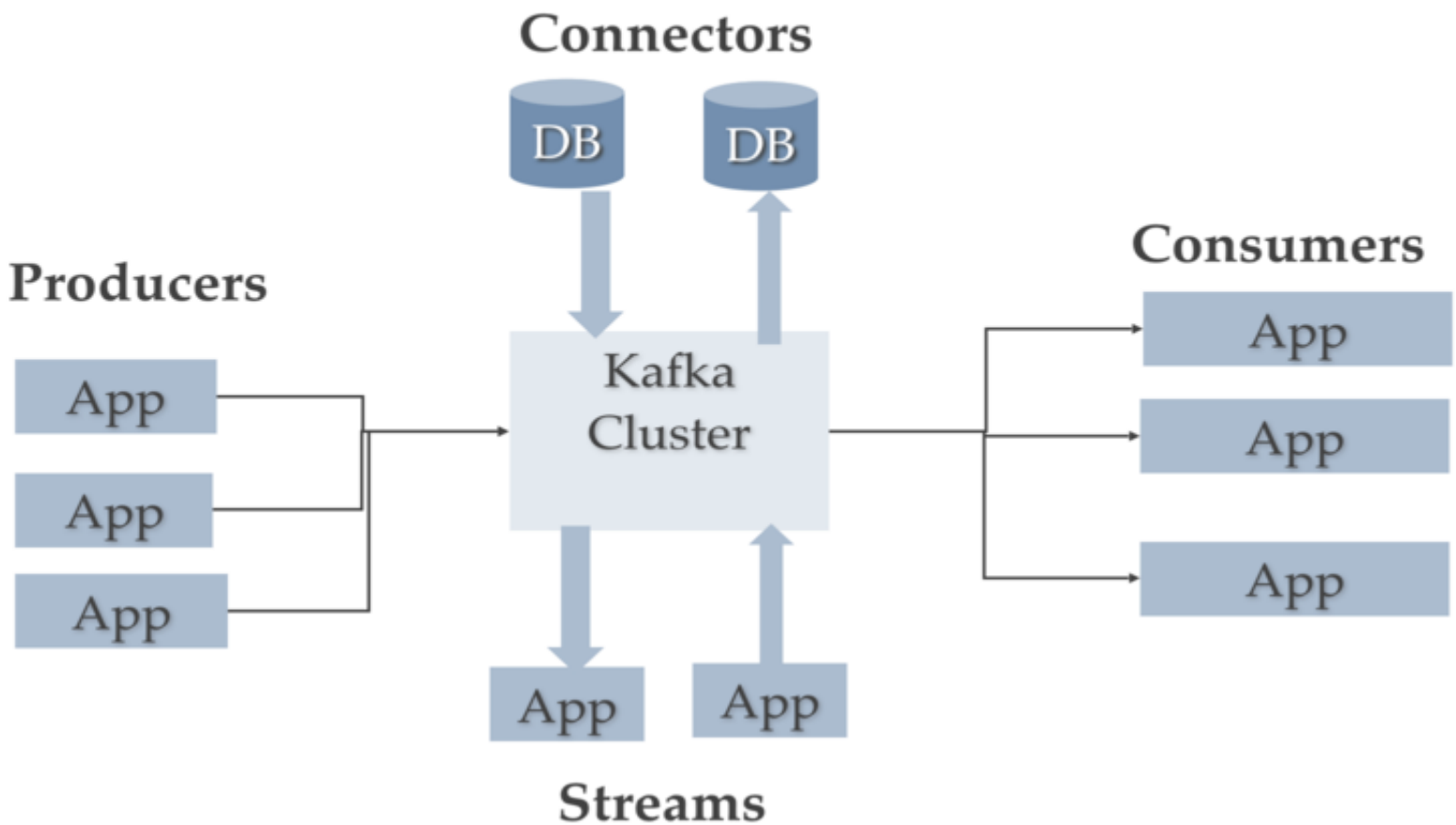
**Kafka Streams** is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters.

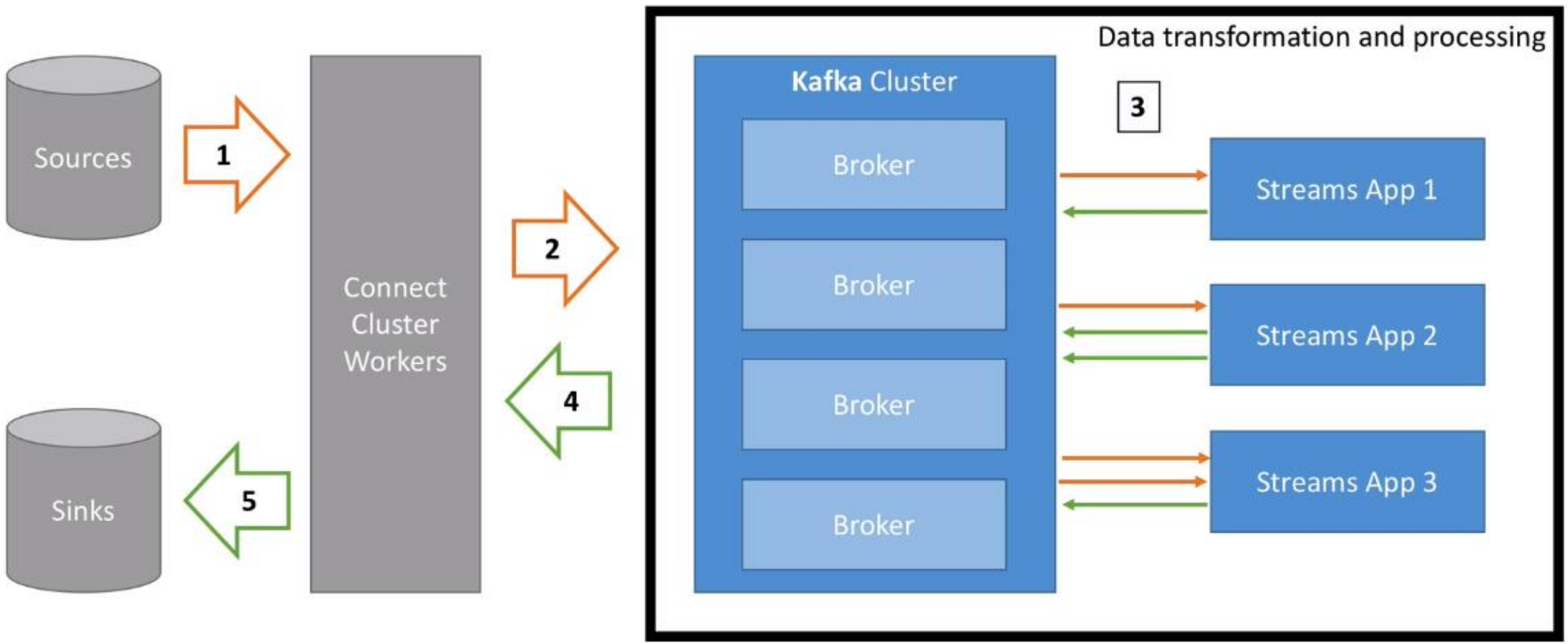
It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology.


Pricing app
Fraud app
Monitoring app
Data
Enrichment

- ✓ Write standard Java applications
- ✓ No seperate processing cluster required
- ✓ Elastic, highly scalable, fault-tolerant
- ✓ Equally viable for small, medium, & large use cases
- ✓ Exactly-once processing semantics
- ✓ Develop on Mac, Linux, Windows
- ✓ Deploy to containers, VMs, bare metal, cloud
- ✓ Fully integrated with Kafka security

# Kafka Connectors and Streams




Sources
1
Connect
Cluster
Workers
2
4
Sinks
5
Data transformation and processing
Kafka Cluster
Broker
Broker
Broker
Broker
3
Streams App 1
Streams App 2
Streams App 3

# Kafka Streams Terminology

There are two special processors in the topology:

