Lazy Like a Duck, Fast Like a DB: PyDuck for Scalable Data Preprocessing

Andre Nandi
University of Michigan
andrenan@umich.edu

Sai Vaka
University of Michigan
saivaka@umich.edu

Evan Zimmerman
University of Michigan
evanzimm@umich.edu

Abstract

Data preprocessing is a critical step in machine learning (ML) workflows. Pandas is the most popular in-process framework for ML data preparation, but its performance is limited when handling large datasets due to its single-threaded execution and memory constraints [7, 3].

DuckDB is an in-process OLAP database designed for analytical workloads [4]. It may be better suited for data preprocessing because it handles larger than memory datasets and offers multi-threaded execution. Although DuckDB can achieve the same data preprocessing results as Pandas through SQL queries, some Pandas functions do not have simple SQL translations, and DuckDB lacks a dataframe API that implements the standard functions of ML data preprocessing frameworks.

To explore DuckDB's ability to accelerate data preparation tasks for ML workflows, we built PyDuck. PyDuck is a DuckDB wrapper with a Pandas-style API that allows users to operate on a dataframe abstraction, called a Quack, that is connected to a table in DuckDB. Our results show that PyDuck is able to achieve much greater performance on some data preparation tasks compared to Pandas, while achieving competitive performance on the same tasks compared to DuckDB. This work shows that a DuckDB wrapper with a well-implemented dataframe abstraction could reduce reliance on traditional dataframe libraries, and simplify end-to-end ML pipelines while improving efficiency.

1. Introduction

With an exponential rise in machine learning workflows, we see a greater need for efficient data preprocessing in order to train and develop models faster. However, data preparation is a massive bottleneck for ML workflows. According to McKinsey, "Many organizations have found that 60 to 80 percent of a data scientist's time is spent preparing the data for modeling... In essence, tuning model parameters has become a commodity, and performance is driven by data selection and preparation." [2]

We currently see a gap in tools commonly used for data cleaning. Pandas is the most used library for data preparation, but since it operates entirely in-memory (it attempts to load the entire dataset into memory), it greatly struggles with large datasets because pandas isn't optimized to perform efficient disk I/Os. Pandas similarly doesn't have optimized indexing and joining/merging capabilities, especially for datasets larger than RAM. Also, Pandas primarily utilizes a single thread, not fully leveraging multi-core processors [7, 3].

Faster data preprocessing can significantly reduce time-to-insight(T2I) for data science and analytic teams, empowering them to focus more on model experimentation, feature engineering, and decision-making rather than tedious data wrangling, ultimately accelerating innovation and business impact.

DuckDB offers optimizations that show promise in outperforming Pandas. DuckDB is optimized for out-of-core processing, meaning it can efficiently handle datasets that don't fit into memory. Because DuckDB doesn't need to load everything into memory, it is highly scalable compared to Pandas. DuckDB is also designed to leverage multi-threading, has optimized indexing and join operations, and has an internal query engine that optimizes both columnar and row-based queries [4].

Creating a Pandas-like wrapper library for DuckDB will make DuckDB more accessible to Software Developers and ML Engineers. This wrapper could bridge the gap between data science and high performance analytics by providing an intuitive API to access DuckDB's efficient query execution.

To fill the gap between ML data preprocessing frameworks and DuckDB, we built PyDuck. PyDuck offers the dataframe abstraction directly on top of data in DuckDB, and implements many of the preprocessing functions that are standard to libraries like Pandas in a pandas-like syntax. The core components of PyDuck are as follows:

1. **Quack**: Quacks are PyDuck's version of dataframes — they allow user's to interact with data in DuckDB through a dataframe-API similar to Pandas. Quacks are immutable objects that stored a chained series of operations and follow a lazy execution model. Quacks

are described in full detail in section 4.1.

2. **SQL Compiler**: DuckDB is often overlooked for ML data pre-processing because efficient SQL implementations of certain Pandas-esque functions (i.e. fillna() which replaces null values with a specified value) is non-trivial. PyDuck addresses this challenge with a standalone SQL compiler module. This module takes as input the chain of operations for any arbitrary Quack, and returns the equivalent SQL query. The SQL Compiler is described in section 4.2.

The rest of this paper is outlined as follows: In section 2 we discuss related work. In section 3, we discuss our initial benchmarking work. In section 4 we cover PyDuck's system design and individual components in-depth. In section 5 we present the results of evaluating PyDuck against DuckDB and Pandas for data preprocessing tasks. Lastly, in section 6, we conclude and discuss future work.

2. Related Work

Popular frameworks for in-process ML data preparation include Pandas, Polars, PySpark, and more [3]. Although the least performant overall, Pandas is the framework that popularized the dataframe API, and its interface laid the groundwork for what is considered the standard dataframe interface today [7]. It is important to note that despite its performance limitations, Pandas greatest strength over other frameworks is its rich ecosystem and its ability to seamlessly integrate with other ML frameworks.

