Experiments & Analyses Paper

Evaluation of Dataframe Libraries for Data Preparation on a Single Machine

Angelo Mozzillo
University of Modena and Reggio Emilia, Italy
angelo.mozzillo@unimore.it

Luca Zecchini
University of Modena and Reggio Emilia, Italy
luca.zecchini@unimore.it

Luca Gagliardelli
University of Modena and Reggio Emilia, Italy
luca.gagliardelli@unimore.it

Adeel Aslam
University of Modena and Reggio Emilia, Italy
adeel.aslam@unimore.it

Sonia Bergamaschi
University of Modena and Reggio Emilia, Italy
sonia.bergamaschi@unimore.it

Giovanni Simonini
University of Modena and Reggio Emilia, Italy
giovanni.simonini@unimore.it

ABSTRACT

Data preparation is a trial-and-error process that typically involves countless iterations over the data to define the best pipeline of operators for a given task. With tabular data, practitioners often perform that burdensome activity on local machines by writing ad hoc scripts with libraries based on the Pandas dataframe API and testing them on samples of the entire dataset—the faster the library, the less idle time its users have.

In this paper, we evaluate the most popular Python dataframe libraries in general data preparation use cases to assess how they perform on a single machine. To do so, we employ 4 real-world datasets with heterogeneous features, covering a variety of scenarios, and the TPC-H benchmark. The insights gained with this experimentation are useful to data scientists who need to choose which of the dataframe libraries best suits their data preparation task at hand.

In a nutshell, we found that: for small datasets, Pandas consistently proves to be the best choice with the richest API; when data fits in RAM and there is no need for complete compatibility with Pandas API, Polars is the go-to choice thanks to its in-memory execution and query optimizations; when a GPU is available, CuDF often yields the best performance, while for very large datasets that cannot fit in the GPU memory and RAM, PySpark (thanks to a multithread execution and a query optimizer) proves to be the best option.

1 INTRODUCTION

Companies and organizations significantly depend on their data to drive informed decisions, such as business strategy definition, supply chain management, etc. Thus, guaranteeing high data quality is fundamental to ensure the reliability of analysis and avoid undesired additional costs [30]. Data is often heterogeneous, i.e., data sources adopt different formats and conventions (e.g., different encodings, different ways to represent dates or numeric values, etc.), and are affected by quality issues, such as missing or duplicate values [23]. Due to that, data scientists dedicate considerable time and resources to perform *data preparation* [24]. Typically, this fundamental process involves multiple operations called *preparators* [29], combined into a pipeline [25], aimed at exploring [41], cleaning [32], and transforming raw data into curated datasets [26, 74, 81].

It is said that data scientists spend up to 80% of their time on data preparation [31]. This also depends on the lack of support in the choice among many different available libraries and in the design of the pipeline that best suits both the dataset and the task at hand. The growing need for standardization and best practices has driven an increasing number of data practitioners to embrace the *dataframe* as the foundational data structure for constructing pipelines and developing libraries. At a high level, the dataframe is a two-dimensional data structure that consists of rows and columns [57]. Libraries for data preparation typically offer a flexible functional interface that allows to conveniently compose pipelines.

Pandas [43] is by far the most widely adopted library for manipulating dataframes, and is considered by many as the *de facto* standard for all preparators [33, 40, 57, 69]. This library has become so popular in the data science ecosystem that even the popularity of Python itself has been sometimes attributed to its wide adoption [14].

Despite its success, Pandas notoriously presents multiple severe limits [22, 39]. Firstly, it has not been designed to work with large datasets efficiently. In fact, Pandas operates in a single-threaded manner, lacks support for cluster deployment, and does not implement any memory optimization mechanism (the entire data is kept in main memory until the end of the pipeline execution). To overcome these limitations, several alternative libraries have been developed and are gaining popularity among data scientists. However, these libraries come with heterogeneous features (e.g., lazy evaluation, GPU support, etc.), making it hard for data scientists to navigate among them and choose the most suitable solution.

In particular, data scientists lack the support of extensive and rigorous studies to evaluate the performance of such libraries in the data preparation scenario. Public wisdom about dataframe libraries is mostly scattered across several not peer-reviewed sources doing mutual comparisons [1, 60, 76] (whose major claims are mostly confirmed by our evaluation), while related examples in scientific literature [58, 73, 77] either focus on orthogonal aspects or only cover small subsets of libraries and preparators.

Our contribution. It is common for data scientists to perform countless iterations on samples of the datasets on their laptop or PC to define a proper data preparation pipeline for the task at hand. Hence, the faster the employed library is, the less idle time they have. To support data scientists in the choice of the library that can speedup their workflow, we present the first extensive experimental comparison of Pandas and its most popular

dataframe-based alternatives, namely PySpark, Modin, Polars, CuDF, Vaex, and DataTable. For the sake of reproducibility we developed Bento, a framework that provides a general interface to design and deploy the data preparation pipeline with any library. Bento[1] allows the definition of a common Pandas-like API for all libraries and enables the execution in a containerized environment using Docker. This process is streamlined with the help of a convenient configuration file[2].

To cover a variety of scenarios and to provide results that are representative of real-world use cases, we consider four datasets with heterogeneous size and features, previously adopted in related work on data preparation [29] and dataframe libraries [58]. All these datasets are collected from Kaggle [38], the most popular platform for data science challenges. This allows us to exploit published data preparation pipelines that have proven to work well for downstream machine learning tasks and have been validated by the world's largest data science community. Moreover, we further validate our findings by testing the different libraries with the TPC-H benchmark [67].

We identify four essential data preparation stages: input/output (I/O), exploratory data analysis (EDA), data transformation (DT), and data cleaning (DC). Thus, we evaluate the performance of the libraries both on a single preparator and on sequences of preparators (i.e., the entire pipeline or its subsets corresponding to the four different stages), highlighting the impact of the query optimization techniques supported by some of them.

For whom is this paper relevant. Since it represents by far the most common situation for data scientists [22], we only focus on the single-machine scenario, aiming to explore the distributed scenario in a future extension[3]. Our findings are useful to those data scientists that: *(i)* are employing Pandas in their workflow when devising data preparation pipelines on their laptop or PC; *(ii)* want to replace Pandas with a more efficient library that has the same interface. Thus, we evaluate scalability by measuring the performance of the dataframe libraries on three distinct machine configurations (laptop, workstation, and server), while increasing the size of the dataset. Hence, we provide the readers with several insights about their performance based on multiple factors, such as the size and the features of the dataset, the configuration of the underlying machine, and the type of preparators applied.

Outline. We describe in detail the dataframe data structure and the libraries to be compared in Section 2, then we present our evaluation setup and the results in Section 3 and 4, respectively. After reviewing the related literature in Section 5, we report the key takeaways in Section 6 and draw the conclusions in Section 7.

2 DATAFRAMES

A *dataframe* [57] is a data structure that organizes the data into a two-dimensional table-like format, where columns represent the schema and rows represent the content. Each column has a specific data type, and the schema does not need to be declared in advance. A dataframe provides a wide set of operators acting on all data dimensions, enabling and simplifying various data preparation tasks such as data analysis, transformation, and cleaning. Beyond this variety of features, dataframes became popular thanks to their simple and flexible structure, which allows them to effectively handle different types of data. In fact, compared to specific *data preparation tools* [29], dataframes offer greater simplicity and customization, enabling users to manipulate the data by writing Python code without the effort of learning to use a new tool.

Pandas [53] has been the first widely-adopted Python library implementing dataframes (inspired by the corresponding data structure from the R language [43]) and provides efficient and expressive data structures optimized to work with structured datasets. Pandas has established itself over time as the standard library for data manipulation and analysis. The main reasons behind its widespread popularity reside in its robust functionality and user-friendly design, making it the go-to choice for handling datasets in Python. Given its enormous diffusion, Pandas is a very mature library and the existence of a huge amount of documentation and courses makes its learning extremely straightforward.

Nevertheless, Pandas also presents several notorious limitations [44], which become especially evident when dealing with large-scale datasets, as it lacks the optimizations required to process large amounts of data in an efficient way. For instance, Pandas does not support multiprocessing and parallel computing, and it is designed to work with in-memory datasets, lacking support for datasets that exceed memory limits. Further, it needs to materialize the intermediate result of every operation (this execution strategy is known as *eager evaluation*), exposing it to out-of-memory risks and preventing it from applying query optimization techniques.

To address such limitations, many solutions have emerged to seamlessly replace Pandas in Python workflows, aiming for more efficient data processing. The selection of the Python libraries to include in our evaluation was based on three main criteria, requiring them to: *(i)* be based on the dataframe data structure (excluding therefore SQL-based solutions such as DuckDB [70]); *(ii)* be compatible with the Pandas API, to enable a fair and straightforward evaluation of each operation (note that many of these libraries, recognizing Pandas as the *de facto* standard, already aim at Pandas-compatibility for their API [45] or describe the parallelism between their API and the Pandas API [18]); *(iii)* have gained more than 1k stars on GitHub, denoting the existence of a significant user base and community support. The selected libraries are: Pandas, PySpark, Modin, Polars, CuDF, Vaex, and DataTable.

Dataframe Libraries

We identify five major features exploited by Pandas alternatives to achieve high efficiency:

- *Multithreading*: using multithreading to speed up the execution of dataframe operations, making the most of the available hardware resources.
- *GPU acceleration*: leveraging the parallel computing potential of graphics processing units to further improve the performance of dataframe operations.
- *Resource optimization*: implementing strategies to efficiently manage memory, reducing the impact on the available resources, hence improving the overall performance.
- *Lazy evaluation*: maintaining a logical plan of the operations, triggering their execution only when a specific output operation is invoked, to apply query optimization techniques.