**Source Processor**: A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.

**Sink Processor**: A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

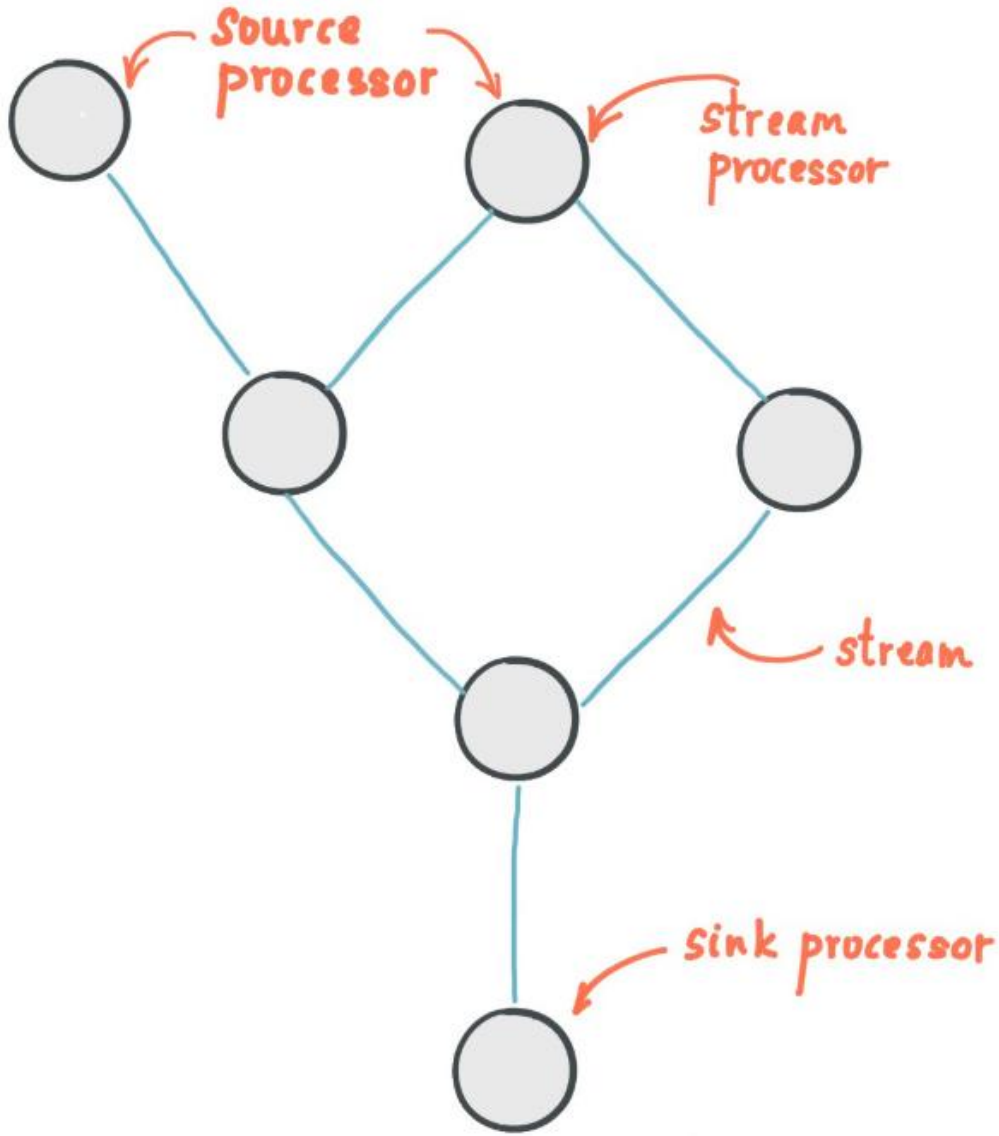
## Processor Topology

A processor topology or simply topology defines the computational logic of the data processing that needs to be performed by a stream processing application.

A topology is a graph of stream processors (nodes) that are connected by streams (edges). Developers can define topologies either via the low-level Processor API or via the Kafka Streams DSL, which builds on top of the former.

```
Topology topology = streamsBuilder.build();
KafkaStreams streams = new KafkaStreams(topology, props);
```

Topology is a logical representation of a ProcessorTopology. A topology is an acyclic graph of sources, processors, and sinks. A source is a node in the graph that consumes one or more Kafka topics and forwards them to its successor nodes




PROCESSOR TOPOLOGY

We can define the processor topology with the Kafka Streams APIs:

**Kafka Streams DSL**
A high-level API that provides the most common data transformation operations such as map, filter, join, and aggregations out of the box.

**Processor API**
A low-level API that lets you add and connect processors as well as interact directly with state stores. The Processor API provides us  with even more flexibility than the DSL but at the expense of requiring more lines of code.

## Processor API

```
public static class CustomProcessor implements Transformer<String, String,
KeyValue<String, Long>> {

  private Long cap;
  private String stateStoreName;
  private ProcessorContext context;
  private KeyValueStore<String, Long> kvStore;

  @Override
  public void init(ProcessorContext context) {
    this.context = context;
    this.kvStore = (KeyValueStore<String, Long>)
context.getStateStore(stateStoreName);
  }
  ...
}
```

## Kafka Streams DSL

```
KTable<String, Long> wordCounts = textLines
        // 2 - map values to lowercase
                // <null, "kafka kafka streams">
        .mapValues(textLine -> textLine.toLowerCase())
         // 3 - flatmap values split by space
        // <null,"kafka">,<null,"kafka">, <null,"streams">
        .flatMapValues(textLine -> Arrays.asList(textLine.split("\\W+")))
        // 4 - select key to apply a key (we discard the old key)
        // <"kafka","kafka">,<"kafka","kafka">, <"streams","streams">
        .selectKey((key, word) -> word)
        // 5 - group by key before aggregation
        // (<"kafka","kafka">,<"kafka","kafka">), (<"streams","streams">)
        .groupByKey()
        // 6 - count occurences
        // <"kafka",2>, <"streams",1>
         // Used to describe how a StateStore should be materialized.
        .count(Materialized.as("Counts"));
```

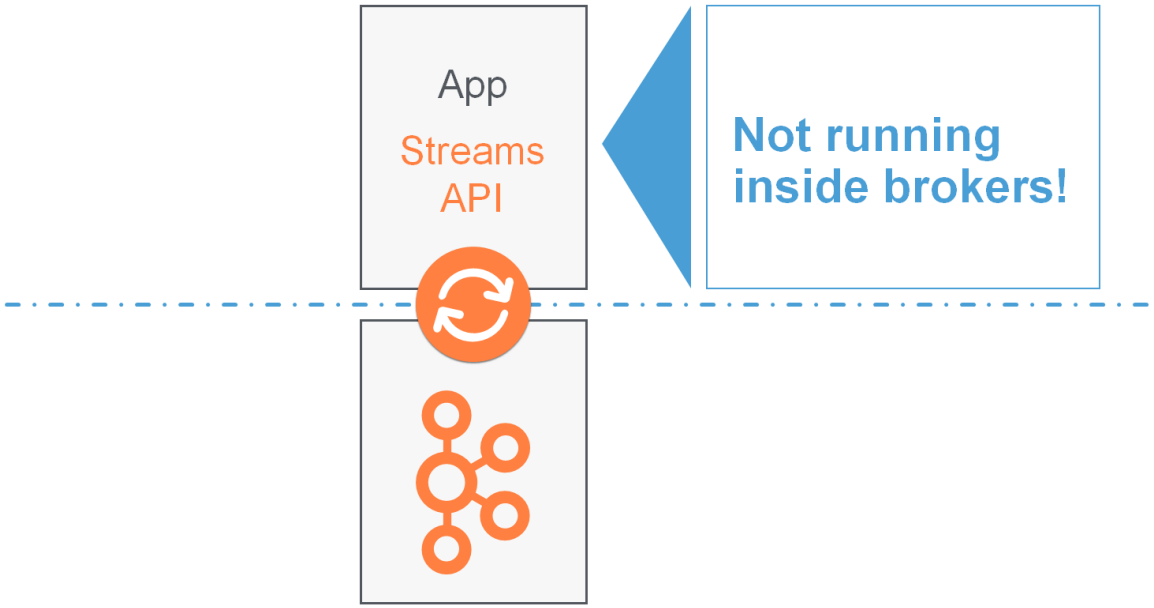
**KSQL** : Stream processing could be expressed by anyone using SQL as the language.

It offers an easy way to express stream processing transformations as an alternative to writing an application in a programming language such as Java.

Moreover, processing transformation written in SQL like language can be highly optimized by execution engine without any developer effort.

The stream processing application doesn't run inside a broker.

Instead, it runs in a separate JVM instance, or in a separate cluster entirely.



**Internal Topics:**

Kafka streams create the below internal topics:

**Repartitioning** topics :  in case we start transforming the key of our stream, a repartitioning will happen at some processor.

**Changelong** topics: In case we perform aggregations kafka streams will save compacted data in these topics.

**Internal topics:**

Are managed by Kafka Streams
Are used to save/restore state and re-partition data
Are prefixed by application.id parameter

Note:
1. Repartition topic that will hold our transformed messages.
2. The changelog topic basically keeps track of the updates made to the state store

## Stream processing : stateless or stateful

With stateless processing, a record can be processed independently without the need of additional information from preceding records.

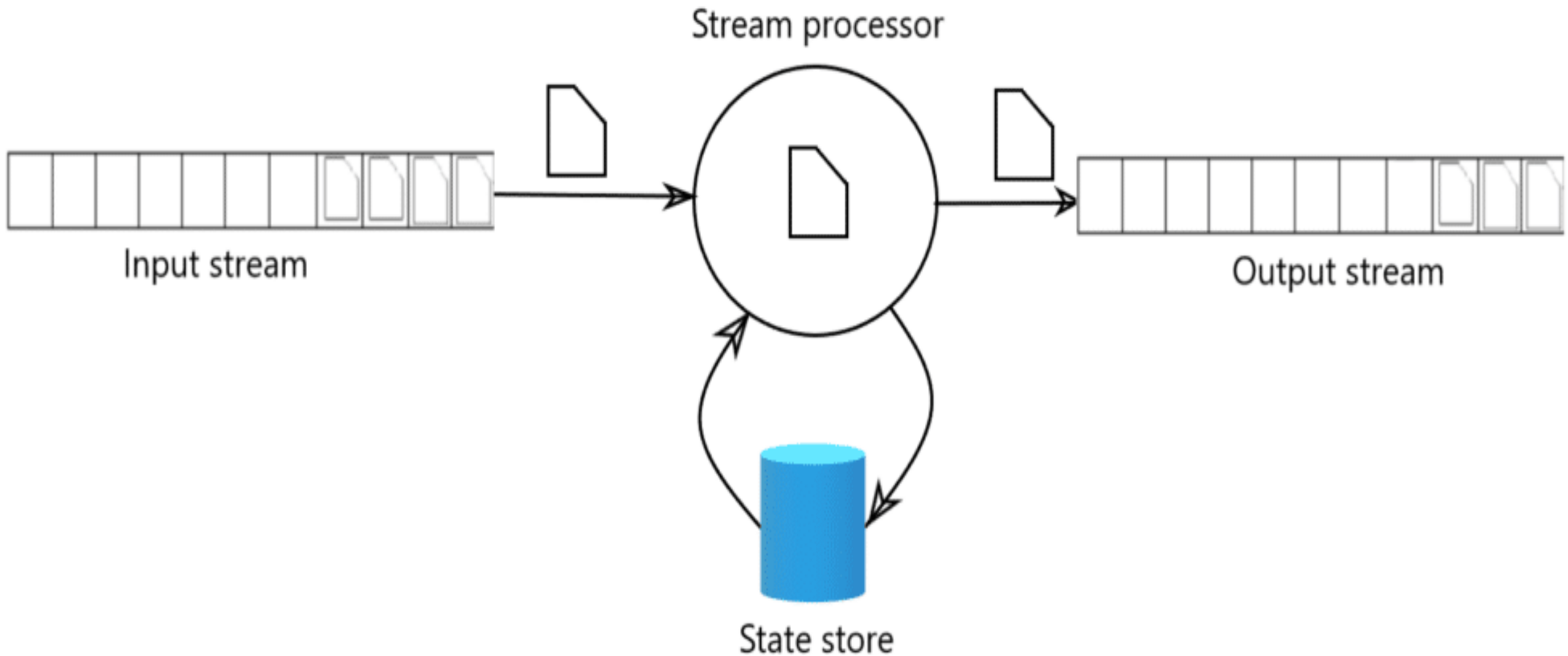
One example of stateless stream processing is filtering out a specific type of event from the data stream to use it to trigger some responding actions. In this situation, the processor does not have to maintain any internal state, it only reads an incoming event, checks whether the event matches some criteria and either drops it or forwards it to downstream processing.

