Streams Processing - Project 2
Taxi Trips from New York City

André Bastos, Nº 56969
Carolina Goldstein, Nº 57213
Rafaela Cruz, Nº 56926

Introduction

In this project, we processed a dataset from ACM DEBS 2015 Grand Challenge, which contains information about taxi trips in New York City. Each record in the dataset corresponds to one taxi ride and contains multiple variables, such as the taxi identifier, the date and time of the pickup and the dropoff of the ride, the duration and distance of the ride, the latitude and longitude of the pickup location and dropoff location, as well as information related to payment. Thus, using this information, we tried to answer several questions that might be useful to the taxi drivers, namely when choosing the areas where they should wait to pickup clients.

To do that, we used the WSO2 Streaming Integrator Tooling, writing our queries in SiddhiQL. We started by creating streams with the processed dataset, in order to exclude locations outside the stipulated bounds and null and extreme values that might affect computations, and to convert longitude and latitude values to grid cells. Afterwards, we used the processed streams to answer the questions. All the computations performed throughout the project are explained in the next sections.

Processed Streams

We started by defining our source, which allows Siddhi to consume events from external systems, and map the events to adhere to the associated stream. In our case, the source type that was used was Kafka and the mapping type was CSV. We defined our source stream using the same column names as in the dataset.

```
@source(type="kafka", topic.list="debs", partition.no.list="0", threading.option="single.thread", ↩
    group.id="group", bootstrap.servers="kafka:9092", @map(type="csv"))
define stream TaxiRidesProductionStream (medallion string, hack_license string, pickup_datetime ↩
    string, dropoff_datetime string, trip_time_in_secs double, trip_distance double, ↩
    pickup_longitude double, pickup_latitude double, dropoff_longitude double, dropoff_latitude ↩
    double, payment_type string, fare_amount double, surcharge double, mta_tax double, tip_amount ↩
    double, tolls_amount double, total_amount double);
```

Afterwards, in order to facilitate the answering of the questions, we processed the source stream. We started by adding two new columns with the timestamp of the pickup datetime and the dropoff datetime in milliseconds. These timestamps will be useful in some of the queries, namely in query 3, as we will explain in its respective section. We also filtered the rows that had extreme or null values for the trip distance, the trip time in seconds and the total paid amount, and the rows with a longitude or latitude outside the specified bounds. Note that the coordinates provided, (41.474937, -74.913585), mark the center of the first cell. Thus, to define the grid, we added 250 meters north in the latitude (obtaining 41.47718278) and 250 meters west in the longitude (obtaining -74.916578), in order to mark the upper left corner of the grid. Then, we computed the bounds as shown Figure 1.

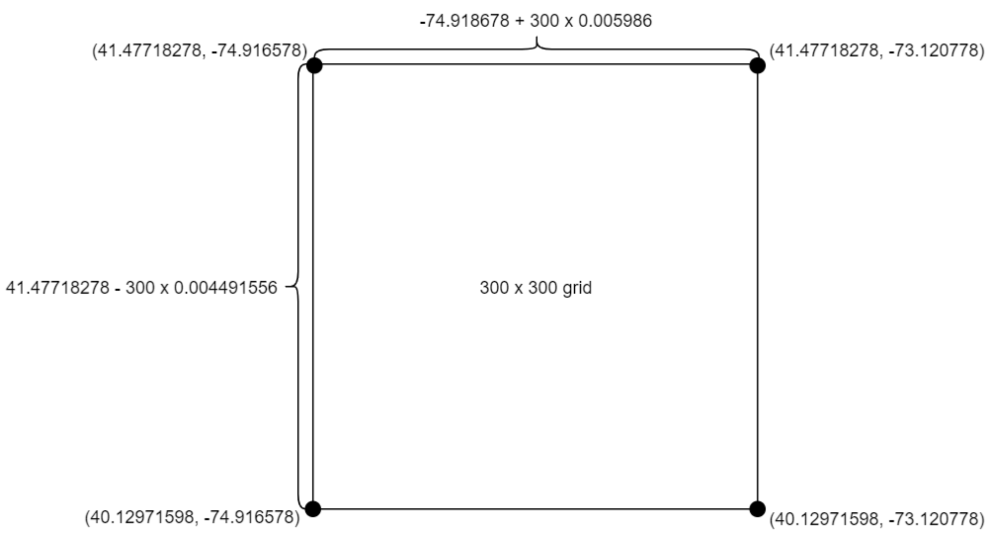


Figure 1: Calculations performed to determine the bounds of the grid.

Finally, we inserted this processed stream into a sink called `ProcessedStream`.

```
from TaxiRidesProductionStream
select medallion, hack_license, pickup_datetime, dropoff_datetime, time:timestampInMilliseconds(↩
    pickup_datetime,"yyyy-MM-dd HH:mm:ss") as pickup_datetime_millisec, time:↩
    timestampInMilliseconds(dropoff_datetime,"yyyy-MM-dd HH:mm:ss") as dropoff_datetime_millisec, ↩
    trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, ↩
    dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, ↩
    total_amount
having pickup_longitude >= -74.916578 and pickup_longitude <= -73.120778 and dropoff_longitude >= ↩
    -74.916578 and dropoff_longitude <= -73.120778 and pickup_latitude >=40.12971598 and ↩
    pickup_latitude <= 41.47718278 and dropoff_latitude >= 40.12971598 and dropoff_latitude <= ↩
    41.47718278 and trip_distance <= 25 and trip_distance > 0 and trip_time_in_secs <= 3600 and ↩
    trip_time_in_secs > 0 and total_amount <= 80 and total_amount > 0
insert into ProcessedStream;
```

Using this `ProcessedStream`, we converted the latitude and longitude of the pickup and dropoff locations to grid cells, each cell being a square of 500 m by 500 m. To do that, we computed the difference between the current latitude and the latitude of the upper left corner of the grid and divided this difference by 0.004491556, which corresponds to a shift of 500 meter south. We did the same for the longitude (dividing the difference by 0.005986, which corresponds to a shift of 500 meter east), and concatenated the result using a "-" as separator. We did this both for the pickup location and the dropoff location, obtaining a stream with all the previous information plus two new columns: the `pickup_cell` and the `dropoff_cell`.

```
from ProcessedStream
select medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, ↩
    dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude,↩
     dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount↩
    , tolls_amount, total_amount, str:concat(convert(convert(math:ceil((pickup_latitude - ↩
    41.47718278)/-0.004491556),"int"),"string"), "-", convert(convert(math:ceil((pickup_longitude +↩
     74.916578)/0.005986),"int"),"string")) as pickup_cell, str:concat(convert(convert(math:ceil((↩
    dropoff_latitude - 41.47718278)/-0.004491556),"int"),"string"), "-", convert(convert(math:ceil↩
    ((dropoff_longitude + 74.916578)/0.005986),"int"),"string")) as dropoff_cell
insert into BigAreaStream;
```