[1] https://github.com/dbmodena/bento
[2] https://github.com/dbmodena/bento/tree/master#write-a-test-file
[3] Note that we decided to include libraries conceived for distributed execution, such as PySpark or Modin, since through multithreading and optimized execution plans they can improve the performance of dataframe operations even on a single machine.

Table 1: Features of the compared dataframe libraries.

	Pandas	PySpark	Modin	Polars	CuDF	Vaex	DataTable
Multithreading		✓	✓	✓		✓	✓
GPU acceleration					✓		
Resource optimization		✓	✓	✓	✓	✓	✓
Lazy evaluation		✓		✓			
Deploy on cluster		✓	✓				
Native language	Python	Scala	Python	Rust	C/C++	C/Python	C++/Python
Licence	3-Clause BSD	Apache 2.0	Apache 2.0	MIT	Apache 2.0	MIT	Mozilla Public 2.0
Other requirements		SparkContext	Ray/Dask		CUDA		
Considered version	2.2.1	3.5.1	0.29.0	0.20.23	24.04.01	4.17.0	1.1.0

- *Deploy on cluster*: enabling the distribution of dataframe processing across a cluster of machines, leveraging parallel computing (as stated in Section 1, we aim to consider this dimension in future work).

For each dataframe library, Table 1 lists which of these features it implements, along with information about its implementation and the version considered here.

PySpark [6] is the Python API for Apache Spark, which is primarily implemented in Scala.

PySpark DataFrame is a type of Dataset (i.e., a distributed collection of data) organized into named columns, resembling a relational table and supporting relational operators [9]. PySpark also supports lazy evaluation, relying on the Catalyst optimizer [11] and a disk spillover mechanism that automatically offloads data from RAM to disk when memory limits are reached [10]. This allows PySpark to process datasets that exceed the machine's physical memory capacity. While Spark owes its popularity to the capability of processing large-scale datasets on a cluster of machines, *standalone* (i.e., single-machine) mode is not only supported, but also surprisingly fast in multiple scenarios, as shown in Section 4.

PySpark supports two different APIs: *(i)* Spark SQL, which allows combining SQL queries with Spark programs to work with structured data; *(ii)* Pandas on Spark (denoted as SparkPD and previously known as Koalas [17]), which enables (by adding an index to the conventional Spark dataframe) to distribute Pandas workloads across multiple nodes without requiring modifications to the original Pandas code for most API functions (~80%).

Modin [46] is a Python library that provides a parallel alternative to Pandas [57, 58]. Modin adopts the Pandas data format as the default storage layer and employs a set of 15 *core operators* to simplify Pandas functions and build its own Pandas-like API. When core operators cannot handle a function, it switches to the *default to Pandas* mode (affecting performance due to communication costs and Pandas single-threaded nature), reverting to a partitioned Modin dataframe after completion. It also implements *opportunistic evaluation* [88], enabling execution based on interactions, and facilitates incremental query construction through intermediate dataframe results. Modin is designed to dynamically switch between different partition schemes (row-based, column-based, or block-based) depending on the operation. Each partition is then processed independently by the execution engine: Dask or Ray (we consider both solutions, denoted as ModinD and ModinR, respectively).

Dask [16] is a Python library for distributed computing, which allows working with large distributed Pandas dataframes [75]. It is designed to efficiently extend memory capacity using disk space, making it suitable for systems with limited memory, and its scheduler provides flexibility to the execution. While we also considered the inclusion of Dask as an independent library, we do not report its results since we found it to perform better in combination with Modin (as Dask is not well suitable for a single machine) and it only covers ~55% of the Pandas API (while Modin covers ~90%) [47].

Ray [72] is a general-purpose framework for parallelizing Python code, using an in-memory distributed storage system and Apache Arrow [3] (a language-independent columnar memory format, recently adopted even by Pandas [52]) as data format [49]. Unlike Dask, Ray does not have built-in primitives for partitioned data. While the two engines serve different use cases, their primary goal is similar: optimizing resource usage by changing how data is stored and Python code is executed.

Polars [64] is a Python library written in Rust and built on top of arrow2, the Rust implementation of the Arrow format. The adoption of Arrow as the underlying data structure provides Polars with efficient data processing capabilities, optimized through parallel execution, cache-efficient algorithms, and efficient usage of resources.

Polars does not use an index, but each row is indexed by its integer position in the table. Further, it has developed its own Domain Specific Language (DSL) for transforming data, whose core components are Expressions and Contexts. Expressions facilitate concise and efficient data transformations, while Contexts categorize evaluations into three main types: filtering, grouping/aggregation, and selection. Polars optimizes queries through early filters and projection pushdown. Moreover, it supports both eager and lazy evaluation. The lazy strategy allows to run queries in a streaming manner: instead of processing the entire data at once, they can run in batches, lightening the load on memory and CPU, hence allowing to process bigger datasets (even larger than memory) in a faster manner.

CuDF [50] is a component of the NVIDIA RAPIDS framework. Written in C/C++ and CUDA (hence only compatible with NVIDIA GPUs), it offers a Pandas-like API to run general-purpose data science pipelines on GPUs, leveraging their computational power for accelerating data processing.

CuDF is built on top of Arrow and leverages parallelization to execute operations on different parts of columns simultaneously across all available GPU cores. Note that CuDF uses a single GPU, while Dask-CuDF can be used for multi-GPU parallel computing. Thus, it can perform efficient and high-performance computations, although it does not provide any optimization strategy for the execution of the pipelines.

Table 2: Features of the selected datasets.

	Athlete	Loan	Patrol	Taxi
CSV Size (GB)	0.03	1.6	6.7	10.9
# Rows ($\times 10^6$)	0.2	2	27	77
# Columns	15	151	34	18
# Num - Str - Bool	5-10-0	113-38-0	5-27-2	15-3-0
% Null	9%	31%	22%	0%
Str Len Range	(1, 108)	(1, 3988)	(1, 2293)	(1, 19)

Vaex [85] is a Python library[4] designed to handle extremely large tabular datasets (such as astronomical catalogs), producing fast statistical analysis and visualization. Supported by its vaex-core extension, written in C, Vaex achieves memory optimization through streaming algorithms, memory-mapped files, and a zero-copy policy, supporting the exploration of datasets that exceed the available memory.

Vaex enables efficient column-wise operations by wrapping a series of NumPy arrays as columns. Only a small subset of NumPy functions benefits of lazy evaluation, storing their results as computation instructions and computing them only when needed. This is achieved through virtual columns, expressions that can be added to datasets without incurring additional memory usage. Vaex can process random data subsets and export data in random order for efficient resource utilization. Moreover, it also supports multithreading for faster computation on multi-core CPUs.

DataTable [28] is a Python package that relies on the Frame object, similar to Pandas dataframes or SQL tables (i.e., data arranged in a two-dimensional array with rows and columns).

DataTable, which uses a native-C implementation for all data types, is developed to support column-oriented data storage, optimizing data access and processing. DataTable enables memory-mapping of data on disk, allowing the seamless processing of out-of-memory datasets. DataTable incorporates multithreaded data processing, leveraging all available cores for time-consuming operations, and offers efficient algorithms for sorting, grouping, and joining operations. To minimize unnecessary data copying, it adopts the copy-on-write technique for shared data, reducing memory overhead and improving the overall performance.

3 BENTO

In this section, we provide all details about the configuration of Bento, our framework for evaluating different dataframe libraries on data preparation pipelines.

Datasets. For a comprehensive evaluation of the dataframe libraries on general, real-world data preparation use cases, we selected four datasets covering a variety of sizes, complexities, and features: *(i)* `Athlete` [35], collecting 0.2M records about the results achieved by the athletes through 120 years of Olympics; *(ii)* `Loan` [36], containing 2M rows about loan applicants and their financial profiles from the LendingClub company; *(iii)* `Patrol` [37], composed of 27M records about 11 years of traffic stops by California law enforcement agencies [59]; *(iv)* `Taxi` [34], containing 77M records about taxi trips in New York City in 2015. A summary of their main features is displayed in Table 2. Note that we compute the percentage of missing values as the number of null cells over all cells in the dataframe. Collected from Kaggle, all of these datasets had previously been adopted in related work about data preparation and dataframe libraries. The three largest datasets were indeed used to evaluate the performance of Modin [58], due to the variety of their features. Instead, `Athlete` was exploited to analyze how different preparators are supported by commercial data preparation tools [29].

Data Preparation Pipelines. For each dataset, we selected three notebooks among the top-voted Kaggle entries that proposed a non-trivial solution for the task at hand (i.e., Kaggle competitions) and extracted its data preparation pipeline—typically the first part of the notebook, which precedes the definition of a machine learning model.

Overall, the pipelines employ a set of 27 preparators (reported in Table 3) that can be clustered into four main stages:

- *Input/output* (I/O): preparators to handle the input/output of data in various formats, such as databases, CSV or Parquet files, Web APIs, etc.
- *Exploratory data analysis* (EDA): preparators to support the exploratory analysis of data features, to better understand the data at hand and detect errors or anomalies.
- *Data transformation* (DT): preparators for transforming the data to make it more suitable for analysis, such as categorical encoding, join, aggregation, etc.
- *Data cleaning* (DC): preparators to improve the quality of the data, hence the reliability of the results of its analysis (e.g., handling missing or erroneous data and correcting or removing outliers).

Pipeline Specification. Bento is a Python framework that provides a general shared interface to design the data preparation pipeline, ignoring the differences in the APIs of the considered libraries. For preparator names, we adopted the convention proposed by Hameed and Naumann [29]. Each pipeline can be defined through a JSON file by specifying the sequence of preparators to employ. Our tool automatically deploys the pipeline for every specific library.

As shown in Table 3, not all preparators are available in every dataframe library. In such cases, we implemented them with our best effort to avoid the transition to Pandas and back. When a preparator is directly available in the API of the library, we denote whether its interface fully aligns with Pandas (✓✓) or differs in the adopted name (✓).

