Data Engineering for HPC with Python

Vibhatha Abeykoon[*†§], Niranda Perera[*§], Chathura Widanage[*§], Supun Kamburugamuve[†§]
Thejaka Amila Kanewala[‡§], Hasara Maithree[‖], Pulasthi Wickramasinghe[*],
Ahmet Uyar[†], and Geoffrey Fox[*†]
[*]Luddy School of Informatics, Computing and Engineering, IN 47408, USA
{vlabeyko,dnperera,pswickra}@iu.edu
[†]Digital Science Center, Bloomington, IN 47408, USA
{cdwidana, skamburu, auyar, gcf}@iu.edu
[‡]Indiana University Alumni, IN 47408, USA
thejaka.amila@gmail.com
[‖]Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka
hasaramaithree.15@cse.mrt.ac.lk

Abstract—**Data engineering is becoming an increasingly important part of scientific discoveries with the adoption of deep learning and machine learning. Data engineering deals with a variety of data formats, storage, data extraction, transformation, and data movements. One goal of data engineering is to transform data from original data to vector/matrix/tensor formats accepted by deep learning and machine learning applications. There are many structures such as tables, graphs, and trees to represent data in these data engineering phases. Among them, tables are a versatile and commonly used format to load and process data. In this paper, we present a distributed Python API based on table abstraction for representing and processing data. Unlike existing state-of-the-art data engineering tools written purely in Python, our solution adopts high performance compute kernels in C++, with an in-memory table representation with Cython-based Python bindings. In the core system, we use MPI for distributed memory computations with a data-parallel approach for processing large datasets in HPC clusters.**

Index Terms—**Python, MPI, HPC, Data Engineering**

I. Introduction

In the last two decades data has played a major role in the evolution of scientific discoveries. This impacts a wide range of domains such as medicine, security, nature, climate, astronomy, and physics. Massive amounts of data are collected daily by a plethora of scientific applications, especially with the evolution of the Internet of Things (IoT)[1, 2]. The transformation of raw data to a form suitable for analytics is a complex process known as *data engineering* [3]. Data engineering involves many data formats, as well as the transformation, movement, and storage of said data. To support these operations a major contribution has been done by the big data community. Among these contributions, Apache Spark[4], Apache Flink[5], Apache Beam[6], Twister2[7] and Hadoop[8] can be considered as widely used systems for data engineering.

Data scientists who design analytical models often use programming languages such as R, Matlab and Python. The flexibility of these languages offers ideal prototyping required for experiments. Additionally, the emergence of machine learning (ML) and deep learning (DL) frameworks such as Scikit-Learn[9], PyTorch[10], Tensorflow[11] and MxNet[12] has inclined the data analytics domain to rely on Python. Although the core system of most prominent tools from the big data community are written in JVM-based languages (like Java and Scala), Python APIs are an essential feature for bridging the gap between data engineering and data analytic frameworks. PySpark, PyHadoop, PyFlink and PyTwister2 are some notable examples. This interfacing affects the efficiency of the system, since the data has to be serialized/deserialized back-and-forth the Python runtime and JVM runtime. There has been some efforts taken to improve this performance bottleneck by using columnar data formats like Apache Arrow[13]. But we observe that the performance can be further improved by using high performance compute kernels.

Analytical engines increasingly rely on high performance computing clusters for training large deep learning models[14–16]. The data engineering frameworks must be able to leverage the full power of these clusters to feed the data efficiently to such applications. However, current big data systems are not directly compatible with HPC frameworks such as MPI[17, 18], PGAS[19] or Slurm. There are several data engineering frameworks contributed from the Python community. Python support is required to blend with most of the data analytical engines written with Python APIs. Pandas[20], Modin[21], Dask[22] and Mars[23] are a few examples of such well-known solutions written in Python. These frameworks are not enhanced with HPC-oriented compute kernels or with HPC-oriented distributed computing frameworks. This shows that there is an opportunity to design high performance data engineering libraries optimized for HPC resources. Further language bindings like Cython[24] can be used to develop Python API around high performance kernels.

Extending from the best practices of the big data and Python communities, we believe that data engineering can be enhanced to suit the performance demands of HPC environments. Since our effort is to seamlessly bridge data engineering and data analytics, we must understand the expectations of state-of-the-art data analytics engines. Among many data structures

[§]These authors contributed equally.

