Dias: Dynamic Rewriting of Pandas Code

Stefanos Baziotis
University of Illinois (UIUC)
Champaign-Urbana, U.S.A
sb54@illinois.edu

Daniel Kang
University of Illinois (UIUC)
Champaign-Urbana, U.S.A
ddkang@illinois.edu

Charith Mendis
University of Illinois (UIUC)
Champaign-Urbana, U.S.A
charithm@illinois.edu

ABSTRACT

In recent years, dataframe libraries, such as pandas have exploded in popularity. Due to their flexibility, they are increasingly used in *ad-hoc* exploratory data analysis (EDA) workloads. These workloads are diverse, including custom functions which can span libraries or be written in pure Python. The majority of systems available to accelerate EDA workloads focus on bulk-parallel workloads, which contain vastly different computational patterns, typically within a single library. As a result, they can introduce excessive overheads for ad-hoc EDA workloads due to their expensive optimization techniques. Instead, we identify program rewriting as a lightweight technique which can offer substantial speedups while also avoiding slowdowns. We implemented our techniques in Dias, which rewrites notebook cells to be more efficient for ad-hoc EDA workloads. We develop techniques for efficient rewrites in Dias, including dynamic checking of preconditions under which rewrites are correct and just-in-time rewrites for notebook environments. We show that Dias can rewrite individual cells to be 57× faster compared to pandas and 1909× faster compared to optimized systems such as modin. Furthermore, Dias can accelerate whole notebooks by up to 3.6× compared to pandas and 26.4× compared to modin.

1 INTRODUCTION

In recent years, *dataframe*-based libraries, such as pandas, have become increasingly popular with users ranging from social scientists to business analysts [43, 53]. This growth is driven by many reasons, including the flexibility of such libraries, the ability to work within a notebook environment, and the interoperability with other libraries.

Due to the popularity of dataframe libraries, academic and industrial work has focused on improving the scalability of pandas *in the context of bulk-parallel operations*. For example, libraries including modin [40], dask [34], and PySpark [12] focus on parallel or distributed dataframes. Many of these libraries focus on scaling out pandas across multiple servers, as pandas will fail if the dataframe does not fit in main memory.

However, there has been an emerging class of important workloads that operate on a *single* machine, combined with ad-hoc functions. For example, in our conversations with law professors at Stanford University, we have found that provisioning and managing distributed clusters is challenging and time-consuming for social scientists. As a result, much of the work done by such social scientists is done on a single machine. Similarly, Kaggle and Google Colab provide single-machine notebooks for data scientists to explore datasets. Furthermore, many tasks require custom user-defined functions (UDFs) that are not well suited to working directly within the pandas API.

```
for i in range(1, 15):
  lhs = DF_PH.loc[i, 'VendorID']
  rhs = DF_PH.loc[i-1, 'VendorID']
  if lhs == rhs:
    counter += 1
  else:
    counter = 1
  DF_PH.loc[i, 'discourse_nr'] = counter
```

Figure 1: Loop which Accesses Individual Elements (extracted from a Kaggle notebook [7]). This loop can be hundreds of times slower in bulk-parallel frameworks like modin, dask, Koalas and PolaRS, which are not optimized for individual accesses.

While these bulk-parallel dataframe libraries improve the horizontal scalability of dataframes, as we show in this work, they unfortunately can *fail to accelerate a wide range of single-machine, ad-hoc workloads*. For example, modin, dask, and PySpark are all 2-200× slower than pandas for a range of operations: when interfacing with numpy, looping over individual rows, and even for simple operations like multiplying two columns (on a single machine). For example, a simple loop (Figure 1) can be many *hundreds* of times slower (see Section 6.4). Furthermore, all distributed dataframe libraries we are aware of do not maintain full pandas compatibility, requiring domain experts to learn new libraries.

We propose an alternative approach to address the *vertical* scalability of dataframe libraries: *rewriting* notebook cells to accelerate dataframe computations by utilizing faster, but semantically equivalent code sequences. To understand the potential for rewriting notebook cells, consider the two cells in Figure 2. The first cell is a simplified cell from a real-world, Kaggle notebook. The second cell is an optimized cell with identical semantics. While identical semantically, the second cell can run up to 1000× faster, showing that simple rewrites of notebook cells can accelerate workloads.

A natural question that emerges is why can't the users write optimized code themselves. As witnessed in compilers, automatic tools that accelerate code can reduce developer effort and improve comprehensibility. Furthermore, severalrewrite rules in this paper were *not apparent to the authors* (e.g., all the rules in Table 1), even after devoting considerable time studying the internals of Python and pandas. To understand how this translates to non-experts, there are whole videos and articles dedicated to patterns for speeding up pandas code via manual rewriting [8, 10, 13, 20, 46, 49]. Even then, optimizing code correctly is challenging for non-expert users and can lead to subtle bugs.

```
def weighted_rating(x, m=m, C=C):
  v = x['vote_count']
  R = x['vote_average']
  return (v/(v+m) * R) + (m/(m+v) * C)

df.apply(weighted_rating, axis=1)
```

(a) Loop through rows (extracted from a Kaggle notebook [5]). This, effectively, loops sequentially over each row, and the operations are performed in the Python interpreter.

```
def weighted_rating(x, m=m, C=C):
  v = x['vote_count']
  R = x['vote_average']
  return (v/(v+m) * R) + (m/(m+v) * C)

# Pass the whole `df` directly.
weighted_rating(df)
```

(b) The function contains only column operations and thus can be applied directly to the whole `DataFrame`.

Figure 2: A rewrite example where we avoid `apply()`. The rewritten version, which uses vectorized, native execution, can run up to 1000× faster.

To realize the vision of rewriting notebook cells transparently, we propose Dias, a library that automatically rewrites notebook cells via cell annotations. As we show, Dias can accelerate notebook cells by up to 57× completely transparently to the user.

In order to rewrite cells, Dias must overcome several challenges. First, it must operate within interactive time scales: the overhead of rewriting cannot dominate cost savings. Second, the rewrites must not change program semantics. This is particularly challenging in the context of a dynamically-typed language like Python, which has no precise formal semantics and liberal typing rules. In Python, the types of variables or even classes of *other* modules can change arbitrarily and there are no standard scoping rules.

We designed Dias' rewrite engine with two components: a pattern matcher and a rewriter with design decisions that specifically address the aforementioned challenges. The rewrite engine is lightweight, with a fast pattern matcher that can quickly match patterns that we can rewrite into faster versions and a rewriter which emits, or performs, necessary static and runtime precondition checks to guarantee correctness, within interactive latencies.

We show that on real-world Kaggle notebooks, Dias can accelerate cells by up to 57× (1.18× geometric mean) and whole notebooks by up to 3.6× (1.29× geometric mean). We also compare Dias with `modin` and show that it can be up to 26.4× faster for whole notebooks (4.1× geometric mean). Furthermore, Dias can avoid rewriting cells that cause slowdowns, resulting in overheads that are only due to the pattern matcher. We show that these overheads, even in the cases where cells are not rewritten, are below noise thresholds. Finally, Dias uses no extra memory or disk capacity.

In summary, we make the following contributions.

(1) We identify program rewriting as a lightweight technique to speed up `pandas`-heavy EDA workloads. We introduce rewrite rules that can significantly speed up `pandas` code, including non-trivial ones that cross library boundaries.
(2) We develop Dias to apply these rewrite rules automatically, at runtime. Dias verifies whether applying a rule is correct by either injecting checks in the code or by slicing the execution and performing checks in between.
(3) We evaluate Dias on real-world notebooks and show that it can speed up cells by up to 57× and notebooks by up to 3.6×, with almost no memory or disk overheads. We further compare Dias with `modin` [40] and show that it can be up to 26.4× faster for whole notebooks (4.1× geometric mean).

2 BACKGROUND

2.1 Setting

In this work, we focus on the broad class of workloads commonly referred to as *exploratory data analytics* (EDA) [42]. In EDA workloads, the data is iteratively analyzed for interesting patterns. Since the patterns of interest are unknown ahead of time, much of this work is done interactively, in a notebook environment (e.g., Jupyter notebooks, other REPLs) using a dataframe library. We focus on `pandas` and related libraries in this work.

In one common setting, analysts are interested in analyzing large datasets, which typically do not fit in main memory on a single server. In order to accelerate these workloads, much effort from both industry and academia has gone towards accelerating *bulk-parallel* workloads.

Frameworks including `modin` [40] and `dask` [34] aim to accelerate such workloads. They operate by providing APIs close to the standard `pandas` API, distributing data across servers, and evaluating functions lazily. When working within these libraries, they can accelerate workloads by up to 100× [40].

Unfortunately, these bulk-parallel libraries have several drawbacks. In particular, these libraries were not designed for *ad-hoc, single-machine* workloads.

Ad-hoc operations. The primary drawback of these libraries is that they have poor support for ad-hoc operations outside of the library API. For example, operations such as looping over rows, column-wise operations that require intermediate materialization for inspection (e.g., comparing a column to a constant), or inspecting the first n rows can be 30-1900× *slower* than standard `pandas` on a single machine. For example, as we explain in Section 6.4, a simple loop, shown in Figure 1, can be many *hundreds* of times slower.

Single-machine overheads. In addition to slowdowns for ad-hoc operations, these libraries can add substantial memory overheads. We selected 20 random EDA notebooks from Kaggle (under criteria described in Section 6.1), which had heavy `pandas` usage. `modin` generally increased memory usage, with the peak memory usage being up to 127× higher than native `pandas`. The peak memory usage increased 4.7× on average (geometric mean).

```python
pd.Series(df['A'].tolist() + df['B'].tolist())
```

(a) Original: Concatenate `Series` by first turning them into lists. Extracted from a Kaggle notebook [50].

```python
pd.concat([df['A'], df['B']], ignore_index=True)
```

(b) Rewritten: Use a `pandas`-provided function for concatenation

Figure 3: Rewrite example that crosses library boundaries, and thus cannot be performed by previous techniques. The rewritten version can be up to 11× faster.

