**Birla Institute of Technology & Science, Pilani**

**Work Integrated Learning Programmes Division**

**First Semester 2022-2023**

**Assignment - 2**

Course No. : DSECLZG556

Course Title: Stream Processing and Analytics

Group No: **Group 20**

|  |  |
| --- | --- |
| **Group Member Name** | **Email ID** |
| Varinder Singh | 2021fc04070@wilp.bits-pilani.ac.in |
| Bandaru Raja Sekhar | 2021fc04074@wilp.bits-pilani.ac.in |
| Mikhil P.A. | 2021fc04326@wilp.bits-pilani.ac.in |
| Yogesh Tiwari | 2021fc04825@wilp.bits-pilani.ac.in |

Table of Contents

[Overall Architecture Diagram 3](#_Toc129604791)

[Technology Stack Used: 3](#_Toc129604792)

[Criteria used for order match-making 4](#_Toc129604793)

[The streaming data pipeline architecture 4](#_Toc129604794)

[Kafka Topics 6](#_Toc129604795)

[Steps for starting Kafka Cluster 6](#_Toc129604796)

[Components used and their purpose 6](#_Toc129604797)

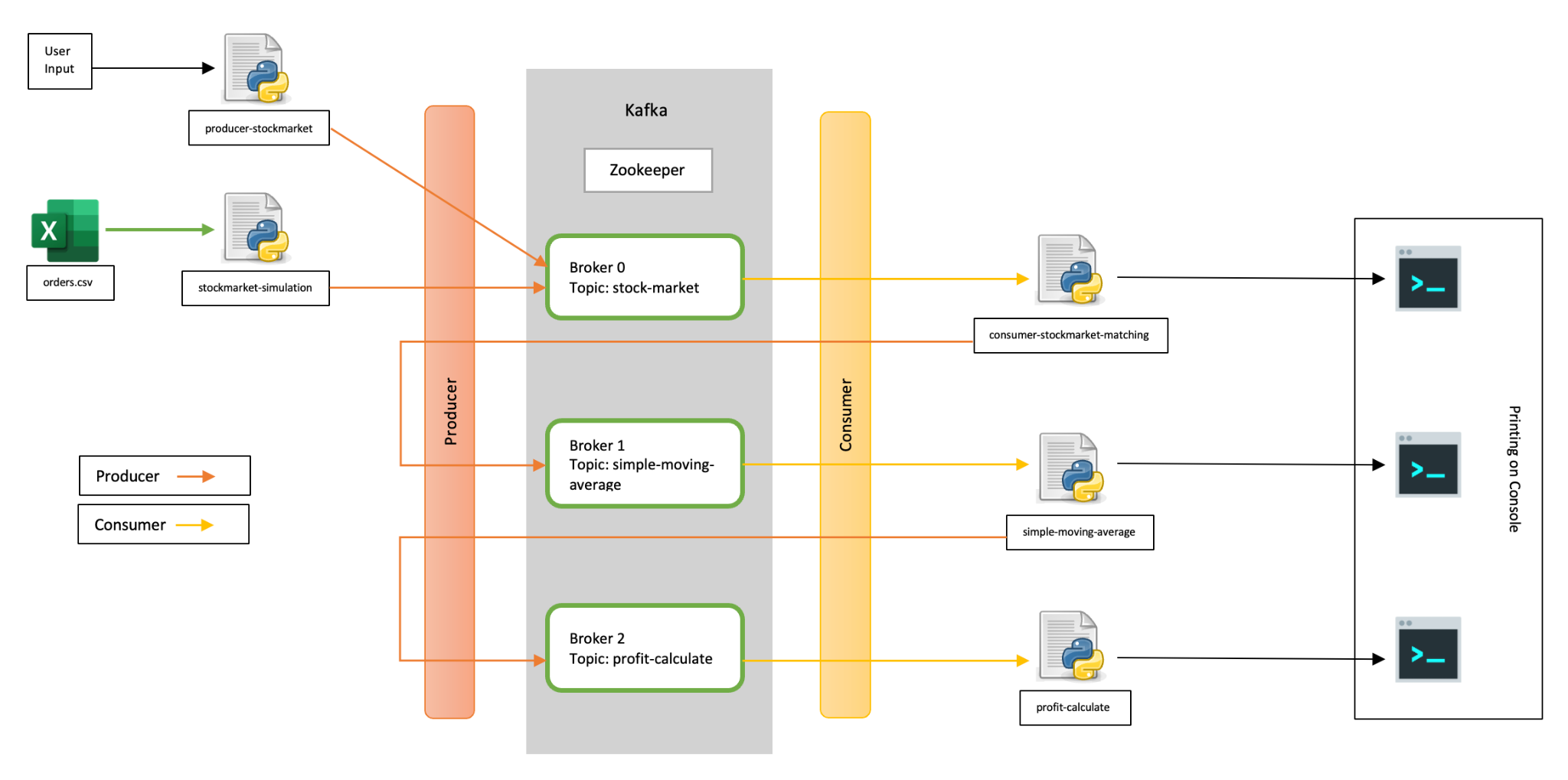
[Data structure and its flow across Kafka topics 7](#_Toc129604798)

