MS-BGDA – INFO-H515

Phase 1 – Correlation of Bike Sensors

Distributed Data Management

AMRI Hakim
hakim.amri@ulb.be
000 459 153

BAGUIA Rania
rania.baguia@ulb.be
000 459 242

CAILLIAU Julian
julian.cailliau@ulb.be
000 459 856

JDAOUDI Mehdi
mehdi.jdaoudi@ulb.be
000 457 507

Project submitted under the supervision of
Prof. Dimitrios Sacharidis,
Prof. Gianluca Bontempi
and teaching assistant Antonios Kontaxakis

Academic year
2022 - 2023

In the context of the
Specialised Master in Big Data and Data Sciences

Table of Content

List of Figures

List of Tables

List of Equation

Project Video

You can see the presentation video here:

https://universitelibrebruxelles.sharepoint.com/:v:/s/GRP_PROJ-INFO-H515BigDataMan.Ana/EXLrPskX_uZOuug6obwDGWYBPbjNPefatfMsFMSQSBMkPw?e=cgeytP

1. Introduction

For several years, the city of Brussels has been encouraging its citizens to use their bikes. To monitor the growth of bike use, 18 sensors recording the number of bikes passing by have been placed all over Brussels. The objective of this work is to analyse this bike traffic by finding correlated sensors. For this purpose, the Pearson Correlation can be used. It allows to see which sensors were the most correlated between December 6, 2018, and March 31, 2023. To go further, it is interesting for the city of Brussels to be able to analyse this correlation in real time. The Pearson correlation will be also applied on a simulated data stream to compute the correlation between sensors at real time. Finally, to analyse the changes in correlation depending on different periods, a sliding window is applied to the stream. To this end, the Spark engine will be used since, in addition to allowing the parallelization of computations, the tool allows the analysis of data streams.

2. Data Extraction and Pre-processing

The data from the Brussels Mobility Bike Counts API was provided as such: historical data containing the number of bikes crossing various sensors placed in Brussels every 15 minutes, and a JSON file containing the list of the different sensors placed in Brussels.

2.1. Procedure

1. Before beginning any computation, the required data has been retrieved and directly parallelized for pre-processing. The sensors names were retrieved and parallelized in a Spark context. The data from 2018-12-06 to 2023-03-31 was then retrieved for each sensor and the sensor name was added with the foreach() Spark function.
2. Missing values then had to be handled as well. The aim is to replace all missing value by 0 meaning that no bike passed by the sensor at the selected time. The following technique was used to create the table and complete the missing values:
    a. First, a "perfect" table was created. This table contains, for all sensors, all the time gaps for every day between 06/12/2018 and 31/03/2023. The values of the "count" and "speed" are set to 0 and -1 respectively in all rows of the table. The table is then parallelized. The identifier of the sensors is added to the table and a key which includes Date-Time gap-Sensor ID is created.
    b. Then, the retrieved data and the perfect table are joined using a "Left Join".
    c. Finally, to facilitate future calculations, a timestamp is added to each observation. To do so, each observation of a sensor is ordered by “date” and “time gap”. Then the timestamp is added, each new timestamp being greater than the previous ones.

The final RDD contains the values retrieved from the API and values equal to 0 and -1 for the count and speed of the periods where nothing was recorded. The data is exported from the final RDD into single CSV file usable for the tasks. The columns in this file are: Date – Time Gap – Count – Average Speed – Sensor – Timestamp.

3. Project Tasks

3.1. Task 1 – Batch processing

3.1.1. Methodology

For the sake of simplicity, a rearranged version of the Pearson Correlation formula has been used. This formula (Equation 1) suggests a convenient single-pass algorithm for calculating sample correlations which makes it easily parallelizable.

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\,\sqrt{n\sum y_i^2 - (\sum y_i)^2}}.$$

Equation 1: Pearson Correlation Coefficient Rearranged

The "map()" function is used a first time to create, for each observation, a tuple containing: datetime, Time gap, count, speed, sensor and timestamp of the observation with the right data type. In order to compute the correlation, the function will use the two following functions: one computing the cumulative sums and another one computing the product between each observation at one timestamp.