Unless differently stated, we load data from CSV files and measure the execution time under three key settings: *(i) function-core*, which evaluates each preparator alone (if the library adopts lazy evaluation, it requires to force the execution after each API call); *(ii) pipeline-stage*, which measures the runtime for the execution of each of the four stages (i.e., I/O, EDA, DT, and DC); *(iii) pipeline-full*, which measures the runtime for the execution of the entire pipeline. Finally, our framework also supports the execution of the pipeline in Docker containers, which allow to specify core and memory usage and isolate Python environments to prevent package conflicts.

Hardware and Software. All measurements are obtained as the average value over ten runs on a machine equipped with two AMD EPYC Rome 7402 CPUs (48 threads running at 2.8 GHz) and 512 GB of RAM, using Python 3.9, when not specified otherwise. The machine also features a NVIDIA A100 GPU with 40 GB of RAM and CUDA 12.1. The Ray engine was configured with default settings, resulting in 48 worker threads running. The

[4]Since the last commit on GitHub is dated about one year ago, the project might not be actively maintained at the moment.

Table 3: Compatibility of dataframe libraries with Pandas API. (✓✓) fully matches Pandas interface; (✓) different interface; (○) missing from the API, but implemented by us to the best of our efforts.

	Preparator	SparkPD	SparkSQL	Modin	Polars	CuDF	Vaex	DataTable
I/O	load dataframe (*read*)	✓✓	✓	✓✓	✓✓	✓✓	✓	✓
	output dataframe (*write*)	✓✓	✓	✓✓	✓	✓✓	✓	✓✓
EDA	locate missing values (*isna*)	✓✓	○	✓✓	✓	✓✓	○	✓
	locate outliers (*outlier*)	✓✓	✓	✓✓	✓✓	✓✓	✓	○
	search by pattern (*srchptn*)	✓✓	✓	✓✓	✓✓	✓✓	✓✓	✓✓
	sort values (*sort*)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
	get columns list (*getcols*)	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓
	get columns types (*dtypes*)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓
	get dataframe statistics (*stats*)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	○
	query columns (*query*)	✓✓	✓	✓✓	✓✓	✓✓	✓	○
DT	cast columns types (*cast*)	✓✓	✓	✓✓	✓	✓✓	✓✓	○
	delete columns (*drop*)	✓✓	✓✓	✓✓	✓✓	✓✓	○	○
	rename columns (*rename*)	✓✓	○	✓✓	✓✓	✓✓	✓✓	○
	pivot table (*pivot*)	✓✓	✓	✓✓	✓	✓✓	○	○
	calculate column using expressions (*calccol*)	✓✓	○	✓✓	✓✓	○	✓✓	○
	join dataframes (*join*)	✓✓	○	✓✓	✓	✓✓	○	○
	one hot encoding (*onehot*)	✓✓	○	✓✓	✓✓	✓✓	✓	○
	categorical encoding (*catenc*)	✓✓	✓	✓✓	✓	✓✓	✓	○
	group dataframe (*group*)	✓✓	✓	✓✓	✓✓	✓✓	✓✓	✓✓
DC	change date & time format (*chdate*)	✓✓	✓	✓✓	○	✓✓	○	○
	delete empty and invalid rows (*dropna*)	✓✓	✓	✓✓	✓	✓✓	✓✓	○
	set content case (*setcase*)	✓✓	✓	✓✓	✓	✓✓	✓✓	✓✓
	normalize numeric values (*norm*)	✓✓	✓	✓✓	✓✓	✓✓	✓✓	○
	deduplicate rows (*dedup*)	✓✓	✓	✓✓	✓	✓✓	○	○
	fill empty cells (*fillna*)	✓✓	✓	✓✓	○	✓✓	✓✓	○
	replace values occurrences (*replace*)	✓✓	✓	✓✓	○	✓✓	✓	○
	edit & replace cell data (*edit*)	✓✓	○	✓✓	✓	✓✓	✓✓	✓✓

Dask engine was configured comparably, leading to the creation of 8 workers and 48 threads for each execution.

To ensure accuracy and avoid warm-up overhead, the assessment of execution time occurs once the system (e.g., JVM) has completed its warm-up process. To assess the performance of the libraries across machines with different hardware specifications, we simulated three distinct machine configurations, as detailed in Table 4. Finally, we evaluated scalability on incremental samples of `Taxi` and `Patrol`.

4 EVALUATION RESULTS

In this section, we report and analyze the results of the extensive experimental evaluation of the presented dataframe libraries. Our main goal is to provide data scientists and practitiones with useful insights for supporting them in the selection of the best solution for their data preparation tasks. Therefore, our comparison is designed to assess the performance of dataframe libraries based on the operations to carry out (considering both the distinct preparators and the benefits introduced by lazy evaluation, when supported), the size and the features of the dataset at hand, and the configuration of the machine on which the pipeline is executed.

In particular, we organize our evaluation in multiple subsections, each designed to answer one of the following research questions:

Q1. What is the performance of the dataframe libraries in running data preparation pipelines on datasets of different size and features? (Section 4.1)

Q2. How does lazy evaluation impact on the performance of the libraries that support it? (Section 4.2)

Q3. How do libraries scale by varying the size of the dataset and the configuration of the underlying machine (i.e., from laptop to server)? (Section 4.3)

Q4. How do libraries perform on the standard queries of the TPC-H benchmark? (Section 4.4)

4.1 Evaluation on Data Preparation Pipelines

Summary—For EDA, Polars is generally the best performer. For DT, if a GPU is available, CuDF generally outperforms other libraries. For DC, Vaex achieves notable results on the largest datasets. Finally, CuDF and Polars appear to be the best choices to read and write CSV files, respectively.

For each of the selected datasets (`Athlete`, `Loan`, `Patrol`, `Taxi`), Figure 1 and Figure 2 show the average speedup over `Pandas` achieved by its alternatives in the execution of the three data preparation pipelines per dataset, focusing on the stages of EDA, DT, and DC. In particular, Figure 1 considers each stage in its entirety, allowing to perform lazy evaluation at the stage level when supported, Figure 2 shows the performance separately for each distinct preparator (i.e., we force the execution for each of them). The I/O stage is considered separately in Figures 3 and 4. The bars in Figure 1 and the markers in Figure 2 denote the speedup achieved over Pandas (the red line), defined as follows:

$$speedup = \frac{Time\langle Pandas, prep/stage\rangle}{Time\langle lib, prep/stage\rangle}$$

where $Time\langle lib, prep/stage\rangle$ is the time required by the library *lib* to run the considered preparator/stage. Thus, a value above (below) the red line denotes that the library outperforms (fall

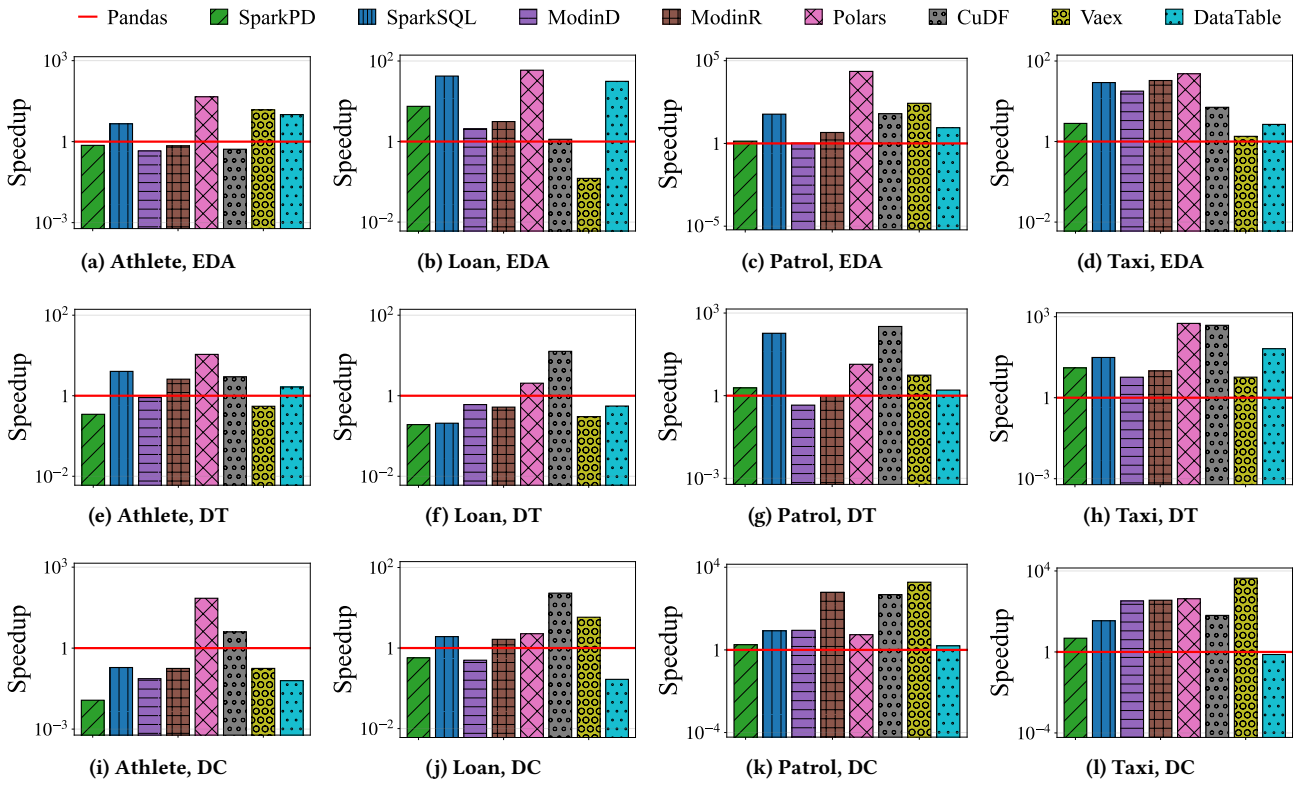


Figure 1: Average speedup over Pandas computed for each stage (EDA, DT, DC) on the three data preparation pipelines.

behind) Pandas. Further, Figure 2 shows for each preparator the number of calls in each of the three pipelines and the impact on its stage, depicted by the bars in the background and defined as a percentage as follows:

$$Impact = \frac{Time\langle dataset, prep\rangle}{Time\langle dataset, stage\rangle} \times 100$$

where $Time\langle dataset, prep\rangle$ is the average time for each stage preparator and $Time\langle dataset, stage\rangle$ is the sum of the average times for all preparators of the stage on the three dataset pipelines[5].

