PolyFrame: A Retargetable Query-based Approach to Scaling Dataframes

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

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

ABSTRACT

In the last few years, the field of data science has been growing rapidly as various businesses have adopted statistical and machine learning techniques to empower their decision-making and applications. Scaling data analyses to large volumes of data requires the utilization of distributed frameworks. This can lead to serious technical challenges for data analysts and reduce their productivity. AFrame, a data analytics library, is implemented as a layer on top of Apache AsterixDB, addressing these issues by providing the data scientists' familiar interface, Pandas Dataframe, and transparently scaling out the evaluation of analytical operations through a Big Data management system. While AFrame is able to leverage data management facilities (e.g., indexes and query optimization) and allows users to interact with a large volume of data, the initial version only generated SQL++ queries and only operated against AsterixDB. In this work, we describe a new design that retargets AFrame's incremental query formation to other query-based database systems, making it more flexible for deployment against other data management systems with composable query languages.

PVLDB Reference Format:
Phanwadee Sinthong and Michael J. Carey. PolyFrame: A Retargetable Query-based Approach to Scaling Dataframes. PVLDB, 14(11): 2296 - 2304, 2021.
doi:10.14778/3476249.3476281

1 INTRODUCTION

Extracting valuable trends and intelligence for enhanced decision-making is becoming a common necessity for many organizations in this age of big data. The growing interest in interpreting large volumes of user-generated content on social media sites for purposes ranging from business advantages to societal insights motivates the development of data analytic tools. The requirements that large-scale modern data analysis imposes on these tools are not met by a single system. Data scientists are thus required to integrate and maintain multiple separate platforms, such as HDFS [33], Spark [2], and TensorFlow [16], which demands systems expertise from analysts who should instead be focused on data modeling, selection of machine learning techniques, and data exploration.

AFrame [34] is a data exploration library that provides a Pandas-like DataFrame experience on top of Apache AsterixDB [17]. AFrame leverages AsterixDB's distributed data storage and management in order to accommodate the rapid rate and volume at which modern data arrives. Storing such massive data in a traditional file system is no longer an ideal solution because its analysis then requires complete file scans to retrieve even a modest subset of the data. Database management systems are able to store, manage, and utilize indexes and query optimization to efficiently retrieve subsets of the data, enabling much more interactive data manipulation.

Depending on the nature of the analysis, large amounts of data can be stored in different types of databases (e.g., document, time-series, or graph). AFrame, however, is language-dependent. It relies on specific features of AsterixDB and it is tightly-coupled with SQL++, limiting its adoption and usage. Instead of requiring data to be in a specific database system, PolyFrame enables users to retarget their data manipulation operations to their existing data stores. PolyFrame makes AFrame language-independent by creating a retargetable mapping between dataframe operations and composable database queries. The language-independence of PolyFrame is achieved by abstracting AFrame's language translation layer and retargeting its incremental query formation mechanism to operate against other database systems. We establish a set of rewrite rules to provide an extensible template for supporting other query languages, thus allowing AFrame to operate against other query-based database systems. As a proof-of-concept, we have applied our language rewrite framework to four different query languages, namely SQL++ [20], SQL [22], MongoDB Query Language [9], and Cypher [24], to retarget AFrame against AsterixDB [1], PostgreSQL [12], MongoDB [8], and Neo4j [10, 30] respectively.

In this paper we describe how we have re-architected AFrame to make it retargetable to other query-based database systems to allow their users to accomplish large-scale data analysis. The contributions of the resulting PolyFrame system are the following:

(1) We enable large-scale data analysis using a Pandas-like syntax on a variety of query-based database systems of choice.
(2) We identify common mapping rules between dataframe operations and database queries. This allows the system to reuse any combinations of the rules to construct queries that represent the supported dataframe operations.
(3) We extract and separate generic and language-specific rules to make it easy to introduce a new language, as the query composition mechanism is separated from the query syntax.
(4) We decompose complex Pandas dataframe operations (e.g., get_dummies, describe) into a sequence of simple operations via generic rewrite rules allows PolyFrame to utilize subqueries, which provides a simple localized model for language-specific mappings.

The rest of this paper is organized as follow: Section2 discusses background and related work. Section 3 provides an overview of our retargetable query-based design along with examples. Section

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476281

4 describes a set of performance experiments and results from running PolyFrame against different backend database systems. We conclude and describe open research problems in Section 5.

2 BACKGROUND AND RELATED WORK

We are rearchitecting AFrame (to create PolyFrame) by making it platform-independent, via flexible language rewrite rules, to create a framework that can support large-scale data analysis against a variety of backend databases. Here we review the basics of Pandas, AFrame, its current backend (AsterixDB), and related work.

2.1 Pandas

Pandas [11] is a Python data analytics framework that reads data from various file formats and creates an object, a DataFrame, with rows and columns similar to Excel. Pandas works with Python machine learning libraries such as Scikit-Learn [31]. The rich set of features that are available in Pandas makes it one of the most preferred and widely used tools in data exploration. However, its limitation lies in its lack of scalability. In addition, Pandas' internal data representation is inefficient as Wes McKinney (Pandas' creator) stated in [15] that a "rule of thumb for Pandas is that you should have 5 to 10 times as much RAM as the size of your dataset".

2.2 Apache AsterixDB

Apache AsterixDB is an open source Big Data Management System (BDMS) [17]. It provides distributed data management for large-scale, semi-structured data using the AsterixDB Data Model (ADM), which is a superset of JSON. In AsterixDB, data records are stored in datasets. Each dataset has a datatype that describes the stored data. ADM datatypes are open so that users can provide a minimal description even when the stored data can have additional contents.

