Efficient Dataframe Systems: Lazy Fat Pandas on a Diet

Bhushan Pal Singh
singhbhushan@cse.iitb.ac.in
Indian Institute of Technology Bombay, Mumbai
NewSpace India Limited, Bengaluru
India

Priyesh Kumar*
priyesh9875@gmail.com
Indian Institute of Technology Bombay
Mumbai, India

Chiranmoy Bhattacharya†
chiranmoy358@gmail.com
Indian Institute of Technology Bombay
Mumbai, India

S. Sudarshan
sudarsha@cse.iitb.ac.in
Indian Institute of Technology Bombay
Mumbai, India

Abstract

Pandas is widely used for data science applications, but users often run into problems when datasets are larger than memory. There are several frameworks based on lazy evaluation that handle large datasets, but the programs have to be rewritten based on the chosen framework. In this paper, we present an optimization framework that allows programmers to code in plain Pandas, but get the benefit of not only scalability, but also multiple optimizations based on a combination of "just-in-time" static analysis of the program and lazy-evaluation based run-time-optimizations. The programmer only needs to add a couple of lines of code to use our framework, and to choose from any of several backend engines (currently Pandas, Dask, Pandas On Spark, DuckDB using Ibis, Polars using Ibis, and Modin). Performance results on a variety of programs show the significant benefits of our optimization framework compared not only to Pandas, but also compared to the direct use of Dask, Modin, DuckDB using Ibis, Polars using Ibis, and Pandas on Spark.

Keywords

Pandas, Dataframe Systems, Lazy Evaluation, Static Optimization

1 Introduction

Python is widely used for data science applications, and in particular dataframe-based libraries and frameworks have become the default model for many of these applications. Pandas is the most popular framework among these and is the tool of choice for applications that use smaller datasets that fit in the available system memory. To address the needs of applications that have larger datasets that do not fit in memory, several scalable frameworks have been created, such as Dask [1], Modin [2], PySpark/Pandas on Spark [3], Magpie [4], among others. Some frameworks such as Dask and Spark are lazy evaluation frameworks that create a task graph lazily, optimize it, and then execute it when results are needed. Others like Modin support eager evaluation.

Many users develop their applications using Pandas, and test them on small datasets; performance issues, especially out-of-memory issues, are not obvious until much larger datasets are used. Even if production datasets fit in memory at a point in time, an increase in data size often causes problems at a later point in time.

*Current affiliation: Dream11, India
†Current affiliation: Fujitsu Research of India

EDBT '26, Tampere (Finland)


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

Figure 1: Code changes to use Lazy Fat Pandas

While users can avoid these problems by using scalable frameworks, there are several challenges. Frameworks based on lazy evaluation can speed up evaluation by using optimizations of the task graph, but require users to modify their code to work with lazy evaluation, for example, by explicitly forcing of computation before any non-lazy operation (e.g. print()) is executed on a dataframe. Which framework is optimal for an application depends on factors such as data size. To use a framework, the user needs to deal with variations in their APIs, including lack of support for some Pandas features. Rewriting code to work on a different framework is thus non-trivial. This makes it hard to use the optimal backend if the application is coded against a different backend.

In this paper, we describe a system, which we call *Lazy Fat Pandas* (*LaFP*), which we have developed to address the above-mentioned challenges faced by the data science user community. Our system allows users to use plain Pandas as front-end, but rewrites the program to optimize it and to execute it using any one of the supported back-end systems. LaFP uses static analysis along with a lazy API wrapper to perform a number of optimizations for efficient execution of these applications.

In order to benefit from LaFP, users can continue to write their programs in Pandas and just need to perform a couple of lines of code changes, as highlighted in Figure 1. Users must replace an import of Pandas by an import of our LaFP library, and add a call to pd.analyze(). Additionally, the users can specify the desired backend (automated choice of backend is an area of ongoing work). Users do not need to worry about the variations between APIs and execution models of different back-end executors.[1] LaFP uses a combination of program rewriting and a lazy wrapper to introduce lazy evaluation and optimization, and to provide workarounds to deal with limitations of the chosen framework.

The specific technical contributions of this paper include the following.

(1) We present a novel optimization architecture (Section 2) based on *static analysis*, to perform source-to-source transformation of Python programs.

[1]Support for a few APIs such as apply() or get_dummies() on backends that do not support them, is a work in progress, as discussed in Section 5.1.

10.48786/edbt.2026.14

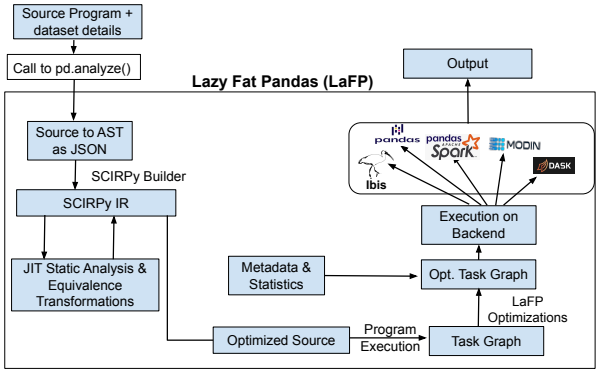


Figure 2: Overview of the Lazy Fat Pandas System

The Python source program is first converted to the SCIRPy intermediate representation (IR), which we have created, which is compatible with the Soot static analysis framework [5].
Based on static analysis, the IR is rewritten to optimize execution, and then is converted back to Python for execution. We discuss the optimizations shortly.

(2) We designed and implemented a novel *just-in-time (JIT) static analysis* technique, which requires only the addition of a single function call, pd.analyze(), to the program. Our analyze() function uses reflection to find and rewrite the source code of the program, and replace the execution of the original program by the execution of the optimized rewritten program. No changes are required to the outer-level systems that invoke the Python programs, greatly simplifying the task of deploying the optimizations.

(3) We have developed a runtime wrapper API which allows easy substitution of invocations of Pandas API calls by invocations of lazy versions of the same calls in LaFP. Lazy evaluation allows the construction of a task graph, which is a DAG of operators, which can be optimized before execution, when results are needed.
The LaFP wrapper supports diverse back-ends, such as Pandas, Dask, Modin, Pandas on Spark (formerly called Koalas), DuckDB [6] using Ibis [7], and Polars [8] using Ibis as backends. The programmer just adds one line of code to specify the desired back-end. Lazy backends require computation to be forced at certain points; forcing of computation is added automatically by our system, based on static analysis.
The back-ends differ in the support for specific Pandas API functionality. If a chosen back-end does not support a specific Pandas API functionality, LaFP can convert data from the back-end representation back to Pandas, to execute the original Pandas function. Thus, the user need not worry about the differences between back-ends, or their specific limitations (we note however that Dask, DuckDB, and Polars do not preserve ordering of rows in a dataframe, so users who choose any of these as the back-end should be aware of this difference).

(4) Our system supports lazy execution for Python functions other than Pandas dataframe operations. For example, LaFP provides a lazy print function, which is a wrapper around normal print(), which becomes part of the lazy task graph. Lazy print allows evaluation to be delayed to a later point, increasing opportunities for performance optimization. However, where such lazy evaluation is not possible, our system rewrites the program to add forcing of computation.

(5) We present a number of optimization techniques based on static analysis (Sections 3). For example, static analysis allows our optimizer to look ahead to predict which dataframes are live, and what parts of the dataframe (based on column selections) will get used later in the program, and rewrite the program to avoid redundant fetches/computation (Section 3.1). We also use static analysis to enable other optimizations such as lazy print of dataframes (Section 3.2), forced computation (Section 3.3), and common computation reuse (Section 3.4).
We note that lazy runtimes optimize task graphs before they are executed. However, programs often require multiple task graphs to be created and executed, each of which is independently optimized. Static analysis provides a global view of the program, across multiple task graphs, supporting optimizations that examine later parts of the program. For example, a task graph execution may materialize a dataframe with multiple columns. Static analysis can detect that only a few of the columns are used later in the program, and optimize the computation by fetching only the required columns. Purely run-time optimizers cannot implement such optimizations.

(6) We have implemented the LaFP framework, along with static and run-time optimizations. Our performance studies (Section 5) show that our optimization methodology can significantly improve the performance of such programs, across all the backends (including those that have their own optimizers), with up to 20x speedup of execution time. Memory usage is also substantially reduced, by up to 95%.

