UNIVERSITY OF CALIFORNIA,
IRVINE

A Retargetable Query-based Approach to Scaling Dataframes

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Phanwadee Sinthong

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Ramesh Jain

2021

DEDICATION

To my loving family.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation would not have been possible without the guidance and support from my advisor, Professor Michael Carey. He has been an excellent advisor and an understanding teacher who I will always look up to. I still remember vividly our first meeting in his office when I was still figuring out my research topic. Every meeting since then has been some of the most valuable learning experiences for me. Throughout my Ph.D. journey, I have learned from Professor Carey how to conduct research in the field of data management and more importantly, how to grow as a Ph.D. student. I am fortunate to have him as my advisor. I could not thank him enough for this remarkable opportunity.

I would like to thank Professor Chen Li and Professor Ramesh Jain for joining my dissertation committee. The first data management class I took was with Professor Li. He has given valuable and constructive feedback that has helped improve the quality of this dissertation in terms of important system requirements. Professor Jain has inspired an important part of the work in the last chapter of this dissertation. His expertise in usability analysis has guided me to conduct performance comparisons that significantly strengthen this dissertation and highlight its impact in a tangible and meaningful way.

I am very fortunate to have worked with Professor Heri Ramampiaro, Xikui Wang, Wail Alkowaileet, and Glenn Galviso. I very much value our meetings, stimulating conversations, and fruitful discussions. Their comprehensive observations and objective critiques have always inspired me to think outside the box and approach problems from different perspectives. I would like to thank the AsterixDB team especially Ian Maxon, Dmitry Lychagin, and Till Westmann for their insightful technical discussions and supports. I am tremendously grateful for their collaboration.

My journey as a Ph.D. student would not have been rewarding without my friends and colleagues. I am indebted to Praveen Venkateswaran, Shiva Jahangiri, Chen Luo, Taewoo Kim, Jianfeng Jia, Sumaya Almanee, Sameera Ghayyur, and Norrathep Rattanavipanon for their support and encouragement during the past few years. Our random conversations have kept me entertained and provided moral support, which made it possible for me to soldier through tough times and difficult moments.

I would like to thank Onnicha Krittayajaroenpong for her love, unwavering support, and understanding over the past five years. My journey would not have been as enjoyable without her.

Most importantly, I would like to thank my family for their unconditional love and support. I would not have made it this far without their understanding, patience, and encouragement.

The work reported in this dissertation has been supported in part by a UCI/ICS Exploration Award, by the Donald Bren Foundation (via a Bren Chair), and by NSF awards IIS-1954962 and CNS-1925610.

VITA

Phanwadee Sinthong

EDUCATION

Doctor of Philosophy in Computer Science **2021**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2015**
University of California, Los Angeles *Los Angeles, California*

Bachelor of Arts in Computer Science **2014**
University of Virginia *Charlottesville, Virginia*

PUBLICATIONS

PolyFrame: A Retargetable Query-based Approach to Scaling Dataframes **2021**
Proceedings of the VLDB Endowment (PVLDB)

Scale-independent Data Analysis with Database-backed Dataframes: A Case Study **2021**
Workshop on Data Analytics and Machine Learning Made Simple @ EDBT

Scaling DNN-Based Video Analysis by Coarse-grained and Fine-grained Parallelism **2020**
IEEE International Conference on Multimedia and Expo (ICME)

AFrame: Extending DataFrames for Large-scale Modern Data Analysis **2019**
IEEE International Conference on Big Data (Big Data)

End-to-End Machine Learning with Apache AsterixDB **2018**
Workshop on Data Management for End-To-End Machine Learning @ SIGMOD

ABSTRACT OF THE DISSERTATION

A Retargetable Query-based Approach to Scaling Dataframes

By

Phanwadee Sinthong

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Professor Michael J. Carey, Chair

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 makings and applications. Scaling up analysis, possibly including the application of custom machine learning models, to large volumes of data, requires the utilization of distributed frameworks which can introduce serious technical challenges to data analysts and reduce their productivity.

In order to efficiently support the full Big Data analysis lifecycle without requiring extensive distributed systems knowledge, we extend data scientists' familiar tool, Pandas dataframe, to operate on managed data at scale. We introduce AFrame, a new scalable analysis package that integrates a Pandas-like user experience with data management systems to provide analysts with a familiar working environment while scaling out the evaluation of the analytical operations over a large data cluster to enable analysis on large-scale managed datasets.

There are four aspects involved in this dissertation: The first is constructing a new framework ("AFrame"). We have implemented AFrame on top of Apache AsterixDB by transparently converting dataframe operations to SQL++ queries. The second aspect is making AFrame more flexible for deployment with other composable query languages by retargeting AFrame's incremental query formation to other query-based database systems. The third aspect is creating a benchmark to evaluate our framework's performance. The fourth and final aspect is to demonstrate the feasibility and efficacy of our framework through a case study analysis.

Chapter 1

Introduction

In this era of big data, extracting useful patterns and intelligence for improved decision-making is becoming a standard requirement for many businesses. The growing interest in interpreting large volumes of user-generated content for purposes ranging from business advantages to societal insights motivates the development of data analytic tools. As the volume of data grows but little is known about the data, data scientists have to iteratively perform analyses over large volumes of data. As a result, various libraries and frameworks have been developed to ease the processing of big data. However, efficiently utilizing these tools requires distributed system and data management knowledge from data scientists who should only be focused on data modeling, selection of machine learning techniques, and data exploration, which in turn lower their productivity. To address these limitations, in this dissertation we propose a scalable data analytic framework that integrates a data scientists' familiar tool, dataframe, with data management capabilities. There are four aspects involved in this work.

First, constructing a scalable data analytics library, AFrame, that provides a Pandas-like dataframe interface on top of Apache AsterixDB [3]. AFrame leverages distributed data storage and management in order to accommodate the rapid rate and volume at which modern data arrives. AFrame translates dataframe operations and incrementally constructs SQL++ queries. It leverages lazy evaluation and only sends the queries for execution when the results are required. This design decision allows AFrame to take advantage of AsterixDB's data management capabilities and optimizations to efficiently interact, transform, and analyze large amounts of data efficiently, enabling much more interactive data manipulation.

The second aspect is to make AFrame more flexible to enable a wider audience in the data science community to leverage its scale-independent data analysis and data management capabilities. This is achievable by abstracting AFrame's existing 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 easily extensible template for supporting other composable query languages, thus allowing AFrame to operate against other query-based database systems. As a proof-of-concept, we have applied our language rewrite rules to four different query languages SQL++ [35], SQL [34], MongoDB's Query Language [23], and Cypher [40] to retarget AFrame to work against AsterixDB [3], PostgreSQL [25], MongoDB [22], and Neo4j [24, 52] respectively.

The third aspect of this dissertation is a distributed dataframe benchmark for general data analytics. The performance of a big data system is greatly affected by the characteristics 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 [10, 36, 38, 44, 50] 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.

Fourth, demonstrating the usability and performance benefits of using PolyFrame to perform end-to-end data analysis on real datasets. The case study analysis is also helpful in identifying the current limitations and potential future optimization opportunities of PolyFrame. We present a performance comparison in each stage of an end-to-end exploratory data analysis and highlight the benefits of database-backed dataframes in comparison to a Pandas dataframe baseline.

The rest of this dissertation is organized as follows: Chapter 2 explains the background for this work. Chapter 3 discusses the related work. Chapter 4 presents our library (AFrame), outlines its architecture, and describes a user model. Chapter 5 presents our Dataframe benchmark and details AFrame experiments. Chapter 6 describes the design of the rearchitected version of AFrame, PolyFrame. Chapter 7 illustrates the usability of PolyFrame through two case studies. Finally, Chapter 8 concludes this dissertation and discusses potential future research directions.

Chapter 2

Background

2.1 Pandas

Pandas [15] is a Python data analytics framework that reads data from various file formats and creates a Python object, a DataFrame, with rows and columns similar to Excel. Pandas works with Python machine learning libraries such as Scikit-Learn [53] and it can also be integrated with scientific visualization tools such as Jupyter notebooks [43]. 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, as its strength has typically been for in-memory computation on a single machine. In addition, Pandas' internal data representation is inefficient; as Wes McKinney (Pandas' creator) stated in [2] 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 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 [55]. 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 [46]. 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.

2.3 Apache AsterixDB

Apache AsterixDB [3, 33] 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++ [35], a highly expressive semi-structured query language for users that are familiar with SQL, to explore stored NoSQL data.

Figure 2.1 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 [3], [28]). Figure 2.1 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);
```

Figure 2.1: SQL++ queries

Chapter 3

Related Work

Limitations in Pandas' dataframe facility has lead to the recent development of various scalable frameworks. In this chapter, we categorize the existing systems into three main categories which are big data platforms, scalable dataframe technology, and polystore systems. Here we briefly summarize each of the approaches and describe the relationship to our work.

3.1 Big Data Platforms

Here we consider frameworks that can operate on distributed data.

3.1.1 Apache Spark

Apache Spark [58] is a general-purpose cluster computing system that provides in-memory parallel computation on a cluster with scalability and fault tolerance. SparkSQL [31] is a module to simplify users' interactions with structured data. SparkSQL integrates relational processing with Spark's functional programming. MLlib [47], which is built on top of Spark, provides the capability of constructing and running machine learning models on large-scale datasets. However, Spark does not provide data management and it requires the installation and configuration tuning of a distributed file system like HDFS.

3.1.2 Hive

Apache Hive [4] 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 [7] and Apache Spark [6] 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.

3.2 Scalable Dataframes

These libraries try to deliver a Pandas-like experience and scale operations onto large volumes of file-based data using different methods. They either provide a similar Pandas-like interface on a distributed compute engine, execute several Pandas dataFrames in parallel, or use memory mapping to optimize the computation and speed up data access. More recent efforts have been to develop a Pandas-like interface directly on top of database systems where large volumes of data are stored. We categorize some of the well-known scalable dataframe libraries into three categories: parallel execution frameworks, interfaces to distributed compute engines, and interfaces to database systems. Here we mention some of the well-known libraries in each category.

3.2.1 Parallel Execution Frameworks

Frameworks in this category address the scalability limitations in Pandas using parallel execution strategies. They execute multiple Pandas dataframes in parallel. Examples are Dask and Modin. Dask[9] is a framework for scaling Python data analytic libraries like Pandas, Scikit-learn [53], and NumPy [14] to run in a distributed environment. Dask offers a Pandas dataframe-based implementation that scales to multiple machines by segmenting large datasets into multiple small Pandas dataframes and processing them in parallel. Modin [13] scales Pandas by distributing the data and operations using a shared-memory framework called Ray [49]. Modin is similar to Dask in the sense that it uses Pandas dataframes internally, but its bottom-up scheduling policy where each worker submits tasks to be executed is different from Dask's which uses a centralized scheduler. Modin also uses eager evaluation, making it more like Pandas. Modin is now being extended to work on Dask in a cluster environment and support for other distributed execution engines and other custom dataframe interfaces are also being considered. However, running Pandas dataframes internally means that these frameworks lack query optimization, which is important when operating on datasets at scale.

3.2.2 Distributed Compute Engines

Spark also provides DataFrames [30], an API built on top of Spark SQL [31] for distributed structured data manipulation. However, Spark's DataFrame syntax is different from Pandas' in several respects. As a result, Koalas [21], 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 [26] to create DataFrames, but it uses the database systems mainly as a data store and continues to process most operations in its own runtime environment. 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.

3.2.3 Scaling Dataframes with Databases

Efforts to scale dataframes have started gaining traction in the database community. Recently, Jindal et. al introduced Magpie [42], 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 [20], 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 [39], 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.

3.3 Polystores

Polystore systems (e.g., BigDAWG, BigIntegrator, and Polybase) provide integrated and transparent access to multiple data stores with heterogeneous storage engines through a common language. In [51], polystores are categorized into three different groups based on the level of coupling with the underlying data stores: loosely coupled, tightly coupled, and hybrid systems. These systems 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, and delivers the results to the user.

3.4 Relationship to This Work

Our work can be categorized as a database-backed dataframe library. We try to scale Pandas dataframes by translating its operations into database queries. Our library is different from the other libraries in this category as it does not maintain an intermediate representation for the purpose of pre-optimizing the queries or selecting between different backends. We focus 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. The differences between our library and polystore systems are their interactions and intended usages. We provide a common language of dataframe operations for users to interact with a query-based database system of their choosing where their data is stored. Our library does not aim to communicate or orchestrate queries across multiple data stores.

Chapter 4

AFrame

4.1 Introduction

This chapter presents the first version of our library that provides a ‘scale-independent’ user experience when moving from a local exploratory data analysis environment to a large-scale distributed data environment. We present AFrame, an Apache AsterixDB [29] based extension of dataframe. AFrame is a data exploration library that provides a Pandas-like dataframe [45] 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.

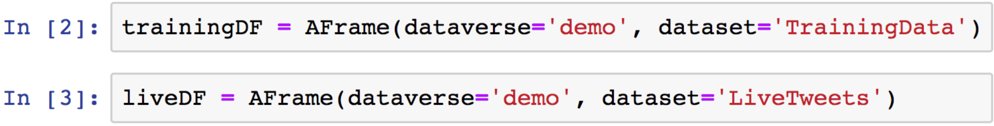
The remainder of this chapter is organized as follows: Section 4.2 describes the AFrame’s user model and illustrates its basic functionalities. Section 4.3 summarizes the underlying architecture of AFrame. Section 4.4 discusses AFrame’s incremental query formation mechanism. Section 4.5 concludes the chapter.

4.2 User Model

Our goal in the AFrame project is to create a unified system that can efficiently support all of the various stages [48] 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. Here we illustrate AFrame's basic functionality and its user model through a small running example that shows how to perform a simple sentiment analysis on ever-growing Twitter data.

4.2.1 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 webservice using its RESTful API. Figure 4.1 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 Chapter 2 in Figure 2.1. 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 2.1 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')
```

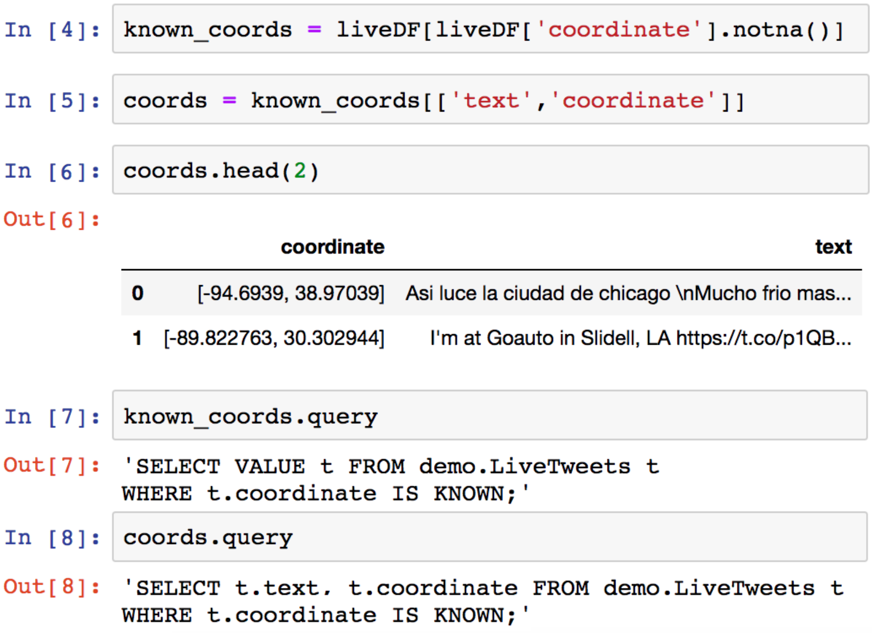
Figure 4.1: Initializing AFrame Objects

4.2.2 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 4.2 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 4.1. 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.

Being in a relatively 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 (both elementwise and tablewise), 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...

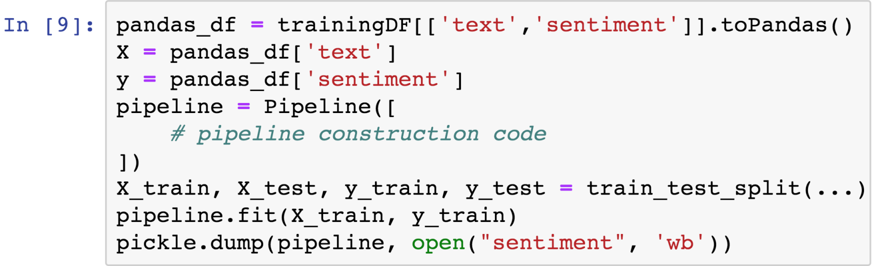
```
In [7]: known_coords.query

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;'
```

Figure 4.2: DataFrame expressions and underlying queries

4.2.3 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 4.3 illustrates a sentiment classifier training session using Python, Scikit-Learn [53], Pandas, and AFrame. It trains a classifier on the training dataset from Figure 4.1. This is a dataset, publicly available on Kaggle [12], containing Twitter posts related to users' experiences with U.S. airlines released by CrowdFlower [8]. The dataset contains labeled tweet sentiments which are positive, negative, and neutral. The first step in Figure 4.3 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.



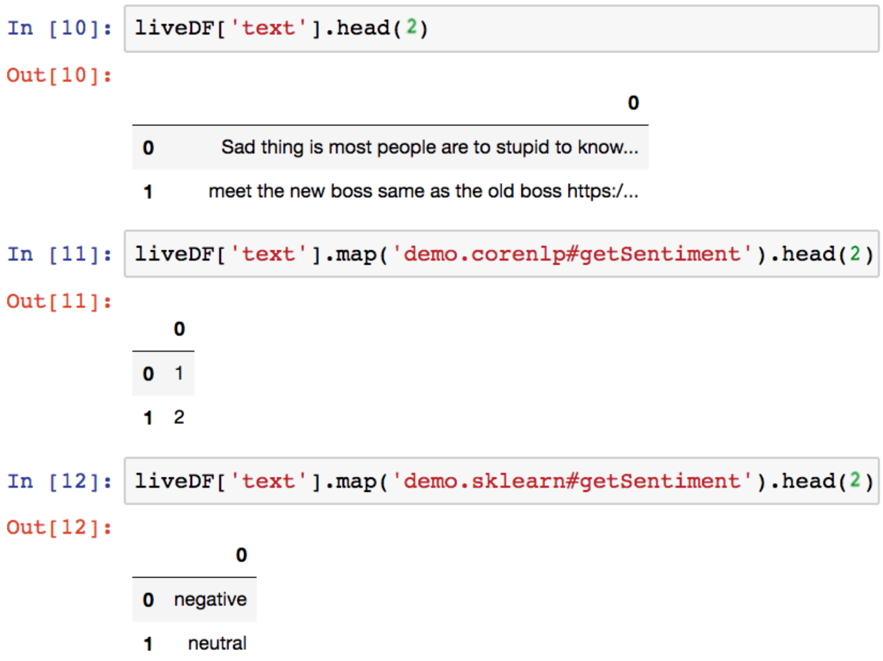
```
In [9]: pandas_df = trainingDF[['text','sentiment']].toPandas()
        X = pandas_df['text']
        y = pandas_df['sentiment']
        pipeline = Pipeline([
            # pipeline construction code
        ])
        X_train, X_test, y_train, y_test = train_test_split(...)
        pipeline.fit(X_train,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

Figure 4.3: Training a Scikit-Learn Pipeline

In Figure 4.4, we show sample code for applying machine learning models in AFrame. We first apply a pre-trained model (from Stanford CoreNLP) and then apply our custom Scikit-Learn sentiment analysis model (created in Figure 4.3) 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 4.1. Input 11 applies the pre-trained Stanford CoreNLP sentiment analysis model [57] 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 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.

[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.

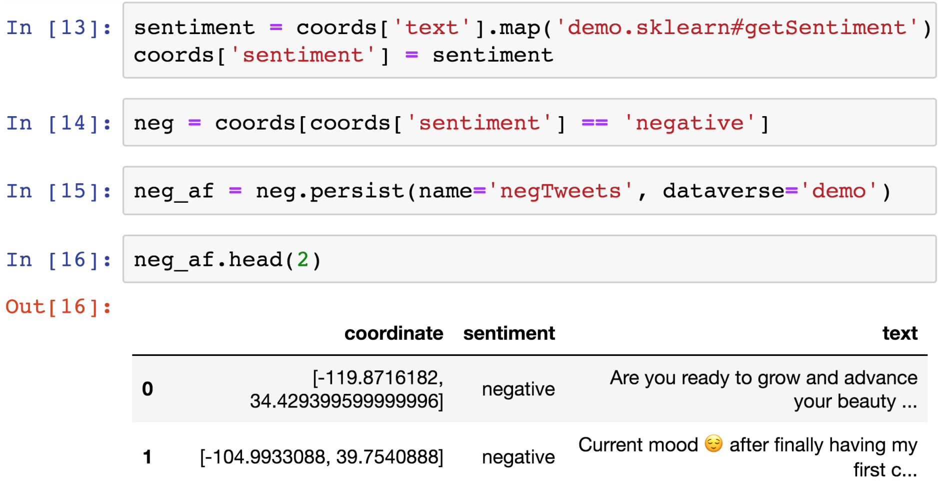


```
In [10]: liveDF['text'].head(2)
Out[10]:
                                               0
     0       Sad thing is most people are to stupid to know...
     1     meet the new boss same as the old boss https:/...

