a

Birla Institute of Technology & Science, Pilani
Work Integrated Learning Programmes Division
First Semester 2022-2023

BITS Pilani
Pilani | Dubai | Goa | Hyderabad
Work Integrated Learning Program

Assignment II

Course No. : DSECL ZC556
Course Title : Stream Processing and Analytics
Nature of Exam : Take Home
Weightage : 15%
Duration : 30 days

No. of Pages = 4
No. of Questions = 5

Group Details

Number	Group 106

Members

Name	ID	Contribution (%)
Ritresh Girdhar	2021FC04145	100
Varshini M	2021FC04149	100
Abhishek Shah	2021FC04153	100
Subhi Singh	2021FC04061	100

Problem Statement - Design and Development of Electronic Stock Exchange

Stock trading broadly refers to any buying and selling of stock, but is colloquially used to refer to more short-term investments made by very active investors. Most stocks are traded on physical or virtual exchanges. The New York Stock Exchange (NYSE), for example, is a physical exchange where some trades are placed manually on a trading floor—yet, other trading activity is conducted electronically. NASDAQ, on the other hand, is a fully electronic exchange where all trading activity occurs over an extensive computer network, matching investors from around the world with each other in the blink of an eye. Investors and traders submit orders to buy and sell shares, either through a broker or by using an online platform such as an E*Trade.

A buyer bids to purchase shares at a specified price (or at the best available price) and a seller asks to sell the stock at a specified price (or at the best available price). When a bid and an ask match, a transaction occurs and both orders will be filled. In a very liquid market, the orders will be filled almost instantaneously. In a thinly traded market, however, the order may not be filled quickly or at all. On an electronic exchange, such as NASDAQ, buyers and sellers are matched electronically. Market makers (similar in function to the specialists at the physical exchanges) provide bid and ask prices, facilitate trading in certain security, match buy and sell orders, and use their own inventory of shares, if necessary.

Active trading is when an investor who places 10 or more trades per month. They often use strategies that rely heavily on timing the market. They try to take advantage of short-term events (at the company or in the market) to turn a short-term profit. Day trading means playing hot potato with stocks — buying and selling the same stock in a single trading day. Day traders care little about the inner workings of the businesses. They try to make a few bucks in the next few minutes, hours or days based on daily price swings.

You have to design and implement an electronic exchange where day trading is facilitated. Make necessary assumptions while designing and implementing such a system and ensure that you have carefully drafted out the critical design decisions.

In order to realize the above expectations, you need to design and construct a streaming data pipeline integrating the various technologies, tools and Programmes covered in the course that will harvest this real time data of trader's orders and produces the trades / analytics that can be sent on the trader's mobile devices. You can think of various aspects related to streaming data processing such as:

- Real time streaming data ingestion
- Data's intermittent storage
- Data preprocessing – cleaning, transformations etc.
- Data processing – filters, joins, windows etc.
- Business logic for placing the offers
- Final representation of the outcome

Submission requirements:

1) Document describing the architecture and tech-stack
2) Python program for order placement with sample input and outputs
3) Short note on criteria used for order match-making
4) Python program for trade generation
5) Short note on describing

- the streaming data pipeline architecture
- components used and purposes of the same
- data flows
- business logic used

6) Programs / Queries used in exercise 4 and 5
7) A short demo describing the overall thought process for approaching this problem and data flow through the pipeline. **Share the Google drive link for the same.**

References:

1) What Happens When You Buy or Sell Stocks?
2) How Stock Trading Works?
3) Kaggle Stock Market Analysis
4) Simple Moving Average (SMA)
5) Zerodha Kite Con nect 3 / API documentation

Exercise 1: Architecture [3 marks]

Provide a suitable architecture diagram with appropriate description matching the system specification described in the following exercises.

