Table of Contents

Real Time Data Analytics with Apache Kafka Stream

CS 6650 Building Scalable Distributed Systems Fall 2018

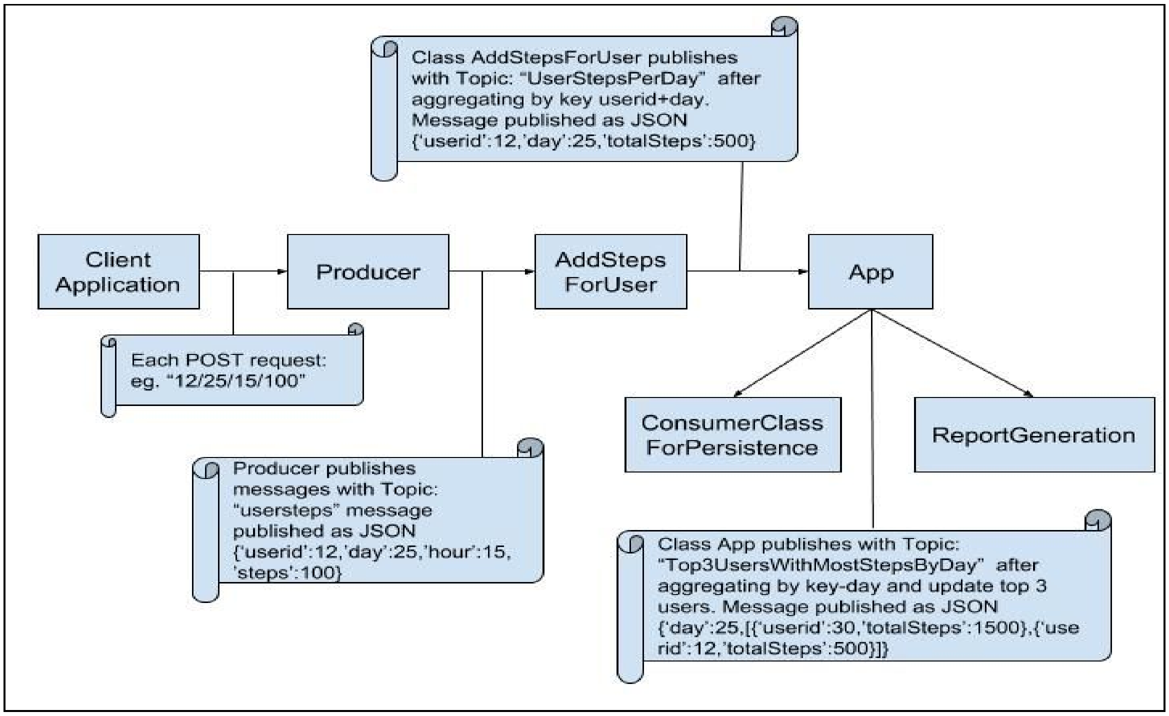
Team : Dessert Stream
Members: Survi Satpathy & Prakriti Dave

Hypothesis:

“Finding the Running Top N users who have the maximum number of steps per day using Apache Kafka Streams will be faster than traditional relational database systems.” To prove that the former is faster we do a latency comparison of the GET requests to find top 3 periodically using two applications - one with Apache Kafka Streaming and the other with MySQL.

The motivation behind this experimentation was to get an in hand experience of using Apache Kafka Streams for real time data analysis used for metric generation. Kafka is often used for operational monitoring of data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Project Implementation Idea for the Apache Kafka Application:



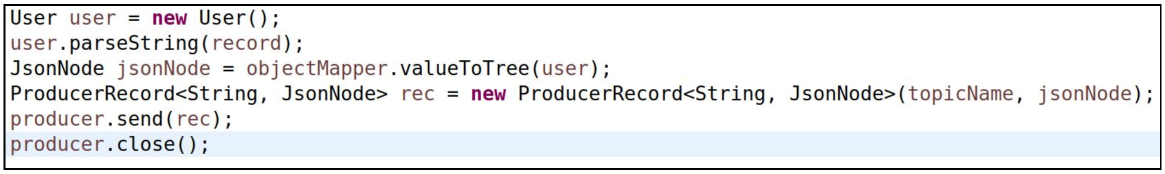
Class AddStepsForUser publishes with Topic: “UserStepsPerDay” after aggregating by key userid+day. Message published as JSON {‘userid’:12,’day’:25,’totalSteps’:500}

Each POST request: eg. “12/25/15/100”

Producer publishes messages with Topic: “usersteps” message published as JSON {‘userid’:12,’day’:25,’hour’:15, ’steps’:100}

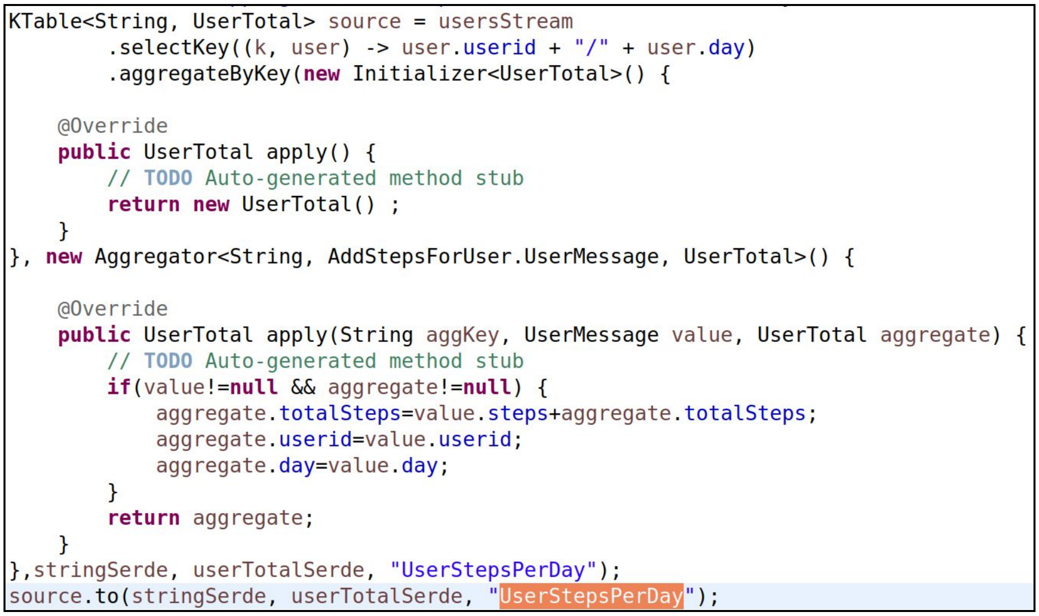
Class App publishes with Topic: “Top3UsersWithMostStepsByDay” after aggregating by key-day and update top 3 users. Message published as JSON {‘day’:25,[{‘userid’:30,’totalSteps’:1500},{‘userid’:12,’totalSteps’:500}]}

With the existing problem scenario i.e. Fitbit application with multithreaded POST requests generating data arranged as {‘UserID’:Integer,’Day’:Integer,’Hour’:Integer,’Steps’:Integer}; we pass each such request to the Apache Kafka Producer class. This uses the send API available from the Apache Library to send all the POST requests to a created topic - **‘usersteps’** . This is done in the **Producer.java** class.



```
User user = new User();
user.parseString(record);
JsonNode jsonNode = objectMapper.valueToTree(user);
ProducerRecord<String, JsonNode> rec = new ProducerRecord<String, JsonNode>(topicName, jsonNode);
producer.send(rec);
producer.close();
```

The Apache Kafka Stream class called **AddStepsPerUser.java** acts as the Consumer of the messages published by the ‘usersteps’ topic. This class aggregates the records by the key: userid. This aggregated result is streamed by a new KStream topic: ‘**UserStepsPerDay’**. The code below shows the KTable used to compute the aggregated steps per user filtered per day and streamed to the new topic-’UserStepsPerDay’.

