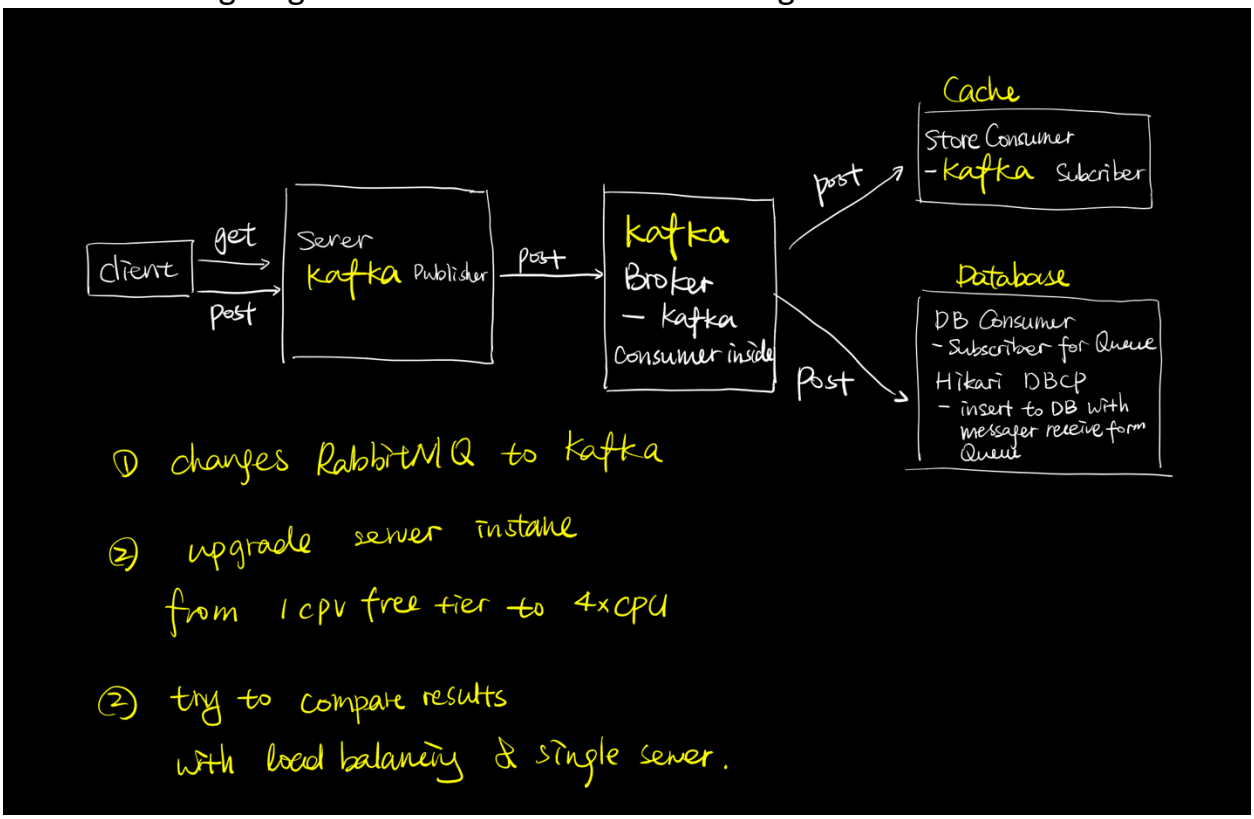
Student Name: Siqi Li
GitHub Repo Under NEU Organization:
https://github.ccs.neu.edu/lisiqi/CS6650-DSBS-Repo

Explanation About Client Structure and How it works:

1. Try to replace RabbitMQ with Kafka
2. partitioning the RDS database into 4
3. different results for larger CPU instance
4. including single server test and load balancing



➔ *Total Three Packages @ "SuperMarketServer":*

MVC pattern used

- model: (general utils for Server)
  - PurchaseRecord:
    - the data access object for each purchase request received, contains attributes match the database purchases table for each column
- Utils: (utils for RabbitMQ and Kafka)
  - ChannelFactory: utils for building RabbitMQ connection
  - ChannelPool: utils for building RabbitMQ connection
  - RabbitMQClient: used to publish the purchase request to RabbitMQ
  - MyPartitioner: used for portioning message to different Kafka broker, partition into 4 broker that matched it with associated database

    - KafkaProducerObject: config Kafka producer to use PubSub model insert to DB
- tools: (the purchase servlet)
    - PurchaseServlet: the server that handles purchase requests
        - have a RabbitMQ ChannelPool to build connections with RabbitMQ
        - valid http request if contains required URL path as header information
        - valid http request body is not empty while have a HTTP Post request
        - Get information from the URL path and request body to generate a PurchaseRecord to publish into Kafka Broker
        - write captured information to response body, and send back

