High Performance Dataframes from Parallel Processing Patterns

Niranda Perera[1](✉), Supun Kamburugamuve[2], Chathura Widanage[2], Vibhatha Abeykoon[2], Ahmet Uyar[2], Kaiying Shan[3], Hasara Maithree[4], Damitha Lenadora[5], Thejaka Amila Kanewala[2], and Geoffrey Fox[6]

[1] Luddy School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN 47408, USA
`dnperera@iu.edu`
[2] Indiana University Alumni, Bloomington, IN 47405, USA
[3] University of Virginia, Charlottesville, VA 22904, USA
[4] University of Moratuwa, Bandaranayake Mawatha, Moratuwa 10400, Sri Lanka
[5] University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
[6] Biocomplexity Institute and Initiative, University of Virginia, Charlottesville, VA 22904, USA

Abstract. The data science community today has embraced the concept of *Dataframes* as the de facto standard for data representation and manipulation. Ease of use, massive operator coverage, and popularization of R and Python languages have heavily influenced this transformation. However, most widely used serial Dataframes today (R, `pandas`) experience performance limitations even while working on even moderately large data sets. We believe that there is plenty of room for improvement by investigating the generic distributed patterns of dataframe operators.

In this paper, we propose a framework that lays the foundation for building high performance distributed-memory parallel dataframe systems based on these parallel processing patterns. We also present *Cylon*, as a reference runtime implementation. We demonstrate how this framework has enabled *Cylon* achieving scalable high performance. We also underline the flexibility of the proposed API and the extensibility of the framework on different hardware. To the best of our knowledge, *Cylon* is the first and only distributed-memory parallel dataframe system available today.

Keywords: Dataframes · High performance computing · Data engineering · Relational algebra · MPI · Distributed Memory Parallel

1 Introduction

The Data Science domain has expanded monumentally in both research and industry communities over the past few decades, predominantly owing to the *Big Data* revolution. Artificial Intelligence (AI) and Machine Learning (ML) offer even more complexities to data engineering applications, which are now required to process terabytes of data. Typically, a significant amount of *developer time* is spent on data exploration, preprocessing, and prototyping while developing AI/ML pipelines. Therefore, improving its efficiency directly impacts the overall pipeline performance.

With the wide adoption of R and Python languages, the data science community is increasingly moving away from established SQL-based abstractions. *Dataframes* play a pivotal role in this transformation [14] by providing a functional interface and interactive development environment for exploratory data analytics. `pandas` is undoubtedly the most popular dataframe library available today. Its open source community has grown significantly, and the API has expanded up to 200+ operators. Despite this popularity, both R-dataframe and pandas encounter performance limitations even on moderately large data sets. In our view, dataframes have now exhausted the capabilities of a single computer, which paves way for distributed dataframe systems.

There are several significant engineering challenges related to developing a scalable and high performance distributed dataframe system (Sect. 2.1). In this paper, we analyze dataframe operators to establish a set of generic distributed operator patterns and present an open-source high performance distributed dataframe system framework based on them, *Cylon*. We take inspiration from Mattson et al's *Patterns for Parallel Programming* [13]. Our main focus is to present a mechanism that promotes an existing serial/ local operator into a distributed operator (Sect. 2.2, 3). The proposed framework is aimed at a distributed memory system executing in a Bulk Synchronous Parallel (BSP) [8,20] environment. This combination has been widely employed by the high performance computing (HPC) community for exascale computing applications with admirable success.

2 Dataframe Systems

A *dataframe* is a heterogeneous data structure containing a set of arrays that are individually homogeneous. In contrast, deep learning or machine learning use *tensors* which are homogeneously typed multidimensional arrays. These two data structures are integrated to support end-to-end data engineering workloads. Dataframes were first introduced by the S language in 1990, and their popularity grew exponentially with R and Python languages [14]. These libraries contain a large number of SQL-like statistical, linear algebra and, relational algebra operators and are sequential in execution. With the increasing size of data, there have been some attempts to scale dataframe execution both in the cloud and high performance computing environments such as, Dask [19], Modin [18], and Koalas.

2.1 Engineering Challenges

While there is a compelling need for a distributed dataframe system, there are several engineering challenges.