Stateful processing, the tasks involve using information about the state cumulated from other processed records.

We can persist and query the state for the stream processor.

There are numerous stateful processing operators.

For instance, counting the number of data records with the same attribute value, aggregating data such as moving average, finding pre-determined pattern on the data stream.


Stream processor
Input stream
Output stream
State store

## Stateless vs Stateful Operations

**Stateless transformations** do not require state for processing and they do not require a state store associated with the stream processor.

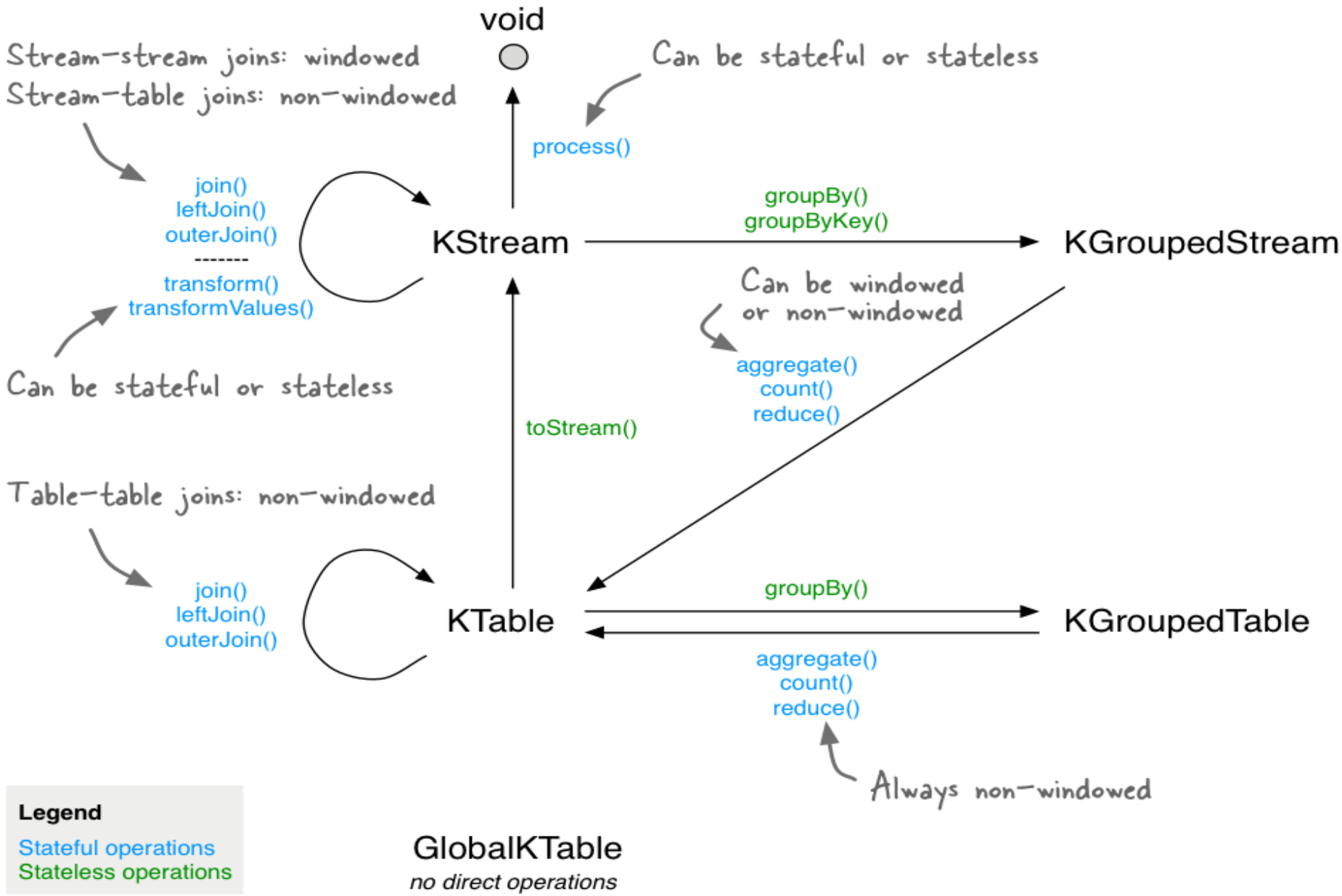
Examples:
Map()
MapValue()
FlatMap()
GroupBy()
SelectKey()
Filter()
KStream → KStream
KTable → KStream

**Stateful transformations** depend on state for processing inputs and producing outputs and require a state store associated with the stream processor.

For example, in aggregating operations, a windowing state store is used to collect the latest aggregation results per window.

In join operations, a windowing state store is used to collect all of the records received so far within the defined window boundary.

join()  aggregate()  count()
reduce()  process()


void
Stream–stream joins: windowed
Stream–table joins: non–windowed
Can be stateful or stateless
process()
join()
leftJoin()
outerJoin()
---
transform()
transformValues()
KStream
groupBy()
groupByKey()
KGroupedStream
Can be windowed
or non–windowed
aggregate()
count()
reduce()
Can be stateful or stateless
toStream()
Table–table joins: non–windowed
join()
leftJoin()
outerJoin()
KTable
groupBy()
KGroupedTable
aggregate()
count()
reduce()
Always non–windowed
Legend
Stateful operations
Stateless operations
GlobalKTable
no direct operations

**KStreams**

- All inserts
- Similar to a log
- Unbounded data streams

**KTables**

- All upserts on non null values
- Delete on null values
- Similar to a table

## KStream vs KTable

- KStream reading from a topic that's not compacted
- KTable reading from a topic that's log-compacted (aggregations)
- KStream if new data is partial information / transactional
- KTable if we need a structure that's like a "database table", where every update is self sufficient (account balance)

**Streams marked for re-partition**

As soon as an operation can possibly change the key, the stream will be marked for repartition:

- Map
- FlatMap
- SelectKey

So, only use these APIs if we need to change the key, otherwise use their counterparts:

- MapValues
- FlatMapValues

Repartitioning is done seamlessly behind the scenes but will incur a performance cost.

## MapValues / Map

Transform the value of each input record into a new value (with possible new type) of the output record.

**MapValues**
> Is only affecting values
> does not change keys
> does not trigger a repartition
> For Kstreams and KTables

**Map**
> Affects both keys and values
➢ Triggers a re-partitions > For KStreams only

```
uppercasedata = stream.mapValues(value ->
value.toUpperCase());
```

**map :**

Transform each record of the input stream into a new record in the output stream (both key and value type can be altered arbitrarily). The provided KeyValueMapper is applied to each input record and computes a new output record.

```
 KStream<String, String> inputStream = builder.stream("topic");
 KStream<String, Integer> outputStream = inputStream.map(new
KeyValueMapper<String, String, KeyValue<String, Integer>>
{
    KeyValue<String, Integer> apply(String key, String value) {
       return new KeyValue<>(key.toUpperCase(), value.split("
").length);    }
 });
```

**Mapvalues** :

Transform the value of each input record into a new value (with possible new type) of the output record. The provided ValueMapper is applied to each input record value and computes a new value for it.

```
 KStream<String, String> inputStream = builder.stream("topic");
 KStream<String, Integer> outputStream =
inputStream.mapValues(new ValueMapper<String, Integer> {
    Integer apply(String value) {
        return value.split(" ").length;
    }
 });
```

**Filter / FilterNot**

Takes one record and produces zero or one record

**Filter**
> does not change keys / values
> does not trigger a repartitions
> for KStreams and KTables

**FilterNot**
Inverse of Filter

```
KStream<String, Long> onlyPositives = stream.filter((key,value)->
value>0);

KStream<String, Long> onlyPositives = stream.filterNot((key,
value) -> value <= 0);
```

**FlatMapValues / FlatMap**

Takes one record and produces zero, one or more record

**FlatMapValues**
> does not change keys
> does not trigger a repartition  > For KStreams only

**FlatMap**
> Changes keys
> triggers a repartition  > For KStreams only

```
words = sentences.flatMapValues
            (value->Arrays.asList(value.split("\\s+")));
```

(rushi, rushi is great)  creates 3 records :  (rushi,rushi), (rushi, is), (rushi,great)

**Difference between map() and flatMap()**

The function we pass to map() operation returns a single value. The function we pass to flattMap() operation returns Stream of values. The flatMap() is a combination of map and flat operation. The map() is used for transformation, but flatMap() is used for both transformation and flattening.

map() example:

```
List listOfIntegers = Stream.of("1","2","3","4")
        .map(Integer::valueOf)
        .collect(Collectors.toList());
```

flatMap() example:

```
List numbers = Stream.of(events, odds, primes)
        .flatMap(list -> list.stream())
        .collect(Collectors.toList());
```

**SelectKey**

Assigns a new Key to the record (from old key and value)
marks the data for re-partitioning.

```
//Use the first letter of the key as the new key
rekeyed = stream.selectKey((key,value) -> key.substring(0,1))
```

The **Aggregator** interface for aggregating values of the given key. This is a generalization of Reducer and allows to have different types for input value and aggregation result.

Aggregator is used in combination with Initializer that provides an initial aggregation value.

Aggregator can be used to implement aggregation functions like count.

The **Reducer** interface for combining two values of the same type into a new value.

In contrast to Aggregator the result type must be the same as the input type.

The provided values can be either original values from input KeyValue pair records or be a previously computed result from apply(Object, Object).

Reducer can be used to implement aggregation functions like sum, min, or max.

