
**Indian Institute of Technology Bombay**

Department of Computer Science and Engineering

Report Submitted in Fulfillment of Credits for

CS691 : R&D Project

---

# Optimizing Access to Pandas Using LaFP/Dias

---

Submitted by :

Pranab Paul (23M0800) & Chaitra Gurjar (23M0831)

Under the Supervision of :

Prof. S. Sudarshan

May 11, 2024

# Acknowledgement

We would like to express our sincere gratitude to Prof. S. Sudarshan for guiding us throughout this project and providing valuable inputs during each discussion and review. We also feel thankful towards Chiranmoy Bhattacharya, Priyesh Kumar and Bhushan Pal Singh for setting up a benchmark for this project. We would especially like to thank Chiranmoy for giving insightful suggestions throughout the project. Finally we would like to acknowledge the grading committee from the department of CSE for giving us the opportunity to present our work.

**Abstract**

Pandas is one of the most popular libraries in Python used for data analysis and data access. However, it imposes the condition of the entire dataset being present in the memory at once for processing. This leads to inefficient access of data calling for the need of optimizations. Additional frameworks like Modin [3] and Dask [1] are used for scalability, but they require the user to work through a different API. Contributions from DBridge in CSE, IITB use compile time optimizations with attribute analysis to create a rewriting tool called ScirPy. The rewritten program is then executed efficiently using lazy evaluation in a tool called LazyFatPandas. Dias is a reqriting tool which works on a pure pandas program, to improve performance in regular Pandas APIs. In this report, we discuss these contributions and explore further opportunities of runtime optimizations. We incorporate the techniques of Dias into LazyFatPandas to enhance time and memory performance. We provide quantitative assessments of these optimizations and conclude the report by hinting at future works.

# Contents


**1 Introduction** **5**

**2 Related Work** **6**
2.1 ScirPy [4] . . . 6
2.2 LazyFatPandas [2] . . . 6
2.3 DIAS : Dynamic Rewriting of Pandas Code [5] . . . 7

**3 Runtime DIAS Optimizations** **7**
3.1 Motivation . . . 7
3.1.1 Runtime vs Compile Time Optimizations . . . 7
3.1.2 Purpose . . . 8
3.2 Rules . . . 8
3.2.1 The nsmallest rule . . . 8
3.2.2 The concat rule . . . 8
3.2.3 The fusable isin rule . . . 9
3.2.4 The replace rule . . . 9
3.3 Psuedo Code . . . 10

**4 Performance Evaluation** **10**
4.1 Execution Time Analysis . . . 10
4.2 Memory usage . . . 12

**5 Conclusion** **14**

**6 Future Work** **14**

**References** **15**


# List of Figures


1 The ScirPy Framework . . . 6
2 The LazyFatPandas Overview . . . 7
3 The nsmallest Rule . . . 8
4 The concat Rule . . . 9
5 The fusable isin Rule . . . 9
6 The replace Rule . . . 9
7 Size of dataframe in GB vs Time in seconds (concat rule) . . . 11
8 Size of dataframe in GB vs Time in seconds (fuse in rule) . . . 11
9 Size of dataframe in GB vs Time in seconds (nsmallest rule) . . . 12
10 Size of dataframe in GB vs Time in seconds (replace rule) . . . 12
11 Size of dataframe in GB vs Peak Memory Usage in GB (concat rule) . . . 13
12 Size of dataframe in GB vs Peak Memory Usage in GB (fuse in rule) . . . 13
13 Size of dataframe in GB vs Peak Memory Usage in GB (nsmallest rule) 14

# 1 Introduction

Pandas is a popular and robust Python open-source framework for data analysis and manipulation. Its expressive and straightforward syntax helps with data cleaning, transformation, and exploration. One of the main advantages of Pandas is the DataFrame object, which offers a tabular data structure like to a spreadsheet or a SQL table and enables users to quickly perform operations on rows and columns. In addition, Pandas provides an extensive collection of functions and techniques for data preprocessing, statistical analysis, and visualisation. As such, it is an indispensable tool for a wide range of jobs. However, like any software tool, Pandas is not without its limitations. Pandas could use some work in the field of large dataset optimisation, especially given the increasing size of datasets used in contemporary data analysis. Pandas may experience performance difficulties when working with huge datasets because of things like memory utilisation, computational complexity, and ineffective techniques. Pandas DataFrames store data in memory, which can become a limitation when working with large datasets that exceed available memory.

