Team 10:

Topic: Implementation and analysis of continuous queries using MavStream

What we implemented?

We implemented 5 queries on a financial dataset to gain insights on the varying stock prices.

How?

We used the MavStream system to query Facebook's historical data spanning over 5 years. We have used different operators provided by the MavStream system such as aggregate(average, min, max) groupby and join.

Packages and libraries used: No additional packages or libraries were required since MavStream system already had the necessary files.

Steps of execution:

1. Import the dsms.server, dsms.core and dsms.client java projects into an IDE of your choice. The following steps are for Eclipse

- Unzip the downloaded MavVStream folder
- Go to File -> Import -> General -> Existing Projects into Workspace
- Check "Select root directory" radio button
- Browse to the location where you have the unzipped folder. For example, F:\MavVStream
- Select all the three projects i.e. dsms.server, dsms.core and dsms.client
- Click on finish

2. Import the launch file. The steps are as follows:

- Go to File -> Import -> Run/Debug -> Launch Configuration
- Browse to the unzipped folder. Example, F:\MavVStream
- On selecting the check box on the left, you will be able to view all the available launch files
- Select all of them and click on Finish

3. Running the Server:

- From the Package Explorer, go to dsms.server -> src -> edu.uta.dsms.server ->DSMSServer.java
- Go to Run -> Run Configurations
- Under the Java Application option, Select and run the DSMSServer launch configuration
- The server will begin listening on Port 8000

4. Go to dsms.client -> src -> edu.uta.dsms.client. Select any of the SimpleSampleClient_VideoQuery.java files and run it as a java application.

For running your queries over a dataset of your choice follow these steps:

1. Start with getting your dataset ready. For this you will need a text file which have single space separated values in it. Column names are not needed since you will be writing your field names for them. Dataset files should look something like this:

123.45 567.78 907.78 089.89 1 1

145.75 327.98 134.90 089.00 2 2

145.56 512.67 127.48 980.89 3 3

112.41 517.18 007.78 009.89 4 4

If all your data is just numbers, then try to keep the numbers in same format so that they will have same digits after and before the decimal. So, add 0s before a single digit number to make it 3 digits. Last two columns are Source TimeStamp and System TimeStamp. You will have to manually add these values. Just add 1,2,3,4…..up to the number of rows you have like the highlighted numbers in the above example. If you have strings in your dataset do not enclose them in quotes, just write them as it is. For e.g.:

134.44 Lincoln_Square 145.56 12/12/2012 1 1

2. Now add your own dataset into the system

For this open dsms.server -> Test -> txtfiles -> streamDefinitions.json file.

This file has all the previous datasets defined in it. Definition of a single dataset would look something like this:

{

“tableName”: “tableName as per your choice”,

“streamURI”: “../../Test/your_text_file_having_the_data.txt”,

```
"fields": [
        [
            "tableName.firstFieldName", (you can give your own field names here)
            "varchar",
            "0"
        ],
        [
            "tableName.secondFieldName",
            "number(double)",
            "1"
        ],
```

This way add all the fields which are there in the dataset. For floating or double precision numbers write number(double), for strings write varchar and for integers write number(long).

Keep incrementing the number for every field that you add.

The next two fields are mandatory to add in your dataset definition. These fieldNames never change. Only change the tableName part.

```
        [
            "tableName.tbSourceTS",
            "number(long)",
            "last numer + 1"
        ],
        [
            "tableName.tbSystemTS",
            "varchar",
            "0"
        ],
}
```

Here is how you can run the queries defined by me project team:

Query 1: Calculating the average stock price for every year in 5 years

1. Run the DSMSServer.java file
2. Run average.java

   The query is as follows:

```
CQ_Stream fbDataFinal = new CQ_Stream(2, "fbDataFinal", "fbDataFinal");

CQ_Aggregate average = new CQ_Aggregate(1, "avg", "AVERAGE", "fbDataFinal.close", "fbDataFinal.close");

CQ_Project  root     = new CQ_Project(0, "projectFields", "average(fbDataFinal.date)");

root.addInput(average);
average.addInput(fbDataFinal);


long startTime = 0;
long endTime   = 10000L;

CQ_ClientQuery cq = new CQ_ClientQuery(
        new CQ_ContinuousQuery(
            "testQuery",
            qos,
            root,
            startTime,
            endTime,

            Long.parseLong("253"),Long.parseLong("253"),Long.parseLong("253"),
            SchedulingStrategy.RoundRobinSS.toString(),
            "null",
            10));
```

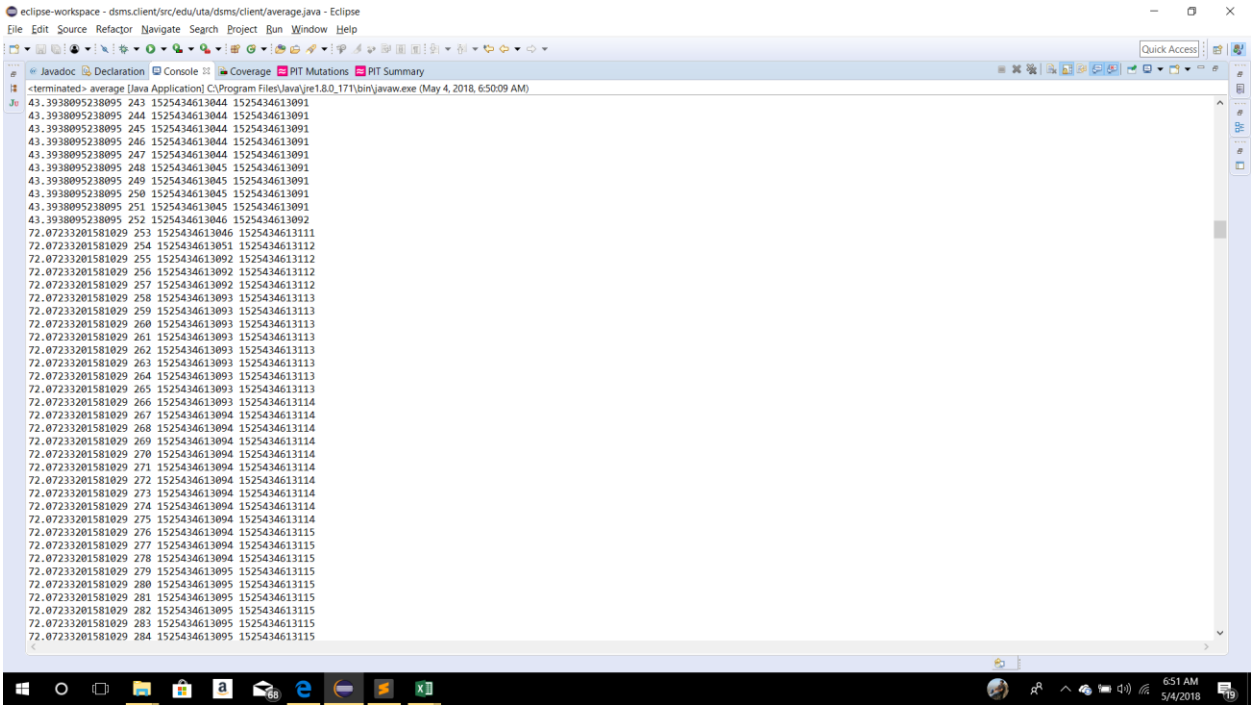
   Here in the first line “fbDataFinal” is the table name and the second argument is again same as the first one.
   Second line uses the CQ_Aggregate operator. You can see all the operators in dsms.core -> src -> edu.uta.dsms.core.query -> CQ_..... .java
   fbDataFinal.close and fbDataFinal.date are the fields which are there in our dataset.

   Keep everything else the same. The line which says Long.parseLong(“253”) is the line where you will be able to give window based parameters.
   First parameter is hopSizeLB, second is hopSizeUB and the last one is initialWindowSize.