Polars is a rust-based dataframe library built for high performance. It offers multi-threaded execution and uses Arrow as the underlying data format which allows for efficient resource utilization [6]. PySpark is the Python API for Apache Spark, which is known for processing data over a cluster of machines, however it has a single machine mode as well. PySpark also offers multi-threading, effectively spills large datasets to disk if necessary, and processes data in columnar format with lazy execution [9, 3].

DuckDB functions as a traditional relational DBMS; however, since DuckDB is in-process it has an "SQL on Pandas" feature that allows users to execute SQL queries a Pandas dataframe using DuckDB's execution engine [5]. This feature never takes the data out of its Pandas binary format, following a "Pandas-in, Pandas-out" execution model. While this approach can offer performance gains for data preprocessing, it still requires users to be proficient SQL writers to generate queries that can mimic some built-in pandas functions (i.e. get_dummies()). DuckDB does not offer its own dataframe interface with a dataframe-style API.

3. Initial Benchmarking

Because DuckDB does not offer its own dataframe abstraction, it is often left out of comparative studies that evaluate dataframe frameworks (i.e. [3]). However due to it's in-memory nature it can be used in a similar fashion (locally loading and transforming a csv, parquet file, etc.) to achieve the same preprocessing tranformations.

To understand how DuckDB compares to other popular dataframe frameworks for data pre-processing, we measured how well DuckDB, Pandas, PySpark, and Polars performed on the same preprocessing tasks on the same dataset at various sizes. The hardware used for this benchmarking was an 11th Gen Intel i7-1165G7 CPU with 16GB RAM running Windows 64-bit, and the dataset was the NYC Yellow Taxi Trip Data from Kaggle at 1MB, 10MB, 100MB, 1GB, 2GB [1]. All functions were run 10 times and the average runtime was recorded. The full table of results are shown in the appendix (section 7.1).

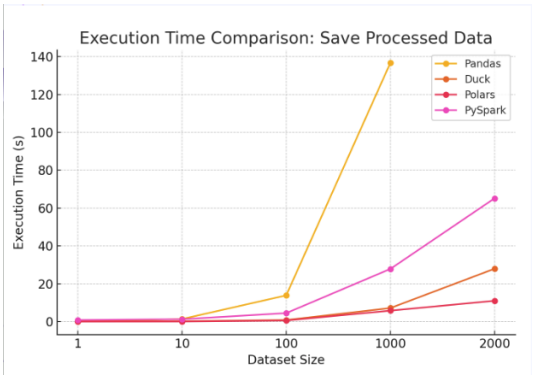

Figure 1. Execution Time of Saving Processed Data to CSV

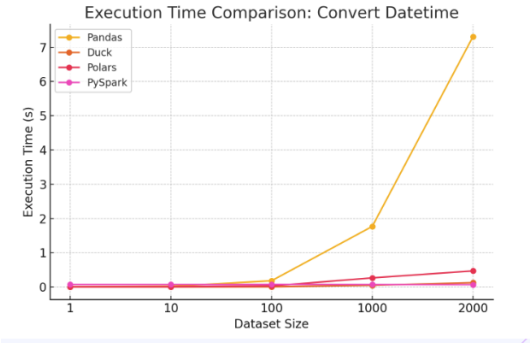

Figure 2. Execution Time of Convert String to Datetime

Figure 1 and Figure 2 show the execution time of saving processed data to csv format and converting a column to datetime format, respecitvely. From the figures, it is clear than Pandas does not scale as well as the other frameworks. DuckDB performs much better than Pandas and is competitive with the other frameworks for these tasks. It is also

worth noting that Pandas failed to save data to csv for a 2GB dataset, highlighting its inefficient memory usage.

4. PyDuck Design

This section will describe how we designed PyDuck. We will begin by discussing our goals, from which we motivate our design. Then, we will explore the system design behind PyDuck and explore each individual component. We finish by comparing PyDuck's usability to Pandas.

4.1. Goals and Design Choices

The design of PyDuck was motivated by the need for a data preprocessing library that offers both the usability of Pandas and the scalability of modern analytical engines. While Pandas is widely adopted in data science workflows, it suffers from large performance bottlenecks because it is limited by RAM. It's single-threaded and in-memory execution model prevents pandas from being usable or efficient for large datasets. PyDuck addresses these limited by providnig a familiar Pandas-like API that allows Pandas users to easily transition to DuckDB. Using DuckDB introduces a high-performance OLAP engine that can bring major improvements to data processing workflows and reach new levels of efficiency. The following subsections describe the key principles that guided the design of PyDuck, including its core abstraction, execution model, and emphasis on modularity and transparency.