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
```

Figure 4.4: Applying CoreNLP and Scikit-Learn models

4.2.4 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 4.5 shows an example of how to apply the previously-constructed Scikit-Learn sentiment function to the ‘text’ field of a queried subset (coords) of the live Twitter records resulting from the operations in Figure 4.2. 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 4.5. 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 data re-loading or 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...

Figure 4.5: Persist Sentiment Analysis Results

4.3 System Architecture

Having seen its user model, we now turn to AFrame's system architecture. Figure 4.6 displays an overview of AFrame's internal working mechanism. When an AFrame object is initialized, an initial SQL++ query is embedded as part of the object's attributes. The red dotted box highlights the incremental query formation process supported by AFrame. Inspired partly by Spark, there are two types of operations in AFrame: transformation and action. Transformations are operations that result in a new AFrame object with a new underlying query resulted from the incremental query formation process. These operations do not trigger query execution. Actions are operations that request for result visualization. These operations trigger a query execution. The underlying query will be sent over to AsterixDB for execution.

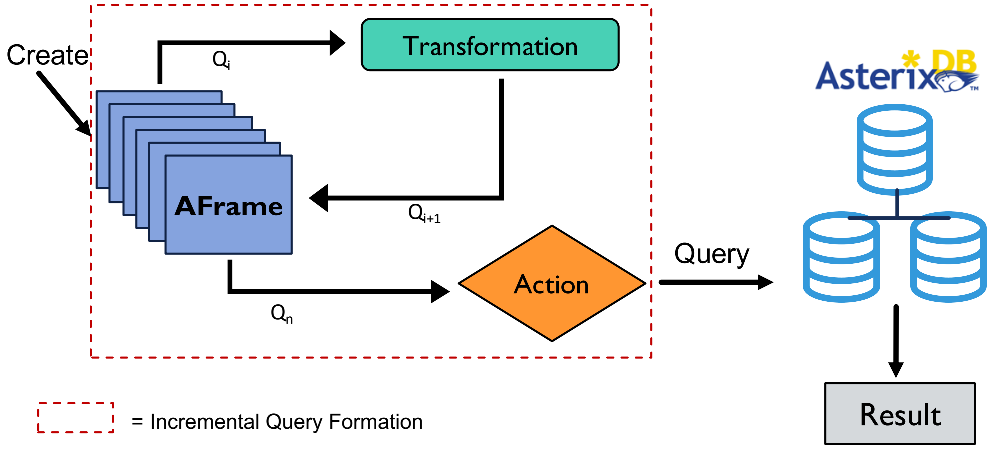


Figure 4.6: Initializing AFrame Objects

4.4 Incremental Query Formation

Internally, AFrame 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 4.7 shows an example of six SQL++ queries generated as a result of Pandas DataFrame operations. The Dataframe operations are listed on top of each AFrame object (the numbered rectangles). The corresponding SQL++ queries are listed below the objects. The first AFrame 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, AFrame 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, AFrame 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.

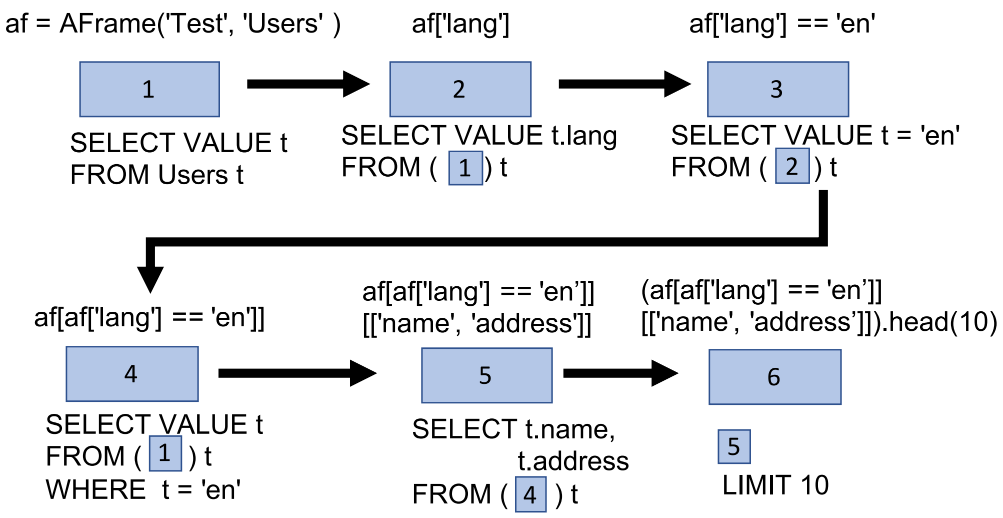


Figure 4.7: Incremental Query Formation

4.5 Conclusion

In this chapter, 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 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. We also described AFrame's incremental query formulation architecture, which enables AFrame to 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.

Chapter 5

A Dataframe Benchmark

5.1 Introduction

To our knowledge, there is no standard benchmark for evaluating dataframe libraries. In order to evaluate our library and compare its performance to that of other distributed DataFrame libraries, we have constructed a preliminary DataFrame benchmark. Inspired by the early Wisconsin Benchmark [37] 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. 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. 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.

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.

The rest of this chapter is organized as follows: Section 5.2 describes our benchmark dataset. Section 5.3 details a set of our benchmark queries. Section 5.4 presents the AFrame's benchmark results. Section 5.5 concludes the chapter.

5.2 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 dataframe 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 Figure 5.1. 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 [41] 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 [5] 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 5.1: Scalable Wisconsin benchmark: attributes [37]

5.3 Benchmark Queries

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()])

Table 5.2: Dataframe Benchmark Operations (df, df2 = DataFrame objects, x,y,z = variables representing random values within an attribute's range)

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 5.2 summarizes the details of our initial DataFrame benchmark expressions. 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. 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 5.2. It is important to note here that when we first created our benchmark we had 12 benchmark expressions. Therefore, the first version of our framework (AFrame) was evaluated using 12 benchmark expressions. The thirteenth expression was added to evaluate PolyFrame on different database systems to compare their performance when handling null values. 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.

5.4 Comparisons with Other Dataframe Libraries

Here we present the environmental setups and results of running our benchmark on various dataframe libraries in both single node and multi-node settings.

5.4.1 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 section.

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, 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 described in section A.2 of the appendix. AFrame's translated SQL++ queries for all benchmark expressions are presented in section C.2 of the appendix. 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, it is worth mentioning that the project has started to implement support for large datasets using disk as an overflow for in-memory DataFrames.

5.4.2 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 time and the DataFrame expression execution time). Each system's timing point is provided below.

- **Pandas & Pandas on Ray Timing**

```
# DataFrame creation time
df = pd.read_json(file_path)
# Expression-only time
df.head()
```

- **Spark Timing**

```
# DataFrame creation time
df = sparkContext.read.json(file_path)
# Expression-only time
df.head(5)
```

- **AFrame Timing**