**KStream & KTable Duality**

**Stream as Table** : A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table.

**Table as Stream**: A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream (a stream's data records are key-value pairs).

**Transforming a KTable to KStream**

It is sometimes helpful to transform a KTable to KStream in order to keep a changelog of all the changes to the KTable.

```
KTable<byte[], String>  table =...;

//Also, a variant of 'toStream' exists that allow you
// to select a new key for the resulting stream

KStream<byte[], String> stream = table.toStream();
```

**Transforming a KStream to a KTable**

Two ways:

> Chain a groupByKey() and an aggregation step (count,aggregate,reduce)

```
KTable<String, Long>  table = userAndColours.groupByKey().count();
```

> Write back to Kafka and read as KTable

```
stream.to("intermediary-topic");

KTable<String, String> table = builder.table("intermediary-topic");
```

## GlobalKTable

Like a KTable, a GlobalKTable is an abstraction of a changelog stream, where each data record represents an update.

A GlobalKTable differs from a KTable in the data that they are being populated with, i.e. which data from the underlying Kafka topic is being read into the respective table.

GlobalKTable provides the ability to look up current values of data records by keys. This table-lookup functionality is available through join operations.

Also, when joining against a global table, the input data does not need to be co-partitioned.

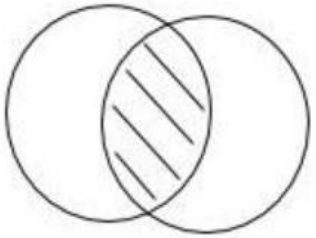
Imagine we have an input topic with 5 partitions. In our application, we want to read this topic into a table. Also, we want to run our application across 5 application instances for maximum parallelism.

If we read the input topic into a KTable, then the "local" KTable instance of each application instance will be populated with data from only 1 partition of the topic's 5 partitions.

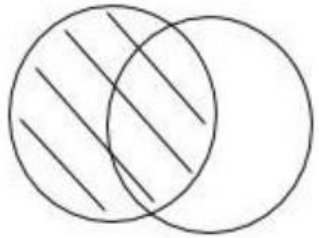
If we read the input topic into a GlobalKTable, then the local GlobalKTable instance of each application instance will be populated with data from all partitions of the topic.

## Kafka Stream - join

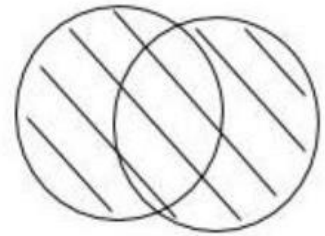
Joining means taking a KStream and / or KTbale and creating a new KStream or KTable from it.



inner:
*KStream-KStream*
*KTable-KTable*
*KStream-KTable*
*KStream-GlobalKTable*



left:
*KStream-KStream*
*KTable-KTable*
*KStream-KTable*
*KStream-GlobalKTable*



outer:
*KStream-KStream*
*KTable-KTable*

**Inner Joins:** Emits an output when both input sources have records with the same key.

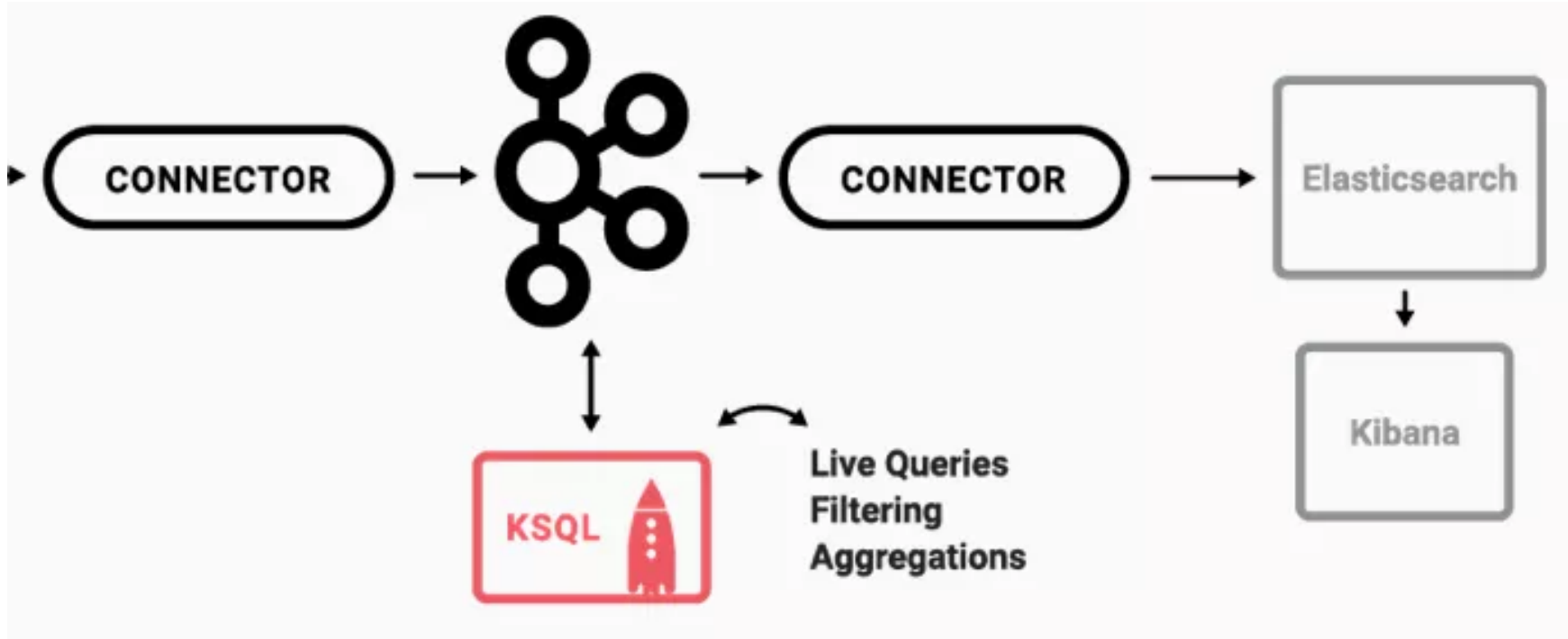
**Left Joins:** Emits an output for each record in the left or primary input source. If the other source does not have a value for a given key, it is set to null.

**Outer Joins**: Emits an output for each record in either input source. If only one source contains a key, the other is null.

# What is KSQL

KSQL is a is distributed, scalable, reliable, and real time SQL engine for Apache Kafka. It allows you to write SQL queries to analyze a stream of data in real time. A query with KSQL will keep generating results over the stream of unbounded data until you stop it.

It is built on top of Kafka Streams which means that KSQL offers similar concepts as to what Kafka Streams offers, but all with a SQL language: streams (KStreams), tables (KTables), joins, windowing etc.


CONNECTOR
CONNECTOR
Elasticsearch
Kibana
KSQL
Live Queries
Filtering
Aggregations

**Kafka Streams API / KSQL:**

Applications wanting to consume from Kafka and produce back into Kafka, also called stream processing.

Use KSQL if you think you can write your real-time job as SQL-like, use Kafka Streams API if you think you're going to need to write complex logic for your job.

## KSQL - Advantages

KSQL is not directly part of the Kafka API, but a wrapper on top of Kafka Streams.

While Kafka Streams allows us to write some complex topologies, it requires some substantial programming knowledge and can be harder to read, especially for newcomers.

KSQL wants to abstract that complexity away by providing you with a SQL semantic.

## KSQL - Limitations

If we want to have complex transformations, explode arrays, or need a feature that’s not yet available, sometimes we have to revert back to using Kafka Streams.

## Comparing a KSQL query to a Relational Database query

Queries in relational databases are one-time queries that are run once to completion over a data set, eg. SELECT statement on finite rows.

In contrast, what KSQL runs are continuous queries, transformations that run continuously as new data passes through them, on streams of data in Kafka topics.

**Real-time monitoring and real-time analytics**

It can be used in defining custom business-level metrics that are computed in real-time and that you can monitor and alert off of at real-time

Example KSQL query,

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

## Fraud detection application

KSQL gives a simple, sophisticated, and real-time way of defining aggregation and anomaly detection queries on real-time streams.

Example KSQL query,

```
CREATE TABLE possible_fraud AS SELECT
card_number,
count() FROM authorization_attempts
WINDOW TUMBLING (SIZE 5 SECONDS)
GROUP BY card_number HAVING count() > 3;
```

## Online Data Integration

KSQL queries can be used to integrate and join different streams of data in real-time.

```
CREATE STREAM vip_users AS SELECT userid, page,
action FROM clickstream c
LEFT JOIN users u ON c.userid = u.user_id
WHERE u.level = ‘Platinum’;
```

# AVRO

# JSON

## Advantages:

Data can take any form (arrays, nested elements)
JSON is a widely accepted format on the web
JSON can be read by any languare
JSON can be easily shared over a network

## Disadvantages:

```
// Despite the firstName key being repeated, this is still valid JSON
{
  "id" : 001,
  "firstName" : "John",
  "firstName" : "Jane",
  "lastName" : "Doe"
}
```

Data has no schema enforcing
JSON Objects can be quite big in size because of repeated keys
No comments, metadata, documentation

**AVRO**

It is defined by a schema (written in JSON)

Avro is JSON with a schema attached to it

## AVRO

### Advantages

Data is fully typed
Data is compressed automatically
Schema (defined using JSON) comes along with the data
Documentation is embedded in the schema
Data can be read across any language
Schema can evolve over time, in a safe manner (schema evolution)

### Disadvantages

Can't print the data without using avro tools
(because it is compressed and serialised)

```
$ curl -i -X POST -H "Content-Type:
application/vnd.kafka.avro.v1+json" --data '{
   "value_schema": "{\"type\": \"record\",
\"name\": \"User\", \"fields\": [{\"name\":
\"username\", \"type\": \"string\"}]}",
   "records": [
     {"value": {"username": "testUser"}},
     {"value": {"username": "testUser2"}}
   ]
}' \
http://localhost:8082/topics/avrotest
```

Note : Both $key_schema and $value_schema parameters are optional and provide JSON strings that represent Avro schemas to use to validate and serialize key(s) and value(s).