Usability. In discussions with social scientists and law professors at Stanford University and the University of California, Berkeley, we have found that learning new APIs is challenging and time-consuming. In particular, these bulk-parallel libraries are not direct drop-in replacements. To show this, we sampled 20 notebooks from Kaggle at random (under criteria described in Section 6.1). Five of these notebooks (25%) were unable to run when `pandas` was replaced with `modin`.

Furthermore, setting up distributed clusters can be difficult in these settings. As a result, the distributed speedups are difficult to realize in the settings we focus on.

In this paper, we introduce program rewriting as an automatic optimization technique for Python code that interfaces with `pandas`, focusing on accelerating single-server, ad-hoc EDA workloads.

2.2 Rewriting as an alternative optimization

Rewriting, for optimization purposes, is the process of replacing some part of code with a functionally equivalent but faster version. Rewriting avoids the previously mentioned drawbacks of library-based optimization systems. First, it inherently does not suffer from a lack of API support because it is not a replacement for `pandas` and it can leave the code untouched if it cannot handle it. Second, rewriting is a lightweight technique incurring minimal overheads, which scale proportionally only to the code, not the data.

Additionally, there are fundamental advantages Dias has over library-based optimization approaches. The rewrite system is transparent. When the user observes a speedup, they can always see the code that the rewriter used. In other words, the user does not need to understand the system to understand the cause of the speedup. At the same time, the user's code remains intact. Further, rewriting has the benefit of being able to optimize across library boundaries. For example, Dias can automatically perform the rewrite in Figure 3 (taken from a real-world notebook). The original code crosses the library boundaries (twice!) as we move from `pandas` to Python (by converting to a list) and then back to `pandas`. To perform this rewrite, a tool needs to view all the code and understand semantic equivalences and differences across library boundaries (e.g., `pandas` and the host language, Python). This is not possible with optimization approaches that purely aim at accelerating the `pandas` API.

```python
df[['a', 'b']] = df['C'].str.split('(', expand=True)
```

(a) Splitting a `pandas.Series` using `pandas.Series.str.split()`. Extracted from a Kaggle notebook [1].

```python
a = []
b = []
ls = df['C'].tolist()
for it in ls:
    spl = it.split('(', 1)
    a.append(spl[0])
    b.append(spl[1] if len(spl) > 1 else None)
df['a'] = pd.Series(a, df['C'].index)
df['b'] = pd.Series(b, df['C'].index)
```

(b) Splitting a `pandas.Series` in pure Python

Figure 4: Splitting in pandas and Python. Surprisingly, the pure Python implementation is up to 7× faster.

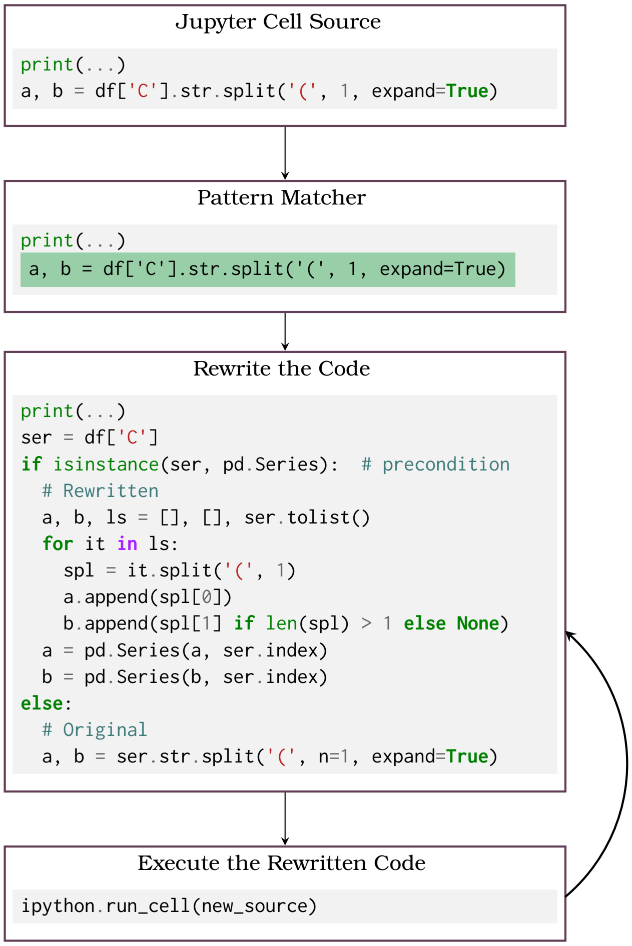
Rewriting appears simple, but it can be challenging when performed manually. There are many non-obvious rewrites that the user may not be able to discover easily. For example, it might seem that the only way to make `pandas` code faster through rewriting is by replacing it with other `pandas` code, or using a similar library such as `numpy`. This has been reinforced over years of data scientists being trained to remain within `pandas`/`numpy` as much as possible because these use native, vectorized implementations and are thus deemed to be much faster than pure Python. It might, then, be surprising that moving out of `pandas` and into pure Python can lead to significant speedups. One example is shown in Figure 4. The task here is to split a `Series` of strings by the delimiter `'('`. The code in Figure 4a (extracted from a Kaggle notebook) does it by using a `pandas`-provided function. One would expect that this is the best way to perform this operation. Nevertheless, the version in Figure 4b is 3.5× faster. It moves from `pandas` to pure Python (by converting `df['C']` to a Python list) and performs the operation with a sequential Python loop (in our case studies in Section 6.5, we explain why this version is faster).

It is unreasonable to expect general `pandas` users to comprehend Python, `pandas`, and `numpy` to such an extensive level to be able to discover such equivalent versions and evaluate their relative performance. Second, even if the user succeeds in these tasks, the rewritten version can be significantly harder to write and read, as is evident from Figure 4. This can further lead to correctness concerns about the rewrite. Third, manual rewriting breaks the library abstraction. In the original code of Figure 4, the user has to think only of *what* `split()` does. But, to come up with the rewritten version, this abstraction's veil has to be removed as the user needs to think of *how* to implement it.

These issues motivated us to build Dias, a system that performs such rewrites *automatically*, by guaranteeing *correctness* and with minimal overhead. Section 3 provides an overview of Dias.

3 DIAS OVERVIEW

We now present the high-level architecture of Dias, a rewrite engine that automatically applies rewrite rules to improve the performance of ad-hoc EDA workloads.



Jupyter Cell Source

```
print(...)
a, b = df['C'].str.split('(', 1, expand=True)
```

Pattern Matcher

```
print(...)
a, b = df['C'].str.split('(', 1, expand=True)
```

Rewrite the Code

```
print(...)
ser = df['C']
if isinstance(ser, pd.Series):  # precondition
  # Rewritten
  a, b, ls = [], [], ser.tolist()
  for it in ls:
    spl = it.split('(', 1)
    a.append(spl[0])
    b.append(spl[1] if len(spl) > 1 else None)
  a = pd.Series(a, ser.index)
  b = pd.Series(b, ser.index)
else:
  # Original
  a, b = ser.str.split('(', n=1, expand=True)
```

Execute the Rewritten Code

```
ipython.run_cell(new_source)
```

Figure 5: Dias overview. Dias identifies patterns in the source code, which it rewrites using its rewriter. The rewriting is not always valid. Dias preserves the original semantics by inserting checks in the code (shown here), or by slicing the execution and performing checks in between.

We designed Dias with two high-level components. First, Dias' *syntactic pattern matcher* matches the input code against the left-hand side (LHS) parts of the rewrite rules and validates the syntactic preconditions. The second component is a *rewriter*, which checks the runtime preconditions of the rewrite rules and on success, rewrites the code to the right-hand side (RHS) version and executes it. We show a high-level overview in Figure 5.

We have several desiderata for Dias: it should facilitate applying complex rewrites automatically with minimal overhead. Further, it should guarantee that the rewritten code is semantically equivalent to the original code. There are two main challenges in achieving these goals.

Preserving Correctness of the Rewrites. Dias needs to emit dynamic checks to ascertain whether runtime preconditions are preserved. Some of these checks are quite involved, for example, checking the form of whole functions. Further, certain checks concerning a statement can only be performed after all the code up to that statement has been executed. For these reasons, the rewriter should be a rewriter, an executor, and a dynamic checker.

```
df['A'].sort_values().head(n=5)
```

(a) Select the 5 smallest elements by sorting first. Extracted from a Kaggle notebook [35].

```
df['A'].nsmallest(n=5)
```

(b) Select the 5 smallest elements directly. This avoids sorting.

Figure 6: Selecting the 5 smallest elements. By comprehending the pandas API, Dias is able to recognize that the second version is equivalent to, and faster than, the first.

```
@{expr: called_on}
  .sort_values()
  .head(n=@{Constant(int): first_n}) ↦
@{called_on}.nsmallest(n=@{first_n})
```

(a) LHS ↦ RHS

```
type(@{called_on}) == pandas.DataFrame
```

(b) Preconditions

Figure 7: A rewrite rule example. If we match the LHS in the source code, we can replace it with the RHS only if the preconditions hold (at runtime).

