Short Paper

PyFroid: Scaling Data Analysis on a Commodity Workstation

Venkatesh Emani
Microsoft
Venkatesh.Emani@microsoft.com

Avrilia Floratou
Microsoft
Avrilia.Floratou@microsoft.com

Carlo Curino
Microsoft
Carlo.Curino@microsoft.com

ABSTRACT

Almost every organization today is promoting data-driven decision making leveraging advances in data science. According to various surveys, data scientists spend up to 80% of their time cleaning and transforming data. Although data management systems have been carefully optimized for such tasks over several decades, they are seldom leveraged by data scientists who prefer to use libraries such as Pandas, sacrificing performance and scalability in favor of familiarity and ease of use. As a result, data scientists are not able to fully leverage the hardware capabilities of commodity workstations and either end up working on a small sample of their data locally or migrate to more heavyweight frameworks in a cluster environment. In this paper, we present PyFroid, a framework that leverages lightweight relational databases to improve the performance and scalability of Pandas, allowing data scientists to operate on much larger datasets on a commodity workstation. PyFroid has zero learning curve as it maintains all the Pandas APIs and is fully compatible with the tools that data scientists use (e.g., Python notebooks). We experimentally demonstrate that, compared to Pandas, PyFroid is able to analyze up to 20X more data on the same machine, provide comparable or better performance for small datasets as well as near-memory data sizes, and consume much less resources.

1 INTRODUCTION

Python is the de-facto language of choice for data scientists [23] and Pandas is the most favored data manipulation library [32] as it integrates seamlessly within the Python ecosystem and has no separate infrastructure requirements. A recent survey [24] estimates that more than 75% of data scientists operate on data sizes in the MB or GB range, which can fit on a single laptop/workstation. These workstations are single node, have limited RAM, and a few cores of CPU processing power with optional GPUs.

Unfortunately, current dataframe systems are inefficient when deployed on laptops/commodity workstations. For instance, according to the creator of Pandas [25], Pandas was not designed to scale for large datasets as it requires 5X-10X times the RAM to process a given dataset, resulting in failures or performance regressions as the data size increases. There have been efforts to improve the performance and scalability of Pandas workloads (e.g., [2, 4, 29]) by using a distributed execution framework to chunk and parallelize dataframes and operations. However, these systems retain the inherent scalability limitations of Pandas dataframes, and their benefits are conditional on the availability of a big cluster or a large amount of RAM, which are impractical for low-resource workstations. Another direction of work aimed at making dataframes lean is by translating imperative dataframe operations into SQL [3, 19], thereby leveraging decades-long work in optimizing relational databases for data manipulation. However, not all Pandas dataframe operations can be translated into database queries [29]. Current systems bypass this by either proposing their own dataframe APIs [18, 21], which breaks existing scripts, or by shifting the burden of deciding what operations to push into SQL onto the user [17, 20], which is tedious and error-prone. Due to these limitations, our customers are forced to either (a) upgrade to a bulkier machine (which may be costlier), or (b) use only a sample of the data (which may lead to sub-optimal results), or (c) resort to micro-management of resources programmatically (which hurts data scientist productivity).

In this paper, we present PyFroid, a system that addresses this gap by providing a lightweight, robust and efficient framework for scaling data analysis on a single node. Specifically, we develop novel techniques that use dual engine (Pandas and SQL) execution to automatically identify and pushdown operations from Pandas into SQL whenever possible or execute untranslatable operations in the stock Pandas engine. Our techniques employ lazy evaluation to execute SQL queries on demand and automatically take care of moving data between the two engines when required. Further, our techniques can optimize Pandas operations interspersed with other imperative programming statements in complex control flows making PyFroid practical in real-world applications.

PyFroid can be easily installed using Python package managers and does not rely on external database infrastructure. With PyFroid, there is no learning curve, and existing workloads can immediately benefit from its features without requiring any code modifications. We use DuckDB [33] as the database accelerator for PyFroid because it is accessible as a Python library, supports a wide range of SQL operators including those beyond relational algebra, operates on both raw files and Pandas dataframes, and delivers good query performance. Our techniques are not specific to any database engine or query dialect; other SQL databases that support cost-based planning, vectorized and multi-core execution, as well as common operations (type casting, text functions, date operations, sub-queries and WITH, etc.), can also be integrated with PyFroid and will likely provide performance and resource utilization benefits compared to purely executing the workload using Pandas.

Contributions. Our key contributions in this paper are as follows:

- A prototype Python library, PyFroid, that provides Pandas APIs backed by the DuckDB engine.
- A hybrid lazy evaluation mechanism that uses imperative-to-declarative translation for relational Pandas operations and Pandas runtime for untranslatable imperative statements.
- An evaluation of PyFroid using top-voted Kaggle notebooks that we make publicly available for others to experiment with, as well as production Pandas workloads to establish the efficiency and effectiveness of our system.

