High Performance Data Engineering Everywhere

Chathura Widanage[†§], Niranda Perera[*§], Vibhatha Abeykoon[*§], Thejaka Amila Kanewala[‡§],
Supun Kamburugamuve[†§], Hasara Maithree[‖], Pulasthi Wickramasinghe[*],
Ahmet Uyar[†], Gurhan Gunduz[†], and Geoffrey Fox[*†]
[*]Luddy School of Informatics, Computing and Engineering, IN 47408, USA
{dnperera, vlabeyko, pswickra}@iu.edu
[†]Digital Science Center, Bloomington, IN 47408, USA
{cdwidana, skamburu, auyar, ggunduz, 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 We present *TwisterX*, a high performance distributed data processing library that can be seamlessly integrated with with existing data analytics and AI/ML frameworks. It's flexible C++ core, exposes language bindings to Java and Python, which allows TwisterX to be imported as a library or operate as a standalone framework. Data engineering**

***Index Terms*—data-engineering, relational algebra, deep learning, ETL, big data**

I. Introduction

Large-scale data processing/ data engineering has gone through remarkable transformations through out the past few decades. Developing fast and efficient *Extract, Transform and Load* frameworks on commodity cloud hardware has taken a center-stage in handling the *information explosion and Big Data*. Subsequently, we have seen a wide adoption of big data frameworks from Apache Hadoop [1], Twsiter2 [2], Apache Spark [3] to Apache Flink [4] in both enterprises and research communities. Today, Artificial Intelligence (AI) and Machine Learning (ML) have further broaden the scope of data engineering, which imposes faster and more integrable frameworks that can operate on both specialized and commodity hardware.

One question that does not have a very clear answer is whether those existing Big Data frameworks utilize the full potential of the computing power and parallelism to process data. Both Big Data and AI/ML applications spend a good amount of time pre-processing data. Minimizing the pre-processing time clearly increase the throughput of these applications. Productivity is another important aspect of these frameworks. Most of the existing data analytics tools are implemented using a rapid programming languages such as Java, Python or R. Ability to develop applications without diverging into complex data processing code, lets data engineers to be efficient. However, we rarely see these two aspects (high-performance and productivity) meet each other in the existing Big Data frameworks. We have also seen the world increasingly moving towards user-friendly frameworks such as NumPy [5], Python Pandas [6] or Dask [7]. Big Data frameworks have been trying to match this, by providing similar APIs (for example, PySpark, Dask-Distributed). But it was done at the cost of performance, owing to overheads arising from switching between runtimes.

We believe that a data processing framework focused on high performance and productivity would provide a more robust and efficient data engineering pipeline. Hence, in this paper, we introduce *TwisterX*: a high-performance, *MPI* (Message Passing Interface) based distributed-memory data parallel library for processing structured data. TwisterX implements a set of relational operators to process data. While "Core TwisterX" is implemented using system level C/C++, TwisterX provides multiple language interfaces (Python and Java) to seamlessly integrate with existing applications; enabling both data and AI/ML engineers to invoke data processing operators in a familiar programming language.

Large scale extract, transform and load (ETL) operations mostly involved with mapping data to distributed relations and applying queries on them. There are distributed table APIs implemented on top of big data frameworks such as Apache Spark [8] and Apache Flink [9] which are predominantly based on Java programming language. Furthermore SQL interfaces are developed on top of these to enhance the usability. Back then, these features were very useful for the initial use cases of Big Data frameworks, which were mostly based on text data (for an example, analyzing web scrolls of the internet/social media, time series data, etc.). Data engineering have ventured beyond text data, to analyzing multi-dimensional data such as, image, video, text-speech, etc. This makes it harder to integrate current big data frameworks with AI/ML applications written in C++/Python and scientific applications. Also they don't work with high performance computing frameworks such as MPI implementations and Slurm. Furthermore, these large scale data processing operations being available as separate frameworks, also limits their use cases due to maintenance overheads.

We envision data processing as a high performance library function that should be available everywhere including deep

[§]These authors contributed equally.

learning, data processing frameworks, distributed databases and even services. This paper discusses how we achieve this goal using TwisterX. It's flexible C++ core and language interfaces (Java and Python) allows it to be imported as a library to applications or can be run as a standalone framework. It exposes a table abstraction and a set of fundamental relational algebraic operations that are widely used in data processing systems. This couples TwisterX seamlessly with existing AI/ML and data engineering infrastructure. Internally it uses a compact Apache Arrow [10] data format. The initial results show significant performance improvements compared to the state of the art data processing frameworks. In Section II, we elaborate the functionality, architecture and features of TwisterX. Section III discusses the idea of "data processing everywhere" and how TwisterX achieves it. Section IV elaborates how TwisterX performs against popular data processing frameworks.

