AFrame: Extending DataFrames for Large-Scale Modern Data Analysis

Phanwadee Sinthong
Dept. of Computer Science
University of California, Irvine
psinthon@uci.edu

Michael J. Carey
Dept. of Computer Science
University of California, Irvine
mjcarey@ics.uci.edu

***Abstract*—Analyzing the increasingly large volumes of data that are available today, possibly including the application of custom machine learning models, requires the utilization of distributed frameworks. This can result in serious productivity issues for "normal" data scientists. This paper introduces AFrame, a new scalable data analysis package powered by a Big Data management system that extends the data scientists' familiar DataFrame operations to efficiently operate on managed data at scale. AFrame is implemented as a layer on top of Apache AsterixDB, transparently scaling out the execution of DataFrame operations and machine learning model invocation through a parallel, shared-nothing big data management system. AFrame incrementally constructs SQL++ queries and leverages AsterixDB's semistructured data management facilities, user-defined function support, and live data ingestion support. In order to evaluate the proposed approach, this paper also introduces an extensible micro-benchmark for use in evaluating DataFrame performance in both single-node and distributed settings via a collection of representative analytic operations. This paper presents the architecture of AFrame, describes the underlying capabilities of AsterixDB that efficiently support modern data analytic operations, and utilizes the proposed benchmark to evaluate and compare the performance and support for large-scale data analyses provided by alternative DataFrame libraries.**

***Index Terms*—DataFrames, distributed data management, large-scale data analysis, data science, benchmark**

I. Introduction

In this era of big data, extracting useful patterns and intelligence for improved decision-making is becoming a standard practice for many businesses. Modern data increasingly has three main characteristics: the first characteristic is that much of it is generated and available on social media platforms. The rapid growth in the numbers of mobile devices and smartphones, Facebook users, and YouTube channels all combine to create a data-rich social media landscape. Information distribution through this landscape reaches a massive audience. As a result, social media is now used as a medium for advertisement, communication, and even political discourse.

The second characteristic of modern data is the rapid rate at which the data is continuously being generated. In order to accommodate the rate and frequency at which modern data arrives, distributed data storage and management are required. Storing such massive data in a traditional file system is no longer an ideal solution because analysis often requires a complete file scan to retrieve even a modest subset of the data. In order to minimize time-to-insight, analyses need to be performed in close to real-time on the ever-arriving data. Database management systems are able to store, manage, and utilize indexes and query optimization to efficiently retrieve subsets of their data, enabling interactive data manipulation.

The third characteristic of modern data is the richness of the information encapsulated in the data. Modern data is not only massive in size but is also often nested and loosely-structured. For example, Twitter [15] provides JSON data containing information related to each message along with information about the user who posted that message and their location details if available. Other social media sites such as Facebook and Instagram provide similar information through their web services. As a result, modern data enables analyses that go beyond interpreting content; one can also analyze the structure and relationships of the data, such as identifying communities. Information extraction from modern data requires complex custom algorithms and analyses using machine learning.

The growing interest in collecting, monitoring, and interpreting large volumes of modern data for business advantages motivates the development of data analytic tools. The requirements that modern, at-scale data analysis impose on analytic tools are not met by a single current system. Instead, data scientists are typically required to integrate and maintain several separate platforms, such as HDFS [35], Spark [5], and TensorFlow [16], which then demands systems expertise from analysts who should instead be focusing on data modeling, selection of machine learning techniques, and data exploration.

In this paper, we focus on providing a 'scale-independent' user experience when moving from a local exploratory data analysis environment to a large-scale distributed workflow. We present AFrame, an Apache AsterixDB [18] based extension of DataFrame. AFrame is a data exploration library that provides a Pandas-like DataFrame [27] experience on top of a big data management platform that can support large-scale semi-structured data exploration and analysis. AFrame differs from other DataFrame libraries by leveraging a complete big data management system and its query processing capabilities to efficiently scale DataFrame operations and optimize data access on large distributed datasets.

The second contribution of this paper is a distributed DataFrame benchmark for general data analytics. The performance of a big data system is greatly affected by the charac-

978-1-7281-0858-2/19/$31.00 © 2019 IEEE

teristics of its workload. Understanding these characteristics and being able to compare various systems' performance on a set of related analytic tasks will lead to more effective tool selection. Various benchmarks [9], [22], [24], [26], [32] have been developed for big data framework assessment, but these benchmarks are either SQL-oriented benchmarks for OLTP or OLAP operations or focus on end-to-end application-level performance. To our knowledge, there is no standard DataFrame benchmark yet for large-scale data analytic use cases.

In order to evaluate the performance of our framework, we have designed a micro-benchmark to compare various distributed DataFrame libraries' performance by issuing a set of common analytic operations. Our DataFrame benchmark provides a detailed comparison of each analytic operation by separating the data preparation time (e.g., DataFrame creation) and expression execution time to give better insight into each system's performance and operation overheads.

The rest of this paper is organized as follows: Section 2 discusses background and related work. Section 3 provides an overview of the AFrame system architecture, user model, and data analytic support. In Section 4, we describe the proposed DataFrame benchmark. Section 5 details our initial experiments and discusses their results. We discuss future improvements and conclude the paper in Section 6.

II. Background

An important motivation for the AFrame project comes from the need to make the management of large-scale modern data available to the larger audience of the data science community by integrating the DataFrame user experience with a big data management system. Here we discuss the foundations of exploratory data analysis and some advantages and disadvantages of its standard evaluation strategy.

A. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) [38] is an investigation process employed by analysts to extract information, identify anomalies, discover insights, and understand the underlying structures and characteristics of a dataset. The goal of EDA is to provide analysts with clues and a better understanding of the data in order to formulate reasonable hypotheses. Important applications of EDA include, but are not limited to, data exploration, cleaning, manipulation, and visualization.

Frameworks and technologies often used in these applications span across the fields of statistics and machine learning. There are a large number of prepackaged machine learning libraries that cover a wide variety of user requirements. However, not all of the machine learning frameworks that work out of the box are designed to work in a distributed environment. As a result, analysts have to resort to large-scale machine learning frameworks such as MLlib [29] because extensive effort is required to make locally constructed models operate on big data. Often times, these large-scale machine learning frameworks do not cover all types of analysis and models.

Since EDA involves visualizing data, collecting statistics from the data, and is iterative in nature, most of the available tools targeting these types of analyses only accommodate smaller datasets and leave big data processing and the scaling out of the algorithmic process for data engineers to implement. In order to reduce the turn-around time and increase productivity for data analysts, several issues need to be addressed: 1) distributed application of custom machine learning models; 2) providing a seamless migration from a local workflow to a distributed environment; 3) having a scalable system that can acquire and operate on ever-changing incoming data.

B. Eager vs. Lazy Evaluation

EDA frameworks such as Pandas target a local workstation environment and often rely on in-memory processing. These frameworks require data to be loaded into memory before any analysis operations can be performed on the data. Once the data is loaded into memory, analysis operations are evaluated eagerly, meaning as soon as they are initiated. However, a similar evaluation strategy is not efficient on large-scale ever-arriving data, as processing every declared operation without any optimization would be expensive as it may result in repetitive scans over massive data.

Eager and lazy evaluation are strategies used in programming languages to determine when expressions should be evaluated [34]. While eager evaluation causes programs to evaluate expressions as soon as they are assigned, lazy evaluation is the opposite and delays their evaluation until their values are required. With eager evaluation, programmers are responsible for ensuring code optimization to prevent performance degradation due to unnecessary operations over large datasets. Lazy evaluation, on the other hand, delays execution until values are required; it is employed to help with operation optimizations where multiple operations can be chained together, extended, and a single iteration over the source collection can be processed, e.g., as in LINQ [28]. As a result, lazy evaluation is more suitable for exploratory operations on large-scale data. Its performance improvement becomes critical as the size of the data grows.

C. Related Platforms

We can compare and contrast existing systems in terms of Big data platforms and DataFrame technology.

1) ***Big Data Platforms:*** Here we consider frameworks that can operate on distributed data.