In the next subsections, we analyze in detail the results obtained for each of the main data preparation stages, considering both the single preparators and the entire stages.

Explorative Data Analysis (EDA). For EDA, Polars clearly stands out as the best performer on all datasets (Figures 1a-1d). As depicted in Figure 2, Polars always registers the best performance for *isna* and *outlier*, the preparators with the greatest impact on `Athlete` and `Loan`, respectively. For the former preparator, Polars is up to 10,000 times faster than Pandas and 3 times faster than the second-best performer, i.e., DataTable. Both libraries avoid element-wise comparisons by using a special encoding to efficiently track null values: while Polars relies on the validity bitmap used by most types of Arrow arrays in their metadata [63], DataTable encodes them with sentinel values [28]. For the latter preparator, outliers are located by filtering the dataframe using the values returned by the *quantile* function. Pandas relies on NumPy exact quantile computation (involving sorting and interpolation) [55], while Polars and Spark employ approximate methods that avoid sorting [7]. The performance gap between SparkSQL and SparkPD arises from differences in their underlying implementations.

Although Vaex tends to be very efficient in column-wise operations (e.g, *srchptn*) and can handle filtering well by internally tracking selected rows without copying data, operations based on percentiles are penalized by the complexity of the required calculations (i.e., determining min/max column values, cumulative sums, and grid interpolation) [84].

For *sort*, CuDF and Polars are the fastest libraries overall. While CuDF leverages on high-performance C++ parallel algorithms from the Thrust library [51], Polars achieves high performance through its efficient multi-threaded Rust implementation [65]. SparkSQL shows remarkable advantages over Pandas as the datasets grow larger, while SparkPD performs similarly to the baseline. Modin performs significantly worse than Pandas on `Athlete` as it requires to apply Pandas implementation of the sorting preparator within each data partition [48], introducing significant latency to partition and merge results. For *stats*, DataTable outperforms Pandas by up to 50 times by computing statistics either during the creation of the Frame object or efficiently on-demand when needed, while Modin (with both Ray and Dask) is up to 200 times faster than the baseline on `Taxi` thanks to its multi-threading capabilities.

Vaex achieves notable performance for *query*, for which it excels on the `Taxi` dataset and is only outperformed by Polars, which enables fast querying by leveraging mask operations [64], on the smaller ones. Finally, Pandas generally maintains overall

[5]To minimize bias from particularly fast or slow libraries, both in the numerator and in the denominator time values below the 20th percentile and above the 80th percentile were excluded.

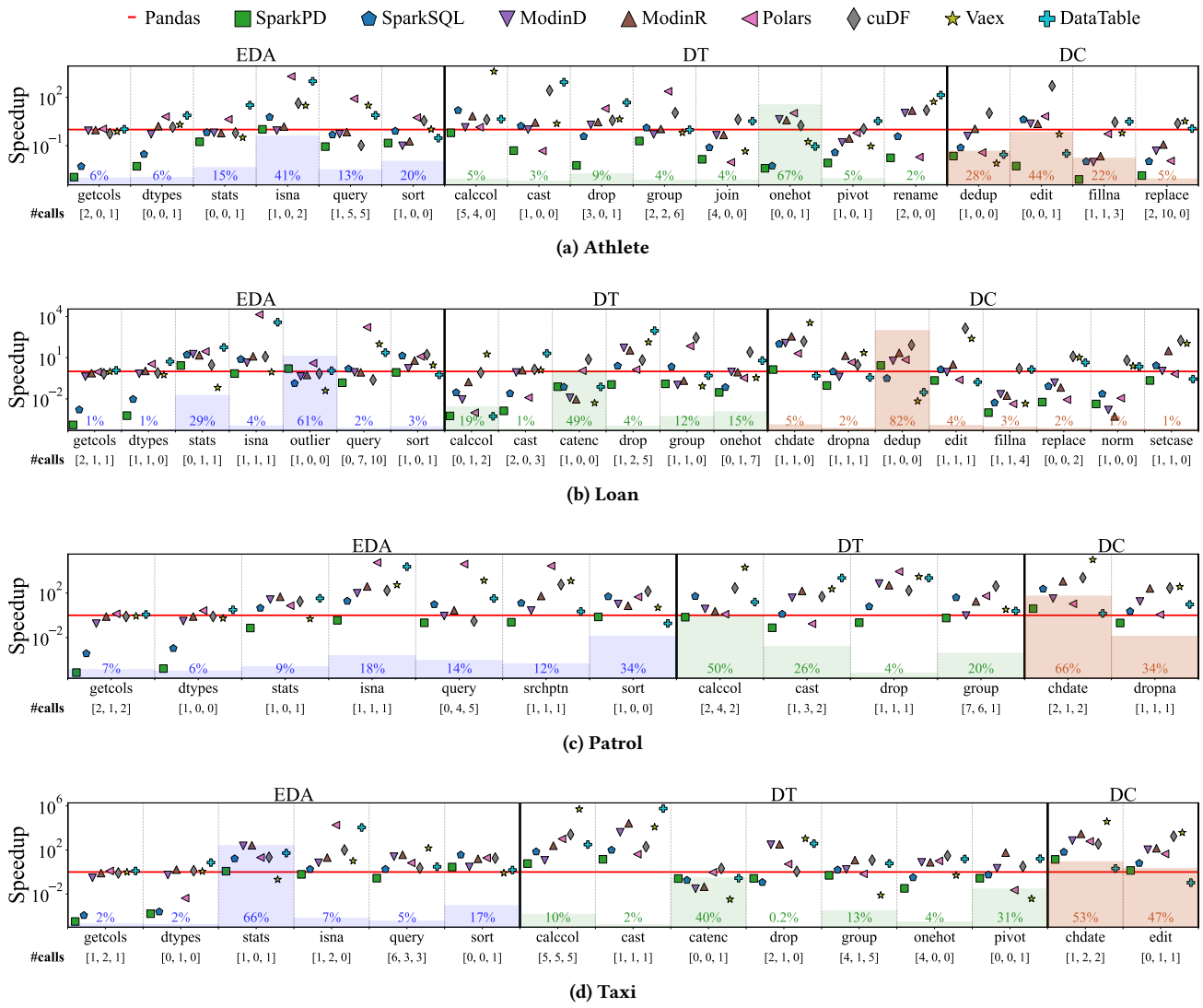


Figure 2: Average speedup over Pandas computed for each preparator of the three data preparation pipelines per datasets. For each preparator, we report the number of calls in each pipeline and its average impact on the stage execution time.

good performance for *getcols* and *dtypes*, where PySpark is much slower due to the overhead associated to its inherently distributed architecture, optimized for complex tasks rather than simple ones.

Data Transformation (DT). In this stage, CuDF and Polars emerge as the best performers overall, highlighting the benefits of GPU acceleration and Rust code optimizations, respectively. In particular, CuDF outperforms Polars on `Loan` (Figure 1f) and `Patrol` (Figure 1g), while the opposite is true on `Athlete` (Figure 1e). As depicted in Figure 2, CuDF always achieves the best results for *catenc*, while for *onehot* and *group* it is the best performer on all datasets but `Athlete`, where Polars outperforms the alternatives. SparkSQL shows excellent results on `Patrol` (Figure 1g): despite not excelling for single preparators (Figure 2c), lazy evaluation provides remarkable benefits. Moreover, it clearly outperforms SparkPD, which results as the worst performer on several preparators, mainly due to the latency introduced by translating Pandas operations into Spark execution plans.

Vaex optimizations lead to outstanding performance on column-wise preparators (e.g., *calccol* and *drop*), but also to drastic drops moving to operations such as *group*, *join*, or *pivot*. Modin, designed for resource-intensive applications, improves its performance as datasets scale up, with ModinR being the best performer for *pivot* on `Taxi`. DataTable registers remarkable performance on that preparator too, despite not supporting it natively [18], but particularly excels for *cast*, due to direct memory manipulation and in-place casting [19]. On the other hand, libraries relying on the Arrow columnar format (e.g., Polars), with its safety check and abstraction layers, may experience additional overhead [61].

Data Cleaning (DC). When it comes to DC, Polars achieves the best results on `Athlete` (Figure 1i), outperforming CuDF (which performs better for all preparators if considered separately, as depicted in Figure 2a) thanks to lazy evaluation. CuDF excels on `Loan` (Figure 1j) thanks to its performance with *dedup*, which has by far the highest runtime among the preparators, thus deserving an in-depth analysis in the following. CuDF implements *dedup* natively, by using factorization to identify duplicates [54]. Differently, Polars tracks and manages the presence or absence of unique values using bitmasking [64], while SparkSQL uses

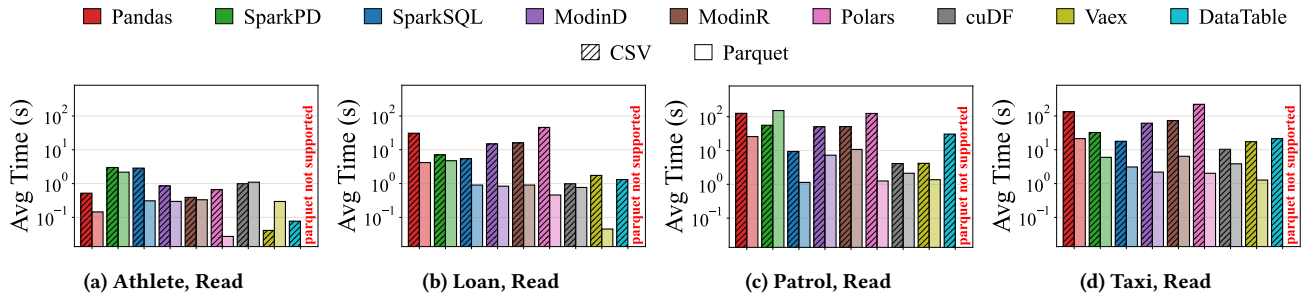


Figure 3: Average runtime for reading CSV and Parquet files (lower is better).