4.1.1 Quack-Based

PyDuck is centered around the concept of a `Quack`, a lightweight, chainable abstraction that represents a virtual view of a table stored in DuckDB. A `Quack` is essentially PyDuck's version of a Pandas dataframe. Each `Quack` instance encapsulates a conneciton to a duckDB database, a base table name, and a sequence of transformations, referred to as the *operations list*. These operations are not executed immediately, but rather stored for deferred compilation into an optimized SQL query. To ensure compatibility with existing pandas workflows, PyDuck easily supports conversions between `Quacks` and Panda dataframes.

4.1.2 Lazy Execution

Unlike Pandas' eager evaluation model, PyDuck employs a lazy execution strategy. Operations like `filter()`, `groupby()`, and `dropna()` do not trigger immediate computation; instead, they return new `Quack` objects with updated operation chains. This deferred execution allows PyDuck to optimize the full query plan before execution, minimizing intermediate materialization and enabling more efficient SQL compilation. This allows us to take full advantage of DuckDB's query optimizer that is constantly under improvement from a large yet active Open-Source community. Execution is only triggered explicitly when users request access to the dataframe via `to_df()` or an internal/external `execute()` call is made on the Quack object, at which point the entire operation chain is compiled into a single SQL query.

4.1.3 RDD-like Lineage

Inspired by Apache Spark's RDD model, PyDuck models its transformation pipeline as an immutable sequence of operations [8]. Each transformation returns a new `Quack` object without mutating the original. This design allows for traceability, reproducibility, and potential optimizations throughout the transformation lineage. This is most clear when you think of workflows filled with filters and projections. Chains of filters or projections can be combined and simplified during SQL compilation, reducing query complexity and execution time. Below is an example of a Quack workflow and the implicitly generated lineage of Quack objects.

```
from pyduck.quack import Quack
import duckdb

# Step 1: Load customer (1st Quack)
q1 = Quack("TABLE", duckdb.connect("db"))

# Step 2: Filter on row (2nd Quack)
q2 = q1.filter("row1>1000")

# Step 3: Select 2 rows (3rd Quack)
q3 = q2[["row2", "row3"]]

# Step 4: Group by and agg (4th Quack)
q4 = q3.groupby("row4")
       .agg({"row5": "mean"})

# Step 5: Convert to dataframe
result = q4.to_df()
. . .
```

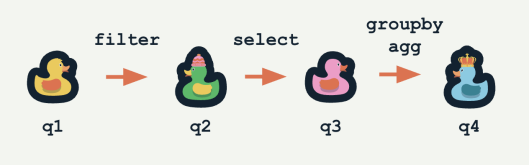

Figure 3. Example of the implicit PyDuck Lineage

4.1.4 Transparency

A key design goal of PyDuck is to provide full visibility into the underlying computations. The `to_sql()` method returns the final SQL query that will be executed on DuckDB.

This allows developers to inspect, debug, and understand how their Pandas-like operations are mapped to SQL, fostering trust and making the library suitable for educational and production use alike.

4.1.5 Composability

PyDuck operations are highly composable. Each transformation is modular and implemented independently in the `operations/` directory. This modularity facilitates reuse, testing, and extensibility. Developers can easily introduce new operations by defining a transformation function and integrating it into the SQL compiler pipeline, without needing to modify the core logic of `Quack` or `SQLCompiler`. Each Quack object is also standalone, meaning that each can monitor it's own operations and query the DuckDB database independently without needing to manual track dependencies or metadata.

4.2. System Design

The following diagram explains the high-level system design of PyDuck. First, a user will initialize a Quack ob-

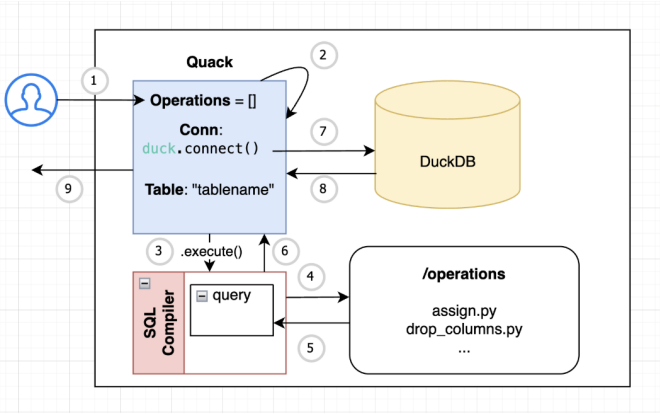

Figure 4. System Design of PyDuck

ject. The user will call operations directly on the quack object (1). Each operation generates a new quack and appends to its operation list. Each quack keeps track of all operations that have been called on it (2). An `.execute()` call is triggered by `to_df()` or a manual call, which initializes the SQL Compiler (3). The SQL compiler maps the operations in the operations list to Python files in the `operations/` directory that will generate SQL commands (4). The compiler collects and formulates the updated compound SQL query (5). The complete SQL query is returned to the Quack object (6), which is then sent to the DuckDB instance for execution (7). The results are returned to the Quack object as a dataframe (8), which is then given back to the user (9).