2 Architecture

Users can run existing Pandas-based programs using our Lazy Fat Pandas (LaFP) framework with minimal changes as shown in Figure 1. LaFP currently supports 120 major Pandas API functions out of around 240 APIs, including the bulk of the widely used API functionality; adding support for further API calls, with the goal of 100% compatibility with Pandas, is an ongoing activity.

Figure 2 shows an overview of our proposed framework. LaFP first converts programs to a lower-level internal representation (IR) using Just-in-Time (JIT) static analysis, and based on it, rewrites the program to optimize evaluation.

The optimized program, where calls to Pandas dataframe operations have been replaced by their lazy wrapper versions, is then executed. The lazy wrapper functions create a task-graph and add each API call as a node to task graph. When computation is forced, the task graph is optimized by the LaFP runtime, and then executed. The runtime optimization makes use of metadata and statistics to perform optimizations that cannot be done at compile time. LaFP uses any one of multiple back-ends to execute task graphs.

2.1 Intermediate Representation

We use the Soot [5] framework, originally developed for Java, for static analysis of Python programs; Soot works on an intermediate representation (IR) of the program. While IRs for JVM bytecode are built in, Soot allows creation of new IRs.

We developed an intermediate representation which we call SCIRPy IR (short for Soot Compatible Intermediate Representation for Python) to support analysis. SCIRPy supports the Python Abstract Syntax Tree (AST) representation, while being compatible with Soot [5].

Most constructs in the SCIRPy IR, such as if statements, assignment statements, etc., extend the Jimple IR of Soot, allowing us to extend and use the built-in static analysis functionality of Soot. SCIRPy further maintains compatibility with Soot by extending Soot's class, method, and body for Python.[2]

Python source is parsed into an abstract syntax tree (AST) using an existing parser (written in Python), and the AST is then translated into a JSON (JavaScript Object Notation) object. This JSON object is passed to our static analysis code, which transforms it into SCIRPy. For compatibility with Soot, our static analysis code is written in Java.

We used Soot as it supports control flow graph construction and data flow analyses, which we extend and use in our static analysis. There are several static analysis tools for Python based on 'ast' module of Python or its derivatives, such as Pylint [9], Pyflakes [10], Mypy [11], Prospector [12], and Bandit [13], among others. However these do not support the static analysis techniques used in our optimizations.

2.2 Static Analysis and Optimization

Static analysis is performed in SCIRPy by building Control Flow Graphs (CFG) and performing data flow analysis (DFA).

Control Flow Graph (CFG): The CFG is a representation of the flow of control within a program [14]. It represents various paths that program execution may take. It is a directed graph in which every node represents a Basic Block (BB). A basic block is a sequential fragment of code without any branch or loop. CFGs are generally constructed on intermediate representation (IR). We construct a CFG from the SCIRPy IR using Soot.

Data Flow Analysis (DFA): DFA is a technique to identify how a program or a method manipulates its data [15], and is performed on CFG. We have currently defined and implemented two data flow analyses: live attribute analysis (Section 3.1) and live dataframe analysis (Section 3.4) to statically optimize data science programs.

The results of static analysis are used to perform a variety of optimizations. Static analysis provides information on preconditions, as well as other information needed to carry out these optimizations.

For example, consider the Pandas program in Figure 3 (taken from [4]), which fetches data from files into in-memory dataframes, and performs transformation operations such as data filtering, feature addition, and aggregation, on the dataframes. The optimized version (output of compile time optimization) of the program is as shown in Figure 4 with highlighted comments explaining the changes.

The original program fetches all 22 columns from the dataset. Only 3 of these are used in the program, which is inferred by a static analysis technique called live attribute analysis, as discussed in Section 3.1. The optimized program after applying the column selection optimization fetches only the required 3 columns, by passing them in the usecols option to read_csv.

We also use compile time analysis to enable other optimizations such as lazy print (Section 3.2), forced computation (Section 3.3), and common computation reuse (Section 3.4).

[2]Extending the IR to handle exceptions is a part of future work.

2.3 Rewriting to Python

In order to convert optimized SCIRPy back to Python source, we first convert the CFG-based IR to an intermediate representation based on program regions. Program regions represent the hierarchical structure of block-structured programs. These regions could be basic block regions, loop regions, branch (if-then-else) regions, or sequential regions, each of which could hierarchically contain other subregions. For example, a loop region may contain a sequential region which in turn may be composed of a branch region and another loop region.

Creating regions from the graph-based SCIRPy representation is done using techniques described in [16], which are also used in [17]. The region-based representation is then translated to Python source code.

We note that static analysis cannot handle dynamically generated code that is executed using the exec() function. Static analysis of Python has other challenges, such as not knowing which overloaded function is being invoked when decorators are used to implement overloading based on types, due to lack of static typing. Further, we currently do not handle global variables and closures. Also, we support a limited form of inter-procedural static analysis. However, Pandas applications typically do not use features that cause such issues, and our analysis is sound as long as such features are not used. *Soundiness*, i.e. use of analysis that are sound only as long as some rare constructs are not used, is widely adopted in static analysis [18]. Transformations based on static analysis are not performed when features that affect soundness are used.

2.4 Just-in-Time Static Analysis

One of the novel contributions of our approach is the Just-in-Time (JIT) static analysis, which performs static analysis at the start of program execution. Other static analysis tools require users to perform static analysis and program rewrite as a separate phase, following which the rewritten program must be executed. In contrast, our approach does not require any change in the flow of code optimization/execution.

The process of JIT static analysis is described in Figure 5. The pd.analyze() method transfers the control to LaFP. LaFP identifies the source program code, parses it, converts it to SCIRPy, and performs static analysis and compile time optimizations. Further, code transformations are performed during this phase to enable runtime/lazy optimizations discussed in Section 3. As discussed, the optimized IR is converted back to Python and executed lazily using our lazy runtime wrapper.

2.5 Task Graphs and Lazy Evaluation

A task graph is a directed acyclic graph (DAG) in which the nodes represent a computational task or operation and the edges denote the precedence constraints among these computational tasks. In eager evaluation, an expression is evaluated as soon as it is reached during the execution of a program. In contrast, in lazy evaluation, when an expression is reached during program execution, instead of evaluating it eagerly, an expression node is created, and added to a task graph. The task graph is evaluated only when it is needed, by calling a function that forces the evaluation. Dask, Pandas on Spark, and Ibis with DuckDB/Polars are examples of systems based on lazy evaluation of dataframes. Evaluation is forced only when a function such as `compute()` is called to actually execute the operations; Pandas on Spark internally forces computation when the contents of a dataframe are

```
import lazyfatpandas.pandas as pd
pd.analyze() # transfer control to LaFP
df = pd.read_csv('data.csv', parse_dates=['
     tpep_pickup_datetime']) #fetch data
df = df[df.fare_amount > 0] # filter bad rows
df['day'] = df.tpep_pickup_datetime.dt.dayofweek # add
     features
df = df.groupby(['day'])['passenger_count'].sum()
print(df) # use dataframe
```

Figure 3: Sample Program

function PD.ANALYZE(())
    source_file ← get_source_code()
    SCIRPy ← python_to_SCIRPy(source_file)
    opt_SCIRPy ← static_analysis_opt(SCIRPy)
    opt_code ← SCIRPy_to_python_opt(opt_SCIRPy)
    executor (opt_code)
end function

Figure 5: Just-in-Time Static Analysis

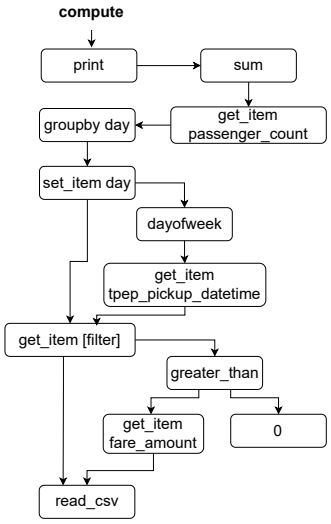


Figure 6: LaFP Task graph of the Program in Figure 3

used, but the other lazy frameworks require an explicit compute() call.

Our Lazy Fat Pandas (LaFP) framework is a lazy framework, which acts as a wrapper layer, allowing execution of dataframe operations to be done using any of the supported backends, including Pandas itself.

