A Fast, Scalable, Universal Approach For Distributed Data Reductions

Niranda Perera*§, Vibhatha Abeykoon*†§, Chathura Widanage*§, Supun Kamburugamuve†§
Thejaka Amila Kanewala‡, Pulasthi Wickramasinghe*,
Ahmet Uyar†, Hasara Maithree‖, Damitha Lenadora ‖, 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, damitha.15}@cse.mrt.ac.lk

***Abstract*—In the current era of Big Data, data engineering has transformed into an essential field of study across many branches of science. Advancements in Artificial Intelligence (AI) have broadened the scope of data engineering and opened up new applications in both enterprise and research communities. Reductions/ aggregations are an integral functionality in these applications. They are traditionally aimed at generating meaningful information on large data-sets, and today, they are used for engineering more effective features for complex AI models. Reductions are usually carried out on top of data abstractions such as tables/ arrays and are combined with other operations such as grouping of values. There are frameworks that excel in the said domains individually. But, we believe that there is an essential requirement for a data analytics tool that can universally integrate with existing frameworks, and thereby increase the productivity and efficiency of the entire data analytics pipeline. *Cylon* endeavors to fulfill this void. In this paper, we present *Cylon*'s fast and scalable aggregation operations implemented on top of a distributed in-memory table structure that universally integrates with existing frameworks.**

***Index Terms*—HPC, Data Engineering, Aggregations, Relational Algebra, Big Data, Reductions**

I. Introduction

The massive amount of data generated by a wide variety of applications are used as an input to Artificial Intelligence (AI) and Machine Learning (ML) applications for further processing. However, these AI/ML applications cannot use the raw data as it is; hence the need for a preprocessing step to convert raw data into a consumable form. Aggregations are an essential class of operations used in the preprocessing stage of a data processing application. Sum, maximum, minimum, count, mean, median and standard deviation are some of the most widely used aggregation operations. They are commonly applied after categorising (grouping) data to extract the meaningful input to the AI/ML applications.

The faster we can group and aggregate data, the sooner we can get the final result from the data processing applications. The performance of the aggregation operations can be increased by executing an operation in parallel. Furthermore, with the amount of data generated today, the aggregations and grouping needs distributed execution to increase scalability. While parallel and distributed execution gives us performance and scalability, they inherently add complexity to the implementations of aggregation and grouping operations. Hiding those complexities from AI/ML users and providing them an easy-to-use, familiar abstraction is challenging, but it increases AI/ML user productivity. In this paper, we present a fast, scalable and easy-to-use "group by" and "a set of aggregation operators" implementation.

Aggregators and group by operations are implemented as part of the Cylon [1]: Online Analytical Processing (OLAP) system. Cylon utilizes distributed-memory parallel execution model. In Cylon, data are distributed among multiple compute nodes which process data using Bulk Synchronous Parallel (BSP) [2] model, employing a columnar data structure to represent data. Our implementation inherits these concepts from Cylon and contributes to the performance and scalability of the implementation. We present a framework for aggregation implementation in Cylon and we currently implement four essential distributed memory parallel aggregators using this framework: min, max, count, and sum. The framework consists of phases, each phase maximizing computation to improve the performance. For group by, we present two approaches: Hash-based and Pipeline-based. These are discussed in detail in IV-B. In the same section we illustrate how we adopted the existing group of aggregation techniques in traditional RDBMS databases and existing Big Data frameworks for our implementation. The core of Cylon is developed using C/C++ to achieve maximum performance. Similar to other Cylon operators, the aggregations and group by operations implemented are also exposed with Python bindings so that they can be integrated with AI/ML applications.

For this research, our main objectives are: 1.) provide efficient compute and communication kernels for aggregation and group by operations; 2.) offer efficient language bindings;

§These authors contributed equally.

and 3.) integrate seamlessly with existing data engineering and data science systems. In Section II, we discuss the role of aggregations in data engineering and high performance computing paradigms. Section III showcases how we designed the Cylon system to facilitate high performance data engineering. Section IV details in depth how our aggregation operations are embedded within the Cylon system. We demonstrate a set of experiments conducted on Cylon compared to state-of-the-art data engineering systems in Section V. Finally in Sections VII and VIII, we highlight the conclusions drawn from our work and extensions to our research respectively.

II. Reductions in Data Engineering

Data engineering focuses on practical applications of data collection, analysis, and prediction. It involves data extraction, transformation and loading (ETL) workloads. The *transformation* phase employs relational and linear algebra operators in which data reducing functions play a vital role. This is evident from the presence of a large number of aggregation queries in the decision support benchmarks such as TPC [3]. They also play a vital role in recent AI and ML applications.