```
# DataFrame creation time
df = AFrame(dataverse, dataset)
# Expression-only time
df.head()
```

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.

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 6.2 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 5.3: Dataset Summary (mil = million)

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 5.4, 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 5.4: 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 5.5, 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

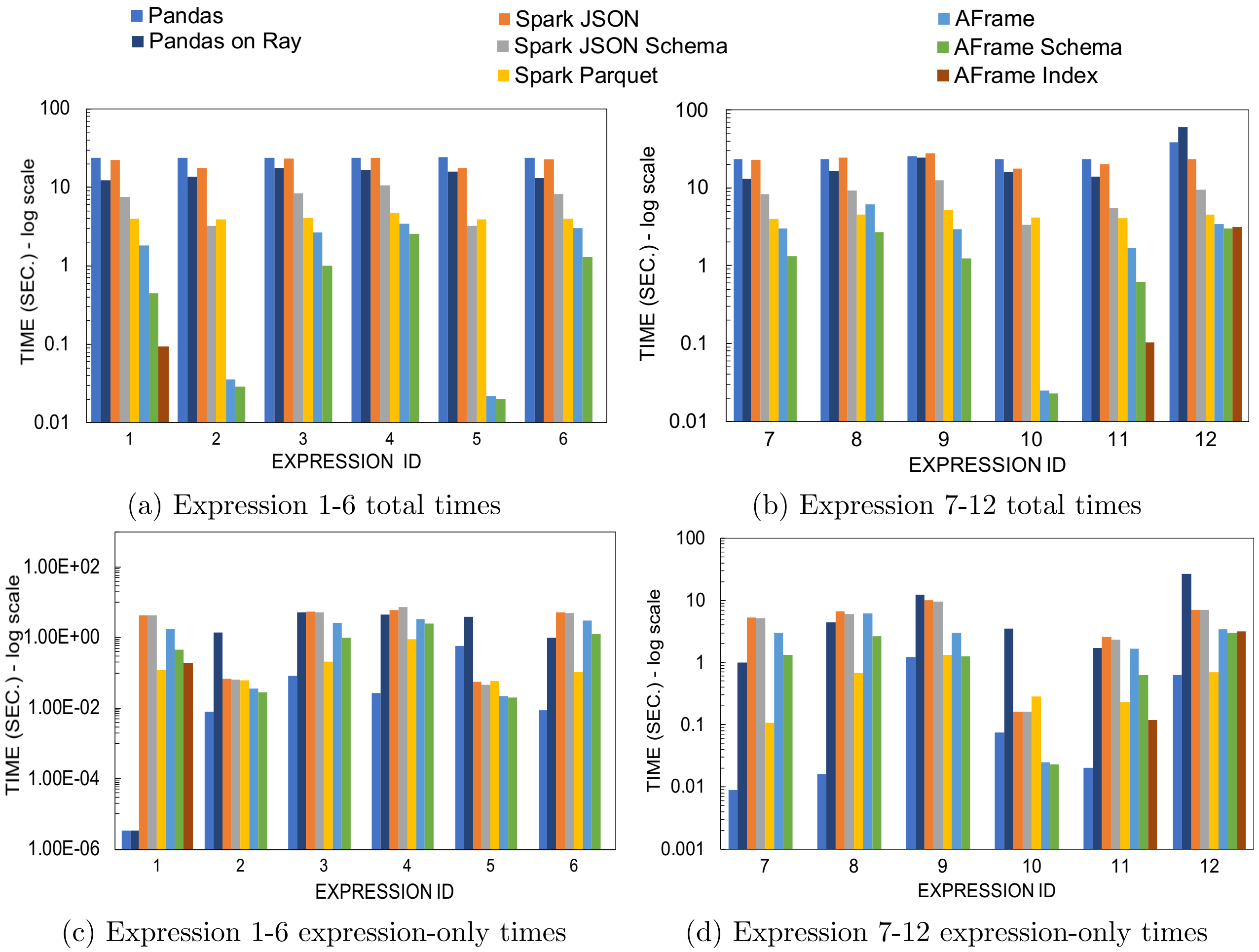
Table 5.5: Scaleup Experiment Setup

5.4.3 Preliminary 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 5.1. 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. 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).

The XS results are presented in Figure 5.1. Figures 5.1a and 5.1b show the total run times (including DataFrame creation). Figure 5.1a displays expression 1-6's results and Figure 5.1b displays expression 7-12's results. Figures 5.1c and 5.1d show the expression-only execution times. Figure 5.1c displays expression 1-6's results, and Figure 5.1d displays expression 7-12's results. The differences between the total times and expression-only times indicate that the DataFrame creation process can significantly impact system performance.

Pandas requires data to be loaded into memory before its operation evaluations. Since it



(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

Figure 5.1: XS Results of Single Node Evaluation

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 5.1c and 5.1d, Pandas performed the best in 10 of the 12 expressions. The two cases where Pandas was not the fastest were operations 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 5.1c and 5.1d, 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 performance results for the datasets with indexes are an order of magnitude faster, as seen for Expressions 1, 11, and 12. 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 5.1d).

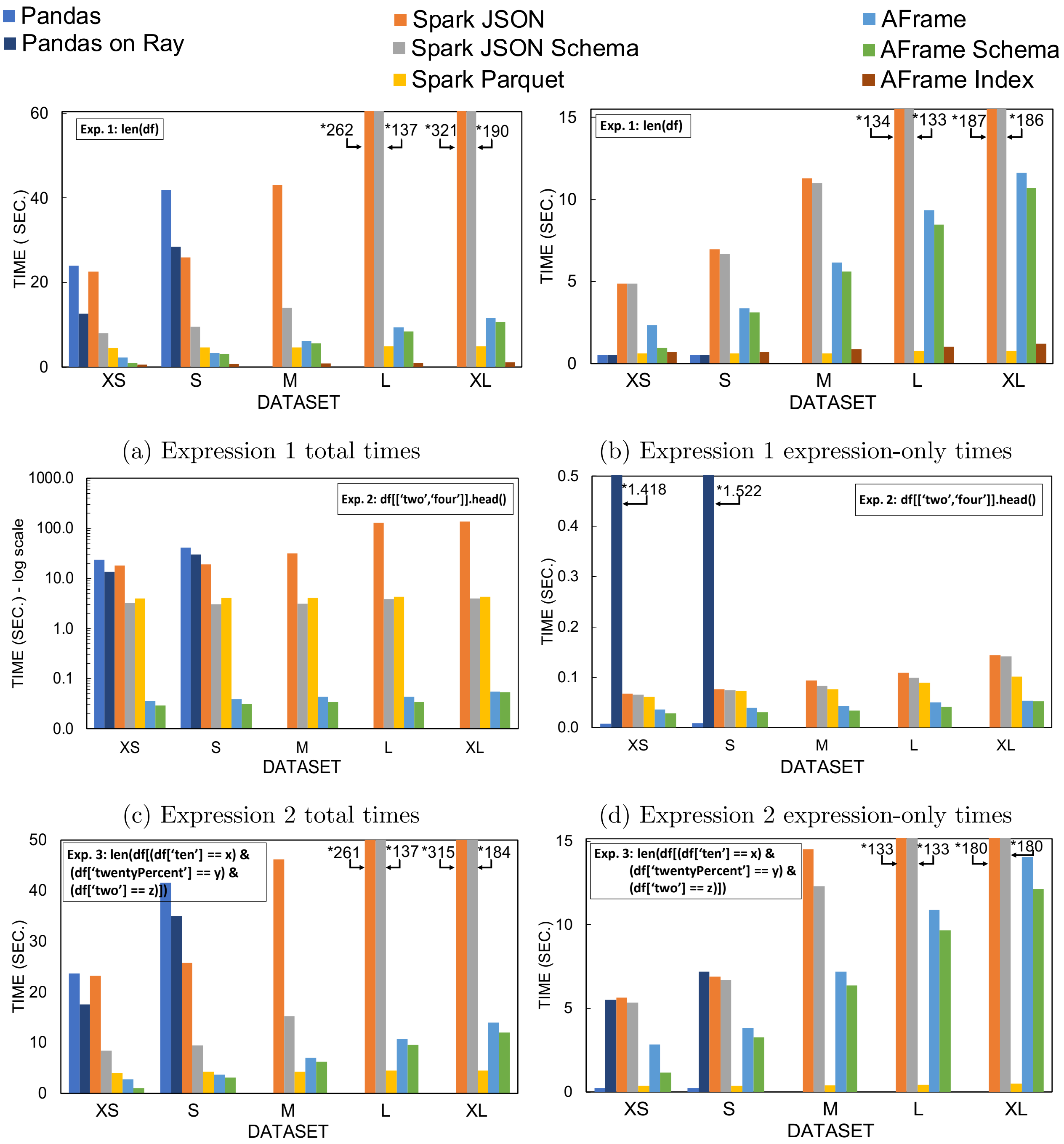
5.4.4 Single-node Results

After the first evaluation round, we evaluated the systems' single-node data scalability by running each expression on all five different data sizes. As we can see from Figures 5.2 - 5.5

Pandas and Pandas on Ray 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 (e.g., Figures 5.2a, 5.2b, 5.2e, 5.2f, 5.3a,5.3b, etc.). These results are much slower than the other datasets 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



(a) Expression 1 total times

(b) Expression 1 expression-only times

(c) Expression 2 total times

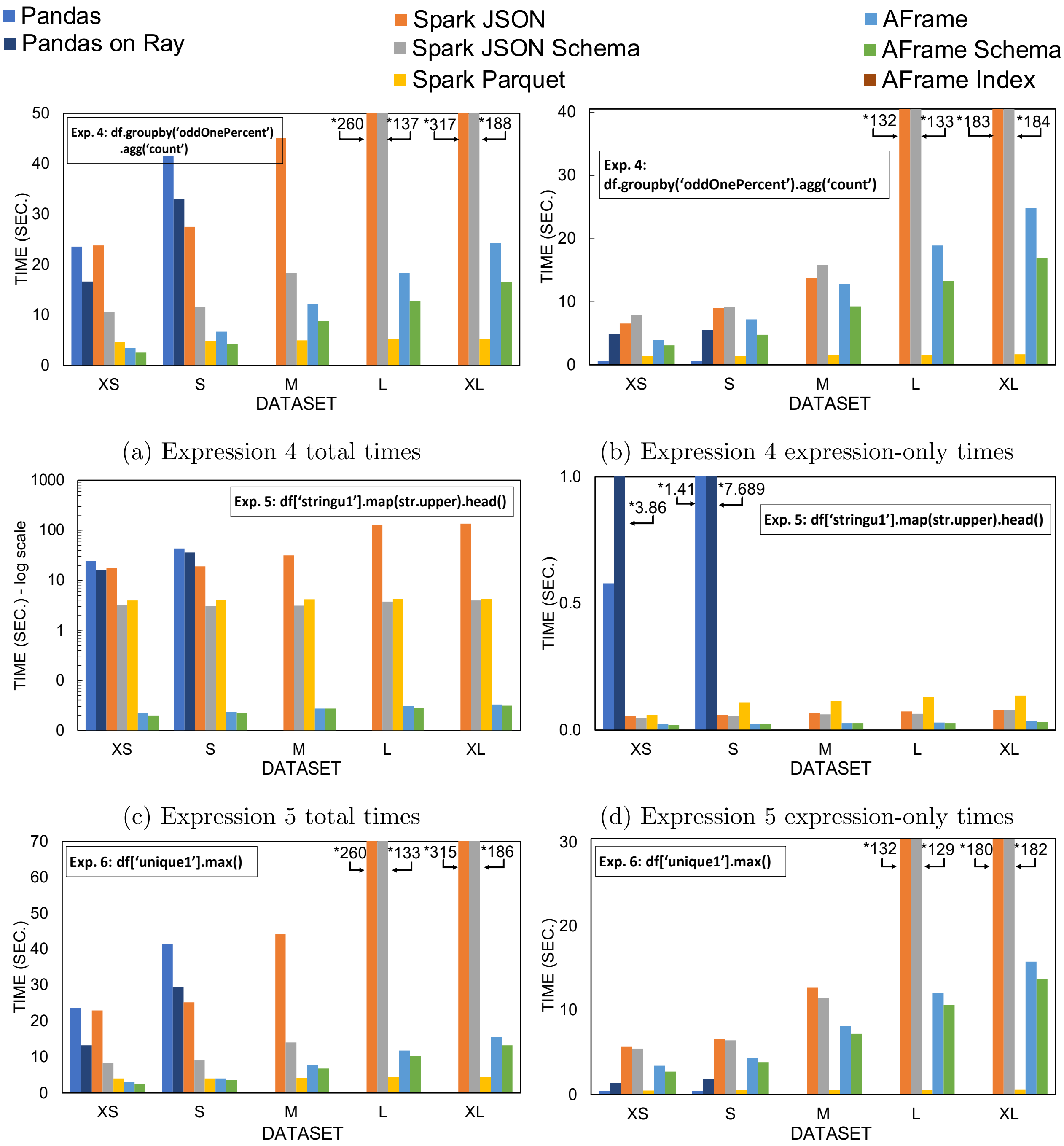
(d) Expression 2 expression-only times

(e) Expression 3 total times

(f) Expression 3 expression-only times

Figure 5.2: Single Node Evaluation: Expression 1-3 Results (* = value where the bar ends)

factor to keep in mind is that even the Parquet-based DataFrame requires some DataFrame creation overhead. Figure 5.2e 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



(a) Expression 4 total times

(b) Expression 4 expression-only times

(c) Expression 5 total times

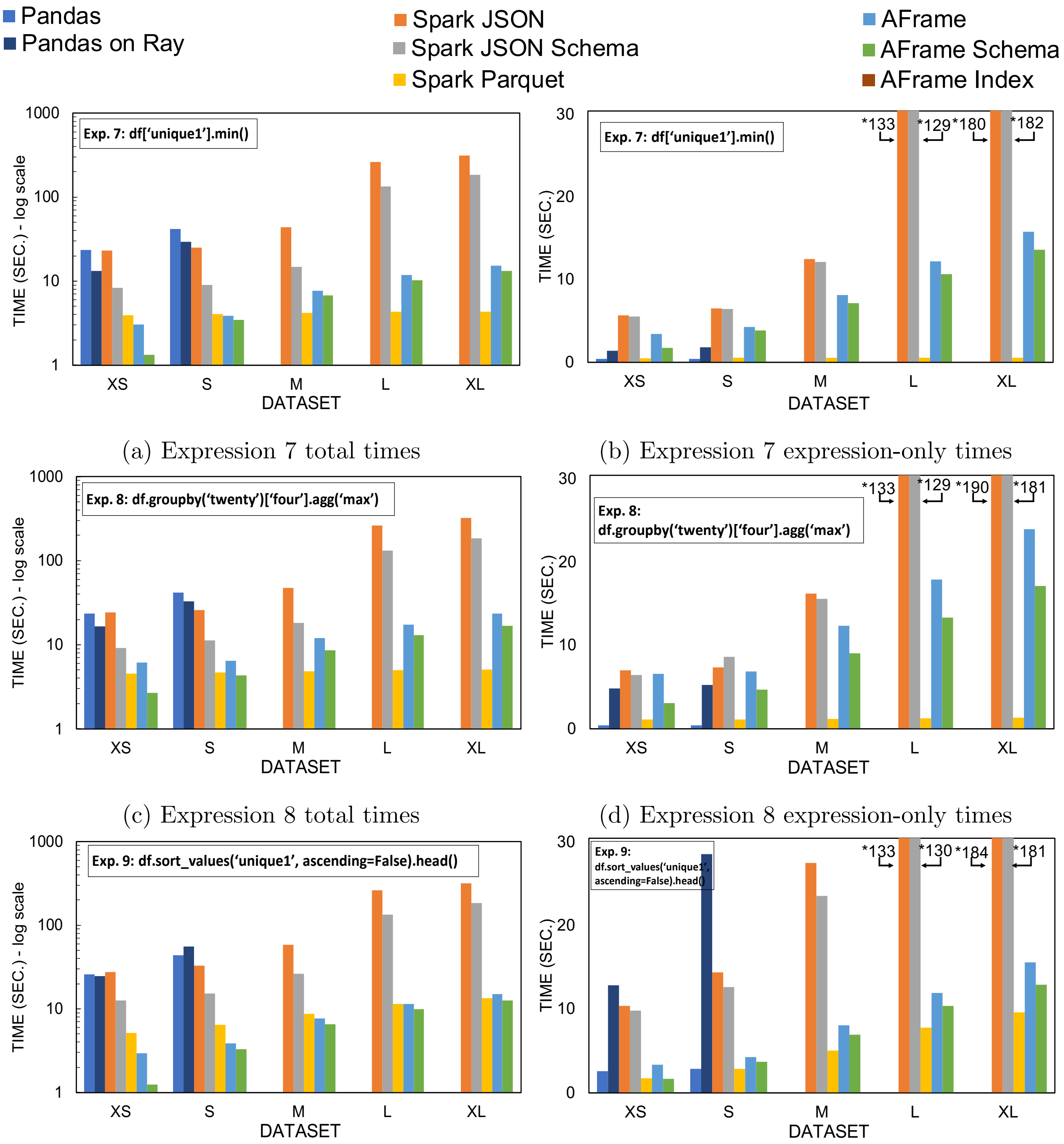
(d) Expression 5 expression-only times

(e) Expression 6 total times

(f) Expression 6 expression-only times

Figure 5.3: Single Node Evaluation: Expression 4-6 Results (* = value where the bar ends)

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



(a) Expression 7 total times

(b) Expression 7 expression-only times

(c) Expression 8 total times

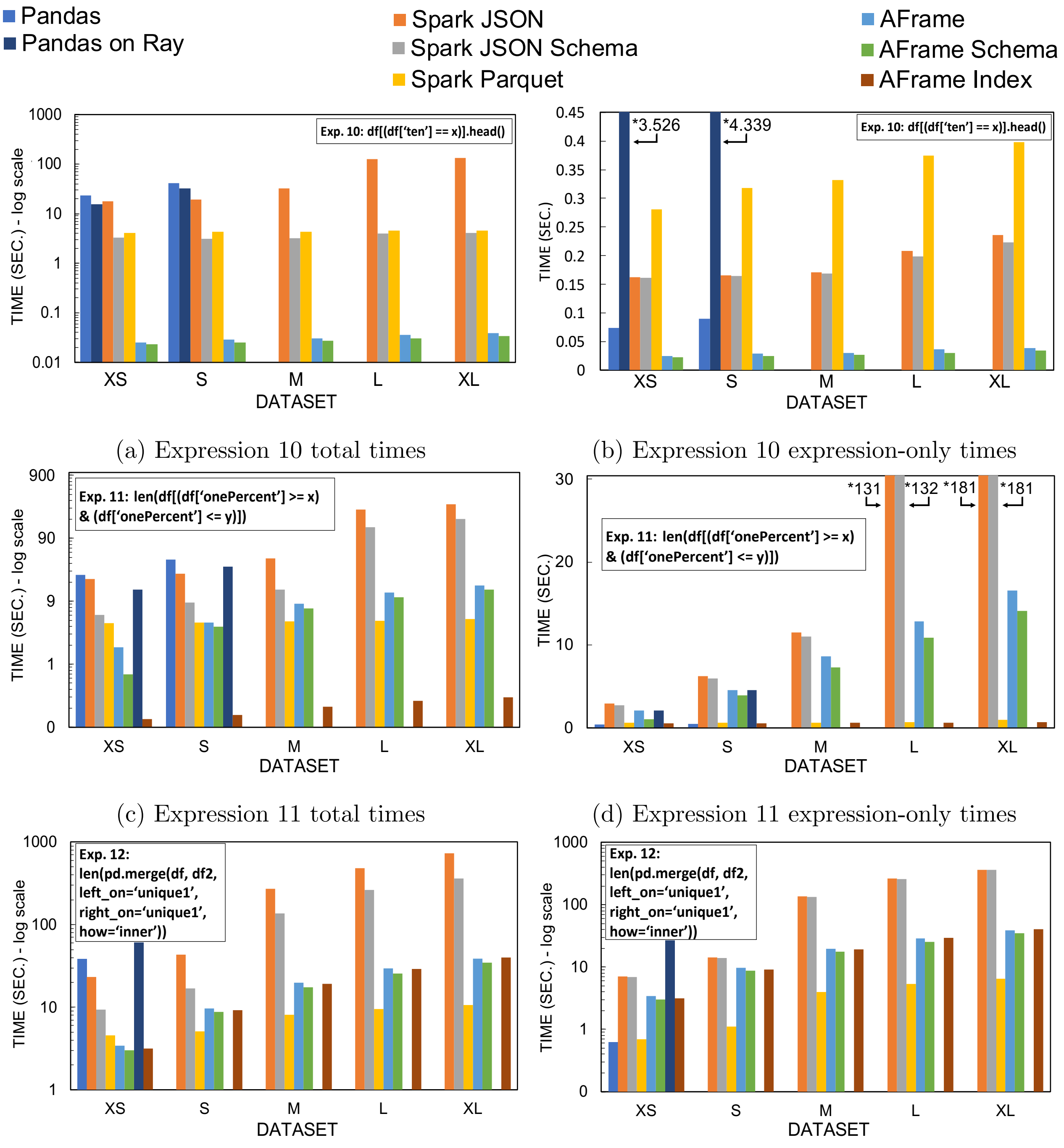
(d) Expression 8 expression-only times

(e) Expression 9 total times

(f) Expression 9 expression-only times

Figure 5.4: Single Node Evaluation: Expression 7-9 Results (* = value where the bar ends)

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 expressions 5 and 10, as seen in Figures 5.3d and 5.5b, Spark's JSON-based DataFrame



(a) Expression 10 total times

(b) Expression 10 expression-only times

(c) Expression 11 total times

(d) Expression 11 expression-only times

(e) Expression 12 total times

(f) Expression 12 expression-only times

Figure 5.5: Single Node Evaluation: Expression 10-12 Results (* = value where the bar ends)

performed significantly better than its Parquet-based (columnar) DataFrame. Even in the case that includes the DataFrame creation time, shown in Figures 5.3c and 5.5a, Spark's

JSON-based DataFrame with a pre-defined schema was faster than Parquet for all data sizes for expressions 5 and 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. Similary, in expression 11 as shown in Figures 5.5c and 5.5d, AFrame with an index on the range attribute was the fastest in the total time case and was competitive in the expression only case as well.

AFrame also benefits from having indexes on the join attributes (Expression 12), as shown earlier in Figure 5.1b; 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. For this expression, both datasets are identical in content and size. As we can see from Figures 5.5e and 5.5f, Pandas and Pandas on Ray even failed to complete DataFrame creation on dataset S as they had to load twice as much data. It is important to note that equality-based joins in AsterixDB default to use hybrid hash join algorithm (as it has good cost characteristics when joining large datasets). For AFrames without index (labeled AFrame and AFrame Schema), the join method used in expression 12 was a hash join algorithm while AFrame Index benefited from index nested-loop join. Hash joins are more efficient for large datasets, which is the reason why we started to see AFrame outperforming AFrame Index on dataset XL as probing a hash table by scanning the dataset in its entirety once could be faster than performing too many index lookups and traversing a b-tree index.

AFrame was faster than Spark's JSON-based DataFrames in most of the test cases in Figure 5.1 and continued to be so as shown in Figures 5.2 - 5.5 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 5.5a and 5.5b), AFrame without indexes performed better than Spark Parquet and were the best for both the expression-only and total run time evaluations.

5.4.5 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.

Speedup Results

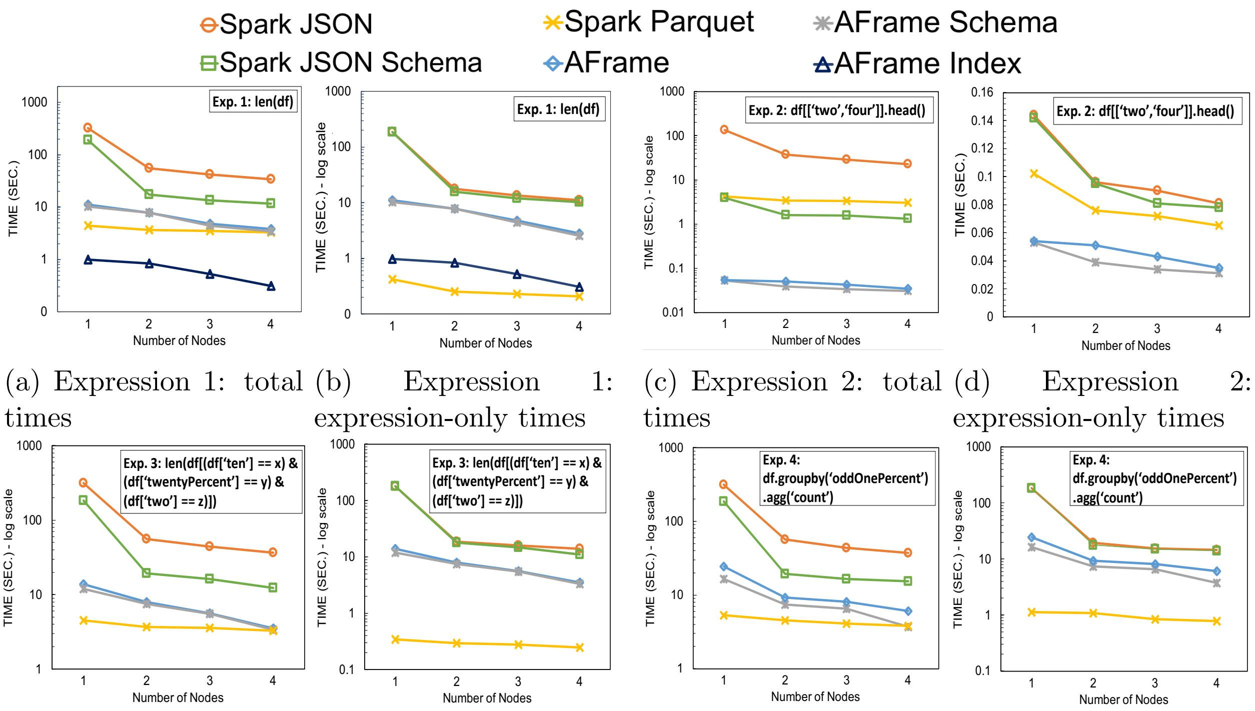
The results for both Spark and AFrame are consistent with their single-node results in terms of their performance rankings. Both systems processed the tasks faster when increasing the number of processors while maintaining the same data size. Spark's performance improved drastically when the distributed data begin to fit in memory in the case of JSON DataFrames. Figures 5.6 and 5.7 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. This is especially more visible in the total run time case. 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 (Figures 5.6a and 5.6b), 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 5.7l across different numbers of nodes. AFrame with and without schema are the fastest in both expression-only (5.7l) and total time (Figures 5.7k) for this expression. However, for expression 11 (Figures 5.7m and 5.7n), AFrame with an index on the range attribute was the fastest in both expression-only and total time evaluations because Spark Parquet incurred certain DataFrame creation overheads while AFrame translated this expression into a query that was executed as an index-only query on AsterixDB.

For expression 12 (Figures 5.7o and 5.7p), the similar behavior from the single node evaluation is also visible here. AFrame was faster than Spark JSON-based DataFrames on all cluster sizes but it was slower than Spark Parquet. AFrame and AFrame Schema display results of hash join on the XL dataset (10 GB JSON data) with increasing number of processing nodes while AFrame Index displays results of a broadcast-based index nested-loop join.



(a) Expression 1: total times (b) Expression 1: expression-only times (c) Expression 2: total times (d) Expression 2: expression-only times

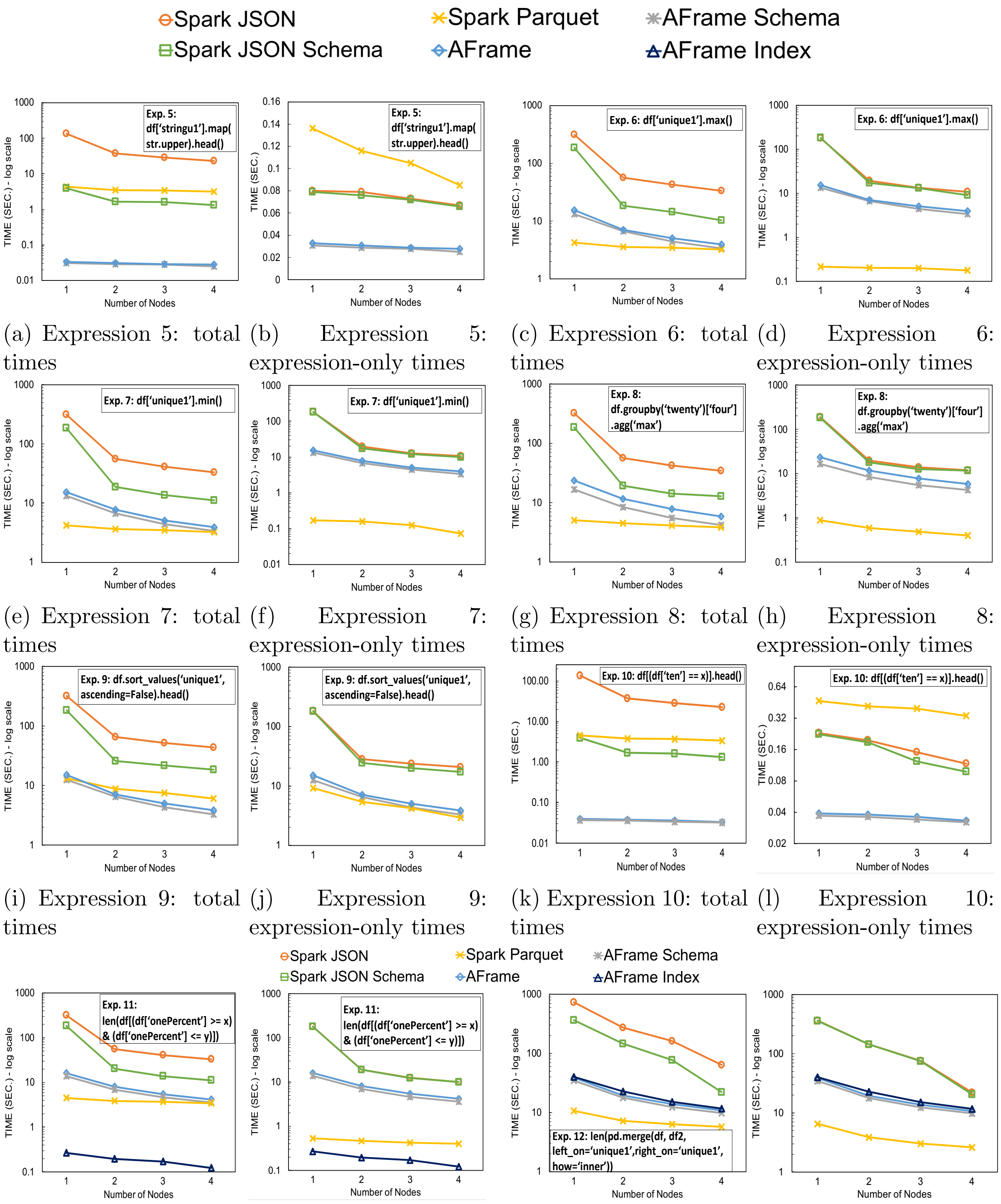
(e) Expression 3: total times (f) Expression 3: expression-only times (g) Expression 4: total times (h) Expression 4: expression-only times

Figure 5.6: Multi-Node Speedup Evaluation Results

Scaleup Results

In Figures 5.8 and 5.9, 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 (e.g., expressions 6 and 7) and was consistently competitive, but it also incurred an overhead for DataFrame creation. However, for row-based expressions (e.g., expressions 5 and 10), AFrame continued to follow the same trend from the single node case with the XL dataset, outperforming Spark Parquet.

As we saw in Figure 5.8, in the total time evaluations, by providing the 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. Expressions 2 (Figure 5.8c), 5 (Figure 5.8i), and 10 (Figure 5.9c) only sample a few records from a large dataset, which causes the schema



(a) Expression 5: total times

(b) Expression 5: expression-only times

(c) Expression 6: total times

(d) Expression 6: expression-only times

(e) Expression 7: total times

(f) Expression 7: expression-only times

(g) Expression 8: total times

(h) Expression 8: expression-only times

(i) Expression 9: total times

(j) Expression 9: expression-only times

(k) Expression 10: total times

(l) Expression 10: expression-only times

(m) Expression 11: total times

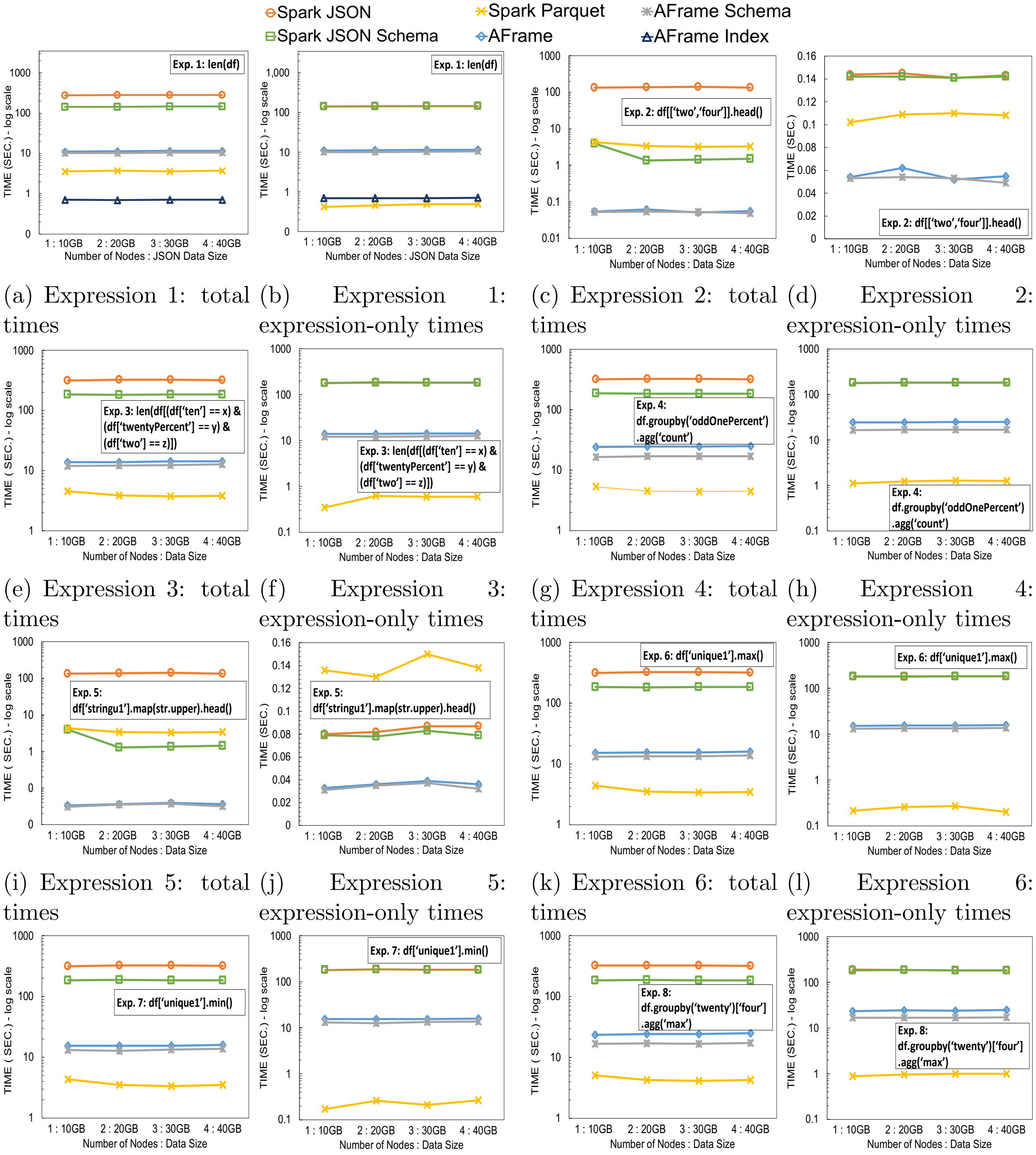
(n) Expression 11: expression-only times

(o) Expression 12: total times

(p) Expression 12: expression-only times

Figure 5.7: Multi-Node Speedup Evaluation Results (continued)

inference time to otherwise dominate the actual expression execution time.



(a) Expression 1: total times

(b) Expression 1: expression-only times

(c) Expression 2: total times

(d) Expression 2: expression-only times

(e) Expression 3: total times

(f) Expression 3: expression-only times

(g) Expression 4: total times

(h) Expression 4: expression-only times

(i) Expression 5: total times

(j) Expression 5: expression-only times

(k) Expression 6: total times

(l) Expression 6: expression-only times

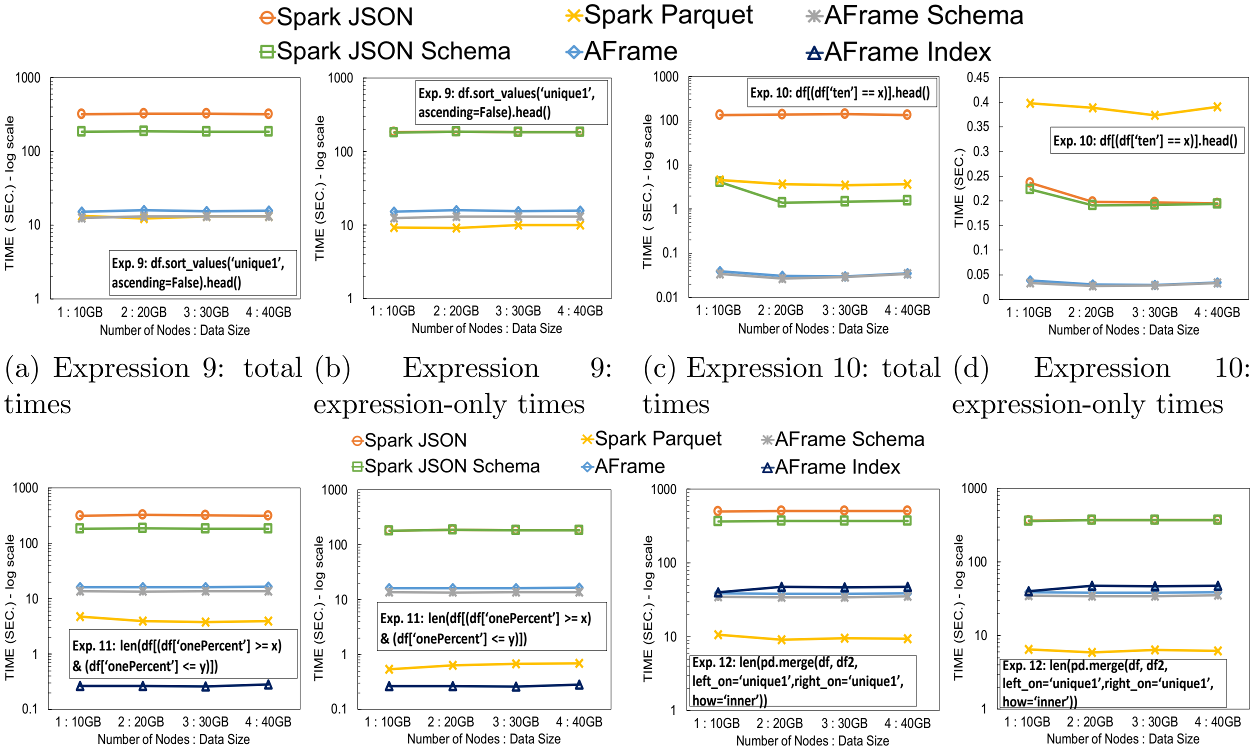
(m) Expression 7: total times

(n) Expression 7: expression-only times

(o) Expression 8: total times

(p) Expression 8: expression-only times

Figure 5.8: Multi-Node Scaleup Evaluation Results



(a) Expression 9: total times (b) Expression 9: expression-only times (c) Expression 10: total times (d) Expression 10: expression-only times

(e) Expression 11: total times (f) Expression 11: expression-only times (g) Expression 12: total times (h) Expression 12: expression-only times

Figure 5.9: Multi-Node Scaleup Evaluation Results (continued)

5.4.6 Result 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. Exploratory 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 [16], where the authors provide an explanation of the issues and give insights as to when the benefit of its work distribution will be significant. 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. It thus eliminates the disconnect between data-intensive analytical tools (e.g., Pandas) and database management systems. 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 (which arguably reflect the time from question to insight).

5.5 Conclusion

The DataFrame benchmark is preliminary work that has served a purpose by allowing us to evaluate the feasibility of AFrame for analytic operations and to compare its initial performance against other frameworks. 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 query 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.

Chapter 6

PolyFrame

6.1 Introduction

Depending on the nature of the analysis problem and environment at hand, 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. In this chapter, we describe a new design that retargets AFrame's incremental query formation to other query-based database systems as well, making it more flexible for deployment against other data management systems with composable query languages.

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++ [35], SQL [34], MongoDB Query Language [23], and Cypher [40], to retarget AFrame against AsterixDB [3], PostgreSQL [25], MongoDB [22], and Neo4j [24, 52] respectively. 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.
5. We support user-defined rewrites to allow users to specify their own custom rewrite rules to leverage a system's language-specific optimizations.

The rest of this chapter is organized as follow: Section 6.2 provides an overview of our retargetable query-based design. Sections 6.3 and 6.4 discuss the details of our language rewrite rules along with examples. Section 6.5 presents the experimental evaluation of PolyFrame. We conclude in Section 6.6.

6.2 System Architecture

Here we briefly describe PolyFrame's architecture 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.

In order to make AFrame language-independent, PolyFrame separates the language syntax from the original incremental query formation process. Figure 6.1 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.

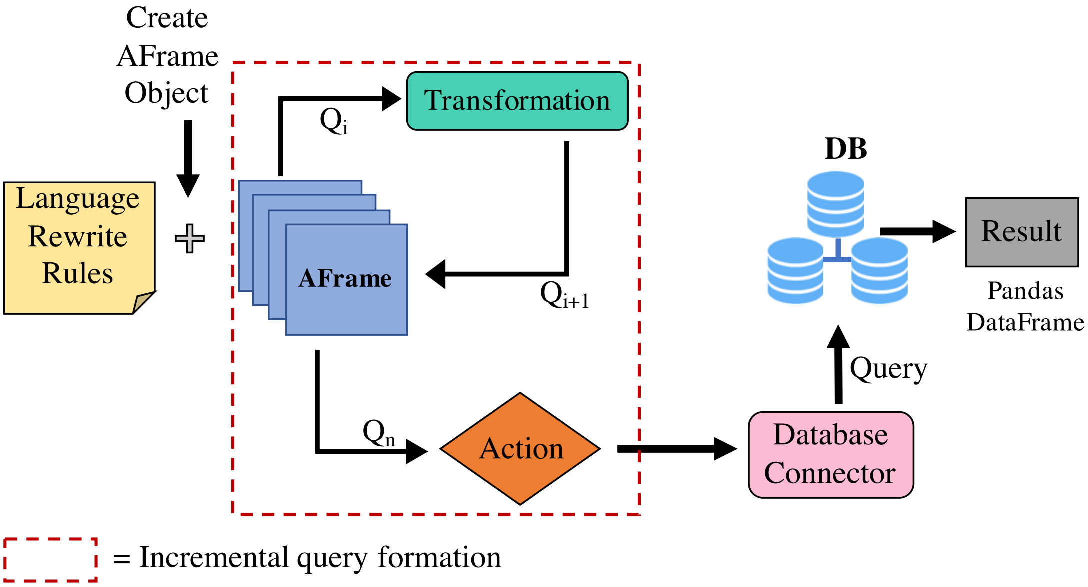


Figure 6.1: AFrame's New Architecture (PolyFrame)

6.3 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 6.2 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 6.3.2) to map each operation to one or multiple language-specific rules (described in Section 6.3.3). 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.

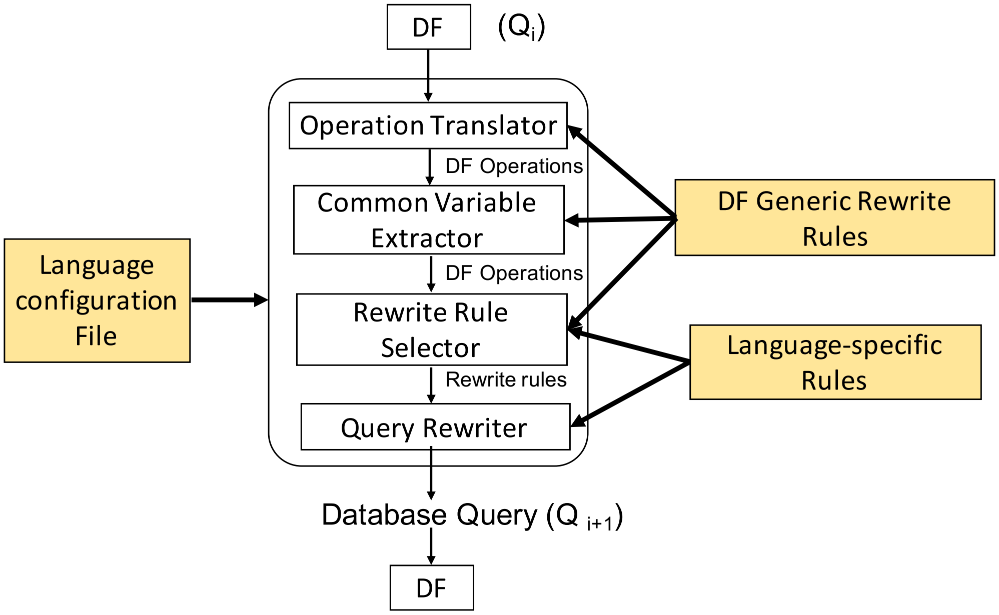


Figure 6.2: Flowchart of a query rewrite

