Partial Evaluation for Pandas in Python

Xavier Adettu
University of Wisconsin – Milwaukee
Milwaukee, USA
xadettu@uwm.edu

Tian Zhao
University of Wisconsin – Milwaukee
Milwaukee, USA
tzhao@uwm.edu

Abstract

Pandas programs often perform more Python level work than necessary due to row-wise `apply` calls, small control loops, and repeated constant computations. These patterns break vectorization and introduce significant interpreter overhead on large datasets. We present a lightweight partial evaluation framework for Python that automatically simplifies Pandas code before execution. The system uses a compact Pandas-oriented DSL, a two-point binding-time analysis to separate static and dynamic expressions, and an online interpreter-based specializer to execute static work early and emit optimized residual code. The prototype performs constant folding, loop unrolling, UDF inlining, and `apply` fusion. Across synthetic workloads, the specialized programs achieve up to 112× speedups on `apply`-heavy pipelines while preserving full Pandas semantics.

CCS Concepts

• **Software and its engineering** → **Source code generation**; **Software maintenance tools**.

Keywords

Partial Evaluation, Pandas, Python, DSL, Optimization

ACM Reference Format:
Xavier Adettu and Tian Zhao. 2026. Partial Evaluation for Pandas in Python. In *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26), March 23–27, 2026, Thessaloniki, Greece.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3748522.3779923

1 Introduction

Pandas library is widely used for data cleaning, feature engineering, and exploratory analysis in Python [6, 9]. While many operations are internally vectorized, real-world workloads frequently use row-wise `apply` calls, small loops over fixed ranges, or repeated constant expressions. These patterns force Python-level interpretation and limit performance, especially on large datasets [11].

Partial evaluation (PE) [3, 5, 7] precomputes expressions that depend only on static inputs, leaving a residual program with only dynamic computation. Because many Pandas pipelines contain static elements, such as literal lists, loop bounds, and column names, PE can simplify computation significantly before a DataFrame is even touched.

Modern DataFrame accelerators such as Dask [15], Modin [12], Vaex [2], Polars [13], and cuDF [14] improve execution or memory layout, but they assume the program is written in a vectorized style.

Our goal is complementary, which is to transform un-vectorized Pandas code into an efficient vectorized form before it runs.

In this paper, we introduce a lightweight PE framework that (1) parses a Pandas-oriented Domain Specific Language (DSL) into a normalized Abstract Syntax Tree (AST), (2) performs binding-time analysis (static vs. dynamic), and (3) interprets the program online to evaluate static work early and produce optimized residual Pandas code. This paper focuses on lightweight specialization of common Pandas patterns and does not aim to cover the full Python language. We target practical pre-processing workloads where static structure and small control loops are prevalent.

In the rest of the paper, we motivate this research in Section 2, explain the approach in Section 3, show our experimental result in Section 4, and discuss related works in Section 5. The code and data of this paper are available at https://github.com/uwm-se/PE.

2 Motivation

Pandas scripts commonly mix two kinds of information:

- **Static context:** constants, literal lists, and small ranges (e.g. `range(3)`).
- **Dynamic data:** DataFrame contents and values known only at runtime.

This mixture leads to unnecessary interpretation overhead.

- **Row-wise overhead:** `Series.apply` calls invoke a Python callback per row.
- **Tiny static loops:** static loops iterate over small fixed ranges unnecessarily.
- **Repeated constants:** expressions like `"T" + str(i)` or `df["x"] * 3` repeat static work.

For example, the code below includes a small loop and applies a User-Defined Function (UDF) to a DataFrame column.

```
def square(x): return x*x
for i in range(2):
    df[f"Z{i}"] = df["POP"] + i
df["sq"] = df["COUNT"].apply(square)
```

After optimization, the loop is unrolled and the UDF is converted into a vectorized expression with no user annotations.

```
df["Z0"] = df["POP"] + 0
df["Z1"] = df["POP"] + 1
df["sq"] = df["COUNT"] * df["COUNT"]
```

3 Approach

Our framework processes input programs in three steps.

3.1 AST Normalization

We define a small DSL with assignments, expressions, column reads/writes, simple functions, `Series.apply`, and basic control flow (`if`, `for`, `while`). The DSL parser converts the program into a uniform AST that makes the analysis and specialization deterministic.

For example, below are a column update and its AST node.

```
df["sq"] = df["x"].apply(square)

("col_op", "df", "sq",
  ("method_call", ("col_access", "df", "x"),
    "apply", [("var", "square")]))
```

3.2 Binding-Time Analysis (Static vs Dynamic)

A two-point lattice $\{\mathbf{S}, \mathbf{D}\}$ determines when an expression is known during specialization: $\mathbf{S} \sqsubset \mathbf{D}$. Static expressions (constants, literal lists, static loop bounds) become **S**. Any dependency on DataFrames or Series becomes **D**. Static loops and branches may be executed immediately, while dynamic ones remain in the residual program.