Approaches for reductions depend on how data is laid out on the physical memory. When looking at general applications on data engineering, data layouts can be broadly categorize into 1. tables, and 2. arrays (or tensors). In both categories, data can be laid out on row-major or column-major fashion.

Reduction operations are also an integral part of the grouping/ categorizing operations. Applying a reduction operation on grouped data, extracts a summarized the insight on the grouped data. Grouping and reduction may reduce the size of the dataset, but the grouping operation may require substantial computational overheads, such as, moving data, randomly access memory, etc.

A. Reductions in Tables

A Table abstraction (also referred as data-frames) carry heterogeneously typed data defined by a schema.The framework architecture would choose to use row or column-major structure but reductions/ aggregations are usually carried out on a column. Hence, reductions on table with a columnar data structure would be very efficient because it seeks contiguous memory locations and allows trivial SIMD parallelization.

Tables are the backbone of Big Data systems. Apache Hadoop with *map-reduce* [4] [5] marked the first generation of Big Data analytics. Subsequently Apache Spark [6], Apache Flink and Apache Storm were introduced, featuring better scalability and performance. These systems are designed on a JVM-based back-end. They are mostly geared for commodity cloud environments and enterprise clusters. The task-based data-flow execution in these systems promotes usability usability, but whether they achieve the native hardware performance is questionable.

B. Reductions in Arrays/ Tensors

Compared to tables, arrays and tensors entail homogeneously typed multi-dimensional data. Reductions/ aggregations are carried out on a particular indices of these data. Advantages of seeking contiguous memory also holds for array data. Arrays/ tensors may have limitations in representing relational data.

These are the main data structure used for High performance computing applications in domain sciences (i.e. Physics, Chemistry, Biology, etc.). Highly optimized linear algebraic computations are available on arrays through BLAS routines. OpenMP, MPI and PGAS are some of the systems designed to provide distributed computing capability for these structures. These systems are developed on top of C/C++/Fortran to achieve native hardware performance, and they are most often deployed on specialized hardware.

AI/ML has taken center stage in the data engineering research community in recent years. While multidimensional array data (termed "tensors") are used for computations, AI/ML models depend on well-defined preprocessed inputs from large heterogeneously typed datasets. A good example of this is Facebook's DLRM (Deep Learning Recommendation Model) application [7].

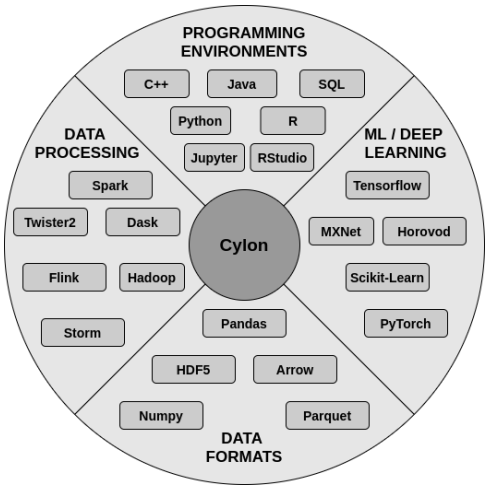

Fig. 1. High performance data engineering everywhere

III. Cylon

In designing a futuristic framework for data engineering, it is vital to pay attention to both performance and usability. An ideal data engineering framework design should be able to benefit from both Big Data and HPC worlds. With AI/ML also becoming a key driver, it is impossible for a single system to provide every feature under one roof. As an example, Apache Spark developed MLlib alongside the data analytics engine. Ultimately it lost popularity to Tensorflow and PyTorch, yet Spark is still being used as a preprocessing engine for AI/ML applications. Hence a better solution would be to create a fast and scalable framework that can universally integrate with other systems.

The goal of Cylon is to fulfill this requirement, and enable *"high performance data engineering everywhere!"* [8]. We will showcase that the architecture of Cylon not only enables

fast and scalable distributed data aggregations, but also provides universal integration bindings to other frameworks as shown in the Figure 1.

A. Data Model

Cylon is a framework that mainly focuses on handling Online Analytical Processing (OLAP) workloads. Unlike Online Transaction Processing (OLTP) systems, OLAP workloads can benefit greatly from data models that have been optimized for homogeneous sequential reads. Hence Cylon has built its core on a columnar data format based on Apache Arrow [9] while providing a Table API abstraction atop a collection of data columns.

