DATA+AI SUMMIT
BY databricks

RAPID PYSPARK IMPLEMENTATION ON TIME SERIES BIG DATA

Megha Rajam Rao | Gary Garcia Molina
June 12th, 2024

DATA+AI SUMMIT
BY databricks

Rapid Pyspark custom processing on time series Big data in Databricks

Powered by

Data Science and Machine Learning (Advanced)
Breakout Session

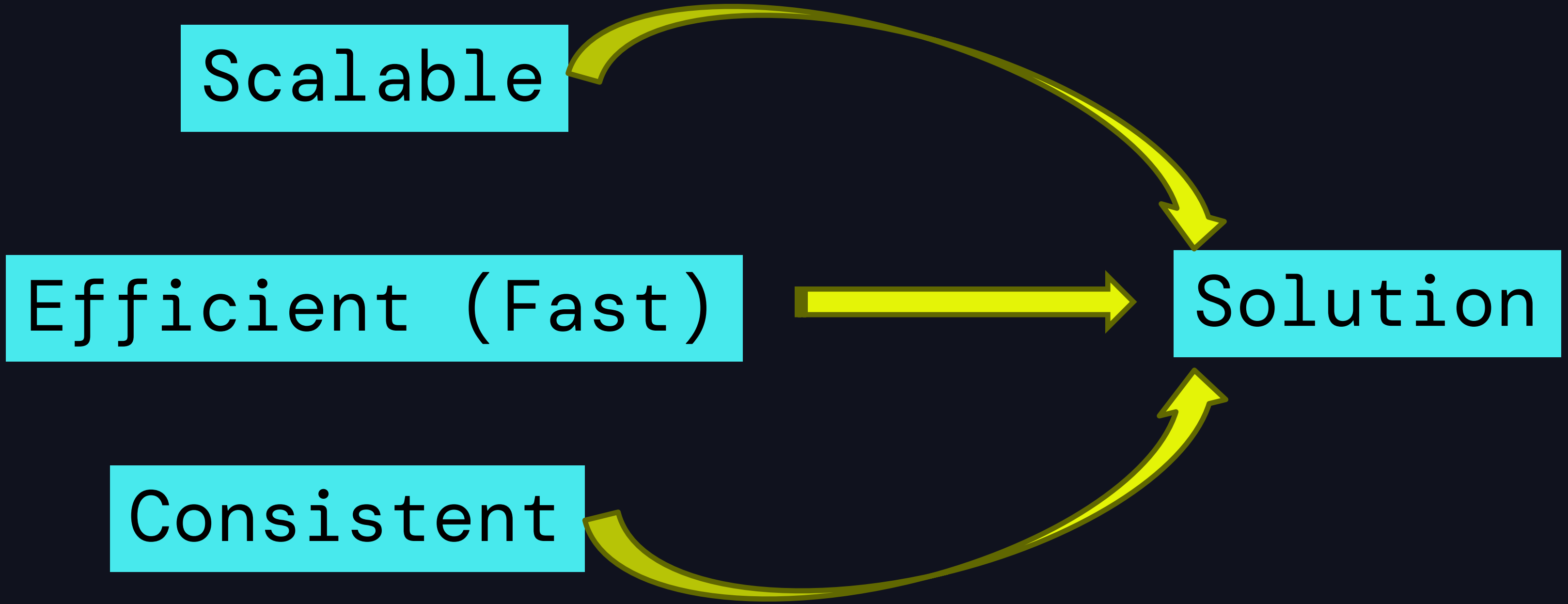
Powered by

sleep number

databricks

Is Big data processing time consuming?

DATA+AI SUMMIT



Agenda

- Introduction
- Dataset
- Clusters
- Methods
- Results
- Conclusion

INTRODUCTION

OVERVIEW

A brief preview

BACKGROUND

- **Goal:** To quantify weight changes and their association with sleep using Sleep Number Smart beds equipped with force sensors.
- **Problem:** Raw readings were noisy due to user movements necessitating denoising by cleaning each rolling window of the big data.

METHOD

- **Methodology:** Entropy measure calculated using Pandas and Pyspark implementations were utilized to clean and denoise the dataset.
- **Experimentation:** Different configurations of single and multi-node clusters in Databricks were tested on datasets with 10 to 50 million datapoints for optimal performance evaluation.

SOLUTION

- **Result:** The recommended Pyspark method rapidly processed 50 million records in nearly 0.3 seconds in Databricks.
- **Inference:** It performed complex custom calculation on rolling windows of time series big data in constant time complexity irrespective of data size.

Background

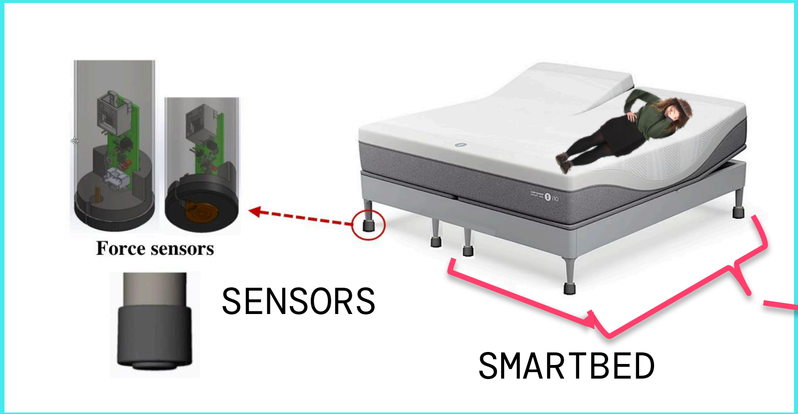
Model

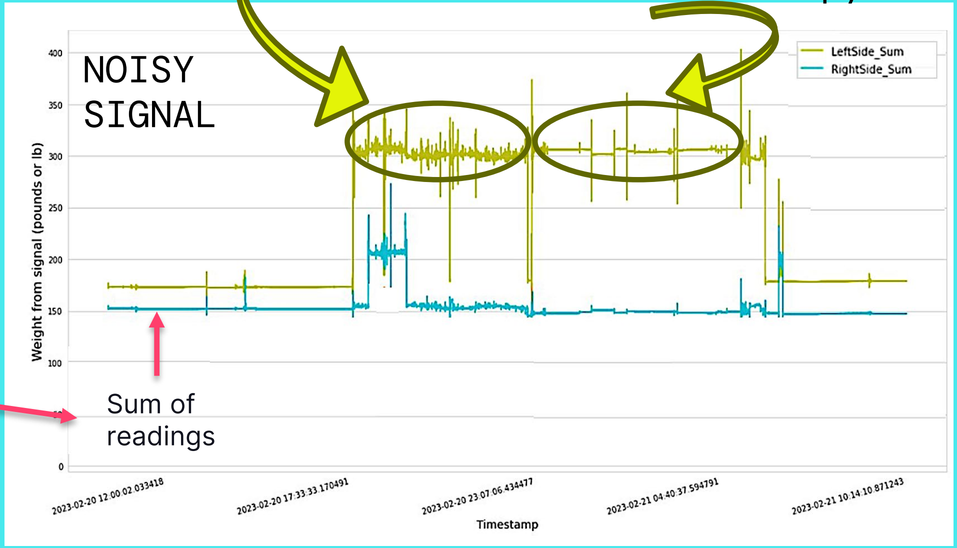
High quality data

High entropy

Low entropy

Suppose subject sleeps on left side





Entropy => Quantifies randomness, disorder or uncertainty
Time complexity of the entropy-calculation algorithm in literature is O(N^2) (Liu et al., 2022)

CHALLENGES

Problem Overview

GENERAL PROBLEM

- The goal was to quantify weight changes & their association with sleep. However, the sensors under the Smart bed send overnight force signals which are inherently noisy due to user movements & position in bed.
- Due to fluctuations in readings on each side of bed, an intricate quality assessment needed to be performed to select stable data segments characterized by low entropy.

TECHNICAL PROBLEM

- For calculating the entropy at a granular level, a custom formula had to be applied to slices of high-resolution time series big data at 1 Hz with 10s of millions of records.
- The initial implementation using Pandas did not suffice due to memory and time constraints.

TERMINOLOGY

Overview & General use

Pandas

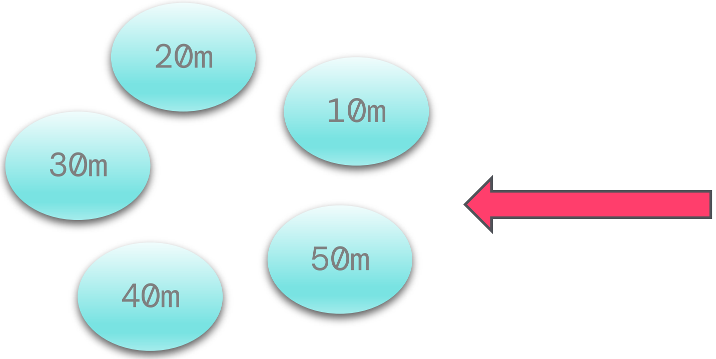
- Pandas is a powerful and flexible python library commonly used for data analysis, manipulation and machine learning tasks.
- It is open source and built on top of the Python programming language.