The task graph for the program in Figure 3 is shown in Figure 6. An edge $(A \rightarrow B)$ represents that the task B depends on task A. Such an edge may be created when the result of the operation at node B is an input for the operation at node A[3]. This dependency is also used to enforce output order for lazy print statements and other output functions supported by the LaFP lazy wrapper.

The optimized source code generated by JIT static analysis and rewriting has calls to lazy versions of the Pandas API calls, defined in our Lazy Fat Pandas (LaFP) framework, which supports the Pandas dataframe API, but using the LaFP's lazy fat-dataframe.

[3]The direction of the edge follows the convention for task graphs and dependency graphs, although the flow of data is in the opposite direction.

```
import lazyfatpandas.pandas as pd
from lazyfatpandas.func import print  # Use lazyPrint
SO_columns = ['pickup_datetime','passenger_count','
     fare_amount']  #
     Live columns passed to read_csv()
df = pd.read_csv('data.csv',usecols=SO_columns,
     parse_dates=['tpep_pickup_datetime'])
df = df[df.fare_amount > 0]
df['day'] = df.tpep_pickup_datetime.dt.dayofweek
df = df.groupby(['day'])['passenger_count'].sum()
print(df)
pd.flush() # Forces computation and printing
```

Figure 4: Optimized Version of Sample Program

A call to the LaFP's API function does not immediately execute the operation. Instead, each operation creates a new lazy fat-dataframe object, which is then linked to the task graph based on its inputs. Most API operations operate on dataframes and output other dataframes. The lowest level operations create dataframes, typically by reading data from file representations such as CSV, or Parquet, or from databases, although dataframes may also be created from constants or random number generators.

The task graph is executed only when an operation result needs to be passed to an operation that needs a materialized (non-lazy) dataframe, or at the end of the program, when computation can no longer be deferred. The task graph is optimized by LaFP before it is executed using any of the back-ends supported by LaFP.

LaFP allows the back-end to be chosen in the program. For small datasets that fit in memory, Pandas is usually faster than Dask or Pandas on Spark, or other scalable frameworks, and is the preferred choice. In this case LaFP optimizations help speed up the program compared to directly running it on Pandas. If the back-end chosen is lazy, it may perform its own optimizations; in that case the optimizations performed by LaFP complement the back-end optimizations.

One of the distinctive features of our approach compared to just using a lazy framework is that it is capable of utilizing information generated using JIT static analysis phase. This information provides look-ahead beyond points where lazy computation cannot be deferred further, allowing us to detect for example that some columns are not used later in the program and therefore can be projected away, or not even computed earlier in the execution. Static-analysis based rewriting also allows our framework to detect which API calls can handle LaFP dataframes; all other API calls (like plotting) default to Pandas dataframes, and our optimizer introduces calls to force computation before invoking such APIs.

2.6 Run-Time Optimizations and Execution

LaFP allows the task graph to be executed on any one of the supported back-ends. LaFP currently supports Pandas, Dask, Pandas on Spark (formerly known as Koalas), Modin, DuckDB using Ibis, and Polars using Ibis, with the default being Dask. The user can select the required back-end by just adding one line of code, for example:

`pd.BACKEND_ENGINE=pd.BackendEngines.PANDAS`

The choice of the back-end significantly impacts execution time and memory usage for a program. Pandas and Modin use an eager evaluation approach with all data required to be in memory

(or distributed memory in the case of Modin). In contrast, Dask, Ibis (DuckDB/Polars), and Pandas on Spark employ lazy evaluation, and support data sizes larger than memory. An appropriate back-end should be chosen based on the system requirements; automating this choice is an area of future work.

Irrespective of the back-end, LaFP utilizes static analysis and the lazy runtime wrapper to perform its optimization, before executing the task graph using the selected back-end. Lazy frameworks such as Spark and Dask optimize the task graph further before execution, which can reduce execution costs further. Examples of optimizations include removal of unused computations, pushing selections and projections, and fusing of operators to reduce data movement.

Similarly, the LaFP runtime module performs several optimizations in the task graph (DAG) at runtime. Two of these are done without using any information from static analysis, namely Predicate Pushdown within the task graph, and Metadata Analysis which can change attribute types to reduce storage cost. Others benefit from information collected by static analysis. Column selection (i.e. projection pushing) benefits from live attribute analysis. Common computation reuse can be done within a task graph, but benefits from static analysis information about reuse later in the program.

When a lazy back-end such as Dask or Pandas on Spark is chosen, the optimizations performed in LaFP complement optimizations in the lazy back-end. When an eager back-end such as Pandas itself or Modin is chosen, the back-end cannot perform optimization across nodes, and thus LaFPs lazy evaluation optimizations are more important.

The execution of the task graph in LaFP is done as follows, for the case where the back-end is eager. The LaFP task graph is executed in topological order. After evaluating a task graph node, the result is stored in a field called `result`. This `result` field is cleared once all its dependent nodes have been evaluated, and the result is no longer needed, minimizing memory usage. This is managed by counting the in-degree of each task graph node before executing the task graph and decrementing the count after a node is used to generate another node's result. When the count reaches zero, the `result` field is cleared, allowing Python's garbage collector to reclaim the memory. As the task graph is executed from bottom to top, the results of lower nodes, which have been evaluated and used, are deleted to keep memory usage to a minimum.

For example, consider the task graph in Figure 6, `read_csv` is executed first, and the result is stored in the node. This result is cleared after its dependent nodes `get_item fare_amount`, and `get_item [filter]`, have been executed. Finally, the `result` of the node on which `compute` was called is returned. If `compute()` is called on a print node, then `None` is returned.

In case the back-end is a lazy framework such as Dask or Pandas on Spark, instead of executing the operation when traversing the task graph, the API call is transformed to the compatible API call for the selected lazy back-end. The execution of the operations on the lazy back-end is initiated when the root of the task graph is reached, or when the results are required for an intermediate operation such as an external API call which expects a computed dataframe. Nodes whose results are used more than once can be persisted using persist() on a Dask dataframe, avoiding recomputation for subsequent uses. Pandas on Spark does not support persist; persistence could be implemented by converting Pandas on Spark dataframes to PySpark dataframes but there is a significant cost to conversion, so we do not currently implement it.

API Comptability with Pandas: For back-ends other than Pandas, LaFP performs some transformations to deal with incompatibilities between Pandas and the selected back-end. For example, pandas 'read_csv' API call supports a keyword argument 'index_col' to specify the column(s) to use as row labels for the Dataframe. Dask Dataframe does not support this keyword argument. However, similar behavior can be obtained by making another API call 'set_index' on the Dask Dataframe after 'read_csv'. Our framework is capable of identifying several such inconsistencies across multiple frameworks and performing additional operations transparently such that the end result is the same after the execution of the API calls, irrespective of the back-end.

Further, the changes required to enable execution using lazy frameworks are implemented by a combination of rewriting based on static analysis, for example, to force computation, and via wrapper functions in the LaFP API. The wrapper functions use appropriate backend API calls to implement Pandas API functionality where possible, and in other cases (such as inplace updates, column rename, shape changing operations, etc) convert the backend dataframes to Pandas, applies the function in Pandas, and converts the dataframe back to the backend.

3 Optimizations

In this section, we discuss a number of static and runtime optimizations that are implemented in LaFP. These optimizations are performed in two settings: (i) Rewriting of imperative programs based on static analysis (ii) Optimizing the generated task graph at runtime, before it is executed. As discussed, some of the optimizations we describe exploit a combination of information from static analysis and runtime information.

3.1 Column Selection

In many programs, not all the columns (attributes) from the input dataset are used. Fetching such unused columns into the memory leads to increased memory usage and extra IO operations. Column selection optimization identifies and fetches into memory only those columns that are used later in the program.

Live variable analysis (LVA): A variable is *live at a program point* if there exists a path from that point to the exit of the program along which the current value of the variable may be used. LVA [14], identifies which variables are live at any point in the program.

Live attribute analysis (LAA): We define *live attribute analysis* (*LAA*) based on *LVA*. *LAA* treats attributes (columns) of a dataframe as variable and computes the liveness of individual attributes (columns) of dataframes. Similar to a variable, a dataframe column is live at a program point if there is a path to the program exit along which it may be used. However, columns are different from variables in that assignments or uses can happen at the level of entire dataframes:

(1) If the whole dataframe is used at a program point, all columns of that dataframe become live.
(2) Similarly, all columns of a dataframe are killed at the point of definition of a dataframe.
(3) If a dataframe is derived from another dataframe, its liveness information is used to determine liveness information for the source dataframe.