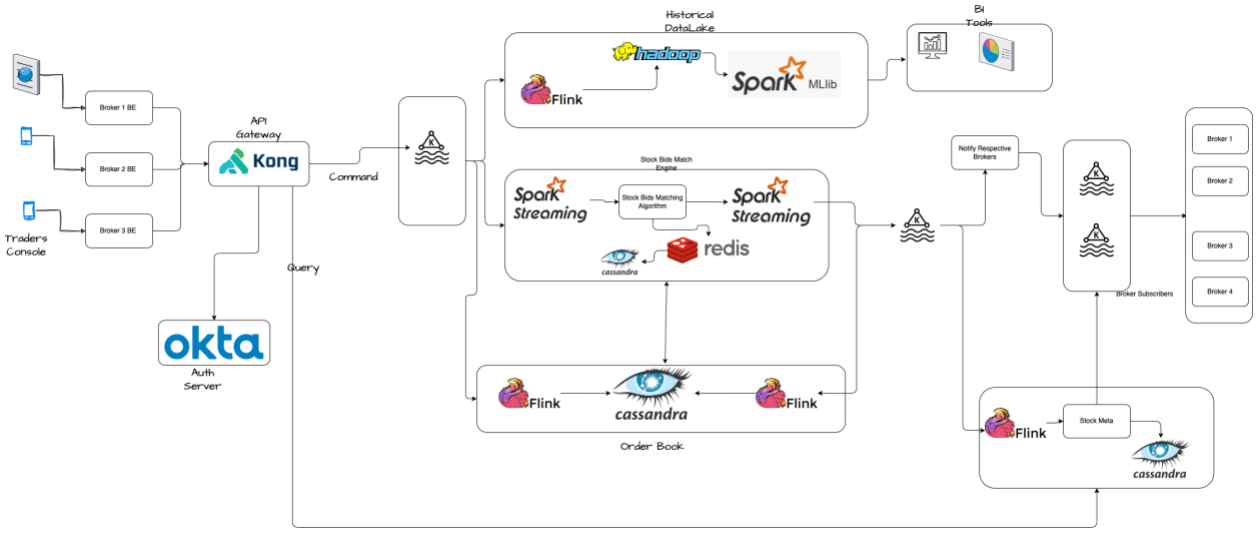


Figure-1

API Gateway -

An API gateway can play a crucial role in an electronic stock exchange application by providing a single-entry point for all incoming requests from various clients, such as trading bots, mobile applications, and web applications. The API gateway can perform a variety of functions to enhance the security, scalability, and reliability of the application. Such as

- authenticate and authorize incoming requests,
- verifying API keys and credentials,
- manage traffic flow and load balancing.
- can also perform caching, logging, and monitoring functions.

Why Kong ?

Kong is one of the leading Api Gateway which supports all the above-mentioned roles of api gateway that is why we mentioned Kong.

Authentication Server

One of the primary roles of an Authentication Server in an electronic stock exchange application is to verify the identity of users. The server can authenticate users using a variety of mechanisms, such as username and password, two-factor authentication, biometric authentication, or digital certificates.

Authentication Server can also perform authorization, ensuring that users have the necessary permissions to access specific resources or perform certain actions.

Why Okta?

Okta platform provides all above-mentioned benefits other than being an identity and access management platform.

Message Broker (Incoming orders Or Trade Orders)

A Message Broker can play a crucial role in a stock exchange application by providing a scalable and reliable messaging infrastructure that enables efficient communication between different components of the system.
Message broker is Scalable, Reliable and Decoupled tool which will consume all the incoming orders from various broker and persist in the sequence.

Why Kafka?

Kafka is a distributed messaging system that provides several benefits for businesses and organizations that need to process and analyze large amounts of data in real-time. Some of the reasons why Kafka is used include:

1. Scalability
2. Durability
3. Real-time processing
4. Flexibility
5. Integration

Instrument Bid Matching Engine

A match engine is a critical component in a stock exchange application that facilitates the matching of buy and sell orders for different financial instruments, such as stocks, bonds, and derivatives. The role of a match engine in a stock exchange application includes the following:

1. Order matching:
2. Price discovery
3. Trade execution
4. Order management
5. Scalability

Why Spark Streaming ?

Spark Streaming is a real-time processing framework that is part of the Apache Spark project.

