

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.

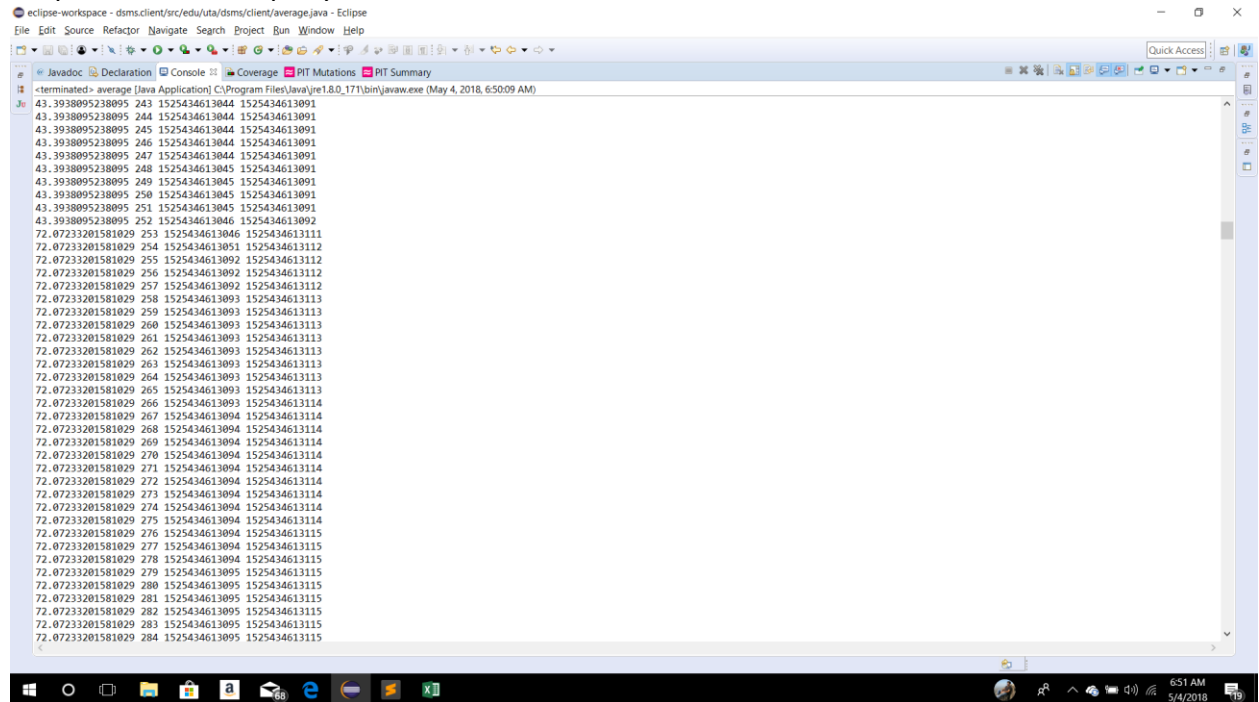
First parameter is hopSizeLB, second is hopSizeUB and the last one is initialWindowSize.