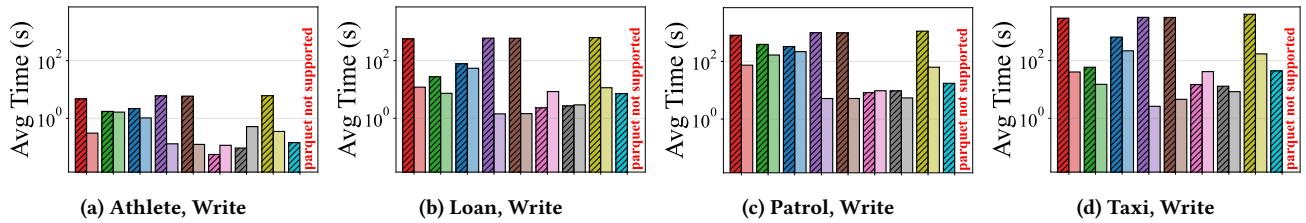


Figure 4: Average runtime for writing CSV and Parquet files (lower is better).

grouping and aggregations to find duplicate rows [8]. Finally, Vaex and DataTable lack an official implementation for *dedup*.

Results change when moving to larger datasets (Figures 1k and 1l), with Vaex achieving the best results both on `Patrol` (followed closely by CuDF and ModinR) and on `Taxi` (with remarkable performance by Polars and both Modin versions). On `Patrol`, Vaex is the best performer on both preparators (Figure 2c): *dropna*, where it exploits its efficient filtering mechanisms, and *chdate*, for which it relies on NumPy operations (faster compared to more complex operations found in other libraries) to enhance conversion and manipulation of columns [85].

Input/Output (I/O). Loading data represents a fundamental step for the success of any data-driven process. In particular, the CSV format is considered as the *de facto* standard for reading and writing raw data, and Pandas is recognized as an efficient and effective tool for its ingestion [86]. Nowadays, the diffusion of Arrow is pushing towards an increasing adoption of the Parquet format [5], known for its efficient data compression. In particular, developers often load Parquet data into memory and convert it to Arrow [42], recognized as an ideal in-memory transport layer for such data [4].

Figures 3 and 4 show performance in reading and writing both file formats, CSV and Parquet. When reading small datasets like `Athlete`, Parquet and CSV often exhibit similar performance, with exceptions such as Polars or SparkSQL (Figure 3a). In some cases (e.g., Vaex or CuDF), the use of Parquet can even lead to worse performance, due to the need for decompressing the data. When the size increases, the use of Parquet generally produces better results; for instance, reading Parquet files with Polars is over 100 times faster than reading CSV files (Figure 3d).

Overall, CuDF and Vaex appear to be the best performers in reading CSV files. Vaex is particularly fast thanks to chunked reading, conversion to optimized formats (e.g., HDF5), and efficient use of memory [83]. CuDF, relying on GPU acceleration and efficient memory management, maintains similar performance for both file formats. DataTable, which maps the file into memory and navigates through it using pointers, performs good in reading CSV files, but it does not support Parquet. Polars and Vaex excel in reading Parquet files thanks to their capability to load them directly into memory using Arrow.

Write operations exhibit a wide range of performance. Overall, Polars and CuDF are the top performers and Parquet performs better than CSV across all datasets as it integrates efficient compression and encoding approaches, while CSV is plain text and uncompressed [42]. Nevertheless, the overhead introduced by compression and encoding operations can penalize Parquet in some scenarios, as highlighted by CuDF on small datasets (e.g., `Athlete` and `Loan`). Further, Polars presents a reported issue in writing Parquet files, which determines slower performance and larger file size [64]. Finally, Modin (with both engines) demonstrates the best performance on Parquet, particularly for large datasets, by leveraging parallel processing across multiple cores [45].

4.2 Impact of Lazy Evaluation

> *Summary*—CuDF generally outperforms other libraries when running the entire pipeline, while SparkSQL and ModinR achieve remarkable results on all datasets. Lazy evaluation often brings relevant benefits, improving performance by 20% on average (up to almost 80%) over its eager counterpart.

Figure 5 reports the average speedup over Pandas achieved by its alternatives for the execution of the entire data preparation pipelines on each dataset. For libraries supporting lazy evaluation (i.e., SparkPD, SparkSQL, and Polars), we also point out the difference compared to eager evaluation.

Considering the overall performance on the entire pipeline, CuDF generally outperforms other libraries by effectively exploiting the power of GPU, with the only exception of `Athlete` (Figure 5a), where Polars is the best performer. PySpark (except

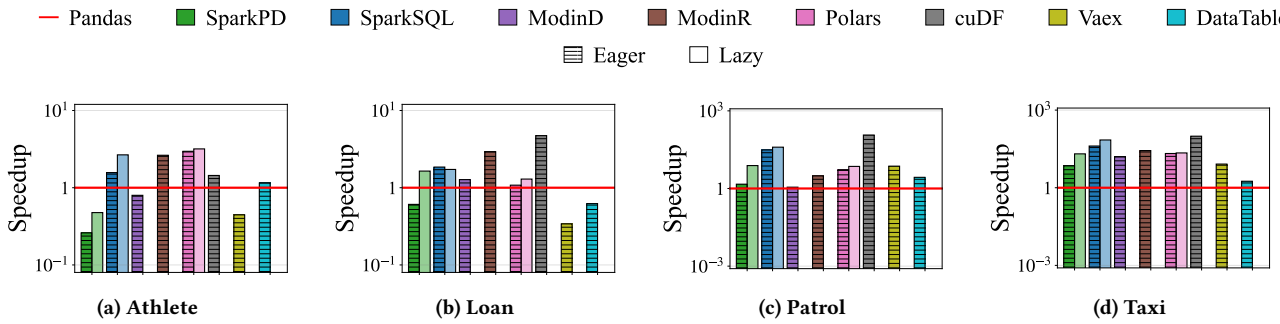


Figure 5: Average speedup over Pandas for the execution of the entire data preparation pipelines.

Table 4: Specifications of each machine configuration.

	Laptop	Workstation	Server
# CPUs	8	16	24
# RAM (GB)	16	64	128
Dask (workers - threads)	4-8	4-16	6-24
Ray (workers)	8	16	24

for SparkPD on `Athlete`), ModinR, and Polars show very solid performance overall, as well as Vaex on the largest datasets. Pandas performs relatively well on the smallest datasets, but it is outperformed by all alternatives on larger ones, due to its eager approach causing high memory consumption [79].

Polars generally achieves good results with both kinds of evaluations, as its eager API is highly optimized and in many cases it internally relies on its lazy counterpart before immediately collecting the result. Lazy evaluation leverages techniques such as streaming processing, early filtering, and projection pushdown, achieving a performance improvement of up to 25% on `Patrol` (Figure 5c). On one hand, it can be noticed how long logical plans can limit the optimizations introduced by lazy evaluation. Yet, the execution plan overhead of SparkSQL does not yield substantial improvements for either smaller or larger datasets, with an improvement of 40% on `Taxi` and almost no changes in performance on `Loan`. Libraries like SparkPD, leverage to Pandas-specific optimizations (e.g., vectorized operations and efficient memory management) that translate for instance into a 80% performance improvement on `Patrol` (Figure 5c).

4.3 Scalability

Summary—SparkSQL is the best library for scaling to large datasets on a single machine. Indeed, for both `Patrol` and `Taxi`, it is the only one able to execute the entire pipelines with the laptop configuration. Polars follows closely, but requires a lot of resources in terms of RAM, exhibiting the worst scalability on machines with limited resources.

In this section, we assess how dataframe libraries scale by varying the size of the dataset and the configuration of the underlying machine. In particular, we take into account three different machine configurations, whose CPU and RAM are incrementally increased as described in Table 4 (which also reports the specifications of the Dask and Ray engines used by Modin) to reproduce typical configurations for laptops, workstations, and servers.

For this experiment, we selected the most computationally expensive pipeline among the three evaluated, which is the first one[6]. Since CuDF scales according to GPU memory, it is not included in this experiment. If the dataset fits in the GPU, CuDF is able to efficiently handle it and complete the pipeline. However, this dependency does not make it a good candidate for dealing with very large datasets on machines with limited GPU resources.

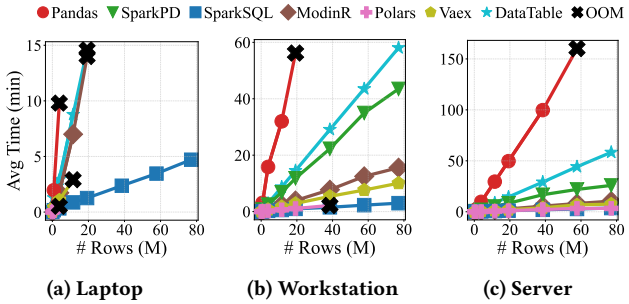
Figure 6 shows the performance of the libraries for running the entire pipeline on incremental samples of `Taxi`. SparkSQL clearly stands out as the best performer, being the only library capable of handling the full pipeline execution on laptop configuration (Figure 6a). This capability is attributed to its combination of lazy evaluation and disk spillover mechanisms [10]—optimizing execution plans and offloading data to disk when memory limits are reached during computations. SparkPD suffers instead from the overhead due to internal conversions between Pandas and Spark dataframes and increased JVM memory usage, causing out-of-memory (OOM) errors on the laptop configuration.

