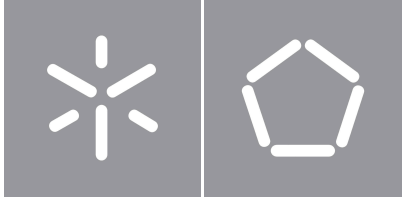
University of Minho
School of Engineering

André Carvalho da Cunha Martins

Analysis of trade-offs between performance and energy efficiency of scalable Dataframes tools

november 2023



University of Minho
School of Engineering

André Carvalho da Cunha Martins

Analysis of trade-offs between performance and energy efficiency of scalable Dataframes tools

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Ricardo Manuel Pereira Vilaça

november 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

First, I would like to thank my supervisor, Professor Ricardo Manuel Pereira Vilaça, for all the support, availability, and knowledge shared during this thesis. Always looking for the best results and solutions, Professor Ricardo helped me during this project and supported my personal and academic growth, sharing all his knowledge and experience in the area and making this closure of my academic path the most rewarding and proud that I could be. My biggest thank you. I would, also, like to thank all the MACC collaborators, especially Miguel Peixoto, for all the help with technical problems and availability when needed.

Second, I would like to thank INESC TEC for the possibility of doing this project while being on an academic scholarship. This opportunity motivated me to reach all my objectives and goals while being able to be in close contact with other scholarship holders and share knowledge and experiences. Being able to publish an article in INForum 2023 was only possible due to that support.

Personally, an enormous thank you to all my friends who have done this beautiful journey with me, that started 5 years ago. Being able to share the most memorable stories, special moments and experiences is something that will stay with me for the rest of my life. This work was accomplished because of all of you. You made me believe that I could make it and you gave me the strength that I needed to do it. Forever with me, no matter where or when! A very special thank you to Dona Rosa and Senhor Manuel for all the kindness and affection given during these 5 years!

And finally, my biggest thank you to my family, Mom, Dad, and Eduarda. For all the support given since I started my degree, for all the daily motivation, and for all the belief in me. Always available to help me through difficult times and always giving me the incentive to achieve my master's degree. This victory was only possible because of you, and it is dedicated to you. I will always be able to give back all the support that I had, especially to my little sister, Eduarda, who is now beginning the most beautiful and rewarding stage of her life.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, november 2023

Assinado por: **André Carvalho da Cunha Martins**
Num. de Identificação: 15857705
Data: 2023.11.21 09:58:05 +0000

André Carvalho da Cunha Martins

Abstract

Nowadays, we have the ability to trace everything, to extract valuable data from wherever we want, all to keep us connected and to improve our lifestyle. This huge amount of information, produced every day, needs to be treated, manipulated, and analysed, requiring convincing data structures to do so.

Dataframes, regularly used worldwide, are powerful data structures used to analyse and manipulate data of any kind. A Dataframe organizes data into a 2-dimensional table of rows and columns, similar to SQL tables or CSV files. Furthermore, it can span alongside thousands of computers or servers, making it easier to work with huge amounts of data, called big data, using distributed systems and parallel computing.

This Dataframe's distributed nature led to the rise of distinct scalable and parallel Dataframe tools. The most used Dataframe tool, pandas, only performs on sequential execution and has some limitations when there is the need to handle huge volumes of data, and some tools such as Modin, Polars, RAPIDS, and so forth, appeared in order to overcome those limitations. The vast offer of these scalable tools brought the need to make an analysis and comparison between these frameworks and pandas, studying their behaviour and results with different workflows. This comparison is not linear and there is a need to use a benchmarking tool, in order to produce a homogeneous and reliable evaluation of the different frameworks.

To perform this analysis, we worked with several workflows, manipulating real and synthetically produced data on distributed and parallel environments and on different hardware configurations.

We designed and developed a benchmarking tool that supports a set of Dataframe frameworks, is flexible to the addition of new frameworks, and is able to perform micro-benchmarking evaluation with the analysis of a group of individual and common operations used on data science, and macro-benchmarking evaluation with the analysis of workflows that represent a set of chained operations. Both of these evaluations aggregate performance and energy consumption results for each framework.

Keywords Dataframe, Distributed and Parallel Computing, Performance, Energy Consumption, Benchmark

Resumo

Hoje em dia, é possível extrair dados de onde quer que queiramos, mantendo-nos todos conectados e contribuindo para um melhor estilo de vida. Esta quantidade enorme de informação, produzida diariament, precisa de ser tratada, manipulata e analisada, precisando de recorrer a estruturas de dados capazes de fazê-lo.

Dataframes, utilizados globalmente, são uma poderosa estrutura de dados capaz de analisar e manipular qualquer tipo de dados. Organiza-se numa tabela de 2 dimensões de columas e linhas, como uma tabela de SQL ou um ficheiro CSV. Um *Dataframe* consegue ser dividido por múltiplos servidores, facilitando o trabalho com enormes quantidades de dados, uma vez que é possível utilizar computação paralela.

Esta característica de paralelismo dos *Dataframes* levou ao aparecimento de várias ferramentas escaláveis e distribuídas. A ferramenta de Dataframes mais utilizada, pandas, apenas é capaz de executar sequencialmente, tendo algumas limitação quando há a necessidade de trabalhar com enormes quantidades de dados, e algumas ferramentas como o Modin, Polars, Rapids, entre outros, apareceram para superar essas mesmas limitações. A oferta vasta destas ferramentas escaláveis trouxe a necessidade de fazer uma análise e comparação entre estas ferramentas e o pandas, estudando o seu comportamento e resultados com diferentes *workdflows*. Esta comparação não é linear e existe a necessidade de utilizar uma ferramenta de *benchmarking*, para gerar uma avaliação homogénea e fiável. Para fazer esta análise, trabalhamos com *workflows* de vários tipos, manipulatdos dados reais e sinteticamente produzidos em ambientes distribuídos e em diferentes configurações de *hardware*.

Prototipamos e desenvolvemos uma ferramenta de *benchmarking*, com suporte a várias ferramentas distribuídas, e flexível à adição de novas ferramentas, que é capaz de realizar avaliações de *micro-benchmarking*, com a análise de operações individuais, e *macro-benchmarking*, com a análise de *workflows* que representam um conjunto de operações encadeadas. Ambas as avaliações agregam resultados sobre a performance e o consumo energético de cada *framework*.

Palavras-chave Dataframe, Computação Distribuída e Paralela, Performance, Consumo Energético, Benchmark

Contents

List of Figures

List of Tables

Acronyms

COW Copy-on-Write. 1, 32

DBMS Database Management System. 1, 4, 31

HDFS Hadoop Distributed File System. 1, 29

JIT Just-in-time. 1, 28

MPI Message Passing Interface. 1, 28

NYC TLC New York City Taxi and Limousine Commission. 1, 39, 41, 47, 51

OLAP Online Analytical Processing. 1, 13

OLTP Online Transaction Processing. 1, 13

PDR Preliminary Dissertation Report. 1

RAPL Running Average Power Limit. 1, 7, 8, 48

SIMD Single Instruction Multiple Data. 1, 14, 32

SOA State of the Art. 1

SPMD Single Program Multiple Data. 1, 28

SSL Secure Sockets Layer. 1

TLS Transport Layer Security. 1

TPC-* Transaction Processing Performance Council. 1, 8, 20, 21

YCSB Yahoo! Cloud Serving Benchmark. 1, 20, 21

Chapter 1

Introduction

Data analysis can be described as a set of processes and operations for cleaning, transforming, and modeling data to support decision-making. These processes help us to better understand and learn how to use the huge amount of data that our digital world produces every single day. It is used in every field, from finance markets to healthcare or governments. With the help of data analysis, organizations can understand, through patterns and relationships, what the data represents and make decisions based on stronger and better evidence.

Data Science, Artificial Intelligence, and Machine Learning are 3 of the software engineering fields that have the highest demand on the market, even though, there is a huge deficit of professionals in these areas. According to the US Bureau of Labor Statistics, it is expected that jobs that require skills and knowledge in the Data Science area will increase by 27.9% by the year 2026, increasing the Data Science market to an unbelievable 322.9 billion USD. We can explain these rising numbers by the huge abundance of data that we can find, with big companies working with gigantic datasets, that are expected to exponentially increase. But, what data structures does a software system use to store and process all data?

Dataframes are one of the most used data structures in data analysis. Providing a 2-dimensional table of rows and columns, Dataframes allows us to easily manipulate data efficiently. It is possible to filter, select, sort, group data from a dataset and even join and merge data from different datasets, enabling the combination of data from different sources and performing more complex analysis. Moreover, a Dataframe can span alongside thousands of computers or servers, allowing them to scale to huge datasets. This way, and using distributed systems and parallel computing, it is possible to split datasets, that were too heavy for a single machine, through a set of clusters, obtaining performance gains and fault tolerance. As datasets are getting bigger and storing more data, this feature is a crucial advantage to optimize data analysis performance.

As we can imagine, alongside the data analysis of huge datasets, comes a lot of computer power

and energy consumption. Hardware resources such as CPU and RAM are crucial to process all the data contained in a Dataframe but have huge costs associated. So, looking just for the performance results and ignoring all the resources and energy consumption metrics isn't the best approach to finding the best working tool. However, we can not fully dismiss the seriousness that performance represents and how important is the time invested to do a given task or process. This situation takes us to a point where it is not clear what is the best choice or what path should we follow, raising several questions and doubts.

What is a better option for a software engineer? Use all the available resources for a shorter period, or partially use the available resources for a longer period?

This dichotomy does not have a direct answer and it is not easy to find the perfect balance between energy consumption and performance. However, we need to keep in mind all the important factors associated with data analysis and study the best options that can keep a good trade-off between the aspects referred to previously, to reach the best possible solution for an established scenario.

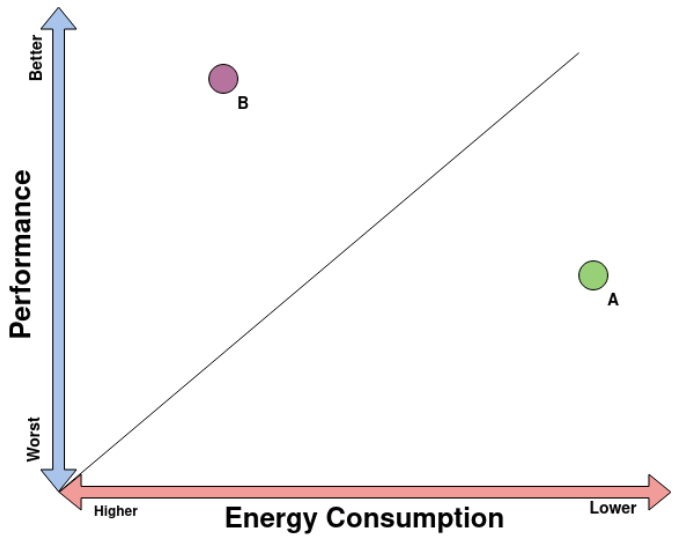


Figure 1: This diagram illustrates the balance between performance and energy consumption.

Undeniably, and using Figure 1 as an example, the best possible tool would be on the top right corner of the diagram presented above, having the best performance and lower energy consumption metrics. On the same page, the worst tool that we can identify is in the bottom left corner, presenting low performance and high energy consumption metrics. But, having a realistic and reasonable approach, we know that those two scenarios are very uncommon and the answers we are looking for are not so direct or undeniable.

By studying some of the most popular scalable Dataframes tools, we can analyse their balance between performance and energy consumption and make an evaluation of the obtained results.

1.1 Motivation

As described in the previous topic, Dataframes are one of the most used data structures in data analysis. There are a vast set of different Python frameworks to manipulate and work with Dataframes. The most popular is pandas, a Python framework developed back in 2009 that helps users to manipulate and analyse datasets easily, implementing an intuitive and easy-to-understand API. Although pandas has enormous popularity, it does not scale to several clusters, performing all the operations on a single machine. This brings up limitations when datasets are bigger than the available resources, causing performance issues.

Due to these limitations, some scalable Dataframes tools have risen through the years. Implementing parallel computing principles and new approaches, those frameworks were able to bring advantages such as the implementation of Out-of-Core principles, lazy evaluations principles, the usage of another available resource like GPU, and the usage of more than one machine, through multi-node executions. With these new tools presenting different proposals and claiming to have achieved performance improvements when compared to the traditional, and most used, sequential execution, the common user has, now, a wider set of options to choose from.

But how we prove those performance improvements? How can we state that those tools are successfully parallelizing Dataframes and are a better option than pandas, which is the most used framework? How do we know which framework will perform better when facing the need to work with a huge workload?

To produce statistics and acquire results about the different Dataframes tools we need a benchmarking tool to execute the same complex operations with all the frameworks, analysing all of them on the same distributed environments, and, finally, compare the obtained results working with graphs and diagrams that illustrate both performance and energy consumption metrics. By performing these actions, we have concrete proof and results to verify the advantages of those frameworks and to show how working with Dataframes in distributed environments can be expressed in performance improvements when working with heavy workloads.

Allied with the performance analysis and execution times comparisons between frameworks, energy consumption, and its impact have become a big concern in the community in recent years. There is a known lack of knowledge of energy efficiency by programmers, as stated by a survey in 2015, where the participants highlighted the unfamiliarity with software energy consumption and how does a software consumes energy. "These results highlight the need for training on energy consumption." [1] Nonetheless, there is an emerging preoccupation and awareness in this field, with the community being more active and more worried about how can we improve what we are used to doing and how can we build software

that represents ecologically energy consumption results and, at the same time, offers good performance metrics in a well-working application. A study done by Gustavo Pinto, Fernando Castor, and Yu David Liu [2] helps to understand and prove that programmers are becoming more aware of the importance of energy consumption.

- *Energy consumption questions sustain a near-linear growth in the last 5 years, with contributors from 9 geographic regions. This suggests that energy consumption is an emerging topic.*

Facing this information, we understand the need to attach energy consumption results when evaluating scalable Dataframe tools. These statistics are gaining even more importance and highlight, especially when we know that we are living in an era of rising electricity costs worldwide, where the energy costs to power a data center, for example, are expected to double every five years. [3]

Even though the incorporation of both energy consumption and performance statistics can produce a complete analysis, there is an absence of benchmarking tools that are capable of doing it and applying these evaluations to a considerable set of scalable tools. The development of a tool that is capable of merging these features and is able to analyse these emerging scalable frameworks can be seen as an innovative and distinct approach to all previous benchmarking analysis that was performed through recent years. The most used benchmarking tools to evaluate this type of framework are based on a reduced set of frameworks, being inflexible and *ad hoc* to specific tools, or are fully focused on just energy consumption or performance metrics, separately. For example, databases and **DBMS** have more complex, general, and standard known benchmarks, while Dataframes processing tools benchmarks are in an embryonic stage of development, without filling the need to answer to some of the rising questions and needs on the community.

As databases are globally used for a way longer time than Dataframes, it is natural that evaluation tools for these data structures are rising and being developed only in recent years. Nevertheless, the innovation and progress invested in new benchmarking tools can help sustain the existing scalable tools and inspire the development of new tools to suppress the flaws and limitations identified in different environments. The more embracing, wide, and flexible the benchmarking tools are, the more these frameworks will grow and become better.

1.2 Objectives

We defined two main goals for this work, deeply related to each other. Our work can be divided into two major stages, with a first stage dedicated to the design, prototype, and development of a benchmarking

tool and a second stage that is focused on the production of performance and energy consumption results, using our tool, for a group of scalable frameworks and consequent evaluation and analysis.

The development of the benchmarking tool aimed to produce a product that can evaluate a set of scalable Dataframe frameworks. The conjunct of frameworks that are able to be tested is not restricted, existing the capacity to add, at any occasion, more frameworks. Similarly to the addition of new frameworks, the tests that are performed are also flexible and open to new additions. These tests will be executed with workloads with variable sizes and in different environments. Our tool is capable of performing a wide number of operations on several frameworks, making this a general and embracing benchmark.

Alongside the development of this tool, we pointed to deepening the study of scalable frameworks in distributed environments. Using the results obtained with the execution of our benchmark, we focused on the comparison of the scalable frameworks between them and between the sequential execution of a non-scalable framework like pandas. Using different environments, hardware configurations, and tests, we desired to produce an analysis, evaluation, and comparison of the frameworks implemented on our tool. More than just the development of a general benchmark, we looked for a tool that is capable of producing a strong basis and coherent results that support the study of distributed frameworks, being able to identify advantages, flaws, performance gains, and energy consumption indicators of those frameworks and, at the same time, giving a scientific and objective explanation about the results produced. The work invested in this detailed study helps to consolidate our benchmark as a solid option and cohesive tool to be used for further applications.

1.3 Results

This work culminated in a benchmarking tool prepared to evaluate a wide group of scalable Dataframe tools. Built to produce a homogeneous, reliable, and wide analysis of these distributed frameworks, this benchmark is flexible to the addition of new frameworks and new workflows, while allowing evaluation with real and synthetic data workflows, on multiple distributed environments. This tool produces, as well, graphics to translate the results' information and ease the understanding and comparison of all implemented frameworks.

The benchmarking tool is available at https://github.com/accunhamartins/PerfEnerPy.