Pandas is extended by tools like Dask [1] and Modin [3] to handle scalability and performance issues that arise while handling big datasets. Dask provides parallel computing capabilities by tilizing distributed computing across multiple cores or clusters. It manages the segmentation and parallel execution of operations transparently, all the while providing a familiar Pandas-like interface. On the other hand, Modin is able to perform both row and coloumn parallel operations. Modin also relies on eager-evaluation, unlike lazy evaluation in Dask. Unfortunately, many end-users struggle to choose the best framework for their use case due to a lack of thorough understanding about them. It is also possible that the size of the input dataset may change over time. Writing an effective program in the chosen framework is made more difficult by the several different APIs accompanied by them.
ScirPy [4] is a source to source program translation tool that makes use of static analysis techniques to select only the available attributes. LazyFatPandas [2] is a framework built to perform lazy evaluations on the rewritten source from ScirPy. It chooses its backend engines, which act as scalable executors, from Dask, Modin and Pure Pandas to run programs efficiently.

In order to overcome the vertical scalability issue with data frame libraries, Dias [5] suggests an alternative method that involves rewriting notebook cells to use quicker, yet semantically comparable, code sequences to speed up data frame operations. Dias is made up of two parts : a pattern matcher followed by a rewriter. With its lightweight design and swift pattern matcher that enables us to rewrite patterns into quicker variants, the rewrite engine ensures correctness within interactive latencies by performing appropriate static and runtime checks.

In this project, we implement some of the rewrite rules from Dias into LazyFatPandas framework. During runtime, LazyFatPandas generates a task graph indicating all the useful actions performed in a pandas program and their order of execution. We remake this task graph by optimizing nodes according to the rewrite rules. We observe significant changes in performance in terms of time and memory.

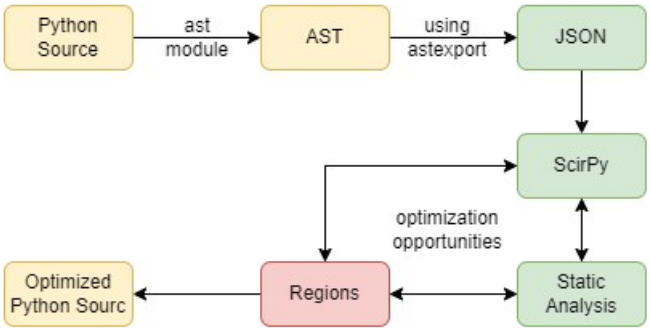


Figure 1: The ScirPy Framework

# 2 Related Work

## 2.1 ScirPy [4]

ScirPy stands for Soot Compatible Intermediate Representation for Python. Certain optimizations (mentioned below) used to rewrite programs show major improvement.

- Data Selection : Live attribute analysis identifies and selects only the vital columns used within a program.
- Metadata Based Optimizations : Exploiting metadata to obtain appropriate data types for dataframes boosts efficiency in terms of storage and computation.
- Multistage Data Fetching : Loading only those columns which are used immediately, as well as expelling columns which are out of use, solves memory overloading issues.

A summary of SCIRPy is presented in Figure 1. The input programme is transformed into an AST (Abstract Syntax Tree) form. Soot is an analysis framework for Java programs that is used by SCIRPy. We must accurately translate Python's AST into a format compatible with Soot, as the tool is primarily meant for Java programs. An Intermediate-Representation that is compatible with Soot is created from the AST. Next, SCIRPy does the aforementioned static optimisations using Soot. Additionally, it collects potentially helpful metadata and statistics regarding upcoming data. The program that has now been statically optimised is run in a standard manner.

## 2.2 LazyFatPandas [2]

LazyFatPandas bypasses the eager evaluation and in-memory computation that is the bottleneck of pure Pandas. LaFP works in three steps -

- ScirPy : The original python program is taken as input and ScirPy optimizations are applied to it in order to create a task graph.