   Once you run the file(assuming you have started the DSMSServer) it will do a bunch of things, so wait till the console says monitor sleepy…….then stop the execution manually and press display selected console which will show you the output.
   Our dataset is Facebook’s historical stock data downloaded from https://finance.yahoo.com/quote/FB/history?p=FB

   Since the stock market is only open 5 days a week, the dataset has approximately 253 rows for a single year. That’s why the window and hop size is 253. So, the initial window

size will be completed once the system encounters 253 rows. It will calculate the average and again wait for the next 253 rows. The output is a bit messy and the system keeps on repeating the output for 253 rows since it's a 253 - row window. So, the output for the first query was like this:



Output explained:
The first column is the average stock price, second column is just the timestamp value which we add manually in the dataset. Third and fourth column are system generated so ignore those.

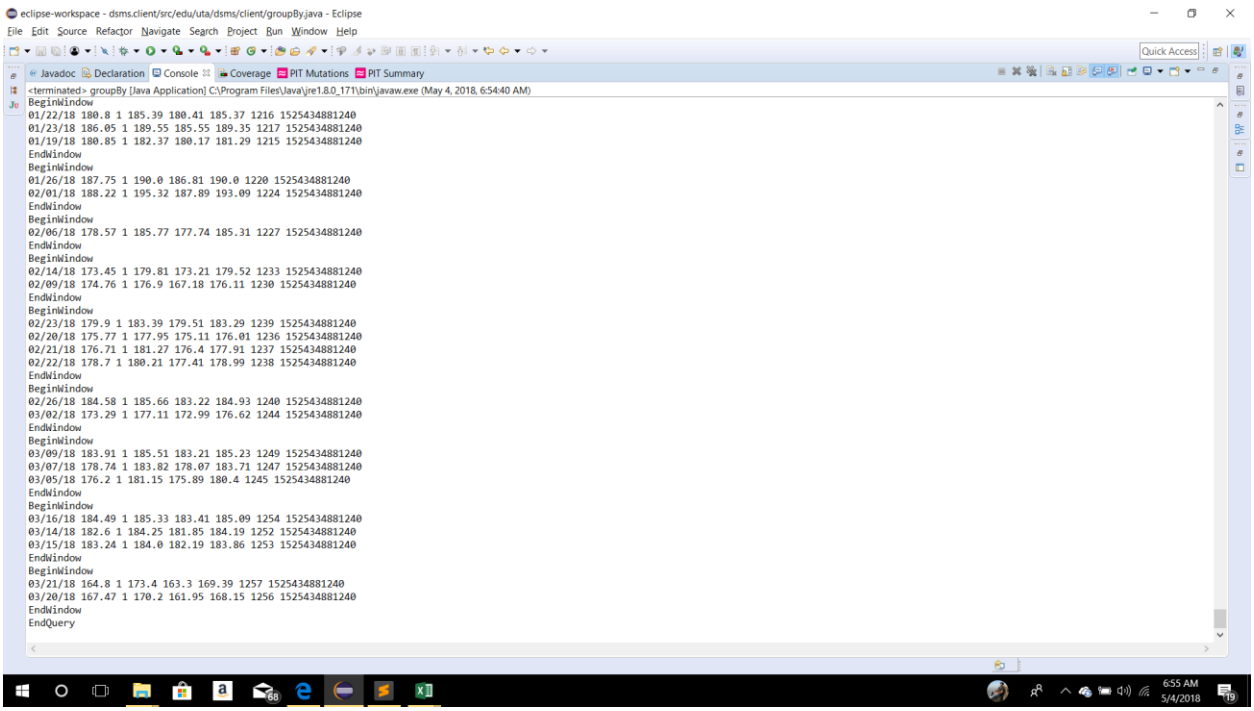
Query 2: Grouping the days in a week where the opening price was less than the closing price in 5 years.