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 6.3 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.

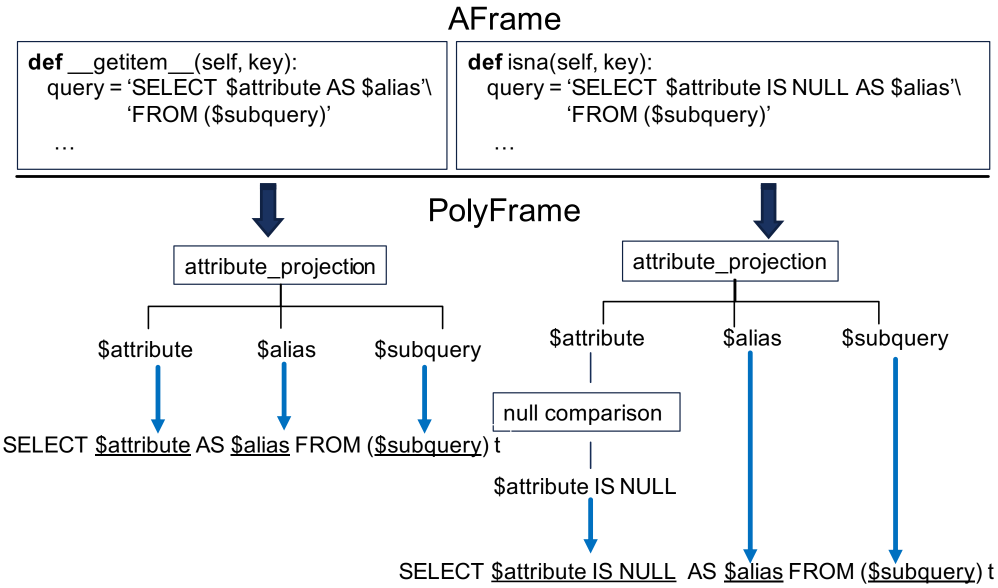

AFrame
def __getitem__(self, key):
query = 'SELECT $attribute AS $alias'\
'FROM ($subquery)'
...
def isna(self, key):
query = 'SELECT $attribute IS NULL AS $alias'\
'FROM ($subquery)'
...
PolyFrame
attribute_projection
$attribute
$alias
$subquery
SELECT $attribute AS $alias FROM ($subquery) t
attribute_projection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

Figure 6.3: AFrame vs. PolyFrame query construction

6.3.1 Supported Language Requirement

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, this latter requirement is already an important property of many database systems and is an ongoing area of research.

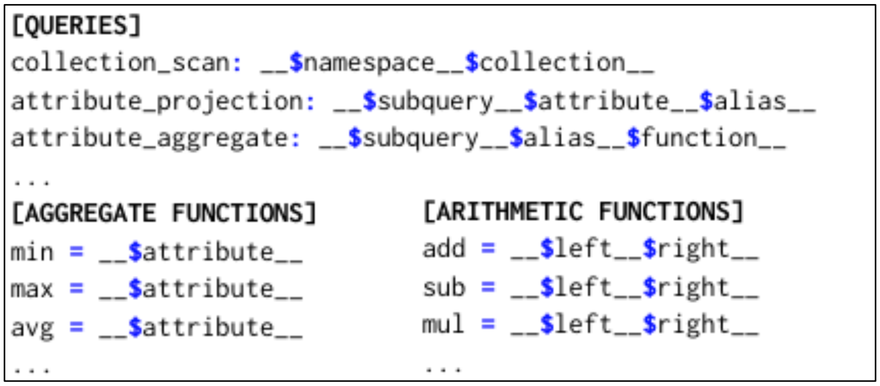
Currently, we support rewrites of many relational algebra operations in Pandas dataframes such as selection, projection, join, group by, aggregation, and sorting. Operations that require access to rows by indices are not supported because row ordering is not widely enforced in database systems. A challenge 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.

6.3.2 Generic Rewrite 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 6.5 together to form a query that asks for aggregate values (min, max, average, count, and standard deviation) of specified attributes. We chain together operations 3-7 according to a pre-defined language-specific attribute rewrite rule and use them to rewrite operation 1 and 2. Thus, instead of creating a rule for each function of Pandas Dataframe, these generic rules allow PolyFrame to efficiently utilize common components to form more complex queries that perform the desired function.

6.3.3 Language-specific Rewrite 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 6.4. 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 6.4: Configuration Template Overview

Figure 6.5 shows a few implementation examples of the language-specific rewrite rules from Figure 6.4 in three query languages. The rule samples for Cypher and MQL including handling of joins can be found in Appendices B.2 and B.3. For these particular examples, SQL happens to share the same syntax as SQL++ for all operations except operation 1, so 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 (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 6.5, aggregate functions

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 6.5: Sample Rewrite Rules

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.

6.4 Per-language Rewrite Examples

To demonstrate the generality of our approach, we have implemented a first prototype of PolyFrame that operates against AsterixDB, PostgreSQL [25], MongoDB [22] and Neo4j [24] by translating Dataframe operations into SQL++, nested SQL queries, MongoDB aggregation pipeline stages, and Cypher queries using 'WITH' statements. Table 6.1 displays query rewrites for SQL++, regular SQL, MongoDB, and Cypher that correspond to the AFrame

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

Table 6.1: PolyFrame's Incremental Query Formation

operations from Chapter 4 in Figure 4.7. 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 6.1 - 6.4.

Listing 6.1: 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 6.2: 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 6.3: 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 6.4: 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 6.1 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 6.3 displays a complete MongoDB aggregation pipeline for operation 6 from Table 6.1. 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.

6.5 Experimental Evaluation

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 the data from database systems and a framework (PolyFrame) that uses a database system to process the data. The other set of experiments illustrates our new architecture operating against different database systems to compute results and compare that to Pandas Dataframes. We conducted our experiments using the Dataframe benchmark detailed in Chapter 5. 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).

6.5.1 Experimental Setup

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

We used the Dataframe Benchmark detailed in Chapter 5 that used to evaluate AFrame here. We also used the Wisconsin benchmark data generator [41] to generate the data in

JSON file format in 5 different sizes ranging from 1GB (0.5 million records) to 10 GB (5 million records). We labeled these as the XS through XL datasets as shown in Table 6.2. The benchmark timing points for Pandas, Spark, and PolyFrame are listed in section C.1 of the Appendix.

	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

Table 6.2: Single Node's Dataset Summary (mil = million)

Comparison with Spark (Single and Multi-Node)

These experiments are included for the benefit of readers who may wonder why Spark plus its database connectivity are not the ultimate scaling answer.

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 [27] 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 [19] 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.

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

For the multi-node benchmark, we ran the benchmark only on PolyFrame operating on top of AsterixDB, MongoDB, and Greenplum (esentially parallel PostgreSQL). The community version of Neo4j does not run on sharded clusters. Since Greenplum uses PostgreSQL v.9.5 (which is different from the PostgreSQL version that we used for the single node evaluation), some of PostgreSQL's latest optimizations were not available to it. As a result, we also ran Greenplum on a single node before conducting its multi-node experiment. The evaluated cluster sizes ranged from 2-4 nodes.

Speedup and scaleup are the two preferred and widely used metrics to evaluate the processing performance of distributed systems, so we evaluated PolyFrame on each multi-node system using these two metrics. Table 6.3 displays the dataset sizes that we used for each of the cluster experiment evaluations. The aggregate memory listed is the sum of all of the available memory in the cluster. We conducted speedup experiments using a fixed-size dataset while increasing the number of processing machines from one up to four. For the scaleup experiments, we increased both the number of processing machines and the amount of data proportionally to measure each system's performance.

	1 node	**2 nodes**	**3 nodes**	**4 nodes**
Aggregate Memory	8 GB	16 GB	24 GB	32 GB
Speedup: JSON File Size	10 GB	10 GB	10GB	10 GB
Scaleup: JSON File Size	10 GB	20 GB	30GB	40 GB

Table 6.3: Multi-Node Experiment Setup

6.5.2 Spark Comparison Results

Here we present benchmark results on a single and a three-node cluster. On a single workstation PolyFrame and Spark operated on the same MongoDB instace. On a three-node cluster both systems operated on top of a Vertica cluster.

Single-node Results

For this experiment, we ran the benchmark on all the dataset sizes on a single node, we will first present the results for the 10 GB dataset (which exceeds a single node memory capacity) and then describe a few of the operations' results in detail.

Figure 6.6 shows the results for PolyFrame and Spark for all of the dataframe benchmark's expressions. It is important to note that the plot is in log scale, which understates the significant differences in runtimes. PolyFrame performed the best across all of the expressions (lower is better). It was faster than both variants of Spark.

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

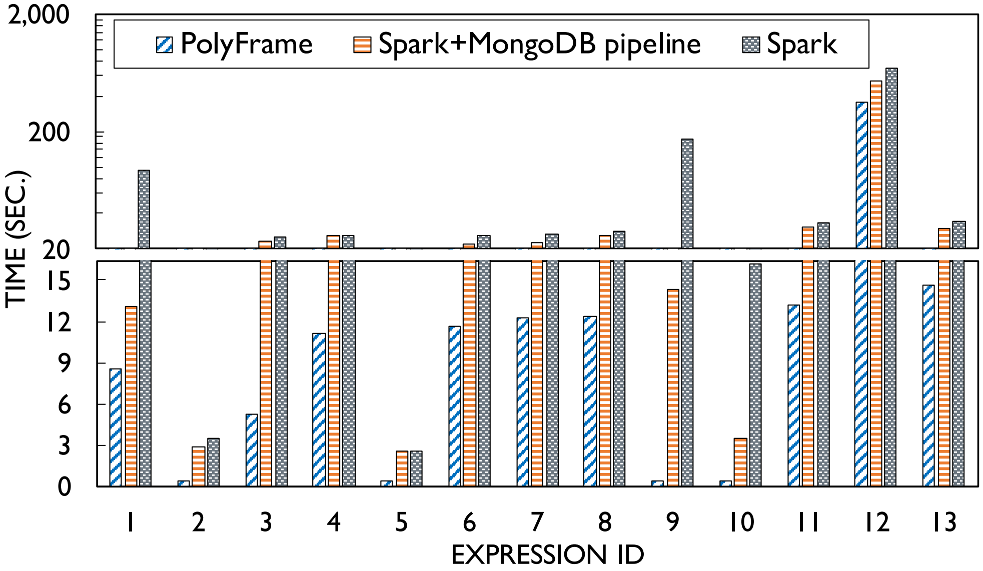


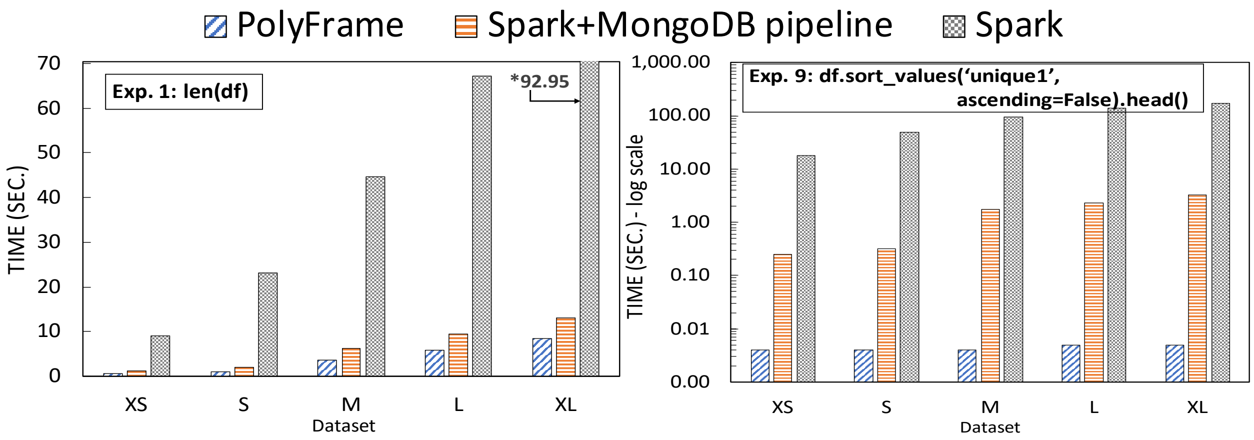
Figure 6.6: Single-node Experiment with Spark on MongoDB

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[1]. Post-processing is then done at the Spark level. We will describe some of the test cases and their results in more detail next.

Figure 6.7a displays the performance of expression 1 on all dataset sizes. This expression asks for the count of records from the dataset. Spark was significantly slower than PolyFrame here because it has to first load the data input into memory to perform the operation, and the data size was larger than the available memory. When a pipeline is passed from Spark

[1] Spark spawns a worker per core on a single machine, so MongoDB's connector defaults to partitioning the data in order to achieve maximum parallelism with Spark.



(a) Expression 1

(b) Expression 9

Figure 6.7: Selected single node Spark and PolyFrame comparisons (*=value where the bar ends)

to MongoDB, its performance was an order of magnitude faster. This is because MongoDB then performed count operations and only sent the numbers of records from each of its data partitions back to Spark to be aggregated. PolyFrame performed the best because it avoids transferring any data and having to perform any aggregation outside the database.

Figure 6.7b displays the performance result for expression 9, which asks for the five records with the highest values in a unique field. Because of this operation, we created an index on the sort attribute at the database level. MongoDB was able to then use this index to perform a backward index scan to retrieve the requested records. PolyFrame's generated MongoDB pipeline allows for such an optimization, which resulted in the best performance across all data sizes. However, we can see that Spark even with the MongoDB pipeline was still significantly slower than PolyFrame issuing the equivalent query. This is because, even when using the index, the sorted records from all partitions have to be globally sorted again in the Spark environment. However, the Spark+pipeline results were orders of magnitude faster than when Spark read the entire data from MongoDB and performed the sort operation itself.

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 6.8 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.

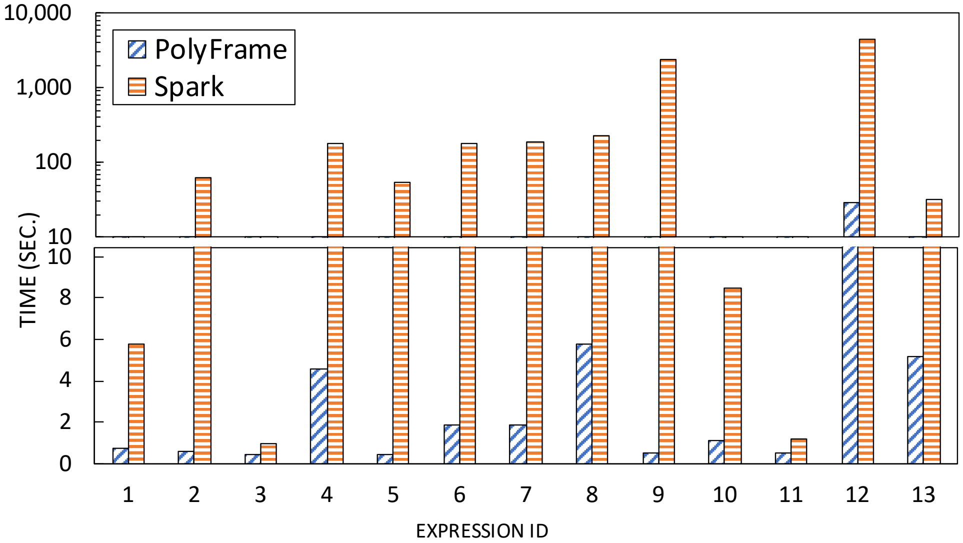
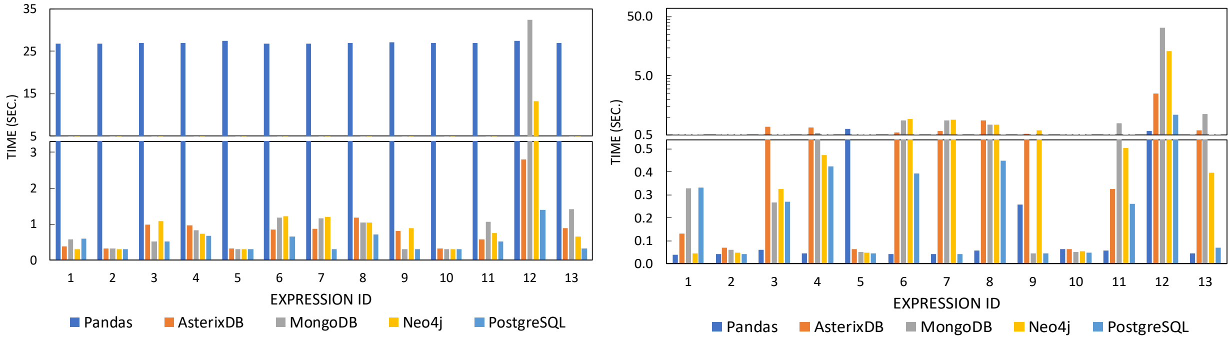


Figure 6.8: Cluster Experiment with Spark on Vertica

6.5.3 PolyFrame's Heterogeneity Results: Single-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. First, we executed the benchmark on the XS 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 6.9 presents the XS results for the single node evaluation. 6.9a displays the total runtimes for expressions 1-13, and 6.9b 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 (more on that shortly).



(a) Expression 1-13 total times

(b) Expression 1-13 expression-only times

Figure 6.9: XS Results of Single Node Evaluation

After the first XS round, we ran the benchmark on the four other dataset sizes, S, M, L and XL, to evaluate the data scalability of Pandas and of PolyFrame on a single node. The single node results are presented in Figures 6.10 to 6.13. The first thing to note here is that Pandas threw an out-of-memory error on dataset sizes M, L, and XL, while all variants of PolyFrame were able to complete all operations on all of the tested dataset sizes. A programmatic workaround for Pandas could be to partition the data and then programmatically compute and combine intermediate results. However, we did not consider this solution; a partition size would need to be specified and that would directly affect the total runtime. Figure 6.10 displays the total runtimes and expression-only runtimes for expressions 1-4. Figure 6.11 displays the total and expression-only runtimes for expressions 5-8. Figure 6.12 displays the total and expression-only runtimes for expressions 9-12 and Figure 6.13 displays expression 13 results. We discuss the results below; our discussion is based on having inspected the query plans for each operation on each system.

For Expression 1, Figure 6.10b shows that PolyFrame operating on Neo4j was the fastest across all data sizes. This is because Neo4j keeps separate data-stores specifically for its nodes and its relationships metadata, so retrieving the count of records is an instant metadata lookup. PolyFrame operating on AsterixDB was competitive and was able to take advantage of a primary key index for this particular expression, while MongoDB and PostgreSQL resorted to table scans.

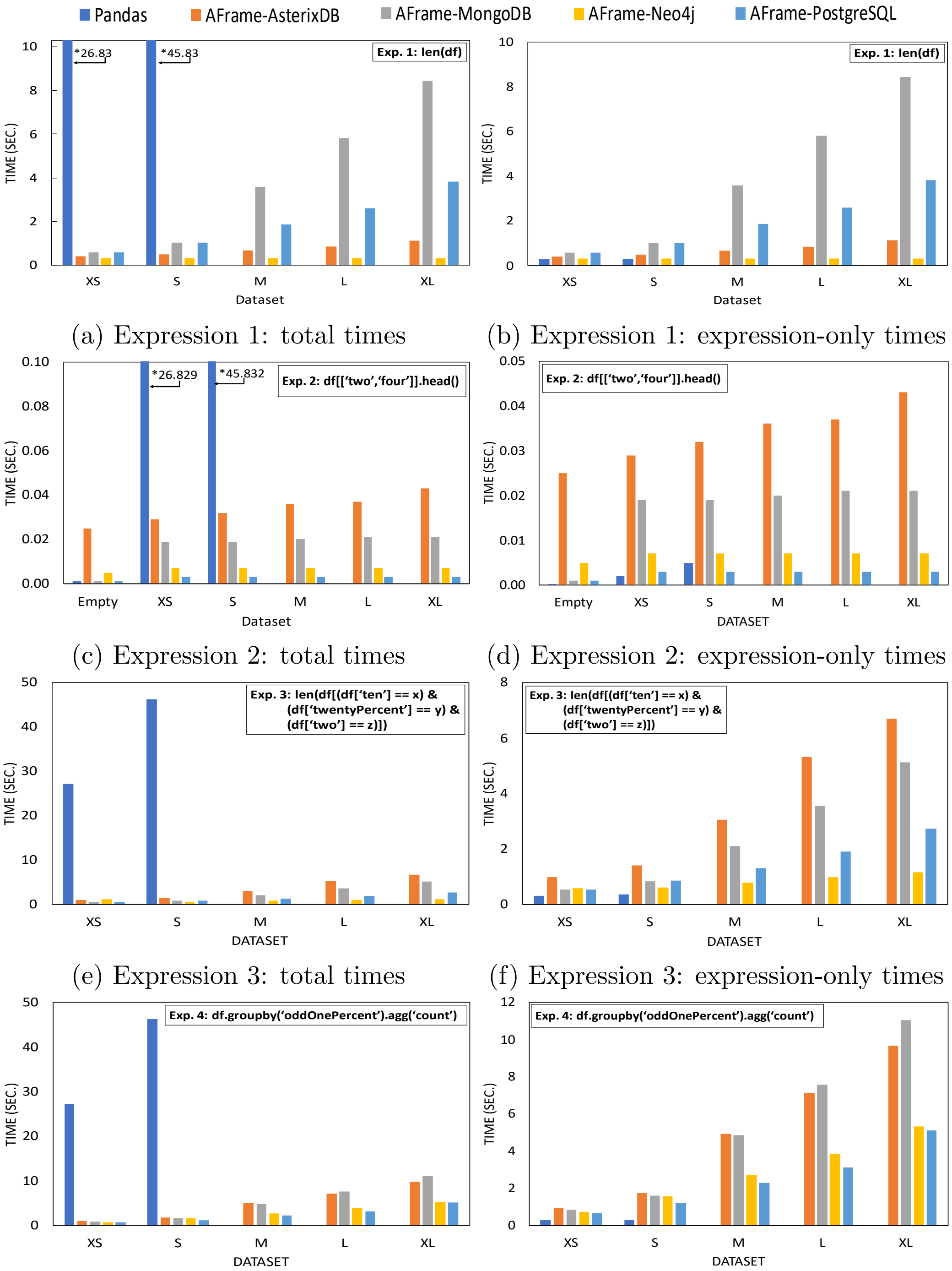
Since we are targeting data analysis at scale, for relatively 'small' queries in terms of computation such as expressions 2 and 10, whose timings are in the tens of millisecond range, we also show results for the 'Empty' dataset as a baseline for consideration. Expression 2 is asking for a projection of two attributes from a small subset of data (five records). As shown by the 'Empty' results in Figure 6.10d, all of the evaluated database systems do have some query preparation overhead, especially AsterixDB (which is designed to operate efficiently on big data rather than being fast on 'small' queries).

Expressions 5 (Figure 6.11b) and 10 (Figure 6.12d) are good demonstrations of the advantage of lazy evaluation. These two expressions involve the application of repetitive operations over a small subset of data. For both of these expressions, even in the expression-only runtime case, Pandas was slower than all variants of PolyFrame. This is because PolyFrame's lazy evaluation allows all of the evaluated database systems to take advantage of their indexes and query optimizations to limit the amount of data needed for computation. However, Pandas suffered from eagerly evaluating these expressions and needed to compute intermediate results.