Pyspark

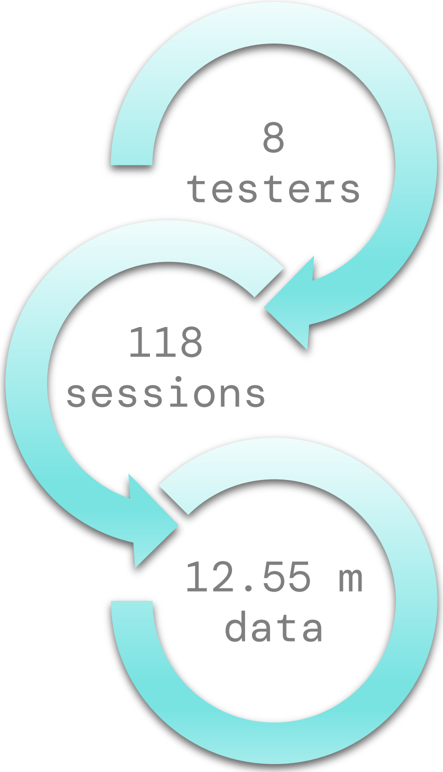
- PySpark is the Python API gateway to Apache Spark for data processing, analysis and machine learning tasks.
- It enables real-time large-scale data processing in a distributed environment using the Python programming language.

DATASET

DATASET



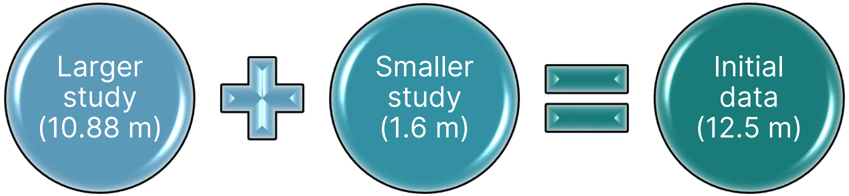
To test scalability & consistency



50, 40, 30, 20 and 10 million datasets generated from a combination of synthetic and actual datapoints from 12.55 million data rows originating from 118 overnight sessions with 8 Testers.

DATA CURATION

(1) INITIAL DATA

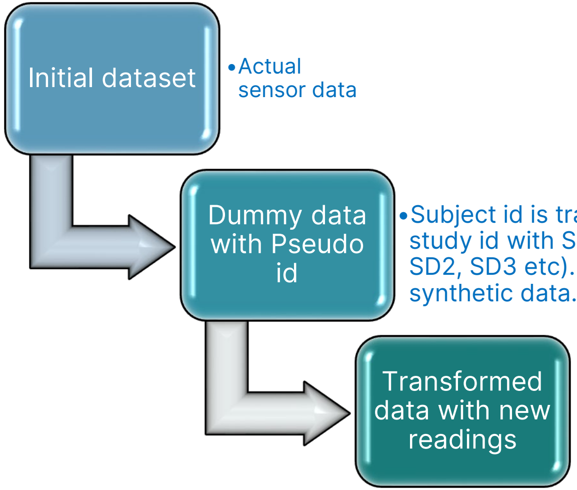


5 testers
102 sessions

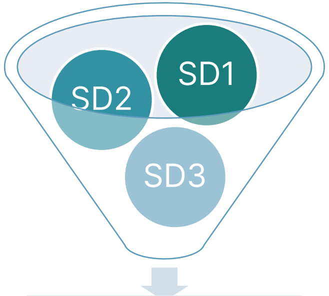
3 testers
16 sessions

8 testers
118 sessions

(2) SYNTHETIC DATA



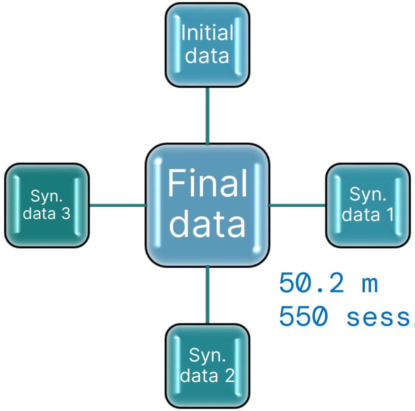
- Actual sensor data
- Subject id is transformed by replacing study id with SDn (for example, - SD1, SD2, SD3 etc). Here SD stands for synthetic data.
- A constant value (multiple of 25 for each dataset) is added to all readings



3 synthetic datasets

3 iterations (with unique pseudo ids and unique constants)

(3) FINAL DATA



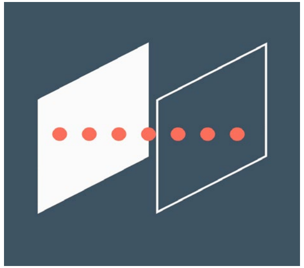
50.2 m
550 sessions

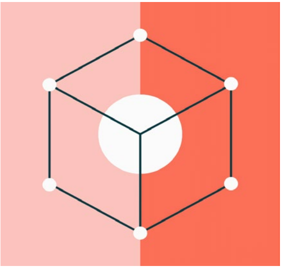
Merge all datasets to multiply the initial data by 4 times

CLUSTERS


Exp1 - Storage Optimized (Active memory: 256 GB; Active cores: 32)
1
Cluster 1
_Single_node_temp_1
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
i4i.8xlarge • 256 GB • 32 Cores
2
Cluster 2
_Multi_Node_temp_1
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
i4i.2xlarge • 64 GB • 8 Cores
Workers (3)
i4i.2xlarge • 192 GB • 24 Cores
Exp2 - Memory Optimized (Active memory: 256 GB; Active cores: 32)
3
Cluster 3
_Single_node_temp_2
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
r5d.8xlarge • 256 GB • 32 Cores
4
Cluster 4
_Multi_Node_temp_2
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
r5d.2xlarge • 64 GB • 8 Cores
Workers (3)
r5d.2xlarge • 192 GB • 24 Cores
Exp3 - General Purpose (Active memory: 256 GB; Active cores: 64)
5
Cluster 5
_Single_node_temp_3
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
m6g.16xlarge • 256 GB • 64 Cores
6
Cluster 6
_Multi_Node_temp_3
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
m5d.4xlarge • 64 GB • 16 Cores
Workers (3)
m5d.4xlarge • 192 GB • 48 Cores
Exp4 - Compute Optimized (Active memory: 256 GB; Active cores: 128)
7
Cluster 7
_Single_node_temp_4
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
c6id.32xlarge • 256 GB • 128 Cores
8
Cluster 8
_Multi_Node_temp_4
Runtime
DBR 14.3 LTS • Spark 3.5.0 • Scala 2.12
Driver
c6gd.8xlarge • 64 GB • 32 Cores
Workers (3)
c6gd.8xlarge • 192 GB • 96 Cores

METHODS





DATA & LOGIC

- Accessed 10 to 50 million distinct datasets at 10 million increments from delta lake.
- Applied user-defined function (udf) to calculate entropy to 30-sec rolling windows with 1 second shift of the signal for each sleeper & session.

ALGORITHM

- For algo design, examined the efficient & brute force implementations of Pandas & Pyspark libraries.
- Evaluated both single & multi-node clusters of varying configurations based on total time taken.

LIMITATIONS

- Unable to evaluated the Pandas implementation beyond 40 m as it keeps failing at failed at 50m.
- The Pandas did not suffice due to memory & time constraints until the operation was augmented using Pyspark.

