I

Streaming 1: Real time data processing of crypto-currencies

Author : Aakash Mandlik

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

Streaming data is information that is delivered immediately after collection. There is no delay in the timeliness of the information provided. Streaming data is often used for navigation or tracking.

What are we solving?

Building a real time dashboard to see the trends of crypto currencies fluctuations in real time. The dashboard should be refreshed in a few seconds and the graphs and charts should be refreshed in accordance with the real time data.
For the scope of this session, we are going to cover the basics of the streaming data ecosystem with a partial solution to our problem statement. The other partial will be covered in the next session.
Lets gear up and capture this knowledge in **real time**.

Where is the data coming from?

- We will be using yahoo finance data apis for getting the real time data for bitcoin crypto currency. The data has other crypto currencies data as well.
- Right now we are going with some of the popular ones like **bitcoins, Ethereum, Tether, Binance Coin, USD Coin**.

Datetime	Open	High	Low	Close	Adj Close	Volume
2022-03-09 11:39:00+00:00	42073.042969	42073.042969	42073.042969	42073.042969	42073.042969	6000640
2022-03-09 11:40:00+00:00	42065.621094	42065.621094	42065.621094	42065.621094	42065.621094	47939584
2022-03-09 11:41:00+00:00	42060.316406	42060.316406	42060.316406	42060.316406	42060.316406	0
2022-03-09 11:42:00+00:00	42064.351562	42064.351562	42064.351562	42064.351562	42064.351562	0
2022-03-09 11:43:00+00:00	42067.289062	42067.289062	42067.289062	42067.289062	42067.289062	0

- Every row above explains the bitcoin value for that particular minute. Primary reason for considering this dataset for analysis is because the fluctuation in crypto currencies happens every few seconds, thus making an ideal case for our use case.
- Before going to the solution directly, let's understand what real time data is.

What is real time data?

- Real-time data is data that is available as soon as it's created and acquired.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- Rather than being stored, data is forwarded to users as soon as it's collected and is immediately available — without any lag — which is crucial for supporting live, in-the-moment decision making.
- This data is at work in virtually every part of your lives, powering everything from bank transactions to GPS to the many COVID-19 maps that have emerged during the pandemic.

Why is real time data useful for organizations?

- **A more proactive approach**: For much of IT's history, data analytics has been a reactive process. Current technology, however, enables a more proactive posture through data mining, predictive strategies and machine learning algorithms to identify patterns that weren't easily uncovered using previously available methods and tools.
- **Greater visibility**: Modern IT infrastructures are a heterogeneous mix of physical and virtual servers, public and private clouds, databases and applications with complex interdependencies — all of which create a host of visibility challenges for IT teams. Real-time data coupled with a unified monitoring and analytics tool provides teams with a single, comprehensive view into their environment and makes it easier to correlate data across elements and produce actionable insights.
- **Reduced downtime**: Real-time data provides the grist to predict, prevent or detect failing components, service spikes, security threats and other infrastructure issues. By anticipating or quickly identifying these types of problems, teams can resolve them before they significantly impact customers.
- **Cost savings**: Real-time data analytics helps reduce IT infrastructure costs by giving administrators more insight into resource allocation and consumption, system health and security weaknesses, among other things. With the opportunity to optimize infrastructure elements, ITOps can achieve significant cost savings.

Lets understand more about the real time ecosystem in our next segment. Stay tuned.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

What is the real time ecosystem: Part 1?

Let's look at what is the real time ecosystem and what all it consists of? We will discuss the first half of the ecosystem in this lecture. Later half will be in scope for the next session.

Messaging Queues:

- A messaging queue which keeps the data, enabling the producers and consumers to work independently.
- In traditional systems, the producers and consumers were coupled together, such that if a consumer is down, then the message will be lost.
- Newer messaging systems like Kafka provided an async way , such that even if a consumer is down , the producer will keep producing messages and none of the messages will be lost.
- What are the different messaging queues available?
    - **Kafka** :
        - It is an open-source software platform developed by the Apache Software Foundation.
        - Kafka requires you to manage the infrastructure(brokers).
    - **Kinesis** :
        - This is also a messaging queue framework, similar to Kafka.
        - Amazon Kinesis is fully managed and runs your streaming applications without requiring you to manage any infrastructure.
        - It is developed by Amazon and comes under AWS hood. It is not open source.
    - **Pub/Sub**:
        - This is also a messaging queue framework, similar to Kafka.
        - This is provided by google under google cloud platform hood.
        - It runs your streaming applications without requiring you to manage the infrastructure.
        - It is not open source.

What are we going to use?

We will be using the following stack for our use case :

- **Kafka** : Kafka is considered because it's an open source streaming messaging queue.

