Painless and Efficient Scaling of Pandas Programs: Demo

Bhushan Pal Singh
IIT Bombay & NSIL, Bengaluru
singhbhushan@cse.iitb.ac.in

Priyesh Kumar
IIT Bombay[#]
priyeshkumar9875@gmail.com

Chiranmoy Bhattacharya
IIT Bombay[*]
chiranmoy358@gmail.com

Utkarsh Shetye
IIT Bombay
utkarshetye@gmail.com

Manish Kumar
IIT Bombay
cse.manishkumar@gmail.com

S. Sudarshan
IIT Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

Data science applications written in Pandas are very widely used. But Pandas programs often do not work well on very large datasets, running out of memory and/or running very slowly. A number of frameworks have been developed to provide scalability and support for optimization. However, Pandas programs have to be modified in different ways to run them on different frameworks, which can be tedious and error-prone.

We present LaFP (Lazy Fat Pandas) - an optimization framework that allows programmers to code in plain Pandas, but get the benefit of scalability, as well as the ability to run Pandas programs on any of the supported frameworks, with only minor configuration changes. Further, LaFP also supports multiple optimizations based on a combination of "just-in-time" static analysis of the program and lazy-evaluation-based run-time optimizations. The demonstration showcases the ability to run Pandas programs on chosen backends, as well as the performance benefits.

PVLDB Reference Format:
Bhushan Pal Singh, Priyesh Kumar, Chiranmoy Bhattacharya, Utkarsh Shetye, Manish Kumar, and S. Sudarshan. Painless and Efficient Scaling of Pandas Programs: Demo. PVLDB, 19(12): 4710 - 4713, 2026.
doi:10.14778/3827998.3828103

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lazyfatpandas/public.

1 INTRODUCTION

DataFrame-based libraries and frameworks are the tool of choice for most data science applications, with Pandas being one of the most popular frameworks among these. Pandas, however, has performance bottlenecks as it runs on a single node, and can handle only datasets that fit in memory. Many users test their Pandas application on small datasets, but run into performance and scalability issues on large production datasets.

A number of scalable/parallel frameworks have been developed to address these issues, including Dask [10], Modin [7], Pandas on Spark [9], Magpie [6], Ibis [5], Polars [8], DuckDB [2], P2D [4], Grizzly [3] among others. However, there are multiple challenges faced by users in switching from Pandas to these frameworks. There are API incompatibilities between Pandas and the scalable frameworks, as well as API incompatibilities between the different frameworks. Users also have to deal with differences between the eager evaluation model of Pandas and the lazy evaluation model of the scalable frameworks. Manual rewriting is error-prone and misses opportunities for optimization.

import pandas as pd		import lazyfatpandas.pandas as pd
# other import stmts	⇒	# other import stmts
		pd.analyze()
		pd.BACKEND=pd.Backends.MODIN
# rest of program		# rest of program

Figure 1: Pandas Code changes to use LaFP

In our recent paper [11], we presented the *Lazy Fat Pandas* (LaFP) framework, which addresses the above-mentioned challenges faced by the data science user community. Users can continue to write their programs in Pandas, and get the benefit of scalability and efficient execution on any of a number of backends, including those based on lazy evaluation, with the addition of only a couple of lines of code, as highlighted in Figure 1. LaFP supports over 120 Pandas API functions. LaFP handles details of differences in APIs of the supported backends, as well as issues that arise when converting Pandas code to use lazy-evaluation-based frameworks. Currently we support Dask, Modin, DuckDB using Ibis, Polars (natively and with Ibis), Pandas on Spark, and Pandas itself, as backends.

Users also benefit from multiple optimizations implemented in LaFP, which speed up the execution of the programs up to 19X as described in [11]. LaFP performs these optimizations using Just-in-Time (JIT) static analysis and rewriting of Pandas programs, coupled with a lazy runtime that performs a variety of runtime optimizations, allowing optimizations to combine run-time optimization opportunities with look-ahead capabilities using JIT static analysis. The optimized program is executed using the chosen back-end.

In this paper, we demonstrate our framework, Lazy Fat Pandas. Our demonstration showcases how users can run the Pandas programs on different frameworks with minor code modifications, using LaFP. In contrast, the manual porting of the Pandas program to specific frameworks often requires significant code rewrites. We also showcase the significant performance benefits of using LaFP, including its ability to handle large datasets, across different backends. We also showcase the impact of JIT static-analysis by showing a visual difference between the original and rewritten program. For

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828103

[#] Current Affiliation: Dream11.
[*] Current Affiliation: Fujitsu Research, India.