[Business assumptions and logic used 9](#_Toc129604799)

[Demo Video link: 11](#_Toc129604800)

# Overall Architecture Diagram

Below is the overall architectural diagram for the design and development of the electronic stock exchange.



# Technology Stack Used:

1. Python Notebook
   1. producer-stockmarket
   2. stockmarket-simulation
   3. consumer-stockmarket-matching
   4. simple-moving-average
   5. profit-calculate
2. Kafka Cluster at the local machine
   1. Zookeeper - localhost:2181
   2. Broker0 - localhost:9092
   3. Broker1 - localhost:9093
   4. Broker2 - localhost:9094

# Criteria used for order match-making

We have defined the match\_orders() function for order match-making. The criteria used for order match-making are:

* Order Action(BUY or SELL)
* Order Quantity

The order match-making function is used to check the order type and does match it with the orders which already exist in the orders dictionary. The steps which are taken in this function are as follows:

1. Order is received from Kafka topic- “stock-market” in JSON format and it is stored in the orders dictionary.
2. First of all, the clean\_dict() function is run. This function is used to clear expired orders from the current orders dictionary. The expired orders are identified by the time when the order was placed; the timing is stored in the order JSON received from the Kafka topic “stock-market”.
3. For every order(order) in the orders dictionary. The following steps are taken:
   1. Order action is checked whether the order is to BUY and current orders are checked with order action with SELL and the same price and stored in a match list.
   2. Order action is checked whether the order is to SELL and current orders are checked with order action with BUY and the same price and stored in a match list.
   3. Else, it is passed; assuming it has incorrect action for the order.
   4. For every order(m) in the match list:
      1. The quantity of the order(m) is compared with the quantity of the order(order) if it is less than or greater than, accordingly the quantity of the order is changed and the trade is completed for the matching quantity. The completed trade is sent to Kafka topic- “simple-moving-average”. The completed order(m or order) is deleted from the orders dictionary and another one is updated with the remaining quantity. The completed trade is printed on the console.
      2. If the quantity of the order(m) is equal to the quantity of the order(order), then the trade is completed. The completed trade is sent to Kafka topic- “simple-moving-average”. The order is deleted from the orders dictionary. The completed trade is printed on the console.