Live variable analysis is done by using dataflow analysis, which is based on the Gen and Kill sets at each node in the

control flow graph (CFG). The dataflow equations for live attribute analysis, which are modified from those for live variable analysis, are as below:

$$Gen_n = \{d.i \mid i \text{ is a column of dataframe } d, \text{ and either } d.i \text{ or all of } d \text{ (without specifying any column) is used in basic block } n, \text{ prior to any assignment to } d.i \text{ or to } d\}. \quad (1)$$

$$Kill_n = \{d.i \mid i \text{ is a column of dataframe } d, \text{ and either } d.i \text{ or all of } d \text{ (without specifying any) column is assigned in } n\} \quad (2)$$

Note that if a dataframe is passed as an attribute of a function called from n, we assume that all columns of the dataframe are used in n. Global variables pose another challenge, and if a dataframe is assigned to a global variable, we assume conservatively that all its columns are used in any function called from n. Further, aggregate operations kill all columns except those used in the aggregate or in the groupby operation.

We next define sets In_n and Out_n which merge local information provided by Gen_n and $Kill_n$, with information from successor nodes of n, to identify global liveness information.

$$Out_{\mathrm{n}} = \bigcup_{s \in succ(n)} In_{\mathrm{s}} \quad (3)$$

$$In_{\mathrm{n}} = Gen_{\mathrm{n}} \cup \{Out_{\mathrm{n}} - Kill_{\mathrm{n}}\} \quad (4)$$

The above equations are solved to get the Gen, Kill, In and Out sets for each basic block n (basic blocks are defined in Section 2.2). The live attributes at the end of a basic block n are those that are in the set Out_n. In_n represents liveness information immediately before the block and Out_n represents liveness information immediately after the block.

Once LAA is performed, liveness information is available for all columns of all dataframes at all program points. The column selection optimization modifies the IR to fetch only those dataframe columns that are live (in Out_n) of the program point n where the dataframe is created from an input dataset, e.g. by a read_csv() call.

We now consider how live attribute analysis works on the program in Figure 3. This program has only one dataframe, i.e., df. The last statement of the program prints the dataframe, so all columns are live at 'In' of this point. Line 8 results in only columns day and passenger_count being live. At line 6, the column pickup_datetime becomes live, whereas column day is killed as it is assigned and thus not alive before that. Line 4 makes fare_amount live. The columns live at 'Out' of the Line 3 are 'pickup_datetime', 'passenger_count' and 'fare_amount', and only these need to be read from the csv file. The optimized version of the program, which reads only the above columns, is shown in Figure 4.

We also note that informative API functions df.head(), df.info() and df.describe() are frequently used to get an idea of the dataset contents and, their output does not affect the intended program result. Treating these as using all attributes of df would result in unnecessary column retrieval, so, as a heuristic, we ignore the attribute usage of these functions.

3.2 Lazy Print

Dataframe computations in lazy frameworks are deferred until computation is forced by a call to a `compute()` (or similar) method. Computation needs to be forced when passing dataframes to functions, for example print(), that cannot accept lazy data frames. If compute can be deferred, the task graph could include later parts of the code, which can enable other optimizations that may not be possible if compute has to be done earlier. Thus, postponing invocation of `compute()` can help to reduce the execution cost.

Print is one of the most common functions that forces computation. We introduce a lazy version of the print operator, allowing compute calls to be deferred beyond the (lazy) print calls. With our novel approach, lazy print statements are treated as operations and added to the task graph. When the task graph is executed, the print nodes are processed, and the data is printed. The delay in computation enabled by lazy print can allow task graphs that would otherwise be separately executed to be combined and executed together, which can reduce the cost significantly compared to separate executions. However, care must be taken to ensure that outputs are generated in the correct order, and we now describe how we enforce that.

Figure 7 presents an example with multiple print statements. Its optimized version generated by our rewriting techniques is shown in Figure 8, and the equivalent task graph is shown in Figure 9. The optimized program overrides the built-in print method with LaFP's lazy print method by importing print from lazyfatpandas.func. The library lazyfatpandas.func also provides lazy versions of some other functions; for example the lazy version of Python's len() function, when applied to a lazy dataframe, returns a lazy integer, else it behaves like the normal len() function.

When LaFP's lazy print is called, the node representing the print operation is added to the task graph, with the lazy dataframes as the source nodes. A dependency edge is added to the previous print operation (if any) to maintain the correct print order. All the lazy dataframes used in the print statement are identified and appropriate edges added to the task graph to ensure that these dataframes are computed before the lazy print is (eventually) executed.

At the end of the program, `pd.flush` is called, which internally invokes `compute` on the last print node, forcing the computation of the task graph. The print statements are processed in the correct order due to the dependency edges between print nodes. The statements to override print, as well as the call to `pd.flush()` are automatically inserted in the source program by program rewriting, thus fully automating the process.

Python allows objects, including dataframes, to be used in Python's formatted strings (f-strings), for example:

```
print(f'Average fare: {avg_fare}')
```

in Figure 7, where `avg_fare` is a dataframe. Creation of the formatted string would require the dataframe to be computed.

To defer the computation of the formatted string, while retaining the link to the correct dataframe (since the variable may get assigned in a subsequent step before the print is executed) the "lazyprint()" wrapper function replaces the dataframe variable by the unique ID of the task graph node representing the dataframe, along with an escape sequence to mark the unique ID.

When the constructed string is processed by the execution of the deferred print function, the function checks for the escape sequence to identify the unique ID of the task graph node. Further, this node must be computed before the lazy print is processed at the end of the program, which is ensured by the runtime.

3.3 Insertion of Forced Computation

When a program in a lazy framework calls a function that expects an evaluated Pandas dataframe, computation needs to be

```
import lazyfatpandas.pandas as pd
pd.analyze()
df = pd.read_csv("data.csv")
print(df.head())
df["day"] = df.pickup_datetime.dt.dayofweek
p_per_day = df.groupby(["day"])["passenger_count"].sum
    ()
print (p_per_day)
avg_fare = df.fare_amount.mean()
print(f"Average fare: {avg_fare}")
```

Figure 7: Program with Multiple Print Statements

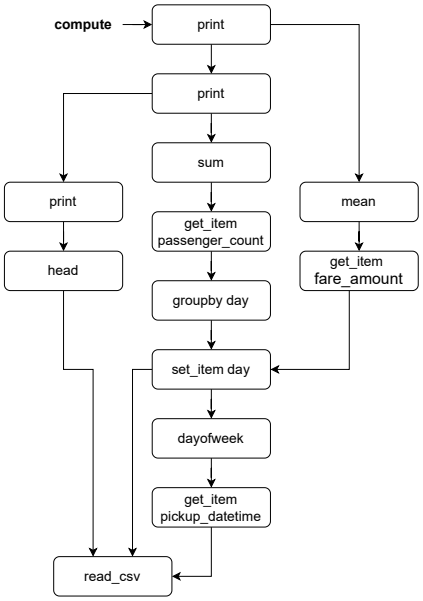


Figure 9: Task Graph for Program in Figure 8

forced before the dataframe is passed to such a function call. Commonly used functions include print() and plotting functions from `matplotlib`. Programmers using lazy frameworks such as Dask or Ibis with DuckDB or Polars have to manually insert code to force computation before calling any such function. Pandas on Spark internally forces computation when a dataframe is converted to a string, which happens for example on a print() call, and also provides wrappers for some other functions such as plotting which force computation, but for other functions computation needs to be forced by converting a Pandas on Spark dataframe to a Pandas dataframe.

For some cases, like for print(), our framework provides a lazy wrapper, allowing the function to be invoked when the dataframe is eventually computed; we are currently implementing lazy wrappers for some other functions. However such lazy wrappers cannot be used in general, for example with functions which return values that are used subsequently.