Embracing Apache Arrow's columnar format comes with many other advantages, such as inter-portability with existing popular frameworks (Spark, Numpy, Pandas, Parquet, etc.) and optimized memory operations at multiple storage levels ranging from disk (compression) to CPU cache (SIMD operations, efficient cache utilization due to contiguous data layout).

B. Operators

Cylon's table operators can be categorized broadly into two categories based on how they rely on the hardware:

1) Local Operators
2) Distributed Operators

The performance of the local operations are mainly bound by the memory (Disk, RAM and Cache) and CPU, while distributed operations are additionally bound by the network. We currently provide the following relational and aggregation operators.

TABLE I
CYLON RELATIONAL AND AGGREGATION OPERATORS

Operator	Description
select (σ)	Filters out some records based on the value of one or more columns
project (π)	Creates a different view of the table by dropping some of the columns
union ($\cup$)	Applicable on two tables having similar schemas to retain all the records from both tables and remove duplicates
intersect ($\cap$)	Applicable on two tables having similar schemas to retain only the records that are present in both tables
difference ($-$)	Retains all the records of the first table, while removing the matching records present in the second table
join ($\bowtie$)	Combines two tables based on the values of columns. Cylon supports Left, Right, Full, Outer and Inner join modes.
Sort	Sorts the records of the table based on a specified column
Group by	Creates multiple tables (groups) based on a specified criteria
Aggregate	Performs a calculation on a set of values (records) and outputs a single value (record)

Fig. 2. Cylon Distributed Aggregations With C++

```
int main() {
  auto mpi_config = cylon::net::MPIConfig::Make();
  auto ctx = cylon::CylonContext::InitDistributed(mpi_config);
  cylon::Status status;

  std::shared_ptr<Table> input;
  status = cylon::FromCSV(ctx, "/tmp/input.csv", input);
  CHECK_STATUS(status, "Reading csv1 failed!")

  // Sum operation
  std::shared_ptr<cylon::compute::Result> sum;
  status = cylon::compute::Sum(input, 1, output);
  CHECK_STATUS(status, "Sum failed!")

  // Group-by sum operation
  std::shared_ptr<Table> groupby;
  status = cylon::GroupBy(input, 0, {1},
                          {GroupByAggregationOp::SUM},
                          groupby);
  CHECK_STATUS(status, "Sum failed!")

  ctx->Finalize();
  return 0;
}
```

Fig. 3. Cylon Distributed Aggregations With Python

```
from pycylon import Table, CylonContext
from pycylon.net import MPIConfig
from pycylon.io import read_csv

mpi_config = MPIConfig()
ctx = CylonContext(config=mpi_config, distributed=True)

tb = read_csv(ctx, "/tmp/input.csv")

# Sum operation
tb_sum = tb.sum(1)

# Group-by sum operation
tb_gby_sum = tb.groupby(0, 1, [AggregationOp.SUM])

ctx.finalize()
```

C. Distributed Memory Execution

Cylon applications can be run either in local mode or distributed mode, where local mode will be contained to a single node, and distributed mode can be scaled across a cluster of nodes. When running distributed mode, a Cylon table defined in one node can be considered a partition of a dataset that has been distributed across multiple nodes. Thus when applying most of the operators mentioned in Table I, Cylon will be internally performing an all-to-all communication to rearrange the data partitions based on the operator's requirements. The current implementation of Cylon uses MPI at the communication layer, and is capable of using TCP, remote direct memory access (RDMA) or any other software-driven or hardware-accelerated transport layer protocol based on the availability of the resources.

IV. CYLON AGGREGATIONS ARCHITECTURE

Cylon aggregation operations are currently provided by the *compute* API. It is broadly divided into two sections: Aggregate operations and group by followed by aggregate operations. The following subsections will discuss the architecture and design considerations behind these operations. Figures 2 and 3 shows an example code snippet of C++ and Python respectively.