➔ *Total Two Class @ "StoreConsumer":*

- PurchaseRecord:
    - the data access object for each purchase request received, contains attributes match the database purchases table for each column
- RedisConsumer:
    - used to subscribe messages from Kafka broker
    - save received message in Redis cache with storeID as key associated with all itemID as member and its scores, and another cache with ItemID as key associated with all storeID as member and its scores

➔ *Database Design:*

- One table names "Puchases" to store all purchase request as each row input
- Table contains:
    - UUID as primary key, and avoid data duplication
    - StoreID as Integer
    - CustomerID as Integer
    - Data as String -> with the given date send by client2 purchase URL
    - CreateTime as Timestamp, takes the server system time while insert to DB
    - PurchaseBody as Json that contains all purchase items information

➔ *Total Three Packages @ "DBConsumer":*

- dal: (the DB connection)
    - DBCPDataSource:
        - Use HikariCP as the Database Connection Pool configuration
        - set a max pool size of 15
        - set max possible lifetime for connection in pool for 20 mins
    - PurchaseDao:
        - Setup and close Database Connection Pool (DBCP) connections
        - Insert data to purchase table with executing SQL queries
        - Enforce successful insertion, when insertion fail, keep inserting until successful, avoid failed insertion that give a failed http response back to client and cause a resend of duplicate purchase call
- model: (general utils for Server)

    - PurchaseRecord:
        - the data access object for each purchase request received, contains attributes match the database purchases table for each column
- Tools:
    - DBConsumer: The Database subscriber for Kafka, have four partitioned database based on Kafka broker partition, insert to different DB by mathing the purchase record's storeID % 4

➔ *Total Two Packages @ "SuperMarketClient":*

- model: (general utils for both client2)
    - Record: the record object used to store all latency results for Client2
    - LogStderr: the error logger for write out system error to err.txt file
    - HttpMethod: Enum for POST and GET
    - CmdParser: set up required and optional command line input options to run client2
- part2: (package for client 2 to generate requested outputs)
    - Clinet2:
        - Main for execute all threads by calling SingleThread in package part2
        - Check if command line arguments are valid
        - Generate output results for part 2, print out to console
    - SingleThread:
        - the single runnable object represents as one thread to make purchase requests with remote server
        - when send purchase request failed, retry maximum 3 times to make purchase request send successfully
        - Use AtomicInteger to count the total successful/unsuccessful requests
        - Use CountDownLatch to initiate phase 2 and phase 3 threads
        - designed for Client 2 to capture start and end time of each request, in order to get latency time
        - use LogStderr to log errors into txt file
    - RecordWriter:
        - Runnable object used to generate Records object and write to csv file
        - Using BlockingQueue, and work as a Consumer, when consider each SingleThread are all producers of the queue
    - ReportGenerator:
        - Generate output results for part 2, print out to console
        - Read in all Records stored in CSV and get all Latency in to a list and sort list
        - Analysis all latency list, and get mean, median, max and p99 response time

Below are the output results for Client2 and analysis Charts:

➔ Results on Database are all Free Tier RDS

Single Server

➔ *With 256 Threads Single Server in A2:*

```
[ec2-user@ip-172-31-26-136 ~]$ java -jar SuperMarketClient.jar -ip 3.90.186.171 -s 256
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: 3.90.186.171, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  93.342 milliseconds
4. Median response time for POSTs:  93.000 milliseconds
5. The total run time (wall time) for all phases to complete: 327.218 second
6. Throughput: 2112.353 requests/second
7. P99 (99th percentile) response time for POSTs: 286 milliseconds
8. Max response time for POSTs: 1456 milliseconds
```

➔ *With 256 Threads Single Server Using RabbitMQ and Redis in A3:*

➔ *USE 1*CPU server instance,* 4*CPU client, 8*CPU Ubuntu RabbitMQ server and put Redis on it

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 256 -ip 34.221.135.149
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: 34.221.135.149, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  68.489 milliseconds
4. Median response time for POSTs:  57.000 milliseconds
5. The total run time (wall time) for all phases to complete: 233.774 second
6. Throughput: 2956.702 requests/second
7. P99 (99th percentile) response time for POSTs: 246 milliseconds
8. Max response time for POSTs: 3190 milliseconds
```

Single Server for A2 VS A3 did show higher throughput with using the RabbitMQ technology

➔ *With 256 Threads Single Server with Kafka in A4:*

➔ *USE 1*CPU server instance,* 4 CPU client, 8 CPU Kafka server

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 256 -ip 18.237.246.42
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: 18.237.246.42, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  77.360 milliseconds
4. Median response time for POSTs:  3.000 milliseconds
5. The total run time (wall time) for all phases to complete: 271.909 second
6. Throughput: 2542.027 requests/second
7. P99 (99th percentile) response time for POSTs: 191 milliseconds
8. Max response time for POSTs: 26528 milliseconds
```