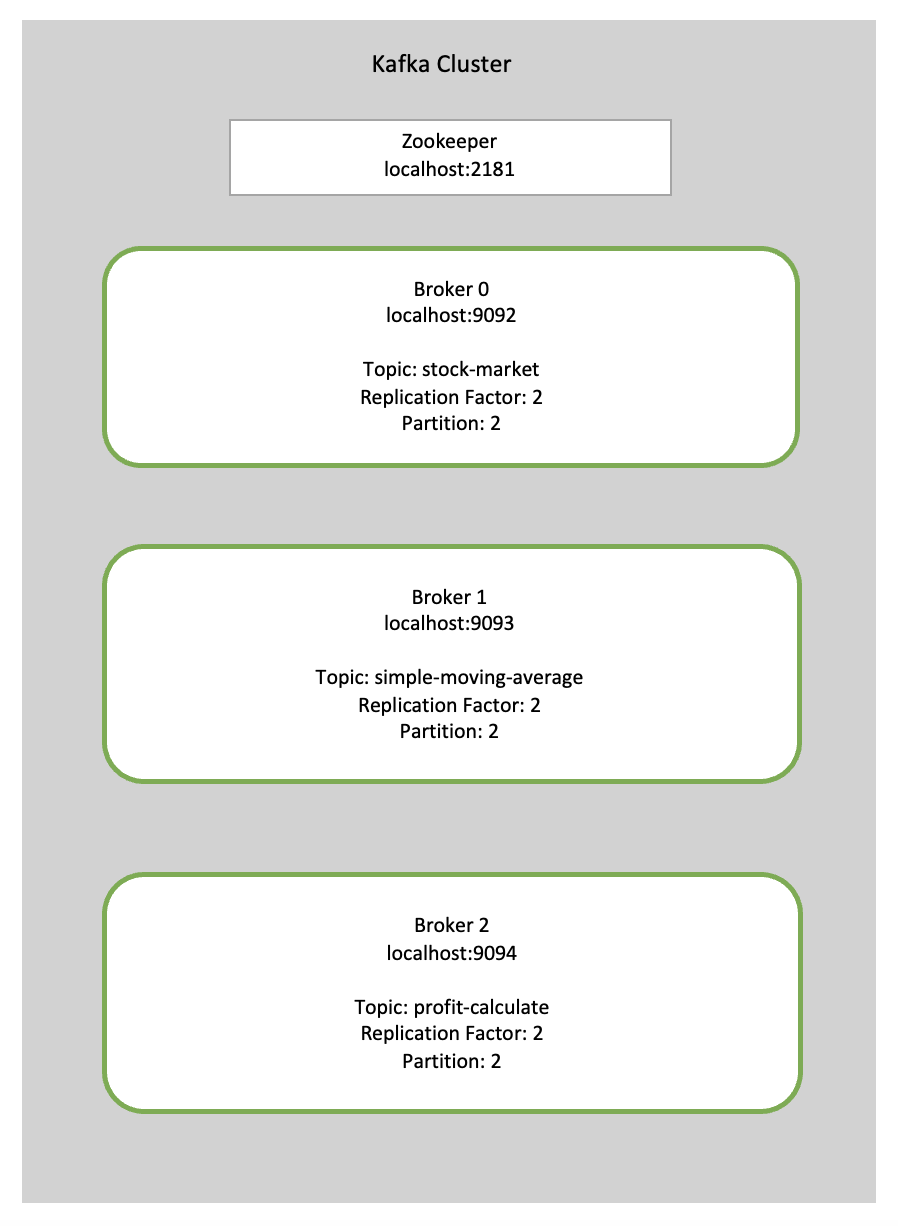
# The streaming data pipeline architecture

Apache Kafka works on a built data pipeline which carries streaming data or messages. A streaming pipeline is where we send data from a source to a target as the data happens, in a stream. Contrast this to a batch world where we wait for a period of time (maybe hours or days) before collecting a bunch of the data and then sending it to the target. There are several good reasons for wanting to use a streaming pipeline, including

1. Ensuring more accurate data in the target system
2. Reacting to data as it changes, while it is current and relevant
3. Spreading the processing load and avoiding resource shortages from a huge influx of data

In Apache Kafka, a streaming data pipeline means ingesting the data from sources into Kafka as it's created and then streaming that data from Kafka to one or more targets. In this example, we're offloading transactional data from a database to an object store, perhaps for analytical purposes. The messaging layer of Kafka partitions data for storing and transporting it. Kafka Streams partition data for processing it. In both cases, this partitioning is what enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of partitions and tasks as logical units of its parallelism model based on Kafka topic partitions. There are close links between Kafka Streams and Kafka in the context of parallelism:

1. Each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition.
2. A data record in the stream maps to a Kafka message from that topic.
3. The keys of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

In our scenario, we need to create a stock exchange where the messages are sent to consumers from producers. And, processing the data as we go with it. As we are doing it on our local machine, we have created three brokers for our case and all three brokers have replication-factor and partition as 2 which suits the current requirement. For real-life practice, we can increase these parameters to the business requirements. The details of the Kafka cluster are given below:

# Kafka Topics

We have created three Kafka topics in our setup. The name of these Kafka topics are as follows:

1. stock-market
2. simple-moving-average
3. profit-calculate

Below are the details for the broker details and its respective topic:

* stock-market → localhost:9092
* simple-moving-average → localhost:9093
* profit-calculate → localhost:9094

# Steps for starting Kafka Cluster

**Start Zoo-keeper:**

* bin/zookeeper-server-start.sh config/zookeeper.properties

**Start Kafka-server:**

1. bin/kafka-server-start.sh config/server0.properties
2. bin/kafka-server-start.sh config/server1.properties
3. bin/kafka-server-start.sh config/server2.properties

# Components used and their purpose

The components used and their purposes are mentioned as follows:

1. Kafka Brokers:

Brokers are Kafka’s distributed message brokers that form the core of the Kafka cluster. Brokers receive, store, and service messages to clients that subscribe to Kafka topics. Each broker holds a subset of the topic’s partitions and acts as a leader or follower for these partitions. Brokers are horizontally scalable, allowing Kafka clusters to handle large volumes of data. We used three brokers which are as follows:

* Broker 0 - localhost:9092
* Broker 1 - localhost:9093
* Broker 2 - localhost:9094

1. Kafka Topics:

Topics are logical channels or categories to which producers can write messages and consumers can read messages. Topics are partitioned and replicated across brokers in a Kafka cluster, allowing for high availability and fault tolerance. We used three topics which are as follows:

* stock-market on Broker 0
* simple-moving-average on Broker 1
* profit-calculate on Broker 2

1. Zookeeper:

Zookeeper is used by Kafka brokers to determine which broker is the leader of a given partition and topic and perform leader elections. Zookeeper stores configurations for topics and permissions. Zookeeper sends notifications to Kafka in case of changes. We created a zookeeper at localhost:2181.

1. Kafka Producers:

Producers are applications that write data to Kafka topics. Producers can send messages to specific partitions or rely on the partitioning strategy configured for the topic.

1. Kafka Consumers:

Consumers are applications that read data from Kafka topics. Consumers subscribe to one or more topics and receive messages from the partitions assigned to them by the Kafka cluster.