While Modin (both Dask and Ray) supports operations on datasets larger than available memory [45], the efficacy of this feature is constrained by two primary factors. Firstly, it requires additional memory for managing internal operations, metadata, and intermediate results. Secondly, certain operations (e.g., apply lambda function), necessitate in-memory data access. DataTable shares this limitation as well, as a results these two libraries encounter OOM issues when processing datasets larger than 25% of the full dataset. Similarly, Vaex experiences OOM errors with datasets exceeding 15% of the full dataset size. This is due to the fact that in Vaex the output of some operations (e.g., groupby) is held entirely in memory, potentially exceeding available RAM [85].

Despite Polars excellent performance in other experiments, scalability appears its weakness—it reaches OOM with 4 million and 40 million rows in Figures 6a and 6b, respectively. This limitation arises from its in-memory execution model, which requires all data to be loaded into RAM. While this approach offers speed and efficiency for smaller datasets, it becomes a bottleneck when handling larger datasets, limiting its scalability. Finally, Pandas confirms its well-known severe limitations about scalability, being the only library not able to complete the pipeline on `Taxi` even on a server configuration (Figure 6c).

To provide a comprehensive overview of how libraries scale across datasets with different characteristics, Table 5 outlines instead the minimum configuration required by each library to successfully run the full pipeline on progressively larger samples of `Patrol` and `Taxi`.

[6]On average, this pipeline is approximately three times more computationally expensive than the others.



(a) Laptop (b) Workstation (c) Server

Figure 6: Average runtime for running the entire pipeline on incremental samples of Taxi.

SparkSQL is the only library able to run the pipelines with the laptop configuration on the entire datasets—thanks to lazy evaluation and disk spillover. The second best in scalability is DataTable, which excels on `Patrol` requiring minimal resources through efficient memory mapping on disk.

ModinR demonstrates better scalability across all machine configurations compared to its Dask counterpart. The Dask engine uses a centralized scheduler to distribute data across multiple cores, which results in higher memory consumption and leads to OOM issues more easily [15]. In contrast, Ray can scale far beyond Dask due to its distributed task scheduling scheme, specifically employing a distributed bottom-up scheduling approach [71]. This optimizes resource utilization and minimizes overhead by initiating tasks at the lowest level of computation and aggregating results upwards. Notably, Modin was originally built to work on top of Ray, making this integration more mature and optimized [45].

4.4 Performance on the TPC-H Queries

Summary—CuDF consistently achieves the best performance across all queries, while Polars significantly outperforms other CPU-only dataframe libraries.

TPC-H [82] is a decision support benchmark consisting of a suite of business-oriented ad-hoc queries and concurrent data modifications [67]. It is widely adopted to compare end-to-end database systems [21], allowing to comprehensively evaluate their performance and efficiency through the execution of complex queries on large volumes of data. Consistently with previous related work [66] exploiting TPC-H queries to assess the performance of a subset of dataframe libraries (see Section 5 for more details), we also adopt this benchmark to provide further support to the outcomes of the comparative analysis presented above.

In particular, we take as a reference a publicly available translation of the TPC-H queries into Pandas API operations [13], replicating it for the other libraries, then use Bento for their seamless execution. Figure 7 illustrates the performance of each library on the 22 queries of the TPC-H 10GB benchmark. We select a scale factor of 10, because it represents the largest dataset that can be processed by the 40 GB RAM of our GPU, which therefore would not be able to handle a scale factor of 100. Moreover, we included DuckDB [70], being a popular in-process SQL analytical database management system. DuckDB can execute parallel queries directly on Pandas DataFrames, Parquet/CSV files, or Arrow tables without a separate import step, and can write results back to these formats. This interoperability allows seamless

Table 5: Minimum machine configuration for running the entire pipeline on incremental dataset samples.

	Patrol							Taxi						
% Sample	**1%**	**5%**	**15%**	**25%**	**50%**	**75%**	**100%**	**1%**	**5%**	**15%**	**25%**	**50%**	**75%**	**100%**
# Rows (M)	**0.2**	**1.2**	**4.5**	**6.7**	**13.5**	**20.2**	**27**	**0.7**	**3.8**	**11.5**	**19.2**	**38.8**	**57.7**	**77**
Pandas														×
SparkPD														
SparkSQL														
ModinD													×	×
ModinR														
Polars														
Vaex														
DataTable														

Laptop Workstation Server ×OOM

integration with other data science libraries. We executed the TPC-H queries on DuckDB using the provided implementation in the Polars benchmark [66]. Notice that DuckDB does not provide Pandas API and only supports SQL. Thus, by expressing the pipelines as SQL queries, it can take advantage of all the well-known query optimization and execution strategies typical of relational database management systems. For this reason, we do not compare it with the other libraries throughout the paper, but only report its performance with TPC-H to provide a valuable reference point w.r.t. OLAP database management systems.

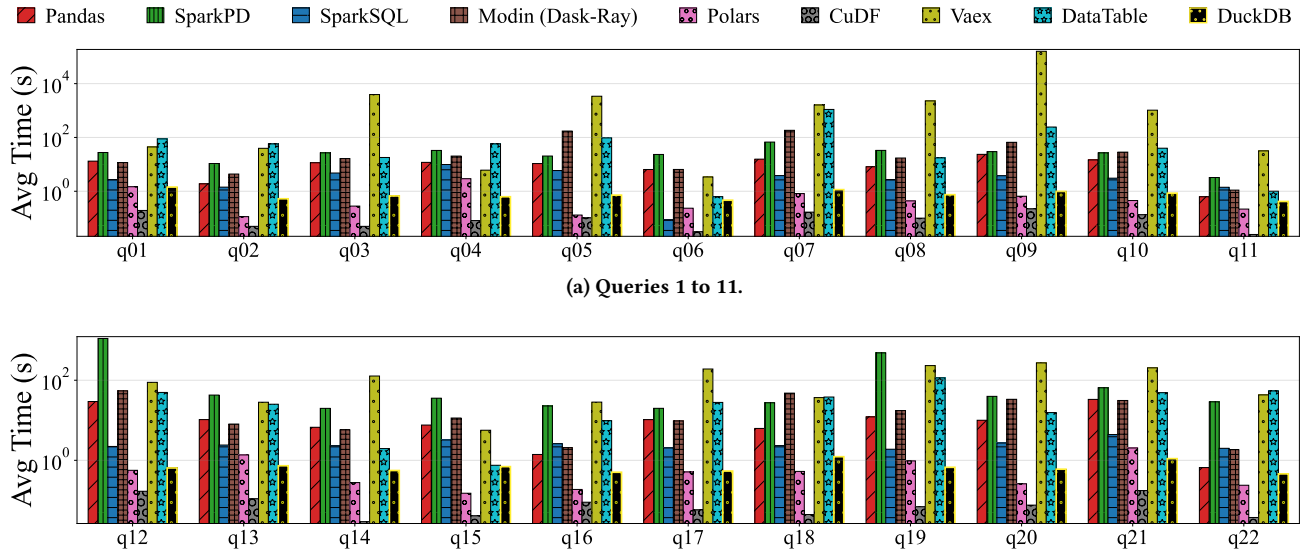
Overall, CuDF consistently achieves the best performance by leveraging GPU power. Among the CPU-only libraries, Polars is instead the clear winner. This confirms what is observed in the data preparation pipelines, for EDA and DT. SparkSQL outperforms Pandas on several queries, registering notable results, especially on q06, where it is even faster than Polars (note that this query selects line items shipped within a year interval, reducing the dataset by 85%). By leveraging lazy evaluation, SparkSQL significantly reduces the amount of data processed early in the query execution—this is one of the key reasons why it improves performance in data preparation pipelines by approximately 60%. Pandas is never the worst performer, but on most queries, it is among the slowest. Modin (which registers the same results with both engines, Dask and Ray) shows diverse performance instead, strongly dependent on the query, confirming the inconsistent performance observed across the different data preparation stages.

As mentioned above, SparkPD suffers from the additional latency due to the translation of Pandas operations into Spark execution plans, making it one of the worst performers. DataTable is one of the worst performers too, since it is slow in grouping operations (as also testified by the execution of *group* preparator in DT stage and their own benchmark [27]) and its API only supports joins on columns with unique values, requiring therefore to switch to the default Pandas API (e.g., q09). Vaex is by far the worst performer overall, as it is significantly slow in grouping operations (as also pointed out in the previous sections) and lacks support for multi-column joins.

After CuDF, the second best performers are DuckDB and Polars. They excel in different aspects of the benchmark, with neither consistently outperforming the other across all queries—they adopt two different approaches for query optimizations.

5 RELATED WORK

To the best of our knowledge, our paper presents the first rigorous and extensive experimental comparison of existing dataframe libraries on data preparation tasks. In fact, public wisdom about such libraries is mostly scattered across several not peer-reviewed sources [1, 60, 76], which generally compare the performance of few libraries (e.g., Pandas vs PySpark vs Polars) on a handful of



(a) Queries 1 to 11.

(b) Queries 12 to 22.

Figure 7: Performance of the dataframe libraries on the TPC-H 10GB queries (lower is better).

operations. The major claims by such analysis are mostly confirmed by our evaluation; nevertheless, despite often providing some useful insights, they only offer a partial and very fragmented knowledge, far from a complete and detailed overview covering all libraries and operations. In the literature, only partial comparisons have been performed among libraries, which we list in the following.