For expressions 6, 7, and 13, PolyFrame running on PostgreSQL was as competitive as Pandas in the case of its expression-only runtime as shown in Figures 6.11d, 6.11f and 6.13b, respectively. This is because PostgreSQL evaluated expressions 6 and 7 using index-only query plans. For expression 13, PostgreSQL was uniquely able to use an index on the attribute. Even though this particular dataframe expression asks for null or missing data, null and missing values are only recorded in the attribute's index in PostgreSQL. On the other hand, AsterixDB, Cypher and MongoDB do support data with missing attributes, but missing values are not present in their indexes.

Figure 6.12b displays the expression-only runtime of Expression 9, which is asking for a sample of records sorted in descending order on an attribute. Even when excluding the DataFrame creation time, PolyFrame operating on MongoDB and PostgreSQL were both faster than Pandas on the smallest dataset, and all variants of PolyFrame were competitive with Pandas for dataset size S. For this expression, MongoDB and PostgreSQL were both able to take advantage of backward index scans to efficiently retrieve the requested records.

The expression-only runtimes for expression 12 are displayed in Figure 6.12h. This expression asks for the count of records resulting from a join of two identical datasets. PolyFrame operating on AsterixDB was able to evaluate this expression using an index-only query, while PostgreSQL, Neo4j, and MongoDB each used index nested loop joins followed by data scans.



(a) Expression 1: total times

(b) Expression 1: expression-only times

(c) Expression 2: total times

(d) Expression 2: expression-only times

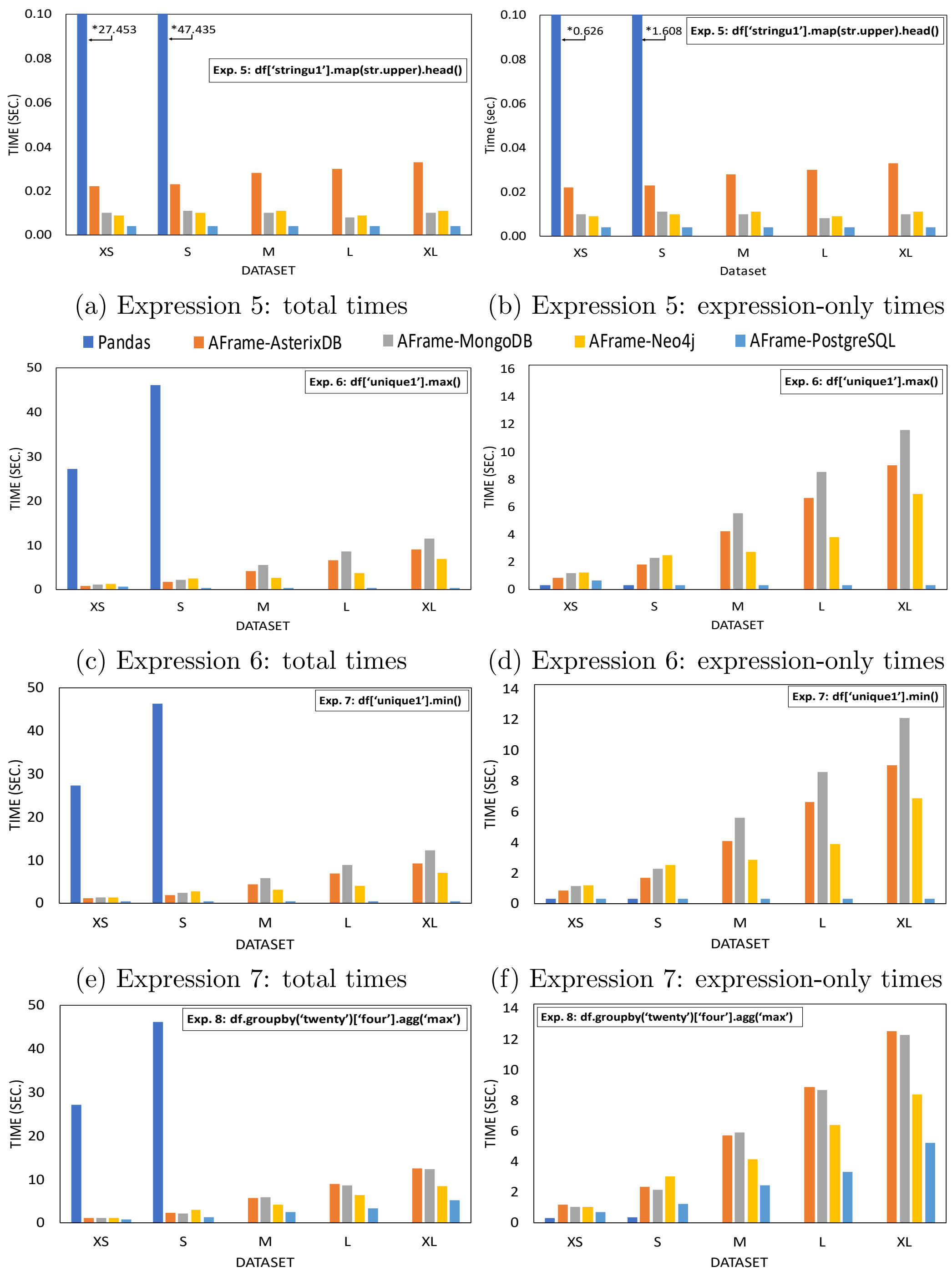
(e) Expression 3: total times

(f) Expression 3: expression-only times

(g) Expression 4: total times

(h) Expression 4: expression-only times

Figure 6.10: Exp.1-4 Single Node Evaluation Results (*=value where the bar ends)



(a) Expression 5: total times

(b) Expression 5: expression-only times

(c) Expression 6: total times

(d) Expression 6: expression-only times

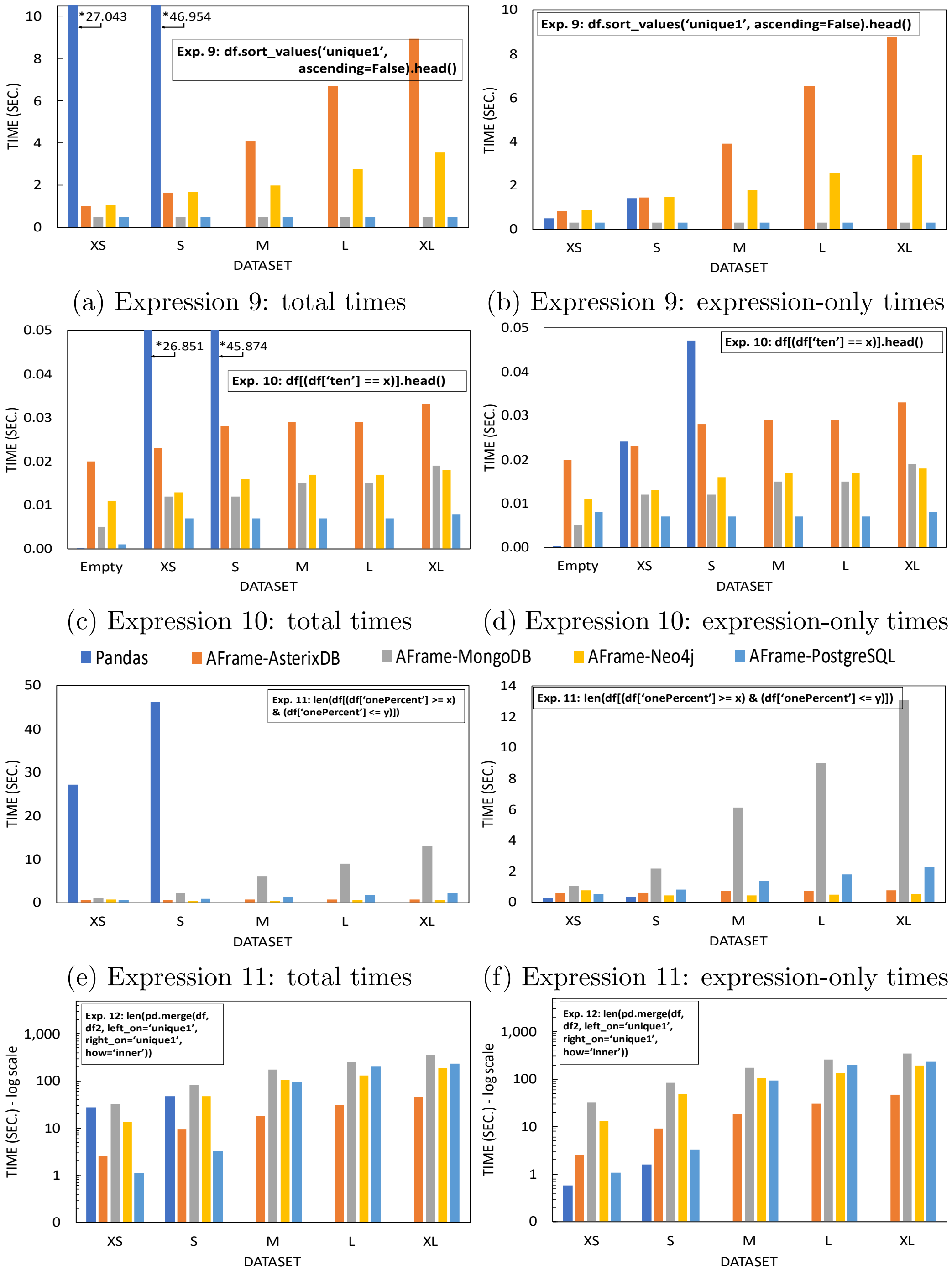
(e) Expression 7: total times

(f) Expression 7: expression-only times

(g) Expression 8: total times

(h) Expression 8: expression-only times

Figure 6.11: Exp.5-8 Single Node Evaluation Results (*=value where the bar ends)



(a) Expression 9: total times

(b) Expression 9: expression-only times

(c) Expression 10: total times

(d) Expression 10: expression-only times

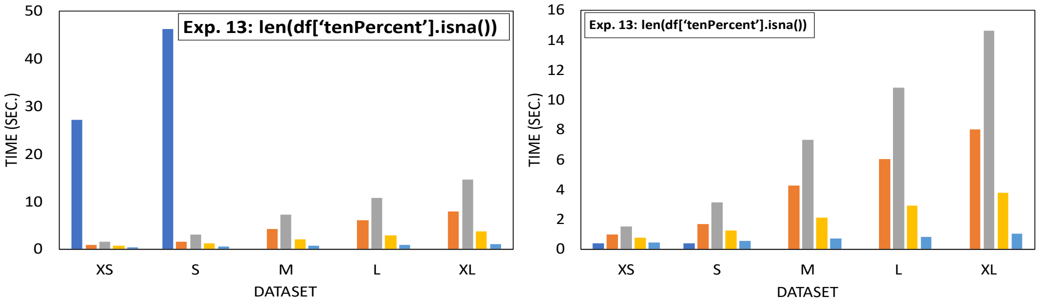
(e) Expression 11: total times

(f) Expression 11: expression-only times

(g) Expression 12: total times

(h) Expression 12: expression-only times

Figure 6.12: Exp.9-12 Single Node Evaluation Results



(a) Expression 13: total times (b) Expression 13: expression-only times

Figure 6.13: Exp.13 Single Node Evaluation Results

6.5.4 PolyFrame's Heterogeneity Results: Multi-node

As mentioned in the experiment set up part of this chapter, we only performed cluster experiments on PolyFrame. We conducted a multi-node evaluation to demonstrate PolyFrame's horizontal scalability on different database systems. We ran the benchmark on PolyFrame operating on AsterixDB, MongoDB, and Greenplum (distributed PostgreSQL) because the Neo4j community edition does not provide support for sharded multi-node clusters. 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 XL (10GB) dataset. Here we evaluated PolyFrame 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. Figures 6.14 and 6.15 display the multi-node speedup and scaleup evaluation results respectively. We only display the total time results here since PolyFrame operating on a database system does not require first loading the data into memory.

Speedup Results

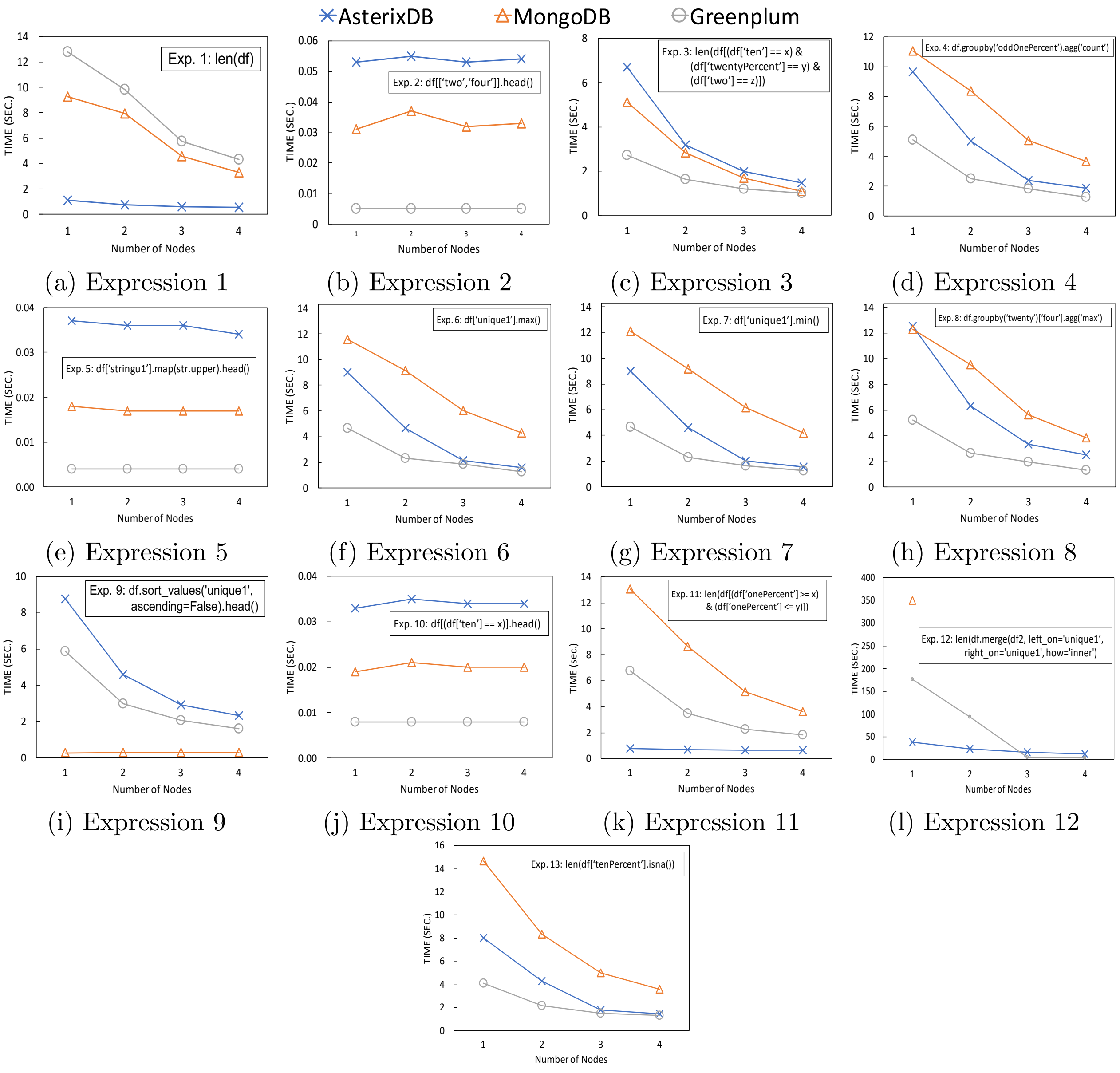
Figure 6.14 displays the speedup results for expressions 1-13 running on cluster sizes ranging from 1-4 machines. The speedup evaluation results for PolyFrame are mostly consistent with the single node results on the XL dataset except for some of Greenplum's performance. This is due to database optimizations in the latest PostgreSQL version that were not present in the version of PostgreSQL that Greenplum uses. As shown in Figure 6.14i's times, Greenplum was not able to use the backward-index scan that the latest PostgreSQL (version 12) used in the single node evaluation; instead it did a table scan. Expressions 6 (Figure 6.14f) and 7 (Figure 6.14g), which ask for the maximum and minimum values of an attribute, were evaluated as index-only queries in PostgreSQL version 12; however, for the version used in Greenplum, this was not the case.

Scaleup Results

Figure 6.15a displays the scaleup results for expressions 1-13. No single system performed the best across all tasks, but all systems were able to operate at scale when we increased the workload in proportion to the number of processing machines. The scale-up evaluation results for PolyFrame running on AsterixDB, MongoDB, and Greenplum are also consistent with the single node evaluation on the XL dataset with the same exceptions discussed in the speedup results section. Again, the exceptions are due to the older PostgreSQL version used by Greenplum.

6.5.5 Result Discussion

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 experi-



(a) Expression 1

(b) Expression 2

(c) Expression 3

(d) Expression 4

(e) Expression 5

(f) Expression 6

(g) Expression 7

(h) Expression 8

(i) Expression 9

(j) Expression 10

(k) Expression 11

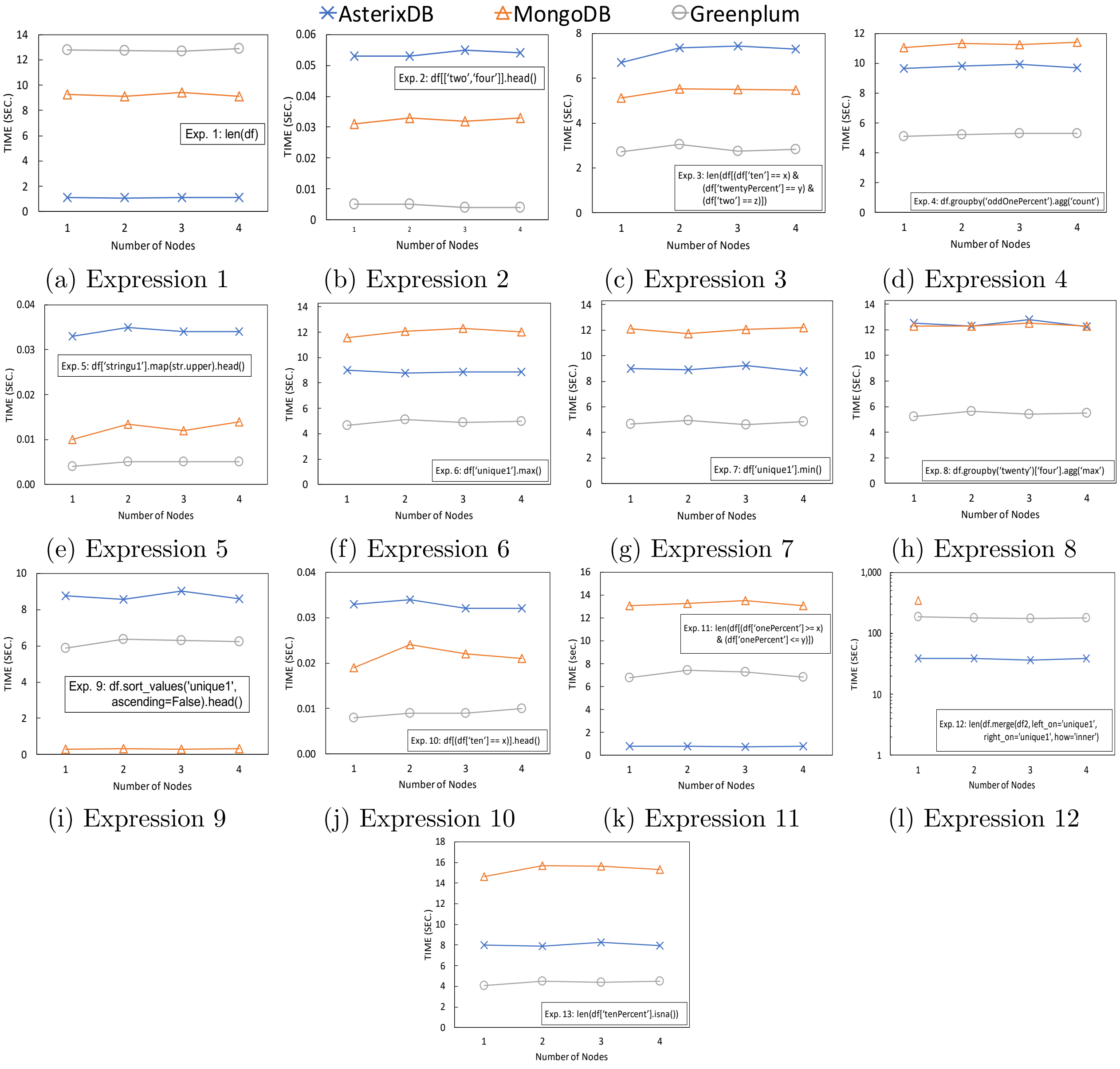
(l) Expression 12

(m) Expression 13

Figure 6.14: Speedup Evaluation Results

ments. 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 under-resourced utilization and scalability as it only utilizes a single processing core and only operates on a



(a) Expression 1
(b) Expression 2
(c) Expression 3
(d) Expression 4
(e) Expression 5
(f) Expression 6
(g) Expression 7
(h) Expression 8
(i) Expression 9
(j) Expression 10
(k) Expression 11
(l) Expression 12
(m) Expression 13

Figure 6.15: Scale-up Evaluation Results

single machine. On repetitive tasks that return a small subset of larger data, Pandas did not perform as well as PolyFrame operating on database systems that employ parallel processing and that utilize indexes to retrieve only a small subset of the data.

PolyFrame utilizes lazy evaluation by only sending queries to an underlying database system when an action is invoked. This allows PolyFrame to take advantage of database systems' optimizations. As mentioned before, we conducted our single node evaluation to compare PolyFrame's lazy evaluation approach and Pandas' eager in-memory evaluation approach rather than comparing the performance of the different database systems. We demonstrated that operating on top of the database systems 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 loading data into memory prior to computing expression results; this results in lower total runtimes across all benchmark expressions. In terms of the expression-only runtime, lazy evaluation utilizing database indexes and query optimization has better performance than eagerly evaluating certain repetitive tasks. For the multi-node evaluation, we have demonstrated the horizontal scalability of PolyFrame operating on three parallel database systems for both speedup and scale up metrics.

By configuring PolyFrame to work against a sampling of significantly different database systems and query languages, we have also 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. As a side effect of our experimental study, we were also able to identify certain potentially effective optimizations across the evaluated database systems, as discussed next.

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, and null value statistics using its indexes.

For Neo4j, apart from the usual database optimizations (e.g., index search, fast metadata lookup), we found that its storage layout and record structure also contribute to its performance, especially for this particular benchmark dataset. Neo4j stores attributes (node properties) as a linked list of fixed-size records. It stores string attributes in a separate record store, storing only pointers to these string attributes in the attribute linked list. Because of this feature, Neo4j's record structure is particularly suitable for the Wisconsin benchmark dataset since it contains multiple long string attributes. Since only one of our benchmark expressions required access to the string attributes, Neo4j had the advantage of scanning shorter records than the other (row store based) database systems.

As mentioned before in the case of MongoDB, PolyFrame utilizes its aggregation pipeline language in order to preserve AFrame's incremental query formation and to support the language rewriting process. As a result, certain query optimizations were not considered by MongoDB. For example, MongoDB also supports a fast lookup of certain metadata (similar to Neo4j) that includes the total count of records in a collection. However, this particular optimization is not enabled as part of a MongoDB aggregation pipeline, so PolyFrame was not able to take advantage of this optimization. Another limitation for MongoDB is related to joins. MongoDB only supports the joining of unsharded data, which requires at least one of the joined datasets to be stored on a single machine. As such, we could not run expression 12 on MongoDB in the distributed environment.

6.6 Conclusion

In this chapter, 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 data analysis at scale.

In its current stage, PolyFrame has already shown promising results for enabling a scale-independent data analysis experience. Moving forward, we would like to extend the PolyFrame framework's support to cover the extensive list of Pandas operations. A recent paper [54] 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.

Chapter 7

Case Studies

7.1 Introduction

Our goal for PolyFrame is to deliver a scale-independent data analysis experience. Data analysts should be able to use PolyFrame interchangeably with Pandas dataframes and utilize their favorite ML libraries to perform each of their analyses on large volumes of data with minimum effort. In order to explore the usability, efficacy, and identify current limitations of our framework, we present two separate case studies that use PolyFrame to perform end-to-end data analysis on real-world datasets. The first case study is a usability assessment, where we asked a user to use AFrame (PolyFrame on AsterixDB) to conduct a classification analysis of a San Francisco police department historical incident report dataset [17] to predict the probability of incidents being resolved. The second case study uses PolyFrame on PostgreSQL to try to mimic an existing EDA on Airbnb datasets and compare PolyFrame's performance against the Pandas dataframe baseline.

The rest of this chapter is organized as follows: Section 7.2 illustrates a usability assessment of AFrame through a case study. Section 7.3 details an EDA that uses PolyFrame on

PostgreSQL. Section 7.4 concludes the chapter.

7.2 Classification Case Study

In this section, we illustrate the usability of AFrame (PolyFrame on AsterixDB) through a case study that uses it to perform an end-to-end data analysis. We highlight some of AFrame's functionalities that help simplify Big Data analysis through each of the data analytics lifecycle stages. The notebook that we use in this chapter is also available [1].

There are several methodologies that provide a guideline for the stages in a data science lifecycle, such as the traditional method called CRISP_DM for data mining and an emerging methodology introduced by Microsoft called TDSP [48]. They have defined general phases in a data analysis lifecycle that can be summarized as follows: business understanding, data understanding and preparation, modeling, evaluation, and deployment.

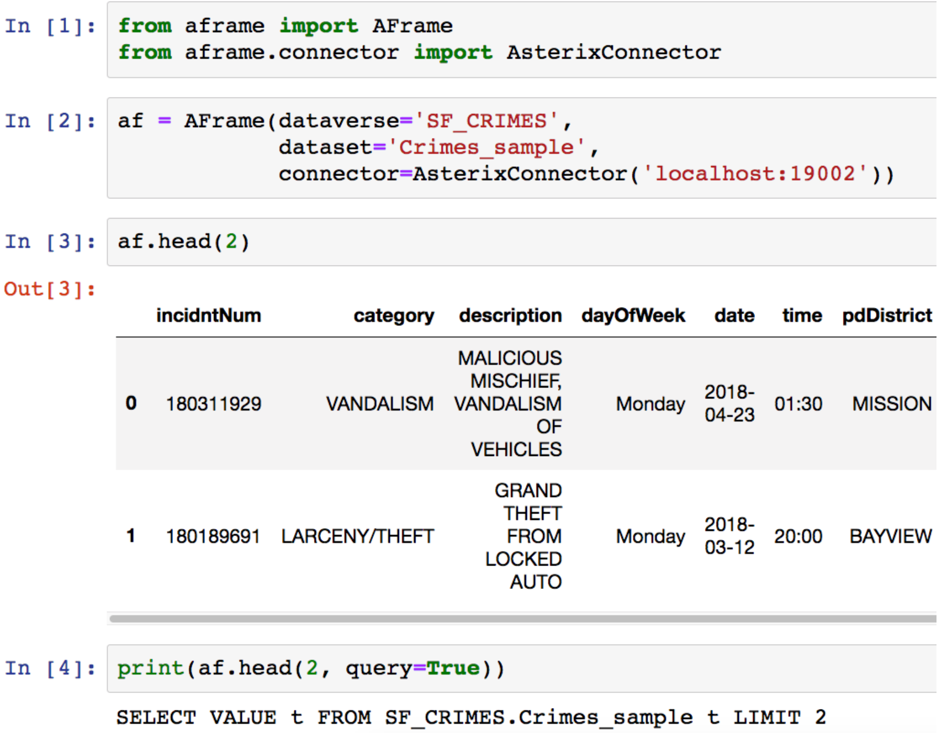
In order to see if AFrame delivers up to our expectations, we conducted a case study by asking a user to perform a data analysis using AFrame in places where Pandas DataFrames would otherwise be used (with an exception of training machine learning models). We used a running example of an analysis of a San Francisco police department historical incident report dataset [17] to predict the probability of incidents being resolved.

We present the case study here through each of the previously mentioned data science lifecycle stages.

[1] https://nbviewer.jupyter.org/github/psinthong/SF_CRIME_Notebook/blob/master/sf_crimes_paper.ipynb

7.2.1 Data Preparation

The goal of this stage in the data science lifecycle is to obtain and clean the data to prepare it for analysis. Figure 7.1 is a snapshot from a Jupyter notebook that shows a process of creating an AFrame object and displaying two records from a dataset. Input line 2 labeled 'In[2]' shows how to create an AFrame object by utilizing AFrame's AsterixDB connector and providing a dataverse name and a dataset name. For this example, the dataset is called 'Crimes_sample' and it is contained in a dataverse named 'SF_CRIMES'. The AsterixDB server is running locally on port 19002. Input line 3 displays a sample of two records from the dataset, and input line 4 displays the underlying SQL++ query that AFrame generated and extended. More about AFrame's incremental query formation process can be found in [56].