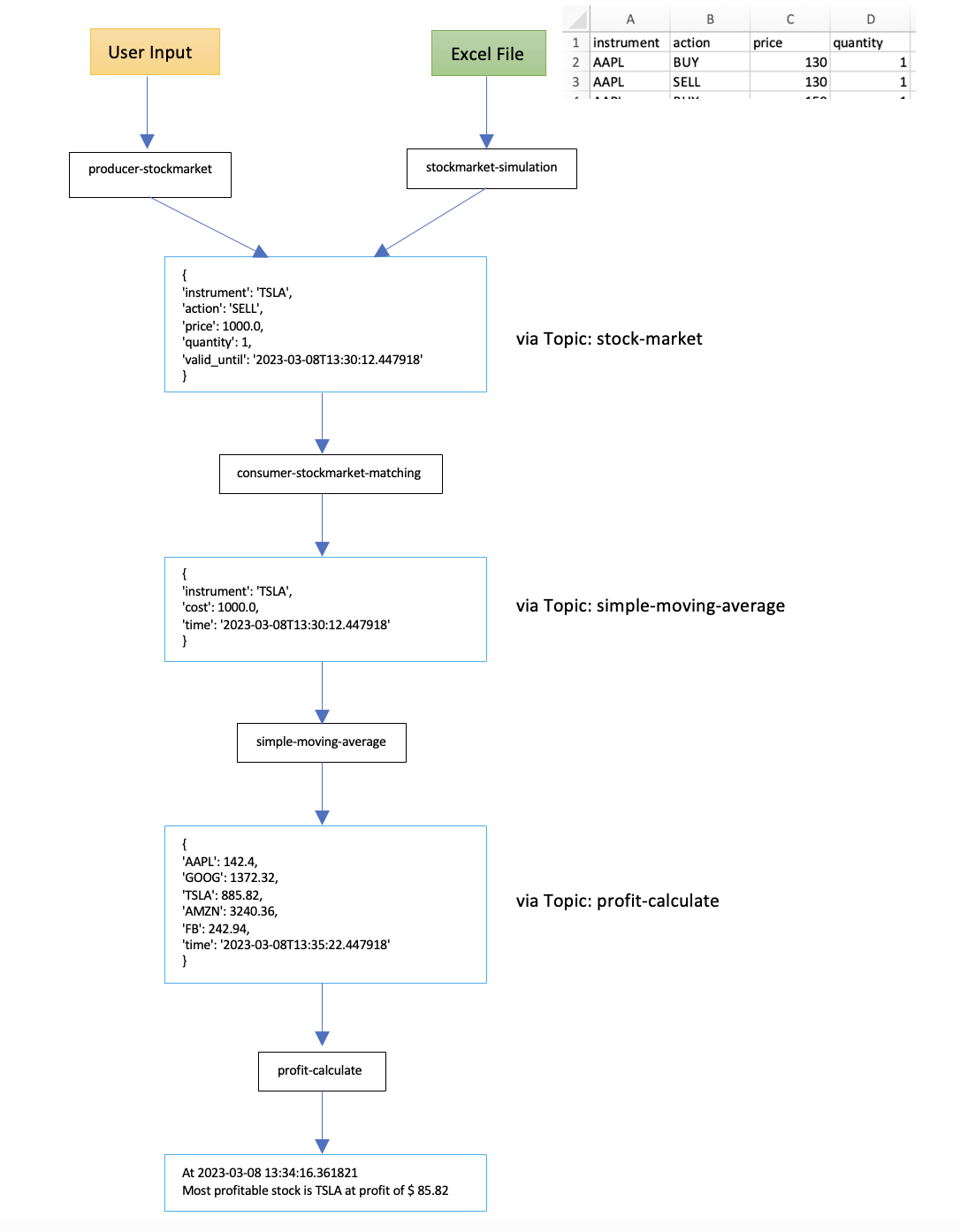
1. Python Notebooks

The Jupyter Notebook is an open-source web application; that enables users to create and share documents that include live code, equations, computational output, visualisations, multimedia resources, and explanatory text.

# Data structure and its flow across Kafka topics

The Data Flow in Kafka refer to the movement of data through the Kafka ecosystem. In Kafka, data is produced by one or more data sources and is sent to a Kafka cluster where it is processed and consumed by one or more data consumers. The data flow in Kafka is unidirectional, meaning that data is always flowing from the producer to the consumer. Data is organized into topics in Kafka, which act as logical containers for messages. A producer sends messages to a specific topic and one or more consumers subscribe to that topic to receive messages. In our case, the data flow is flown from producer to consumer via 3 different topics. All these topics carry the data at different stages of data processing.

1. Firstly, it carries data on orders placed whether it is coming from user input or from a CSV file which simulates the real stock exchange.
2. Secondly, after the trade is completed, the trade data for an instrument is carried out for further processing which is to find a simple moving average.
3. Thirdly, the averages calculated are carried out in order to find the most profitable stock at a particular instance.



# Business assumptions and logic used

1. For our scenario, we are considering only 5 instruments for stock exchange trade, streaming data and processing. The five stock instruments used are as follows:
   1. AAPL (Apple)
   2. GOOG (Google)
   3. TSLA (Tesla)
   4. AMZN (Amazon)
   5. FB (Facebook)

For the initial trading or when the first trade is happening, we are assuming the opening price for all the above-mentioned stocks which are as follows:

1. AAPL (Apple) – 130.00
2. GOOG (Google) – 1500.00
3. TSLA (Tesla) – 800.00
4. AMZN (Amazon) – 3200.00
5. FB (Facebook) – 250.00

Similarly, when we are calculating profits and the most profitable stock we are assuming the above-mentioned price for the first opening prices for the first profit calculation. After the first window has run, then dynamically profits are calculated with opening and closing prices as the subsequent rounds of window runs happen.