Apache Spark: Apache Spark [40] is a general-purpose cluster computing system that provides in-memory parallel computation on a cluster with scalability and fault tolerance. SparkSQL [20] is a module to simplify users' interactions with structured data. SparkSQL integrates relational processing with Spark's functional programming. MLlib [29], which is built on top of Spark, provides the capability of constructing and running machine learning models on distributed data. However, Spark does not provide data management and it requires the installation of a distributed file system like HDFS.

Hive: Apache Hive [3] is data warehouse software built on top of Apache Hadoop for providing data summary, query, and analysis capabilities. The introduction of Hive reduced the

complexity of having to write pure MapReduce programs by providing a SQL-like interface and translating the input queries into MapReduce programs to be executed on the Hadoop platform. Now Hive also includes Apache Tez [6] and Apache Spark [5] as alternative query runtimes. However, to leverage Hive's processing power, knowledge of SQL is essential in addition to being able to install and appropriately configure and manage Hadoop and HDFS.

Apache AsterixDB: Apache AsterixDB [2], [18] is a parallel open source Big Data Management System (BDMS) that provides full distributed data management for large-scale, semi-structured data. AsterixDB utilizes a NoSQL style data model (ADM) which is a superset of JSON. Before storing data into AsterixDB, a user can create a Datatype, which describes known aspects of the data being stored, and a Dataset, which is a collection of objects of a Datatype. Datatypes are "open" by default, in that the description of the data does not need to be complete prior to storing it; additional fields are permitted at runtime. This allows for uninterrupted ingestion of data with ever-changing data schemas. AsterixDB provides SQL++ [21], a highly expressive semi-structured query language for users that are familiar with SQL, to explore stored NoSQL data.

Figure 1 shows an example of creating an open datatype 'Tweet' with only the field 'id' being pre-defined and two datasets called 'TrainingData' and 'LiveTweets' which store records of this Tweet datatype. The TrainingData dataset is populated by reading data from a local file system. In this example, it is being populated using a labeled airline sentiment dataset. AsterixDB also provides support for user-defined functions (UDFs) and built-in live social media data acquisition through its data feed feature. The LiveTweets dataset is populated by connecting a data feed called 'TwitterFeed' that continuously ingests Twitter data. (More details on how to create a live Twitter feed can be found in [2], [17]). Figure 1 also creates two indexes on the LiveTweets dataset.

```
CREATE TYPE Tweet AS{id: int64};
CREATE DATASET TrainingData(Tweet);
CREATE DATASET LiveTweets(Tweet);
LOAD DATASET TrainingData USING localfs
 (("path"="1.1.1.1:///airline_data.json"),
  ("format"="adm"));

CREATE  FEED  TwitterFeed  WITH {...};
CONNECT  FEED  TwitterFeed  TO  LiveTweets;
START  FEED  TwitterFeed;

CREATE PRIMARY INDEX ON LiveTweets;
CREATE INDEX coordIdx ON LiveTweets(coordinate);
```

Fig. 1: SQL++ queries

*2) **DataFrames for Data Science**:* Here we consider libraries that provide a DataFrame facility.

Pandas: Pandas [12] is an open source data analysis tool that provides an easy-to-use data structure built specifically to support data wrangling in Python. Pandas reads data from various file formats (e.g., CSV, SQL databases, and Parquet) and creates a Python object, DataFrame, with rows and columns similar to a table in Excel. Pandas can be integrated with scientific visualization tools such as Jupyter notebooks [25]; Jupyter notebooks provide a unified interface for organizing, executing code and visualizing results without referring to low-level systems' details. The rich set of features that are available in Pandas makes it one of today's most popular tools for data exploration. However, its limitation lies in scalability. Pandas does not provide either data storage or support for interacting with distributed data, as its focus has been on in-memory computation on a single node. Another well-known Pandas' limitation is its memory consumption. This is caused by the underlying internal memory requirements about which the Pandas creator, McKinney, advised: "you should have 5 to 10 times as much RAM as the size of your dataset" [1].

R Data Frames: R [14] is a language originally built for statistical computing and graphics. Since R is primarily used for statistical analysis, R has become one of the most popular languages in the data science community. R also provides Data Frame as a built-in native data structure, but working with data larger than memory in R still requires a distributed framework and data storage setup. For example, SparkR [39] is an R package created by Apache Spark that supports distributed operations like R Data Frames but on large datasets.

Spark DataFrames: Spark also provides a DataFrame API [19] to enable the wider audience of the data science community to leverage distributed data processing. This API is designed to support large-scale data science applications with inspirations from both the R DataFrame and Python Pandas. Spark employs the lazy evaluation technique to perform computations only when values are required. This is different from the eager evaluation strategies used in Python and R. Lazy evaluation is exploited by Spark's query optimizer, which understands the structure of the data and the operations. In order for Spark to determine the input data schema for unstructured data, a process called 'schema inference' is required and can result in long wait times for data that does not fit in memory.

Pandas on Ray: Pandas on Ray, which recently become a part of the Modin project [11], is a recent attempt to make Pandas DataFrames work on big data by providing the Pandas syntax and transparently distributing the data and operations using Ray [31]. Ray uses shared memory and employs a distributed scheduler to manage a system's resources. Pandas on Ray automatically utilizes all available cores on a machine or a cluster to execute operations in parallel. Since the Ray framework handles large data through shared memory, it requires a cluster with sufficient aggregate memory to hold the entire dataset. In addition, Pandas on Ray uses Pandas as a black box at its core, which does not address the high memory consumption issue of Pandas.

III. AFrame System Architecture

Exploratory tools such as Pandas work well against locally stored data that fits in the memory of a single machine, but this is not a solution for large-scale analysis. Still, Pandas is one of the most widely used libraries for data exploration due to the analyst-friendly characteristics of its data structure. As a result, we set out to integrate a Pandas-like user experience

with big data management capabilities to provide analysts with a familiar environment while scaling out their analytic operations over a large data cluster to enable big data analysis.

Our goal in the AFrame project is to create a unified system that can efficiently support all of the various stages [30] in data science projects, from data understanding to model deployment and application, thus enabling very large-scale analysis and requiring little or no modification to analysts' existing local workflows. Instead of building such a system from scratch, we extend Apache AsterixDB with support for the use of machine learning libraries and with interactive data exploration capabilities. Here we describe the underlying architecture of AFrame, the relevant AsterixDB features, and illustrate AFrame's basic functionality through a small running example that shows how to perform a simple sentiment analysis on ever-growing Twitter data.

A. Acquiring Data

AFrame is an API that provides a DataFrame syntax to interact with AsterixDB's datasets; it targets data scientists who are already familiar with Pandas DataFrames. AFrame works on distributed data by connecting to AsterixDB's web-service using its RESTful API. Figure 2 shows how users can use AFrame in a Jupyter notebook to access datasets stored in AsterixDB. Input 2 (labeled "In [2]") creates an AFrame object (trainingDF) from the TrainingData dataset initialized via the SQL++ statements in Figure 1. Input 3 creates another AFrame object (liveDF) from the LiveTweets dataset, which is connected to a data feed that continuously ingests data from Twitter. Building on top of AsterixDB allows AFrame to operate on such live data the same way as it does on a static dataset without requiring additional knowledge about how to setup a streaming engine. Since Figure 1 created indexes on the LiveTweets dataset, the incoming data is also appropriately stored and indexed for efficient data access.

```
In [2]: trainingDF = AFrame(dataverse='demo', dataset='TrainingData')

In [3]: liveDF = AFrame(dataverse='demo', dataset='LiveTweets')
```

Fig. 2: Initializing AFrame Objects

B. Operating on Data

As most EDA tools are designed to work with in-memory data, the eager evaluation strategy can suffice even when a session involves multiple scans over the entire dataset. However, multiple scans over a large distributed dataset would be very costly and have a negative effect on system performance.

