Dias: Dynamic Rewriting of Pandas Code

STEFANOS BAZIOTIS, University of Illinois (UIUC), U.S.A
DANIEL KANG, University of Illinois (UIUC), U.S.A
CHARITH MENDIS, University of Illinois (UIUC), U.S.A

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 source-to-source, external program rewriting as a lightweight technique which can optimize across representations, and offer substantial speedups while also avoiding slowdowns. We implemented Dias, which rewrites notebook cells to be more efficient for ad-hoc EDA workloads. We develop techniques for efficient rewrites in Dias, including checking the preconditions under which rewrites are correct, dynamically, at fine-grained program points. 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 27.1× compared to modin.

CCS Concepts: • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: pandas, rewriting, dynamic, cross-representation

1 INTRODUCTION

In recent years, *dataframe*-based libraries, such as pandas, have become increasingly popular with users ranging from social scientists to business analysts [40, 51]. 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 [36], dask [31], and PySpark [10] 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

Authors' addresses: 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.

```python
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
```

Fig. 1. Loop which accesses individual elements (source: Kaggle [6]). This loop can be hundreds of times slower in bulk-parallel frameworks like modin, dask etc. and PolaRS, which are not optimized for individual accesses.

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.

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 6.6). 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, several rewrite rules in this paper were *not apparent to the authors* (e.g., the rules used to perform the rewrites in Figure 3 and Figure 4), 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 [7, 9, 11, 16, 44, 47]. Even then, optimizing code correctly is challenging for non-experts and can lead to subtle bugs.

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

Dias is the first source-to-source, dynamic rewriter for a Python library. Source-to-source rewriting/compilation, especially over Python code, is challenging because source languages are high-level, while traditional compiler optimizations work better in low-level intermediate representations. But, we observe that operating at the source level creates novel opportunities. One such opportunity is that it enables us to rewrite across different representations (e.g., Python and pandas). Such

```python
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.

```python
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.

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

rewrites are high-level (leading to significant speedups), and thus, the source language is a suitable representation for performing them. It may be useful to compare and contrast a high-level rewrite like the one in Figure 3 with a traditional low-level rewrite such as a*2 → a«1.

To perform these rewrites, DIAS needs to operate externally, in contrast with previous work (e.g., TensorFlow Grappler [46], modin [36], BELE [53]), which are implemented as libraries. Namely, it views all user's code compared to just library code and can modify any part of it. This allows DIAS to rewrite across library boundaries (e.g., Figure 3), which is infeasible with current techniques.

However, implementing a source-to-source, external rewriter introduces new challenges. First, DIAS needs to understand more than one representation (i.e., pandas and Python). Second, providing any guarantees when operating in a high-level language is naturally harder, especially when it comes to Python which has no formal semantics and liberal typing and scoping rules. A specific challenge was performing the precondition checks at the correct program points, as we explain in Section 4.2. Finally, DIAS must operate within interactive time scales: the overhead of rewriting cannot dominate cost savings.

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× (1.27× geometric mean) and whole notebooks by up to 3.6× (1.31× geometric mean). We also compare DIAS with modin and show that it can be up to 27.1× faster for whole notebooks (4.9× 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.

(2) We develop DIAS, the first external rewriter to rewrite code across different representations in a dynamic setting. We introduce rewrite rules that can significantly speed up pandas code, including ones that cross library boundaries.

(3) Dias applies these rewrite rules automatically, at runtime, and verifies whether applying a rule is correct by injecting checks at fine-grained program points.

(4) 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 [36] and show that it can be up to 27.1× faster for whole notebooks (4.9× 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) [39]. 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. To accelerate such workloads, both industry and academia have invested in accelerating *bulk-parallel* workloads.

Frameworks including modin [36] and dask [31] 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× [36].

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 6.6, 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 772× higher than native pandas. The peak memory usage increased 53× on average (geometric mean).

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). Six of these notebooks (30%) 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.

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

(a) Original: Concatenate `Series` by first turning them into lists. Extracted from a Kaggle notebook [48].

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

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

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

```
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].

```
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

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

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.

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.

We designed DIAS with two high-level components. First, DIAS' *syntactic pattern matcher* matches the input code against the syntactic patterns the rewrite rules. The second component is a *rewriter*, which checks the runtime preconditions of the rewrite rules and on success, rewrites the code to an equivalent, but faster 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 i.e., that it is sound, which presents the main technical challenge.

To guarantee soundness, we first formalize the rewrites (Section 3.1). Second, we accurately describe the conditions under which a rewrite can be applied, some of which can be quite involved. For example, some require DIAS to check the form of whole functions. And finally, most of these conditions can only be checked at runtime, and checking them at the correct program points requires delicate program transformations (Section 4.2).

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.

The most general form of a rule consists of the following:

(1) **LHS**: A code fragment with some parts that may vary. We call those the **varyingSet**. This LHS must be recognizable with a subtree search on the AST.
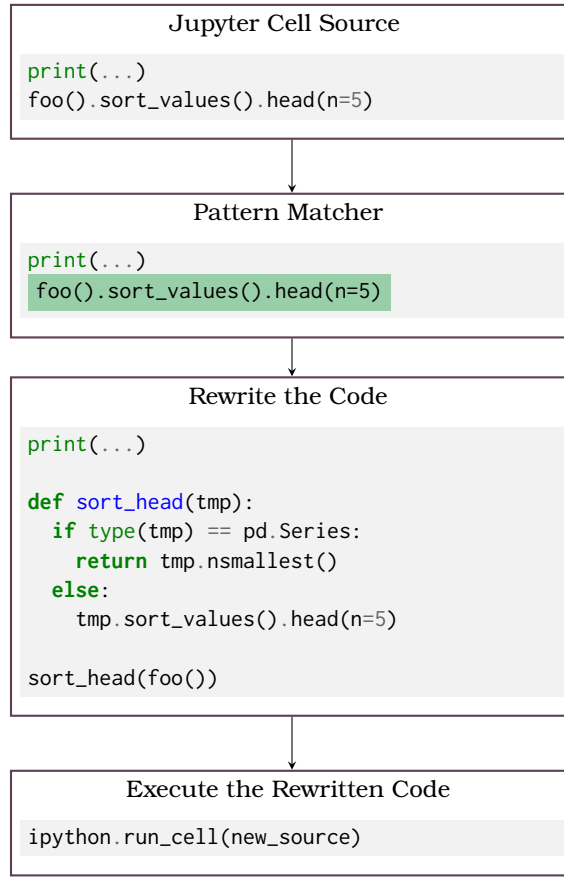
```
                        Jupyter Cell Source

print(...)
foo().sort_values().head(n=5)
```

```
                          Pattern Matcher

print(...)
foo().sort_values().head(n=5)
```

```
                          Rewrite the Code

print(...)

def sort_head(tmp):
  if type(tmp) == pd.Series:
    return tmp.nsmallest()
  else:
    tmp.sort_values().head(n=5)

sort_head(foo())
```

```
                     Execute the Rewritten Code

ipython.run_cell(new_source)
```

Fig. 5. DIAS overview. DIAS identifies patterns in the source code, which it rewrites using its rewriter. The optimized version is used only if certain dynamic checks pass, to ensure correctness.

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

(a) Select the 5 smallest elements by sorting first. Extracted from a Kaggle notebook [32].

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

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

Fig. 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.

(2) **TransformLHS**: An AST transformer that transforms the LHS. We call the result the RHS of the rule.

(3) **RuntimePrecond**: An algorithm for checking the runtime preconditions under which using the RHS instead of the LHS is semantic-preserving. This is essentially a callback function, with inputs a subset of the **varyingSet**, but evaluated. This callback also gets a state snapshot of the interpreter as input. It returns true or false.

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

(a) LHS ⟼ RHS

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

(b) Preconditions

Fig. 7. An example of a rewrite rule, named **SortHead**. If we match the LHS in the source code, we can replace it with the RHS only if the preconditions hold (at runtime).

```
"{}.nsmallest(n={})".format(called_on, first_n)
```

Fig. 8. A sketch of the implementation of the RHS of the rule in Figure 7

```
def foo(x):
  return True if x['Fare'] > 10 else False
df.apply(foo, axis=1)
```

(a) A function applied to the whole dataframe.

```
def foo(x):
  return True if x > 10 else False
df['Fare'].apply(foo)
```

(b) The function touches only one column so we can increase locality by iterating only that column.

Fig. 9. Apply a function to a single column instead of a whole dataframe and increase locality

LHS. We introduce some notation to show the structure of the LHS. 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 varying part of the rule (i.e., an element of the **varyingSet**). These parts can be matched to multiple valid options. Inside the curly brackets, we describe 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, which we refer to as Const for short, we optionally specify the type of the constant inside parentheses.[1] So, @{Const(int)} denotes that any integer constant can appear in its place. We need to refer to elements of the **varyingSet** in other components of the rewrite rule. So, we bind them to names. For example, the code string df.sort_values().head() matches the LHS of Figure 7 and called_on is bound to (the string) 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.

TransformLHS. This is a function which takes as input the **LHS** as an AST and outputs the RHS as an AST. The simplest form is a function that outputs a constant string, with some elements

---

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

```
@{expr: called_on}.apply(@{Name: func}, axis=@{expr: axis})
```

Fig. 10. RemoveAxis1 rule's LHS

of **varyingSet** (also strings) interpolated. For example, the RHS of Figure 7 can be implemented as shown in Figure 8. The values of `called_on` and `first_n` are part of the **varyingSet**. This LHS transformer is quite simple, but others can be significantly more complex and they can also depend on *dynamic* information (the transformer we just described does not as `called_on` and `first_n` are extracted from the text).

RuntimePrecond. The runtime preconditions describe conditions which have to hold at *runtime* for the original (LHS) and the RHS to be semantically equivalent. 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.

In general, **RuntimePrecond** imposes some restrictions which are up to the rule designer and implementer. For example, it should be the case that the subset of **varyingSet** used in the preconditions is evaluated regardless of the result of the precondition check.

In their generic form, the runtime preconditions are also functions that depend on dynamic information. These can be simple, like the one in Figure 7, or more complicated (i.e., whole algorithms).

Table 1 shows nine more rewrite rules we use in Dias. The first two correspond to the examples in Figure 4 and Figure 3, respectively. The TransformLHS's are simple transformers which output a paremeterized string, similar Figure 8 which we saw above. Also, we introduce some notation for syntactic preconditions, which are prefixed by 𝔖. These are simple syntactic conditions (i.e., equalities on ASTs) which are part of the LHS (and which are checked in the pattern matcher). They differ from the runtime preconditions (**RuntimePrecond**), which are prefixed by ℜ.

Together with the rules in Figure 7, the rule that achieves the rewrite in Figure 2 (named ApplyOnlyMath), RemoveAxis1 (which we discuss below) and the VectorizedConditionals rule (which we discuss as a case study in Section 6.5), they make up all the rules used in Dias. We note that we leave for future work the formal description of the latter three rules.

RemoveAxis1. We will briefly discuss one more rule, called RemoveAxis1, whose **RuntimePrecond** and **TransformLHS** components are significantly more complicated. An example of applying this rule is shown in Figure 9. This rule targets cases where `apply()` is applied to a whole `DataFrame`. It checks whether the function passed to `apply()` touches only a single column and if so, it rewrites the code so that the function is applied only to this column.