1. Order placement and expiring placed orders after 2 minutes
2. When an order is placed using an excel file, the number(quantity) of shares in an order placed is taken(assumption) as 1 for the sake of simplicity.
3. For every order placed, an attribute “valid\_until” is added to it with time till when the placed order is valid. Once that time is passed that order should be expired and should not be considered for future orders. For our scenario, we have kept the order expiration at 2 minutes; which means that the order placed at T0 will be valid till (T0 + 2 minutes). Beyond (T0 + 2 minutes), the orders will be expired and be removed from the list of current orders in the stock market.
4. Match-making logic

We have defined logic for the match\_orders() function for order match-making. The criteria used for order match-making are Order Action(BUY or SELL) and Order Quantity. The order match-making function is used to check the order type and does match it with the orders which already exist in the orders dictionary. For matching the order for a successful trade – the order id, stock instrument, action for the order(BUY or SELL) and the price of the share are considered. The steps which are taken in this function are as follows:

1. Order is received from Kafka topic- “stock-market” in JSON format and it is stored in the orders dictionary.
2. First of all, the clean\_dict() function is run. This function is used to clear expired orders from the current orders dictionary. The expired orders are identified by the time when the order was placed; the timing is stored in the order JSON received from the Kafka topic “stock-market”.
3. For every order(order) in the orders dictionary. The following steps are taken:
4. Order action is checked whether the order is to BUY and current orders are checked with order action with SELL and the same price and stored in a match list.
5. Order action is checked whether the order is to SELL and current orders are checked with order action with BUY and the same price and stored in a match list.
6. Else, it is passed; assuming it has incorrect action for the order.
7. For every order(m) in the match list:

i. The quantity of the order(m) is compared with the quantity of the order(order) if it is less than or greater than, accordingly the quantity of the order is changed and the trade is completed for the matching quantity. The completed trade is sent to Kafka topic- “simple-moving-average”. The completed order(m or order) is deleted from the orders dictionary and another one is updated with the remaining quantity. The completed trade is printed on the console.

ii. If the quantity of the order(m) is equal to the quantity of the order(order), then the trade is completed. The completed trade is sent to Kafka topic- “simple-moving-average”. The completed order is deleted from the current orders dictionary. The completed trade is printed on the console.

1. Simple moving average at every 5 min with 10 min window

To calculate Simple Moving Average, here we are considering the closing price of all the successful trades happening in the stock exchange for the last 10 minutes window. This SMA calculation is triggered every 5 minutes and the resultant SMA for all 5 instruments is printed on the console.

First of all, all the successful trades are captured with the help of the consumer topic of ‘stock-market’ and these trades are stored in a panda data frame. After every 5 minutes, an average calculation is triggered which considers only those successful trades which have been completed in the last 10 minutes and the mean is calculated for each instrument; which in return is printed on the console.

1. Profit Calculate at every 5 min with 10 min window

To calculate the profit, we consider the average opening and closing price of the instrument received from the topic - ‘profit-calculate’. It compares the opening and closing prices and finds the positive and most profit values and then the corresponding stock is printed on the console. For the very first profit calculation, we have assumed the opening average price of all 5 instruments as per the initial trading price assumed earlier.

As we receive average values of all the instruments which have already triggered after every 5 minutes. The message is consumed by the topic ‘profit-calculate’, and then profit is calculated by subtracting the last received averages as the opening average price and the currently received as the closing average price. The profit calculated for all 5 instruments is compared with each other and the instrument with the maximum profit is chosen to print on the console with the profit value.

# Demo Video link:

<https://drive.google.com/file/d/1USl4j8QcJRijaZbzw_XRxcU5s2SWvqWO/view?usp=sharing>