AFrame leverages lazy evaluation. AFrame operations are incrementally translated into SQL++ queries that are sent to AsterixDB (via its RESTful API) only when final results are called for. Figure 3 shows an example of some expressions in AFrame when issuing Pandas-like DataFrame expressions. Input 4 (labeled In [4]) issues a selection predicate on the live dataset declared in Figure 2. Input 5 performs attribute projections. Neither inputs 4 or 5 trigger query evaluation; they only modify an underlying AFrame query. Input 6 performs an action that requests the actual output of two records, so AFrame takes the underlying query, appends a 'LIMIT 2' clause to it, sends it to AsterixDB for evaluation, and displays the requested data. For debugging purposes, AFrame allows users to observe the underlying query resulting from the incremental query formation process. Input 7 prints the underlying query resulting from Input 4. Input 8 prints the underlying query of Input 5 (which adds projected attributes to the selection query). These are examples of queries that correspond to simple DataFrame operations. However, even complex DataFrame expressions that result in nested SQL++ queries are efficiently translated into optimized query plans in order to minimize data access. This is another benefit of operating on AsterixDB and utilizing its query optimizer.

In its early development stage, AFrame today covers essential Pandas' operations for exploratory analyses that are suitable for large-scale unordered data. Currently, AFrame's supported operations include column selection and projection, statistical operations (e.g., describe), arithmetic operations (e.g., addition, subtraction, etc.), applying functions, joining, categorizing data (sorting and ordering), grouping (group by and aggregation), and persisting data.

```
In [4]: known_coords = liveDF[liveDF['coordinate'].notna()]

In [5]: coords = known_coords[['text','coordinate']]

In [6]: coords.head(2)
Out[6]:
```

	coordinate	text
0	[-94.6939, 38.97039]	Asi luce la ciudad de chicago \nMucho frio mas...
1	[-89.822763, 30.302944]	I'm at Goauto in Slidell, LA https://t.co/p1QB...

```
In [7]: known_coords.query
Out[7]: 'SELECT VALUE t FROM demo.LiveTweets t
        WHERE t.coordinate IS KNOWN;'
In [8]: coords.query
Out[8]: 'SELECT t.text, t.coordinate FROM demo.LiveTweets t
        WHERE t.coordinate IS KNOWN;'
```

Fig. 3: DataFrame expressions and underlying queries

C. Support for Machine Learning Models

Following the data wrangling and hypothesis forming process, distributed systems are often required to accommodate the development and usage of customized machine learning models. The goal of the modeling step is to create an effective machine learning model that can make accurate predictions. With AFrame, analysts can apply either a prepackaged model or create a custom machine learning model from their local environment that can be applied to a distributed dataset directly from within a Jupyter notebook.

Figure 4 illustrates a sentiment classifier training session using Python, Scikit-Learn [33], Pandas, and AFrame. It trains a classifier on the training dataset from Figure 2. This is a dataset, publicly available on Kaggle [10], containing Twitter posts related to users' experiences with U.S. airlines released by CrowdFlower [7]. The dataset contains labeled

```
In [9]: pandas_df = trainingDF[['text','sentiment']].toPandas()
        X = pandas_df['text']
        y = pandas_df['sentiment']
        pipeline = Pipeline([
            # pipeline construction code
        ])
        X_train, X_test, y_train, y_test = train_test_split(...)
        pipeline.fit(X_train, y_train)
        pickle.dump(pipeline, open("sentiment", 'wb'))
```

Fig. 4: Training a Scikit-Learn Pipeline

tweet sentiments which are positive, negative, and neutral. The first step in Figure 4 selects a subset of attributes from the training dataset. Since the subsetted training data is small enough to fit in a single node's memory[1], here we convert it to a Pandas DataFrame and use it to build and train a Scikit-Learn pipeline to classify sentiment values. The last step after training the model saves it as an executable which can then be dropped into AsterixDB and utilized as a UDF.

In Figure 5, we show sample code for applying machine learning models in AFrame using the Pandas-style map function syntax on the 'text' column to get sentiment value predictions. Input 10 in the figure displays a sample of the text column from the liveDF dataset created in Figure 2. Input 11 applies the pre-trained Stanford CoreNLP sentiment analysis model [37] to the text column and displays two records. The CoreNLP sentiment annotator produces 5 sentiment classes ranging from very negative to very positive (0-4). Input 12 applies our custom Scikit-Learn sentiment analysis model (created in Figure 4) to the same data.

Under the hood, AFrame utilizes AsterixDB's UDF framework to enable users to import and then apply their own machine learning models written in popular programming languages (e.g., Java and Python) as functions.

```
In [10]: liveDF['text'].head(2)
```

Out[10]:

	0
0	Sad thing is most people are to stupid to know...
1	meet the new boss same as the old boss https:/...

```
In [11]: liveDF['text'].map('demo.corenlp#getSentiment').head(2)
```

Out[11]:

	0
0	1
1	2

```
In [12]: liveDF['text'].map('demo.sklearn#getSentiment').head(2)
```

Out[12]:

	0
0	negative
1	neutral

Fig. 5: Applying CoreNLP and Scikit-Learn models

D. Result Persistence

After constructing a model, the next step would be to deploy the model and to apply it on real data. Input 13 in Figure 6 shows an example of how to apply the Scikit-Learn sentiment function to the 'text' field of a queried subset (coords) of the live Twitter records resulting from the operations in Figure 3. It then saves the sentiment prediction as a new field called 'sentiment'. Input 14 selects only records with negative sentiment for future root cause analysis. In AFrame, the result of an AFrame operation can optionally be persisted as another dataset by issuing the 'persist' command and providing a new dataset name, as shown by Input 15 in Figure 6. Persisting an analysis result is efficient here, as the data has never left AsterixDB storage and the new dataset (demo.negTweets) can be accessed right away without having to wait for a file scan. Input 16 displays sampled records from the new dataset created using AFrame; their sentiment is negative and they only contain a subset of the attributes from the original dataset.

```
In [13]: sentiment = coords['text'].map('demo.sklearn#getSentiment')
         coords['sentiment'] = sentiment

In [14]: neg = coords[coords['sentiment'] == 'negative']

In [15]: neg_af = neg.persist(name='negTweets', dataverse='demo')

In [16]: neg_af.head(2)
```

Out[16]:

	coordinate	sentiment	text
0	[-119.8716182, 34.429399599999996]	negative	Are you ready to grow and advance your beauty ...
1	[-104.9933088, 39.7540888]	negative	Current mood 😌 after finally having my first c...

Fig. 6: Persist Sentiment Analysis Results

E. Summary

We have demonstrated through an example how to use AFrame to acquire live Twitter data, manipulate the data, train and apply a custom Scikit-Learn model to get sentiments from the data, and save an analysis result for further investigation. AFrame provides a Pandas-like user experience without suffering from Pandas' single-node and in-memory requirements. AFrame does not load all data from a file or store its intermediate analysis results in memory. It can utilize database features to efficiently retrieve data and accelerate data manipulation on large-scale distributed data. By offloading data management to a distributed database system, AFrame remains a lightweight library that provides a scale-independent user experience to data scientists with any level of expertise.

IV. A DataFrame Benchmark

In order to evaluate our AFrame implementation and compare its performance to that of other distributed DataFrame libraries, we have constructed a preliminary DataFrame benchmark. Inspired by the early Wisconsin Benchmark [23] from the relational world, we propose a benchmark that evaluates DataFrames in several key dimensions that are important to conducting large-scale data analyses. This is similar to how the Wisconsin Benchmark was used to assess early relational database system performance. We also aim to provide members of the data science community with a tool to help them select a framework that is best suited to their project.

[1]Scikit-Learn's model training is required to take place on a single-node, but we are then able to utilize its trained models in a distributed setting.

Our DataFrame Benchmark is designed to evaluate the performance of DataFrame libraries against data of various sizes in both local and distributed environments. As an initial set of evaluated systems, we selected the following DataFrame frameworks: Pandas, PySpark, Pandas on Ray (Modin), and AFrame. There are several factors that contributed to our framework selection. First, since our goal is to support DataFrame syntax on large-scale data, it is appropriate to compare how systems perform with regard to the original Pandas DataFrames in a single node environment. Second, Apache Spark is a popular framework for distributed processing of large-scale data, so comparing against Spark DataFrames gives us a good understanding and comparison to a commercial and well-maintained DataFrame project. Pandas on Ray is another project that is trying to solve the same data scientists' problem, but using a different approach, so we also include it in our initial set of platforms.