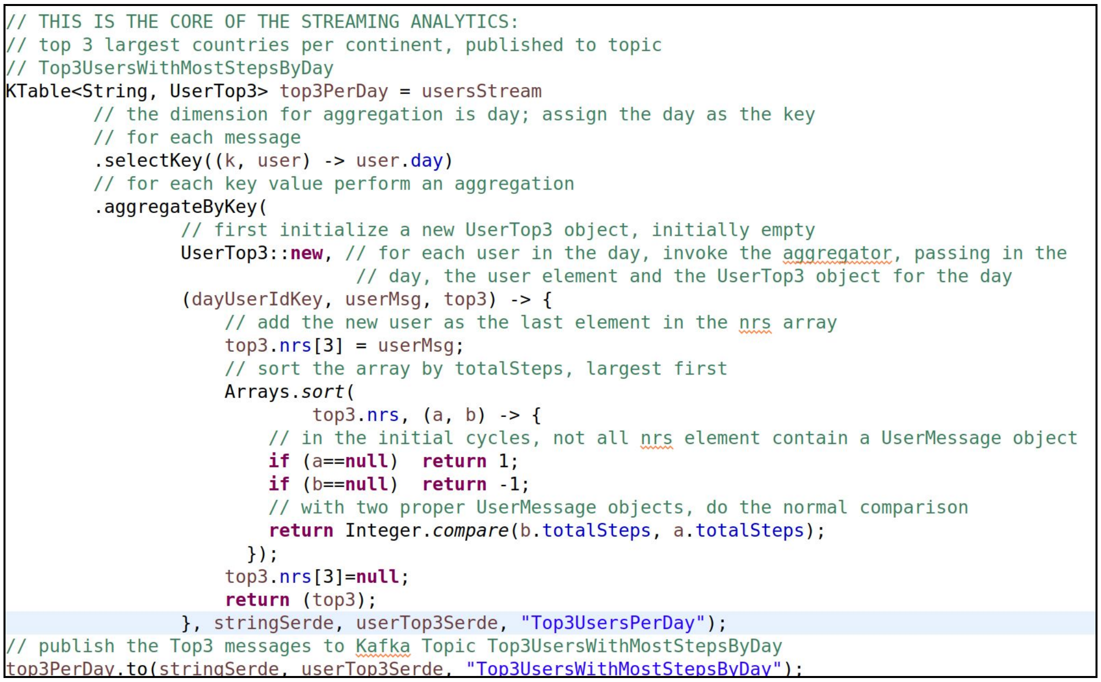


```
KTable<String, UserTotal> source = usersStream
        .selectKey((k, user) -> user.userid + "/" + user.day)
        .aggregateByKey(new Initializer<UserTotal>() {

    @Override
    public UserTotal apply() {
        // TODO Auto-generated method stub
        return new UserTotal() ;
    }
}, new Aggregator<String, AddStepsForUser.UserMessage, UserTotal>() {

    @Override
    public UserTotal apply(String aggKey, UserMessage value, UserTotal aggregate) {
        // TODO Auto-generated method stub
        if(value!=null && aggregate!=null) {
            aggregate.totalSteps=value.steps+aggregate.totalSteps;
            aggregate.userid=value.userid;
            aggregate.day=value.day;
        }
        return aggregate;
    }
},stringSerde, userTotalSerde, "UserStepsPerDay");
source.to(stringSerde, userTotalSerde, "UserStepsPerDay");
```

The messages streamed as above will be consumed by another KStream class called **App.java**. This class aggregates by the key as ‘day’. Aggregator implementation in lambda that performs the actual aggregation – the operation aggregateByKey (aggregateByKey) is invoked with an Initializer(Initializer) that returns the initial instance of the UerTop3 object (per day) on which the aggregate will be built, an Aggregator (Aggregator) that receives the day, the UserTop3 object and the next UserMessage and upgrades the UserTop3 object to include the new UserMessage, the Serdes (serializer/deserializer) for the key and the value and a String that is the name of the resulting KTable. Below is a screenshot of the KTable from App.java.



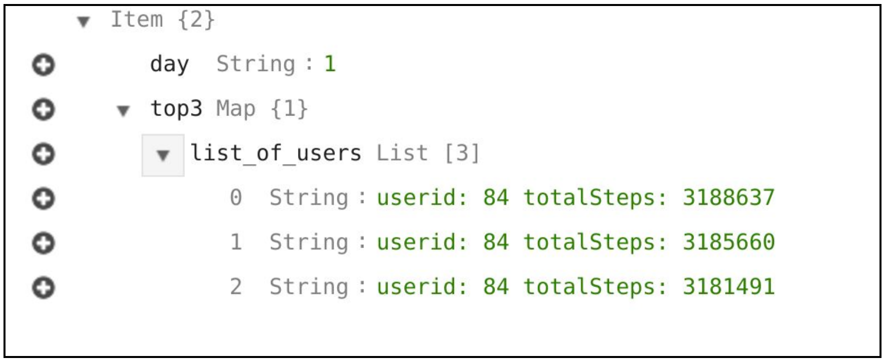
```
// THIS IS THE CORE OF THE STREAMING ANALYTICS:
// top 3 largest countries per continent, published to topic
// Top3UsersWithMostStepsByDay
KTable<String, UserTop3> top3PerDay = usersStream
        // the dimension for aggregation is day; assign the day as the key
        // for each message
        .selectKey((k, user) -> user.day)
        // for each key value perform an aggregation
        .aggregateByKey(
                // first initialize a new UserTop3 object, initially empty
                UserTop3::new, // for each user in the day, invoke the aggregator, passing in the
                                // day, the user element and the UserTop3 object for the day
                (dayUserIdKey, userMsg, top3) -> {
                    // add the new user as the last element in the nrs array
                    top3.nrs[3] = userMsg;
                    // sort the array by totalSteps, largest first
                    Arrays.sort(
                            top3.nrs, (a, b) -> {
                        // in the initial cycles, not all nrs element contain a UserMessage object
                        if (a==null)  return 1;
                        if (b==null)  return -1;
                        // with two proper UserMessage objects, do the normal comparison
                        return Integer.compare(b.totalSteps, a.totalSteps);
                      });
                    top3.nrs[3]=null;
                    return (top3);
                }, stringSerde, userTop3Serde, "Top3UsersPerDay");
// publish the Top3 messages to Kafka Topic Top3UsersWithMostStepsByDay
top3PerDay.to(stringSerde, userTop3Serde, "Top3UsersWithMostStepsByDay");
```

This generates the output as shown below to the topic '**Top3UsersWithMostStepsByDay**':

Note: The topic 'Top3UsersWithMostStepsByDay' is created with partition count as 2 because there as two consumer classes listening to this topic - ConsumeClassForPersistence.java and ReportGeneration.java