AFrame [78, 79] reports an evaluation against Pandas, Spark, and ModinR through a micro-benchmark using the synthetic Wisconsin benchmark dataset [20] to assess acceleration and scalability across distributed environments. Petersohn et al. [58] evaluate the benefits of Modin by comparing against Dask, Koalas, and Pandas. Grizzly [39] evaluates against Modin and Pandas, while translating dataframe pipelines to SQL. Shanbhag et al. [77] compare Pandas, Vaex and Dask on a set of single operations to gain insights into their energy consumption.

Dataframe libraries have been compared also in data science benchmarks. For instance, Polars [66] compares against Pandas, PySpark, Dask, Modin, and DuckDB on TPC-H [67]. The outcome of their evaluation is consistent with our findings (we adopt the same scale factor for TPC-H), but their evaluation only covers the first 7 queries out of 22. Similarly, DataTable [27] uses a set of queries to asses the scalability of popular database and dataframe-like systems only limited to group by and join operations. Sanzu [87] proposes a benchmark designed to evaluate five popular data science frameworks and systems (R, Anaconda [2], Dask, PostgreSQL [68], and PySpark) on data processing and analysis tasks. It comprises both a micro-benchmark for testing basic operations in isolation and a macro-benchmark for evaluating series of operations representing concrete data science scenarios on real-world datasets. Similarly, FuzzyData [73] is a workflow generation system that allows to compare dataframe-based APIs on workflows composed of a small subset of operations.

Other works have explored the problem of optimizing/rewriting dataframe pipelines, reporting comparative evaluations of popular libraries. For instance, PyFroid [22] translates from the Pandas API to the DuckDB API to efficiently scale Pandas workloads on a commodity workstation, using top-voted Kaggle notebooks for the evaluation. Dataprep [56] is a task-centric EDA tool that can use different dataframe libraries as back-end engines, including Modin, PySpark, and Dask. Dias [12] is a system for dynamically rewriting Pandas code to optimize performance. All of these studies offer valuable insights into dataframe library performance across some real-world scenarios, but they do not provide a comprehensive evaluation on data preparation tasks with all dataframe libraries considered here.

6 KEY TAKEAWAYS

Our evaluation confirmed the well-known limitations of Pandas. However, despite the abundance of dataframe libraries designed to overtake it, there is no silver bullet to perform the preparation of tabular data with dataframes on a single machine.

The relative performance of the libraries varies significantly among the variegate datasets and stages of the data preparation pipelines that we considered. None of the tools emerges as the clear winner for all considered scenarios. Nevertheless, certain characteristics of the dataset and the pipeline can help narrow the user's choice of the tool. Below, we list three questions to assist in this selection.

Does the dataset fit in the GPU memory? If a GPU with enough memory to fit the data is available, CuDF appears to consistently yield the best overall performance and has quasi-complete compatibility with the Pandas API[7]. However, CuDF lacks of an optimizer; thus, libraries like Polars or SparkSQL might be faster when running the entire pipeline, avoiding unnecessary materializations [62]

Individual stage or complete pipeline? If a user is only interested in an individual stage, i.e., input/output (I/O), explorative data analysis (EDA), data transformation (DT), or data cleaning (DC), CuDF and Polars stand out as the fastest libraries for reading and writing operation respectively. Polars emerges as the

[7]This assumption is based on a single-machine scenario. It is important to note that while GPUs can provide substantial performance advantages, comparing GPUs and CPUs requires careful consideration of factors like costs (e.g., pay-per-use fees in a cloud environment) and the growing availability of high-core-count CPUs.

best performer for EDA, while SparkSQL represents a consistent solution for DT. Moreover, for DC, Vaex records the best performance among all CPU-only libraries when running the entire pipeline on larger datasets.

What is the size of the datasets? For **small datasets** (less than 500k rows), all of the libraries show similar average runtimes. Even if Polars is often recommended for such a scenario, and indeed registers the best overall performance if memory resources are not limited, Pandas might still be the most reasonable choice, as their performance does not differ substantially—at least, not enough to replace entire parts of the pipeline [60].

In the case of **datasets of medium size** (from 2 to 19 million rows) primarily composed of numeric columns (e.g., 5% `Taxi` sample), Vaex offers a robust solution if the pipeline involves many column-wise operations and Pandas compatibility is not needed. On the other hand, for datasets with a high proportion of missing values (e.g., `Loan`, where more than 30% of values are missing), ModinR and PySpark (both version) show good performance. Finally, for datasets with many string columns (e.g., 25% `Patrol`), SparkSQL appears to be the best choice.

For **larger datasets** (over 20 million rows), SparkSQL stands out as the best option [80]. Furthermore, when the pipeline requires dealing with heavy-duty column operations, particularly in datasets that have a multitude of string columns and a good amount of null values (e.g., `Patrol`), Vaex seems to be the best move—and there is no need to set up any environment, as required for PySpark.

7 CONCLUSION

We presented a comprehensive experimental comparison of the most used dataframe libraries to support practitioners in the selection of the most suitable solution to carry out their data preparation tasks on a single machine. To guarantee comparable results and increase usability, we developed Bento, a general framework for assessing the performance of dataframe libraries on four major data preparation stages: input/output (I/O), exploratory data analysis (EDA), data transformation (DT), and data cleaning (DC).

We exploited Bento to perform a thorough comparative analysis of the libraries on four heterogeneous datasets (varying in size, complexity, and features) previously adopted in literature and publicly available on Kaggle. In particular, we relied on three of the most voted Kaggle notebooks per dataset to assess the performance of the libraries on real-world data preparation pipelines validated by a large community. Further, we also evaluate their performance on the popular TPC-H benchmark to support our findings, including DuckDB to assess performance in comparison with an SQL-based system.

Our experimental evaluation shows that one size does not fit all when it comes to dataframe libraries for data preparation. However, we have distilled key takeaways that data practitioners can use as starting points for selecting the appropriate library for their task at hand.

As future work, we plan to compare the libraries in a distributed environment and study the possible adoption of machine learning to suggest optimal library combinations based on datasets, tasks, hardware, and previous executions. Further, we plan to explore how the libraries perform in combination with tools that optimize and rewrite entire dataframe-based data preparation pipelines, such as Dataprep [56] and Dias [12].

ACKNOWLEDGEMENTS

We extend our appreciation to the master's students of the 2021-2022 Big Data Management and Governance course at the University of Modena and Reggio Emilia for their contributions in testing the libraries. We warmly thank Leonardo S.p.A., co-funding the PhD program of Angelo Mozzillo. Further, this work was partially supported by the "Enzo Ferrari" Engineering Department (University of Modena and Reggio Emilia) within the projects FARD-2022 and FARD-2023, and by MUR within the project "Discount Quality for Responsible Data Science: Human-in-the-Loop for Quality Data" (code 202248FWFS).

REFERENCES

[1] Jonathan Alexander. 2023. Beyond Pandas: Spark, Dask, Vaex and other big data technologies battling head to head. *Medium* (2023). https://towardsdatascience.com/beyond-pandas-spark-dask-vaex-and-other-big-data-technologies-battling-head-to-head-a453a1f8cc13

[2] Anaconda. 2024. Anaconda Website. https://www.anaconda.com/

[3] Apache Arrow. 2024. Apache Arrow GitHub Repository. https://github.com/apache/arrow

[4] Apache Arrow. 2024. Reading and Writing the Apache Parquet Format. https://arrow.apache.org/docs/python/parquet.html

[5] Apache Parquet. 2024. Apache Parquet Website. https://parquet.apache.org/

[6] Apache Spark. 2024. Apache Spark GitHub Repository. https://github.com/apache/spark

[7] Apache Spark. 2024. pyspark.sql.DataFrame.approxQuantile. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.approxQuantile.html

[8] Apache Spark. 2024. Source Code for pyspark.sql.dataframe. https://spark.apache.org/docs/latest/api/python/_modules/pyspark/sql/dataframe.html

[9] Apache Spark. 2024. Spark SQL, DataFrames and Datasets Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html

[10] Apache Spark. 2024. Storage Level. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.StorageLevel.html

[11] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[12] Stefanos Baziotis, Daniel D. Kang, and Charith Mendis. 2024. Dias: Dynamic Rewriting of Pandas Code. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 1, Article 58 (2024), 27 pages. https://doi.org/10.1145/3639313

[13] Bodo. 2024. Bodo Examples. https://github.com/Bodo-inc/Bodo-examples

[14] Thomas Claburn. 2017. Python explosion blamed on pandas. *The Register* (2017). https://www.theregister.com/2017/09/14/python_explosion_blamed_on_pandas

[15] Dask. 2024. Dask Documentation. https://docs.dask.org/en/stable/

[16] Dask. 2024. Dask GitHub Repository. https://github.com/dask/dask

[17] Databricks. 2024. Koalas GitHub Repository. https://github.com/databricks/koalas

[18] DataTable. 2024. Comparison with Pandas. https://datatable.readthedocs.io/en/latest/manual/comparison_with_pandas.html

[19] DataTable. 2024. datatable.as_type. https://datatable.readthedocs.io/en/latest/api/dt/as_type.html

[20] David J. DeWitt. 1993. The Wisconsin Benchmark: Past, Present, and Future. In *The Benchmark Handbook for Database and Transaction Systems*.