A. Benchmark Datasets

In order to discover useful information from large volumes of modern data, most data science projects rely on data exploration. DataFrames are one of the most popular data structures used in data exploration and manipulation. A mature DataFrames library must be able to handle exploratory data manipulation operations on large volumes of data efficiently. The design of our DataFrame micro benchmark aims at reflecting these expectations in its workload.

For our benchmark datasets, we have chosen to use a synthetically generated Wisconsin benchmark dataset instead of using data from social media sites to allow us to precisely control the selectivity percentages, to generate data with uniform value distributions, and to broadly represent data for general analysis use cases (not just social media). A specification of the attributes in the Wisconsin benchmark's dataset is displayed in Table I. The unique2 attribute is a declared key and is ordered sequentially, while the unique1 attribute has 0 to (cardinality-1) unique values that are randomly distributed. The two, four, ten and twenty attributes have a random ordering of values which are derived by an appropriate mod of the unique1 values. The onePercent, tenPercent, twentyPercent, and fiftyPercent attributes are used to provide access to a known percentage of values in the dataset. The dataset also contains three string attributes: stringu1, stringu2, and string4. The stringu1 and stringu2 attributes derive their values from the unique1 and unique2 values respectively. The string 4 attribute takes on one of four unique values in a cyclic fashion; its unique values are constructed by forcing the first four positions of a string to have the same value chosen from a set of four letters: [A, H, O, V].

For our DataFrame benchmark, we used a JSON data generator to generate Wisconsin datasets of various sizes ranging from 1 GB (0.5 million records) to 40 GB (20 million records). In addition to JSON, we also evaluate systems using other widely used input formats, namely Parquet [4] and CSV.

Attribute name	Attribute domain	Attribute value
uniquel	O..(MAX-1)	unique, random
unique2	O..(MAX-1)	unique, sequential
two	0..1	uniquel mod 2
four	0..3	uniquel mod 4
ten	0..9	uniquel mod 10
twenty	0..19	uniquel mod 20
onePercent	0..99	uniquel mod 100
tenPercent	0..9	uniquel mod 10
twentyPercent	0..4	uniquel mod 5
fiftyPercent	0..1	uniquel mod 2
unique3	O..(MAX-1)	uniquel
evenOnePercent	0,2,4, ...,198	onePercent*2
oddOnePercent	1,3,5, ...,199	(onePercent *2)+ 1
stringul	per template	derived from uniquel
stringu2	per template	derived from unique2
string4	per template	cyclic: A, H, O, V

TABLE I: Scalable Wisconsin benchmark: attributes [23]

B. Benchmark Queries

The essential characteristic that makes DataFrame an appealing choice for data scientists is its stepwise syntax for exploratory tasks and data manipulation. As a result, we have designed our benchmark queries to target a set of core exploratory operations and visualization tasks. Table II summarizes the details of our initial DataFrame benchmark expressions. All evaluated frameworks except Pandas on Ray provide support for all of our benchmark expressions. Pandas on Ray defaults back to Pandas if the given expression has not yet been implemented to take advantage of its parallel processing engine. (In our case, these expressions are expressions 4, 8, and 12 in Table II.) Our initial set of expressions consist of analysis operations that include selection, projection, grouping, sorting, aggregation, and join. For expressions 2, 5, 9, and 10, we only asked for sampling because loading the entire dataset into memory would not be desirable in an exploratory big data context. For the join expression, both datasets are of the same size with the same number of records ranging from 1 GB to 40 GB. When executing the benchmark, each expression is run 15 times, and the first five results were excluded from the calculation to account for any JVM warm-up overheads. The recorded results are averaged over 10 runs. Our DataFrame benchmark expressions are detailed in Table II. We randomly generated values for the expression predicates (e.g., df['ten'] == $x) that fall within the tested attributes' range to reduce the effect of any in-memory caching between runs.

C. Evaluated System Details

The details of each systems' setup are provided below.

Pandas: Pandas DataFrame only works on a single machine environment and on data that fits in memory. It is important to note that Pandas only utilizes a single core for processing and that we use it with its default settings (without any additional configuration). It is labeled "Pandas" in the experimental results presented in this paper.

Spark: Spark indicates in its DataFrame API document that there is a significant difference in its DataFrame creation time when reading from JSON files if a data schema is provided. This performance benefit comes from eliminating its initial schema inference step. As a result, a dataset schema was also included in our benchmark. For single node experiments,

ID	Operation	Description	DataFrame Expression
1	Total Count	Total count	`len(df)`
2	Project	Project records on attributes two and four	`df[['two','four']].head()`
3	Filter & Count	Count records that satisfy column conditions	`len(df[(df['ten'] == x) & (df['twentyPercent'] == y) & (df['two'] == z)])`
4	Group By	Count records with the same column value	`df.groupby('oddOnePercent').agg('count')`
5	Map Function	Apply a function to a column	`df['stringu1'].map(str.upper).head()`
6	Max	Retrieve a max column value	`df['unique1'].max()`
7	Min	Retrieve a min column value	`df['unique1'].min()`
8	Group By & Max	Retrieve the max column value for each group	`df.groupby('twenty')['four'].agg('max')`
9	Sort	Order records based on a column	`df.sort_values('unique1', ascending=False).head()`
10	Selection	Retrieve some records that satisfy column conditions	`df[(df['ten'] == x)].head()`
11	Range Selection	Count records in a selected range	`len(df[(df['onePercent'] >= x) & (df['onePercent'] <= y)])`
12	Join & Count	Count records resulting from an inner join	`len(pd.merge(df, df2, left_on='unique1', right_on='unique1',how='inner'))`

TABLE II: Benchmark Operations (df, df2 = DataFrame objects, x,y,z = variables representing random values within range)

we used Spark in its local standalone operating mode. In the distributed environment, we configured HDFS as its distributed storage and used its standalone cluster manager. We evaluated Spark's DataFrame on both JSON and Parquet data using the default setup configurations. The three evaluated Spark variations are labeled "Spark JSON", "Spark JSON Schema", and "Spark Parquet" in the experimental results section.

AFrame: In order to evaluate AFrame, the benchmark datasets are expected to be resident in AsterixDB (as opposed, e.g., to HDFS) when running the operations. Similar to the Wisconsin benchmark queries, some of the expressions can benefit from indexes, so we executed the queries on both indexed and non-indexed data. Also, even though AsterixDB's default data typing is open, there is some benefit when a data schema is provided. Since we also provided Spark with a schema, we decided to also evaluate AFrame on a closed data type with the same pre-defined schema. The three evaluated AFrame variations are labeled "AFrame", "AFrame Schema", and "AFrame Index" in the experiments presented here.

Pandas on Ray: When we began evaluating the systems, Pandas on Ray had not yet provided cluster installation instructions, so we executed the DataFrame benchmark only on its single node setup. Notably, Pandas on Ray has implemented an impressive number of Pandas' operations to utilize all of the available cores in the given system. (For functions that have not been parallelized, it defaults back to using the original Pandas' operations.) When we did a preliminary run of the benchmark to check supported expressions, we noticed that Pandas on Ray had not yet parallelized Pandas' load_json method, so we decided to evaluate Pandas on Ray using CSV files instead. Pandas on Ray is based on a shared, in-memory architecture; its strength lies in in-memory computation. However, the project has started to implement support for large datasets using disk as an overflow for in-memory DataFrames.

D. Experimental Setup

Our DataFrame benchmark provides a set of configurable parameters to enable both single-node and cluster performance evaluations. The same suite of benchmark queries were applied to both settings. Each evaluated framework handles DataFrame creation differently, and some utilize an eager evaluation strategy while the others employ lazy evaluation. On top of that, depending on the flow of an analysis session, data might or might not already be available in memory, resulting in additional time to create a DataFrame before issuing analytic operations. Sometimes, when only a small subset of the data is needed, DataFrame creation time can dominate the overall actual operation time. As a result, we separately consider expression-only run times and total run times (which include both the DataFrame creation and expression execution times).