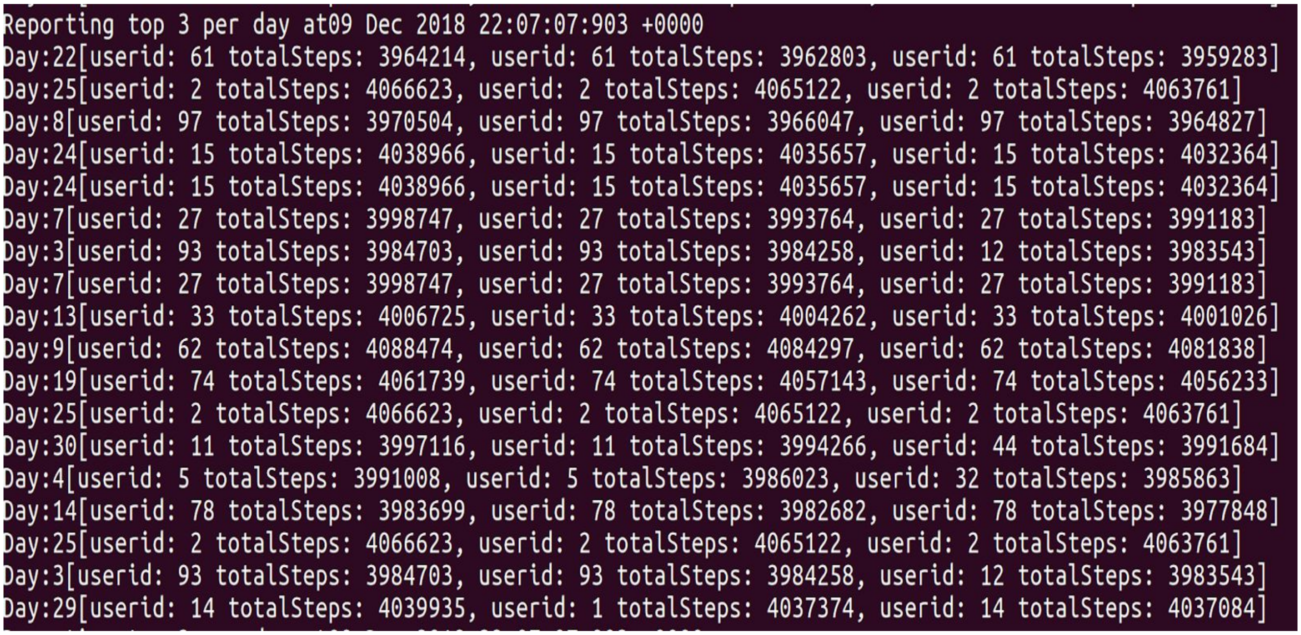
The Class '**ConsumeClassForPersistence.java**' is a Apache Kafka consumer class which listens to messages published by App.java and pushes them to a DynamoDB with primary key as 'day'. This is solely for the purpose of retaining the running top 3 users results generated per day. Storing the result in DB enable us the use the monthly data for statistical analysis in the future.

Below is the screenshot of one such row from the DynamoDB:



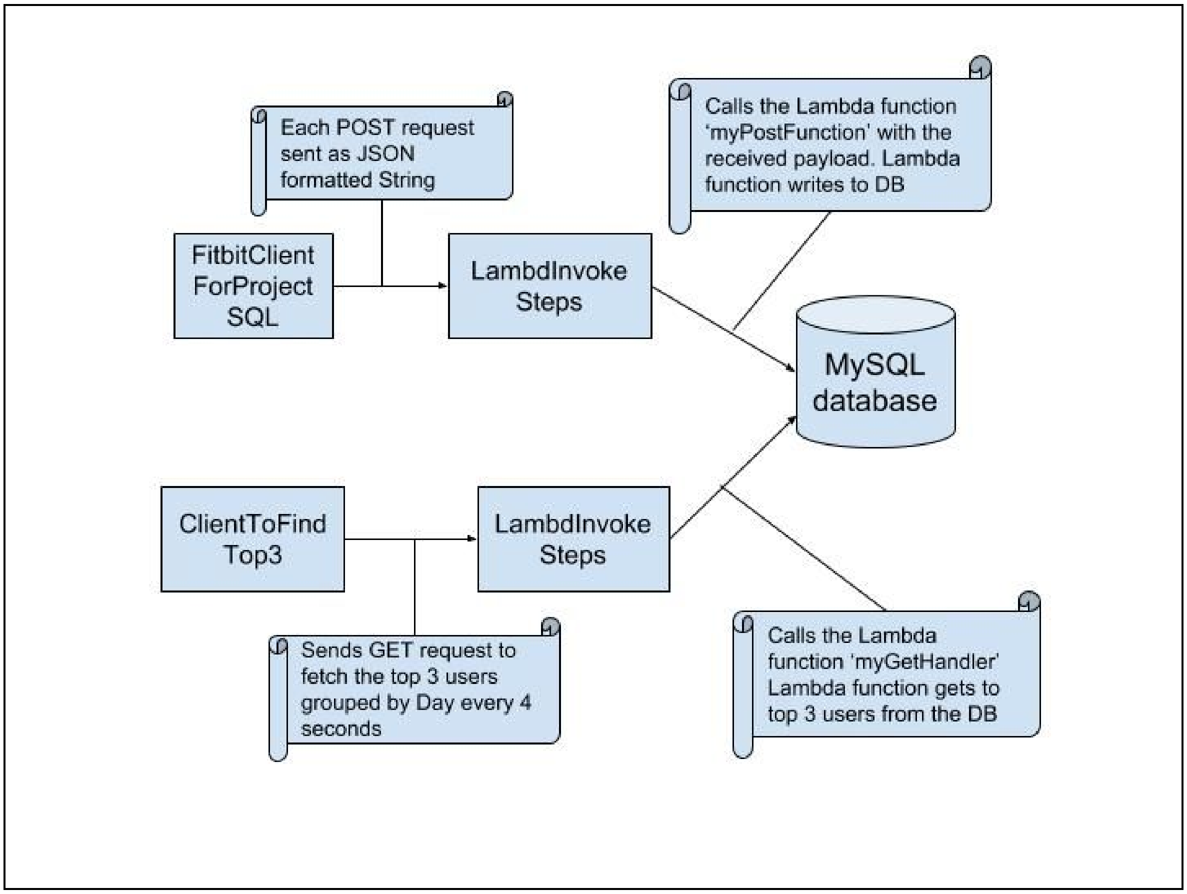
```
▼ Item {2}
    day  String : 1
  ▼ top3 Map {1}
      ▼ list_of_users List [3]
            0  String : userid: 84 totalSteps: 3188637
            1  String : userid: 84 totalSteps: 3185660
            2  String : userid: 84 totalSteps: 3181491
```

The Class ‘**ReportGeneration.java**’ is an Apache Kafka consumer class which **polls every 4 seconds** the messages published by App.java and generates a running log of the results. This is the class used for capturing the latency of each such poll operation. Below is the sample output of the ReportGeneration class. It logs the top 3 users for the day along with their steps for the day in decreasing order.