2 MOTIVATION

Consider the example program shown in Figure 1. The program, abridged from public notebook N8 (Section 6) uses Pandas APIs

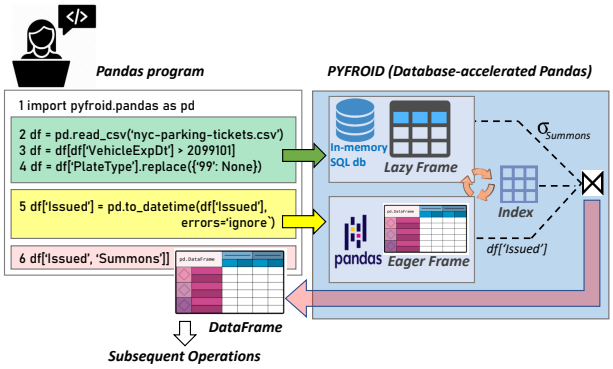

Figure 1: Architecture of PyFroid

Operator count based:			
Total	SQL-able	UDF-able	SQL+UDF
217	122 (56%)	27 (12%)	149 (69%)

Operator frequency based (5m scripts):	
Total occurrences	SQL-able
178117867	137212960 (77%)

Table 1: Feasibility of Pandas APIs to SQL Translation.

to read in a parking tickets dataset, then filters expired vehicles followed by data cleaning and transformation operations. Using our techniques, lines 2-4 of the program are automatically translated into a single SQL query that records all the operations in those lines - however, the SQL query is not yet executed. In line 5, the datetime conversion of the column *Issued* is executed in Pandas because of the *errors='ignore'* option that results in dynamic types, which is not supported by SQL. Finally, in line 6, a subset of the columns is requested for subsequent operations. At this point, PyFroid executes a projection SQL query to obtain only the required column (*Summons*), joins it with data from Pandas (*Issued*), and orders it appropriately using metadata that is maintained separately (*Index*) to provide a resultant Pandas dataframe.

This example illustrates how PyFroid's techniques enable scaling to large datasets. First, the filter (line 3) has been pushed into the database, thus a fraction of the rows is retrieved using optimized query execution plans. Note that more complex operations such as grouped aggregations, joins, etc. may be pushed down into SQL. Next, the update on line 4 is simply recorded but never executed as the program does not use the value of the column *PlateType*. Finally, only a subset of the entire dataset is materialized in memory and on-demand. We discuss challenges and underlying techniques including imperative to SQL translation, computation distribution, and how PyFroid synchronizes state across engines in Section 3.

To further motivate our work, we conducted a detailed examination of all the `pandas.DataFrame` APIs [5]. The results are shown in Table 1. Based on operator count, around 70% of the Pandas operations (with their commonly used parameters) can be expressed using SQL and SQL user-defined functions (UDFs). In this paper, we focus only on SQL-translatable operators; translating into UDFs is part of future work. We then analyzed Pandas operations in all publicly available GitHub Python notebooks from 2020 [32], amounting to a total of 5 million notebooks. Based on frequency of occurrences, a higher fraction of these operations are directly translatable to SQL (77% compared to 56% using plain operator count), reinforcing that the set of Pandas operators that are most commonly used by data scientists can be pushed down into SQL. At the same time, it is clear that SQL alone cannot handle all the available data operations motivating the need for hybrid approaches like ours.

3 APPROACH

Our system is rooted in providing fully compatible Pandas APIs to users, while allowing the system to automatically push operations down into SQL. There are multiple challenges to achieving this. Firstly, there is an impedance mismatch between SQL and Pandas. A SQL query usually consists of a number of operations including selections, projections, joins, grouped aggregations, etc. that are specified declaratively and optimized and executed together in one plan. Pandas queries, on the other hand, may span over multiple (often tens or hundreds of) lines. Secondly, SQL queries are expressed over *tables*, which are two-dimensional datasets with labeled columns and optional indexes. Queries in Pandas are expressed over *dataframes* which are two-dimensional ordered datasets with labeled and ordered columns and indexes. Operations on dataframes implicitly preserve the data ordering based on the index, whereas SQL engines are free to reorder data unless an explicit ORDER BY is specified. Pandas also provides a *Series* data structure that is a one-dimensional labeled array and shares many of the APIs on dataframes, with appropriate semantics for a single column of data. Lastly, but equally importantly, fragments of Pandas code that can be pushed down into SQL may be interspersed with other untranslatable operations. Handling this intermixing not only requires careful management of metadata and state, but also a way to distribute operations across the two execution engines. Next, we discuss the structured and techniques we propose to handle these challenges.

Structures. PyFroid internally uses the following structures: *HybridFrame*, *LazyFrame*, *EagerFrame* and *Index*. HybridFrame is what the user of the library interacts with, and it provides the same APIs as Pandas DataFrame. Under the hood, HybridFrame encompasses three structures: LazyFrame, EagerFrame and Index.

