# Apache Kafka

**The truth is the log**
(Helland 2015)

**Team Members:**

Ankush Sharma

René Gómez

INFO-H-415: Advanced Databases

**Supervised by:**

Esteban Zimányi

# Table of content

## Executive Summary

It is true that the digitalization of companies has lead to multiple data sources with structured and unstructured data. But the problem doesn't stop there, we have also created different specialized tools to store, query and analyse such data. The combination of more data sources and the need to get this data into diverse systems leads to a huge data integration problem.

On the other hand, from the architectural point of view the digitalization of companies and the rise of event data have forced to change from monolithic applications to more scalable systems with Services Oriented Architectures (SOA) and more recently Microservices. When systems reach a critical level of dynamism we have to change our way of modelling and designing them. However, this also increase the complexity of the communication systems required to properly transport data from the multiple sources to the multiple target systems. The appropriate systems architecture for this inherent dynamic nature to complex engineered systems is what is called event driven architecture, built around the production, detection, and reaction to events that take place in time.

Events are important because basically companies are active process, continuously reacting and operating as events occur; the aim of stream processing platforms as Apache Kafka is precisely provide the capacities to process events in real time.

In this document, we explore some concepts behind event data, stream processing and specifically Apache Kafka, its features and the problems it tries to solve. We also present the typical architecture of application using Kafka,, the data abstraction and its importance in the whole Kafka's ecosystem. At the end of the document we present the use cases and some real production environments that evidence Kafka's performance in companies like Twitter and Uber.

# 1. Data integration and stream processing

Software development and infrastructure technologies have mainly focused on how to build applications and the data stores that support those applications in order to provide services. This starts to be a problem when the company grows, requiring lots of applications and services with different technologies and protocols that must be integrated.

Companies frequently offer different business lines that normally require their own infrastructure and again different applications that provide multiple services to their clients. In addition, nowadays it must be considered the existence of several data sources beyond transactional data like sensors, mobile logs, metrics and social networks to name a few. Moreover, with the exponential growth in data, it has become increasingly difficult to build data pipelines that integrate with different data sources. The problem, therefore, is how to build an infrastructure that is:

- Decoupled
- Evolvable
- Operationally transparent
- Resilient to traffic spikes
- Highly available
- Distributed

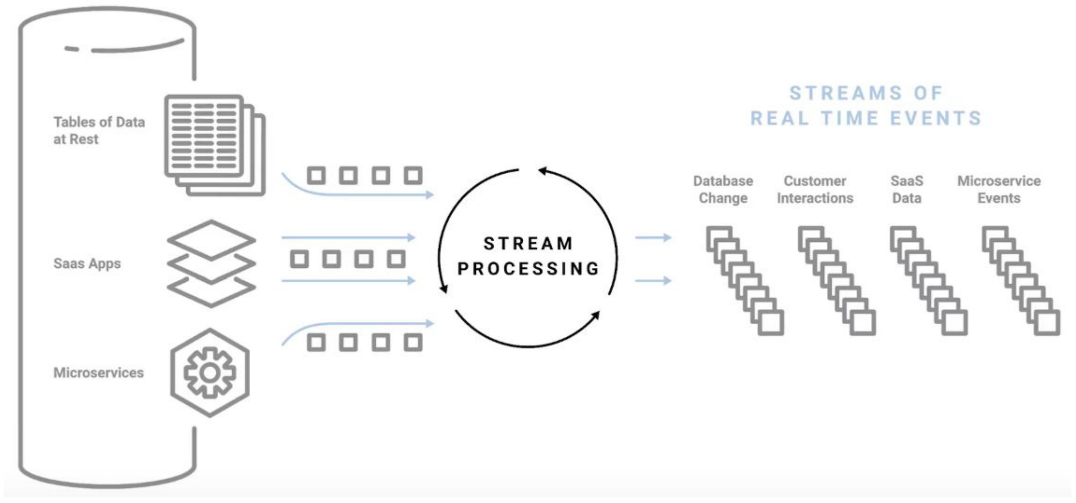
Consider this from a business perspective. More than data stores, a company is an active process, continuously reacting and operating as events occur. In consequence, event centric design have emerged and with it event driven architectures, a design pattern built around the production, detection, and reaction to events that take place in real time (Fowler 2017). Companies are rethinking their business as a stream of events and how to respond to those events. This world view lets companies to model what happens in their business as events: the sales, the orders, the customer experience and behavior are streams of events that again, model the operation of the business. They key is to detect such events, find relations between them and react in a proper manner.

Both situations, data integration and events processing require technological solutions. The data generated continuously by thousands of data sources that send data records (messages or events) simultaneously and normally in small sizes is called **streaming data** (AWS 2018). The ability to process/react in *real time* to messages/events is called **stream processing**. Kafka is a stream processing platform. For this reason, Kafka clusters are part of the data processing architecture of a lot of companies like LinkedIn, Yahoo!, Twitter, Netflix, Spotify, Uber and many more.

Streaming data includes a wide variety of data such as ecommerce purchases, information from social networks or geospatial services from mobile devices. Streaming data needs to be processed sequentially and incrementally in order to, for instance, integrate different applications that consumes the data, or storing and processing the data to update metrics, reports, and summary statistics in response to each arriving data record. It is better suited for real-time monitoring and response functions such as (Data Artisans 2017):

- Classifying a banking transaction as fraudulent based on an analytical model then automatically blocking the transaction
- Sending push notifications to users based on models about their behavior
- Adjusting the parameters of a machine based on result of real-time analysis of its sensor data

Data stores somehow are based on the illusion of static data and use tables as the data abstraction. The purpose of streaming platforms as Kafka is to model change explicitly, thinking in data flows and using a log as data abstraction. In words of Kafka co-creator Jay Kreps (Narkhede et al. 2017) "Kafka got its start as an internal infrastructure system at LinkedIn. Our observation was really simple: there were lots of databases and other systems built to *store* data, but what was missing in our architecture was something that would help us to handle the continuous ***flow*** of data".