Tight Latency Budget. EDA notebooks are created incrementally, where the user is in a continuous write-execute-inspect loop. Thus, we cannot optimize the notebook offline because we do not have the code. This demands that the system operate at *runtime*, which enforces a tight latency budget. Ideally, rewrites should be applied within interactive speeds (i.e., under 300ms).

In the subsequent sections, we describe how we designed the pattern matcher (Section 4.1) and the rewriter (Section 4.2) to overcome these challenges. First, we introduce the structure of the rewrite rules briefly in Section 3.1.

3.1 Pandas Rewrite Rules

The abstract form of the rewrite rules Dias supports can be modeled as transforming a Left Hand Side (LHS) set of statements to Right Hand Side (RHS) set of statements subject to certain preconditions on the LHS. We introduce some notation to show the structure of our parameterized rewrite rules. The parameterized portions of the rewrite rules are general and can match multiple valid code segments subject to certain conditions (e.g. types). For example consider the original code in Figure 6(a) rewritten to Figure 6(b) using the rewrite rule shown in Figure 7. A @{. . .} entry denotes a parameterized part of the rule. These parts can be matched to multiple valid options by Dias. Inside the curly brackets, we describe

LHS	RHS	Preconditions

`@{Name: df}[[@{Constant(str): a}, @{Constant(str): b}]]=`
`  @{expr: ser}.str.split(`
`    @{Constant(str): sep},`
`      expand=@{Constant(bool): expand})`
	`a, b = [], []`
	`for it in @{ser}.tolist():`
	`    spl = it.split(@{sep})`
	`    a.append(spl[0])`
	`    y = spl[1] if len(spl) > 1 \`
	`          else None`
	`    b.append(y)`
	`@{df}[@{a}] = pandas.Series(a, @{ser}.index)`
	`@{df}[@{b}] = pandas.Series(b, @{ser}.index)`
		`𝔖: @{expand} == True`
		`ℜ: type(@{ser}) == pandas.Series`

`@{expr: ser}.apply(`
`  lambda @{Name: par1}:`
`    @{Constant(str): needle}`
`      in @{Name: par2})`
	`res = @{ser}.tolist()`
	`res = [(@{needle} in s) for s in res]`
	`pandas.Series(res, @{ser}.index)`
		`𝔖: @{par1} == @{par2}`
		`ℜ: type(@{ser}) == pandas.Series`

`pd.Series(`
`  @{expr: e1}.tolist() +`
`  @{expr: e2}.tolist())`
	`pd.concat([@{e1}, @{e2}],`
	`  ignore_index=True)`
		`ℜ: pd == pandas`
		`ℜ: type(@{e1}) == pandas.Series`
		`ℜ: type(@{e2}) == pandas.Series`

Table 1: Examples of Rewrite Rules. If any of the LHS's is matched, it can be replaced with the corresponding RHS, provided that the preconditions hold. The symbol 𝔖 denotes syntactic preconditions while ℜ denotes runtime ones.

these valid options using a derivation rule of the Python grammar [2]. For example, `@{expr}` denotes that any expression can appear in its place. For `Constants`, we optionally specify the type of the constant inside parentheses.[1] So, `@{Constant(int)}` denotes that any integer constant can appear in its place. We need to refer to the parts of the LHS that are parameterized in the preconditions and the RHS. So, we bind these parts to names. For example, the code string `df.sort_values().head()` matches the LHS of Figure 7 and `called_on` is bound to `df`. Everything that is not in `@{}` should appear as is. With these in mind, we can read the LHS of Figure 7 as matching any Python expression on which `sort_values()` is applied, followed by `head()` with any constant integer as the argument of the formal parameter `n`.

There are two kinds of preconditions, syntactic and runtime ones. Syntactic preconditions describe conditions related to the matched *text*. Usually, they require that two matched entries of the LHS are *syntactically* equal. The runtime preconditions describe conditions which have to hold at *runtime* for the original (LHS) and the rewritten code (RHS) to be semantically equivalent and they are expressed in Python syntax and semantics. For example, in Figure 7, the result of the `called_on` expression that was matched in the LHS should be a `pandas.DataFrame`. The runtime preconditions implicitly impose an order of evaluation. In this example, `called_on` must be evaluated first, then the preconditions are checked on the resulting object, and then this object is used in place of `called_on` in the RHS. Note that unconditionally evaluating `called_on` is correct even if the conditions do not hold because it would be evaluated anyway in the original.

Table 1 shows three more rewrite rules we use in Dias. The first two correspond to the examples in Figure 4 and Figure 3, respectively. Rules can have both runtime and syntactic conditions. For example, in the second rule, we have the syntactic precondition `@{par1} == @{par2}` requires that the two names be equal. To differentiate between the two kinds of preconditions, we prefix the syntactic preconditions with 𝔖 and the runtime ones with ℜ.

Listing 1 A Sketch of Dias' Pattern Matcher

```
function LambdaSubstrSearch(stmt)
    // Return True if stmt is a ast.Lambda
    // that performs substring search.
end function
function PatternMatch(stmt)
    for all node in stmt do
        if node is ast.Call then
            if node.func.attr = "apply" then
                arg0 ← node.func.args[0]
                if LambdaSubstrSearch(arg0) then
                    return SubstringSearchApply
                end if
                if isinstance(arg0, ast.Name) then
                    if node.func.args[1] is axis=1 then
                        return ApplyAxis1
                    end if
                end if
            end if
        end if
    end for
    return None
end function
```

4 DIAS REWRITE SYSTEM

Dias consists of two main parts: a syntactic pattern matcher and a rewriter that rewrites the code matched against patterns. We now describe how the two parts were designed in detail.

[1] We can determine the type of constants from the AST [3].

4.1 Dias Pattern Matcher

The pattern matcher is responsible for matching a sequence of statements with the LHS part of any rewrite rule. This is reminiscent of regular expressions, but the language we match is not regular. So, instead, we parse the code as a Python abstract syntax tree (AST) [2] and do pattern matching at the AST level.

To minimize matching overhead, we designed the pattern matcher to factor patterns based on their commonalities. The common parts are matched first before hierarchically matching more specific components of a rule. This eliminates repeatedly matching components that are common to multiple rules.

Consider the pattern-matching code that matches two patterns: the third pattern of Table 1 and the one that enables the rewrite of Figure 2. The LHS of the former is shown in the table. The LHS of the latter is `@{expr: e}.apply(@{Name: fun}, axis=1)` (see Section 4.2). Notice that these LHS's share parts; they both require a function call, that is an attribute of some expression and the name of the function is `apply`. We want to check the common parts of the pattern at a single place to exploit commonalities across patterns. Listing 1 shows a sketch of the pattern-matching code that matches these two patterns. It recursively loops through all the AST nodes of type `stmt` and checks for the two patterns by first checking for an attribute function called `apply` and then matching against either one of the patterns specifically.

Lastly, the pattern matcher needs to be able to match patterns that span multiple statements. Having a function that matches single-statement patterns (like the one in Listing 1), there is another function that matches multiple statements. The latter function operates on a higher level, viewing multi-statement patterns as sets of smaller ones. So, for a 2-statement pattern, if it matches the first part, it will then checks the next statement for the second part.

4.2 Dias Rewriter

When a piece of code is successfully matched with a rewrite rule's LHS, if there are no preconditions, the rewriter can emit the RHS code in place of the LHS and execute it. This is simple and it is done with a series of AST transformations. However, checking preconditions is challenging because the rewriter must check them at points, during the execution, when it is correct to do so.

To check the preconditions, the rewriter needs to derive facts about the execution of the Python program (e.g., the type of an object at a particular point). Ideally, we would derive such facts from static program analysis. However, since Python is dynamically typed, we cannot statically determine how a Python program will behave when executed. Further, Python does not have a complete description of its formal semantics (e.g., [18] is not complete). Thus, even advanced program analyses over Python are limited [16].

The natural alternative is to derive these facts dynamically during execution. In particular, we are interested in knowing the type of Python objects. Python programs can introspect their own objects and so the simplest way to perform a check is to include the code that performs the check as part of the rewritten code. This is conceptually simple and it fits simple checks (like the type-related ones). This is what we do in Figure 5. There, we insert code that checks the preconditions. We also insert the RHS and the program will branch to it if the preconditions are satisfied. Otherwise, we branch to the original code. However, notice that this style of a check requires that we know the (concrete version of the) RHS a priori. This is not the case for all rewrite rules.

Sliced Execution. Let us consider a more demanding rule; that of Figure 2. For this rewrite to be correct, the function passed to `apply()` must have a certain form. Then, a rewrite rule for this rewrite could have an LHS that matches a function definition with that form, followed by a call (on an `expr`) to `apply()` with the name of the defined function as the first argument. Such a rule would have a subtle benefit. Observe that when we perform this rewrite, we rewrite the function passed to `apply()`, which means we need to *know* the code of the function at the time of rewriting. Because the function body is matched as part of the LHS, we do know it.

However, this rule is weak. More specifically, the function definition may not appear in the same cell or even notebook (it might be part of an external library). One way to strengthen the rule is to relax the LHS onto just `@{expr: df}.apply(@{Name: fun}, axis=1)` (so, the function definition is not part of the LHS and the first argument can be any `Name`) and add the condition that the function should follow a certain form to the runtime conditions.[2] The rewriter now needs to insert the code that checks the form of the function as part of the rewritten code, similar to checking the type of an object.

But what is the RHS that we emit? Remember that to create the rewritten version we need the code of the function. Thus, we end up having to also emit the code that does the rewriting as part of the rewritten code! On confronting this situation, we searched for an alternative solution as that one would result in huge code duplication and unintelligible generated code [3].