2.3 AFrame

AFrame [34, 35] is a library that provides a Pandas DataFrame [27] based syntax to interact with data in Apache AsterixDB. AFrame targets data scientists who are already familiar with Pandas DataFrames. It works on distributed data by connecting to AsterixDB using its RESTful API. Inspired in part by Spark, AFrame leverages lazy evaluation to take advantage of AsterixDB's query optimizer. AFrame operations are incrementally translated into SQL++ queries that are sent to AsterixDB only when final results are called for.

2.4 Related Work

There are currently several scalable Pandas-like dataframe implementations. Here we compare PolyFrame with some of the better-known scalable dataframe libraries, polystore systems, and recent efforts that provide a dataframe interface on top of databases.

Spark: Apache Spark [37] is a general-purpose cluster-based computing framework. Spark also provides DataFrames [18], an API built on top of Spark SQL [19] for distributed structured data manipulation. However, Spark's DataFrame syntax is different from Pandas' in several respects. As a result, Koalas [6], a new open source project, was established to allow for easier transitioning from Pandas to Spark. Koalas provides a Pandas-like Dataframe API and uses Spark for evaluation. Koalas implements an intermediate data representation in order to support Pandas features such as row ordering in the Spark environment, which can result in performance trade-offs. Spark does not provide its own data storage, indexing, or data management. Spark can, however, load data from sources including databases via its Data Sources API [13] to create DataFrames. Users supply Spark with a database driver that implements support for read and write operations; the API allows filter and projection pushdown for performance optimization.

Modin: Modin [7] is another attempt to scale Pandas DataFrames by supporting the Pandas syntax and distributing the data and operations using a shared memory framework called Ray [28]. Modin supports Dask as its alternative backend. By running on Ray and Dask [4], Modin automatically utilizes all the available cores on a machine to execute Pandas operations in parallel. However, Modin does not provide data storage and it uses Pandas internally.

Polystores: Polystore systems provide integrated access to multiple data stores with heterogeneous storage engines through a common language. In [29], polystores are described as systems that typically share a common mediator-wrapper architecture in which a mediator process accepts input queries, interacts with data stores to obtain and merge the results. The differences between polystore systems and PolyFrame are their interactions and intended usages. PolyFrame provides a common language of DataFrame operations for users to interact with a query-based database system. PolyFrame does not communicate or orchestrate queries between data stores.

Scaling dataframes with databases: Efforts to scale dataframes have started gaining traction in the database community. Recently, Jindal et. al introduced Magpie [26], a system that provides a Pandas-like API and automatically determines an optimal backend for query execution. Magpie in turn is built on top of Ibis [5], a Python polystore-like engine that provides its own proprietary API and is capable of interacting either eagerly or lazily with backends including Spark, Pandas, and RDBMSs. A concurrent effort, Grizzly [23], introduced by Hagedorn et. al, translates the Pandas API into nested SQL queries with additional feature support for lambda expressions as UDFs and external file ingestion. In contrast, PolyFrame focuses on incrementally building queries by utilizing an identified set of mappings between dataframe operations and database queries that can be applied to a wide variety of composable query languages.

3 POLYFRAME SYSTEM OVERVIEW

In this section, we briefly describe PolyFrame's architecture, its incremental query formation process, and the new language rewrite component which is an architectural extension to make AFrame extensible for deployment against different widely used query-based database systems.

3.1 PolyFrame Overview

AFrame provides a Pandas-like DataFrame experience while scaling out the evaluation of its operations over a large cluster. The current implementation of AFrame does this by operating against Apache AsterixDB [17] and incrementally constructing nested SQL++ queries in order to mimic Pandas' (eager) evaluation characteristics while enabling lazy execution of the resulting queries. In order to make AFrame language-independent, PolyFrame separates its query language rewriting component from the original incremental query formation process.

Figure 1 outlines the new AFrame architecture (PolyFrame). An AFrame runtime object is created using a set of language-specific rewrite rules by selecting from those that we provide (e.g., SQL++, SQL, Cypher, MongoDB) or providing a set of custom rules. Inspired by Spark, each operation in AFrame can be categorized as either a transformation or an action. Transformations are operations that transform data. These operations are functions that take an underlying query (Qi) from an AFrame object and produce a new AFrame object with a new query (Qi+1). Transformatios will not trigger query evaluation, hence AFrame does not produce any intermediate results. Actions are operations that trigger query evaluation. This is done through a database connector that sends the underlying query (Qn) of an AFrame object to a database. The database connector is an abstract class in AFrame that makes connections to database engines. It also performs AFrame initialization, pre-processing of queries before sending them to the database, and post processing of queries' results from the database. A new database connector can be added by providing an implementation of these three required methods. Query's results are returned as a Pandas Dataframe.

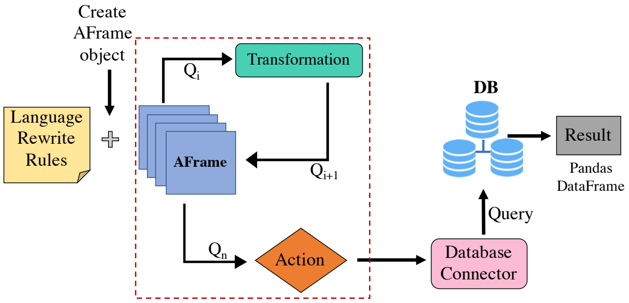


Figure 1: AFrame's New Architecture (PolyFrame)

3.2 Incremental Query Formation

PolyFrame incrementally constructs queries in order to mimic Pandas' eager evaluation characteristics and record the order of operations. However, it utilizes lazy evaluation to take advantage of databases' query optimization. Figure 2 shows an AsterixDB example of six SQL++ queries generated as a result of Pandas DataFrame operations. The Dataframe operations are listed on top of each PolyFrame object (the numbered rectangles). The corresponding SQL++ queries are listed below the objects. The first PolyFrame object (marked 1) is created by passing in the dataverse and dataset name of an existing dataset in AsterixDB. Notice how each subsequent SQL++ query is composed from the query resulting from the previous operation. Operations 1 to 5 are transformations. For these types of operations, PolyFrame does not load any data into memory nor execute any query. Operation 6 (which is asking for a sample of 10 records) is an action that triggers the actual query evaluation. For this operation, PolyFrame appends a 'LIMIT 10' clause to the underlying query and uses a connection to a database (AsterixDB in this case) to send the underlying query and retrieve its results.

