Magpie: Python at Speed and Scale using Cloud Backends

Alekh Jindal
Gray Systems Lab, Microsoft
alekh.jindal@microsoft.com

K. Venkatesh Emani
Gray Systems Lab, Microsoft
k.emani@microsoft.com

Maureen Daum*
University of Washington
mdaum@cs.washington.edu

Olga Poppe
Gray Systems Lab, Microsoft
olga.poppe@microsoft.com

Brandon Haynes
Gray Systems Lab, Microsoft
brandon.haynes@microsoft.com

Anna Pavlenko
Gray Systems Lab, Microsoft
annapa@microsoft.com

Ayushi Gupta*
Apple
ayushi.iiit@gmail.com

Karthik Ramachandra
Microsoft Azure Data
karam@microsoft.com

Carlo Curino
Gray Systems Lab, Microsoft
carlo.curino@microsoft.com

Andreas Mueller
Gray Systems Lab, Microsoft
andreas.mueller@microsoft.com

Wentao Wu
Microsoft Research
wentao.wu@microsoft.com

Hiren Patel
Microsoft
hirenp@microsoft.com

ABSTRACT

Python has become overwhelmingly popular for ad-hoc data analysis, and Pandas dataframes have quickly become the de facto standard API for data science. However, performance and scaling to large datasets remain significant challenges. This is in stark contrast with the world of databases, where decades of investments have led to both sub-millisecond latencies for small queries and many orders of magnitude better scalability for large analytical queries. Furthermore, databases offer enterprise-grade features (e.g., transactions, fine-grained access control, tamper-proof logging, encryption) as well as a mature ecosystem of tools in modern clouds.

In this paper, we bring together the ease of use and versatility of Python environments with the enterprise-grade, high-performance query processing of cloud database systems. We describe a system we are building, coined `Magpie`, which exposes the popular Pandas API while lazily pushing large chunks of computation into scalable, efficient, and secured database engines. `Magpie` assists the data scientist by automatically selecting the most efficient engine (e.g., SQL DW, SCOPE, Spark) in cloud environments that offer multiple engines atop a data lake. `Magpie`'s common data layer virtually eliminates data transfer costs across potentially many such engines. We describe experiments pushing Python dataframe programs into the SQL DW, Spark, and SCOPE query engines. An initial analysis of our production workloads suggest that over a quarter of the computations in our internal analytics clusters could be optimized through `Magpie` by picking the optimal backend.

*Work done while at Microsoft.

11th Annual Conference on Innovative Data Systems Research (CIDR '21). January 10-13, 2021, Chaminade, USA.

1 INTRODUCTION

Python has become the lingua franca for ad-hoc data analysis (typically over text or CSV files), driven primarily by its concise, comprehensible code that requires less time and effort to write relative to other languages. Furthermore, there is a rapid convergence towards dataframe-oriented data processing in Python, with Pandas dataframes being one of the most popular and the fastest growing API for data scientists [46]. Many new libraries either support the Pandas API directly (e.g., Koalas [15], Modin [44]) or a dataframe API that is similar to Pandas dataframes (e.g., Dask [11], Ibis [13], cuDF [10]). This trend has resulted in a language surface for data science in Python that is increasingly well defined and converging on a common set of primitives. Notwithstanding this popularity, scaling data processing with Pandas and achieving good performance in production remains a substantial challenge [9, 20, 25, 42].

Cloud environments, on the other hand, have enabled users to manage and process data at hyper scale. Furthermore, in contrast to the effort of setting up a database on-premise, the cloud has made it ridiculously easy to try out and operate a variety of *backends*, i.e., database services with either co-located or disaggregated storage, on demand [4, 5]. As a result, modern enterprises are more likely to already have their data in the cloud and operate multiple backends, each optimized for different scenarios. The question therefore is whether we can bring these two worlds together: the versatility and easy-to-use aspects of Python data processing environments, and the scale and performance of cloud environments.

The above question exposes a daunting set of challenges. Pandas evaluates data operations eagerly over in-memory data while cloud backends process large query graphs that are pushed closer to the data in distributed storage. It is often tedious to embed Python code (usually via UDFs) into cloud backends, which generally provide a SQL interface. Despite the rise of newer managed services such as Azure Synapse [7] which allow customers to effortlessly switch between different backends, choosing between these backends is a challenging problem (as has been observed in previous polystore systems [26]). Finally, providing a good data science experience on

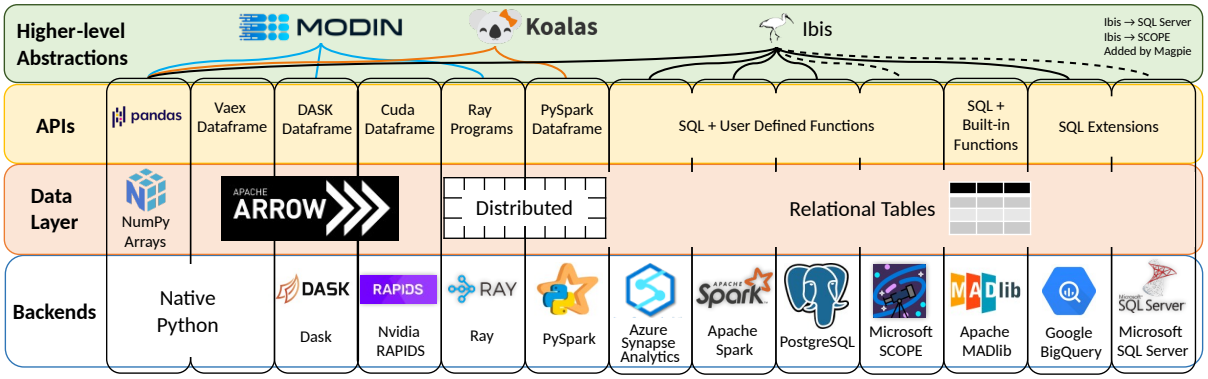


Figure 1: The data science jungle from a subset of current data science tools.

top of cloud data services remains nontrivial, especially given the unprecedented growth in data volume and complexity.