To obtain a grid of 250 m by 250 m, we did the same computations as above, but divided the latitude difference by 0.004491556/2 and the longitude difference by 0.005986/2.

```
from ProcessedStream
select medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, ↩
    dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude,↩
     dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount↩
    , tolls_amount, total_amount, str:concat(convert(convert(math:ceil((pickup_latitude - ↩
    41.47718278)/(-0.004491556/2)),"int"),"string"), "-", convert(convert(math:ceil((↩
    pickup_longitude + 74.916578)/(0.005986/2)),"int"),"string")) as pickup_cell, str:concat(↩
    convert(convert(math:ceil((dropoff_latitude - 41.47718278)/(-0.004491556/2)),"int"),"string"), ↩
    "-", convert(convert(math:ceil((dropoff_longitude + 74.916578)/(0.005986/2)),"int"),"string")) ↩
    as dropoff_cell
insert into SmallAreaStream;
```

In the end, we used the BigAreaStream, with the 300 x 300 grid (cells of 500 m x 500 m) to create a new stream with the route traveled in each ride. To do that, we simply concatenated the pickup cell with the dropoff cell using a "/" as a separator, as shown in the code below.

```
from BigAreaStream
select medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, ↩
    dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude,↩
     dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount↩
    , tolls_amount, total_amount, str:concat(pickup_cell, "/", dropoff_cell) as route
insert into RouteStream;
```

Below, we present a diagram showing all the streams created.

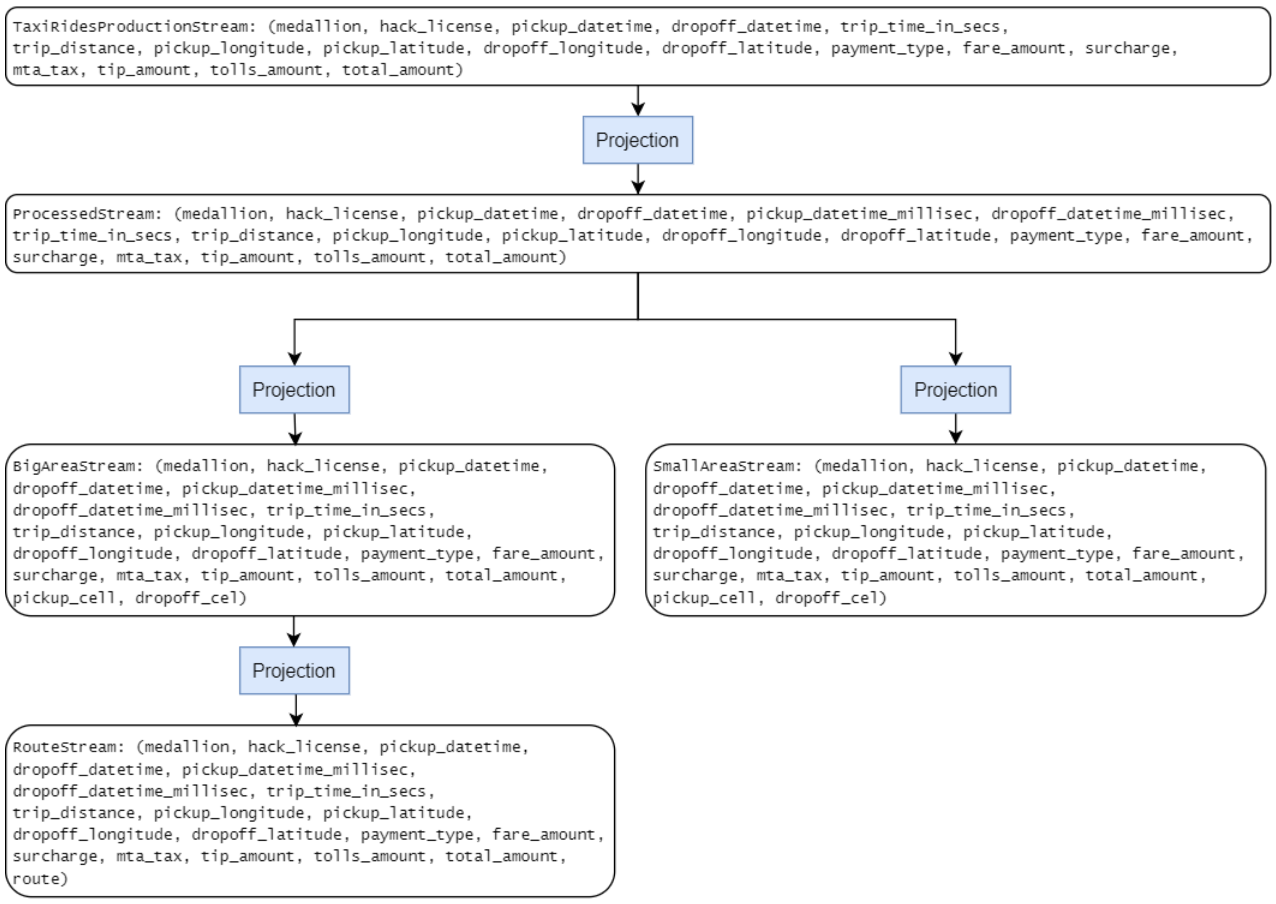

TaxiRidesProductionStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount)
Projection
ProcessedStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount)
Projection
Projection
BigAreaStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, pickup_cell, dropoff_cel)
SmallAreaStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, pickup_cell, dropoff_cel)
Projection
RouteStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, route)

Figure 2: Diagram with the streams created in the processing steps.

Having created all the streams with the processed information, we can now use them to address the proposed questions.

Query 1 - Most Frequent Routes

In this query, the goal was to find the top 10 most frequent routes during the last 30 minutes, using a 300 x 300 cell grid.