3.3 Online Partial Evaluation

We use an interpreter-driven online PE strategy [5]. During the traversal of the AST, we perform the following tasks:

- Static expressions are evaluated immediately.
- Dynamic expressions become residual code.
- Static loops are unrolled.
- Pure single-expression UDFs used in `apply` are inlined.
- Function calls with static arguments are specialized.

The result is clean and executable Pandas code with the same semantics but less Python-level overhead.

3.4 Design Principles

Our design follows three principles that make partial evaluation effective for Pandas:

1. *Deterministic rewriting.* Transformations are applied only when binding-time information guarantees correctness, avoiding unpredictable changes.
2. *Separation of concerns.* The DSL frontend normalizes syntax, while the interpreter performs specialization. This keeps each component small and modular.
3. *Readable output.* The residual code remains standard Pandas, enabling users to inspect, debug, or further optimize using existing tools.

Properties. The analysis is flow sensitive at the statement level, while path insensitive overall. Recursion is disallowed in the DSL, ensuring predictable specialization. Loop unrolling is explicitly bounded, and all dynamic control flow is residualized, guaranteeing termination of the specializer.

3.5 Pandas-Specific Optimizations

Constant Folding. Static arithmetic, range expressions, string concatenation, such as the one below, are computed early.

```
tag = "T" + str(7)   ->   tag = "T7"
```

Loop Unrolling. Loops with static bounds (e.g. `range(3)`) are expanded to straight line assignments.

UDF Inlining for `Series.apply`*.* Single expression UDFs like `lambda v: v*v` become vectorized Pandas expressions.

Apply Fusion. Consecutive `apply` operations are fused:

```
df["x"] = df["x"].apply(f1).apply(f2)
-> df["x"] = df["x"].apply(lambda v: f2(f1(v)))
```

Function specialization. When a function is called with one or more static arguments, the specializer creates a specialized version in which these static values are substituted directly into the function body. This avoids repeated interpretation of constant parameters and exposes further optimization opportunities, such as constant folding and dead code elimination.

To prevent unbounded creation of specialized versions (a classical problem in online partial evaluation), we maintain a memoization table keyed by the tuple of static argument values. If an identical specialized instance was previously generated, the specializer reuses it rather than creating a new one. This polyvariant caching strategy ensures both termination and predictable specialization behavior, following standard techniques described by Jones *et al.* [7] and Cook and Lämmel's online PE model [5].

For example, function call `scale_and_tag(df1,2,7)`, as shown below, yields a specialized function `def scale_and_tag__S1(df)` where 2 and 7 are embedded as constants. Further calls with the same static arguments reuse the cached specialized definition.

```
# Original
def scale_and_tag(df, scale, tag):
    df["VAL"] = df["VAL"] * scale
    df["TAG"] = "T" + str(tag)
# Call site
scale_and_tag(df1, 2, 7)

# Specialized (cached) version
def scale_and_tag__S1(df):
    df["VAL"] = df["VAL"] * 2
    df["TAG"] = "T7"
# Reused for identical static arguments
scale_and_tag__S1(df1)
```

4 Evaluation

We evaluate six micro-benchmarks and one macro-benchmark on Python 3.11.14 and Pandas 2.3.3 on a Linux server (x86_64, kernel 6.14.0-1014-azure) running on AMD EPYC 9004 CPU. The micro-benchmarks test constant folding, loop unrolling, UDF lifting, `apply` fusion, function specialization, and column assignment. The macro-benchmark is a `groupby` and aggregation pipeline over semi-synthetic event data. Table 1 reports, for each benchmark and input size, the original running time T_{orig} (in seconds) and the speedup $T_{\text{orig}}/T_{\text{resid}}$ obtained by partial evaluation.

The six micro-benchmarks correspond to the transformations implemented by our system. These patterns reflect the most common sources of Python-level overhead in real Pandas workloads. The Pandas performance guide explicitly identifies row-wise `apply`

Table 1: Original running time T_{orig} (seconds) and speedup ($SU = T_{\mathrm{orig}}/T_{\mathrm{resid}}$) for six microbenchmarks and one groupby macrobenchmark.

Benchmark	10^5 rows		5×10^5 rows		10^6 rows	
	T_{orig}	SU	T_{orig}	SU	T_{orig}	SU
Constant folding	0.0015	1.04	0.0048	1.03	0.0039	1.02
Loop unrolling	0.0031	1.05	0.0059	1.01	0.0041	0.98
UDF lifting	0.31	321	1.57	1092	2.89	1399
`apply` fusion	0.61	1.78	2.70	1.58	5.68	1.78
Function special.	0.0012	1.10	0.0022	1.02	0.0037	1.02
Column assign.	0.0012	0.78	0.0034	0.80	0.0054	0.78
Groupby & agg.	0.069	4.09	0.32	5.66	0.64	5.67

calls, small Python loops, and repeated constant computations as typical bottlenecks [11]. Empirical studies of real notebooks similarly report heavy use of per-row UDFs and non-vectorized control flow in everyday data preparation [1, 10], consistent with earlier observations about Pandas usage patterns [9].