In order to provide a reproducible environment for evaluating these systems, we set all of the evaluated systems up and executed our benchmark on Amazon EC2 instances. For each node, we selected the m4.large instance type with the Linux 16.04 operating system, 2 cores, 8 GB of memory, and 100 GB of SSD.

*1) **Single-Node Setup**:* We generated the Wisconsin benchmark as JSON data in various sizes ranging from 1 GB (0.5 million records) to 10 GB (5 millions records). The Parquet and CSV datasets were created by converting the JSON files; they contained the exact same logical records as the JSON datasets. Table III shows the numbers of records and the byte sizes of each dataset for all file formats. The sizes of the Parquet files are significantly smaller due to its compression and its internal data representation. The JSON structure is based on key-value pairs. Each JSON record contains all of the necessary information about its content, and in principle each record could contain different fields in different orders. CSV is more compact than JSON due to the facts that its schema is only declared once for the whole file and that each record has an identical list of fields in the exact same order. Parquet is a column-oriented binary file that contains metadata about its content. Parquet is the most compact file format among the three formats tested.

	Dataset Name				
	XS	S	M	L	XL
Number of Records	0.5 mil	1.25 mil	2.5 mil	3.75 mil	5 mil
JSON File Size	1 GB	2.5 GB	5 GB	7.5 GB	10 GB
Parquet File Size	43 MB	110 MB	217 MB	317 MB	426 MB
CSV File Size	715 MB	2.3 GB	4.6 GB	6.8 GB	9.3 GB

TABLE III: Dataset Summary (mil = million)

*2) **Multi-Node Setup**:* For the multi-node setting, we only evaluated Spark and AFrame. The evaluated cluster size ranged from 2-4 nodes, where each node is a worker except for one node that is also a master. Speedup and scaleup are the two preferred and widely used metrics to evaluate the processing

performance of distributed systems, so we evaluated the two systems using these two metrics.

Speedup Experiment: Ideal speedup is when increasing resources by a certain factor to operate on a fixed amount of data results in the overall task processing time being reduced by the same factor. As a result, speedup reduces the response time, which also makes resources available sooner for other tasks. Linear speedup is not always achievable due to reasons such as start up cost and system interference between parallel processes accessing shared resources.

For our DataFrame benchmark, we conducted speedup experiments using a fixed size dataset while increasing the number of machines from one up to four. The details are summarized in Table IV, where aggregate memory is the sum of all of the available memory in the cluster.

	1 node	2 nodes	3 nodes	4 nodes
Aggregate Memory	8 GB	16 GB	24 GB	32 GB
JSON File Size	10 GB	10 GB	10GB	10 GB
Parquet File Size	426 MB	426 MB	426 MB	426 MB

TABLE IV: Speedup Experiment Setup

Scaleup Experiment Ideal scaleup is the system's ability to maintain the same response time when both the system resources and work (data) increase by the same factor.

For the scaleup experiments, we increased both the number of machines and the amount of data proportionally, as summarized in Table V, to measure each system's performance.

	1 node	2 nodes	3 nodes	4 nodes
Aggregated Memory	8 GB	16 GB	24 GB	32 GB
JSON File Size	10 GB	20 GB	30GB	40 GB
Parquet File Size	426 MB	818 MB	1.33 GB	1.75 GB

TABLE V: Scaleup Experiment Setup

V. Initial Benchmark Results

In this section, we present the initial experimental results from both the single-node and cluster environments.

A. Single Node Results

For the single-node evaluations, we ran the test suite first on the XS Wisconsin dataset as a preliminary test to determine the level of feature support in each framework and to observe their relative performance across all twelve expressions. The XS results are displayed in Figure 7. After the first round, we ran the benchmark on four other dataset sizes, S, M, L and XL to evaluate the data scalability of each framework on a single node. Due to space limitations, Figure 8 only displays selected results that illustrate some key advantages and disadvantages of each evaluated system. Note that Figures 7a to 7d and 8e are all in log scale. As mentioned in the experimental setup section, we present both the expression run times and the total run times (which include the DataFrame creation times).

1) ***Baseline Performance Results:*** The XS results are presented in Figure 7. Figures 7a and 7b show the total run times (including DataFrame creation). Figure 7a displays expression 1-6's results and Figure 7b displays expression 7-12's results. Figures 7c and 7d show the expression-only execution times. Figure 7c displays expression 1-6's results, and Figure 7d displays expression 7-12's results. The differences between the total and expression-only times indicate that the DataFrame creation process can significantly impact performance.

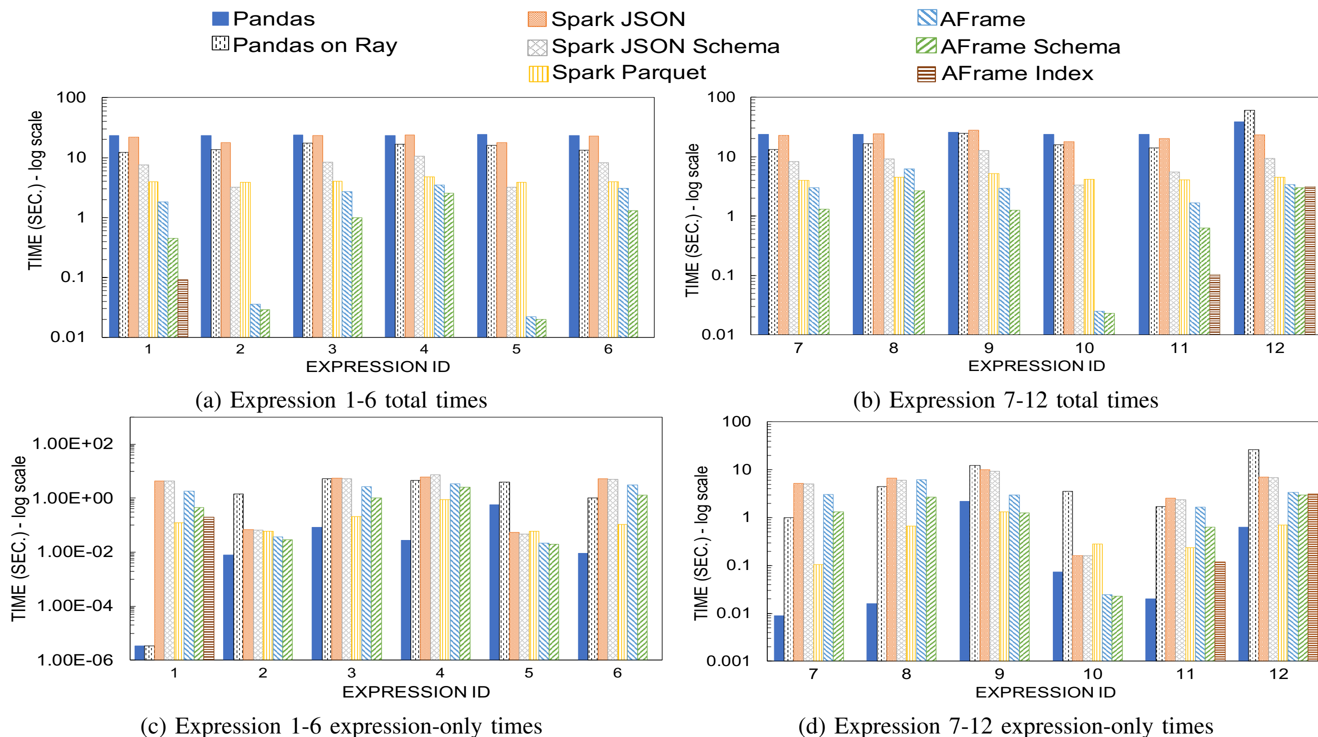
Pandas requires data to be loaded into memory before its operation evaluations. Since it was not designed for parallel processing, the total run time including DataFrame creation was high for Pandas in all of the test cases. However, once the data was loaded into memory, as shown in Figures 7c and 7d, Pandas performed the best in 10 of the 12 expressions. The two cases where Pandas was not the fastest were Expressions 5 and 10, and the reason was Pandas' eager evaluation strategy. Expression 5 applies a function to a string column, while expression 10 selects rows that satisfy a column predicate. However, in the end, both expressions 5 and 10 require only a small subset (head()) of rows from the dataset. The strict nature of Pandas' eager evaluation caused both the function and the predicate to be applied to the whole dataset before selecting only a few samples to return. On the other hand, with lazy evaluation, the expressions can be applied to just the subset of data needed to fulfill the result's required size.