LazyFrame is a wrapper around a SQL query corresponding to Pandas operations that could be pushed down to SQL. In our work, the SQL queries are represented as *DuckDB relations* that denote virtual tables in the database. These virtual tables do not actually contain any data - they simply denote a tree of built up operations. In addition, LazyFrame supports a *materialize()* method that can force the DuckDB relation to be executed and return the result as a Pandas dataframe. Query execution is forced in conditions described in Algorithm 1, whereby some of the columns are retrieved from the database/file. The EagerFrame contains a Pandas DataFrame and is used to store these retrieved columns. The IndexFrame contains within it a Pandas dataframe with as many rows as the EagerFrame (as would be the number of rows in the LazyFrame once materialized) and three columns: *rank* – the rank of the current row based on the current ordering, *index_col* – the value of the index column (unique row identifier), and *index_col_modified* – optionally, the modified value of the index column. The last column is required because in dataframes, the index can be updated just as any other column. Each row in the EagerFrame and LazyFrame also contains the row identifier. Note that by merging the EagerFrame, materialized LazyFrame and the IndexFrame, we have all the information required to reconstruct the dataframe result.

Algorithm. Our approach is outlined in Algorithm 1. The entry

Algorithm 1

Input: *H* (base object on which method is invoked), *op* (Pandas operation), *args, kwargs* (arguments)
Output: *HybridFrame* or *Pandas DataFrame*

```
1:  procedure HybridLazyEval(H, op, args, kwargs)
            ▷ refer to description for details of subroutines used below
2:      if columns across LazyFrame and EagerFrame
3:       are accessed then
4:          mat ← Materialize(H)
5:          return op(mat, args, kwargs)
6:      else if only LazyFrame columns are accessed and
7:       op can be translated to SQL then
8:          lazyf ← ToSQL(H, op, args, kwargs)
9:          index ← UpdateIndex(H.I, lazyf)
10:         eagerf ← UpdateEager(index, H.E)
11:         return CreateHybridFrame(lazyf, eagerf, index)
12:     else                              ▷ op needs to happen in Pandas
13:         H ← MoveColumnsToEager(H, op, args, kwargs)
14:         eagerf ← op(H.E, args, kwargs)
15:         index ← UpdateIndex(H.I, eagerf)
16:         lazyf ← UpdateLazy(index, H.L)
17:         return CreateHybridFrame(lazyf, eagerf, index)
18:     end if
19: end procedure
```

point to our system is the *pyfroid.pandas* library, which is a drop-in replacement for Pandas. To leverage PyFroid the user simply has to import the *PyFroid* package instead of *pandas*. Our system then intercepts each Pandas API call, and subsequent actions are governed by Algorithm 1. Intuitively, the system checks for each Pandas operation, whether that operation can be pushed down into SQL. This depends on two factors: whether the operation has a SQL counterpart and whether past operations on the columns referenced have been pushed into SQL. If so (lines 6 to 11), the equivalent SQL operation is registered for execution on demand; otherwise, relevant columns are retrieved from the database and the operation is performed eagerly using the Pandas engine (lines 12 to 17). In this way, we achieve partial pushdown of operations into SQL. If an operation requires columns spread across the SQL and Pandas engines, then we merge the data from both the engines into a new Pandas dataframe and the operation is routed onto the dataframe (lines 2 to 5).

4 DEEP DIVE

In this section, we illustrate the working of our approach using the running example from Figure 1, and highlight key aspects of Pandas to SQL translation and computation distribution across engines. For simplicity, let's assume that dataset contains only the columns referenced in Figure 1, namely *VehicleExpDt, PlateType, Issued* and *Summons*, along with an index column *Id*.

Pushdown to SQL. Line 2 of Figure 1 contains a *read_csv* operation, which can be performed in SQL. It triggers the connection to an empty in-memory DuckDB database and creates a LazyFrame *L* with the DuckDB query *SELECT * FROM read_csv ('path/to/csv')*. The query reads relevant metadata and creates a virtual DuckDB relation, say *rel*. Along with this, we also create an IndexFrame with two columns: *Id* and an arbitrarily assigned *rank*. In line 3, the condition can again be pushed into SQL, and the query above is updated with a WHERE clause. Similarly for line 4. At the end of line 4, the query becomes:

```
Q1: SELECT CASE WHEN t0.PlateType = '99' THEN NULL
       ELSE t0.PlateType END CASE as PlateType, *
    FROM (SELECT *
          FROM (SELECT * FROM rel) AS t1
          WHERE t1.VehicleExpDt > 2099101) AS t0
```

Note that at any point during program execution, the LazyFrame *L* contains the accumulated query expression for the slice of computations encountered so far that contribute to the value being computed. Although the queries obtained, such as Q1, may be nested and verbose due to the step-by-step nature of our construction, they do not usually lead to performance degradation as database engines such as DuckDB simplify away these idiosyncrasies of query expression and generate an optimized query plan during execution.