Single Server for A3 VS A4 show similar throughput, not about to compare difference between Kafka and RabbitMQ. But with comparison to the median and P99, Kafka is much better than RabbitMQ

Load Balancing

➔ *With 256 Threads Load Balanced Server with Hikari in A2:*

➔ *USE 4*CPU server instance, 4*CPU client*

```
[ec2-user@ip-172-31-26-20 ~]$ java -jar SuperMarketClient.jar -ip DSBSLoadBalancer-485728369.us-east-1.elb.amazonaws.com -s 256
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: DSBSLoadBalancer-485728369.us-east-1.elb.amazonaws.com, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  26.217 milliseconds
4. Median response time for POSTs:  15.000 milliseconds
5. The total run time (wall time) for all phases to complete: 94.82 second
6. Throughput: 7289.601 requests/second
7. P99 (99th percentile) response time for POSTs: 142 milliseconds
8. Max response time for POSTs: 367 milliseconds
```

➔ *With 256 Threads Load Balanced Server Using Non-persistent RabbitMQ and Redis in A3:*

➔ *USE 1*CPU server instance,* 4*CPU client, 8*CPU Ubuntu RabbitMQ server and Redis on it

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 256 -ip my-load-balancer-80924670.us-west-2.elb.amazonaws.com
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: my-load-balancer-80924670.us-west-2.elb.amazonaws.com, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 10
0000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  18.817 milliseconds
4. Median response time for POSTs:  5.000 milliseconds
5. The total run time (wall time) for all phases to complete: 106.543 second
6. Throughput: 6487.521 requests/second
7. P99 (99th percentile) response time for POSTs: 130 milliseconds
8. Max response time for POSTs: 60064 milliseconds
```

➔ *With 256 Threads Load Balanced Server Using Persistent RabbitMQ and Redis in A3:*

➔ *USE 1*CPU server instance,* 4*CPU client, 8*CPU Ubuntu RabbitMQ server and Redis on it

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 256 -ip my-load-balancer-80924670.us-west-2.elb.amazonaws.com
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: my-load-balancer-80924670.us-west-2.elb.amazonaws.com, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId: 1
0000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  23.626 milliseconds
4. Median response time for POSTs:  4.000 milliseconds
5. The total run time (wall time) for all phases to complete: 121.322 second
6. Throughput: 5697.235 requests/second
7. P99 (99th percentile) response time for POSTs: 135 milliseconds
8. Max response time for POSTs: 60070 milliseconds
```

➔ *With 256 Threads Load Balanced Server with Kafka and Hikari in A4:*

➔ *USE 1*CPU server instance,* 4 CPU client, 8 CPU Kafka server

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 256 -ip my-load-balancer-80924670.us-west-2.elb.amazonaws.com
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: my-load-balancer-80924670.us-west-2.elb.amazonaws.com, Max Store: 256, numOfCustomersPerStore: 1000, maxItemId:
100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 691200
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  25.646 milliseconds
4. Median response time for POSTs:  3.000 milliseconds
5. The total run time (wall time) for all phases to complete: 103.791 second
6. Throughput: 6659.537 requests/second
7. P99 (99th percentile) response time for POSTs: 97 milliseconds
8. Max response time for POSTs: 60133 milliseconds
[ec2-user@ip-10-0-0-251 ~]$
```

Load balanced Server for A3 VS A4 with 256 threads, with comparison to the median and P99, Kafka is much better than RabbitMQ no matter persistent queue or non-persistent, and also Kafka show slightly increase on throughput comparing to Persistent RabbitMQ. But feel like using different queue will same configuration of server, it is relatively similar, this result might due to the network latency, since the monitor shows CPU usage only rans below 50%, mighe be a bottle neck.

➔ *With 512 Threads Load Balanced Server in A2:*