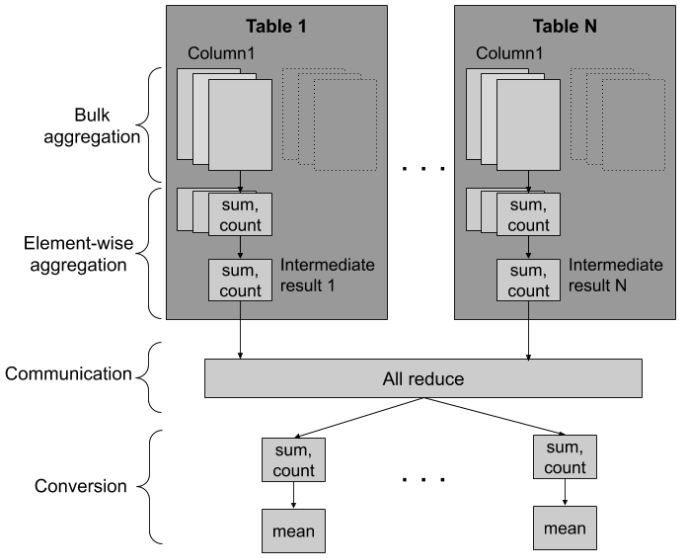

Fig. 4. Operation flow of *mean* aggregation

A. Aggregate Operations

As explained in Section III, Cylon employs Apache Arrow [9] columnar data structure underneath it. A table is partitioned into multiple shards across distributed processes. A column of a table may contain multiple chunks of data. We define an aggregation operation as "a reduction of all values in a (distributed) column". Based on this setup, we identify core components to implement an aggregation operation on columnar data tables.

- Intermediate and final result definition
- Bulk Reduction
- Element-wise Reduction
- Communication of intermediate results
- Final result conversion

This approach has been widely adopted in the OLAP columnar database domain (ex: ClickHouse Database [10]). Additionally, a similar approach has inspired the Apache Arrow *Chunked Array* aggregation operations in their *Table API*. Figure 4 depicts the operation flow of a mean aggregation.

1) Intermediate and final result definition: We posit that distributed aggregation operations can be more complex than a trivial *MPI_Allreduce* operation. Operations would need to track multiple intermediate values before arriving at the global final result. It all depends on the "moment" of the statistic/operation. For example, to compute the *mean* (first moment statistic), the sum and count would have to be tracked individually for every column chunk in each shard of the table. For higher moments, such as *standard deviation*, sum of squares, sum, and count would need to tracked. These intermediate results/state would then need to be converted to the final result. An additional consideration would be the data types of intermediate results. Hence an aggregation operation would have to define these attributes.

2) Bulk Reduction: The role of bulk reduction is to aggregate partial array data into an intermediate result/state. Columnar data being available on contiguous memory locations allows fast and efficient bulk reductions. These operations benefit from efficient CPU cache and registry usages and are further optimized from SIMD (Single-Instruction-Multiple-Data) instructions. Thus it is an obvious choice to enable bulk aggregation capability, as it will be used to reduce partitioned data into individual elements of intermediate results.

Bulk aggregations become sub-optimal if the data elements are randomly distributed in the column. A good example of such a situation is group by aggregations. These will be discussed in more detail in Section IV-B.

Cylon uses Apache Arrow [9] Compute API for bulk aggregations on chunked arrays. Arrow Compute API supports optimized aggregation kernels for

3) Element-wise Reduction: Element-wise aggregations would need to be used when aggregating multiple intermediate results/states. These also form the basic aggregation function. In a scattered data environment/row-based data distribution, element-wise data reduction could be more dominant, as we see in Twister2 Keyed-Reduce operations [11].

4) Communication of intermediate results: Since the intermediate results are distributed across processes, it would require additional communication operations to arrive at the final result. These communication operations could be semantically similar to reduce/all-reduce. This would aggregate intermediate results element-wise. Cylon currently uses OpenMPI [12] as the communication fabric and reduces each element of the intermediate results individually. Another approach would be to use a custom MPI_Op that corresponds to the aforementioned element-wise aggregations.

5) Final result conversion: At the point of returning the aggregated function, the final result would be calculated.

B. Group By Operations

Group by is a widely used operation in traditional Big Data analysis. There are multiple approaches for group by execution, but the main idea is to bring rows together based on a particular key/index column (one or more) and apply an aggregation operation on the rows with the same key. This is semantically equivalent to the *Map-Reduce* [5] Big Data execution paradigm. *Pivoting* is another derivation of group by operation that has become very popular on tabular data.

Cylon currently supports two approaches for group by execution: Hash-based and Pipeline-based. Figure 5 depicts the typical execution flow of group-by operation.