This sends an HTTP request using the POST method to the endpoint http://localhost:8082/topics/avrotest, which is a resource representing the topic avrotest.

The content type, application/vnd.kaka.avro.v1+json, indicates the data in the request is for the Kafka proxy (application/vnd.kafka), contains Avro keys and values (.avro), using the first API version (.v1), and JSON encoding (+json).

The payload contains a value schema to specify the format of the data (records with a single field username) and a set of records. Records can specify a key, value, and partition, but in this case we only include the value.

The values are just JSON objects because the REST Proxy can automatically translate Avro's JSON encoding, which is more convenient for your applications, to the more efficient binary encoding you want to store in Kafka.

The server will respond with:

```
HTTP/1.1 200 OK
Content-Length: 209
Content-Type: application/vnd.kafka.v1+json
Server: Jetty(8.1.16.v20140903)
{
  "key_schema_id": null,
  "value_schema_id": 1,
  "offsets": [
    {"partition": 0, "offset":0, "error_code": null, "error":
null},
    {"partition": 0, "offset":1, "error_code": null, "error":
null}
  ]
}
```

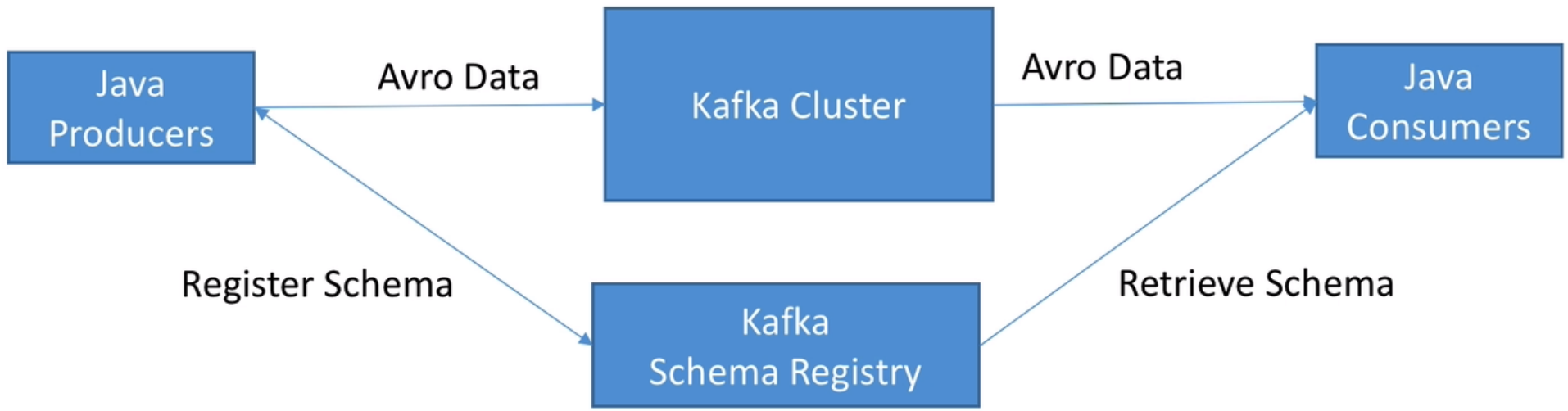
This indicates the request was successful (200 OK) and returns some information about the messages.

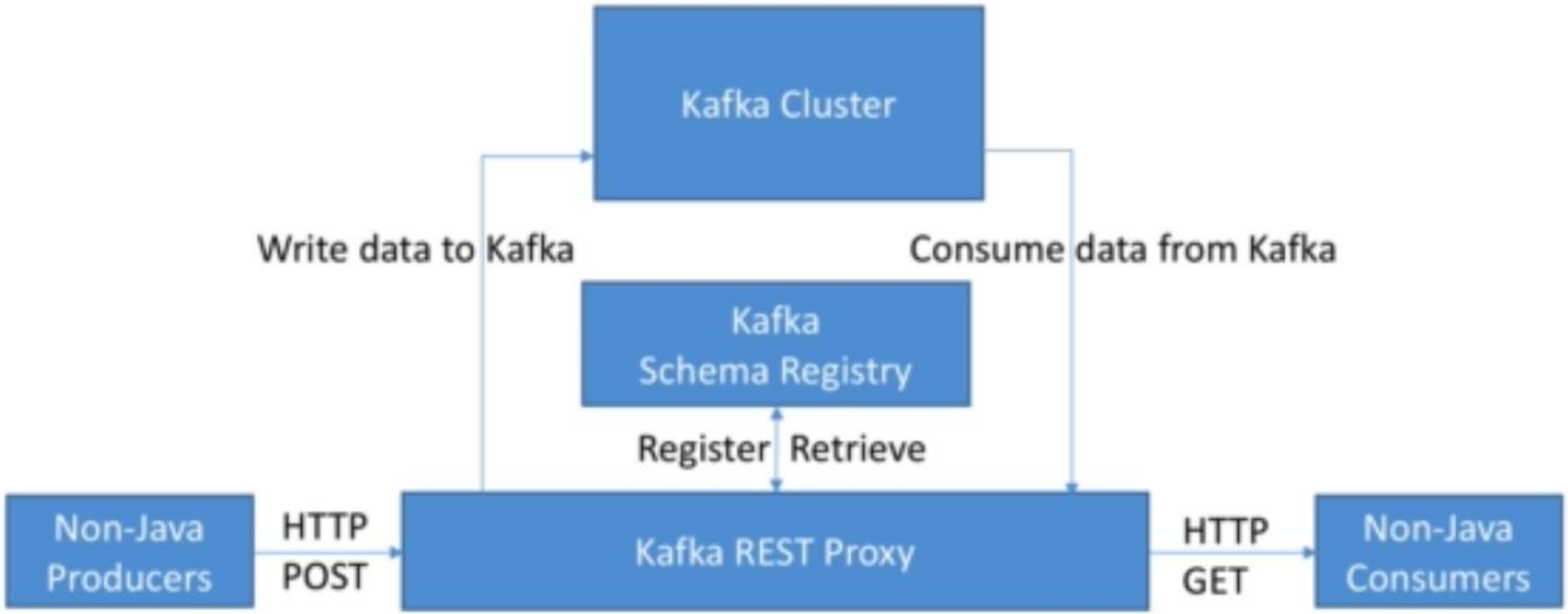
Schema IDs are included, which can be used as shorthand for the same schema in future requests.

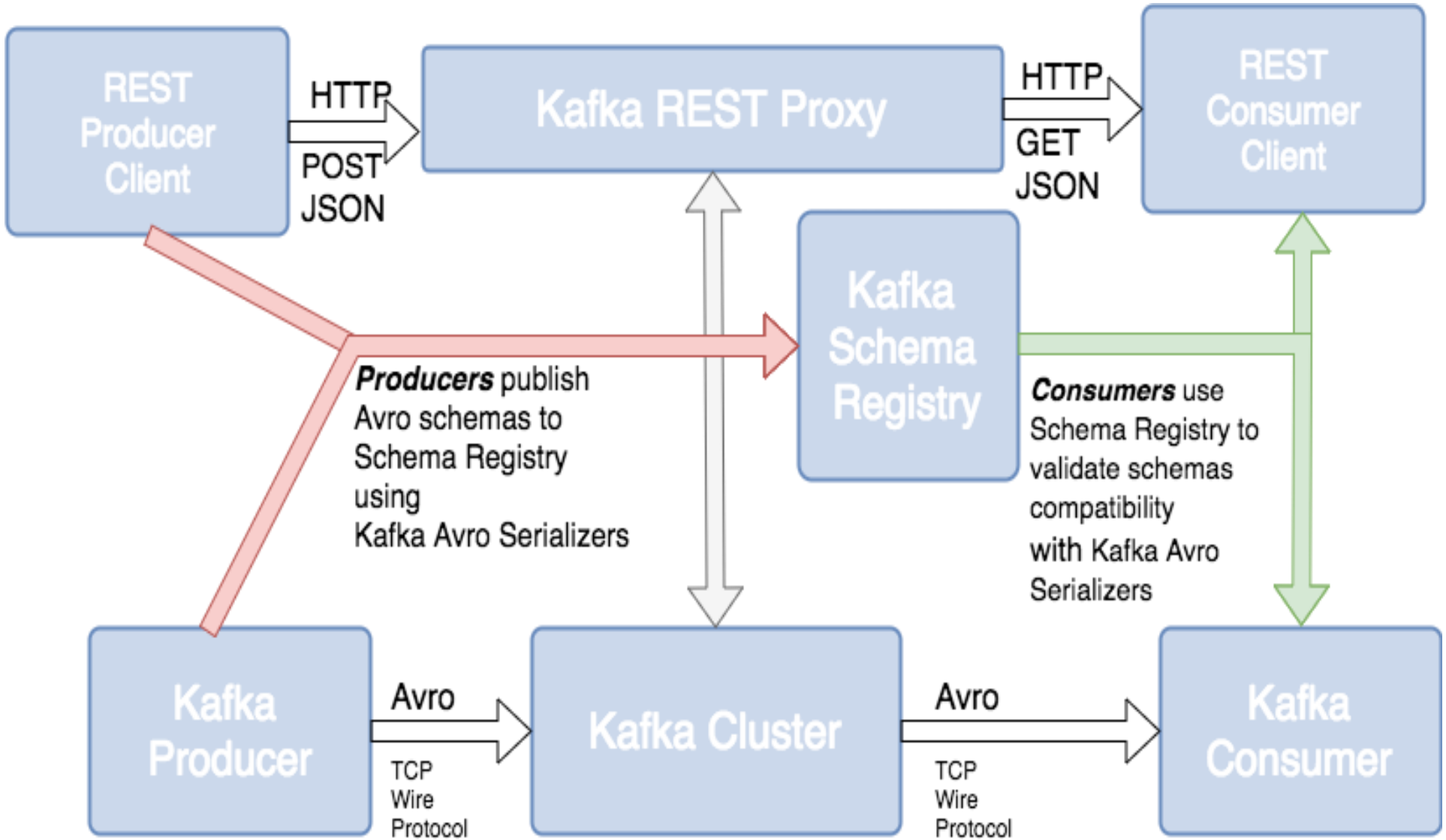
Information about any errors for individual messages (error and error_code) are provided in case of failure, or are null in case of success.