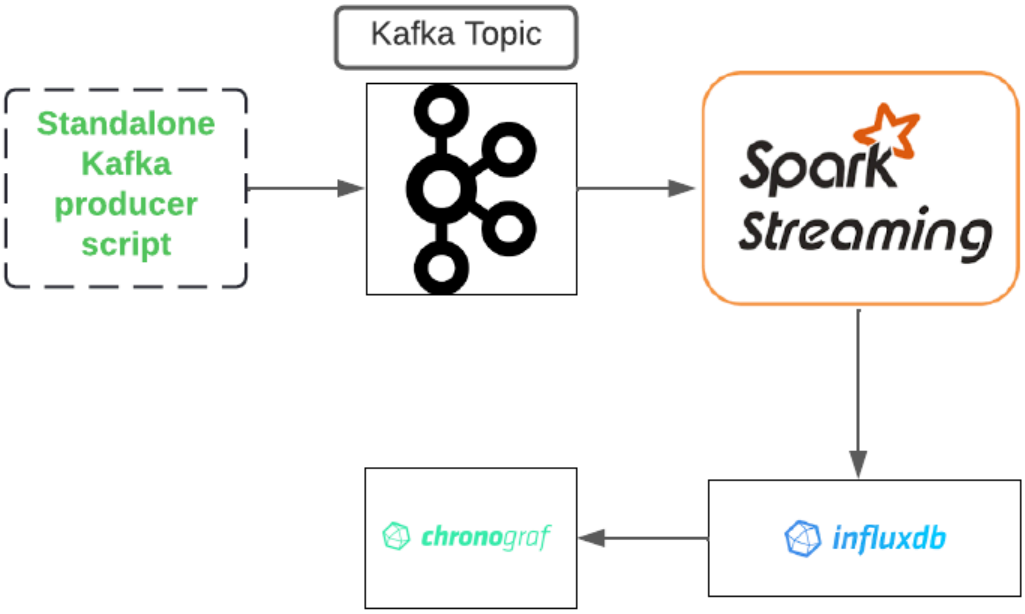
- **Spark streaming** : Spark streaming is considered because it is the best tool to work with micro batches.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- **InfluxDb :** InfluxDb is considered because it supports a wide range of data types like int, float, string etc, thus a wide range of data can be saved in influxDb.

- **Grafana** : Grafana is considered because with it multiple data sources are supported. It also supports having non time-series data as a source to the dashboard.



As part of the current session, we are going to cover the partial pipeline. We will cover the data produced at the kafka message queue. After which we will read from kafka in micro-batches and write to InfluxDb. Using InfluxDb to create real time dashboards will be covered in the next session.

Apache Kafka: A messaging system:

Apache Kafka originated at LinkedIn and later became an open sourced Apache project in 2011, then First-class Apache project in 2012.

Apache Kafka is a publish-subscribe based fault tolerant messaging system. It is fast, scalable and distributed by design.

What is a Messaging System?

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it.

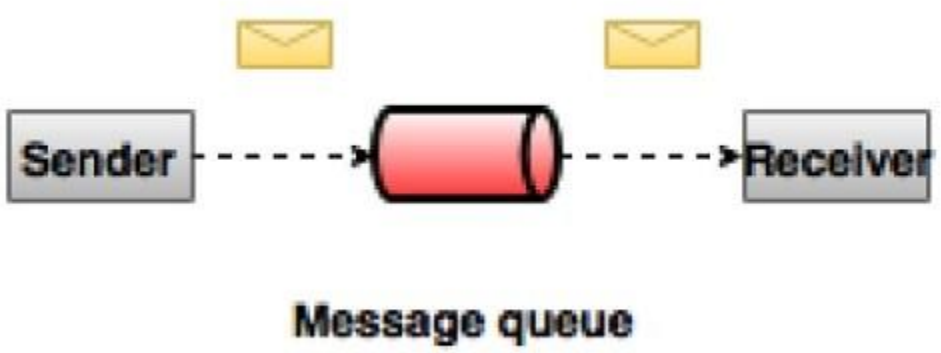
Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging systems.
- Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow pub-sub.

Point to Point Messaging System(Non-Kafka):

- In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only.
- Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

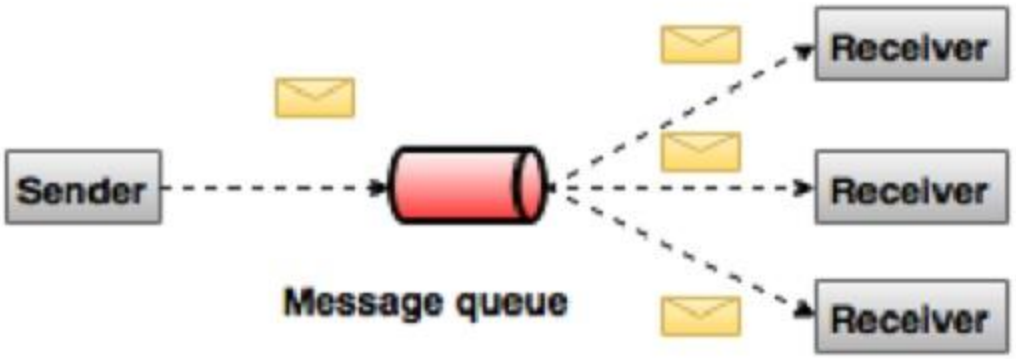


Publish-Subscribe Messaging System(Kafka):

- In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point systems, consumers can subscribe to one or more topics and consume all the messages in that topic.
- In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers.
- A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I



What is Kafka?