1. Run the DSMSServer.java file
2. Run groupBy.java

The query is as follows:

```
CQ_Stream fbData = new CQ_Stream(3, "fbDataFinal", "fbDataFinal");

CQ_Select   select1  = new CQ_Select(2, "selectTuples","fbDataFinal.open < fbDataFinal.close ");

CQ_GroupBy g = new CQ_GroupBy(1, "grp", "fbDataFinal.open", "count","fbDataFinal.open", "fbDataFinal.open" );

g.addInput(select1);
select1.addInput(fbData);


long startTime = 0;
long endTime   = 10000L;

CQ_ClientQuery cq = new CQ_ClientQuery(
        new CQ_ContinuousQuery(
            "testQuery",
            qos,
            g,
            startTime,
            endTime,
            //both hop and window are 5 so that all the days satisfying the condition in a week will be generated
            Long.parseLong("5"),Long.parseLong("5"),Long.parseLong("5"),
            SchedulingStrategy.RoundRobinSS.toString(),
            "null",
            10));
```

Here the window parameters are 5 since its grouping for a week. Since stock market is only open 5 days a week there were 5 rows in the dataset for a week. Output for this query was like this:



Here it groups the rows which satisfy the condition in the select query, in a 5-row window. The first column is the fbDataFinal.date field, second is the fbDataFinal.open,

third column is system generated, fourth is fbDataFinal.high, fifth is fbDataFinal.low, sixth is fbDataFinal.close and seventh is the manually added timestamp value of that row. Last two columns are again system generated so ignore them.

Query 3: Combining two different streams of stock information i.e.

1. Stream -1: Date and Opening Price
2. Stream -2: Date and Volume

To run the query:

1. Run the DSMSServer.java file
2. Run join.java

The query is as follows:

```
CQ_Join     join1 = new CQ_Join(1,
                                "joinStreams",
                                stream1,
                                stream2,
                                "dateOpen.date == dateVol.date");


CQ_Select   select1  = new CQ_Select(2, "selectTuples","dateOpen.open > '0.0' ");

CQ_Project  root     = new CQ_Project(0, "projectFields",
         "dateOpen.date ,"+
         "dateOpen.open ," +
         "dateVol.volume ,");
```

Here to test the join operator we manually divided our dataset to make two different datasets. These two datasets are small datasets and their filenames are fboneyear.txt and fbdataforjoin.txt

These datasets look like these(Not the exact values) :
First file:
03/26/2013 25.11 1 2
03/27/2013 26.78 1 2
03/28/2013 25.53 1 2
04/05/2013 26.85 1 2
Second file:
03/26/2013 21231241 1 2
03/27/2013 26781231 1 2
03/28/2013 25531231 1 2
04/05/2013 26854141 1 2

Output for this query was like this:

```
BeginWindow
3/26/2013 25.21 3/26/2013 26957200 1 1 1524716928392
4/1/2013 25.53 4/1/2013 22249300 1 1 1524716928413
3/28/2013 25.58 3/28/2013 28585700 1 1 1524716928413
3/27/2013 26.09 3/27/2013 52297400 1 1 1524716928413
4/5/2013 27.39 4/5/2013 64566600 1 1 1524716928414
4/4/2013 27.07 4/4/2013 82016800 1 1 1524716928414
4/3/2013 26.25 4/3/2013 48195200 1 1 1524716928414
4/2/2013 25.42 4/2/2013 35153300 1 1 1524716928414
EndWindow
```

Here two streams are joined where the date is the same. First column is date from first stream and second column is opening price from the first stream. Third and fourth columns are date and volume from the second stream. Fifth, sixth and seventh columns are system generated.