To compute the cumulative sums, the observations of each sensor are shuffled and grouped with a groupByKey(). Tuples having as key the sensor id and as value the set of observations linked to this sensor are created. These tuples are then passed to the "compute_cum_counts()" function with a mapValues() transformation. This allows to calculate, for each timestamp “t”, the sum and the sum of the square of all the bike counts counted by the sensor since the first timestamp. The result is the RDD composed of the initial tuple for each observation to which has been added the cumulative sum and the cumulative sum of squares up until the timestamp t. Grouping the observations by sensors with groupByKey() allows Spark to distribute all observations of a sensor in the same partition. All that remains is to perform additions and multiplications with a mapValues() on all the observations of this sensor. This does not require a shuffle and thus, computations can be performed within the same partition.

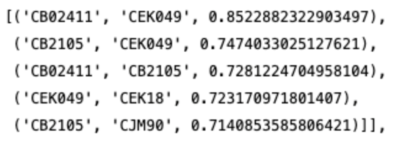
Afterwards, it remains to compute the cumulative sum of the products of the observations ($\sum x_i y_i$) for each pair of sensors. After creating each pair of sensors, the observations are grouped by timestamps. The productAtForPair() function will, for each pair of sensors multiply their observations at the timestamp "t". In terms of parallelisation, a shuffle is operated when the data is partitioned by timestamps. Then, the use of flatmap() allows to multiply observations without shuffling data between the partitions.

Finally, after computing all elements required, the correlation() function can be applied. It follows the Pearson correlation formula to get the correlation for any timestamp t for any pair of sensors. To do that, the data is grouped used groupByKey() with the key being a pair of sensor. All elements for the correlation computation being already computed and recorded; the only thing left is to strictly apply the formula. Since the correlations do not depend on each other, they can be calculated in parallel with a mapValues() for each pair of sensor at each timestamp.

To conclude, this method allows most of the computations to be done fast and in parallel. However, it also requires increasing the size of the initial data by adding new values such as the cumulative sums and the observations products.

3.1.2. Results and sanity check

To visualize the results, a graph containing the evolution of the correlation between a chosen sensor and all the other 17 sensors can be found in the appendix (Figure 3). It can be noticed that the bigger the number of timestamps considered, the more the correlation between two sensors converges. The top-5 most correlated pairs at the timestamp 69 on day 2018-12-14 can be found in Table 1.



```
[('CB02411', 'CEK049', 0.8522882322903497),
 ('CB2105', 'CEK049', 0.7474033025127621),
 ('CB02411', 'CB2105', 0.7281224704958104),
 ('CEK049', 'CEK18', 0.723170971801407),
 ('CB2105', 'CJM90', 0.7140853585806421)]],
```

Table 1 - Top 5 most correlated pairs of sensors for batch processing

To verify the obtained results, two sanity checks were performed. Firstly, the correlations computed must be between [-1:1]. None of the correlations calculated were outside the boundaries. Secondly, the results were assessed by comparing them with the Pearson correlations computed with Numpy's built-in function "corrcoef()" and the obtained results were similar.

3.2. Task 2 – Stream Processing

3.2.1. Methodology

Producer

The socket-based approach has been applied in this work to create the data stream which allows to send batches of data at regular intervals. The sending interval (Δ) of data was set to 5 seconds and the batch size (Π) to 30 days of data. To be able to send the batch, its format was changed using the array2string() function.

Consumer

On the consumer side, the same methodology and correlation formula as in the previous task was used. To be able to apply the correlation, some modifications were apported to the process to handle states updating at each batch.

The first step is to connect to the same socket as the one used for the producer side. Batches are received at 5 seconds interval and contain the data of 30 days of observations.

The second step is to create the state variables that will be updated with each new batch. Four of them were needed: the first keeps track of the last timestamp of the batch, the second encodes, for each sensor separately, the cumulative sum ($\sum x_i$) until its latest observation, the third similarly keeps the cumulative sum of the squared counts ($\sum x_i^2$) and finally, the last variable encodes the sum of the products ($\sum x_i y_i$) for each pair of sensors.