Fortunately, we identify four key enablers to bridge the gap between the Python and cloud worlds. First, there is an active ongoing effort, led by the broader Python community, to *standardize* the Python APIs for data by bringing different variants of dataframe APIs under a common umbrella [1]. Second, many dataframe operations can be mapped to relational algebra, as also shown in prior work such as Modin [44] and Ibis [13], and hence can be *pushed down* to a database (we discuss related work further in Section 2). Third, there is a shift from polystores (e.g., [26, 38, 39]) to *polyengines* operating on top of common disaggregated storage in the cloud. As a result, data is no longer locked away in database stores and query engines are increasingly fungible. Finally, there is an emergence of a *common data format*—Apache Arrow[2]—which significantly reduces data transfer costs and makes backend interoperation practical.

Motivated by these trends, in this paper we present a vision of scalable and efficient data science on top of cloud backends. Our goal is to let data scientists focus on data analysis using familiar Python APIs and then transparently execute their workloads on the best-performing backend. We present `Magpie`, a data science middleware that exposes the popular Pandas API, automatically selects the best-performing cloud backend for each Pandas script, lazily pushes down large chunks of Pandas operations to those backends, and uses Arrow and ArrowFlight to communicate efficiently and cache intermediate results at the data layer. We illustrate the benefits with `Magpie` in two different analytics environments at Microsoft—a data warehousing environment consisting of data already within Azure SQL Data Warehouse (SQL DW) and a big data analytics environment consisting of SCOPE and Spark query engines with disaggregated storage. Our experiments show that pushing Pandas down into SQL DW can improve the performance by more than 10×, while judiciously picking between SCOPE and Spark backends could improve up to 27% of our production workloads (with a median performance improvement of 85%).

The rest of the paper is organized as follows. In Section 2, we discuss how the current tools are hard to navigate for data scientists and related efforts to address this problem. We present an overview of `Magpie` in Section 3. In Sections 4 and 5, we discuss the techniques behind pushing Pandas operations into backends. Section 6 describes how `Magpie` chooses the best engine for a workload. We describe our common data layer in Section 7 and conclude in Section 8 with a discussion of the road ahead.

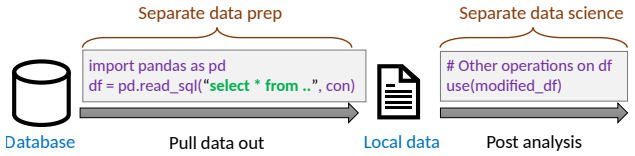
2 THE DATA SCIENCE JUNGLE

As illustrated in Figure 1, current data science tools confront data scientists with a jungle of higher-level abstractions, APIs, data layers, and backends. The plethora of tools is driven by two factors. First, the Pandas API is the de-facto choice for data scientists to analyze data. This is supported by the fact that nearly all the Python-based tools in Figure 1 provide dataframes-based APIs that resemble the Pandas API. Second, Pandas is not designed to scale to large data, as noted by the creators of Pandas [41].

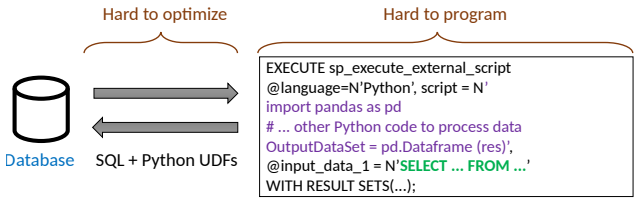
On the one hand, solutions have evolved within the Python ecosystem to scale Pandas. Modin, for instance, is backed by the Ray/Dask parallel processing engines that use distributed Pandas dataframes to leverage multiple cores for faster processing. CuDF is backed by RAPIDS and provides libraries to execute data analytics pipelines on GPUs, building on NVIDIA CUDA primitives for low-level compute optimizations. PySpark provides a Python API to interact with the Spark ecosystem, and Koalas takes it a step further by mimicking the Pandas API backed by Spark distributed processing. Furthermore, Apache Arrow in-memory columnar data format coupled with ArrowFlight for data transmission is fast developing as a unifying data storage format across data engines and APIs.

On the other hand, hybrid approaches have evolved within the database ecosystem to push the capabilities of relational engines to support dataframe and tensor computations. The Apache MADlib project [3] provides SQL-based parallel implementations of algorithms for in-database analytics and machine learning at scale. Most popular databases including PostgreSQL and SQL Server, and big data processing systems including Microsoft SCOPE and Google BigQuery support Python interoperability either natively or through separate solutions such as Apache Beam.

We can see how different tools are siloed across similar but different sets of APIs (variants of dataframes or SQL), different data layer abstractions that range from arrays and in-memory columns to distributed file systems and relational tables, and very different



(a) Data science often involves multiple, fragmented scripts that (expensively) extract data to be analyzed locally.



(b) Performing data science within a single system results in complex programs that are difficult to optimize.

Figure 2: Current approaches to data science for data already in a database.

processing backends ranging from custom Python backends to cloud data processing systems. As a result, the current data science approach is *fragmented* and *difficult to program and optimize*.

To illustrate the above challenges, consider the case when data is already within a database (e.g., SQL DW), a standard scenario in the cloud. The typical data science process (Figure 2a) involves an ad hoc patchwork of scripts that pulls relevant data out (e.g., using Python libraries such as PyODBC [18] or SQLAlchemy [23]) that is then separately analyzed locally (e.g., in a Jupyter notebook [14]). Pulling data out of a database forces data scientists to optimize data movement and data preparation before performing the actual data analysis. This process also makes subsequent scaling difficult.