3.3 Query Rewrite

In order to separate the language syntax from the query formation process, we re-architected AFrame and established two sets of rewrite rules that govern how each query is constructed for a particular dataframe operation. Figure 3 shows the sequence of steps in PolyFrame's query rewriting process. In PolyFrame, a language configuration file contains query mappings that the system uses during the query formation process. Each PolyFrame object has an underlying query (Qi). When an operation is called on a PolyFrame object, the underlying query is passed into the rewriting process. A dataframe operation is inspected, and if possible, decomposed into multiple simple dataframe operations. Variables from each dataframe operation will also be extracted. The system uses generic rewrite rules (described in Section 3.4.1) to map each operation to one or multiple language-specific rules (described in Section 3.4.2). Query rewriting is then performed on each identified language-specific rule using string pattern matching to replace each token with the extracted common variables. The result is a new database query (Qi+1) encapsulated in a new PolyFrame object.

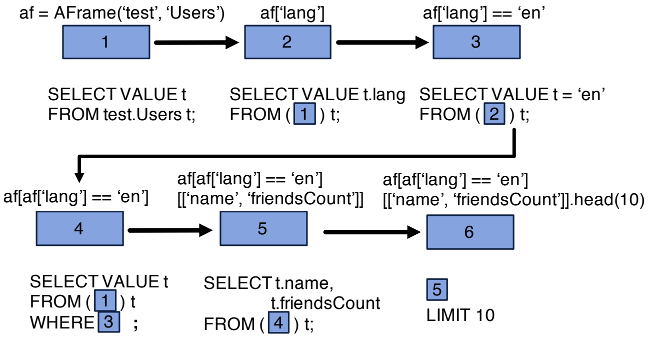


Figure 2: Incremental Query Formation

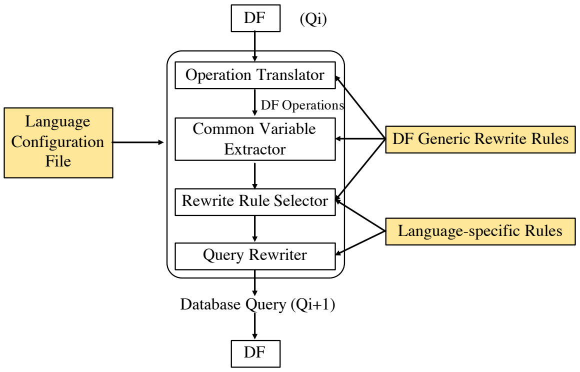


Figure 3: Flowchart of a query rewrite

Originally in AFrame, a SQL++ query was hard-coded in each dataframe operation body, while in PolyFrame, we use two levels of rewrite rules to create the operations' queries. Figure 4 shows two operation implementations (attribute projection and null value detection) in AFrame in comparison to PolyFrame. In PolyFrame, each operation uses the attribute-projection rule with three rewrite variables ('attribute', 'alias', 'subquery') that are overwritten at runtime. The 'isna' operation uses the null-operator rewrite rule in addition to the attribute-projection rule to construct its query. The separation of the target query language from the query construction allows PolyFrame to be easily extensible. The two-level approach also helps reduce the number of rewrite mappings required since the system can combine and reuse the rules to construct its queries.

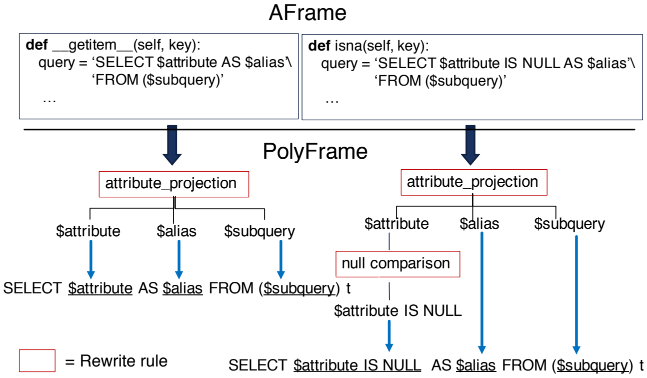

AFrame
def __getitem__(self, key):
query = 'SELECT $attribute AS $alias'\
'FROM ($subquery)'
…
def isna(self, key):
query = 'SELECT $attribute IS NULL AS $alias'\
'FROM ($subquery)'
…
PolyFrame
attribute_projection
$attribute
$alias
$subquery
SELECT $attribute AS $alias FROM ($subquery) t
attribute_projection
$attribute
$alias
$subquery
null comparison
$attribute IS NULL
SELECT $attribute IS NULL AS $alias FROM ($subquery) t
= Rewrite rule

Figure 4: AFrame vs. PolyFrame query construction

3.4 Per-Language Rewrite Rules

In order to preserve AFrame's incremental query formation and subquery characteristics, we target query languages that are composable. Another important requirement that any of PolyFrame's target database systems must satisfy is having an efficient query optimizer. Executing subqueries without optimization could result in unnecessary data scans that would affect performance. Fortunately, good query optimization is an important feature of databases.