Total number of experiments

	PANDAS	PYSPARK
Dataset	4	5
Methods	2	2
Cluster configs	8	8

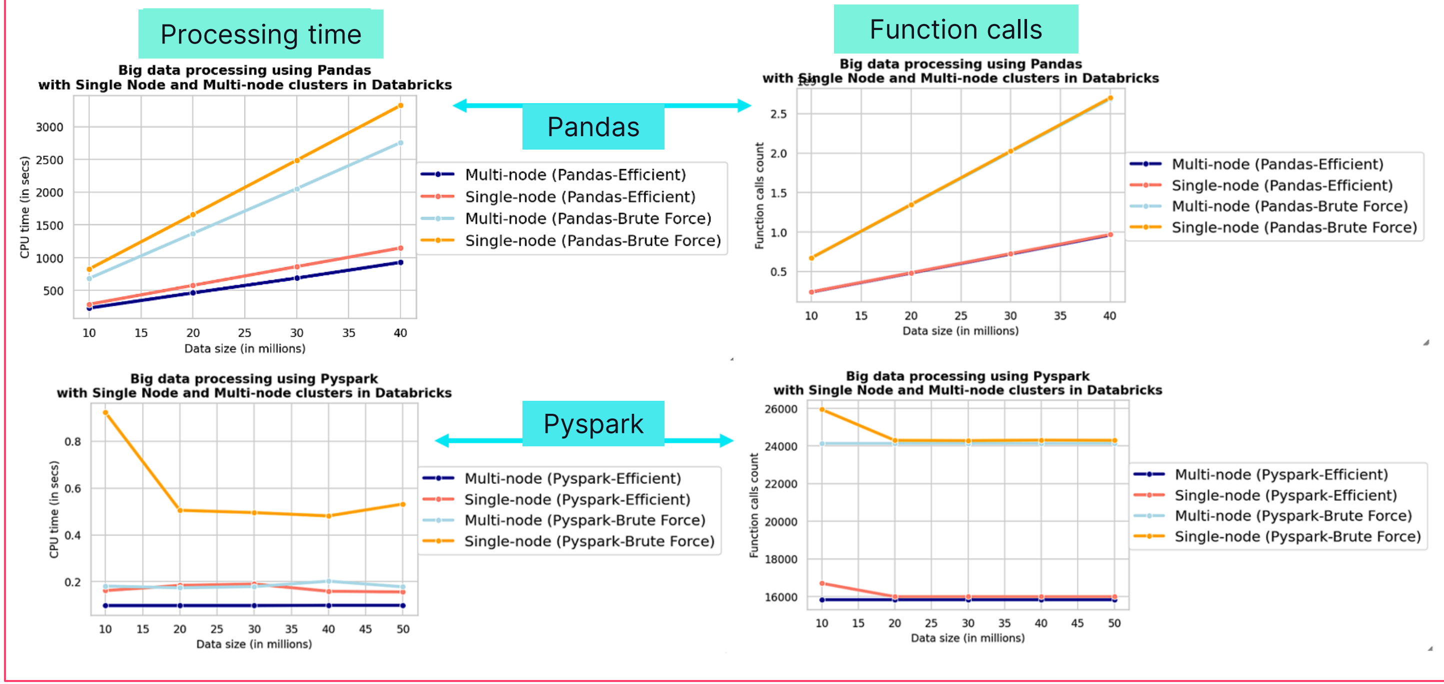
64 + 80

144 experiments

RESULTS

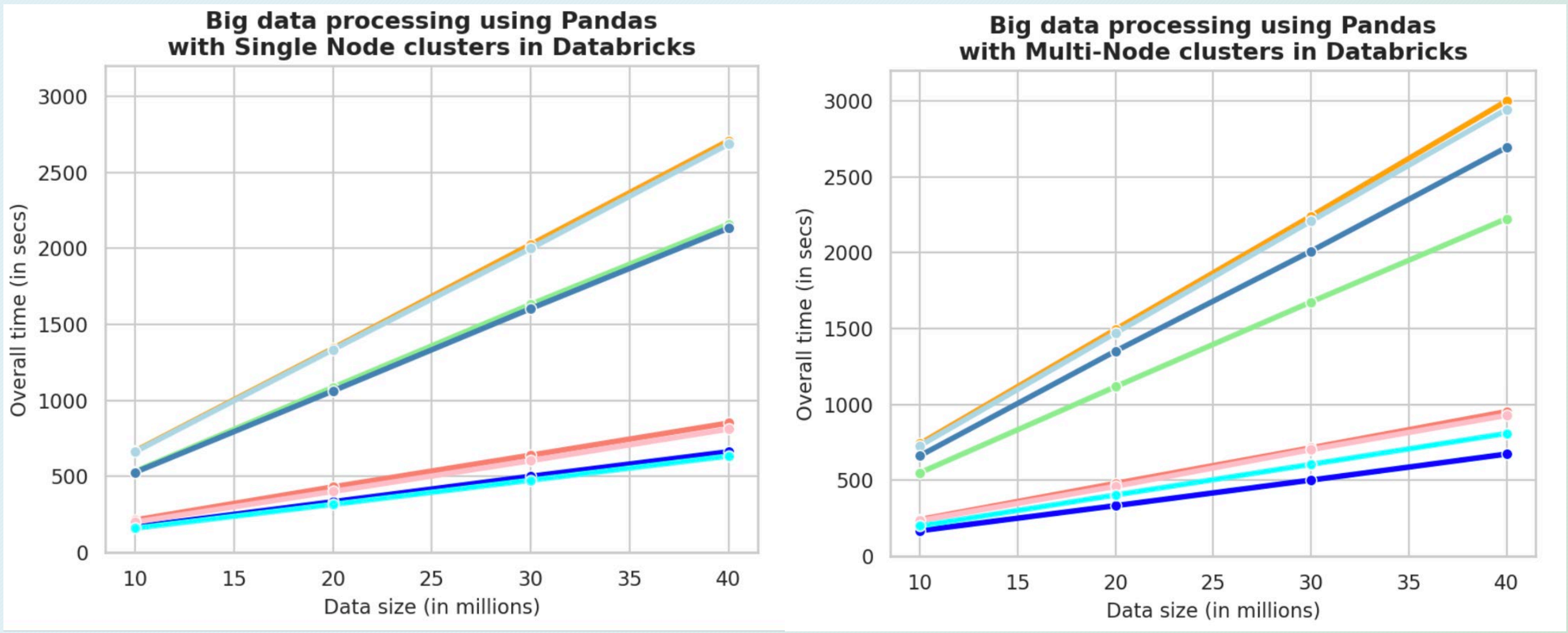
RESULTS

Pandas vs. Pyspark

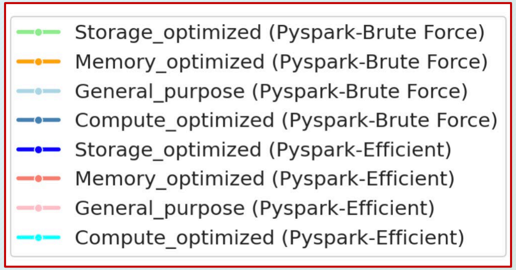


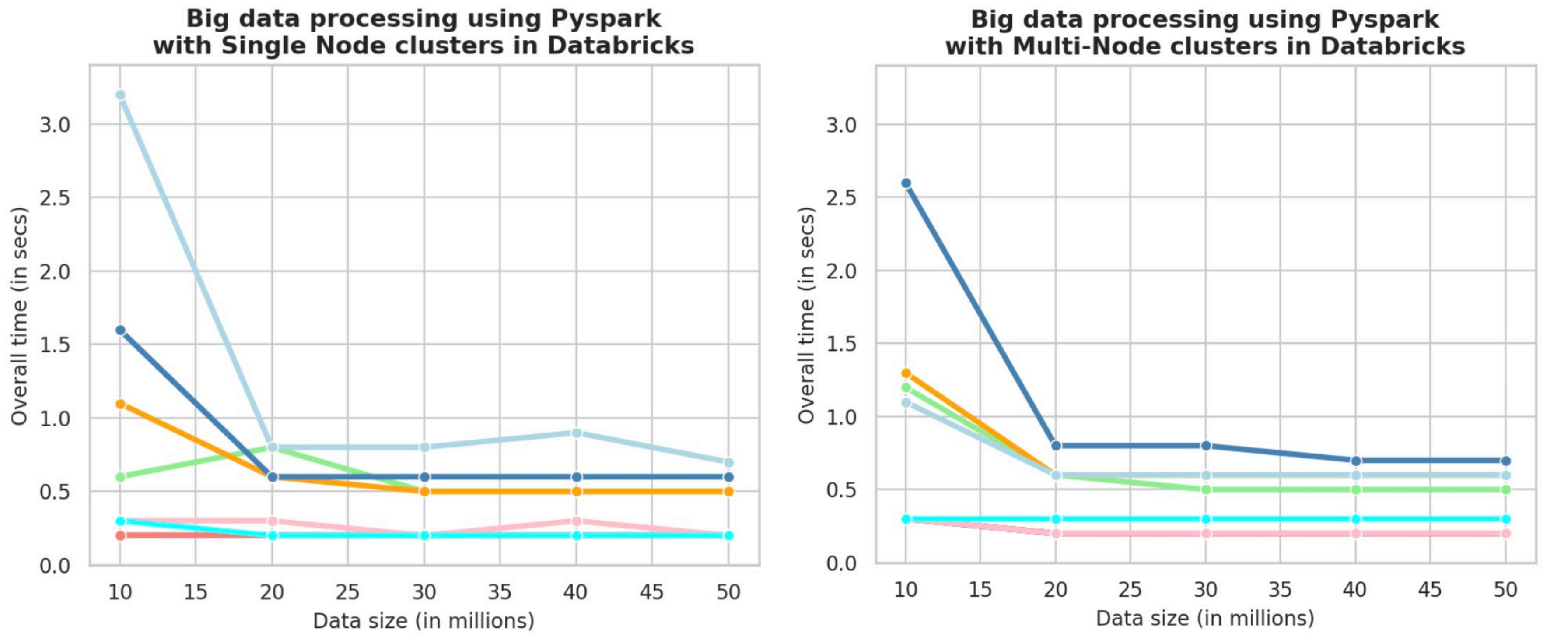
- Pyspark efficient method has constant time complexity O(1) due to constant total time & function calls irrespective of data size.
- The Pyspark operation generates custom windows based on the criterion defined and applies the Pandas function to data in each individual window.
- It performed complex rolling calculation on 50 million records in less than 0.3 seconds with both single & multi-node Databricks clusters.