Alternatively, data scientists might attempt to perform work using a single system, as illustrated in Figure 2b. This approach, however, poses significant implementation- and optimization-related challenges. It involves intermixing SQL queries with Python user defined functions (UDFs) to push computation inside the database engine (e.g., using PL/Python [17], MadLib [3], Myria [51], or SQL Server ML Services [22]). Furthermore, mixing SQL and Python UDFs requires expertise in multiple languages, often in a nonstandard dialect (e.g., PostgreSQL or SQL Server), and it has been historically very hard to optimize imperative UDFs pushed into the database engine [47]. Furthermore, Python has a fast growing set of libraries and supporting different versions of them within a database (along with a good debugging experience) is nontrivial.

Several recent works attempt to bridge the gap between Python and database ecosystems. Closely related efforts include Modin [44], Dask [11], Koalas [15], Ibis [13], Vaex [24] and others, which provide a dataframes surface and push down computations into backends. We compare various aspects of these efforts below.

Language surface. Modin and Koalas provide Pandas compatibility, while Ibis, Dask and Vaex provide a variant of the Pandas dataframes API. The mismatch in API requires data scientists to rewrite their code from Pandas into the framework specific APIs to benefit from scaling. Also, Modin does not consider the typical data processing backends available in current clouds, while Koalas only maps to Spark (other than Pandas) even though there might be other backends available in analytics environment.

Lazy evaluation. Lazy evaluation refers to an evaluation strategy that delays the evaluation of an expression until its value is needed [52]. Lazy evaluation is usually supported by functional languages, and it's use for improving performance of database programs has been explored in [29]. Lazy evaluation of dataframes has been used in [11, 13, 15, 44] to build expression trees for execution on a backend. However, lazily constructed expressions could be further used for optimizations such as caching and automated backend selection.

Supported backends. Modin currently provides Pandas APIs on Ray, Dask, or PyArrow as alternate backends. Koalas provides Pandas APIs that can run on the Spark ecosystem. The Ibis framework is able to map dataframe operations expressed in Ibis APIs onto a number of relational and big data processing backends supported by Ibis (the complete list of supported backends is given in [13] including CSV and Pandas backends). The Ibis framework is also extensible to add support for other backends, such as SQL Server and SCOPE.

Dataframe Algebra. Petersohn et al. [44] presented a *dataframe algebra* to formalize the set of operations offered by dataframe APIs such as Pandas. Petershohn et al. identify that these operations are a combination of relational algebra, linear algebra, and spreadsheet computations. There are other efforts to unify relational and linear algebra with an aim to support dataframe operations in databases [31, 37, 50]. Thus, a direct translation of Pandas into database backends (which are based on relational algebra) may not always be possible.

However, the need for a query planner for Pandas expressions is well established [41]. Database backends provide query planning and optimization capabilities for relational algebra expressions. To examine the significance of relational operations in the Pandas API, we conducted an analysis on real world data science notebooks from the GitHub Archive dataset [12]. We analyzed notebooks with star rating $\geq$ 10, and ranked the number of invocations of each Pandas method in descending order. Of the top-20 methods (totaling over 50K occurrences), more than 60% correspond to those methods that perform relational operations or operations that are commonly supported by popular database systems (e.g., exporting results to CSV). This suggests that a significant number of popular Pandas operations can be pushed down to database backends.

Pushing imperative code to databases. Pushing imperative code to databases is an area that has received significant attention. Related efforts include techniques for pushing Java code using object-relational mapping APIs into SQL queries [28, 32], query batching [29, 35], partitioning of application programs [27], cost-based transformations for database applications [34], etc. The Myria system [51] translates programs specified using a PySpark-like API into various backends, including pushing down Python UDFs. In this work, we propose a runtime approach with a focus on programs using the Pandas API interspersed with other Python code, such as visualizations and model training.

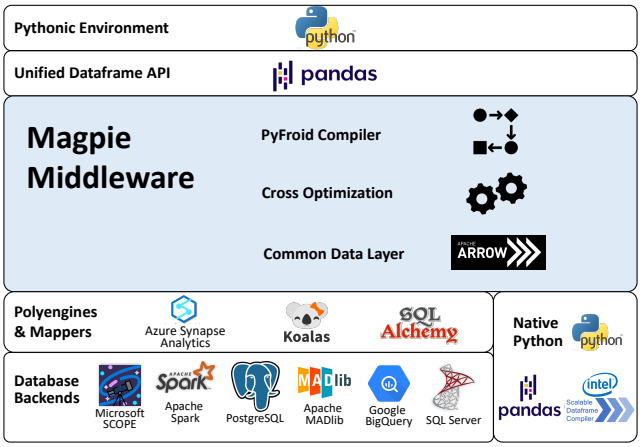


Figure 3: Our vision for a more simplified, unified, and efficient data science stack.

3 MAGPIE OVERVIEW

Figure 3 shows our vision for a simplified, unified, and efficient data science stack. Our goal is to bridge the Pythonic environments at the top with the cloud backends at the bottom. Specifically, we want to let data scientists write programs in the Pandas API and map them to the Python layer that already exists for most cloud backends[1] To achieve this, the `Magpie` middleware consists of a compiler, a backend optimizer, and a common data layer. The compiler converts Pandas computations into logical expressions, batches them together using Ibis [13], and decides when to materialize them using the underlying backends. The backend optimizer selects the most-performant backend amongst the available ones using a decision tree classifier learned from past observations available on the cloud. The common data layer helps improve the database interface, facilitate interactivity by caching results in memory, and combining data from different sources. Overall, `Magpie` helps improve the data science lifecycle in several ways:

Pandas without regret. One of the key factors behind the wide adoption of Pandas besides the imperative interface and relatively small learning curve is its interoperability with other popular systems such as `matplotlib`, Jupyter notebooks, `sklearn`, etc. In one of our recent projects at Microsoft, data scientists implemented a data cleaning module using Pandas. They tested locally using a sample dataset, but later they had to scale it to a large dataset on Cosmos. They ended up *rewriting* their module using SCOPE, the query engine on Cosmos. The data scientists could have tried alternate engines like Dask [11] to scale their programs, however, operationalizing new engines in a cloud environment is non-trivial and something which data scientists are not expert at. Examples like this illustrate the gap between data science APIs and the data processing tools, and exposes the dilemma that data scientists face everyday. `Magpie` relieves data scientists of this dilemma and lets them focus on their tasks using the Pandas API without any subsequent regret. We achieve this by transparently executing Pandas computations on backends and taking care of materializing the results when required. Our approach is to push down operations to backends whenever possible; for other operations with no external side effects, we extract a User Defined Function (UDF) that can run on a co-located Python runtime at the database. It may also be possible to use frameworks such as Dask for out of core processing on resulting data after partially pushing down computations into backends. Our preliminary evaluation (refer Section 5) shows that relational operations can be significantly sped up by pushing down to backends.