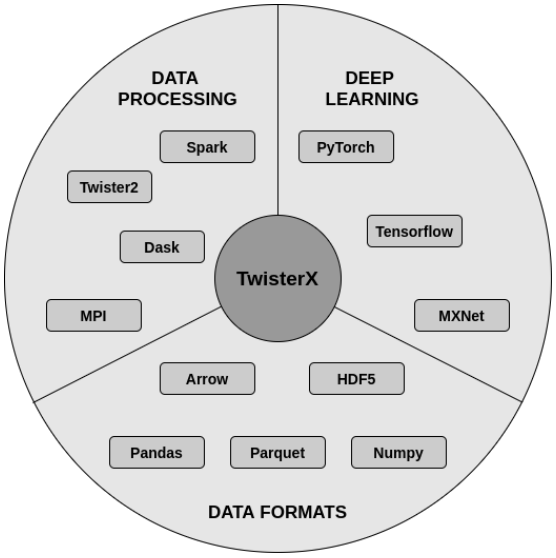

Fig. 1: Data processing everywhere

II. TwisterX

TwisterX is a data engineering toolkit designed to work with ML/AI systems and integrate with data processing systems. This vision is shown in Figure 1 where it supports common data structures and systems. TwisterX can be used either as a library or a framework. Big Data systems like Apache Spark, Apache Flink and Twister [2] can use TwisterX to boost the performance in the ETL pipeline. For ML/AI systems like PyTorch [11], Tensorflow [12] and MXNet [13], TwisterX can be used as a library to enhance ETL performance. Additionally, TwisterX is expanding towards a generic framework for supporting ETL and efficient distributed modelling of AI/ML workloads.

TwisterX currently provides a set of distributed data-parallel operators to extract, transform and load structured relational data. These operators are exposed as APIs in multiple programming languages (e.g., C++, Python, Java) that are commonly used in Machine Learning and Artificial Intelligence platforms, hence enabling tight integration with them. When an operator is invoked in a ML/AI platform, that invocation is delegated to the "Core TwisterX" framework which implements the actual logic to perform the operation in a distributed setting. A high level overview of the TwisterX along with its core framework is depicted in Figure 2.

TwisterX core has a table abstraction to represent structured data. When a table is created in a distributed context, each worker or process will hold a partition of the data that logically belongs to the table. However, each process can work on their own portion of the table, as if they are working on the entire data-set. TwisterX "local operators" are executed on local data and distributed operators depend on these local operators after distributing data as per operator requirements. Distributed operators are implemented based on the Bulk Synchronous Parallel (BSP) approach and the framework synchronizes local operations as needed. In order to take the complexity of distributed programming away from the user, TwisterX internally performs network level operations and abstracts out the distributed nature of the operators. Those network operations function on top of a layer where communication can take place over either TCP, Infiniband or any other protocol supported by the underlying communication layer of a given framework. In the following subsections we discuss each of these layers in detail.

A. Data Model

Traditionally, data processing systems are divided into two main 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 low volume of transactions.

Columnar data formats have a number of advantages for analytical workloads. Predominantly, each column will be contiguously on disk, and will be homogeneously typed. This increases the performance by allowing SIMD operations, utilizing caches better due to the data locality, and also allowing effective compression of data. This was the basis for Apache Parquet columnar data format [14], developed for Hadoop ecosystem. Apache Arrow [10] extends this strategy to an in-memory language-agnostic data structure.

TwisterX workloads predominantly falls under the OLAP category. Hence, the Data Layer of TwisterX is based on Apache Arrow. "Arrow Columnar Format" provides the foundation for TwisterX table API. This also allows seamless integration of other open source frameworks such as Spark and libraries such as Pandas, Parquet and NumPy. Additionally, Arrow also provides zero copy reads, which drastically reduces overheads of switching between language runtimes.

B. Operators

On top of the table abstraction, TwisterX implements five fundamental operations that can be used to create a complete

data processing system. Each operator has two modes of execution: *local* and *distributed*.

Local operators don't use the communication layer and they entirely work on the data available and accessible locally to the process. However, these operators are capable of performing computations on the data in volatile memory as well as on the data in the disk. The performance of these operations are bound usually by IO and CPU capacity. Hence, local operators are optimized to utilize disk-level and memory-level caches efficiently. Some of the local operations we have implemented in the initial version of the TwisterX are Join, HashPartition, Union, Sort, Merge, and Project.

Distributed operators use the network layer at one or multiple points during the operator's life-cycle (beginning, middle, or end). In other words, a distributed operator is one or more local operators coupled with one or more network operators. Network operators can be considered as IO and network bound operators that involve minimum amount of CPU based on the underlying protocol. Initially, we have implemented the "All to All" network operator which is widely required when implementing the distributed counterparts of the local operators. Figure 3 shows how the distributed join operation has been composed by combining two local operators(HashP artition, Local Join) and one network operator(AllToAll).The performance of distributed operators is bound by IO, CPU, and network performances. Current operators implemented in TwisterX are listed in the Table I and this list is expected to grow.

1) Select: Select is an operation that can be applied on a table to filter out a set of rows/records based on the values of all or a subset of columns. When select is called in a distributed environment, TwisterX applies the predicate function provided by the user on the locally available partition of the distributed table. This operation is a pleasingly parallel operation where network communication is not required at all.