- If you enter the Kafka website, you'll find the definition of it right on the first page: ***A distributed streaming platform***
- What is an "A distributed streaming platform"? First, we need to define what a stream is. Streams are just infinite data, data that never end. It just keeps arriving, and you can process it in real-time.
- And distributed? Distributed means that Kafka works in a cluster, each node in the cluster is called Broker. Those brokers are just servers executing a copy of apache Kafka.
- So, basically, Kafka is a set of machines working together to be able to handle and process real-time infinite data.
- Its distributed architecture is one of the reasons that made Kafka so popular. The Brokers is what makes it so resilient, reliable, scalable, and fault-tolerant. That's why Kafka is so performer and secure.

How Kafka Works?

- Apache Kafka is a publish-subscribe-based durable messaging system. A messaging system sends messages between processes, applications, and servers.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

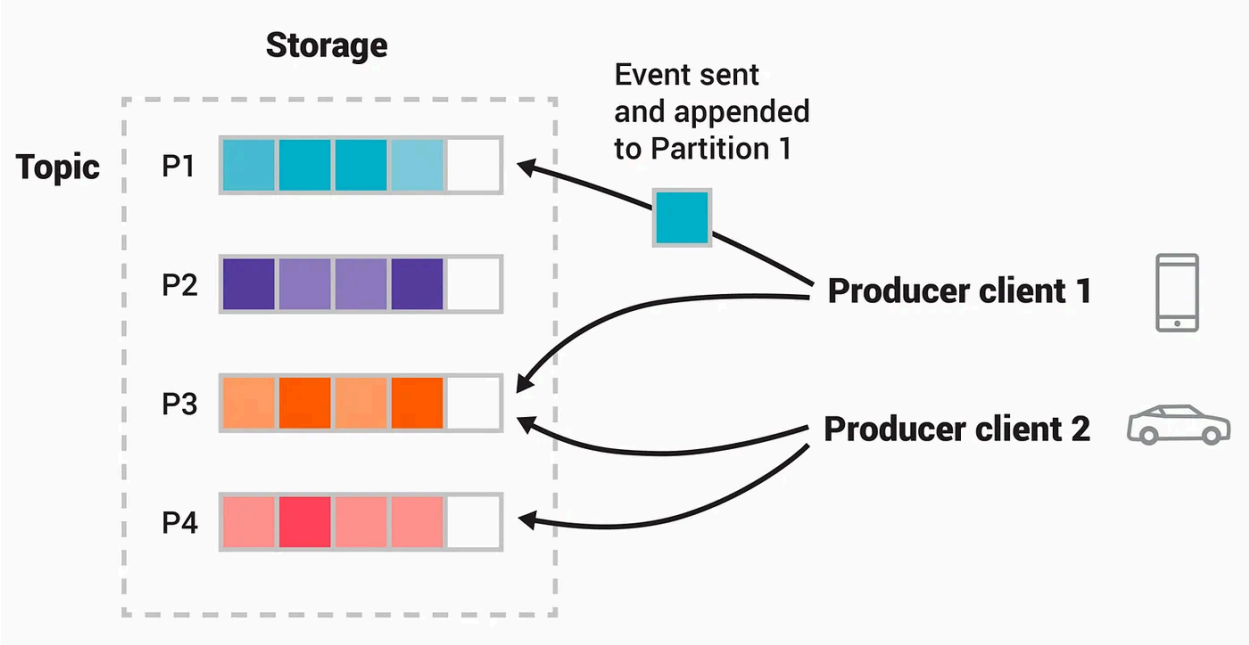


Figure: This example topic has four partitions P1–P4.

- Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Producer 1 is publishing events in partition 1 and partition 3, likewise Producer is publishing events in partition 3 and partition 4.
- Events with the same key (denoted by their color in the figure) are written to the same partition.
- Like in figure every partition has four events and in each partition, events with the same key are denoted with similar colors. In partition 1, event 1 and event 4 have the same key which is why they are denoted by the same colors.

Note that both producers can write to the same partition if appropriate.

Why Kafka? Benefits and Use Cases

- Kafka is used by over 100,000 organizations across the world and is backed by a thriving community of professional developers, who are constantly advancing the state of the art in stream processing.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- Due to Kafka's high throughput, fault tolerance, resilience, and scalability, there are numerous use cases across almost every industry - from banking and fraud detection, to transportation and IoT. We typically see Kafka used for purposes like those below.

1) Data Integration:

   Kafka can connect to nearly any other data source in traditional enterprise information systems, modern databases, or in the cloud. It forms an efficient point of integration with built-in data connectors, without hiding logic or routing inside brittle, centralized infrastructure.

2) Metrics and Monitoring:

   Kafka is often used for monitoring operational data. This involves aggregating statistics from distributed applications to produce centralized feeds with real-time metrics.

3) Log Aggregation:

   A modern system is typically a distributed system, and logging data must be centralized from the various components of the system to one place. Kafka often serves as a single source of truth by centralizing data across all sources, regardless of form or volume

4) Stream Processing:

   Performing real-time computations on event streams is a core competency of Kafka. From real-time data processing to dataflow programming, Kafka ingests, stores, and processes streams of data as it's being generated, at any scale.

5) Publish-Subscribe Messaging:

   As a distributed pub/sub messaging system, Kafka works well as a modernized version of the traditional message broker. Any time a process that generates events must be decoupled from the process or from processes receiving the events, Kafka is a scalable and flexible way to get the job done.