1) Early Aggregation: Early aggregation is a common technique used in traditional RDBMS databases that could speed up group by aggregation operations [13]. In a distributed processing environment, early aggregation helps in reducing the communication overhead and memory required for intermediate results. This early aggregation is semantically equal to the element-wise aggregation step from Section IV-A, but would be operating on multiple rows. Cylon takes advantage

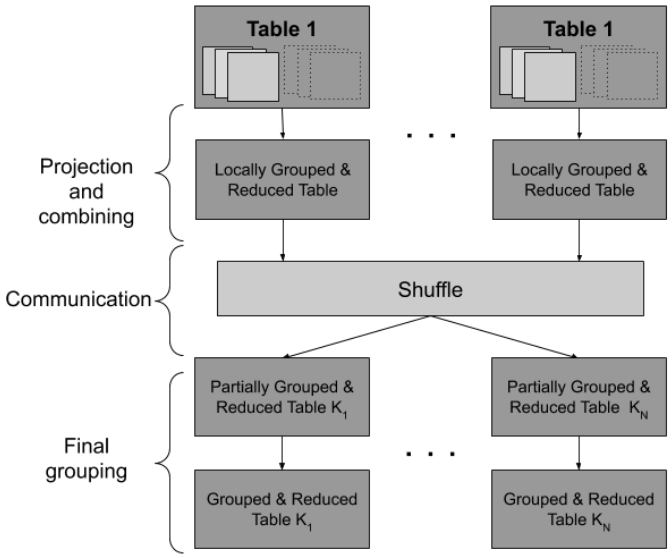

Fig. 5. Operation flow of Cylon group by operation

of this technique for its hash group by implementation, which is explained in the following section.

2) Indices of Groups: A common approach being used by frameworks such as Python Pandas [14], Apache Spark [6], etc., is treating group by as a separate operation which returns the indices corresponding to each group. The aggregation function will subsequently be applied for these groups. Similar to *Join* operations, indices of groups can be generated by either using *hash-based* or *sort-based* approach [15]. This effectively allows multiple aggregation operations on the same groups. Even though this seems like a very intuitive approach, it could have a performance penalty, especially for columnar data. Since values will be aggregated for each group, value columns would be accessed randomly, which could lead to poor cache performance.

Furthermore, the concept of indices of groups becomes rather obscure in a distributed table setup. In this case, groups could well be partitioned across table shards. Tables would have to be shuffled to bring all the indices to the same process.

Sorting can be further extended to *sorting the entire table*. This could benefit columnar data aggregations because the grouped data will be on consecutive memory locations. Then the aggregation kernels could call the *bulk aggregation* on value column slices. This would be an efficient execution, provided that there is a sizeable number of records in each group, but as discussed by Muller et al [16], this leads to higher maintenance costs which could hinder overall performance (Refer Section IV-B5).

3) Group by with early aggregations: In traditional SQL, group by queries are always accompanied by aggregation operations. As mentioned in the previous section, this allows early aggregations even while determining groups.

Cylon currently supports this approach with hash-based grouping. While creating the hash table, the values will be aggregated into intermediate values and later be written to the locally grouped table. Since Cylon works on a distributed environment, these local results would have to be shuffled amongst the processes. Then the resultant table will be grouped again to aggregate the intermediate results of the same groups.

4) Using Local Combiner: The worst-case scenario of the above-mentioned approach can be asymptotically analyzed as follows. Let us take a distributed table partitioned across P processes, with a total N records in G unique groups ($G \leq N$). Then for *each process*,

1) Local group by (Combiner) $= O(N/P)$
2) Shuffle communication $= O(G)$ or $O(N/P)$ without combiner
3) Final local group by $= O(G)$ or $O(N/P)$ without combiner

From this analysis, it is evident that the ratio $N/P : G$ is very important. If these values are relatively similar (i.e. there is less duplication amongst keys), then steps 1 and 3 would take comparable amounts of time, leading to increased total time for operation. In such a scenario, dropping the combiner step could improve the total execution time. Experimental results to support this scenario are provided in Section V-C.

5) Pipeline Group by: Pipeline Group by is a special case of group by operations that can be applied on *sorted* tables. In the distributed setup, it is sufficient to have the table shards sorted locally. The term *Pipeline Group By* seems to have originated from the Vertica [17] columnar database. As explained in Section IV-B2, this could make use of bulk aggregation operations provided that there is sufficient key duplication ($G <<< N/P$). Nevertheless, this approach significantly reduces the memory footprint of the group by operation, and allows further optimizations using multi-threading.

Cylon also supports this approach. The experimental results are provided in Section V-D.

V. Experiments

We analyzed the strong scaling performance of Cylon for the following scenarios and compared the performance against popular Big Data analytics framework Apache Spark [6]. Furthermore we have analyzed the performance of Cylon's aggregation implementation on the following aspects.

1) Strong scaling performance comparison between Cylon vs. Spark on aggregates and group by operations.
2) The effect of group size on the local combiner step
3) Hash group by vs. pipeline group by
4) Overhead comparison between Cylon's Python and Java bindings.

A. 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 16 cores from each node were used, reaching a maximum parallelism of 160.