4.2.1 Quack

The `Quack` class, defined in `quack.py`, is the core interface for PyDuck users. It stores the connection to the DuckDB database instance, the base table name, and an ordered list of operations. Each method in `Quack` (e.g., `filter()`, `assign()`, `groupby()`, `dropna()`, `fillna()`) returns a new `Quack` object with an extended operations list. This design ensures immutability and clean separation of transformation steps. The `__getitem__` method enables Pandas-like column selection, while `merge()` supports multi-table joins.

4.2.2 SQL Compiler

The `SQLCompiler` class in `compiler.py` is responsible for converting the operations list into a valid SQL query. The compiler begins with a base query (`SELECT * FROM <table_name>`) and iteratively applies each operation by calling the `apply_operation()` function, which maps high-level operations to valid SQL snippets. By sequentially modifying the query string, the compiler produces a complete SQL statement that reflects the full transformation pipeline, which can then be executed efficiently in DuckDB. The compiler leverages DuckDB's support for subqueries, CTEs, and vectorized execution to efficiently assemble and execute the query. Lazy evaluation enables the compiler to analyze the entire operations list and optimize the final SQL output before execution.

4.2.3 Compilation Example

Here is an example of such a query forming based on the following. As a reminder, this query is formed lazily, but we are showing the intermediate steps for clarity.

First, a user will initialize a Quack object. The base query will be a basic select all query from the DuckDB table associated with the quack object.

```
q1 = Quack("TABLE", duckdb.connect("db"))
```

```
SELECT * FROM TABLE
```

During a filter call, the compiler will wrap the query with a new select and introduce a `WHERE` clause.

```
q1.filter("row1>1000")
```

```
SELECT * FROM (
  SELECT * FROM TABLE
) WHERE row1 > 1000
```

During a select call, the compiler will modify the `SELECT *` portion of the query to only `SELECT` the desired columns.

```
q3 = q2[["row2", "row3"]]
```

```
SELECT row2, row3 FROM (
  SELECT * FROM TABLE
) WHERE row1 > 1000
```

For groupby/aggregate calls, the compiler will wrap the existing query with the necessary subquery.

```
q4 = q3.groupby("row4").agg({"row5": "mean"
    })
```

```
SELECT "row4", AVG("row5") AS "row5_mean"
FROM (
  SELECT row2, row3 FROM (
    SELECT * FROM TABLE
  ) WHERE row1 > 1000
) AS sub
GROUP BY "row4"
```

Each operation wraps or modifies the previous one. While the resulting SQL statement is quite naive, the nested query can be optimized and executed as a single SQL statement in DuckDB.

4.2.4 `Operations/`

Each operation supported by PyDuck is implemented as a standalone module in the `operations/` directory. These include `filter.py`, `dropna.py`, `fillna.py`, `groupby.py`, `drop_duplicates.py`, `get_dummies.py`, etc. Each module defines an `apply()` function, which takes the current SQL string, the operation arguments, and optionally the DuckDB connection and table name. The modular structure decouples transformation logic from execution logic and simplifies testing and extension.

4.3. Comparison with Pandas

Table 1. Comparison between Pandas and PyDuck

Feature	Pandas	PyDuck
Exec Model	Eager	Lazy
Back-End	In-memory (Python)	DuckDB (vectorized)
Scalability	RAM-limited, single-threaded	Disk-backed, multi-threaded
Performance	Fast on small data	Scales better with large data

PyDuck closely mimics the Pandas API while offering significant scalability and performance improvements. Pandas operates entirely in-memory and executes eagerly, which leads to memory pressure and slower performance on large datasets. Pandas is inherently single-threaded and limited by Python's Global Interpreter Lock (GIL), which prevents it from fully using multi-core CPUs during computation.

In contrast, PyDuck:

- Executes lazily, enabling whole-query optimization.
- Uses DuckDB as the backend, providing multi-threaded and out-of-core execution.
- Scales efficiently beyond RAM capacity, thanks to DuckDB's ability to spill to disk.

5. Evaluation

5.1. Experimental Setup

Our benchmarks were conducted on a MacBook Pro M2 Max with 128GB of RAM running macOS. Each benchmark was executed using Python 3 in a virtual environment with the following software stack:

- **DuckDB:** v1.2.1
- **Pandas:** v2.2.3
- **PyDuck:** Our custom Pandas wrapper built on DuckDB

We benchmarked three frameworks (Pandas, DuckDB, PyDuck) across multiple TPC-H scale factors: 0.005, 0.01, 0.1, 1, 2, 4, 8, and 10, where scale = 1 is approximately 1 GB. This is why we used a high end hardware system of 128GB RAM since we wanted to observe how Pandas scales in large, industrial sized datasets. We originally tried using a 16GB RAM hardware on an intel processor and found that it couldn't handle past a scale of 1.5, which is why we made the switch to a much larger system to handle datasets of 10+ GBs. For each scale, we measured execution time (in seconds (s)) over 10 trials for each bencmarking function and reported the average.