This gave rise to sliced execution. In this mode, we execute the code up to, but not including, the call to `apply`. Then, stop and inspect the code of the passed function and check the preconditions. Upon passing, we rewrite the code on the fly, and then we execute it. Essentially, we have two forms of rewrite rules: The regular and the deferred ones. For regular rewrite rules, all the checks can happen either statically or they are simple enough that we bake them in the rewritten code. On the other hand, for the deferred ones, we need to have executed all the code up to some statement involved in the rule to perform the necessary dynamic checks and rewriting. For this reason, we may need to do many rounds of check-rewrite-execute, which effectively means that we split the cell into slices. We signify this flow with the back-edge in Figure 5.

5 IMPLEMENTATION

5.1 IPython Integration

Dias is built on top of IPython [52], which is an enhanced Python interpreter. This implies that a current limitation of our implementation is that does not work with standard Python. In practice, this is not a problem because the dominant platform for the notebooks we target is the IPython notebook (usually accessed through Jupyter [38]), which requires IPython.

[2] A function is an object in Python, that we can inspect and recover its source code. So, checking this condition at runtime is possible.

[3] Note, however, that this solution is possible.

An IPython notebook consists of a collection of code snippets called *cells*. Each cell can be executed individually, which is commonly done in interactive EDA workloads. The core feature of IPython that allows Dias to operate transparently is the magic function [47]. Dias leverages magic functions to invoke its rewriter upon cell execution. Unfortunately, the user needs to add a small annotation on top of every cell such that Dias is invoked on every execution. We hope to remove this in a future version.

An important detail is that Dias runs on the *same* IPython instance as the notebook, having access to the same namespace as the underlying cells. This is necessary for sliced execution, because Dias needs to inspect the names, types, objects, etc. of the program.

Upon running a cell, Dias gets a single argument, which is the cell code as a string, which it first parses as an AST. For that, we use the Python `ast` library [2], which parses Python code. This implies a limitation because cells can contain invalid Python syntax (but valid for IPython, e.g., other magic functions), which this library will not handle. This did not cause serious problems in practice but we hope to fix in the future. After the code is parsed as an AST, we just match patterns on it using the pattern matcher, and rewrite/execute it using the rewriter.

5.2 Crossing Library Boundaries

It might seem that we could achieve the same optimizations simply by modifying the `pandas` library. For example, consider the problem we described in Section 4.2, where we have to use complex logic to get the code of the function passed to `apply`. If instead of operating as an external tool, we modified the internals of `apply`, much of this complexity would vanish. Specifically, `apply` gets the function to be called as its first parameter. So, Dias could check the code when `apply` is called, possibly rewrite it and call it, all that without changing the interface.

The reason we implemented Dias as an external tool is it needs to view *all* the user code (and not just the calls to the library) to perform rewrites that cross the library boundaries (e.g., Figure 3). Currently, we only explore the direction of going from `pandas` to Python, which leads to surprising, multiple-fold speedups. The inverse i.e., going from Python to `pandas`, still leads to speedups, and it is something we hope to explore in the future using rewriting.

6 EVALUATION

6.1 Experimental Setup

All the experiments, except if mentioned otherwise, were performed on a system with a 12-core AMD Ryzen 5900X, 32GB of main memory, Samsung 980 PRO NVMe SSD and Ubuntu 22.04.1 LTS.

Benchmark. Our goal was to evaluate Dias on real workloads and so we picked notebooks from Kaggle. We chose Kaggle as it is a popular repository for data science workloads and it also contains both the data and notebooks used. The overarching hypothesis that we want to validate is that a rewrite system like Dias can offer substantial speedups on real-world notebooks, through rewriting, with minimal slowdowns, minimal memory consumption and disk usage, and without changing the API.

In this work, we focus on ad-hoc EDA, `pandas`-heavy workloads. In order to find such notebooks, we chose notebooks randomly from Kaggle subject to the following conditions:

- At least 50% of static function calls are `pandas` calls
- Using datasets of size approximately 2GB or less

We chose the first criterion because we focus on EDA notebooks. In particular, many of the notebooks we excluded focused on machine learning and plotting, which are out of scope for this work. In the notebooks we picked, we disabled such code for our evaluation.

Our second criterion was to filter out notebooks that were already hand-optimized. These notebooks typically operated on large datasets. Optimization is necessary in this setting as Kaggle has resource constraints (both computational and memory). However, hand-optimization requires significant effort. Dias is an automatic and transparent system and we want to evaluate its effectiveness without users having to expend that effort.

For the datasets that were significantly lower than 2GB, we replicated them so that they reach at least several hundred MBs (otherwise our measurements would be dominated by noise). Also, we modified any notebook that used a sample/subset of the dataset to instead operate on the full dataset.

We sampled 20 notebooks satisfying our criteria. There are rewrite opportunities in 10 of these 20 notebooks, which we coded in Dias. We focus on these 10 notebooks in our evaluation. We further executed Dias on the remainder of the notebooks where no patterns were matched to study Dias' overhead. We describe further experiments that include all 20 notebooks in an extended version of this manuscript [6]. We compare Dias with `pandas` (version 1.5.1) and `modin` [40] (version 0.17.0).

6.2 End-to-End vs Pandas

We first investigated whether Dias can accelerate cells and notebooks compared to standard `pandas`. To do so, we ran each sampled notebook with and without Dias. We ran 10 trials each and measured execution time at the cell level. Our primary metric was the speedup of cells and notebooks with Dias compared to standard `pandas`. We report the geometric mean of the speedups.

Per-Notebook Speedups. We show per-notebook relative speedups in Figure 9. As shown, Dias can provide substantial speedups *at the notebook level* of up to 3.6×. Overall, Dias provides significant speedups in half of the notebooks (five) and moderate speedups in one other notebook. We emphasize that these notebooks were selected randomly from Kaggle, showing the applicability of Dias.

Furthermore, Dias does not significantly slow down any notebook, with a maximum slowdown of 3%. Dias rewrites cells in these notebooks but it does not achieve speedups.

Per-Cell Speedups. We show per-cell speedups in Figure 8. For clarity, we excluded cells that run for fewer than 50ms in the original version and excluded all speedups and slowdowns when run with Dias within 10% of the original cell runtime.

As shown, Dias can achieve per-cell speedups of up to 57×. The cell with the highest speedup is matched by the pattern shown in Figure 2. The second largest speedup is due to the "Vectorized Conditionals" pattern discussed in Section 6.5. The majority of cells we consider are improved by Dias. The maximum slowdown in

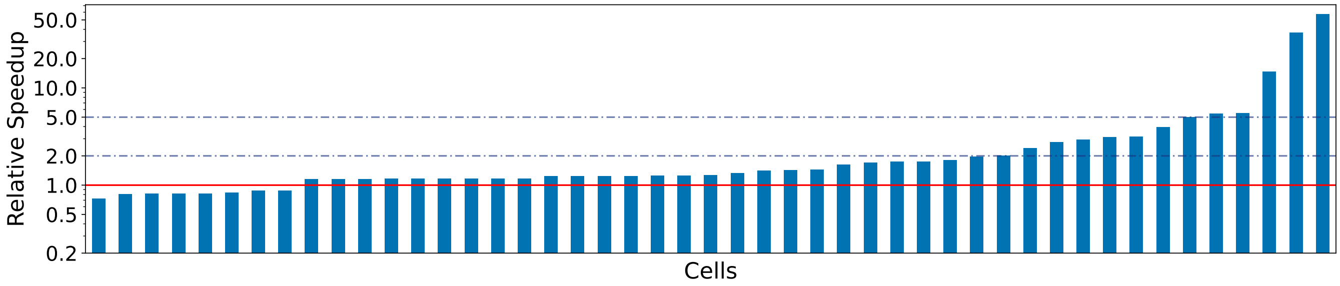


Figure 8: Cell-level relative speedups (excluding cells that originally ran for less than 50ms and also all the cells that got a speedup or slowdown within the 10% range). Again, Dias provides significant speedups by up to 57×. There are also slowdowns, which are not substantial (see Figure 10).

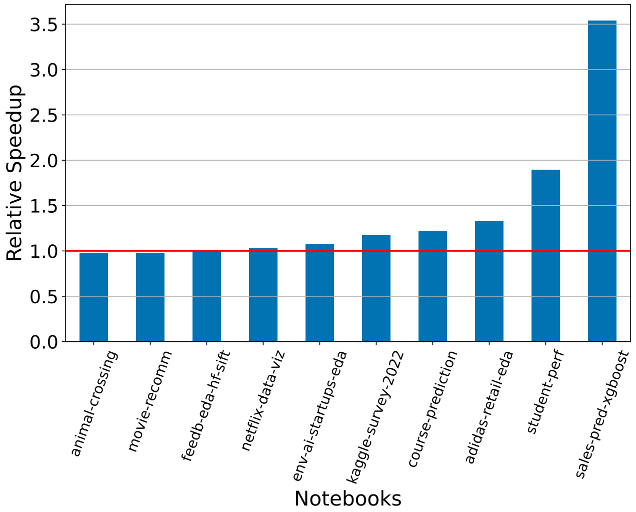


Figure 9: Relative speedups on whole notebooks. Dias speeds up notebooks by up to 3.6× while not slowing down any notebook by more than 3%.

an individual cell is 28%. In general, the cells that have the highest slowdown are fast cells, i.e., those already within interactive latencies, both before and after rewriting.

Overhead of Dias. We further investigated the cause of slowdowns. We first measured the overhead of deploying Dias (on all 20 notebooks). We find that Dias never has an overhead of more than *23 ms* with a geometric mean overhead of 0.99ms.