Abstracting data processing complexity. Data scientists may not be experts in the underlying techniques for data management. As a result, it may be hard for them to figure out details like data partitioning, indexing, movement, joins, etc., especially when different cloud backends implement these concepts differently. Even working with different data formats, given that CSV is the de facto file format for data science, could be challenging. `Magpie` abstracts the data processing details by automatically mapping to different cloud backends and even picking amongst different ones for different scenarios. Using `Magpie`, data scientists can start off their explorations locally on a sample of the data using CSV files. Later, they can easily port their solutions to the backend of choice using leveraging `Magpie`'s automatic push down from Pandas into various backends.

Write once, execute anywhere. As discussed earlier in Section 1, polyengines are now commonplace thanks to disaggregated cloud storage, and different backends may be suitable for data science at different times, e.g., as the data volume grows. `Magpie` allows data scientists to write their programs once and execute anywhere, e.g., switch to more powerful or scalable cloud backends as the data grows, debug locally on smaller data using just the Pandas backend, or even port the same code to completely different environments or different clouds.

We note here that `Magpie` requires users to provide a list of backends available to them for computations on their data. From these backends, `Magpie` can choose the best backend and run the workload efficiently on the selected backend.

In-situ data science. Data movement is often the biggest blocker to getting started with the data science, and so it is desirable to run data science right where the data is without moving it around. The advent of GDPR regulations has established the need for in-database data science solutions [46]. This is further emphasized with newer HTAP scenarios that require analytics on top of the operational data. However, the lack of a familiar and unified API makes it difficult for data scientists to build solutions efficiently.

4 PANDAS SURFACE FOR CLOUD BACKENDS

`Magpie` enables users to use the familiar Pandas API without sacrificing the scale and performance provided by cloud backends. This is achieved through a compiler, coined PyFroid, that translates Pandas into a backend-agnostic intermediate representation, which is then translated into an executable query on a *target backend* selected by `Magpie`. We defer the underlying details about constructing and optimizing the intermediate representation to Section 5.

Figure 4 illustrates the journey from Pandas to cloud backends. For simplicity, we hereafter refer to programs that use the Pandas

[1]We added a Python layer for SCOPE.

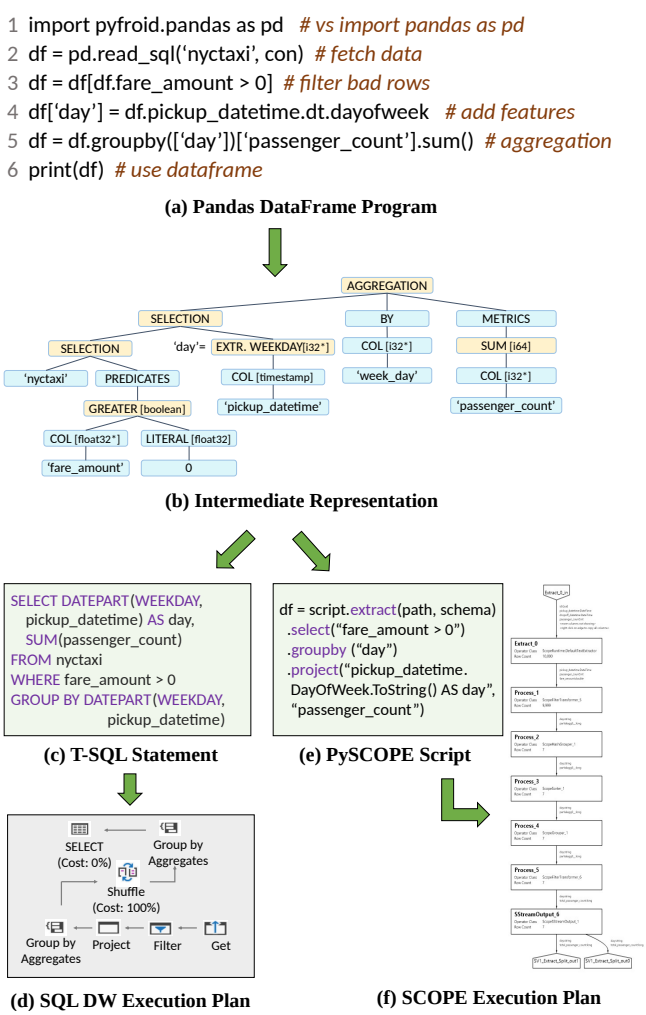


Figure 4: Pushing Pandas dataframes to cloud backends.