Streaming processing

[source (Byzek et al. 2018) ]

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue. This allows multiple application subscribed to a same or different data sources that produce data in one or different topics.
- Store streams of records in a fault-tolerant durable way. This means different clients can access all the events (or a fraction of them) at any time, at their own pace.
- Process streams of records as they occur. This allows filtering, analysing, aggregating or transforming data.

Up to this point, the possibilities that arise using Kafka have been reviewed. However, there are a lot of components and subjacent technologies that must be studied to understand the reasons behind each design decision of Apache Kafka and its use cases.

# 2. Apache Kafka

## 2.1. What is Kafka

The core components and the multiple use cases of Kafka produce different perspectives to define it. At the beginning Kafka was often described as a "distributed commit log", but currently it is defined as a "distributing streaming platform". This is because in its core, Kafka is a publish/subscribe messaging system designed to solve the problem of managing continuous data flows.

It is important to remember that in the publish–subscribe pattern, senders of messages (called the publishers or producers) do not sent messages to specific receivers (called subscribers or consumers). Instead each published message is categorize into classes without knowledge of which subscribers, if any, there may be. Similarly, consumers subscribe to one or more messages classes and only receive messages of those classes, without knowledge of which publishers, if there are any. This pattern enhances scalability since it facilitates to add and remove producers or consumers; the trade-off is the decreased flexibility to modify the publisher and the structure of the published data.

According to it's documentation Kafka is generally used for two broad classes of applications (Apache Software Foundation 2017):

- Building *real-time* streaming data pipelines that reliably get data between systems or applications

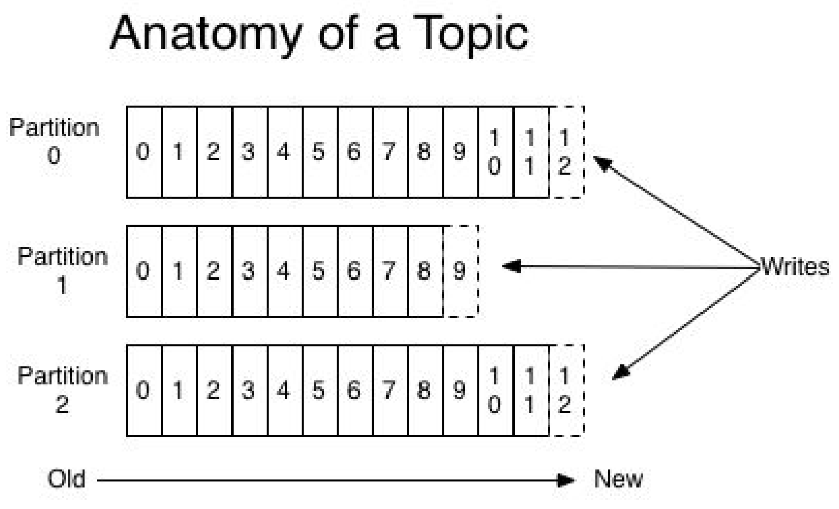
- Building *real-time* streaming applications that transform or react to the streams of data

## 2.2. Components & Key concepts

Kafka use four different Application Programming Interfaces (API) to enable building different and some key concepts components:

### 2.2.1. Producers

The **Producer API** allows an application to publish a **stream of records** to one or more topics. In Kafka, the data records are know as **messages** and they are categorized into **topics**. Think of messages as the data records and topics as a database table.



Anatomy of a Topic

(source SOOK)

Topics are additionally broken down into a number of *partitions* to be stored is a single log. This means messages are written down in partitions in an append-only fashion, and are read in order from beginning to end by consumers. Topics are divided into partitions to allow distribution across multiple servers if it is required. This provides redundancy and scalability. This is why at the beginning Kafka was considered a distributed commit log.

A message is made of up two components:

1. Key: The key of the message. This key would determines the partition the message would be sent to. Careful thought has to be given when deciding on a key for a message. Since a key is mapped to a single partition, an application that pushes millions of messages with one particular key and only a fraction with other key would result in an uneven distribution of load on the Kafka cluster. If the key is set to null, the producer will use a Round robin algorithm and assign a key to random partition
2. Body: Each message has a body. Since Kafka provides APIs for multiple programming languages, the message has to be serialized in a way that the subscribers on the other end can understand it. There are encoding protocols that the developer can use to serialize a message like JSON, Protocol Buffers, Apache AVRO, Apache Thrift, to name a few

### 2.2.2. Consumers

The **Consumer API** on the other hand allows application subscription to one or more topics to store/process/react to the stream of records produced to them. To accomplish this, Kafka adds to each message a unique integer value. This integer is incremented by one for every message that is

This value is known as the **offset** of the message. By storing the offset of the last consumed message for each partition, a consumer can stop and restart without losing its place. This is why Kafka allows different types of applications to integrate to a single source of data. The data can be processed at different rates by each consumer.

Here is also important to know that Kafka allows the existence of **consumer groups**, which are nothing more than consumers working together to *process* a topic. The concept of consumer groups allows to add scale processing of data in Kafka, this is will be reviewed in the next section. Each consumer group identifies itself by a **Group Id**. This value has to be unique amongst all consumer groups. In a consumer group, each consumer is assigned to a partition. If there are more consumers than the number of partitions, some consumers will be sitting idle. For instance, if the number of partitions for a topic is 5, and the consumer group has 6 consumers, one of those consumers will not get any data. On the other hand, if the number of consumers in this case is 3, then Kafka will assign a combination of partitions to each consumer.