1. Real-time processing:
2. Scalability
3. Fault tolerance

In this demo, for the ease of set up we are using ksqlDB SQL Engine

ksqlDB is a distributed streaming SQL engine that is built on top of Apache Kafka. It allows businesses to process, analyze, and query real-time streaming data using a familiar SQL-like syntax. Some of the benefits of using ksqlDB include:

1. Real-time data processing
2. SQL-like syntax
3. Scalability
4. Stream processing
5. Easy Integration with Kafka

Order Book

An Order Book system plays a crucial role in a stock exchange application by maintaining a record

- Order management
- Transparency

Notification Engine

A Notification Engine can play a critical role in a stock exchange application by providing real-time notifications to market participants about important events, such as the execution of trades, changes in prices, and other market conditions. The role of a Notification Engine in a stock exchange application includes the following:

1. Real-time notifications
2. Customization
3. Integration
4. Security

Historical Data layer

A Historical Data layer in a stock exchange application is responsible for collecting and storing historical market data, such as prices, volumes, and trades, over a period of time. The role of a Historical Data layer in a stock exchange application includes the following:

- Market analysis
- Regulatory compliance
- Backtesting
- Risk management
- Business intelligence

Why Hadoop Distributed File System?

HDFS is a distributed file system designed to store and manage large amounts of data across multiple nodes in a cluster. It is often used to store historical data for several reasons:

- Cost effectiveness
- Scalability
- Fault Tolerance
- Easy access
- Data retention

Why Spark Machine learning?

Spark Machine Learning (ML) is a powerful tool for businesses looking to develop and deploy machine learning models at scale. Some of the benefits of using Spark ML include:

- Performance
- Flexibility
- Integration
- Ease of use

CQRS

CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security.
Here, We are trying to present the API architecture as CQRS, Broker will push Order place command to one application i.e Match Stream but will read the changes from the other application i.e stock meta/Order Book (explained in Figure 1)

Assumptions:

1. As we are building solution for electronic exchange like Nasdaq or NSE. We would be receiving orders from various brokers like Zerodha, Icici direct or Moti Oswal on the behalf of traders who have chosen these various brokers for their various reasons.
2. Our system should be generic/scalable enough to handle multiple lists of brokers without extra code.
3. Our Stock Exchange only fulfil Full order not supporting partial Order execution.
4. API Model is kind of CQRS Pattern. Mock Exchange API will consume Order and push to streaming engine for the Query Broker will invoke separate system that will be Order Book. Or Order Book's Notification engine will invoke Brokers BE system in async mode soft real time mode.

Exercise 2: Order Producer [3 marks]

Write a Python program that enables the traders to place various types of orders on the variety of instruments traded on the electronic exchange.

- Program should first read the list of instruments (with relevant details) available on the exchange.
- Based on the received data from the exchange, trader should be able to place an order which is valid for certain time only.
- You have to write your own code for order placement.
- Add comments at appropriate place so that it's easy to understand your thought process.
- The program should clearly output the order details on the console.

Solution:

Here, we have implemented Client and mockExchangeServerAPI

Client - Broker's Jupiter notebook

Which will behave as a client, using which trader will be placing orders and monitoring the status. In the real world, this are not a command-based tool, but features enrich web UI.

File - SPAAssignment2.ipynb

Download - https://github.com/RitreshGirdhar/StockExchangeDemo/blob/master/brokersClient/SPAAssignment2.ipynb

Mock Exchange Server API

- Mock Stock Exchange API will receive the Order post request and push the message to message broker Kafka.
- Framework used for Backend API is FLASK
- It will host rest api on localhost 5000
- Download - https://github.com/RitreshGirdhar/StockExchangeDemo/blob/master/mockExchangeServer/spaexchange.py

How to start

```
FLASK_APP=spaexchange.py flask run --debugger
```

Exercise 3: Exchange – Match Maker [5 marks]

As an outcome of Exercise 2, your system will receive the orders placed by various traders for different instruments.

- Apply a match making algorithm so that the trades are completed for the instruments.
    - Explain your logic behind match making algorithm.
    - Program should output both the orders resulted into a trade.

Market makers could listen to the trades and decide the current price of the stock and populate to the brokers and customers.

Stock data analysis is used by investors and traders to make critical decisions related to the stocks. Investors and traders study and evaluate past and current stock data and attempt to gain an edge in the market by making decisions based on the insights obtained through the analyses.