We show the LHS in Figure 10. The **RuntimePrecond** component, which we do not include here for clarity purposes, is a whole algorithm which basically checks that only a single column of the `DataFrame` is accessed. The **TransformLHS** component replaces all `Subscript`'s inside `@{func}` with just their object and it removes the `axis` argument.

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.

LHS	TransformLHS	RuntimePrecond

`# SplitInPython`
`@{Name: df}[[@{Const(str): a}, @{Const(str): b}]] =`
`  @{expr: ser}.str.split(`
`    @{Const(str): sep}, expand=True)`
	`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)`
		$\mathfrak{R}$: `type(@{ser}) == pandas.Series`

`# InplaceUpdate`
`@{Name: df1}[@{Const(str): c1}] = \`
`  @{Name: df2}[@{Const(str): c2}]`
`    .@{Name: f}(@{expr: arg})`
	`@{df1}[@{c1}].@{f}(@{arg}, inplace=True)`	$\mathfrak{R}$: `type(@{df1}) == pandas.DataFrame`
		$\mathfrak{S}$: `@{df1} == @{df2}`
		$\mathfrak{S}$: `@{c1} == @{c2}`
		$\mathfrak{S}$: `@{f.id} in {"fillna", "replace",`
		`"rename", "dropna", "sort_values",`
		`"drop_duplicates", "sort_index"}`

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

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

`# ReplaceRemoveList`
`@{expr: e}.replace([@{Const(str): x1}],`
`              @{Const(str): x2})`
	`@{e}.replace(@{Const(str): x1},`
	`              @{Const(str): x2})`
		$\mathfrak{R}$: `type(@{e}) == pandas.DataFrame`

`# InToString` `@{Const(str): x} in @{expr: e}.to_string()`	`@{e}.astype(str).str.contains(x).any()`	$\mathfrak{R}$: `type(@{e}) == pandas.Series` $\mathfrak{R}$: `@{e}.index.dtype == np.int64` $\mathfrak{R}$: `no_whitespace(@{x})`
`# FuseIsIn` `(@{Name: o1}[@{Const(str): x1}].isin(@{Name: s1}))	\` ` (@{Name: o2}[@{Const(str): x2}].isin(@{Name: s2}))`	`@{o1}[@{x1}].isin(@{s1}+@{s2})`	$\mathfrak{R}$: `type(@{o1}) == pandas.DataFrame` $\mathfrak{S}$: `@{o1} == @{o2}` $\mathfrak{S}$: `@{x1} == @{x2}`

`# FuseApply`
`@{expr: e}.apply(@{expr: f1}).apply(@{expr: f2})`
	`def fused_apply(e, f1, f2)`
	`  res = []`
	`  for it in e.tolist():`
	`    x = f1(it)`
	`    y = f2(x)`
	`    res.append(y)`
	`  return pd.Series(res, e.index)`
	`fused_apply(@{e}, @{f1}, @{f2})`
		$\mathfrak{R}$: `type(@{e}) == pandas.Series`

`# FuseStrSplit`
`@{Name: o1}[@{Const(str): x1}] = \`
`  @{Name: o2}[@{Const(str): x2}].\`
`    str.split(@{Const(str): sep})`
`@{Name: o3}[@{Const(str): x3}] = \`
`  @{Name: o4}[@{Const(str): x4}].str[1]`
	`def fused_str_split(df, col, sep)`
	`  res = []`
	`  for it in df[col].tolist():`
	`    spl = it.split(sep)`
	`    x = spl[1] if len(spl) > 1 else None`
	`    res.append(x)`
	`  return pd.Series(res, df[col].index)`
	`@{o1}[@{x1}] = fused_str_split(@{o1}, @{x1})`
		$\mathfrak{R}$: `type(@{o1}) == pandas.DataFrame`
		$\mathfrak{S}$: `@{o1} == @{o2} == @{o3} == @{o4}`
		$\mathfrak{S}$: `@{x1} == @{x2} == @{x3} == @{x4}`

`# FuseReplaceUnique`
`@{Name: o1}[@{Const(str): x1}] =\`
`  @{Name: o2}[@{Const(str): x2}] .replace(@{Name: d})`
`@{Name: o3}[@{Const(str): x3}].unique()`
	`uniq, res = set(), []`
	`for s in @{e}.tolist():`
	`  try:`
	`    s = d[s]`
	`  except:`
	`    pass`
	`  res.append(s)`
	`  uniq.add(s)`
	`@{o1}[@{x1}] = pandas.Series(res,`
	`  @{o1}[@{x1}].index)`
	`np.array(uniq, dtype=@{o1}[@{x1}].dtype)`
		$\mathfrak{R}$: `type(@{o1}) == pandas.DataFrame`
		$\mathfrak{R}$: `type(@{d}) == dict`
		$\mathfrak{S}$: `@{o1} == @{o2} == @{o3}`
		$\mathfrak{S}$: `@{x1} == @{x2} == @{x3}`

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 $\mathfrak{S}$ denotes syntactic preconditions while $\mathfrak{R}$ denotes runtime ones. The name of the rule appears as a comment in the LHS column.

```
test(mod_x(), foo().read_x().sort_values().head())
```

Fig. 11. A nested expression with global state access.

```
def sort_head(tmp):
  return tmp.nsmallest() if type(tmp) == pd.Series
    else tmp.sort_values().head()

test(mod_x(), sort_head(foo.read_x()))
```

Fig. 12. A correct dynamic check with a local binding.

4.1 Dias Pattern Matcher

The pattern matcher is responsible for matching a sequence of statements with the **LHS** part of any rewrite rule. Whether any **LHS** (represented as an AST) matches any part of the source AST, is essentially a sub-tree search problem. The pattern matcher performs a greedy search and it returns the first LHS it matches.