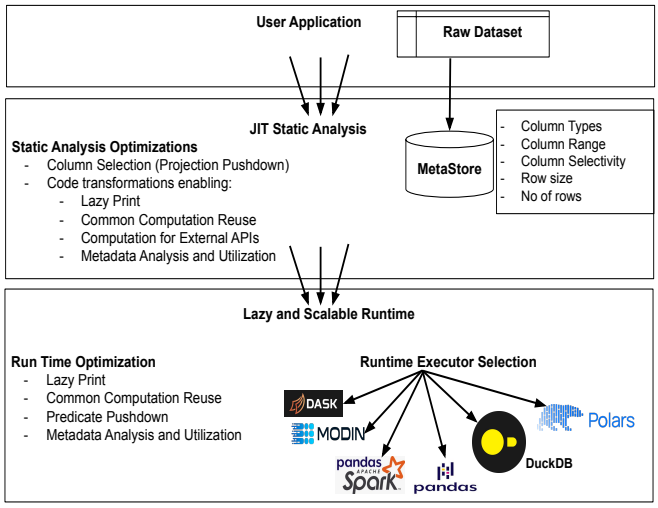

Figure 2: Overview of the Lazy Fat Pandas System

the demo, we also provide a GUI that allows viewers to select any of our benchmark programs and datasets, and compare the impact of individual optimizations. We also display visual task graphs and execution plans both before and after optimizations performed by LaFP for these programs.

2 SYSTEM ARCHITECTURE

Lazy Fat Pandas (LaFP) is composed of two major modules, i.e. JIT static analysis & rewrite module and LaFP's lazy runtime wrapper module. Figure 2 shows the overall flow of optimized execution for LaFP. LaFP offers the unique advantage of combining static-analysis based optimizations with lazy-runtime based optimizations, which is not possible with any of the earlier frameworks. We provide a brief overview of LaFP's architecture here. Further details may be found in [11].

2.1 Just In Time Static Analysis and Rewriting

When control is transferred from the program to LaFP by calling the function `pd.analyze()`; LaFP first optimizes the program using Just-in-Time (JIT) static analysis and rewriting phase. This phase performs source-to-source transformation of Python programs. During this phase, LaFP builds Soot Compatible Intermediate representation i.e SCIRPy IR of the program. The SCIRPy IR was developed by us based on Soot [1]; for more details, refer [11]. LaFP then performs static analysis on this IR, and based on the analysis results, LaFP carries out optimizations by rewriting (transforming) the IR. The optimized IR is then converted back to Python, and the resultant Python program is lazily executed in place of the original program. The optimization phase performs code rewriting to support use of lazy evaluation by forcing computation where required, as well as rewriting to support some run-time optimizations.

2.2 Lazy Run Time

The optimized program is then executed in a lazy fashion. As in other lazy frameworks, instead of executing each Pandas operation eagerly, LaFP creates a task graph: on each invocation of an (LaFP overridden) Pandas API call, a node is created and added to a task graph (DAG). The resultant DAG of operators is executed only when computation is forced, in order to materialize results. When a call to force computation is invoked, the DAG is first optimized by LaFP, then converted to a DAG on the chosen backend, and finally executed on the chosen backend. The chosen backend may perform its own optimizations before executing the program.

LaFP also provides a novel way of lazily executing some non-lazy operations such as `print()` which are not part of Pandas. The use of lazy print allows deferring of computation beyond occurrences of print statements, providing further optimization opportunities.

2.3 Optimizations

Our overall optimization architecture combines optimization based on static analysis with run-time optimization, to have the best of both worlds. Static analysis provides look-ahead to perform optimizations not possible otherwise, such as to predict which dataframes, or parts (columns) of dataframes, are live (i.e., used later) at different points in the program. Rewriting ensures that unused dataframes/dataframe columns are not fetched or computed.

Such optimizations are not possible with the purely run-time optimizers of lazy frameworks such as Dask, Ibis or (Pandas on) Spark, since those optimizers cannot look ahead beyond points where dataframes are materialized. Our optimizations in LaFP run-time are complementary in nature to those already available in lazy frameworks such as Dask or Spark. Optimizations and rewritings implemented in LaFP include projection pushdown (a.k.a. column selection), predicate pushdown (a.k.a. row selection), lazy print, forcing of computation wherever materialized results are needed, common computation reuse, and metadata-based optimizations; see [11] for more details.