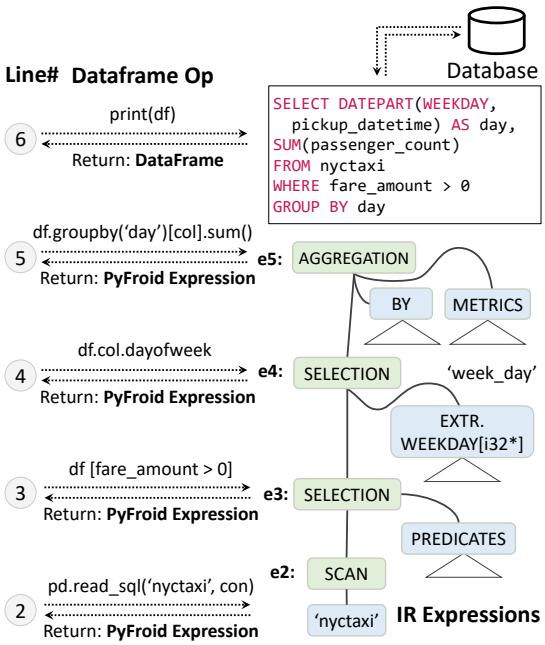


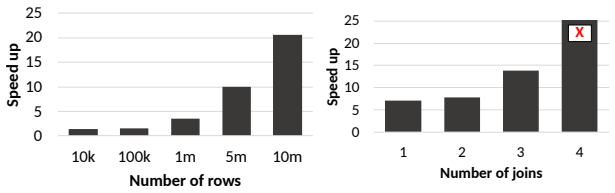
Figure 5: PyFroid Lazy Evaluation

API as Pandas programs. The Pandas program in Figure 4a computes the number of taxi trips per weekday over the NYC Taxi [16] dataset, a common benchmark for data science and exploration. We created a Pandas script that includes commonly used operations including selections, projections, and group by aggregations. Data scientists can run their existing Pandas programs using Magpie by just modifying one line in their scripts (as shown in line 1 of Figure 4a). Figure 4b shows the compiled version of this program, capturing computations across multiple imperative statements (lines 2 to 5 in the program) into a single logical query tree. As a result, Pandas programs can now be executed as large chunks of computations, potentially in parallel, compared to the default eager evaluation offered by Pandas. Representing Pandas programs as a logical query tree also allows us to apply database-style query optimization techniques, i.e., it decouples the Pandas programs from their physical execution on cloud backends. Our current implementation uses Ibis [13] for the logical query tree representation.

PyFroid currently handles selections, projections, feature additions, joins, and aggregations. However, for unsupported or untranslatable operations, PyFroid falls back to ordinary Pandas operations (prior translatable operations are still executed on the backend). This fallback is important as the Pandas API contains a large set of rapidly growing operations, some of which may not have counterparts in the database backend (e.g., `DataFrame.interpolate()`). PyFroid can also push Python functions as UDFs to the database, which then may be executed on a co-located interpreter. This further opens up interesting research challenges for applying a number of optimizations. Preserving the ordering of data in dataframe computations (explicit ordering such as that obtained through `sort` or implicit ordering as obtained from other data sources) during push down to backends is an important challenge. Techniques to handle ordering from prior work on translating imperative code to declarative representations [28, 33] can be leveraged for this purpose. The unrestricted combination of Pandas statements with other Python code, such as visualizations and model training, presents new research challenges such as handling complex control flow, using program analysis, and rewriting to maximize the computation pushed into the database.

5 PUSHING *DATA SCIENCE* DOWN

We saw in the previous section how Pandas programs could be translated to logical query trees. In this section, we describe how the trees expressions are lazily constructed and evaluated. Then, we describe pushing them down to two popular cloud backends at Microsoft: SQL DW and the SCOPE big data engine in Cosmos. Finally, we show some of the benefits of pushing data science down.



(a) By input size **(b) By query complexity**

Figure 6: Pushdown performance

Lazy Evaluation. Pandas eagerly evaluates each method call. By contrast, PyFroid does the following: (a) it parses the Pandas statement to extract information about the underlying dataframe computation, (b) registers the computations as part of an encapsulated intermediate representation, called a *PyFroid Expression (PFE)*, and (c) returns a copy of the modified `PFE` object. This process is illustrated in Figure 5. For instance, after line 2, the table expression `e2` (i.e., `SCAN(nyctaxi)`) is added to the PFE. After line 3, the PFE is updated to a new expression `e3` rooted at `SELECTION` operator and children as `e2` and a `PREDICATES` subexpression, which contains the filter `fare_amount > 0`. In this way Pandas computations are incrementally accumulated into a PFE.

Evaluation of a `PFE` is delayed until the dataframe needs to be materialized for use in the program (i.e., a *sink* statement). For example, in Figure 5, line 6 contains the sink statement `print`, which forces the dataframe to be materialized. At this point, PyFroid translates the PFE into a query tree, submits it to the cloud backend for optimization and execution, and returns the result as a dataframe. We currently identify the following statements as sinks to force dataframe materialization: (a) statements that require the contents of dataframes to be printed or passed on to other library functions, (b) occurrences of loops that iterate over the dataframe, or (c) statements that use parts of the Pandas API that are that unsupported by PyFroid. For unsupported operations, PyFroid materializes a result and returns a `DataFrame` object (instead of a `PFE`). This enables graceful fallback to Pandas.

Cloud Backends. PyFroid leverages Ibis [13] for query tree representation and Ibis provides support for a number of relational and hadoop-based backends, including Spark, PostgreSQL, BigQuery, etc. Ibis specifies a set of common APIs to be implemented by each backend, to facilitate code generation from Ibis expressions to backend-specific queries and translation between database results and dataframes. Ibis provides driver routines that invoke specific implementations of these APIs depending on the backend.

As part of our work, we have added support for SQL DW and SCOPE backends. Figure 4c and Figure 4d shows the SQL DW query and the corresponding execution plan for our running example. In this case, the database engine chose a parallelizable plan consisting of partial group by aggregates followed by shuffle and global aggregates. Alternatively, `Magpie` might select SCOPE as the target backend. In this case, Figure 4e and Figure 4f shows the corresponding PyScope (our Python layer for SCOPE) expressions and the SCOPE query plan. Note that users do not have to worry about implementing the SQL or SCOPE queries, or tuning the underlying query plans. They only write their Pandas programs and the system takes care of pushing them down to the cloud backends.

Our system design amortizes the cost of adding a new backend. The PyFroid compiler exposes a common Pandas interface that can be compiled down to any backend supported by Ibis. Thus, to add a new backend, we only need to add a new backend to Ibis. This was also a design goal of Ibis and our work extends the benefit of this amortization, enabling existing data science solutions to be migrated onto cloud engines without rewriting.