However, in addition to the overhead from deploying Dias, Dias may also cause downstream effects. We find that in some cases, cells that are not modified by Dias can experience degradations in performance. The highest magnitude of those appear only in notebooks where Dias rewrites cells. Because of this, and because some of these slowdowns are much larger than any overhead that Dias can cause, we hypothesize that rewriting is not the cause of the slowdown. Rather, it seems that the rewritten version of a cell, while faster than the original version of this same cell, causes a slowdown in another cell of the same notebook. Nonetheless, these slowdowns are not substantial. In Figure 10 we show only the cells from Figure 8 that get slowdowns along with the absolute slowdown. That figure shows that even when the relative slowdown is large, the absolute slowdown is below interactive latency times (i.e., below 300ms).

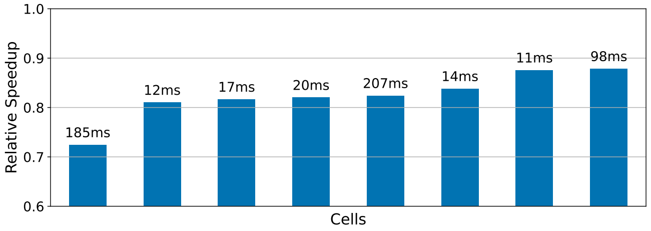


Figure 10: The subset of cells from Figure 8 that got slowed down. Above the bars, we show the *absolute* slowdown. The slowdowns are within interactive latencies (i.e., less than 300ms), with the maximum overhead of Dias being 23ms.

6.3 Comparison with Modin

We compare Dias with `modin` [40] (using Ray as the underlying engine which is the default). We chose `modin` because it enjoys wide adoption and is supposed to be a drop-in replacement for `pandas`.

We focus on deploying `modin` on a single server as this is the setting we focus on in this work. Unfortunately, we find that deploying `modin` in this setting is difficult for two reasons: excess memory utilization and lack of support for the full `pandas` API.

For the notebooks we consider, `modin` consumes substantially more memory resources than standard `pandas`. Even when using a powerful AWS server, the AWS `c5.24xlarge` with 96 vCPUs and 192 GB of RAM, `modin` was unable to execute five of the ten notebooks we consider. As a result, we modify the default `modin` settings to execute on 4 to 12 cores depending on the notebook and we also had to reduce the dataset replication on 3 of the 10 notebooks. With these modifications, we are able to run the notebooks with `modin`, using our original setup.

We further find that `modin` does not support 100% of the `pandas` API. As a result, we could not run two of the ten notebooks. We changed the impeding snippets to ones that are functionally close. Given our new setup, we compared `modin`, Dias, and vanilla `pandas`.

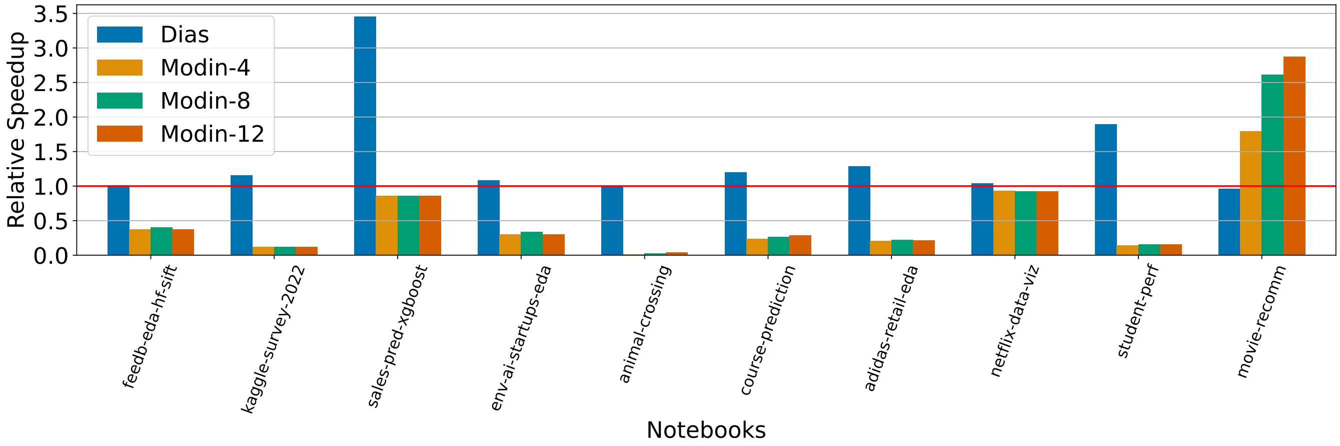


Figure 11: Comparing Dias with modin [40]. Dias is faster for 9 out of 10 notebooks (Up to 26.4× faster with 4.1× geometric mean). modin is, in many cases significantly, slower than the original for these 9 notebooks. For the one notebook where Dias is slower, it is no more slower than 3% compared to the original.

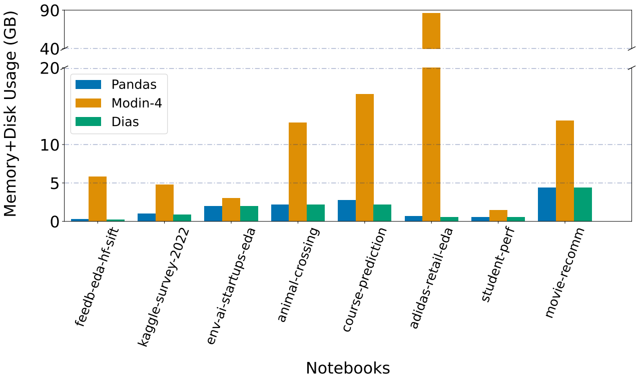


Figure 12: RAM and disk usage comparison in modin, Dias and pandas. Dias and pandas do not use the disk and they use almost the same amount of RAM in all cases. modin uses the RAM and disk aggressively, surpassing the 80GB threshold for a notebook where pandas/Dias use less than 5GB.

As shown in Figure 11 [4], modin *slows down* 9 of the 10 total notebooks we consider compared to vanilla pandas. It speeds up one notebook which is dominated by a call to apply(), which modin is able to parallelize. As witnessed in this notebook, one advantage of modin is that it can scale with the availability of more hardware resources in cases where it can parallelize. Dias does not enjoy such scaling benefits. However, we find that modin cannot parallelize the majority of the notebooks we consider diminishing any scaling benefits. Overall, Dias is up to 26.4× faster than modin (4.1× geometric mean) for whole notebooks.

We further show that modin uses memory resources (RAM and disk) aggressively, with results in Figure 12 [5]. When deploying modin exclusively across multiple servers, it is generally acceptable to use all the available hardware resources. However, many of the users of ad-hoc EDA workloads have limited hardware resources, further highlighting the deployment issues with modin. Note that Dias, (like pandas), makes no use of the disk.

6.4 Comparing Various Dataframe Libraries

To further understand how modin and other dataframe libraries perform on ad-hoc EDA workloads, we perform a series of targeted experiments using common patterns we have found in such workloads. In addition to studying modin, we also study three other common dataframe libraries: dask [34] (version 2022.12.1), Koalas [39] (version 0.32.0), and PolaRS [15] (version 0.7). dask is another widely adopted parallel dataframe library with a slightly different API from that of pandas. Koalas implements the pandas API over PySpark [12]. PolaRS [15] is a pandas replacement (using Rust under the hood), which, however, has a different API.

We use a c5.24xlarge AWS instance with 96 vCPUs and 192 GiB of RAM. We use 12 vCPUs for modin, dask, Koalas and PolaRS. The dataset used is the NYC Yellow Taxi Dataset 2015 - January [48] (except for one case mentioned below) with a size of around 1.8GB. We picked this dataset because (a) it is large (the subset we use is the largest we could run the experiments with, using the libraries mentioned, on this machine) and these libraries specialize in large datasets and (b) it has been used in previous work [40] and in multiple notebooks throughout the Internet [21].

Column-Wise Operations. A common pattern in pandas is to perform column-wise operations. In fact, this pattern does not involve interoperability with other libraries or complex Python

[4] Dias' results in Figure 11 look slightly different from those in Figure 9, even though the same notebooks are used. This is because of the changes we had to perform on some of the notebooks (i.e., less replication and API changes) to run them with modin.

[5] The only way we found to measure modin's memory consumption somewhat reliably was using ray memory, which however was still unreliable and very slow to query. We could not obtain memory measurements for 2 notebooks.

```
df['pickup_longitude'] + df['pickup_latitude']
```

(a) Add Two Series Element-Wise

```
df['pickup_longitude'].std()
```

(b) Compute the Standard Deviation of a Series

Figure 13: Common column operations, which are fast in `pandas`. But `modin`, `dask` and Koalas are tens to hundreds of times slower.

```
np.sum(df['pickup_longitude'])
```

(a) Sum Column using `numpy`

```
col_a = 'pickup_longitude'
col_b = 'pickup_latitude'
np.where(
  pandas_df[col_a] < pandas_df[col_b],
  10, 20)
```

(b) Vectorized Conditional Assignment using `numpy`

Figure 14: Interacting with `numpy`. Users expect the interaction to be fast because `pandas` uses `numpy` as a building block. Yet, `modin`, `dask` and Koalas are tens to hundreds of times slower.

code. Because this pattern is common, pandas uses vectorized implementations. This means that not only does it process elements in bulk but also does it so completely in native code, thereby avoiding Python's overheads. We show two examples in Figure 13.

In the first example, we add two pandas columns element-wise. modin, dask, Koalas and PolaRS are 55.1×, 136.8×, 9.4× and 7.6× slower than pandas, respectively. In the second example, we perform a standard reduction over a column: computing its standard deviation. modin, dask and Koalas are 2.6×, 4.6× and 31.2× slower than pandas, respectively. Therefore, even for simple and ubiquitous operations within the pandas space, pandas replacements can incur significant slowdowns. Interestingly, PolaRS is 1.1× faster for this example.