As explained in the figure1 – Architecture of Electronic exchange. We mentioned that the order match maker will be a stream-based systems.

In this assignment we are using most popular stream server by https://www.confluent.io/ written on top of best open-source message broker application that is Apache Kafka

Design

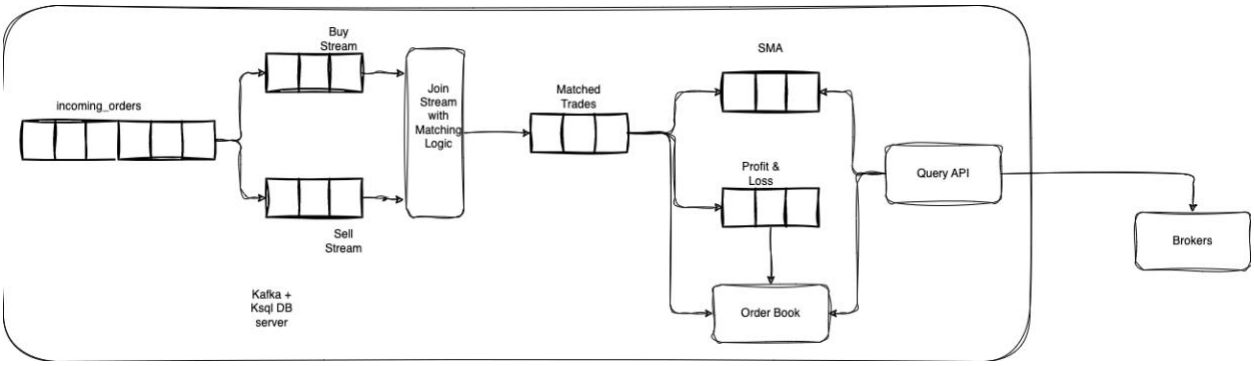


Figure 2 – Actual Implementation

Layers implemented for the assignments

1. Incoming orders - for all active orders received from the various brokers.
2. Buy Orders – Will contain all the orders with type BUY.
3. Sell Orders - Will contain all the orders with type SELL.
4. Match Making Join – Will join both the buy stream and sell stream based on the instrument and the buy.price <= sell.price. Those matched records will be pushed to next immediate stream that is Matched Stream

5. Matched Stream – Will contain all the orders which got executed along with buyer and select order id and the quantity and the price. This stream will be consumed by
    a. Order Book system to update the orders and trade information.
    b. Simple Moving Average stream to calculate the average price.
    c. Profit and Loss Stream to calculate profit or loss for each instrument.
6. Simple Moving Average Stream
7. Profit and Loss Stream

Install : Tools/Software

kafka_2.11-2.4.0
confluent-7.3.2
Open JDK 1.8
Python 3.1.2
Anaconda/Jupyter Notebook

Start Zookeeper > bin/zookeeper-server-start.sh config/zookeeper.properties
Start Kafka > bin/kafka-server-start.sh config/server.properties
Start Confluent KsqlDb > bin/ksql-server-start ./etc/ksqldb/ksql-server.properties
Create topic (kafka) > ./kafka-console-producer.sh --broker-list localhost:9092 --topic incoming_orders
KSQLDB SQL Engine > bin/ksql http://localhost:8088

What is Stream and Table in KSQLDB

KSQLDB is an open-source, distributed streaming SQL engine that provides an easy way to process, analyze and query real-time data streams using SQL-like syntax. In KSQLDB, there are two primary data types: streams and tables.

- A stream in KSQLDB is an unbounded sequence of immutable events or records, with each event containing a key and a value. Streams are used to represent continuous, real-time data and are best suited for situations where data is constantly changing, such as event processing or real-time analytics.

- A table in KSQLDB is a structured, immutable collection of data, with each record containing a key and a value. Tables are used to represent a snapshot of data at a specific point in time and are best suited for situations where data is static or changes infrequently, such as reference data or materialized views. Tables can be used to join with streams, aggregate data, filter data, and perform lookups.