– **Lack of Specification**: Despite the popularity, there is very little consensus on a specification/standard for dataframes and their operators amongst the systems available today. Rapid expansion in applications and the increasing demand for features may have contributed to this divergence. The current trend is to use `pandas` as the reference API specification [18], and we also follow this approach for the work described in this paper.

- **Massive API**: `pandas` API consists of 240 operators [3,18]. There is also significant redundancy amongst the operators. It would be a mammoth undertaking to parallelize each of these operators individually. Petersohn et al. [18], have taken a more practical approach by identifying a core set of operators (*Dataframe Algebra*) listed in Table 1. In this paper, we have taken a different approach by identifying distributed patterns in dataframe operators, and devise a framework that can best scale them in a distributed memory parallel environment.

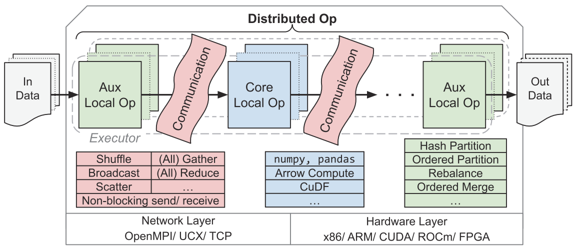


Fig. 1. Distributed Memory Dataframe Abstraction

Table 1. Modin DataFrame Algebra [18]

Selection	Window
Projection	Transpose
Union	Map
Difference	Aggregation*
Join	ToLabels
Unique	FromLabels
GroupBy	Rename
Sort	

*Not categorized in Modin

- **Efficient Parallel Execution**: Distributed data engineering systems generally vary in their execution model. Dask, Modin, and Koalas dataframes are built on top of a fully asynchronous execution environment. Conversely, Bulk-Synchronous-Parallel (BSP) model is used in data parallel deep learning. This mismatch poses a challenge in creating a fully integrated scalable data engineering pipeline. Our framework attempts to bridge this gap by taking an HPC approach to parallelizing Dataframe operators.

2.2 System Considerations

There are multiple aspects that need to be considered when developing a distributed data processing framework [11]. Our distributed dataframe model is designed based on the following considerations.

- **BSP Execution**: The most widely used **execution models** are, 1) *Bulk Synchronous Parallel* [8,20] and 2) *Fully Asynchronous*. The former assumes all the tasks are executing in parallel, and the executors synchronize with each other by exchanging messages at certain points. The sections of code between communication synchronizations execute independently. In the latter, tasks would be executed independently. Input and output messages will be delivered using queues, and often this requires a central scheduler to orchestrate the tasks. Many recent data engineering frameworks (e.g. Apache Spark, Dask, etc.) have adopted fully asynchronous execution. Our framework is based on BSP execution in a distributed memory environment. Gao et al. [9] recently published a similar concept for scaling joins over thousands of GPUs. We intend to show that this approach generalizes to all operators and achieves commendable scalability and high performance.

- **Distributed Memory**: Most often the parallel **memory model** of a system is a choice between, 1) *Shared*: multiple CPU cores in a single machine via threads/ processes (e.g. OpenMP), 2) *Distributed*: every instance of the program is executed on an isolated memory, and data is communicated via message passing (e.g. MPI), and 3) *Hybrid*: combines shared and distributed models. Our framework is developed based on Distributed memory.
- **Columnar Data Format**: Most of dataframe operators access data along columns, and using a columnar format allows operators to be vectorized using SIMD and hardware accelerators (e.g. GPUs). As a result, the patterns described in this paper focus on columnar dataframes.
- **Row-based Partitioning**: Dataframe partitioning is semantically different from traditional matrix/tensor partitioning. Due to the homogeneously typed data storage, when a matrix/ tensor is partitioned, the effective computation reduces for each individual partition. By comparison, dataframe operator patterns (Sect. 3.3) show that not all columns of a dataframe contribute equally to the computation, e.g. `join` is performed on *key* columns, while the rest of the columns move alongside the keys. Both Apache Spark [23] and Dask [19] follow a row-based partitioning scheme, while Modin [18] uses block-based partitioning with dynamic partition ID allocation. Our framework employs BSP execution on a distributed memory parallel environment. We would like to distribute the computation among all available executors to maximize the scalability. We also use row-based partitioning because it allows us to hand over the data partitions with identical schema to each executor.