Interaction with NumPy. Another standard operation we find common is to interoperate numpy and dataframe libraries. We hypothesize that this is a common operation because numpy is a core primitive used in pandas: pandas columns are stored as numpy arrays and many pandas operations use numpy.

We tried two simple operations shown in Figure 14. In Figure 14a we perform a simple sum over a column. In Figure 14b, we use np.where() to conditionally assign values to a pandas.Series (this is a standard way to speed up pandas [8]).

For the first example, modin, dask and PolaRS are 18.3×, 179.8× and 1.4× slower than pandas, respectively. Koalas was unable to perform this operation within the memory limits. We instead used a smaller dataset, Iris [11], a common dataset in the pattern recognition literature. Koalas is 190.4× slower in this case.

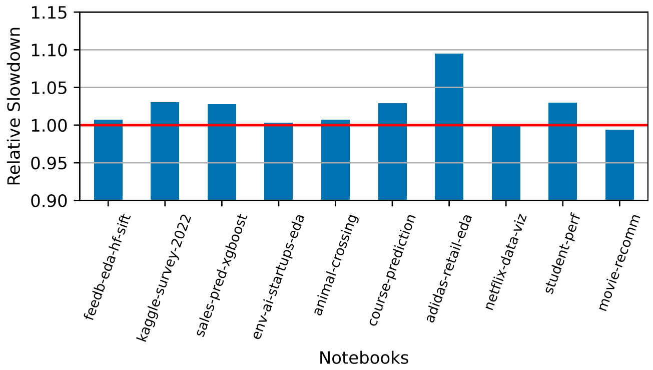


Figure 15: Slowdowns on whole notebooks without sliced execution. Disabling sliced execution leads to slowdowns and no significant speedups.

For the second example, modin, dask, Koalas and PolaRS are 54.5×, 111.7×, 202× and 3× slower than pandas, respectively.

As we can see, modin, dask and Koalas do not interoperate well with numpy. Nevertheless, such pandas usage is common. We have 1 notebook (10%) using np.sum() and another 3 (30%) using np.where() (one of which makes heavy use of it). PolaRS also incurs a slowdown, but it is much smaller.

Iterative Access of Individual Elements. The final common pattern we consider is the iterative access to individual dataframe elements. We show an example in Figure 1, which we extracted from a real notebook. modin and Koalas are 1914.5× and 155× slower than pandas, respectively (note that we were able to run the loop only for 5 iterations without out-of-memory errors in Koalas, so we compared with 5 iterations of pandas). PolaRS is 20.8× slower for 15 iterations and 99.7× slower for 100 iterations (it was realistically impossible to run the other frameworks for 100 iterations).

It was more difficult to run the experiment for dask. In general, dask.DataFrame is not intended for individual element access. This loop is not supported as-is by the dask API and we could not find a reasonable way to translate it. But for completeness, we access a single individual element in dask and we compare it with doing 15 iterations of the loop above in pandas. dask is 506× slower.

Discussion. As our results show, ad-hoc EDA workloads contain diverse code. Given the limited hardware resources available in these settings, we see that bulk-parallel dataframe libraries like modin, dask and Koalas, are not well suited for ad-hoc EDA workloads. PolaRS, can give small performance improvements compared to pandas, and its slowdowns are smaller compared to other libraries. However, it can also cause considerable slowdowns (e.g., with the iterative element access) and has a significantly different API. For example, the pandas snippet df['A'] = 1 is translated to

```
df = df.with_column(pl.lit(1).alias('A'))
```

in PolaRS. As a result, it requires learning new syntax.

```python
def foo(row):
  if row['A'] == row['B'] and row['A'] < row['C']:
    return 'X'
  elif row['A'].startswith('Y'):
    return 'Y'
  elif row['B'] in ls:
    return 'Z'
  else:
    return 'NA'

df.apply(foo, axis=1)
```

(a) Original pandas `apply()`. It processes each row sequentially, using the interpreter.

```python
conditions = [
  (df['A'] == df['B']) & (df['A'] < df['C']),
  df['A'].str.startswith('Y'),
  df['B'].isin(ls)
]
choices = [
  'X', 'Y', 'Z'
]
np.select(conditions, choices, default='NA')
```

(b) Vectorized execution using `numpy.select()`

Figure 16: Vectorized `apply()` with conditions, which can be hundreds of times faster [8]. However, performing this rewrite automatically is challenging.

6.5 Understanding Dias' Performance

To understand the performance gains of Dias, we conduct an ablation where we remove sliced execution and we also discuss two case studies in detail.

Disabling Sliced Execution. As we explained in Section 4.2, sliced execution increases the complexity of Dias. To investigate whether or not sliced execution improves performance, we ablated sliced execution. Figure 15 shows the relative *slowdown* compared to having sliced execution enabled.

As shown, all notebooks except one run faster without sliced execution. In fact, our largest cell-level speedup (57×) is lost. Furthermore, although only two patterns that hit require sliced execution, patterns presented in online tutorials [8] and which require sliced execution, can give dramatic speedups of up to 380×.

Vectorized Conditionals. We further study two case studies, starting with vectorized conditionals.

We show an example of rewriting a pandas `apply()` function with numpy's `np.select()` in Figure 16 [8]. Both versions output a certain value per row based on some conditions. The second one gives many-fold speedups, 36× in our evaluation and up to 380× in other situations [8], mainly due to the use of vectorized execution.

To do this rewrite, Dias checks that the function `foo` contains only an `if-else` chain and the conditions are such that we can translate them to equivalent that apply to whole columns (for example, we cannot translate `if bar()` for some random function `bar`). Also, the return values should be such that can be converted to numpy arrays. The constant `'X'` is such a value but if it were `bar(row['A'])`, we would not, in general, be able to translate it.

```python
arr = df['C'].values
n = len(arr)
res = np.empty(n, dtype=arr.dtype)
for i in range(n):
  spl = arr[i].split(',', maxsplit=1)
  res[i] = spl

df_temp = pd.DataFrame(res, columns=['a', 'b'])
a = res['a']
b = res['b']
```

Figure 17: `pandas.Series.str.split()` Implementation (Simplified)

Verifying these conditions is not the only tricky part; producing the rewritten version can be challenging too. For example, the original uses Python's *logical-AND* (i.e., and) to compare elements, but we need to use Python's *bitwise-AND* (i.e., &) when translating to pandas and the parentheses around the two sides are required. Similarly, a condition like `a in ls` needs to be translated to a call to the pandas `isin()` function. These are subtleties of rewriting that can be easily missed if we carry it out manually. Besides leading to bugs, they require extensive knowledge of the pandas API.

As explained in Section 4.2, these checks, and the rewriting, cannot be performed a priori because the code of `foo` might not be available yet at the start of the cell. Thus, the rewriter employs sliced execution to perform these actions on demand.

Finally, if the user changes `foo` such that it does not abide to the above conditions, the rewriter cannot perform the rewrite. At the same time, however, the original code remains intact. Thus, the code will never be slower than the original. Moreover, had the user performed the rewrite by hand, they would have to convert it back to the `apply()` version, but this effort disappears with the rewriter.

Translating to Pure Python. We present a case study of a non-intuitive result: translating an "optimized" pandas call to pure Python (Figure 4). In general, users expect pandas to be more efficient than pure Python since pandas uses vectorized, native code, while also avoiding the interpreter, when possible.

However, `.str.split()` is a *string* operation and these cannot in general be vectorized by numpy. So, a call to `.str.split()` reaches a standard Python loop to carry out the operation [33].

We would then expect the pandas version to be in par with our version. We have to look more closely to understand the discrepancy. In Figure 17, we show a simplified version of `.str.split()`'s implementation. Specifically, the important thing is that in the loop, we gather a collection of (2-element) *lists* in `res` (`res` is a numpy array but it could be any container without much difference in performance; e.g., it could be a list. The important thing is what it stores.). Then, we create our two results, our two `Series` (via creating a `DataFrame`, but the particular way of doing it is irrelevant).

In particular, we split these lists "vertically" and in half so that all the first elements of the lists create the Series a and all the second elements create the Series b.

One should contrast this with our rewritten version. There, we create only two lists (a and b). At every iteration of the loop, we create one list, the result of split(), append the individual elements to a and b and then *throw it away*. Notice that in the pandas version, the result of split has to be saved. So, while on the surface, the two loops allocate the same number of lists, in our version, the same space can be reused for every iteration.

Finally, we convert a and b (both lists of strings) to Series. Under the hood, a list of strings is a contiguous block of memory in which every element is a pointer to the string. A Series of strings is also a contiguous block of memory in which every element is a pointer to a string. So, the conversion from the one to the other is cheap. However, in the pandas version, the elements are stored together in lists "horizontally", but we want to store them together "vertically" (if we imagine a matrix where every row is a list coming from split). This halving and regrouping is expensive.

In this example, the rewriter enables us to optimize a library *without changing the library*. As we have explained earlier, the rewriter can cross library boundaries and thus it can optimize across Python, pandas and numpy, without the need to provide custom versions of these libraries.