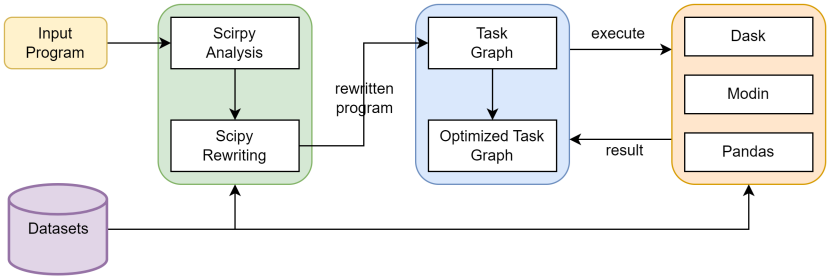


Figure 2: The LazyFatPandas Overview

- Runtime Optimizations : Optimizations beyond static analysis help improve performance on-the-go. Techniques such as column selection, row selection, redundant operation elimination and cost estimation negate memory bound errors. An optimized task graph is generated as a result of this step.
- Backend Engine Selector : Finally, the engine for execution is chosen based on size of data. Dask and Modin are preferred for larger datasets, while Pandas works the quickest on im-memory computation.

Figure 2 shows the overall picture and steps followed in LazyFatPandas.

## 2.3 DIAS : Dynamic Rewriting of Pandas Code [5]

DIAS generates an AST of the given code and majorly consists of two steps :

- Syntactic Pattern Matcher : DIAS has a set of predefined rules, which it aims to match with any piece of code in the program. These rules usually involve certain APIs of Pandas on the left hand side, which can be optimized using other Python calls, which make up the right hand side. The Pattern Matcher identifies these LHS rules as a part of the code, and pass them on to the rewriter.
- Rewriter : If the matched LHS does not depend on any pre-conditions, the rewriter transforms it into RHS in runtime. The tranformed code is then executed using Pandas to acheive a significant speedup.

# 3 Runtime DIAS Optimizations

## 3.1 Motivation

### 3.1.1 Runtime vs Compile Time Optimizations

It is important to distinguish between compile-time and run-time optimisations when optimising a Pandas code. Optimizations at compile time aim to increase the effectiveness of code generation and implementation prior to runtime. Static code analysis is one technique utilised in these optimisations; it can find potential performance bottlenecks, variables that are not being used, or computations that are duplicated during compile time. This allows for their removal or optimisation prior to set execution. While runtime optimisations take place during code execution and are aimed at improving speed

dynamically, compile-time optimisations can also involve employing more effective data structures or algorithms to accomplish operations, reducing the total computational complexity of the code. Pandas runtime optimisations can include techniques like caching frequently accessed data to minimise redundant calculations or lazy evaluation, which postpones actions until they are absolutely essential. Furthermore, data manipulation activities can be completed more quickly by using parallelization techniques to divide computations across several CPU cores or even clusters. Developers can improve the scalability and execution speed of their Pandas programmes by combining compile-time and runtime optimisations. This allows for the handling of huge datasets more quickly.

### 3.1.2 Purpose

The purpose of this project is to incorporate DIAS rewrite rules into LazyFatPandas. We perform these optmizations on the generated task graph to identify nodes which match the LHS of a rewrite rule. A new optmized node is then returned containing the substituted RHS of the rule. This fuses the lazy calls of LaFP and runtime upgrades of DIAS. This also runs using the pure python interpreter, instead of the ipython interpreter for notebook cells as mandatory in DIAS.

## 3.2 Rules

### 3.2.1 The nsmallest rule

Due to differences in their underlying algorithms and computational complexity, the nsmallest technique in Pandas is frequently more performant than sort followed by head. The nsmallest approach finds the smallest n elements in a DataFrame quickly and effectively without having to sort the entire dataset by using a specialised algorithm (like a min-heap). It is more effective for tasks where only a tiny subset of the data is required because it quits once it identifies the n smallest items. In particular for large datasets, nsmallest can be more memory-efficient because it eliminates the need to store the complete sorted dataset in memory. An example is shown in figure 3.

```
# DIAS_VERBOSE
our = train_df['total_score'].sort_values().head(4)

### Dias rewrote code:
our = train_df['total_score'].nsmallest(n=4)
```

Figure 3: The nsmallest Rule

**Variations of nsmallest rule.** The sort_values() and head() functions may also be called from inside another function. Handling the arguments in this case can be tricky and the dependencies of functions must be checked before matching the rules.