In general, Pandas operators contain many parameters and multiple syntactic variants can be used to express the same data processing operation. For instance, the Pandas `merge` operator contains 12 parameters. In a simple variation, join keys may be specified using either a single parameter (`on`) if the column is present in both the dataframes, or separate parameters (`left_on`, `right_on`). PyFroid takes care of analyzing the parameters and translating equivalent variants into a canonicalized query expression tree. Our current prototype handles common variations of the following Pandas operations for SQL pushdown: read data (*read_csv, read_sql*), inspection (*head*, *describe*, *shape*, *info*, *columns*, *__repr__*, *__str__*), full table aggregations (*sum*, *min*, *max*, *mean*), grouped aggregations (*groupby*, *value_counts*), filter using a condition or boolean series (*df[. . .]*), projection of single or multiple columns (*df.attr*, *df[. . .]*, *drop*), handling null values (*dropna*, *fillna*, *isna*, *notna*), renaming and adding computed columns (*assign, df[. . .]*), type casting (*astype, to_datetime*), order by one or more columns (*sort*), joins (*merge*), union (*concat*) distinct (*unique*), accessing using integer index (*iloc*), arithmetic and logical operations (*AND, OR, >, <, +, -, *, etc.*), and plotting (*df.plot*), among others. The operations we support are among the most commonly used Pandas operations and enables SQL acceleration for a large number of scripts (refer Table 2). Pushing down other operations and more variations of these constructs is ongoing effort.

A major challenge when translating procedural code to declarative queries is the presence of control flow statements such as if-else, loops, and function calls. Since PyFroid intercepts Pandas API calls to construct query expressions at runtime, jumps in program flow are naturally supported; thus, optimized SQL statements can be generated. For instance, consider the following (abridged) loop fragment adapted from program P12:

```
df_join_all = []
for date in ["17", "18", "19", "20"]:
    df1 = pd.read_csv(f'op_{date}.csv')
    df2 = pd.read_csv(f'token_{date}.csv')
    join = df1.join(df2)
    df_join_all.append(df_join)
pd.concat(df_join_all)
```

Using PyFroid, we are able to translate this fragment into a single query that is a union of joins. In each iteration of the loop, the query expression for the corresponding join is appended into the list *df_join_all*, and *concat* then gets translated into a union of the list of expressions. In some cases, control flow constructs may force the query expression to be materialized, for example, when iterating with `df.iterrows()`.

Computation Distribution. We discussed previously the construction of LazyFrame *L* with the SQL query Q1 through lines 1 to 4 from Figure 1. In line 5, the data conversion operation with dynamic types (*errors='Ignore'* option) is not translatable to

SQL. So, the required column, namely *Issued* along with the index column *Id* is moved to the EagerFrame E - this entails executing a projection on top of Q1 and obtaining the dataframe. The Pandas *datetime* operation and the update to *Issued* is executed within the Pandas engine using E. Additionally, the query expression in L is updated with a projection that excludes the *Issued* column.

In line 6, columns across both the E (*Issued*) and L (*Summons*) are referenced. Based on Algorithm 1, at this point, we materialize the dataframe contents. This entails executing the query for L to obtain a dataframe, say L_{df}, and merging it with E and I to obtain the dataframe, say H_{df}. Post the merge, the operation on line 6 and subsequent ones are performed on H_{df}. Our current approach is to realize the entire dataframe contents when columns across the EagerFrame and LazyFrame are referenced. This means that the data for all the columns in L is retrieved.

In our work, explicit Pandas merge operations occurring in the script are translated to database joins for optimal performance. However, the reader may have observed that we perform the implicit join between internal structures (L_{df}, E and I) in the Pandas engine. In general, joins in Pandas may be slow. However, in our implementation we observe that once the data is filtered/grouped, for the reduced dataset size, performing the merge of L_{df}, E and I in Pandas does not regress the performance of our system.

We pushdown operations into SQL whenever possible. While this is usually beneficial, in some cases it may lead to regressions (see Section 6). Our approach of partially materializing the dataframe on demand enables high scalability even in low-resource settings. A downside of our approach is that multiple parts of the dataframe may be materialized separately, and each materialization leads to a database query. Compared to having the entire data in memory up front (as is the case in Pandas), repeated querying can be slower for simple operations, especially when the dataset is small. However, for complex queries, our approach is still beneficial. Including these factors in a cost-based decision is an interesting next step in our work.

5 RELATED WORK

There have been efforts to improve the performance and scalability of Pandas workloads including parallel processing on distributed dataframes for faster runtimes (Modin [29] and Dask [2]), memory mapping for scaling to large datasets (Vaex [10], Polars [30]), leveraging GPUs (CuDF [1] and [9]), dataframe APIs for relational databases (Ibis [3], Grizzly [19], PySpark [7], Koalas [4]), and federated execution on cloud backends (Magpie [22], Ponder [6]). Figure 2 classifies existing frameworks based on key requirements for scalable single node systems: (i) Compatibility with the popular Pandas APIs to enable broader adoption and easy migration of existing workloads, (ii) Full hardware utilization (iii) Lightweight solutions that do not rely on distributed big data frameworks such as Spark, Ray, etc.