```
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: DSBSLoadBalancer-485728369.us-east-1.elb.amazonaws.com, Max Store: 512, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 1382400
2. Total number of unsuccessful requests: 2
3. Mean response time for POSTs:  45.610 milliseconds
4. Median response time for POSTs:  12.000 milliseconds
5. The total run time (wall time) for all phases to complete: 164.513 second
6. Throughput: 8402.983 requests/second
7. P99 (99th percentile) response time for POSTs: 265 milliseconds
8. Max response time for POSTs: 7708 milliseconds
```

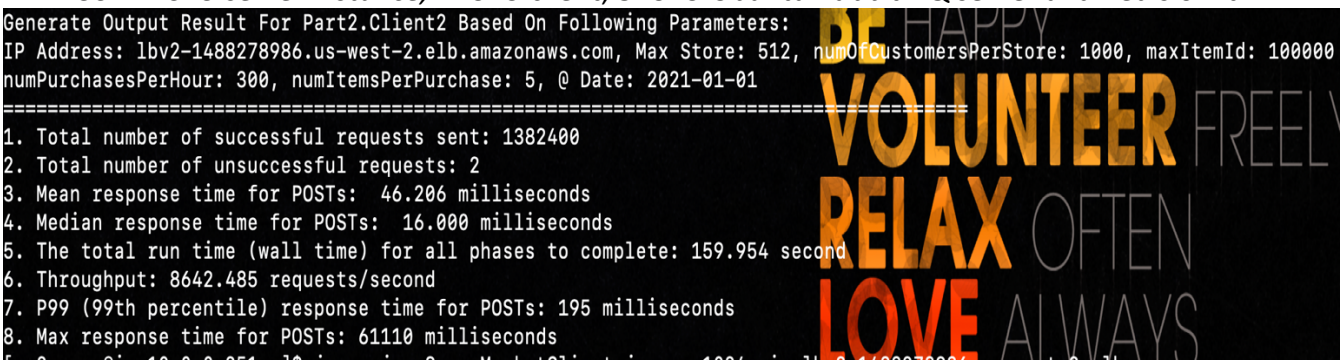
➔ *With 512 Threads Load Balanced Server Using RabbitMQ and Redis in A3:*

➔ *USE 1*CPU server instance,* 4*CPU client, 8*CPU Ubuntu RabbitMQ server and Redis on it

```
[ec2-user@ip-10-0-0-251 ~]$ java -jar SuperMarketClient.jar -s 512 -ip my-load-balancer-80924670.us-west-2.elb.amazonaws.com
Generate Output Result For Part2.Client2 Based On Following Parameters:
IP Address: my-load-balancer-80924670.us-west-2.elb.amazonaws.com, Max Store: 512, numOfCustomersPerStore: 1000, maxItemId: 100000
numPurchasesPerHour: 300, numItemsPerPurchase: 5, @ Date: 2021-01-01
================================================================================
1. Total number of successful requests sent: 1382400
2. Total number of unsuccessful requests: 0
3. Mean response time for POSTs:  39.452 milliseconds
4. Median response time for POSTs:  5.000 milliseconds
5. The total run time (wall time) for all phases to complete: 158.956 second
6. Throughput: 8696.746 requests/second
7. P99 (99th percentile) response time for POSTs: 312 milliseconds
8. Max response time for POSTs: 60169 milliseconds
[ec2-user@ip-10-0-0-251 ~]$
```

➔ *With 512 Threads Load Balanced Server Using in A4:*

➔ *USE 4*CPU server instance,* 4*CPU client, 8*CPU Ubuntu RabbitMQ server and Redis on it

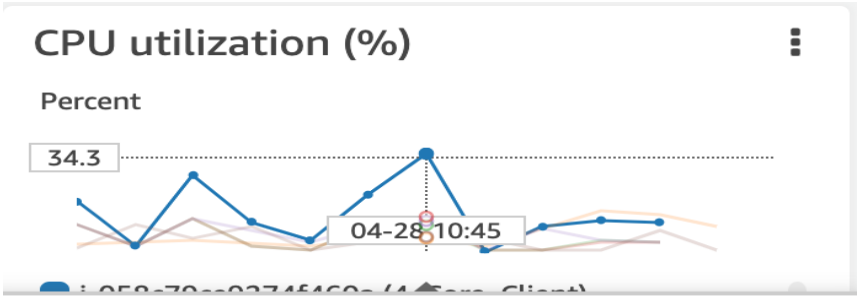


Conclusion:

Load balanced Server for A3 VS A4 with 512 threads, Kafka show slightly slower on throughput comparing to Persistent RabbitMQ, but I don't think it really make sense by looking at the monitor of CPU usages on server, it could be highly influenced by the network latency inside AWS VPC, or maybe I am not fully utilized Kafka's ability. But looking at the P99 and median, Kafka is faster on average for 256 or 512 threads on different machine configurations. So as of this point, thinking Kafka has better performance comparing to RabbitMQ, also I had partitioned the RDS insertion into 4, this could be having KafkaConsumer consume messages faster.

Failed to run on 1024 threads, API gateway time-out ☹

Below is the image of my cloud instance CPU utilization, so above testing has been highly influencing by internal configuration and network latency on VPC.



2021-04-28 10:45 UTC

1.	i-058c79ce9274f460a (4_Core_Client)	**35.775**
2.	i-02639002a34ad16e5 (ServerV2)	**12.6533435213**
3.	i-01458341e1378271c (ServerV3)	**11.125**
4.	i-0e533c473315bcad7 (ServerV1)	**9.64166666667**
5.	i-0b3b5d8b3fe0b283b (ServerV4)	**5.35833333333**
6.	i-0f50d8109183ddb66 (Kafka_Server)	**4.8**