Let's Start - Create Following Streams

Incoming_orders Stream

It will receive the list of orders placed by trader's broker system for BUY or SELL side.

```
-- Create a stream for incoming orders
CREATE STREAM incoming_orders (
   order_id VARCHAR,
   stock_symbol VARCHAR,
   quantity INT,
   side VARCHAR,
   order_type VARCHAR,
   price DOUBLE,
   time_in_force VARCHAR
) WITH (
   KAFKA_TOPIC='incoming_orders',
   VALUE_FORMAT='JSON',
   PARTITIONS=1
);
```

Output

```
(base) ritgirdh@WKMIN1307242 confluent-7.3.2 % ./bin/ksql http://localhost:8088

                  ===========================================
                  =       _              _ ____  ____       =
                  =      | | _____  __ _| |  _ \| __ )      =
                  =      | |/ / __|/ _` | | | | |  _ \      =
                  =      |   <\__ \ (_| | | |_| | |_) |     =
                  =      |_|\_\___/\__, |_|____/|____/      =
                  =                   |_|                   =
                  =        The Database purpose-built       =
                  =        for stream processing apps       =
                  ===========================================

Copyright 2017-2022 Confluent Inc.

CLI v7.3.2, Server v7.3.2 located at http://localhost:8088
Server Status: RUNNING

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
ksql> -- Create a stream for incoming orders
>CREATE STREAM incoming_orders (
>    order_id VARCHAR,
>    stock_symbol VARCHAR,
>    quantity INT,
>    side VARCHAR,
>    order_type VARCHAR,
>    price DOUBLE,
>    time_in_force VARCHAR
>) WITH (
>    KAFKA_TOPIC='incoming_orders',
>    VALUE_FORMAT='JSON',
>    PARTITIONS=1
>);
>

 Message
----------------
 Stream created
----------------
```

Buy_orders Stream

- It will filter the data from Incoming_orders stream and consume only BUY Order.
- In the practical world, we could add separate stream for the type of Instrument or further divide till the level of individual Stock. But with each layer complexity of handling traffic, processing orders immediately and consistency issues surged.

```
-- Create a stream for buy orders
CREATE STREAM buy_orders AS
SELECT *
FROM incoming_orders
WHERE side = 'BUY';
```

```
```

Output

```
ksql> -- Create a stream for buy orders
>CREATE STREAM buy_orders AS
>SELECT *
>FROM incoming_orders
>WHERE side = 'BUY';
>

 Message
----------------------------------------
 Created query with ID CSAS_BUY_ORDERS_3
----------------------------------------
```

Sell Orders Stream

It will filter the data from Incoming_orders stream and consume only SELL Order

```
-- Create a stream for sell orders

CREATE STREAM sell_orders AS
SELECT *
FROM incoming_orders
WHERE side = 'SELL';
```

Output

```
>CREATE STREAM sell_orders AS
>SELECT *
>FROM incoming_orders
>WHERE side = 'SELL';
>

 Message
----------------------------------------
 Created query with ID CSAS_SELL_ORDERS_5
----------------------------------------
ksql>
```

Matched trades Stream

Here we are Joining BUY_orders and SELL_orders stream based on the stock symbol and SELL Price and the Quantity.

Criteria

- Buy and Sell stock Symbol should be same
- Quantity should be least of Sell Quantity and buy Quantity
- Sell Price should be less than equal to Buy Price.

```
-- Join the buy and sell orders streams on the stock symbol and price
CREATE STREAM matched_trades AS
SELECT
  buy.order_id AS buy_order_id,
  sell.order_id AS sell_order_id,
  buy.stock_symbol,
  buy.price AS buy_price,
  sell.price AS sell_price,
  LEAST(buy.quantity, sell.quantity) AS quantity
FROM
  buy_orders buy
  INNER JOIN sell_orders sell
  WITHIN 8 HOURS
  ON buy.stock_symbol = sell.stock_symbol
  where buy.price >= sell.price and sell.quantity=buy.quantity;
```