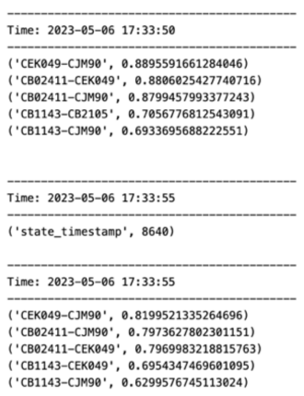
Next, the variables are computed. Regarding the cumulative sum of each sensor of the batch ($\sum x_i$), the data is first mapped in key-value pairs where the sensor is used as a key. In this way, the data can be grouped by sensor with a groupByKey(). The cumulative sum over all timestamps observed at each sensor can be therefore computed. Finally, for each sensor, with updateStateByKey(), the cumulative sum is summed to the one stored in the state variable. The same logic is applied for the computation of $\sum x_i^2$. These operations require one shuffle during the groupByKey() and the rest is done in parallel.

For the computation of $\sum x_i y_i$, the data is mapped in key-value pairs with the timestamps as a key and grouped by key. For each pair of sensor, the counts observations are multiplied at each timestamp with a mapValues(). Then, the data is grouped by pair of sensors to compute the cumulative sum of the product before updating the states.

The last step is to retrieve from the updated states the required values to compute the correlation for each pair of sensors. The 5 best pairs are then returned. With this the methodology, each transformation is done in parallel. However, to get all the variables pre-computed, tables needed to be joined each which shuffles the data each time.

3.2.2. Results

The following Table 2 - Top 5 most correlated pairs of sensors in streaming contextTable 2 shows an example of the results obtained for the five most correlated pairs in stream processing context for two consecutive batches. In this case, the batch interval is 5 seconds with one batch containing the data of 30 days of observations. To verify the veracity of the results, these were compared to the results obtained during the batch processing task and were found similar.



```
-------------------------------------------
Time: 2023-05-06 17:33:50
-------------------------------------------
('CEK049-CJM90', 0.8895591661284046)
('CB02411-CEK049', 0.8806025427740716)
('CB02411-CJM90', 0.8799457993377243)
('CB1143-CB2105', 0.7056776812543091)
('CB1143-CJM90', 0.6933695688222551)


-------------------------------------------
Time: 2023-05-06 17:33:55
-------------------------------------------
('state_timestamp', 8640)

-------------------------------------------
Time: 2023-05-06 17:33:55
-------------------------------------------
('CEK049-CJM90', 0.8199521335264696)
('CB02411-CJM90', 0.7973627802301151)
('CB02411-CEK049', 0.7969983218815763)
('CB1143-CEK049', 0.6954347469601095)
('CB1143-CJM90', 0.6299576745113024)
```

Table 2 - Top 5 most correlated pairs of sensors in streaming context

Thanks to the Spark interface, it is possible to see on the Figure 4 in appendix that the calculations are well distributed on the ten cores of the computer. Figure 5 in the appendix show the times taken by the different steps of the calculation. The total duration for the processing of the batch is 3 seconds.

Finally, some experimental results were obtained by varying the batch size and the batch interval. The computation time grows slower than the data size which indicates a good scalability. Another indicator of good scalability is the fact that increasing the number of cores allows to process data a lot more rapidly when the data size gets bigger as it can be seen in Figure 1.