To deal with the above issue, the program rewriting phase adds a compute() call to force computation before execution of any function call for which a lazy implementation is not available; the materialized (computed) dataframe is then passed to the function call. Static analysis allows us to automate the forcing of materialization, which would have to be done manually if a lazy framework is used directly.

```
import ...
from lazyfatpandas.func import print # Use lazyPrint
...[ optimized read csv as in Figure 4 ]
print(df.head())  # lazy print
df['day'] = df.pickup_datetime.dt.dayofweek
p_per_day = df.groupby(['day'])['passenger_count'].sum
    ()
print(p_per_day)  # lazy print
avg_fare = df.fare_amount.mean()
print(f'Average fare: {avg_fare}')  # lazy print
pd.flush() # Force computation and printing
```

Figure 8: Optimized Program with Lazy Print

The program in Figure 10 and its optimized version in Figure 11 demonstrate forced computation. Our static analysis detects when a dataframe is passed as an argument to a function that is not known to support lazy dataframes. To handle such functions, a call to `compute` is added; for example line 9 of Figure 10 is rewritten as shown in Figure 11 line 10 to force computation before invocation of of the plt.plot() function. (We pass an argument live_df=[df] to the compute() function, which is an optimization that is discussed later).

To invoke `compute` on a dataframe, we need to figure out which variables are dataframe variables. This information is inferred from the types of the Pandas API calls. For example, Pandas functions like read_csv() or read_parquet(), as well as most Pandas functions on dataframes return dataframes.

Further, function calls from external modules like `matplotlib.pyplot` can generate output, which can conflict with lazy printing since the output order may get changed. To solve this issue, pending print operations are processed when a dataframe is forced to compute, maintaining the correct output order.

In line 11 of Figure 11, when `p_per_day.compute()` is invoked, the task graph containing all the lazy calls up to that point, including lazy prints, are executed, and the result of the node `sum` (`p_per_day`) passed to `plt.plot`, generating a plot image. Subsequently, when `pd.flush` is called in line 15, the print in line 14 is processed, along with lazy operations that are pending. Note that the shared subexpression corresponding to the dataframe df computed in line 6 would get recomputed on further execution in the program when pd.flush() is called. When the shared subexpression is first computed, information related to the future usage of the dataframe in the rest of the program is unavailable to the lazy framework. We can use static analysis to avoid recomputation, as discussed next in Section 3.4.

3.4 Common Computation Reuse

As discussed in the previous section, forced computation is needed in some cases, but it can lead to recomputation of shared subexpressions. We can avoid recomputation of common subexpressions by persisting dataframes that are used in more than one place, before and after a force computation boundary. However, in lazy evaluation frameworks, information about future reuse beyond the current task graph is not available when the computation is forced, so we do not know which dataframes will be reused. Naively persisting every intermediate result just in case it is reused is not practical since it would drastically increase memory footprint and slow down computation.

Therefore, we make use of static analysis to identify useful (live) sub-expressions to be cached when compute is invoked on a

```
import lazyfatpandas.pandas as pd
import matplotlib.pyplot as plt #external module
pd.analyze()
df = pd.read_csv("data.csv")
print(df.head())
df["day"] = df.pickup_datetime.dt.dayofweek
p_per_day = df.groupby(["day"])["passenger_count"].sum()
print (p_per_day)
plt.plot(p_per_day)
plt.savefig("fig.png")
avg_fare = df.fare_amount.mean()
print(f"Average fare: {avg_fare}")
```

Figure 10: Program with External Module Invocation

```
import lazyfatpandas.pandas as pd
import matplotlib.pyplot as plt #external module
from lazyfatpandas.func import print # lazy print
...[# optimized read csv as in Figure 4 ]
print(df.head()) # lazy print
df['day'] = df.pickup_datetime.dt.dayofweek
p_per_day = df.groupby(['day'])['passenger_count'].sum()
print(p_per_day) # lazy print
# call to compute() below forces computation & printing
plt.plot(p_per_day.compute(live_df=[df]))
plt.savefig('fig.png')
avg_fare = df.fare_amount.mean()
print(f'Average fare: {avg_fare}') # lazy print
pd.flush() # Force computation and printing
```

Figure 11: Optimized Version of Program From Figure 10

dataframe. We perform Live DataFrame Analysis (LDA) which is similar to Live Attribute Analysis (discussed earlier in Section 3.1) to identify live (useful) dataframes at each program point. When we force computation, we check which dataframes are live after the point when the computation is forced, and provide the list of such dataframes to the compute method, which persists (caches) any common sub-expressions between the expressions defining the live dataframes, and the expressions in the task graph nodes being computed.

When the task graph is executed, any node marked for persistence has its result persisted on its first execution. In later executions, the persisted result is reused, instead of being recomputed.

In Figure 11, the compute method is called on p_per_day, and an argument named live_df is introduced, and static analysis is used to generate the list of dataframes live after that program point and pass it as the value for the live_df argument in line 11. After line 11, df is the only live dataframe, and it is used later to compute avg_fare. Since df is a shared sub-expression (common node) between both p_per_day and avg_fare, the compute call at line 11 includes the parameter "`live_df = [df]`", which is live and a common subexpression, so it will be cached during computation.

Once all uses of a persisted dataframe have been completed, it can be safely discarded to release memory. Our lazy computation framework discards persisted dataframes after their last use when they are no longer subexpressions of dataframes in live_df list passed to the compute().

3.5 Predicate Push Down

Performing selection operation early in databases, also known as predicate push down, reduces the size of relations and therefore reduces the computation to be performed during other operations like joins. In dataframe systems, filter operations reduce the size of datafames. We identify the filter operations in the task graph, and move them as close to the data source as possible.

Predicate pushdown on the task graph is a standard optimization, which is already performed by lazy backends. However, we implement it on our task graph to benefit non-lazy backends such as Pandas itself, which lack this optimization. Unlike predicate pushdown on relational expression trees, pushdown on task graphs needs to take into account multiple uses of a result, and also needs to take into account the variety of ways in which filtering can be expressed in Pandas, including boolean indexing, loc/iloc, in addition to the filter() method.

Static analysis can allow predicate pushdown through the control flow graph, across task graph computations, similar in spirit to the predicate pushdown performed by MagicPush [19]. This can enable some cases of predicate pushdown beyond points where computation is forced, which cannot be done by predicate pushdown on task graphs. Implementing predicate pushdown as part of static analysis is an area of ongoing work.

3.6 Using and Computing Metadata

Data type information and data statistics are both very important for efficient computation. The widely used csv format does not provide type information or statistics, although formats such as Parquet provide type information, and some statistics. Where metadata is not available, we compute metadata for each source data file. Statistics can be computed from a sample of the values in a collection. To get correct datatypes we have to scan the entire file, although we can do it based on the first few rows or a sample, at some risk. The metadata for a file is computed by running a script on the file, and stored for later use. Information like modified time, column names and types, approximate size of each row, and approximate number of rows in the dataset are currently maintained in the metadata.

Metadata is used during runtime optimization. The modified time metadata is used to ensure that the metadata is up-to-date.

We implement a number of optimizations based on metadata, such as using datatypes to reduce storage overhead when reading data, and replacing a string type by a category type which is based not only on metadata information, but also on static analysis information to ensure that the column is read only. Metadata is particularly important for Dask; for example, apply() in Dask requires the output datatype to be specified. We omit details for the sake of brevity.

4 Related Work

There has been a large body of work on optimized execution of Pandas which we describe in this section. However, to the best extent of our knowledge, none of the earlier systems support optimizations based on static analysis, and are all restricted to run-time optimization. The combination of static and run-time optimization sets our system approach apart from all the earlier work. All of the optimizations described in Section 3, except our current implementations of metadata and predicate pushdown

optimizations, make use of static analysis. As described earlier, we are working on using static analysis to improve these two optimizations as well.

Dask [1] allows dataframes to be partitioned, and operations on dataframes to run in parallel, and to be larger than memory. Moreover, Dask supports a lazy API which allows multiple operations to be collected, and executed only when a compute() function is executed. However, Dask requires changes to Pandas programs to deal with lazy evaluation. Modin [20] supports eager computation, and provides a "drop-in" replacement for Pandas, while also supporting parallel/multi-core execution. Ray [21] is another framework for distributed programs, which supports scalable datasets (dataframes) among other features. Modin uses Ray and Dask as its back-end options.

Pandas on Spark, formerly called Koalas, [3] supports the Pandas DataFrame API on top of Apache Spark; it supports lazy computation in the back-end, while allowing the eager Pandas API to continue to be used at the front end. Unlike Dask it preserves ordering, but like Dask it does not support all of the Pandas API. In contrast, PySpark, which supports access to the native Spark API from Python, does not support the Pandas API.

PyFroid [22] supports a lazy back-end based on the embedded relational database DuckDB, while providing an eager Pandas API with many but not all Pandas functions. PyFroid works only on a single system unlike Dask or Spark. Magpie [4] focuses on pushing Pandas computation to back-end databases on the cloud; internally it used an earlier version of the PyFroid engine. Ibis [7] provides a dataframe interface to data stored in any of a large number of back-end databases, including DuckDB and Polars, but does support many Pandas API functions.

