



Kristianstad
University
Sweden

Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

DA399D: Independent project (degree project), 15 credits, for the Degree of Bachelor of Science (180 credits) with a major in Computer Science
Spring Semester 2023
Faculty of Natural Sciences

A Comparative Analysis of Pandas DataFrames and Apache Spark DataFrames

Authors

Meron Habtemichael
Md Masudul Islam

Title

A Comparative Analysis of Pandas DataFrames and Apache Spark DataFrames

Supervisor

Charlotte Sennersten

Examiner

Niklas Gador

Abstract

This thesis presents a comprehensive comparison of Pandas and Spark DataFrames, focusing on their architectural differences and performance for data processing tasks. DataFrames, a common data structure in both frameworks, play a vital role in facilitating efficient data manipulation and analysis. However, selecting the appropriate framework for a specific use case requires a thorough understanding of their respective capabilities and limitations. In our investigation, we delve into the fundamental concepts, features, and components of both Pandas and Spark DataFrames. We evaluate their execution time using the COVID-19 Research dataset as a case study. Our findings reveal that Pandas DataFrames are well-suited for better performance on a single-node computer. Nevertheless, they are limited by the available memory on a single machine, making them less suitable for large-scale data processing. Conversely, Spark DataFrames are designed for large-scale data processing, providing an efficient and scalable solution for handling massive datasets. While both Pandas and Spark are powerful data processing libraries, their performance characteristics can vary significantly depending on the nature of the task, the size of the dataset, and the hardware architecture. The observed performance difference in your experiments can be attributed to the interplay of these factors, and a deep understanding of these factors can provide valuable insights for optimizing data processing tasks. These findings contribute to the growing body of knowledge on data processing frameworks, helping data scientists and developers make informed decisions to optimize their data analysis workflows.

Keywords

Pandas DataFrame, Spark DataFrame, Data processing, Performance analysis, Architectural differences.

Table of Contents

Acknowledgments	1
1 Introduction.....	2
1.1 Problem Definition.....	3
1.2 Motivation	3
1.3 Research Questions	3
1.3.1 Research Question 1	3
1.3.2 Research Question 2	3
1.4 Research Objectives	3
1.5 Variables	4
1.5.1 Independent Variables:	4
1.5.2 Dependent Variables:	4
2 Methodology	4
2.1 Literature Study	5
2.1.1 DataFrames	6
2.1.2 Importance of Data Processing Frameworks	6
2.1.3 Pandas DataFrames Architecture	7
2.1.4 Spark DataFrame Architecture.....	14
2.1.5 Architectural Comparison: Spark vs. Pandas DataFrame.....	23
3 Results from Experiment	26
3.1 Experiment Setup	26
3.1.1 Data Preparation and Segmentation	27
3.2 Performance Evaluation	29
3.2.1 Data Loading	30
3.2.2 Filtering and Selection.....	31
3.2.3 Data Transformation	32
4 Results Analysis.....	33
4.1 Data Loading.....	33
4.2 Filtering and Selection	34
4.3 Transformation.....	36
5 Discussion.....	36
6 Conclusion	37
6.1 Future work	37
7 References	39

Acknowledgments

We would like to express our deepest appreciation and heartfelt gratitude to our supervisor, Charlotte Sennersten, Senior Lecturer in Computer Science at the Faculty of Natural Science, for her unwavering support, continuous guidance, and consistent availability, even amidst her demanding schedules. We also want to extend our gratitude to our examiner, Niklas Gador, for his constructive feedback.

Our special thanks go to Kamilla Klonowska for her efforts in organizing the course, providing additional guidance when needed, and fostering an atmosphere of collaboration and learning among students. We are also grateful to our fellow classmates, colleagues, and friends who provided inspiration, motivation, and support throughout this journey.

Finally, we would like to acknowledge the contributions of the researchers and developers whose work laid the foundation for our thesis, as well as the open-source communities that continue to create and maintain the tools and frameworks we used in our research. Their collective support, encouragement, and expertise have been valuable in the successful completion of this thesis.

1 Introduction

In the age of big data, where zettabytes of data are created and processed annually, proficient data processing frameworks are critical for not only managing voluminous datasets but also distilling valuable insights that drive decision-making [5]. In the realm of data science, two such prominent frameworks, namely Pandas and Spark, have become the keystone for data management and analysis, given their versatile DataFrame structures [2]. This research undertakes a comprehensive examination of these two frameworks, particularly contrasting the architectural difference between Pandas and Spark DataFrames and assessing their consequential effects on performance metrics. The dataset used for our analysis, titled "COVID-19 Research", is sourced from Kaggle [3]. This extensive corpus consists of over one million scholarly articles related to COVID-19, SARS-CoV-2, and correlating coronaviruses. The compilation, an effort by leading research institutions in collaboration with the White House, has been scrupulously curated to expedite scientific discoveries in this critical period of global health crisis [3]. The total volume of this dataset - 87.52 GB in total - is demonstrative of the scale of big data involved in modern research. For the specific purposes of this study, however, we have chosen to focus on a single CSV file containing volume of 15.58 GB. The techniques and approaches for exploring this dataset will be detailed further in the ensuing methodology section. It provides an in-depth analysis of the execution time for different operations such as data loading, filtering, selection, and transformation. The results are based on experiments conducted on a MacBook Air M1 2020, equipped with the M1 chip, 8 GB RAM, and 8 CPU cores. Whether you are a data scientist looking to optimize your data processing workflows, a software engineer working on big data applications, or a researcher exploring the latest tools and techniques in data analytics, this document offers valuable insights and practical knowledge. The aim is to provide developers with a comprehensive understanding of the strengths and weaknesses of Pandas and Spark in different scenarios. This knowledge can guide the choice of tool for specific tasks, leading to more efficient and effective data processing. The document also delves into the technical details of how these libraries operate under the hood, discussing concepts like vectorization, task scheduling, data partitioning, and more. Python's combination of readability, simplicity, and vast library support makes it an ideal language for conducting this comparative study.

We, the authors acknowledge the inherent design differences between Spark and Pandas. Spark is architected for distributed computing, meaning it is designed to process data across multiple machines or nodes. Its core functionality is built around the concept of parallel processing, where a large task is divided into multiple smaller tasks that are executed simultaneously across different nodes in a cluster [2]. On the other hand, Pandas is designed for single-node processing. It operates entirely within the memory of a single machine and does not have built-in capabilities for distributed computing [6]. In this study, the comparison is conducted on a single machine with multiple cores. This approach is chosen because comparing Pandas and Spark in a

distributed environment would not be appropriate or meaningful. Pandas is not designed to operate in a distributed environment, and therefore, it wouldn't be a fair or relevant comparison. However, even on a single machine, the architectural differences between Spark and Pandas can lead to different performance outcomes. In essence, this study aims to observe how the unique architectural features of both frameworks translate into differing performance results on a single machine, providing insights into their relative strengths and weaknesses in this specific context.

1.1 Problem Definition

The primary goal of this study is to explore the architectural differences between Pandas DataFrames and Spark DataFrames and assess their impact on performance [1] [2]. Additionally, the research will compare the performance of both frameworks when analyzing the COVID-19 Research dataset to establish their suitability for real-world data processing tasks.

1.2 Motivation

As data generation and processing continue to expand, choosing the appropriate data processing framework becomes increasingly important for efficiency and scalability [5]. By comparing Pandas DataFrames and Spark DataFrames, this study will seek to provide insights into their respective advantages and disadvantages, helping developers and data scientists make informed decisions when selecting a framework for their data analysis projects.

1.3 Research Questions

1.3.1 Research Question 1

How do the architectural differences between Pandas DataFrames and Spark DataFrames impact their performance?

1.3.2 Research Question 2

How does the performance of Pandas DataFrames and Spark DataFrames compare in analyzing the Covid-19 Research dataset?

1.4 Research Objectives

The research objectives aim to address two main queries: exploring architectural differences between Pandas and Spark DataFrames and evaluating their performance using the COVID-19 Research dataset. The objectives are:

- Investigate architectural differences and assess their impact on performance metrics, including execution time for various data processing scenarios.
- Analyze the COVID-19 Research dataset with both frameworks, measuring the execution time of its common built-in method for data loading, filtering and selecting, and data transformation.

1.5 Variables

1.5.1 Independent Variables:

The architectural differences between Pandas DataFrames and Spark DataFrames serve as the primary independent variable, influencing the overall performance and suitability of each framework for various data processing tasks.

1.5.2 Dependent Variables:

The performance of data processing frameworks, as the dependent variable, is influenced by the architectural differences between them. Execution time is employed to measure this performance, providing a comprehensive understanding of the frameworks' capabilities.

- **Execution time:** These measures how long it takes for a given operation or query to complete using either Pandas DataFrames or Spark DataFrames.

2 Methodology

To answer our research questions, we will employ a combination of experimental and analytical methodologies. The first step in our methodology involves collecting representative sample data from Kaggle, specifically the COVID-19 Research dataset [3]. This dataset has been chosen due to its complexity and real-world relevance, making it an ideal candidate for evaluating the performance of Pandas DataFrames and Apache Spark DataFrames under varying conditions [2] [1]. For our research project, we have chosen Python as our programming language of choice. As part of the setup process, we have utilized the pip package manager to install pandas and pyspark packages, which will allow us to utilize the Pandas and Spark libraries, respectively, in our code.

Our research experiment will involve loading and manipulating the dataset using both Pandas and Spark DataFrames and then measuring the performance of several built-in methods, such as data loading, filtering, and selection, and data transformation, others [1] [25]. We will systematically increase the dataset size to assess the performance of each DataFrame implementation with the larger data size. Furthermore, we will delve into the architectural aspects of both Pandas and Spark DataFrames to understand how resources are utilized. By examining the architecture, we aim to identify the key factors that contribute to the observed

performance differences between the two DataFrame implementations. This comprehensive methodology will allow us to not only measure the performance differences but also provide insights into the underlying causes, facilitating a thorough understanding of the advantages and limitations of each data frame in data processing tasks [6] [22].

2.1 Literature Study

This study aims to provide a comprehensive understanding of Pandas and Spark DataFrames by comparing their architecture, functionality, and performance. To gather relevant literature for this research, we conducted a systematic and narrative literature review. The literature search is performed using five primary keywords: "Pandas DataFrame," "Spark DataFrame," "Data processing," "Performance analysis," and "Architectural differences."

The search strategy for identifying relevant studies included using different combinations of these keywords and searching in Google Scholar. The following (Table 1) outlines the search strategy and the number of hits obtained for each search string:

Database	Keywords and Strings	Hits
Google Scholar	"Pandas DataFrame" AND "Architecture" AND "Performance" AND "execution time"	288
Google Scholar	"Spark DataFrame" AND "Architecture" AND "Performance" AND "execution time"	151
Google Scholar	"Pandas DataFrame" AND "Spark DataFrame"	89
Google Scholar	"Pandas DataFrame" AND "Spark DataFrame" AND "Data processing"	77
Google Scholar	"Pandas DataFrame" AND "Spark DataFrame" AND "Performance analysis"	7

Table 1: Literature Search Strategy.

After carefully reviewing the search results and selecting the most relevant articles, we proceed to present the findings and analysis of the study. The following sections are structured to provide a smooth transition from an overview of DataFrames and their importance to a detailed comparison of Pandas and Spark DataFrames.

2.1.1 DataFrames

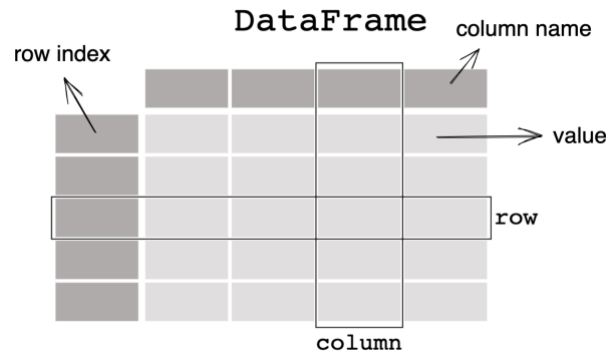


Figure 1: Overview of DataFrame.