Terminologies in Kafka :

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

Components	Description
Topics	A stream of messages belonging to a particular category is called a topic. Data is stored in topics. Topics are split into partitions. For each topic, Kafka keeps a minimum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.
Partition	Topics may have many partitions, so it can handle an arbitrary amount of data.
Partition offset	Each partitioned message has a unique sequence id called as offset.
Replicas of partition	Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

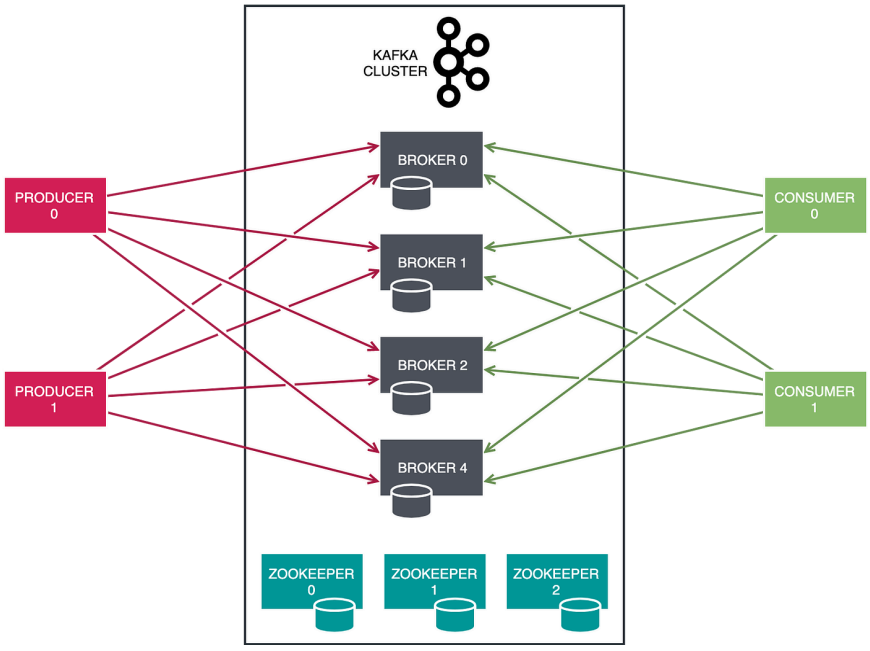
I

Brokers	• Brokers are a simple system responsible for maintaining the pub-lished data. Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition. • Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic. • Assume if there are N partitions in a topic and less than N brokers (n-m), each broker will have one or more partitions shared among them. This scenario is not recommended due to unequal load distri-bution among the broker.
Kafka Cluster	Kafka's having more than one broker is called the Kafka cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.
Producers	Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer pub-lishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producers can also send messages to a partition of their choice.
Consumers	Consumers read data from brokers. Consumers subscribe to one or more topics and consume published messages by pulling data from the brokers.
Leader	Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.
Follower	Nodes which follow leader instructions are called as follower. If the leader fails, one of the followers will automatically become the new leader. A follower acts as a normal consumer, pulls messages and updates its own data store.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

What are the key components of Kafka?



**Broker 0 is the leader and active cluster controller*

Kafka is a distributed system comprising several key components:

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

1) Broker nodes: It is responsible for the bulk of I/O operations and durable persistence within the cluster. Brokers accommodate the append-only log files that comprise the topic partitions hosted by the cluster.

   Partitions can be replicated across multiple brokers for both horizontal scalability and increased durability — these are called replicas. A broker node may act as the leader for certain replicas while being a follower for others.

   A single broker node will also be elected as the cluster controller — responsible for the internal management of partition states. This includes the arbitration of the leader-follower roles for any given partition

2) ZooKeeper nodes: Under the hood, Kafka needs a way of managing the overall controller status in the cluster. The actual mechanics of controller election, heart-beating, and so forth, are largely implemented in ZooKeeper.

   ZooKeeper also acts as a configuration repository of sorts, maintaining cluster metadata, leader-follower states, quotas, user information, ACLs and other housekeeping items.

3) Producers: it is responsible for appending records to Kafka topics. Because of the log-structured nature of Kafka, and the ability to share topics across multiple consumer ecosystems, only producers are able to modify the data in the underlying log files

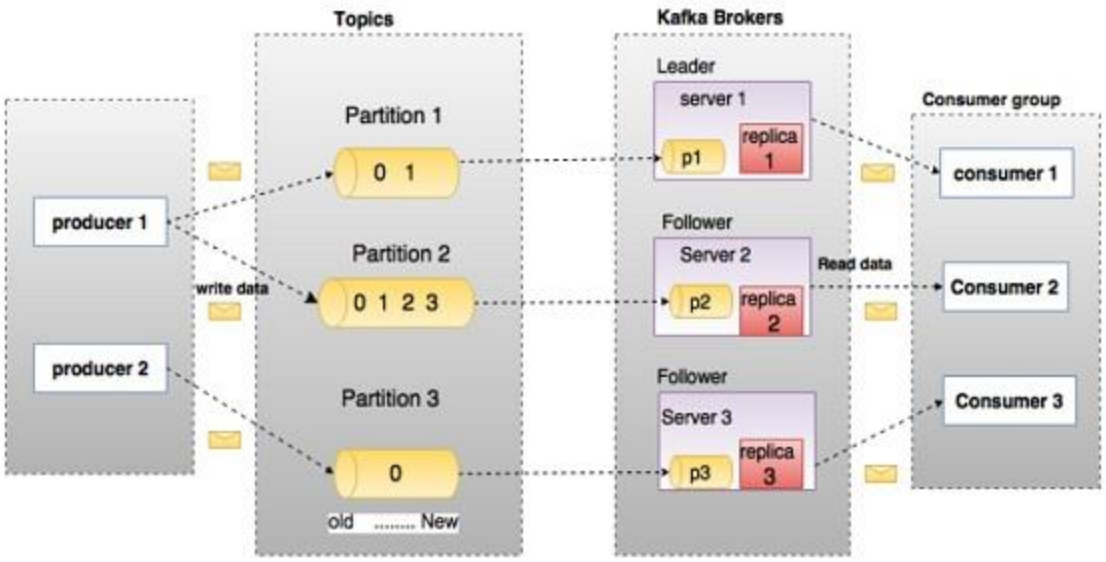
   The actual I/O is performed by the broker nodes on behalf of the producer clients. Any number of producers may publish to the same topic, selecting the partitions used to persist the records