Using the `RouteStream` defined beforehand, we started by defining a sliding window of 30 seconds. Note that the time in the dataset is accelerated, with 1 second in real life corresponding to 1 minute in the dataset. Thus, if we want to know the most frequent routes in the last 30 minutes, we should create a window of 30 seconds.

We then selected the route and counted the records, grouping the results by route. We inserted the routes and the counts for each route in a stream called `CountRoutes`.

```
from RouteStream#window.time(30 sec)
select route, count() as count
group by route
having count > 0
insert into CountRoutes;
```

Using the `CountRoutes` stream, we ordered the counts in descending order and limited the results to show only the top 10 routes with the highest counts. The top 10 routes and their respective counts were then inserted in a stream called `FrequentRoutes`.

```
from CountRoutes
select route, count
order by count desc
limit 10
insert into FrequentRoutes;
```

The diagram with the streams created for this query is shown below.

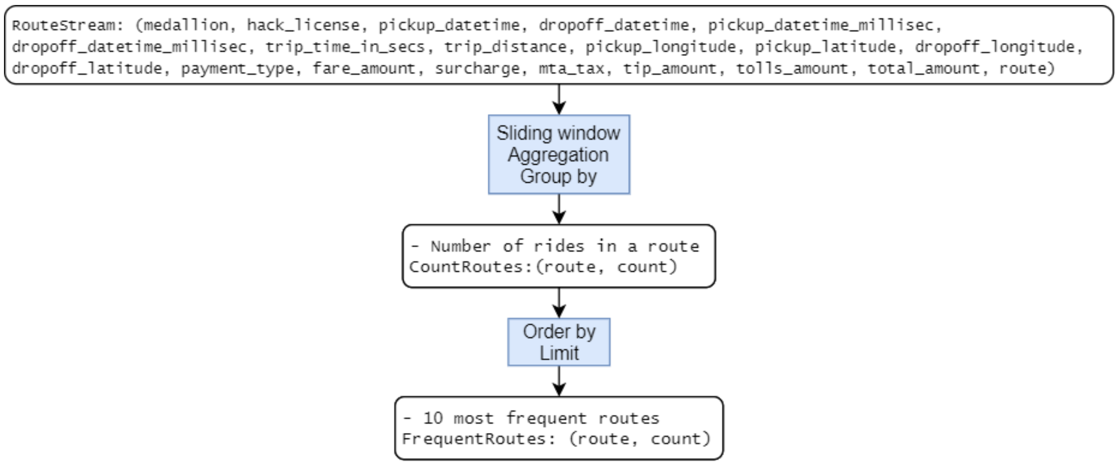
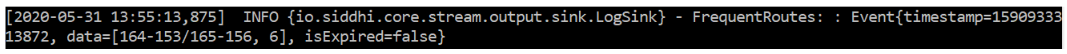

RouteStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, route)
Sliding window Aggregation Group by
- Number of rides in a route
CountRoutes:(route, count)
Order by Limit
- 10 most frequent routes
FrequentRoutes: (route, count)

Figure 3: Diagram with the streams used for Query 1.

We also show an example of the results, that allow taxi drivers to know which areas and routes have the highest number of clients.



```
[2020-05-31 13:55:13,875]  INFO {io.siddhi.core.stream.output.sink.LogSink} - FrequentRoutes: : Event{timestamp=15909333
13872, data=[164-153/165-156, 6], isExpired=false}
```

Figure 4: Results obtained for Query 1.

Query 2 - Most Profitable Areas

In this query we were asked to find the most profitable areas for taxi drivers. We defined the profitability of an area as the ratio between the profit generated in the area within the last 15 minutes and the number of empty taxis in this same area in the last 30 minutes. The profit generated in an area is computed by calculating the average fare and tip for trips that started in the area (that is, have this area indicated as the pickup area) and ended within the last 15 minutes. The empty taxis in a certain area are the ones thay had a dropoff in this area and no following pickup. We divided the average profit by the number of empty taxis, so that we penalize areas where there are a lot of empty taxis, as this will lead to more competition between taxi drivers and it is consequently harder for a driver to get a client.

For this problem we used a cell size of 250 m x 250 m, i.e., we are representing New York in a 600 x 600 grid.

Using the `SmallAreaStream` defined beforehand, we started by defining a sliding window of 15 seconds, to represent 15 minutes in the real scenario.

We then selected the pickup cell and calculated the average profit (which is defined as the fare amount plus tip amount), grouping the results by pickup cell. We inserted the pickup cell and the average profit for each pickup cell in a stream called `AvgProfitStream`.

```
from SmallAreaStream#window.time(15 sec)
select avg(fare_amount+tip_amount) as avg_profit, pickup_cell
group by pickup_cell
insert into AvgProfitStream;
```

Also using the `SmallAreaStream` stream, we created a new stream called `Taxis30` where we selected only the needed information for computing the number of empty taxis, namely the dropoff cell, pickup cell and the medallion. For that, we used a sliding window of 30 seconds (representing 30 minutes in the dataset).

```
from SmallAreaStream#window.time(30 sec)
select dropoff_cell, pickup_cell, medallion
insert into Taxis30;
```

In order to compute the number of empty taxis in a given area, we decided to use an approach slightly different from the one proposed. First, using the `Taxis30` stream, we computed the total number of taxis that had a dropoff in a given cell in the last 30 minutes, and inserted the results in a stream called `AllTaxis`.

```
from Taxis30
select dropoff_cell, pickup_cell, count(medallion) as count_taxis
group by dropoff_cell
insert into AllTaxis;
```

Then, using the `Taxis30` stream we computed the number of busy taxis and inserted this in a stream called `BusyTaxis`. To do that, we considered taxis that had a dropoff followed by a pickup in the last 30 minutes in the same cell.

```
from every e1 = Taxis30 -> e2 = Taxis30[e1.dropoff_cell == e2.pickup_cell]
select e1.dropoff_cell, count(e2.medallion) as count_busy_taxis
group by e1.dropoff_cell
insert into BusyTaxis;
```

In the first line we look for a pattern of two events: e1 followed by e2, for the same cell, i.e. the dropoff cell of the first ride must be the same as the pickup cell of the second ride. Doing so, we can count the number of taxis that had a dropoff followed by a pickup, that is, the number of busy taxis.