Multiple libraries use lazy evaluation to optimize data science applications, such as Cunctator [23], DelayRepay [24], and Weld [25].

None of the above systems support optimization based on static analysis and rewriting, unlike our system. While several of our run-time optimizations are also implemented by lazy evaluation based frameworks, our wrappers allow these optimizations to be used even with eager backends that do not natively support optimization.

Our earlier short paper [26] briefly describes column selection based on static analysis, but does not cover other optimizations described in this paper such as JIT static analysis, lazy evaluation, or forcing of computation based on static analysis.

Dias [27] optimizes Python notebook based Pandas programs by program rewriting, based on a pattern matcher and rewriter applied to one cell of a notebook at a time, after earlier cells have been executed. However, it does not perform static analysis and cannot benefit from look ahead at later parts of the program, unlike our optimizations. Further Dias can only work with data that (after optimization) fits in memory.

Magicpush [19] uses a program synthesis approach coupled with verification based on symbolic execution to implement predicate pushdown in data science applications. It is limited in its applicability since it does not perform any other optimizations. Our optimizer also performs predicate pushdown on the task graph; pushdown based on static analysis is an area of future work.

5 Performance Evaluation

In this section, we study the benefits of our optimization techniques on a variety of programs, across different dataset sizes. We ran our single-node experiments on a hexa-core AMD Ryzen 5 3600 with base clock at 3.6 GHz with 32GB of DDR4 3200MHz RAM. We use the following library versions for the performance studies: Pandas 2.3.2, Dask 2025.1.0, Modin 0.31.0, Ibis-DuckDB 10.8.0, Ibis-Polars 10.8.0, and Pandas on Spark from PySpark 3.5.1.

5.1 Benchmark Programs and Datasets

To benchmark LaFP, we have taken 10 real workloads from a variety of sources, including programs used in Dias [27], Magpie [4], and MagicPush [19]. These programs execute a variety of operations like data filter, data augmentation using feature addition, data aggregation (like mean,max,sum), data merge and group-by, and informational operations, analyzing data from domains such as movie rating systems, taxi data, startup analysis etc. Each program contained between 5 and 29 operators, with an average of 13 operators per program. We have shared the benchmark programs and datasets at https://github.com/lazyfatpandas/public.

To test the impact of data size, for each program, we created datasets of different sizes by replicating or pruning the original datasets that were available with the programs. The datasets thus created had on-disk (csv file) sizes of close to (within 5% of) 1.4 GB, 4.2 GB, and 12.6 GB respectively. When loaded into memory as a Pandas dataframe the sizes expand by a factor of 3X to 8X depending on the datatypes used in the different datasets. Thus the 12.6 GB on disk datasets ranged from around 35 GB to 100 GB in memory, ensuring the dataset did not fit in-memory.

For each program, we compare the performance with and without our optimizations (including both runtime and rewrite optimizations), using Pandas, Dask, Modin, Pandas on Spark, and DuckDB using Ibis and Polars using Ibis, as the back-ends. For comparison with direct use of the back-end frameworks, we manually transformed the programs to work on each of the backends.

Executing Pandas programs on Modin is straightforward since Modin is designed for Pandas compatibility. Rewriting for Dask was more complicated due to the need for forcing of computation, lack of support for some API methods, and lack of in-place updates. Even though Pandas on Spark uses a lazy backend, rewriting Pandas programs to use it only requires a change to an import statement; unlike Dask, Pandas on Spark hides lazy evaluation from the programmer, with the Pandas on Spark dataframe forcing computation automatically whenever its result is required. Rewriting Pandas programs to run on DuckDB & Polars using Ibis required a significant rewriting effort since forcing of computation is required before using a dataframe result, and many of the Pandas APIs are not supported. To ensure that these programs run successfully using DuckDB and Polars, we commented two API calls (apply() in programs *stu* and *env*, and get_dummies() in program *emp*) which are not supported by Ibis. Support for these in Polars is currently under implementation, while handling DuckDB is harder since it uses an SQL backend. Conversion to Pandas dataframes is an option, but would significantly increase the cost, and be unfair to Polars.

In all other cases, LaFP automatically handles the API differences for whichever backend is chosen. Thus no manual rewriting is required, making it very easy to switch between backends.

Modin can use different execution engines, such as Ray or Dask. We use Ray as the default executor for Modin and LaFP Modin programs. In all cases where a program/dataset combination could not be executed using Ray, we used Dask as executor

Data Size	P	OP	M	OM	D	OD	PoS	OPoS	MDu	ODu	MPo	OPo
1.4GB	10	10	10	10	10	10	10	10	10	10	10	10
4.2GB	10	10	9	9	10	10	9	9	10	10	10	10
12.6GB	2	7	4	7	8	9	9	9	9	9	9	9

Table 12: Number of Programs Successfully Executed

for Modin. For such programs, Dask is used as executor for LaFP Modin as well for uniform comparison.

5.2 Applicability of LaFP

A major goal of our optimizations was to ensure that Pandas programs can run successfully even on large datasets. The first set of experiments therefore checks how many programs could complete successfully using the different frameworks, with and without optimization.

Our framework allows any back-end to be used without any manual program rewrite (barring an import of the lazyfatpandas library, a configuration line to choose the back-end, and a call to analyze()).

Users, however, need to be aware that using Dask, Polars or DuckDB may affect the order of rows in dataframes, which may potentially affect subsequent operations that are order-sensitive. If the program has such order-sensitive operations, the user should not choose these as the back-ends (with or without LaFP).[4] Passing input/output types to apply() function is required in Dask. Adding it automatically is under implementation; we manually added the type for one program that needed it.

Table 12 shows how many of the 10 programs could execute successfully on different back-ends with different data sizes. The original versions and optimized versions with different backends are denoted as: Pandas (P and OP), Modin (M and OM), Dask (D and OD), Pandas on Spark (PoS and OPoS), DuckDB (MDu and ODu), and Polars (MPo and OPo). Note that we use MDu and MPo to denote that the original Pandas program required significant manual rewriting to run on Ibis with DuckDB and Polars as backends, while ODu and OPo were automatically rewritten.

For example, with Pandas and Modin only 2 and 4 programs, respectively, run successfully with 12.6 GB dataset whereas 7 programs could run with our optimizations on Pandas and Modin. The improvements were because our rewrites could reduce space usage, whereas the un-optimized Pandas/ Modin programs ran out of memory. Dask could execute 8 programs whereas optimized Dask could handle 9 programs. The remaining backends, i.e., Pandas on Spark, DuckDB using Ibis and Polars using Ibis, could all execute 9 programs with both the original and optimized versions, since these backends are designed to be scalable. The one program where all backends failed was the 'emp' program; on inspection we found that there was a call to an external plot function which required materializing a large dataframe as a Pandas dataframe, which resulted in out-of-memory error, regardless of the backend used.

We also built a regression test framework to ensure that the datasets computed with our optimizations were identical to the results on Pandas without any optimization, by computing and comparing (order independent) hashes of the dataset results, computed by md5 hash of each row combined using exclusive or; our optimized programs on different platforms all passed these tests.

[4]Automated detection of order-sensitivity is an area of future work.

5.3 Execution Time

We first consider absolute execution times for different backends, with and without optimization, on the 1.4 GB dataset. For this dataset, all the configurations could run successfully. The results are shown in Figure 13.

For programs labeled with '*', as well as overall average, labeled 'Avg*', the programs for DuckDB and Polars are modified as explained earlier, so only relative comparison across the optimized and unoptimzed programs on same platform should be done. On programs that were the same across all platforms, Polars gave the best performance, while DuckDB was second best.

Across all programs, the benefit of optimization for DuckDB and Polars were 18% and 17%, while for Pandas and Modin were 27% and 28%, while Dask and Pandas on Spark gave 62% and 18% respectively. On individual programs, the benefits on different backends varied, with maximum benefits ranging from 49% to 94%, and minimum benefit ranging from -9% to 12%. We note also that the scalable backends all have their own optimizers, yet our optimized versions of the programs gave further improvements.