2) Project: TwisterX doesn't enforce any limits on the amount of columns or rows that a table can have. As long as hardware resources permit, it can handle a data-set of any complexity. Project can be used to create a simpler view of an existing table by dropping one or more columns. Project can be considered as the counterpart of Select, that works on columns instead of rows. Similar to Select, Project is also a pleasingly parallel operation and TwisterX applies Project on distributed partitions of the table without having to perform any synchronization over the network.

3) Join: Join operation can be used to combine two tables based on the values of a common column. TwisterX implements four types of join operations;

1) Inner Join : Includes records that have matching values in both tables.
2) Left(Outer) Join : Includes all records from the left table and just the matching records from the right table
3) Right(Outer) Join : Includes all records from the right table and just the matching records from the left table.
4) Full Outer Join : Includes all records, but combines the left and right records when there is a match.

TwisterX implements two algorithms to perform above four operations.

1) Sort Join : Sorts both tables based on the join column before performing the join operation. Scans both sorted relations from top to bottom while merging matching records to create the new joined table.
2) Hash Join : Hashes the join column of one relation(preferably the smallest relation), and keep the hashes in a hash map. Scans through the second relation while hashing the join column to find the matching records from first table's hash map.

Although Join is a straightforward operation when performed locally, all matching records need to be in the same process when performed in a distributed context. Hence TwisterX couples Join operation with a shuffling phase to redistribute data based on the join column value. We use a hash based partitioning technique where the records with the same join column hash will be sent to a designated worker/process. At the end of the shuffling phase, local join can be applied on the re-partitioned table.

4) Union: Union operation can be applied on two homogeneous tables(tables having similar schema) to create a single table which includes all the records from both source tables with duplicates removed. Similar to Join, in order to perform Union on a distributed data-set, all similar records needs to be in the same process. Hence, TwisterX performs shuffling as the initial step of the distributed Union operation. Unlike in Join, union considers all the columns(properties) of a record when finding duplicates. For that reason, the hash value of the entire record(row) is considered when performing the hash based partitioning.

5) Intersect: Intersection operation, when applied on two homogeneous tables produces a table with similar records from both tables. TwisterX couples this operation with a shuffling phase similar to the Union operator's distributed implementation.

6) Difference: This operation can be considered as the opposite of Intersect operation. When the Difference operation is applied on two homogeneous table, it produces the final table by adding all the records from both tables, but removing the similar records. Since the similar records need to be identified, this operation has also been coupled with a shuffling phase.

C. Communication Layer

Although the communication layer of TwisterX is initially developed based on OpenMPI [15], that implementation can be easily replaced with a different implementation such as UCX [16]. This will enhance TwisterX's compatibility to run on a wide variety of hardware devices that have different native capabilities including GPUs and different processor architectures such as ARM and PowerPC. Transport layer options will also be widened with different communication layer implementations.

A TwisterX application can utilize multiple communication layer implementations within the same process by defining

Operator	Description
Select	Select operator works on a single table to produce another table by selecting a set of attributes that matches 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 inputs 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 equal number of columns and identical types to produce another table. The produced table will be a combination of the input tables with duplicate records removed.
Intersect	Intersect operator works on two tables with equal number of columns and identical types to produce another table that holds only the similar rows from source tables.
Difference	Difference operator works on two tables with equal number of columns and identical types to produce another table that holds only the dissimilar rows from source tables.

TABLE I: TwisterX operations

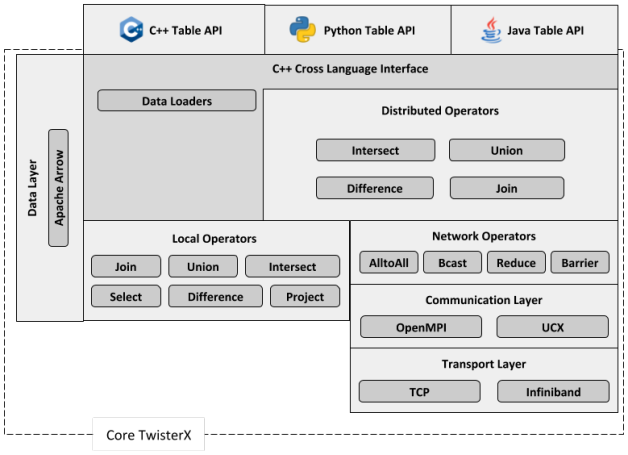

Fig. 2: TwisterX Core Architecture

multiple TwisterX Contexts, which is the API layer abstraction of the underlying stack of communication and transport layers.

Data analytic systems either use event driven model where data producers and consumers are decoupled or synchronous models where producers and consumers work together at the same time. Classic parallel computing with Message Passing Interface (MPI) [17] uses the latter approach where senders synchronize with the receivers for transferring messages. Big data systems use a mix of these two approaches with batch systems using the even driven model and streaming systems using the synchronous model.