Then we finally computed the number of empty taxis and inserted it in a stream called `EmptyTaxis`. To do that, we computed the difference between the total number of taxis and the number of busy taxis in a given cell in the last 30 minutes. Note that, if the number of empty taxis in a given area is zero, we replace this value with 0.5, to increase the relative profitability of that area.

```
from AllTaxis#window.time(30 sec) as A join BusyTaxis#window.time(30 sec) as B on (A.dropoff_cell ↩
    == B.dropoff_cell)
select B.dropoff_cell, ifThenElse(A.count_taxis - B.count_busy_taxis == 0, 0.5, convert(A.↩
    count_taxis - B.count_busy_taxis,"double")) as count_empty_taxis
group by B.dropoff_cell
insert into EmptyTaxis;
```

Lastly, to answer this query we computed the average profitability for each area and inserted it in a stream called `Profitability`. For that, we divided the profit in the last 15 minutes by number of empty taxis in the last 30 minutes. We then ordered the results by the profitability and limited them to 5, in order to find the 5 most profitable areas.

```
from AvgProfitStream#window.time(15 sec) as P join EmptyTaxis#window.time(30 sec) as E on P.↩
    pickup_cell == E.dropoff_cell
select P.pickup_cell, (P.avg_profit/E.count_empty_taxis) as profitability
group by P.pickup_cell
order by profitability desc
limit 5
insert into Profitability;
```

The diagram with the streams created for this query is shown below.

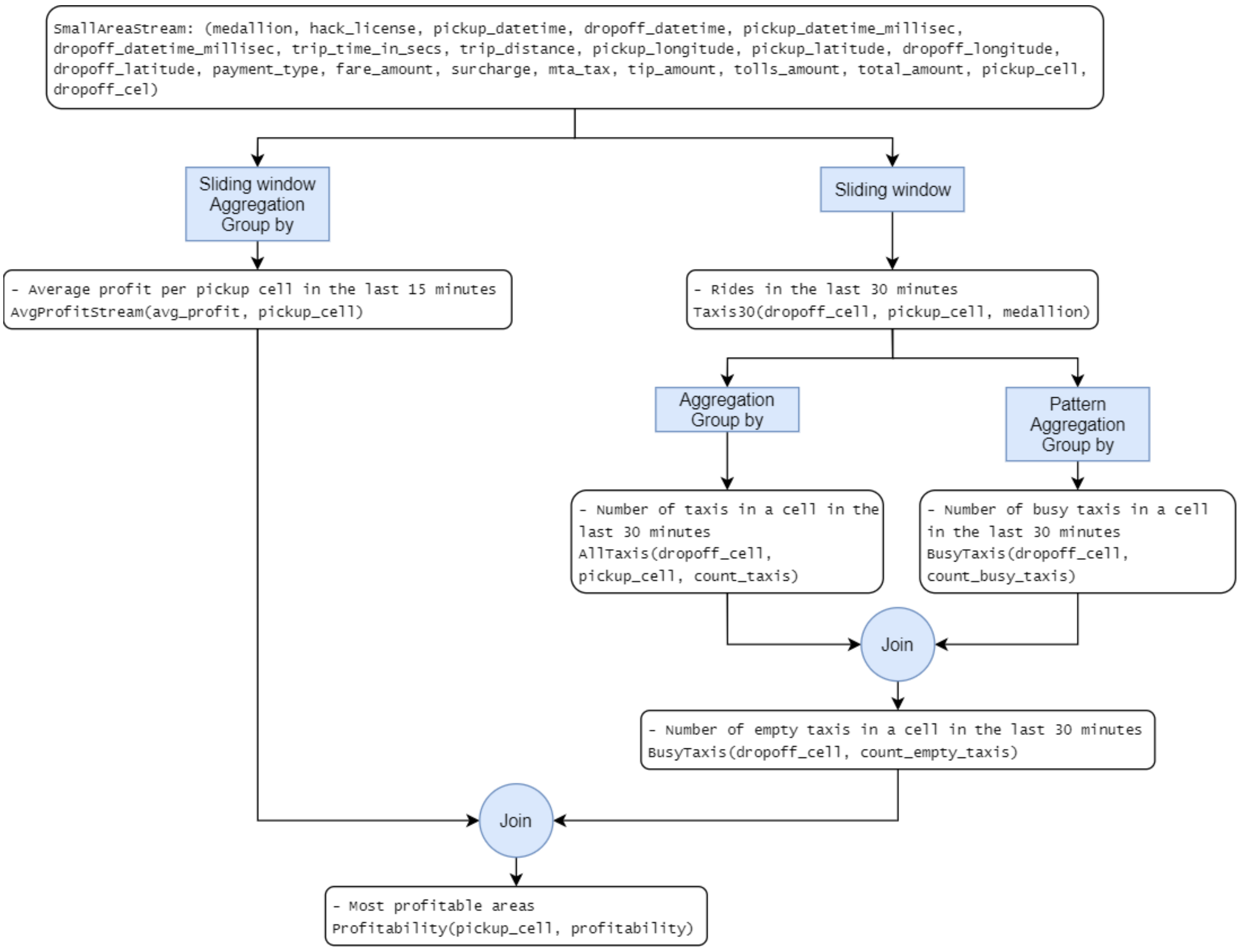
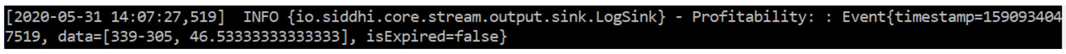

SmallAreaStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, pickup_cell, dropoff_cel)
Sliding window Aggregation Group by
Sliding window
- Average profit per pickup cell in the last 15 minutes AvgProfitStream(avg_profit, pickup_cell)
- Rides in the last 30 minutes Taxis30(dropoff_cell, pickup_cell, medallion)
Aggregation Group by
Pattern Aggregation Group by
- Number of taxis in a cell in the last 30 minutes AllTaxis(dropoff_cell, pickup_cell, count_taxis)
- Number of busy taxis in a cell in the last 30 minutes BusyTaxis(dropoff_cell, count_busy_taxis)
Join
- Number of empty taxis in a cell in the last 30 minutes BusyTaxis(dropoff_cell, count_empty_taxis)
Join
- Most profitable areas Profitability(pickup_cell, profitability)

Figure 5: Diagram with the streams used for Query 2.