Together with the development of our benchmarking tool, we produced a results analysis and evaluation, where is made a description of the results obtained, both for micro-benchmarking and macro-benchmarking tests and for both real and synthetic data artifacts. This evaluation embraced two different

hardware configurations, with distinct available resources, and was tested on distinct distributed environments, such as single-node and multi-node executions.

The work developed during this thesis resulted in the publication of a scientific paper in the INFORUM 2023 - Simpósio de Informática.

1.4 Structure

This work is composed of the following chapters: Chapter 1, **Introduction**, Chapter 2, **State of the Art**, Chapter 3, **Benchmark Design and Implementation**, Chapter 4, **Evaluation, Testing Environments and Results** and Chapter 5, **Conclusions and Future Work**.

This chapter, Chapter 1, makes an introduction, problem contextualization, and description of our motivation to do this work. In this chapter, we explain the problem that we wanted to approach and the motivation that guided us to build and develop our benchmarking tool and consequent results.

State of the Art, Chapter 2, where we describe the existing benchmarking tools and studies that would help our work, understanding their advantages, their flaws, and gaps, to build a more stable and complete tool, able to answer to questions and to make comparisons that have not been made so far. Allied with this research, we aim to understand how a Dataframe works, how distributed and parallel frameworks handle that data structure, and how they optimize this process when compared to the sequential, and most used, execution made with pandas. This way, we analysed and understood the results obtained in Chapter 4, making a deeper and more accurate differentiation between tools.

Benchmark Design and Implementation, Chapter 3, where we describe how our benchmarking tool works and how is it built and deployed, with a detailed architecture and requirements description. This chapter also made a detailed explanation of the implementation of our tool.

Evaluation, Testing Environments, and Results, Chapter 4, where we explain how we performed our energy consumption and performance analysis, describing the hardware configurations used, the tests that were made, and which frameworks were selected to be deeply studied. Presenting results generated by our tool, we proved the difference, both in execution performance and energy consumption, between every framework, revealing their strong relation to the complexity of operations, the available resources, the hardware configurations, and the workloads that were executed.

Finally, Conclusions and Future Work, Chapter 5, where is made a conclusion about our work and results produced and some considerations for future work that can be developed from our benchmark.

Chapter 2

State of the Art

2.1 Energy Consumption

Energy efficiency, energy consumption, and how we must have greener behaviors are real problems that are getting more attention every day. Ambient and climate crisis is a real threat and we must act in order to reduce our digital footprint. With the rising prices of energy in our households, the need to use more forms of renewable energy, and to use much less fossil energy like coal and petroleum, our society must change its lifestyle and try to make our daily basis a "greener" one. Software systems and devices play a crucial and indispensable role in our schools, hospitals, government systems, courts, and in every possible aspect of our daily routine. But how can we measure and collect information about how much energy a software system consumes? How do we know if our programs are using energy in an efficient way?

2.1.1 Energy consumption measurement

Energy consumption from an arbitrary software system can be measured through both hardware and software applications. Hardware tools like energy meters or power monitors are the most accurate ways to measure energy consumption, although, not ignoring their extreme precision, these types of tools are extremely difficult to set up, often requiring changes to the system itself. Software applications for energy measurement are more commonly used, and easier to acquire and set up. Libraries like **Running Average Power Limit** (**RAPL**), developed by Intel and capable of measuring energy consumption from CPU, and memory in real-time, can be used directly or indirectly through other applications like Powerjoular or Intel Power Gadget. **RAPL** allows, as well, to set power limits on our devices, reducing energy consumption.

Powerjoular [4] is a tool capable of monitoring power consumption from CPU and GPU for computers, servers, or devices like Raspberry Pi, named single-board computers. It can be used jointly with other

tools like JoularJX, for example. In this case, JoularJX will use the data provided by Powerjoular to monitor the power consumption of methods and source code of Java applications. This tool was built to assist roles such as system administrator, or even a common software developer, to understand and analyse the energy consumption of their devices and software.

Implementing Intel **RAPL**, this tool will automatically detect the device configuration and support modules, in order to provide accurate power data. In CPU power monitoring, it will use RAPL power data through Linux powercap interface with the help of system files, detecting which power domains are supported by the CPU. In the case of GPU, Powerjoular will use NVIDIA SMI to verify if GPU power monitoring is possible for that graphic card. If that condition is fulfilled, Powerjoular will proceed to make GPU power consumption readings every second. Finally, the tool will combine the power readings from both CPU and GPU and provide overall power consumption data.

SLURM [5] also implements plugins that are capable of measuring energy consumption. For example, the Energy Accounting Plugin, is able to collect, through hardware sensors and **RAPL**, energy consumption data. This plugin allows the SLURM user to perform hardware monitoring, reporting instantaneous power and cumulative energy consumption for each node. It also permits the profile of power used by a job over time, per node, and total energy consumption by a job. [6]

Energy-aware benchmarks

TPC-* founded back in December 2007 the Energy Specification Subcommittee, in order to support and complement their existing benchmarking tools. The Energy Specification [7] creation was based on energy efficiency rising awareness, becoming one crucial factor to be analysed and evaluated. This tool was developed to help manufacturers measure energy usage by an average user of their system. Allied with Energy Measurement System, an online package that eases the implementation of **TPC-*** Energy Specification, and provides services like power instrumentation interfacing, power and temperature logging or report generation, **TPC-*** aims to successfully add energy measurement and energy consumption reports to their vast group of benchmarks.

Another example of this type of benchmarking tool is EA-HAS-Bench, presented as a pioneer in large-scale energy-aware benchmarks. This tool was developed for studying AutoML methods, in order to produce better trade-offs between performance gains and energy consumption efficiency. [8]

This tool, more connected with Artificial Intelligence and Machine Learning fields, provides data scientists with training energy costs and model performance on a specific architecture and hyperparameter configuration, without the need to actually train the model.

2.2 Benchmarking

Benchmarking is responsible for testing performance in a controlled manner, giving a strong basis to compare choices and to understand performance limitations before these are found later in production. These limits can be system resources, locally or in cloud computing and virtualized environments, or limits from the target application itself. [9]

Benchmarking can be used for several different reasons, such as system design, troubleshooting, and even marketing-related issues. It is a common practice used in different areas, from software development to big companies marketing. Taking cloud computing as an example, benchmarking tools and methodologies are very helpful in making the best choices for the analysed problem, once this type of system allows to rapid create a large-scale environment, perform a benchmark run and then destroy the environment at low cost. [9]

Even though it may look like simple work to do, benchmarking is very difficult to perform well and to produce accurate and correct analysis. It can lead to a lot of mistakes, misinterpretations, and oversights. Bad interpretations of accurate results are not trustworthy and can lead to major issues in systems development. What looks like an efficient and well-performing system can be the system with a hidden flaw that may appear later on the development life-cycle.
Upcoming, we will present some common mistakes and issues that can lead to these issues.

2.2.1 Common benchmarking mistakes

As written in the book *Systems Performance* [9], authored by Brendan Gregg, there are some common issues related to benchmarking evaluations and tools that need to be avoided.

Casual Benchmarking

Benchmarking is a complex operation and results interpretation may not be as simple as it seems. It requires some experience and field knowledge to understand and judge the obtained results, once they may not look suspicious or incorrect on a first-look evaluation.

Another important aspect is the need to really understand what we are analysing and what components are being evaluated. For example, some benchmark tools may say that they will execute a disk performance but they will actually perform a file system performance. There is a major difference between these two and can mislead to wrong interpretations and conclusions about the results given, even though the benchmark tool is correct and gives good and accurate results.

Benchmarking Faith

Despite that are a lot of benchmarking tools that are trustworthy, open source, and being used by a lot of companies, blindly believing that they are flawless and produce totally accurate results is a mistake. We should analyse all the benchmark tools that we use and give some time to look at their results. This process and judgment require some experience and knowledge about the expected results and, in the majority of the cases, the problem is not related to the benchmark software itself (although these tools are not flawless and might not be working correctly) but to the interpretation of the results.

Giving numbers without proper analysis

When producing a results analysis, there must be an explanation for the given results. This kind of analysis takes time to be performed and just dropping numbers without context is a bad methodology. There is the need to replicate the evaluation a few times, understand the provided numbers by the tools used, and not make blind assumptions, for example, the disk I/O test actually measures disk I/O or the benchmark tool drove disk I/O to its limit, as intended. Again, this process requires experience and dedicated time and it is crucial to give the necessary effort and focus in order to produce the analysis that we want.

Ignoring errors

This may look like an obvious mistake but it is commonly done, once it is easy to accept that the tests were well performed if they produce a result. But, in some cases, tests may produce a result that reflects an error, and all the requests done by the benchmark tool were not executed. Once again, it is very important to analyse all the results and confirm that all the intended requests were actually performed.

2.2.2 Deployment

Benchmarking tools include several software systems where they can perform evaluations and testing procedures. From benchmarking microservices that can be deployed on Kubernetes to tailor-made benchmarks for a specific system evaluation, these tools can be deployed and executed through a variety of different software environments and operative systems.

SLURM

SLURM [5] is an open-source cluster manager and job scheduling tool. Known for its high scalability and fault tolerance, this system was designed to manage Linux clusters of several sizes. Requiring no kernel

modifications, this system has three major functions:

- **Resources management:** Responsible for giving exclusive or non-exclusive access to the available resources.
- **Job execution:** Responsible for job execution and monitoring work on the select set of allocated nodes of the cluster.
- **Job management:** Responsible for managing a queue of pending jobs.

SLURM's architecture is characterized by a central manager, called slurmctld, that is responsible for all the resources and work handling. There is in each node a daemon process, slurmd, that can be compared to a remote shell. This daemon will be on a waiting status while there is not any work allocated to it, then it will execute the respective jobs, return its status and return to its waiting status. These processes provide fault-tolerant hierarchical communications.

The common user can communicate with SLURM workload manager through a REST API from a daemon process called slurmrestd, using commands srun to initiate a job, scancel to cancel a job, or squeue to see the overall status of running and pending jobs. There are also administrative commands like scontrol that report the state information of the cluster and of its jobs.

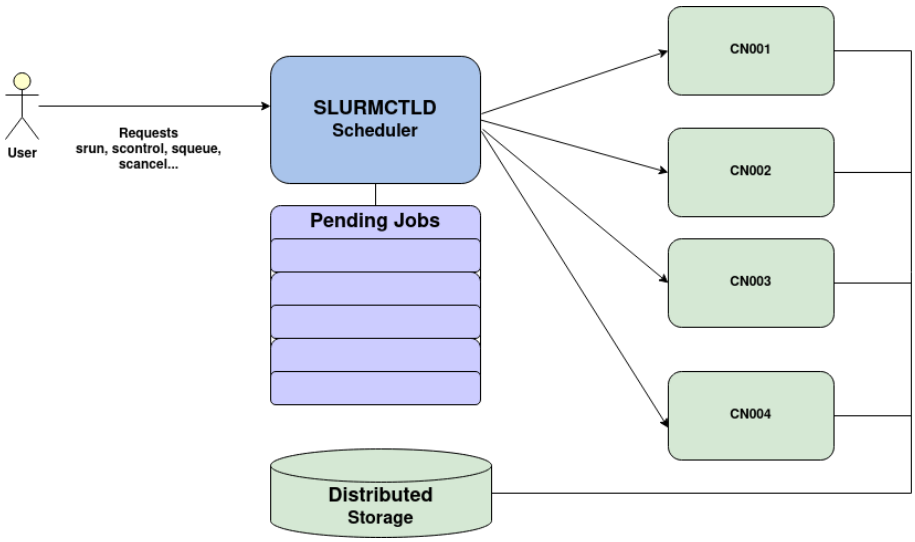


Figure 2: SLURM Architecture.

2.3 Memory Formats

2.3.1 Row vs Columnar Storage Formats

The most common storage format that we are used to seeing and working with is the row-oriented storage format. This format makes it easier, when dealing with small-size files, to search and update records, representing operations that are instantly executed. But, when we are dealing with big data files that have a huge volume of data, these operations can be a very time-consuming process, having poor performance and high execution times for simple operations such as searching for a record or deleting a record.

This way, columnar storage format can be a solution to improve overall performance and be more efficient. While working with row-oriented storage format there is the need to load the entire row to then try to find a specific record, in this format only the necessary data will be loaded, working only with the particular data that is needed for a specific query. This simple process represents a visible reduction in the time needed to execute a query.

The figure below is a representation of the major difference between row and columnar storage formats.

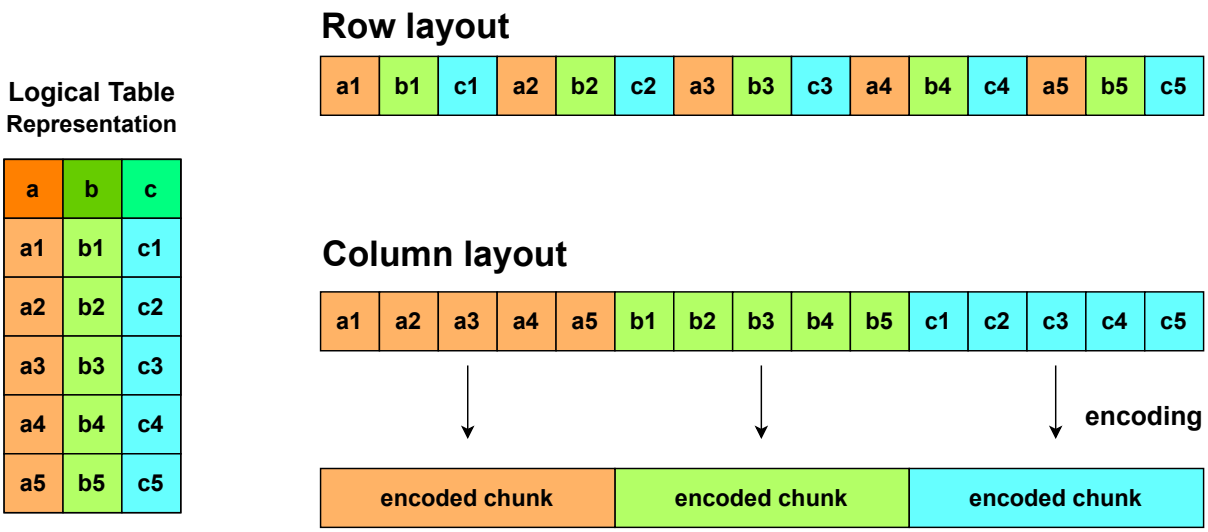


Figure 3: Row vs Columnar Storage Formats.

Parquet and ORC are two of the most known examples that implement column-oriented storage formats.

2.3.2 Apache Parquet

"*Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.*" [10]

Developed by Apache Foundation Software, Parquet was built to provide a column-oriented data file format, taking advantage of compressed, efficient columnar data representation. Parquet is language agnostic and is used for **Online Analytical Processing** (**OLAP**) use cases, in conjunction with **Online Transaction Processing** (**OLTP**) databases. This format is also very flexible, supporting complex data types and advanced nested structures.

Parquet proves to be a solid option when working with the storage of big data of any kind, from structured data tables to images or videos. It also saves cloud storage space, since uses highly efficient column-wise compression. Also presents performance improvements and increased data throughput by using data skipping, allowing queries that fetch just a specific column value not to read the entire row.

Parquet's flexible compression offers the possibility to compress data using one of several codecs available, allowing different data files to be compressed in different ways. This file format was also optimized to work and execute queries that will process large volumes of data. Using its column-oriented data format, Parquet provided hardware savings and minimized latency for accessing data.

When compared with CSV files, Parquet files offer clear advantages in efficiency and costs. Using a study [11] done by Thomas Spicer as an example, the author shows how Parquet highlights itself from CSV files. After the CSV files were converted to Parquet files, and later compressed and partitioned, it is possible to observe the clear gains with the following image made by the author.

Dataset	Size on Amazon S3	Query run time	Data scanned	Cost
Data stored as CSV file	1TB	236 seconds	1.15TB	$5.75
Data stored as Parquet file	130GB	6.78 seconds	2.51GB	$0.01
Savings / Speedup	87% less with Parquet	34x faster	99% less data scanned	99.7% savings

Figure 4: Amazon S3 storage pricing for different data formats.

Parquet shows a clear advantage in every aspect analysed, representing 87% less space occupied on Amazon S3, which will represent 99.7% of cost savings. Performance has also greatly improved, once queries ran 34 times faster and 99% of less data was scanned.

Although this file format presents considerable performance gains and more efficient query execution, one of the biggest disadvantages associated is that Parquet is only machine-readable, which makes it too

difficult for developers to understand the data stored by just looking at the file. Besides that, if there is the need to add a new record, the whole file needs to be recreated, once Parquet files are fully immutable and do not allow the addition of new records to the existing ones.

2.3.3 PyArrow

PyArrow is the Python implementation of Apache Arrow, providing a Python API for the functionalities of Arrow, allowing the integration of Python frameworks like pandas or NumPy. Apache Arrow is a language-independent columnar memory format made for flat and hierarchical data. Built to process and transport large datasets, Apache Arrow was developed to improve the performance of analytical algorithms and the efficiency of moving data from one system to another.

It is in-memory columnar format [12] is a language-agnostic specification to represent structured datasets in memory. Using this format, Arrow allows computational routines and execution engines to maximize and improve their efficiency when dealing with and analyzing large datasets. In fact, the continuous columnar layout enables vectorization using **Single Instruction Multiple Data** (**SIMD**) operations.