Next, we consider larger datasets, where Pandas and Modin are not able to run successfully in several cases. Figure 14 shows the execution time improvements using our optimizations, across different back-ends for 4.2 and 12.6 GB datasets. Improvement is defined as:

$$Improvement = \frac{(Original\ Runtime - Optimized\ Runtime)}{Original\ Runtime}$$

In cases where the original program and dataset combination could not execute successfully on the relevant back-end, we treat the original execution time as infinity. This results in 100% improvement in performance provided the optimized program runs successfully. Missing data points in Figure 14, such as for 'emp' on 12.6GB, represent cases where neither the original program nor its optimized version could not be executed using the relevant back-end.

It can be seen that for Pandas and Modin, multiple programs could not be executed on the 12.6 GB dataset, but several of them could be executed successfully using our optimizations in LaFP. For the remaining (scalable) backends, only the emp program failed across all backends, for reasons described earlier in Section 5.2.

Ignoring the cases where the original program did not execute successfully, our optimizations give up to nearly 60% reduction (2.5X speedup) in execution time on Pandas, up to 90% reduction (10X speedup) with Modin, up to 94% reduction (20X speedup) with Dask, up to 70% reduction (3X speedup) with Pandas on Spark, up to 56% reduction (2.3x speedup) with DuckDB and up to 57% reduction (2.3x speedup) with Polars.

We note also that the scalable backends all use lazy evaluation with runtime optimization, yet our optimized versions of the programs gave significant further improvements in many cases.

There were only a few cases where our optimizations increased execution time, the worst case being approximately 15% more time as compared to the original Pandas on Spark program, with DuckDB and Polars also having a few cases which took up to 13% more time. These numbers represent cases where the run-time transformations, which are not cost-based, made wrong choices, adding more overhead than any benefit gained. We are working on reducing the overhead of certain transformations on specific back-ends/optimizations. In most cases the optimizations did make the right choice, but cost-based optimization is an area of future work.

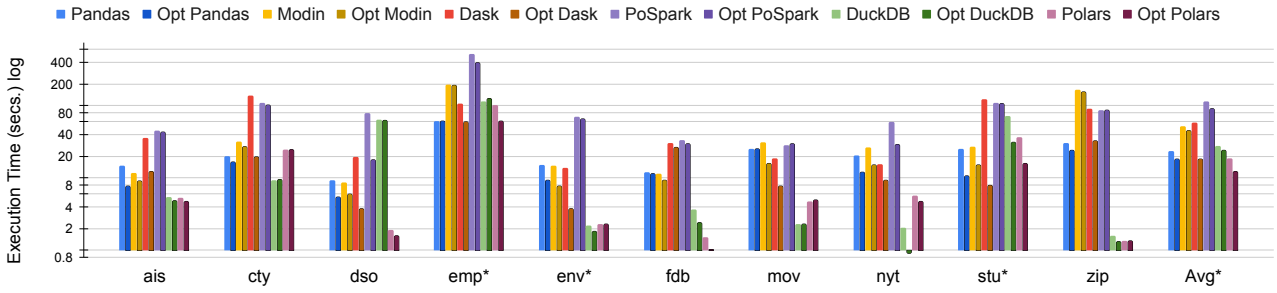
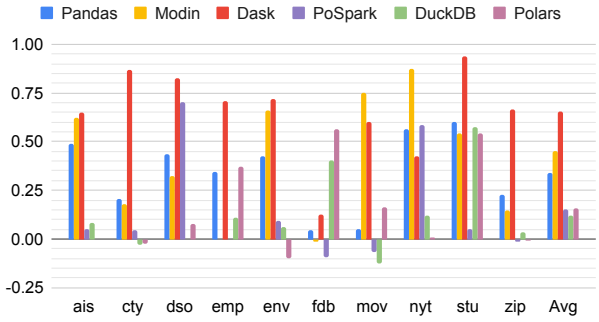
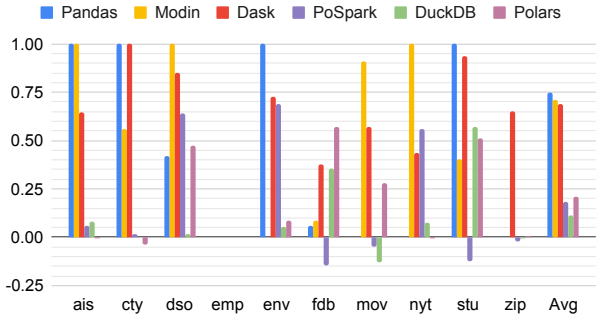


Figure 13: Execution Times on 1.4 GB Dataset



(a) 4.2 GB Dataset



(b) 12.6 GB Dataset

Figure 14: Execution Time Improvement

Program	nyt	dsp	mov	ais	stu	env	fdb	zip	emp	cty
Size(GB)	1.3	2.1	1.1	0.6	0.2	0.2	1.9	2.2	0.3	0.2
Pandas	12.5	2.2	51.6	15.4	50.6	22.4	**29.0**	109.3	311.7	**37.6**
OPandas	3.1	**1.1**	50.6	15.5	49.8	22.9	29.4	**91.6**	332.2	39.0
Dask	2.5	7.5	20.0	20.8	129.7	27.4	60.6	226	167.9	115.8
ODask	**1.9**	1.7	**13.9**	**14.4**	**9.1**	**16.4**	81.8	117.6	**130.6**	42.8

Table 15: Execution Time with Parquet on 4.2GB Data

Parquet Format. In order to validate performance of LaFP with Parquet file format, which provides compression as well as type information, we executed the benchmark programs using parquet format by transforming 4.2GB csv files to Parquet. The time taken (in seconds) is shown in the Table 15.

Compression done by Parquet reduces dataset sizes significantly in many cases, as can be seen in the table. Due to this, data could be loaded faster in the memory. Parquet allows fetching a few columns much faster compared to a csv dataset, and therefore the column selection optimization was more efficient. However, the metadata optimization was not useful since Parquet already maintains metadata. Overall, when using Parquet datasets, our optimizations continued to give significant benefits on Dask as well as on Pandas.

Benchmark	nyt	dso	mov	env	fdb
Dataset(GB)	130	128	132	107	116
Dask Time (secs)	353.2	378.7	355.3	Out of mem.	Out of mem.
ODask Time (secs)	188.4	174.3	277.9	172.8	336.5

Table 16: Performance on Cluster With Large Data

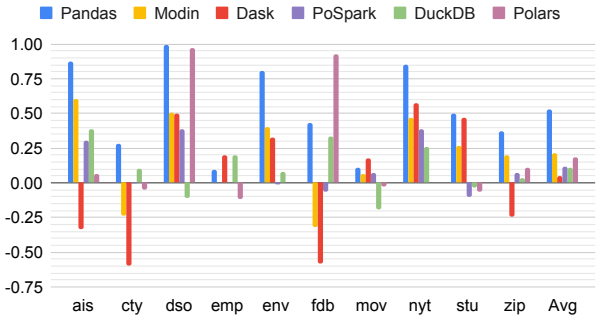


Figure 17: Memory Consumption Reduction (4.2 GB Dataset)

Execution on Cluster. To verify the scalability of our approach, we ran a sample of the benchmark programs on an AWS cluster with 12 nodes (type: m6i.xlarge with 16 GiB memory and 4vCPU) with a total of 192 GiB memory and 48 cores. We use coiled.io to run programs on a Dask cluster. Each program is executed using native Dask and LaFP with Dask. Execution times (in seconds) are shown in Table 16.

Despite Dask supporting out-of-memory dataframes, two of the programs ran out of memory with Dask, but ran successfully with our optimizations which inferred dtype as category for some of the columns, reducing memory usage. It can be seen that even on a cluster, ODask provided significant time and memory benefits over native Dask; LaFP inserted column selection and/or data type optimizations in several of these programs.

Optimization Overhead. We also measured the overhead due to our static analysis and rewriting optimization techniques. The time taken by JIT static analysis phase and rewriting for various programs is in the range of 0.04 sec - 0.59 sec, which is a very small fraction of the execution times of the programs. Collection of metadata is done asynchronously, once per dataset/file, requiring only a relatively inexpensive single scan or sample of the data, and incurs no cost during program execution.