Pandas Total time:
Single node & Multi-node clusters

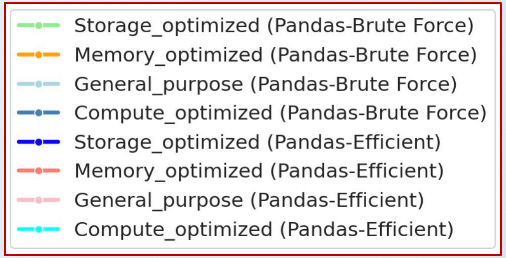


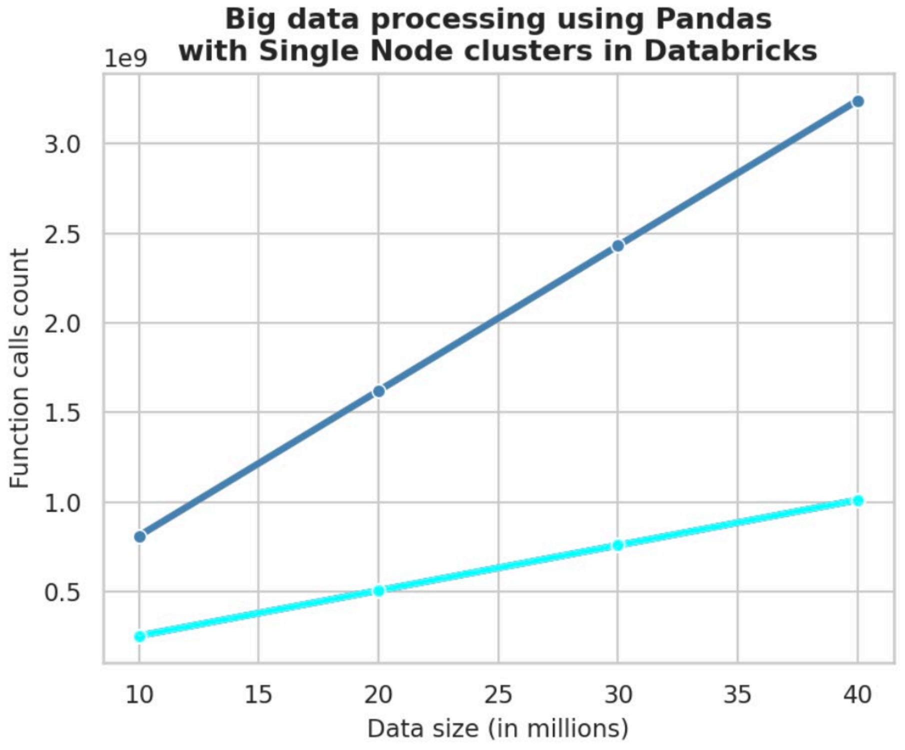
Pyspark Total time:
Single node & Multi-node clusters

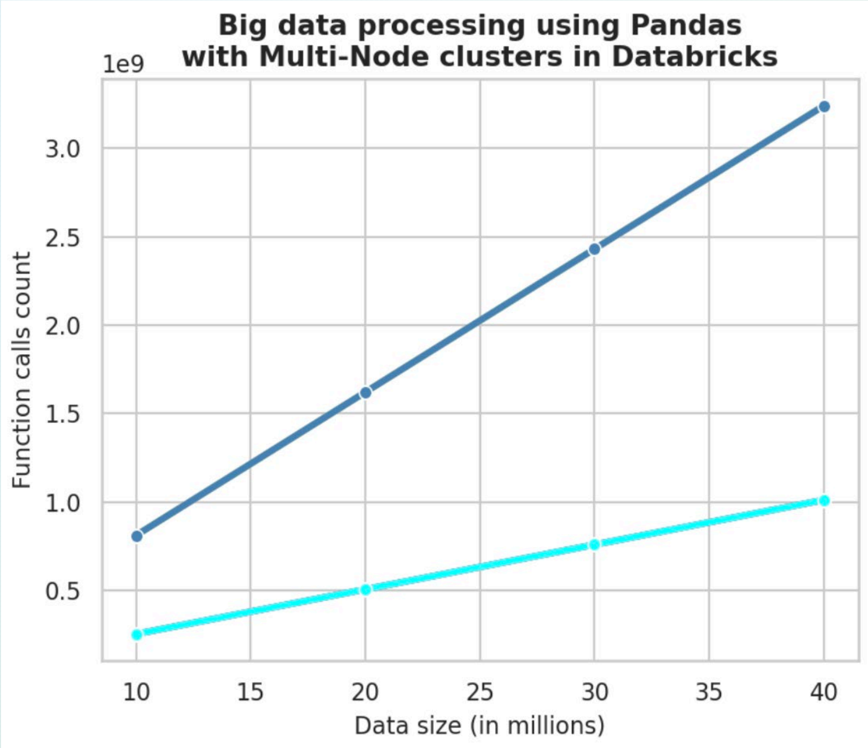




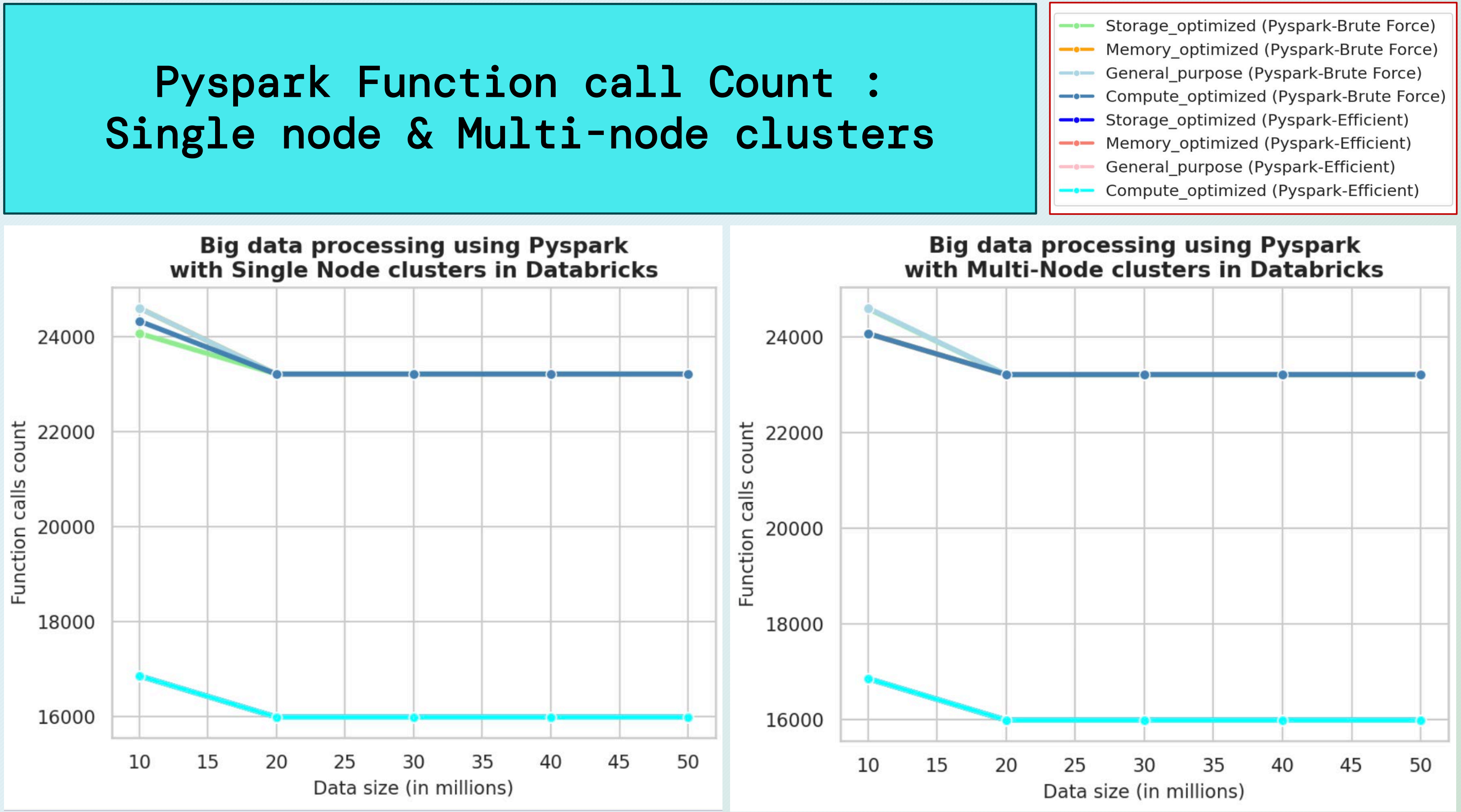
Pandas Function call Count: Single node & Multi-node clusters