3 Distributed Memory Dataframe Framework

The lack of a specification presents a challenge in properly defining a *dataframe* data structure. It is not quite a relation in an SQL sense, nor a matrix/multidimensional array. For our distributed memory model, we borrow definitions from Petersohn et al. [18]. Dataframes contain heterogeneously typed data originating from a known set of *domains*, $Dom = \{dom_1, dom_2, ...\}$. For dataframes, these *domains* represent all the data types they support.

Definition 1. *A **Schema** of a Dataframe, S_M is a tuple (D_M, C_M), where D_M is a vector of M domains and C_M is a vector of M corresponding column labels. Column labels usually belong to String/ Object domain.*

Definition 2. *A **Dataframe** is a tuple (S_M, A_{NM}, R_N), where S_M is the Schema with M domains, A_{NM} is a 2-D array of entries where actual data is stored, and R_N is a vector of N row labels belonging to some domain.* Length *of the Dataframe is N, i.e. the number of rows.*

3.1 Distributed Memory Dataframe

"How to develop a high performance scalable dataframe runtime?" is the main problem we aim to address in our framework. We attempt to promote an already available *serial (local) operator* into a distributed-memory parallel execution environment (Fig. 1). For this purpose, we extend the definition of a dataframe for a distributed memory parallel execution environment with row-based partitioning (Fig. 2).

Definition 3. *A* ***Distributed-Memory Dataframe*** *(DMDF) is a* virtual *collection of P* Dataframes *(named* Partition*s) of lengths $\{N_0, ..., N_{P-1}\}$ and a common Schema S_M. Total length of the DMDF is $\Sigma N_i = N$, and the row labels vector is the concatenation of individual row labels, $R_N = \{R_0 R_1 ... R_{P-1}\}$.*

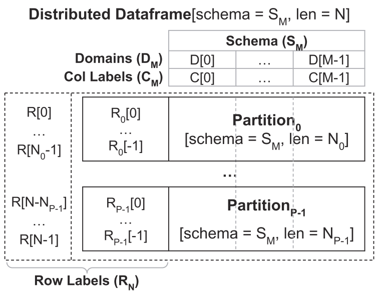


Fig. 2. Distributed Memory Dataframe

Table 2. Communication semantics in Dataframe Operators and the frequency of occurrence

Operation	Data Structure		
	Dataframe	Array	Scalar
Shuffle (AllToAll)	Common	Rare	N/A
Scatter	Common	Rare	N/A
Gather/AllGather	Common	Common	Common
Broadcast	Common	Common	Common
Reduce/AllReduce	N/A	Common	Common

3.2 Building Blocks

As shown in Fig. 1, a distributed operator is comprised of multiple components/ building blocks, such as,

1. **Data Structures**: The distributed memory framework we employ uses three main data structures: dataframes, arrays, and scalars. While most of the operators are defined on dataframes, arrays and scalars are also important because they present different communication semantics.
2. **Serial/Local Operators**: These refer to single-threaded implementations of core operators (Table 1). There could be one or more libraries that provide this functionality (e.g. `numpy`, `pandas`, RAPIDS CuDF, Apache Arrow Compute, etc). Choice of the library depends on the language runtime, the underlying memory format, and the hardware architecture.
3. **Communication Routines**: A BSP execution allows the program to continue independently until the next communication boundary is reached (Sect. 2.2). HPC message passing libraries such as MPI (OpenMPI, MPICH, MSMPI) and UCX provide communication routines for memory buffers (works for homogeneously typed arrays). The most primitive routines are tag-based *async send* and *async receive*. Complex patterns (generally termed

collectives) can be derived on top of these two primitive routines (e.g. MPI-Collectives, UCX-UCC). The columnar data format represents a column by a tuple of buffers and a dataframe is a collection of such columns. Therefore, a communication routine would have to be called on each of these buffers. We identified a set of communication routines required to implement distributed memory dataframe operators. These are listed in Table 2.

4. **Auxiliary Operators**: *Partition* operators are essential for distributed memory applications. Partitioning determines how a local data partition is split into subsets so that they can be sent across the network. This operator is closely tied with *Shuffle* communication routine. The goal of *hash partitioning* is to assign a partition ID to each row of the dataframe so that at the end of the communication routine, all the equal/key-equal rows end up in the same partition. *Ordered Partitioning* is used when the operators (e.g. *Sort*) need to be arranged based on sorted order. Parallel sorting on multiple key-columns further complicates the operation by accessing values along row-dimension (cache-unfriendly). *Rebalance* repartitions data across the executors equally or based on a sequence of rows per partition. On average, an executor would only have to exchange data with its closest neighbors to achieve this. To determine the boundaries, the executors must perform an *AllGather* on their partition lengths. *Merge* is another important auxiliary operator. It is used to build the final ordered dataframe in *Sort* operator to merge individually ordered sub-partitions (∼merge-sort).

