P2D: A Transpiler Framework for Optimizing Data Science Pipelines

Yordan Grigorov[*]
Cisco Systems
dgrigoro@cisco.com

Haralampos Gavriilidis
Technische Universität Berlin
gavriilidis@tu-berlin.de

Sergey Redyuk[*]
DFKI
sergey.redyuk@dfki.de

Kaustubh Beedkar
Indian Institute of Technology Delhi
kbeedkar@cse.iitd.ac.in

Volker Markl
Technische Universität Berlin, DFKI
volker.markl@tu-berlin.de

ABSTRACT

In this paper, we propose a transpilation-based approach to optimize data science pipelines that comprise database management systems (DBMSes) and data science runtimes (e.g., Python). Our approach allows to identify DBMS-supported operations and translate them into SQL to leverage DBMSes for accelerating data science workloads. The optimization target is twofold: First, to improve data loading, by reducing the amount of data to be transferred between runtimes. Second, to exploit DBMS processing capabilities by "pushing down" certain pre-processing operations. Our optimizations are based on an intermediate representation, which allows supporting different data science libraries and DBMSes as frontends and backends respectively, making it suitable for different data science pipelines. Our evaluation with real-world and synthetic datasets shows that our approach can accelerate data science workloads by up to an order of magnitude over state-of-the-art approaches.

ACM Reference Format:
Yordan Grigorov, Haralampos Gavriilidis, Sergey Redyuk, Kaustubh Beedkar, and Volker Markl. 2023. P2D: A Transpiler Framework for Optimizing Data Science Pipelines. In *Data Management for End-to-End Machine Learning (DEEM '23), June 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3595360.3595853

1 INTRODUCTION

Nowadays, the two most popular languages among data scientists are Python and R. These languages offer a plethora of libraries, packages, and tools that are specifically designed for data science tasks including data manipulation, analysis, and model training. While these languages offer convenient APIs, building efficient data science pipelines (DSPs) for large datasets remains a challenge, as processing and analyzing large datasets requires special expertise.

Database management systems (DBMSes) are important components of DSPs, as they provide storage and querying capabilities for large datasets. However, DBMSes are underutilized, as data scientists often use them only as storage backends and use a data science runtime to carry out data science tasks. For example, consider a DSP for predicting housing prices in Berlin using the popular Python Pandas and sklearn libraries, with historical data stored in a PostgreSQL DBMS. Listing 1 illustrates the DSP: First, we load the real-estate data from PostgreSQL into Pandas dataframes (lines 3–6). Then, in lines 8–12 we perform data manipulation tasks including filtering certain rows and columns and joining different datasets. Finally, in lines 14–17 we train a linear regression model.

Even though the above example DSP seems straightforward from a data scientist's perspective, it underutilizes PostgreSQL leading to suboptimal performance. This is because: 1) using a DBMS only as a storage backend leads to high data movement cost, as raw data is moved between different runtimes, and 2) data science runtimes are less efficient for relational operations (e.g., filters, projections, or joins), which are common during pre-processing tasks.

To achieve better performance, data scientists must implement their DSPs with multiple programming abstractions and leverage underlying DBMSes' processing capabilities. In the above example, this would require implementing the preprocessing tasks in SQL and the remaining tasks in Python, as shown in Listing 2. This approach, however, has the drawback that data scientists get exposed to multiple APIs. Furthermore, DSPs become less maintainable. To avoid this, users tend to sacrifice efficiency for convenience.

A key challenge in optimizing DSPs is to offer convenient user abstractions while at the same time optimizing the execution by automatically offloading certain tasks to DBMSes. This is, however, far from trivial as data science runtimes cannot holistically optimize DSPs over DBMSes. Moreover, state-of-the-art approaches for optimizing DSPs propose to either scale out the execution [14, 16], or to expose alternative APIs [7, 10, 19, 22] that enable holistic optimizations. While both approaches improve runtime performance, neither of them completely addresses the challenge of keeping existing user interfaces and transparently optimizing the execution.