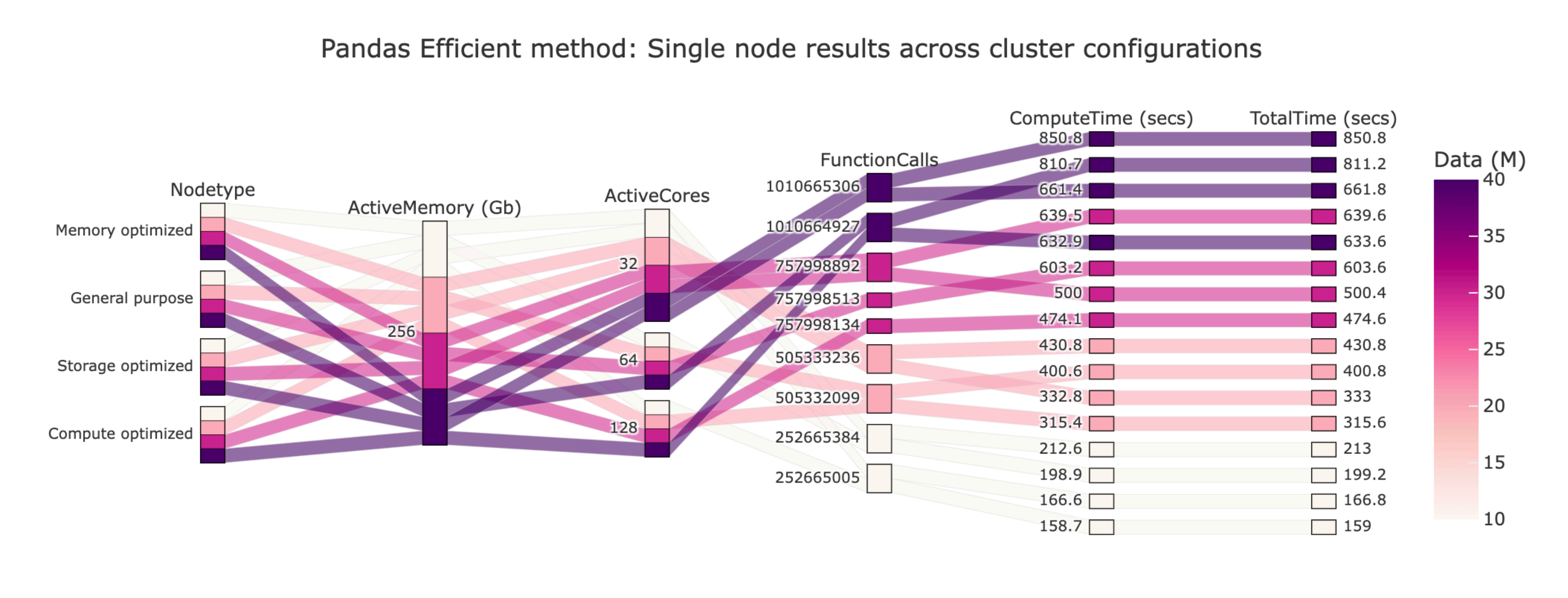
Pyspark Function call Count : Single node & Multi-node clusters



CONSISTENCY & SCALABILITY

Pandas
Single-node

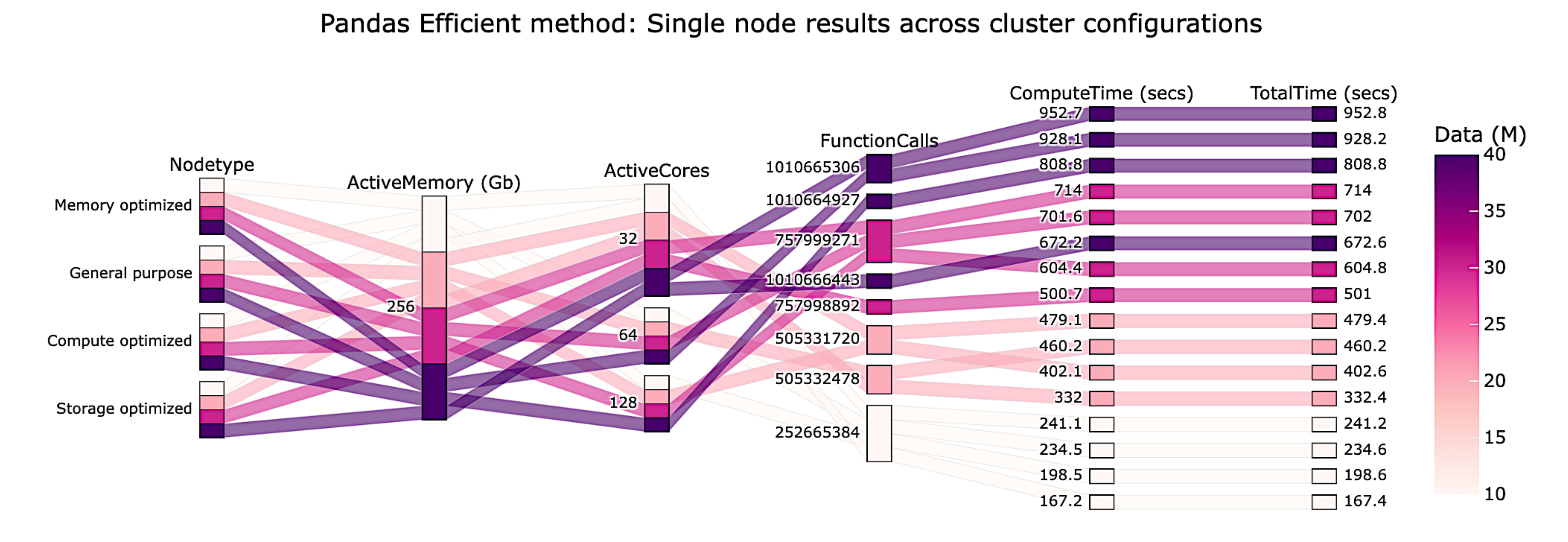
Single-node evaluation of the Efficient Pandas implementation



CONSISTENCY & SCALABILITY

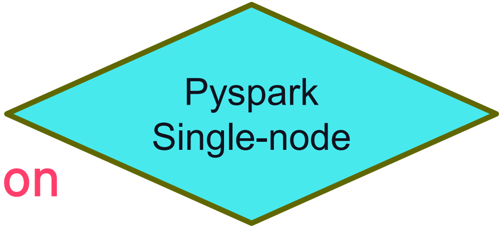
Multi-node evaluation of the Efficient Pandas implementation

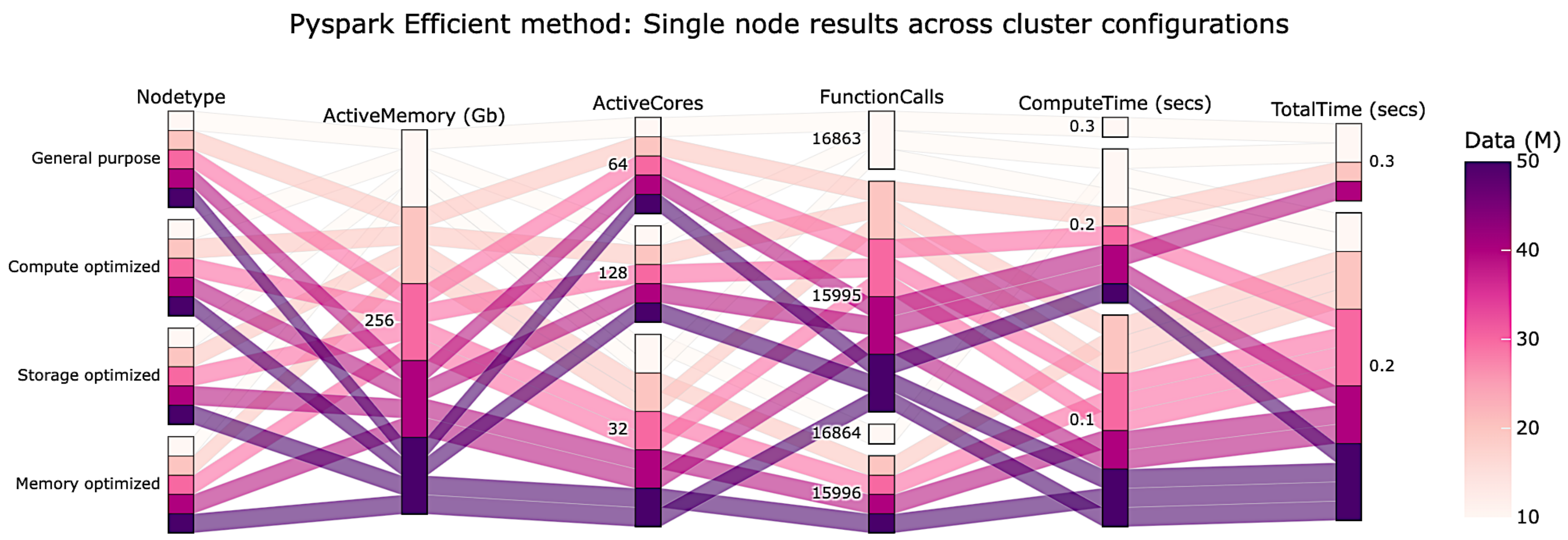
Pandas Multi-node



CONSISTENCY & SCALABILITY

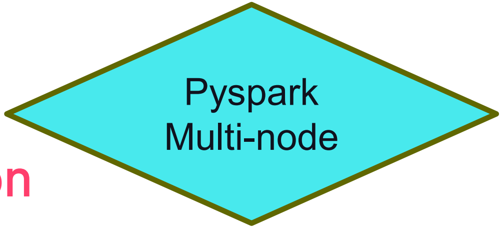
Single-node evaluation of Efficient Pyspark implementation