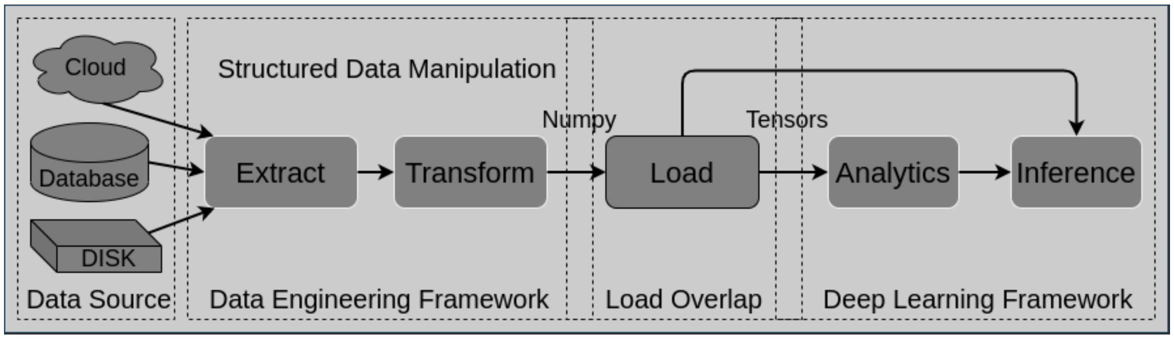

Fig. 1: Data Engineering to Data Analytics

available for representing data in data engineering phases, tables are one of the most widely used abstractions. Relational algebraic operations are a natural fit for processing table data, and SQL interfaces can further enhance usability. Tables can be partitioned into distributed nodes, and the operations can work the partitioned tables. With an efficient data representation and with high performance compute kernels, we believe that data engineering and data analytics can be bridged efficiently and effectively. Figure 1 depicts the connection between the data engineering and the data analytic in a data pipeline.

In this paper we present a high performance Python API with a C++ core to represent data as a table and provide distributed data operations. We compare our work with existing data engineering libraries in Python and big data. The rest of the paper is organized as follows. Section II discusses the data engineering paradigm with high performance computing. In Section III we elaborate on architecture and implementation details, while in Section IV demonstrates how Python-bindings are written on Cylon high performance compute kernels. In Section V demonstrates how our implementation compares with existing data engineering solutions. Section VI reviews the existing literature. We reach our conclusions in Section VII and VIII with an analysis of the current progress and future goals in our effort to improve data engineering with Python in HPC environments.

II. Data Engineering

The data used by ML/DL applications comes in the form of vectors, matrices or tensors. In most cases, the original data are not in formats that are directly convertible to the data structure expected by ML/DL applications. This data can be in many forms inside relational/graph/NoSQL databases or files. The path from raw to readable data is not a one-step process and can be highly complicated depending on the use case. Vector/matrix/tensor data structures are represented by arrays (contiguous memory) with homogeneous data values in the main memory.

In contrast, depending on the original data format, there are many structures available to model them, such as tables, graphs and trees. It is fair to say that table is the most common structure used to represent data. A table contains a set of rows and columns viewed as a grid with cells. The columns can contain data of different types compared to a matrix. Natural operations for tables come from relational algebra. The fundamental relational algebra operations are shown in Table I.

To support large-scale data engineering on top of a table structure, we need to have an in-memory table representation, a partitioning scheme to distribute the table across nodes, and distributed data operations on the partitioned table. As with matrices, tables can be represented using a column or row format. They can also be partitioned row-wise, column-wise or using a hybrid approach with both column and row partitioning. The operations shown in Table I can be implemented on a distributed table partitioned into multiple compute nodes. Both big data and Python systems have used the table abstraction to process data. Another advantage in a table abstraction is that it can be queried using SQL.

A. Big Data Systems

Big data systems adopt the dataflow model, where functional programming is heavily implemented to support a series of data operations. Figure 2 shows an example dataflow operation in PySpark. The big data systems are designed to run on commodity cloud environments. Referring to recent advancements in DL frameworks, eager execution (inverse of lazy execution) has been the most widely adopted programming model as it mimics the Python programming style. Eager execution is not supported by the Python bindings of existing big data systems such as PySpark, PyFlink, PyHadoop and Beam-Python (Apache Beam Python API), which adopt the dataflow model. Integrating dataflow frameworks with high performance computing resources is a challenging task. Frameworks such as Twister2[7] and APIs like TSet[25] were developed to bridge this gap between the big data and HPC systems. Although these efforts enabled big data systems to run on HPC environments, working with multi-language runtimes produces a bottleneck. In such integrations, we observe the following common approaches in big data systems.

- JVM-based back-end handling computation and communication
- Python-based API for defining the dataflow application

Operator	Description
Select	Select operator works on a single table to produce another table by selecting a set of attributes matching a predicate function that works on individual records.
Project	Project operator works on a single table to produce another table by selecting a subset of columns of the original table.
Join	Join operator takes two tables and a set of join columns as input to produce another table. The join columns should be identical in both tables. There are four types of joins with different semantics: inner join, left join, right join and full outer join.
Union	Union operator works on two tables with an equal number of columns and identical types to produce another table. The result will be a combination of the input tables with duplicate records removed.
Intersect	Intersect operator works on two tables with an equal number of columns and identical types to produce another table that holds only the similar rows from the source tables.
Difference	Difference operator works on two tables with an equal number of columns and identical types to produce another table that holds only the dissimilar rows from both tables.

TABLE I: Fundamental relational algebra operations

Fig. 2: PySpark Dataflow Operations

```
df_r = sqlContext.read.format('com.databricks.spark.csv') \
        .options(header='true', inferschema='false') \
        .load(..).toDF(...).repartition(...).cache()
```

Although JVM-Python bridging is done to enable switching between language runtimes, it involves data serialization and deserialization. This creates a performance bottleneck in large-scale applications since (a) it consumes a significant amount of additional CPU cycles for data serialization/deserialization, and (b) this approach also tends to create additional copies of data on both language runtimes' memory spaces, reducing the effective memory available for the application. With this setting, enabling high performance kernels for legacy big data frameworks poses a challenge.

B. *Python for Data Engineering*

PyData and Python community-designed frameworks emerged well ahead of the *big data explosion*. Pandas was initially released in 2008, compared to Apache Spark in 2014. Its user-friendly front-end heavily influenced the front-end APIs of big data frameworks that followed. In the last decade, Pandas became the center for all data engineering tasks associated with ML/DL problems. It is predominantly written in Python and seamlessly integrates with ML/DL frameworks with Python front-ends.