Successfully recorded messages include the partition and offset of the message.

# Confluent  Schema Registry


Java
Producers
Avro Data
Kafka Cluster
Avro Data
Java
Consumers
Register Schema
Kafka
Schema Registry
Retrieve Schema


Kafka Cluster
Write data to Kafka
Consume data from Kafka
Kafka
Schema Registry
Register
Retrieve
Non-Java
Producers
HTTP
POST
Kafka REST Proxy
HTTP
GET
Non-Java
Consumers


REST Producer Client
HTTP
POST
JSON
Kafka REST Proxy
HTTP
GET
JSON
REST Consumer Client
Kafka Schema Registry
Producers publish Avro schemas to Schema Registry using Kafka Avro Serializers
Consumers use Schema Registry to validate schemas compatibility with Kafka Avro Serializers
Kafka Producer
Avro
TCP Wire Protocol
Kafka Cluster
Avro
TCP Wire Protocol
Kafka Consumer

**Primitive Types**

The set of primitive type names is:

null: no value
boolean: a binary value
int: 32-bit signed integer
long: 64-bit signed integer
float: single precision (32-bit) IEEE 754 floating-point number
double: double precision (64-bit) IEEE 754 floating-point number
bytes: sequence of 8-bit unsigned bytes
string: unicode character sequence

{"type": "string"}

## Complex Types

Avro complex types are:

Enum
Arrays
Maps
Unions
Calling other schemas as types

**Avro Complex Types:**

Enums Example :

{"type": "enum", "name": "AccountType", "symbols" : ["Savings" , "Current", "Loan"]}

Arrays  Example:

{"type": "array", "items": "string",  "value": ["one", "two", "three"]}

Maps Example:

{"type": "map", "values": "int", "value": ["one": 1, "two": 2, "three": 3]}

## Avro Record Schemas

It is defined using JSON

It has some common files:

Name:  Name of the schema
Namespace : (equivalent of package in java)
Doc: Documentation to explain the schema
Aliases: Optional other names for the schema
Fields :
    Name : Name of the field
    Doc: Documentation for that field
    Type: Data type for that field (can be a primitive type)
    Default: Default value for that field

## Avro in Java

A **GenericRecord** is used to create an avro object from schema, the schema being referenced as  "file / string".

It is not the most recommended way of creating Avro objects because things can fail at runtime, but it the most simple way.

```
Schema mainSchema = new Schema.Parser().parse(new
ClassPathResource("avro/movie-v1.avsc").getInputStream());

        //Create avro message with defined schema
        GenericRecord avroMessage =
                        new GenericData.Record(mainSchema);

        //Populate avro message
        avroMessage.put("movie_name", " Gods Must Be Crazy ");
        avroMessage.put(" category", " comedy ");
```

**Specific Record**

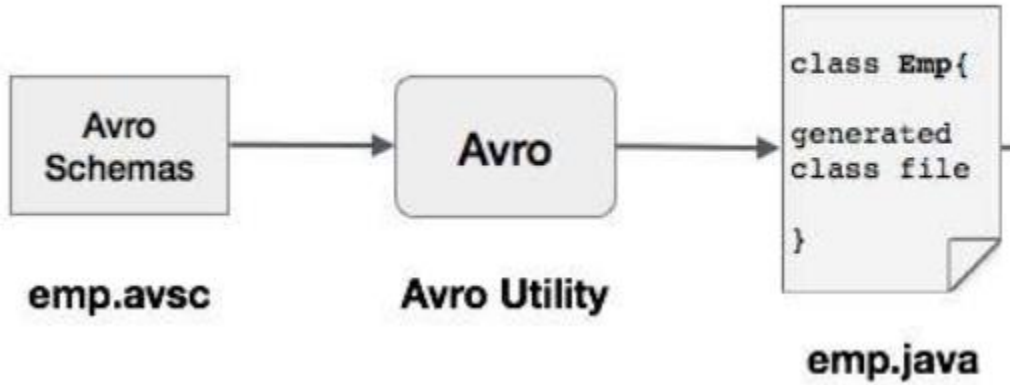
It is also an Avro object, but is obtained using code generation from an Avro schema.

There are different plugins for different build tools (gradle, maven,sbt etc.,)

**Avro Schema  -> Maven Plugin -> Generation Code**

## Specific Record

Compile the schema using Avro utility & Generate a java class



1. Download avro-tools-1.8.2.jar

2. Create emp.avsc

```
{   "namespace": “demo.employee",
   "type": "record",
   "name": "emp",
   "fields": [
      {"name": "name", "type": "string"},
      {"name": "id", "type": "int"},
      {"name": "salary", "type": "int"},
      {"name": "age", "type": "int"},
      {"name": "address", "type": "string"}   ]}
```

3. java -jar <path/to/avro-tools-1.8.2.jar>  compile schema <path/to/schema-file>  <destination-folder>

## Schema Evolution

Avro enables us to evolve our schema over time, to adapt with the changes from the business.

Example:

Initial version(v1) of the customer request contains First Name and Last Name. But, future version(v2) may require email, phone number etc.,

We should make the schema evolve without breaking programs reading our stream of data.

## Schema Evolution

There are 4 types:

**Backward** : it is compatible change is when a new schema can be used to  read old data.

**Forward** : it is compatible change is when an old schema can be used to read new data.

**Full** : which is both backward and forward

**Breaking**: which is none of those

## Backward Compatible

We can read old data with new schema with a default value. In case the field doesn't exist, Avro will use the default.

With backwards, we can successfully perform queries over old and new data using new schema.

Old schema

```
....
"fields":
  [{"name": "firstName", "type": "string"},
   {"name": "lastName", "type": "string"}
   ]}
```

New schema

```
....
"fields":
  [{"name": "firstName", "type": "string"},
   {"name": "lastName", "type": "string"},
   {"name": "phone", "type": "string",
         "default" : "000-000-000"}
   ]}
```

## Forward compatible

We can read new data with the old schema, Avro will just ignore new fields.

Forward compatible is used, when we want to make a data stream evolve without changing our downstream consumers.

Old schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"}
    ]}
```

New schema

```
....
"fields":
   [{"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "email", "type":“string"}
    ]}
```

## Fully compatible

Only add fields with defaults.
Only remove fields that have defaults.
When writing you schema changes, most of the time you want to target full compatibility.

Old schema

```
....
"fields":
  [{"name": "firstName", "type": "string"},
   {"name": "lastName", "type": "string"}
   ]}
```

New schema

```
....
"fields":
  [{"name": "firstName", "type": "string"},
   {"name": "lastName", "type": "string"},
   {"name": "phone", "type": "string",
         "default" : "000-000-000"}
   ]}
```

**Not compatible**

Here are examaples of changes that are NOT compatible:

Adding / Removing elements from an Enum
Changing the type of  a field (string => int etc.,)
Renaming a required field (field without defaultl)

**Don't do that.**

**Wire Format**

Most users can use the serializers and formatter directly and never worry about the details of how Avro messages are mapped to bytes. However, if we are working with a language that Confluent has not developed serializers for, or simply want a deeper understanding of how the Confluent Platform works, we may need more detail on how data is mapped to low-level bytes.

The wire format currently has only a couple of components:

| Bytes | Area | Description |
|---|---|---|
| 0 | Magic Byte | Confluent serialization format version number; currently always `0`. |
| 1-4 | Schema ID | 4-byte schema ID as returned by Schema Registry |
| 5-... | Data | Avro serialized data in Avro's binary encoding. The only exception is raw bytes, which will be written directly without any special Avro encoding. |

In the following example, we send strings and Avro records in JSON as the key and the value of the message, respectively.

```
bin/kafka-avro-console-producer --broker-list localhost:9092 --topic t2 \
  --property parse.key=true \
  --property key.schema='{"type":"string"}' \
  --property value.schema='{"type":"record","name":"myrecord","fields":[{"name":"f1","type":"string"}]}'
```

In the shell, type in the following.

```
"key1" {"f1": "value1"}
```

The following example reads both the key and the value of the messages in JSON.

```
bin/kafka-avro-console-consumer --topic t2 \
  --bootstrap-server localhost:9092 \
  --property print.key=true
```

You should see following in the console.