Source: Overview of DataFrame [Online]

As shown in Figure 1 above, the standard DataFrame format offers a concise representation of data in a structured way. They are a widely adopted data structure for representing and manipulating structured datasets, offering a general overview and intuitive representation of tabular data similar to a spreadsheet or a relational database table [7]. They were developed as a convenient and expressive alternative to SQL tables and other data models, providing an integrated approach to data manipulation and analysis [2]. One of the key advantages of DataFrames is their flexibility and expressiveness, which allow users to perform complex data operations with ease, such as filtering, aggregation, and transformation, without requiring extensive knowledge of SQL or other query languages [8].

In addition to their convenience, DataFrames also offer significant advantages in terms of data visualization. They can be easily integrated with various plotting libraries, such as Matplotlib, Seaborn, and Plotly, enabling users to create a wide range of visualizations directly from the data [16]. While the integration process may vary across different types of DataFrames, the concept of integrating and utilizing additional libraries remains a prevalent practice. The extensible and interoperable design of DataFrames [1] and Pandas Development Team [23] allows for compatibility with other data processing tools and libraries, equipping users to tap into the extensive ecosystem of data analysis tools available in languages like Python and R.

2.1.2 Importance of Data Processing Frameworks

Data processing frameworks, such as Pandas and Apache Spark, have gained significant popularity in the data science community due to their ability to simplify and optimize the processing of large datasets [2]. These frameworks leverage the DataFrame data structure, which provides an intuitive and efficient way to represent and manipulate tabular data. The adoption of DataFrames in data processing frameworks has facilitated a more seamless and expressive approach to data manipulation, enabling users to perform complex operations with minimal effort and thus enhancing productivity and reducing the learning curve associated with data analysis

tasks [5]. One of the primary reasons for the popularity of DataFrames in data processing frameworks is their flexibility and expressiveness, which enable users to perform a wide range of data manipulation tasks, such as filtering, aggregation, and transformation, with ease [2]. Furthermore, the integration of DataFrames with various visualization libraries allows for quick and efficient data exploration and analysis, enabling researchers to derive insights and discover patterns more rapidly [21]. This combination of simplicity, expressiveness, and powerful data manipulation capabilities has made DataFrames an indispensable tool for data processing in various domains, including finance, healthcare, marketing, and scientific research [22], which allows users to perform complex data operations with ease, such as filtering, aggregation, and transformation, without requiring extensive knowledge of SQL or other query languages [8].

Building on our general discussion of DataFrames, we now proceed to present Pandas and Spark DataFrames in separate subchapters to provide a clear and distinct understanding of each framework, which will help address our research questions. In this section, we will focus on the architecture of Pandas DataFrames and its core components.

2.1.3 Pandas DataFrames Architecture

2.1.3.1 Overview of Pandas Framework

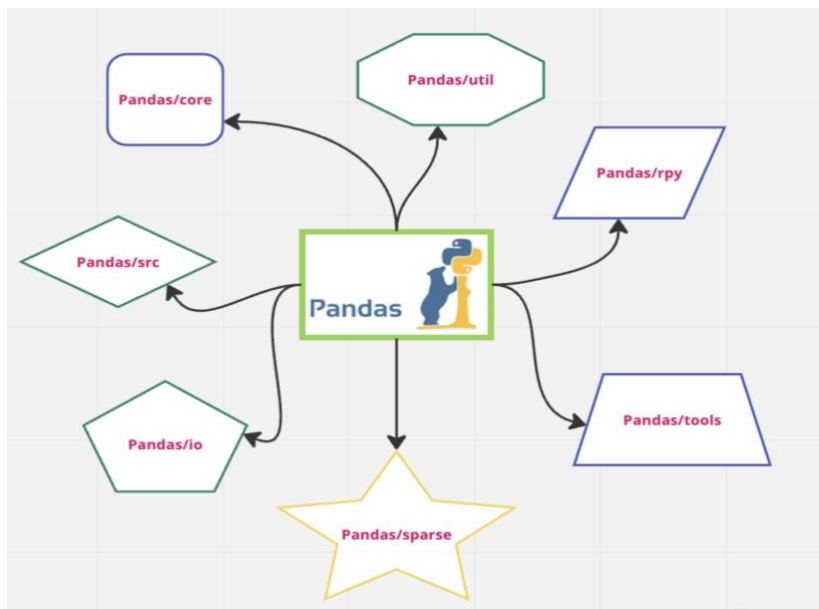


Figure 2: *Pandas Library Architecture, File*

Source: Pandas Library Architecture, File [Online]

Pandas, an open-source data analysis and manipulation library for Python, offers data structures and functions necessary for manipulating structured data. Its foundation relies on two fundamental Python libraries - Matplotlib, which facilitates data visualization, and NumPy, which enables mathematical operations. McKinney [2] developed this library in 2008, aiming to provide an effective, user-friendly tool for data

manipulation library and it has been popular in the field of data science ever since. It is a vast library consisting of 7 crucial and well-documented file hierarchies as shown in Figure 2. This research focus is on the sparse file, which consists of sparse versions of various data structures like DataFrames and Series. Pandas provides a flexible API for data manipulation, particularly with labeled data [6]. Unlike NumPy, which is more focused on numerical operations with integer-indexed data, pandas are designed to work with data that have more complex relationships, such as metadata that can be used to group, select, combine, filter, and transform the data.

Pandas is designed to be fast and performant, which is crucial when dealing with large datasets. However, Python's dynamic nature can cause overhead in certain operations. For example, when looping over a large array, Python's need to look up the definition of a function or a value each time it's used can slow down the process. This overhead becomes significant when dealing with arrays that have hundreds of thousands or millions of elements [6].

To mitigate this, one can use the Python C API, which allows for more control over memory management and can optimize certain operations. However, using the Python C API is non-trivial and requires a good understanding of both Python and C. Another issue is the memory overhead of individual Python objects. For example, a Python list is an object that contains pointers to other Python objects, each of which has its own memory overhead. This can lead to inefficient memory usage and slower operations. To address this, pandas leverages NumPy arrays, which are more memory efficient. A NumPy array is essentially a pointer to a block of memory where the data is stored, with each element in the array being the same size. This allows for better cache performance and faster operations [23].

However, writing functions for NumPy arrays requires knowledge of C, which can be a barrier for some users. This is where Cython comes in. Cython is a Python-like language that allows for more efficient computations by enabling type declarations [38]. It is a programming language, a superset of the Python programming

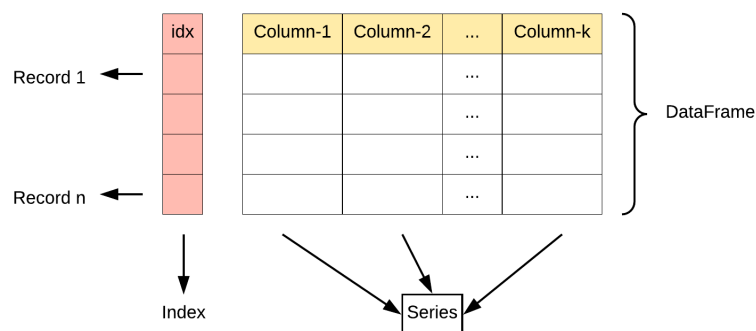


Figure 3: Pandas DataFrame Components.

Source: w3resource, (2023), *Pandas DataFrame Components*. [Online]

language, designed to give C-like performance with code that is written mostly in Python with optional additional C-inspired syntax, compiles to C, allowing you to write code that's closer to the machine and hence faster, while still being able to write Python-like code [38]. So, this implies that whenever Pandas is used to run a specific code and use vectorization operations on a Series object (which will be discussed in the next section), then this means a very optimized and high-speed performance is achieved by compiling the code into C [38]. This makes Pandas a very fast tool for data analysis. As shown in Figure 3, the Pandas DataFrame consists of several key components, including the index, columns, and data [23]. Let's explore each of these components individually in the following section.

2.1.3.2 Pandas DataFrame Components

- **Series Object**

The Series object (see Figure 3) is the most basic data structure in Pandas and serves as the building block for more complex structures such as DataFrames [7] [24]. A Series is a labeled array in one dimension that has the capability to store various types of data, such as integers, floats, strings, and other data types. It has two primary components: the data and the associated index. The data is a collection of values, while the index is a collection of labels that uniquely identify each value in the data. The index can be either explicit, where the user defines the index labels, or implicit, where Pandas automatically generates integer-based index labels. The Series object is designed to support various operations, including arithmetic operations, aggregation functions, and data alignment, all of which are carried out using vectorized operations for increased performance [24].

- **DataFrame Object**

The DataFrame (Figure 3) is the central data structure in Pandas, providing a two-dimensional labeled data structure with columns that can have different data types [8] [25]. Essentially, a data frame is a collection of a series of objects that share a common index, forming a tabular structure like a SQL table or an Excel spreadsheet. DataFrames enable users to perform a wide range of data manipulation tasks such as selecting, filtering, aggregating, transforming, and reshaping data. Pandas provides a vast array of built-in methods to perform these tasks, making it a powerful tool for data analysis in Python [25].

- **Index Object**

The Index object (see Figure 3), an integral component of the Pandas DataFrame architecture, serves as an immutable mapping mechanism that assigns labels to the rows and columns of a DataFrame [9]. This mapping system is pivotal in executing operations such as alignment, merging, and joining, as it associates labels with their corresponding integer indexes. The Index object's immutability ensures data integrity and consistency,

allowing for the sharing of indexes between components and enabling the execution of potentially complex operations to construct an index. Furthermore, it plays a crucial role in data alignment, facilitating operations across multiple data frames with varying indexes by automatically aligning data based on the index labels. The Index object also enhances the performance and usability of DataFrames by expediting data lookups, selections, and join operations [26].

- **Interplay of Components**

The Pandas DataFrame architecture (see Figure 3), comprising Series, DataFrame, and Index objects, facilitates efficient and intuitive data manipulation and analysis in a tabular format. The interplay of these components allows users to perform complex operations on data with relative ease while also benefiting from the library's optimized performance [26]. The flexibility and power of the Pandas working with Python, as it simplifies the process of cleaning, transforming, and analyzing data in a variety of formats and structures.

Now that we have reviewed each component and its relationship with the others, it's time to examine how data is stored and processed within the Pandas library.

2.1.3.3 In-memory Columnar Data Storage

Pandas DataFrame utilizes an in-memory, columnar data storage format, allowing for efficient and fast data manipulation and analysis [6]. Unlike row-based storage, where data is stored sequentially by rows, columnar storage stores data in contiguous memory blocks based on columns. This arrangement results in several performance advantages, particularly when it comes to data analysis tasks that involve a subset of columns or require column-based calculations. The columnar data storage format in Pandas has several benefits, including improved performance, better data compression, and enhanced query execution [6]. Since data within a column is typically homogeneous, it can be stored more compactly, reducing memory usage [10]. Furthermore, operations on columns can take advantage of Single Instruction Multiple Data (SIMD) vectorizations, which allows for faster computations [11]. Another advantage of columnar storage is that it enables efficient filtering and aggregation operations, as it reduces the need to read and process irrelevant data from other columns. In summary, columnar storage significantly enhances the performance of data analysis tasks in Pandas DataFrames.

2.1.3.4 Data Manipulation in Pandas: Vectorized Operations and Efficient Algorithms

Pandas DataFrames provide a comprehensive suite of data manipulation operations, such as filtering, aggregation, transformation, and sorting, which are essential for data analysis and pre-processing tasks [6] [29]. The library employs vectorized operations and efficient algorithms to optimize performance and ensure that these tasks are executed quickly and effectively. Let's look at them one by one.

- **Vectorized Operations**

Vectorized operations in Pandas, which are powered by the underlying NumPy library, are characterized by element-wise computations on entire arrays or columns simultaneously. This approach significantly reduces computational overhead and leads to substantial performance improvements [14]. The concept of vectorization is rooted in the architecture of NumPy, which is designed to work with arrays. This design is a manifestation of the principle of data locality, which is a critical aspect of efficient computation. Data locality refers to the practice of storing related data in close proximity to each other to minimize data access time. In the case of NumPy and, by extension Pandas, data in arrays are stored in contiguous blocks of memory. NumPy offers a type of N-dimensional array, known as the ndarray, that represents a group of elements of identical type. These elements can be accessed using N integers, for instance. All ndarrays are uniform, signifying that each element occupies an equal memory block size, and all these blocks are processed in an identical manner [13].