The benchmarks included:

- **TPC-H queries:** 4 TPC-H queries: Query 1, Query 4, Query 6, Query 11. These composite queries consisted of multiple merges (JOIN) , filters (WHERE), group by and aggregation, and also sorting values. These queries can visualize how each library peforms in complex workflows
- **Single data preprocessing functions:** sample, drop duplicates, drop columns, fillna, dropna, null sums, and get dummies (one hot encode). We chose these functions as these are very popular preprocessing functions. These individual benchmarking functions can visualize the clear benefits and potential drawbacks of Pandas and Pyduck

All benchmarks were implemented using our custom `frameworkTester` class, which loads the TPC-H dataset from DuckDB and runs equivalent operations in each of the 3 library frameworks: `pyduckTester`, `pandaTester`, `duckdbTester`

5.2. Results

Our experimental results demonstrate that PyDuck, outperforms Pandas and closely tracks DuckDB's performance across several common data processing tasks and composite query workflows. On the other hand, there are some benchmarks that revealed the limitations of Pyduck.

In total, it took about 3 hours to run all the benchmarks with all the raw resutls displayed in the appendix: Table 6 (Pandas), Table 7 (Pyduck), and Table 8 (DuckDB). Below we will visualize and analyze both the benefits and drawbacks of Pyduck according to the benchmarking data.

Note: for each figure, we transformed our x axis from scale points to actual rows processed depending on which tables each benchmarking functions where actually performing their queries on. This way, it is easier to grasp how the execution time of function scales with the number of rows processed. Additionally, we tried to record the peak memory consumption of each benchmarking function under different scales for each library. However, the `malloc` python library used to measure peak memory could not correctly capture the memory consumption for DuckDB and PyDuck queries. We theorize that `malloc` cannot trace memory of the sql query itself, returning extremely low memory consumption for both libraries, but was giving accurate memory information for Pandas. Due to the inconsistencies, we decided to only stick to the average execution time metric for conclusive analysis.

5.2.1 PyDuck Advantages

Out of all the benchmarking functions, we found that PyDuck excelled in group by and aggregation, null sum ,and all the TPC-H queries. As seen in figure 5 and figure 6, PyDuck and DuckDB perform better than Pandas by magnitudes. Figure 5 shows a normal scale while figure 6 displays a logarithmic scale, where pandas scale exponentially with the number of rows it has to process. Additionally, we observe that PyDuck is on par with DuckDB, which shows how implementation was efficient and adds little to no additional time overhead to DuckDB.

Additionally, group-by aggregation (figure 7) and null-counting individual functions support that PyDuck excels in functionality that is efficient in columnar SQL execution. As shown in Figure 7, PyDuck matches DuckDB in performance and both consistently outperform Pandas across all scales. At scale factors greater than 1 (over 6 million rows), Pandas runtime increases exponentially, reaching over 4 seconds by scale 10. In contrast, both PyDuck and DuckDB take under 0.036 seconds at scale 10, which is a speedup of more than 100x over Pandas. Similarly, Figure 8 shows PyDuck's advantage in computing null value counts using `isna().sum()` by out performing Pandas by magnitudes.

This speedup highlights how DuckDB's vectorized execution model and PyDuck's lazy SQL compilation help maintain efficient performance at high scales. The lazy SQL compilation allows a the query engine to optimize the final query and combined with columnar execution, it is no surprise it performs better than Pandas by magnitudes.

Overall, These benchmarks clearly show how PyDuck takes advantage of SQL operations internally to reduce both runtime compared to Pandas, specifically as data scales toward real world dataset sizes.

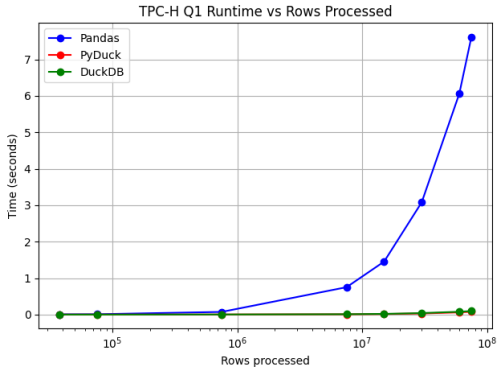

Figure 5. TPC-H Query 1: Composite Query Runtime vs. Rows Processed

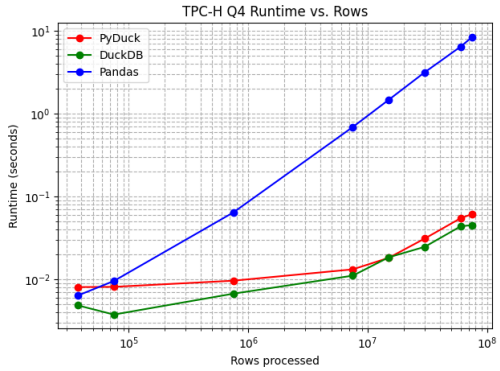

Figure 6. TPC-H Query 4: Composite Query Runtime vs. Rows Processed

5.2.2 Limitations

While it seems that PyDuck has a clear advantage over Pandas without little to no overhead on DuckDB, there were some limitations of DuckDB and our implementation. Specifically for `test_get_dummies()` and