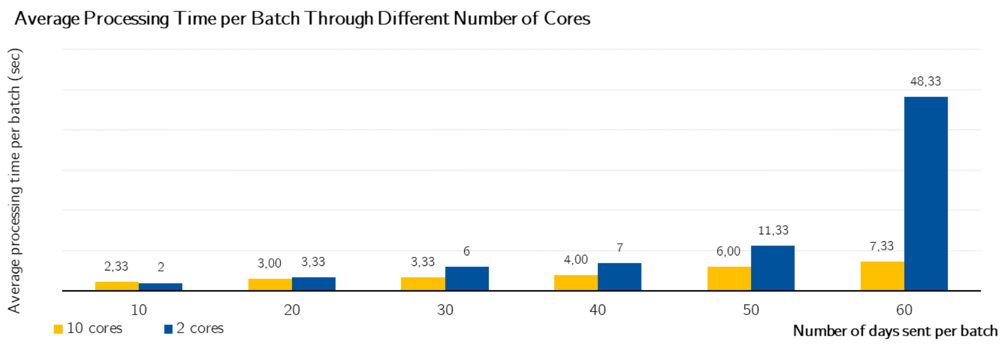


Figure 1- Experimental results when varying the batch size and number of cores

3.3. Task 3 - Sliding Window Processing

3.3.1. Methodology

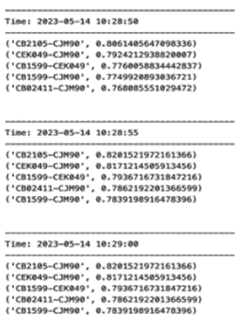
To realize the sliding window, a sliding interval (Δ) of 5 seconds and a window length (W) of 30 seconds were used. Thus, the sliding window computes every 5 seconds the last 6 batches sent by the producer.

The method for calculating the correlation is similar to the one applied for the stream processing task. The only difference is that keeping states is not required anymore. Thus, every 5 seconds, the correlation calculation starts from zero and is applied on the last 6 batches received in the window. The parallelization strategy is also identical to the one explained in the previous task. As shown on Figure 6 and Figure 7 in the appendix, the computations are done in parallel, and 2 seconds are necessary to process the whole window with the set of parameters presented above.

A problem encountered during the final analysis is that the consumer applies the calculation sometimes on 5 batches instead of 6. This can be due to the consumer receiving batches from the producer a little too late. It then considers only the last 5 batches received within 30 seconds, the last one not having arrived yet. This does not have much impact on the correlation for several reasons. The first one is that the window length is large enough to avoid a large bias in case of missing batches. The second is that if a batch is missing, it will be part of the next window. Thus, the correlation will end up being consistent.

3.3.2. Results

Computation of the 5 highest correlations pairs from 30 seconds of output displays results are shown in Table 3. To recall, Δ = 5 sec and W = 30 sec.



```
-------------------------------------------
Time: 2023-05-14 10:28:50
-------------------------------------------
('CB2105-CJM90', 0.8061405647098336)
('CEK049-CJM90', 0.7924212938820007)
('CB1599-CEK049', 0.7760058834442837)
('CB1599-CJM90', 0.7749920893036721)
('CB02411-CJM90', 0.768085551029472)

-------------------------------------------
Time: 2023-05-14 10:28:55
-------------------------------------------
('CB2105-CJM90', 0.8201521972161366)
('CEK049-CJM90', 0.8171214505913456)
('CB1599-CEK049', 0.7936716731847216)
('CB02411-CJM90', 0.7862192201366599)
('CB1599-CJM90', 0.7839198916478396)

-------------------------------------------
Time: 2023-05-14 10:29:00
-------------------------------------------
('CB2105-CJM90', 0.8201521972161366)
('CEK049-CJM90', 0.8171214505913456)
('CB1599-CEK049', 0.7936716731847216)
('CB02411-CJM90', 0.7862192201366599)
('CB1599-CJM90', 0.7839198916478396)
```

Table 3 - Top 5 most correlated pairs of sensors in sliding window context

The experimental results obtained by varying first the window length and then the sliding interval are represented on Figure 2. It can be seen that the computation time grows slower than the data size indicating good scalability. Moreover, the more cores are added, the best the computation time is for bigger batch sizes. Finally, Figure 6 and Figure 7 in the appendix show a representation of the parallelisation for a batch coming from the Spark interface.