Figure 5: Columnar memory format.

This format has some key advantages such as:

- Data adjacency for sequential access (scans)
- Random access performed in constant time (O(1))
- **Single Instruction Multiple Data** and vectorization
- Relocatable without "pointer swizzling", allowing for true zero-copy access in shared memory (Pointer swizzling is the conversion of references based on name or position into direct pointer references

(memory addresses)).

This tool also offers standardization saves by, due to its columnar data format, making it no longer necessary for every database and language to implement its own data format, creating a lot of waste and making data moving from one system to another a costly process of serialization and deserialization. This would also lead to the need to rewrite every algorithm for each data format. Columnar data format offers a solution to the previous problem, by allowing systems that support Apache Arrow to transfer data between them with almost no cost associated. They don't need to implement custom connectors to other systems and using this standardized memory format, makes the re-utilization of algorithms an easier process, even across different programming languages. [13]

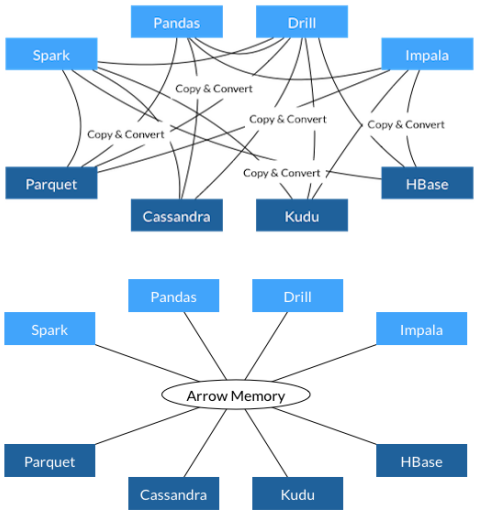


Figure 6: Standardization Saves.

Similarly to other frameworks, Apache Arrow has an API Dataframe, as its columnar memory format gives support to other tools, like Polars.

2.4 Dataframes

2.4.1 The beginning of Dataframes

Dataframes origin can be associated with the S programming language, developed in 1976 at Bell Laboratories in order to support and help statistical computation. Dataframes were later developed and first introduced in 1990, presented by Chambers, Hastie, and Pregibon at the Computational Statistics conference. Then, the authors stated in a book [14] that Dataframes are a more general data structure than

matrices, once matrices assume that all elements have the same type, for example, all numeric, all logical, or all strings, not allowing the presence of more than one type on a single matrix. On the other hand, Dataframes have full support for matrix-like computation, with variables as columns and observations as rows, while they also allow computations where variables act as separate objects, that can be referred to by name.

In 1995, the R programming language evolved from the S programming language and is presented as an open-source version of S with some modern add-ons. The R programming language, after the release of its stable version in 2000, gained a lot of fame and was widely used by the statistics community.

Jumping into 2008, Wes Mckinney presents pandas, a modern framework that pursued to bring the known Dataframes capabilities from the R programming language to Python. After that transition, Dataframes gained even more popularity and are now globally used, not only in the statistics community but also in Machine Learning, Artificial Intelligence, and Data Science.

Using an illustration from a Ph.D. thesis, authored by Devin Petershon [15], we can illustrate what a Dataframe Data Model looks like.

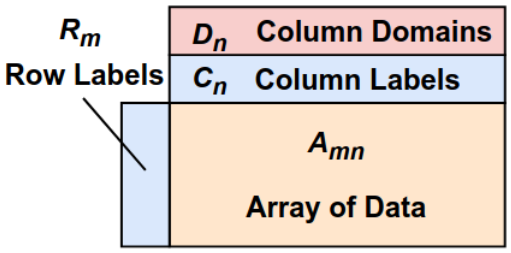


Figure 7: The Dataframe Data Model.

2.4.2 Dataframes Data Model and Operations

Dataframe Data Model

As shown on the previous topic, the Dataframe Data Model was defined by Devin Petherson [15] as *"a tuple (A_{mn}, R_m, C_n, D_n), where A_{mn} is an array of entries from the domain Σ^*, R_m is a vector of row labels from Σ^*, C_n is a vector of column labels from Σ^*, and D_n is a vector of n domains from Dom, one per column, each of which can also be left unspecified. We call D_n the schema of the Dataframe. If any of the n entries within D_n is left unspecified, then that domain can be induced by applying S(·) to the corresponding column of A_{mn} to get its domain i and then p(·) to get its values."*

Given the definition above, it is important to enhance that the domain Σ^* is the set of finite strings over an alphabet Σ, serves as a default, uninterpreted domain, and can be called, in some examples of Dataframes libraries, an Object.

A particular and important characteristic of a Dataframe is that we can induce *post hoc*, from data, the domain of its columns, unlike relational models, where these domains must be declared *a priori*. Dataframe's rows and columns can be referenced in two different ways. We can either index them using a number, called positional notation, or we can index them using a label, called a named notation. The traditional relational model only gives this type of indexing for its columns, not for its rows. Positional notation also allows us to index individual values through *(row, col)* references, just like a simple matrix. However, there are also a few differences between Dataframe's columns and rows. One of those differences is that columns own a schema, but rows do not, in other words, we parse any cell value based on the domain of its column.

Focusing now on D_n (Column Domains), we will notice that it provides some *schematic asymmetry*, besides all the notational symmetry between rows and columns. Petherson presents a pretty good example as he considers a schema like D_n = **[category, int, float]**, to show that although the column types differ, the type of each row will always remain the same, on specific **[category, int, float]**. This shows that each column has a single domain, but each row will have a vector of domains. When we have the case where the schema D_n has the same domain for all of *n* columns, presenting a **homogeneous Dataframe**, whose rows and columns have the exactly same domain and only differ in dimension.

This work by Petherson deeply studies Dataframes and the algebra behind it, resulting in great research about the theme and how it works. Following, we will focus on and illustrate just a few pieces of information about Dataframes algebra and operations. The remaining information about all the Dataframes algebra concepts and formulations can also be found in his Ph.D. thesis [15].

Dataframe Operations

During this work, while studying several Dataframes tools, we noticed that there is a core of Dataframes operations that all tools provide and allows us to read, alter, sort, and even delete the data from a Dataframe. Petherson made a table, presented below, that describes all the main operations of Dataframes manipulation and visualization. These operations are based on pandas and R API's, and other Dataframes tools might implement new operations, but we believe that these are the most important and most used ones.

Operator	Description
SELECTION	Eliminate rows
PROJECTION	Eliminate columns
UNION	Set union of two Dataframes
DIFFERENCE	Set difference of two Dataframes
CROSS PRODUCT / JOIN	Combine two Dataframes by element
DROP DUPLICATES	Remove duplicate rows
GROUPBY	Group identical values for a given (set of) attribute(s)
SORT	Lexicographically order rows
RENAME	Change the name of a column
WINDOW	Apply a function via a sliding-window (either direction)
TRANSPOSE	Swap data and metadata between rows and columns
MAP	Apply a function uniformly to every row
TOLABELS	Set a data column as the row labels column
FROMLABELS	Convert the row labels column into a data column

Table 1: Dataframes Operations [15]

These are the most used and most common operations when manipulating Dataframes using pandas or R API's. These API's are globally used and are gaining a rising popularity in the last 20 years.

As Data Science mixes software engineering and statistical and mathematical knowledge, Dataframes are seen as an essential tool that offers everything that a Data Scientist needs to do his job.

2.4.3 pandas Dataframes

pandas was first developed in 2008, by Wes McKinney, and became an open source tool by 2009. As mentioned by its developer, this Python library provides intuitive routines for performing data manipulation operations and analysis on datasets. It was developed to help and complement the existing scientific Python stack, at the same time implementing some features that were very useful for data analysis and were only available in programming languages such as R. With pandas, Wes McKinney helped the Python community to rise on data analysis and data science fields. [16]

pandas' popularity is explained by Python's increasing influence in scientific areas that were dominated by programming languages like MATLAB, R, Stata, or SAS. Back in 2011, when McKinney made the article

mentioned, it was still difficult for some statisticians to choose Python over R, for example. By the date, unlike R and SAS which had a lot of tools to work with labeled data sets, Python had only a few that could provide a similar level of quality to work with these types of datasets. Looking at these major issues, McKinney focused on the development of this new framework, aiming to compete with the traditional tools that were market leaders.

pandas' author focused on providing equivalent functionality and, also, implement many new features, such as automatic data alignment and hierarchical indexing. These features were not readily available in such a tightly integrated way in any other libraries or computing environments until this date. pandas was initially developed for financial data analysis applications, however, the developers stated that pandas would enable scientific Python to be a more attractive and practical statistical computing environment, both for academic purposes and industry practitioners. [16]

The name **pandas** came from **pan**el **da**ta, a common term for multidimensional data sets encountered in statistics and econometrics.

Problems dealing with large datasets

Aside from pandas being one of the most popular Python frameworks to deal with Dataframes manipulation, the major flaw that is pointed to it is the difficulty to deal with big datasets. Once pandas uses in-memory computation, it is very usual and propitious to out-of-memory errors when dealing with large datasets. It is not easy to objectively rate a dataset as being a large dataset, once it depends on some factors and the tools we are using. However, it is given for granted that pandas starts to struggle to deal with datasets whose size starts to threaten memory use, using almost all the available memory.

This pandas' flaw can be explained by the non-use of distributed and parallel operations that are a "turning point" that exponentially improve the performance when handling large datasets. Following this idea of taking advantage and exploiting every advantage that distributed systems and parallel programming offer to us, lately, there has been an emergence of scalable Dataframe tools that aim to be faster and more efficient than pandas, at the same time that keeps pandas' simplicity and ease to manipulate data and work with Dataframes. pandas is only able to use one core at a time, which is a huge limitation that does not exploit the computation power of our devices. However, Dataframes nature and model allow to use of parallel programming and work with distributed Dataframes, providing solid results and performance gains.

To surpass the memory limitation that large datasets could bring to our devices there are tools that implement Out of Core algorithms, described as an algorithm that exploits the resources available on

external storage to support data volumes whose size cannot fit on primary memory. [17]

These algorithms are able to split the data into smaller chunks that will be processed separately, and possibly with parallel computing, merging all the results once every chunk is processed. This approach allows to processing of a large amount of data without the need to store it all at once in memory.

On the following topic, we describe some of the most used scalable Dataframe tools, detailing the way they work, how they handle big datasets, and, finally, the performance gains that are expected for each tool. All the mentioned tools focus on parallel computing principles or distributed systems and implement API's that are very similar to pandas, which ease the utilization of these tools and provide a quick adaptation and fast learning of its features by new users that were used to work with pandas.

2.4.4 Dataframes Benchmarking

Currently, there is a known absence of general and standard Dataframes benchmarks. Unlike database benchmarks, which have benchmarks developed by **Transaction Processing Performance Council** (**TPC-***) or **Yahoo! Cloud Serving Benchmark** (**YCSB**) there are widely used, there isn't such offer to better study or analyse Dataframe frameworks.

One project [18] developed by **H2O.ai** is seen as one of the most standard Dataframes benchmarks developed. Supporting more than 10 Dataframe frameworks, this tool tests them with 2 different operations, join and group by, and with datasets of 0.5 GB, 5 GB, and 50 GB. Although it only tests two simple operations, this benchmark is able to run a considerable amount of different frameworks and allows the user to interactively analyse all the obtained results with detail. This tool is a pioneer in Dataframe benchmarking and can be seen as a reference. Figure 8 is an example of how this tool shows the results obtained when executing the **join** operation on a **5 GB** workload.

Besides the tool presented above, there are some more custom and tailor-made benchmarks that evaluate some specific frameworks. For example, a tool [19] developed by Xinrong Meng and Hyukjin Kwon that analyses specifically Dask and PySpark frameworks. Although this tool only focuses on two frameworks, the set of operations that are tested is way bigger and more complex than the previous tool, testing both frameworks on local execution and distributed execution. Comparing results from complex arithmetic to statistics and standard deviations, this tool makes a complex and full evaluation of PySpark and pandas.

Another example of custom benchmarking [20] is one developed to test Modin. This tool was built to support Modin's advantages when compared with 3 other popular Dataframe frameworks (pandas, Dask, and PySpark). This benchmark focused on three major questions:

- Compared to existing Dataframe systems, including PySpark, Dask Dataframe, and pandas, how well does MODIN scale Dataframe operations over a large number of CPU cores?
- How much do the optimization techniques, eager data pipelining, and selective data exchange, reduce the execution time?
- What is the end-to-end performance of MODIN and how much does lazy type inference reduce the execution time?

As this tool was built specifically to test Modin and its features, it is not very flexible or scalable to other frameworks, providing concrete results about Modin's performance and improvements when compared with 3 other frameworks is short to make a deeper and more complex analysis about Dataframe frameworks in general.

There are, also, some web pages that report some *ad-hoc* benchmarking studies on some frameworks. For example, the benchmarking to high-performance pandas alternatives study [21] focuses on execution time performance and memory usage by pandas, Polars, Vaex, and Datatable. However, similar to previously described tools, this benchmark is not flexible, or scalable and represents a quite heterogeneous process to evaluate each framework, which does not ease the extension of the study to other tools, workloads, or environments.

Dataframe's energy consumption analysis is, as well, an area that lacks investigation and previous studies. For example, there is an article [22] that reports energy consumption for three Python Dataframe frameworks (pandas, Vaex, and Dask) that does not evaluate any complex operation or test different hardware configurations and environments. It is, also, a strict study that is not scalable or open to the addition of new frameworks. Besides the existence of some Dataframes benchmarking tools, these integrally focus on performance evaluation, discarding energy consumption and efficiency evaluation.

The previously described benchmarks are just examples of tools built for one specific goal, lacking the standardization that H2O.ai's tool offers. While database clients' benchmarks are quite consolidated on the market, with **TPC-*** and **YCSB**, for example, having complex and popular benchmarking tools, Dataframes benchmarking is a couple of steps behind. This can be explained by the fresh rise of new Dataframe tools, especially scalable Dataframe tools, that are on the market and being developed only a few years ago.

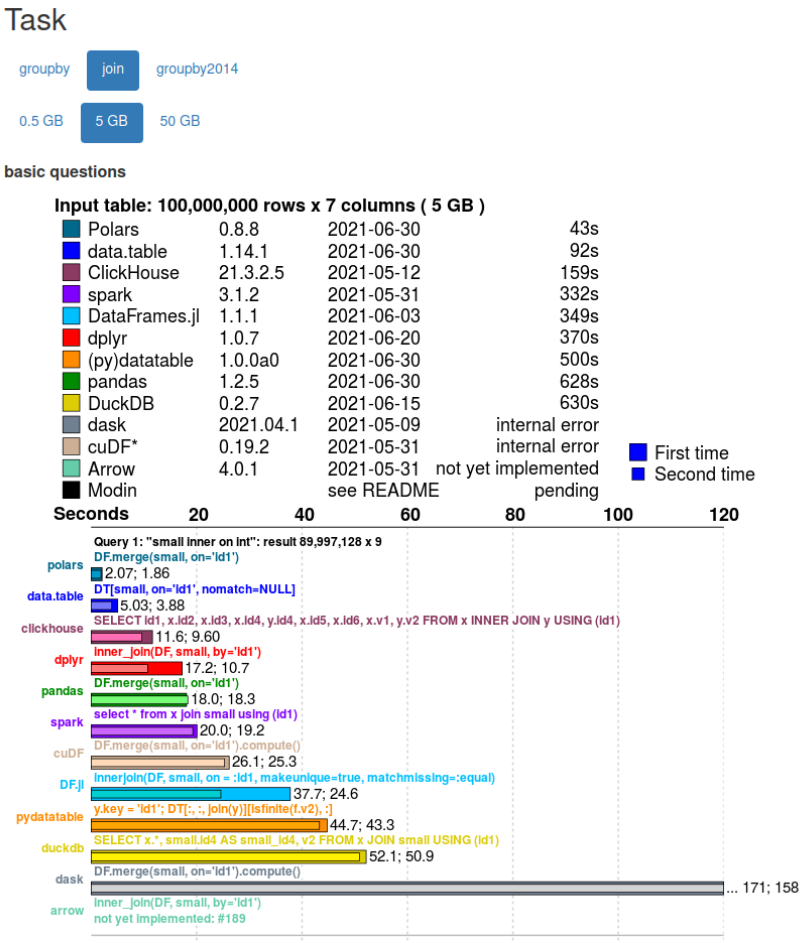


Figure 8: Database-like ops benchmark by H2O.ai [18].

Synthetic Data Bencharking

The benchmarking tools and studies that were presented previously execute their tests and evaluations with real data. Fuzzydata [23] is an open-source tool that is able to perform benchmarking tests on synthetic artifacts. This tool allows to use of workflows manually specified by the user or randomly generated, scaled, and replayed by the tool itself, producing a benchmarking analysis and comparisons between a group of Dataframe frameworks.

This tool implements an abstract model, that specifies artifacts, operations, and workflows in an implementation-agnostic manner. They describe an artifact as being an abstract representation of a Dataframe, giving flexibility to the tool to support several data access and manipulation API's.

Operations are described, also, as an abstract representation of one or more transformations that can be applied to one or more artifacts, with the result produced by these operations being a new artifact. Each operation has a model that consists of metadata, for example, the source artifact label, a list of

transformations and their respective arguments, and the new label that will be assigned to the new artifact produced.