2.4 Support for Multiple Back-Ends

LaFP currently supports Pandas, Modin, Dask, Pandas on Spark, DuckDB using Ibis, Polars (natively and using Ibis), and Modin frameworks as executors, and can be extended to support other scalable frameworks. Pandas, Modin, and the eager API of Polars support only eager execution, while the other backends support lazy evaluation.

If an eager back-end is chosen by the user, LaFP still creates a task graph, optimizes it, and then traverses the task graph, and invokes individual operations in sequence to execute the task graph. In case of a lazy back-end, LaFP performs some optimizations, and then converts the task graph to an operator DAG in the back-end framework. Lazy back-ends subsequently perform their own optimizations on the DAG created by LaFP.

If a chosen back-end lacks support for a specific Pandas API functionality, LaFP uses an alternate equivalent API in the back-end, if available. Otherwise, LaFP converts the data from the back-end representation back to that of Pandas and performs the intended operation using the original Pandas API. The LaFP wrapper functions take care of differences between Pandas back-ends, and the user need not worry about these differences, or the limitations of specific backends. The choice of back-end is a configurable parameter, allowing the user to select a desired back-end.

Line#	Sample Program
1	import lazyfatpandas.pandas as pd
2	import matplotlib.pyplot as plt
3	-pd.analyze()
-	
4	pd.BACKEND_ENGINE = pd.BackendEngines.DASK
-	
5	~df = pd.read_csv('s1.csv',parse_dates=['pickup_datetime'])
6	df['day'] = df.tpep_pickup_datetime.dt.dayofweek # add features
7	df = df[df.fare_amount > 0] # filter bad rows
8	df = df[df['passenger_count'] > 2] # get group travels
9	p_per_day = df.groupby(['day'])['passenger_count'].sum() # aggregation
10	df = df[df.fare_amount > 0] # filter bad rows
11	df = df[df['passenger_count'] > 2] # get group travels
12	print (p_per_day)
13	~plt.plot(p_per_day)
14	plt.savefig('fig.png')
15	df = df[df.fare_amount > 5] # filter high fares
16	avg_fare = df.fare_amount.mean()
17	print(f'Average fare: {avg_fare}')
18	print(ans)
-	

Line#	LaFP's Optimized Program
1	import lazyfatpandas.pandas as pd
2	import matplotlib.pyplot as plt
-	
3	+print = pd.lazyPrint
4	pd.BACKEND_ENGINE = pd.BackendEngines.DASK
5	+SO_cols = ["tpep_pickup_datetime","passenger_count","fare_amount"]
6	~df = pd.read_csv('s1.csv',parse_dates=['pickup_datetime'],usecols=SO_cols)
7	df['day'] = df.tpep_pickup_datetime.dt.dayofweek
8	df = df[(df.fare_amount > 0)]
9	df = df[(df['passenger_count'] > 2)]
10	p_per_day = df.groupby(['day'])['passenger_count'].sum()
11	df = df[(df.fare_amount > 0)]
12	df = df[(df['passenger_count'] > 2)]
13	print(p_per_day)
14	~plt.plot(p_per_day.compute(live_df=[df]))
15	plt.savefig('fig.png')
16	df = df[(df.fare_amount > 5)]
17	avg_fare = df.fare_amount.mean()
18	print(f'Average fare: {avg_fare}')
19	print(ans)
20	+pd.flush()

Figure 3: Original Program vs LaFP's Optimized Version of the Same Program

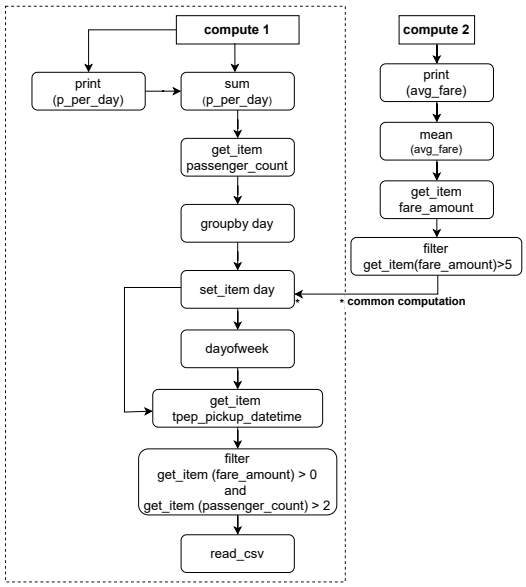

Figure 4: Task graph for program in Figure 3

API Support: LaFP currently supports more than 120 Pandas API functions, including the bulk of the widely used API functionality; and we are adding support for further API calls. Row order is not preserved by most lazy backends, and so such backends should be used only if the program does not depend on row ordering.