In this paper, we propose P2D, which is a source-to-source transpiler for optimizing DSPs. P2D allows data scientists to express their tasks in their favorite data science language and automatically translates user source code to an optimized version. For example, P2D can automatically translate the Python code in Listing 1 to the code shown in Listing 2. P2D depends on system-specific mappings that allow for translating DSPs to our intermediate representation (IR), which captures the semantics of the data science operations. Such an IR allows us to reason holistically about and optimize DSPs through static code analysis techniques. P2D achieves

[*]Work partially done while author was at Technische Universität Berlin

```
# Skipping import statements for brevity
# Connect to database
engine = create_engine('postgresql://user:password@host:port/dbname')
# Load data from the database
h_df = pd.read_sql('SELECT * FROM housing', engine)
s_df = pd.read_sql('SELECT * FROM schools', engine)
# Preprocessing
h_df = h_df[['num_bedrooms', 'median_income', 'price']]
h_df['price'] = h_df['price'].apply(lambda x: x*1.07)
h_df = h_df[h_df['num_bedrooms'] > 3 && h_df['city'] == 'Berlin']
num_schools = s_df.groupby('zipcode')['school_id'].nunique()
data = h_df.merge(num_schools.rename('num_schools'), on='zipcode')
# Fit linear regression model
X = data.drop('price', axis=1)
y = data['price']
model = LinearRegression()
model.fit(X, y)
```

Listing 1: Unoptimized Python script

```
# Skipping import statements for brevity
# Connect to the database
engine = create_engine('postgresql://username:password@host:port/dbname')
# Load data from the database with pre-processing included
query = """ SELECT   h.num_bedrooms, h.median_income,
                     COUNT(s.school_id) AS num_schools, h.price
            FROM     housing h, schools s
            WHERE    h.zipcode = s.zipcode AND h.num_bedrooms > 3 AND
                     h.city = 'Berlin'
            GROUP BY h.num_bedrooms, h.median_income, h.price """
data = pd.read_sql(query, engine)
data['price'] = data['price'].apply(lambda x: x*1.07)
# Fit linear regression model
X = data.drop('price', axis=1)
y = data['price']
model = LinearRegression()
model.fit(X, y)
```

Listing 2: Optimized Python script

this optimization by re-ordering operations and translating the IR (using the system-specific mappings) to an optimized DSP.

Our evaluation of P2D using real-world and synthetic datasets showed that DSPs produced by P2D outperform state-of-the-art approaches [16, 21] by up to an order of magnitude. Furthermore, we conducted a case study where we analyzed over 45,000 publicly available Python DSPs. We found that 85% of Pandas functions can be translated to SQL, which demonstrates P2D's potential impact.

2 P2D: DATA SCIENCE PIPELINE TRANSPILER

In this section, we describe P2D, a transpiler framework to bridge the gap between expressive data science APIs (e.g., Pandas) and performant query processors (e.g., DBMSes). P2D takes DSP source code as input (e.g., Listing 1) and translates it into an optimized representation, which is again DSP source code (e.g., Listing 2). Figure 1 shows the framework overview, comprising three main modules. First, the `Code Preprocessor` translates the DSP code into an abstract syntax tree (AST). Then, the `Statement Aggregator` transforms the AST into an intermediate representation (IR), which the `Optimizer` and `Code Generator` transform to optimized DSP code.

2.1 Code Preprocessor

The key idea behind our approach is to use static code analysis to identify DSP operations that DBMS backends can support. To conveniently reason about supported operations on a canonical representation of the user code, we transform the code into the Administrative Normal Form (ANF). During this transformation we do several steps to simplify our representation, such as removing redundant syntactic sugar and constructs (e.g., spaces, braces, newlines, and indentations), type inference, and "normalizing" function calls with default parameters to identify function calls that "look" different but are the same. For example, in Pandas a projection

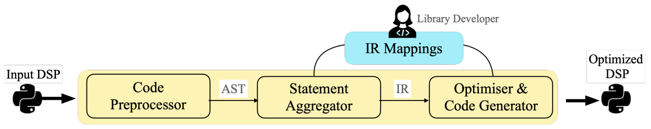


Figure 1: P2D Overview

on a dataframe can be expressed both as `df[df['col']]` and as `df.loc[df['col']]`. The output of our preprocessing phase is an AST for the Statement Aggregator.