Finally, Fuzzydata describes a workflow as being a directed acyclic graph (DAG) W = (V, E). In this graph representation, the vertices V represent the group of artifacts and the group of edges E is a representation of the operations that will be used to transform an arbitrary artifact V1 $\in V$ into another artifact V2 $\in V$. There are, also, artifacts with no incoming edges, being a representation of source artifacts that are loaded from disk or randomly generated by Fuzzydata itself.

Fuzzydata is very flexible and open to the addition of new frameworks and workflows. The tool is capable of rapidly scaling source artifacts with hundreds of rows to artifacts with millions of rows while having the ease of replaying the same workflow to all its implemented frameworks.

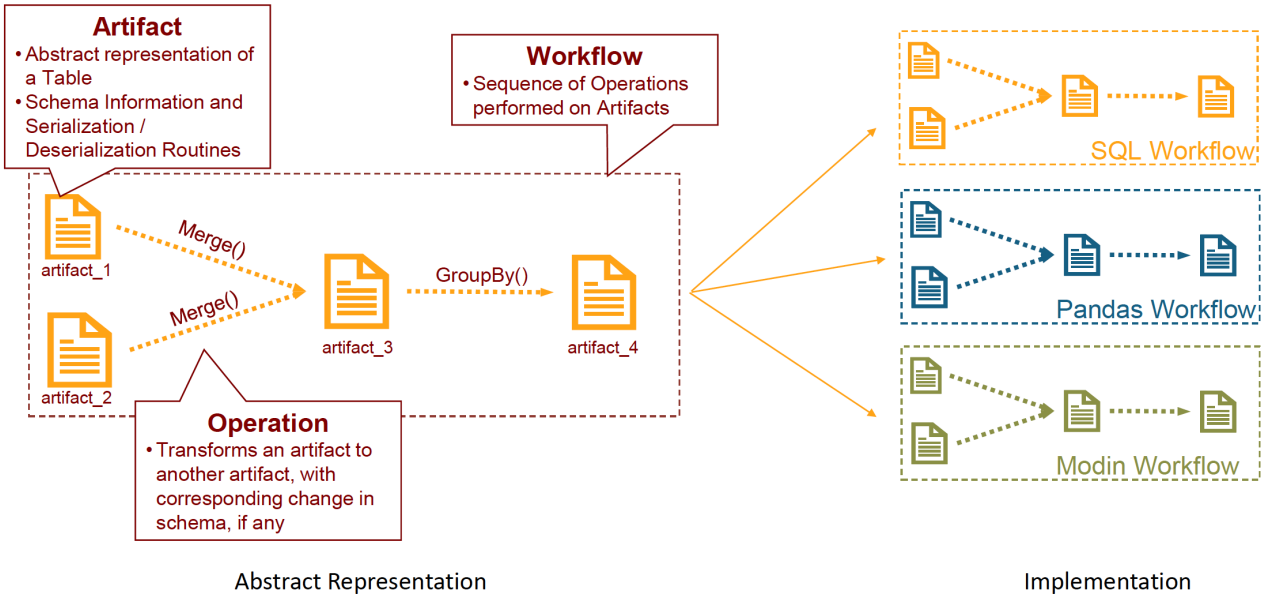


Figure 9: The design of FuzzyData.

2.5 Scalable Dataframes Tools

As targeted in the previous topic, pandas faces some difficulties when it needs to handle large datasets, mostly causing memory problems and limitations. Now, we present some of the most used and popular scalable Dataframes tools, built and developed in order to handle big datasets, using principles and paradigms of parallel computing and distributed systems. These tools, also, implement API's very similar to pandas, making it easy to use for new developers and making the dataset manipulation as easy as it would be using the pandas framework.

2.5.1 Modin

Modin is an open-source tool that labels itself as a drop-in replacement for the pandas framework. While pandas is known for being limited to a single-thread execution, Modin makes it a stronger point and features the ability to execute and speed up all workflows scaling pandas and making it use all available cores of a device. Modin was developed to work especially well when handling large datasets that pandas can not handle so successfully. [24] [15]

Modin offers Ray and Dask as main execution engines, and there isn't a need to specify the available hardware, once Modin does all that work. Also, it isn't necessary to specify how the data will be distributed. To use Modin it is only necessary to import its Python module to a previous program or notebook.

```
# import pandas as pd
import modin.pandas as pd
```

After importing the module we are already using all Modin features, keeping the pandas API. Modin's architecture is illustrated in the next figure.

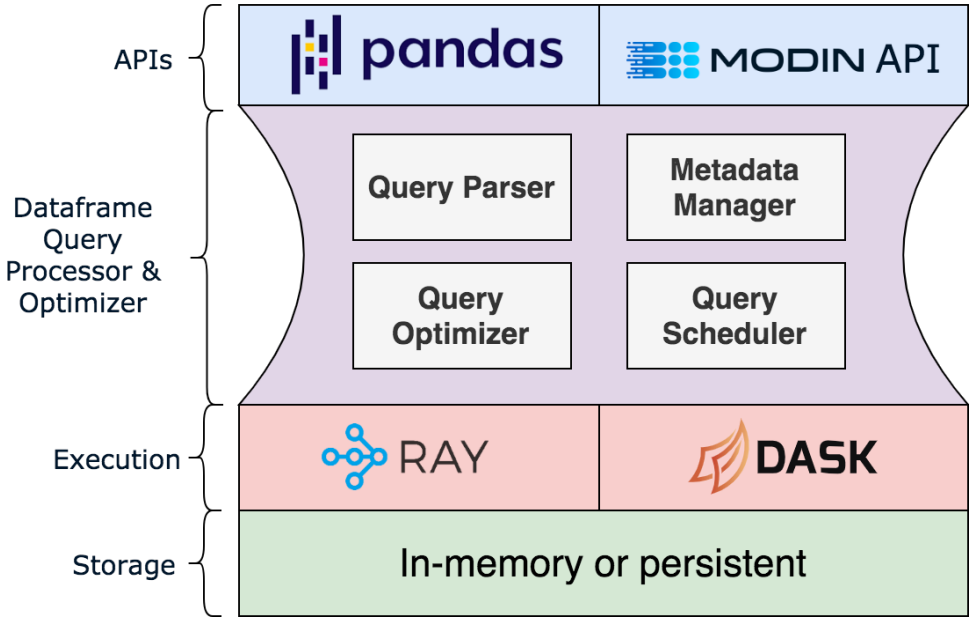


Figure 10: Modin architecture.

The Modin Dataframe is extremely light-weight and parallel, that will be distributed by all the physical cores available. This lightweight Dataframe allows speed-ups of up to 4 times the pandas' execution on a laptop that only has 4 physical cores. Modin, due to its scalable nature and lightweight Dataframes, offers good results and performance for datasets from 1MB to more than 1 TB.

Modin Backend Engines

Modin implements two major backend engines, Ray and Dask. Besides that, this framework is also able to work with more backend engines. Although Ray and Dask are the most stable and most used backend engines, Modin's developers are implementing more, in order to make this a more flexible tool and to offer advantages and features that other engines cannot.

- **Dask: [25]** Open-source that implements parallelism to the already existing Python stack. This tool eases the integration in existing Python code while being able to scale up to 1000+ cores cluster, locally or in the cloud. This backend engine is more detailed on further topics.
- **Ray: [26]** Unified application built to scale Python applications. This tool allows you to scale any Python application from a laptop to a cluster, without the need for other platforms. Ray Core implements three key abstractions:
  - **Tasks:** Stateless functions that are executed on the cluster.
  - **Actors:** Stateful worker processes created at the cluster.
  - **Objects:** Immutable values accessible across the cluster.

2.5.2 Vaex

Vaex is an open source framework [27] that is built for lazy Out-of-Core (set of techniques that allow the user to manipulate data that is larger than the available RAM by reading chunks from the disk) Dataframes to handle big tabular datasets. Implementing an API very similar to pandas, the framework can calculate the main statistics such as mean, sum, count, and standard deviation, on an N-dimensional grid up to a billion objects/rows per second.

Vaex can handle large datasets pretty well, even if they are bigger than the available memory of the device. This is achieved by using techniques like memory mapping and lazy evaluation. Vaex uses files with memory-mappable formats like Apache Arrow and HDF5, and its only limitation is the existing free space on disk. Through these techniques, allied with Out-of-Core algorithms, Vaex can load smaller chunks of data to memory, taking advantage of caching procedures when the same chunk of data is accessed constantly, improving the performance of some operations, once it's no longer necessary to always read the data from disk, which is a much slower process.

Vaex has a zero memory copy policy, which means operations like filtering a Dataframe do not copy the data and cost very little amount of memory. For example, if we create a Dataframe df2 to filter the

Dataframe df1, df2 will not take any extra memory, because it is a shallow copy of df1. A shallow copy is a type of object copy whose properties share the same references as the original Dataframe, which means it points to the same underlying values. Vaex creates a binary mask when df2 is created, that will be later applied to df1 Dataframe. The estimated memory cost for filtering one billion rows Dataframe is only nearly 1.2GB RAM, much lower than some traditional tools.

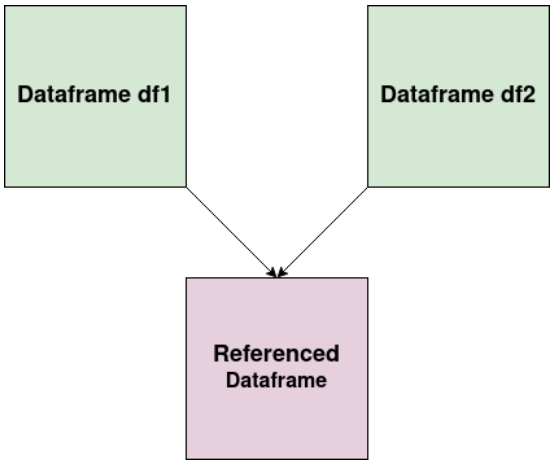


Figure 11: Dataframe shallow copy.

Another interesting feature of this framework is the creation of virtual columns when transforming Dataframes columns into new ones. These columns are just like the normal ones, but take up no memory, for the reason that Vaex only knows the expression that defines the column but does not know the values upfront. This process lazy evaluates the columns only when necessary, keeping the memory usage low.

Vaex, if possible, will use all available cores on the device by using the Python Module *concurrent.futures*, which allows to run tasks in multiple threads or processes concurrently. Vaex can also be used with distributed computing frameworks like Dask, for example. When using Vaex and Dask, it is possible to parallelize data through a cluster of multi-core machines, which is extremely useful in improving the performance of big data processing.

Finally, Vaex is an extremely fast tool and has solid performance results. Operations and column methods implement very efficient algorithms, all implemented in C++. All of those work in Out-of-Core, allowing us to process more data than our RAM size grants while using all available cores.

2.5.3 PySpark

PySpark is an open-source tool that provides a similar pandas API and allows it to work with Dataframes on Apache Spark. Built to work with big data and large datasets, PySpark makes it easier to work on

distributed environments by using a familiar pandas API. This way, it isn't necessary to learn an all-new framework or to write complex and hard-to-understand distributed code, and we can still manipulate and analyse huge amounts of data and very large datasets. [28]

Talking about Apache Spark [29], the open-source platform where PySpark is built on top, was developed for large-scale data processing. Due to its distributed nature, Spark can run on a cluster of several machines, making the processing of large datasets more quick and efficient. This platform is widely used in machine learning, Big Data analytics, and data science. Written in Scala programming language, its in-memory compute engine can process data much faster than the traditional MapReduced-based systems. It is also highly scalable and easy to deploy on a cluster with multiple machines, making the workload distribution for the available resources and processing its data in parallel, presenting pretty good results even when working with datasets with terabytes of size. [30]

Just like the previous scalable tools, this one presents itself as a drop-in replacement of pandas, allowing the users to use very similar code that they would use with pandas, but with the advantage of their code being run on Spark and using its distributed features and capabilities, making the transition from working singly with pandas to work with Apache Spark more smooth and easier for the data scientists.

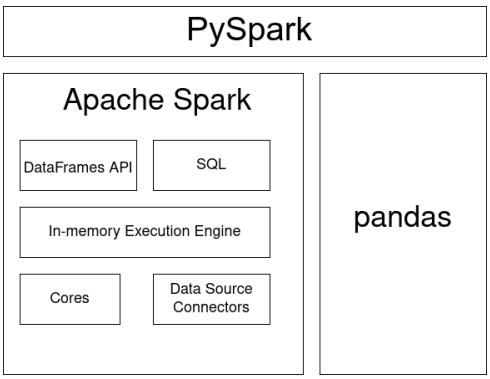


Figure 12: PySpark architecture.

2.5.4 Bodo

Bodo is a tool that offers supercomputing performance and scalability to native Python analytics automatically. Due to its just-in-time inferential compiler, Bodo presents itself as an improvement to basic Python analytics just like pandas or NumPy. This tool allows the programmer to use the native pandas or Numpy API's. It also offers extreme performance improvements and scalability, due to the use of true parallelism operations and some advanced compiler technologies. [31]

Clarifying now what is a **Just-in-time** (**JIT**) compilation and how Bodo handles all the workflow, it is important to first demonstrate how easily Bodo uses this feature. Without the need to alter any piece of pandas' Python code, Bodo, by simply using the @bodo.jit decorator, transforms the previous Python function into a Dispatcher object. After the first call of the Dispatcher object, Bodo compiles the function, but, afterwards, the function will only be recompiled if there is a call of the same function but with different arguments. In the next example, we can see how easy it is to apply the **JIT** compilation workflow with pandas' API.

```
@bodo.jit
def bodo_load(file_path):
    return pandas.read_parquet(file_path)
```

Bodo is responsible for transforming simple Python functions for parallel execution and uses **Message Passing Interface** (**MPI**), following the **Single Program Multiple Data** (**SPMD**) paradigm to do so. Using this parallel execution model, the Dispatcher object does not launch processes or threads on the fly. Rather, all processes are created and launched at the beginning and run the same file with the mpiexec command. Bodo, using the @bodo.jit decorator, will distribute all data across all available processes. Each process will run the exactly same code on some chunk of data, with Bodo handling the communication between processes as necessary, thanks to MPI.

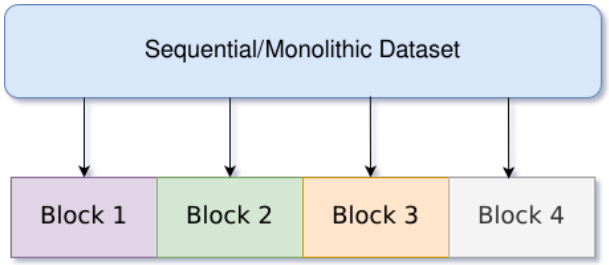


Figure 13: Bodo chunk partition. [31]

As there is some data that will not be divided into chunks, we can identify two different data distribution schemes:

- **Replicated:** Variable associated data is the same on every process.
- **One-dimensional:** Data divided into chunks, split along one dimension. In the case study, we can give the example of a row from a Dataframe.

The data distribution schemes are automatically chosen by Bodo, allowing the parallelization for any given number of cores or cluster size, without the need for an extra API layer.

2.5.5 Dask

Dask is an open-source built to provide parallelism to the already existing Python stack, providing integration to frameworks like pandas, NumPy, or scikit-learn and allowing parallel execution across several cores, processors, and even computer, without the need to learn a new language or a new framework. Dask's ease of natively integrating existing Python code and scaling up to 1000+ cores cluster, locally or in the cloud, without the need to rewrite code is the strongest feature that offers us. [32]

Dask can be divided into two distinct parts:

- **Collections API for parallel lists, arrays, and Dataframes:** Responsible for natively scaling the Python stack to run in larger-than-memory or distributed environments. Dask collections are the parallel version of the underlying framework, for example, a Dask Dataframe consists of a parallel Dataframe composed of many smaller pandas Dataframes, split along the index. These collections run on top of the task scheduler.

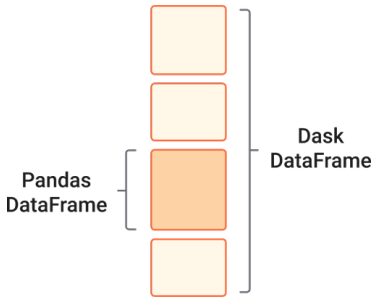


Figure 14: Dask Dataframe.

- **Task scheduler:** Responsible for building task graphs and coordinating, scheduling, and monitoring tasks optimized for interactive workloads across CPU cores and machines. The scheduler can work with a single machine or with multiple workers.

  When using multiple workers, the scheduler coordinates all of them and moves the computation to the correct worker, while keeping a fluid and non-blocking conversation. This approach can work with **HDFS** file systems and with cloud object storage, such as Amazon S3. This also allows to have several simultaneous users on the same system.

  In a single-machine scenario, the scheduler was developed and optimized for large-than-memory

use, being responsible for tasks handled through different processes or threads, making this a low-overhead approach.

Dask is a complex tool yet it is very easy to use, requiring little or no configuration at all. It is a resilient framework, handling possible worker failures gracefully and without compromising all the other workers. Its elasticity is also a good add-on, making it possible to take advantage of new nodes added on the fly, and presents good performance results, operating with low latency and low overhead.[25]

2.5.6 Ibis

Ibis [33] is an open-source framework built for data analysis that ensures a full Python workflow on top of some big data systems. Using this tool, developers can still use Python for data analysis while working with big data systems, removing some of the limitations that Python, more specifically pandas, has when dealing with large datasets.