We also show an example of the results, that could help the taxi drivers choose the area where to wait for clients.



```
[2020-05-31 14:07:27,519]  INFO {io.siddhi.core.stream.output.sink.LogSink} - Profitability: : Event{timestamp=159093404
7519, data=[339-305, 46.53333333333333], isExpired=false}
```

Figure 6: Results obtained for Query 2.

Query 3 - High Idle Time

The goal of this query was to alert the city whenever the average idle time of taxis is greater than a certain amount of time (in this case, we considered 5 minutes). Since we are not interested in the pickup or the dropoff areas, we computed the idle time using the stream `ProcessedStream`.

The idle time of a taxi is the time mediating between the dropoff time of a ride, and the pickup time of the following ride. Thus, to compute the idle time of a taxi, we used patterns, as shown in the following code.

```
from every e1 = ProcessedStream -> e2 = ProcessedStream[e1.medallion == e2.medallion]
select e2.pickup_datetime_millisec - e1.dropoff_datetime_millisec as idle_time
having idle_time > 0 and idle_time < 3600000
insert into IdleTime;
```

In the first line we look for a pattern of two events: e1 followed by e2, for the same taxi (i.e. the medallion must be the same for e1 and e2). Then, we compute the idle time between the dropoff time of e1 (the first ride) and the pickup time of e2 (the following ride), using the timestamps in milliseconds computed before. Due to some errors in the dataset, this idle time can be negative sometimes, so we filtered the obtained idle time in order to keep only the times above zero. Moreover, we want the idle times only for the taxis that are available. It is assumed that a taxi is available if it had at least one ride in the last hour. Therefore, we created another filter for the idle times, in order to keep only the times below one hour.

The idle times computed in the stream `IdleTime` are in milliseconds, so we need to convert them to minutes. To do that, we use the function `extract`, as shown below.

```
from IdleTime
select time:extract(idle_time, "minute") as idle_time_min
insert into IdleTimeMin;
```

Afterwards, we only need to compute the average idle time from the times emitted by the stream `Idle-TimeMin`. To do that, we defined a sliding window of 30 seconds (which corresponds to 30 minutes in the dataset), in order to compute the average only for the last 30 minutes. Lastly, we created a stream that, if the average idle time computed is greater than 5 minutes, emits an alert message and the corresponding average idle time.

```
from IdleTimeMin#window.time(30 sec)
select avg(idle_time_min) as average_idle_time
insert into AvgIdleTime;

from AvgIdleTime[average_idle_time > 5]
select "ALERT: High Idle Time" as alert, average_idle_time
insert into AlertStream;
```

The diagram with the streams created for this query is shown below.

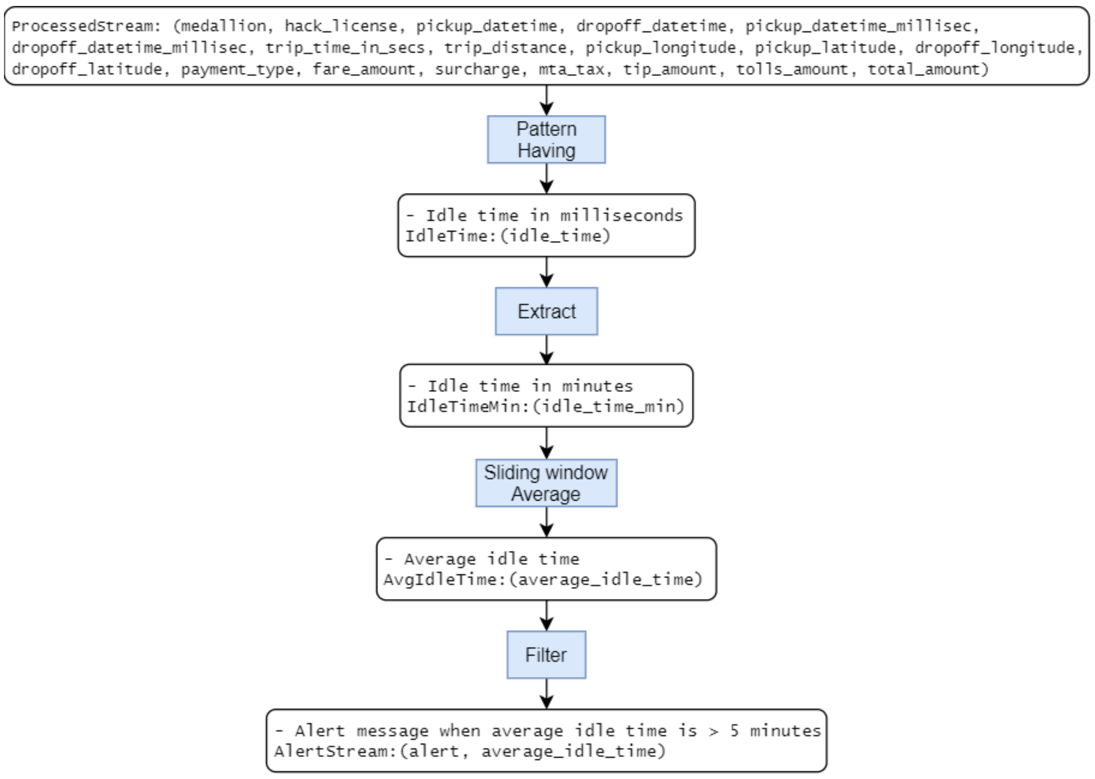
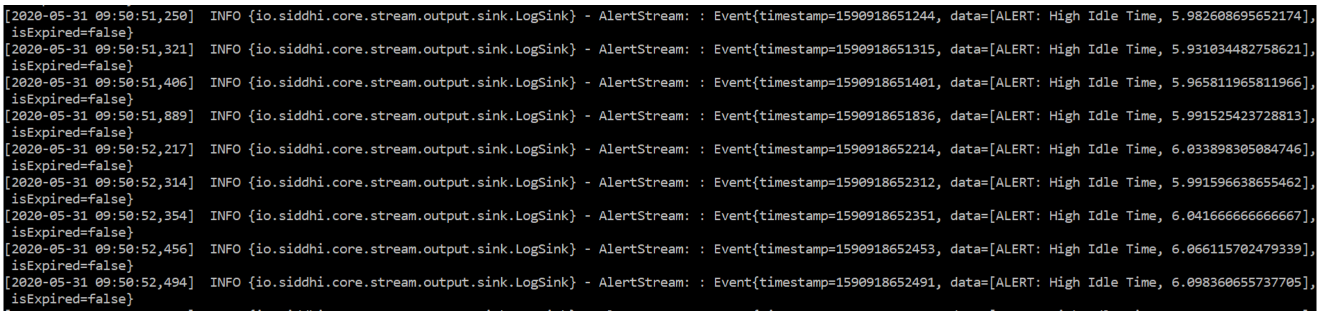

ProcessedStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount)
Pattern Having
- Idle time in milliseconds IdleTime:(idle_time)
Extract
- Idle time in minutes IdleTimeMin:(idle_time_min)
Sliding window Average
- Average idle time AvgIdleTime:(average_idle_time)
Filter
- Alert message when average idle time is > 5 minutes AlertStream:(alert, average_idle_time)

Figure 7: Diagram with the streams used for Query 3.