When a consumer in the consumer group is removed, Kafka will reassign some partitions that were originally for this consumer to other consumers in the group. If a consumer is added to a consumer group, then Kafka will take some partition from the existing consumers to the new consumer.

### 2.2.3. Brokers

Take into account that in the Kafka cluster a single Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second» (AKF n.d.).

### 2.2.4. Clustering and Leadership Election

As stated before, a Kafka cluster is made up of brokers. Each broker is allocated a set of partitions. A broker can either be the leader or a replica of the partition. Each partition also has a replication factor that goes with it. For instance, imagine that we have 3 brokers namely b1, b2 and b3. A producer pushes a message to the Kafka cluster. Using the key of the message, the producer will decide which partition to send the message to. Let's say the message goes to a partition p1 which resides on the broker b1 which is also the leader of this partition, and the replication factor of 2, and the replica of partition p1 resides on b2.

When the producer sends a message to partition p1, the producer API will send the message to broker b1, which is the leader. If the "acks" property has been configured to "all", then the leader b1 will wait till the message has been replicated to broker b2. If the "acks" has been set to "one", then b1 will write the message in its local log and the request would be considered complete.

In the event of the leader b1 going down, Apache Zookeeper initiates a Leadership election. In this particular case, since we have only broker b2 available for this partition, it becomes the leader for p1.

## 3. Connector API

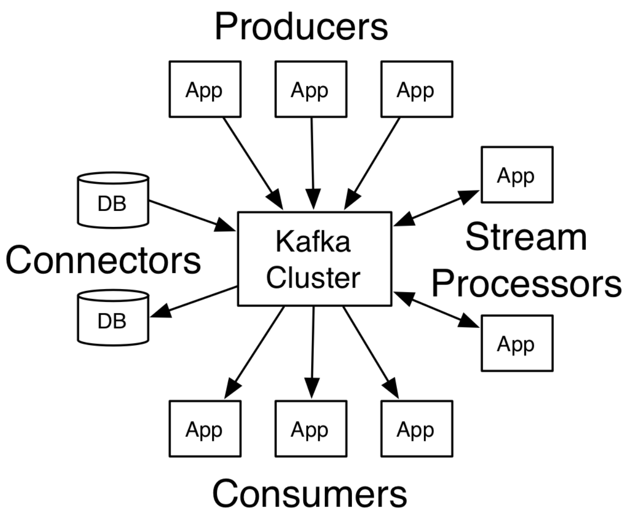
The **Connector API** allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table. If you think in a ETL system, connectors are involved in the Extraction and Load of data (AKF n.d.).

#### 3.1.1. Streams API

The **Streams API** allows an application 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. Basically, the streams API helps to organize the data pipeline. Again, thinking in a ETL system, Streams are involved in data transformation.

#### 3.1.2. Apache Zookeeper

Apache Zookeeper is a distributed, open-source configuration, synchronization service that provides multiple features for distributed applications. Kafka uses it in to manage the cluster, storing for instance shared information about consumers and brokers. The most important role of Apache Zookeeper is to keep a track of consumer group offsets.



Kafka APIs
[source (Apache Software Foundation 2017) ]

# 4. Why use Apache Kafka

## 4.1. Messaging - Publishers/Subscribers

**Read and write streams of data like a messaging system.**

As mentioned before *producers* create new messages or *events*. In general, a message will be produced to a specific topic. In the same way, *consumers (also known as subscribers)* read messages. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages.

Here is an important concept to explore: A message queue allows you to scale processing of data over multiple consumer's instances that process the data. Unfortunately, once a message is consumed from the queue the message is not available anymore for others consumers that may be interested in the same message. Publisher/subscriber in contrast allows you to *publish* broadcast each message to a list of *consumers or subscribers*, but by itself does not scale processing. Kafka offers a mix of those two messaging models: Kafka publishes messages in topics that broadcast all the messages to different consumer groups. The consumer group acts as a message queue that divides up processing over all the members of a group.

## 4.2. Store

**Store streams of data safely in a distributed, replicated, fault-tolerant cluster.**

A file system or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically (Narkhede et al. 2017). This last point is important, in a message queue once the message is consumed it disappears, a log in contrast allows to maintain the message (or events) during a configurable time period, this is known as *retention*.

Thanks to the retention feature, Kafka allows different applications to consume and process messages at their own pace, depending for instance in its processing capacity or its business purpose. In the same way, if a consumer goes down, it can continue to process the messages in the log once it is recovered. This technique is called event sourcing, which is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future (Fowler 2017).

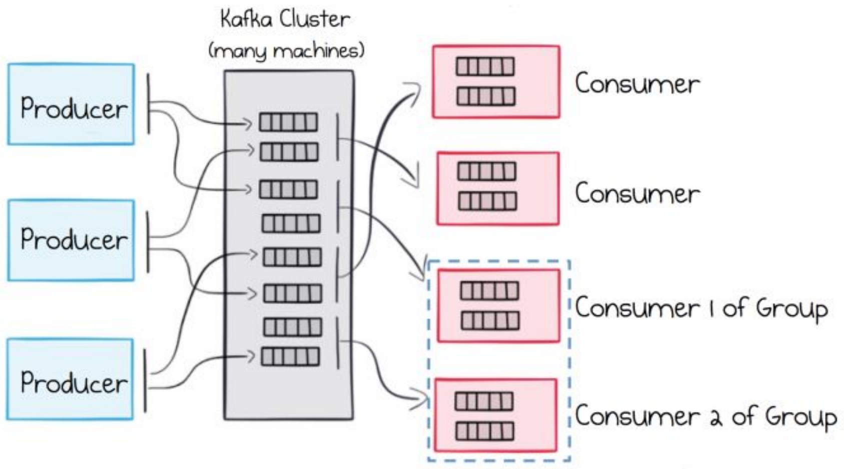
In addition, since the data is distributed within the system it provides additional protection against failures, as well as significant opportunities for scaling performance (Helland 2015). Kafka is therefore a kind of special purpose distributed file system dedicated to high-performance, low-latency commit log storage, replication, and propagation. This doesn't mean Kafka purpose is to replace storage systems, but it results helpful in keeping data consistency in a distributed set of applications.