As shown in Figure 2, only Koalas and Modin offer full Pandas compatibility among these frameworks. Koalas is based on the Spark ecosystem, which is essentially a big data system so it is quite heavy when deployed on a single node. Modin a multi-threadead drop in replacement for Pandas. It is based on distributed schedulers such as Ray or Dask, and is installed through Python package managers. Although Modin can parallelize data processing to use all available cores, under the hood it distributes Pandas dataframes, which are quite RAM-heavy. Consequently, Modin requires a lot of memory even at small data sizes. Further, for complex operations such as joins, Modin may not use

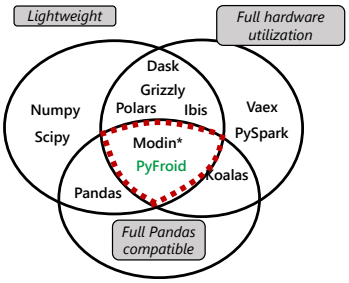

Figure 2: Taxonomy of existing frameworks

optimized execution plans and requires users to specify the join order [28]. PyFroid leverages the database cost-based optimizer to provide optimal plans even for complex queries. In our evaluation (Section 6), we show that Modin has sub-optimal performance on a single node compared to our system.

Grizzly and Polars also use lazy evaluation. However, their APIs are not fully Pandas-compatible. Consequently, users need to rewrite their scripts to leverage these systems, which is tedious and impractical. Also, the optimizations available in these systems are limited. As opposed to PyFroid, which automatically decides when to do eager vs lazy evaluation, when to push operations into the database and when to fallback to Pandas, Grizzly users need to make these decisions themselves and manually invoke appropriate constructs in their scripts to translate Grizzly objects into Pandas dataframes when needed. Similarly, unlike PyFroid, Polars does not leverage a database and thus optimizations such as predicate pushdown and join ordering are implemented from scratch and may be limited in number and scope. Further, Polars does not support dataframe indexes [31], which is a key feature of dataframe APIs.

Recent advances in large language models (LLMs) [12, 14] present interesting opportunities for code translation. However, LLMs do not provide any guarantees as to the correctness of the translation, whereas the SQL queries generated by PyFroid are always correct. Further, PyFroid is also an execution engine and handles complexities of fallback to Pandas in case of untranslatable operations.

Finally, our work is inspired by recent efforts in translation of data processing and machine learning operations into SQL such as [11, 13, 15, 16, 34], with a focus on dataframe operations.

6 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation and analysis of PyFroid. We evaluate PyFroid with respect to performance, resource utilization and data scalability. For our experiments, we use an Azure Standard DS4 v3 machine with 4 virtual CPUs and 16GB RAM, running Ubuntu 20.04.

Datasets: In our experiments, we use top-voted Kaggle notebooks (N1 to N10) as as well as production notebooks (P11 and P12) used by data scientists in Microsoft. The Kaggle notebooks were obtained by querying Kaggle for notebooks that use CSV datasets between 1GB and 5GB and tagged “pandas” sorted by the number of votes. We trimmed the notebooks to remove non-Pandas operations (i.e., invocation of other ML libraries for model training) and excluded notebooks that contained errors or use data encodings not supported in DuckDB. The notebooks we used can be found at [8]. Together, these notebooks contain a variety of data analysis and plotting operations. Table 2 provides a detailed breakdown of operations in these notebooks.

ID	Operations
N1	read, inspect, project columns, plot
N2	read, project cols, update index, filter, update, type cast
N3	read, inspect, limit, distinct, project cols, replace, aggregation (with and without groupby), filter, type cast, plot
N4	read, update index, type cast, remove duplicates, resample (time series op)
N5	read, type cast, filter, drop columns, rolling aggregates, add column, plot
N6	read, index, type cast, remove duplicates, resample, add columns, inspect, plot, reshape (matrix op)
N7	read, project single col, export col as list, filter, aggregates, reshape
N8	read, inspect, filter, drop cols, type cast, update col, distinct, export column values, in, project cols, groupby, plot
N9	read, inspect, filter, drop cols, sort, project single col
N10	read, modify dataframe attributes, distinct, filter, inspect, plot
P11	read, add columns, groupby aggregation
P12	read, join, concat (within a loop), groupby aggregates, inspect, plot

Table 2: Operations in notebooks used for evaluation

Figure 3: Micro-benchmarks (dotted red bars denote failure)

Evaluation Metrics and Goals. We evaluate PyFroid with respect to resource utilization (memory and CPU), data scalability and performance (end-to-end latency). Our experiments aim to answer the following questions:

- What are the overheads of SQL pushdown vs Pandas for common data processing operations?
- How does the performance and resource utilization of PyFroid compare to that of Pandas on a commodity workstation?
- How does PyFroid compare to other frameworks such as Modin on Ray on a commodity workstation? We note that these frameworks are designed to perform best on cluster environments but can be used to process small datasets as well [27]. We present a comparison with them on a single node to illustrate their strengths and limitations for this setup and PyFroid's contributions.