Pandas provided a rich API for data engineering and wraparound tabular format data processing for heterogeneous data, establishing a convenient way to preprocess the data. Yet Pandas suffers from performance bottlenecks for several reasons. Firstly, it only works on a single core. This gives much less room to scale out and process larger datasets in parallel. Secondly, the compute kernels are entirely written in Python without high performance compute kernels. For higher performance and effective prototyping, Apache Spark introduced PySpark which includes a similar functionality. But it is also constrained by the data movement between the Python runtime and JVM.

Since Pandas first premiered, there have been some attempts made by the Python community to improve the performance for large-scale problems. A distributed dataframe [26] was introduced by Dask. This is a full-fledged Python library for parallel computing and it allows for scaling the dataframe to a larger number of machines. The API in Dask is based on dataflow-like lazy execution that uses Python Futures.

Extending from Pandas dataframes, Modin[21] optimizes the query compiling and internal components of Pandas. It also supplies distributed computation by using Dask or Ray as the execution back-end. The internal optimizations have provided better performance over Pandas. Similar to Modin, a framework called Mars [23] was released by Alibaba with a Ray-based back-end. Both these frameworks are developed in Python and do not support high performance compute kernels.

C. *HPC for Data Engineering*

We observe that ML/DL applications commonly consume large predefined and preprocessed datasets. They frequently use data-parallel execution mode in distributed execution. This indicates that data engineering and ML/DL execution occur as two separate functions. We believe that there is an opportunity to bridge this gap using a high-performance low-overhead framework for data engineering, thereby increasing the efficiency and performance of the ML/DL pipeline. We believe introducing HPC for data engineering would enable this transformation. Data engineering in GPU resources using Cudf [27] dataframes is a good example of such an effort, as it delivers processed data immediately to the ML/DL pipeline.

Data engineering would benefit by using distributed memory computations. Furthermore, there are well-defined HPC compute kernels for numerical data analysis. BLAS[28], MKL[29] and Boost[30] are a few examples of such tools being regularly used. Developing systems with these HPC resources provides a better opportunity to improve the performance in data engineering. In addition, HPC compute kernels written in Fortran/C/C++ can be easily integrated with Python, allowing for a seamless integration among HPC resource, data engineering and data analytics.

Since high performance Python is tightly coupled with kernel implementations written in C/C++, the compute kernels related to data engineering can also be written in C/C++. Python bindings must be written carefully without degrading performance and usability. Frameworks like Swig [31], Pybind11 [32] and Cython [24] are mainly used to write efficient Python bindings. In building frameworks or complex applications, the most recommended methods are Python bindings written in either Pybind11 or Cython. This is evident from open source tools such as PyTorch(PyBind11), Numpy(Cython) and Cudf(Cython). Pybind11 favors more on programming with C++ 11 standards, while Cython focuses on both C++ and Python approaches.

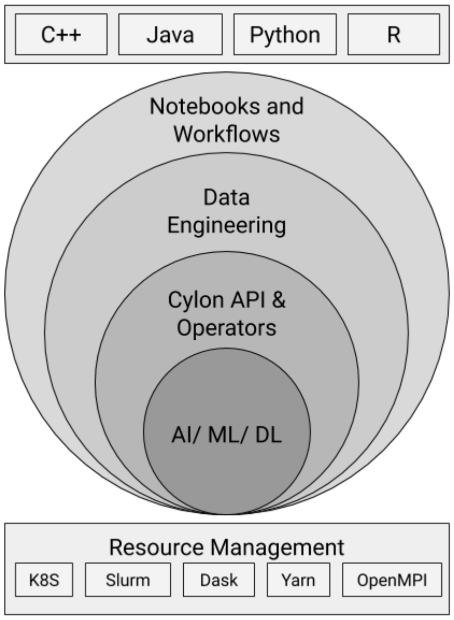

Fig. 3: PyCylon's position in data engineering

The aforementioned tools are mainstream CPU data engineering engines. Rapids AI developed CuDF [27], which offers a GPU dataframe written on top of high performance GPU kernels. CuDF houses a Cython-based Python binding on top of the core GPU dataframe API. Similar to other Python-based frameworks, CuDF uses Dask as a back-end for distributed execution. Limited memory availability in GPUs is often seen as the main drawback for CuDF. Even though GPUs offer much faster kernel computations, CPU-based solutions are still the best fit for big data-related data engineering. Furthermore, we observe that the CPU manufacturers are constantly improving CPU hardware architectures with more threads, faster cache memory, and low power consumption. Therefore a CPU-based solution may provide a less expensive alternative for larger dataset operations.

D. Jupyter Notebooks

Jupyter Notebooks is a powerful means to share and maintain data engineering workflows. When it comes to distributed workloads, Jupyter Notebooks does not support running programs in a cluster with the interactive look and feel generally provided for sequential programs. For Cylon we use an existing set of plugins created by the Python community to enable this on Jupyter Notebooks, with IPyParallel[33] and a distributed extension to IPython[34] kernels. This distributed integration currently supports only an MPI cluster. Using this integration, a data engineering job can be seamlessly integrated with distributed ML/DL jobs. Figure 4 shows the architecture of the current implementation.