```
In [1]: from aframe import AFrame
        from aframe.connector import AsterixConnector

In [2]: af = AFrame(dataverse='SF_CRIMES',
                    dataset='Crimes_sample',
                    connector=AsterixConnector('localhost:19002'))

In [3]: af.head(2)
```

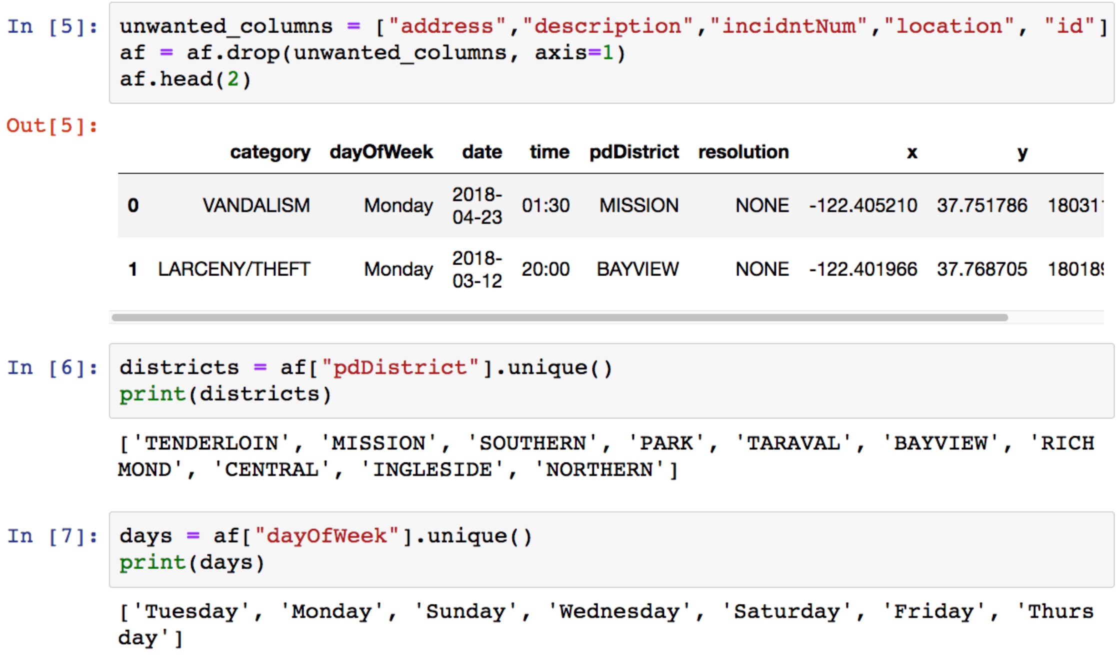
Out[3]:

	incidntNum	category	description	dayOfWeek	date	time	pdDistrict
0	180311929	VANDALISM	MALICIOUS MISCHIEF, VANDALISM OF VEHICLES	Monday	2018-04-23	01:30	MISSION
1	180189691	LARCENY/THEFT	GRAND THEFT FROM LOCKED AUTO	Monday	2018-03-12	20:00	BAYVIEW

```
In [4]: print(af.head(2, query=True))

        SELECT VALUE t FROM SF_CRIMES.Crimes_sample t LIMIT 2
```

Figure 7.1: Acquire data

Next, our user drops some columns from the dataset and explores the data values, as shown in Figure 7.2. Input line 5 drops several columns using the Pandas' 'drop' function and prints out two records from the resulting data. Our user then prints out the unique values from the columns 'pdDistrict' and 'dayOfWeek' as shown in input lines 6 and 7 respectively.