2.2 Statement Aggregator

After bringing the code to a canonical representation, the Statement Aggregator builds a program in our IR. Having an IR next to our (in our example Python) AST, helps to i) easily support new frontends and backends, and ii) to apply optimizations, e.g., by reordering operations. We base our IR on relational algebra, since preprocessing tasks in DSPs often consist of filtering rows and columns, and joining and aggregating tables. In particular, our IR is a directed acyclic graph (DAG) where nodes are processing operations and edges are dependencies between operations. We further distinguish processing operations as either *supported* or *unsupported*. Supported operations refer to those that can be executed by the underlying DBMS, while unsupported operations cannot be performed by it.

Supported operations are operations that library developers define through *frontend mappings*. The Statement Aggregator traverses the AST and tries to match frontend mappings to create the corresponding operation in our IR. Our approach is extensible in that library developers can provide mappings of library functions and their arguments to our IR. For example, Listing 3 shows an example of mapping a projection from the Python Pandas library to our IR. In this case, we map `pandas.DataFrame`'s method `__getitem__` with its key (attributes) to a `PROJECTION` primitive.

```
"req":{
    "__comment": "df[['col1', 'col2']]",
    "parenttype": "pandas.DataFrame",
    "attrname":"__getitem__",
    "kwarg_types":{"key":"list"}
},
"maps":{
    "ir":"PROJECTION({key},{__instance})",
    "type":"pandas.DataFrame",
    "state": []
}
```

Listing 3: Frontend mapping for Pandas projection

All operations not having a specified frontend mapping are translated to unsupported operations in the IR. Essentially, unsupported operations are treated as black boxes where we only keep track of their properties and dependencies. The final output of the Statement Aggregator is a program in our IR, subject to optimizations.

2.3 Optimizer and Code Generator

Having a program in our IR allows us to reorder operations and generate the optimized DSP source code, as we are aware of its semantics. The benefit of having a relational algebra based IR is that we can reuse its rules for rewrites and equivalence. Our Optimizer's goal is to push down as many supported operations below the unsupported ones, such that we get a DSP like in Listing 2. To ensure safe reorderings with unsupported black-box operations, we rely on existing approaches employing static code analysis [9].

We note that our optimizations target reordering supported and unsupported operations and not blocks of relational operations.

After optimizing our IR program, the `Code Generator` transforms it back to an AST, to then generate the optimized DSP source code. For that, the Code Generator transforms the statements corresponding to supported operations into DBMS-specific statements. For this, we rely on developer-specified *backend mappings*. Like their frontend counterparts, backend mappings map IR operations to DBMS-specific instructions. For example, Listing 4 shows an example for mapping the projection and selection operators from our IR to SQL statements.

```
def PROJECTION(cols: list, source: str):
    return f"SELECT {', '.join(cols)} FROM {_wrapas(source)}"

def SELECTION(column, operator, operand, condition: str, source: str):
    return f""" SELECT * FROM {_wrapas(source)}
                WHERE {column} {operator} {operand}"""
```

Listing 4: Backend mapping for projection to SQL

After our transformations, the generated source code includes all offloaded DBMS operations within the data loading part. In this case, we output the supported operations in the language of the DBMS backend, in this case, SQL (Listing 2: lines 5–10, unnested for simplicity). Executing this DSP will "push down" all supported operations to the DBMS and execute only the remaining ones in Python. This leads to performance improvements, as we both reduce data transfer (in most cases) and at the same time leverage efficient compute kernels, indexes, and other optimizations available in DBMSes.