### 3.2.2 The concat rule

This optimization is performed when it is required to merge two or more series into a single series. Instead of using the '+' operator over lists to merge the two series, we can

use the concat() function over attributes from the dataframe. The concat() function in pandas does not iterate over individual elements, instead it performs operations over the entire set of data. This is achieved by low level libraries and parallelism. It can also directly work on Series objects rather than converting them to lists beforehand. Finally, it results in a clean sequential index over series, especially if the original series was not indexed. An example is shown in figure 4.

```
# DIAS_VERBOSE
defa = pd.Series(df['Name'].tolist() + df['Ticket'].tolist())

### Dias rewrote code:
defa = pd.concat([df['Name'], df['Ticket']], ignore_index=True)
```

Figure 4: The concat Rule

### 3.2.3 The fusable isin rule

If we want to filter the same column where cell data belongs to two lists, we can fuse them into one list using '+'. In terms of readability and conciseness, this might be better, but both ways are probably going to perform similarly in terms of efficiency because they effectively do the same thing. However, we observed that in the paper [5] they were unable to fuse more than two lists. We are able to overcome this to fuse any number of conditions. An example is shown in figure 5.

```
# DIAS_VERBOSE
our = df2.loc[(df2['Industry'].isin(env)) | (df2['Industry'].isin(ai))].reset_index(drop=True,inplace=False)

### Dias rewrote code:
our = df2.loc[df2['Industry'].isin(env + ai)].reset_index(drop=True,
    inplace=False)
```

Figure 5: The fusable isin Rule

### 3.2.4 The replace rule

If we want to replace some values in a column with other values, we use the replace function. We can add the 'inplace=True' argument to the replace function as an optimization. Especially for larger datasets, this might have a minor speed and memory use benefit. This is due to the fact that it alters the dataframe in-place rather than copying the column, which can eliminate the requirement for extra memory allocation and conserve memory. This, however, may not be noticeable for smaller datasets due to im-memory computations. An example is shown in figure 6.

```
# DIAS_VERBOSE
popul_df['name'] = popul_df['name'].replace(['Buck(Brows)'],"Buck")

### Dias rewrote code:
popul_df['name'].replace('Buck(Brows)', 'Buck', inplace=True)
```

Figure 6: The replace Rule

### 3.3 Psuedo Code

**Algorithm 1** An algorithm to optimize task graph from DIAS rules

```
1:  procedure OptimizeNode(root)                    ▷ root is of the task graph
2:      visited ← {}
3:      procedure DFS(node)
4:          if node in visited then
5:              return
6:          end if
7:          if node matches pattern then
8:              dependencies ← traverseChildren(node)
9:              if no dependencies then
10:                 new_node ← Node(node_info, new_args)
11:                 return new_node
12:             end if
13:         end if
14:         while node.sources not None do
15:             DFS(node.source)
16:         end while
17:     end procedure
18:     new_node ← DFS(root)
19:     return new_node
20: end procedure
```

# 4 Performance Evaluation

Our experiments are conducted on a machine equipped with an AMD Ryzen 5600H CPU, 8 GB of RAM, a 512GB SSD, and running Ubuntu 22.04 LTS. We have mainly analysed the Execution time and Peak Memory usage with respect to the dataset size.

## 4.1 Execution Time Analysis

Here we have given a detailed analysis of execution time with varying dataframe size, and also give insights why our model is performing better for some special cases. Figure 7 shows the performance analysis of the *concat rule* mentioned in section 3.2.2 with respect to the time and dataframe size. It can be observed that Pandas and the LaFP both take equal amount of time at the begining for the smaller dataframes, but after a certain point LaFP started taking more executioin time continously. But for rescue the LaFP with Dias implementation comes and it is clearly visible that our implementation takes less execution time throughout the experiment and for larger dataframe sizes the execution time improves significantly. Overall we achieve 10% to 20% improvement in overall execution time.