7 RELATED WORK

Parallel and Distributed Dataframe Libraries. Most previous work on optimizing dataframe libraries falls under the use of parallel and distributed execution. Our technique is orthogonal to these techniques. First, to the best of our knowledge, no other system has used rewriting at the interface boundary of Python and pandas. Systems like modin [40], dask [34], Koalas [39], PolaRS [15], Ponder [4], PolyFrame [45] and Magpie [24] are all essentially custom versions of pandas (some are full rewrites, while others implement the pandas API over some underlying system). They use many different techniques, like the use of parallel execution using anything from Rust threads to engines like Ray [36] and Spark [55], partition schemes, query optimization and the use of the hard disk. But none uses rewriting as we do, and all of these techniques are performed *within* the library. This is the main conceptual difference, but there are also other practical drawbacks as we outlined in the previous sections, mainly arising from the fact that these systems do not focus on single-machine, ad-hoc, diverse use cases.

Optimizing Dataframe Libraries for Interactive Settings. A slightly different and interesting line of work focuses on optimizing dataframe queries for interactive workloads [27, 54]. Some of their optimizations include displaying partial results (e.g., applying head() on an expression), reordering operations and performing computation during *think-time*, i.e., when the user is inspecting results. We also recognize the importance of interactive workloads, which include the EDA, single-machine, ad-hoc workloads we focus on in this paper, but we are taking a different path in optimizing them. We use rewriting at the interface boundary, which is fundamentally different from the techniques used in this previous work.

Rewrite systems in compilers. Program rewriting is prevalent in compilers. Production-level compilers use peephole optimizers to perform local rewrites. LLVM [25] uses InstCombine [29] and VectorCombine [30] to perform IR rewrites on scalars and vectors respectively. Further, there have been many works such as Alive [31], Alive2 [32], Souper [44] that try to prove or automatically find such rewrites inside traditional compilers. TASO [23] and PET [51] have looked into how rewrites can be used to optimize tensor computations in tensor compilers. Domain specific languages such as Halide [41] include extensive rewrite engines to perform optimizations [37]. Even complicated optimization passes such as dataflow optimizations [28] and vectorization [9] can be expressed as a series of rewrites. In fact, the compiler infrastructure MLIR [26] is rooted on the premise of rewriting to express complex IR transformations. Dias takes inspiration from these systems that mainly perform static program rewrites and performs rewrites for pandas implemented in the dynamically-typed Python language.

Dynamic Optimization. There has been a large body of work that optimizes programs at runtime. Just-in-time (JIT) compilation is one common technique applied to interpreted languages like Javascript (TraceMonkey [17], V8 [14]) and non-native languages like Java (HotSpot [19]). Recently, Python also started to enjoy significant speedups from optimization at runtime, with the release of the specializing adaptive interpreter [22]. All these methods differ in one key aspect from our method: they optimize the host language, focusing on low-level optimizations (on each language's bytecode) and not the host-library combination. On the other hand, our technique can perform higher-level (i.e., library-level), and more impactful improvements because it understands the semantics of both the host language and the library.

8 CONCLUSION

In this paper, we identified program rewriting as a lightweight technique for optimizing ad-hoc, single-machine EDA workloads. Performing rewrites is valid only under conditions, which need to be checked at runtime, a setting that imposes strict latency boundaries. We implemented Dias, a system which rewrites pandas code automatically and transparently, while simultaneously addressing the requirements and constraints of condition-checking. Dias applies rewrite rules automatically, and it verifies whether applying a rule is correct by either injecting checks in the code or by slicing the execution and performing checks in between.

We experimentally showed that Dias was able to achieve significant speedups (up to 57× for individual cells and 3.5× for whole notebooks), both compared to pandas and modin, in *real-world*, randomly sampled notebooks. At the same time, Dias incurs minimal runtime and memory overheads, while making no use of the disk, whether Dias succeeds to optimize code or not. Last but not least, this paper showed a new direction for optimization, that of crossing library boundaries, the key aspect of which is to employ techniques that understand both the library and the client code.

ACKNOWLEDGMENTS

We would like to thank Marc Canby and Stratos Vamvourellis for inisghtful comments and suggestions. This work was supported by the AWS Cloud Credit for Research and the Open Philanthropy project.

REFERENCES

[1] AIEducation. 2022. What course are you going to take? https://www.kaggle.com/code/aieducation/what-course-are-you-going-to-take/. Accessed: 2022-12-09.

[2] Python ast module. 2022. https://docs.python.org/3/library/ast.html. Accessed: 2022-12-09.

[3] Python ast module: Constant. 2022. https://docs.python.org/3/library/ast.html#ast.Constant. Accessed: 2022-12-09.

[4] Ponder | Pandas at Scale. 2022. https://ponder.io/. Accessed: 2022-12-09.

[5] Rounak Banik. 2017. Movie Recommender Systems. https://www.kaggle.com/code/rounakbanik/movie-recommender-systems. Accessed: 2022-12-09.

[6] Stefanos Baziotis, Daniel Kang, and Charith Mendis. 2022. Dias: Dynamic Rewriting of Pandas Code (Extended Version). https://baziotis.cs.illinois.edu/papers/dias.pdf. Accessed: 2022-12-09.

[7] Erik Bruin. 2022. NLP on Student Writing: EDA. https://www.kaggle.com/code/erikbruin/nlp-on-student-writing-eda. Accessed: 2022-12-09.

[8] Nathan Cheever. 2019. 1000x faster data manipulation: vectorizing with Pandas and Numpy. https://www.youtube.com/watch?v=nxWginnBklU. Accessed: 2022-12-09.

[9] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[10] Atanu Dan. 2020. Pandas DataFrame: Performance Optimization. https://medium.com/@atanudan/pandas-dataframe-performance-optimization-8b87db24c2c4.

[11] Iris Dataset. 1936. https://archive.ics.uci.edu/ml/datasets/Iris. Accessed: 2022-12-09.

[12] PySpark Documentation. 2022. https://spark.apache.org/docs/latest/api/python/. Accessed: 2022-12-09.

[13] Pandas Documentation. 2023. Enhancing performance. https://pandas.pydata.org/docs/user_guide/enhancingperf.html.

[14] Javascript V8 Engine. 2022. https://v8.dev/. Accessed: 2022-12-09.

[15] PolaRS: Lightning fast DataFrame library for Rust and Python. 2022. https://www.pola.rs/. Accessed: 2022-12-09.

[16] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods*, Aaron Dutle, César Muñoz, and Anthony Narkawicz (Eds.). Springer International Publishing, Cham, 185–202.

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[18] Dwight Guth. 2013. A formal semantics of Python 3.3. (2013).

[19] Christian Häubl and Hanspeter Mössenböck. 2011. Trace-Based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 129–138. https://doi.org/10.1145/2093157.2093176

[20] Sofia Heisler. 2017. No More Sad Pandas Optimizing Pandas Code for Speed and Efficiency, PyCon 2017. https://www.youtube.com/watch?v=HN5d490_KKk.

[21] NYC Taxi Dataset Used in Kaggle Competition. 2017. https://www.kaggle.com/c/nyc-taxi-trip-duration. Accessed: 2022-12-09.

[22] Python Specializing Adaptive Interpreter. 2021. https://peps.python.org/pep-0659/. Accessed: 2022-12-09.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[24] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends.. In *CIDR*.

[25] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO 2021*.

[27] Doris Jung Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: Always-on Visualization Recommendations for Exploratory Data Science. *CoRR* abs/2105.00121 (2021). arXiv:2105.00121 https://arxiv.org/abs/2105.00121

[28] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations from Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (aug 2021), 29 pages. https://doi.org/10.1145/3473579

[29] LLVM. 2022. InstCombine. https://llvm.org/doxygen/InstructionCombining_8cpp_source.html. Accessed: 2022-12-09.

[30] LLVM. 2022. VectorCombine. https://llvm.org/doxygen/VectorCombine_8cpp_source.html. Accessed: 2022-12-09.

[31] Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI'15, Portland, OR, USA.* ACM. https://www.microsoft.com/en-us/research/publication/provably-correct-peephole-optimizations-alive/

[32] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[33] Pandas 1.5.1: _map_infer_mask(). 2022. https://github.com/pandas-dev/pandas/blob/91111fd99898d9dcaa6bf6bedb662db4108da6e6/pandas/_libs/lib.pyx#L2863. Accessed: 2022-12-09.

[34] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 126 – 132. https://doi.org/10.25080/Majora-7b98e3ed-013

[35] Fahad Mehfooz. 2021. ClubHouse EDA. https://www.kaggle.com/code/fahadmehfoooz/clubhouse-eda. Accessed: 2022-12-09.

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications *(OSDI'18)*. USENIX Association, USA, 561–577.

[37] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (nov 2020), 28 pages. https://doi.org/10.1145/3428234

[38] Jupyter Notebooks. 2022. https://jupyter-notebook.readthedocs.io/en/latest/notebook.html. Accessed: 2022-12-09.

[39] Koalas: pandas API on Apache Spark. 2022. https://koalas.readthedocs.io/en/latest/. Accessed: 2022-12-09.

[40] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2022. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proc. VLDB Endow.* 15, 3 (feb 2022), 739–751. https://doi.org/10.14778/3494124.3494152

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[42] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[43] Python for Social Scientists San Diego State University, Linguistics/BDA 572. 2022. https://gawron.sdsu.edu/python_for_ss. Accessed: 2022-12-09.

[44] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. https://doi.org/10.48550/ARXIV.1711.04422

[45] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-Based Approach to Scaling Dataframes. *Proc. VLDB Endow.* 14, 11 (oct 2021), 2296–2304. https://doi.org/10.14778/3476249.3476281

[46] Sunny Solanki. 2021. How to Speed up Code involving Pandas DataFrame using Numba? https://coderzcolumn.com/tutorials/python/guide-to-speed-up-code-involving-pandas-dataframe-using-numba.

[47] IPython: Magic Command System. 2022. https://ipython.readthedocs.io/en/stable/interactive/reference.html#magic-command-system. Accessed: 2022-12-09.