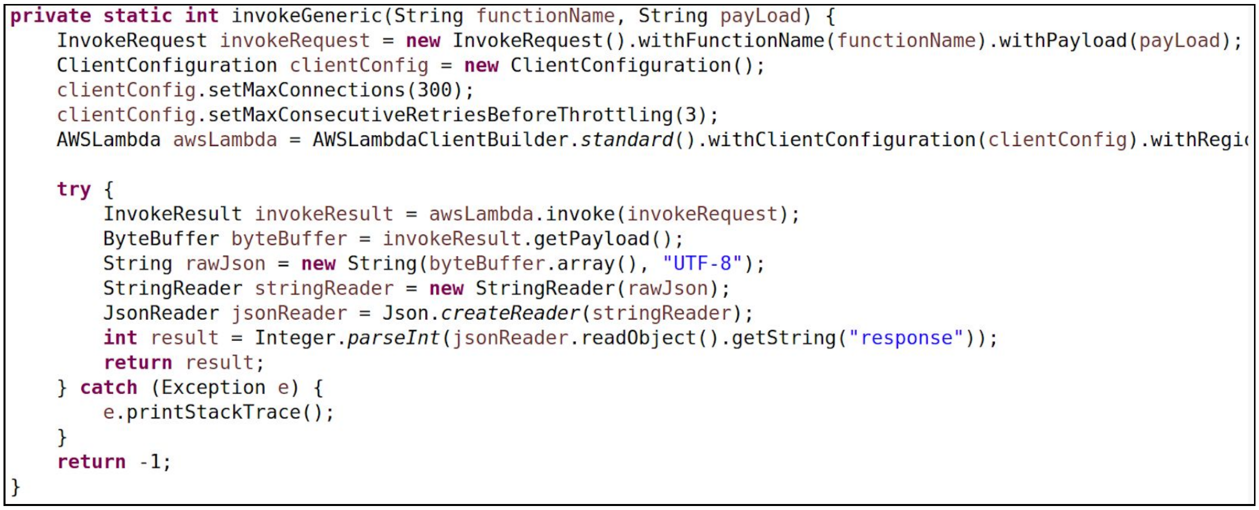


```
Reporting top 3 per day at09 Dec 2018 22:07:07:903 +0000
Day:22[userid: 61 totalSteps: 3964214, userid: 61 totalSteps: 3962803, userid: 61 totalSteps: 3959283]
Day:25[userid: 2 totalSteps: 4066623, userid: 2 totalSteps: 4065122, userid: 2 totalSteps: 4063761]
Day:8[userid: 97 totalSteps: 3970504, userid: 97 totalSteps: 3966047, userid: 97 totalSteps: 3964827]
Day:24[userid: 15 totalSteps: 4038966, userid: 15 totalSteps: 4035657, userid: 15 totalSteps: 4032364]
Day:24[userid: 15 totalSteps: 4038966, userid: 15 totalSteps: 4035657, userid: 15 totalSteps: 4032364]
Day:7[userid: 27 totalSteps: 3998747, userid: 27 totalSteps: 3993764, userid: 27 totalSteps: 3991183]
Day:3[userid: 93 totalSteps: 3984703, userid: 93 totalSteps: 3984258, userid: 12 totalSteps: 3983543]
Day:7[userid: 27 totalSteps: 3998747, userid: 27 totalSteps: 3993764, userid: 27 totalSteps: 3991183]
Day:13[userid: 33 totalSteps: 4006725, userid: 33 totalSteps: 4004262, userid: 33 totalSteps: 4001026]
Day:9[userid: 62 totalSteps: 4088474, userid: 62 totalSteps: 4084297, userid: 62 totalSteps: 4081838]
Day:19[userid: 74 totalSteps: 4061739, userid: 74 totalSteps: 4057143, userid: 74 totalSteps: 4056233]
Day:25[userid: 2 totalSteps: 4066623, userid: 2 totalSteps: 4065122, userid: 2 totalSteps: 4063761]
Day:30[userid: 11 totalSteps: 3997116, userid: 11 totalSteps: 3994266, userid: 44 totalSteps: 3991684]
Day:4[userid: 5 totalSteps: 3991008, userid: 5 totalSteps: 3986023, userid: 32 totalSteps: 3985863]
Day:14[userid: 78 totalSteps: 3983699, userid: 78 totalSteps: 3982682, userid: 78 totalSteps: 3977848]
Day:25[userid: 2 totalSteps: 4066623, userid: 2 totalSteps: 4065122, userid: 2 totalSteps: 4063761]
Day:3[userid: 93 totalSteps: 3984703, userid: 93 totalSteps: 3984258, userid: 12 totalSteps: 3983543]
Day:29[userid: 14 totalSteps: 4039935, userid: 1 totalSteps: 4037374, userid: 14 totalSteps: 4037084]
```

Project Implementation idea for Application with MySQL:


Each POST request sent as JSON formatted String
Calls the Lambda function 'myPostFunction' with the received payload. Lambda function writes to DB
FitbitClient ForProject SQL
LambdInvoke Steps
MySQL database
ClientToFind Top3
LambdInvoke Steps
Sends GET request to fetch the top 3 users grouped by Day every 4 seconds
Calls the Lambda function 'myGetHandler' Lambda function gets to top 3 users from the DB

The implementation idea for the second half of the project where we use traditional database is fairly similar to the last assignment on AWS Lambda. We have a class **'FitClienForProjectSQL.java'** which generates multi threaded POST requests to add data to the database. These requests as sent to the **'LambdaInvokeSteps.java'** class. This class has the code to call the respective lambda function with the payload for a user's steps. Below is the screenshot of the invoke function which takes as arguments the lambda function name and the User data as a String.

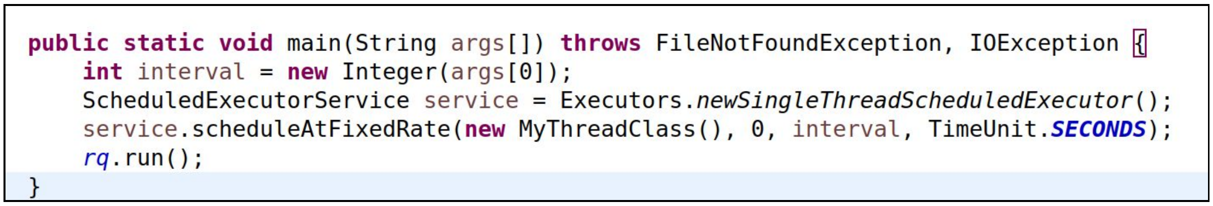


```
private static int invokeGeneric(String functionName, String payLoad) {
    InvokeRequest invokeRequest = new InvokeRequest().withFunctionName(functionName).withPayload(payLoad);
    ClientConfiguration clientConfig = new ClientConfiguration();
    clientConfig.setMaxConnections(300);
    clientConfig.setMaxConsecutiveRetriesBeforeThrottling(3);
    AWSLambda awsLambda = AWSLambdaClientBuilder.standard().withClientConfiguration(clientConfig).withRegi

    try {
        InvokeResult invokeResult = awsLambda.invoke(invokeRequest);
        ByteBuffer byteBuffer = invokeResult.getPayload();
        String rawJson = new String(byteBuffer.array(), "UTF-8");
        StringReader stringReader = new StringReader(rawJson);
        JsonReader jsonReader = Json.createReader(stringReader);
        int result = Integer.parseInt(jsonReader.readObject().getString("response"));
        return result;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return -1;
}
```