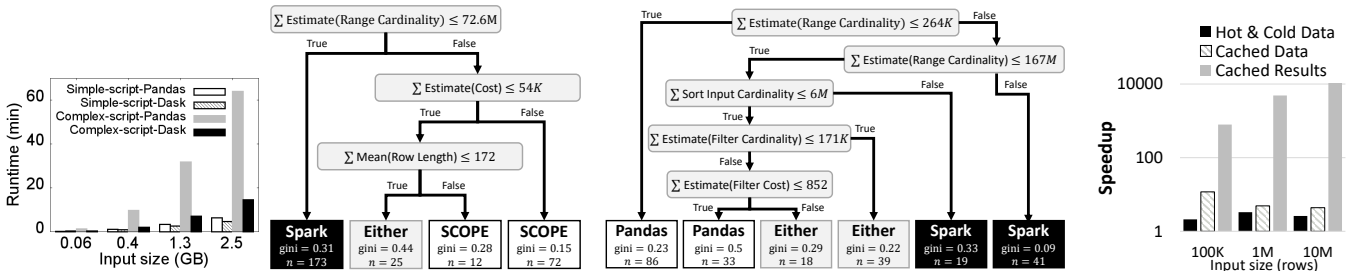
Pushing data science down matters. Figure 6 summarizes our early evaluation results. Our experiments are run on SQL DW (Gen2 DW 100c) as the data store, with Pandas programs accessing the data from an Azure Standard D8s v3 machine (8 vCPUs, 32 GiB memory) running Ubuntu 18.04. First, we run the running example (Figure 4a) using Pandas and PyFroid (i.e., pushed into database) by varying the table size. Figure 6a shows the speedup when the program is run using PyFroid compared to ordinary Pandas. Next, we vary the complexity of the script in terms of number of joins involved and compare in-database vs in-Pandas performance. For this experiment, we fix the size of nyctaxi at one million rows (around 120MB on database disk); each other table participating in the join is obtained by randomly selecting five rows from nyctaxi. The results are shown in Figure 6b. The *X* indicates that Pandas ran out of resources and was killed. As seen in Figure 6, as the data size or complexity of computations increases, PyFroid significantly outperforms Pandas. However, the choice of which engine to choose based on data statistics and workload complexity is a challenging problem, which motivates our next section.

6 MANAGING THE ENGINE ZOO

As discussed earlier, polyengines are a reality in modern clouds. The underlying data could be in a variety of locations such as operational stores, data warehouses, or data lakes. For operational data, there is increasingly a trend to make it available for hybrid transaction and analytical processing (HTAP). For example, Synapse Link makes operational data from Cosmos DB available to Synapse [8]. For data warehouses, in order to minimize data movement, we currently push data science to the respective store. However, future work should explore whether it makes sense to pull out intermediate results from the data warehouse for more efficient subsequent local processing. Data lakes [48] on the other hand provide shared, disaggregated storage that can be accessed by multiple engines, including data warehouses [19, 21], *without any additional data movement costs*.

Despite this reality, current tools require the data scientists to choose database backends manually. Modin, for instance, requires users to pick between Dask, Ray, or Pandas backend. This is challenging since a data scientist must study available engines and select one for each specific use case. Below we describe three production scenarios to illustrate why backend selection is nontrivial for data scientists and how the backend optimizer in `Magpie` can help them when working on a data lake.

Load Prediction on the Cloud. We recently deployed an infrastructure, called Seagull, for predicting the load of database services and to optimize resource allocation [45] To ensure scalability across



(a) Manual experiments (b) Decision tree: Spark & SCOPE (c) Decision tree: Spark & Pandas

Figure 7: Manual vs automatic choice of engine

Figure 8: Speedups via caching in ArrowFlight

all Azure regions, we had to pick the most efficient backend for the data processing component written using Pandas APIs and executed using Azure Machine Learning [6] containers that fetch data from Azure Data Lake Storage [48]. To this end, we manually ran scripts of various complexity against input data of different sizes on Pandas and on Dask (Figure 7a). We concluded that speed-up of the multi-threaded execution on Dask increases with growing input size and script complexity compared to the single-threaded execution on Pandas. For a complex script and 2.5GB of input, Dask achieves over 4× speed-up compared to Pandas on an Ubuntu VM with 16 cores and 64GB of RAM. Running these experiments manually is labor-intensive, time-consuming, error-prone, and hard to generalize to any script or input data. Data scientists have neither time nor expertise to conduct a rigorous study for all engines. The above Seagull experience further motivated our exploration for backend selection, even on a single machine, and demonstrated the feasibility of an automated approach.

Backend Selection in Cosmos. Previously, Cosmos had SCOPE as the only query engine. SCOPE scripts perform petabyte-scale preparation and analysis of production telemetry. Recently, Spark has been introduced in Cosmos as well, which means both SCOPE and Spark can process data from the same underlying Azure Data Lake Storage. Now, the question is which SCOPE scripts can be executed faster on Spark. We ran a workload of equivalent PyScope and PySpark scripts on SCOPE and Spark on Cosmos. Based on typical data preprocessing steps in Cosmos, we automatically generated 68 script templates and executed them on inputs with sizes ranging from 1MB to 1TB. The scripts contained operators such as filters, projections, groupbys, aggregations, joins, reads, and writes. Each script contained between 3 and 20 operators, with an average of 7 operators per script.

We extracted features from this workload such as query operators, input and output cardinalities, row length, and per-engine runtimes [49]. Based on these features, we trained a decision tree that automatically picks an engine with minimal expected runtime for a given script (Figure 7b). We chose a decision tree since product teams (and customers) want to understand why a particular engine was chosen—particularly in case of an incident or unexpected result. We observed cases where neither engine always performs better for a given script due to variance in runtimes and so we added a third label "Either" for such cases. We trained the tree using scikit-learn [43] and used cross-validation to pick the appropriate values of the hyperparameters `max_depth`, `min_samples_split`, `max_leaf_nodes`, and `min_samples_leaf`.