[21] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment (PVLDB)* 13, 8 (2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[22] Venkatesh Emani, Avrilia Floratou, and Carlo Curino. 2024. PyFroid: Scaling Data Analysis on a Commodity Workstation. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 61–67. https://doi.org/10.48786/edbt.2024.06

[23] Wenfei Fan. 2015. Data Quality: From Theory to Practice. *ACM SIGMOD Record* 44, 3 (2015), 7–18. https://doi.org/10.1145/2854006.2854008

[24] Alvaro A. A. Fernandes, Martin Koehler, Nikolaos Konstantinou, Pavel Pankin, Norman W. Paton, and Rizos Sakellariou. 2023. Data Preparation: A Technological Perspective and Review. *SN Computer Science* 4, 4 (2023), 425:1–425:20. https://doi.org/10.1007/S42979-023-01828-8

[25] Tim Furche, Georg Gottlob, Leonid Libkin, Giorgio Orsi, and Norman W. Paton. 2016. Data Wrangling for Big Data: Challenges and Opportunities. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 473–478. https://doi.org/10.5441/002/edbt.2016.44

[26] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *VLDB Journal* 29, 4 (2020), 867–892. https://doi.org/10.1007/S00778-019-00586-5

[27] H2O. 2021. Database-like Ops Benchmark. https://h2oai.github.io/db-benchmark/

[28] H2O. 2024. DataTable GitHub Repository. https://github.com/h2oai/datatable

[29] Mazhar Hameed and Felix Naumann. 2020. Data Preparation: A Survey of Commercial Tools. *ACM SIGMOD Record* 49, 3 (2020), 18–29. https://doi.org/10.1145/3444831.3444835

[30] Anders Haug, Frederik Zachariassen, and Dennis Van Liempd. 2011. The costs of poor data quality. *Journal of Industrial Engineering and Management* 4, 2 (2011), 168–193. https://doi.org/10.3926/jiem.2011.v4n2.p168-193

[31] Joseph M. Hellerstein, Jeffrey Heer, and Sean Kandel. 2018. Self-Service Data Preparation: Research to Practice. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 41, 2 (2018), 23–34. http://sites.computer.org/debull/A18june/p23.pdf

[32] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning.* https://doi.org/10.1145/3310205

[33] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2021/papers/cidr2021_paper08.pdf

[34] Kaggle. 2017. New York City Taxi Trip Duration. https://www.kaggle.com/competitions/nyc-taxi-trip-duration

[35] Kaggle. 2018. 120 Years of Olympic History: Athletes and Results. https://www.kaggle.com/datasets/heesoo37/120-years-of-olympic-history-athletes-and-results

[36] Kaggle. 2019. All Lending Club Loan Data. https://www.kaggle.com/datasets/wordsforthewise/lending-club

[37] Kaggle. 2019. Stanford Open Policing Project. https://www.kaggle.com/datasets/faressayah/stanford-open-policing-project

[38] Kaggle. 2024. Kaggle Website. https://www.kaggle.com/

[39] Steffen Kläbe and Stefan Hagedorn. 2021. Applying Machine Learning Models to Scalable DataFrames with Grizzly. In *Proceedings of the Conference on Database Systems for Business, Technology and Web (BTW)*. 195–214. https://doi.org/10.18420/btw2021-10

[40] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, et al. 2021. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. *Proceedings of the VLDB Endowment (PVLDB)* 15, 3 (2021), 727–738. https://doi.org/10.14778/3494124.3494151

[41] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. 2018. *Data Exploration Using Example-Based Methods.* https://doi.org/10.2200/S00881ED1V01Y201810DTM053

[42] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proceedings of the VLDB Endowment (PVLDB)* 16, 11 (2023), 3044–3056. https://doi.org/10.14778/3611479.3611507

[43] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the Python in Science Conference (SciPy)*. 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

[44] Wes McKinney. 2017. Apache Arrow and the "10 Things I Hate about Pandas". *Archives for Wes McKinney* (2017). https://wesmckinney.com/blog/apache-arrow-pandas-internals/

[45] Modin. 2024. Modin Documentation. https://modin.readthedocs.io/en/latest

[46] Modin. 2024. Modin GitHub Repository. https://github.com/modin-project/modin

[47] Modin. 2024. Modin vs. Dask DataFrame vs. Koalas. https://modin.readthedocs.io/en/stable/getting_started/why_modin/modin_vs_dask_vs_koalas.html

[48] Modin. 2024. System Architecture. https://modin.readthedocs.io/en/stable/development/architecture.html

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 561–577. https://www.usenix.org/conference/osdi18/presentation/nishihara

[50] NVIDIA. 2024. CuDF GitHub Repository. https://github.com/rapidsai/cudf

[51] NVIDIA. 2024. Thrust Documentation. https://nvidia.github.io/cccl/thrust/

[52] Pandas. 2023. What's New in 2.0.0. https://pandas.pydata.org/docs/dev/whatsnew/v2.0.0.html

[53] Pandas. 2024. Pandas GitHub Repository. https://github.com/pandas-dev/pandas

[54] Pandas. 2024. pandas.DataFrame.drop_duplicates. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop_duplicates.html

[55] Pandas. 2024. pandas.DataFrame.quantile. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.quantile.html

[56] Jinglin Peng, Weiyuan Wu, Brandon Lockhart, Song Bian, Jing Nathan Yan, Linghao Xu, Zhixuan Chi, Jeffrey M. Rzeszotarski, and Jiannan Wang. 2021. DataPrep.EDA: Task-Centric Exploratory Data Analysis for Statistical Modeling in Python. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2271–2280. https://doi.org/10.1145/3448016.3457330

[57] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment (PVLDB)* 13, 11 (2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[58] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proceedings of the VLDB Endowment (PVLDB)* 15, 3 (2021), 739–751. https://doi.org/10.14778/3494124.3494152

[59] Emma Pierson, Camelia Simoiu, Jan Overgoor, Sam Corbett-Davies, Daniel Jenson, Amy Shoemaker, Vignesh Ramachandran, Phoebe Barghouty, Cheryl Phillips, Ravi Shroff, and Sharad Goel. 2020. A large-scale analysis of racial disparities in police stops across the United States. *Nature Human Behaviour* 4 (2020), 736–745. https://doi.org/10.1038/s41562-020-0858-1

[60] Ben Pinner. 2023. Data Processing: Pandas vs PySpark vs Polars. *Medium* (2023). https://medium.com/@benpinner1997/data-processing-pandas-vs-pyspark-vs-polars-fc1cdcb28725

[61] Polars. 2024. Casting. https://docs.pola.rs/user-guide/expressions/casting/

[62] Polars. 2024. Comparison with Other Tools. https://docs.pola.rs/user-guide/misc/comparison/

[63] Polars. 2024. Missing Data. https://docs.pola.rs/user-guide/expressions/missing-data/

[64] Polars. 2024. Polars GitHub Repository. https://github.com/pola-rs/polars

[65] Polars. 2024. polars.LazyFrame.sort. https://docs.pola.rs/api/python/stable/reference/lazyframe/api/polars.LazyFrame.sort.html

[66] Polars. 2024. Updated TPC-H Benchmark Results. https://pola.rs/posts/benchmarks/

[67] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *ACM SIGMOD Record* 29, 4 (2000), 64–71. https://doi.org/10.1145/369275.369291

[68] PostgreSQL. 2024. PostgreSQL Website. https://www.postgresql.org/

[69] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, K. Venkatesh Emani, Wentao Wu, et al. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *ACM SIGMOD Record* 51, 2 (2022), 30–37. https://doi.org/10.1145/3552490.3552496

[70] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1981–1984. https://doi.org/10.1145/3299869.3320212

[71] Ray. 2024. Ray Documentation. https://docs.ray.io/en/latest/index.html

[72] Ray. 2024. Ray GitHub Repository. https://github.com/ray-project/ray

[73] Mohammed Suhail Rehman and Aaron J. Elmore. 2022. FuzzyData: A Scalable Workload Generator for Testing Dataframe Workflow Systems. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 17–24. https://doi.org/10.1145/3531348.3532178

[74] El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. 2019. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 1954–1957. https://doi.org/10.14778/3352063.3352108

[75] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the Python in Science Conference (SciPy)*. 126–132. https://doi.org/10.25080/Majora-7b98e3ed-013

[76] Markus Schmitt. 2020. Scaling Pandas: Comparing Dask, Ray, Modin, Vaex, and RAPIDS. *Data Revenue* (2020). https://www.datarevenue.com/en-blog/pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray

[77] Shriram Shanbhag and Sridhar Chimalakonda. 2023. An Exploratory Study on Energy Consumption of Dataframe Processing Libraries. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 284–295. https://doi.org/10.1109/MSR59073.2023.00048

[78] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *Proceedings of the IEEE International Conference on Big Data (BigData)*. 359–371. https://doi.org/10.1109/BigData47090.2019.9006303

[79] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-based Approach to Scaling Dataframes. *Proceedings of the VLDB Endowment (PVLDB)* 14, 11 (2021), 2296–2304. https://doi.org/10.14778/3476249.3476281

[80] Travis Tang. 2023. Polars: Pandas DataFrame but Much Faster. *Medium* (2023). https://towardsdatascience.com/pandas-dataframe-but-much-faster-f475d6be4cd4

[81] Ignacio G. Terrizzano, Peter M. Schwarz, Mary Roth, and John E. Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf

[82] TPC-H. 2024. TPC-H Benchmark Documentation. https://www.tpc.org/tpch/default5.asp

[83] Vaex. 2024. I/O. https://vaex.readthedocs.io/en/latest/guides/io.html

[84] Vaex. 2024. Vaex API. https://vaex.readthedocs.io/en/docs/api.html

[85] Vaex. 2024. Vaex GitHub Repository. https://github.com/vaexio/vaex

[86] Gerardo Vitagliano, Mazhar Hameed, Lan Jiang, Lucas Reisener, Eugene Wu, and Felix Naumann. 2023. Pollock: A Data Loading Benchmark. *Proceedings of the VLDB Endowment (PVLDB)* 16, 8 (2023), 1870–1882. https://doi.org/10.14778/3594512.3594518

[87] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray. 2017. In *Proceedings of the IEEE International Conference on Big Data (BigData)*. 263–272. https://doi.org/10.1109/BigData.2017.8257934

[88] Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 44, 1 (2021), 66–78. http://sites.computer.org/debull/A21mar/p66.pdf