Ibis can be divided into two major components:

- **Dataframe API for Python:** Python developers can write Ibis code in order to manipulate tabular data like Dataframes.
- **Deferred execution:** Ibis pushes the code execution to the query engine, allowing users to execute code at the speed of the chosen backend, not the local machine.

Ibis developers aim for their tool to be a future-proof solution to interacting with data using Python. This tool's main features are:

- **Familiar API:** Ibis' API is very similar to popular API's like pandas or dplyr.
- **Consistent syntax:** Universal Python API for tabular data, such as Dataframe. It is not related to or affected by the dataset's size.
- **Deferred execution.**
- **Interactive mode:** Allows users to quickly identify problems and bugs, do explanatory data analysis, and mock-up workflows locally.
- **More than 10 supported backends.**

- **Minimize rewrites:** Most of the Ibis code can be kept the same, whether the developers make some changes on the chosen backend, like increasing or decreasing computing power or, even, switching backend engines.

Ibis most used practical cases are:

- **Speed up prototype to production:** Scale to distributed systems with minimum rewrites.
- **Boost performance of existing Python code:** Due to its deferred execution, Ibis will significantly improve pandas' code performance, having a much faster execution. Ibis also eases the switching process to a faster database engine, requiring minimal changes to the written code.

2.5.7 DuckDB

DuckDB [34] is an open-source high-performance analytical database system, designed to support analytical query workloads, characterized by complex and long-running queries, responsible for processing big portions of stored data, such as joins between several large tables.

To have efficient support for this kind of workload, DuckDB has a columnar-vectorized query execution engine, in order to reduce the amount of CPU cycles that are expended in each individual value. Using this technique, queries are still interpreted but a large set of values, a vector, is processed in just one operation, greatly reducing the usual overhead existing in tools like PostgreSQL or SQLite, which process every single row sequentially. Vectorized query execution presents far better performance and faster results.

DuckDB presents no external dependencies, both in run-time and compilation. There is no DBMS server software to install or maintain, once DuckDB runs entirely embedded within a host process. This makes data transfer from and to the database much faster, making it possible, in some cases, to process foreign data without copying. Giving pandas as an example, DuckDB can run queries directly on pandas without importing or copying any data.

Once DuckBD is an embedded **DBMS**, the database and the client application will share the same address space. This way, the chunk is handed over without requiring a copy. Appending data to tables works the same way, with the client application filling chunks with its data. Once filled, they will be handed over to DuckDB and appended to persistent storage. All APIs are built around bulk value handling, in order to prevent function call overhead from becoming a bottleneck. The bulk API offers efficient data transfer to and from R and pandas or Numpy. [35]

2.5.8 Polars

Polars [36] is an open-source tool, implemented in Rust, that uses Apache Arrow Columnar Format as a memory model. This tool aims to improve performance on Dataframe management, using all available cores to achieve it. Polars was designed for the parallelization of Dataframe queries, reducing redundant copies and transverse memory cache efficiently.

Polars is lazy and semi-lazy, allowing it execute most of the work in an eager way, similar to pandas, but also providing an expression syntax that will be optimized and executed on the query engine. This lazy Polars' approach allows us to optimize the whole query, improving performance and memory pressure. Polars handles queries on a logical plan, that is reordered and optimized before running the query. All available work will be distributed to different executors, that implement the eager API. Once the whole query is known to both the optimizer and the executors, processes that have separate data can be parallelized on the fly.

Polars implements **Single Instruction Multiple Data** (**SIMD**) principles and is written in Rust, being capable of running several operations in parallel and, just like explained on PyArrow, using Apache Arrow to store data, which is much more efficient than pandas' data storage. Polars is driven by the reduction of redundant copies, traverse the memory cache efficiently, and minimize contention in parallelism.

Finally, on Polars' official page [36] , there is a list of the features implemented:

- **Copy-on-Write** (**COW**) semantics
- Appending without clones
- Column-oriented data storage based on Apache Arrow
- **SIMD** vectorization

2.5.9 Datatable

Datatable is an open-source tool to manipulate tabular data and Dataframes. Inspired by the R library *data.table*, Datatable supports out-of-memory datasets, multi-threading data processing operations, and simple API. This tool started back in 2017, aiming to perform big data (up to 100GB) operations on single-node machines, trying to achieve the maximum speed possible. This tool focuses on the needs of modern machine-learning applications, once they require to process large datasets and perform complex operations. The first official user of Datatable was the project Driverless.ai, a project designed to use AI to make AI, with automation encompassing data science best practices across key functional areas.

On Datatable's official GitHub page [37] we can see a set of features that this tool aims to implement, such as:

- Column-oriented data storage.
- Native-C implementation for all datatypes, including strings. (pandas does not do that for strings).
- Support for date-time and categorical types.
- All types should support null values, with as little overhead as possible.
- Data should be stored on disk in the same format as in memory.
- Work with memory-mapped datasets to avoid loading into memory more data than necessary for each particular operation.
- Multi-threaded data processing.
- Efficient algorithms for sorting/grouping/joining.
- Minimal amount of data copying, copy-on-write semantics for shared data.
- Interoperability with pandas / numpy / pyarrow / pure python.

2.5.10 RAPIDS

RAPIDS [38] is a group of software libraries, developed by NVIDIA, that implement NVIDIA CUDA optimizations, low-level advantages, and GPU parallelism while using a user-friendly API. These libraries also feature multi-node and multi-GPU programming, allowing them to deal with and handle huge volumes of data. RAPIDS have different libraries that were built for different goals, for example:

- **cuDF:** Dataframe manipulation library based on Apache Arrow.
- **cuML:** Set of Machine Learning libraries that implement GPU version of algorithms from scikit-learn.

cuDF

cuDF is a Dataframe manipulation library based on Apache Arrow [39] [40], that implements a pandas-like API. This library was developed to improve the overall performance of core operations, such as data loading, filtering, and manipulation, both for data preparation and model training. This library is seen as

Table 2: Examples of Python frameworks for scalable data science

Name	Out-of-core	Multi-Node	Lazy Evaluation	GPU
Modin	✓	✓	✓	
Vaex	✓		✓	
PySpark		✓	✓	
Bodo	✓	✓	✓	
Dask	✓	✓	✓	
Ibis	✓	✓	✓	
DuckDB	✓		✓	
Polars			✓	
Datatable	✓		✓	
RAPIDS		✓		✓

one of the most complete and versatile, allowing a fast and easy adaptation from pandas' users, due to its similar API.

2.5.11 Frameworks Comparison

Compared to relational databases, Dataframe frameworks are not optimized or well-suited to every workload. It is possible to identify that different tools perform better to a given data set or group of operations, once every framework applies different trade-offs, as Table 2 illustrates. The principal known trade-offs are the following:

- **Out-of-core** Some frameworks focus on out-of-core parallelization of big volumes of data not limited to available memory, using persistent storage.
- **Multi-Node** Provide support for a complete analysis pipeline in a distributed environment, either based on MPI for more HPC computing or DAG dataflows for more traditional Cloud computing.
- **Lazy Evaluation** Lazy evaluation of dataframes is used to create expression trees for execution on a backend. Nonetheless, there is potential for further optimization through the use of lazily constructed expressions, such as caching and automatic backend selection.

Even with different trade-offs applied, all frameworks are developed with common goals. To facilitate data treatment and manipulation of huge data volumes, the implementation of familiar and intuitive API's,

overcoming resource limitations, and achieving performance gains are examples of objectives that all scalable frameworks aim to achieve.

Chapter 3

Benchmark Design and Implementation

In this chapter we will detail our benchmark's architecture, requirements and how is it deployed and implemented. We will describe the requirements that were the base of the development of our tool, being a solid guideline to build a cohesive tool that is capable of answering the needs that we identified through the study of another benchmarking tool. We will also illustrate and detail how is our tool built and how is it capable of producing the results for performance and energy consumption analysis.

3.1 Requirements

Following an analysis of existing Dataframe benchmarking tools and what are the main objectives that we aim to achieve with our tool, we formulate a set of requirements that our benchmark must achieve and be able to implement. These requirements answer to the lack identified on existing tools while ensuring that our benchmark is a solid and reliable tool, capable of producing quality results. The requirements are the following:

1. **Flexible Workloads Implementation:** Set of operations that range from the most simple arithmetic operations, executed individually, to complex chained execution of operations. This way, our benchmark covers a set of operations that are widely used and recreates a scenario that is commonly used in real data science studies.

2. **Real and synthetic artifacts:** Incorporation of workloads composed of real data and workloads composed of synthetically produced data.

3. **Extensible to new frameworks additions:** Simple addition of new frameworks to our benchmarking tool, making it flexible and with a homogeneous implementation for any framework. Unlike *ad-hoc* benchmarks, this tool can be extended to any framework with ease.

4. **Simple usability:** Simplicity of deployment and execution of our tool on distributed environments, both on HPC and on more traditional environments such as Cloud or *on-permise*.

5. **Statistics collection:** Perform statistics collection from performance and energy consumption, with non-intrusive tools.

6. **Metrics visualization and analysis:** Using graphics and other data analysis tools.

7. **Reproducibility:** Making constant evaluations and repetition of tests, in order to achieve solid and reliable results.

8. **Baseline comparison with Pandas:** Once Pandas perform in sequential environments, this will be our baseline comparison to any other distributed framework.

9. **Scalability:** Streamline the process to run frameworks single node with a variable number of cores and run framework multi-node with a variable number of machines, as well as increasing the data sets used.

3.2 Benchmark Architecture

In Figure 15 we illustrate the architecture of our benchmark and how is it structured. Following, we will describe each main component and what is their role in the whole benchmark.

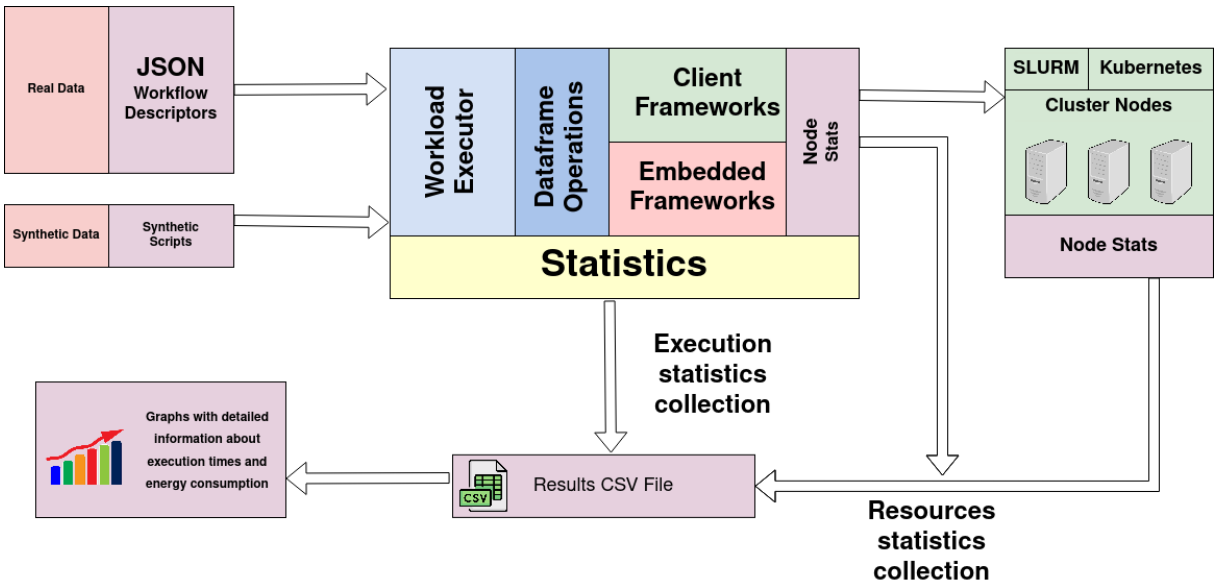


Figure 15: Benchmark Architecture.

- **Workflow Descriptors:** We use JSON files as workflow descriptors. Present on these files is the information necessary for the workload executor to perform a benchmark execution, such as the workload type, which operations will be executed, how many runs will be performed, and which dataset will be loaded.
- **Client Frameworks:** These frameworks, like Dask and Bodo, will be executed on external nodes, with the client behind executed on our tool.
- **Embedded Frameworks:** These frameworks are executed on our benchmark itself, being embedded into our tool, running as a library.
- **Results CSV files:** All results (from a given workflow descriptor) will be saved in two files, assembling all information about execution times, performance, and energy consumption metrics. Statistics collection can be made from workers and from our tool, using non-intrusive software applications.
- **Information display:** Using Python's library **Matplotlib**, we will display all results contained on the results CSV files in graphics that allow us to compare frameworks, execution times, energy efficiency, and scalability. These graphics allow an intuitive and simple comparison between all frameworks.

3.3 Phases

Our benchmark has 5 main phases: **Deployment Phase**, **Artifact Generation Phase**, **Loading Phase**, **Execution Phase**, **Results Analysis Phase**.

3.3.1 Deployment Phase

To initially set up our tool and all frameworks implemented, it is necessary to create virtual environments to facilitate the general execution of our benchmark. Once we are executing various frameworks with different dependencies and packages that have to be installed, every framework will have its own virtual environment (Conda or Python) where will be executed. This way, it is possible to have a clear separation between all frameworks and it is easier to solve any possible dependency conflicts or installation errors that might occur during the first setup of our tool or during the addition of a new framework.

This phase is responsible, in addition, to starting the non-intrusive tools that will measure the energy consumption statistics. Once we run external tools to perform the statistics collection for energy consumption, this phase will start the respective processes and micro-services necessary.

3.3.2 Artifact Generation Phase

Our benchmark will work with artifacts of two types, **real data artifacts** and **synthetic data artifacts**, as pointed on requirement 2. These artifacts are represented by Parquet files.

The real data artifacts are obtained through the download of **New York City Taxi and Limousine Commission** (**NYC TLC**) datasets. The benchmark users can define the timeline of data that want to be downloaded, in order to produce and execute our benchmark with datasets from different sizes. For example, if a user wants to work with complete years of data, which will produce a file with nearly 9 GB of raw data, he just has to define which year and month they want to start and which year and month they want to finish.

The synthetic data artifacts will be generated with the help of Fuzzydata. To work with this type of artifact, the user will select how many rows and columns they want to randomly generate for the base artifact, where further, and also randomly generated, operations will be executed. The artifacts synthetically generated are also represented by Parquet files.

Offering the execution with two different types of artifacts helps to ensure a more flexible workload and workflow implementation, perform more embracing and wide tests, and allow the benchmark user to explore different use cases from one unique benchmark.

3.3.3 Loading Phase

Once the artifacts are downloaded, or generated, they are ready to be used by any framework to create Dataframes. This phase works similarly for both micro and macro-benchmarking executions, and the load operation in any framework is equal for all implemented frameworks.

When working with real data, every framework will load the Parquet file and create a Dataframe from it, which will be used for every further operation performed. It is during the execution of this phase that **Client Frameworks** will start their respective client processes and create the necessary workers and similar processes for the framework execution, as it happens, for example, with Dask. In some cases, like Modin, during this phase is performed the Dataframe partition.

For synthetically produced data, this phase will use the base artifact produced during the random artifact generation, with the number of rows and columns defined by the benchmark users. This base

artifact can be scaled up, with no effect on the loading phase itself.

3.3.4 Execution Phase

After the loading phase is completed, each framework is ready to perform the set of operations with the Dataframe that was created in the previous phase. This phase can be split into the two workload types that our benchmark supports, **Micro-benchmarking** and **Macro-benchmarking**. These two workload types allow a more general and embracing analysis, once the user of our benchmark is able to both perform a study of each operation individually, with Micro-benchmarking, and to perform an analysis of a workflow composed by a group of operations that will be executed consecutively, with Macro-benchmarking.

During this phase, we have frameworks that are embedded and will be executed on the benchmark itself, running as a library and the statistics collected will be from the machine where our benchmarking is running. We also have implemented client frameworks, that will be executed with the aid of external nodes, implementing multi-node computing. In this case, we will have statistics collected from the node where our benchmarking is running and from the set of nodes that will be used during the execution of those frameworks.

At the end of the execution of both workload types, the results generated are written on CSV files, saving the reports from performance and energy consumption.

Micro-benchmarking

On this workload type, which works exclusively with real data, our benchmarking tool aims to evaluate each operation isolated, obtaining results for each framework and for a given set of pre-defined operations. This mode is open to the addition of new operations and we have implemented the following 7 operations, representing a basic core of Dataframe analysis operations:

- **Mean:** Mathematical mean of a given column
- **Sum:** Summatory of a given column
- **Unique Rows:** Counts of unique values of a given column
- **Group By:** Grouping the data according to one category and applying a function to it
- **Multiple Group By:** Grouping the data according to 2 or more categories and apply a function to the categories

- **Join:** Outer join the Dataframe created from the **NYC TLC** with a Dataframe manually created.
- **Sort:** Sort a given column

Macro-benchmarking

This workload type can be executed with both real and synthetic data. During this execution, our benchmark will evaluate the performance and energy consumption of a workflow, that is composed of a group of operations to be executed. Contrarily to micro-benchmarking, here our benchmark will produce results from a chained execution of the operations given, reporting execution times for the all workflow as a whole, and not for every operation individually. The loading time is taken into account for the whole execution time, however, for energy consumption metrics, the benchmark reports the total energy consumed per operation executed (J/Operation), except the loading operation.