The decision tree in Figure 7b achieves 87% accuracy on the test set, which contained 25% of the scripts. For the scripts that executed more quickly on Spark than on SCOPE, we observed up to 98% improvement in runtime, with a median improvement of 85%.

As Figure 7b illustrates, if the input data contains fewer than 72M rows, Spark wins over SCOPE, while SCOPE wins when the input data is large and the script has high estimated cost. Based on this decision tree, we can automatically translate and redirect respective scripts using the `Magpie` API from SCOPE to Spark to reduce their runtime. In summary, we conclude that the runtime of more than a quarter of scripts on Cosmos can be reduced by up to half by moving them from SCOPE to Spark.

Local vs Cluster Execution. Given that data scientists are used to working with local CSV files, it is not easy for them to decide when to switch to cluster-based execution in Cosmos, i.e., when the input is small, it may be most efficient to execute the script on a single node using Pandas. However, as the size grows it may be more efficient to move the data to the data lake and process using one of the available cloud engines. Based on this intuition, we trained another decision tree to decide whether Pandas (running locally) or Spark on Cosmos (after moving the data to the data lake) has shorter expected runtime using the same features and training procedure as above. We measured runtimes by executing equivalent scripts written using Pandas on a Surface Pro 7 with an Intel Core i7 CPU and 16GB of RAM. The tree shown in Figure 7c achieves 84% accuracy on the test set, which contained 25% of the scripts. For the scripts that ran more quickly on Pandas, we observed up to 99% improvement in runtime, with a median improvement of 84% in runtime.

An alternative approach is to frame backend selection as a regression problem and predict the runtime of scripts on each engine. However, training high-quality runtime prediction models is difficult, and recent work on a similar problem of index selection has shown that positioning the decision as a classification task can lead to superior performance [30]. While the search space for this problem is unbounded, few features matter in practice. And since identifying the decision points is nontrivial, `Magpie` can leverage large cloud workloads to provide robust backend selection decisions.

7 BETTER TOGETHER

`Magpie` uses Arrow [2] and ArrowFlight [40] as a common format for data access across each of its backend engines. It serializes intermediate results as Arrow tables and uses ArrowFlight as an

in-memory and wire format when moving data between systems. For co-located engines, this virtually eliminates transfer cost, and greatly reduces it for inter-node transfer. We now describe two scenarios where Magpie's common data layer brings things together.

Different data sources together. A common scenario in the cloud is to combine cold and near-archival data with hot and near-operational data (e.g., combining data from Cosmos and SQL DW). The typical approach is to ingest data from one source to the other before running the analysis. Magpie, however, pushes the computations (as much as possible) down the two sources *before* combining the result. Furthermore, given that the cold data is typically large and more stationary by definition, Magpie caches it in ArrowFlight, whereafter it can be quickly joined with the constantly changing hot data. Figure 8 shows 2–3× speedups in our running example query over the NYC taxi dataset, when combining $100K$ rows of hot data in SQL DW with varying sizes of cold data in Cosmos. Equally importantly, Magpie allows processing over various file formats (e.g., CSV and Cosmos' StructStream), presenting a unified view to data scientists.

Different data scientists together. Data science is both an iterative and a collaborative process, with data scientists often running partially or fully same computations on the same data. Magpie can cache these common computations or even just the common frequently used datasets (e.g., recent sales or telemetry) in Arrow flight server to avoid going back to cloud backends every time. For instance, it is desirable to hit the SQL DW backends less frequently to make it more available for data warehousing workloads. Figure 8 shows the speedups if the data or the results of our running example were cached in ArrowFlight. We show that with increasing sizes, the speedups decrease for caching data but increase for caching the results directly. Caching and reusing also deduplicates the dataframes loaded in memory and reduces workspace memory footprint.

A common, efficient serialized and wire format across data engines is a transformational development. Many previous systems and approaches (e.g., [26, 36, 38, 51]) have observed the prohibitive cost of data conversion and transfer, precluding optimizers from exploiting inter-DBMS performance advantages. By contrast, in-memory data transfer cost between a pair of Arrow-supporting systems is effectively *zero*. Many major, modern DBMSs (e.g., Spark, Kudu, AWS Data Wrangler, SciDB, TileDB) and data-processing frameworks (e.g., Pandas, NumPy, Dask) have or are in the process of incorporating support for Arrow and ArrowFlight. Exploiting this is key for Magpie, which is thereby free to combine data from different sources and cache intermediate data and results, without needing to consider data conversion overhead.

8 THE ROAD AHEAD

In this paper we introduced Magpie, a system that we are building to bring together the increasingly popular Pandas API as an interface for data science and decades of performance and scalability optimizations found in database backends on the cloud.

Many challenges remain. With the Pandas community converging rapidly on a unified API, Magpie should ultimately strive for full compatibility and evolve with the standard. This opens up additional opportunities for optimizations (e.g., matrix multiply) and new operations to be pushed down. Ultimately, full unification will require even deeper system integration.

Finally, backend selection remains a longstanding and formidable research challenge. The trend toward disaggregated storage and a common data format offers a new opportunity for resolving this challenge. However, many open problems remain, such as maximizing user experience (e.g., unifying error messages across dissimilar backends), exploring federated or multi-backend execution, and efficiently integrating new backend systems into a learned model.

Acknowledgments. We would like to thank the following teams and individuals for their invaluable assistance, insight, and support: Milos Sukovic and Brenden Niebruegge for discussions on Arrow based processing, Christopher Ostrouchov for efforts on incorporating SQL Server into Ibis, and the SCOPE (especially Hiren Patel) and Spark on Cosmos (especially Newton Alex, Akshat Bordia, Manoj Kumar, Sriram Ganesh, Abhishek Modi, Prakhar Jain, Abhishek Tiwari) teams for their integration expertise and feedback.