3.3 Generic Operator Patterns

Table 3. Generic Dataframe Operator Patterns

Pattern	Operators	Result Semantic	Communication
Embarrassingly parallel	Select, Project, Map, Row-Aggregation	Partitioned	-
Loosely Synchronous			
– Shuffle Compute	Union, Difference, Join, Transpose	Partitioned	Shuffle
– Combine Shuffle Reduce	Unique, GroupBy	Partitioned	Shuffle
– Broadcast Compute	Broadcast-Join*	Partitioned	Bcast
– Globally Reduce	Column-Aggregation	Replicated	AllReduce
– Globally Ordered	Sort	Partitioned	Gather, Bcast, Shuffle, AllReduce
– Halo Exchange	Window	Partitioned	Send-recv
Partitioned I/O	Read/Write	Partitioned	Send-recv, Scatter, Gather

*Specialized join algorithm

Our key observation is that dataframe operators can be categorized into several generic parallel execution patterns. We believe a distributed framework based on these patterns would make the parallelization of the massive API more tractable. These generic patterns (Table 3) have distinct distributed execution semantics, and individually analyzing the semantics allowed us to recognize opportunities for improvement. Rather than optimizing each operator individually, we can focus more on improving bottlenecks of the pattern, and thereby benefiting all operators derived from it.

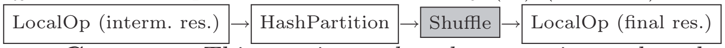
Result Semantic: A local dataframe operator may produce dataframes, arrays, or scalars as results. When we promote a local operator to distributed memory, these result semantics could be nuanced (a global-viewed dataframe). Distributed memory dataframes (and arrays) are partitioned, and therefore a dataframe/array result (e.g. `select, join, etc.`) should also be partitioned. By contrast, scalars cannot be partitioned, so when an operator produces a scalar, it needs to be *replicated* to preserve the overall operator semantic.

Embarrassingly Parallel (EP). EP operators are the most trivial class of operators. They do not require any communication to parallelize the computation. *Select*, *Project*, *Map*, and *Row-Aggregation* fall under this pattern. While *Select* and *Map* apply to rows, *Project* works by selecting a subset of columns. These operations are expected to show linear scaling. Arithmetic operations (e.g. `add`, `mul`, etc.) are good examples of this pattern.

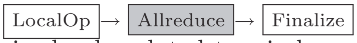
Loosely Synchronous

1. **Shuffle-Compute**: This is a common pattern that can be used for operators that depend on *Equality/Key Equality of rows.* Of the core dataframe operators, `join, union` and `difference` directly fall under this pattern, while `transpose` follows a more nuanced approach.
   Hash partitioning and shuffle communication rearrange data in such a way that equal/key-equal rows are on the same partition. Corresponding local operation can then be called trivially. *Join, Union* and *Difference* operators follow this pattern:
   $$\begin{tikzcd} \boxed{\text{HashPartition}} \arrow[r] & \boxed{\text{Shuffle}} \arrow[r] & \boxed{\text{LocalOp}} \end{tikzcd}$$
   The local operator may access memory randomly, and allowing it to work on in-cache data improves the efficiency of the computation. We could also simply attach a *local hash partition* block at the end of the shuffle to achieve this since hash-partitioning can stream along the columnar data and is fairly inexpensive.
   $$\begin{tikzcd} \boxed{\text{HashPartition}} \arrow[r] & \boxed{\text{Shuffle}} \arrow[r] & \boxed{\text{LocalHashPartition}} \arrow[r] & \boxed{\text{LocalOp}} \end{tikzcd}$$
   A more complex scheme would be to hash-partition data into much smaller sub-partitions from the start. Possible gains on each of these schemes depend heavily on runtime characteristics.
   Transpose is important for dataframe *Pivot* operations. It can be implemented without communication in a block partitioned environment [18]. In a row partitioned setup, a *shuffle* is required at the end of block-wise local transpose to rearrange the blocks.
2. **Combine-Shuffle-Reduce**: An extension of the *Shuffle-Compute* pattern, Combine-Shuffle-Reduce is semantically similar to the famous MapReduce paradigm. The operations that reduce the resultant dataframe length such as *Groupby* and *Unique*, could benefit from this pattern. The initial local operation would reduce data into a set of intermediate results (similar to the combine step in *MapReduce*) e.g. `groupby.std`, creating `sum_`x^2, `sum_`x, and