```
"key1" {"f1": "value1"}
```

Avro message with new schema version (02)

kafka-avro-console-producer --broker-list localhost:9092 --topic t2  --property parse.key=true  --property key.schema='{"type":"string"}'
--property value.schema='{"type":"record","name":"myrecord","fields":[{"name":"f1","type":"string"},{"name":"f2","type":"string","default":"any"}]}'
"key1" {"f1":"apache","f2":"kafka"}

==========

1. From controle centre -> select topic -> Schema  and check the version

2. Select Version history

3. Turn on version diff

4. All topics-> t2-> Schemavalue-> Edit schema -> Compatibility Mode

==========

*Auto Schema Registration*

By default, client applications automatically register new schemas. If they produce new messages to a new topic, then they will automatically try to register new schemas.

Best practice is to register schemas outside of the client application to control when schemas are registered with Schema Registry and how they evolve.

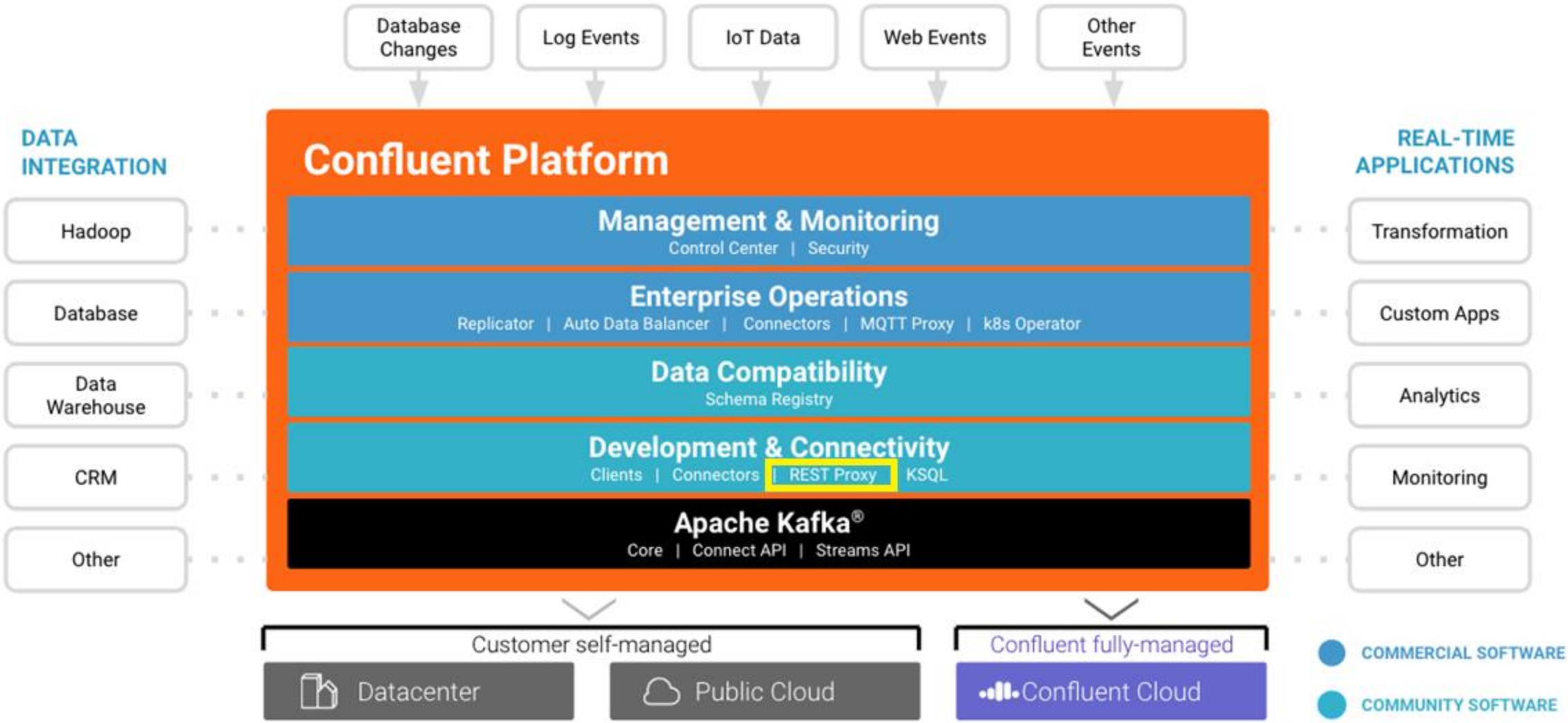
Within the application, you can disable automatic schema registration by setting the configuration parameter auto.register.schemas=false, as shown in the example below.