Micro-benchmarks. We first present a set of micro-benchmark that highlight the benefits of SQL pushdown for various common operations. The results are shown in Figure 3, where we used a data scale of 1.5GB from our running example dataset. The labels *pandas, sql as rs, and sql as df* respectively denote whether the operation is executed in Pandas, in SQL with results in DuckDB format or in SQL with results serialized to Pandas dataframe. We see that SQL pushdown is beneficial both in terms of latency and memory usage for most of the operations, with benefits especially large where only aggregated results need to be retrieved, as in *groupby* and *unqiue*. Data reading (*read*) in DuckDB is fast and cheap because we only need to read the file metadata. For complex operations such as *merge* (join), Pandas fails to run even at the small data size whereas SQL query runs to completion. Serializing large datasets into the Pandas dataframe format is inefficient as evidenced by the failures in *sql as df* for *load*, *merge* and *sort* operations. Next, we move on to evaluations on end-to-end workloads.

Resource Utilization. We consider notebooks N1 to P12 and vary the input data size for each notebook from very small (100MB) to near-memory (10.5GB). We use random duplication to make the dataset sizes uniform across various datasets. For each notebook and dataset size, we run the notebook with PyFroid and without (i.e., using Pandas) and measure the peak memory utilization. For some Pandas operations in these notebooks that can be translated into SQL queries but are not yet handled in our implementation (refer Section 4 for details), we simulate the translation and rewrite following Algorithm 1. Naturally, Pandas operations that cannot at all be pushed down into SQL queries are executed using the Pandas engine as described in Section 3. The results are shown in Figure 4a. The label ∞ denotes that only PyFroid could successfully run the notebook at that data size, whereas Pandas failed. The label × denotes that both PyFroid and Pandas failed.

As we see in Figure 4a, PyFroid is much less resource intensive than Pandas. Memory utilization improvements due to PyFroid range from 1.1X to 84X less than Pandas even when Pandas succeeds at that scale. In some cases, PyFroid consumes slightly higher peak memory than Pandas, as seen in N4, N5 and N6. When most operations in a notebook can be pushed down into the database, PyFroid consumes much less memory than Pandas as no intermediate data is materialized. In cases where operations cannot be pushed down especially early in a notebook, PyFroid needs to execute the query and convert almost the entire dataset into a Pandas dataframe. This can be expensive as we saw in Figure 3; further, the costs compound with materialization for multiple expressions. Specifically, in N4, N5 and N6, PyFroid falls back early to the Pandas engine due to untranslatable operations such as time series sampling, rolling aggregates, reshape, etc.

GB	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	P11	P12
0.1	1.2	2	3.1	0.9	0.9	1.3	1	2	1.4	1	2	3.2
1.5	1.6	9.9	57.2	0.9	1	0.9	1.1	7.2	12.9	×	24.4	∞
4.5	∞	∞	∞	0.7	0.9	0.6	1.2	∞	58.7	×	84.1	∞
7.5	×	∞	∞	0.7	1	0.7	1.4	∞	∞	×	∞	∞
10.5	×	∞	∞	0.8	1.1	0.8	1.6	∞	∞	×	∞	∞

(a) Peak memory utilization

GB	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	P11	P12
0.1	1	1.2	0.5	1	1.2	1.2	0.8	0.9	1.3	1	1.5	0.7
1.5	0.8	1.7	0.4	1.1	1.3	1.4	0.9	0.7	2.4	×	3	∞
4.5	∞	∞	∞	1.3	1.5	1.7	1.1	∞	1.6	×	1.4	∞
7.5	×	∞	∞	1	1.2	1.5	1.2	∞	∞	×	∞	∞
10.5	×	∞	∞	1.1	1.3	1.6	1.2	∞	∞	×	∞	∞

(b) Running Times

Figure 4: Pandas vs PyFroid (numbers denote the fraction *Pandas/PyFroid*, greener is better).

ID	Memory (GB)			Latency (s)		
	Modin	PyFroid	Benefit	Modin	PyFroid	Benefit
N1	6.1	3.6	1.7	59	71.2	0.8
N2	6.0	1.5	4.0	42.7	15.5	2.7
N3	10.3	0.1	102.5	170.8	236.8	0.7
N5	2.6	1.1	2.4	57	21.5	2.7
N7	6.0	4.9	1.2	42.9	33.3	1.3
N9	2.5	0.1	24.8	23	7.3	3.2
P11	5.0	0.05	100.6	19.1	14.5	1.3

Table 3: Modin vs PyFroid (Benefit denotes the fraction *Modin/PyFroid*, higher is better).