3 DEMONSTRATION

We explore several scenarios in our demo as described below. We will use a GUI (described later in this section) to make it easy to explore various aspects of our system.

Scenario 1: *Ease of using LaFP and execution of Pandas programs on large datasets using LaFP+Pandas where native Pandas fails.*

Figure 3 shows a sample program motivated from [11], modified by adding an import of lafyfatpandas and an invocation of the pd.analyze() method. We will use this example in our demo to show how rewriting works, but also allow viewers to explore other programs from a set of programs we have collected, or can be downloaded online. Figure 3 also shows the optimized program, generated by LaFP using JIT static analysis of the sample program. The side-by-side display of the original and optimized programs is automatically generated by our system by passing an extra argument `diff_mode='html'` to `pd.analyze()`.

One of the optimizations performed by our system is Live Attribute Analysis (see [11] for details). This analysis determines which attributes are live at each point in the program, i.e., whose value at the program point is used later in the program. The live attributes in our example program at the point where `read_csv()` is called are `pickup_datetime`, `passenger_count`, and `fare_amount`.

Data fetching (using `read_csv()`, `read_parquet()` etc.) is then optimized by fetching only the live attributes. This information is passed to `read_csv()` using the `usecols` parameter (lines 5 and 6).

In line 14, the function `compute(live_df=[df])` is added to force computation, with the parameter specifying live dataframes after the call, based on live dataframe analysis. The live dataframes are materialized when they are computed, and re-used subsequently. More details on these optimizations are available in [11].

All these optimizations are performed on the SCIRPy IR, and then the optimized IR is translated back to Python. The optimized Python code as shown in Figure 3 is executed, with Pandas dataframe API calls replaced by their lazy equivalents in our lazy run-time wrapper. Execution of the lazy API calls is forced at points where materialized results are needed (for example to execute a function that does not handle lazy dataframes); a `compute()` call is added as part of static rewriting at such points.

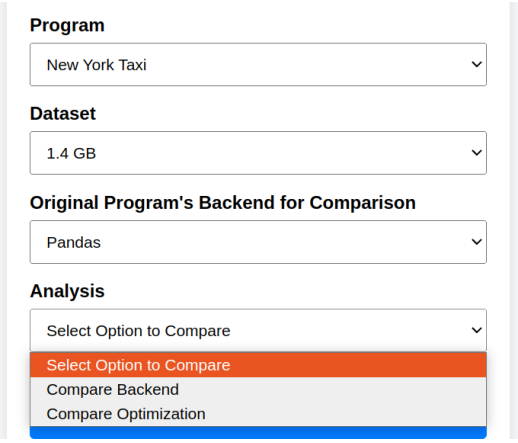

Figure 5: GUI for LaFP Demo

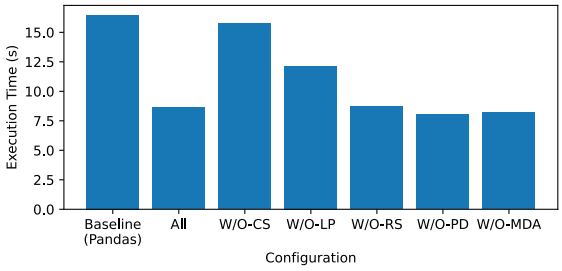

Figure 6: Execution time for Sample Program

For our example program, the lazy calls result in the creation of the task DAG shown in Figure 4, and computation is forced before the `plot()` API call at line 14 (compute 1 in task graph). It may be observed that filter operations (row selection) have been pushed closer to the data source as a run-time optimization of the task DAG. The task graph generation will be part of the demo, although here we show a manually redrawn task graph to reduce space.