The producers and consumers are decoupled in time in an event driven model. The data generated by a producer can be consumed in a later time, suitable for the consumer and not the time imposed by the producer. This allows greater flexibility in designing applications with distributed parts that can work independently.

TwisterX uses synchronized producers and consumers for transferring messages. In contrast, Apache Spark uses an event driven model for communication between its tasks.

D. Transport Layer

At the time of writing, TwisterX has the capability to communicate using any transport layer protocol supported by OpenMPI including TCP and Infiniband. Additionally, all the tuning parameters of the OpenMPI are applicable for TwisterX since the initial implementation is entirely written based on the OpenMPI framework.

Traditionally, data processing systems are divided into two wide categories, (1) Online Transaction Processing (OLTP) and (2) Online Analytical Processin (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 low volume of transactions. As the name suggests, TwisterX falls under the latter category.

Columnar data formats have a number of advantages for analytical workloads. Predominantly, each column will be contiguously on disk, and will be homogeneously typed. This increases the performance of traversing through a column, and also allows effective compression of data. This was the basis for Apache Parquet columnar data format [14], which was developed for Hadoop ecosystem.

III. Data processing Everywhere

One of the main goals of TwisterX to be a versatile library that can provide data processing as a function to provide efficient data engineering across different systems. When working across these systems data representation and conversion is a key factor affecting performance and interoperability. TwisterX internally uses Apache Arrow data structure which is supported by many other frameworks such as Apache Spark, TensorFlow, and PyTorch. Apache Arrow can be converted into other popular data structures such as NumPy, Pandas efficiently. Furthermore our core data structures can work with zero copy across languages. For example, when TwisterX creates a table in CPP, it is available to the Python or Java interface without any data copying.

Further TwisterX has efficient C++ kernels supporting data loading and data processing. These functions can be used as in a distributed setting as a whole or can be used as local functions. Most of the deep learning libraries like Pytorch,

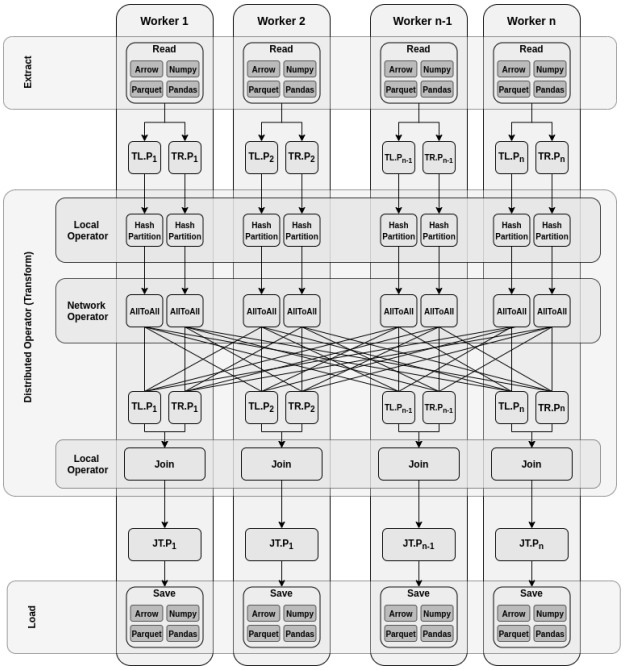

Fig. 3: TwisterX distributed operations

Tensorflow and MXNet are designed on top of such high performance kernels. Similar to TwisterX, their APIs are made available to the user via Python. Similarity in designs leads to lower friction when integrating these systems. With these design points, we envision the following scenarios where TwisterX will work with other systems to create rich applications.

1) Data processing as a library
2) Data processing as a framework
3) Accelerate existing data processing frameworks

A. Data Processing Library

TwisterX can be directly imported to application written in another framework as a library. In a Python program, this integration is a simple module import. TwisterX Python API currently supports Google Colab with an experimental version and supports Jupyter Notebooks with fully-fledged compatibility. In the model prototyping stage, additional set up configurations and installation details becomes a bottleneck to the scientists. Having an easy integration, make this process much easier.

A sample program snippet where TwisterX is imported to Pytorch application is shown in 4. TwisterX can be used as a library to load data efficiently either by using Arrow data loaders or TwisterX data loaders. Then the Table API can be used for the data pre-processing. After the data pre-processing stage, the data can be converted to Pandas and then to Tensors in the DL framework. For data loading TwisterX Python API currently provides support for PyTorch distributed data loaders. This minimizes the effort of integrating TwisterX Python APIs and PyTorch data loading. After this stage, the

Fig. 4: TwisterX with PyTorch

```
import numpy as np
from torch import Tensor as TorchTensor
from pytwisterx.data.table import Table, csv_reader
from pyarrow import Table as PyArrowTable

...
file = "data.csv"
tb = csv_reader.read(file, ",")

# Does data pre-processing

...

tb_arw = Table.to_arrow(tb)
npy = tb_arw.to_pandas().to_numpy()
tensor = torch.from_numpy(npy)

...

# DL Training
```

usual data loader object can be used to extract the tensors and run the training program.