Pandas on Ray leverages parallel processing by utilizing all available cores in a system to load and process the data. However, there are overheads associated with distributing a DataFrame, as we can see from Figures 7c and 7d, where Pandas outperformed Pandas on Ray on all but one expression. However, Pandas on Ray's total run time was better than Pandas' due to parallel data loading. As the size of the data grows, so does the time taken to process the data. Pandas on Ray outperforms Pandas when the task processing time dominates its work distribution overheads.

Among the three Spark DataFrames, the Parquet-based DataFrame (Spark Parquet) outperformed the JSON-based DataFrame (Spark JSON) and the JSON-based DataFrame with a pre-defined schema (Spark JSON Schema) in most of the tested cases for both the total and expression-only evaluation metrics. Spark produces different runtime plans for the JSON-based DataFrame and the Parquet-based DataFrame, resulting in the difference in their task execution times even after the schema inferencing step.

AFrame was the fastest in terms of the total-time evaluation since its DataFrame creation process does not involve first loading data into memory from a file. In addition, AFrame also benefits from the presence of database indexes. Its total-time performance results for the datasets with indexes are an order of magnitude faster, as seen for Expressions 1 and 11. Even in terms of just the expression-only time, AFrame with an index on the range attribute performed better than Spark Parquet on Expression 11 (see Figure 7d).

2) ***Scalability:*** After the first evaluation round, we evaluated the systems' single-node data scalability by running each expression on all five different data sizes. Due to space limitations, we only display a selected subset of the results. (The full set of results can be found in [36].) As we can see from Figures 8a and 8b, Pandas and Pandas on Ray



(a) Expression 1-6 total times

(b) Expression 7-12 total times

(c) Expression 1-6 expression-only times

(d) Expression 7-12 expression-only times

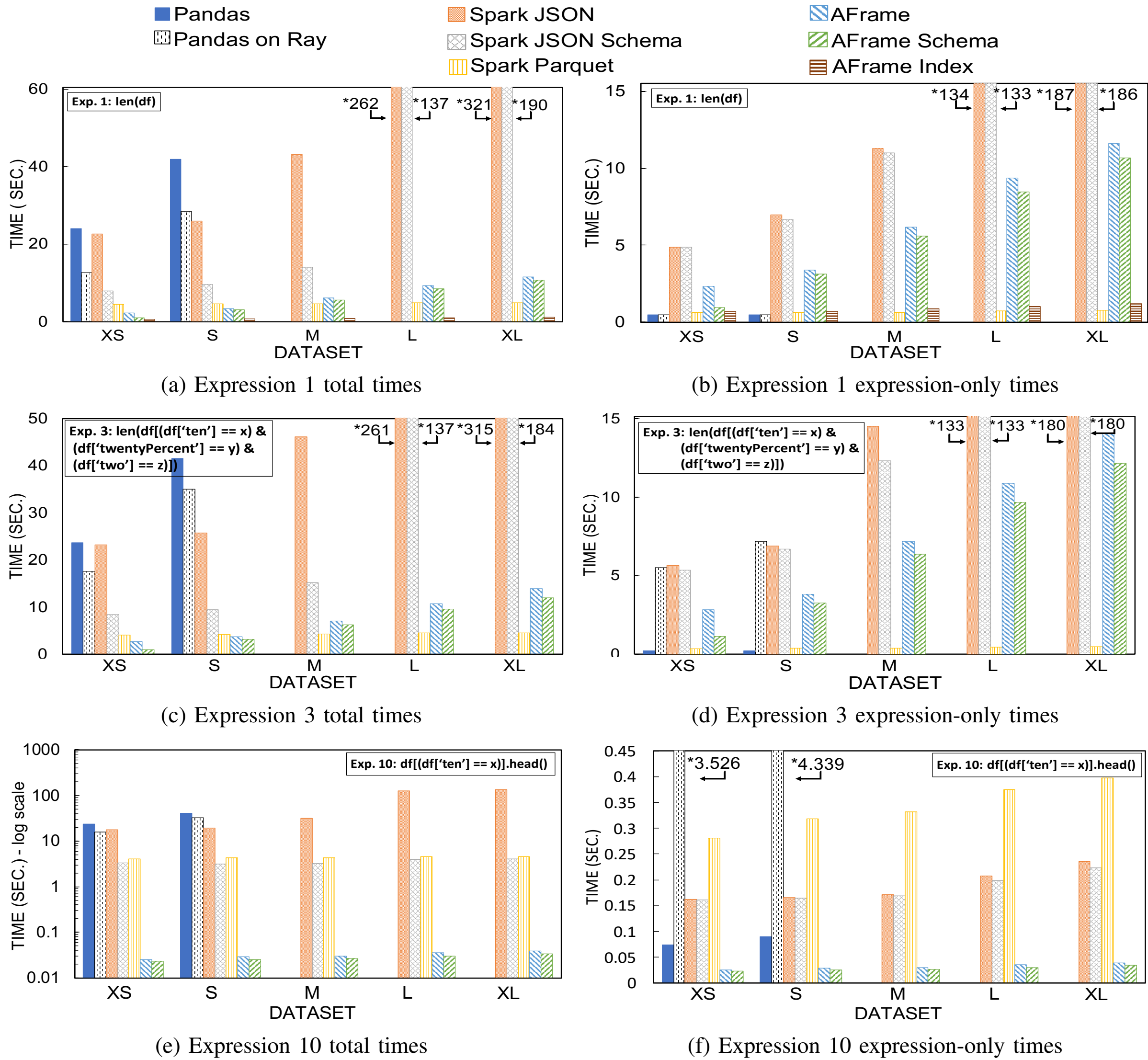
Fig. 7: XS Results of Single Node Evaluation

were not able to complete the DataFrame creation process for the M-XL datasets (5-10 GB) due to insufficient memory. A possible workaround would be to load the data in smaller chunks; we did not consider applying this workaround because it would result in customizing the data chunk size and that would directly affect the performance evaluation. Pandas on Ray suffers from the same memory limitations as Pandas since it uses Pandas internally.

The results of our single node scalability evaluation are largely consistent with those from our first run of functionality checking. There are some interesting results in the cases of running Spark on L and XL datasets, which are 7.5 and 10 GB of JSON data (Figures 8a to 8d). These results are much slower than the others in terms of both the total and expression-only elapsed times. These results are explained by Spark's default settings and its memory management policy. By default, Spark reserves one GB less than the available memory (MAX_MEMORY - 1) for its executor's memory. In our case this results in 7 GBs of memory being reserved for the executor tasks. When working with data that is larger than the available memory, Spark processes it in partitions and spills data to disk if it has insufficient memory. The L and XL datasets require Spark to spill to disk in order to complete the tasks, which results in long task execution times. In the Spark JSON case, providing a schema when creating a DataFrame from JSON files allows Spark to completely skips the schema inference step. This results in a lower total run time than when a schema is not provided. However, excluding the DataFrame creation time, whether or not the schema was provided, there was no significant performance difference between Spark JSON and Spark JSON Schema across all expressions.