## 4.3. Process

**Write scalable stream processing applications that react to events in real-time.**
As previously described, a stream represents data moving from the producers to the consumers through the brokers. Nevertheless, it is not enough to just read, write, and store streams of data; the purpose is to enable real-time processing of streams. Kafka allows building very low-latency pipelines with facilities to transform data as it arrives, including windowed operations, joins, aggregations, etc.

## 4.4. Kafka features

Kafka offer a lot of out-of-the-box features that allow developers to focus in the business implementation and not in the underlying details of streaming processing.



Producers spread messages over many partitions, on many machines, where each partition is a little queue. Load balanced consumers (denoted a Consumer Group) share the partitions between them.

[source (Byzek et al. 2018) ]

### 4.4.1. Multiple Producers

Kafka is able to seamlessly handle multiple producers that help to aggregate data from many data sources in a consistent way. A single producer can send messages to one or more topics. Some important properties that can be set at the producer's end are as follows:

- acks: The number of acknowledgements that the producer requires from the leader of the partition after sending a message. If set to 0, the producer will not wait for any reply from the leader. If set to 1, the producer will wait till the leader of the partition writes the message to its local log. If set to all, the producer will wait for the leader to replicate the message to the replicas of the partition
- batch.size: The producer will batch messages together as set by this parameter in bytes. This improves performance as now a single TCP connection will send batches of data instead of sending one message at a time
- compression.type: The Producer API provides mechanisms to compress a message before pushing it to the Kafka cluster. The default is no compression, but the API provides gzip, snappy and and lz4

### 4.4.2. Multiple Consumers

In the same way, the publish/subscribe architecture allows multiple consumers to process the data. Each message is broadcasted by the Kafka cluster to all the subscribed consumers. This allows to connect multiple applications to the same or different data sources, enabling the business to connect new services as they emerge. Furthermore, remember that for scaling up processing, Kafka provides the consumer groups. Whenever a new consumer groups comes online, and is identified by its group id, it has a choice of reading messages from a topic from the first offset, or the latest offset. Kafka also provides a mechanism to control the influx of messages to a Kafka. Some interesting properties that could be set at the consumers end are discussed below:

- auto.offset.reset: One can configure the consumer library of choice to start reading messages from the earliest offset or the latest offset. It is useful to use "earliest" if the consumer group is new and wants to processing from the very beginning
- enable.auto.commit: If set to true, the library would commit the offsets periodically in the background, as specified by the auto.commit.interval.ms interval
- Max.poll.records: The maximum number of records returned in a batch

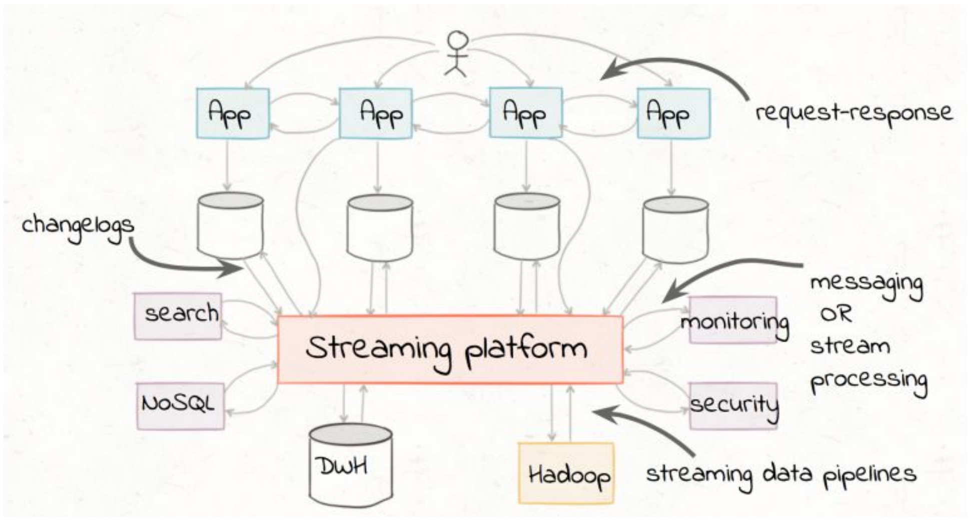
### 4.4.3. Disk-Based Retention

- log.retention.bytes: The maximum size a log for a partition can go to before being deleted
- log.retention.hours: The number of hours to keep a log file before deleting it (in hours), tertiary to log.retention.ms property
- log.retention.minutes: The number of minutes to keep a log file before deleting it (in minutes), secondary to log.retention.ms property. If not set, the value in log.retention.hours is used
- Log.retention.ms: The number of milliseconds to keep a log file before deleting it (in milliseconds), If not set, the value in log.retention.minutes is used

### 4.4.4. High performance

Kafka's flexible allows adding multiple brokers to makes scale horizontally. Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole. All these features converge to improve Kafka's performance.