We also show a few examples of the results, which can be very helpful for the taxi drivers, to know whenever they have to wait longer to pickup clients from one ride to another.



```
[2020-05-31 09:50:51,250]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918651244, data=[ALERT: High Idle Time, 5.982608695652174],
 isExpired=false}
[2020-05-31 09:50:51,321]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918651315, data=[ALERT: High Idle Time, 5.931034482758621],
 isExpired=false}
[2020-05-31 09:50:51,406]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918651401, data=[ALERT: High Idle Time, 5.965811965811966],
 isExpired=false}
[2020-05-31 09:50:51,889]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918651836, data=[ALERT: High Idle Time, 5.991525423728813],
 isExpired=false}
[2020-05-31 09:50:52,217]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918652214, data=[ALERT: High Idle Time, 6.033898305084746],
 isExpired=false}
[2020-05-31 09:50:52,314]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918652312, data=[ALERT: High Idle Time, 5.991596638655462],
 isExpired=false}
[2020-05-31 09:50:52,354]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918652351, data=[ALERT: High Idle Time, 6.041666666666667],
 isExpired=false}
[2020-05-31 09:50:52,456]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918652453, data=[ALERT: High Idle Time, 6.066115702479339],
 isExpired=false}
[2020-05-31 09:50:52,494]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AlertStream: : Event{timestamp=1590918652491, data=[ALERT: High Idle Time, 6.098360655737705],
 isExpired=false}
```

Figure 8: Results obtained for Query 3.

Query 4 - Congested Areas

This query's objective was to emit an alert whenever there is a congested area. For this, we considered that a congested area might be one that, when the taxis enter there, the rides increase in their duration. Thus, there should be an alert when a taxi has a peak in the duration of a ride, followed by at least 3 rides with increasing duration. The alert should contain the location where the taxi started this ride which had the peak duration.

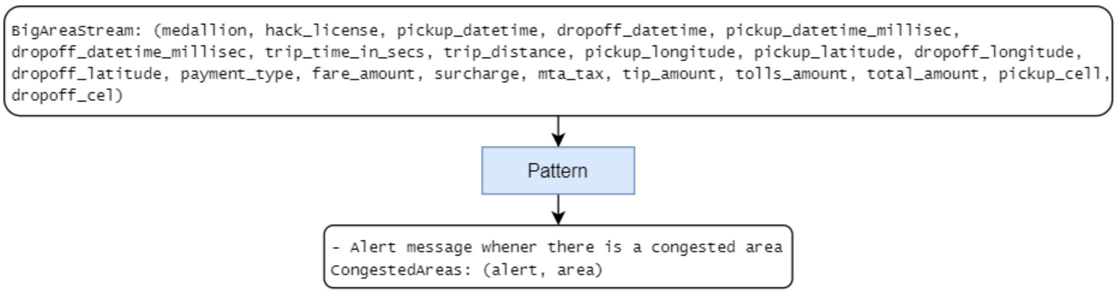
Since we want to obtain our results for areas (cells) of 500 m by 500 m, we used the `BigAreaStream` stream. Starting from this stream, we use the perfect tool for this type of problem: Siddhi patterns. These will be comparing different events for a match that represents our situation. In this case, we initialize 6 events. The first one will not need any condition as there are no others to compare to.

As we initialize the rest of the events, we need to specify the comparison condition to be applied. The first one is that both event's pickup cell is the same, as we want to track an area behavior. The other condition is related to the duration of the rides in each event. The second event must have a duration greater than the first and the third ones (i.e., it must be a peak). Then, the fourth, fifth and sixth events must have increasing duration.

In the end, we select a message to be used as alert and the area where these events occurred, and insert this information in a stream called `CongestedAreas`.

```
from every event1 = BigAreaStream,
            event2 = BigAreaStream[event1.pickup_cell == event2.pickup_cell and event1.↩
                trip_time_in_secs < event2.trip_time_in_secs],
            event3 = BigAreaStream[event2.pickup_cell == event3.pickup_cell and event2.↩
                trip_time_in_secs > event3.trip_time_in_secs],
            event4 = BigAreaStream[event3.pickup_cell == event4.pickup_cell and event3.↩
                trip_time_in_secs < event4.trip_time_in_secs],
            event5 = BigAreaStream[event4.pickup_cell == event5.pickup_cell and event4.↩
                trip_time_in_secs < event5.trip_time_in_secs],
            event6 = BigAreaStream[event5.pickup_cell == event6.pickup_cell and event5.↩
                trip_time_in_secs < event6.trip_time_in_secs]
select "ALERT: Congested Area" as alert, event2.pickup_cell as area
insert into CongestedAreas;
```

The diagram with the streams created for this query is shown below.



```
BigAreaStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec,
dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude,
dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, pickup_cell,
dropoff_cel)
```

```
- Alert message whener there is a congested area
CongestedAreas: (alert, area)
```

Figure 9: Diagram with the streams used for Query 4.

This query is quite useful to the taxi drivers, because they can avoid certain areas that are more congested. However, we did not get any results, probably because in the time where we ran the source stream there were no congested areas, as this is a quite specific situation.

Query 5 - Most Pleasant Taxi Driver

The goal of this query was to show the most pleasant taxi driver in a given day. If a taxi driver is pleasant, then it makes sense that the passengers give a higher tip. However, the tip also depends on the total price of a ride. Therefore, we computed the tip percentage, i.e., how much of the total amount paid in a ride corresponds to the tip given by the passenger. It is best to compute this percentage to compare between taxi drivers, because there can be taxi drivers that had longer and more expensive rides and so the total tip is higher than taxi drivers that had shorter and less expensive rides, even if the former are not as nice as the latter.