The efficiency of vectorized operations is further enhanced by Single Instruction, Multiple Data (SIMD) processing capabilities. SIMD is a computing model within Flynn's Taxonomy that executes a single instruction on multiple data elements simultaneously, enabling efficient data-level parallelism. It is a superior computing model to SISD (Single Instruction, Single Data) for certain tasks due to its ability to process multiple data elements simultaneously, enabling efficient data-level parallelism and performance improvements [36]. Figure 4 shows the overview and highlights the difference between SIMD and SISD operations. In SIMD, instructions are implemented in modern CPUs using vector registers, which can store multiple data elements in a single register [11]. The width of the vector register determines the number of data elements that can be processed in parallel. These modern CPUs have SIMD instruction sets, such as Intel's SSE and AVX or ARM's NEON [11]. These instruction sets include operations that can perform a single operation on multiple data points at once, which can significantly speed up certain types of computations. This is crucial, especially in applications like image or signal processing, where many computations are trivially parallelizable.

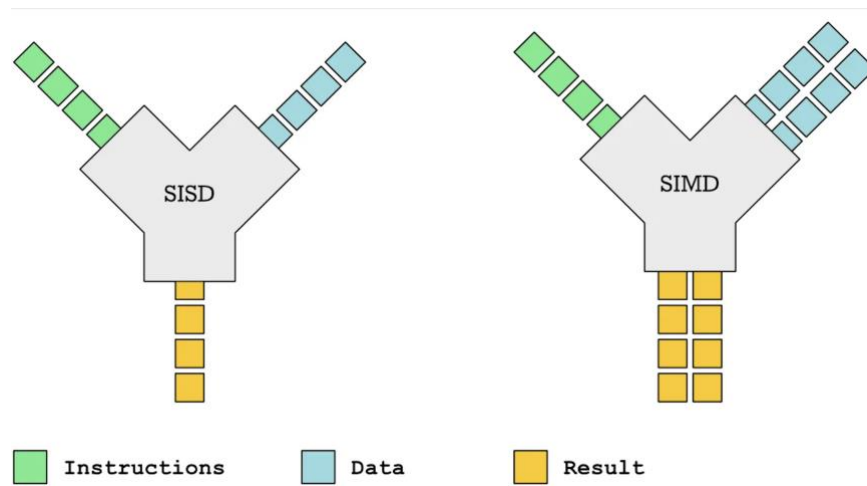


Figure 4: SIMD vs SISD

Source: SIMD vs SISD [Online]

• Understanding SIMD: A Deep Dive

To understand how SIMD works, it is fundamental to understand how the CPU makes use of registers. A CPU register is a tiny storage area that is part of a CPU's architecture. It holds data that the CPU needs to access quickly for efficient processing. There are many types of registers within the CPU alone, but for this simplified example, we refer to the registers within the Arithmetic Logic Unit (ALU). The size of these registers determines how much data the CPU can process at once. In the context of SIMD (Single Instruction, Multiple Data), the size of the SIMD registers is particularly important because it determines the degree of parallelism that can be achieved [37].

For example, a CPU with 128-bit SIMD registers within ALU. This means that each register can hold 128 bits of data at once. Now, if we're dealing with 32-bit floating-point numbers (a common format for representing real numbers in computing), each number takes up 32 bits of space. Therefore, a 128-bit register can hold four 32-bit floating-point numbers at once (since 128 divided by 32 equals 4). When a SIMD instruction is executed, it operates on all four of these floating-point numbers simultaneously. This is where the speedup comes from; instead of executing four separate instructions to process four numbers, the CPU can execute a single SIMD instruction to process them all at once [37]. Now, regarding the experiment in this research, a MacBook Air M1 2020, the M1 chip is based on ARM architecture, which includes a feature called NEON. NEON is ARM's version of SIMD, and it supports 128-bit SIMD operations, just like in the above example. This means that each of the M1's CPU cores can process up to four 32-bit floating-point numbers at once using SIMD instructions. This means that approximately $8 * (4 * 32\text{-bit})$ floating-point numbers are being processed

at once. The actual performance of a system, even when using SIMD instructions, depends on a multitude of factors. These include the specific code being executed, the overall system load, memory availability, and the efficiency of the operating system's scheduling and memory management algorithms, among others.

- **Impact on Data Analysis Task and Limitations**

These vectorized operations accelerate code performance by using low-level machine code to iterate over data and execute the required operations, a process that proves extremely advantageous when performing identical operations on large, homogeneous data sets [6]. However, the benefits of vectorization come with several limitations. For instance, only supported operations can maintain the high speed; reverting to Python loops and functionality would compromise this efficiency, making the code slow again. Consequently, this compels libraries like NumPy to implement their own extensive functionality to maintain speed. Additionally, vectorization only expedites bulk operations on similar data types. If calculations deviate from this pattern, the efficiency gain from vectorization diminishes. Moreover, tools like Pandas can encounter memory constraints due to their in-memory data storage model, particularly with large datasets, suggesting a potential need for alternative data processing frameworks like Apache Spark [12]. Thus, while vectorization significantly boosts data analysis efficiency, its utility is contingent on specific scenarios and may necessitate alternative solutions in other contexts.

2.1.3.5 Pandas in Action (Execution Operation)

In this section of the research, Pandas in Action, we explore the functionality of pandas on a single machine. It's worth noting that Python and its libraries, including Pandas, don't utilize multiple cores by default due to the Global Interpreter Lock (GIL) inherent in the language's design [42]. The GIL ensures that only one thread executes Python bytecodes at a time. This essentially serializes thread-level access to Python objects, making multi-core parallelism challenging to achieve in pure Python. However, there are ways to circumvent the GIL and take advantage of multicore processors. One method is to use the multiprocessing module in Python, which creates separate Python processes, each with its own interpreter and memory space, therefore bypassing the GIL. Yet, Pandas does not use this method directly. Instead, it relies on lower-level libraries that do [21].

Pandas and NumPy, which Pandas is built upon, use lower-level C and Fortran code, which are not subject to the GIL, allowing them to use multiple cores [42]. In particular, NumPy and pandas use the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) libraries for complex computations, and these libraries are typically optimized for multithreading and can take advantage of multiple cores [21] [42].

Now, let's talk about vectorization and SIMD. Vectorization is a powerful method to improve performance in data processing. It involves performing the same operation on multiple data points simultaneously, which is where SIMD (Single Instruction, Multiple Data) comes in [11].

Modern processors, as discussed in the previous section, have SIMD instructions that allow them to drastically improve their performance [11]. For example, when you write a Pandas operation like `df['column1'] + df['column2']`, it is interpreted by Python and then pandas. Pandas then call appropriate routines in NumPy, which in turn relies on underlying C or Fortran code (like BLAS). This code is compiled with knowledge of the SIMD instructions of the target machine [43]. The compilation of this underlying code is typically done using a C or Fortran compiler like GCC or GFORTRAN, which includes flags to vectorize loops where possible [43]. During the compilation, the compiler takes the source code and converts it into machine code that the CPU can execute directly. The compiler also includes optimizations to make the code as efficient as possible, such as using SIMD instructions when performing operations on arrays [43].

So, although pandas and Python might not natively distribute across cores due to the GIL [42], the reliance on lower-level, compiled libraries allow pandas to take advantage of multiple cores and SIMD instructions. However, it's important to understand that not all operations will be parallelized. The degree of parallelization depends on the specific operation, the implementation in the underlying library, and the number of available cores. In summary, Pandas does take advantage of multi-core CPUs due to their usage of lower-level libraries that can bypass the Python GIL, coupled with the power of vectorization and modern CPU SIMD instruction sets that drastically enhance the computational performance [11].

2.1.4 Spark DataFrame Architecture

2.1.4.1 Introduction to Spark Data Processing Framework

- **Overview of Spark**

Apache Spark is an open-source, distributed data processing framework designed for large-scale data processing and analytics tasks [1] [31]. It was developed to overcome the limitations of Hadoop MapReduce, offering enhanced performance, flexibility, and ease of use. Spark supports a variety of programming languages, including Python, Scala, Java, and R, allowing developers and data scientists to work with their preferred language [5]. We have opted to utilize Python as our programming language of choice, and as such, we have decided to use the Pyspark library for our data processing needs. Spark provides a unified platform for various data processing tasks, such as batch processing, interactive queries, machine learning, and graph processing, making it a popular choice for big data applications.

- **Spark's Core Abstraction - Resilient Distributed Datasets (RDDs)**

The Resilient Distributed Dataset (RDD) is a crucial concept in Spark. It serves as an unchangeable distributed aggregation of objects that permits parallel processing across a machine cluster. RDDs are designed to be fault-tolerant, allowing Spark to automatically recover from node failures and ensure the reliability of the data processing tasks. In addition to RDDs, Spark also offers higher-level abstractions like DataFrames and Datasets, which are built on top of RDDs and provide more advanced functionality and optimizations for structured data processing [1].

At the core of the Spark framework are several components that work together to enable distributed data processing. These include the Spark Driver Program, the Cluster Manager, and Executor Processes. Let's see them one by one in the following sub sections.


The Resilient Distributed Dataset (RDD), a fundamental data structure in Apache Spark, is an immutable distributed collection of objects partitioned across a cluster of machines. RDDs are designed to be fault-tolerant, meaning they can automatically recover from failures, thereby ensuring the reliability of data processing tasks. This fault tolerance is achieved through a concept known as lineage information, which allows Spark to rebuild lost data partitions due to node failures [1]. The RDD abstraction is a powerful one, as it allows developers to perform complex transformations and actions on data without having to worry about the underlying details of distribution and fault tolerance. This high-level abstraction is one of the key reasons for Spark's popularity for big data processing tasks [24].

However, while RDDs are a powerful tool, they have some limitations, particularly when it comes to processing structured data. To address these limitations, Spark introduced two additional abstractions built on top of RDDs: DataFrames and Datasets. The main interest of this research is Apache Sparks DataFrames. The experiment is done by using that DataFrame API and then comparing the performance, which is the execution time to Pandas DataFrame, as explained in the introduction.

DataFrames and Datasets provide a higher-level, more user-friendly API for manipulating data, and they take advantage of Spark's Catalyst optimizer for query optimization, which is discussed in section 2.1.4.3. This can result in significantly better performance compared to raw RDD operations [24]. Furthermore, these abstractions provide a more intuitive interface for dealing with structured data, similar to what you might find in a relational database or in data analysis libraries like Pandas. In the context of a single-node, multi-core machine, these abstractions still provide significant benefits. While you won't see the benefits of distributed computing on a single node, you can still take advantage of parallel processing across multiple cores [5]. Spark can partition your data across multiple cores and perform operations on each partition in parallel, significantly speeding up processing time for large datasets [22].

In Apache Spark, operations on data are categorized into two types: transformations and actions [5] [17].

- *Transformations* are operations that create a new RDD (Resilient Distributed Dataset) or DataFrame from an existing one. Examples of transformations include `filter()`, `map()`, and `groupByKey()`. These operations are lazily evaluated, meaning they are not executed immediately when called but are recorded and only executed when an action is called.
- *Actions*, on the other hand, are operations that return a value to the driver program or write data to an external storage system. Examples of actions include `count()`, `first()`, `take()`, and `collect()`. Actions trigger the execution of the transformations that have been recorded.



```
In [5]: from pyspark.sql import SparkSession
# Create a SparkSession
spark = SparkSession.builder \
    .appName("Transformation and Action Example") \
    .getOrCreate()

In [6]: # Create a DataFrame
df = spark.createDataFrame([
    ("scale",),
    ("Java",),
    ("Hadoop",),
    ("spark",),
    ("Akka",),
    ("spark vs Hadoop",),
    ("pyspark",),
    ("pyspark and spark",)
], ["Words"])

# Transformation: Filter the DataFrame to only include rows where the word starts with 's'
df_s = df.filter(df.Words.startswith('s'))

# Action: Count the number of rows in the filtered DataFrame
count_s = df_s.count()

print("Number of words starting with 's':", count_s)

Number of words starting with 's': 3
```

Figure 5: Transformation and Action in Spark

Let's consider an example in which we begin with a DataFrame and apply several transformations and actions. For this particular example and this research in general, we will utilize PySpark, the Python API for Spark [41].