Figure 5 shows the system components of a TwisterX-based deep learning integration. Further to enhance the distributed operations we can add specific support to deep learning settings such as NCCL [18].

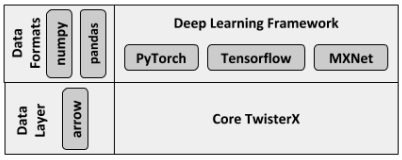

Fig. 5: TwisterX For Deep Learning

B. Data Processing Framework

TwisterX can be used as a separate standalone distributed framework to process the data. As a distributed framework, TwisterX should bring up the processes in different cluster management environments such as Kubernetes, slurm and Yarn. After it brings up the processes it can use the TwisterX core library to process the data. TwisterX has a distributed backend abstraction to plugin various cluster process management systems. At the moment TwisterX works as a standalone framework with the MPI backend. Connections to cluster environments such as Kubernetes will be added through the Twister2 distributed backends. Fig. 6 shows the high level architecture of TwisterX standalone.

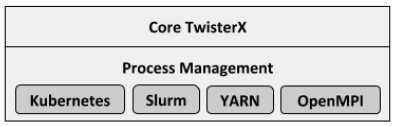

Fig. 6: TwisterX as a Standalone Framework

C. Accelerate existing data processing frameworks

There are two main areas in which TwisterX can accelerate existing data frameworks. In terms of non-deep learning workloads, TwisterX can be used as a plugin to provide efficient communication API which encapsulates the state of the art communication libraries like MPI, NCCL, UCX, etc. In addition to this TwisterX can be used as a high performance kernel for any big data system to perform relation algebra functions through the Table API. For JVM-oriented distributed data processing engines like Apache Spark, Apache Flink or Apache Storm, TwisterX could provide a valuable addition with the Java and Python APIs. We are currently integrating TwisterX kernels with the Twister2:TSet abstraction to provide efficient data pre-processing.

D. Integrate with workflow systems

Any of the above three modes of TwisterX can be used with workflow systems. Here the workflow systems refers to a continuously integrated application. Data pre-processing, training, inference, post-analytics are linked tasks. The tasks in the sequence is always linked the state and input of the previous task. In supporting such scenarios, TwisterX being currently improved in providing seamless integration with the existing workflow engines and dataflow engines. In the modern usecases, data oriented action triggering has become a valuable component. In solving problems associated with both dataflow and workflows, by only using the state of the art workflow engines will not provide the capability of linking up multiple systems in a seamless way. This is one of research components that is being developed with TwisterX. Figure 7 depicts a design of a complex distributed deep learning environment setup on Jupyter environments. This is a component that we are currently designing to bridge all the major tasks associated in a data science workflow. In this workflow, raw data movement, raw data pre-processing, data modelling (training of AI models), model evaluation (inference of AI models), post analysis and end-user consumption are depicted as a unified workflow on top of a Jupyter environment.

IV. Experiments

We have analyzed the distributed performance of TwisterX, against Apache Spark, which is seen as the emerging leader in ETL frameworks. We evaluated the two frameworks based on their scalability of the following workloads.

1) Join - Joins are a common use case of columnar-based traversal
2) Union - Unions without duplicates, is a use case for row-based traversal

Additionally, we have compared TwisterX performance against Dask + Dask-Distributed Python library, which is popularly used in the data science community. We present results of Dask side-by-side with TwisterX and Spark.

Furthermore, we also analyzed the overheads of using TwisterX on C++ and Python environments. This provides an indication of how well, TwisterX can be integrated with Deep Learning and AI frameworks, such as PyTorch, Tensorflow and MXNet.

A. Setup

Tests were carried in a cluster with 10 nodes. Each node has 2 sockets with 24 Intel® Xeon® Platinum 8160 cores each. A node has a total RAM of 255GB and mounted SSDs were used for data loading. Nodes are connected via an Infiniband 40Gbps connection.

TwisterX was built using g++ (GCC) 8.2.0 and OpenMPI 4.0.3 was used as the distributed runtime. *Mpirun* was mapped by nodes, and bound to both nodes and sockets. Additionally, Infiniband was enabled for MPI. For each experiment, 16/48 cores of each node was used, totalling 160 cores.

Apache Spark 2.4.6 (hadoop2.7) prebuilt binary was used. Apache Hadoop/ HDFS 2.10.0 was used as the distributed file system for Spark with all data nodes mounted on SSDs. Both Hadoop and Spark cluster was sharing the same 10-node cluster. To match MPI setup, *SPARK_WORKER_CORES* was set to 16 and *spark.executor.cores* was set to 1.

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

For each test case, CSV files were generated with 4 columns (1 int_64 as index and 3 doubles). 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).

B. Scalability

1) Weak Scaling: For weak scaling, tests were carried out for parallelism 1 to 160 while allocating 2 million rows per core per relation (left/ right). Hence, the first case would have a total work of 2 million while the last case would have 320 million. The results are depicted in Figure 8.