It may also be noted that due to our innovative implementation of lazy print, computation is not forced when a print method is invoked and is instead added as a node in the task graph.

In line 20, where `pd.flush()` is called, the nodes in the not-yet computed part of the task graph are executed.

Scenario 2: We provide a GUI as shown in Figure 5 for viewers to interact with LaFP to explore different programs, datasets, and backends, with all/none/selected optimizations enabled. We will provide at least 10 different real-world programs that execute a variety of operations, such as data filter, feature addition, data aggregation, data merge, and group-by, having between 5 and 29 operators, with an average of 13 operators per program; and each program with multiple dataset sizes (450 MB, 1.4 GB, 4.2 GB).

The demo using the GUI will showcase the ease of modifying a Pandas program to enable optimization using LaFP, and its execution on any backend from amongst the supported backends. This will be contrasted with the complexity of manual code rewriting to target different backends. The demo will showcase the speed up due to optimizations.

As an example, the execution times of a sample program along with ablation studies on different optimizations, as plotted through GUI, are shown in Figure 6. The figure shows time taken to execute program natively in Pandas (baseline); using LaFP + Pandas with all optimizations (All); disabling Column Selection (W/O-CS); disabling Lazy Print (W/O-LP); disabling row-selection (W/O-RS); disabling persisting shared dataframes (W/O-PD); and disabling metadata analysis (W/O-MD). As can be seen, using all optimizations reduced execution time by nearly 50%. Disabling Column Selection removes most of the benefits, while Disabling Lazy Print (W/O-LP) also leads to significant loss of benefits, indicating that these are important optimizations for this program. Disabling other optimizations did not have a significant impact for this program.

Further details of performance across multiple programs can be found in [11].

Scenario 3: In this part of the demo, we will provide insight into the optimized execution plan and task graph, which are displayed by using `df.explain()`.

To demonstrate the lazy runtime of LaFP, we showcase task graphs before and after LaFP's optimization along with the execution plan. We have grouped 2 task graphs in Figure 4 and shown only the optimized task graphs to save space for the paper. In the demo, these would be generated and shown separately with initial and optimized task graphs. We also showcase the execution plans for these task graphs.

The demo will raise interesting questions on what class of programs can be handled. While there are clearly Python programs where static analysis cannot be done, in our experience typical Pandas programs do not exhibit such hard-to-handle structures. The main issue we face in general is programs depending on row ordering; while some scalable frameworks, such as Modin, preserve row ordering, others do not. Detecting such cases, and warning the user if they choose back-end frameworks that do not preserve ordering, is an area of ongoing work.

4 CONCLUSION

Our Lazy Fat Pandas (LaFP) optimization framework allows Pandas users to use scalable back-ends with minimal code modifications, and also get benefits of optimizations using just-in-time static-analysis based rewriting, combined with a lazy runtime wrapper. This demonstration showcases the benefits of our approach, allowing viewers to explore alternative backends and optimizations.

REFERENCES

[1] Soot Contributors. 2012. Soot: a Java Optimization Framework. https://www.sable.mcgill.ca/soot/.
[2] DuckDB 2025. DuckDB. https://duckdb.org/. Retrieved 2025-10-05.
[3] Stefan Hagedorn et al. 2021. Putting Pandas in a Box. In *CIDR '21*. Online. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper07.pdf
[4] Yordan Grigorov et al. 2023. P2D: A Transpiler Framework for Optimizing Data Science Pipelines. In *DEEM '23*. ACM. https://doi.org/10.1145/3595360.3595853
[5] Ibis [n.d.]. Ibis: Python data analysis framework for Hadoop and SQL engines. https://github.com/ibis-project/ibis Retrieved 2025-02-27.
[6] Alekh Jindal et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *Conf. on Innovative Data Systems Research (CIDR)*.
[7] Modin 2024. https://github.com/modin-project/modin.
[8] Polars 2025. DataFrames for the new era. https://pola.rs/. Retrieved 2025-10-05.
[9] PySpark 2024. Pandas API on Spark. https://spark.apache.org/docs/latest/api/python/user_guide/pandas_on_spark. Accessed Sep 2024.
[10] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science conference*. Citeseer.
[11] Bhushan Pal Singh, Priyesh Kumar, Chiranmoy Bhattacharya, and S. Sudarshan. 2026. Efficient Dataframe Systems: Lazy Fat Pandas on a Diet. In *EDBT*.