Each of the *microbenchmarks* isolates a single transformation in isolation while the `groupby` macrobenchmark exercises PE on the Python-level filtering and weighting logic surrounding an already vectorized `groupby` operator and serves as a sanity check that specialization does not regress performance on library-optimized kernels. At 10^6 rows, for example, UDF lifting reduces runtime from 2.89 s to 0.00207 s (a 1399× speedup), while the groupby aggregation drops from 0.64 s to 0.11 s (a 5.67× speedup) after specialization.

5 Related Work

Classical PE foundations come from Jones, Gomard, and Sestoft [7], Consel and Danvy [3], and Cook and Lämmel [5]. Staged interpretation and lightweight modular staging [8, 16] demonstrate efficient specialization in dynamic languages.

Dataframe accelerators such as Dask, Modin, Vaex, Polars, and cuDF [2, 12–15] improve execution but assume well vectorized code. Our work complements these by rewriting unvectorized Pandas scripts automatically. In addition to classical PE techniques, hybrid or selective partial evaluation approaches have been proposed to balance static analysis with online decisions, notably Shali and Cook's hybrid PE framework [17] and the Tempo system for controlled specialization of real programs [4].These systems emphasize predictable specialization and effect awareness, ideas that influence our design of bounded loop unrolling, purity checks for UDF inlining, and memoized function specialization.

Recent work on Pandas optimization also explores transparent rewriting. Dias [1] dynamically rewrites Pandas pipelines at runtime, primarily targeting expression fusion and projection pruning. Our approach is complementary, whereas Dias focuses on execution time rewrites, our method removes Python level overhead *before* execution through static specialization.

6 Conclusion

We presented a lightweight partial evaluator for Pandas that simplifies Python level operations through constant folding, loop unrolling, UDF inlining, apply fusion, and function specialization. The residual program preserves Pandas semantics and remains readable, while achieving substantial performance gains on `apply`-heavy workloads. Our DSL only covers a small subset of Python. Features such as classes, decorators, generators, and closures with side effects are intentionally excluded to maintain predictable binding-time analysis. UDF inlining is limited to pure, single expression functions. Complex Pandas operators, including joins, grouped aggregations, and window operations, require additional effect and alias analysis. As future work, we plan to extend our DSL to cover richer Pandas behaviors, incorporate optional user annotations for advanced transformations, and integrate with engines such as Dask and Modin to enable backend-aware specialization.

Acknowledgments

This work was partially supported by the Northwestern Mutual Data Science Institute (NMDSI) under grant number SS184.

References

[1] Stefanos Baziotis, Daniel Kang, and Charith Mendis. 2024. Dias: Dynamic Rewriting of Pandas Code. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 1, Article 58 (2024), 27 pages. doi:10.1145/3639313 Published February 2024.

[2] Maarten A. Breddels and Jovan Veljanoski. 2020. Vaex: Big Data Exploration in Python with Efficient Out-of-core Algorithms. *Astronomy and Computing* 32 (2020), 100384.

[3] Charles Consel and Olivier Danvy. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 493–501. doi:10.1145/158511.158707

[4] Charles Consel, Julia L. Lawall, and Gilles Muller. 2004. A Tour of Tempo: A Program Specializer for the C Language. *Science of Computer Programming* 52, 1–3 (2004), 341–370. doi:10.1016/j.scico.2004.03.011

[5] William R. Cook and Ralf Lämmel. 2011. Tutorial on Online Partial Evaluation. In *IFIP Working Conference on Domain-Specific Languages (EPTCS 66)*. 81–104. doi:10.4204/EPTCS.66.8

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. Array Programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[7] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall. Freely available author edition.

[8] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. In *Proceedings of the 2018 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

[9] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference (SciPy)*. 56–61. doi:10.25080/Majora-92bf1922-00a

[10] Angelo Mozzillo, Luca Zecchini, Luca Gagliardelli, Adeel Aslam, Sonia Bergamaschi, and Giovanni Simonini. 2023. Evaluation of Dataframe Libraries for Data Preparation on a Single Machine. arXiv:2312.11122.

[11] Pandas Developers. 2023. Pandas User Guide: Enhancing Performance. https://pandas.pydata.org/docs/user_guide/enhancingperf.html. Accessed: 2025-09-11.

[12] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2033–2046. doi:10.14778/3407790.3407807

[13] Polars Developers. 2022. Polars: Fast DataFrame library written in Rust. https://www.pola.rs/.

[14] RAPIDS AI. 2018. RAPIDS cuDF: GPU DataFrame Library for Python. https://rapids.ai/.

[15] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference (SciPy)*. 130–136. doi:10.25080/Majora-7b98e3ed-013

[16] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compilation. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*. 127–136. doi:10.1145/1868294.1868314

[17] Amin Shali and William R. Cook. 2011. Hybrid Partial Evaluation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 375–390. doi:10.1145/2048066.2048098