## 5. Typical architecture in real scenarios

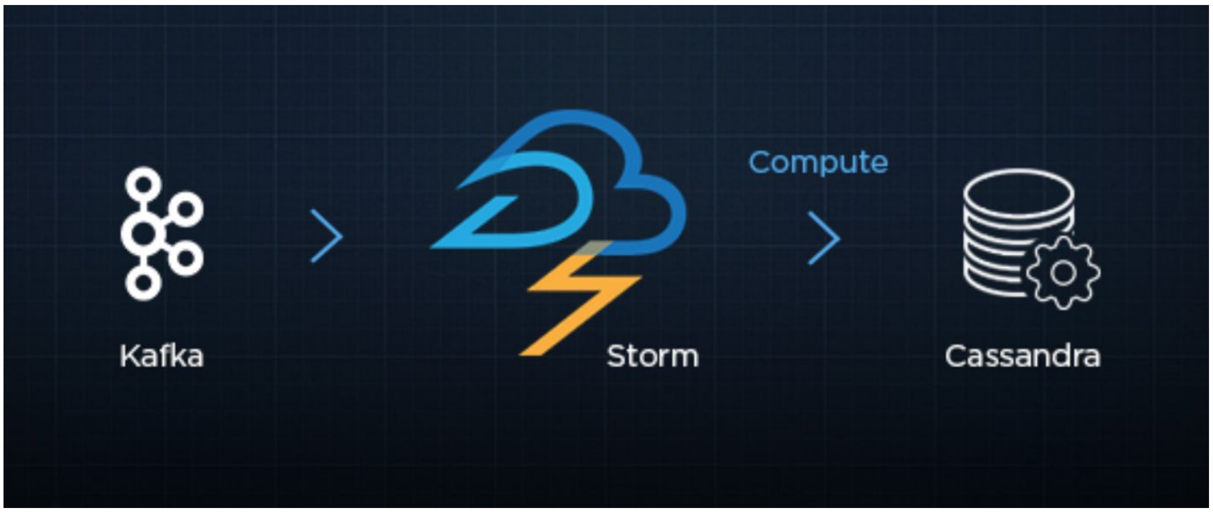


[source (Byzek et al. 2018) ]

To get an idea of a general implementation, consider a system that manages thousands of request/response from different applications. Kafka provides connectors to extract data from those multiple sources, the Kafka Streams API performs transformations and analysis (for instance fraud detection, or monitoring user behaviour) in your core applications, and more Kafka connectors to load transformed data to another systems (for instance data warehouses, hadoop, etc.).

It is important to highlight again that Kafka is not a tool for batch processing (as for instance Apache Spark is), rather than that, there are two goals: real time processing (or stream processing) in one hand and integrate data to later use the batch analysis tools in the other. The idea with the former is to handle lots of events, associate events from the same or different sources and react to them in real time, this create the business capacity to eliminate threats or benefit from opportunities. Notice as well that normally companies have both objectives, and that's why Kafka becomes relevant in managing data and normally is present on both sides of data processing. To illustrate more this point, some real companies are presented in the next sections.

### 5.1. Answers - Twitter



Twitter use Kafka for two different tasks: To help them to receive, retain (in case of failures) and store the data. They also use another subscriber to process data in real time. Here are some aparts from their engineering blog (Twitter Engineering n.d.):

**Archival**

*Because Kafka writes the messages it receives to disk and supports keeping multiple copies of each message, it is a durable store. Thus, once the information is in it we know that we can tolerate downstream delays or failures by processing, or reprocessing, the messages later (...) we configure our Kafka cluster to retain information for a few hours (enough time for us to respond to any unexpected, major failures) and get the data to our permanent store, Amazon Simple Storage Service (Amazon S3), as soon as possible.*

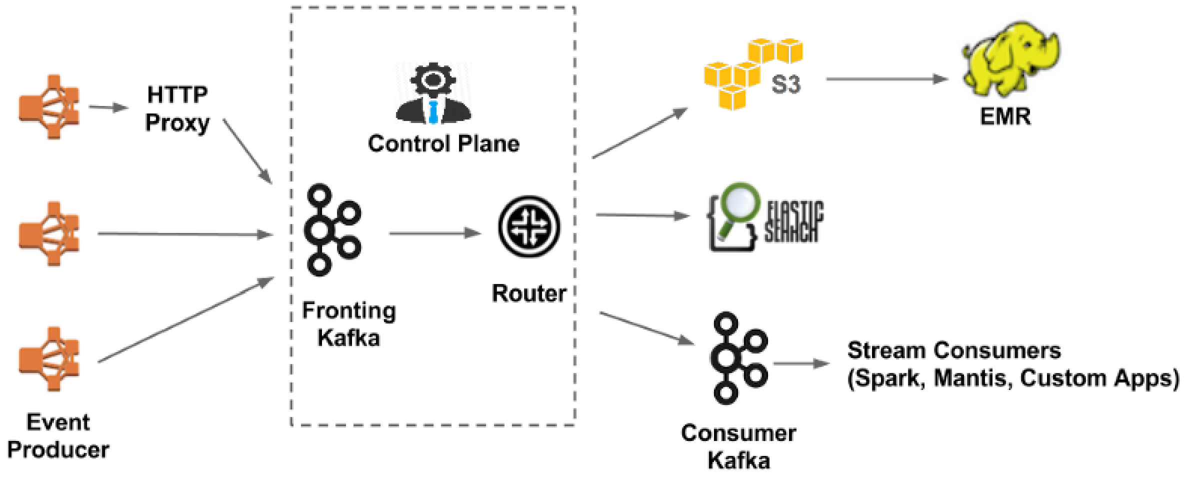
**Speed computation**

*An independent Storm topology consumes the same Kafka topic as our archival topology and performs the same computations that our MapReduce jobs do, but in real time. The outputs of these computations are written to a different independent Cassandra cluster for real-time querying.*

This is a regular architecture for real solutions in companies with a huge demand for storing and processing data. Notice here Kafka helps to mitigate the load on the database and enable real time processing. This is why we consider this technology so interesting for advanced databases course. We made a demo using a similar topology to show how slow and fast consumers use the same source of data.