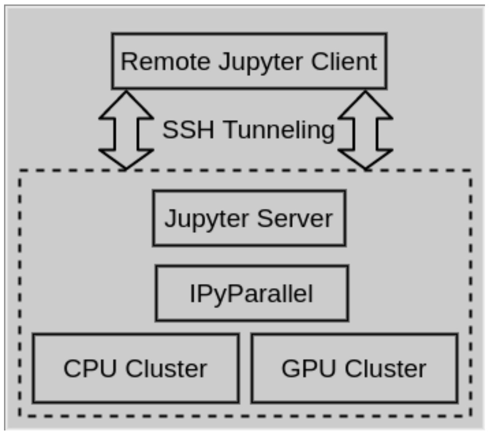

Fig. 4: Cylon Jupyter Cluster

III. Cylon

Having recognized the cost of performance bottlenecks when designing a high performance data engineering library, we created a library/framework Cylon[1]. Cylon a is a distributed memory data table consisting of core relational algebra operators written in C++ high performance kernels. Figure 3 shows PyCylon's position on data engineering.

A. Data Model

Generally data processing systems are divided into two categories: (1) Online Transaction Processing (OLTP) and (2) Online Analytical Processing (OLAP). OLTP usually deals with a large number of atomic operations such as inserts, updates, and deletes, while OLAP is focused on bulk data processing (bulk reads and writes) with a low volume of transactions. Cylon falls into the OLAP category. For the in-memory data representation, Cylon follows the Apache Arrow format. Apache Arrow uses columnar in-memory representation. This allows Cylon to seamlessly integrate with Arrow-based frameworks like Pandas, Apache Spark, Modin, Parquet and Numpy.

B. Distributed Memory Execution

In order to handle massive volumes of data which do not fit into the memory of a single machine, Cylon employs distributed memory execution techniques to slice a large table into small pieces across a cluster of computing nodes. Applying an operation on a table applies that operation concurrently across all the table partitions that reside on different nodes. Hence Cylon embraces the data-parallel approach to parallelizing

[1]https://github.com/cylondata/cylon

the computing tasks. Additionally, the Cylon worker process spawns only a single thread for execution. We assume that the user will start multiple instances of Cylon workers within a single node to match the number of available CPU cores and thus fully utilize the computing capacity of the hardware.

C. Operators

Cylon provides a set of operators for communication-related to distributed computing and relational algebra operators for processing tabular data. The set of operations that we have implemented in Cylon requires an additional communication step when running in a distributed environment. Cylon performs a key-based partition followed by a key-based shuffle through the network to collect similar records into a single process. We have implemented an AllToAll communication utilizing the asynchronous send and receive capabilities of the underlying communication framework to support this cause. The initial implementation of Cylon is written with OpenMPI to handle the communication, which can be easily pluggable with a different framework such as UCX. Cylon can utilize the RDMA capabilities or any other hardware-level network accelerators supported by OpenMPI to improve CPU utilization, throughput, and the latency of the distributed operations. When running on OpenMPI, apart from calling Cylon's built-in functions to manipulate data on the tables, users are free to do any OpenMPI API call at any point of execution to handle additional computing or message passing requirements. Also, Cylon supports a set of core relational algebra operators used in manipulating tabular data. These are supported as local and distributed operators. The core data table operators are shown in Table I.

IV. PyCylon

PyCylon is the Python API written on top of Cylon high performance kernels. Cython is used to link the C++ kernels with Python. The core compute kernels for relational algebra operations are written in C++ using the same memory buffer allocated in C++ when doing computations from the Python API. Since the in-memory data representation is based on Apache Arrow, we have also extended our support to PyArrow tables via our Cython API. This seamlessly integrates the computations from PyArrow extended libraries to PyCylon. We do not expose the communication API to the data scientist; instead the communications are internally handled when doing distributed computations on data tables. Figure 5 shows the API overlay of Cylon.

Even though Cylon is a distributed memory framework, it can also be used as a high performance library to speed up the data processing workloads written in similar Python libraries like Pandas, Modin and Dask. PyCylon includes a DataTable API which is being continuously improved to provide advanced functionality to the data engineering workloads.

Figure 6 shows the PyCylon data interoperability. PyCylon currently supports a few input data formats and input data libraries. CSV support is provided via core Cylon C++ and with PyArrow and Pandas. A PyCylon table can also be created by passing a Pandas dataframe or a PyArrow table. Once the distributed computation is complete, the output can be converted to CSV, Pandas dataframe, etc.

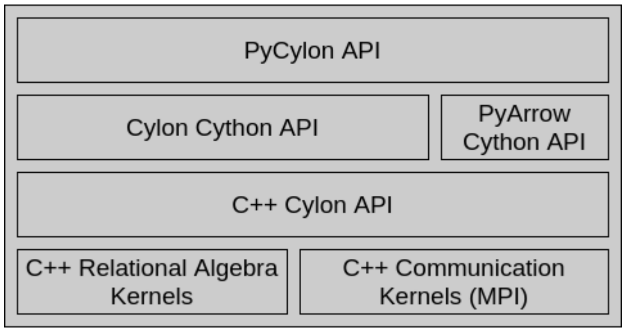
PyCylon API
Cylon Cython API
PyArrow Cython API
C++ Cylon API
C++ Relational Algebra Kernels
C++ Communication Kernels (MPI)