To minimize matching overhead, we designed the pattern matcher to be hierarchical, by factoring 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.

Lastly, the pattern matcher needs to be able to match patterns that span multiple statements. Having a function that matches single-statement patterns by performing a greedy search, 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 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 runtime pre-conditions (i.e., **RuntimePrecond** just returns True), then the rewriter can invoke **TransformLHS** on the **LHS**, and replace the **LHS** with the result.

For example, consider the rule in Figure 7. @{called_on} needs to be evaluated first, let us name the resulting object res, then execution needs to *stop*, check the precondition on res, and then continue (i.e., evaluating either res.sort_values().head() or res.nsmallest(), depending on the precondition result).

This is more difficult than it looks because we do not have arbitrary control over the evaluation of the code, since we are operating at the source level. So, we need to *effectively* do the same thing but using source-level transformations. This is difficult because we are not allowed to evaluate @{called_on} twice. The obvious solution is to just evaluate it once and reuse it.

However, this requires careful orchestration. Consider for example the code in Figure 11. The evaluation-and-reuse should happen *exactly* where the original sub-expression (involving sort_values()) appears. Otherwise (e.g., if we save it in a variable by adding a statement above), it is possible that read_x() will read a stale value.

To solve that in general, we need a local binding of the evaluation of `@{called_on}` [2]. However, Python does not have local bindings, so the workaround is to create a function and call it at the place of the original expression, as in Figure 12. The local binding here is the binding to the function's parameter.

Dynamic RHS. Observe that in the solution we just mentioned, to create the function `sort_head()`, we need to know the RHS a-priori. This is true when **TransformLHS** does not depend on dynamic information and thus we can "run" it offline. This is the case for the rule in Figure 7. However, this is not the case for the `RemoveAxis1` rule, because it depends on the body of `@{func}`, which is not known offline. To implement such rules, all of which involve `apply()` and which face the same problem, we just overwrite `apply()`. In the overwritten body, we invoke **RuntimePrecond**, which depends on the body of `@{func}`, which is however available because it is passed as an argument. If the preconditions pass, we rewrite the body on the fly and invoke it appropriately. For example, in the case of `RemoveAxis1`, we rewrite the body as described earlier and we call `self[theOneSeries].apply(new_body)`. Note that `self` is the evaluated `@{expr}`. We never see this `@{expr}`, but we know this fact because `self` is bound to the object on which (the overwritten) `apply()` gets called.

5 IMPLEMENTATION

5.1 IPython Integration

To work automatically, DIAS currently depends on IPython [50], which is an enhanced Python interpreter. This implies that a current limitation of our implementation is that DIAS is not automatic in 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 [34]), which requires IPython. However, DIAS can still be used as a library even with a standard Python interpreter. In the rest of this section, we will assume that DIAS is running on top of IPython.

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 goal of our implementation to invoke DIAS automatically before a cell is executed, and rewrite it automatically, on the fly. A key feature of IPython that allows us to do that is the input transformer [18]. This allows us to register a function that runs before a cell gets executed. That function gets the cell content as a string and returns a new string which becomes the new cell. Our input transformer just inserts a call to DIAS' main routine, with the original cell passed as an argument. DIAS then potentially rewrites and finally executes the cell. This is all automatic; the user just needs to import DIAS.

DIAS' main routine gets 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., magic functions), which this library will not handle. This did not cause serious problems in practice but we hope to fix it in the future.

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 because DIAS needs to inspect dynamic information like names, types, function objects, etc.

5.2 Crossing Library Boundaries

It might seem that we could achieve the same optimizations simply by modifying the pandas library. In fact, for some rules, the implementation is effectively a replacement of pandas routines

---

[2]In the style of a `let` expression in OCaml.

with our own. For example, `RemoveAxis1` is implemented by overriding pandas's `apply()`. This way, we get both the `@{func}` and the `@{called_on}` components easily, without needing to do source-level transformations (like the ones we described in Section 4.2). The former is accessed through the `self` implicit argument, and the latter through the `func` argument to `apply()`.

However, this approach does not suit all cases. First, in some cases, it is simply easier to operate as an external rewriter. For example, the rule in Figure 7 can be applied if `sort_values()` is followed by `head()`. If we overwrite `sort_values()`, we cannot know what happens with its result. If we overwrite `head()`, we cannot know how the object it is applied to came to be. There are ways to work around these limitations and one popular one is lazy execution, which has been employed for similar purposes [53] (Modin also employs it).

In summary, in lazy execution, we do not execute the code but rather we log which functions have been called. After a call chain is evaluated, the result is a computation graph that captures the whole computation. We can evaluate it in the trivial way (e.g., actually calling the functions in sequence), or in an optimized way (e.g., by calling `nsmallest`).

The problem, however, is that this requires a lot of bookkeeping to know when exactly to evaluate the computation graph, to hide from the user the fact that the functions do not return the type the user expects them to return, to build this computation graph, etc. In other words, lazy execution is a hack around the fact that we do not see all the computation and we are just trying to reconstruct it from inside a library. With an external rewriter, this is trivially solved because we just view all the code. So, we just apply well-known code transformation techniques.

But the most important benefit of an external rewriter is that it can effortlessly rewrite across representations. For example, it would be nearly impossible to support the rewrite shown in Figure 3 with lazy execution by keeping the API intact because `tolist()` is supposed to actually return a Python list, thereby moving us away from the library's control. So, the lists will unavoidably get concatenated in pure Python. But an external rewriter just views the code and it can trivially perform the rewrite.