Output

```
ksql> -- Join the buy and sell orders streams on the stock symbol and price
>CREATE STREAM matched_trades AS
>SELECT
>  buy.order_id AS buy_order_id,
>  sell.order_id AS sell_order_id,
>  buy.stock_symbol,
>  buy.price AS buy_price,
>  sell.price AS sell_price,
>  LEAST(buy.quantity, sell.quantity) AS quantity
>FROM
>  buy_orders buy
>  INNER JOIN sell_orders sell
>  WITHIN 8 HOURS
>  ON buy.stock_symbol = sell.stock_symbol
>  where buy.price >= sell.price and sell.quantity=buy.quantity;
>

 Message
----------------------------------------------
 Created query with ID CSAS_MATCHED_TRADES_7
----------------------------------------------
WARNING: DEPRECATION NOTICE: Stream-stream joins statements without a GRACE PERIOD will not be accepted in a future ksqlDB version.
Please use the GRACE PERIOD clause as specified in https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-reference/select-push-query/
ksql>
```

This matched_trades topic will be subscribe by order_book and then execution will happen.

To subscribe to this topic use below kafka command

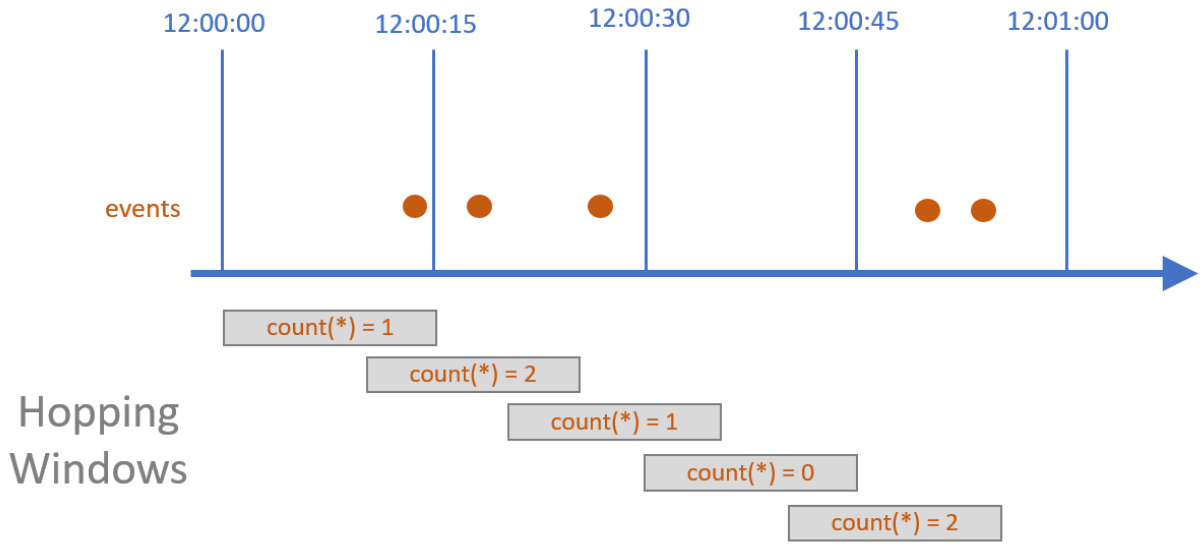
```
cd <kafka_home>

./bin/kafka-console-consumer.sh --broker-list localhost:9092 –topic matched_trades
```

Exercise 4: Exchange – Simple Moving Average Calculator [2 marks]
As an outcome of Exercise 3, your system has generated the trades for the various instruments. The simple moving average can be used to identify buying and selling opportunities. Moving averages are one of the core indicators in technical analysis, and there are a variety of different versions. SMA is the easiest moving average to construct. It is simply the average price over the specified period. The average is called "moving" because it is plotted on the chart bar by bar, forming a line that moves along the chart as the average value changes.

- Closing prices are used mostly by the traders and investors as it reflects the price at which the market finally settles down. The SMA (Simple Moving Average) is a parameter used to find the average stock price over a certain period based on a set of parameters
- The simple moving average is calculated by adding a stock's prices over a certain period and dividing the sum by the total number of periods.
- Calculate the simple moving average closing price of the four instruments in a 5-minute sliding window for the last 10 minutes.