With real data, Macro-benchmarking is able to perform all the operations implemented on the Micro-benchmarking mode, with the advantage of allowing the benchmark users to select the arguments to pass to each operation. While on Micro-benchmarking, the users select which operations they pretend to evaluate, with fixed arguments, on this mode the user references which operations they want to execute, on which artifact they want the operation to be executed and what are the arguments to be given to each operation.

When working with synthetic data, the workflow and, consequently, the group of operations to be executed will be randomly generated on the artifact creation. However, the users can select the number of artifact versions they pretend to generate, as well as the number of rows and columns of the base artifact, the number of operations to be executed before producing a new artifact, and the workflow branching factor. These features are executed with the incorporation of Fuzzydata. Although the energy consumption data will be saved identically to Micro-benchmarking, the performance data will be saved on a different format CSV file that has the elapsed time since an artifact generated another artifact. For example, considering N as the number of operations to be executed before producing a new artifact, in a case with N = 5, the CSV file will have an entry for the elapsed time since Artifact Y was generated from Artifact X from the execution of 5 random operations.

3.3.5 Results Analysis Phase

Once all desired workflows finish their respective executions, we are able to generate graphics that will help us to analyse, compare and understand the different tests executed. Using Matplotlb Python library

to generate these illustrations, the main goal is to produce information that rapidly allows us to compare different frameworks executing datasets of different sizes on different workload types. Having a direct and side-by-side comparison between a group of frameworks is an easier and more intuitive method to understand how the different tools behave in a given scenario, in this case represented by a given workflow. This approach is also a better representation for a scalability analysis, once we picture the results for datasets, or synthetic artifacts, of different sizes when executed on the same framework and compare them with all the remaining frameworks.

3.4 Implementation

Our benchmarking tool is managed and operated through a workload executor, responsible for executing the tests and executions pretended. Parallel to the execution of the workload executor actions and the implemented frameworks are running the tools to measure the power consumption. To orchestrate all these steps and streamline all the required processes to obtain both performance and energy consumption data, we defined a group of scripts that will be responsible for executing all the desired actions:

- The **setup** script creates all the necessary Conda and Python Virtual Environments and installs all the necessary Python dependencies to execute each framework. This script also downloads Parquet files that will be used by the frameworks.
- The **run** script that will execute workflows for real data, being responsible for operating each framework on its virtual environment and executing the power consumption measurement tool
- The **plot** script will generate all the graphs from the information contained in the results CSV files.
- The **synthetic_scripts**, used to work with synthetic data. These scripts are responsible for the random artifact and workflow generation, artifact scaling, and workflow execution. Synthetic data does not require the utilization of a workflow descriptor file, with all the necessary arguments being passed directly to the workload executor.

3.4.1 Workload Executor

The workload executor will work as a core element of our benchmark, being responsible to organize and execute the frameworks and workloads that are described on the workflow descriptor. When dealing with real data, there are 4 arguments that the workload executor requires in order to execute a given workload:

- **Type:** The type of workload that will be executed. We have implemented the Micro-benchmarking and the Macro-benchmarking types.
- **Dataset:** The dataset that will be analysed and manipulated.
- **Runs:** The number of runs that will be executed. If omitted, the default number of runs made by the benchmarking tool is two, in order to grant requirement 7, **reproducibility**. In micro-benchmarking mode, each operation individually will be executed for the number of runs that are defined, while in macro-benchmarking mode the chained operations are consecutively executed with the defined number of runs.
- **Operations:** The group of operations that will be executed. There are 2 different ways to pass the set of operations. When executing on Micro-benchmarking mode, the set of operations is a list containing the name of the operations that are implemented, with no arguments defined by the user. When executing on Macro-benchmarking mode, the set of operations will be a list of operations and associated arguments to be executed, defined by the user. This way we grant a **flexible workload execution**, as explained on requirement 1.

The workflow descriptor files are represented by JSON files, Python software can easily parse a JSON file and they are simple to be created by the common user. Knowing the required arguments that must be on the file, the benchmark users only need to create the JSON file to represent a workload they intend to evaluate, granting a **simple usability** to our tool, requirement 4, with easy deployment and executing processes. Here are two examples of JSON files that can be used on our benchmark:

Example of JSON workflow descriptor for Micro-benchmarking:

```
{
    "type": "micro",
    "dataset": "yellow_tripdata_2019_01-2019_12",
    "runs": 4,
    "operations": ["mean", "sum", "unique_rows", "groupby", "multiple_groupby"]
}
```

Example of JSON workflow descriptor for Macro-benchmarking:

```
{
    "type": "macro",
```

```
    "dataset": "yellow_tripdata_2019_01-2019_4",
    "runs": 4,
    "operations":
        [{
          "op": "sort",
          "artifact": "source",
          "label": "artifact_1",
          "column": "tip_amount"
        },
        {
          "op":"groupby",
          "artifact": "artifact_1",
          "label": "artifact_2",
          "args":{
              "group_columns": [
                "passenger_count"
              ],
              "agg_column": "tip_amount",
              "agg_function": "mean"
            }
          }
    ]
}
```

The results produced by the N number of runs will be reported with a warmup execution time and an average execution time. The warmup time reports the execution time relative to the first run, once, normally, this execution time can lead to some misunderstanding of the results due to systems initialization or similar processes that affect the execution time. To have a more precise and reliable execution time, we report the mean of the execution times from all N runs.

3.4.2 Synthetic Data Workflows

When working with synthetic data, the interaction with our benchmarking tool is slightly different. In this use case, there is no need to define workflow descriptors through JSON files, however, there are a few arguments that must be passed to the executor in order to generate random artifacts and workflows:

- **Workflow Client:** The framework that will be used to execute the workflow.
- **Columns:** The number of columns to generate on a base artifact.
- **Rows:** The number of rows on a base artifact.
- **Versions:** The total number of artifacts that will be generated.
- **Workflow Branching factor:** Oscillating between 0.1 and 100, where 0.1 represents a linear branching factor and 100 is a star-like branching factor.
- **Materialization frequency:** The number of operations to be executed before generating a new artifact.
- **Replay Directory:** Optional and used to replay the same workflow to all different frameworks.

With the above arguments given, our benchmark will use Fuzzydata to generate all the synthetic artifacts and workflows from the information detailed. Working with this type of data allows us to quickly generate several different and wide workloads and, also, offers fast scalability for the artifacts generated.

Synthetic data utilization gives support to the results obtained for real data testing. Testing the same framework with both real and synthetic data, and comparing the results achieved, confirms the behavior observed with both artifacts and helps to test a framework with a vast variety of operations and artifacts, producing more complete and extensive results.

3.4.3 Frameworks

The flexibility of our benchmark, as stated on requirement 3, and its homogeneous implementation for any framework make the addition of new frameworks a simple process. We have implemented a total of 10 scalable frameworks in our tool, as represented in Table 3.

The majority of the frameworks have implemented both workload types and have a similar API when compared to pandas' API. There are some exceptions, such as DuckDB which has a unique implementation, once it requires the execution of SQL queries. Datatable also has some API differences when compared to pandas'. Dask, Modin, RAPIDS, and PySpark are able to perform in multi-node environments.

The Dask framework was implemented with LocalCluster, for single-node execution, and SLURMCluster for multi-node execution. For synthetic workflow execution, we implemented Modin (Ray and Dask backend engines), Polars, and RAPIDS.

Framework	**Micro benchmarking**	**Macro benchmarking (Real|Synthetic)**	**Multi Node**	**pandas' API**	**Observations**
Modin	✓	✓ | ✓	✓	✓	Ray and Dask backend engines.
Vaex	✓	✓		✓	
PySpark	✓	✓	✓	✓	
Bodo	✓	✓		✓	
Dask	✓	✓	✓	✓	
Ibis	✓	✓		✓	Pandas was used as the selected backend engine *
DuckDB	✓				API much different from pandas' Works with SQL queries
Polars	✓	✓ | ✓		✓	
Datatable	✓	✓			API with some differences when compared to pandas'
RAPIDS	✓	✓ | ✓	✓	✓	Multi-node execution using Dask

Table 3: Scalable frameworks implemented

Notes: The implementation of more Ibis' backend engines implies code modifications.

3.4.4 Graphics

After a given workflow's execution is completed, we have all the results and statistics saved on CSV files. However, this is not a readable and easy way for a common user to understand and analyse the results produced, once it has a lot of dumped information that seems to have no correlation or scientific context.

So, to accomplish requirement 6, we used Matplotlib Python's library to produce graphics from the results' files generated, making the frameworks' analysis and comparisons an easier, more understandable, and quicker task.

The main objective of this data visualization procedure was to ensure an intuitive form of comparison between all frameworks executed on a workflow. For the micro-benchmarking workload type, we produced a graph with execution times for every individual operation executed, for the loading phase, and for energy consumption, representing the total energy consumed by each framework. For the macro-benchmarking workload type, we produced graphs for the loading phase, similar to micro-benchmarking, for the execution time of the set of operations contained on the workflow descriptor and for energy consumption, representing the energy consumed by the operation executed. On both workload types, these graphs also had the information from two simultaneous workload dataset sizes, in order to be able to make an analysis of the frameworks' behaviour when the dataset's size increases.

As requirement 8 refers to the baseline comparison with pandas' performance and energy consumption, the graphs enable that analysis, once pandas' performance and energy consumption are represented side to side to scalable frameworks. An example of such a figure is shown in Figure 16.

3.4.5 NYC TLC Dataset

To handle real data operations we chose a well-known dataset that is used widely through the data science community. The **New York City Taxi and Limousine Commission** (**NYC TLC**) datasets are a suitable option, once they offer a big volume of data, making it possible to produce heavy workloads, and it is considered a dataset relatively clean.

This dataset is composed of a total of 19 columns, representing a variety of information such as pick-up and drop-off locations and datetimes, itemized fares, rating and payment types, passenger count, and the trip elapsed distance. The columns can have four different data types, int, double, string, and timestamp.

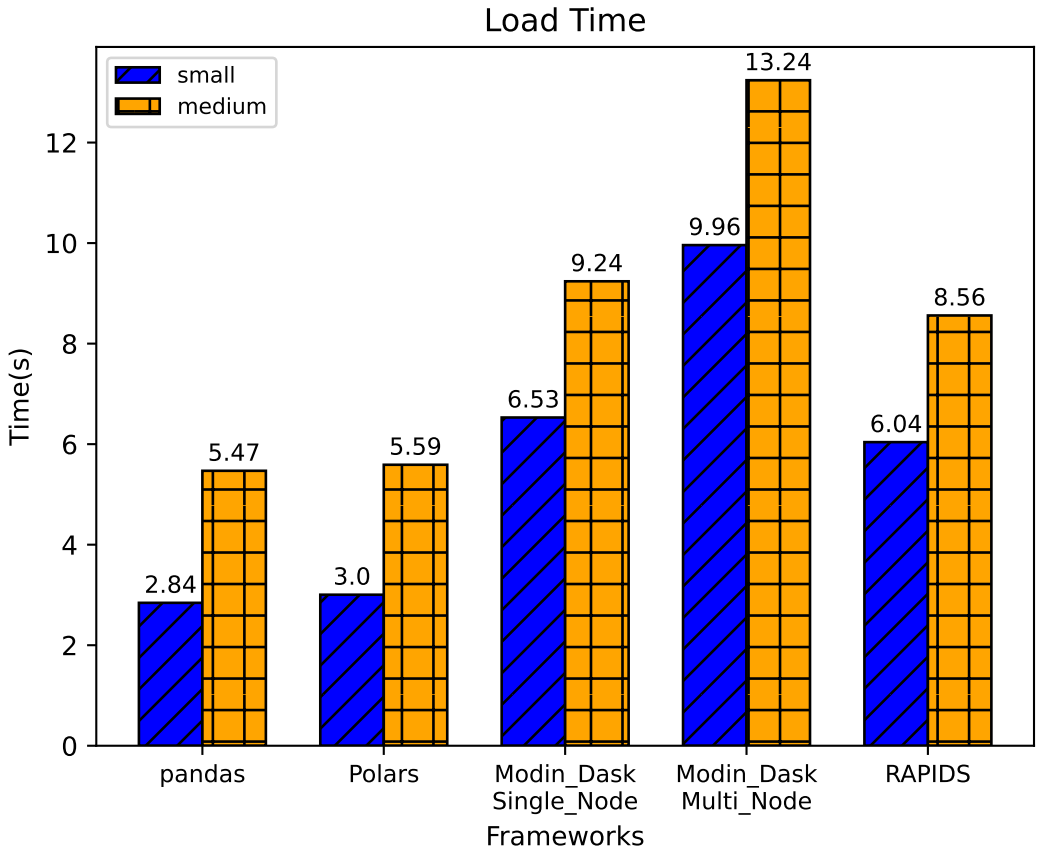


Figure 16: Example of graph generated for loading phase.

3.4.6 Performance and Energy Consumption metrics

As stated on requirement 5, our benchmark will collect statistics from both performance and energy consumption data to report the results desired.

Focusing on performance and execution times measurement, we will evaluate each framework individually by calculating the elapsed between the beginning and the end of the execution of each operation, on micro-benchmarking mode, and the elapsed time between the beginning and the end of the execution of the set of operations defined on the workflow descriptor, on the macro-benchmarking mode. It is important to note that the data loading time is measured separately, as an isolated operation on both micro and macro-benchmarking workload types.

For energy measurement we used two non-intrusive tools, **Powerjoular** and **Perf**, and both tools were executed on *system wide* mode, collecting information from the energy consumption of all systems during the execution of the workflows. Both of these tools rely on **RAPL** to collect the power statistics from the system.

Powerjoular collects statistics from CPU utilization, Total Power, and Total Energy Consumed. We save this information on a CSV file, generated by Powerjoular itself, that can be consulted by the user.

Perf was operated by using *perf stat* and the event *power/energy-cores*, reporting the total energy consumed by the system during the execution of the workflows. This tool was used for multi-node energy measurement, being executed on master and worker nodes to report the total energy used by all systems involved in this type of execution.

We collected, as well, memory usage statistics, using **dstat**. Using this tool's features and versatility, we were able to collect information about memory used, free memory, and cached memory.

Chapter 4

Evaluation, Testing Environments and Results

In this chapter, we will explain all the tests that were done using our benchmarking tool, as well as make a description of the testing environments and resources that were used. We will, also, explain the set of frameworks that were analysed and tested and why we chose those frameworks.

4.1 Frameworks

Although our benchmark has full support for 10 different frameworks, we decided to select a subgroup of those frameworks to perform our study and to analyse deeply. The frameworks chosen were **pandas**, **Modin (Ray and Dask Single and Multi-Node)**, **Polars** and **Rapids (Single Node and Multi-Node with Dask)**. We selected this subgroup of tools once it embraces the main approaches that are associated with distributed and parallel frameworks, such as **Out-of-core** tools, **Multi-node** tools, or **Lazy execution**. With this set of tools, we can obtain results and compare all the different approaches that are associated with distributed tools, while using a smaller sample of implemented frameworks, which facilitates the overall analysis of the results.

4.2 Testing Environments

Our evaluation was done using two different hardware configurations:

- **CPU Server:** 1 server with 2 Intel Xeon Gold 6342 CPU with 24 Cores at 2.80GHz, 192GB RAM, 1 NVMe 1.6TB, and Gigabit network. This machine uses Ubuntu Linux 20.04.5 as operative system, with kernel version 5.40 and Python 3.8.16
- **GPU Cluster:** 4 servers, each one with 1 Intel Core i7-8700 CPU with 6 cores at 3.20GHz, 32GB RAM, 1 NVMe with 240 GB, one NVIDIA GeForce RTX 2080 Ti with 12GB and Gigabit network.

These machines use Rocky Linux 8 as operative system, with kernel version 4.18.0 and Python 3.9.5

All these experiments were executed using the versions pandas 1.5.3, Polars 0.17.15, Modin 0.20.1, Dask 2023.3.2, Ray 2.4.0, and Rapids 23.4.0.

4.3 Tests

Using the two testing environments explained above, we made a set of tests for the 2 modes that our benchmark has implemented.

4.3.1 Workload types

- **Micro-benchmarking tests:** The tests presented on this mode were executed on the **GPU Cluster**. We evaluated a total of 6 basic operations, as well as the loading time of the data from the Parquet files to the dataframes, executing a total of 4 runs for each dataset. The final results exhibit the total energy consumed during the execution of all operations and the data loading, moreover, they all show the execution time for each operation and for the data loading.

  To perform these tests we executed the same workload with 3 different sizes, all using real data from the **New York City Taxi and Limousine Commission** (**NYC TLC**). The small series has a raw total size of 1.5 GB and 14 745 97 entries. The medium series has a raw total size of 3.07 GB and 30 088 556 entries. The big series has a raw total size of 8.7 GB and 84 598 444 entries. The biggest test was only performed by **Modin Dask in Multi Node** and will be better explained in a further section. The energy consumption results are presented as total energy consumed, Joules.