Fig. 5: Cylon API Overlay

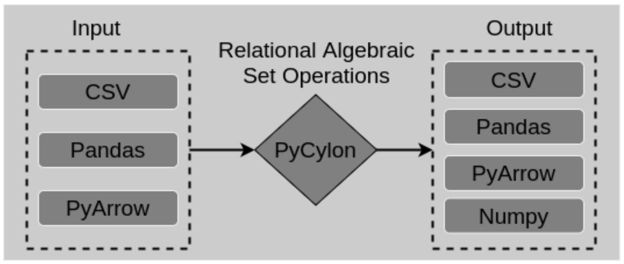

Fig. 6: Cylon Data Interoperability

A. PyCylon Table

The higher level API we have included is the DataTable API. This provides parallelism-unaware API endpoints so that a data scientist can prototype the model without worrying about complex parallel computing concepts. The Cylon context manages the initialization of a distributed environment, so the data scientist need only specify the back-end name used (i.e MPI or UCX). Currently our implementation supports MPI.

Fig. 7: PyCylon Sequential Join

```
from pycylon.data.table import csv_reader
from pycylon.data.table import Table
from pycylon.ctx.context import CylonContext

ctx: CylonContext = CylonContext()

tb1_file = f'{/path/to/file_1}'
tb2_file = f'{/path/to/file_2}'

tb1: Table = csv_reader.read(ctx, tb1_file, ',')
tb2: Table = csv_reader.read(ctx, tb2_file, ',')

configs = {'join_type':'left', 'algorithm':'hash',
               'left_col':0, 'right_col':0}

tb3: Table = tb1.join(ctx, table=tb2,
        join_type=configs['join_type'],
        algorithm=configs['algorithm'],
        left_col=configs['left_col'],
        right_col=configs['right_col'])

tb3.show()
ctx.finalize()
```

Fig. 8: PyCylon Distributed Join

```
from pycylon.data.table import csv_reader
from pycylon.data.table import Table
from pycylon.ctx.context import CylonContext

ctx: CylonContext = CylonContext("mpi")

tb1_file = f'file_1_{ctx.get_rank()}'
tb2_file = f'file_2_{ctx.get_rank()}'

tb1: Table = csv_reader.read(ctx, tb1_file, ',')
tb2: Table = csv_reader.read(ctx, tb2_file, ',')

configs = {'join_type':'left', 'algorithm':'hash',
           'left_col':0, 'right_col':0}

tb3: Table = tb1.distributed_join(ctx, table=tb2,
        join_type=configs['join_type'],
        algorithm=configs['algorithm'],
        left_col=configs['left_col'],
        right_col=configs['right_col'])

tb3.show()
ctx.finalize()
```

Fig. 9: PyCylon Conversions

```
# pycylon_table = Table.from_arrow(pyarrow_table),
# pycylon_table = csv_reader.read(...)
tb3: Table = <pycylon_table>
# converts PyCylon table to Pandas
pdf: pd.DataFrame = tb3.to_pandas()
# converts PyCylon table to Numpy, specify
npy: np.ndarray = tb3.to_numpy(order='C')
```

The difference between a sequential and distributed table API call is the distributed prefix for the method name, 'mpi' argument for the context initialization, and specifying unique file names for each process. In the current API we have kept the MPI look and feel so that users can extend PyCylon programs to existing MPI projects or vice versa. We also support table conversions and initialization as follows. Figures 7 and 8 show the PyCylon code for sequential and distributed join respectively. Figure 9 displays the code-wise data interoperability previously illustrated in Figure 6.

We continue to improve the API endpoints and are adding more functionality for users. We currently offer an experimental Numpy support with PyCylon. The Numpy conversion is the direct link to the tensors in deep learning. In PyTorch Numpy especially, NdArray to tensor conversion takes a negligible amount of time.

V. Experiments

We analyzed the strong scaling performance of PyCylon for the following scenarios and compared the performance against popular Python-based frameworks Dask (Distributed), Modin, and PySpark.

1) Strong scaling performance comparison between the frameworks. Join operation was used here as it is a common use case of columnar-based traversal.
2) Larger test with the best performing frameworks of the above experiment.
3) Overhead comparison between Cylon's Python and Java bindings.

Hardware Setup: The tests were carried out in a cluster with 10 nodes. Each node is equipped with Intel® Xeon® Platinum 8160 processors. A node has a total RAM of 255GB and mounted SSDs were used for data loading. Nodes are connected via Infiniband with 40Gbps bandwidth.

Software Setup: Cylon was built using g++ (GCC) 8.2.0 with OpenMPI 4.0.3 as the distributed runtime. *Mpirun* was mapped by nodes and bound sockets. Infiniband was enabled for MPI. For each experiment, a maximum of 40 cores from each node were used, reaching a maximum parallelism of 400.

For the baseline sequential experiments we used Pandas 0.25.3 version. Apache Spark 2.4.6 (hadoop2.7) prebuilt binary was chosen for this experiment alongside its PySpark release. Apache Hadoop/ HDFS 2.10.0 acted as the distributed file system for Spark, with all data nodes mounted on SSDs. Both Hadoop and Spark clusters shared the same 10-node cluster. To match MPI setup, *SPARK_WORKER_CORES* was set to 16 and *spark.executor.cores* was set to 1. Additionally we also tested PySpark with *spark.sql.execution.arrow.pyspark.enabled* option, which would allow PyArrow underneath PySpark dataframes.