In this example, as depicted in Figure 5 above, our initial step involves creating a `SparkSession`, which serves as the entry point for all Spark functionalities. We then create a `DataFrame` with a list of words. The `filter()` method is a transformation that creates a new `DataFrame`, including only the rows where the word starts with 's'. The `count()` method is an action that returns the number of rows in the `DataFrame`. When we call `count()`, this triggers the execution of the `filter()` transformation [22].

This example demonstrates the fundamental concept of lazy evaluation in Spark which will be discussed in section 2.1.4.4, a powerful performance feature of Spark. The `filter()` transformation is not actually executed

when it's called. Instead, Spark records this transformation and only executes it when an action is called. This allows Spark to optimize the entire sequence of transformations at once, leading to more efficient execution [22]. The lower-level details on how the code is compiled and then processed is discussed further in the coming sections.

In order to have a clear understanding of how Spark handles code and to gain a good overview of Spark's architecture components, let's explore the key components that make up the Spark framework. These components work together seamlessly to enable efficient and scalable distributed data processing.

2.1.4.2 Spark Core Components

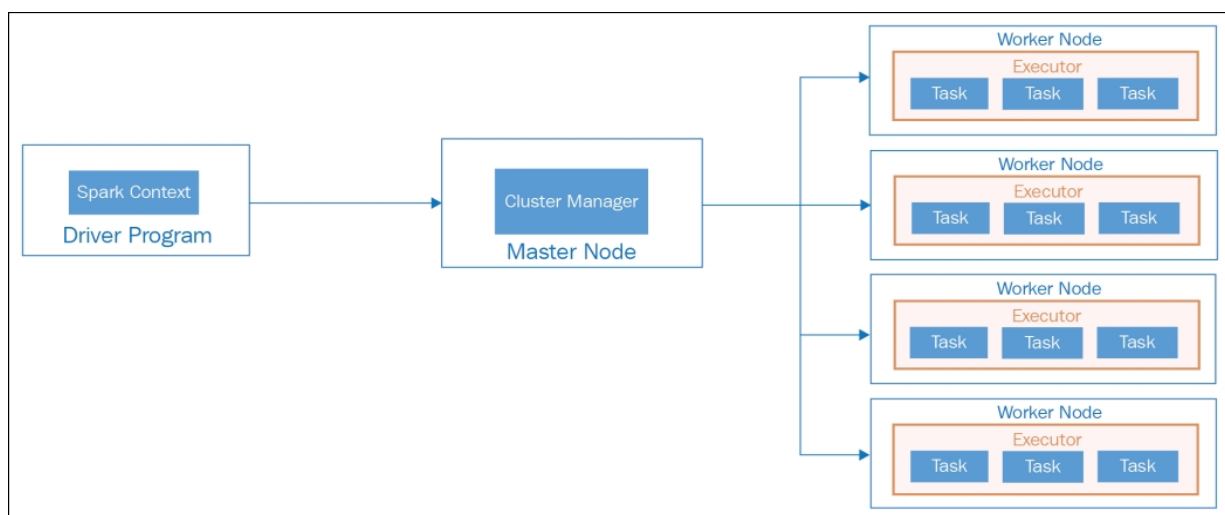


Figure 6: Apache Spark Architecture.

Source: Chambers, B., (2017), *Apache Spark Architecture*. [Online]

The Apache Spark framework is composed of several core components that work together to facilitate distributed data processing. These components include the Driver Program, the Cluster Manager, and Executor Processes as shown in the above Figure 6. In the context of a single-node, multi-core machine, these components function in a slightly different manner compared to a distributed multi-node cluster. Let's discuss what they really are and how they function one by one.

- Driver Program:** The Driver Program is the central coordinator of the Spark application. It runs the user-defined code and submits Spark jobs for execution. In PySpark, the Python script you write serves as the Driver Program. The Driver Program creates a `SparkSession`, which is the main entry point for interacting with the Spark framework [5]. The `SparkSession` manages cluster resources, schedules tasks, and coordinates operations. It provides APIs for creating RDDs and DataFrames, registering User-Defined Functions (UDFs), and custom data types [1] [17]. In a single-node setup, the Driver Program and the `SparkSession` run on the same machine. In such environments, the Driver Program

and the Spark Session would run on the same machine. The Spark Session would manage the resources of the machine, treating each core as a separate worker node. For example, if Pyspark is used to create a DataFrame, the Spark Session would partition the DataFrame across the available cores and schedule tasks to process each partition. The scheduling of tasks would take into account the available resources (CPU and memory) on the machine.

- **Cluster Manager:** The Cluster Manager is responsible for managing resources and scheduling tasks. In a single-node setup, the Cluster Manager's role is somewhat simplified[1]. It doesn't need to manage resources across multiple machines, but it still needs to allocate resources across the different cores of the machine. The Cluster Manager allocates resources to each Spark application, such as CPU, memory, and network bandwidth, based on the application's requirements and the available resources on the machine [18]. In a single-node, multi-core setup, the Cluster Manager would allocate resources across the different cores of the machine.
- **Executor Processes:** Executors are JVM processes that run on worker nodes and are responsible for executing tasks. In a single-node setup, these would run as separate threads across the different cores of the machine. Each executor has its own JVM and runs multiple tasks in separate threads [17]. The executor processes execute the tasks assigned to them by the SparkSession and returns the results to the Driver Program. They also cache intermediate data, such as RDD partitions or DataFrame partitions, in memory or on disk. This cached data can be reused across multiple tasks, reducing the overhead of data shuffling and improving overall performance [17]. In a single-node, multi-core setup, the executor processes would run as separate threads across the different cores of the machine [18].

2.1.4.3 Spark SQL, DataFrames, Query Optimization and Tungsten Execution Engine

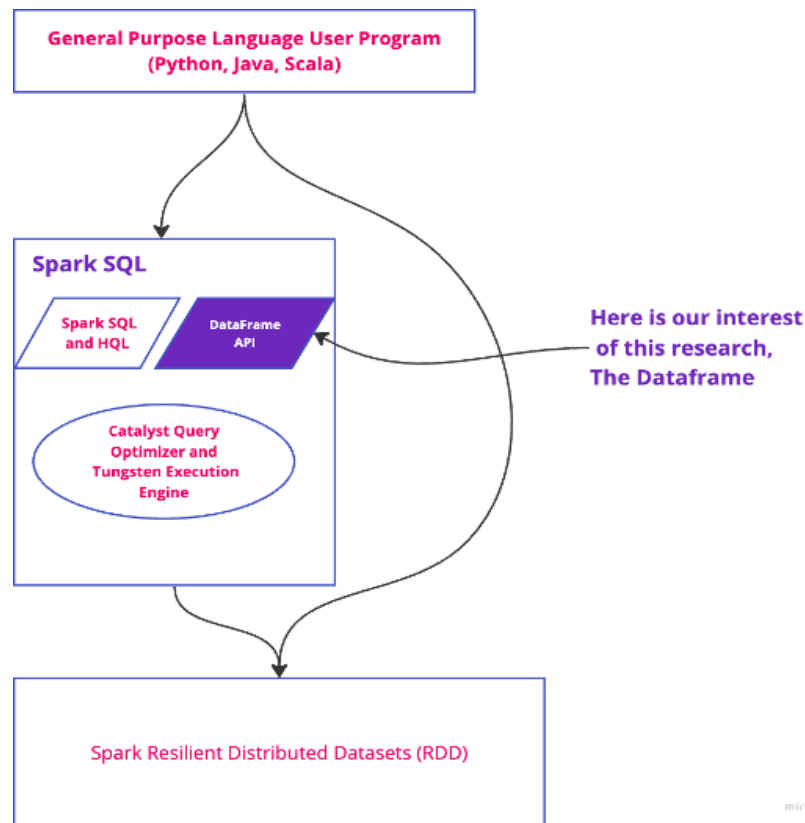


Figure 7: *Spark SQL Module.*

After establishing an understanding of the core components of the Spark framework, this section delves deeper into its Spark SQL Module which contains our interest i.e., the DataFrame (see Figure 7) and its capabilities and the optimization techniques it employs. A critical aspect of Spark's performance advantage is its ability to use Spark SQL and DataFrames for structured and semi-structured data, along with its sophisticated query optimization strategies [22].

◦ Spark SQL and DataFrames

Spark SQL is a Spark module for structured data processing. It provides a programming interface for data manipulation using a relational or structured query language (SQL) as well as a runtime for executing queries [18]. Users can use it to execute SQL queries written using either a basic SQL syntax or HiveQL.

DataFrames, which are the primary focus of this research, are a distributed collection of data organized into named columns [6]. They are designed to make large data sets processing even easier, inspired by data frames in R and Python (Pandas). DataFrames also allow Spark to manage schema, i.e., it lets Spark understand the structure of the data, its computation, and storage format, which leads to more efficient execution using the Catalyst Optimizer [19].

- **Catalyst Query Optimizer**

The Catalyst Query Optimizer optimizes the logical plan of a DataFrame operation or a Spark SQL query to make it execute more efficiently. Catalyst uses a tree transformation framework, where each transformation is a rule that can pattern-match on the query plan and make specific changes to optimize it [18]. Catalyst performs two types of optimizations: rule-based and cost-based. Rule-based optimizations apply general rules that simplify the query plan, such as predicate pushdown or constant folding. Cost-based optimizations use statistics about the data to make decisions, such as which join algorithm to use or in which order to join tables [19].

- **Tungsten Execution Engine**

The Tungsten project aims to improve the efficiency of memory and CPU for Spark applications to push performance closer to the limits of modern hardware. This effort includes three initiatives [39]:

1. Memory Management and Binary Processing: Spark leverages application semantics to manage memory explicitly and eliminate the overhead of the JVM object model and garbage collection.
2. Cache-aware computation: Spark lays out data in memory to exploit the CPU's memory hierarchy and to allow vectorized CPU instructions.
3. Code generation: Spark generates bytecode at runtime that collapses the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data.

On a single-node, multi-core machines like the one used in this research, i.e., M1 MacBook Air, both the Catalyst Optimizer and the Tungsten Execution Engine play crucial roles in optimizing the execution of Spark operations [39]. The Catalyst Optimizer analyzes the query plan and optimizes it based on the characteristics of the data and the available resources on the machine [18]. For instance, it might decide to use a broadcast join instead of a sort-merge join if one of the tables is small enough to fit in memory [39]. This decision-making process is based on the understanding of the data and the computational resources, which allows Spark to perform operations more efficiently.

Simultaneously, the Tungsten Execution Engine manages memory explicitly to reduce the overhead of the JVM object model and garbage collection, which is particularly beneficial in a resource-constrained environment like a single-node machine. Tungsten lays out data in memory in a cache-aware manner to exploit the CPU's memory hierarchy, which can significantly speed up data access times [19]. Furthermore, Tungsten generates bytecode at runtime to collapse the entire query into a single function. This approach allows the CPU to execute the query more efficiently by eliminating virtual function calls and leveraging CPU registers

for intermediate data [19]. The Catalyst Optimizer and the Tungsten Execution Engine work in tandem to optimize both the logical plan of the operations and the physical execution of the operations, ensuring efficient utilization of the machine's resources and high-performance data processing [22].

2.1.4.4 Spark in Action (Execution Operation)

Apache Spark, a distributed computing framework, is primarily written in Scala and runs on the Java Virtual Machine (JVM). It provides APIs in several languages, including Python via PySpark. When you write code in PySpark, your Python code is translated into a series of transformations and actions that are executed on



Figure 8: Spark Lazy Evaluation.

Source: Shukla, R., (2017), *Spark Lazy Evaluation*. [Online]

the JVM. This translation process is complex and involves several steps and components. When you define transformations and actions in PySpark, you create a logical execution plan. This plan is a Directed Acyclic Graph (DAG) of transformations on your data. However, Spark employs a strategy known as lazy evaluation, meaning that these transformations are not immediately executed [1]. Instead, execution is deferred until an action is called, such as `count()`, `collect()`, etc., as shown in Figure 8 below. Rather than executing the operations as soon as they're defined RDD1, RDD2, and RDD3, Spark postpones all executions until an action is invoked [1]. This allows Spark to create an optimized execution plan by grouping multiple narrow transformations into a single stage [22]. This helps reduce the need to create and move data between different stages, making it easier and more efficient. The underlying concept is quite straightforward: Spark merges transformations, such as 'map' and 'filter' operations, that contribute to only one output partition from a single input partition, thereby minimizing the need for extensive data shuffling [17].