3 PRELIMINARY EXPERIMENTS

We have conducted preliminary experiments to evaluate the performance benefits of our P2D transpiler in the context of exploratory data analysis and DSPs with pre-processing and model training.

Hardware & Software Setup: We have used a machine running Ubuntu 22.04, with an Intel i7-8550U CPU and 32GB of memory. We have implemented our prototype in Python, and provided frontend mappings for 11 Pandas operations, and backend mappings for projection, selection, join, and aggregates in SQL (for PostgreSQL 13).

Baselines: We have considered native Pandas (v1.5.2) and Modin [16] (v0.18 on Ray v2.3), a "drop-in replacement for Pandas". Its multithreaded engine with the provided re-implementation of Pandas operators, let Modin scale out the loading and execution. Furthermore, we have considered ConnectorX [23] (v0.3.1), a library that optimizes data movement between Pandas and DBMSes by parallelizing operations and reducing data copies. Moreover, we have considered SCIRPy [21], an approach that "pushes down" projections on Pandas dataframes to loading methods to improve performance. In our experiments, we simulate SCIRPy by queries with pre-defined projections to the DBMS through ConnectorX, to even optimize further.

Exploration and Decision Support: We first considered DSPs based on TPC-H queries (scale factor 1), as it resembles data warehousing and exploration scenarios. Here, our goal was to see the "maximum" we can accelerate by pushing everything to a DBMS. Therefore, we have re-implemented queries 1-10 on the Pandas API. We show the results in Figure 2; here P2D pushes down all operations to the underlying DBMS. This leads to up to one order of magnitude performance improvements, compared to the native pandas implementation. Modin and ConnectorX drastically improve loading by employing parallelism and avoiding redundant data copies, however, do not match P2D's performance, as operations in the Python runtime are not as efficient as in PostgreSQL. While for some queries, e.g., Q1 and Q6, our baselines spent almost the same amount of execution time as the DBMS backend, other queries, e.g. Q7–Q10 show that PostgreSQL is more efficient for the execution. Overall, these experiments suggest that leveraging a DBMS for supported operations is indeed beneficial and can accelerate DSPs, both because of reduced data movement and efficient execution.

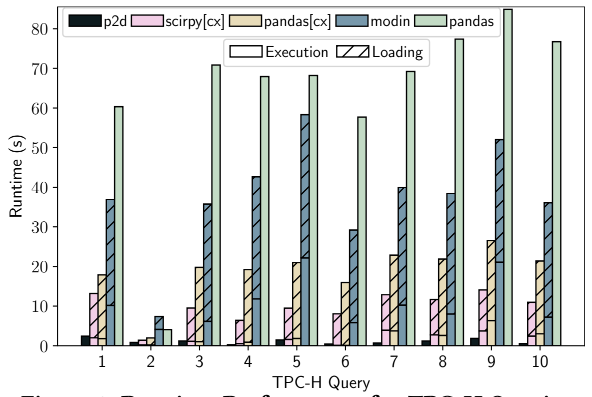
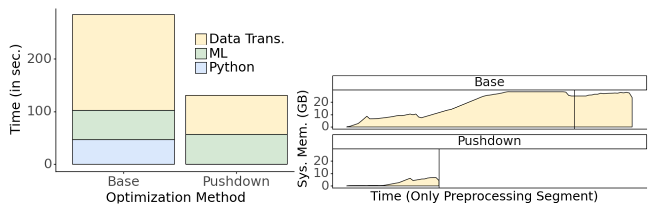


Figure 2: Runtime Performance for TPC-H Queries



(a) Runtime Performance **(b) Memory consumption**

Figure 3: Kaggle Module 4 Pipeline

Preprocessing with Model Training: We have also considered a real-world DSP from a Kaggle competition [17] (c.f. Section 4), which uses Twitter hashtags, sentiment scores, and music streaming session data (total 3GB) from a music streaming application. The DSP first loads, preprocesses, and joins the three datasets before building a linear regression model that predicts the next song.