We start by creating a tumbling window of 24 minutes, which corresponds to 24 hours in the dataset, of the already processed data stream. Grouping by hack license, we computed the division between the sum of the tip amounts and the sum of the total amounts. We applied the average to this relative value and selected it along with the hack license. We ordered the results by the average tip percentage and selected only the first result (top 1). The results were then sent to the corresponding sink.

```
from ProcessedStream#window.timeBatch(24 min)
select hack_license, avg(sum(tip_amount)/sum(total_amount)) as avg_tip_percentage
group by hack_license
```

```
order by avg_tip_percentage desc
limit 1
insert into PleasantTaxiDriver;
```

The diagram illustrating the computations performed is shown below.

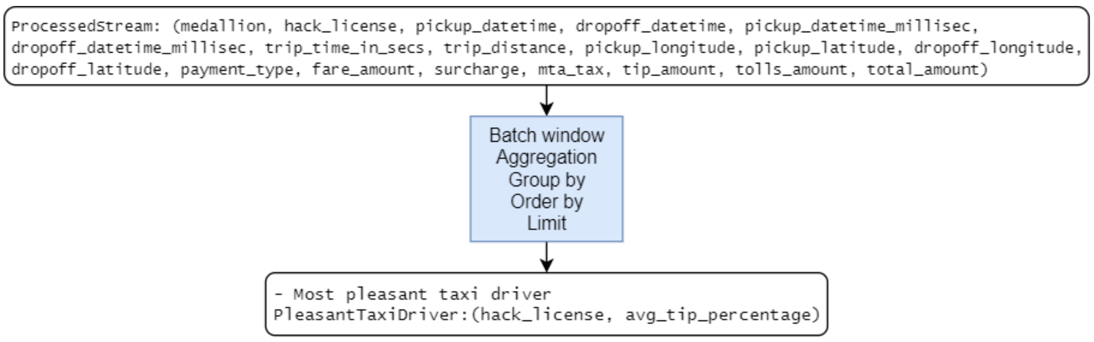
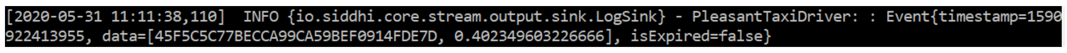

ProcessedStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount)
Batch window Aggregation Group by Order by Limit
- Most pleasant taxi driver
PleasantTaxiDriver:(hack_license, avg_tip_percentage)

Figure 10: Diagram with the streams used for Query 5.

We also show an example of the results in Figure 11, where we can see that the most pleasant taxi driver for that day received an average of 40% of the total paid amount in tips.



```
[2020-05-31 11:11:38,110]  INFO {io.siddhi.core.stream.output.sink.LogSink} - PleasantTaxiDriver: : Event{timestamp=1590
922413955, data=[45F5C5C77BECCA99CA59BEF0914FDE7D, 0.402349603226666], isExpired=false}
```

Figure 11: Results obtained for Query 5.

Query 6 - Revenue Discrimination for the Highest Paying Areas

As an optional query we decided to look into how the revenue discrimination looks like for the pickup areas where the taxi drivers make most money, in each 2 hour window. A pickup area may look very lucrative when the total amount received by the drivers is high, but is it really profit or is there a high percentage of surcharge or tolls included, masking the actual revenue?

In this query, we wanted to present the results in percentage, rounded to two decimal cases. However, because the `round` function in SiddhiQL only rounds a number to zero decimal cases, we started by creating a function that returns a given value rounded to two decimal cases.

```
define function round_value[JavaScript] return double {
    var value = data[0];
    return Math.round(value*100)/100; };
```

Then, from the `BigAreaStream`, we used a 2 hour window in time batches and calculated for each pickup area the total received amount and the percentages of each kind of payment in the total received amount. This gives us the revenue discrimination. In the end we ordered by the total amount and limited the results to 5, so that we show for each 2 hour window the revenue discrimination of the top 5 highest paying areas, and inserted this in a new stream called `Statistics`.

```
from BigAreaStream#window.timeBatch(2 min)
select pickup_cell, sum(total_amount) as sum_total_amount, round_value((sum(fare_amount)/sum(↩
    total_amount))*100) as fare, round_value((sum(tip_amount)/sum(total_amount))*100) as tip, ↩
    round_value((sum(surcharge)/sum(total_amount))*100) as surcharge, round_value((sum(tolls_amount↩
    )/sum(total_amount))*100) as tolls, round_value((sum(mta_tax)/sum(total_amount))*100) as ↩
    mta_tax
group by pickup_cell
order by sum_total_amount desc
```

```
limit 5
insert into Statistics;
```

The diagram illustrating the computations performed is shown below.

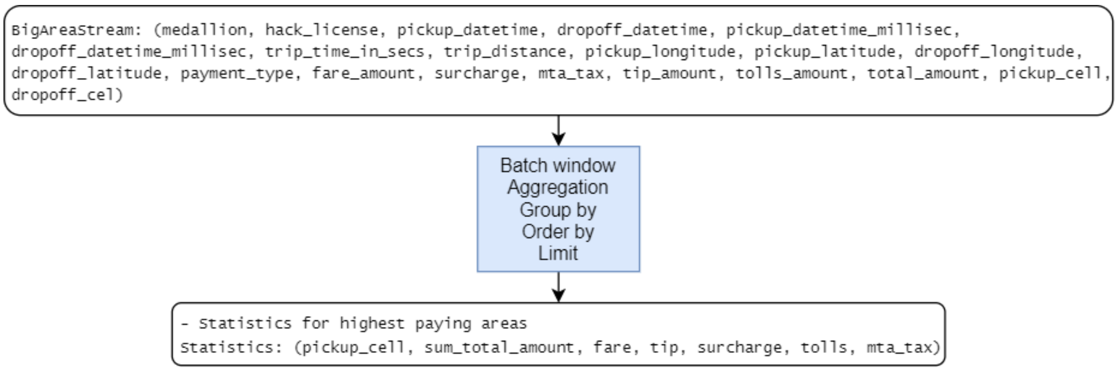
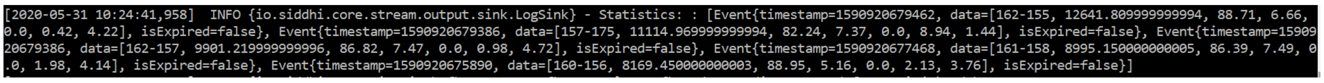

BigAreaStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, pickup_cell, dropoff_cel)
Batch window
Aggregation
Group by
Order by
Limit
- Statistics for highest paying areas
Statistics: (pickup_cell, sum_total_amount, fare, tip, surcharge, tolls, mta_tax)

Figure 12: Diagram with the streams used for Query 6.