Upon the invocation of an action, Spark's Catalyst optimizer comes into play. The Catalyst optimizer takes the logical plan and optimizes it based on various rules, generating a physical plan. This physical plan takes into account the characteristics of the data and how it's partitioned, and it represents the actual operations that will be executed on the cluster. The ability to partition data is a critical enhancement to the performance of Spark DataFrames on single-node machines [17]. This process, explains how sizable datasets can be divided into smaller portions, known as partitions. As illustrated in Figure 9, three partitions (Partition 1, Partition 2, and Partition 3) are created from a five-row data frame. The partitions are done in a low-level structures RDD. For instance, if we want to apply the filter in this data frame, Partition 1, Partition 2, and Partition 3's row is filtered independently. Optimal partitioning and Spark strike a balance between read performance and write

performance [17]. Too many partitions can slow down read time and burden Spark with creating more tasks than necessary to process the data, which can cause an out-of-memory error in the driver.

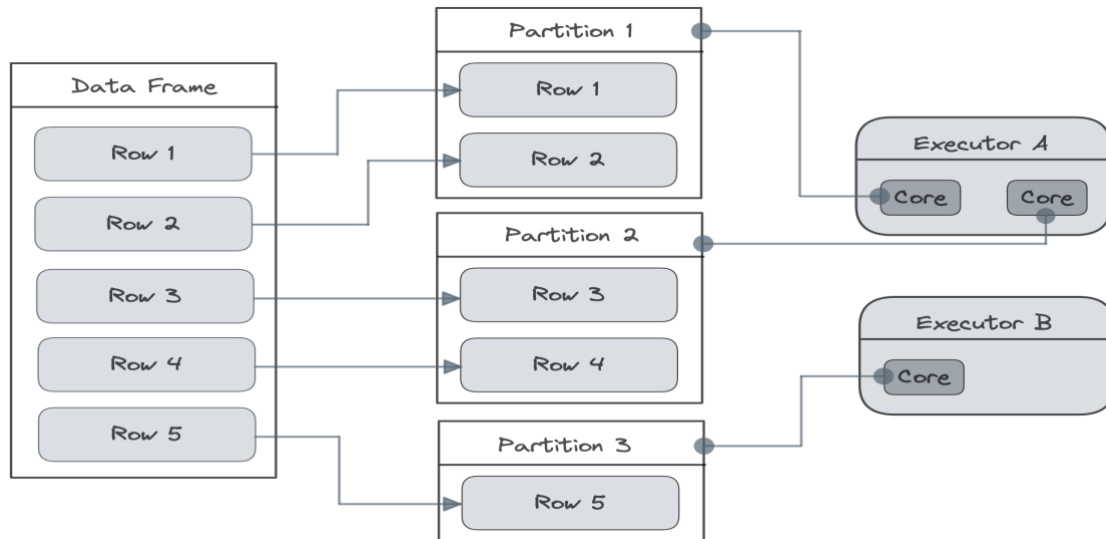


Figure 9: Spark Data Partitioning.

Source: Nayagan, S., (2022), *Spark Data Partitioning*. [Online]

The physical plan is then compiled into bytecode that can be executed on the JVM [40]. This is achieved using a technique known as whole-stage code generation, which fuses multiple operations together into a single function. This function is then compiled into bytecode. This approach reduces the overhead of interpreting the operations and allows the JVM to further optimize the code. The bytecode is then dispatched to the Spark executors, which are running on the JVM [41]. These executors could be running on the same machine in local mode which this research is focused on, or on different machines in a cluster [40]. The executors execute the bytecode in parallel across multiple threads, which can be run on different cores or even different machines. When data needs to be transferred between the Python driver program and the JVM, it needs to be serialized. PySpark uses a library called Pyrolite for this purpose [41]. Pyrolite is capable of serializing Python objects into a format that can be deserialized by the Java library Pyro.

When you're using the DataFrame API in PySpark, under the hood, everything is expressed in terms of Resilient Distributed Datasets (RDDs), which are Spark's fundamental data structure. When you call a DataFrame transformation, it becomes a set of RDD transformations. This translation process is transparent to the user but is crucial for the execution of the operations on the JVM [41]. In some cases, you might need to drop down to the RDD API for more fine-grained control over your data processing tasks. For instance, you might need to use some legacy code, implement a custom partitioner, or update and track the value of a variable over the course of a data pipeline's execution [41]. While the RDD API gives you more control, it comes at

the expense of the optimizations provided by the DataFrame API and Catalyst optimizer. PySpark provides a Pythonic interface to Spark, but the heavy lifting is done by the Spark engine running on the JVM [40]. The Python code you write is translated into a series of transformations and actions that are optimized and executed on the JVM. This process is complex and involves many components and layers of abstraction, but it allows Spark to provide a high-level, user-friendly API without sacrificing performance.

2.1.5 Architectural Comparison: Spark vs. Pandas DataFrame

In terms of a deep technical comparative analysis, let's dive into the considerations for Pandas and Spark in a single machine with multiple cores, primarily from a speed of execution perspective.

2.1.5.1 Pandas Execution Time Perspective:

Pandas primarily operate in memory, meaning that the speed of their operations can be tightly linked to the memory hierarchy of your machine. Data that are kept in CPU cache (L1, L2, or L3) can be accessed significantly faster than data stored in RAM [2]. Consequently, Panda's operations that fit within the CPU cache can be executed with a substantially reduced latency.

Underneath, Pandas uses NumPy for numerical computations. NumPy internally leverages libraries like OpenBLAS or Intel MKL, which are written in low-level languages like C or Fortran and are highly optimized [13]. These libraries have parallelized routines that use multiple cores for computations, thus overcoming Python's GIL limitation [13]. This means that complex mathematical operations on large arrays can be performed at near C-speed thanks to these libraries.

However, this doesn't imply all operations are parallelized. In particular, operations that can't be vectorized (for instance, operations that depend on the value of the previous operation) might not see speed improvement with more cores. Moreover, the overhead of splitting tasks and gathering results (task scheduling overhead) could make multi-threading less effective for smaller data sizes or simpler tasks.

2.1.5.2 Spark Execution Time Perspective:

Spark was originally designed for distributed computing but can operate on a single machine with multiple cores [1] [18]. Each task in Spark runs in a separate JVM thread, which means that in a multi-core environment, it can execute multiple tasks in parallel. Spark's RDDs (Resilient Distributed Datasets) and DataFrames are split into partitions, which can be processed independently and in parallel across different threads [1].

While Spark can theoretically utilize all available cores effectively, in practice, it might face some limitations in a single-machine, multi-core scenario. Firstly, each Spark task carries a certain amount of overhead - tasks

need to be serialized and sent to executors, and results need to be collected and deserialized [22]. For smaller datasets or simpler tasks, this overhead might significantly affect the speed of execution, making Spark slower than Pandas. Secondly, Spark's lazy evaluation model means that transformations aren't executed until an action is called [18]. While this allows for optimizing the overall execution plan, it might lead to higher latency for individual operations. Lastly, if you are using PySpark, there is an additional overhead due to the data having to be serialized and deserialized between Python and JVM (Java Virtual Machine), which might affect performance [22].

- **Takeaway and reflection on research question one**

In a single-machine, multi-core environment, Pandas tends to be faster for smaller datasets (that fit in memory) or tasks that can be efficiently vectorized, thanks to the highly optimized libraries it uses underneath [2] [6].

Spark, however, might be slower for these kinds of tasks due to the overhead of task serialization, scheduling, and (if using PySpark) data serialization between Python and JVM [5]. But for larger datasets (that can still fit in the memory of a single machine) or more complex tasks involving multiple transformations, Spark can potentially outperform Pandas thanks to its ability to parallelize tasks across multiple cores and optimize execution plans [22].

However, these are general trends. The actual performance can vary depending on many factors, including the specific hardware (CPU architecture, cache size, memory speed, etc.), the nature of the tasks, and the implementation of specific operations in Pandas and Spark [12]. Hence, when performance is critical, it's always a good idea to benchmark different approaches on representative workloads and hardware [5].

Criteria	Pandas DataFrame	Spark DataFrame
Execution Environment	Python interpreter, leveraging multi-core through low-level libraries	Runs on the JVM, can use multiple cores on a single node through multi-threading
Parallelism	Implicit, through underlying libraries like OpenBLAS or Intel MKL	Explicit, through partitioning of RDDs and DataFrames
Language	Python, with underlying libraries in C and Fortran	Written in Scala, with APIs in Python, Scala, Java, and R

Execution Speed	Typically, faster for smaller datasets (fit in memory) and operations that can be vectorized	Potentially slower for small datasets due to overhead, but can outperform on larger data or complex tasks
Lazy Execution	Operations are executed as they are called, Poor execution planning.	Yes, operations are optimized and executed only when an action is called
Overhead	Lower, mainly due to task scheduling for multi-threaded operations	Higher due to task serialization, scheduling, and (if PySpark is used) data serialization between Python and JVM
Memory Management	Less efficient, primarily relies on Python's memory management, but efficient when using Numpy	More efficient and fine-grained through the use of JVM
GIL (Global Interpreter Lock)	Bypassed through lower-level C/Fortran code in NumPy and other libraries	Not an issue as Spark runs on JVM

Table 2: *Summarization of Architectural Differences.*

So far until this stage, this research has provided:

- A comprehensive examination of the Pandas architecture, including an in-depth exploration of the DataFrame object and its dependence on NumPy, as well as the lower-level operations that govern its behavior.
- An extensive discourse on the architecture of Spark, with a particular focus on the SQL module - which houses the DataFrame. We delve into the mechanisms of code execution and optimization algorithms used by Spark.
- A theoretical response to our first research question: "How do the architectural differences between Pandas DataFrames and Spark DataFrames impact their performance?"

3 Results from Experiment

In this section, we delineate the findings of our experiment designed to address the second research question of our study: “How does the performance of Pandas DataFrames and Spark DataFrames compare when processing the COVID-19 research dataset?”. Building on the theoretical foundation laid by the topics, we endeavor in this section to contextualize these theories in a practical setting. We aim to evaluate and quantify the performance of equivalent built-in functions of these two DataFrames, offering an empirical perspective on how the intricacies of their respective architectures influence execution time, which serves as our performance metric.

3.1 Experiment Setup

The performance assessment of Pandas and Spark DataFrames in handling the COVID-19 research dataset involves a multi-faceted procedure. Starting with Data Collection and Segmentation, wherein we partition the dataset into manageable segments, we then proceed to measure the execution time. We conducted our experiments on a single-node computer, specifically, a MacBook Air M1 2020 model as discussed earlier and notably for process monitoring, and we employed Htop. Htop is a process viewer and a text-mode application for system monitoring. Have a look at the following Figure 10 to see the interface of Htop.