```
In [5]: unwanted_columns = ["address","description","incidntNum","location", "id"]
        af = af.drop(unwanted_columns, axis=1)
        af.head(2)
```

Out[5]:

	category	dayOfWeek	date	time	pdDistrict	resolution	x	y	
0	VANDALISM	Monday	2018-04-23	01:30	MISSION	NONE	-122.405210	37.751786	18031
1	LARCENY/THEFT	Monday	2018-03-12	20:00	BAYVIEW	NONE	-122.401966	37.768705	18018

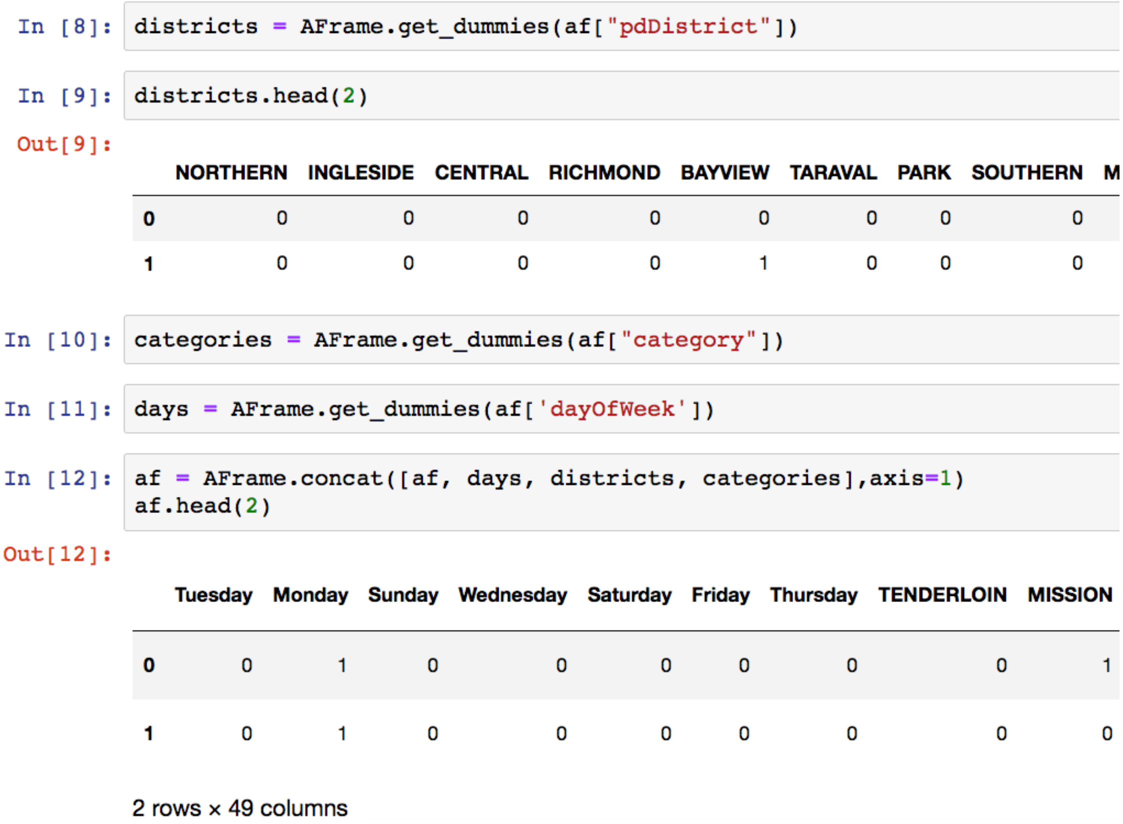
```
In [6]: districts = af["pdDistrict"].unique()
        print(districts)

        ['TENDERLOIN', 'MISSION', 'SOUTHERN', 'PARK', 'TARAVAL', 'BAYVIEW', 'RICH
        MOND', 'CENTRAL', 'INGLESIDE', 'NORTHERN']

In [7]: days = af["dayOfWeek"].unique()
        print(days)

        ['Tuesday', 'Monday', 'Sunday', 'Wednesday', 'Saturday', 'Friday', 'Thurs
        day']
```

Figure 7.2: Data cleaning and exploration

7.2.2 Modeling

The next stage in a data science project lifecycle is to determine and optimize features through feature engineering to facilitate machine learning model training. This stage also includes machine learning model development, which is the process to construct and select a model that can predict the target values most accurately considering their success metrics.

In Figure 7.3, the user applies one-hot encoding by utilizing the Pandas' 'get_dummies' function to create multiple features from the columns that he previously explored. Input line 8 applies one-hot encoding to the 'pdDistrict' column and line 9 displays the resulting data with ten new columns each indicating whether or not the record has that particular feature. Input lines 10 and 11 perform the same operation on the ‘category’ and ‘dayOfWeek’ columns respectively. The user then appends all of their one-hot encodings to the original data in input line 12.



```
In [8]: districts = AFrame.get_dummies(af["pdDistrict"])
```

```
In [9]: districts.head(2)
```

Out[9]:

	NORTHERN	INGLESIDE	CENTRAL	RICHMOND	BAYVIEW	TARAVAL	PARK	SOUTHERN	M
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	1	0	0	0	

```
In [10]: categories = AFrame.get_dummies(af["category"])
```

```
In [11]: days = AFrame.get_dummies(af['dayOfWeek'])
```

```
In [12]: af = AFrame.concat([af, days, districts, categories],axis=1)
         af.head(2)
```

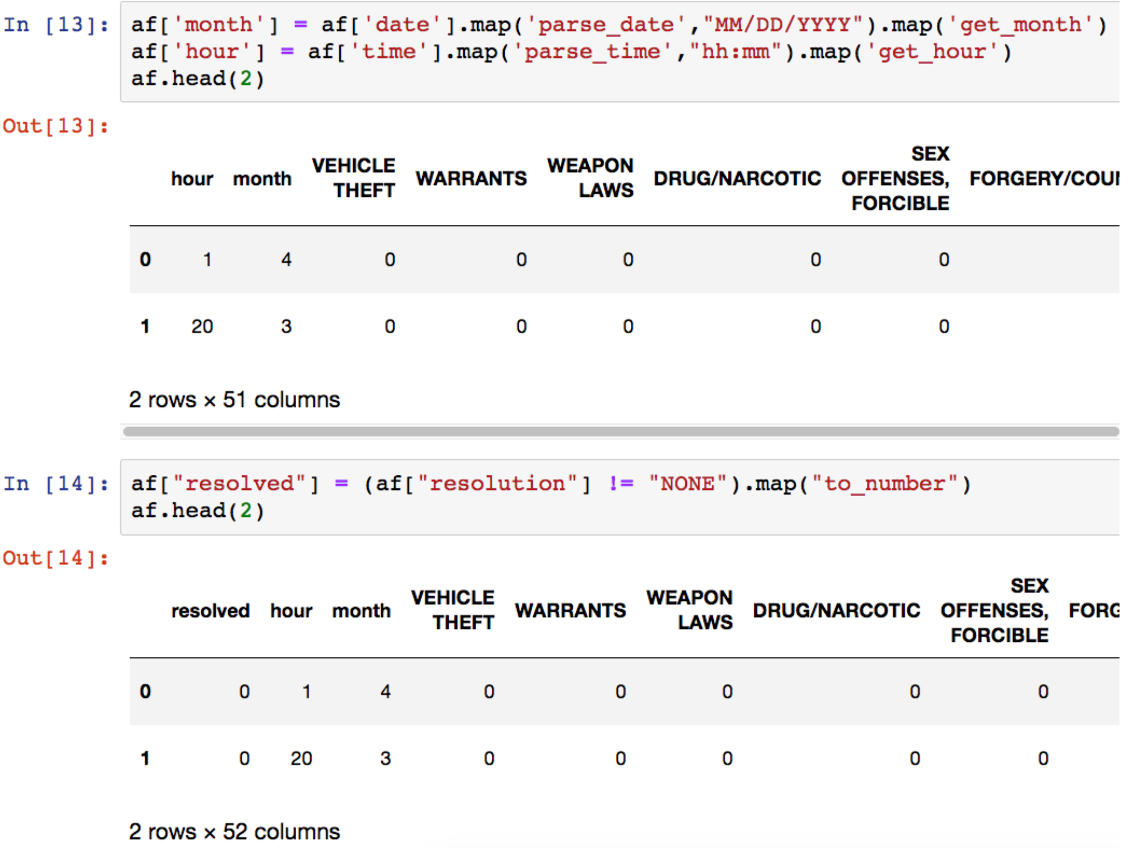
Out[12]:

	Tuesday	Monday	Sunday	Wednesday	Saturday	Friday	Thursday	TENDERLOIN	MISSION
0	0	1	0	0	0	0	0	0	1
1	0	1	0	0	0	0	0	0	0

2 rows × 49 columns

Figure 7.3: One-hot encodings

In order to extract important features from the data, users can also apply AsterixDB’s builtin functions directly on the entire data or part of the data. This is done through the ‘map’ and ‘apply’ functions. The complete list of all available builtin functions can be found at [3]. Figure 7.4 shows an example of using the map operation on a subset of the data attributes. Input line 13 creates two new columns, ‘month’ and ‘hour’. For ‘month’, the user applies AsterixDB’s ‘parse_date’ to the ‘date’ column to generate a date object and then applies the ‘get_month’ function to extract only the month before appending it as a new column called ‘month’. Similarly, for the ‘hour’ column, ‘parse_time’ is applied to the 'time' column followed by the 'get_hour' function to extract the hour of day from the data before appending it as a new column. Finally, to finish up the feature engineering process, the target column 'resolution' is converted into a binary value column called 'resolved' using AsterixDB's 'to_number' function.



```
In [13]: af['month'] = af['date'].map('parse_date',"MM/DD/YYYY").map('get_month')
         af['hour'] = af['time'].map('parse_time',"hh:mm").map('get_hour')
         af.head(2)
```

Out[13]:

	hour	month	VEHICLE THEFT	WARRANTS	WEAPON LAWS	DRUG/NARCOTIC	SEX OFFENSES, FORCIBLE	FORGERY/COUI
0	1	4	0	0	0	0	0	
1	20	3	0	0	0	0	0	

2 rows × 51 columns

```
In [14]: af["resolved"] = (af["resolution"] != "NONE").map("to_number")
         af.head(2)
```

Out[14]:

	resolved	hour	month	VEHICLE THEFT	WARRANTS	WEAPON LAWS	DRUG/NARCOTIC	SEX OFFENSES, FORCIBLE	FORG
0	0	1	4	0	0	0	0	0	
1	0	20	3	0	0	0	0	0	

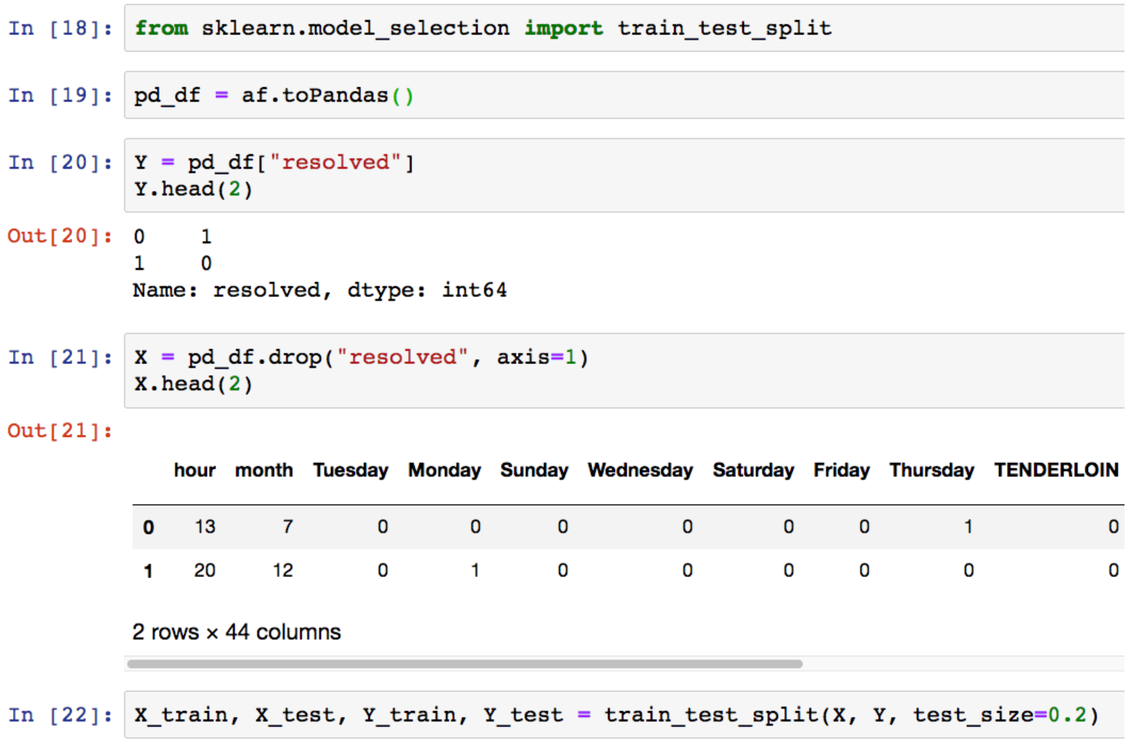
2 rows × 52 columns

Figure 7.4: Applying functions to create new columns

Once the feature engineering process is done, the data is split into training and testing sets for use in training and evaluating machine learning models. Figure 7.5 shows the process of splitting the data. Input line 19 converts the data referenced by an AFrame object into a Pandas DataFrame. Currently, feeding data into existing Scikit-learn models from a database system is not supported. As such, AFrame provides an operation called 'toPandas' which converts data into Pandas DataFrame objects[2]. On input line 20, 'Y' is the binary encoded

[2] The resulting data is required by Pandas to fit in memory. This currently limits the training dataset size, but there is no limit to the amount of data to which the model may be applied (see Section 3.3).

‘resolved’ column and the remaining columns will be used to train the models. Input line 22 splits the data into an 80% training set and a 20% testing set.



```
In [18]: from sklearn.model_selection import train_test_split

In [19]: pd_df = af.toPandas()

In [20]: Y = pd_df["resolved"]
         Y.head(2)

Out[20]: 0    1
         1    0
         Name: resolved, dtype: int64

In [21]: X = pd_df.drop("resolved", axis=1)
         X.head(2)

Out[21]:
```

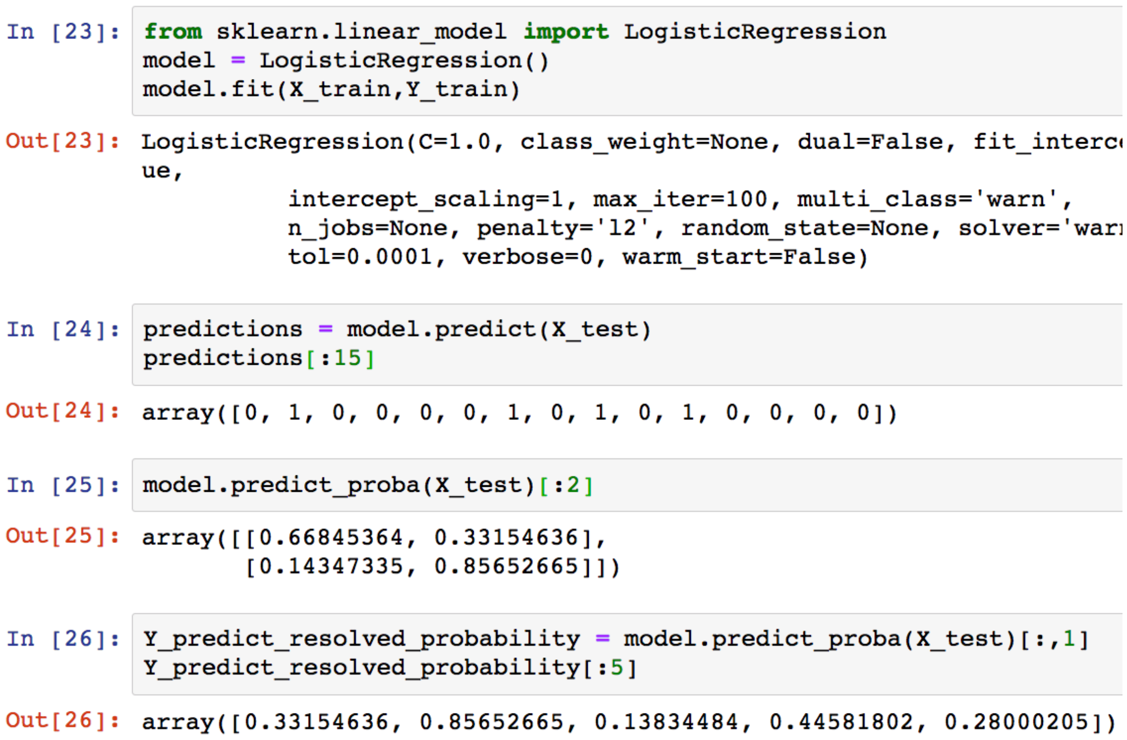
	hour	month	Tuesday	Monday	Sunday	Wednesday	Saturday	Friday	Thursday	TENDERLOIN
0	13	7	0	0	0	0	0	0	1	0
1	20	12	0	1	0	0	0	0	0	0

2 rows × 44 columns

```
In [22]: X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)
```

Figure 7.5: Preparing data for model training

Input lines 23 - 26 in Figure 7.6 are standard Scikit-learn model training steps that take a Pandas DataFrame as their input. Input line 23 trains a Logistic Regression model on the training data, while input line 24 calls the ‘predict’ function on the test data and displays a subset of the results. Instead of returning binary results indicating whether or not a particular incident will get resolved, users can utilize Scikit-learn’s ‘predict_proba’ method to get the raw probability values that the model outputs for each of the prediction labels (0 and 1 in our case). Input line 25 shows the probability values that the model outputs for each of the labels in order (0 followed by 1) on a subset of the records. Our user decided to use the ‘predict_proba’ function and output the probability of an incident getting resolved as shown in line 26.

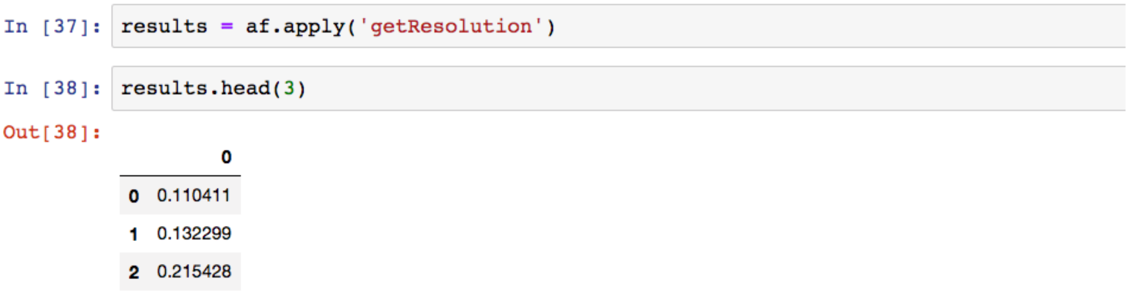