Solution
We are implementing SMA using the concept **Hopping Windows**

What is Hopping Windows in Data Streaming?
Hopping windows group events for aggregation. Hopping windows are equal in duration but overlap at regular intervals. They can help with regular calculations where the time basis for aggregation is different from the frequency at which the calculation should be performed.



The advantage of using a hopping window is that it can provide a continuous stream of results without waiting for the entire data set to be collected. This makes it well-suited for Simple Moving Average Processing.

Create stream for Simple Moving Average for the last 10 minutes and 5-minute sliding window

```
CREATE TABLE sma_hopping_table AS
SELECT BUY_STOCK_SYMBOL,
AS_VALUE(BUY_STOCK_SYMBOL) as STOCK_SYMBOL,
AVG(SELL_PRICE) AS sma
FROM matched_trades
WINDOW HOPPING (SIZE 10 MINUTES, ADVANCE BY 5 MINUTES)
GROUP BY BUY_STOCK_SYMBOL
EMIT CHANGES;
```

```
ksql> CREATE TABLE sma_hopping_table AS
>SELECT BUY_STOCK_SYMBOL,
>AS_VALUE(BUY_STOCK_SYMBOL) as STOCK_SYMBOL,
>AVG(SELL_PRICE) AS sma
>FROM matched_trades
>WINDOW HOPPING (SIZE 10 MINUTES, ADVANCE BY 5 MINUTES)
>GROUP BY BUY_STOCK_SYMBOL
>EMIT CHANGES;
>

 Message
----------------------------------------------
 Created query with ID CTAS_SMA_HOPPING_TABLE_9
----------------------------------------------
```

Test

Kafka Topic which will publish the events to Market Maker

```
> ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic SMA_HOPPING_TABLE
```

```
(base) ritgirdh@WKMIN1307242 kafka_2.11-2.4.0 % ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic SMA_HOPPING_TABLE

{"STOCK_SYMBOL":"AMZN","SMA":2935.984734255091}
{"STOCK_SYMBOL":"AMZN","SMA":2935.984734255091}
{"STOCK_SYMBOL":"GOOG","SMA":1130.2238827258261}
{"STOCK_SYMBOL":"GOOG","SMA":1130.2238827258261}
{"STOCK_SYMBOL":"FB","SMA":386.6136568848735}
{"STOCK_SYMBOL":"FB","SMA":386.6136568848735}
{"STOCK_SYMBOL":"TSLA","SMA":741.0650647668283}
{"STOCK_SYMBOL":"TSLA","SMA":741.0650647668283}
{"STOCK_SYMBOL":"AAPL","SMA":156.36297672193854}
{"STOCK_SYMBOL":"AAPL","SMA":156.36297672193854}
{"STOCK_SYMBOL":"FB","SMA":386.160639200221}
{"STOCK_SYMBOL":"FB","SMA":386.160639200221}
{"STOCK_SYMBOL":"AMZN","SMA":2934.8251935733765}
{"STOCK_SYMBOL":"AMZN","SMA":2934.8251935733765}
{"STOCK_SYMBOL":"AAPL","SMA":156.09327619539192}
{"STOCK_SYMBOL":"AAPL","SMA":156.09327619539192}
{"STOCK_SYMBOL":"GOOG","SMA":1134.7676235234326}
{"STOCK_SYMBOL":"GOOG","SMA":1134.7676235234326}
{"STOCK_SYMBOL":"TSLA","SMA":741.1331934272822}
{"STOCK_SYMBOL":"TSLA","SMA":741.1331934272822}
{"STOCK_SYMBOL":"GOOG","SMA":1135.041535996558}
```