Figure 8 (a) shows to Inner-Join results while (b) shows Union (Distinct) results. Both join and union has a linear complexity over the number of rows. Hence, both can be seen showing a flat line in the log-log plot for TwisterX, indicating that the framework behaves as expected when adding more nodes with similar work. As the number of workers increases, the time for completion grows owing to the increased communication between workers.

Compared to TwisterX joins, union behaves poorly in higher number of nodes. A possible reasoning being the row-based traversal of the table, which could nullify advantages of columnar data format.

Both cases, perform better than Spark in terms of wall clock time for the operation, while both having similar upward trending curves at higher number of workers.

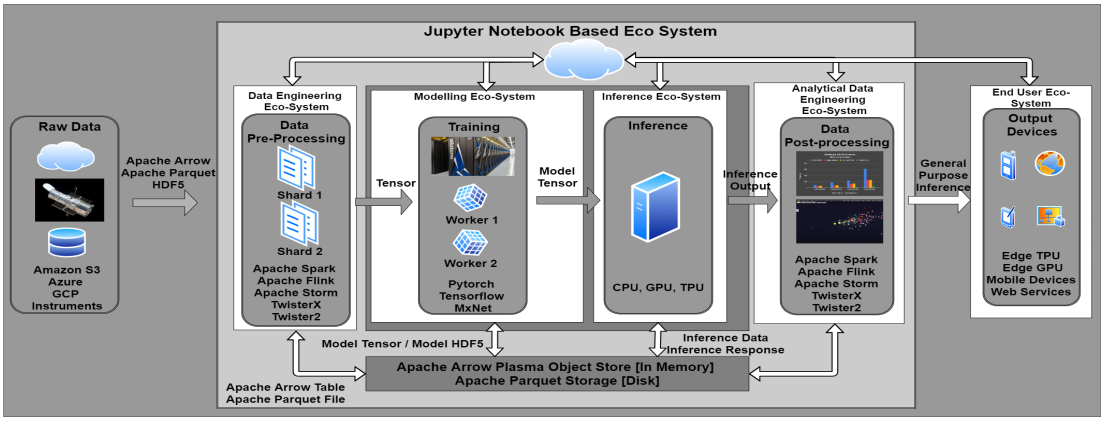

Fig. 7: Jupyter Centric Workflow

Workers	Twx (H)	Twx (S)	Spark	Dask
1	141.5	164.2	586.5	
2	121.2	116.2	332.8	
4	61.6	56.5	207.1	246.7
8	30.7	27.4	119.0	134.6
16	15.0	13.2	62.3	134.2
32	8.1	7.0	39.6	113.1
64	4.5	4.0	22.2	109.0
128	2.8	2.5	18.1	70.6
160	2.5	2.3	18.0	68.9

2) Strong Scaling: For strong scaling, tests were carried out for parallelism 1 to 160 while keeping total work at 200 million rows per relation (left/ right).The results are depicted in Figure 8. It shows individual speed-up over it's own sequential test (ex: TwisterX hash join speed-up over its sequential time)

Figure 8 (c) shows to Inner-Join results while (d) shows Union (Distinct) results. As the number of workers increase, the work per core reduces. Hence both join and union individual speed up is expected to follow a linear trend in the log-log plot. TwisterX plots confirms to this expectation. When the number of workers increase, the speed-up reaches a plateau.

TwisterX shows better strong scaling, reaching a higher individual speedup. For a single worker (serial) Inner-joins TwsiterX Hash, TwsiterX Sort, and Spark took 141s, 164s and 587s respectively. For Union TwsiterX and Spark took, 34s and 75s respectively. Hence, not only TwisterX shows better strong scaling, it gets a better wall-clock speed up because it's serial case wall-clock time is better than Spark.

C. TwsiterX, Spark vs Dask

Figure 9 shows a strong scaling wall-clock time comparison between TwisterX, Spark and Dask. The same strong scaling setup for Inner-Joins was used in this comparison. When comparing Dask, and Spark, TwisterX clearly outperforms them on the wall-clock time. And for this 200 million line join, it scales better than both the other frameworks. It needs to be noted that, Dask failed to complete for the world sizes 1 and 2, even with doubling the resources. It continued to fail even with the factory *LocalCluster* settings, with higher memory.

D. Overheads between C++, Python & Java

Figure 10 shows the time taken for Inner-Join (Sort) for 200 million rows while changing the number of workers. It is evident that the overheads between TwisterX and its Cython Python bindings and JNI Java bindings are negligibly small.

V. Related Work

Databases and Structured Query Language (SQL) was at the heart of ETL pipelines, until the emergence of Apache Hadoop [1]. Termed as the first generation of big data analytics, Hadoop provides a distributed file system (HDFS) and a data processing framework based on the MapReduce programming model [19]. Even though Hadoop scales into terabytes of data over commodity hardware clusters, having a Java-only API made it less attractive for traditional ETL applications.