Apache Spark 2.4.6 (hadoop2.7) pre-built 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.

This notation will be used in the following sections.

- N, Total number of rows in the *distributed* table
- P, Parallelism/number of partitions
- N_P, Rows per partition
- G, Number of unique groups in the dataset ($G \leq N$)

Dataset Formats: For strong scaling test cases, CSV files were generated with two columns (an int_64 as index and a double as value). The same files were then uploaded to HDFS for the Spark setup and output counts were checked against each other to verify the accuracy. Times were recorded only for the corresponding operation (no data loading time considered). All numbers are generated using a uniform random distribution.

B. Scalability

To test the scalability of the aggregation operations, we varied the parallelism from 1 to 160 while keeping total work at 200 million rows per table. The results for both aggregation and group by operations are shown in Figure 6. We have chosen 1 billion records ($N = 1 \times 10^9$) and a $G = 0.99 \times 10^9$, which produces 1.01 average rows per group. This case reflects the worst-case scenario for group by operations (it has no effect on aggregations).

1) Aggregation Scaling: We have tested the performance of calculating the sum of a column. The results are shown in Figure 6(a). As evident from the graphs, Cylon shows almost perfect linear scaling as expected. In comparison, Spark scales much slower. The speed-up of Cylon over Spark increases from *4x* to *27x* as the number of processes increases.

2) Group By Scaling: For group by operations, we have calculated the time spent on group by followed by sum. Group by operation results are shown in Figure 6(b). Cylon seems to demonstrate linear strong scaling. This is expected as the execution becomes increasingly communication-dominant for higher values of P. These results coincide with the Cylon C++ performance in our prior publication [8]. In contrast, Spark seems to plateau earlier than Cylon, leaving a maximum speedup around 2x at 160 processes.

C. Local Combiner vs. Group Size Group By's

To assess the effect of group size on the overall performance, we have varied the average rows per group from $(0.99)^{-1} \approx 1.01$ to $10,000$ while keeping N=200 million. This will effectively change G from 198×10^6 to 20×10^3. We have considered two cases: $P = 64$ and $P = 128$.

As the number of average rows per group increases, the total time reduces. With the local combiner, this effect is much more prominent. The speed-up with the combiner changes from 0.5x to 3-4x. This corresponds to the analysis given in Section IV-B4. When $G = 20 \times 10^3$, $N_P = 1.25 \times 10^6$, therefore having a local combiner significantly reduces the downstream workloads.

D. Hash Group By vs. Pipeline Group By

Figure 8 depicts hash-based group by vs. pipeline group by for a *locally sorted* distributed table with $N = 200$ million records. Tests were carried out varying the average number of rows per group (N/G) for 1, 100, and 10,000. From the graphs, it is evident that pipeline group by performance is adversely affected by the lack of work for bulk aggregations in a group slice. As the number of rows per group increases, pipeline group by becomes more effective than the hash group by.

E. Switching between C++, Python & Java

All of the previous experiments were done on various aspects of the aggregation performance. Since Cylon is engineered to be an integrating approach for all data engineering applications, it is worthwhile evaluating the overheads while switching between language bindings. Table II shows the time ratio for Inner-Join (Sort) for 200 million rows while changing the number of workers. It is clear that the overheads between Cylon and its Cython Python bindings and JNI Java bindings are negligible.

TABLE II
CYLON, PYCYLON VS. JCYLON

World size	PyCylon/Cylon	JCylon/Cylon
16	1.00	1.07
32	0.99	1.04
64	1.01	1.04
128	1.00	1.01

VI. RELATED WORK

Relational databases, Structured Query Language (SQL), and disk-based analytics frameworks such as Apache Hadoop [4] are at the heart of traditional Big Data aggregations. Considered the first generation of Big Data analytics, Hadoop revolutionized the industry by introducing the MapReduce programming model [5]. Apache Spark [6] and Apache Flink [18] subsequently overtook Hadoop by providing faster user-friendly APIs. These also benefited from the boost in hardware advancements that allowed them to perform Big Data processing in-memory.