- **Macro-benchmarking tests:** The tests presented on this mode were executed on both hardware configuration **CPU Server** and **GPU Cluster**. To perform this evaluation we used synthetic random data and real data from the **NYC TLC**.

  We executed synthetic tests with a total of 60 operations (4 operations per artifact, with a total of 15 artifacts per framework), with a star-like distribution. Synthetic data testing used base artifacts whose size varied between 1000 entries and 30 million entries, on the CPU Server, and 1000 entries and 10 million entries on the GPU Cluster.

  With real data and workflows, we used both hardware configurations, having the small and medium series on both. On the CPU Server, the small series has a raw total size of 8.7 GB and 84 598 444

entries. The medium series has a raw total size of 11.23 GB and 109 247 536 entries. On the GPU Cluster, we used the same series as Micro-benchmarking, with the raw sizes of 1.5 GB and 3.07 GB for small and medium, respectively. The workflow used on both hardware configurations was composed of the execution of 6 chained operations, executing a total of 4 runs for each dataset. The operations executed were groupby, mean, sum, and sort, and the energy consumption results are presented as energy consumed per operation performed, J/Operation.

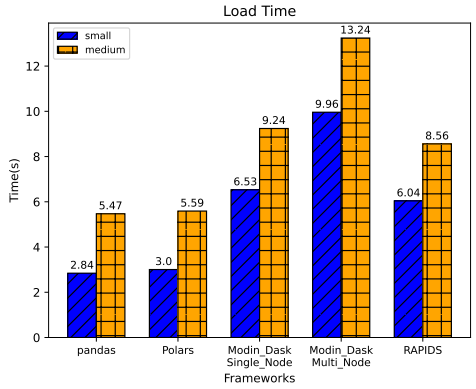
4.3.2 Single and Multi-Node executions

- **Single Node:** Single Node testing was performed both on **CPU Server** and **GPU Cluster**. Polars and pandas are only executable on single-node environments, while Modin and RAPIDS can be both executed on single and multi-node environments. While running on a single node, Modin with Dask backend executes a local client that will launch both master and worker on a unique machine, performing all the computation processes and tasks using the resources of a single node. RAPIDS, using its library cuDF, executes all the operations on a single machine, by default.

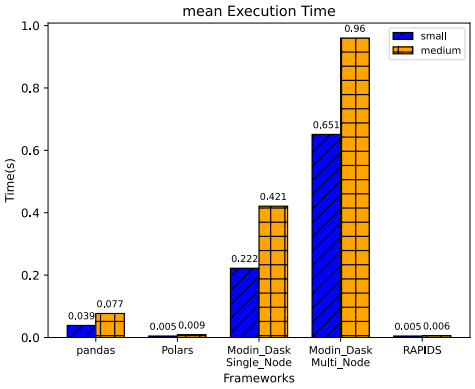
- **Multi Node:** Multi Node testing was performed taking advantage of **GPU Cluster** and **SLURM** workload manager, only being executed on this hardware configuration. From the selected frameworks, Modin with Dask backend and RAPIDS, also using Dask, were the tools that were able to accomplish those tests on a multi-node scenario. SLURM was responsible for all the scheduling and task management and hardly any changes were necessary to be made from the source code of Single Node execution to Multi Node execution. The scaling from 1 node to N node was defined on the framework's code, using Dask's SLURMCluster client.

4.4 Results

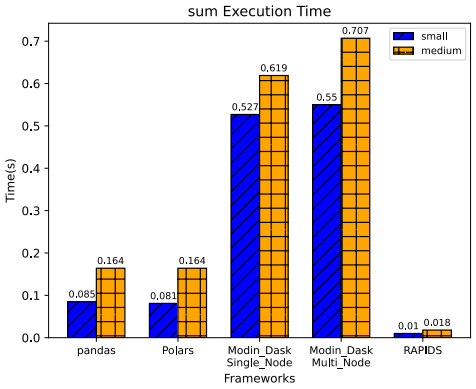
In this section, we will present the results obtained during the execution of the micro and macro benchmarking tests. Following those results, we will elaborate an explanation and analysis for them. Both execution times and energy consumption values are presented and compared between all studied frameworks, as described in the previous chapter.



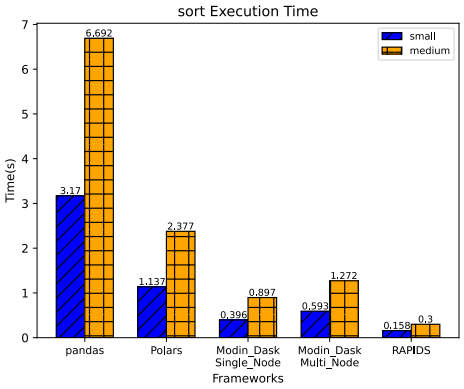
(a) Load



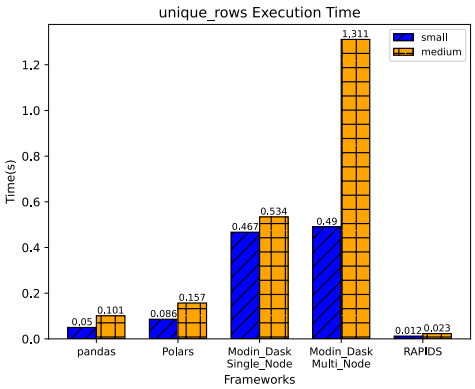
(b) Mean



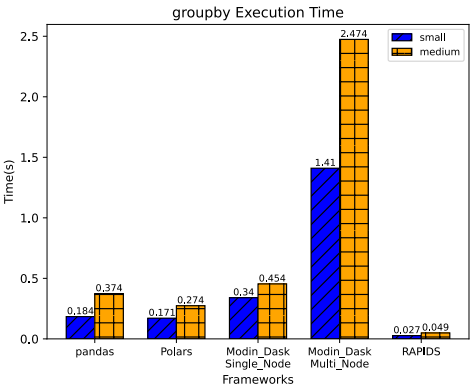
(c) Sum



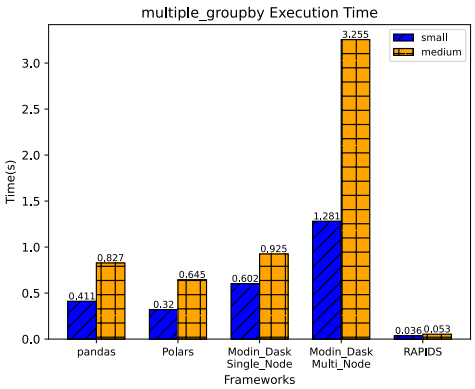
(d) Sort



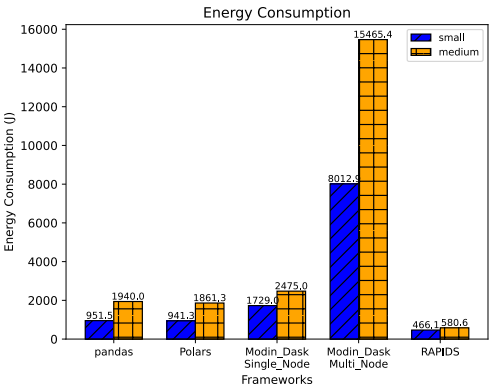
(e) Unique Rows



(f) Group By



(g) Multiple Group By



(h) Energy Consumption

Figure 17: Micro-benchmarking Execution Time and Energy Consumption

4.4.1 Micro-benchmarking

GPU Cluster

- **Execution Time:** Starting with the loading time, Figure 17a, Polars and pandas are highlighted as the frameworks with faster loading times, for both series. RAPIDS, the framework that uses GPU computation power to perform operations, has a higher loading time, which can be explained by the process of data transfer from memory to GPU, causing an additional overhead when compared to Polars or pandas. Modin has the slowest loading time, both on Single and Multi-Node and with both series, explained by the client creation and starting processes that cause additional overheads. Modin, in Multi-Node execution, has the loading time affected by the Dataframe's partitioning and distribution through the worker nodes.

  Analysing the operations executed individually, in Figures 17b - 17g, we point up to RAPIDS, once it is the framework with the fastest execution times for all executed operations. Taking advantage of GPU computation power, applied to parallel computing, this framework has execution times that are up to 15.6 times better than the sequential execution of pandas, which is the case of Multiple Groupby execution with the medium series (0.053 vs 0.827 seconds), Figure 17g.

  On the other hand, we observe that Modin, in Multi-Node execution, has the worst performances in every operation, except for sort, especially when handling the medium series. Once we are evaluating individual operations, we see that the overhead of data distribution and worker synchronization is way too big to deal with this kind of operation, which is executed faster in single node executions by all the remaining frameworks.

  We enhance that the sort operation, Figure 17d, has a quite high execution time on the sequential execution of pandas when compared to all distributed frameworks, which have faster execution times for this operation. pandas, with the small series, execute this operation in a slower time than all the remaining frameworks with the medium series.

- **Energy Consumption:** Observing the Figure 17h, we observe the same pattern and behaviour pictured with the execution times analysis. RAPIDS has the lowest energy consumption values for both series. When relating to the dstat memory usage information collected, it is clear that this framework has the lowest memory usage of all the executed tools. Allied with efficient GPU usage and with fastest execution times, this framework can perform all the desired operations with the lowest energy consumption.

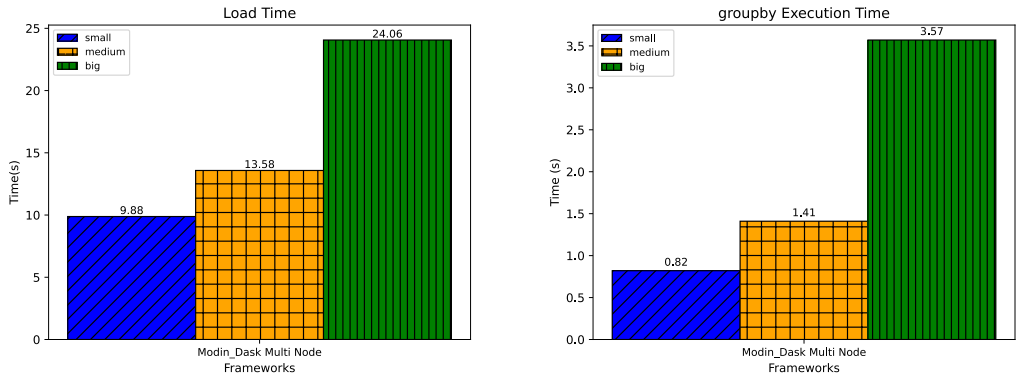
Similar to execution times, pandas and Polars reflect similar energy consumption values, for both series, with Polars reporting values a bit lower. The memory usage for these frameworks is higher for pandas.

Modin, in turn, reports the highest consumption values. On Single Node execution, this framework has the highest memory usage, nearly double when compared with RAPIDS. On the other hand, we observed that Modin, in Multi-Node execution, has a low memory usage both on master node and worker nodes. Once the Dataframe is split through the available nodes, the computation resources used by each individual machine are lower when compared to Single-Node execution, which handles the Dataframe as a whole and needs to use more memory from one singular node. However, Modin in Multi-Node execution has the highest energy consumption values. Even with lower memory usage, this framework uses 4 machines simultaneously, which represents 4 energy-consuming systems during a period of time that is similar, and even slower, than the Single-Node frameworks.

Modin-Dask Multi Node Execution

As shown on the Micro-benchmarking testing results, we were not able to see any major advantages from executing Modin Dask on Multi-Node, both for execution times and for energy consumption. For the analysed series, Multi Node does not reveal any kind of performance gain or optimization when compared to Single Node execution. However, when we studied the execution with a bigger series, with a dataset with a raw size of 8.7 GB, we were able to identify one of the major advantages of this execution mode. Due to the data partitioning that Dask does before the distribution through the cluster nodes, this framework is able to deal with a larger amount of data when compared to Single-Node executions. While the Single Node execution was not able to even load the bigger series, Multi-Node was able to do the data loading and perform all operations with acceptable execution times, as shown in Figure 18a and Figure 18b, for example.

Multi-Node execution with Modin is a crucial advantage, even when compared with other scalable tools, once it can surpass the resource limitations of one singular machine. Once this framework can scale up to N nodes, we can, as well, split a given Dataframe through the N available nodes, making it possible to handle bigger data volumes that were not possible to be analysed on Single Node executions.



(a) Load

(b) Group By

Figure 18: Modin Dask Multi Node Execution Time

RAPIDS Multi Node

During the execution of the tests on a Multi-Node environment, RAPIDS presented poor results with very low performance, both on energy consumption and execution times. RAPIDS uses Dask to execute Multi-Node computation, however, the data distribution through the nodes is not optimized.

Similar to Single Node execution, RAPIDS loads the dataset, without partition, to GPU memory. The data partition through the available nodes is only done after this process, which causes huge overhead during the execution of several operations, resulting in high execution times and high values of energy consumption.

These flaws are reflected by the obtained results during micro-benchmarking execution. Although the load time of the dataset is similar to other frameworks, the execution time of each individual operation is very high, due to the overhead of data partitioning to GPU, after the loading process to memory.

This framework limits the dataset's size, once it can not split the data as Modin Dask does, for example. Modin Dask can handle a huge amount of data when executing in Multi-Node, while RAPIDS is not able to load that same volume of data to memory. Allied with the huge overhead explained previously and the consequent high execution times, generates high energy consumption values, once the cluster is running and executing tasks for a long period of time when compared to the remaining analysed tools.

For the reasons described, we opted to not present the results of RAPIDS in Multi-Node execution, once they were much worse than the remaining, which would hamper the analysis and comparison of results, as shown in Figure 19.

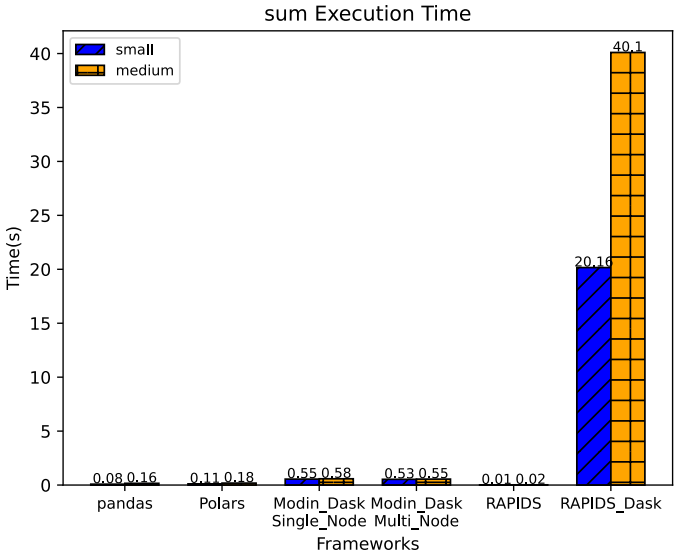


Figure 19: Example of RAPIDS Dask's poor performance.

4.4.2 Macro-benchmarking

GPU Cluster

- **Execution Time:** Analysing the execution time for the workflow defined, Figure 20a, with 6 operations, again we notice that RAPIDS has a clear advantage, for both series when compared to the remaining frameworks. As described on Micro-benchmarking, this framework has an efficient usage of GPU computation power, which translates to the fastest execution time, with a large margin for the second-best framework.

  As we observed on the individual execution of each operation, the sort operation is executed in a slower time by pandas than by any other framework. This way, we can explain how pandas has the second-worst performance for this workflow, for both series.

  Again, Modin in Multi-Node execution presents the poorest performance and the highest execution times, for both series. Similar to the Micro-benchmarking execution, the overhead cost for the data partitioning and node synchronization processes has a huge impact on the performance and execution time of this framework.

- **Energy Consumption:** Figure 21a illustrates the energy consumed by operation executed and we can notice that, despite the notable gap in execution time between pandas and Polars or pandas and Modin Dask in Single-Node, the energy consumed per operation executed by pandas is closer

to Polars and even lower when compared to Modin. Modin has the highest memory usage of all frameworks, which can explain the higher energy consumption values, even when executing the operations faster than pandas. Similarly, Polars reports a higher memory usage on Macro-benchmarking mode than on Micro-benchmarking mode, explaining the near consumption values to pandas.

Modin in Multi-Node execution repeats the highest energy consumption values. Similar to Micro-benchmarking, the memory usage is way lower when compared to Single-Node execution, both on master and worker nodes, but the slower execution times and the utilization of 4 machines represent a more energy-consuming task when compared to the remaining frameworks.

CPU Server

- **Execution Time:** As represented in Figure 20b, we observe that pandas as the slowest execution with both dataset sizes when performing the macro-benchmarking workflow, followed by Modin-Dask with closer execution times. Polars and Modin-Ray detach with the fastest execution times, as they perform the macro-benchmarking workflow faster with the medium series than pandas and Modin-Dask with the small series.

  We observe notable differences in the Modin framework when comparing both backend engines, as Modin-Ray executes the workflow 2.34 times faster with the smaller series and 1.98 times faster with the medium series. From resources statistics collected with dstat, we were able to observe that Modin-Ray has a higher memory usage than the remaining frameworks, allied with a higher cache utilization.

  Observing the framework that reported the fastest execution time, Polars, we noticed that was also the framework with the lowest memory usage of all.