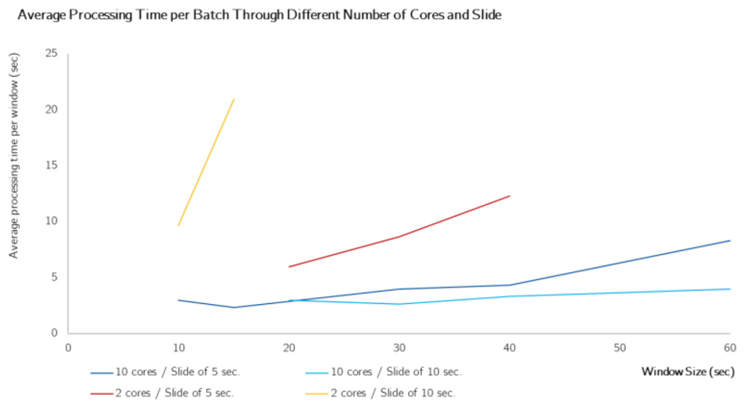


Figure 2 - Experimental results in the window processing context when varying the batch size, number of cores and sliding interval

4. Appendix

4.1. Batch processing task

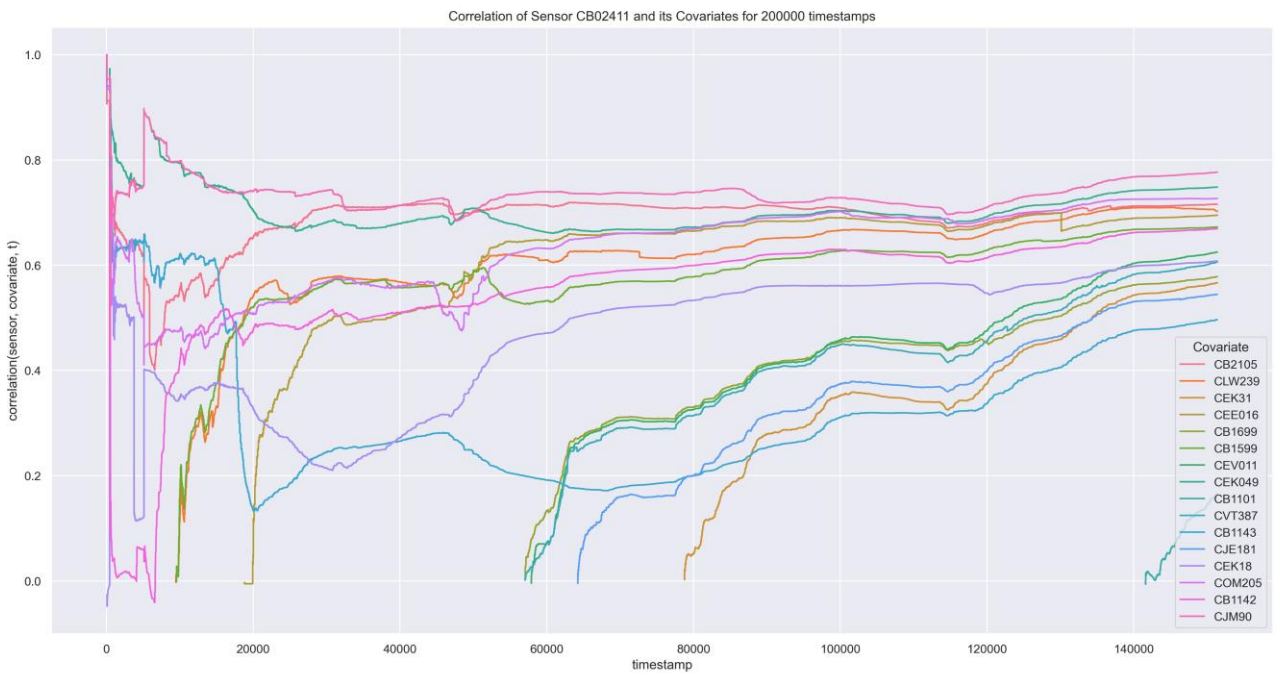
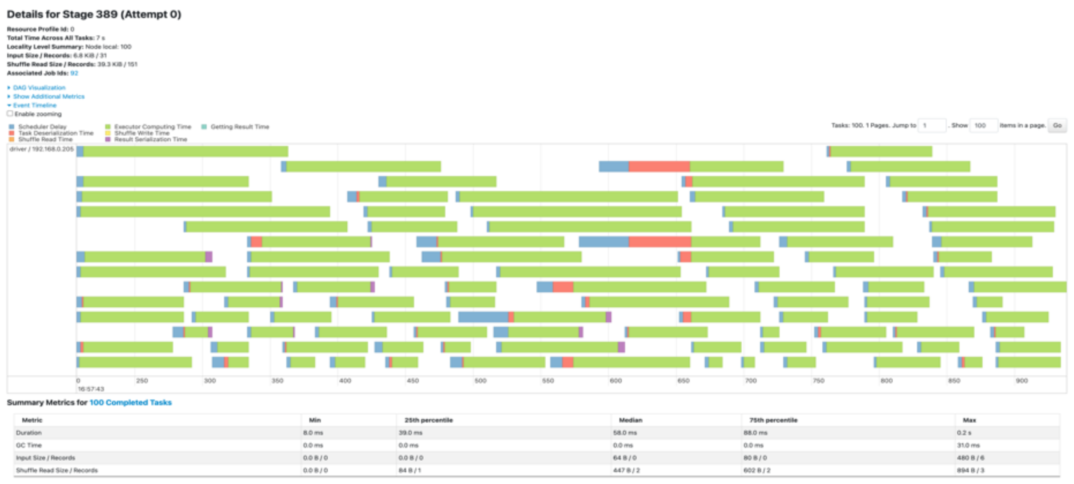