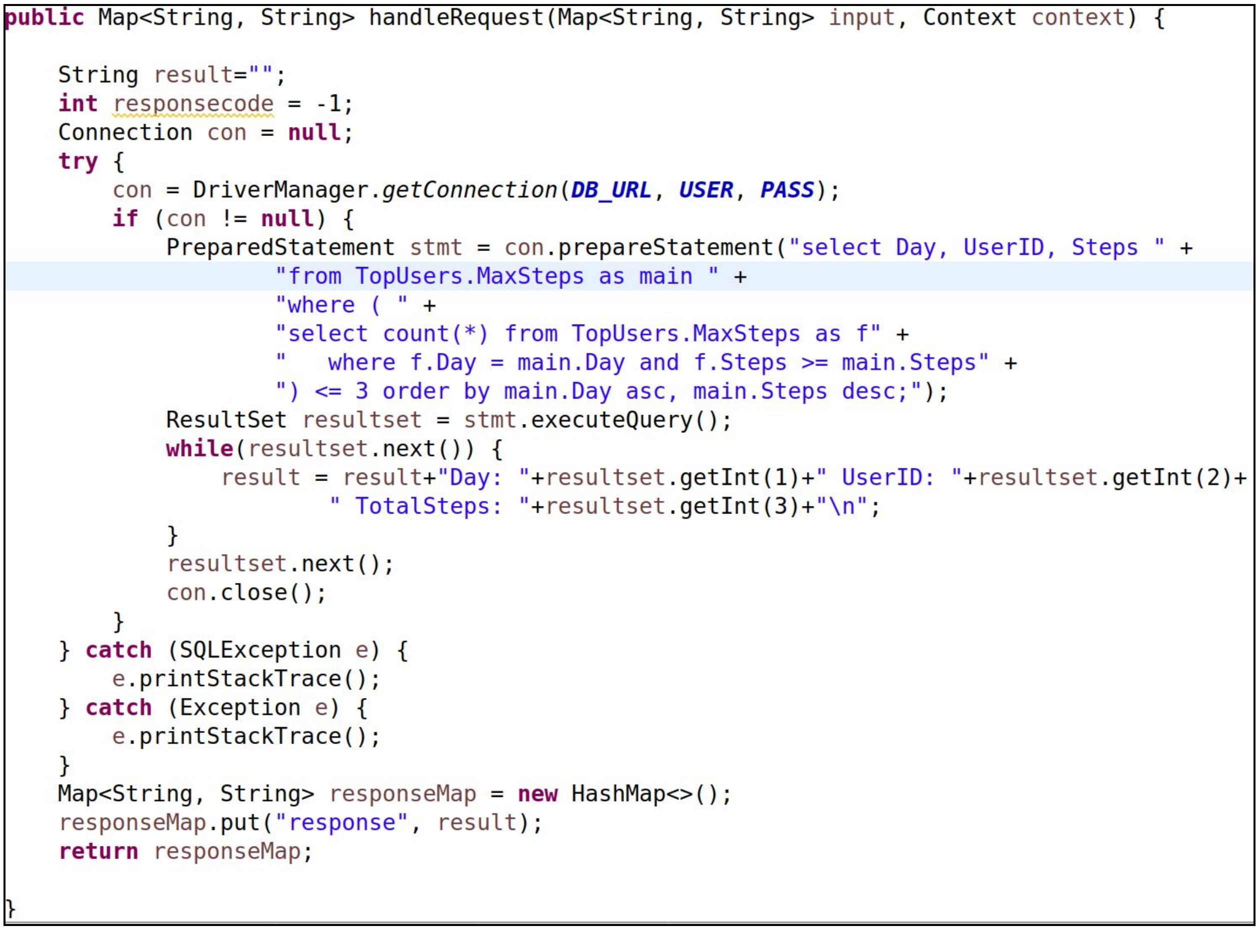
So this function calls the lambda function which handles the POST request. The lambda function handler class **‘PostHandler.java’** implements the RequestHandler interface. This actually has the SQL query to insert a row into the table and in case of duplicate keys, adds the new steps to it. The primary key for the table is a composite key made up of {UserID,Day,Hour}.

Now, let’s talk about the class **‘ClientToFindTop3.java’.** This class essentially does the job of report generation to find the top 3 users with maximum steps for the day. It makes use of ScheduledExecutorService to send a GET request every 4 seconds. This interval can be set by the user as the input argument to the function. Below is the screenshot of the periodic call made from the ClientToFindTop3.java class.



```
public static void main(String args[]) throws FileNotFoundException, IOException {
    int interval = new Integer(args[0]);
    ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
    service.scheduleAtFixedRate(new MyThreadClass(), 0, interval, TimeUnit.SECONDS);
    rq.run();
}
```

Upon getting a response from the server side this class also records the latency of each such request into a text file. This is used for measurement purpose of our hypothesis. The rest of the flow - call to LambdInvokeSteps and fetching data from the Database is similar to the POST request. The interesting bit about this GET request is the Lambda Handler class which send the SQL query to fetch the top 3 users for each row. Given below is the implementation of this query.

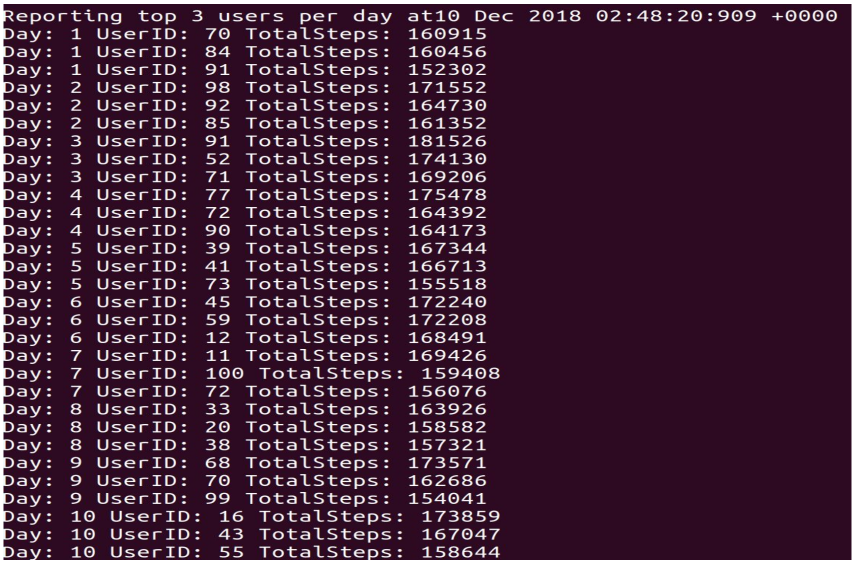


```
public Map<String, String> handleRequest(Map<String, String> input, Context context) {

    String result="";
    int responsecode = -1;
    Connection con = null;
    try {
        con = DriverManager.getConnection(DB_URL, USER, PASS);
        if (con != null) {
            PreparedStatement stmt = con.prepareStatement("select Day, UserID, Steps " +
                    "from TopUsers.MaxSteps as main " +
                    "where ( " +
                    "select count(*) from TopUsers.MaxSteps as f" +
                    "   where f.Day = main.Day and f.Steps >= main.Steps" +
                    ") <= 3 order by main.Day asc, main.Steps desc;");
            ResultSet resultset = stmt.executeQuery();
            while(resultset.next()) {
                result = result+"Day: "+resultset.getInt(1)+" UserID: "+resultset.getInt(2)+
                        " TotalSteps: "+resultset.getInt(3)+"\n";
            }
            resultset.next();
            con.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
    Map<String, String> responseMap = new HashMap<>();
    responseMap.put("response", result);
    return responseMap;

}
```