## 5.2. Netflix

"We currently operate 36 Kafka clusters consisting of 4,000+ broker instances for both Fronting Kafka and Consumer Kafka. We've achieved a daily data loss rate of less than 0.01%. Metrics are gathered for dropped messages so we can take action if needed" (Netflix Technology Blog 2016).



Here are some statistics about our data pipeline:

- ~500 billion events and ~1.3 PB per day
- ~8 million events and ~24 GB per second during peak hours

There are several hundred event streams flowing through the pipeline. For example:

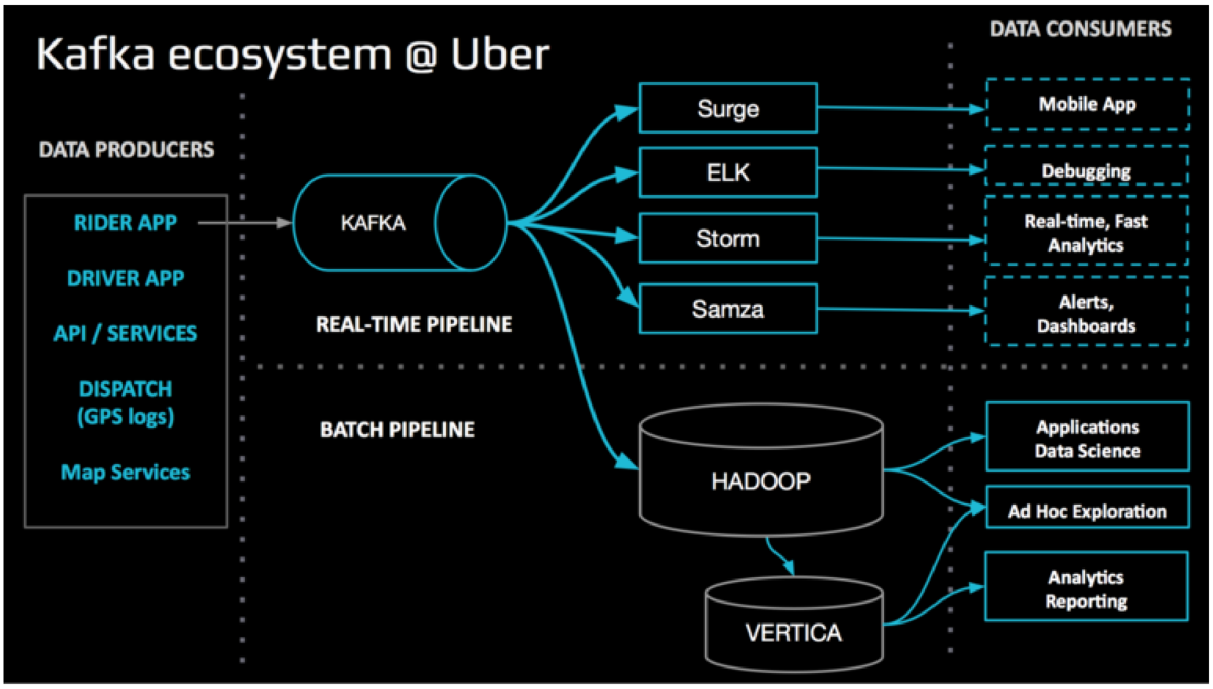
- Video viewing activities
- UI activities
- Error logs
- Performance events

- Troubleshooting & diagnostic events

### 5.3. Uber

As Uber continues to scale, our systems generate continually more events, interservice messages, and logs. Those data needs go through Kafka to get processed (Uber Engineering n.d.). Uber scaled its real time Infrastructure to trillion events per day.

Again, you can find a similar architecture using batch and realtime pipelines.



## 6. Kafka Demo Application

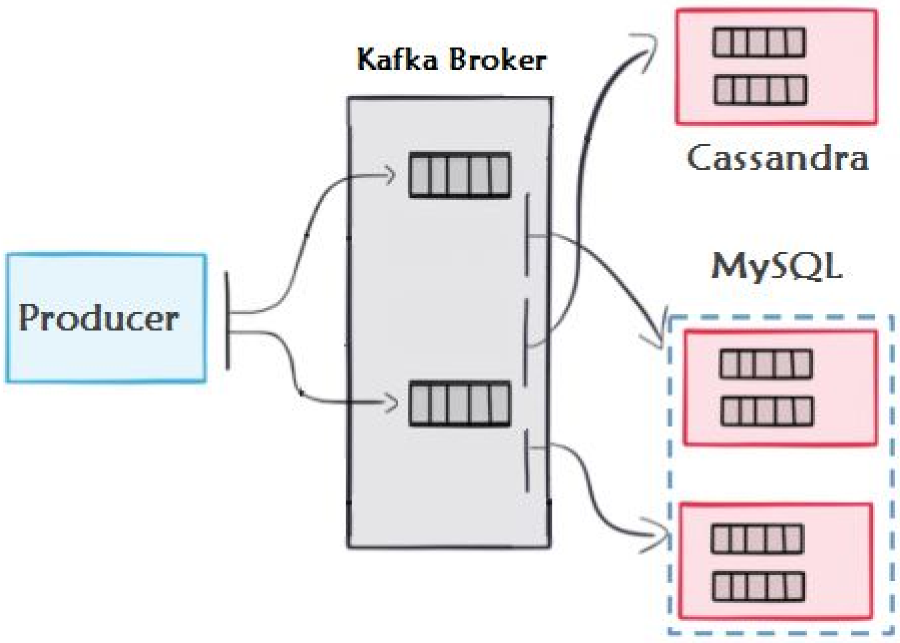
### 6.1. About

Our application will model the simplest form of event in digital advertising domain, the “click” event. A click is an event that can originate anywhere on Earth, on a mobile app or a desktop. The source of the click is this case the publisher. For instance, the source can be google, or facebook. A click is attributed to a campaign. A click has the following properties:

1. uuid: a unique identifier for each click of type UUID
2. campaignId: a unique identifier for a campaign of type Integer
3. pubId: a unique identifier for a publisher of type integer
4. timestamp: the date and time the server reacted to the click
5. ip: the source of the click
6. city: the city the click originated  from
7. country: the country the click originated from
8. browser: the browser the click originated from
9. platform: the platform like android or iOS or macOS
10. platformVersion: the version of the platform like android 5

The Kafka producer will generate fake click events and push them to Kafka to a topic called “clicks”. We also have two consumer groups:

1. kc-clicks-persistor: This consumer extracts (or streams) events from the topic called clicks and saves these events to Cassandra
2. kc-clicks-mysql: This consumer performs the same functionality as kc-clicks-persistor, the only difference being that it saves the events to MySQL.



Our goal with this application is to show how a producer pushes messages to Kafka and how Kafka can stream those events in “near real time” to different consumer groups. You can also use this application to experiment:

1. What happens when a new consumer group goes live
2. What happens when a consumer within a consumer group goes down
3. What happens if one consumer is fast whereas the other one is really slow. For the sake of generality, we assume that in terms of raw write performance, MySQL will be slower than Cassandra, and hence is a good use case to simulate our tests

Environment configuration of the machine (henceforth called the environment) used to develop the application is as follows: MacBook Pro, 2.2 GHz Intel Core i7 . For the producer and consumer API’s, Vert.x Kafka library and Spring Boot was used. For Cassandra and MySQL, Spring Boot Cassandra and Spring Boot JPA were used.

## 6.2. Kafka Cluster

To install Apache Kafka and Apache Zookeeper on a macbook pro, simply run the commands in sequence:

```
brew install zookeeper
brew install kafka
```

The environment has version 0.11 installed. To run kafka, execute the following commands in sequence:

```
cd /usr/local/Cellar/kafka/0.11.0.0/libexec
bin/kafka-server-start.sh config/server.properties
```

Before setting up the producer, let’s first create the topic “clicks” .

```
cd /usr/local/Cellar/kafka/0.11.0.0/libexec
 bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic clicks
```

## 6.3. Setting up the producer

The producer in this particular case is a program that generates fake clicks every millisecond and keeps pushing them to Kafka. We showcase only a small part of the actual program. To view and run the full code, visit this link.

```
@Component
@Slf4j
class ClickLoader {

    private final ClickService clickService

    ClickLoader(ClickService clickService) {
        this.clickService = clickService
    }

    @Scheduled(fixedRate = 1L)
    void generateFakeClicks() {
        log.info "Generating clicks"
        CAMPAIGN_IDS.each { campaignId -> // ----> 1
            PUB_IDS.each { pubId ->       // ----> 2
                def uuid = UUID.randomUUID()
                def ip = IPUtil.generateRandomIp()
                def city = getRandom(CITIES)
                def country = COUNTRIES[city]
                def platform = getRandom(PLATFORM)
                def platformVer = getRandom(PLATFORM_VERSION)
                def browser = getRandom(BROWSERS)
                def click = new Click(   // ---> 3
                        uid: uuid,
                        timestamp: LocalDateTime.now(),
                        campaignId: campaignId,
                        pubId: pubId,
                        ip: ip,
                        country: country,
                        city: city,
                        platform: platform,
                        platformVer: platformVer,
                        browser: browser
                )
                log.debug "Click {}" , click
                clickService.pushToKafka(click) // -----> 4
            }
        }
    }

    private static String getRandom(Set<String> countries) {
        int random = (int) (Math.random() * (countries.size()))
        countries[random]
    }
}
```

Let's break it down one by one:

1. We loop over each campaign id. These campaign ids can have values ranging from 1 to 10
2. We Loop over each pub id. Pub ids can have values ranging from 1 to 5
3. We generate a click object
4. We push click object to Kafka

The method *pushToKafka* encodes the Click object to JSON before pushing to Kafka. This is how it does it:

```
@Service
@Slf4j
class ClickServiceImpl implements ClickService {

    private final KafkaProducer<String, byte[]> kafka

    ClickServiceImpl(KafkaProducer<String, byte[]> kafka) {
        this.kafka = kafka
    }

    @Override
    Future<Void> pushToKafka(Click click) {
        Assert.notNull(click, "click cannot be null")
        def json = Json.encode(click)
        def bytes = json.bytes
        def future = Future.future()
        def record = new KafkaProducerRecordImpl<String,byte[]>(KafkaTopics.CLICK,
                                                              bytes)
        kafka.write(record, { done ->
            if(done.succeeded()) {
                log.debug "Successfully pushed Click to Kafka"
                future.complete()
            }
            else {
                log.error "Could not push Click event to Kafka. Error {}",  done.cause()
            }
        }).exceptionHandler { ex ->
            log.error "", ex
        }
        return future
    }
}
```

## 6.4. Cassandra consumer

This consumer subscribes to the topic “clicks” with group id “kc-clicks-persistor”. The code to subscribe to Kafka is shown below:

```
@Configuration
class KafkaConfig {

    @Autowired
    private Vertx vertx

    @Bean("kafka")
    KafkaConsumer<String,byte[]> kafkaConsumer() {
        def config = new Properties()
        def host = "localhost:9092"
        config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, host)
        config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                   StringDeserializer.class)
        config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                   ByteArrayDeserializer.class)
        config.put(ConsumerConfig.GROUP_ID_CONFIG, "kc-clicks-persistor")
        config.put("enable.auto.commit", "false")  // ---> 1
        config.put("auto.offset.reset", "earliest")
        return KafkaConsumer.create(vertx, config)
    }
}
```