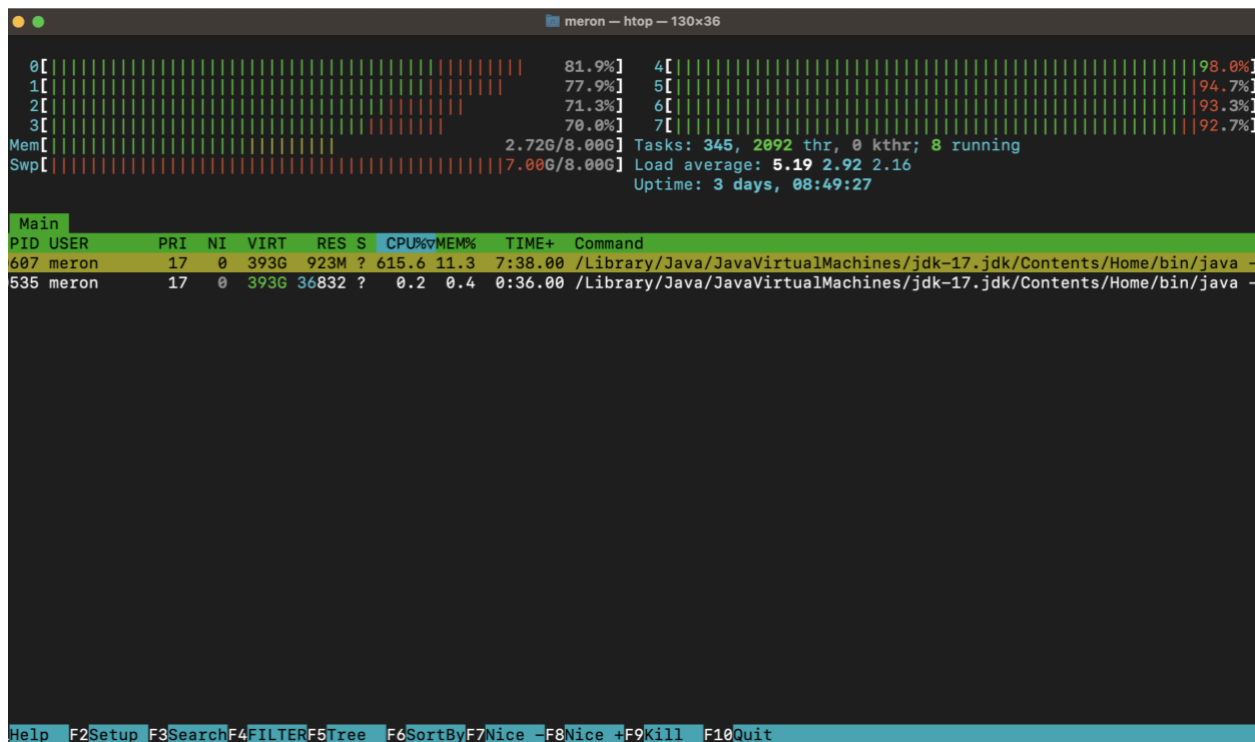


Figure 10: Htop Process Monitoring

Htop is an interactive system-monitoring utility used to view and manage system resources on Unix-like operating systems [44]. It provides a user-friendly interface to monitor processes, CPU usage, memory usage, and other system metrics in real time. The number of CPU cores available on the test machine system is displayed at the top of the Htop interface (0-7), as shown in upper Figure 10. The values fluctuate every two seconds because the CPU continuously allocates and shuffles resources during the execution process. The same applies to Memory, represented as 'Mem' in the Htop interface. The 'Swp' mark below the 'Mem' label in the figure above measures the swapping space, i.e., the virtual memory allocated as a backup when larger RAM space is needed. It is quite elastic, so the maximum virtual memory can approach the available space in the secondary memory. This elasticity made the measurement of CPU and Memory inaccurate in this research. However, this presents a good opportunity for further studies and is referred to in the 'Future Works' section at the end of the research.

It was crucial to optimize the system by terminating all nonessential processes to reduce the swapping rate imposed by the operating system. However, the M1's operating system was still observed to leverage secondary memory (virtual memory) for loading and executing the algorithms pertinent to Pandas and Spark, as a means to circumvent potential memory exhaustion. It is important to note that reaching the limit of secondary memory could lead to a system halt. However, in the experimental setup of this testing machine, the secondary memory was sufficient enough that even when we tried to load larger datasets than the system RAM could handle, the OS was still managing the situation. The experiment specifically investigated how vectorization, enabled by NumPy in Pandas and the lower-level actions discussed in the previous sections, competes with Spark's execution algorithms, specifically the Catalyst Optimizer and the Tungsten engine, in the context of this specific dataset.

3.1.1 Data Preparation and Segmentation

Data segmentation was a crucial step in the performance evaluation of data processing tools, especially when dealing with large datasets. In this context, the dataset, which comprises structured float numbers and contains 1,056,659 samples/rows and 769 columns, is segmented into 25 distinct chunks (see Figure 11 below). These chunks range from 0.1GB to 15GB, providing a comprehensive spectrum for assessing the performance of Pandas and PySpark. The strategic choice of these chunk sizes serves multiple purposes. The smaller chunks, starting from 0.1GB, serve as a baseline for performance assessment with manageable data volumes, and the larger chunks, ranging from 2GB to 15GB, push the boundaries of what can be processed on a single machine. These sizes are more representative of real-world "big data" scenarios and provide valuable insights into how each tool copes with processing such substantial data volumes in a single machine.

Performance_measurement

output

data1.csv

data2.csv

data3.csv

data4.csv

data5.csv

data6.csv

data7.csv

data8.csv

data9.csv

data10.csv

data12.csv

data13.csv

data14.csv

data15.csv

data16.csv

data17.csv

data18.csv

data19.csv

data20.csv

data21.csv

data22.csv

data23.csv

data24.csv

cord_19_embeddings_2022-06-02.csv

Today at 22:36

Today at 22:36

6 Jun 2023 at 17:17

6 Jun 2023 at 17:34

6 Jun 2023 at 17:34

6 Jun 2023 at 17:34

6 Jun 2023 at 17:35

6 Jun 2023 at 17:35

6 Jun 2023 at 17:36

6 Jun 2023 at 17:37

6 Jun 2023 at 17:38

6 Jun 2023 at 16:51

6 Jun 2023 at 17:11

6 Jun 2023 at 17:41

6 Jun 2023 at 17:09

6 Jun 2023 at 17:53

6 Jun 2023 at 18:10

6 Jun 2023 at 18:19

6 Jun 2023 at 18:29

6 Jun 2023 at 18:42

6 Jun 2023 at 19:22

6 Jun 2023 at 19:51

6 Jun 2023 at 19:37

6 Jun 2023 at 20:07

6 Jun 2023 at 20:28

6 Jun 2022 at 20:41

--

--

99,9 MB

199,9 MB

299,9 MB

399,8 MB

499,8 MB

600 MB

700,2 MB

800,4 MB

900,4 MB

1 GB

2 GB

3 GB

4 GB

5 GB

6 GB

7 GB

8 GB

9 GB

10 GB

10,99 GB

11,99 GB

12,99 GB

13,99 GB

15,58 GB

Folder

Folder

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Comm...et (.csv)

Figure 11: *Dataset Chunks after segmentation*

Consequently, the results of this experiment are analyzed and discussed in the Discussion section, aiming to draw a conclusion on the performance comparison of Pandas DataFrames and Spark DataFrames.

3.2 Performance Evaluation

The performance of these operations is measured across different data sizes, ranging from 0.1 GB to 15 GB. This wide range of data sizes allows for a comprehensive evaluation of how these frameworks scale with increasing data size, a key consideration in big data applications, and it is conducted through three primary operations: data loading, data filtering, and data transformation. Each operation is performed in 10 cycles, and the time taken is averaged to balance out minor variations and provide a more accurate measure of the operation's typical performance. This approach helps to mitigate the impact of outliers and provides a more reliable measure of performance. It's important to note that the performance of these operations is dependent on the computational resources available, and the time taken for these operations can vary significantly on different machines with different specifications. Table 3 tries to show the whole result in one numerical structure, as shown below.

Dataset Size	Data Loading		Filtering and Selection (No Action)		Filtering and Selection (Action)		Transformation - Doubling a Single Column	
	Pandas DF	Spark DF	Pandas DF	Spark DF	Pandas DF	Spark DF	Pandas DF	Spark DF
0.1GB	0.36285	1.57070	0.00349	0.08031	0.01468	0.59102	0.00201	0.59104
0.2GB	0.71348	2.80850	0.00473	0.04739	0.02710	0.96527	0.01350	0.47522
0.3GB	1.06954	4.16848	0.00702	0.04467	0.03160	1.47752	0.01650	0.77752
0.4GB	1.44079	5.90993	0.00936	0.04481	0.04065	1.72583	0.02962	0.92683
0.5GB	1.83271	7.11664	0.01178	0.04563	0.04493	1.98006	0.04314	1.08006
0.6GB	2.15857	8.18952	0.01631	0.04437	0.18973	2.34320	0.06990	1.34354
0.7GB	2.57580	9.38334	0.01802	0.04468	0.32440	2.58273	0.09963	1.57263
0.8GB	2.96075	10.13698	0.01983	0.04431	0.45208	2.99867	0.15137	1.74068
0.9GB	3.24031	11.67882	0.03158	0.04232	0.67745	3.46062	0.22397	2.09046
1GB	3.57463	13.04607	0.03112	0.04756	1.13371	4.07885	0.89684	3.47851
2GB	7.41703	27.06401	0.08747	0.04363	1.66391	8.18545	1.36950	8.55526
3GB	11.15301	40.34847	0.18891	0.04363	1.86188	11.68531	1.53778	12.77538
4GB	15.89599	53.89822	0.72862	0.04366	2.39572	14.29990	2.95263	18.39595
5GB	19.46477	67.64213	1.95690	0.04366	2.62604	20.11396	3.39114	23.51394
6GB	25.89928	104.29503	1.55108	0.04396	3.90013	26.22346	5.91480	37.82342
7GB	39.33730	132.03032	4.82007	0.04485	4.16967	33.07037	7.63846	46.97034
8GB	48.52571	149.14590	5.15704	0.04469	6.31762	39.17328	9.76212	55.27322
9GB	60.61468	170.31554	11.44873	0.04246	13.70582	47.76579	16.88578	68.676577
10GB	76.24155	233.77961	19.53639	0.04135	24.42966	69.21383	48.00944	79.31385
11GB	86.79467	250.62885	27.85096	0.03210	33.04504	88.63960	62.13310	88.83967

12GB	97.14161	249.09140	36.64158	0.04341	48.40607	96.57020	74.25676	96.27027
13GB	135.47625	277.08775	50.19761	0.04847	61.89603	109.80649	95.38042	119.70643
14GB	153.77703	341.47879	69.10705	0.04162	93.21290	120.58435	115.50408	138.88432
15GB	161.32567	402.70341	82.91411	0.49303	118.64564	134.09829	135.85202	145.89107

Table 3: Execution Time Result

For more comprehensive understanding and clarity, the following section will focus on the graphical representation of the results individually for each method tested. Seaborn offers a high-level, flexible interface for creating informative statistical graphs. Therefore, it is used to generate the following graphs for the upper shown result in Table 3. Let's see the Built-in methods used to measure the execution time one by one.

3.2.1 Data Loading

Data loading, the initial phase of data processing, involves importing data into the environment for manipulation and analysis. In our comparison of Pandas and Spark DataFrames, we measure the execution time of the data loading to a DataFrame from a CSV file. The method is shown in the following Figure 12 along with the achieved results. This measurement provides a baseline for understanding each framework's performance characteristics. Despite both Pandas and Spark offering data-loading functionalities, their underlying implementations and optimization strategies can differ, leading to performance variations.

```
# Reading a CSV file

# Pandas
def read_csv_pandas(x):
    df = pd.read_csv(x)
    return df

# Spark
def read_csv_spark(x):
    df = spark.read.option('header', 'true').csv(x, inferSchema=True)
    return df
```

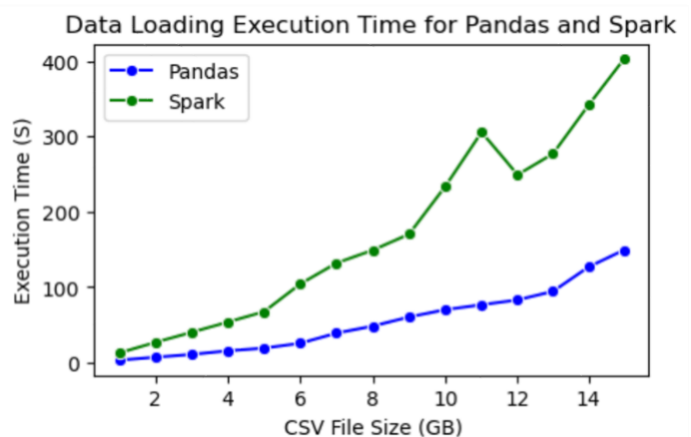
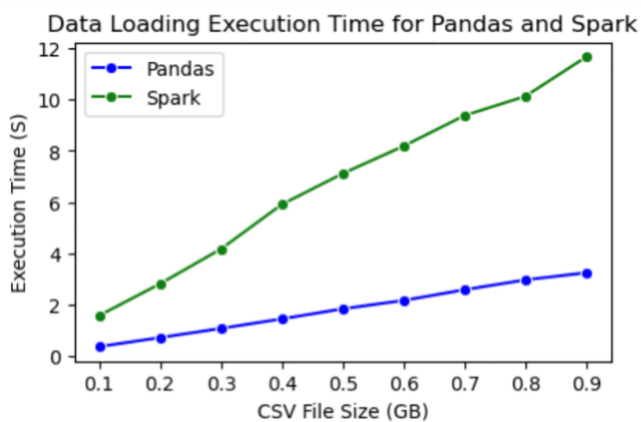