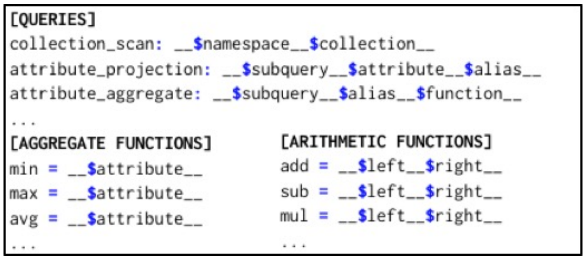
Currently, we support rewrites of many relational algebra operations in Pandas dataframes such as selection, projection, join, group by, aggregation, and sorting. Operations that require access to rows by indices are not supported because row ordering is not widely enforced in database systems. A challenge in generating common rewrite rules for PolyFrame has been distinguishing between configurable and general components across various languages. We have established two main types of rewrite rules. One type is generic rules and the other rule type is language-specific rules.

3.4.1 *Generic rules.*

Generic rules are rewrite rules that are not explicitly defined in a system-specific PolyFrame language configuration file. The purpose of our generic rules is to identify language-specific rule(s) for each dataframe operation. For simple dataframe operations (e.g., projection, unique), generic rules can map an operation directly to a language-specific rule. For complex dataframe operations, generic rules decompose these operations into a sequence of simple operations that can be translated via the existing language-specific rewrite rules. For example, the function 'describe()' in Pandas displays statistics for each attribute in a Dataframe. In PolyFrame, we construct this function by combining operations 1-7 in Figure 6 together to form a query that asks for aggregate values of specified attributes. Thus, instead of creating a rule for every function of Pandas Dataframe, these generic rules allow PolyFrame to efficiently utilize common components to form more complex queries that perform the desired function.

3.4.2 *Language-specific rules.*

These rules are the rewrite rules for translating dataframe operations into (sub) queries that have to be defined in a language configuration file due to the syntax differences across various languages. We supply users with a language configuration template file to allow adding a new query language or a similar query language with semantic variants. A sketch of PolyFrame's template file is displayed in Figure 5. The template file contains rewrite variables that can be rearranged or omitted to represent the required query behavior. Its pre-defined variables will be rewritten at runtime when a user interacts with PolyFrame objects. These rules are defined in such a way that they can be combined to create complex queries. In addition to the general dataframe operations that we support, we also require rules for translating arithmetic operations (addition, subtraction, multiplication, division, etc.), aggregation (e.g., sum, average, count, min, and max), comparison statements (equal, not equal, greater than, less than, etc.), logical operations (and, or, and not), and attribute aliases. A challenge in establishing a set of language-specific rewrite rules was identifying the granularity of the rules while maintaining efficiency. Defining the granularity too fine would yield rules that cannot be reused or combined to compose other methods and would require too much effort to maintain. The rules need to be generalized across different languages and not rely exclusively on system-specific optimizations. Our goal was to identify the common components that are shared across query languages.



```
[QUERIES]
collection_scan: __$namespace__$collection__
attribute_projection: __$subquery__$attribute__$alias__
attribute_aggregate: __$subquery__$alias__$function__
...
[AGGREGATE FUNCTIONS]      [ARITHMETIC FUNCTIONS]
min = __$attribute__       add = __$left__$right__
max = __$attribute__       sub = __$left__$right__
avg = __$attribute__       mul = __$left__$right__
...                        ...
```

Figure 5: Configuration Template Overview

ID	Operation	SQL++	MongoDB	Cypher

1	Return all records	SELECT VALUE t
		FROM
		$namespace.$collection t
			{ "$match": {} }	MATCH(t: ***$collection***)

2	Return an attribute aggregate	SELECT ***$func*** AS ***$alias***
		FROM (***$subquery***) t
			$subquery,
			{"$group": {"_id": {},
			"***$alias***": {***$func***}}},
			{"$project": {"_id": 0 }}
				$subquery
				WITH {`***$alias***`: ***$func***}
				AS t

3	Minimum	MIN(***$attribute***)	"$min": "$***$attribute***"	min(t.***$attribute***)
4	Maximum	MAX(***$attribute***)	"$max": "$***$attribute***"	max(t.***$attribute***)
5	Average	AVG(***$attribute***)	"$avg": "$***$attribute***"	avg(t.***$attribute***)
6	Count	COUNT(***$attribute***)	"$count": "$***$attribute***"	count(t.***$attribute***)
7	Standard deviation	STDDEV(***$attribute***)	"$stdDevPop": "$***$attribute***"	stDevP(t.***$attribute***)

Figure 6: Sample Rewrite Rules

Figure 6 shows a few implementation examples of the language-specific rewrite rules from Figure 5 in three query languages. The rules for all languages including handling of joins can be found in [36]. For these particular examples, SQL happens to share the same syntax as SQL++ for all operations except operation 1, so due to space limitations we only show SQL++, MongoDB, and Cypher here. Operation 2, for example, requires the language to return the aggregate value of an attribute. There are three rewrite variables

Table 1: PolyFrame's Incremental Query Formation

ID	AFrame Operation	SQL++	SQL	MongoDB	Cypher

1	af = AFrame('Test','Users')	SELECT VALUE t
		FROM Test.Users t
			SELECT *
			FROM Test.Users
				{"$match": {} }	MATCH(t: Users)

2	af['lang']	SELECT t. lang
		FROM (***1***) t
			SELECT t. lang
			FROM (***1***) t
				1,
				{"$project": { " lang ": 1 } }
					1
					WITH t{` lang `: t. lang }

3	af['lang'] == 'en'	SELECT VALUE
		t. lang = "en"
		FROM (***2***) t
			SELECT
			t. lang = "en"
			FROM (***2***) t
				2,
				{"$project": {"is_eq": {"$eq": [" lang ", "en"]}}}
					2
					WITH t{`is_eq`: t. lang = "en" }

4	af[af['lang'] == 'en']	SELECT VALUE t
		FROM (***1***) t
		WHERE t. lang = "en"
			SELECT t.*
			FROM (***1***) t
			WHERE t. lang = "en"
				1,
				{"$match": { "$expr": { "$eq": [" lang ", "en"] } } }
					1
					WITH t WHERE t. lang = "en"