[48] New York (N.Y.). Taxi and Limousine Commission. 2015. TLC Trip Record Data. https://dask-data.s3.amazonaws.com/nyc-taxi/2015/yellow_tripdata_2015-01.csv. Accessed: 2022-12-09.

[49] Eyal Trabelsi. 2021. Practical Optimisation for Pandas. https://www.youtube.com/watch?v=zdubYLjXHb0.

[50] Prakritidev Verma. 2017. Notebook673580193d. https://www.kaggle.com/code/prakritidevverma/notebook673580193d. Accessed: 2022-12-09.

[51] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[52] IPython Website. 2022. https://ipython.org/. Accessed: 2022-12-09.

[53] Solving Real-World Business Questions with Python Pandas. 2020. https://medium.com/li-ting-liao-tiffany/solving-real-world-business-questions-with-pandas-70ef8ef02675. Accessed: 2022-12-09.

[54] Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time. *CoRR* abs/2103.02145 (2021). arXiv:2103.02145 https://arxiv.org/abs/2103.02145

[55] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

A EXTENDED RESULTS

In Section 6 we focused only on ten out of the twenty random notebooks we picked (see Section 6.1). Here, we include results for all twenty notebooks.

Per-Cell Speedups. Figure 21 shows the cell-level speedups, corresponding to Figure 8. The plots look almost identical, and this is because Dias does not decelerate notebooks it does not rewrite. Thus, since this plot includes only slowdowns or speedups that are outside the 10% range, there is hardly any discernible difference. Similar observations are derived from Figure 18, where the slowdowns are still under interactive latency, i.e., 300ms.

These results further validate our hypothesis in Section 6.2. That is, the slowdowns we observe are the result of rewriting, independent of who performs it (in this case, Dias).

When we include all twenty notebooks, the geometric mean speedup is 1.1× and the maximum slowdown is 28%.

Per-Notebook Speedups. In Figure 20 we show the notebook-level speedups. This figure corresponds to Figure 9. As we mentioned, Dias does not rewrite code in the the ten new notebooks, so we do not see any additional speedup. However, it remains that the slowdowns, when Dias does not succeed, are minimal. The geometric mean speedup is now 1.13× while the maximum slowdown is 3.5%.

Comparison with Modin. In Figure 22, which corresponds to Figure 11, our conclusions are again unaltered. `modin` slows down all the ten new notebooks and it rarely scales with the number of cores. The geometric mean and maximum speedup remain the same (see Figure 11).

In Figure 19, which corresponds to Figure 12, we show the memory consumption of Dias, `pandas`, and `modin`, when we consider all twenty notebooks. The results are not significantly different for `pandas` and Dias. However, `modin`'s memory consumption becomes even more aggressive. We see that for one notebook, `modin` consumes almost 250GB when `pandas` and Dias consume less than 5GB.

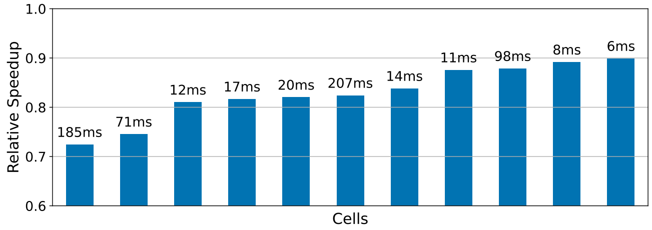


Figure 18: Corresponding to Figure 10. The conclusions are the same. The slowdowns are within interactive latencies (i.e., less than 300ms).

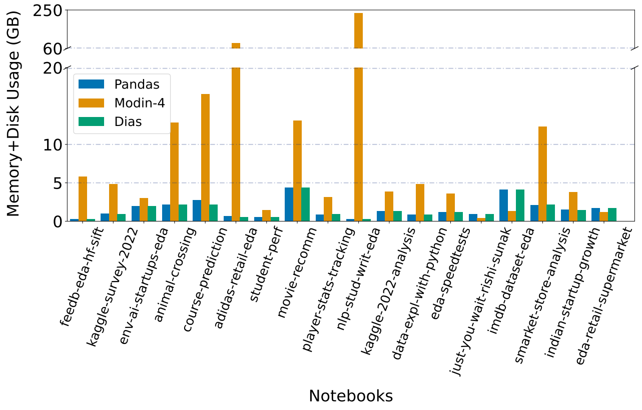


Figure 19: Corresponding to Figure 12. When we include all 20 notebooks, we see even more aggressive memory+disk usage from `modin`. Dias and `pandas` remain on the same scales.

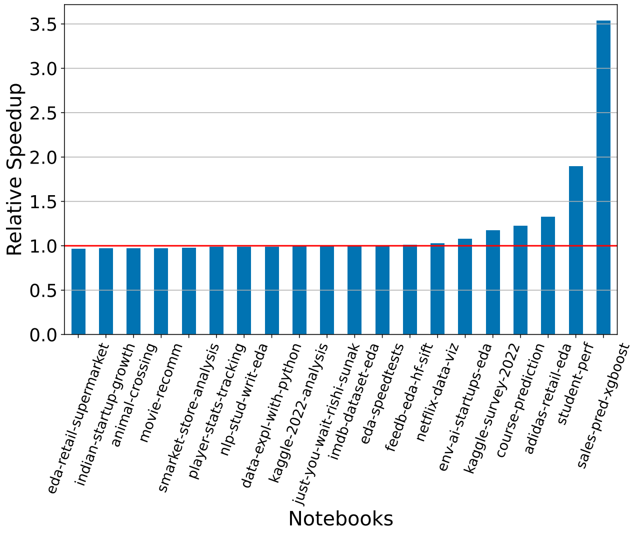


Figure 20: Relative speedups on whole notebooks. We see the same speedups as in Figure 9 with no extra substantial slowdowns when considering all 20 notebooks.

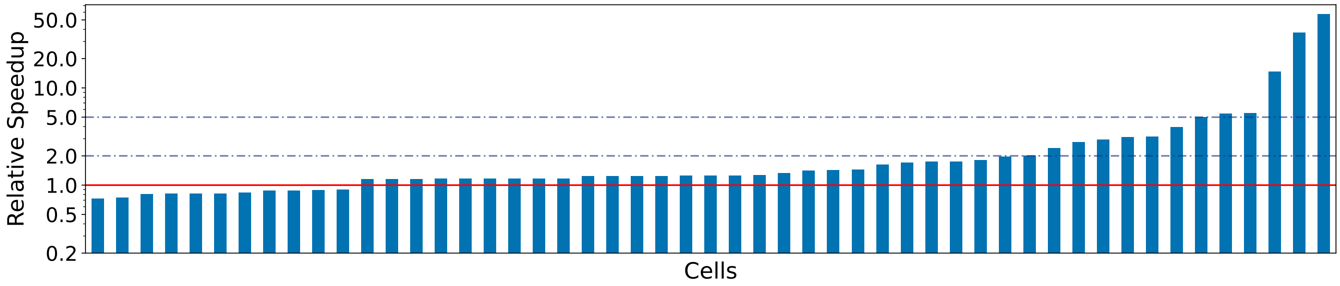


Figure 21: Cell-level relative speedups (excluding cells that originally ran for less than 50ms and also all the cells that got a speedup or slowdown within the 10% range) for all 20 notebooks. Still, Dias provides significant speedups with no substantial slowdowns (see Figure 18).

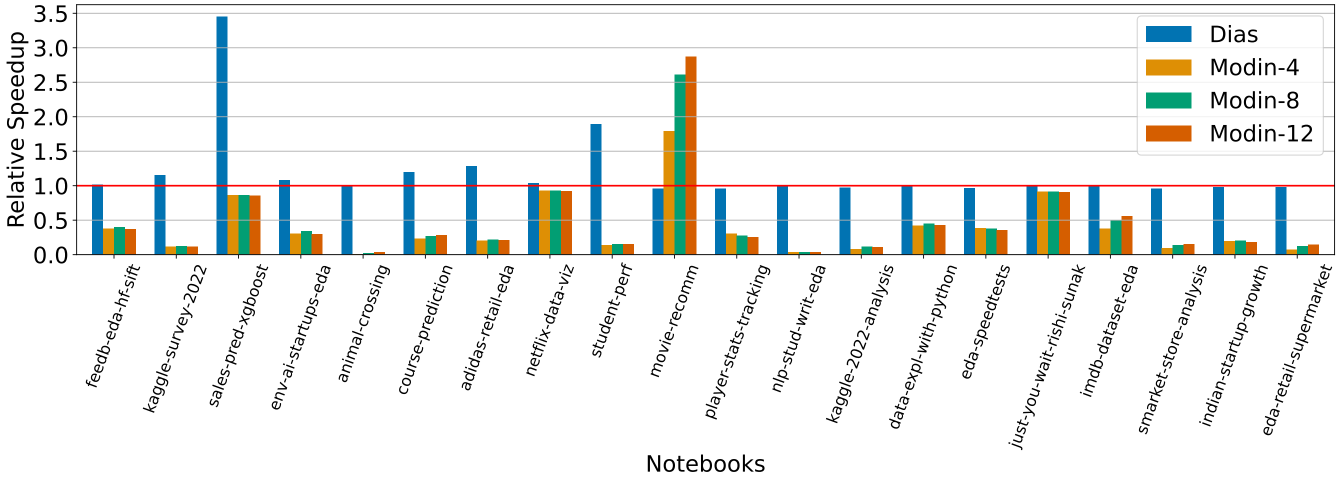


Figure 22: Corresponding to Figure 11. The conclusions are similar as `modin` slows down all the ten new notebooks.