Figure 12: Data Loading and Results

3.2.2 Filtering and Selection

The second measurement tests both DataFrames on their performance when filtering a value from a single column. It is done in two parts as follows:

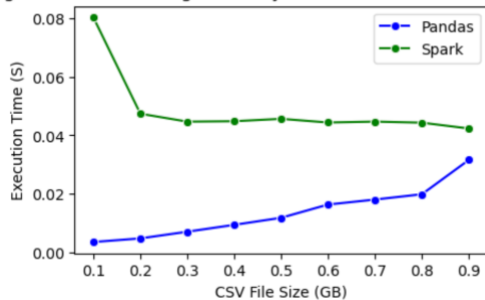
Filtering without an Action: The first set of functions, `filter_and_select` for both Pandas and Spark, perform a filtering operation on the data. They select a row from the DataFrame where the values in the column "5.164162635803223" are between 6 and 7, inclusive. In the context of Pandas, the `filter_it.loc[]` operation is a label-based data selection method, which means that we are referring to the labels of the rows (or columns) that we want to filter out. This operation is straightforward and executes immediately, following the eager execution paradigm of Pandas. On the other hand, the Spark filter function is a transformation operation. In Spark's lazy evaluation model, transformations are not immediately executed. Its influence on the following Figure 13 metrics is further discussion in the Result analytics section.

```
# Filtering from a column "5.164162635803223" which are in values b/n and including 6 and 7.

# Pandas
def filter_and_select(filter_it):
    return filter_it.loc[(filter_it["5.164162635803223"] >= 6) & (filter_it["5.164162635803223"] <= 7)]

# Spark
def spark_filter_and_select(df):
    return df.filter((df["5.164162635803223"] >= 6) & (df["5.164162635803223"] <= 7))
```

Filtering and Selection - eager vs Lazy Execution Time for Pandas and Spark



Filtering and Selection - eager vs Lazy Execution Time for Pandas and Spark

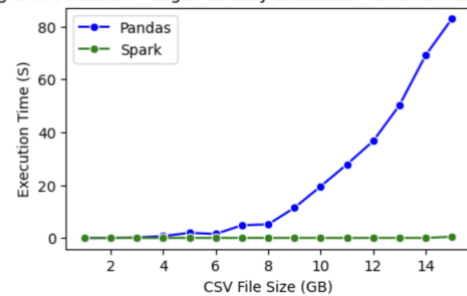


Figure 13: Filter without Action and Results

Filtering with an Action: The second set of functions, `filter_and_select` for both Pandas and Spark, perform the same filtering operation as in Part One, but they also invoke an action operation: `shape[0]` for Pandas and `count()` for Spark. In Pandas, `shape[0]` returns the number of rows in the DataFrame, which forces the computation of the filtering operation. This is consistent with Pandas' eager execution model. In Spark, `count()` is an action operation that returns the number of rows in the DataFrame. When `count()` is called, Spark executes all the recorded transformations (in this case, the filter operation) and returns the result. This is a key aspect of Spark's lazy evaluation model: computations are delayed until an action operation is invoked, which can

lead to more optimized execution plans and can save computational resources when dealing with large datasets. The methods and their result are shown in the following Figure 14.

```
# Filtering with an action

# Pandas
def filter_and_select(filter_it):
    return (filter_it.loc[(filter_it["5.164162635803223"] >= 6) & (filter_it["5.164162635803223"] <= 7)]).shape[0]

# Spark
def spark_filter_and_select(df):
    return df.filter((df["5.164162635803223"] >= 6) & (df["5.164162635803223"] <= 7)).count()
```

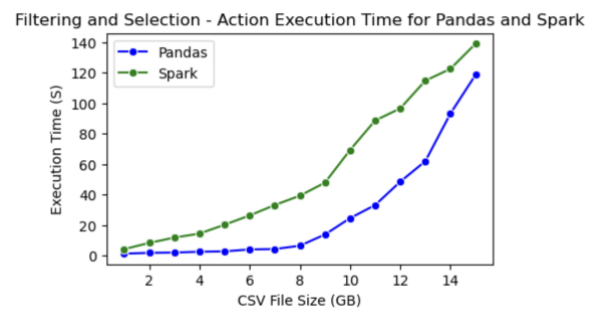
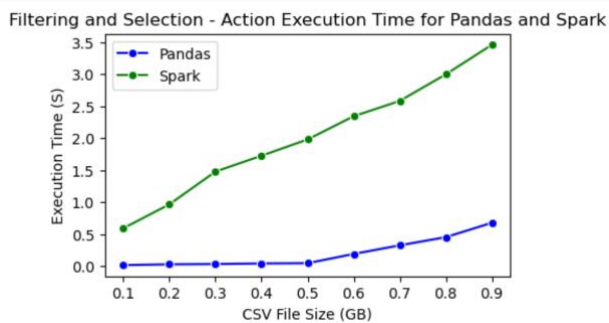


Figure 14: Filter with Action and results

3.2.3 Data Transformation

The `doubler_pandas` and `spark_doubler` functions perform a transformation operation on a specific column ("5.164162635803223") in the DataFrame, doubling its values. Observe the following Figure 15 for the method and its results.

```
# Doubling the values in the column "5.164162635803223".

# Pandas
def doubler_pandas(df):
    df["5.164162635803223"] = df["5.164162635803223"] * 2
    return df.shape[0]

# Spark
def spark_doubler(df):
    return df.withColumn("`5.164162635803223`", col("`5.164162635803223`") * 2).count()
```

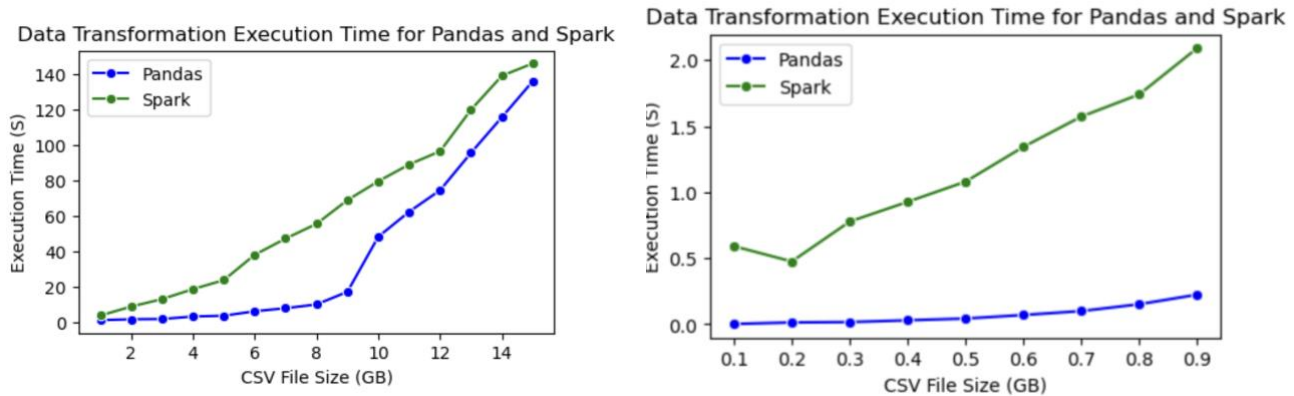


Figure 15: Data Transformation and Results

4 Results Analysis

Having presented the raw results of our comparative study, we now try to drive a deeper analysis of these findings. This section aims to provide a comprehensive interpretation of the results, elucidating the performance characteristics of Pandas and Spark DataFrames in various data processing tasks. We will dissect the execution times of data loading, filtering, and transformation operations and attempt to understand the underlying factors contributing to the observed performance. This understanding can guide us in choosing the right tool for the right task, optimizing our data processing workflows, and even contributing to the development and improvement of these tools. Let's start seeing the results one by one:

4.1 Data Loading

In order to analyze how the results turned out to be, as shown in Figure 12, one needs to consider the underlying architecture of these libraries, the nature of the data being processed, and the hardware on which these operations are performed. Pandas, written in Python, leverages a C-based library, `pandas.io.parsers`, for reading CSV files [38]. This library is highly optimized for performance and can take advantage of modern CPU features when you call `pd.read_csv`, Pandas uses the `pandas.io.parsers` library to read the CSV file. This process involves several low-level operations:

- Disk I/O: The CSV file is read from the disk into the system's memory. This operation's speed depends on the disk's read speed and the efficiency of the operating system's file system.

- **Parsing:** The raw data is parsed into a DataFrame, a process that involves interpreting the CSV format, converting strings into appropriate data types (e.g., integers, floats, dates), and handling missing values [22]. This operation is CPU-intensive and can be influenced by the CPU's speed and the efficiency of the parsing algorithm.
- **Memory Allocation:** The parsed data is stored in a DataFrame in memory. This operation's speed depends on the system's available memory and the efficiency of the memory management system [6].

On the other hand, Spark, written in Scala, operates on the JVM and uses a Java-based CSV parser for parsing the data. Spark's data-loading operation is distributed and can be parallelized across multiple CPU cores in a single machine. This involves additional steps:

- **Partitioning:** The CSV file is divided into partitions, which can be processed independently. The number of partitions is typically much larger than the number of CPU cores to allow for load balancing.
- **Task Scheduling:** Tasks (i.e., reading and parsing operations) are scheduled to run on different CPU cores. This operation is managed by Spark's scheduler, which aims to minimize data movement and balance the load across the cores [1].
- **Disk I/O and Parsing:** Similar to Pandas, these operations involve reading data from the disk and parsing it into a data frame. However, these operations are performed independently on each partition.
- **Shuffling:** After the data is loaded into DataFrames, it may need to be re-distributed across the partitions to ensure balanced data distribution. This operation, known as shuffling, can be expensive as it involves data movement across the network [39].
- **Garbage Collection:** As Spark operates on the JVM, it is subject to garbage collection (GC), which can pause the execution of the program to reclaim memory. Frequent or long GC pauses can significantly impact the performance of Spark applications [34].

In this experiment result finding shows that pandas outperform spark DataFrames when reading CSV files on a single computer.

4.2 Filtering and Selection

- **Filtering without an Action:** Figure 13 shows that Spark Dataframe execution is not being affected by the increase in the data size. In Pandas, the `filter_it.loc[]` operation is used, which is a label-based data selection method. This operation is straightforward and executes immediately, following the eager

execution paradigm of Pandas. The operation involves scanning each value in the specified column and checking if it meets the specified condition. This operation is CPU-intensive, and its performance can be influenced by the CPU's speed and the efficiency of the comparison operation. On the other hand, Spark's **filter()** function is a transformation operation. In Spark's lazy evaluation model, transformations are not immediately executed. Instead, Spark records the transformations applied to the base dataset (e.g., a file), forming a lineage. The transformed data is not materialized until an action operation is invoked. In this case, the **filter()** function simply records the filtering operation in the DataFrame's lineage, and no computation happens at this stage. This is briefly discussed in the Literature Review section. The spike in execution time for the first measurement when the data size is 0.1 GB could be attributed to the initial compilation of instructions and the generation of an execution plan by Spark's Catalyst optimizer. This initial overhead is expected when running Spark jobs for the first time or when there are changes in the data or execution environment. Once the plan is generated, subsequent executions with similar data sizes may not incur the same overhead, resulting in more consistent execution times.

- **Filtering with an Action:** In this result, comparing the performance of filtering operations in Pandas and Spark, but this time, invoking an action operation: `shape[0]` for Pandas and `count()` for Spark. The action operation forces the computation of the filtering operation, which provides a more accurate measure of the performance of the filtering operation in both frameworks. In Pandas, `shape[0]` returns the number of rows in the DataFrame, which forces the computation of the filtering operation. This operation involves scanning each value in the specified column, checking if it meets the specified condition, and then counting the number of rows that meet the condition. In Spark, `count()` is an action operation that returns the number of rows in the DataFrame. When `count()` is called, Spark executes all the recorded transformations (in this case, the filter operation) and returns the result. This is a key aspect of Spark's lazy evaluation model: computations are delayed until an action operation is invoked, which can lead to more optimized execution plans and can save computational resources when dealing with complex operation codes. The performance of the filtering operation with an action in Pandas and Spark on the M1 chip with 8 CPU cores gave Pandas DataFrames to execute faster. Pandas, it's primarily designed for single-threaded operations, meaning it typically utilizes only one core of the CPU. However, some operations in Pandas, like `shape[0]`, leveraged the vectorized operations provided by the underlying NumPy library, which can utilize multiple cores and leverages super-fast language which is C (via Cython) and benefit from the high-performance cores of the M1 chip.