5	af[af['lang'] == 'en']
	[['name', 'address']]
		SELECT t. name ,
		t. address
		FROM (***4***) t
			SELECT t. name ,
			t. address
			FROM (***4***) t
				4,
				{"$project": { " name ": 1, " address ": 1} }
					4
					WITH t{` name `:t. name , ` address `:t. address }

6	af[af['lang'] == 'en'] [['name', 'address']].head(10)	***5*** LIMIT 10 ;	***5*** LIMIT 10 ;	***5***, { "$project": { "_id": 0 } }, { "$limit" : 10 }	***5*** RETURN t LIMIT 10

(italicized), '$subquery', '$alias', and '$func'. A previous operation's underlying query will replace the variable '$subquery' and one of the aggregate functions (e.g., operations 3-7) will replace the variables '$func' and '$alias'. As also indicated in Figure 6, aggregate functions require a rewrite for a variable labeled '$attribute'. This is the name of an attribute. For example, to get the minimum value of 'age' from a dataset named 'Users' in a database named 'Test', PolyFrame will combine the results of operations 1, 2, and 3. First it will rewrite the variable '$namespace' of operation 1 as 'Test' and the variable '$collection' as 'Users'. The result of operation 1 will replace the variable '$subquery of operation 2'. It will then rewrite the variable '$attribute' in operation 3 with the value 'age' and use operation 3 to replace the variable '$func' in operation 2.

3.5 PolyFrame Examples

To demonstrate the generality of our approach, we have implemented a first prototype of PolyFrame that operates against AsterixDB, PostgreSQL [12], MongoDB [8] and Neo4j [10] by translating Dataframe operations into SQL++, nested SQL queries, MongoDB aggregation pipeline stages, and Cypher queries using 'WITH' statements. Table 1 displays query rewrites for SQL++, regular SQL, MongoDB, and Cypher that correspond to the PolyFrame operations from Figure 2. The highlighted parts of each query are generated by PolyFrame's query translation process, while the non-highlighted parts come directly from the provided language-specific rewrite rules. The bold italicized numbers are operation IDs. These IDs refer to query results from the indicated operation. We can see that SQL++ has much in common with SQL, but some differences exist due to the different data models of the two languages. The MongoDB and Cypher rewrites are very different, but the passed-in operation parameters are the same across all four languages. The full finished products of operation 6 rewritten in each of the languages are displayed in Listings 1-4.

Listing 1: SQL++ translation of operation 6

```
SELECT t.name, t.address
FROM (SELECT VALUE t
      FROM (SELECT VALUE t
            FROM Test.Users t) t
            WHERE t.lang = 'en') t
LIMIT 10;
```

Listing 2: SQL translation of operation 6

```
SELECT t.name, t.address
FROM (SELECT *
      FROM (SELECT *
            FROM Test.Users t) t
            WHERE t.lang = 'en') t
LIMIT 10;
```

Listing 3: MongoDB translation of operation 6

```
Test.Users.aggregate([
    { "$match": {} },
    { "$match": {"$expr": {"$eq":["$lang", "en"]}}},
    { "$project": { "name": 1, "address": 1 } },
    { "$project": { "_id": 0 } },
    { "$limit" : 10 }
])
```

Listing 4: Cypher translation of operation 6

```
MATCH(t: Users)
WITH t WHERE t.lang = "en"
WITH t{`name`:t.name, `address`:t.address}
RETURN t
LIMIT 10
```

For MongoDB, PolyFrame uses its aggregation pipeline language in order to obtain the incremental query formation leveraged for AFrame. As a result, certain optimizations might be limited for particular operations in a pipeline (as described in MongoDB's documentation). Operation 1 in Table 1 for MongoDB does not have any variable rewritten because our MongoDB rewrite rules are pipeline stages and pipeline constructions are handled through its database connector. Listing 3 displays a complete MongoDB aggregation pipeline for operation 6 from Table 1. Notice here that we include a '{"$project":{"_id":0}}' statement as part of the MongoDB's rewrite rule to exclude the MongoDB object identifier attribute '_id' from the final results. This attribute is the last attribute to be excluded in the pipeline because its presence is needed to enable index usage.

4 EXPERIMENTS

In order to demonstrate the value of database-backed dataframes and to empirically validate the generality of our language-rewrite approach working against different database systems, we have conducted two sets of experiments. One set illustrates the performance differences between a distributed data processing framework (Spark) that can consume data from databases vs. a framework

(PolyFrame) that instructs a database system to process the data. This set of experiments is included for the benefit of readers who may otherwise wonder why Spark plus its database connectivity are not simply the ultimate scaling answer. The other set of experiments illustrates our new architecture operating against different database systems to compute results as compared to Pandas Dataframes. We conducted our experiments using the Dataframe benchmark detailed in [34]. That Dataframe benchmark was originally developed to evaluate AFrame and to compare its performance with that of other Dataframe libraries. Note that we use the benchmark here as a demonstration of our new architecture (not to compare the performance of the different database systems). Performance results for AFrame versus other distributed dataframe alternatives are available in [34].

4.1 DataFrame Benchmark

To our knowledge, there is no standard benchmark for evaluating dataframe libraries. Therefore, when we first created AFrame we also implemented our own Dataframe benchmark to evaluate AFrame's performance. We use the same benchmark here to evaluate PolyFrame. The benchmark uses Wisconsin benchmark data [21]. An important feature of the benchmark is that it presents two separate timing comparisons. One is the total runtime, which includes both the DataFrame creation and the expression runtimes, and the other is the expression-only runtime. This is done to reflect the impact of the schema inferencing process. The benchmark timing points for Pandas and PolyFrame are fully described in [36].