Our Kafka handler streams events from Kafka and saves to Cassandra

```
@Component
@Slf4j
class ClickStreamKafkaListener  {

    private final KafkaConsumer<String, byte[]> kafka
    private final ClickRepository clickRepository
    ClickStreamKafkaListener(
            KafkaConsumer<String, byte[]> kafkaConsumer,
            ClickRepository clickRepository
    )
    {
        this.kafka = kafkaConsumer
        this.clickRepository = clickRepository
    }

    @PostConstruct
    void streamClickEvents() {
        kafka.handler { record ->
            def offset = record.offset()
            def payload = record.value()
            def json = new String(payload)
            def partition = record.partition()

            log.debug "Partition: {}, OffSet: {}", partition, offset
            def click = Json.toObject(json, Click)
            clickRepository.save(click) // ----> 2

            kafka.commit { ar ->  // ----> 3
                if(ar.succeeded()) {
                    log.debug "Message with Offset {} HAS BEEN commited", offset
                }
                else {
                    println "Could not commit offset {}"
                }
            }
        }

        kafka.subscribe(KafkaTopics.CLICK)
    }

    @PreDestroy
    void cleanup () {
        kafka.unsubscribe { ar ->
            if(ar.succeeded()) {
                log.debug "Successfully unsubscribed Consumer"
            }
            else {
                log.error "Unable to unsubscribe Consumer"
            }

        }
    }
}
```

Let’s go over each step one by one:

1. We set `enable.auto.commit` to false. We explicitly commit each message as it when it arrives to Kafka
2. Save the click object to cassandra
3. Commit the offset to Apache Kafka

### 6.5. MySQL consumer

The MySQL consumer works the same way as kc-clicks-persistor.

```
interface ClickRepository extends JpaRepository<Click, Integer> {
}
```

JPARepository is an abstraction for MySQL

The only difference is that ClickService persists the event to MySQL.

```
@Service
@Slf4j
class ClickServiceImpl implements ClickService {

    private final ClickRepository clickRepository

    ClickServiceImpl(ClickRepository clickRepository){
        this.clickRepository = clickRepository
    }

    @Override
    void save(Click click) {
        Assert.notNull(click,'click cannot be null')
        clickRepository.save(click)
    }
}
```

# 7. CONCLUSIONS

We saw how the growth of data have led companies to change their way of thinking about software systems to a more event driven approach.  As another consequence, there have emerged specialized data stores and search engines that need to be fed data in real time for real time analytics, search indexing, machine learning etc. Being a distributed streaming platform, we saw how Apache Kafka can help companies mitigate these issues by allowing them to build loosely coupled systems  and sophisticated data pipelines allowing them to perform on ETL’s on data that is constantly moving through the system. But more than that, we saw how Kafka can also be tuned as a permanent data store for events, leaving it for the consumer applications to read data as far back in time as they desire. With it, Kafka also provides strong fault tolerant guarantees, thereby making it a nice fit for big data applications in a distributed environment. We created a simple data pipeline, with two consumer applications storing in different data stores, namely Cassandra and MySQL in order to experiment with consumer groups, partitions and offsets and Apache Kafka interaction with different consumers. We tested Kafka's capacities in distribution and fault tolerance.

## 8. REFERENCES

AKF, Apache Kafka Documentation. *Apache Kafka Documentation*. Available at: https://kafka.apache.org/intro [Accessed November 30, 2018].

Apache Software Foundation, 2017. Apache Kafka Docs. *Apache Kafka Docs*. Available at: https://kafka.apache.org/intro [Accessed November 30, 2018].

AWS, 2018. What is Streaming Data? – Amazon Web Services (AWS). *Amazon Web Services, Inc.* Available at: https://aws.amazon.com/streaming-data/ [Accessed November 30, 2018].

Byzek, Y. et al., 2018. Confluent Blog: Apache Kafka Best Practices, Product Updates & More. *Confluent*. Available at: https://www.confluent.io/blog/ [Accessed December 1, 2018].

Data Artisans, 2017. data Artisans - Apache Flink. *Data Artisans*. Available at: https://data-artisans.com/ [Accessed November 28, 2018].

Fowler, M., 2017. What do you mean by "Event-Driven"? *martinfowler.com*. Available at: https://martinfowler.com/articles/201701-event-driven.html [Accessed November 30, 2018].

Helland, P., 2015. Immutability Changes Everything. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*. 7th Biennial Conference on Innovative Data Systems Research (CIDR). Salesforce, pp. 1–6.

Narkhede, N., Shapira, G. & Palino, T., 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*, "O'Reilly Media, Inc."

Netflix Technology Blog, 2016. Evolution of the Netflix Data Pipeline – Netflix TechBlog – Medium. *Medium*. Available at: https://medium.com/netflix-techblog/evolution-of-the-netflix-data-pipeline-da246ca36905 [Accessed December 11, 2018].

Twitter Engineering, Handling five billion sessions a day – in real time. *Twitter Engineering Blog*. Available at: https://blog.twitter.com/engineering/en_us/a/2015/handling-five-billion-sessions-a-day-in-real-time.html [Accessed December 11, 2018].

Uber Engineering, Kafka Archives | Uber Engineering Blog. *Uber Engineering Blog*. Available at: https://eng.uber.com/tag/kafka/ [Accessed December 11, 2018].