More recently, Python Pandas Dataframes [14] emerged as the preferred data analytics abstraction amongst the data science and engineering community. Even though Pandas are limited in performance and scalability, they provide an extremely

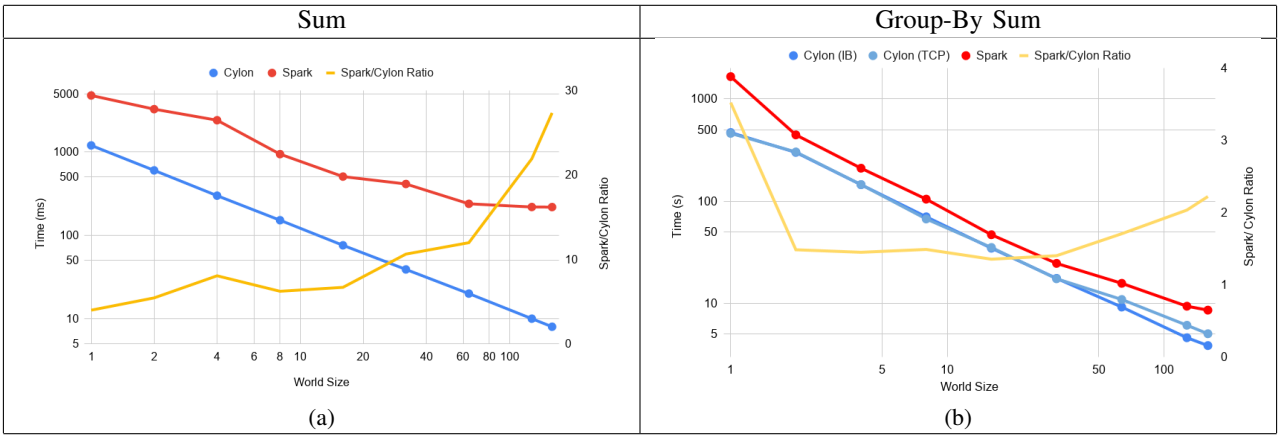

Fig. 6. Cylon vs. Spark strong scaling (Log-Log plots) for $N = 1 \times 10^9$, $G = 0.99 \times 10^9$

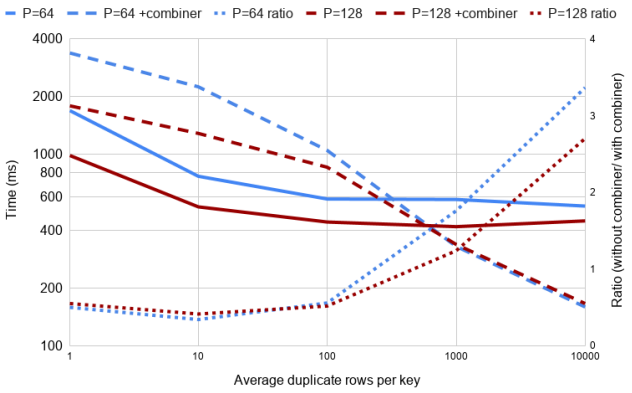

Fig. 7. Local combiner step vs. key distribution for group-by sum (Log-log plot) $N = 2 \times 10^6$

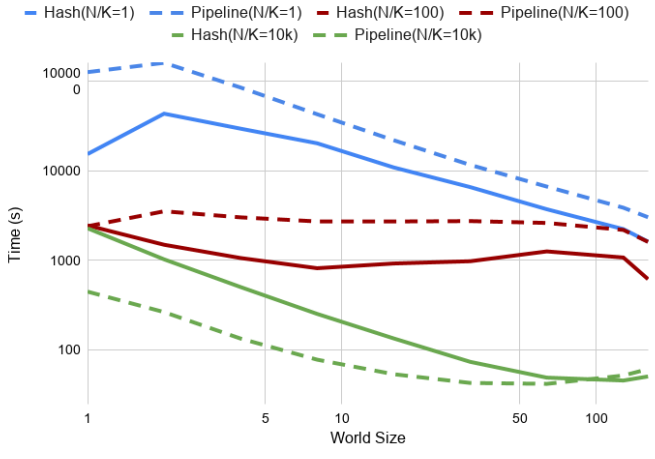

Fig. 8. Hash group-by [solid lines] vs pipeline group-by [dashed lines] followed by sum (Log-log plot) $N = 2 \times 10^6$

convenient programming environment for data processing. Programmer usability became so important that frameworks such as Spark and Flink provided Python wrappers around their data abstractions. But with Java and Python runtimes not being inherently compatible with each other, this hindered their performance. Dask Distributed [19] is a distributed DataFrame abstraction on Python Pandas, and Modin [20] [21] generalized the Pandas API. Later, CuDF [22] emerged as a DataFrame abstraction that could be used for ETL pipelines on top of GPU hardware.