	P	M	D	PoS	Du	Po
Only Static	11%	11%	29%	24%	14%	35%
All Opts	26%	28%	41%	18%	16%	23%
Ratio	42%	39%	70%	133%	88%	152%

Table 18: Impact of Static vs. All Optimizations

Opt.	C+F	CS	LP	ME	PD
ODask	2[1.2-9.9x]	3[1.5-1.9x]	6[1.7-6.5x]	5[1.2-2.2x]	1[1.2x]
OPandas	2[1.2-9.0x]	3[1.5-1.7x]	5[1.8-7.5x]	5[1.1-2.0x]	1[1.2x]

Table 19: Ablation Study: Number of Affected Programs and Regression Factor Ranges

5.4 Maximum Memory Consumption

To get reliable memory size estimates despite multi-threaded execution with Modin and Dask, we created a separate thread that monitored the process memory usage every 200 milliseconds, and we report the maximum memory usage. Figure 17 shows the memory consumption improvements using our optimizations on the 4.2 GB dataset. For lack of space we omit results on the 1.4 GB and 12.6 GB dataset, but note that the results are similar except for the fact that with the 12.6 GB dataset the unoptimized versions of many of the programs could not finish execution as we saw earlier.

Missing values in Figure 17 represent cases where neither the original nor the optimized program could execute successfully.

In case of Pandas, our optimizations significantly reduce memory consumption in most cases, with over 95% reduction in some cases where column selection was particularly helpful. For the programs that can be executed with Modin and Dask, our optimizations reduced memory consumption by up to 60% and 55% respectively, primarily due to column selection (projection pushdown) optimization. For Pandas on Spark and DuckDB using Ibis, reduction in memory consumption is observed in most cases with upto 35% reduction. In case of Polars using Ibis, significant reduction in memory consumption is observed up to 95%.

However, there were some cases where our optimizations increased memory consumption on Dask, with up ot 60% increase in the worst case on 3 programs. These were due to common computation reuse optimization which persists results and reuses them. However, programs *ais* and *cty*, which had increased memory usage, correspondingly had 65% to 85% time improvements.

It may be noted that persisted dataframes are memory-resident not only when using Pandas and Modin, but also with Dask when using Dask's persist() API function, which results in increased memory usage. Persisting Dask dataframes on disk is an area of future work.

5.5 Ablation Studies

We performed an ablation study of the execution time impact of optimizations that we have proposed.

Effect of Static Analysis Based Optimizations: We first consider the impact of optimizations that are based on static analysis, turning off the other optimizations. All of our optimizations, except for Predicate Pushdown (as currently implemented) and Metadata Analysis, depend on static analysis.

The results are shown in Table 18, where the backend names are abbreviated as done earlier in Table 12. It can be seen that for Pandas and Modin, only static gave on average 11% improvement, whereas all optimizations gave 26%and 28% improvement, indicating the benefits of adding lazy computation based optimizations to non-lazy backends. In contrast the benefits of only static optimization are significant across all the lazy backends, ranging from 16% to 35%, with only small or even negative improvements when only runtime optimizations are added. For Pandas on Spark and Polars, only static resulted in better performance than all optimizations, indicating some of our runtime optimizations made wrong choices; cost based choice is an area of future work.

Effect of Individual Optimizations: In order to study the impact of individual optimizations, we disable one optimization at a time and find the increase in cost. We consider the following ablation cases. *C+F*: common computation reuse+forced computation removed, *CS*: column selection removed, *LP*: lazy print removed, *ME*: metadata optimizations removed, *PD*: predicate push down removed.

We run the programs with LaFP with Dask and Pandas as back-ends on the 1.4 GB disk size dataset. The results are shown in Table 19. The cells show the number of affected programs, and the range of increase in costs due to the ablation, compared to the baseline (all optimizations enabled), for these programs.

For example, with ODask there was a sharp slowdown of around 9.9x on disabling common computation reuse on one of the programs, *cty*. We also note that in this case there was also an increase of 2.3X in memory consumption, which is a good trade-off. Similarly, turning off column selection resulted in a slowdown of 1.5X (*ais*) to 1.9X (*dso*).

Disabling lazyprint had an impact of up to 6.5X (*stu*), in programs with multiple print statements. Disabling metadata optimization caused a slowdown of up to 2.2X. Both lazyprint and metadata optimizations improve the performance for *stu*. Similarly, multiple optimizations are applicable for other programs. Predicate pushdown had only a 1.2X impact on one of the benchmark programs we considered, but we have observed much larger benefits on other programs.

6 Conclusion and Future Work

We have described novel optimization techniques for Pandas programs that combine static-analysis based rewriting and lazy runtime frameworks. Our performance study shows the significant benefits of our optimization approach not only compared to Pandas but also compared to other frameworks.

Future work includes the addition of support for more Pandas API calls, direct support for Polars (instead of via Ibis), implementation of automated choice of back-end, support for dataframe attribute reference by position instead of name, completion of read-only attribute analysis to ensure category type can be safely used, and type inference for input/output parameters to be passed to Dask apply() function. Implementation of more optimizations, such as predicate pushdown based on static analysis, is another area of future work.

Acknowledgments

We would like acknowledge the contributions of Utkarsh Shetye and Manish Kumar, who helped implement some parts of the LaFP system, including support for Ibis with DuckDB and Polars as backends, and Sayanti Bhattacharjee and Pranab Kumar Paul, who helped implement some aspects of lazy computation.

Artifacts

Supplementary material including benchmark programs, datasets, and code is available at https://github.com/lazyfatpandas/public.

References

[1] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in science conference*, no. 130-136. Citeseer, 2015.

[2] https://github.com/modin-project/modin, Sep. 2024.

[3] "Pandas api on spark," https://spark.apache.org/docs/latest/api/python, 2024, accessed Sep 2024.

[4] A. Jindal *et al.*, "Magpie: Python at speed and scale using cloud backends," in *Conf. on Innovative Data Systems Research (CIDR)*, 2021.

[5] S. Contributors, "Soot: a java optimization framework," https://www.sable.mcgill.ca/soot/, 2012.

[6] "Duckdb," https://duckdb.org/, 2025, retrieved 2025-10-05.

[7] "Ibis: Python data analysis framework for Hadoop and SQL engines," retrieved 2025-02-27. [Online]. Available: https://github.com/ibis-project/ibis

[8] "Dataframes for the new era," https://pola.rs/, 2025, retrieved 2025-10-05.

[9] "Pylint," https://pypi.org/project/pylint/, 2025, retrieved 2025-10-05.

[10] "Pyflakes," https://pypi.org/project/pyflakes/, 2025, retrieved 2025-10-05.

[11] "Mypy," https://github.com/python/mypy, 2025, retrieved 2025-10-05.

[12] "Prospector - python static analysis," https://prospector.landscape.io/en/master/, 2025, retrieved 2025-10-05.

[13] "Welcome to bandit," https://bandit.readthedocs.io/en/latest/, 2025, retrieved 2025-10-05.

[14] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[15] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 2009.

[16] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," in *Procs. ACM Symp. Theory of Computation (STOC)*, 1972, p. 238–250.

[17] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan, "Extracting equivalent sql from imperative code in database applications," in *Procs. of SIGMOD*, 2016, p. 1781–1796.

[18] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[19] C. Yan, Y. Lin, and Y. He, "Predicate pushdown for data science pipelines," *Proc. ACM Manag. Data*, vol. 1, no. 2, jun 2023.

[20] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2033–2046, Jul. 2020.

[21] "Ray: a unified framework for scaling AI and Python applications," https://github.com/ray-project/ray, 2024, accessed Sep 2024.

[22] V. Emani, A. Floratou, and C. Curino, "Pyfroid: Scaling data analysis on a commodity workstation," in *Extending Database Technology (EDBT)*, 2024.

[23] G. Zhang and X. Shen, "Best-Effort Lazy Evaluation for Python Software Built on APIs," in *ECOOP 2021*, 2021.

[24] J. M. Morton *et al.*, "Delayrepay: Delayed execution for kernel fusion in python," in *ACM SIGPLAN Intl Symp. on Dynamic Languages (DLS)*. ACM, 2020.

[25] S. P. et al., "Weld: A common runtime for high performance data analytics," in *CIDR '17*, 2017.

[26] B. P. Singh, M. Sahu, and S. Sudarshan, "Optimizing data science applications using static analysis," in *Int'l Symp. on Database Prog. Languages (DBPL)*, 2021, p. 23–27.

[27] S. Baziotis, D. Kang, and C. Mendis, "Dias: Dynamic rewriting of Pandas code," *Procs. of ACM on Management of Data (PACMMOD)*, vol. 2, no. 1, Feb. 2024.