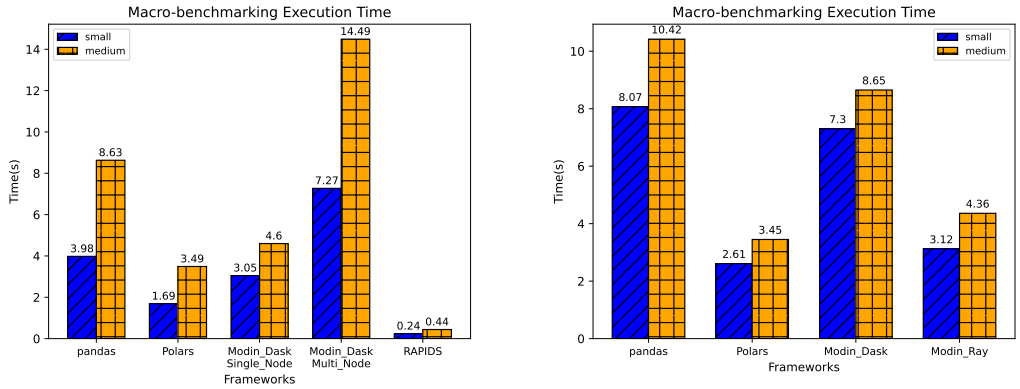
- **Energy Consumption:** Illustrated in Figure 21b, Modin-Ray, Modin-Dask, and Pandas have similar energy consumed per operation executed, in both series, although the execution times have notable differences between them.

  As Modin is a framework that uses all the available resources to optimize the execution while applying parallel computing, the energy consumption associated with this framework is also higher, related to a higher CPU and memory utilization.

  On the other hand, pandas presents the highest energy consumed per operation executed, not due to the intensive resource usage, but rather related to the highest execution times. When compared

with Modin, using the system for a longer time, even with lower CPU and memory utilization, is revealed to be a higher energy-consuming task than using the system for a shorter amount of time and with higher CPU and memory utilization.

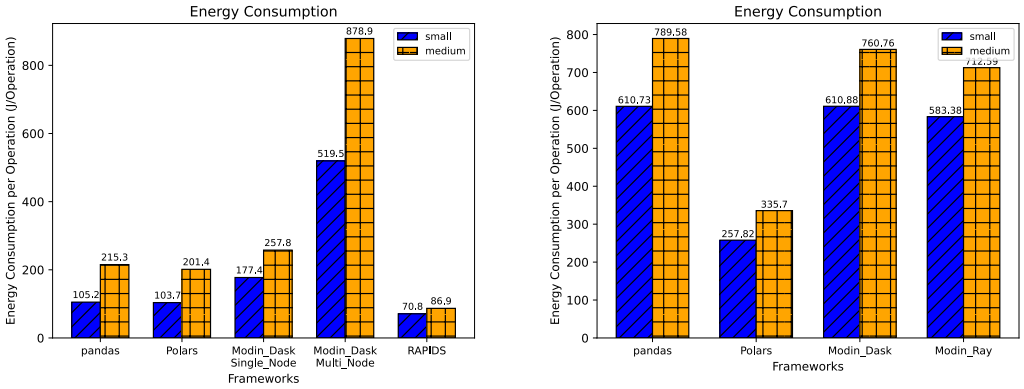
Highlighting as the lowest energy-consuming tool, Polars has the lowest consumption values for the medium series than the remaining frameworks for the small series. As explained by the low memory usage and the fastest execution time, Polars' energy consumption values deserve to be underlined.



(a) GPU Cluster Execution Time

(b) CPU Server Execution Time

Figure 20: Macro-benchmarking Execution Time



(a) GPU Cluster Energy Consumption

(b) CPU Server Energy Consumption

Figure 21: Macro-benchmarking Energy Consumption

Synthetic Data

- **GPU Cluster:** Observing the execution times with synthetic data and random workflows, Figure 22b, we observe some similar tendencies when compared with real data. RAPIDS and Polars remain with a faster execution than pandas, however, RAPIDS reports a slower execution time than Polars, which was not stated when analysing real data workflows. This can be explained, once synthetic data workflows execution times add the loading time to the total elapsed time, while on real data workflows, we did not consider the loading time to be the totality of the execution time for a given workflow. Once RAPIDS has a slower loading time than Polars, it is understandable that, with synthetic data, the total time elapsed during the RAPIDS' execution is higher than Polars'.

  However, that is a different tendency that must be pointed on. While with real data workflows, we observed that Modin, in Multi Node execution, reported execution times that we much slower when compared to Single Node execution, in this case, particularly since the base artifact with 1 million entries, we can detect that Multi Node execution has similar and much closer execution times to Single Node execution. This behaviour can be explained by the workflow that is being executed. While the real data workflow only had 6 operations, this workflow has a total of 60 operations to be executed. Once it is a more demanding procedure, the impact that the overhead of data partitioning and node synchronization has on total execution time is reduced by the complexity of the workflow. This way, and for a certain volume of data, Multi-Node execution can perform with better results when compared to the results reported for real data workflows.

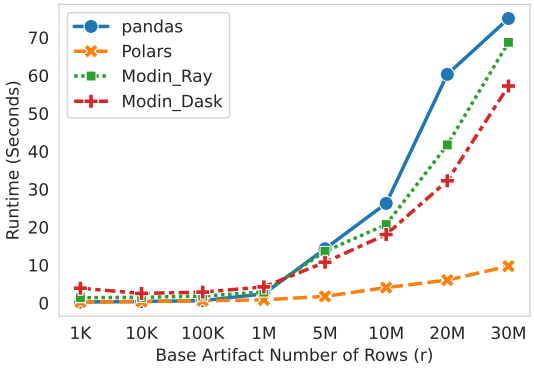
  Observing the energy consumption statistics, Figure 22d, we observe the same patterns described for real data workflows, with Modin in Multi-Node reporting the highest energy consumption and RAPIDS being the framework with the lowest consumption.

- **CPU Server:** Looking for the execution times, shown in Figure 22a, we identify a high increase since the base artifact with 10 million rows. pandas stays as the slowest framework, as Modin_Ray, in this case, performs worse than Modin_Dask. Once we are dealing with more complex workflows (60 operations vs 6 operations with real data), we observe that Modin, with Dask backend, can perform better when working with more complex workflows than Modin with Ray backend, which reported better results with real data workflows. Polars, similar to real data workflows, performed with the fastest execution times.

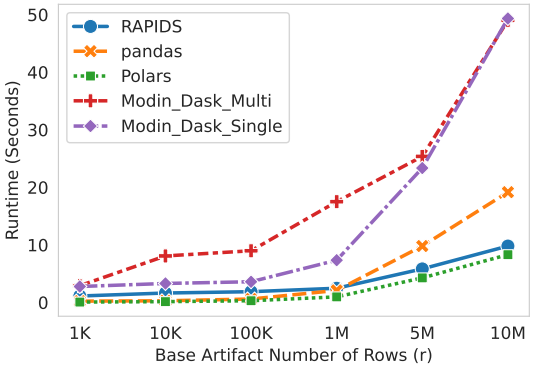
  In terms of energy consumed per operation executed, Figure 22c, we observe that Modin_Ray has the highest consumption values, followed by Modin_Dask. Again, these values can be explained by

the higher memory and CPU utilization of this framework.

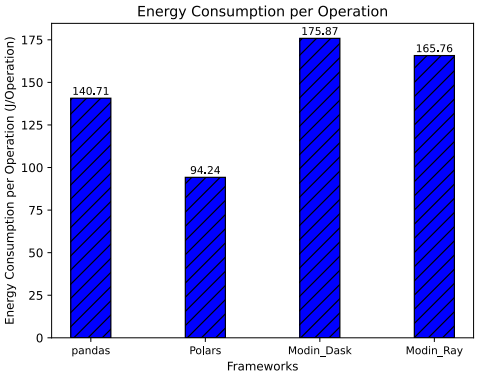
Comparing the results obtained with synthetic data with the results from real data workflows, we observe similar behaviours and tendencies, both for execution times and energy consumption.



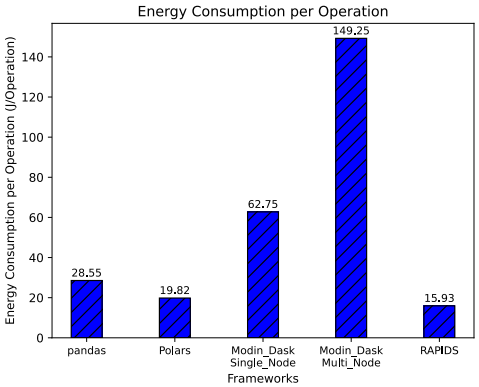
(a) CPU Server execution Time



(b) GPU Cluster execution Time



(c) CPU Server Energy Consumption



(d) GPU Cluster Energy Consumption

Figure 22: Synthetic Data

4.4.3 Hardware Comparison

The two hardware configurations evaluated produced results with some differences between them. As GPU Cluster provided machines that allowed the execution of workflows using GPU computation power, CPU Server made available a more powerful CPU, with more available cores, and more memory.

Looking at the results obtained from the GPU Cluster, it is notable the advantage taken by RAPIDS, the only framework implemented that was able to use GPU to execute the operations. Obtaining, with a notable gap for the remaining frameworks, and the fastest execution times, RAPIDS proves itself with an efficient and powerful utilization of the GPU resources available to perform parallel computing applied to Dataframe's manipulation.

Besides the GPU units that were available on the GPU cluster, each node provided 32 GB of RAM and 6 available cores, while the CPU Server provided 192 GB of RAM and a total of 24 cores. The availability of more resources on the CPU Server, when looking for a single machine execution, benefited particularly Modin's execution. If we analysed the results obtained with synthetic data workflows on the GPU Cluster, Figure 22b, we observed that Modin has a slower execution time when compared to the sequential execution of pandas, while on the CPU Server, Figure 22a, it is notable that Modin performs better, for both analysed backend engines, than pandas. This improvement can be associated with the availability of more cores, as Modin is a framework that uses all available CPU cores to perform the operations.

Polars revealed an efficient performance with both hardware configurations, allied with efficient resource utilization in both cases. This framework obtained the best results for all performed workflows, revealing a great capability to deal with big datasets and several operations, either executed individually or executed on a chained operations workflow. However, the gap between Polars and pandas is bigger when executing workflows on the CPU Server, both for real data workflows and even more notable for synthetic data workflows.

Chapter 5

Conclusions and future work

5.1 Conclusions

Dataframes are one of the most used data structures in data analysis. For massive data processing, several alternatives to pandas appear using Out-of-Core, lazy evaluations, parallel, and distributed techniques.

Based on the requirements that we identified as crucial, from the study of previous existing benchmarks, and the available pandas alternatives, in this thesis we design a new benchmarking tool that is able to evaluate a set of scalable Dataframe frameworks. The tool offers the possibility to be expanded to the utilization of new study cases, with new frameworks, new workloads, and new testing environments.

Using the benchmark developed in this thesis, we did a detailed benchmarking study of the performance and energy consumption of scalable dataframes frameworks, on different hardware platforms. The analysis produced stands as an example of a study that is possible to do with the developed benchmark.

Lastly, we strongly hope that the results, and the whole study, are helpful to the community, especially the data science and big data community, that rely on this type of tools every day. This work, furthermore, aims to ease the choice between distributed tools to execute a given scenario, helping to select the framework that better fits the needs and demands presented.

5.2 Prospect for future work

Once we built our tool with a flexible architecture and with an easy insertion of new frameworks, we strongly encourage any new framework addition, and related study, to our benchmarking. It would be quite interesting to see new frameworks and new hardware configurations being analysed, describing their behaviour in new and diverse distributed environments. With an increasing rise of new scalable and distributed tools, including the latest frameworks, that bring new approaches and optimizations, this work is a procedure that we would follow full of interest and enthusiasm.

Similar to the addition of new frameworks, the addition of new real data workloads, with the study and exploration of other datasets, is an interesting inclusion. These new workloads would bring new workflows to be explored and tested, generating new results and analysis. Workflows that use new datasets can also be merged with the existing workflows, deepen even more the analysis made during this work even more.

Bibliography

[1] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.

[2] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 22–31, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2016.

[4] Adel Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–4, 2022.

[5] SchedMD. Slurm. https://slurm.schedmd.com/overview.html, 2021. Accessed: 2023-06-27.

[6] SchedMD. Slurm energy account plugin. https://slurm.schedmd.com/SUG13/energy_sensors.pdf, 2013. Accessed: 2023-06-27.

[7] Transaction Processing Performance Council. Tpc-energy benchmark development. https://www.tpc.org/tpc_energy/presentations/tpc-energy-2.pdf. Accessed: 2023-01-02.

[8] Shuguang Dou, Xinyang Jiang, Cai Rong Zhao, and Dongsheng Li. Ea-has-bench: Energy-aware hyperparameter and architecture search benchmark. In *The Eleventh International Conference on Learning Representations*, 2022.

[9] Brendan Gregg. *Systems Performance: Entrepise and the Cloud*. Prentice Hall, 2014.

[10] Apache Software Foundation. Apache parquet. https://parquet.apache.org//, 2022. Accessed: 2023-01-02.

[11] Thomas Spicer. Apache parquet: How to be a hero with the open-source columnar data format. https://blog.openbridge.com/how-to-be-a-hero-with-powerful-parquet-google-and-amazon-f2ae0f35ee04/, 2017. Accessed: 2023-01-02.

[12] Apache Software Foundation. Apache arrow documentation. https://arrow.apache.org/docs/format/Columnar.html, 2022. Accessed: 2023-01-02.

[13] Apache Software Foundation. Apache arrow. https://github.com/apache/arrow, 2022. Accessed: 2023-01-02.

[14] Tim Hesterberg. Statistical models in s, 1993.

[15] Devin Petersohn. *Dataframe Systems: Theory, Architecture, and Implementation*. PhD thesis, Ph. D. Dissertation. EECS Department, University of California, Berkeley ..., 2021.

[16] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[17] Imran Ghani. *Handbook of research on emerging advancements and technologies in software engineering*. IGI Global, 2014.

[18] H2O.ai. Database-like ops benchmark. https://h2oai.github.io/db-benchmark//, 2021. Accessed: 2023-01-02.

[19] Xinrong Meng and Hyukjin Kwon. Benchmark: Koalas (pyspark) and dask. https://www.databricks.com/blog/2021/04/07/benchmark-koalas-pyspark-and-dask.html/, 2021. Accessed: 2023-01-02.

[20] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E Gonzalez, Anthony D Joseph, and Aditya G Parameswaran. Flexible rule-based decomposition and metadata independence in modin: a parallel dataframe system. *Proceedings of the VLDB Endowment*, 15(3):739–751, 2021.

[21] Zoumana KEITA. Benchmarking high-performance pandas alternatives. https://www.datacamp.com/tutorial/benchmarking-high-performance-pandas-alternatives, 2023. Accessed: 2023-06-27.

[22] Shriram Shanbhag and Sridhar Chimalakonda. An exploratory study on energy consumption of dataframe processing libraries. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 284–295, 2023.

[23] Mohammed Suhail Rehman and Aaron Elmore. Fuzzydata: A scalable workload generator for testing dataframe workflow systems. In *Proceedings of the 2022 Workshop on 9th International Workshop of Testing Database Systems*, DBTest '22, page 17–24, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *arXiv preprint arXiv:2001.00888*, 2020.

[25] Dask. Dask documentation. https://www.dask.org/, 2022. Accessed: 2023-01-02.

[26] Ray-project. Ray. https://github.com/ray-project/ray, 2022. Accessed: 2023-01-02.

[27] Maarten A Breddels and Jovan Veljanoski. Vaex: Visualization and exploration of out-of-core dataframes. *Astrophysics Source Code Library*, pages ascl–1810, 2018.

[28] Databricks. Koalas. https://github.com/databricks/koalas, 2022. Accessed: 2023-01-02.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[30] Apache Software Foundation. Introduction to apache spark. https://aws.amazon.com/big-data/what-is-spark/, 2021. Accessed: 2023-01-02.

[31] Bodo.ai. Bodo developer documentation. https://docs.bodo.ai/2022.9/, 2022. Accessed: 2023-01-02.

[32] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. SciPy Austin, TX, 2015.

[33] Ibis-project. Ibis. https://github.com/ibis-project/ibis, 2022. Accessed: 2023-01-02.

[34] DuckDB. Duckdb. https://github.com/duckdb/duckdb, 2022. Accessed: 2023-01-02.

[35] Mark Raasveldt and Hannes Mühleisen. Data management for data science-towards embedded analytics. In *CIDR*, 2020.

[36] Pola-rs. Polars documentation. https://pola-rs.github.io/polars-book/user-guide/, 2022. Accessed: 2023-01-02.

[37] h2oai. Datatable. https://github.com/h2oai/datatable, 2022. Accessed: 2023-01-02.

[38] NVIDIA Corporation 2023. Rapids. https://rapids.ai/learn-more/#about, 2023. Accessed: 2023-06-17.

[39] NVIDIA Corporation 2023. Welcome to cudf's documentation! https://docs.rapids.ai/api/cudf/stable/, 2023. Accessed: 2023-06-17.

[40] Abdallah Aguerzame, Benoit Pelletier, and François Waeselynck. Gpu acceleration of pyspark using rapids ai. In *DATA*, pages 437–442, 2019.