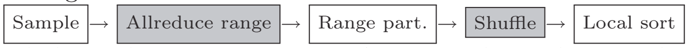
`count_`x, which would then be shuffled. Upon their receipt, a local operation is performed to finalize the results. Perera et al. [17] also discuss a similar approach for dataframe reductions. The effectiveness of *combine-shuffle-reduce* over *shuffle-compute* depends on the *Cardinality* (**C**) (Sect. 3.4).



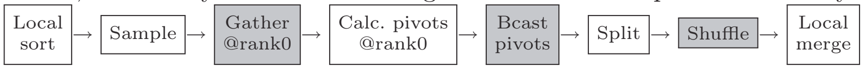
3. **Broadcast-Compute**: This requires a *broadcast* routine rather than *shuffle*. `broadcast_join`, a special algorithm for join, is a good example of this pattern. Broadcasting the smaller length relation to all other partitions and performing a local join is potentially much more efficient than shuffling both relations.
4. **Globally-Reduce**: This is most commonly seen in dataframe *Column Aggregation* operators. It is similar to EP, but requires communication to arrive at the final result. For example, calculating the column-wise `mean` requires a local summation, a global reduction, and a final value calculation. Some utility methods such as *distributed length* and *equality* also follow this pattern. For large data sets, the complexity of this operator is usually governed by the computation rather than the communication.



5. **Halo Exchange**: This is closely related to window operations. `pandas` API supports rolling and expanding windows. For row-partitions, the windows at the boundaries would have to communicate with their neighboring partitions and exchange partially computed results. The amount of data sent/received is based on the window type and individual length of partitions.
6. **Globally Ordered**: Ascending order of rows ($row_i \leq row_j$) holds if all elements in row_i are less than or equal to the corresponding element in row_j. *Ordered partitioning* preserves this order along the partition indices. For a single numerical key-column, the data can be range-partitioned based on a key-data histogram.



For multiple key-columns, we use *sample sort* with regular sampling [12]. It sorts data locally and sends out a sample to a central entity that determines pivot points for data. Based on these points, sorted data will be split and shuffled, and finally all executors merge the received sub-partitions locally.



Partitioned I/O. *Partitioned Input* parallelizes the input data (CSV, JSON, Parquet) by distributing the files to each executor. It may distribute a list of input files to each worker evenly. Alternatively, it receives a custom one-to-many mapping from worker to input file(s) and reads the input files according to the custom assignment. In *Partitioned Output*, each executor writes its own partition dataframe to one file.

3.4 Runtime Aspects

- **Cardinality**: Hash-shuffle in *Shuffle-Compute* pattern roughly takes $O(n) + O(\log P * n)$, where n is average length of a partition. In the *Combine-Shuffle-Reduce* pattern, the initial local operation has the potential to reduce communication order to $n' < n$. This gain depends on the *Cardinality* ($\mathbf{C}$) of the dataframe $\mathbf{C} \in [\frac{1}{N}, 1]$, which is the number of unique rows relative to the length. $\mathbf{C} \sim \frac{1}{N} \implies n' \lll n$, making the combine-shuffle-reduce much more efficient than a shuffle-compute. Consequently, when $\mathbf{C} \sim 1 \implies n' \sim n$ may in fact worsen the combine-shuffle-reduce complexity. In such cases, shuffle-compute pattern is more efficient (5).
- **Data Distribution**: This heavily impacts the partitioning operators. When there are unbalanced partitions, some executors may be underutilized, thereby affecting the overall distributed performance. *Work-stealing* scheduling is a possible solution to this problem. In a BSP environment, pseudo-work-stealing execution can be achieved by storing partition data in a shared object store. Some operations could employ different operator patterns based on the data distribution. (e.g. When one relation is very small, *Join* could use a `broadcast_join`).
- **Logical Plan Optimizations**: An application consists of multiple Dataframe operator. Semantically, they are arranged in a DAG (directed acyclic graph), i.e. *logical plan.* An *optimized logical plan* can be generated based on rules (e.g. predicate push-down) or cost metrics. While these optimizations produce significant gains in real-life applications, this is an orthogonal detail to the individual operator patterns we focus on in this paper.