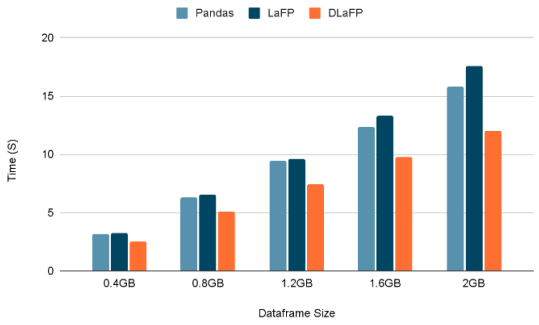


Figure 7: Size of dataframe in GB vs Time in seconds (concat rule)

Figure 8 presents the performance analysis of the *isin rule* discussed in Section 3.2.3 concerning execution time and dataframe size. Initially, the execution times of Pandas and LaFP show similar trends with minimal improvement. However, with the LaFP implementation using the Dias Rule, it is noticeable that the execution time is reduced, showing a modest but significant improvement of approximately 10% to 15% compared to Pandas and standard LaFP.

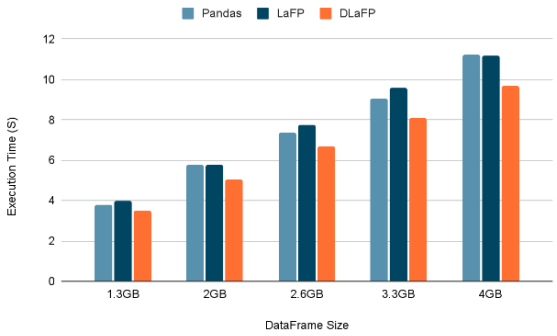


Figure 8: Size of dataframe in GB vs Time in seconds (fuse in rule)

Figure 9 provides insights into the performance of the nsmallest rule discussed in Section 3.2.1, specifically focusing on execution time. It is evident that there is a substantial improvement in execution time compared to Pandas and LaFP. In case of LaFP with Dias implementation the execution time for this rule increases linearly, whereas the other two frameworks experience crashes after reaching a certain threshold. This notable enhancement can be attributed to the nsmallest method, which efficiently identifies the top $n$ smallest values in a column using the min-heap property, resulting in faster performance compared to the sort_values() method that processes the entire column. Therefore, for larger datasets, we achieve nearly 4 times faster execution time using this optimized approach.

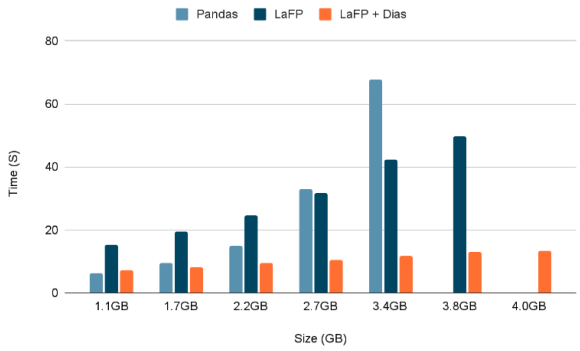


Figure 9: Size of dataframe in GB vs Time in seconds (nsmallest rule)

In Figure 10, it is evident that there is no difference in execution time with LaFP. This is primarily due to the limitation of our current LaFP framework, which does not efficiently support inplace operations. Despite our efforts to implement support for inplace operations, we were unable to achieve significant improvements. The additional overhead required to support inplace operations, combined with the limited capabilities of our current framework, contributed to this outcome.

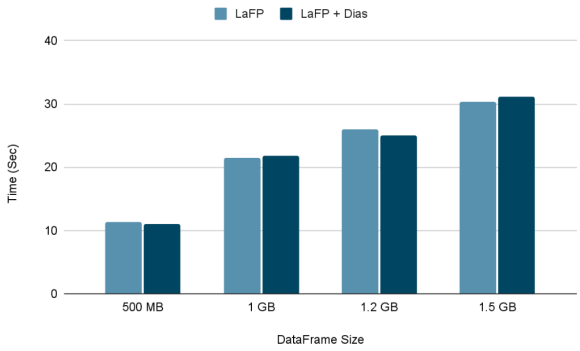


Figure 10: Size of dataframe in GB vs Time in seconds (replace rule)

## 4.2 Memory usage

Here, we have provided a comprehensive analysis of peak memory usage across different dataframe sizes, highlighting the factors that contribute to the superior performance of our model in specific scenarios. Figure 11 demonstrates that our implementation of the concat rule consumes less memory compared to both Pandas and the LaFP framework. This improvement is achieved by leveraging Pandas' built-in functions, which optimize memory usage by avoiding the maintenance of separate lists for each column, unlike the original code. The improvement is ranging from 5% to 20% almost as seen in the plot.