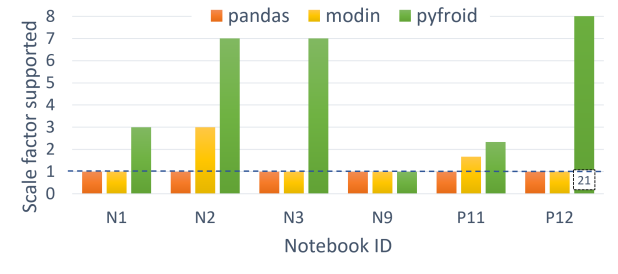

Figure 5: Data Scalability of various frameworks

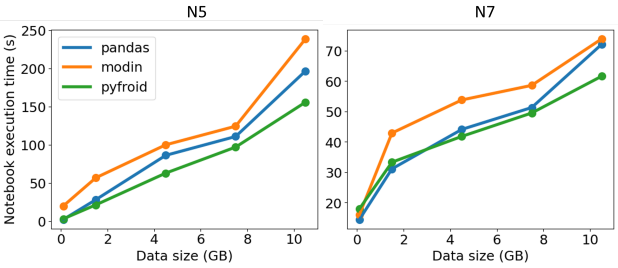

Figure 6: End-to-end latency with various frameworks

Table 3 compares Modin and PyFroid for the evaluated notebooks. Overall, we observed that typically, Modin is more memory intensive than Pandas and PyFroid. Beyond 1.5GB data size, we observed that Modin increasingly only succeeds on very few notebooks, so we choose to report our results for this data size. There are a few failures for Modin even at this scale and we excluded those notebooks from Table 3. We note that we installed and used stock Modin on Ray following the guide at [26]; Modin experts may be able to extract more performance using configuration tuning. Nevertheless, as we see from Table 3, PyFroid provides huge benefits in memory utilization compared to Modin (up to 100x), while also providing comparable to significant benefits in end-to-end latency. The memory utilization benefit obtained depends on the nature of the workload. As noted before, Modin distributes Pandas dataframes, which are memory heavy, across each worker in the underlying execution framework such as Ray. As a result, the memory utilization of Modin shoots up even at small data sizes. For the cases where Modin failed, the memory utilization shot up to 100% before the system crashed.

CPU Utilization: CPU utilization in PyFroid and Modin is usually higher (upto 4X) than in Pandas due to the use of all the cores. However, in our experiments, we observed that for a specific workload, CPU utilization does not rise in correlation with data sizes.

Scalability. In Figure 5, we compare the scalability of PyFroid, Pandas and Modin. We plot for each notebook, the maximum dataset size for which the framework successfully ran the notebook, with Pandas as the baseline (y=1). For instance, in N1, both Pandas and Modin run successfully until data size 1.5GB whereas PyFroid runs successfully until 4.5GB, so the value for Modin would be 1 and PyFroid would be 3. We exclude in the chart notebooks for which all frameworks had the same maximum dataset size (N5 - 10.5, N7 - 10.5 and N10 - 0.5) or if the notebook had APIs yet unsupported by Modin (N4, N6 and N8). As we see in Figure 5, PyFroid can scale up to 20*X* bigger datasets than Pandas and Modin. This is because PyFroid avoids a full load of the data into memory and leverages SQL database for complex operations such as joins (e.g., P12). By analyzing the individual notebook operations, we found that plotting operations that retrieve a lot of data (e..g, correlation matrices) negatively affect the scalability of all three frameworks.

End-to-end Latency. We first compare the end-to-end latency of Pandas vs PyFroid. As we see in Figure 4b, PyFroid consistently outperforms Pandas in most cases, with benefits up to 3x even in notebooks that Pandas succeeds. Note that the benefits in latency are often accompanied by benefits in resource utilization as seen in the two tables in Figure 4, which reinforces the holistic benefits due to our techniques.

In some cases, using PyFroid may lead to performance degradation. For instance, notebook N3 at 1.5GB data scale has 2.5x higher latency using PyFroid compared to Pandas. This is because N3 contains repeated inspection operations (head, info, unique, count, etc.). While Pandas has data materialized up front to resolve these operations quickly, in PyFroid, each such operation forces a separate query execution leading to higher latency. On the plus side, our approach significantly reduces peak memory utilization (over 50x less than Pandas). Overall, the number of regressions due to PyFroid is small and as data size increases, PyFroid continues to complete execution whereas Pandas fails in many cases.

In the next chart (Figure 6), we compare the end-to-end latency using PyFroid, Modin and Pandas on notebooks N5 and N7 at various dataset sizes. We picked those two notebooks as these were the only ones where all three frameworks completed at all dataset sizes. As far as it concerns the remaining notebooks, we note that only PyFroid completed at larger sizes and the other two frameworks failed much earlier. As we see, PyFroid executes similarly or in significantly less time than Pandas and Modin. As the data sizes increase, the benefits due to PyFroid also increase.

7 CONCLUSION

In this paper, we identified a gap in existing systems for efficiently scaling Pandas workloads on a commodity workstation and presented PyFroid, a system that addresses this gap by translating Pandas operations into SQL queries that can run on an embedded database. For operations that cannot be translated to SQL, PyFroid runs them efficiently using the Pandas engine through on-demand materialization, and transparently puts together results from the database and Pandas. Our experiments demonstrate that our system is resource-efficient and can provide significant speed and scale on a variety of workloads.