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:



```
terminated> average [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 4, 2018, 6:50:09 AM)
43.3938095238095 243 1525434613044 1525434613091
43.3938095238095 244 1525434613044 1525434613091
43.3938095238095 245 1525434613044 1525434613091
43.3938095238095 246 1525434613044 1525434613091
43.3938095238095 247 1525434613044 1525434613091
43.3938095238095 248 1525434613045 1525434613091
43.3938095238095 249 1525434613045 1525434613091
43.3938095238095 250 1525434613045 1525434613091
43.3938095238095 251 1525434613045 1525434613091
43.3938095238095 252 1525434613046 1525434613092
72.07233201581029 253 1525434613046 1525434613111
72.07233201581029 254 1525434613051 1525434613112
72.07233201581029 255 1525434613092 1525434613112
72.07233201581029 256 1525434613092 1525434613112
72.07233201581029 257 1525434613092 1525434613112
72.07233201581029 258 1525434613093 1525434613113
72.07233201581029 259 1525434613093 1525434613113
72.07233201581029 260 1525434613093 1525434613113
72.07233201581029 261 1525434613093 1525434613113
72.07233201581029 262 1525434613093 1525434613113
72.07233201581029 263 1525434613093 1525434613113
72.07233201581029 264 1525434613093 1525434613113
72.07233201581029 265 1525434613093 1525434613113
72.07233201581029 266 1525434613093 1525434613114
72.07233201581029 267 1525434613094 1525434613114
72.07233201581029 268 1525434613094 1525434613114
72.07233201581029 269 1525434613094 1525434613114
72.07233201581029 270 1525434613094 1525434613114
72.07233201581029 271 1525434613094 1525434613114
72.07233201581029 272 1525434613094 1525434613114
72.07233201581029 273 1525434613094 1525434613114
72.07233201581029 274 1525434613094 1525434613114
72.07233201581029 275 1525434613094 1525434613114
72.07233201581029 276 1525434613094 1525434613115
72.07233201581029 277 1525434613094 1525434613115
72.07233201581029 278 1525434613094 1525434613115
72.07233201581029 279 1525434613095 1525434613115
72.07233201581029 280 1525434613095 1525434613115
72.07233201581029 281 1525434613095 1525434613115
72.07233201581029 282 1525434613095 1525434613115
72.07233201581029 283 1525434613095 1525434613115
72.07233201581029 284 1525434613095 1525434613115
```

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:

```

<terminated> groupBy [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 4, 2018, 6:54:40 AM)
BeginWindow
01/22/18 180.8 1 185.39 180.41 185.37 1216 1525434881240
01/23/18 186.05 1 189.55 185.55 189.35 1217 1525434881240
01/19/18 180.85 1 182.37 180.17 181.29 1215 1525434881240
EndWindow
BeginWindow
01/26/18 187.75 1 190.0 186.81 190.0 1220 1525434881240
02/01/18 188.22 1 195.32 187.89 193.09 1224 1525434881240
EndWindow
BeginWindow
02/06/18 178.57 1 185.77 177.74 185.31 1227 1525434881240
EndWindow
BeginWindow
02/14/18 171.45 1 179.81 173.21 179.52 1233 1525434881240
02/09/18 174.76 1 176.9 167.18 176.11 1230 1525434881240
EndWindow
BeginWindow
02/23/18 179.9 1 183.39 179.51 183.29 1239 1525434881240
02/20/18 175.77 1 177.95 175.11 176.01 1236 1525434881240
02/21/18 176.71 1 181.27 176.4 177.91 1237 1525434881240
02/22/18 178.7 1 180.21 177.41 178.99 1238 1525434881240
EndWindow
BeginWindow
02/26/18 184.58 1 185.66 183.22 184.93 1240 1525434881240
03/02/18 173.29 1 177.11 172.99 176.62 1244 1525434881240
EndWindow
BeginWindow
03/09/18 183.91 1 185.51 183.21 185.23 1249 1525434881240
03/07/18 178.74 1 183.82 178.07 183.71 1247 1525434881240
03/05/18 176.2 1 181.15 175.89 180.4 1245 1525434881240
EndWindow
BeginWindow
03/16/18 184.49 1 185.33 183.41 185.09 1254 1525434881240
03/14/18 182.6 1 184.25 181.85 184.19 1252 1525434881240
03/15/18 183.24 1 184.0 182.19 183.86 1253 1525434881240
EndWindow
BeginWindow
03/21/18 164.8 1 173.4 163.3 169.39 1257 1525434881240
03/20/18 167.47 1 170.2 161.95 168.15 1256 1525434881240
EndWindow
EndQuery

```

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));