Query 4: Calculating the 50 – day and 200-day moving average of Facebook's closing price over the last 5 years

To run the query:

1. Run the DSMSServer.java file
2. Run movingAverage.java

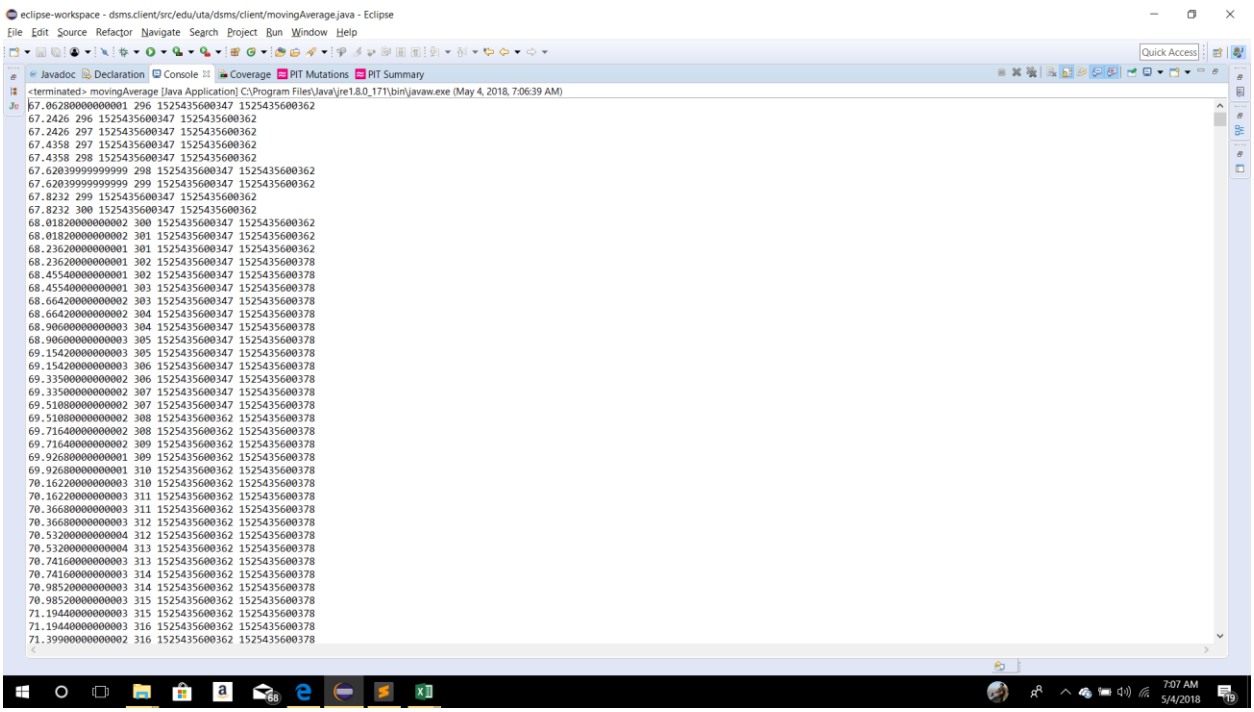
The query is as follows:

```
CQ_Stream fbDataFinal = new CQ_Stream(2, "fbDataFinal", "fbDataFinal");

CQ_Aggregate average = new CQ_Aggregate(1, "avg", "AVERAGE", "fbDataFinal.close", "fbDataFinal.close");

CQ_Project  root     = new CQ_Project(0, "projectFields", "average(fbDataFinal.date)");

root.addInput(average);
average.addInput(fbDataFinal);

long startTime = 0;
long endTime   = 10000L;

CQ_ClientQuery cq = new CQ_ClientQuery(
        new CQ_ContinuousQuery(
            "testQuery",
            qos,
            root,
            startTime,
            endTime,
            //For 50 day -> hopLB will be 1, hopUB will be 1 & initial window will be 50
            //For 200 day -> hop will be 1, hopUB will be 1 & initial window will be 200
            Long.parseLong("1"),Long.parseLong("1"),Long.parseLong("50"),
            SchedulingStrategy.RoundRobinSS.toString(),
            "null",
            10));
```