Dask and Dask-Distributed 2.19.0 was set up with Pip installation. Dask Distributed cluster remained in the same nodes as mentioned previously, with *nthreads=1* and varying *nprocs* based on the parallelism. All workers were equally distributed among the nodes.

Modin [21] 0.6.3 was selected alongside Ray 0.7.3 for the experiments. Note that both these frameworks are several versions behind the latest released versions. We were unable to get Modin's latest version (0.8.0) to work with its corresponding Ray 0.8.7 back-end. At the time of conducting these experiments, there are a number of Github issues reported on the same incident which have not yet been resolved [35][36].

Dataset Formats: For strong scaling test cases, CSV files were generated with four columns (one int_64 as index and three doubles). The same files were then uploaded to HDFS for the Spark setup and output counts were checked against each other to verify the accuracy. Timings were recorded only for the corresponding operation (no data loading time considered). For the extended test case, CSV files with two columns were used (one int_64 as index and one double as payload).

1) Scalability: To test the scalability of the Python data engineering frameworks, we varied the parallelism from 1 to 160 while keeping total work at 200 million rows per relation (left/ right). The results are shown in Figure 10.

According to our findings, PyCylon seemed to scale well as the number of processes increased. Around 160 processes, the speedup reaches its plateau. This was expected, as when the parallelism increases, the operation transforms into a communication-bound operation. These results coincide with the Cylon C++ performance in our prior publication [37].

Additionally, the current Cylon compute kernels do not take into account factors such as NUMA boundaries, in-cache performance, etc. As the number of processes inside a node increases, we can expect resource contention for memory bandwidth and L1/L2 caches. Polychroniou et al [38] show that these factors play a vital role in sorting operations, which is the core task in Cylon joins.

Out of Dask-distributed, Modin, and PySpark, only PySpark achieved the strong scaling we expected. This reaffirms the adoption of Spark as a popular data engineering tool. Even with PyArrow execution enabled, the performance seemed to be identical.

Dask-distributed shows some strong scaling conformity, but since it is developed with a Python back-end, this behavior is nothing out of the ordinary.

Recently Modin [21] emerged as a distributed dataframe successor for Pandas. Authors Pertersohn et al showed that it was able to work on datasets exceeding 100GB [39]. We have tested Modin's experimental distributed execution with Ray back-end, but found it performs poorly for strong scaling. Even though Modin's (nearly) complete Pandas dataframe API looks very promising, it suggests that there is a lot of room for improvement.

2) Larger Load Tests: From the above experiment, PySpark and PyCylon scaled as expected. As such we carried out a larger test to determine how these two frameworks perform under bigger loads. We fixed the processes at 200 and varied the total work from 200 million rows per relation to 10 billion. The results are shown in Figure 11.

As the total work increases from 200 million rows to 10 billion, the ratio between PySpark time and PyCylon time increases from 2.1 to 4.5 times. This indicates that Cylon performs better at larger workloads.

3) Switching Between C++, Python, and Java: Figure 12 shows the time taken for Inner-Join (Sort) for 200 million rows while changing the number of workers. It seems clear that the overheads between Cylon and its Cython Python bindings and JNI Java bindings are negligible. This observation seems to confirm that having a C++ back-end greatly reduces overhead in switching between multiple language runtimes.

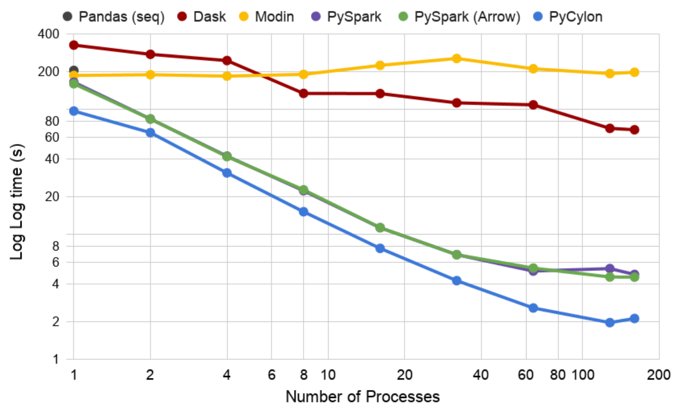

Fig. 10: Strong Scaling with Join Operator. Distributed Join operation was called across 1-160 processes across 10 Nodes (Log-Log plot)

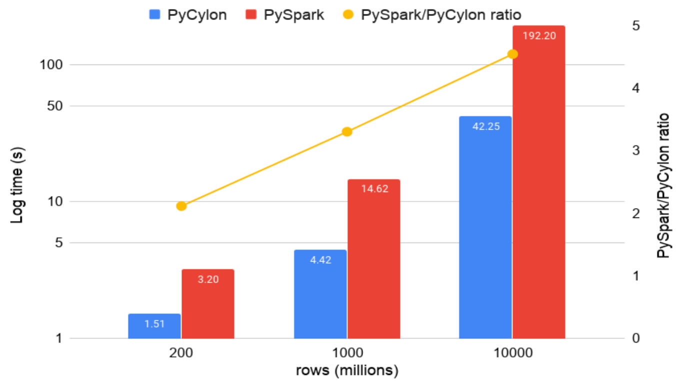

Fig. 11: PyCylon vs. PySpark for Joins with 200 processes at each experiment (Log scale on vertical axis and labels on horizontal axis)