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
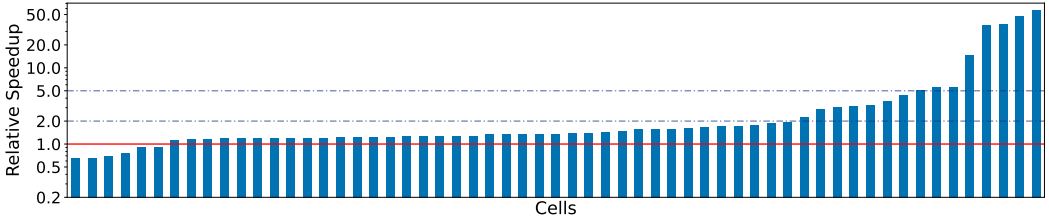
Fig. 13. Cell-level relative speedups (excluding cells that originally ran for less than 50ms and also all the cells that got a speedup or slowdown within the 0.1× range). Again, DIAS provides significant speedups by up to 57×. There are also slowdowns, which are not substantial (see Figure 15).

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 [3]. We compare DIAS with pandas (version 1.5.1) and modin [36] (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 14. 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 1.03×. DIAS rewrites cells in these notebooks but it does not achieve speedups.

Per-Cell Speedups. We show per-cell speedups in Figure 13. 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 0.1x 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 pattern Vectorized Conditionals, which is discussed in Section 6.5. The majority of cells we consider are improved by DIAS. The maximum slowdown in an individual cell is 1.56×. In general, the cells
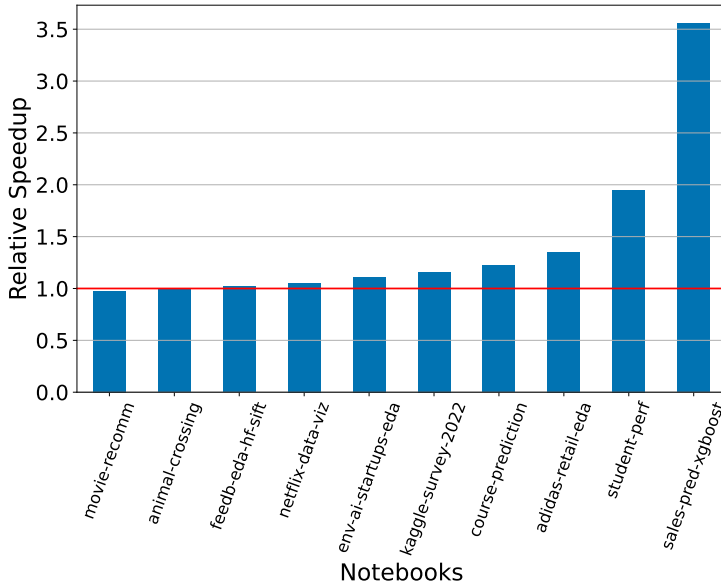
---
[3]Not cited to preserve anonymity

Fig. 14. Relative speedups on whole notebooks. Dias speeds up notebooks by up to 3.6× while not slowing down any notebook by more than 1.03×.

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 *22 ms* with a geometric mean overhead of 0.41ms.
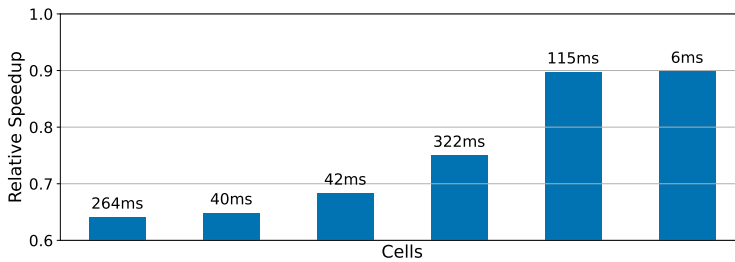
Fig. 15. The subset of cells from Figure 13 that got slowed down. Above the bars, we show the *absolute* slowdown. The slowdowns are within interactive latencies (i.e., less than 300ms), with the maximum overhead of Dias being 23ms.

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 15 we show only the cells from Figure 13 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).
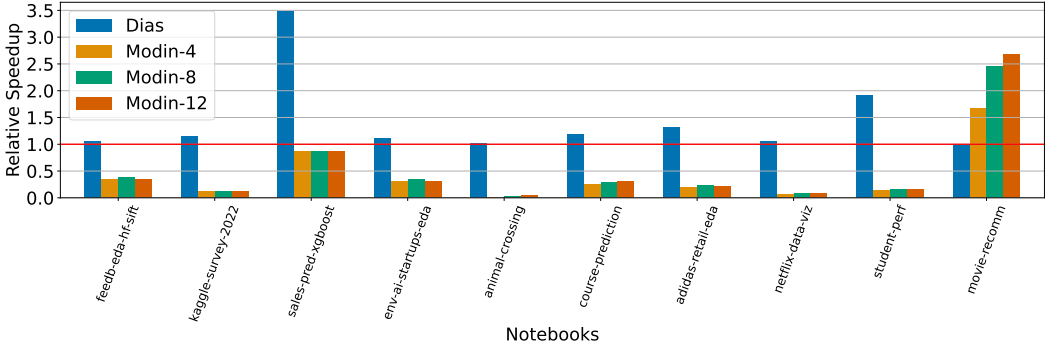

Fig. 16. Comparing DIAS with modin [36]. DIAS is faster for 9 out of 10 notebooks (Up to 27.1× faster with 4.9× geometric mean). modin is, in many cases significantly, slower than the original for these 9 notebooks. For the one notebook where DIAS is slower, it is no more slower than 1.03× compared to the original.

6.3 Comparison with Modin

We compare DIAS with modin [36] (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.

As shown in Figure 16 [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 27.1× faster than modin (4.9× geometric mean) for whole notebooks.