Figure 3: Correlation of sensor CB02411 and its covariate for 200 000 timestamps

4.2. Stream processing task



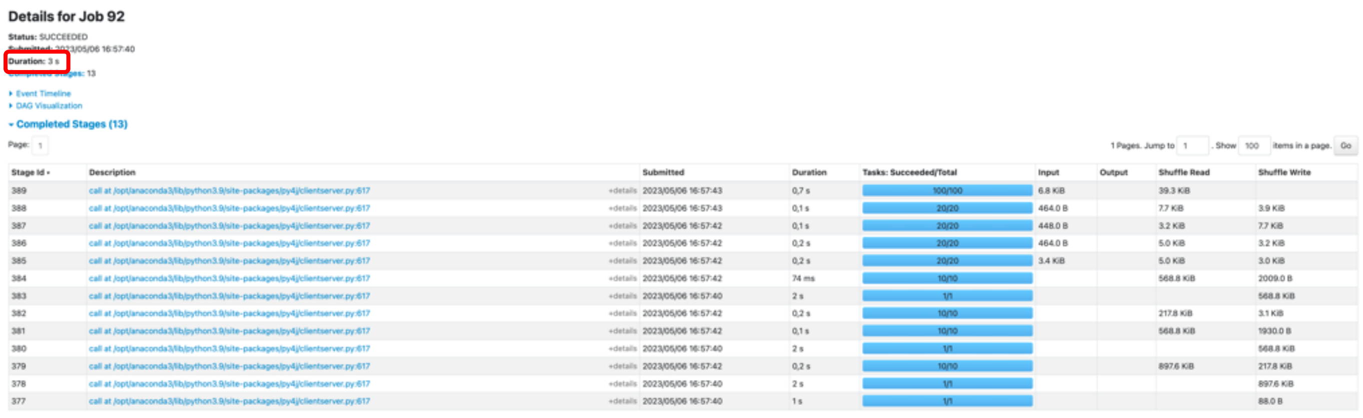
Details for Stage 389 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 7 s
Locality Level Summary: Node local: 100
Input Size / Records: 6.8 KiB / 31
Shuffle Read Size / Records: 39.3 KiB / 151
Associated Job Ids: 92

Summary Metrics for 100 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	8.0 ms	39.0 ms	58.0 ms	88.0 ms	0.2 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	31.0 ms
Input Size / Records	0.0 B / 0	0.0 B / 0	64 B / 0	80 B / 0	480 B / 6
Shuffle Read Size / Records	0.0 B / 0	84 B / 1	447 B / 2	602 B / 2	894 B / 3