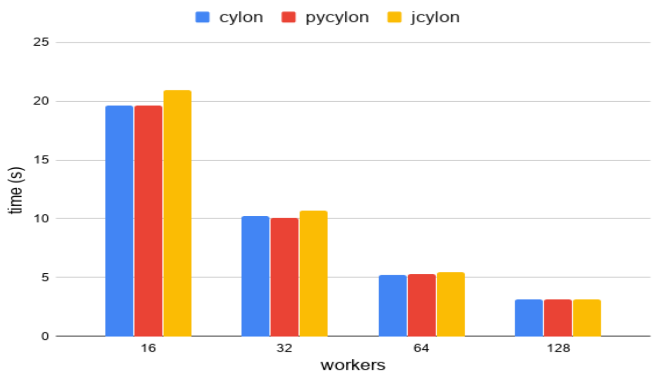

Fig. 12: Cylon Performance Comparison on C++, Python and Java (Linear on vertical axis and labels on horizontal axis)

VI. Related Work

In the data science domain, a widely used data abstraction is the dataframe. Although it has risen to prominence with the advent of deep learning, it was originally developed by S programming language in 1990[40]. For modern day data science, Python programming language affords rapid prototyping capability. Adopting this, Pandas[41] was introduced as a state-of-the-art data representation format. It is a full-fledged Python development based on tabular data. Pandas compute kernels are limited to run only in a single core. With the evolution of big data, an extensive amount of data is being added to data storage every day. For efficient data preprocessing, state-of-the-art big data frameworks like Apache Spark[4, 42] also offer a dataframe abstraction on top of the SQL-based Spark DataSet API. PySpark provides integration between JVM-based data structures and Python-based dataframes. Spark also scales in large-scale big data clusters. One of the main challenges in using PySpark is its function as a framework and not as a library. A data scientist needs to set up a Spark cluster separately to run the data pre-processing workloads. The challenge of data movement from JVM to Python also adds a bottleneck in iterative data preprocessing. Modin [39] provides a Pandas API which can run in large scale with Dask[43] and Ray[44] back-ends. Dask includes a distributed dataframe written on Python. Apart from these CPU dataframes, CuDf, a GPU dataframe, was also introduced by Rapids AI. Unlike existing CPU-based solutions, CuDf[45] utilizes high performance kernels specific for GPUs written in C++. These kernels are exposed to Python via Cython bindings. In addition, frameworks like PyCOMPS[46] along with Dislib[47] provides a better support for distributed computation on array data structures. To optimize an existing Python code, libraries like Numba[48] and Pythran[49] provide high performance capability by optimizing the user code. But in writing frameworks, libraries like Cython[24] and Pybind11[32] are recommended to obtain efficient language bindings. By improving the usability for data analytics, Jupyter[50] Notebooks is also widely popular. Finally, in terms of distributed computations, IPyParallel[33] allows for parallel computation on IPython[34] kernels. IPyParallel is also compatible with Jupyter Notebooks.

VII. Conclusion

The exponential growth of data and deep learning applications means it is vital to provide effective data engineering solutions. After studying the existing data engineering solutions and best practices, we showcased how data engineering can be reinforced to get better performance and scale. One of the major qualitative requirements of data engineering is to write ETL pipeline in Python and retain high performance. Using high performance compute kernels written in C++ and offering Cython-based Python APIs means less overhead across the two runtimes and good scaling in HPC environments. We also show how to use data science best practices and extend distributed data engineering towards Jupyter Notebooks.

From our experiments, we can confirm that Cylon's approach of using Python language bindings with a C++ backend significantly reduces the overheads of switching between language runtimes. As such, data engineering frameworks could benefit from both Python's convenience and HPC backend's performance. Furthermore, we showcased that PyCylon's operations scale better than the popular Python-based data engineering frameworks that are available at the time of writing this paper. We firmly believe that PyCylon performance could be improved further by paying careful attention to memory and network utilization and usage of HPC kernels for computations.

VIII. Future Work

We agree with Petersohn et al's [39] suggestion that confirming to the Pandas dataframe API is an important feature for Python data engineering tools. We are currently developing a dataframe API based on Modin, and thus Cylon would be another distributed back-end for Modin. To expand our compute kernels we are currently focusing on the supporting distributed computing on array data structures. In supporting diverse data formats, we will be integrating HDF5 and Parquet data loading and data processing in a future software release. Additionally, we don't support GPFS or Lustre file systems yet, but we will be focusing on expanding data storage support in the next stages of the development. Furthermore, we are improving Cylon kernels to be NUMA and cache-aware. We believe this would significantly enhance Cylon's performance. We have also recognized recent developments in communication technologies such as RDMA and Infiniband-enabled message passing without the involvement of the CPU. Here, our focus is to integrate the Cylon communication layer with UCX[51]. This permits more flexibility for computation and communication overlap.

Acknowledgment

This work is partially supported by the National Science Foundation (NSF) through awards CIF21 DIBBS 1443054, nanoBIO 1720625, CINES 1835598 and Global Pervasive Computational Epidemiology 1918626. We thank Intel for their use of the Juliet and Victor systems, and extend our gratitude to the FutureSystems team for their support with the infrastructure.

References

[1] A. Castillo O'Sullivan and A. D. Thierer, "Projecting the growth and economic impact of the internet of things," *Available at SSRN 2618794*, 2015.