4 *Cylon*

Cylon is a reference distributed memory parallel dataframe runtime based on Sect. 3. We extended concept to implement a similar GPU Dataframe system, *GCylon*. The source code is openly available in GitHub [6] under Apache License.

4.1 Architecture

- **Arrow Format & Local Operators**: *Cylon* was developed in C++ using Apache Arrow Columnar format, which allows zero-copy data transfer between language runtimes. Arrow C++ Compute library is used for the local operators where applicable. Some operators were developed in-house. Additionally, we use `pandas` and `numpy` in Python for EP operators.
- **Communication**: *Cylon* currently supports MPI (OpenMPI, MPICH, MSMPI), UCX, and Gloo communication frameworks. The communication routines (Table 2) are implemented using a collection of non-blocking routines on internal dataframe buffers. For the user, it would be a blocking routine on dataframes. For example, *Dataframe Gather* is implemented via a series of `NB_Igatherv` calls on each buffer.

- **Auxiliary Operators**: *Cylon* supports all auxiliary operators discussed in Sect. 3. These operators are implemented with utilities developed in-house and from Arrow Compute, and for *GCylon*, we use CuDF utilities where applicable.
- **Distributed Operators** Except for *Window* and *Transpose*, *Cylon* implements the rest of the operators identified in Table 1. As shown in Fig. 1, all of them are implemented as a composition of local, auxiliary and communication operators based on the aforementioned patterns. Currently the `pandas` operator coverage is at a moderate 25%, and we are working on improving the coverage.

4.2 Features

- **Scalability and High Performance**: *Cylon* achieves above-average scalability and higher performance than the commonly used distributed dataframe systems. In Sect. 5, we compare strong scaling of *Cylon*, Modin, and Dask.
- **Flexible Dataframe API**: *Cylon* API clearly distinguishes between local and distributed operators with minimal changes to the `pandas` API semantics. This allows complex data manipulations for advanced users. As an example, a `join` (shuffle) can be easily transformed into a `broadcast_join` just by changing a few lines of code.

```
df1 = read_csv_dist(..., env) # large df
df2 = read_csv(...) if env.rank == 0 else None # read small df at rank 0
df2_b = env.broadcast(df2, root=0) # broadcast
df3 = df1.merge(df2_b, ...) # local join
```

- **Extensibility**: With the proposed model, *Cylon* was able to switch between multiple communication frameworks. Additionally, we extended this model to develop an experimental distributed memory dataframe for GPUs, *GCylon* with minimum development effort.

5 Experiments

Our experiments were carried out in a 15-node Intel® Xeon® Platinum 8160 cluster. Each node has a total RAM of 255 GB, uses SSD for storage and are connected via Infiniband with 40 Gbps bandwidth. A maximum of 40 (of 48) cores were used from each node. The software used: Python v3.8 & Pandas v1.4; *Cylon* (GCC v9.4, OpenMPI v4.1, & Apache Arrow v5.0); Modin v0.12 (Ray v1.9); Dask v2022.1. Uniformly random distributed data was used with two `int64` columns, 10^9 rows (~16 GB), and $\mathbf{C} = 0.9$. This constitutes a worse-case scenario for key-based operators. The scripts to run these experiments are available in Github [7].

The main goal of these operator benchmarks was to show how such generic patterns helped *Cylon* achieve scalable high performance. Dask and Modin operators are compared here only as a baseline. We tried our best to refer to publicly available documentation, user guides and forums while carrying out these tests to get the optimal configurations.