```
props.put(AbstractKafkaAvroSerDeConfig.AUTO_REGISTER_SCHEMAS, false);
```

To manually register the schema outside of the application, you can use Control Center.

First, create a new topic called test in the same way that you created a new topic called transactions earlier in the tutorial. Then from the Schema tab, click Set a schema to define the new schema.

# Kafka REST Proxy


Database Changes
Log Events
IoT Data
Web Events
Other Events
DATA INTEGRATION
Hadoop
Database
Data Warehouse
CRM
Other
Confluent Platform
Management & Monitoring
Control Center | Security
Enterprise Operations
Replicator | Auto Data Balancer | Connectors | MQTT Proxy | k8s Operator
Data Compatibility
Schema Registry
Development & Connectivity
Clients | Connectors | REST Proxy | KSQL
Apache Kafka®
Core | Connect API | Streams API
REAL-TIME APPLICATIONS
Transformation
Custom Apps
Analytics
Monitoring
Other
Customer self-managed
Datacenter
Public Cloud
Confluent fully-managed
Confluent Cloud
COMMERCIAL SOFTWARE
COMMUNITY SOFTWARE

The Kafka REST Proxy is part of Confluent Open Source and Confluent Enterprise distributions.

The proxy provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some example use cases are:

- Reporting data to Kafka from any frontend app built in any language not supported by official Confluent clients.
- Ingesting messages into a stream processing framework that doesn't yet support Kafka.
- Scripting administrative actions

## Features

Eventually, the REST Proxy should be able to expose all of the functionality of the Java producers, consumers, and command-line tools. They are:

**Metadata** - Most metadata about the cluster – brokers, topics, partitions, and configs – can be read using GET requests for the corresponding URLs

**Producers** - Instead of exposing producer objects, the API accepts produce requests targeted at specific topics or partitions and routes them all through a small pool of producers.

**Consumers** - The REST Proxy uses consumer api  to implement consumer-groups that can read from topics.

**Data Formats** - The REST Proxy can read and write data using JSON, raw bytes encoded with base64 or using JSON-encoded Avro.

**REST Proxy Clusters and Load Balancing -** The REST Proxy is designed to support multiple instances running together to spread load and can safely be run behind various load balancing mechanisms

# Apache Spark

## Apache Spark

Spark Streaming API enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Data can be ingested from many sources like Kafka, Flume, Twitter, etc., and can be processed using complex algorithms such as high-level functions like map, reduce, join and window.

Finally, processed data can be pushed out to filesystems, databases, and live dash-boards.


Kafka
Flume
HDFS/S3
Kinesis
Twitter
Spark Streaming
HDFS
Databases
Dashboards

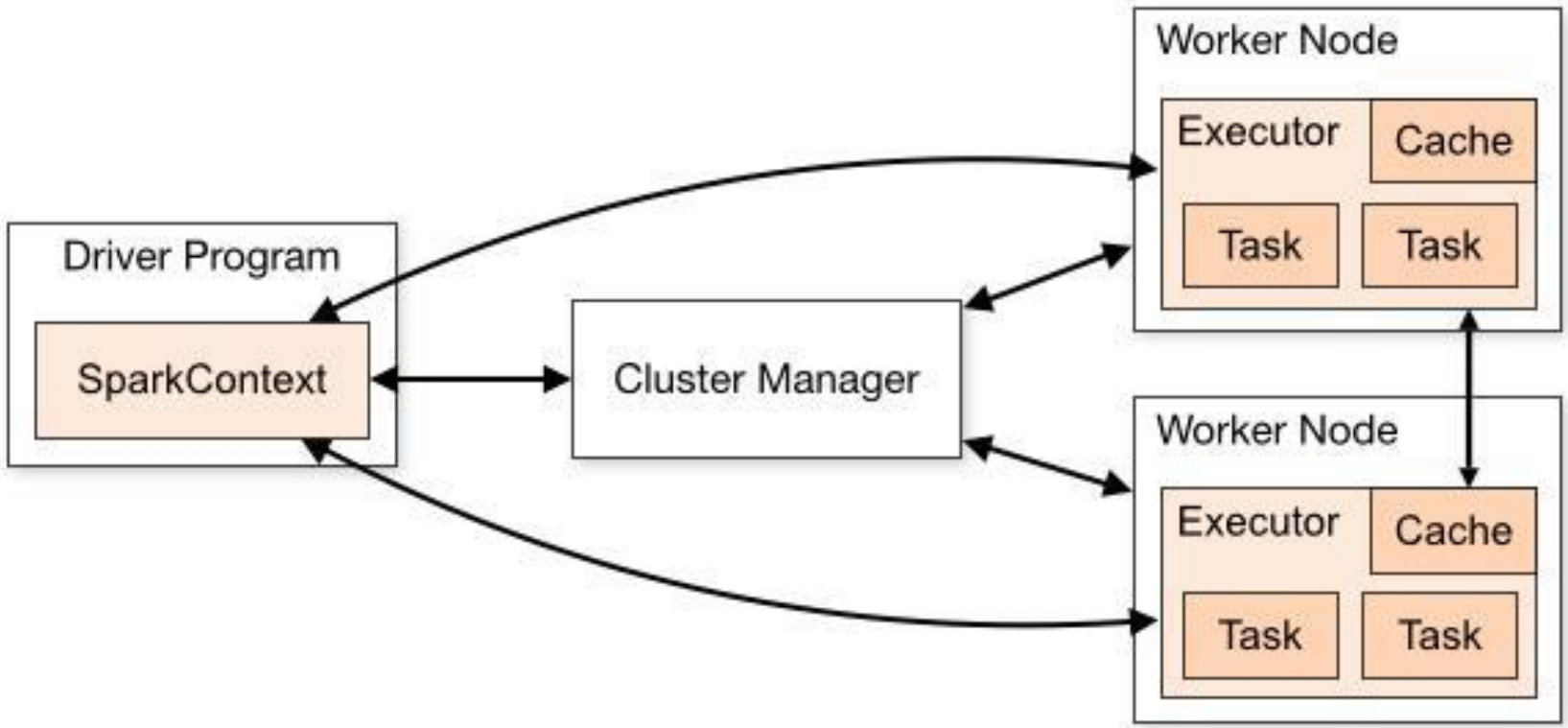
- Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark.
- It is an immutable distributed collection of objects.
- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

**Kafka Integration with Spark**

Kafka is a potential messaging and integration platform for Spark streaming.

Kafka act as the central hub for real-time streams of data and are processed using complex algorithms in Spark Streaming.

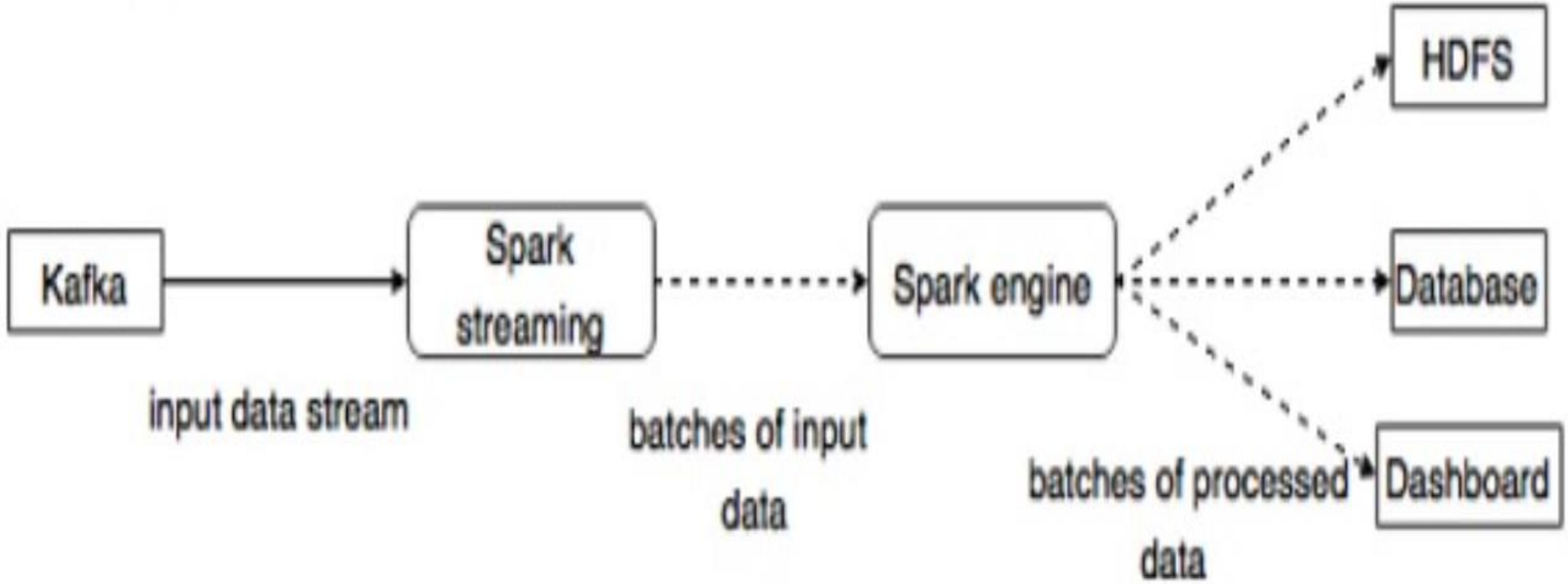
Once the data is processed, Spark Streaming could be publishing results into yet another Kafka topic or store in HDFS, databases or dashboards.


Worker Node
Executor
Cache
Task
Task
Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task

Spark Core uses a master-slave architecture. The Driver program runs in the master node and distributes the tasks to an Executor running on various slave nodes. The Executor runs on their own separate JVMs, which perform the tasks assigned to them in multiple threads.

The Executors execute the tasks and send the result back to the Driver.

The Driver communicates to the nodes in clusters using a Cluster Manager like the built-in cluster manager, Mesos, YARN, etc. The batch programs we write get executed in the Driver Node.


Kafka
Spark streaming
Spark engine
HDFS
Database
Dashboard
input data stream
batches of input data
batches of processed data

DAG(Directed Acyclic Graph) is a graph denoting the sequence of operations that are being performed on the target RDD”.

Spark uses DAG to carry out the in-memory computations

Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

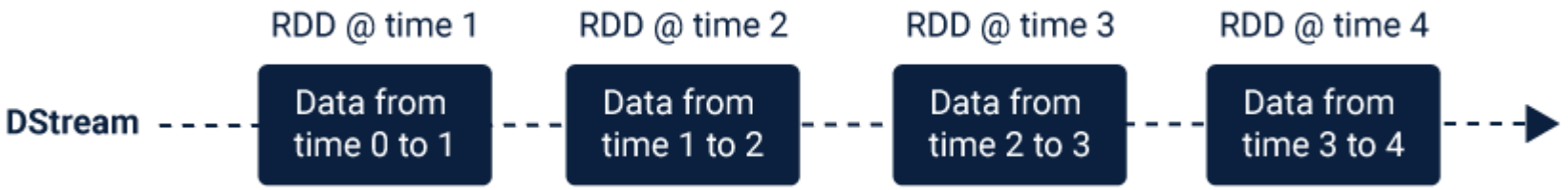
Spark Streaming provides a high-level abstraction called **discretized stream** or DStream, which represents a continuous stream of data.

DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams.

Internally, a DStream is represented as a sequence of RDDs.


Micro batching is a technique where incoming tasks to be executed are grouped into small batches to achieve some of the performance advantage of batch processing, without increasing the latency for each task completion too much

DStreams are sequences of RDDs (Resilient Distributed Dataset), which is multiple read-only sets of data items that are distributed over a cluster of machines. These RDDs are maintained in a fault tolerant manner, making them highly robust and reliable.



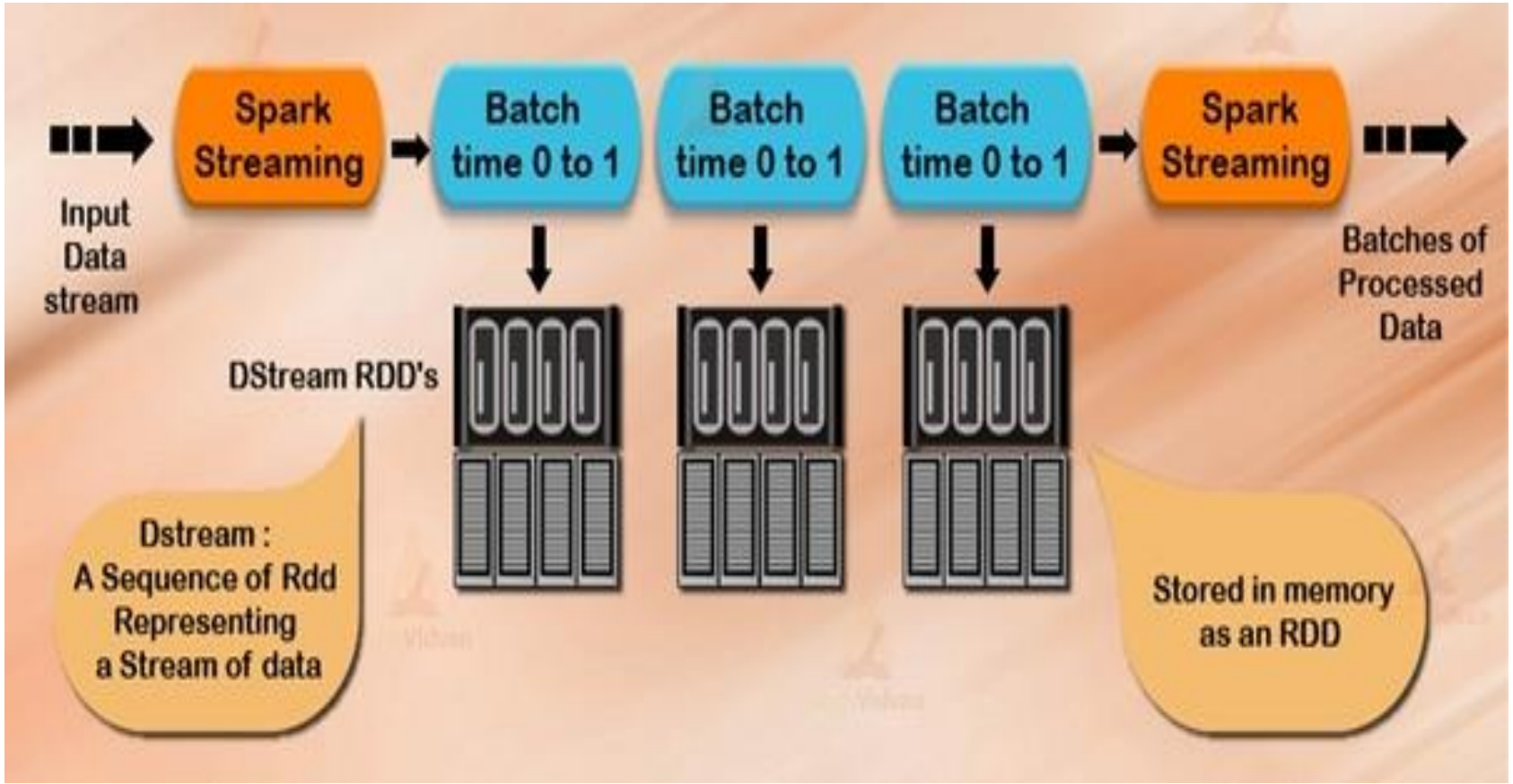

Any operation on a DStream applies to all the underlying RDDs. DStream covers all the details. It provides the developer a high-level API for convenience. As a result, Spark DStream facilitates working with streaming data.

```
JavaDStream<Integer> length = lines.map(x -> x.length());
```

A Discretized Stream (DStream), the basic abstraction in Spark Streaming, is a continuous sequence of RDDs (of the same type) representing a continuous stream of data.

DStreams can either be created from live data (such as, data from HDFS, Kafka or Flume) or it can be generated by transformation existing DStreams using operations such as map, window and reduceByKeyAndWindow.

While a Spark Streaming program is running, each DStream periodically generates a RDD, either from live data or by transforming the RDD generated by a parent DStream.


Input Data stream
Spark Streaming
Batch time 0 to 1
Batch time 0 to 1
Batch time 0 to 1
Spark Streaming
Batches of Processed Data
DStream RDD's
Dstream : A Sequence of Rdd Representing a Stream of data
Stored in memory as an RDD

**processes per data stream(real real-time)**

Kafka Streams is best used in a "Kafka > Kafka" context.

**Data received form live input data streams is Divided into Micro-batched for processing.**

Spark Streaming could be used for a "Kafka > Database" or "Kafka > Data science model" type of context.

The latency for Spark Streaming ranges from milliseconds to a few seconds.

If latency is not a significant issue and we are looking for flexibility in terms of the source compatibility, then Spark Streaming is the best option to go for. Spark Streaming can be run using its standalone cluster mode, on EC2, on Hadoop YARN, on Mesos, or Kubernetes as well.

On the other hand, if latency is a significant concern and one has to stick to real-time processing with time  frames shorter than milliseconds then, we must consider Kafka Streaming.