---

[4]DIAS' results in Figure 16 look slightly different from those in Figure 14, 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.
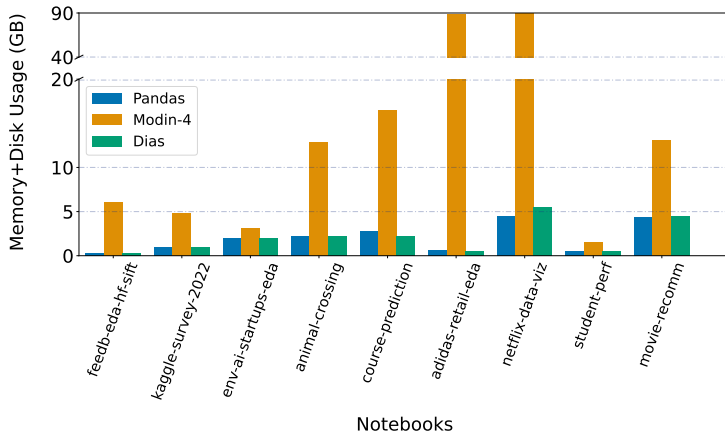
Fig. 17. RAM and disk usage comparison in modin, Dias and pandas. Dias and pandas do not use the disk and they use almost the same amount of RAM in all cases. modin uses the RAM and disk aggressively, surpassing the 80GB threshold for a notebook where pandas/Dias use less than 5GB.

Note that if we include all 20 notebooks in our evaluation, modin slows down *all* 10 new notebooks (19/20 → 95%). Also, for individual cells, Dias reaches speedups up to 1909× compared to modin.

We further show that modin uses memory resources (RAM and disk) aggressively, with results in Figure 17 [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. In fact, if we run modin across all 20 notebooks, it consumes up to 250GB when pandas and Dias consume less than 5GB. Note that Dias, (like pandas), makes no use of the disk.

6.4 An Estimate of Dias' Hit Rate

In this section we take a closer look at the hit rate of Dias. In particular, we wish to answer the question: How frequently does Dias rewrite a cell?

For our analysis, we utilized a subset of KGTorrent [37], a dataset of Jupyter notebooks harvested from Kaggle. This dataset contains notebooks that are outside the scope of our work, like notebooks which do not use pandas at all. After filtering such notebooks out, we extracted *a total of 8,853 notebooks and 177,272 cells*. We cannot run these notebooks because: (a) we found no reliable way to automatically download a notebook's datasets and (b) even with a dataset, many notebooks still don't run without manual modification. Therefore, we perform a static analysis. In particular, we run the pattern matcher over the notebooks and when it matches a rule, we consider it a "hit". Dias' hit rate is 3.2% (5,586 cells) across cells and 27.1% (2,395 notebooks) across notebooks.

In our evaluation, we used 20 notebooks with 652 cells. Dias rewrites in 2% of all cells (5% if we discard cells that run for less than 50ms) and 50% of notebooks. Because 652 cells is a large sample, and because it agrees with the static analysis, we contend that Dias's hit rate is close to 2-3%.

This result is significant considering that Dias currently uses only 12 rules, which is a small set of rules for an automatic rewriter. Production rewriters have hundreds of rules. For example, TensorFlow r1.14 includes 155 rewrite rules [20] (which are also simpler), developed over a long period of time by many engineers and totalling around 53 thousand lines of code. We emphasize

---

[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 1 notebook.

```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()`

Fig. 18. VectorizedConditionals: Vectorized `apply()` with conditions, which can be hundreds of times faster [7]. However, performing this rewrite automatically is challenging.

that the novelty of DIAS is in the system, not the specific rules we happen to use at this snapshot. With DIAS, we wish to encourage such a development of rules for ad-hoc EDA workloads.

6.5 Understanding DIAS' Performance

To understand the performance gains of DIAS, we discuss two case studies in detail.

Vectorized Conditionals. We further study two case studies, starting with a rule named `VectorizedConditionals`.

We show an example of rewriting a pandas `apply()` function with numpy's `np.select()` in Figure 18 [7]. 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 [7], 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()` in 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.

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

```
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']
```

Fig. 19. `Series.str.split()` implementation (Simplified)

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 `pandas`.

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 on-demand dynamic checking.

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, an 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, as shown in Figure 4 (In DIAS, this is covered by the rule `SplitInPython`; see Table 1). 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 [30].

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 19, 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`), which is expensive.

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

(a) Example of a column-wise operation: Add Two Series Element-Wise