4.1.1 ***Benchmark Datasets.*** The DataFrame benchmark issues its expressions against a synthetically generated Wisconsin benchmark dataset. This dataset allows us to precisely control the selectivity percentages, to generate data with uniform value distributions, and to broadly represent data for general analysis use cases. A specification of the attributes in the Wisconsin benchmark's dataset is displayed in Table 2. For our use case, we modified the Wisconsin dataset to include 10% missing values in its 'tenPercent' attribute. The unique2 attribute is a declared key and is ordered sequentially, while the unique1 attribute has a set of unique values that are randomly distributed. Other numerical 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. We used a JSON data generator [25] to generate the datasets.

In order to simulate running PolyFrame in a production environment, we also obtained a real-world data from Criteo [3] to evaluate PolyFrame's performance in a cluster environment. This dataset contains feature values and click feedback on display ads served by Criteo. It has 40 string and numerical attributes.

4.1.2 ***Benchmark Expressions.*** Table 3 displays the complete set of the DataFrame benchmark's expressions (where expression 13 is an added expression since [34]). For the Criteo dataset, we used the same operations but modified the attribute names prior to running the experiment.

4.2 Experimental Setup

In order to present a reproducible evaluation environment, we set up a benchmark cluster consisting of Amazon EC2 m4.large machines. Each machine has 8 GB of memory, 100 GB of SSD, and the Ubuntu Linux operating system.

Table 2: Scalable Wisconsin benchmark: attributes [21]

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 3: Dataframe Benchmark Operations (df, df2 = DataFrame objects, x,y,z = variables representing random values within an attribute's range)

ID	Operation	DataFrame Expression
1	Total Count	`len(df)`
2	Project	`df[['two','four']].head()`

3	Filter & Count	`len(df[(df['ten'] == x)`
		`& (df['twentyPercent'] == y)`
		`& (df['two'] == z)])`

4	Group By	`df.groupby('oddOnePercent').agg('count')`
5	Map Function	`df['stringu1'].map(str.upper).head()`
6	Max	`df['unique1'].max()`
7	Min	`df['unique1'].min()`
8	Group By & Max	`df.groupby('twenty')['four'].agg('max')`
9	Sort	`df.sort_values('unique1',ascending=False).head()`
10	Selection	`df[(df['ten'] == x)].head()`

11	Range Selection	`len(df[(df['onePercent'] >= x)`
		`& (df['onePercent'] <= y)])`

12	Join & Count	`len(pd.merge(df, df2,`
		`left_on='unique1',`
		`right_on='unique1',`
		`how='inner',hint='index'))`

13	Count Missing Value	`len(df[df['tenPercent'].isna()])`

We used the Wisconsin benchmark data generator to generate 1 GB and 10 GB of data in JSON file format.

4.2.1 ***Comparison with Spark (Single and Multi-Node).*** On a single node, we used two different data access methods for PySpark dataframes reading from a MongoDB instance. We used the MongoDB-Spark connector provided by MongoDB to read the data. For the first data access method (labeled 'Spark'), we used the connector to directly read the data from MongoDB. For the second data access method (labeled 'Spark+MongoDB pipeline'), we directly provided the connector with MongoDB aggregation pipelines. This is an optimization that Spark supports to push down a query and utilize database optimizations to lower the amount of data transferred back. The pipelines that we issued to Spark are the same ones that PolyFrame generated, and both Spark and PolyFrame were connected to the same MongoDB instance.

On a three-node cluster, we set up Vertica Community Edition 10.1.0 [14] with Spark workers co-located on the same nodes. For this cluster experiment, we ingested a 100 GB of real-world data subsetted from the Criteo dataset [3] into Vertica. The attributes used for the benchmark queries were changed to work with the underlying Criteo dataset. PolyFrame and Spark both connected to the same Vertica cluster. However, unlike MongoDB, the Vertica-Spark connector only supports projection and selection push-downs to the database.

4.2.2 ***PolyFrame Heterogeneity (Single and Multi-Node).*** To assess the benefits and feasibility of PolyFrame, we ran the DataFrame benchmark on Pandas and on PolyFrame operating on the four different database systems detailed below:

- AsterixDB: v.0.9.5 with data compression enabled
- PostgreSQL: v.12
- MongoDB: v.4.2 Community edition
- Neo4j: v.3.5.14 Community edition

4.3 Experimental Results

Here we present benchmark results for both the PolyFrame comparison with Spark and PolyFrame on different database systems.

4.3.1 ***Comparison with Spark (Single Node).*** For this experiment, we ran the benchmark on 10 GB dataset (which exceeds a single node memory capacity). Figure 7 shows the results for PolyFrame and Spark for all of the dataframe benchmark's expressions. PolyFrame performed the best across all of the expressions.

Spark was significantly slower than PolyFrame, even when operating on the same MongoDB instance, partly due to the data transfer time between MongoDB and Spark. PolyFrame sends queries to MongoDB directly, without first loading any data into memory for processing. This lets MongoDB process the operations and only return the queries' results. This design allows PolyFrame to take advantage of MongoDB's database optimizations (e.g., index) and to avoid loading large amounts of data into memory.

Spark with MongoDB pipelines had better performance than regular Spark because it reduces the amount of data needed to be transferred from the database into the Spark environment for processing. However, one can see that even with passed-down pipelines, Spark was still slower than PolyFrame. This is because the MongoDB pipelines that are passed through the connector are applied to each data partition, and not to the whole dataset. The number of data partitions is determined by MongoDB's partitioner in order to optimize data transfers to multiple Spark workers. Post-processing is then done at the Spark level.

4.3.2 ***Comparison with Spark (Multi-Node).*** For the cluster experiments, we ran the benchmark on 100GB of data from the Criteo dataset residing on a three-node Vertica cluster. We used the SQL language configuration file for operation-to-query translation. The results in Figure 8 are consistent with the single node results for MongoDB. However, unlike the MongoDB connector, the Vertica-Spark connector (provided by Vertica) does not provide an option for manual query pass-down. For Vertica, then, Spark can only take advantage of the database API's selection and projection push-down to limit the amount of data transferred. As a result, Spark and PolyFrame performance is similar on expressions 3 and 11, which issue range and exact-match queries respectively.