On getting a response back from the server the Top 3 for each day is logged onto console.



```
Reporting top 3 users per day at10 Dec 2018 02:48:20:909 +0000
Day: 1 UserID: 70 TotalSteps: 160915
Day: 1 UserID: 84 TotalSteps: 160456
Day: 1 UserID: 91 TotalSteps: 152302
Day: 2 UserID: 98 TotalSteps: 171552
Day: 2 UserID: 92 TotalSteps: 164730
Day: 2 UserID: 85 TotalSteps: 161352
Day: 3 UserID: 91 TotalSteps: 181526
Day: 3 UserID: 52 TotalSteps: 174130
Day: 3 UserID: 71 TotalSteps: 169206
Day: 4 UserID: 77 TotalSteps: 175478
Day: 4 UserID: 72 TotalSteps: 164392
Day: 4 UserID: 90 TotalSteps: 164173
Day: 5 UserID: 39 TotalSteps: 167344
Day: 5 UserID: 41 TotalSteps: 166713
Day: 5 UserID: 73 TotalSteps: 155518
Day: 6 UserID: 45 TotalSteps: 172240
Day: 6 UserID: 59 TotalSteps: 172208
Day: 6 UserID: 12 TotalSteps: 168491
Day: 7 UserID: 11 TotalSteps: 169426
Day: 7 UserID: 100 TotalSteps: 159408
Day: 7 UserID: 72 TotalSteps: 156076
Day: 8 UserID: 33 TotalSteps: 163926
Day: 8 UserID: 20 TotalSteps: 158582
Day: 8 UserID: 38 TotalSteps: 157321
Day: 9 UserID: 68 TotalSteps: 173571
Day: 9 UserID: 70 TotalSteps: 162686
Day: 9 UserID: 99 TotalSteps: 154041
Day: 10 UserID: 16 TotalSteps: 173859
Day: 10 UserID: 43 TotalSteps: 167047
Day: 10 UserID: 55 TotalSteps: 158644
```

AWS setup details:

For Apache Kafka testing:

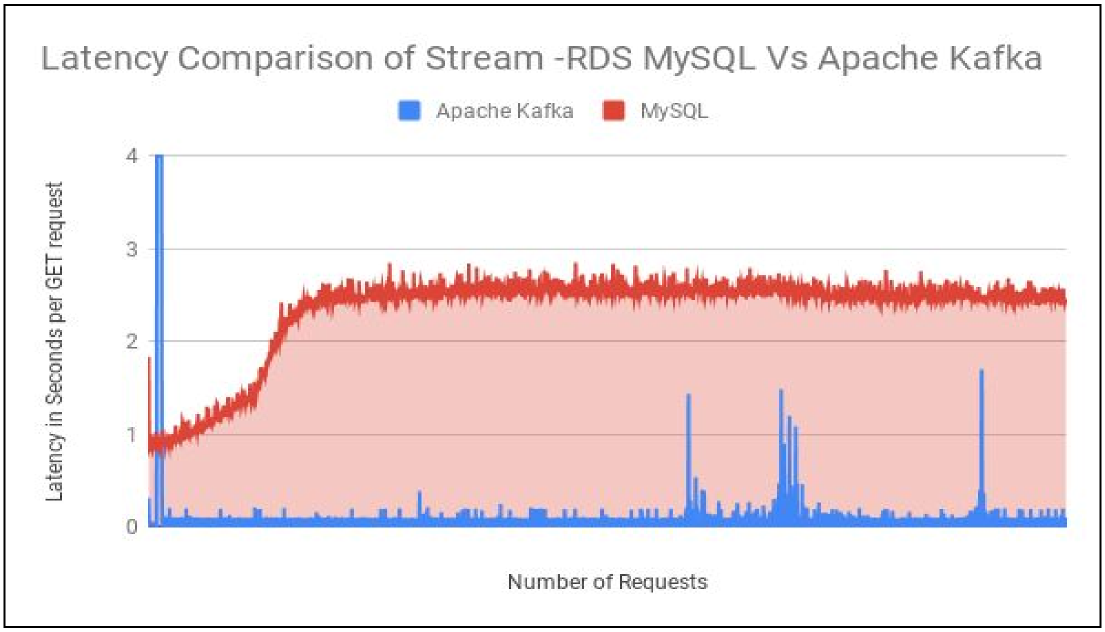
1. Installed Apache Kafka 2.11-2.1.0 on EC2 instance - made use of t3.xlarge AMI instance with 16GB RAM.
2. Setup DynamoDB for storing the data with primary key as ‘day’
3. Started Zookeeper Server and Kafka Server from the kafka_2.11-2.1.0/bin folder.
4. On 5 different terminals ran the following commands:
   a. java -jar ConsumerClass.jar Top3UsersWithMostStepsByDay group-top-3
   b. java -jar ReportGeneration.jar Top3UsersWithMostStepsByDay group-top-3 4
   c. java -jar App.jar
   d. java -jar AddStepsForUser.jar
   e. java -jar FitbitClient.jar 32 localhost 1 100 100

For MySQL testing:

1. Used the same t3.xlarge AMI instance as before.
2. Setup MySQL database for storing the user data.
3. Created the Lambda Function for the POST and GET requests
4. On two different terminals ran the following commands:
   a. java -jar FitbitPost.jar 10 1 100 10
   b. java -jar ClientTop3.jar 4

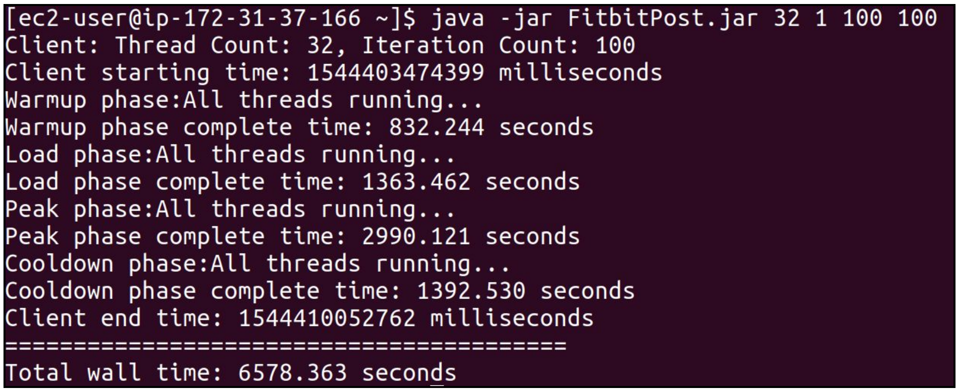
Results and Analysis:

Latency Plot of Apache Kafka Stream Vs MySQL : **32 threads with 100 iteration count**