REFERENCES

[1] Announcing the Consortium for Python Data API Standards. https://data-apis.org/blog/announcing_the_consortium/.
[2] Apache Arrow. https://arrow.apache.org.
[3] Apache MADlib: Big Data Machine Learning in SQL. https://madlib.apache.org/index.html.
[4] AWS Databases. https://aws.amazon.com/products/databases.
[5] Azure Databases. https://azure.microsoft.com/en-us/services/#databases.
[6] Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/.
[7] Azure Synapse Analytics. https://azure.microsoft.com/en-us/services/synapse-analytics/.
[8] Azure Synapse Link. https://docs.microsoft.com/en-us/azure/cosmos-db/synapse-link.
[9] Beyond Pandas: Spark, Dask, Vaex and other big data technologies battling head to head. https://towardsdatascience.com/a453a1f8cc13.
[10] cuDF - GPU DataFrames. https://github.com/rapidsai/cudf.
[11] Dask. https://dask.org/.
[12] GH Archive. https://www.gharchive.org/.
[13] Ibis: Python data analysis framework for Hadoop and SQL engines. https://github.com/ibis-project/ibis.
[14] Jupyter Notebook. https://jupyter.org/.
[15] Koalas. https://github.com/databricks/koalas.
[16] NYC Taxi demo data for tutorials. https://docs.microsoft.com/en-us/sql/machine-learning/tutorials/demo-data-nyctaxi-in-sql.
[17] PL/Python. https://www.postgresql.org/docs/9.0/plpython.html.
[18] pyodbc. https://pypi.org/project/pyodbc/.
[19] Redshift Spectrum. https://aws.amazon.com/blogs/big-data/amazon-redshift-spectrum-extends-data-warehousing-out-to-exabytes-no-loading-required/.
[20] Scaling Pandas: Comparing Dask, Ray, Modin, Vaex, and RAPIDS. //datarevenue.com/en-blog/pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray.
[21] SQL Data Warehouse and Azure Data Lake Store. https://azure.microsoft.com/en-us/blog/sql-data-warehouse-now-supports-seamless-integration-with-azure-data-lake-store/.
[22] SQL Server ML Services. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services.
[23] The Python SQL Toolkit and Object Relational Mapper. https://www.sqlalchemy.org/.
[24] Vaex.io: An ML Ready Fast Dataframe for Python. https://vaex.io/.
[25] Which one should I use Apache Spark or Dask. https://medium.com/@prayankkul27/22ad4a20ab77.
[26] V. G. P. Chen, J. Duggan, A. J. Elmore, B. Haynes, et al. The BigDAWG Polystore System and Architecture. In *HPEC*, pages 1–6, 2016.
[27] A. Cheung, O. Arden, S. Madden, and A. C. Myers. Automatic partitioning of database applications. *arXiv preprint arXiv:1208.0271*, 2012.
[28] A. Cheung et al. Optimizing Database-backed Applications with Query Synthesis. *ACM SIGPLAN Notices*, 48(6):3–14, 2013.
[29] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Leing Lazy is a Virtue (When Issuing Database Queries). In *SIGMOD*, pages 931–942, 2014.

[30] B. Ding, S. Das, R. Marcus, et al. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*, pages 1241–1258, 2019.
[31] J. V. D'silva, F. De Moor, and B. Kemme. Aida: Abstraction for advanced in-database analytics. *Proceedings of the VLDB Endowment*, 11(11):1400–1413, 2018.
[32] K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan. DBridge: Translating Imperative Code to SQL. In *SIGMOD*, pages 1663–1666, 2017.
[33] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting equivalent sql from imperative code in database applications. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796, 2016.
[34] K. V. Emani and S. Sudarshan. Cobra: A framework for cost-based rewriting of database applications. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 689–700. IEEE, 2018.
[35] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, 1(1):1107–1123, 2008.
[36] B. Haynes, A. Cheung, and M. Balazinska. PipeGen: Data Pipe Generator for Hybrid Analytics. In *SoCC*, pages 470–483, 2016.
[37] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 1–10, 2017.
[38] V. Josifovski, P. M. Schwarz, L. M. Haas, and E. T. Lin. Garlic: a New Flavor of Federated Query Processing for DB2. In *SIGMOD*, pages 524–532, 2002.
[39] A. Kaitoua, T. Rabl, A. Katsifodimos, et al. Muses: Distributed Data Migration System for Polystores. In *ICDE*, pages 1602–1605, 2019.
[40] W. McKinney. Introducing Apache Arrow Flight: A Framework for Fast Data Transport. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight.
[41] W. McKinney. Apache Arrow and the "10 Things I Hate About Pandas". https://wesmckinney.com/blog/apache-arrow-pandas-internals/, 2017.
[42] A. Moran. Pandas vs Koalas: The Ultimate Showdown! https://www.youtube.com/watch?v=xcGEQUURAuk.
[43] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine Learning in Python. *JMLR*, 12:2825–2830, 2011.
[44] D. Petersohn, W. Ma, D. Lee, S. Macke, D. Xin, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran. Towards scalable dataframe systems. *arXiv preprint arXiv:2001.00888*, 2020.
[45] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, A. Au, C. Curino, Q. Guo, A. Jindal, A. Kalhan, M. Oslake, S. Parchani, V. Ramani, R. Sellappan, S. Sen, S. Shrotri, S. Srinivasan, P. Xia, S. Xu, A. Yang, and Y. Zhu. Seagull: An infrastructure for load prediction and optimized resource allocation. *Proceedings of the VLDB Endowment*, 14(2):154–162, 2021.
[46] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, et al. Data Science Through the Looking Glass and What we Found There, 2019.
[47] K. Ramachandra, K. Park, K. V. Emani, et al. Froid: Optimization of imperative programs in a relational database. *VLDB*, 11(4):432–444, 2017.
[48] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, et al. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63, 2017.
[49] T. Siddiqui, A. Jindal, et al. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*, pages 99–113, 2020.
[50] P. Sinthong and M. J. Carey. Aframe: Extending dataframes for large-scale modern data analysis. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 359–371. IEEE, 2019.
[51] J. Wang, T. Baker, M. Balazinska, et al. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*, 2017.
[52] D. A. Watt. *Programming language design concepts*. John Wiley & Sons, 2004.