In contrast to JSON, Spark's Parquet-based DataFrame performance results were consistent throughout all data sizes because the Parquet files are much smaller than the JSON files used to generate them. Since Parquet is supplied with a data schema and is a column-oriented format, it is especially suitable for column-based queries such as attribute projections. One factor to keep in mind is that even the Parquet-based DataFrame requires some DataFrame creation overhead. Figure 8c displays the total elapsed time for expression 3, which asks for the count of records that satisfy column conditions. We can see that for the XS and S datasets, the Parquet-based DataFrame total time results were slower than AFrame. However, as the data size increases and the task processing time becomes more prominent, the Parquet-based DataFrame starts to have a better run time than AFrame. The Spark Parquet-based DataFrame starts to benefit when the operation time exceeds the DataFrame creation time. In turn, for the expressions that require access to whole records, such as expression 10, as seen in Figure 8f, Spark's JSON-based DataFrame performed significantly better than its Parquet-based (columnar) DataFrame. Even in the case that includes the DataFrame creation time, shown in Figure 8e, Spark's JSON-based DataFrame with a pre-defined schema was faster than Parquet for all data sizes for Expression 10.

AFrame benefits from database optimizations like query planning and indexing. For expression 1, which asks for a total record count, AFrame with a primary key index performed the best for all data sizes. AFrame benefits from having indexes



(a) Expression 1 total times

(b) Expression 1 expression-only times

(c) Expression 3 total times

(d) Expression 3 expression-only times

(e) Expression 10 total times

(f) Expression 10 expression-only times

Fig. 8: Selected Single Node Evaluation Results (* = value where the bar ends)

on the join attributes (Expression 12), as shown earlier in Figure 7b; also as the size of the dataset gets larger, the others suffer more from long DataFrame creation times because they have to scan an additional dataset for this expression. AFrame was faster than Spark's JSON-based DataFrames in most of the test cases in Figure 7 and continued to be so as shown in Figure 8 for both expression-only and total times evaluations. AFrame without indexes was slower than Spark Parquet in most of the column-based expression-only times. However, for whole-row-based expressions, such as expression 10 (Figures 8e and 8f), AFrame without indexes performed better than Spark Parquet and were the best for both the expression-only and total run time evaluations.

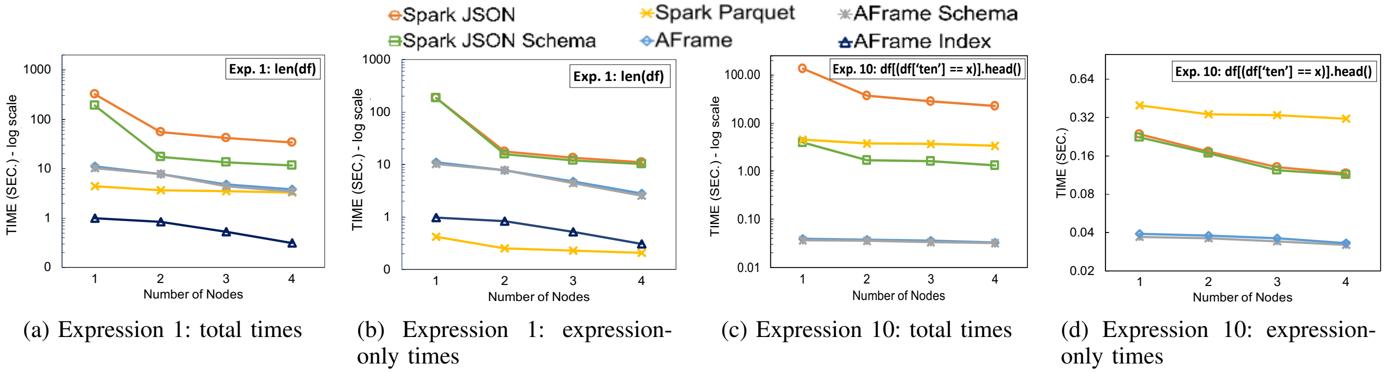
B. Multi-Node Results

For the distributed environment evaluation, as mentioned earlier, we have only evaluated Spark and AFrame. We evaluated Spark on the same three DataFrame creation sources: JSON, JSON with schema, and Parquet. Likewise, we evaluated AFrame on its same three datasets, which are datasets with an open datatype, with a schema, and with an index.

For the multi-node evaluation, we evaluated the systems' performance in a distributed environment. As we observed in the single node evaluation, Spark spills to disk for both the L and XL datasets (7.5 and 10 GB), which significantly affected its performance. In order to observe the effect of clusters processing data that is larger than the available aggregate memory, we chose to start our multi-node evaluation with the 10-GB dataset. Here we evaluated both systems according to both the speedup and scaleup metrics.

The multi-node evaluation was performed on ec2 machines with the same specifications as the single node evaluation. Due to space limitations, Figures 9 and 10 only display selected multi-node scaleup and speedup evaluation results. (Again, the full set of results are available in [36].)

1) Speedup Results: The results for both Spark and AFrame are consistent with their single-node results in terms of their



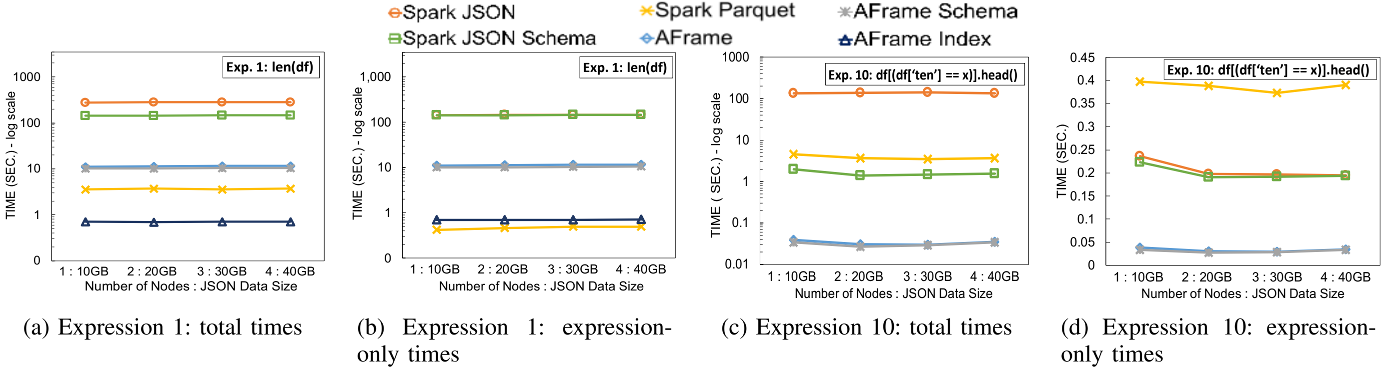
(a) Expression 1: total times

(b) Expression 1: expression-only times

(c) Expression 10: total times

(d) Expression 10: expression-only times

Fig. 9: Selected Multi-Node Speedup Evaluation Results (log scale)



(a) Expression 1: total times

(b) Expression 1: expression-only times

(c) Expression 10: total times

(d) Expression 10: expression-only times

Fig. 10: Selected Multi-Node Scaleup Evaluation Results

performance rankings. Both systems processed the tasks faster when increasing the number of processors while maintaining the same data size. Spark's performance improved drastically when the distributed data begin to fit in memory in the case of JSON DataFrames. Figures 9a and 9b show that increasing the number of processing nodes reduces Spark JSON-based DataFrame's run time by an order of magnitude in the case of going from a single node to a 2-node cluster. However, once the data fits in memory, increasing the number of nodes no longer results in such a drastic change (as we can see from the flatter lines for both of Spark's JSON-based DataFrames going from 2 nodes to 4 nodes). For expression 1, AFrame with an index and Spark's Parquet-based DataFrame performed the best. AFrame operating on a dataset with a primary key index was faster than Spark in the total time case, and Spark's Parquet-based DataFrame was best in terms of the expression-only time. Similar to the single node results, the Parquet-based DataFrame was the slowest in expression-only evaluation when access to the entire data record is required, as seen for expression 10 in Figure 9d across different numbers of nodes. In both Figures 9c and 9d, AFrame with and without schema are the fastest.