Figure 4: Parallelization of a stage performed in the streaming context



Details for Job 92

Status: SUCCEEDED
Submitted: 2023/05/06 16:57:40
Duration: 3 s
Completed Stages: 13

- Event Timeline
- DAG Visualization

Completed Stages (13)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
389	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:43	0,7 s	100/100	6.8 KiB		39.3 KiB	
388	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:43	0,1 s	20/20	464.0 B		7.7 KiB	3.9 KiB
387	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,1 s	20/20	448.0 B		3.2 KiB	7.7 KiB
386	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,2 s	20/20	464.0 B		5.0 KiB	3.2 KiB
385	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,2 s	20/20	3.4 KiB		5.0 KiB	3.0 KiB
384	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	74 ms	10/10			568.8 KiB	2009.0 B
383	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:40	2 s	1/1				568.8 KiB
382	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,2 s	10/10			217.8 KiB	3.1 KiB
381	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,1 s	10/10			568.8 KiB	1930.0 B
380	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:40	2 s	1/1				568.8 KiB
379	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:42	0,2 s	10/10			897.6 KiB	217.8 KiB
378	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:40	2 s	1/1				897.6 KiB
377	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 16:57:40	1 s	1/1				88.0 B

Figure 5: Details and computing time of a job performed in the streaming context.

Number of cores used	Time Period (Days)	Batch i (s)	Batch j (s)	Batch k (s)	Average time (s)
10	10	3	2	2	2,33
10	20	3	3	3	3,00
10	30	3	3	4	3,33
10	40	4	4	4	4,00
10	50	6	6	6	6,00
10	60	7	7	8	7,33
2	10	2	2	2	2
2	20	3	4	3	3,33
2	30	6	6	6	6
2	40	7	7	7	7
2	50	14	11	9	11,33
2	60	67	52	26	48,33

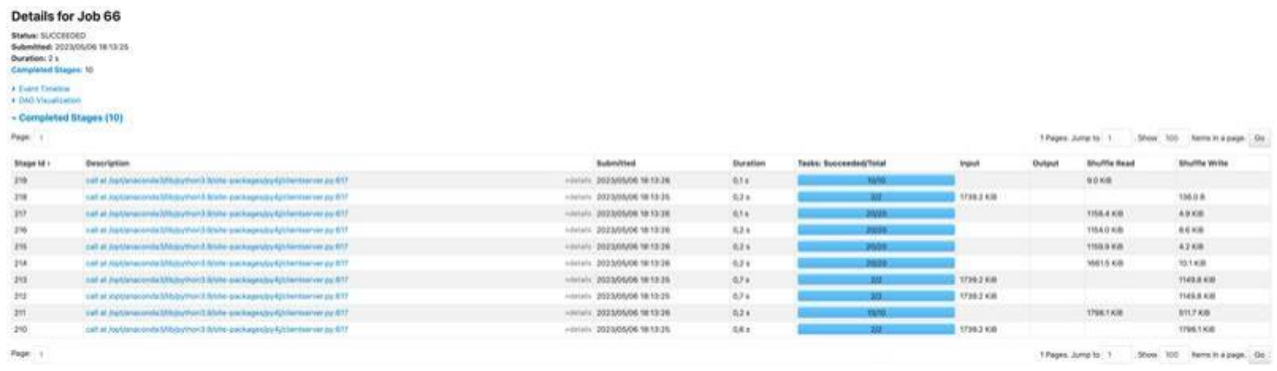
Table 3 - Experimental result when varying the batch size (Π) and number of cores.

4.3. Sliding Window

W(sec)	Slide(sec)	Time Period (days)	Batch i (s)	Batch j (s)	Batch k (s)
10	5	30	3	3	3
15	5	30	2	2	3
30	5	30	4	4	4
40	5	30	4	4	5
60	5	30	8	8	9
20	10	30	3	3	3
30	10	30	2	3	3
40	10	30	3	4	3
60	10	30	4	4	4

Table 5: Experimental results in the window processing context.