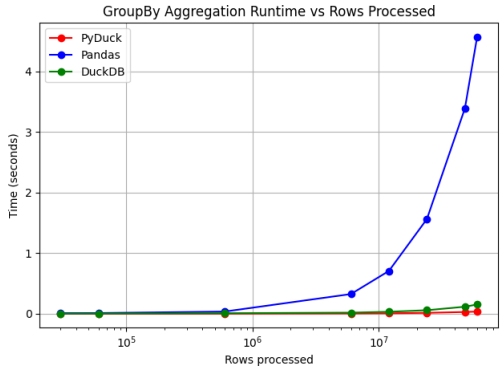

Figure 7. Groupby Aggregation (sum): Runtime vs. Rows Processed

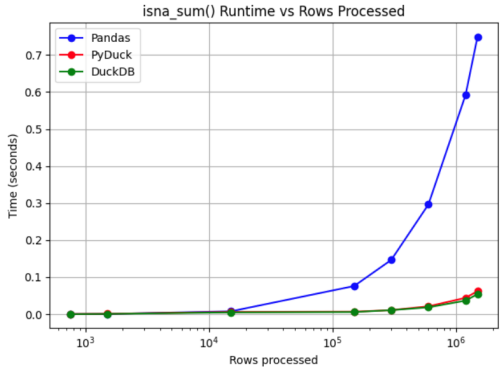

Figure 8. isna-Sum (counting na/null values): Runtime vs. Rows Processed

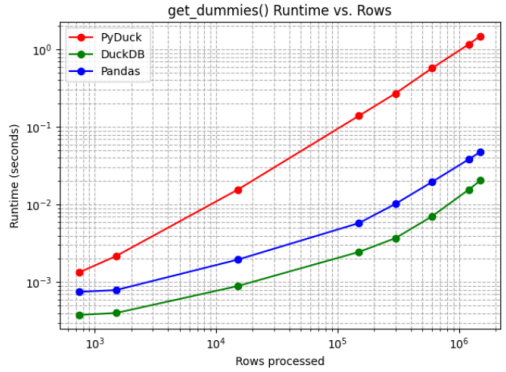

Figure 9. get_dummies() Runtime vs. Rows

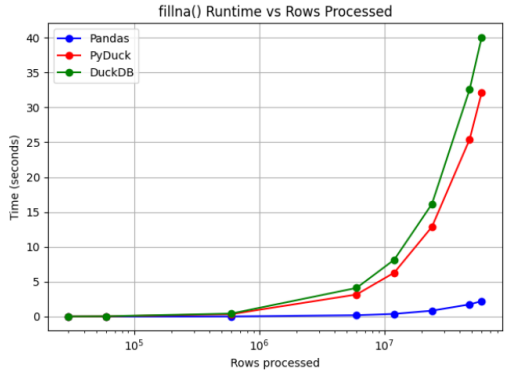

Figure 10. fillna() with mean: Runtime vs. Rows. SQL-based subqueries lead to high overhead for PyDuck and DuckDB.

`test_fillna()`, there were some clear drawbacks for PyDuck.

Figure 9 (`test_get_dummies()`) shows that PyDuck significantly underperforms against Pandas and DuckDB in hot encoding tasks. Even though DuckDB performs the best, PuDuck is still magnitudes worse compared to both Pandas and DuckDB, displaying our lack of efficiency in our implementation, which may have added a lot more overhead and increasing execution runtime. So, for get_dummies and other functions where we find that PyDuck adds overhead, we will need to work on developing a more efficient implementation and retesting to increase speedup to meet with DuckDB demands.

Also shown in Figure 10, PyDuck and DuckDB perform exponentially worse than Pandas in filling na/Null values. This function alone showed that there are some functionality in Pandas that might already be truly optimal for the task it is running. We additionally theorize that column-wise SQL execution lacks in row level updates, while Pandas uses native Numpy functionality and more optimized caching, which may explain Pyduck and DuckDBs poor performance against Pandas.

These limitations overall show that DuckDB and PyDuck alone cannot fully replace and speed up all Pandas functionality. There are some functions that may need better optimization in PyDuck, and there are some that seem to do already really well in Pandas. Furthermore, there were some benchmarks where we found little to no improvement of PyDuck or DuckDB over Pandas, specifically for (`test_drop_columns()`), (`drop_duplicates()`), and (`test_sample()`), allowing one to choose between either Pandas or Pyduck implementation without loss of performance. So going forward, it seems that a hybrid systems combining both PyDuck functionality with native Pandas library can help to achieve higher overall performance benefits for data preprocessing.