Aggregations have been a widely studied area in the database domain for decades. Smith et al [23] formally defined the fundamentals of aggregation operations. Later Gray et al [24] comprehensively analyzed aggregation functions and also introduced a *Data Cube operator* extending the usual group by aggregation behavior. Larson et al [13] studied impact on grouping by early aggregation. More recently, greater focus has been given to columnar datastores such as Vertica (commercialized version of C-Store) [17], MonetDB, GreenPlum, ClickHouse [10], etc., for their efficient access patterns in OLAP query processing. Abadi et al [25] discussed the general design and implementation of such databases.

VII. Conclusion

Big Data analytics and data engineering have experienced an exponential growth in both research effort and applications. This has expanded the boundaries of traditional stand-alone data analytics solutions (databases, analytics frameworks, etc.) beyond their capabilities. But no framework on its own can fulfill all these requirements. Hence we believe that there is an opportunity for a fast, flexible and integrating framework that could bring all these environments together. Cylon strives to serve this purpose.

Cylon's C++ core allows efficient data analytics implementations, and in this paper we confirmed that data aggregations can also benefit from the same architecture. It allows data aggregations to take advantage of both Big Data and high performance computing domains. Another qualitative requirement of data engineering is to write ETL pipelines in popular

languages like Java and Python without compromising on performance. Offering Cython-based Python APIs for compute kernels means less overhead across the runtimes and good scaling [26].

From our experiments, we can confirm that Cylon's architecture achieves superior performance and scalability than the state-of-the-art Big Data systems, and universally integrates with cross-platform frameworks. It also shows potential for further improvements.

VIII. Future Work

Cylon is a project still in its early stages, and we believe that there is a substantial potential for more performance and usability improvements. The current 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 [27] show that these factors play a vital role in sorting and hashing operations. Furthermore, we believe that the relational algebraic operations such as joins, groupings, etc. can benefit from efficient in-place sorting operations.

Currently, Cylon aggregates, group by operations, and their corresponding communication APIs are rather disjointed, and we believe that these APIs can be combined into a more uniform distributed computing API. We are also evaluating the possibility of using UCX [28] as another communication fabric. While developing Twister2 [11], we have experienced that UCX is an effective communication abstraction for distributed systems. Additionally we believe compute kernels should be able to make use of specialized computational subsystems such as GPUs and CUDA compute routines, FPGAs, etc., and we believe that Cylon's architecture supports such integration.

We agree with Petersohn et al's [21] suggestion that conforming to the Pandas dataframe API is an important feature for data engineering tools. We are currently developing a dataframe API based on Modin, and as such Cylon would be another distributed back-end for Modin. To expand our compute kernels, we are currently focusing on supporting distributed computing on array data structures. In supporting diverse data formats, we will be integrating HDF5, Parquet, GPFS/Lustre data loading and data processing in a future software release.

Acknowledgments

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 the FutureSystems team for their support with the Juliet and Victor infrastructure.

References

[1] Cylon: Data engineering everywhere. [Online]. Available: https://cylondata.org/

[2] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of parallel and distributed computing*, vol. 22, no. 2, pp. 251–267, 1994.

[3] Tpc benchmark specifications. [Online]. Available: http://www.tpc.org/

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

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] 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.

[7] Deep learning recommendation model for personalization and recommendation systems. [Online]. Available: https://github.com/facebookresearch/dlrm

[8] 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.

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

[10] Clickhouse database. [Online]. Available: https://clickhouse.tech/

[11] G. Fox, "Components and rationale of a big data toolkit spanning hpc, grid, edge and cloud computing," in *Proceedings of the 10th 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

[12] 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.

[13] P.-A. Larson, "Grouping and duplicate elimination: Benefits of early aggregation," *Manuscript submitted for publication*, 1997.

[14] 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.

[15] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–169, 1993.

[16] S. Müller and H. Plattner, "An in-depth analysis of data aggregation cost factors in a columnar in-memory database," in *Proceedings of the fifteenth international workshop on Data warehousing and OLAP*, 2012, pp. 65–72.

[17] Vertica: The unified analytics warehouse. [Online]. Available: https://www.vertica.com/

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

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

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

[21] 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.

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

[23] J. M. Smith and D. C. Smith, "Database abstractions: aggregation and generalization," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 2, pp. 105–133, 1977.

[24] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[25] D. Abadi, P. Boncz, S. H. Amiato, S. Idreos, and S. Madden, *The design and implementation of modern column-oriented database systems*. Now Hanover, Mass., 2013.

[26] V. Abeykoon, N. Perera, C. Widanage, S. Kamburugamuve, T. A. Kanewala, H. Maithree, P. Wickramasinghe, A. Uyar, and G. Fox, "Data engineering for hpc with python," *arXiv preprint arXiv:2010.06312*, 2020.

[27] 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.

[28] 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.