Apache Spark [3] took over Hadoop with the introduction of Resilient Distributed Dataset (RDD) [20] abstraction. Built on the Scala, Spark provided a fast distributed in-memory data processing framework, with built-in SQL capability, streaming analytics and Python API. In later versions, Spark developers enhanced the performance by optimizing queries, and added more user-friendly, table-like API's that mimic Python Pandas *DataFrames*. Having to communicate between Python and Java run-times, greatly affects PySpark performance. Hence they have introduced an extension that uses Apache Arrow in-memory columnar data format in the Python environment. Unfortunately this enhancement is not available for the Java/ Scala APIs. Spark initially proposed a machine learning library, MLlib, but with the wide adoption of specialized AI/ML and deep learning frameworks such as, Tensorflow, PyTorch, etc, it is now used purely as an ETL framework for AI/ML data manipulation. Apache Flink [9] is another popular data analytics tool that emerged after Apache Spark, which was also developed in Java. Flink is geared for real-time streaming analytics and native iterative processing. It also provides a table-like API on both Python and Java environments. Like Spark, Flink also uses Py4J to communicate between the runtimes, and hence make a trade-off of performance for usability.

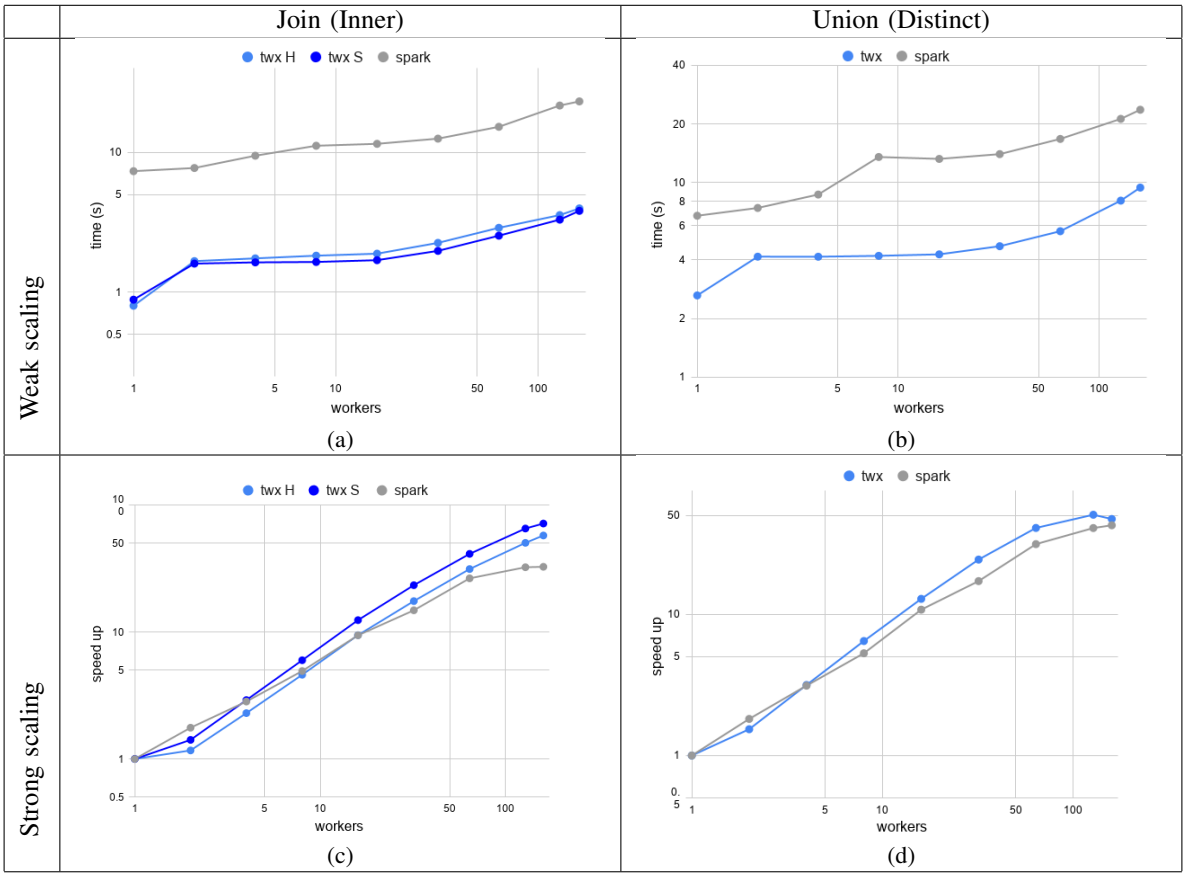

Fig. 8: TwisterX vs Spark scaling comparison (Log-Log plots)