6. Conclusion and Future Work

This paper presented PyDuck, a DuckDB wrapper that presents a dataframe abstraction on top of DuckDB. PyDuck allows Pandas-style data preprocessing directly on data in DuckDB, taking advantage of DuckDB's optimized query planning, vectorized execution, and memory management. The framework is built on immutable Quack objects that serve as PyDuck's version of a dataframe. Quacks al-

low users to chain operations while following a lazy execution model. Only upon explicit execution are a Quack's operations compiled to SQL and run on a table in DuckDB.

Our experimentation showed that PyDuck performs significantly better than Pandas on some data pre-processing tasks but struggles with some non-traditional DBMS operations. For operations that only involve traditional DBMS operators (i.e. filter, groupby, aggregate) that benefit from vectorized processing, PyDuck performs strongly. However, pre-processing frameworks like Pandas also offer unique stand-alone functions like fillna(), and PyDuck performed poorly on those functions.

We believe that PyDuck has many potential improvements, not only in the execution of unique Pandas functions, but at the system-level as well. PyDuck's subpar performance on certain Pandas functions is likely a direct result of generating inefficent SQL queries in PyDuck's compilation phase. To address this issue, we are investigating the bottlenecks in the current inefficeint operations and modifying their SQL translations to improve execution performance. At the system-level, we would like to explore the potential of an eager execution model. With the current lazy execution model, chaining together many operations can lead to deeply nested SQL queries, which can be very challenging for the query optimizer to optimize. With an eager execution model, we remove this challenge for the query optimizer at the cost of needing to store intermediate results. Despite this trade off, shifting to an eager execution model could improve overall runtime for preprocessing tasks.

References

[1] Elemento. Nyc yellow taxi trip data, Dec 2021.

[2] Holger Hurtgen, Sebastian Kerkhoff, Jan Lubatschowski, and Manuel Moller. Rethinking ai talent strategy as automated machine learning comes of age, Aug 2020.

[3] Angelo Mozzillo, Luca Zecchini, Luca Gagliardelli, Adeel Aslam, Sonia Bergamaschi, and Giovanni Simonini. Evaluation of dataframe libraries for data preparation on a single machine, 2025.

[4] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Mark Raasveldt and Hannes Muhleisen. Efficient sql on pandas with duckdb, May 2021.

[6] Ritchie Vink and Chiel Peters. Pola-rs/polars: Dataframes powered by a multithreaded, vectorized query engine, written in rust.

[7] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[8] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[9] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.

7. Appendix:

7.1. Midterm Results: Pandas vs DuckDB vs Polars vs PySpark:

Function	1MB	10MB	100MB	1GB	2GB
compute_speed	0.0007	0.0010	0.0040	0.0395	0.5138
compute_trip_duration	0.0007	0.0014	0.0081	0.0660	0.6371
convert_datetime	0.0039	0.0179	0.1811	1.7670	7.3084
handle_missing	0.0047	0.0160	0.1456	1.5062	3.1978
normalize_features	0.0024	0.0057	0.0401	0.3828	—
one_hot_encode	0.0033	0.0135	0.1432	2.0417	—
save_processed_data	0.1296	1.3556	13.8559	136.6971	—

Table 2. Average runtime (s) of Pandas preprocessing functions

Function	1MB	10MB	100MB	1GB	2GB
compute_speed	0.0017	0.0040	0.0317	0.3494	0.7390
compute_trip_duration	0.0015	0.0034	0.0316	0.3549	0.6864
convert_datetime	0.0015	0.0012	0.0047	0.0477	0.1262
handle_missing	0.0078	0.0169	0.1791	1.6579	3.9133
load_data	0.1244	0.2126	0.5403	3.0195	5.8980
normalize_features	0.0070	0.0194	0.1660	1.6184	7.8065
one_hot_encode	0.0076	0.0482	0.0810	0.6852	2.1904
save_processed_data	0.0193	0.1957	0.8240	7.1596	27.9194

Table 3. Average runtime (s) of DuckDB preprocessing functions

Function	1MB	10MB	100MB	1GB	2GB
compute_speed	0.0020	0.0019	0.0034	0.0089	0.0170
compute_trip_duration	0.0018	0.0018	0.0035	0.0102	0.0154
convert_datetime	0.0038	0.0057	0.0301	0.2643	0.4682
handle_missing	0.0047	0.0089	0.0536	0.4935	1.0542
normalize_features	0.0010	0.0013	0.0123	0.0867	0.1751
one_hot_encode	0.0044	0.0046	0.0079	0.0227	0.0380
save_processed_data	0.0055	0.0494	0.5822	5.7857	11.0032