We also show an example of the results in Figure 13.



```
[2020-05-31 10:24:41,958]  INFO {io.siddhi.core.stream.output.sink.LogSink} - Statistics: : [Event{timestamp=1590920679462, data=[162-155, 12641.809999999994, 88.71, 6.66, 
0.0, 0.42, 4.22], isExpired=false}, Event{timestamp=1590920679386, data=[157-175, 11114.969999999994, 82.24, 7.37, 0.0, 8.94, 1.44], isExpired=false}, Event{timestamp=15909
20679386, data=[162-157, 9901.219999999996, 86.82, 7.47, 0.0, 0.98, 4.72], isExpired=false}, Event{timestamp=1590920677468, data=[161-158, 8995.150000000005, 86.39, 7.49, 0
.0, 1.98, 4.14], isExpired=false}, Event{timestamp=1590920675890, data=[160-156, 8169.450000000003, 88.95, 5.16, 0.0, 2.13, 3.76], isExpired=false}]
```

Figure 13: Results obtained for Query 6.

The results show that, as expected, most of the received amount corresponds to the actual paid fare. For the areas in Figure 13, the percentage of total amount that corresponds to the tip varies between 5% and 7.5%. The surcharge percentage was 0% for these areas. Tolls seem to vary a lot for these areas, from 0.4% to 9%. Mta tax is actually a fixed value per trip of 50 cents, so it is not relevant for this price discrimination.

Query 7 - Routes with Highest Total Amount per Mile

For this optional query, we decided to check in which routes the total paid amount per mile is highest in each hour. This is useful for the taxi drivers, because they can decide in which areas to pickup passengers, based on those routes, since the more a passenger pays per mile, the more advantageous it is for the taxi driver.

We started by creating a tumbling window of 1 minute (to represent 1 hour in the dataset) and grouping the results by route. For each route, we summed the total paid amount and the total distance traveled in all the rides that used that route. Then we divided these two values to get the total amount per mile traveled. Lastly, we ordered by the above variable and limited the results to show only the first 5 rows, to get the 5 most valuable routes per mile.

```
from RouteStream#window.timeBatch(1 min)
select route, sum(total_amount)/sum(trip_distance) as amount_per_mile
group by route
order by amount_per_mile desc
limit 5
insert into AmountPerMile;
```

The diagram illustrating the computations performed is shown below.

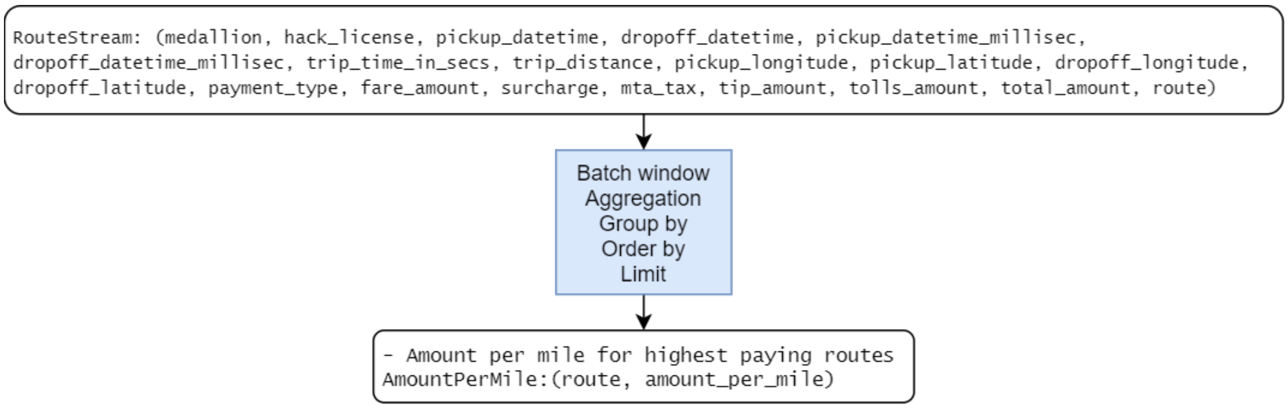
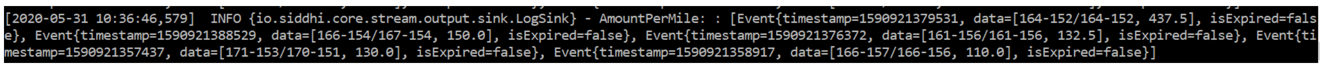

RouteStream: (medallion, hack_license, pickup_datetime, dropoff_datetime, pickup_datetime_millisec, dropoff_datetime_millisec, trip_time_in_secs, trip_distance, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount, route)
Batch window
Aggregation
Group by
Order by
Limit
- Amount per mile for highest paying routes
AmountPerMile:(route, amount_per_mile)

Figure 14: Diagram with the streams used for Query 7.

We also show an example of the results in Figure 15.



```
[2020-05-31 10:36:46,579]  INFO {io.siddhi.core.stream.output.sink.LogSink} - AmountPerMile: : [Event{timestamp=1590921379531, data=[164-152/164-152, 437.5], isExpired=fals
e}, Event{timestamp=1590921388529, data=[166-154/167-154, 150.0], isExpired=false}, Event{timestamp=1590921376372, data=[161-156/161-156, 132.5], isExpired=false}, Event{ti
mestamp=1590921357437, data=[171-153/170-151, 130.0], isExpired=false}, Event{timestamp=1590921358917, data=[166-157/166-156, 110.0], isExpired=false}]
```

Figure 15: Results obtained for Query 7.

Note that the routes with the highest amount per mile are usually very short - some of them even have the same pickup and dropoff cell. This may be because the total paid amount may include some initial fee, which means that the total paid amount is high even though the distance traveled was very short.