REFERENCES

[1] [n.d.]. cuDF - GPU DataFrames. https://github.com/rapidsai/cudf.
[2] [n.d.]. Dask. https://dask.org/.
[3] [n.d.]. Ibis: Python data analysis framework for Hadoop and SQL engines. https://github.com/ibis-project/ibis.
[4] [n.d.]. Koalas. https://github.com/databricks/koalas.
[5] [n.d.]. pandas.DataFrame. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.
[6] [n.d.]. Ponder | Pandas at Scale. https://ponder.io/.
[7] [n.d.]. PySpark Documentation. https://spark.apache.org/docs/latest/api/python/.
[8] [n.d.]. Supplementary material. https://github.com/kvemani/pyfroid_material.
[9] [n.d.]. Taming Model Serving Complexity, Performance and Cost: A Compilation to Tensor Computations Approach. https://scnakandala.github.io/papers/TR_2020_Hummingbird.pdf.
[10] [n.d.]. Vaex.io: An ML Ready Fast Dataframe for Python. https://vaex.io/.
[11] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p17-blacher.pdf
[12] chatgpt 2023. *ChatGPT: Optimizing Language Models for Dialogue.* Retrieved Feb 15, 2023 from https://openai.com/blog/chatgpt/
[13] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. *SIGPLAN Not.* 48, 6 (jun 2013), 3–14. https://doi.org/10.1145/2499370.2462180
[14] codex 2023. *OpenAI Codex.* Retrieved Feb 15, 2023 from https://openai.com/blog/openai-codex/
[15] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL. In *ACM SIGMOD*. ACM, 1663–1666. https://doi.org/10.1145/3035918.3058747
[16] Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *ACM SIGMOD (SIGMOD '22)*. 2389–2392. https://doi.org/10.1145/3514221.3520175
[17] grizzly-example 2023. *grizzly/example.* Retrieved May 17, 2023 from https://github.com/dbis-ilm/grizzly/blob/master/example.py
[18] grizzly-limitations 2023. *GitHub - dbis-ilm/grizzly: A Python-to-SQL transpiler as replacement for Python Pandas.* Retrieved May 17, 2023 from https://github.com/dbis-ilm/grizzly#limitations
[19] Stefan Hagedorn. 2020. When sweet and cute isn't enough anymore: Solving scalability issues in Python Pandas with Grizzly.. In *CIDR*.
[20] ibis-execute 2023. *Base Expression Types.* Retrieved May 17, 2023 from https://ibis-project.org/api/expressions/?h=execute#ibis.expr.types.core.Expr.execute
[21] ibis-for-pandas 2023. *Ibis for pandas Users.* Retrieved May 17, 2023 from https://ibis-project.org/ibis-for-pandas-users/#ibis-for-pandas-users
[22] 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*.
[23] kaggle-survey 2022. *State of Data Science and Machine Learning 2021.* Retrieved Jun 29, 2022 from https://www.kaggle.com/kaggle-survey-2021
[24] kdnuggets-poll 2022. *Largest Dataset Analyzed - Poll Results and Trends - KDNuggets.* Retrieved June 29, 2022 from https://www.kdnuggets.com/2020/07/poll-largest-dataset-analyzed-results.html
[25] Wes McKinney. 2017. Apache Arrow and the "10 Things I Hate About Pandas". https://wesmckinney.com/blog/apache-arrow-pandas-internals/.
[26] modin-github 2023. *Modin: Scale your pandas workflows by changing one line of code.* Retrieved July 17, 2023 from https://github.com/modin-project/modin
[27] modin-mb-to-tb 2023. *Scale your Pandas workflow by changing a single line of code.* Retrieved Feb 15, 2023 from https://modin.readthedocs.io/en/latest/#modin-is-a-dataframe-for-datasets-from-1mb-to-1tb
[28] modin-merge 2023. *Modin: Operation-specific optimizations.* Retrieved May 2, 2023 from https://modin.readthedocs.io/en/latest/usage_guide/optimization_notes/index.html
[29] Devin Petersohn, William Ma, Doris Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *arXiv preprint arXiv:2001.00888* (2020).
[30] polars 2023. *Polars: Blazingly fast DataFrames in Rust, Python, Node.js, R and SQL.* Retrieved May 2, 2023 from https://github.com/pola-rs/polars
[31] polars-no-index 2023. *Coming from Pandas: Polars User Guide.* Retrieved May 17, 2023 from https://pola-rs.github.io/polars-book/user-guide/migration/pandas/#polars-does-not-have-a-multi-indexindex
[32] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, et al. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML. NET Pipelines. *ACM SIGMOD Record* 51, 2 (2022), 30–37.
[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
[34] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, et al. 2017. Froid: Optimization of imperative programs in a relational database. *VLDB* 11, 4 (2017), 432–444.