4) Consumers: it reads data from topics. Any number of consumers may read from the same topic; however, depending on the configuration and grouping of consumers, there are rules governing the distribution of records among the consumers.

Kafka Architecture:

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I



x

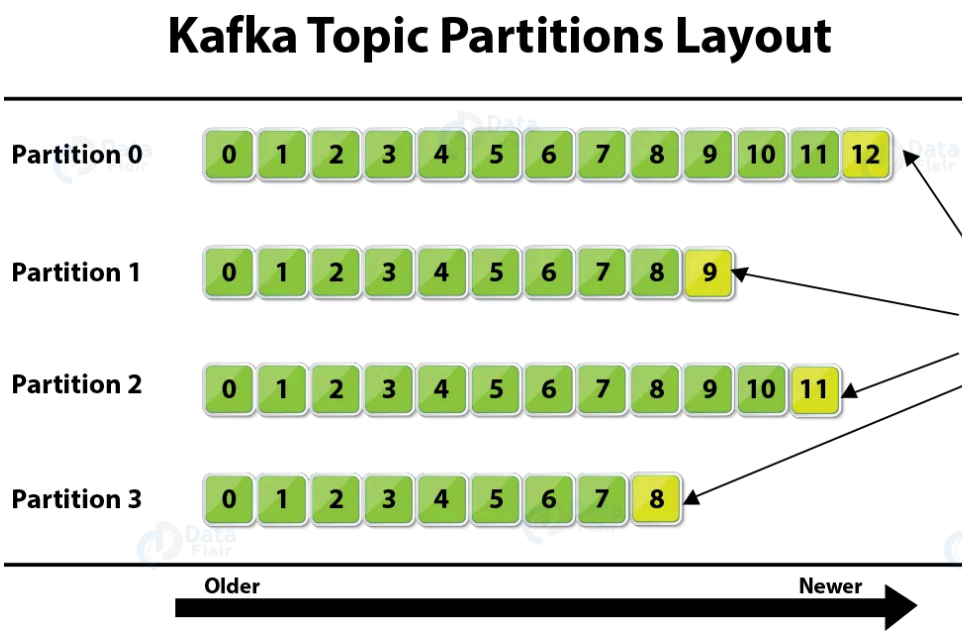
As you can see it is very similar to what we've discussed about messaging, but here we don't have the Queue concept, instead, we have the concept of Topics.

Topic:

- The Topic is a very specific type of data stream, it's very similar to a Queue, it receives and delivers messages as well, but there are some concepts that we need to understand about topics:
- A topic is divided into partitions, each topic can have one or more partitions and we need to specify that number when we're creating the topic. You can imagine the topic as a folder in the operating system and each folder inside her as a partition.
- If we don't give any key to a message when we're producing it, by default, the producers will send the message in a round-robin way, each partition will receive a message (even if they are sent by the same producer).
- The producers use the offset to read the messages, they read from the oldest to the newest. In case of consumer failure, when it recovers, will start reading from the last offset;

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

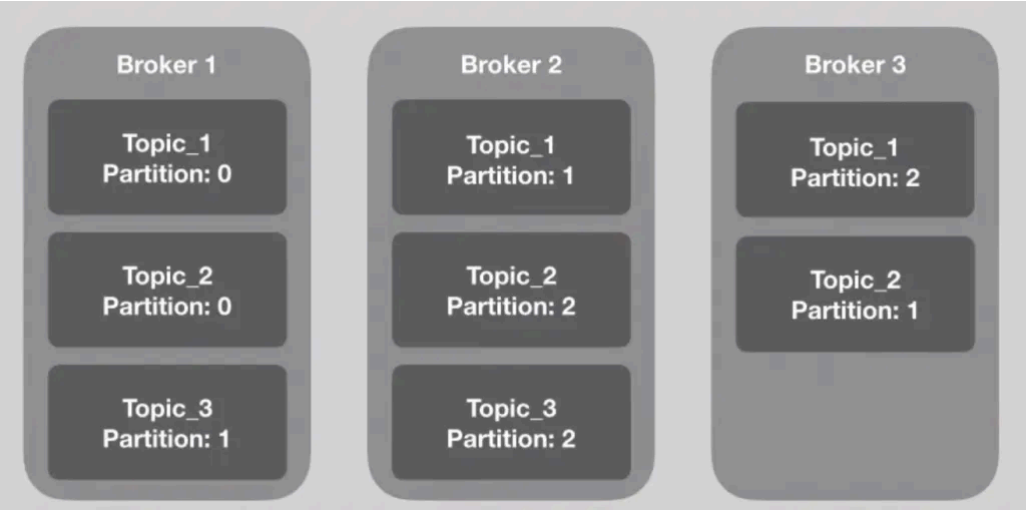
I



**the publisher is writing into the partitions.*

Brokers:

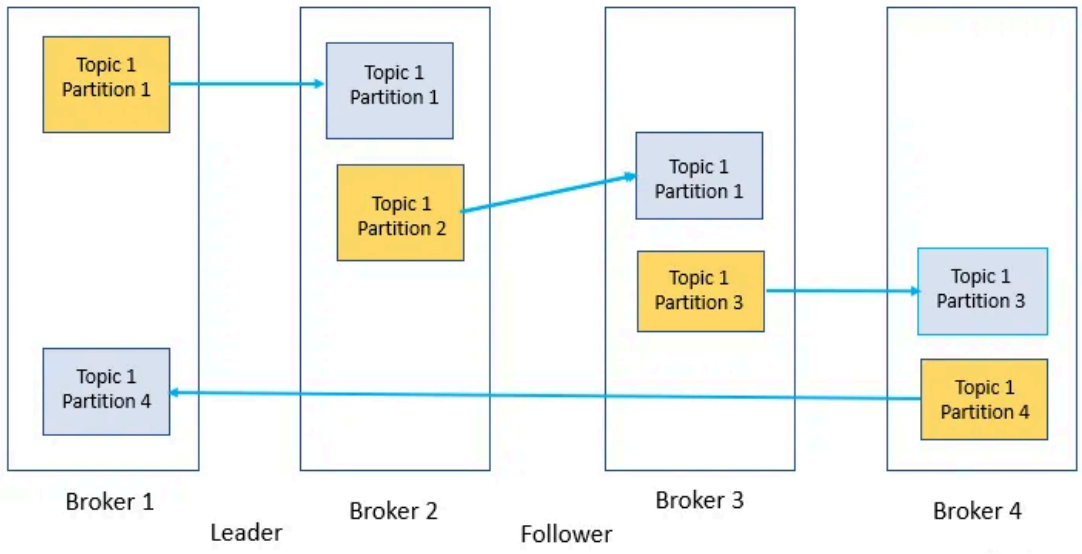
- Each broker in a cluster is identified by an ID and contains at least one partition of a topic. To configure the number of the partitions in each broker, we need to configure something called Replication Factor when creating a topic.
- Let's say that we have three brokers in our cluster, a topic with three partitions and a Replication Factor of three, in that case, each broker will be responsible for one partition of the topic.



Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- As you can see in the above image, Topic_1 has three partitions, each broker is responsible for a partition of the topic, so, the Replication Factor of Topic_1 is three.
- It's very important that the number of the partitions match the number of the brokers, in this way, each broker will be responsible for a single partition of the topic.
- To ensure the reliability of the cluster, Kafka enters with the concept of the Partition Leader. Each partition of a topic in a broker is the leader of the partition and can exist only one leader per partition.
- The leader is the only one that receives the messages, their replicas will just sync the data (they need to be in-sync to that). It will ensure that even if a broker goes down, his data won't be lost, because of the replicas
- When a leader goes down, a replica will be automatically elected as a new leader by Zookeeper.



**correction - in broker 3 its it Topic 1 and partition 2 instead of partition 1*

In the above image, Broker 1 is the leader of Partition 1 of Topic 1 and has a replica in Broker 2. Let's say that Broker 1 dies, when it happens, Zookeeper will detect that change and will make Broker 2 the leader of Partition 1. This is what makes the distributed architecture of Kafka so powerful.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

Producers:

Producers in Kafka are the ones who produce and send the messages to the topics.

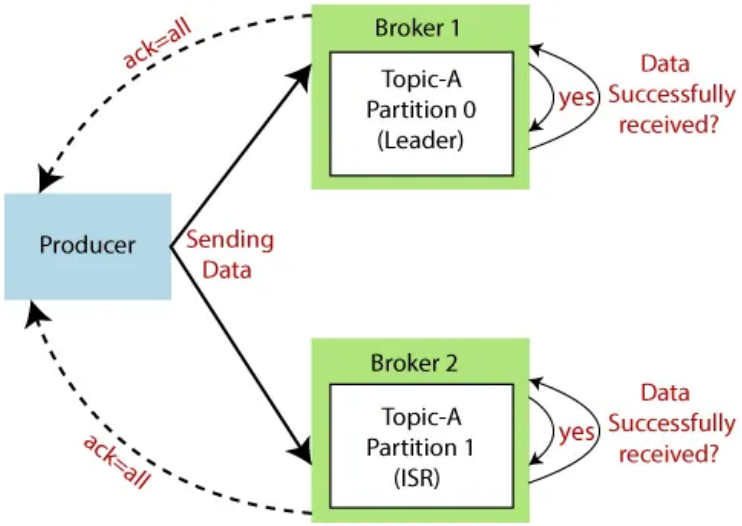
- the messages are sent in a round-robin way. Ex: Message 01 goes to partition 0 of Topic 1, and message 02 to partition 1 of the same topic.
- It means that we can't guarantee that messages produced by the same producer will always be delivered to the same topic. We need to specify a key when sending the message, Kafka will generate a hash based on that key and will know what partition to deliver that message.
- there's something called Acknowledgement (ack). The ack is basically a confirmation that the message was delivered. In Kafka, we can configure this ack when producing the messages. There are three different levels of configuration for that:

    1) ack = 0: in this we're saying that we don't want to receive the ack from Kafka. In case of broker failure, the message will be lost;
    2) ack = 1: This is the default configuration, with that we're saying that we want to receive an ack from the leader of the partition. The data will only be lost if the leader goes down (still there's a chance);
    3) ack = all: This is the most reliable configuration. We are saying that we want to not only receive confirmation from the leader but from their replicas as well.

        This is the most secure configuration since there's no data loss. Remembering that the replicas need to be in-sync (ISR). If a single replica isn't, Kafka will wait for the sync to send back the ack.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I



Data is successfully received

Consumer and Consumer groups:

Consumers are applications subscribed to one or more topics that will read messages from there. They can read from one or more partitions.

- When a consumer reads from just one partition, we can ensure the order of the reading, but when a single consumer reads from two or more partitions, it will read in parallel, so, there's no guarantee of the reading order. A message that came later can be read before another that came earlier,
- That's why we need to be careful when choosing the number of partitions and when producing the messages.

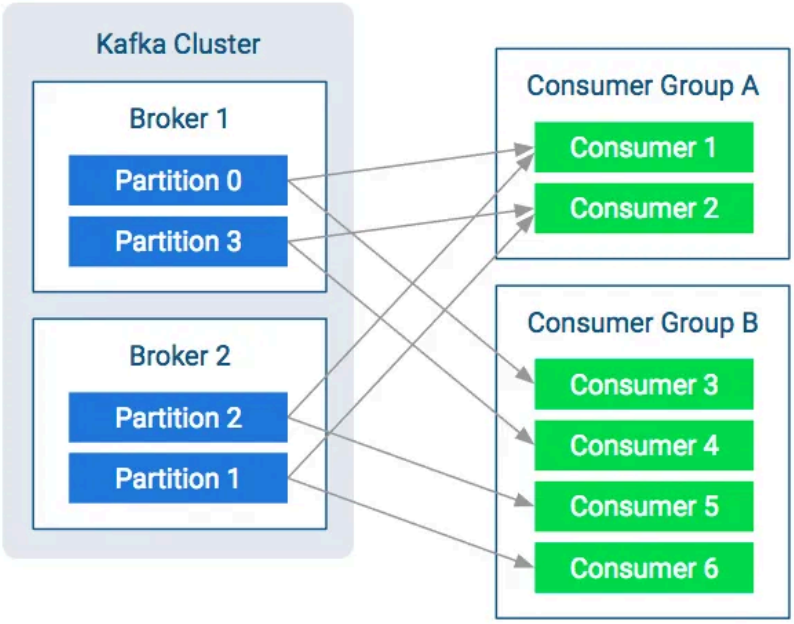
Another important concept of Kafka is the Consumer Groups. It's really important when we need to scale the reading of messages.

- It becomes very costly when a single consumer needs to read from many partitions, so, we need to load-balancing this charge between our consumers, this is when the consumer groups enter.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

- The ideal is to have the same number of consumers in a group that we have as partitions in a topic, in this way, every consumer read from only one. When adding consumers to a group, you need to be careful, if the number of consumers is greater than the number of partitions, some consumers will not read from any topic and will stay idle.



How to ingest data to kafka :

Let's say we have a kafka setup and kafka binaries already available in the system.
Kafka broker is available at localhost:9092. To ingest the data to kafka , we will use following:

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
HelloKafka
```

> My-first-message
> My-second-message

This way two messages will be ingested in the HelloKafka topic.

How to read data from kafka :

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

After writing to the kafka topic, let's read from the kafka topic.

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 –topic HelloKafka --from-beginning
My-first-message
My-second-message
```

Now as we are all clear with kafka , let's move to spark streaming basics in the next section.

How to solve our problem now: Part 1?

As we are familiar with most of the technologies in the real time domain, we can gear up and enter into the trajectory of the real world domain. This is a very vast area.
Our problem statement is to build a real time dashboard to see the trends of bitcoin fluctuations in real time. The dashboard should be refreshed in a few seconds and the graphs and charts should be refreshed in accordance with the real time data.

Setup :

We have used the python3.8 version for both the lectures. Please use the same to be compatible.