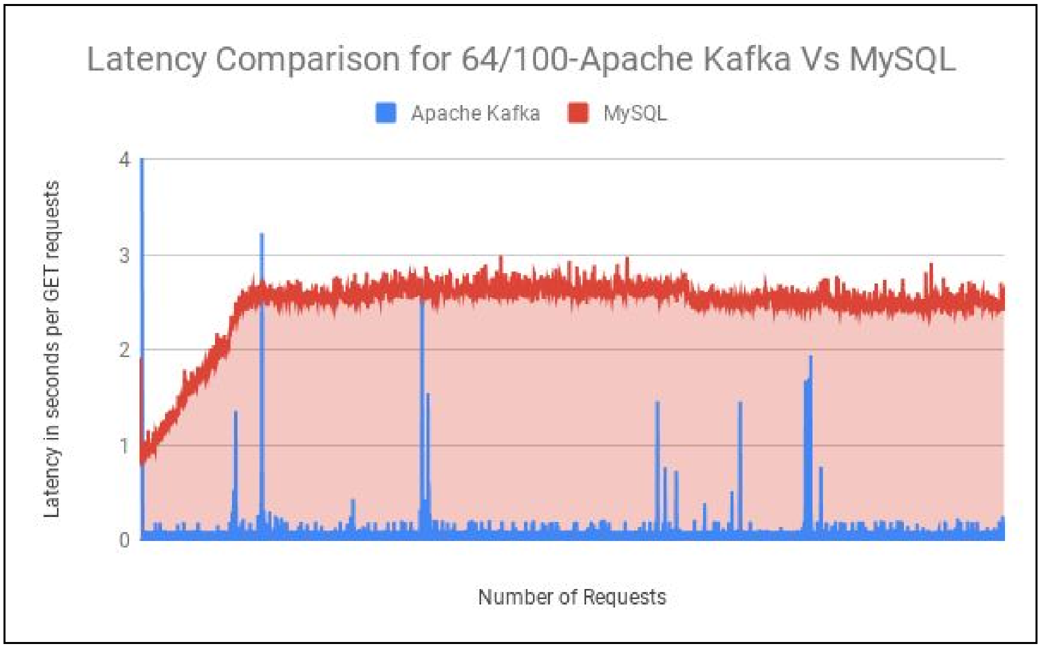
Above is the **Area Graph** of Latencies generated for each periodic call to Kafka stream or DB(in the case of SQL) to get the top 3. The GET requests are called periodically every 4 seconds. As can be interpreted each call to Kafka stream is within 0-1 seconds but in the case of MySQL it rises to 3 seconds or more.

Below is the POST record generation output for SQL application for the same. As you can see it is fairly slow.

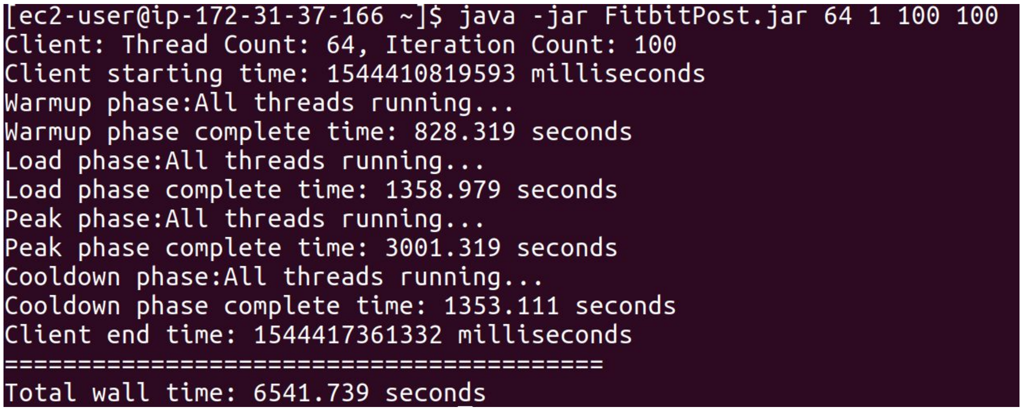


```
[ec2-user@ip-172-31-37-166 ~]$ java -jar FitbitPost.jar 32 1 100 100
Client: Thread Count: 32, Iteration Count: 100
Client starting time: 1544403474399 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 832.244 seconds
Load phase:All threads running...
Load phase complete time: 1363.462 seconds
Peak phase:All threads running...
Peak phase complete time: 2990.121 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 1392.530 seconds
Client end time: 1544410052762 milliseconds
=========================================
Total wall time: 6578.363 seconds
```

Latency Plot of Apache Kafka Stream Vs MySQL : **64 threads with 100 iteration count**

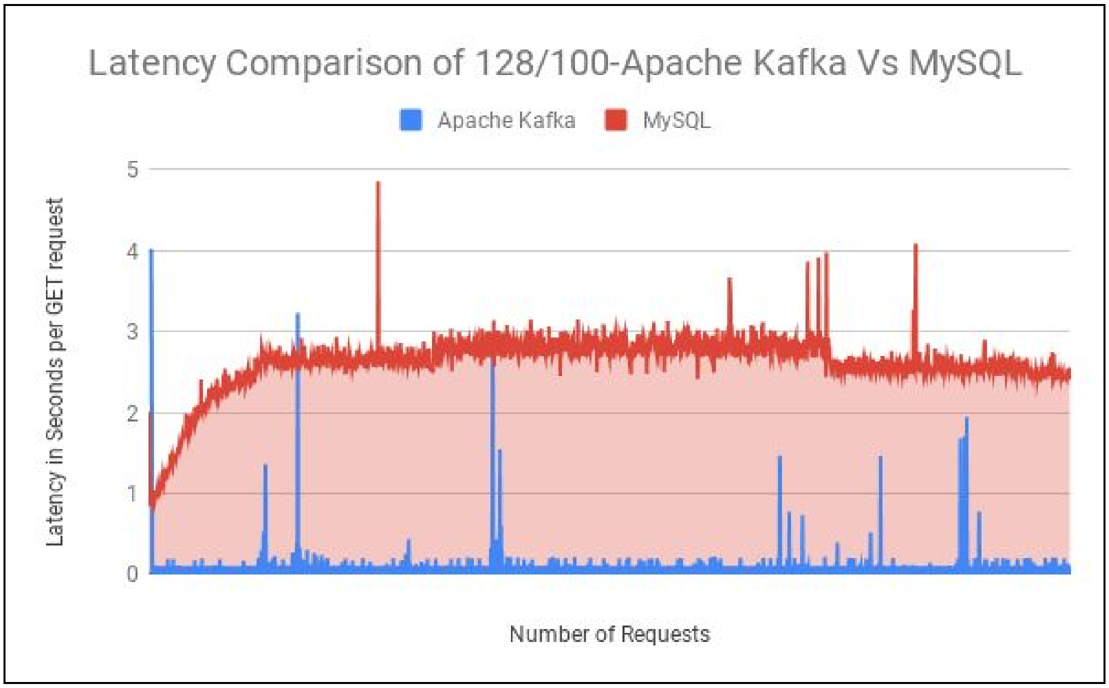


Below is the POST record generation output for SQL application for the same.