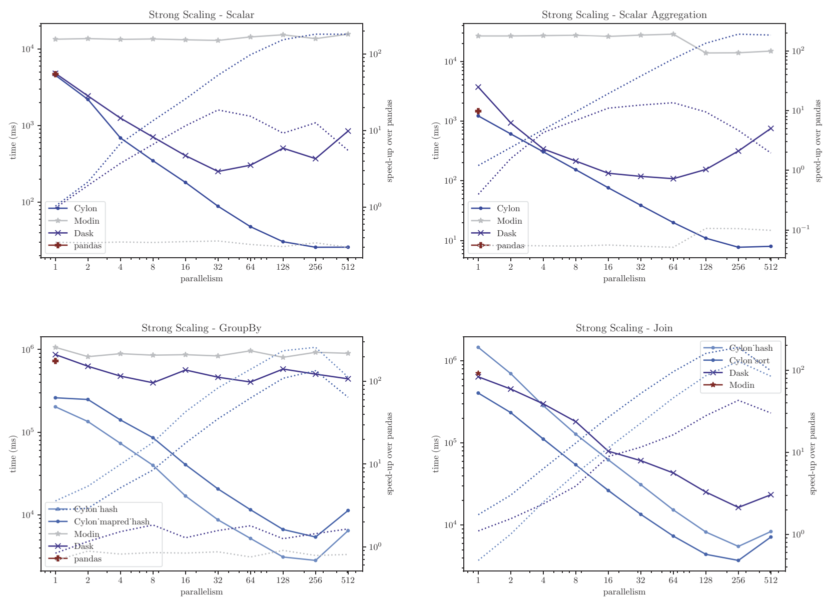


Fig. 3. Strong Scaling (1B rows, Log-Log) with speed-up over `pandas`

- **Scalability**: Figure 3 depicts strong scaling for the patterns. Dotted lines represent the speed-up over `pandas` ($pandas\_time/time$). Compared to Dask, Modin, and `pandas`, *Cylon* shows consistent performance and superior scalability. When the parallelism is increased from 1 to 256, the wall-clock time is reduced, and it takes longer to complete at 512 parallelism. Per executor work is at its lowest in this instance, therefore the communication cost dominates over computation. For EP, a *Barrier* is called at the end and it might carry some communication overhead. *Cylon*'s local operators also perform on par or better than *pandas*, which validates our decision to develop in a C++ backend. Unfortunately, Modin `join` for 1B rows failed, therefore we ran a smaller 100 million row test case (Fig. 4(a)). It only uses `broadcast-join` [15], which explains the lack of scalability. However, we encountered similar problems for the rest of the operators (Fig. 3). Compared to Modin, Dask showed comparable scaling to *Cylon* for `join`s. However, the other operations lacked scalability, especially the scalar operations.
- **Cardinality Impact**: Figure 4(b) illustrates the impact of Cardinality (**C**) on the `groupby` performance. When $\mathbf{C} = 0.9$, hash-groupby (shuffle-compute) consistently outperforms the mapred-groupby (combine-shuffle-reduce), because the local combining step does not reduce the shuffle workload sufficiently. Whereas when $\mathbf{C} = 10^{-5}$, shuffled intermediate result size is significantly lesser, and therefore the latter is much faster. This shows that the same operator might need to implement several patterns and choose an implementation based on runtime characteristics.

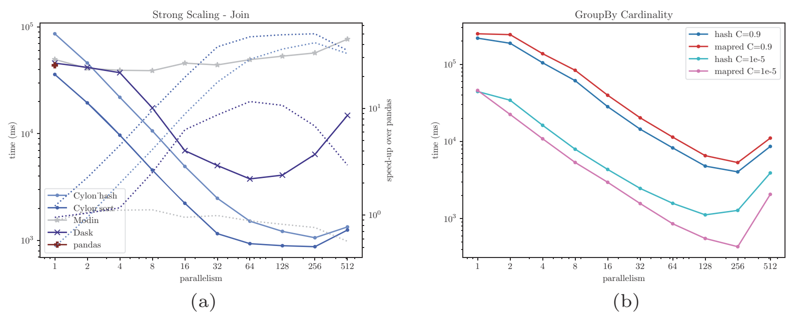


Fig. 4. a: Strong Scaling Joins with Modin (100M rows, Log-Log), b: Cardinality Impact on Combine-Shuffle-Reduce (`groupby`, 1B rows, Log-Log)

6 Related Work

Dask distributed dataframe [19] was the first and foremost distributed dataframe system. It was targeted at providing better performance in personal workstations. RAPIDS CuDF, later extended Dask DDF for GPU dataframes. In large-scale supercomputing environments, HPC-based systems like MPI (Message Passing Interface) [1], PGAS (partitioned global address space) [24], OpenMP, etc. performed better compared to Apache Spark [23] and Dask [2,10,21]). Modin [18], Dask [19], and Koalas (Apache Spark) are some of the emerging distributed dataframe solutions, but the domain shows a lot more room for improvement. HPC-based distributed data engineering systems show promising support for workloads running in supercomputing environments [3,4,17,22], and this is the main motivation for this paper.