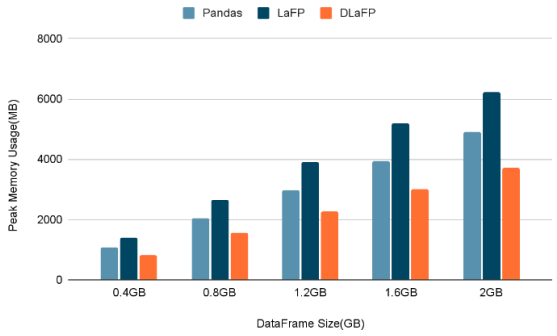


Figure 11: Size of dataframe in GB vs Peak Memory Usage in GB (concat rule)

As seen from the figure 12 it is clearly visible that the memrory footprint is always less than the other two frameworks but not of a significant improvement. This is generally because both the original and the rewritten rule is using a single column for the operation and only the difference is it maintains a single in-built list for the selection and the original does not also instead of doing the selection for the same row multiple times the rewritten rule just scan the column only once.

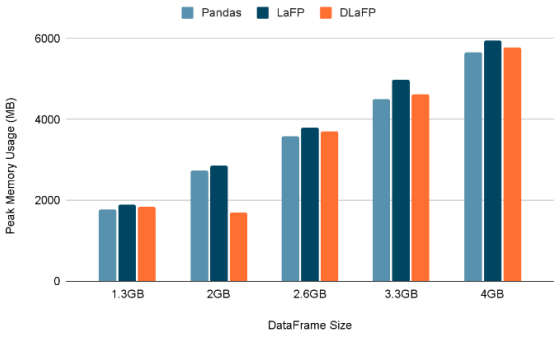


Figure 12: Size of dataframe in GB vs Peak Memory Usage in GB (fuse in rule)

In Figure 13, it is evident that our model consumes significantly less memory compared to the other two frameworks for the nsmallestrule. The memory usage remains relatively constant across varying dataframe sizes. This efficiency is achieved because the rewritten rule selectively loads and sorts only the column required for the operation, extracting the smallest $n$ values. As a result, the overall memory footprint remains nearly constant for the nsmallest rule.

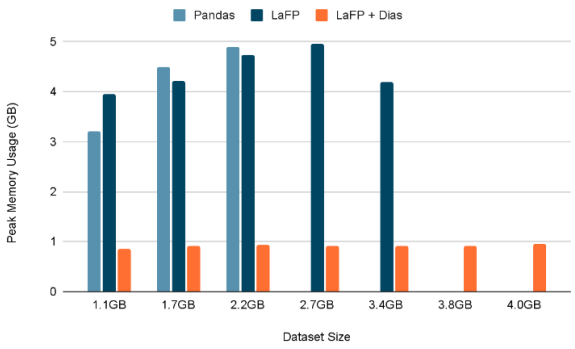


Figure 13: Size of dataframe in GB vs Peak Memory Usage in GB (nsmallest rule)

# 5 Conclusion

In this project, we got to learn about many interesting ideas including lazy evaluation, compile time rewriting, runtime rewriting, source to source tools etc. We were able to incude multiple optimization rules into the existing codebase. We were able to observe significant time and memory improvements in most cases.

# 6 Future Work

We wish to further this project by involving compile time optimizations as well into lazyfatpandas. We also wish to look into memory performance and improvements. We can consider more benchmark datasets and testing suites for these optimizations in the future.

# References


[1] *Dask: Scaling Python code natively.* URL: https://dask.org/.

[2] Priyesh Kumar. "Rewriting Pandas Programs for Optimized Lazy Evaluation". In: *CSE, IITB, Final Thesis* (2022).

[3] *Modin : Faster pandas, even on your laptop.* URL: https://modin.readthedocs.io/en/stable/.

[4] Bhushan Pal Singh, Mudra Sahu, and S. Sudarshan. "Optimizing Data Science Applications using Static Analysis". In: *DBPL '21: The 18th International Symposium on Database Programming Languages, Copenhagen, Denmark* (2021), pp. 23–27.

[5] Baziotis Stefanos, Daniel Kang, and Charith Mendis. "Dias: Dynamic Rewriting of Pandas Code". In: *Proceedings of the ACM on Management of Data* (2024).