Table 4. Average runtime (s) of Polars preprocessing functions

Function	1MB	10MB	100MB	1GB	2GB
compute_speed	0.0515	0.0498	0.0560	0.0496	0.0398
compute_trip_duration	0.0412	0.0396	0.0414	0.0399	0.0334
convert_datetime	0.0723	0.0703	0.0748	0.0703	0.0655
handle_missing	1.7908	2.4613	6.3327	26.2351	54.1678
normalize_features	1.9269	1.9204	3.8620	19.7126	42.5870
one_hot_encode	0.1113	0.1061	0.1281	0.0982	0.0911
save_processed_data	0.8375	1.2660	4.4811	27.8291	65.1096

Table 5. Average runtime (s) of PySpark preprocessing functions

7.2. Full Evaluation Results:

Function/Scale	0.005	0.01	0.1	1	2	4	8	10
test_drop_columns	0.000	0.000	0.001	0.006	0.011	0.021	0.041	0.054
test_drop_duplicates	0.003	0.000	0.000	0.001	0.001	0.001	0.001	0.000
test_dropna	0.001	0.001	0.006	0.063	0.122	0.249	0.490	0.624
test_fillna	0.001	0.002	0.016	0.190	0.385	0.814	1.747	2.218
test_get_dummies	0.001	0.001	0.019	0.058	0.102	0.020	0.038	0.074
test_groupby_agg	0.010	0.011	0.037	0.325	0.707	1.564	3.392	4.572
test_isna_sum	0.001	0.001	0.008	0.077	0.147	0.296	0.591	0.750
test_sample	0.001	0.001	0.003	0.002	0.002	0.004	0.009	0.014
tpc_q1	0.008	0.011	0.075	0.753	1.455	3.082	6.062	7.618
tpc_q11	0.003	0.003	0.004	0.215	0.180	0.040	0.068	0.096
tpc_q4	0.006	0.010	0.064	0.689	1.471	3.159	6.456	8.450
tpc_q6	0.001	0.002	0.010	0.096	0.189	0.387	0.767	0.991

Table 6. Average execution times (in seconds) for each function in Pandas across TPC-H scale factors.

Function/Scale	0.005	0.01	0.1	1	2	4	8	10
test_drop_columns	0.001	0.002	0.017	0.159	0.316	0.658	1.321	1.714
test_drop_duplicates	0.003	0.004	0.004	0.003	0.004	0.013	0.013	0.015
test_dropna	0.005	0.009	0.091	0.871	1.756	3.578	7.162	8.960
test_fillna	0.016	0.032	0.346	3.167	6.299	12.683	25.410	32.131
test_get_dummies	0.001	0.002	0.155	1.398	2.772	0.574	1.164	1.492
test_groupby_agg	0.001	0.001	0.005	0.009	0.016	0.029	0.063	0.036
test_isna_sum	0.001	0.001	0.007	0.011	0.021	0.045	0.063	0.063
test_sample	0.002	0.001	0.013	0.131	0.089	0.013	0.015	0.018
tpc_q1	0.002	0.002	0.010	0.027	0.056	0.113	0.198	0.258
tpc_q11	0.003	0.003	0.008	0.005	0.006	0.008	0.011	0.013
tpc_q4	0.008	0.009	0.032	0.182	0.031	0.055	0.061	0.063
tpc_q6	0.001	0.001	0.001	0.002	0.004	0.007	0.013	0.016

Table 7. Average execution times (in seconds) for each function in PyDuck across TPC-H scale factors.

Function/Scale	0.005	0.01	0.1	1	2	4	8	10
test_drop_columns	0.001	0.002	0.017	0.166	0.329	0.653	1.322	1.658
test_drop_duplicates	0.002	0.002	0.002	0.003	0.003	0.006	0.007	0.008
test_dropna	0.004	0.009	0.090	0.906	1.818	3.584	7.185	8.756
test_fillna	0.020	0.040	0.434	4.106	8.155	16.141	32.563	40.073
test_get_dummies	0.000	0.002	0.025	0.004	0.007	0.015	0.020	0.026
test_groupby_agg	0.004	0.005	0.009	0.018	0.033	0.059	0.117	0.154
test_isna_sum	0.001	0.001	0.005	0.006	0.011	0.019	0.037	0.055
test_sample	0.001	0.001	0.011	0.001	0.001	0.002	0.002	0.003
tpc_q1	0.002	0.002	0.010	0.021	0.042	0.079	0.097	0.123
tpc_q11	0.001	0.002	0.005	0.004	0.005	0.006	0.009	0.011
tpc_q4	0.005	0.004	0.066	0.018	0.024	0.043	0.045	0.045
tpc_q6	0.000	0.000	0.007	0.002	0.004	0.007	0.012	0.014

Table 8. Average execution times (in seconds) for each function in DuckDB across TPC-H scale factors.