4.3 Transformation

The operation in question involves doubling the values in a specific column ("5.164162635803223") of the DataFrame. In the Pandas implementation, the operation `df["5.164162635803223"] * 2` is a vectorized operation that doubles each element in the column. This is a big factor in better performance. This is further discussed in the discussion section. The operation is performed in memory and can take advantage of the multiple high-performance cores. In contrast, in the Spark implementation, the `col("5.164162635803223") * 2` operation is a transformation that creates a new column by doubling the values in the existing column. This operation is lazily evaluated, meaning it's not executed immediately but recorded in the DataFrame's lineage graph. The actual computation is triggered when an action operation, in this case, `count()`, is invoked. Pandas is found to outperform Spark in this scenario, even though the dataset gets bigger, but still, Spark did not stand a chance. The performance of the data transformation operation in Pandas and Spark on the M1 chip is influenced by a complex interplay of factors, including the vectorized operations in Pandas, the lazy evaluation and optimization strategies in Spark, the JVM overhead and GC in Spark, and the architecture of the M1 chip.

5 Discussion

The observed performance difference between Pandas and Spark in this experiments can be attributed to a many factors, each of which is deeply rooted in the design principles, execution models, and optimization strategies of these two data processing libraries. Pandas, being a library designed for in-memory data analysis, is highly optimized for single-node, single-threaded operations. It leverages the power of NumPy, a library that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy's operations are implemented in C, which allows for efficient computation, and are vectorized, meaning they operate on entire arrays at once instead of individual elements. This vectorization allows Pandas to take full advantage of modern CPU features such as Single Instruction, Multiple Data (SIMD), where one instruction operates on multiple data points simultaneously across multiple CPU cores. This is particularly effective on the M1 chip, which has advanced support for SIMD operations. Moreover, Pandas uses Cython for executing critical code paths, designed to give C-like performance with code that is written mostly in Python. It does this by translating Python code to C and allowing Python code to call C functions directly, which lead to significant performance improvements, especially for CPU-bound tasks. This is the main factor of the impressive results achieved in this research.

On the other hand, Spark is optimized for large-scale, multi-node, multi-threaded operations. It uses the JVM for execution, which introduces an additional layer of abstraction and overhead. While the JVM's Just-In-Time (JIT) compiler can optimize the bytecode execution, the JVM overhead, including garbage collection, can still impact the performance. Spark's Catalyst optimizer and Tungsten engine are designed to optimize the

execution plan and provide efficient memory use and faster processing speed. However, For single-node operations, the overhead of task scheduling, data partitioning, and serialization can outweigh the benefits of these optimizations. Additionally, Spark's lazy evaluation model, while it can lead to more optimized execution plans, can also introduce overhead. Each transformation in Spark creates a new DataFrame and records the operation in the DataFrame's lineage graph. When an action operation is invoked, Spark computes the entire lineage graph, which can involve multiple stages of task scheduling, data shuffling, and serialization/deserialization. This overhead can impact the performance, especially for smaller datasets or operations that involve a large number of transformations. Lastly, the I/O operations in Pandas are highly optimized. The `pandas.io.parsers` module provides robust and efficient parsing capabilities for various file formats, including CSV. This led Pandas to perform faster on data loading times compared to Spark.

6 Conclusion

In this thesis, we explored and compared the architectural differences and performance of Pandas and Spark DataFrames for data processing tasks. We discussed the fundamental concepts, features, and components of both frameworks and conducted a comprehensive evaluation of their execution time using the Covid-19 Research dataset obtained from Kaggle. Our findings indicate that Pandas DataFrames excel in performance in a single machine. However, they are limited by the available memory on a single machine and may not be the best choice for large-scale data processing. This research main concern is to provide insights into the Architectural differences of both DataFrames and see their performance features in a single computer. The finding from this research outlines that Pandas leverages the speed of C and it is powerful to be used in one computer. Therefore, when working with Pandas, it's highly recommended to leverage its vectorization capabilities whenever possible. Avoiding traditional loop-based operations not only leads to more efficient code but also makes your code more idiomatic and easier to understand and maintain. This is particularly important when dealing with large datasets where performance can be a critical factor.

6.1 Future work

Exploring Spark DataFrame's scalability on a cluster of computers would provide valuable insights into its distributed processing capabilities. Comparing this with Dask, another popular library for parallel and distributed computing in Python, would offer a comprehensive understanding of their respective strengths and weaknesses in a distributed environment.

Moreover, measuring resource utilization, particularly CPU metrics, would shed light on the efficiency of these data processing frameworks. CPU usage is a critical factor in the performance of data processing tasks, and understanding how these frameworks utilize CPU resources can lead to optimizations that improve

performance. The dynamic nature of CPU and memory utilization, due to the operating system's resource allocation, presents an interesting research challenge. Developing techniques to accurately measure these variables would not only contribute to the understanding of these frameworks' performance but also advance the field of performance measurement in general.

These investigations would indeed enrich the knowledge base of data processing frameworks and provide practitioners with valuable insights to optimize their data analysis workflows. As data continues to grow in volume and complexity, research in this area becomes increasingly important for the development of efficient and scalable data processing solutions.

7 References

- [1] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. and Stoica, I. (2010) 'Spark: Cluster Computing with Working Sets', in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association.
- [2] McKinney, W. (2010) 'Data Structures for Statistical Computing in Python', in *Proceedings of the 9th Python in Science Conference*, pp. 51-56.
- [3] Allen Institute for AI (2023) CORD-19 Research Challenge [Dataset]. Available at: <https://www.kaggle.com/datasets/allen-institute-for-ai/CORD-19-research-challenge> (Accessed: 3 June 2023).
- [4] VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, Inc.
- [5] Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., ... & Ghodsi, A. (2016) 'Apache Spark: A Unified Engine for Big Data Processing', *Communications of the ACM*, 59(11), pp. 56-65
- [6] McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc.
- [7] Pandas (2021) *Series*. Available at: <https://pandas.pydata.org/pandas-docs/stable/reference/series.html> (Accessed: 23 March 2023)
- [8] Pandas (2021) *DataFrame*. Available at: <https://pandas.pydata.org/pandas-docs/stable/reference/frame.html> (Accessed: 25 March 2023).
- [9] Pandas (2021) *Index*. Available at: <https://pandas.pydata.org/pandas-docs/stable/reference/index.html> (Accessed: 27 March 2023).
- [10] Abadi, D., Madden, S. and Hachem, N. (2008) 'Column-stores vs. Row-stores: how different are they really?', in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 967-980.
- [11] Intel (n.d.) *The Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length Extensions Feature on Intel® Xeon® Scalable Processors*, Intel. Available at: <https://www.intel.com/content/www/us/en/developer/articles/technical/the-intel-advanced-vector-extensions-512-feature-on-intel-xeon-scalable.html> (Accessed: 10 May 2023)
- [12] Pandas (2021) *Scaling to Large Datasets*. Available at: https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html (Accessed: 6 April 2023).
- [13] Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). *The NumPy array: a structure for efficient numerical computation*. *Computing in Science & Engineering*, 13(2), 22-30.

- [14] Pandas. (2021). *How we can speed up 50x using vectorized operations*. Available at: <https://plainenglish.io/blog/pandas-how-you-can-speed-up-50x-using-vectorized-operations#vectorized-array> (Accessed: 13 April 2023).
- [15] Augspurger, T. (2016) *Effective Pandas*. Available at: <https://github.com/TomAugspurger/effective-pandas> (Accessed: 15 April 2023).
- [16] Pandas Development Team (n.d.) *Group By: split-apply-combine*. Available at: https://pandas.pydata.org/docs/user_guide/groupby.html (Accessed: 16 April 2023).
- [17] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012) 'Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing', in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association.
- [18] Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R.S., Lian, C., Huai, Y., ... & Meng, X. (2015) 'Spark SQL: Relational Data Processing in Spark', in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM.
- [19] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., ... & Zaharia, M. (2015) 'Scaling Spark in the Real World: Performance and Usability', *Proceedings of the VLDB Endowment*, 8(12), pp. 1840-1843.
- [20] Matei, Z., et al. (2010) 'Spark: Cluster Computing with Working Sets', in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association.
- [21] McKinney, W. (2011) 'Pandas: A Foundational Python Library for Data Analysis and Statistics', *Python for High Performance and Scientific Computing (PyHPC 2011)*.
- [22] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M. (2015) *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, Inc.
- [23] Pandas Development Team. (2021). *Pandas: Powerful Python Data Analysis Toolkit*. Available at: <https://pandas.pydata.org/docs/> (Accessed: 14 April 2023).
- [24] Apache Software Foundation, 2021. Apache Spark™ - A unified analytics engine for large-scale data processing. Available at: <https://spark.apache.org/docs/latest/> (Accessed: 14 April 2023).
- [25] Molloy, D., 2011. *The art of data analysis: how to answer almost any question using basic statistics*. John Wiley & Sons.
- [26] Waskom, M., 2014. *Seaborn: statistical data visualization*. Journal of Open Source Software, 6(60), p.3021.
- [27] Bovet, D.P., and Cesati, M., 2005. *Understanding the Linux Kernel*. O'Reilly Media, Inc.
- [28] Rodola, G., 2021. Psutil - Cross-platform lib for Process and System Monitoring in Python. Available at: <https://github.com/giampaolo/psutil> (Accessed: 16 April 2023).

- [29] Herlihy, M., and Shavit, N., 2012. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers.
- [30] Mertz, D., 2019. *Effective Python: 90 Specific Ways to Write Better Python*. Addison-Wesley Professional.
- [31] Provost, F., and Fawcett, T., 2013. *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*. O'Reilly Media, Inc.
- [32] Gold, R., 2016. *Measurement and Analysis of CPU Usage on a Personal Computer*. Journal of Modern Processes in Manufacturing and Production, 5(3).
- [33] Jones, M., 2014. *Memory Management in Python: The Basics*. In: Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns. Addison-Wesley.
- [34] Gosling, J. et al., 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional.
- [35] Smith, J., 2020. *Linux Performance*. Pragmatic Bookshelf.
- [36] Wikipedia contributors. (2023). Flynn's taxonomy - Wikipedia. [online] Available at: https://en.wikipedia.org/wiki/Flynn%27s_taxonomy [Accessed 4 Jun. 2023].
- [37] Chhugani, J., Macy, W., Baransi, A., Nguyen, A.D., Hagog, M., Kumar, S., Lee, V.W., Chen, Y-K., Dubey, P. (Year of Publication). Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. ACM Digital Library. Available at: <https://dl.acm.org/doi/pdf/10.1145/564691.564709> (Accessed 5 Jun. 2023).
- [38] McKinney, W. (2012). pandas: powerful Python data analysis toolkit (Release 0.7.3) (Accessed 3 Jun. 2023).
- [39] Xin, R., & Rosen, J. (2015, April 28). Project Tungsten: Bringing Apache Spark Closer to Bare Metal. Retrieved from <https://www.databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html> (Accessed 6 Jun. 2023)
- [40] Spiegelberg, L., Yesantharao, R., Schwarzkopf, M. and Kraska, T., 2021. Tuplex: Data Science in Python at Native Code Speed. In: SIGMOD '21: Proceedings of the 2021 International Conference on Management of Data, June 20–25, Virtual Event, China. Brown University and MIT CSAIL. Available at: <https://dl.acm.org/doi/pdf/10.1145/3448016.3457244>. (Accessed June 9, 2023)
- [41] Apache Spark. (2017). PySpark 2.2.0 Documentation. Available at: <https://spark.apache.org/docs/2.2.0/api/python/pyspark.html> (Accessed: 11 June 2023).
- [42] Beazley, D., 2010. *Understanding the Python GIL*. PyCON Python Conference.
- [43] Lawlor, O., 2013. *Loop-Level Parallelism in the Intel® Xeon Phi™ Coprocessor*. The Inner Loop.
- [44] Goddard, W., 2017. *UNIX and Linux System Administration Handbook*, 5th Edition. Pearson Education.