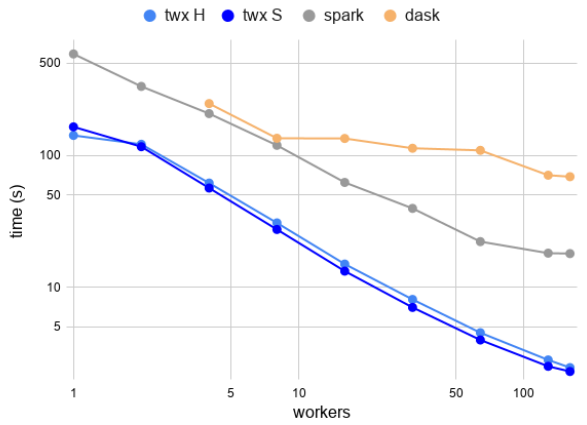

Fig. 9: TwisterX, Spark vs Dask strong scaling (Inner-Join)

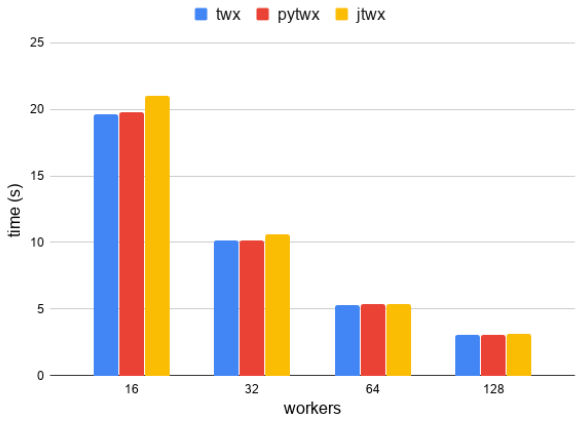

Fig. 10: TwisterX, PyTwisterx vs JTwisterX

Pandas [6] is the most popular library used in the data science community, that provides a rich toolkit for ETL operations. Its built on top of Numpy library. *DataFrame* is the main table-like data structure of Pandas. The concept of *DataFrames* has inspired many other frameworks as mentioned previously. Main drawback of Pandas is, not having a distributed data abstraction, nor distributed operations. Since 2016, Dask [7], a parallel computing library for Python came up with a distributed DataFrame abstraction like Python Pandas. Using distributed communication primitives, Dask provide standard Pandas operations like group-by, join, and time series computations. Since both, Pandas and Dask DataFrames have been developed in Python, they suffer the inherent inefficiencies of the language environment. Hence,

application of DataFrames for large-scale ETL pipelines, is still not very popular, even though they provide a very user-friendly programming interface.

GPU hardware are becoming very popular for deep learning workloads. GPUs have many of SIMD threads that are connected to high-bandwidth memory. CuDF [21] integrates GPU computing capabilities, to Pandas like DataFrame abstraction that can be used for ETL pipelines. It also, integrates with Dask to provide distributed computations on GPU DataFrames. It is also built on top of the Apache Arrow columnar data format. Modin [22] is a similar GPU based dataframe library, also inspired by Pandas API. It uses Ray or Dask for distributed execution. With increasing GPU memory and hardware availability, it is inevitable that they would also be used for ETL workloads, which would compliment deep learning workloads in turn.

Apache Spark is one of the pioneer dataflow systems to support deep learning workloads for a long time. BigDL [23] framework developed on top of Apache Spark ecosystem is one of the prominent research done in integrating deep learning with Apache Spark. Additionally, Horovod [24] another distributed deep learning framework which initially supported TensorFlow, is also using Apache Spark as a backend for data pre-processing. In supporting the existing deep learning frameworks, there has been a recent contribution from another platform called Ray [25]. Ray is an actor-based distributed system, developed on top of a Python-backend. The major advantage of Ray is the easiness to use as a regular Python library to bridge the gap between classical big data systems.

VI. Conclusions & Future work

In this paper we described TwisterX data processing library. We described its data model, how it can interpolate between different systems efficiently, its core operations and performance. We saw significant improvements in efficiency compared to existing systems proving our assumption that there is much room to improve. Furthermore we discussed how TwisterX fits into the overall data engineering of an application.

We are working onto extend the TwisterX operations to use external storage such as disks for larger tables that do not fit into the memory. When such operations are available BSP style APIs are no longer efficient and needs a dataflow API. At the initial stage we have implemented the fundamental relational algebraic operations. We are planning to add more operations to enhance the usability of the system.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and the Indiana University Precision Health initiative. We thank Intel for their support of the Juliet and Victor systems, and extend our gratitude to the FutureSystems team for their support with the infrastructure.

References

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

[2] 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

[3] 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.

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

[5] Numpy - the fundamental package for scientific computing with python. [Online]. Available: https://numpy.org/

[6] 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.

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

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] Apache parquet project. [Online]. Available: https://parquet.apache.org/

[15] 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.

[16] 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.

[17] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: advanced features of the message passing interface*. MIT press, 1999.

[18] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, ""efficient large message broadcast using nccl and cuda-aware mpi for deep learning"," in *Proceedings of the 23rd European MPI Users' Group Meeting*, 2016, pp. 15–22.

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

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

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

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

[23] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, ""bigdl: A distributed deep learning framework for big data"," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[24] A. Sergeev and M. Del Balso, ""horovod: fast and easy distributed deep learning in tensorflow"," *arXiv preprint arXiv:1802.05799*, 2018.

[25] 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.