We show the results in Figure 3. Here, we measured three different phases: Data loading (*Data Trans.*), preprocessing in *Python*, and finally executing the model training (*ML*). We observe that P2D reduces both the preprocessing in Python and the data transfer times. This results in a performance improvement of 2.5×. We have also measured the memory utilization: While native Pandas leads to a consumption of more than 20GB, P2D reduces it more than 2×.

4 CASE STUDY

To further understand the potential impact of our transpilation-based approach, we have analyzed the source code of 45, 147 Python DSPs on Kaggle. Our case study (cf. Table 1) showed that more than 98% of the DSPs use Pandas, a library for manipulating tabular data, closely followed by Numpy (84.5%), a library for linear algebra.

Furthermore, we have analyzed the occurrence of function calls in the aforementioned kernels and compiled a table of the names

Table 1: Libraries

Library	Used Perc.
pandas	98.54%
numpy	84.55%
matplotlib	67.13%
sklearn	53.79%
seaborn	39.83%
mpl_toolkits	16.30%
learntools	11.40%
plotly	9.34%
scipy	7.98%
tensorflow	6.38%

Table 2: Function Calls

Function	#	Function	#	Function	#
__getitem__	1534k	join	30k	set	16k
head	104k	dropna	29k	describe	15k
len	96k	apply	28k	split	15k
read_csv	78k	astype	26k	map	15k
range	67k	nunique	24k	copy	15k
sum	59k	fillna	23k	train_test_split	15k
append	57k	corr	23k	zip	15k
groupby	52k	replace	22k	arange	14k
fit	50k	max	22k	walk	14k
value_counts	49k	round	20k	concat	14k
predict	43k	unique	20k	reshape	13k
drop	39k	isnull	19k	info	13k
DataFrame	39k	min	19k	count	13k
mean	37k	add	18k	to_datetime	13k
sort_values	34k	int	18k	merge	11k
format	34k	array	18k	rename	11k
reset_index	32k	fit_transform	16k		

of the 50 most commonly used ones (cf. Table 2). The Table contains 17 function names that overlap between Pandas and other popular Python classes, marked with yellow. 21 of the names are sufficiently unique to classify a function as originating from the Pandas library, marked with green. A cursory examination of the Pandas documentation reveals that a significant proportion (85%) of the green-colored functions can be translated to SQL. This further substantiates our conjecture that the data preprocessing APIs of high-level procedural programming languages are mappable to traditional declarative data query languages. Our findings are on par with a case study [10], which arrived at a similar conclusion but through a different dataset.

5 RELATED WORK

There have been proposed several lines of work to improve the performance of DSPs. We identify three main related lines of work.

Data Science APIs: Several approaches expose new APIs (in the form of DSLs) backed by their intermediate representations. For example, Lara [12], based on Emma [3] allows to interleave collection processing and machine learning operations. Its holistic view enables optimizing the order and choice of physical operators. SystemML [6] employs cost-based optimizations to generate efficient in-memory single-node and large-scale distributed operations, e.g. on Apache Spark. Approaches such as Magpie [10], Ibis [1], MLearn [20], and AFrame [22] propose APIs inspired by dataframes, and offload the computation to scalable execution backends. While these approaches offer convenient user APIs and great optimization potential, they can not optimize DSPs in existing APIs, e.g., Pandas.

In-Database Machine Learning: Another line of work suggests in-DBMS machine learning, where ML operators are implemented within the DBMS [8, 11, 15]. This allows to move computations closer to data and to optimize holistically over ML and DBMS operators, by enhancing existing optimization techniques in DBMSes.

The aforementioned approaches improve runtime performance, however, they all force users to new APIs, require DBMSes extensions, and can not be used to optimize existing DSPs, as they have to be reimplemented in the respective APIs. In contrast, P2D's goal is optimizing existing DBMS-backed DSPs transparently, i.e., by not being intrusive to users nor to DBMSes.

Optimizing existing DSPs: Weld [14], Modin [16], Ray [13], and Koalas [2] accelerate performance by scaling out the execution of DSPs expressed in existing APIs. While these approaches can be used to optimize existing DSPs they do not exploit DBMSes as computation backends and treat them as black-box storage backends.