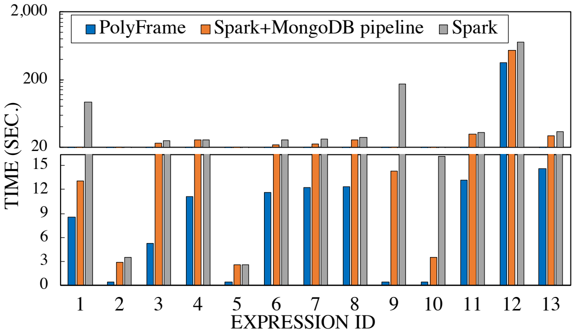


Figure 7: Single-node Experiment with Spark on MongoDB

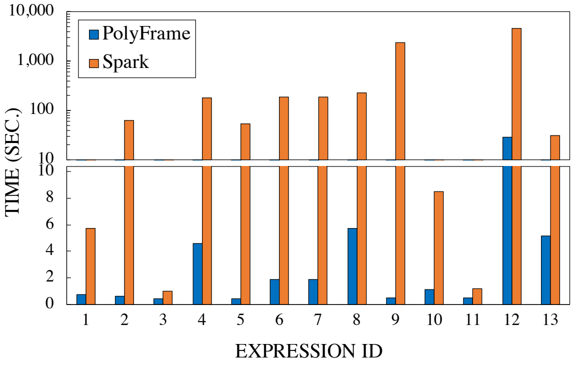
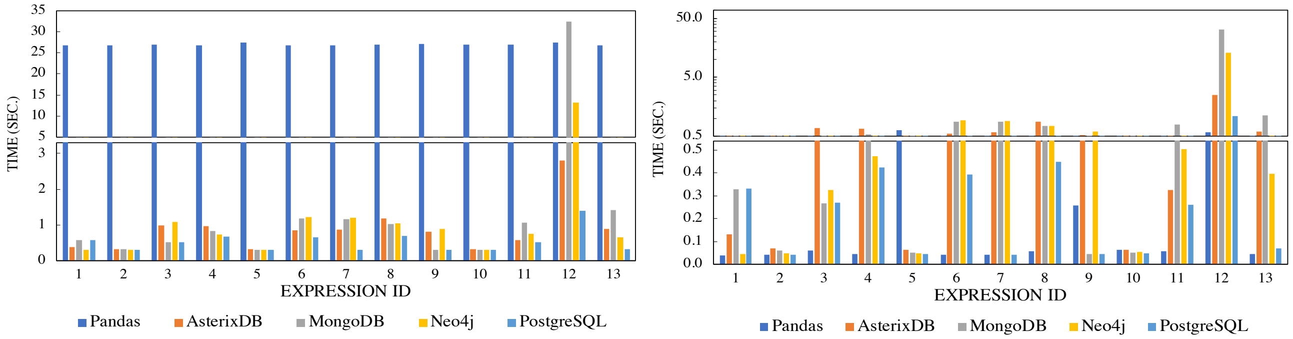


Figure 8: Cluster Experiment with Spark on Vertica

4.3.3 ***PolyFrame Heterogeneity (Single and Multi-Node).*** An important point to note here is that we began with a single node evaluation primarily to compare PolyFrame's database system-based lazy evaluation with Pandas' eager in-memory evaluation approach. We first executed the benchmark on a small benchmark dataset as a preliminary test before running it on the other bigger datasets. As mentioned, the DataFrame benchmark separately presents the DataFrame creation time and the expression-only runtime. Figure 9 presents the results for the single node evaluation. 9a displays the total runtimes for expressions 1-13, and 9b displays the expression-only runtimes for the expressions. The total evaluation times of Pandas were significantly higher than all variants of PolyFrame because Pandas has to load the entire dataset into memory to create its DataFrames. For the expression-only times, Pandas was then the fastest to complete most of the operations due to having the data already available in memory, except for expressions 5 and 10 where Pandas suffered due to eagerly evaluating sub-components of the expressions. In contrast, PolyFrame operating on top of the four database systems did not incur any DataFrame creation times. The four PolyFrame variants were all able to execute all benchmark expressions. The runtime results among these four database systems vary due to their each having different optimizations. For example:

- PolyFrame operating on top of PostgreSQL was able to take advantage of index-only query plans, backward index scans to retrieve a subset of records sorted in descending order (expression 9), and null value statistics using its indexes.



(a) Expression 1-13 total times

(b) Expression 1-13 expression-only times

Figure 9: Results of Single Node Evaluation

- For Neo4j, it provides fast metadata lookup for count of records (expression 1). We also found that its storage layout and record structure, which separate string attributes from numerical attributes, also contributed to its performance.
- For MongoDB, PolyFrame uses aggregation pipelines to facilitate incremental query formation and language rewriting process. Similar to PostgreSQL, a backward index scan and index searches are applied on the same set of expressions.
- AsterixDB offers a primary key index, which enables fast computation for count of records (expression 1). Index-only capabilities and index nested-loop joins significantly helped with calculating the count of records on the join results.

We also conducted multi-node experiments to demonstrate the scalability of PolyFrame on distributed database systems (AsterixDB, MongoDB, and GreenPlum). An instance of each database was installed on all of the machines in the 4-node cluster. The speedup and scaleup results for PolyFrame were mostly consistent with the single node results. Due to space limitations, we refer interested readers to the discussion of our cluster results in [36].

4.4 Discussion of Experiments

We conducted the Spark experiments to show important differences between utilizing database optimizations versus using an optimized compute engine to read and then process the data. It is important to note that passing queries down to a database can significantly lower the amount of transfer data. However, in Spark, doing so requires data scientists to be familiar with the database's query language in order to fully optimize Spark performance. Intuitively, if users can generate the needed database queries and then execute them directly on a database system, that yields the most optimal execution results, as shown in our experiments. However, it significantly reduces the benefits offered by the Dataframe abstraction.