```
np.where(pandas_df['A'] < pandas_df['B'], 10, 20)
```

(b) Example of interacting with numpy: Vectorized conditional

Fig. 20. Examples of operations performed with the pandas alternatives. These are pretty fast with pandas.

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.

6.6   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 [31] (version 2022.12.1), Koalas [35] (version 0.32.0), and PolaRS [13] (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 [10]. PolaRS [13] is a pandas replacement (using Rust under the hood), which, however, has a different API.

Setup and Dataset. 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 [45] (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 [36] and in multiple notebooks throughout the Internet [17].

Operations. We tested several common patterns found in pandas workloads, and which are expected to be pretty fast with pandas [6]. In particular, we tested column-wise operations, interacting with numpy, and an iterative access of individual elements. Figure 20 gives examples of the former two. Figure 1 gives an example of the latter.

Results and Discussion. A subset of our results (for a single example of each category) is shown in Table 2. As is evident, bulk-parallel dataframe libraries like modin, dask and Koalas, are not well suited for ad-hoc EDA workloads. We should note, however, that we observed that PolaRS was significantly more judicious with the resources compared to the other three (especially for memory), and its slowdowns are much smaller. However, it can still cause considerable slowdowns (e.g., with the iterative element access) and has a considerably 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.

---

[6]For example, pandas columns are stored as numpy arrays and many pandas operations use numpy. So, we expect interacting with numpy to be quite efficient.

	pandas	MODIN	dask	Koalas	PolaRS
Column Operations	1x	55.1x	136.8x	9.4x	7.6x
Interaction with numpy	1x	18.3x	179.8x	OOM Failure	1.4x
Individual accesses	1x	1914x	API Incompat.	155x	20.8x

Table 2. *Slowdown* when running common pandas operations with pandas alternatives. OOM Failure is out-of-memory failure and API incompatibility means that an API we used is not supported. As the results show, the current alternatives are not suited to ad-hoc, EDA workloads, both in terms of efficiency and usability.

7   RELATED WORK

Optimizing Pandas. Most previous work on optimizing dataframe libraries focuses on optimizing pandas, mainly through the use of parallel and distributed execution. Systems like modin [36], dask [31], Koalas [35], PolaRS [13], Ponder [4], PolyFrame [43] and Magpie [21] are all essentially custom versions of pandas (some are full rewrites, while others implement the pandas API over some underlying system). Similarly, even techniques like BELE [53], whose optimization targets the pandas-Python interface, are limited *within* the library's boundaries, which weakens the view and control of the surrounding code. In contrast to all these techniques, DIAS is the first system to use dynamic rewriting at the pandas-Python interface, and it does so *externally*, which lets it views all user's code compared to just library code and can modify any part of it. This is the main conceptual difference, but there are also other practical drawbacks as we outlined earlier, mainly arising from the fact that these systems do not focus on single-machine, ad-hoc, diverse use cases.

There is also previous work on optimizing pandas code using static analysis [42], which also utilizes library-specific knowledge. DIAS is different in that it is a dynamic rewriter. Theoretically, the same rules we use could be applied with static analysis, but static analysis in Python is quite inaccurate and it does not fit the read-eval-print-loop (REPL) workflow of EDA workloads.

Pandas for Interactive Settings. A slightly different and interesting line of work focuses on optimizing dataframe queries for interactive workloads [24, 52]. Some of their optimizations include displaying partial results, 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 [22] uses InstCombine [26] and VectorCombine [27] to perform IR rewrites on scalars and vectors respectively. Further, there have been many works such as Alive [28], Alive2 [29], Souper [41] that try to prove or automatically find such rewrites inside traditional compilers. TASO [20] and PET [49] have looked into how rewrites

can be used to optimize tensor computations in tensor compilers. Domain specific languages such as Halide [38] include extensive rewrite engines to perform optimizations [33]. Even complicated optimization passes such as dataflow optimizations [25] and vectorization [8] can be expressed as a series of rewrites. In fact, the compiler infrastructure MLIR [23] 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 [14], V8 [12]) and recently Python [19], but also non-native languages like Java (HotSpot [15]). However, all these methods optimize the host language, focusing on low-level optimizations and not the host-library combination. On the other hand, our technique can perform higher-level (and higher-impact) 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. We implemented DIAS, the first source-to-source, dynamic rewriter for Python, system which rewrites pandas code automatically and transparently, while simultaneously addressing the requirements and constraints of condition-checking.