7 Limitations and Future Work

Cylon Sort and *Window* operators are still under development. Additionally, larger scale experiments have been planned to provide more finer-grained analysis on communication and computation performance. *Cylon* execution currently requires dedicated resource allocation, which may be a bottleneck in a multi-tenant cloud environment. Furthermore, fault tolerance is another feature that is yet to be added. We believe that both BSP and asynchronous executions are important for complex data engineering pipelines and are currently working on integrating *Cylon* with Parsl [5] and Ray [16]. This would enable the creation of individual workflows that run on BSP, each of which can be scheduled asynchronously, that would optimize resource allocation without hindering the overall performance.

8 Conclusion

We recognize that today's data science community requires scalable solutions to meet their ever-growing data demand. Dataframes are at the heart of such applications, and in this paper we proposed a framework based on a set of generic operator patterns that lays the foundation for building scalable high performance dataframe systems. We discussed how this framework complements the existing literature available. We also presented *Cylon*, a reference runtime developed based on these concepts and showcased the scalability of its operators against leading dataframe solutions available today. We believe that there is far more room for development in domain, and we hope our work contributes to the next generation of distributed dataframe systems.

References

1. MPI: A Message-Passing Interface Standard Version 3.0 (2012). http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Technical Report
2. Abeykoon, V., et al.: Streaming machine learning algorithms with big data systems. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 5661–5666. IEEE (2019)
3. Abeykoon, V., et al.: Hptmt parallel operators for high performance data science & data engineering. arXiv preprint arXiv:2108.06001 (2021)
4. Abeykoon, V., et al.: Data engineering for HPC with python. In: 2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC), pp. 13–21. IEEE (2020)
5. Babuji, Y.N., et al.: Parsl: scalable parallel scripting in python. In: IWSG (2018)
6. CylonData: cylon (2021). https://github.com/cylondata/cylon
7. CylonData: cylon experiments (2021). https://github.com/cylondata/cylon_experiments
8. Fox, G., et al.: Solving problems on concurrent processors, vol. 1: general techniques and regular problems. Comput. Phys. **3**(1), 83–84 (1989)
9. Gao, H., Sakharnykh, N.: Scaling joins to a thousand GPUs. In: 12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@ VLDB (2021)
10. Kamburugamuve, S., Wickramasinghe, P., Ekanayake, S., Fox, G.C.: Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink. Int. J. High Perform. Comput. Appl. **32**(1), 61–73 (2018)
11. Kamburugamuve, S., et al.: Hptmt: operator-based architecture for scalable high-performance data-intensive frameworks. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 228–239. IEEE (2021)
12. Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H.: On the versatility of parallel sorting by regular sampling. Parallel Comput. **19**(10), 1079–1103 (1993)
13. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming (2004)
14. McKinney, W., et al.: pandas: a foundational python library for data analysis and statistics. Python High Perform. Sci. Comput. **14**(9), 1–9 (2011)
15. Modin: modin scalability issues (2021). https://github.com/modin-project/modin/issues

16. Moritz, P., et al.: Ray: a distributed framework for emerging {AI} applications. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), pp. 561–577 (2018)
17. Perera, N., et al.: A fast, scalable, universal approach for distributed data reductions. In: International Workshop on Big Data Reduction, IEEE Big Data (2020)
18. Petersohn, D., et al.: Towards scalable dataframe systems. arXiv preprint arXiv:2001.00888 (2020)
19. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th Python in Science Conference, 130–136. Citeseer (2015)
20. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
21. Wickramasinghe, P., et al.: Twister2: tset high-performance iterative dataflow. In: 2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS), pp. 55–60. IEEE (2019)
22. Widanage, C., et al.: High performance data engineering everywhere. In: 2020 IEEE International Conference on Smart Data Services (SMDS), pp. 122–132. IEEE (2020)
23. Zaharia, M., et al.: apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
24. Zheng, Y., Kamil, A., Driscoll, M.B., Shan, H., Yelick, K.: UPC++: a PGAS extension for c++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1105–1114. IEEE (2014)