In the real world application, SMA not only have streaming window but also some other components which consume the messages from the kafka and process and persist into database. also broadcast the changes to brokers or other components of the architecture like historical database.

```
ksql> show tables
>;

 Table Name        | Kafka Topic       | Key Format | Value Format | Windowed
--------------------------------------------------------------------------------
 SMA_HOPPING_TABLE | SMA_HOPPING_TABLE | KAFKA      | JSON         | true
--------------------------------------------------------------------------------
ksql> show streams;

 Stream Name         | Kafka Topic                 | Key Format | Value Format | Windowed
-----------------------------------------------------------------------------------------
 BUY_ORDERS          | BUY_ORDERS                  | KAFKA      | JSON         | false
 INCOMING_ORDERS     | incoming_orders             | KAFKA      | JSON         | false
 KSQL_PROCESSING_LOG | default_ksql_processing_log | KAFKA      | JSON         | false
 MATCHED_TRADES      | MATCHED_TRADES              | KAFKA      | JSON         | false
 SELL_ORDERS         | SELL_ORDERS                 | KAFKA      | JSON         | false
-----------------------------------------------------------------------------------------
```

Exercise 5: Exchange – Profit Calculator [2 marks]
Find the stock out of the four instruments giving maximum profit (average closing price - average opening price) in a 5-minute sliding window for the last 10 minutes.

Profit and Loss Stream – will be responsible to calculate the profit and loss per instrument.
Create stream for profit & loss calculation in the 5-minute sliding window for the last 10 minutes.

Solution

```
CREATE TABLE opening_prices AS
SELECT
  BUY_STOCK_SYMBOL as stock_symbol,
  MIN(buy_price) AS opening_price
FROM matched_trades
WINDOW TUMBLING (SIZE 300000 SECONDS)
GROUP BY BUY_STOCK_SYMBOL;
```

```
ksql> CREATE TABLE opening_prices AS
>SELECT
>  BUY_STOCK_SYMBOL as stock_symbol,
>  MIN(buy_price) AS opening_price
>FROM matched_trades
>WINDOW TUMBLING (SIZE 300000 SECONDS)
>GROUP BY BUY_STOCK_SYMBOL;

 Message
----------------------------------------------
 Created query with ID CTAS_OPENING_PRICES_13
----------------------------------------------
```

```
-- Step 3: Calculate closing price for each stock
CREATE TABLE closing_prices AS
SELECT
  BUY_STOCK_SYMBOL as stock_symbol,
  MAX(sell_price) AS closing_price
FROM
  matched_trades
WINDOW TUMBLING (SIZE 300000 SECONDS)
GROUP BY
  BUY_STOCK_SYMBOL
```

```
ksql> -- Step 3: Calculate closing price for each stock
>CREATE TABLE closing_prices AS
>SELECT
>  BUY_STOCK_SYMBOL as stock_symbol,
>  MAX(sell_price) AS closing_price
>FROM
>  matched_trades
>WINDOW TUMBLING (SIZE 300000 SECONDS)
>GROUP BY
>  BUY_STOCK_SYMBOL
>;

 Message
--------------------------------------------
 Created query with ID CTAS_CLOSING_PRICES_15
--------------------------------------------
```

```
CREATE TABLE stock_profit AS
SELECT
  opening_prices.stock_symbol,
  (closing_price - opening_price) AS profit
FROM
  opening_prices
  INNER JOIN closing_prices
  ON opening_prices.stock_symbol = closing_prices.stock_symbol
  WINDOW TUMBLING (SIZE 600000 SECONDS)
  GROUP BY opening_prices.stock_symbol
  EMIT CHANGES;
```

Test output

```
> ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic  stock_profit
```

Demo Recording
https://drive.google.com/drive/folders/1lQXtNNzUiorc8h6_VtN3yB5W95bdelg9?usp=share_link

Github Repo - https://github.com/RitreshGirdhar/StockExchangeDemo.git

We have uploaded following document.

1. Client Program to allow Trader to place BUY or SELL orders.
2. Client Program contains Sample code to generate orders in Bulk
3. Demo Video
4. Mock Exchange API to consume Brokers order and ingest into Kafka stream.

References

https://kafka.apache.org/quickstart
https://docs.ksqldb.io/en/latest/
https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3
https://www.oreilly.com/library/view/stream-analytics-with/9781788395908/0b6a6289-6fd8-4e89-8eb1-51d5da579f59.xhtml
What Happens When You Buy or Sell Stocks?
How Stock Trading Works?
Kaggle Stock Market Analysis
Simple Moving Average (SMA)
Zerodha Kite Connect 3 / API documentation

****