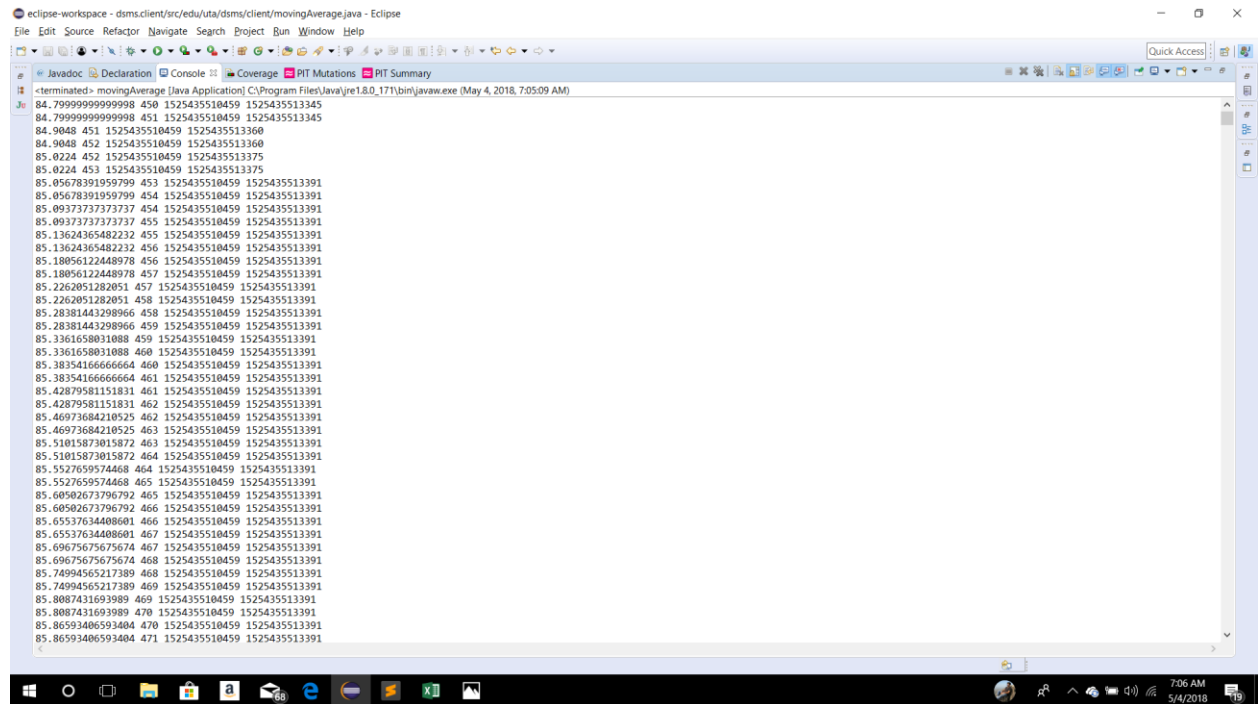
```

This screenshot is 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:



```
<terminated> movingAverage [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 4, 2018, 7:05:09 AM)
84.79999999999998 450 1525435510459 1525435513345
84.79999999999998 451 1525435510459 1525435513345
84.9048 451 1525435510459 1525435513360
84.9048 452 1525435510459 1525435513360
85.0224 452 1525435510459 1525435513375
85.0224 453 1525435510459 1525435513375
85.05678391959799 453 1525435510459 1525435513391
85.05678391959799 454 1525435510459 1525435513391
85.09373737373737 454 1525435510459 1525435513391
85.09373737373737 455 1525435510459 1525435513391
85.13624365482232 455 1525435510459 1525435513391
85.13624365482232 456 1525435510459 1525435513391
85.18056122448978 456 1525435510459 1525435513391
85.18056122448978 457 1525435510459 1525435513391
85.2262051282051 457 1525435510459 1525435513391
85.2262051282051 458 1525435510459 1525435513391
85.28381443298966 458 1525435510459 1525435513391
85.28381443298966 459 1525435510459 1525435513391
85.3361658031088 459 1525435510459 1525435513391
85.3361658031088 460 1525435510459 1525435513391
85.38354166666664 460 1525435510459 1525435513391
85.38354166666664 461 1525435510459 1525435513391
85.42879581151831 461 1525435510459 1525435513391
85.42879581151831 462 1525435510459 1525435513391
85.46973684210525 462 1525435510459 1525435513391
85.46973684210525 463 1525435510459 1525435513391
85.51015873015872 463 1525435510459 1525435513391
85.51015873015872 464 1525435510459 1525435513391
85.5527659574468 464 1525435510459 1525435513391
85.5527659574468 465 1525435510459 1525435513391
85.60502673796792 465 1525435510459 1525435513391
85.60502673796792 466 1525435510459 1525435513391
85.65537634408001 466 1525435510459 1525435513391
85.65537634408001 467 1525435510459 1525435513391
85.69675675675674 467 1525435510459 1525435513391
85.69675675675674 468 1525435510459 1525435513391
85.74994565217389 468 1525435510459 1525435513391
85.74994565217389 469 1525435510459 1525435513391
85.8087431693989 469 1525435510459 1525435513391
85.8087431693989 470 1525435510459 1525435513391
85.86593406593404 470 1525435510459 1525435513391
85.86593406593404 471 1525435510459 1525435513391
```

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:

```
<terminated> minmax [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 4, 2018, 7:07:53 AM)
193.09 01/26/18 1220 1525435677126 1525435677126
193.09 01/29/18 1221 1525435677126 1525435677126
193.09 01/30/18 1222 1525435677126 1525435677126
193.09 01/31/18 1223 1525435677126 1525435677126
193.09 02/01/18 1224 1525435677126 1525435677126
193.09 02/02/18 1225 1525435677126 1525435677126
193.09 02/05/18 1226 1525435677126 1525435677126
193.09 02/06/18 1227 1525435677126 1525435677126
193.09 02/07/18 1228 1525435677126 1525435677126
193.09 02/08/18 1229 1525435677126 1525435677126
193.09 02/09/18 1230 1525435677126 1525435677126
193.09 02/12/18 1231 1525435677126 1525435677126
193.09 02/13/18 1232 1525435677126 1525435677126
193.09 02/14/18 1233 1525435677126 1525435677126
193.09 02/15/18 1234 1525435677126 1525435677126
193.09 02/16/18 1235 1525435677126 1525435677126
193.09 02/20/18 1236 1525435677126 1525435677126
193.09 02/21/18 1237 1525435677126 1525435677126
193.09 02/22/18 1238 1525435677126 1525435677126
193.09 02/23/18 1239 1525435677126 1525435677126
193.09 02/26/18 1240 1525435677126 1525435677126
193.09 02/27/18 1241 1525435677126 1525435677126
185.23 02/28/18 1242 1525435677126 1525435677157
185.23 03/01/18 1243 1525435677126 1525435677157
185.23 03/02/18 1244 1525435677126 1525435677157
185.23 03/05/18 1245 1525435677126 1525435677157
185.23 03/06/18 1246 1525435677126 1525435677157
185.23 03/07/18 1247 1525435677126 1525435677157
185.23 03/08/18 1248 1525435677126 1525435677157
185.23 03/09/18 1249 1525435677126 1525435677157
185.23 03/12/18 1250 1525435677126 1525435677157
185.23 03/13/18 1251 1525435677126 1525435677157
185.23 03/14/18 1252 1525435677126 1525435677157
185.23 03/15/18 1253 1525435677126 1525435677157
185.23 03/16/18 1254 1525435677126 1525435677157
185.23 03/19/18 1255 1525435677126 1525435677157
185.23 03/20/18 1256 1525435677126 1525435677157
185.23 03/21/18 1257 1525435677126 1525435677157
185.23 03/22/18 1258 1525435677126 1525435677157
185.23 03/23/18 1259 1525435677126 1525435677157
EndQuery
```

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.