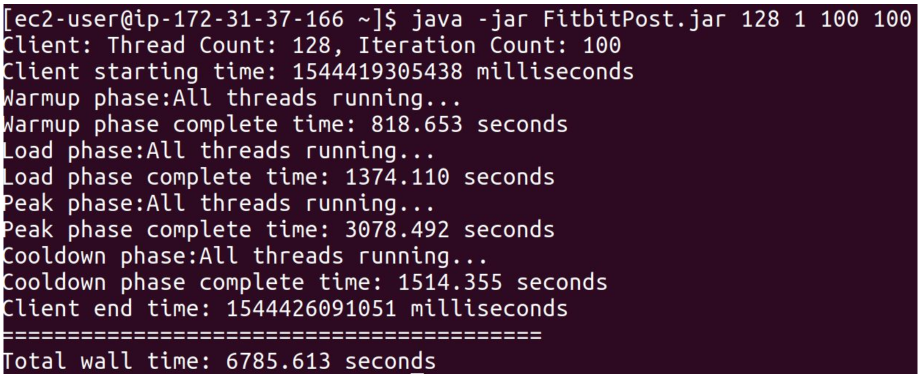


```
[ec2-user@ip-172-31-37-166 ~]$ java -jar FitbitPost.jar 64 1 100 100
Client: Thread Count: 64, Iteration Count: 100
Client starting time: 1544410819593 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 828.319 seconds
Load phase:All threads running...
Load phase complete time: 1358.979 seconds
Peak phase:All threads running...
Peak phase complete time: 3001.319 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 1353.111 seconds
Client end time: 1544417361332 milliseconds
=========================================
Total wall time: 6541.739 seconds
```

Latency Plot of Apache Kafka Stream Vs MySQL : **128 threads with 100 iteration count**

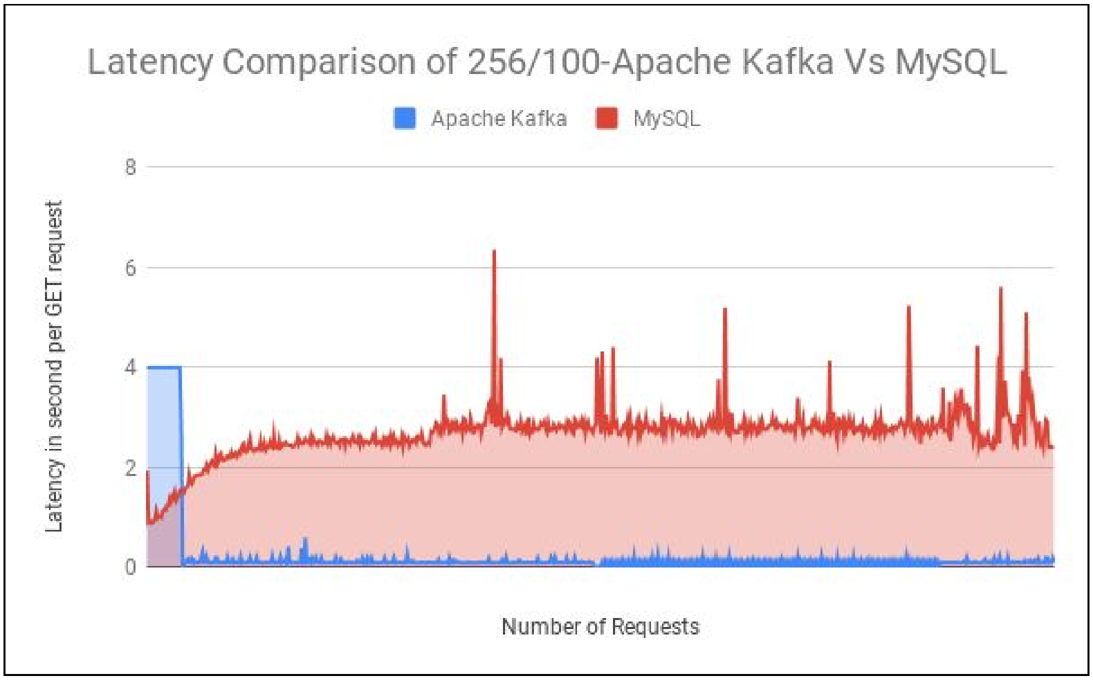


Below is the POST record generation output for SQL application for the same

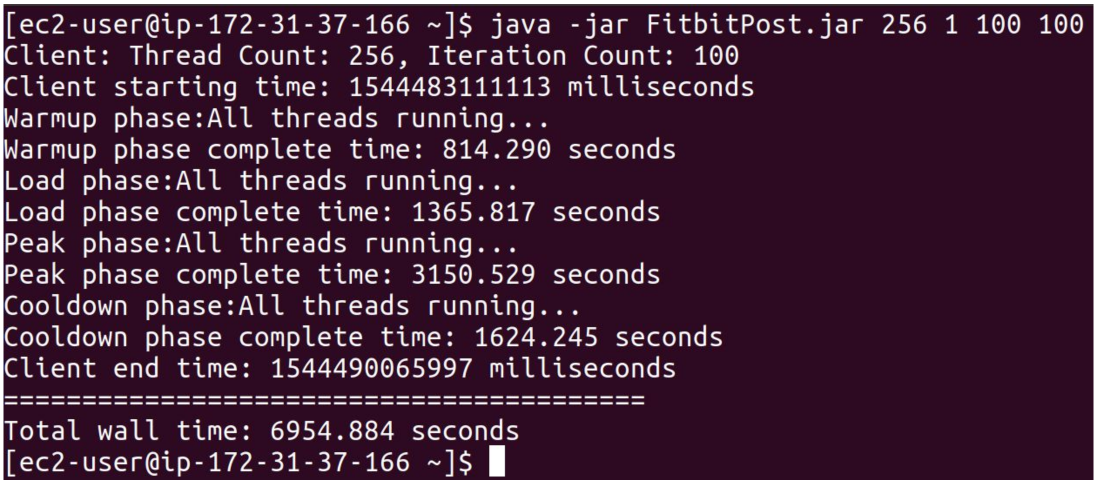


```
[ec2-user@ip-172-31-37-166 ~]$ java -jar FitbitPost.jar 128 1 100 100
Client: Thread Count: 128, Iteration Count: 100
Client starting time: 1544419305438 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 818.653 seconds
Load phase:All threads running...
Load phase complete time: 1374.110 seconds
Peak phase:All threads running...
Peak phase complete time: 3078.492 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 1514.355 seconds
Client end time: 1544426091051 milliseconds
=========================================
Total wall time: 6785.613 seconds
```

Latency Plot of Apache Kafka Stream Vs MySQL : **256 threads with 100 iteration count**



Below is the POST record generation output for SQL application for the same:



```
[ec2-user@ip-172-31-37-166 ~]$ java -jar FitbitPost.jar 256 1 100 100
Client: Thread Count: 256, Iteration Count: 100
Client starting time: 1544483111113 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 814.290 seconds
Load phase:All threads running...
Load phase complete time: 1365.817 seconds
Peak phase:All threads running...
Peak phase complete time: 3150.529 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 1624.245 seconds
Client end time: 1544490065997 milliseconds
=========================================
Total wall time: 6954.884 seconds
[ec2-user@ip-172-31-37-166 ~]$ █
```

Comparison of Total wall time in seconds for POST requests:

Configuration(Threads Vs Iteration)	Apache Kafka	MySQL
32/100	320.092	6578.363
64/100	657.099	6541.739
128/100	1456.279	6785.613
256/100	3170.516	6954.884

Conclusion:

From the above plots of GET requests latencies and POST requests wall time comparison we can conclude that our original hypothesis that “Apache Kafka Stream would be a better option for finding out the running top 3 users with maximum steps per day” is proved successfully.

External Links:

- https://technology.amis.nl/2017/02/12/apache-kafka-streams-running-top-n-grouped-by-dimension-from-and-to-kafka-topic/
- https://github.com/lucasjellema/kafka-streams-running-topN
- https://www.tutorialspoint.com/apache_kafka/apache_kafka_simple_producer_example.htm
- https://www.tutorialspoint.com/apache_kafka/apache_kafka_consumer_group_example.htm

Repository Contents:

1. Project Report PDF
2. Apache Kafka Application - Producer Application and Consumer Application .zip files
3. MySQL Lambda Application - Lambda Server and Lambda Client .zip files
4. Latency Files - has all the executed run’s latencies in seconds generated as .txt files