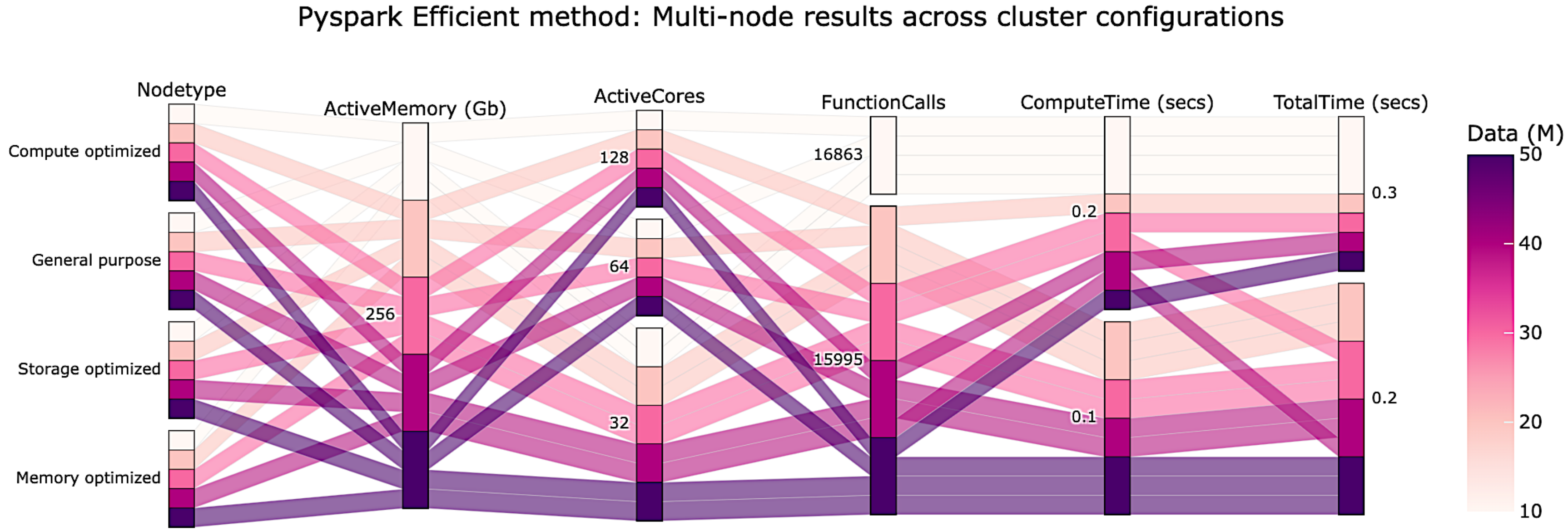




CONSISTENCY & SCALABILITY

Multi-node evaluation of Efficient Pyspark implementation



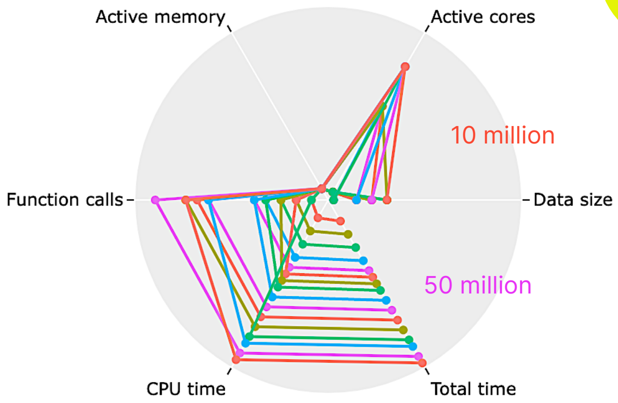


Solution evaluation

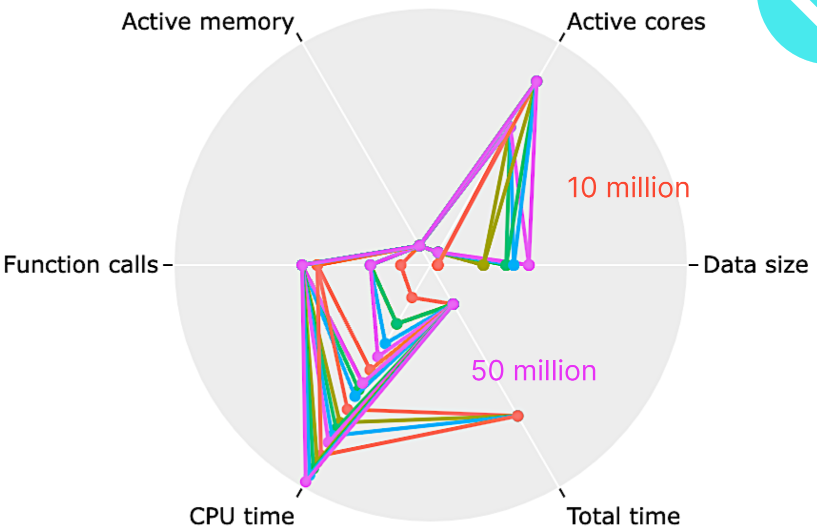
Using Single-node clusters

SingleN_Storage_optimized_10m
SingleN_Storage_optimized_20m
SingleN_Storage_optimized_30m
SingleN_Storage_optimized_40m
SingleN_Storage_optimized_50m
SingleN_Memory_optimized_10m
SingleN_Memory_optimized_20m
SingleN_Memory_optimized_30m
SingleN_Memory_optimized_40m
SingleN_Memory_optimized_50m
SingleN_General_purpose_10m
SingleN_General_purpose_20m
SingleN_General_purpose_30m
SingleN_General_purpose_40m
SingleN_General_purpose_50m
SingleN_Compute_optimized_10m
SingleN_Compute_optimized_20m
SingleN_Compute_optimized_30m
SingleN_Compute_optimized_40m
SingleN_Compute_optimized_50m

Pandas



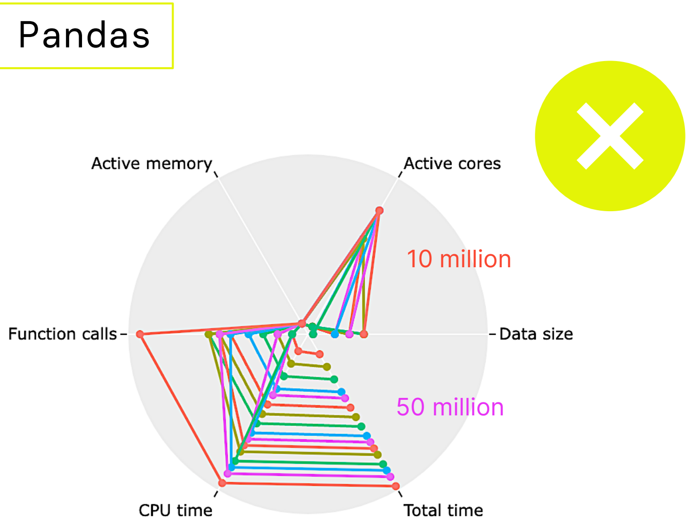
Pyspark

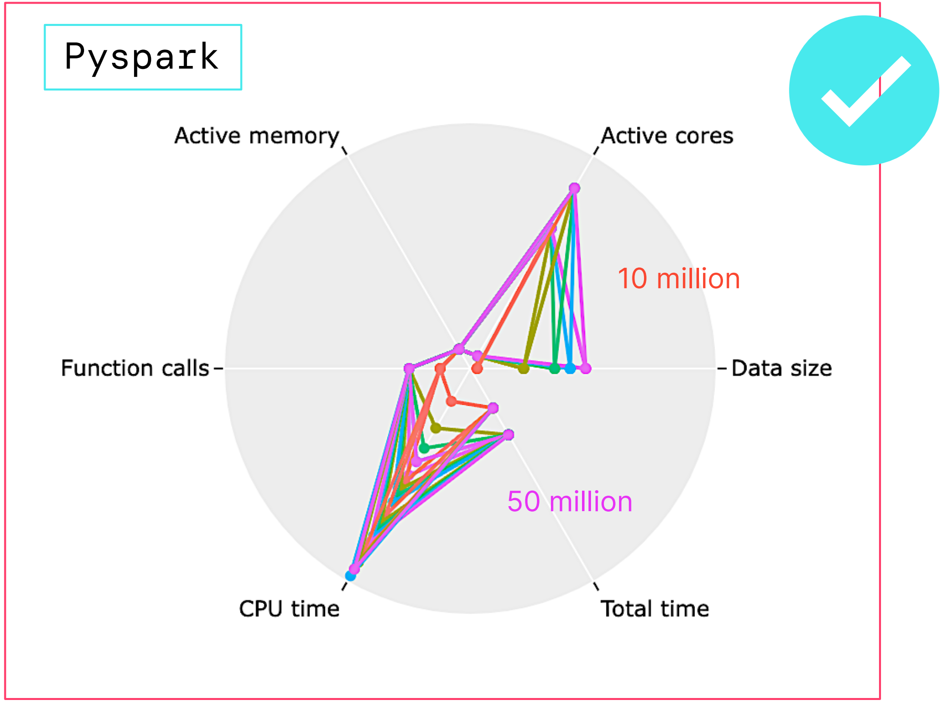


Solution evaluation

Using Multi-node clusters

MultiN_Storage_optimized_10m
MultiN_Storage_optimized_20m
MultiN_Storage_optimized_30m
MultiN_Storage_optimized_40m
MultiN_Storage_optimized_50m
MultiN_Memory_optimized_10m
MultiN_Memory_optimized_20m
MultiN_Memory_optimized_30m
MultiN_Memory_optimized_40m
MultiN_Memory_optimized_50m
MultiN_General_purpose_10m
MultiN_General_purpose_20m
MultiN_General_purpose_30m
MultiN_General_purpose_40m
MultiN_General_purpose_50m
MultiN_Compute_optimized_10m
MultiN_Compute_optimized_20m
MultiN_Compute_optimized_30m
MultiN_Compute_optimized_40m
MultiN_Compute_optimized_50m





DEMO

Multi-node experimentation with assorted-sized Big data...