2) Scaleup Results: In Figure 10, selected scaleup results for Spark and AFrame are presented. No single system performed the best across all tasks. Spark's Parquet-based DataFrame was the fastest for column-based expressions and was consistently competitive, but it also incurred an overhead for DataFrame creation. However, for row-based expressions, AFrame continued to follow the same trend from the single node case with the XL dataset, outperforming Spark Parquet.

As we saw in Figure 10c, by providing the Spark JSON-based DataFrames with a schema, the total time is reduced by an order of magnitude, especially when only a subset of data is required. Expression 10 only samples a few records from a large dataset, which causes the schema inference time to otherwise dominate the actual expression execution time.

C. Discussion

Pandas performed competitively on all tasks for a single node when the data fits in memory. However, its weaknesses lie in resource utilization and scalability. The memory requirement for Pandas is large and it can only take advantage of a single processing core. In addition, Pandas' eager evaluation strategy has disadvantages when expressions involve potentially repetitive tasks. Operations that only view a small subset of the data took longer on Pandas than on frameworks that utilize parallel processing and/or lazy evaluation.

Pandas on Ray did an excellent job in functionally covering Pandas operations. It reroutes operations to the default Pandas when its parallel work distribution has not been enabled for an operation. While treating Pandas DataFrame as a black box does not solve the problem of its memory requirement, it utilizes parallel processing for loading and processing data in order to speed up the computation. Evaluating the system as-is reveals that there can be significant overhead associated with work distribution for Pandas on Ray. This is a known issue which is mentioned in the project's own benchmarking

results [13]. Its experimental out-of-core support will be worth looking into once it is enabled and distributed installation instructions are provided.

Spark DataFrame provides similar syntax to that of Pandas' with the ability to operate on data that exceeds the per-node memory limit; it provides a friendly interface to the Apache Spark distributed compute engine. While Spark can operate on large datasets, its performance drastically degrades when having to work with insufficient cluster memory as its strength lies in in-memory computation. As a result, on large datasets, its JSON-based DataFrame was an order of magnitude slower than AFrame. On the other hand, its Parquet-based DataFrame performed quite competitively across all data sizes. Due to its compression, a Parquet file is much smaller than a JSON file with the same logical data content. Finally, Parquet is a columnar file format, which makes the Parquet-based DataFrame an excellent fit for column-based operations but slower on tasks that require access to the entire payload of each data record.

A unique characteristic that sets AFrame apart from other large-scale DataFrame libraries is its ability to operate on managed and indexed data. AFrame benefits from its AsterixDB backend in several ways. First, it can eliminate repetitive file scans during the DataFrame creation process since datasets have been ingested and stored on disk in AsterixDB. Second, it is able to operate on data larger than the available memory, seamlessly, without requiring additional effort. Third, it eliminates issues that could arise from manually managing large amounts of data from various sources. Flat file storage requires effort to maintain and can be difficult to share between multiple users; modifying data in traditional storage can be prone to corruption because of a lack of transactional support. In addition, by having a distributed data management system as its backend, complex DataFrame operations that would otherwise execute inefficiently can be optimized by a database query optimizer. AsterixDB provides query plan optimization and indexing that enable AFrame to perform competitively, especially in terms of the total time evaluations.

VI. Conclusions & Future Work

In this work, we have shown the practicality of utilizing a distributed data management system to scale data scientists' familiar DataFrame operations to work against modern data at scale without requiring distributed data engineering expertise. We can also increase data analysts' productivity by optimizing their operations' execution times through lazy evaluation and database query optimization. AsterixDB also provides additional benefits to AFrame, such as the ability to utilize pre-trained models from packages such as Scikit-Learn (as-is) without requiring specialized large-scale machine learning skills. Finally, with AsterixDB's built-in social media data feeds, data scientists can operate on live datasets in the same way that they would work with static data.

In order to evaluate our initial AFrame prototype, we have also proposed a DataFrame benchmark for evaluating DataFrame performance on analytic operations. Our benchmark can be used in both single-node and distributed settings. Our experiments showed that AFrame can operate competitively in both settings. We have also demonstrated that optimizations can be crucial when dealing with data at scale. Our DataFrame benchmark, even at this early stage, can help data scientists better understand the performance of their workloads and understand distributed frameworks' tradeoffs.

Moving forward, we have a list of new functionality and improvements that we would like to implement for both AFrame and the DataFrame benchmark. We are adding nested data handling and window functions to AFrame. We also plan to make AFrame less query language specific by abstracting its language layer from the DataFrame operation translation mechanism. We would then be able to deploy AFrame on other query-based data management systems (e.g., Postgres).

The DataFrame benchmark is preliminary work that has served a purpose by allowing us to evaluate the feasibility of AFrame and to compare its initial performance against other frameworks. However, the benchmark is a work in progress and needs more analytic operations to be included in order to evaluate other aspects of distributed DataFrames. We also intend to add more frameworks (e.g., Dask [8]) to our evaluation and to deploy them in a much larger distributed environment.

VII. Acknowledgements

The work reported in this paper was supported in part by a Thai government scholarship, a UCI/ICS Exploration Award, and The Donald Bren Foundation (via a Bren Chair).

References

[1] Apache Arrow and the 10 Things I Hate About pandas. https://wesmckinney.com/blog/apache-arrow-pandas-internals/.
[2] Apache asterixdb. https://asterixdb.apache.org/.
[3] Apache Hive. https://hive.apache.org/.
[4] Apache Parquet. https://parquet.apache.org/.
[5] Apache Spark. http://spark.apache.org/.
[6] Apache Tez. http://tez.apache.org/.
[7] CrowdFlower. http://www.crowdflower.com/.
[8] Dask. http://dask.org/.
[9] GraySort benchmark. http://sortbenchmark.org/.
[10] Kaggle. http://www.kaggle.com/crowdflower/twitter-airline-sentiment/.
[11] Modin. https://modin.readthedocs.io/en/latest/.
[12] Pandas. http://pandas.pydata.org/.
[13] Pandas on Ray. https://rise.cs.berkeley.edu/blog/pandas-on-ray-early-lessons/.
[14] R. http://www.r-project.org/.
[15] Twitter API. http://developer.twitter.com/.
[16] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.
[17] W. Alkowaileet et al. End-to-end machine learning with Apache AsterixDB. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 6. ACM, 2018.
[18] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.
[19] M. Armbrust et al. Scaling Spark in the real world: Performance and usability. *PVLDB*, 8(12):1840–1843, 2015.
[20] M. Armbrust et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.
[21] D. Chamberlin. SQL++ for SQL Users: A Tutorial. September 2018. *Available via Amazon.com*.

[22] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.

[23] D. J. DeWitt. The Wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[24] A. Ghazal et al. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208, 2013.

[25] T. Kluyver et al. Jupyter Notebooks-a publishing format for reproducible computational workflows. In *Proceedings of the 20th International Conference on Electronic Publishing (ELPUB)*, pages 87–90, 2016.

[26] X. Liu et al. Smart meter data analytics: Systems, algorithms, and benchmarking. *ACM Transactions on Database Systems (TODS)*, 42(1):2, 2017.

[27] W. McKinney et al. Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[28] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *SIGMOD*, pages 706–706, 2006.

[29] X. Meng et al. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[30] Microsoft. TDSP: Team Data Science Process. 2017.

[31] P. Moritz et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, 2018.

[32] R. O. Nambiar and M. Poess. The making of TPC-DS. In *PVLDB*, pages 1049–1058, 2006.

[33] F. Pedregosa et al. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12(10):2825–2830, 2011.

[34] D. A. Schmidt. *The Structure of Typed Programming Languages*. MIT press, 1994.

[35] K. Shvachko et al. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[36] P. Sinthong and M. J. Carey. AFrame: Extending DataFrames for large-scale modern data analysis (Extended Version). *arXiv preprint arXiv:1908.06719*, 2019.

[37] R. Socher et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1631–1642, 2013.

[38] J. W. Tukey. *Exploratory Data Analysis: Limited Preliminary Ed*. Addison-Wesley Publishing Company, 1970.

[39] S. Venkataraman et al. SparkR: Scaling R programs with spark. In *SIGMOD*, pages 1099–1104, 2016.

[40] M. Zaharia et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.