[2] A. Whitmore, A. Agarwal, and L. Da Xu, "The internet of things—a survey of topics and trends," *Information systems frontiers*, vol. 17, no. 2, pp. 261–274, 2015.

[3] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 3–10.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, ""apache spark: a unified engine for big data processing"," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[5] Apache flink - stateful computations over data streams. [Online]. Available: https://flink.apache.org/

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," 2015.

[7] G. Fox, "Components and rationale of a big data toolkit spanning hpc, grid, edge and cloud computing," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/3147213.3155012

[8] Apache hadoop project. [Online]. Available: https://hadoop.apache.org/

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[12] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[13] Apache arorw project. [Online]. Available: https://arrow.apache.org/

[14] T. Akiba, S. Suzuki, and K. Fukuda, “Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes,” *arXiv preprint arXiv:1711.04325*, 2017.

[15] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, “Deep learning recommendation model for personalization and recommendation systems,” *arXiv preprint arXiv:1906.00091*, 2019.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[17] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, “Open MPI: A High-Performance, Heterogeneous MPI,” in *2006 IEEE International Conference on Cluster Computing*, Sept 2006, pp. 1–9.

[18] “MPI: A Message-Passing Interface Standard Version 3.0,” 2012, Technical Report. [Online]. Available: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[19] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, “Upc++: a pgas extension for c++,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[20] W. McKinney *et al.*, “pandas: a foundational python library for data analysis and statistics,” *Python for High Performance and Scientific Computing*, vol. 14, no. 9, 2011.

[21] Modin dataframes. [Online]. Available: https://modin.readthedocs.io/en/latest/index.html

[22] Dask framework. [Online]. Available: https://dask.org/

[23] Alibaba, “mars-project/mars: Mars is a tensor-based unified framework for large-scale data computation which scales numpy, pandas and scikit-learn.” https://github.com/mars-project/mars, (Accessed on 09/09/2020).

[24] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The best of both worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[25] P. Wickramasinghe, S. Kamburugamuve, K. Govindarajan, V. Abeykoon, C. Widanage, N. Perera, A. Uyar, G. Gunduz, S. Akkas, and G. Fox, “Twister2: Tset high-performance iterative dataflow,” in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE, 2019, pp. 55–60.

[26] A. Lazar, “Dask processing and analytics for large datasets.”

[27] Cudf gpu dataframes. [Online]. Available: https://docs.rapids.ai/api/cudf/stable/

[28] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, “An updated set of basic linear algebra subprograms (blas),” *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[29] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel math kernel library,” in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[30] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[31] D. M. Beazley *et al.*, “Swig: An easy to use tool for integrating scripting languages with c and c++.” in *Tcl/Tk Workshop*, vol. 43, 1996, p. 74.

[32] W. Jakob, J. Rhinelander, and D. Moldovan, “pybind11–seamless operability between c++ 11 and python,” 2017.

[33] “ipython/ipyparallel: Interactive parallel computing in python,” https://github.com/ipython/ipyparallel, (Accessed on 09/10/2020).

[34] F. Pérez and B. E. Granger, “Ipython: a system for interactive scientific computing,” *Computing in science & engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[35] Ray-project issue: Trying to setup ray on custom cluster using docker. [Online]. Available: https://github.com/ray-project/ray/issues/8033

[36] Ray-project issue: Specify network interface to use / runtimeerror... [Online]. Available: https://github.com/ray-project/ray/issues/9456

[37] C. Widanage, N. Perera, V. Abeykoon, S. Kamburugamuve, T. A. Kanewala, H. Maithree, P. Wickramasinghe, A. Uyar, G. Gunduz, and G. Fox, “High performance data engineering everywhere,” *arXiv preprint arXiv:2007.09589*, 2020.

[38] O. Polychroniou and K. A. Ross, “A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 755–766.

[39] 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.

[40] J. M. Chambers and T. J. Hastie, “Statistical models in s. pacific grove, ca: Wadsworth & brooks,” 1992.

[41] W. McKinney *et al.*, “pandas: a foundational python library for data analysis and statistics,” *Python for High Performance and Scientific Computing*, vol. 14, no. 9, 2011.

[42] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. ” O’Reilly Media, Inc.”, 2015.

[43] M. Rocklin, “Dask: Parallel computation with blocked algorithms and task scheduling,” in *Proceedings of the 14th python in science conference*, no. 130-136. Citeseer, 2015.

[44] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, “Ray: A distributed framework for emerging {AI} applications,” in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[45] R. AI, “rapidsai/cudf: cudf - gpu dataframe library,” https://github.com/rapidsai/cudf, (Accessed on 09/10/2020).

[46] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, “Pycompss: Parallel computational workflows in python,” *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[47] J. Á. Cid-Fuentes, S. Solà, P. Álvarez, A. Castro-Ginard, and R. M. Badia, “dislib: Large scale high performance machine learning in python,” in *2019 15th International Conference on eScience (eScience)*. IEEE, 2019, pp. 96–105.

[48] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[49] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, and A. Raynaud, “Pythran: Enabling static optimization of scientific python programs,” *Computational Science & Discovery*, vol. 8, no. 1, p. 014001, 2015.

[50] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, “Jupyter notebooks-a publishing format for reproducible computational workflows.” in *ELPUB*, 2016, pp. 87–90.

[51] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, “Ucx: an open source framework for hpc network apis and beyond,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.