File Edit View Run Help Last edit was now Provide feedback

Run all

Click on Play > button

Share

(2) Pyspark efficient method

43

```
# Define the main function
def pyspark_method2(pyspark_df):
    # Step 1 - Define the window
    # **Proprietary code**

    # Step 2 - Create a user-defined function
    # **Proprietary code**

    # Step 3 - Apply window and built-in function
    # **Proprietary code**

    # Step 4 - (Optional) Display results
    # **Proprietary code**
```

01:22 PM (2m) 44 Python

```
# Check the data size
print('Data size for the assorted datasets are as follows-\n',\
    pyspark_df_10m.count(),\
        pyspark_df_20m.count(),\
            pyspark_df_30m.count(),\
                pyspark_df_40m.count(),\
                    pyspark_df_50m.count())
```

(15) Spark Jobs

```
Data size for the assorted datasets are as follows-
 10000000 20000000 30000000 40000000 50000000
```

Clear winner = PYSPARK EFFICIENT METHOD

Within
0.2 - 0.3 seconds

Total time taken for 10 to 50 million records

CONCLUSION

DISCUSSION

Technical insights

TECHNICAL DETAILS

- As Pandas failed to process beyond 40 m datapoints, the results at the same size & varying cluster configuration were juxtaposed to compare the processing speed, execution time & function calls to nominate the best method.
- The efficient Pyspark method had a constant time complexity O(1) & static number of function calls. It executed in mere 0.2 to 0.3 seconds.
- In comparison to the Pandas brute force method with O(n) complexity and approximately 45-minute execution time, the top method was four orders of magnitude times faster.
- Similarly, when compared to the efficient Pandas method with O(log(n)) complexity and approximately 15-minute execution time, the top method was 3 orders of magnitude times faster.
- Finally, the top method was twice to thrice as fast as the Pyspark brute-force method with quasi-O(1) complexity and 0.3 to 0.6 second execution time with single and multi-node clusters, respectively.

RECOMMENDATIONS

Based on Pandas vs. Pyspark comparison

Use the Databricks platform to leverage the computing capabilities offered by multitude of cluster configurations

PANDAS

- Ideal for small datasets below 0.5 million datapoints.
- Works well on a single machine.
- Easier to implement with lower learning curve due to simple API & syntax.

Disclaimer
Shared analysis applies to entropy calculation. Other algorithms may require different considerations.

PYSPARK

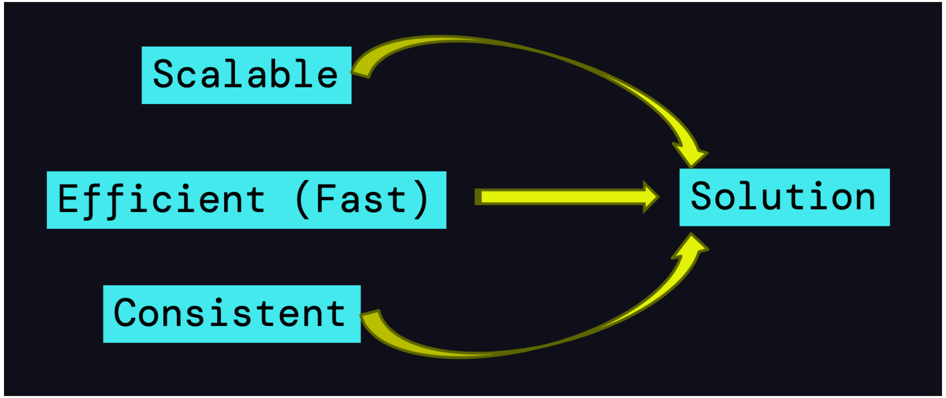
- Ideal for larger datasets above 0.5 million datapoints.
- Works well with distributed processing across clusters.
- Utilizes python's learnability to leverage the powerful capabilities of Apache Spark.

Key takeaway
Gain in performance can be observed for big data algorithms that can be parallelized.

CONCLUSION

Based on 144+ experiments

- The recommended efficient Pyspark method to calculate entropy ensures constant O(1) time complexity.
- The solution is efficient (fast), scalable and consistent as promised.
- The potent synergy of Pyspark Databricks can enable accelerated processing of big data.
- It can perform complex time series rolling window operations using the entropy custom function in less than a second in Databricks.



- Pandas is easy to use & ideal for smaller datasets below 0.5 million
- PySpark is ideal for larger datasets with distributed processing across clusters

Team roles

Name	Role	Responsibilities
Gary Garcia Molina	Long-term Research team leader	Vision, guidance & Ideas
Megha Rajam Rao	Weight research lead	Algorithm design, in-lab & in-home study design and coordination, data collection, protocols, quality assessment & analysis
Dmytro Rizdvanetskyi	Data Architect	Peer review & Algorithm design assessment
Sai Ashrith Aduwala	Research contributor	In-lab data collection, study coordination & data pipeline
Suprit Bansod	Research contributor	Manual annotation for data quality & in-lab data analysis
Shawn Barr	Research contributor	In-lab data analysis & in-home mini-protocol analysis
Kashish Jain	Electrical engineer	Hardware setup & troubleshooting
Dmytro Guzenko	Reviewer	Algorithm Peer review

REFERENCES

- Liu, W., Jiang, Y., & Xu, Y. (2022, April 8). *A super fast algorithm for estimating sample entropy*. Entropy (Basel, Switzerland). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9027109/
- *pandas - Python Data Analysis Library*. (n.d.).
- *PySpark Overview — PySpark master documentation*. (n.d.). https://spark.apache.org/docs/latest/api/python/index.html

THANK YOU

Megha Rajam Rao
Research Scientist
Sleep Number

Gary Garcia Molina
Senior Principal Scientist
Sleep Number

Powered by

sleep number

databricks

Appendix

Best results (20 out of 72 Single node tests)

Pyspark-Efficient had superior performance compared to Pandas

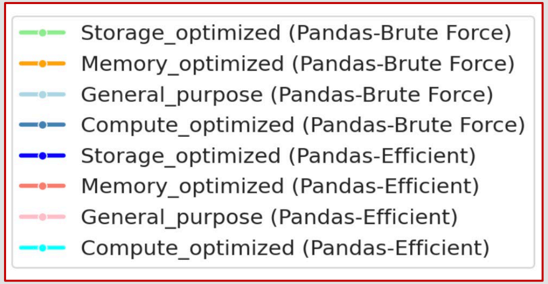
Cluster_No	Cluster_type	Node	Runtime	Node_type	Active memory_gb	Active cores	Method	Dataset_in_Millions	Function_calls	Primitive_calls	CPU_time_secs	Overall_time_secs	Overall_time_mins	Photon_acceleration
1	SingleN_Storage_optimized	Single	14.3 LTS	i4i.8xlarge	256	32	Pyspark-Efficient	20	15996	15951	0.116	0.2	0.00333333	No
1	SingleN_Storage_optimized	Single	14.3 LTS	i4i.8xlarge	256	32	Pyspark-Efficient	30	15996	15951	0.116	0.2	0.00333333	No
1	SingleN_Storage_optimized	Single	14.3 LTS	i4i.8xlarge	256	32	Pyspark-Efficient	50	15996	15951	0.117	0.2	0.00333333	No
1	SingleN_Storage_optimized	Single	14.3 LTS	i4i.8xlarge	256	32	Pyspark-Efficient	40	15996	15951	0.122	0.2	0.00333333	No
2	SingleN_Memory_optimized	Single	14.3 LTS	45d.8xlarge	256	32	Pyspark-Efficient	20	15995	15950	0.135	0.2	0.00333333	No
2	SingleN_Memory_optimized	Single	14.3 LTS	45d.8xlarge	256	32	Pyspark-Efficient	50	15995	15950	0.135	0.2	0.00333333	No
2	SingleN_Memory_optimized	Single	14.3 LTS	45d.8xlarge	256	32	Pyspark-Efficient	30	15995	15950	0.136	0.2	0.00333333	No
2	SingleN_Memory_optimized	Single	14.3 LTS	45d.8xlarge	256	32	Pyspark-Efficient	40	15995	15950	0.143	0.2	0.00333333	No
4	SingleN_Compute_optimized	Single	14.3 LTS	c6id.32xlarge	256	128	Pyspark-Efficient	50	15995	15950	0.144	0.2	0.00333333	No
4	SingleN_Compute_optimized	Single	14.3 LTS	c6id.32xlarge	256	128	Pyspark-Efficient	20	15995	15950	0.145	0.2	0.00333333	No
4	SingleN_Compute_optimized	Single	14.3 LTS	c6id.32xlarge	256	128	Pyspark-Efficient	30	15995	15950	0.147	0.2	0.00333333	No
1	SingleN_Storage_optimized	Single	14.3 LTS	i4i.8xlarge	256	32	Pyspark-Efficient	10	16864	16819	0.154	0.2	0.00333333	No
4	SingleN_Compute_optimized	Single	14.3 LTS	c6id.32xlarge	256	128	Pyspark-Efficient	40	15995	15950	0.159	0.2	0.00333333	No
2	SingleN_Memory_optimized	Single	14.3 LTS	45d.8xlarge	256	32	Pyspark-Efficient	10	16863	16818	0.168	0.2	0.00333333	No
3	SingleN_General_purpose	Single	14.3 LTS	m6g.16xlarge	256	64	Pyspark-Efficient	30	15995	15950	0.17	0.2	0.00333333	No
3	SingleN_General_purpose	Single	14.3 LTS	m6g.16xlarge	256	64	Pyspark-Efficient	50	15995	15950	0.171	0.2	0.00333333	No
3	SingleN_General_purpose	Single	14.3 LTS	m6g.16xlarge	256	64	Pyspark-Efficient	40	15995	15950	0.179	0.3	0.005	No
3	SingleN_General_purpose	Single	14.3 LTS	m6g.16xlarge	256	64	Pyspark-Efficient	20	15995	15950	0.18	0.3	0.005	No
4	SingleN_Compute_optimized	Single	14.3 LTS	c6id.32xlarge	256	128	Pyspark-Efficient	10	16863	16818	0.191	0.3	0.005	No
3	SingleN_General_purpose	Single	14.3 LTS	m6g.16xlarge	256	64	Pyspark-Efficient	10	16863	16818	0.254	0.3	0.005	No