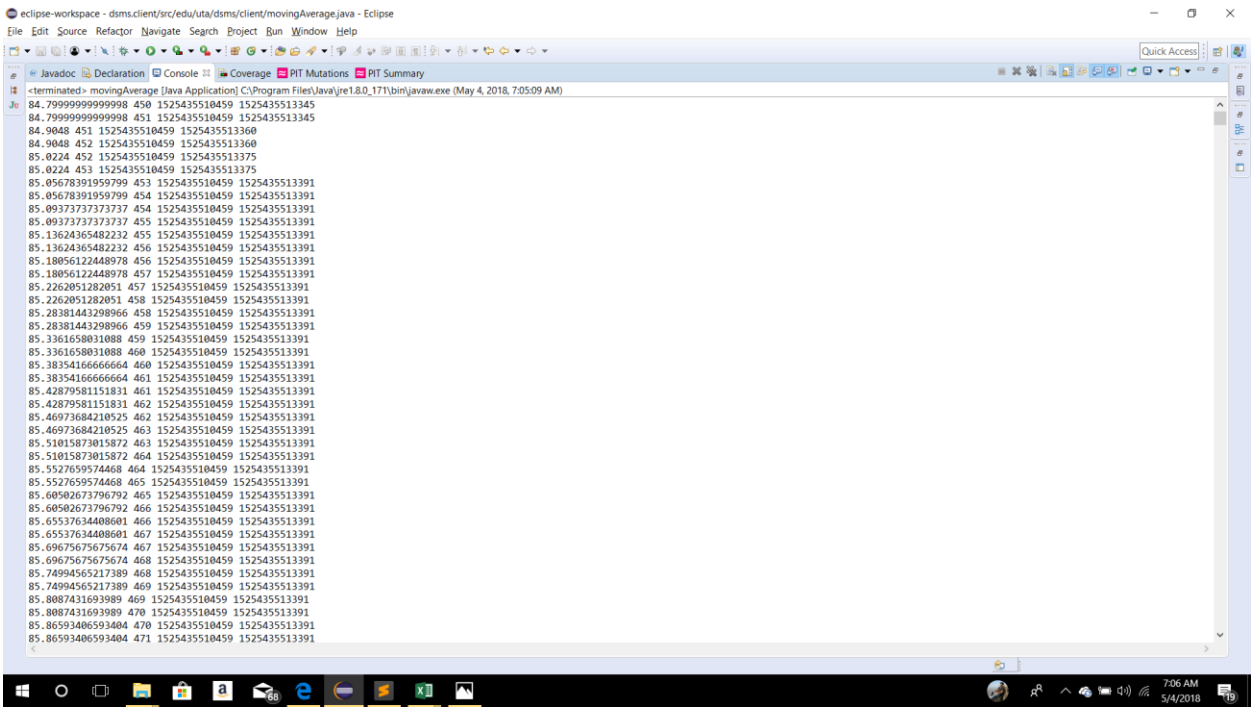
This screenshot if for the 50-day moving average query. For moving average hopsize parameters are 1 and window is 50. Moving averages are used in technical analysis of stock market. For 200-day only the window parameter will change.

Output of this query was as follows:

For 50-day:



For 200-day:



The first column is the moving average which keeps on changing as soon as a new row is encountered. Second column is manually added timestamp in the dataset. Last two columns are system generated.

Query 5: Finding the maximum and minimum closing value of the stock in a window of 30 days (month)

To run the query:

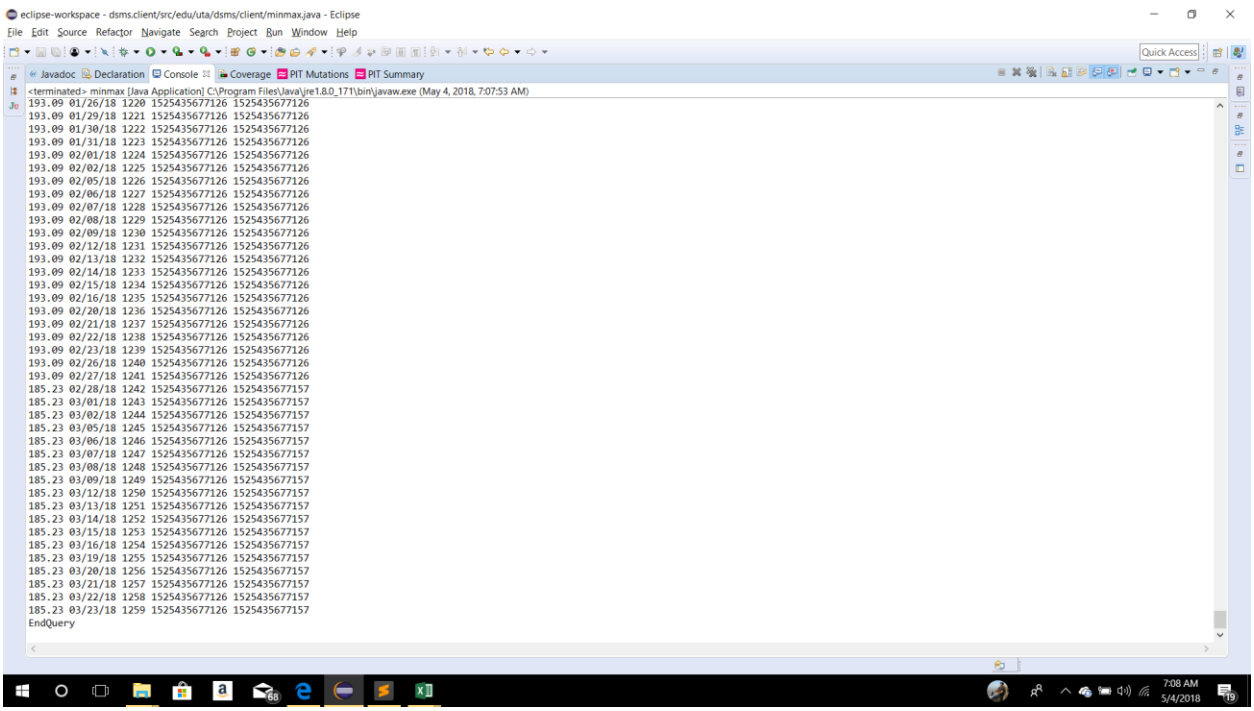
1. Run the DSMSServer.java file
2. Run minmax.java

Query is as follows:

```
CQ_Stream fbDataFinal = new CQ_Stream(2, "fbDataFinal", "fbDataFinal");

//CQ_Select   select1  = new CQ_Select(2, "selectTuples","fbDataFinal.close > '167.60' ");

CQ_Aggregate min = new CQ_Aggregate(1, "max", "MAX", "fbDataFinal.close", "fbDataFinal.close");

CQ_Project  root     = new CQ_Project(0, "projectFields", "max(fbDataFinal.date), fbDataFinal.date");


root.addInput(min);
min.addInput(fbDataFinal);
//select1.addInput(fbData);

long startTime = 0;
long endTime   = 10000L;

CQ_ClientQuery cq = new CQ_ClientQuery(
        new CQ_ContinuousQuery(
            "testQuery",
            qos,
            root,
            startTime,
            endTime,
            // since stock market is approximately open for 23 days in a month
            // both hop and window is 23 so that min or max value of the closing price in a
            //month will be generated
            Long.parseLong("23"),Long.parseLong("23"),Long.parseLong("23"),
            SchedulingStrategy.RoundRobinSS.toString(),
            "null",
            10));
```

Here window and hopSize parameters are 23 since a month (5 days a week for stock market, so weekends are not counted) in our dataset was approximately 23 rows.

Output of this query was as follows:



First column keeps repeating 23 times but it is the minimum or the maximum value depending on the rows which are included in that particular window. Second column is the fbDataFinal.date field. Third column is manually added timestamp and last two columns are system generated.

Now, if you need to run your own queries then typically you will have to change the following things:

1. Add your own tableName in CQ_Stream query.
2. Write the select, project or whatever queries you need in the proper sequence since stream processing needs a query plan and a tree is generated by .addInput() method.
3. Change the window and hop parameters as per your requirements.