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 whether it succeeds or not.
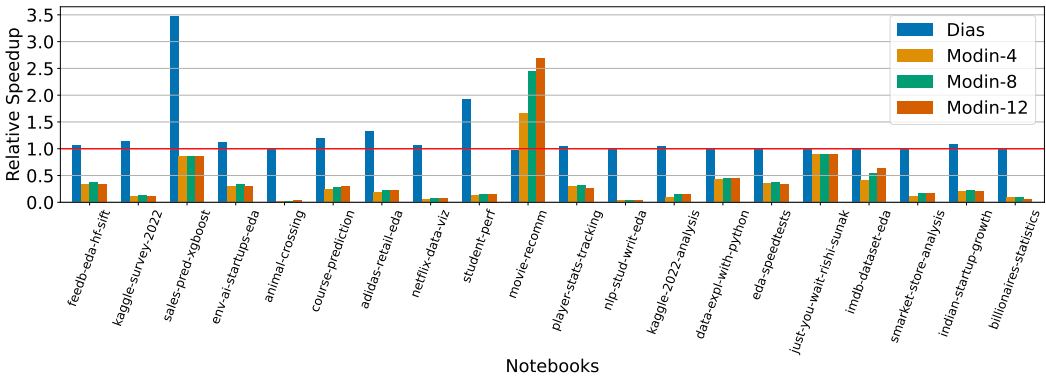
A EXTENDED RESULTS

Fig. 21. Corresponding to Figure 16. The conclusions are similar as modin slows down all the ten new notebooks.
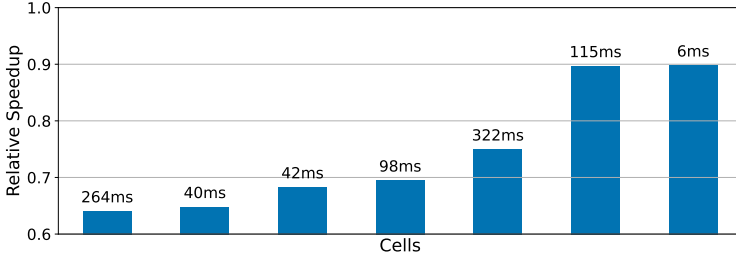
Fig. 22. Corresponding to Figure 15. The conclusions are the same. The slowdowns are within interactive latencies (i.e., less than 300ms).
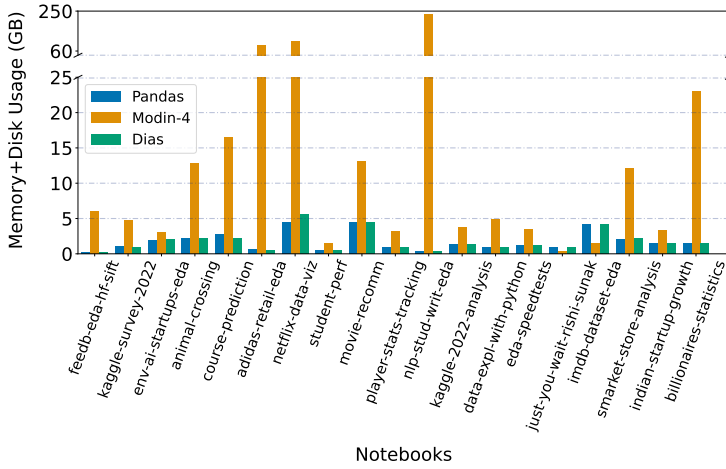

Fig. 23. Corresponding to Figure 17. When we include all 20 notebooks, we see even more aggressive memory+disk usage from modin. Dias and pandas remain on the same scales.

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 25 shows the cell-level speedups, corresponding to Figure 13. 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 0.1× range, there is hardly any discernible difference. Similar observations are derived from Figure 22, 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 1.18× and the maximum slowdown is 1.56×.

Per-Notebook Speedups. In Figure 24 we show the notebook-level speedups. This figure corresponds to Figure 14. 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 1.16× while the maximum slowdown is still 1.03×.
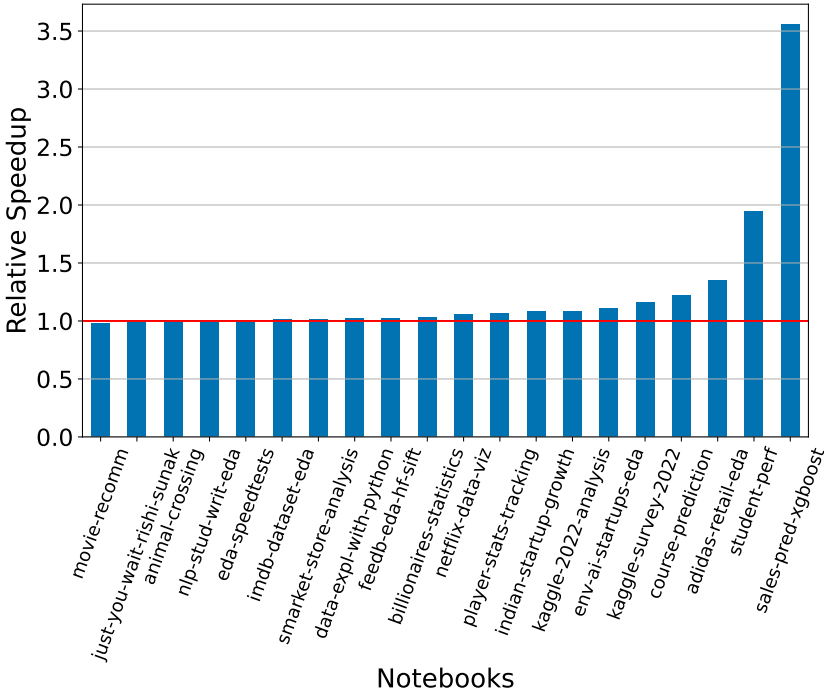
Fig. 24. Relative speedups on whole notebooks. We see the same speedups as in Figure 14 with no extra substantial slowdowns when considering all 20 notebooks.
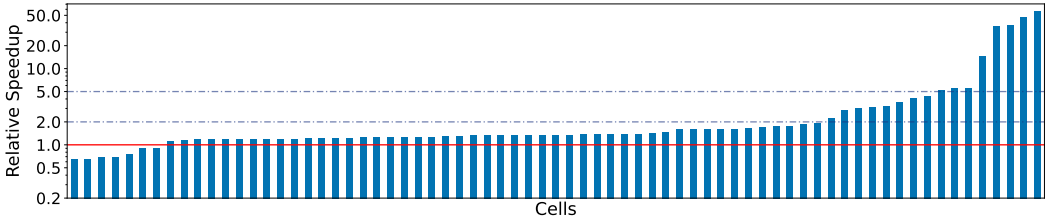

Fig. 25. Cell-level relative speedups (excluding cells that originally ran for less than 50ms and also all the cells that got a speedup or slowdown within the 0.1× range) for all 20 notebooks. Still, DIAS provides significant speedups with no substantial slowdowns (see Figure 22).

Comparison with Modin. In Figure 21 [7], 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.

In Figure 23[8] we show the memory consumption 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.

---

[7]which corresponds to Figure 16.
[8]which corresponds to Figure 17.

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

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

[8] 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

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

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

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

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

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

[14] 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

[15] 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

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

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

[18] IPython: Custom input transformation. 2022. https://ipython.readthedocs.io/en/stable/config/inputtransforms.html#string-based-transformations. Accessed: 2023-05-30.

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

[20] 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

[21] 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*.

[22] 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

[23] 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*.

[24] 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

[25] 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

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

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

[28] 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/

[29] 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

[30] 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.

[31] 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

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

[33] 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

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

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

[36] 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

[37] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2021. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 550–554. https://doi.org/10.1109/MSR52588.2021.00072

[38] 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

[39] 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

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

[41] 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

[42] Bhushan Pal Singh, Mudra Sahu, and S. Sudarshan. 2021. Optimizing Data Science Applications Using Static Analysis. In *The 18th International Symposium on Database Programming Languages* (Copenhagen, Denmark) *(DBPL '21)*. Association for Computing Machinery, New York, NY, USA, 23–27. https://doi.org/10.1145/3475726.3475729

[43] 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

[44] 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.

[45] 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.

[46] TensorFlow. 2023. TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization.

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

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

[49] 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

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

[51] 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.

[52] 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

[53] Guoqiang Zhang and Xipeng Shen. 2021. Best-Effort Lazy Evaluation for Python Software Built on APIs. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:24. https://doi.org/10.4230/LIPIcs.ECOOP.2021.15