Docker :

Download docker desktop. For CLI use `pip3 install docker.`

Kafka :

- Run docker image for setting up kafka and zookeeper.
  - Image : docker run -p 2181:2181 -p 9092:9092 -e ADVERTISED_HOST=127.0.0.1 -e NUM_PARTITIONS=10 johnnypark/kafka-zookeeper
- After #1 , the kafka broker will be available on localhost:9092.
- Download kafka binaries for producing kafka messages(using Kafka CLI) : https://kafka.apache.org/downloads

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

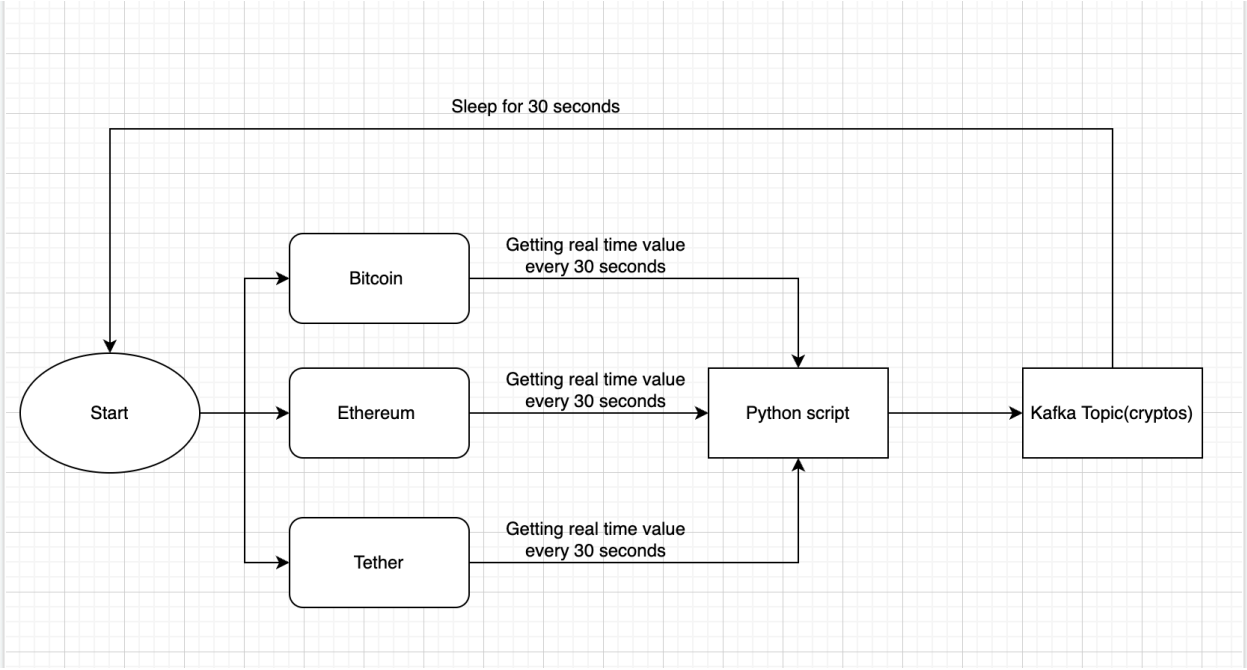
I

Step 1: Load to Kafka

- We will be using yahoo finance data apis for getting the real time data for bitcoin crypto currency. The data has other crypto currencies data as well.
- Right now we are going with some of the popular ones like **bitcoins, Ethereum, Tether, Binance Coin, USD Coin**.

Datetime	Open	High	Low	Close	Adj Close	Volume
2022-03-09 11:39:00+00:00	42073.042969	42073.042969	42073.042969	42073.042969	42073.042969	6000640
2022-03-09 11:40:00+00:00	42065.621094	42065.621094	42065.621094	42065.621094	42065.621094	47939584
2022-03-09 11:41:00+00:00	42060.316406	42060.316406	42060.316406	42060.316406	42060.316406	0
2022-03-09 11:42:00+00:00	42064.351562	42064.351562	42064.351562	42064.351562	42064.351562	0
2022-03-09 11:43:00+00:00	42067.289062	42067.289062	42067.289062	42067.289062	42067.289062	0

- Every row above explains the bitcoin value for that particular minute. Primary reason for considering this dataset for analysis is because the fluctuation in crypto currencies happens every few seconds, thus making an ideal case for our use case.
- We will be sending only Date and Close columns to Kafka for each currency. The data will be fetched from yahoo_finance library and sent to kafka every 30 seconds.




```
import numpy as np
import yfinance as yf
import os
import time

while(1==1):
for crypt in ['BTC-USD','ETH-USD','USDT-USD','BNB-USD','USDC-USD']:
data = yf.download(tickers=crypt, period = '2h', interval = '1m')
data = data.tail(1)
```

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data

I

```
data['Type'] = crypt
data = data[['Type','Close']]
print(data.info())
data = data.to_csv("./data/current_{}.csv".format(crypt.replace('-','_')), index=False, header =
False)
print(type(data))
os.system("kafka-console-producer.sh --broker-list 127.0.0.1:9092 --topic cryptos <
./data/current_{}.csv".format(crypt.replace('-','_')))
time.sleep(30)
```

Conclusion

This is just the first half of the streaming as a whole. We learnt about messaging queues, real time processing frameworks in brief.
Later we discussed different messaging queues and why we are going ahead with Kafka as our preferred messaging queue.
In the next session we will discuss different real time data processing frameworks, what we chose out of them and why.
Also we will discuss different time-series databases and why we preferred the one out of them.
Also we will deep dive into different dashboarding tools available and what is the best option for us in our use case.
Excited!! Stay tuned and keep learning and exploring.

Code : https://github.com/scaleracademy/dsml-de/tree/main/streaming-data