6 CONCLUSIONS

Data scientists tend to sacrifice performance for simplicity, as they implement DSPs on DBMSes in dataframe-like APIs. This leads to underutilized DBMSes and increased data movement between runtimes, as DBMSes are treated as simple storage backends. In this paper, we have proposed the P2D transpiler that aims to find the sweet spot between usability and performance. P2D optimizes DSPs by offloading preprocessing operations to DBMSes. We employ static code analysis and rely on developer-provided mappings for its IR to transpile DSPs. Our preliminary evaluation has shown that P2D can achieve up to 10× performance improvements over state-of-the-art.

Future Work: Our roadmap includes plans to develop mappings for more frontends (e.g., Numpy and sklearn), to enable optimizations across multiple frontends, and to formalize techniques to guarantee correctness for the IR transformations. We also plan to extend P2D's heuristic optimizer with cost-based methods. Finally, we plan to extend P2D to optimize DSPs with data on multiple DBMSes [5], and to integrate it with the Dorian [18] and PolyDB [4] projects.

ACKNOWLEDGEMENTS

This work was funded by the German Ministry for Education & Research as BIFOLD - Berlin Institute for the Foundations of Learning & Data (01IS18025A; 01IS18037A; BIFOLD23B), and as a Software Campus project (PolyDB, 01IS17052).

REFERENCES

[1] 2023. Ibis. https://github.com/ibis-project/ibis
[2] 2023. Koalas. https://github.com/databricks/koalas
[3] Alexander Alexandrov et al. 2015. Implicit parallelism through deep language embedding. In *SIGMOD*.
[4] Haralampos Gavriilidis et al. 2022. Towards a Modular Data Management System Framework. In *CDMS*.
[5] Haralampos Gavriilidis et al. 2023. In-situ cross-database query processing. In *ICDE*.
[6] Amol Ghoting et al. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*.
[7] Stefan Hagedorn. 2020. When sweet and cute isn't enough anymore: Solving scalability issues in Python Pandas with Grizzly. In *CIDR*.
[8] Joseph M Hellerstein et al. 2012. The MADlib Analytics Library. In *PVLDB*.
[9] Fabian Hueske et al. 2012. Opening the Black Boxes in Data Flow Optimization. In *PVLDB*.
[10] Alekh Jindal et al. 2021. Magpie: Python at speed and scale using cloud backends. In *CIDR*.
[11] Mahmoud Abo Khamis et al. 2018. AC/DC: in-database learning thunderstruck. In *DEEM*.
[12] Andreas Kunft et al. 2019. An intermediate representation for optimizing machine learning pipelines. In *PVLDB*.
[13] Philipp Moritz et al. 2018. Ray: A distributed framework for emerging AI applications. In *OSDI*.
[14] Shoumik Palkar et al. 2018. Evaluating end-to-end optimization for data analytics applications in weld. In *PVLDB*.
[15] Kwanghyun Park et al. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *SIGMOD*.
[16] Devin Petersohn et al. 2020. Towards Scalable Dataframe Systems. In *PVLDB*.
[17] Chelsea Power. 2019. Module 4 Project. https://www.kaggle.com/chelseapower/module4-project/.
[18] Sergey Redyuk et al. 2022. DORIAN in action: assisted design of data science pipelines. In *PVLDB*.
[19] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *SCIPY*.
[20] Maximilian E Schüle et al. 2019. Mlearn: A declarative machine learning language for database systems. In *DEEM*.
[21] Bhushan Pal Singh et al. 2021. Optimizing Data Science Applications using Static Analysis. In *DBPL*.
[22] Phanwadee Sinthong et al. 2019. Aframe: Extending dataframes for large-scale modern data analysis. In *Big Data*.
[23] Xiaoying Wang et al. 2022. ConnectorX: accelerating data loading from databases to dataframes. In *PVLDB*.