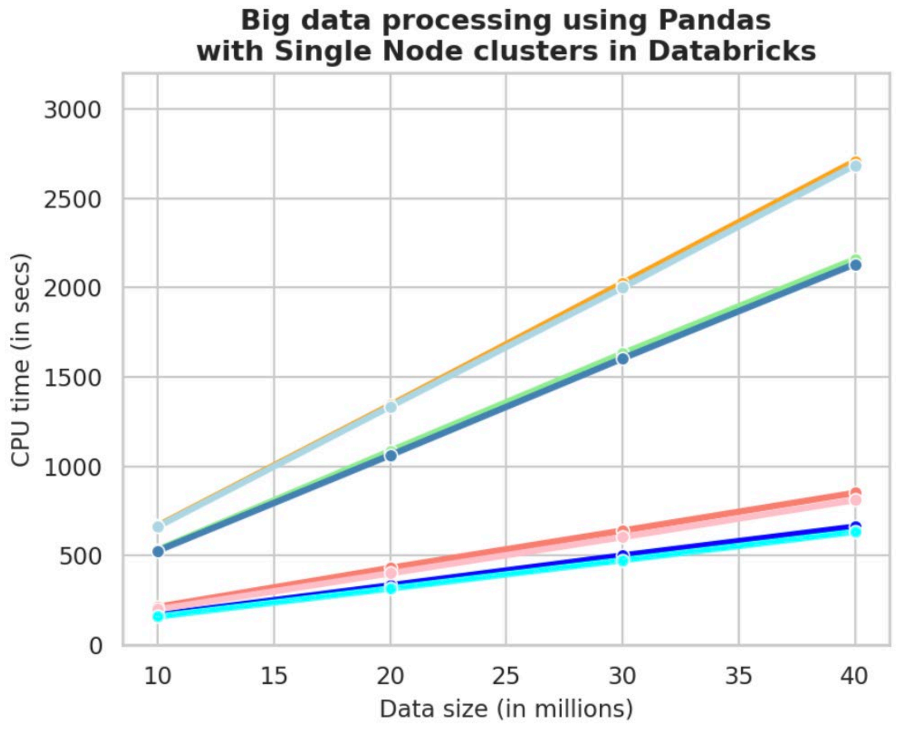
Best results (20 out of 72 Multi-node tests)

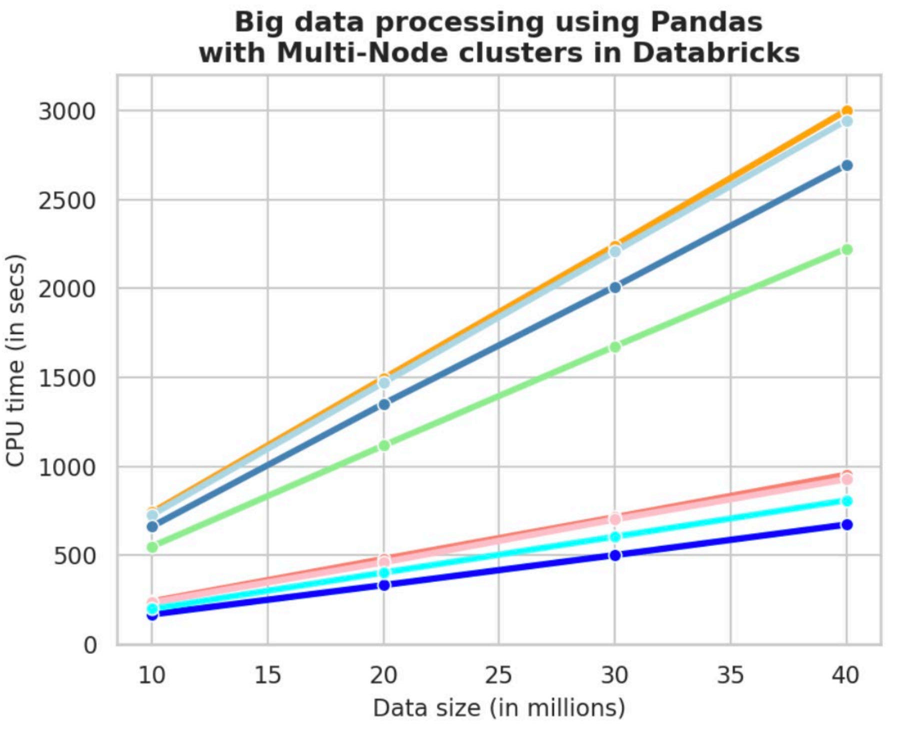
Pyspark-Efficient had superior performance compared to Pandas

Cluster_No	Cluster_type	Node	Runtime	Active memory_gb	Active cores	Method	Dataset_in_Millions	Function_calls	Primitive_calls	CPU_time_secs	Overall_time_secs	Overall_time_mins	Photon_acceleration
1	MultiN_Storage_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	50	15995	15950	0.132	0.2	0.00333333	No
1	MultiN_Storage_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	20	15995	15950	0.133	0.2	0.00333333	No
3	MultiN_General_purpose	Multiple	14.3 LTS	256	64	Pyspark-Efficient	50	15995	15950	0.137	0.2	0.00333333	No
2	MultiN_Memory_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	20	15995	15950	0.141	0.2	0.00333333	No
1	MultiN_Storage_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	40	15995	15950	0.143	0.2	0.00333333	No
2	MultiN_Memory_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	50	15995	15950	0.143	0.2	0.00333333	No
1	MultiN_Storage_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	30	15995	15950	0.146	0.2	0.00333333	No
2	MultiN_Memory_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	30	15995	15950	0.147	0.2	0.00333333	No
3	MultiN_General_purpose	Multiple	14.3 LTS	256	64	Pyspark-Efficient	40	15995	15950	0.148	0.2	0.00333333	No
3	MultiN_General_purpose	Multiple	14.3 LTS	256	64	Pyspark-Efficient	20	15995	15950	0.149	0.2	0.00333333	No
3	MultiN_General_purpose	Multiple	14.3 LTS	256	64	Pyspark-Efficient	30	15995	15950	0.155	0.2	0.00333333	No
2	MultiN_Memory_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	40	15995	15950	0.169	0.2	0.00333333	No
3	MultiN_General_purpose	Multiple	14.3 LTS	256	64	Pyspark-Efficient	10	16863	16818	0.177	0.3	0.005	No
1	MultiN_Storage_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	10	16863	16818	0.18	0.3	0.005	No
4	MultiN_Compute_optimized	Multiple	14.3 LTS	256	128	Pyspark-Efficient	30	15995	15950	0.183	0.3	0.005	No
4	MultiN_Compute_optimized	Multiple	14.3 LTS	256	128	Pyspark-Efficient	50	15995	15950	0.183	0.3	0.005	No
4	MultiN_Compute_optimized	Multiple	14.3 LTS	256	128	Pyspark-Efficient	20	15995	15950	0.184	0.3	0.005	No
4	MultiN_Compute_optimized	Multiple	14.3 LTS	256	128	Pyspark-Efficient	40	15995	15950	0.187	0.3	0.005	No
2	MultiN_Memory_optimized	Multiple	14.3 LTS	256	32	Pyspark-Efficient	10	16863	16818	0.197	0.3	0.005	No
4	MultiN_Compute_optimized	Multiple	14.3 LTS	256	128	Pyspark-Efficient	10	16863	16818	0.223	0.3	0.005	No

Pandas computing (CPU) time:
Single node & Multi-node clusters







Pyspark computing (CPU) time: Single node & Multi-node clusters