Pandas performed competitively on all tasks when data fits in memory. However, due to its eager evaluation approach, it needs to accommodate intermediate computation results, which leads to higher memory consumption. In addition, Pandas suffered from resource under-utilization and has very limited scalability.

PolyFrame utilizes lazy evaluation, sending queries to an underlying database system only when actions are invoked. This allows PolyFrame to take advantage of database systems' optimizations. We conducted our single node experiments to compare PolyFrame's lazy evaluation approach with Pandas' eager in-memory evaluation approach (rather than comparing the performance of the different database systems). We demonstrated that operating on top of a database system indeed allows PolyFrame to take advantage not only of optimizations such as indexes and query optimization, but also of the data management capabilities that go beyond memory limits. PolyFrame does not require the loading of data into memory prior to computing expression results; this resulted in lower total runtimes across all benchmark expressions.

By configuring PolyFrame to work against significantly different database systems and query languages, we have demonstrated the generality and feasibility of its language rewrite rules. The flexibility of our rewrite rules allows PolyFrame to take advantage of each database system's optimizations while maintaining efficiency.

5 CONCLUSIONS AND FUTURE WORK

In this work, we have shown the practicality of retargeting AFrame's incremental query formation approach onto a variety of query-based database systems in order to scale dataframe operations without requiring users to have distributed or database systems expertise. The flexibility of our language rewrite rules enables database-specific optimizations and makes extending the Pandas DataFrame API to custom languages and systems possible. We evaluated PolyFrame versus Pandas DataFrames through a set of analytical benchmark operations. As a result, we have also shown that lazy evaluation, which takes advantage of database optimizations, is an efficient (and important) solution to big data analysis.

In its current stage, PolyFrame has already shown promising results for enabling a scale-independent data analysis experience. A recent paper [32] has given a formal definition to dataframe operators. It may be worthwhile to incorporate their dataframe algebra with our generic rewrite rules to provide an intermediate abstraction for query language mapping. Another research problem being explored in other similar dataframe libraries is how to support Pandas' notion of row labels efficiently in a distributed environment. There is not yet an efficient solution to enable row-indexing on unordered data (which is the data model used in most existing database systems); currently, an order is required in the form of either system-generated internal identifiers or sorted data to enable such a capability in a distributed environment. This results in a performance trade-off that we would like to eliminate if possible.

REFERENCES

[1] 2021. Apache AsterixDB. https://asterixdb.apache.org/.

[2] 2021. Apache Spark. http://spark.apache.org/.

[3] 2021. Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[4] 2021. Dask. http://dask.org/.

[5] 2021. IBIS. https://ibis-project.org/.

[6] 2021. Koalas. http://koalas.readthedocs.io.

[7] 2021. Modin. https://modin.readthedocs.io/en/latest/.

[8] 2021. MongoDB. http://mongodb.com/.

[9] 2021. MongoDB Aggregation. https://docs.mongodb.com/manual/aggregation/.

[10] 2021. Neo4j. http://neo4j.com/.

[11] 2021. Pandas. http://pandas.pydata.org/.

[12] 2021. PostgreSQL. http://www.postgresql.org/.

[13] 2021. Spark Data Sources. https://spark.apache.org/docs/latest/sql-data-sources.html.

[14] 2021. Vertica. https://www.vertica.com.

[15] 2021. Wes McKinney. https://wesmckinney.com/blog/apache-arrow-pandas-internals/.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[17] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *Proceedings of the VLDB Endowment (PVLDB)* 7, 14 (2014), 1905–1916.

[18] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling Spark in the real world: Performance and usability. *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1840–1843.

[19] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data processing in Spark. In *ACM International Conference on Management of Data (SIGMOD)*. 1383–1394.

[20] D Chamberlin. 2018. SQL++ for SQL Users: A Tutorial. September 2018. *Available via Amazon.com* (2018).

[21] David J DeWitt. 1993. The Wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook*, J. Gray (Ed.). Morgan Kaufmann.

[22] James R Groff, Paul N Weinberg, and Andrew J Oppel. 2002. *SQL: the complete reference*. Vol. 2. McGraw-Hill/Osborne.

[23] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting pandas in a box. In *Conference on Innovative Data Systems Research (CIDR)*.

[24] Florian Holzschuher and René Peinl. 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 195–204.

[25] Shiva Jahangiri. 2020. *shivajah/JSON-Wisconsin-Data-Generator*. https://doi.org/10.5281/zenodo.4315937

[26] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at speed and scale using cloud backends. In *Conference on Innovative Data Systems Research (CIDR)*.

[27] Wes McKinney et al. 2010. Data structures for statistical computing in Python. In *Proceedings of the Python in Science Conference*, Vol. 445. 51–56.

[28] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 561–577.

[29] M. Tamer Ozsu and Patrick Valduriez. 2019. *Principles of Distributed Database Systems* (4th ed.). Springer Publishing Company, Incorporated.

[30] Jonas Partner, Aleksa Vukotic, and Nicki Watt. 2013. *Neo4j in action*. Manning Publications Co.

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research* 12, 10 (2011), 2825–2830.

[32] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 2033–2046.

[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10.

[34] Phanwadee Sinthong and Michael J Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *IEEE International Conference on Big Data (Big Data)*. 359–371.

[35] Phanwadee Sinthong and Michael J Carey. 2019. AFrame: Extending DataFrames for large-scale modern data analysis (Extended Version). *arXiv preprint arXiv:1908.06719* (2019).

[36] Phanwadee Sinthong and Michael J Carey. 2020. PolyFrame: A Query-based approach to scaling Dataframes (Extended Version). *arXiv preprint arXiv:2010.05529* (2020).

[37] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.