`



Details for Job 66

Status: SUCCEEDED
Submitted: 2023/05/06 18:13:25
Duration: 2 s
Completed Stages: 10

- Event Timeline
- DAG Visualization

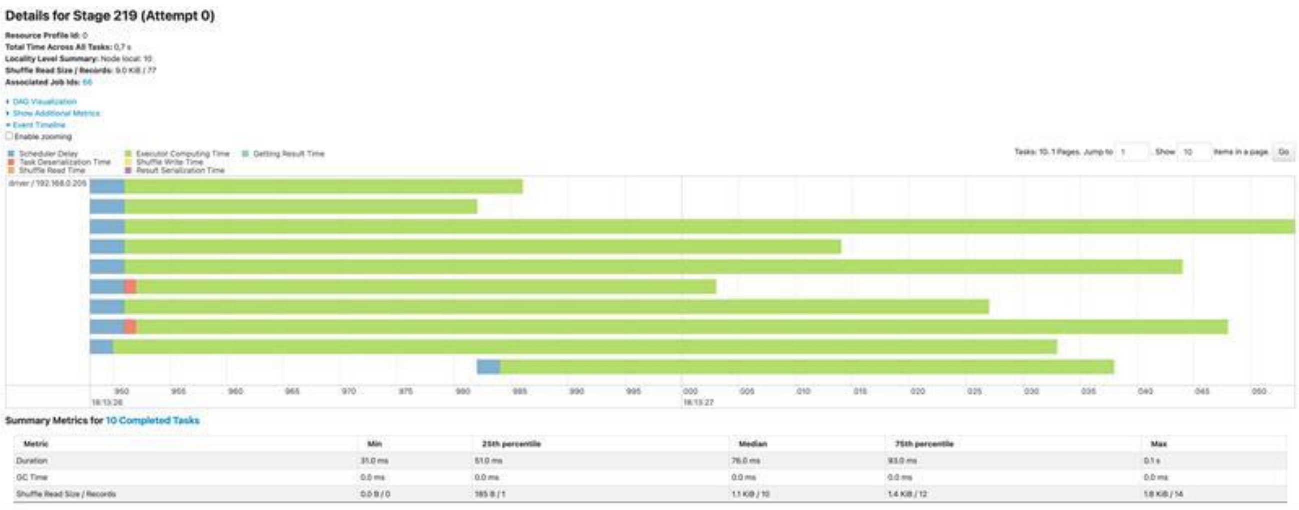
- Completed Stages (10)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
219	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,1 s	10/10			9.0 KiB	
218	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:25	0,2 s	2/2	1738.2 KiB			136.0 B
217	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,1 s	20/20			1158.4 KiB	4.9 KiB
216	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,2 s	20/20			1154.0 KiB	8.6 KiB
215	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,2 s	20/20			1159.9 KiB	4.2 KiB
214	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,2 s	20/20			1661.5 KiB	10.1 KiB
213	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:25	0,7 s	2/2	1738.2 KiB			1149.8 KiB
212	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:25	0,7 s	2/2	1738.2 KiB			1149.8 KiB
211	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:26	0,2 s	10/10			1798.1 KiB	511.7 KiB
210	call at /opt/anaconda3/lib/python3.9/site-packages/py4j/clientserver.py:617	+details	2023/05/06 18:13:25	0,6 s	2/2	1738.2 KiB			1796.1 KiB

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Figure 6: Details and computing time for a job in the window processing context



Details for Stage 219 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0,7 s
Locality Level Summary: Node local: 10
Shuffle Read Size / Records: 9.0 KiB / 77
Associated Job Ids: 66

- DAG Visualization
- Show Additional Metrics
- Event Timeline

☐ Enable zooming

Tasks: 10. 1 Pages. Jump to 1 . Show 10 items in a page. Go

Summary Metrics for 10 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	31.0 ms	51.0 ms	76.0 ms	93.0 ms	0.1 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Shuffle Read Size / Records	0.0 B / 0	185 B / 1	1.1 KiB / 10	1.4 KiB / 12	1.8 KiB / 14

Figure 7: Parallelization in the window processing context