```
In [23]: from sklearn.linear_model import LogisticRegression
         model = LogisticRegression()
         model.fit(X_train,Y_train)

Out[23]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_interc
         ue,
                   intercept_scaling=1, max_iter=100, multi_class='warn',
                   n_jobs=None, penalty='l2', random_state=None, solver='war
                   tol=0.0001, verbose=0, warm_start=False)

In [24]: predictions = model.predict(X_test)
         predictions[:15]

Out[24]: array([0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0])

In [25]: model.predict_proba(X_test)[:2]

Out[25]: array([[0.66845364, 0.33154636],
                [0.14347335, 0.85652665]])

In [26]: Y_predict_resolved_probability = model.predict_proba(X_test)[:,1]
         Y_predict_resolved_probability[:5]

Out[26]: array([0.33154636, 0.85652665, 0.13834484, 0.44581802, 0.28000205])
```

Figure 7.6: Model training and inferencing

7.2.3 Evaluation and Deployment

In order to deploy the model into a production environment and apply it to large datasets at full scale, users can export and package their models using Python's pickle and then deposit them into AsterixDB for use as external user-defined functions (UDFs). In our example, the user deposited their model and created a function called 'getResolution' in AsterixDB that invokes the trained model's 'predict_proba' function on a data record. We omit the steps to deposit the model into AsterixDB and create a new UDF to call it as the required steps are explained in detail in [3]. After creating a UDF, users can then utilize their model using AFrame in the same way that they would use a builtin function. Figure 7.7 shows the user applying their Python UDF 'getResolution' on an AFrame object. Line 37 uses the 'apply' operation to apply the Python UDF on the previously transformed data. The results of the function are the probabilities of crime incidents getting resolved, as displayed in line 38.



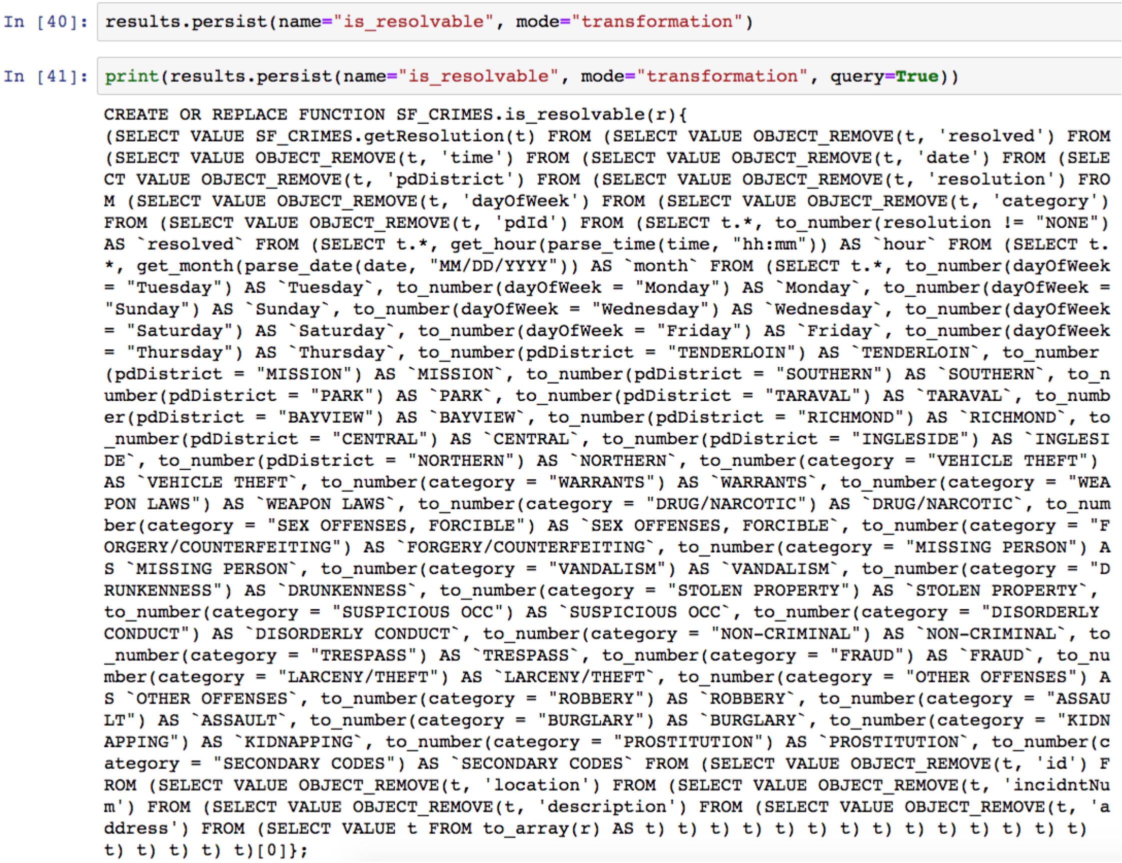
```
In [37]: results = af.apply('getResolution')
```

```
In [38]: results.head(3)
Out[38]:
            0
   0  0.110411
   1  0.132299
   2  0.215428
```

Figure 7.7: Calling the model using the function syntax

At this point in the analysis, our user is done with the training and evaluation process and wants to apply the model to other larger sets of data. However, to apply the model to a different dataset, that dataset has to have the same features arranged in the same order as the training data. To simplify the model inferencing process, AFrame allows users to save their data transformation as a function that can then be applied to other datasets effortlessly. Line 40 in Figure 7.8 displays the persist-transformation syntax. The user has named the resulting function 'is_resolvable'. The underlying SQL++ query that transforms and appends their engineered features to a data record before calling the trained model on it is shown in input line 41.

Finally, applying the data transformation and the ML model on a large distributed dataset can be done through AFrame using the apply function. Figure 7.9 shows the user accessing a different dataset, called 'Crimes', from AsterixDB and applying the 'is_resolvable' function to it. In input line 43, our user filters only the crime incidents that happened in the Central district, possibly assisted under the hood by an index on 'pdDistrict', before applying the trained ML model. The model's prediction results are appended to the dataset as a new column called 'is_resolvable' in line 44. These results can be used to do further analysis or to visualize them using Pandas' compatible visualization libraries by converting the AFrame object into a Pandas DataFrame.

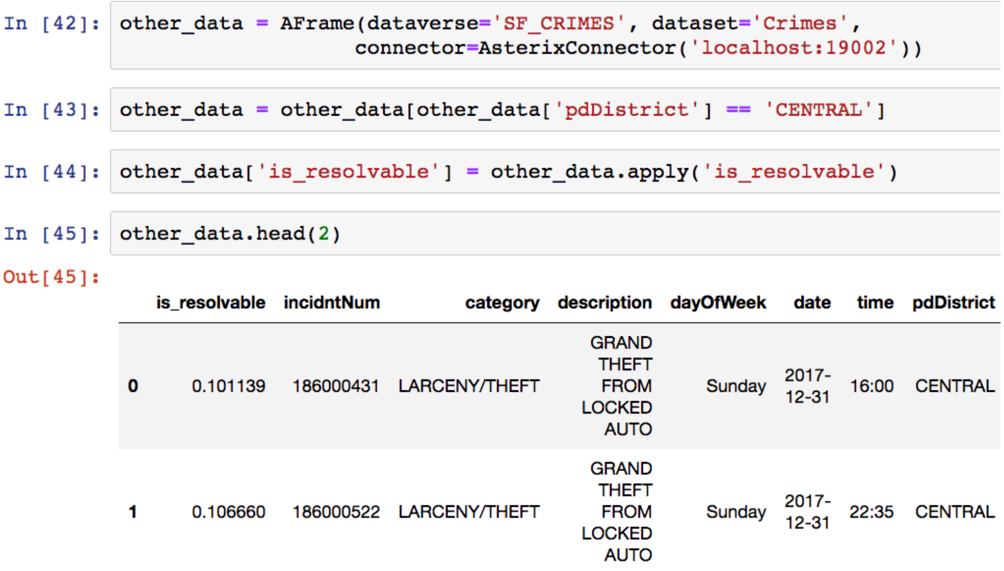


```
In [40]: results.persist(name="is_resolvable", mode="transformation")
```

```
In [41]: print(results.persist(name="is_resolvable", mode="transformation", query=True))
```

```
CREATE OR REPLACE FUNCTION SF_CRIMES.is_resolvable(r){
(SELECT VALUE SF_CRIMES.getResolution(t) FROM (SELECT VALUE OBJECT_REMOVE(t, 'resolved') FROM
(SELECT VALUE OBJECT_REMOVE(t, 'time') FROM (SELECT VALUE OBJECT_REMOVE(t, 'date') FROM (SELE
CT VALUE OBJECT_REMOVE(t, 'pdDistrict') FROM (SELECT VALUE OBJECT_REMOVE(t, 'resolution') FRO
M (SELECT VALUE OBJECT_REMOVE(t, 'dayOfWeek') FROM (SELECT VALUE OBJECT_REMOVE(t, 'category')
FROM (SELECT VALUE OBJECT_REMOVE(t, 'pdId') FROM (SELECT t.*, to_number(resolution != "NONE")
AS `resolved` FROM (SELECT t.*, get_hour(parse_time(time, "hh:mm")) AS `hour` FROM (SELECT t.
*, get_month(parse_date(date, "MM/DD/YYYY")) AS `month` FROM (SELECT t.*, to_number(dayOfWeek
= "Tuesday") AS `Tuesday`, to_number(dayOfWeek = "Monday") AS `Monday`, to_number(dayOfWeek =
"Sunday") AS `Sunday`, to_number(dayOfWeek = "Wednesday") AS `Wednesday`, to_number(dayOfWeek
= "Saturday") AS `Saturday`, to_number(dayOfWeek = "Friday") AS `Friday`, to_number(dayOfWeek
= "Thursday") AS `Thursday`, to_number(pdDistrict = "TENDERLOIN") AS `TENDERLOIN`, to_number
(pdDistrict = "MISSION") AS `MISSION`, to_number(pdDistrict = "SOUTHERN") AS `SOUTHERN`, to_n
umber(pdDistrict = "PARK") AS `PARK`, to_number(pdDistrict = "TARAVAL") AS `TARAVAL`, to_numb
er(pdDistrict = "BAYVIEW") AS `BAYVIEW`, to_number(pdDistrict = "RICHMOND") AS `RICHMOND`, to
_number(pdDistrict = "CENTRAL") AS `CENTRAL`, to_number(pdDistrict = "INGLESIDE") AS `INGLESI
DE`, to_number(pdDistrict = "NORTHERN") AS `NORTHERN`, to_number(category = "VEHICLE THEFT")
AS `VEHICLE THEFT`, to_number(category = "WARRANTS") AS `WARRANTS`, to_number(category = "WEA
PON LAWS") AS `WEAPON LAWS`, to_number(category = "DRUG/NARCOTIC") AS `DRUG/NARCOTIC`, to_num
ber(category = "SEX OFFENSES, FORCIBLE") AS `SEX OFFENSES, FORCIBLE`, to_number(category = "F
ORGERY/COUNTERFEITING") AS `FORGERY/COUNTERFEITING`, to_number(category = "MISSING PERSON") A
S `MISSING PERSON`, to_number(category = "VANDALISM") AS `VANDALISM`, to_number(category = "D
RUNKENNESS") AS `DRUNKENNESS`, to_number(category = "STOLEN PROPERTY") AS `STOLEN PROPERTY`,
to_number(category = "SUSPICIOUS OCC") AS `SUSPICIOUS OCC`, to_number(category = "DISORDERLY
CONDUCT") AS `DISORDERLY CONDUCT`, to_number(category = "NON-CRIMINAL") AS `NON-CRIMINAL`, to
_number(category = "TRESPASS") AS `TRESPASS`, to_number(category = "FRAUD") AS `FRAUD`, to_nu
mber(category = "LARCENY/THEFT") AS `LARCENY/THEFT`, to_number(category = "OTHER OFFENSES") A
S `OTHER OFFENSES`, to_number(category = "ROBBERY") AS `ROBBERY`, to_number(category = "ASSAU
LT") AS `ASSAULT`, to_number(category = "BURGLARY") AS `BURGLARY`, to_number(category = "KIDN
APPING") AS `KIDNAPPING`, to_number(category = "PROSTITUTION") AS `PROSTITUTION`, to_number(c
ategory = "SECONDARY CODES") AS `SECONDARY CODES` FROM (SELECT VALUE OBJECT_REMOVE(t, 'id') F
ROM (SELECT VALUE OBJECT_REMOVE(t, 'location') FROM (SELECT VALUE OBJECT_REMOVE(t, 'incidntNu
m') FROM (SELECT VALUE OBJECT_REMOVE(t, 'description') FROM (SELECT VALUE OBJECT_REMOVE(t, 'a
ddress') FROM (SELECT VALUE t FROM to_array(r) AS t) t) t) t) t) t) t) t) t) t) t) t) t) t)
t) t) t) t) t)[0]};
```

Figure 7.8: Persisting the transformation



```
In [42]: other_data = AFrame(dataverse='SF_CRIMES', dataset='Crimes',
                            connector=AsterixConnector('localhost:19002'))
```

```
In [43]: other_data = other_data[other_data['pdDistrict'] == 'CENTRAL']
```

```
In [44]: other_data['is_resolvable'] = other_data.apply('is_resolvable')
```

```
In [45]: other_data.head(2)
```

Out[45]:

	is_resolvable	incidntNum	category	description	dayOfWeek	date	time	pdDistrict
0	0.101139	186000431	LARCENY/THEFT	GRAND THEFT FROM LOCKED AUTO	Sunday	2017-12-31	16:00	CENTRAL
1	0.106660	186000522	LARCENY/THEFT	GRAND THEFT FROM LOCKED AUTO	Sunday	2017-12-31	22:35	CENTRAL

Figure 7.9: Model inferencing

7.2.4 Lessons Learned

The case study was not only helpful in proving the usability of AFrame but it also helped us identify useful and missing features. For example, the transformation saving mode was created to record the data transformation steps as a function so that they can easily be applied to other datasets using the existing function syntax. Unique Pandas' functions (e.g., describe, get_dummies) are now implemented in AFrame by internally calling multiple simple operations in a sequence. This design decision was influenced by the engineered features that the user manually created.

7.3 Exploratory Data Analysis Case Study

In the first case study we asked a user to use PolyFrame to perform an analysis in order to assess PolyFrame's usability and identify the missing features. After that, we decided to conduct another case study where we applied PolyFrame to an existing exploratory data analysis on real-world datasets that originally used Pandas dataframes.

Exploratory Data Analysis (EDA) is a process to identify patterns, explore relations, and extract information from unfamiliar datasets. The process is iterative, as data scientists often begin with limited knowledge about the data and its structure. During an EDA, data scientists perform a wide range of data preparation and cleaning steps such as identifying outliers, normalizing value ranges, and eliminating or filling null attributes to align the data with the goals of the analysis. The end goal can also evolve over the course of the analysis as meaningful patterns begin to be drawn from the data. Efficiently processing large amounts of data usually requires establishing a complex computational infrastructure. As a result, performing EDA operations on large amounts of data today is a challenging and time-consuming task for data scientists, as the language that is used to query the data in a data store is typically different from the descriptive language used in analyses.

Here we investigate the usability and efficacy of PolyFrame through a detailed case study. In our end-to-end case study analysis, we highlight PolyFrame's functionality through each of the stages in the data science project lifecycle [48], from data retrieval to data cleaning and through to deployment. We model our case study after an online Jupyter notebook [1] outlining an EDA on an Airbnb dataset. Our case study analysis notebooks are available here[3]. We evaluate the performance of PolyFrame by measuring the runtime of the dataframe operations throughout the analysis over increasingly large datasets and compare that to Pandas' dataframe performance. The performance comparison is not only helpful in illustrating PolyFrame's optimization benefits, but it also reveals Pandas' limitations in eagerly evaluating dataframe expressions. Pandas was not able to complete the analysis even when operating on moderate-sized data. On the other hand, PolyFrame running on the same datasets was able to reduce the amount of data materialization due to operating on a database system with an effective query optimizer. The objectives of this exercise are the following:

1. Empirically investigate the usability and performance benefits of PolyFrame, a database-backed dataframe library, through an end-to-end case study analysis.
2. Identify PolyFrame's current limitations and potential future optimization opportunities.

The rest of this section is organized as follows: Subsection 7.3.1 describes functionality supported in PolyFrame. Subsections 7.3.2 - 7.3.5 provide detail of our case study analysis. Subsection 7.3.6 describes a series of performance measurements obtained from running PolyFrame and Pandas on PostgreSQL.

[3] https://github.com/psinthong/PolyFrame_Case_Study

We have selected the case study presented here as an example for exploratory data analysis using PolyFrame. We model this case study after an online Jupyter notebook that tries to answer a set of questions about vacation rentals in Seattle using Airbnb datasets [1]. Some of the questions that its author is trying to address are the average price per night, time to rent to maximize revenue, the off-peak season for maintenance, the common group size of Seattle travelers, bedroom configurations to maximize booking rates, and factors that affect ratings and listing prices.

The author's notebook provides details about the datasets used, which are listings and reviews. The datasets are obtained from 'Inside Airbnb' [11], an online website that collects Airbnb data. The author cleans both datasets of any anomalies, generates data summaries, and applies machine learning models to extract information from the data. First, we describe the dataframe functions used in our own notebooks. After that, we present our case study and how we scale it and discuss PolyFrame's support through each of the stages in this analysis, which are data acquisition, data preparation, data analysis, and data visualization.

7.3.1 Functionality Supported

Petersohn et al [54] have identified three types of functions in the Pandas dataframe library - relational algebra, linear algebra, and spreadsheet functions. PolyFrame focuses on support for the relational algebra portion of the library, as it relies on general features supported by database systems. PolyFrame provides a data conversion mechanism to Pandas dataframe for other non-supported functions.

It is worth noting that the Pandas library contains over hundreds of dataframe functions for data transformations. As stated in [32], given a specific input data and an expected output, there can be multiple sequences of functions that can be invoked to generate such a result. Therefore, we try to utilize a different PolyFrame supported dataframe function or a sequence of functions in places in the original notebook where the requested function cannot be directly implemented. However, if similar functions are not present, we convert the PolyFrame objects into Pandas dataframes and then proceed.

Table 7.1 displays the functionality supported in PolyFrame for each of the functions used in the case study notebook. The table indicates PolyFrame's support for each function in one of three modes: First, the function is fully supported. Second, the function is partially supported or a similar function is used to achieve the same result. Third, the function is not supported and an object conversion from PolyFrame to Pandas dataframe is required.

Operation	**Functionality Support Levels**		
	Supported	**Use similar operation**	**Not supported**
Data acquisition			
Load listing dataset	✓		
Load review dataset	✓		
Data preparation			
Describe (aggregate functions)	✓		
Column projections	✓		
Check null/missing values	✓		
Fill null/missing data		✓	
Type conversion		✓	
String manipulation		✓	
Rename columns	✓		
Data analysis			
Left join	✓		
Inner join	✓		
Group and get aggregate values	✓		
Sort	✓		
Correlation			✓
One-hot encodings	✓		
Linear regression model			✓
Decision tree model			✓
Data visualization			
Bar plot			✓
Location plot			✓
Heatmap			✓

Table 7.1: Functionality Support Levels

As indicated in the table, all of the data acquisition and preparation operations used can be performed using PolyFrame. There are some data preparation functions that are not identically supported in PolyFrame so we used other dataframe functions that produce the same results instead. Operations such as type conversions and string manipulations rely on specific built-in functions, which are different for each database system. During the data analysis stage, the relational algebra and statistical functions are fully supported, but machine learning models (linear regression and decision tree) are not supported. Pandas dataframe object conversion is applied in that case. Similarly, visualization functions are also applied to converted dataframe objects because the Scikit-learn and matplotlib functions that are used in this notebook cannot operate on database-resident data.

Next, we highlight some of the operations and their usages as supported in Pandas and PolyFrame in each of the stages in the analysis in detail. We implemented the case study using two different Jupyter notebooks; one entirely using Pandas and the other one using PolyFrame.

7.3.2 Data Acquisition

Pandas dataframes can be created by reading a file from a local file system or by reading from a table in a database system. For this case study analysis, we set up a PostgreSQL database to store the data. Pandas dataframes in this analysis are created by providing the PostgreSQL's server address. There are two datasets used in the original notebook, 'listings' and 'reviews'. We created listings and reviews tables prior to starting the analysis.

In PolyFrame, a dataframe object can be created by providing a connecter to a table in a database system. As shown in the last import statement in line 1 in Figure 7.10b, PolyFrame provides SQLConnector class which internally uses SQLAlchemy [18] to connect to various SQL database systems. PolyFrame also provides an easily extensible database connector interface to allow users to implement their own socket connection to an underlying database system. PolyFrame also supports dataframe creation from external CSV or JSON files. The file content will then be used to create a table in the database.

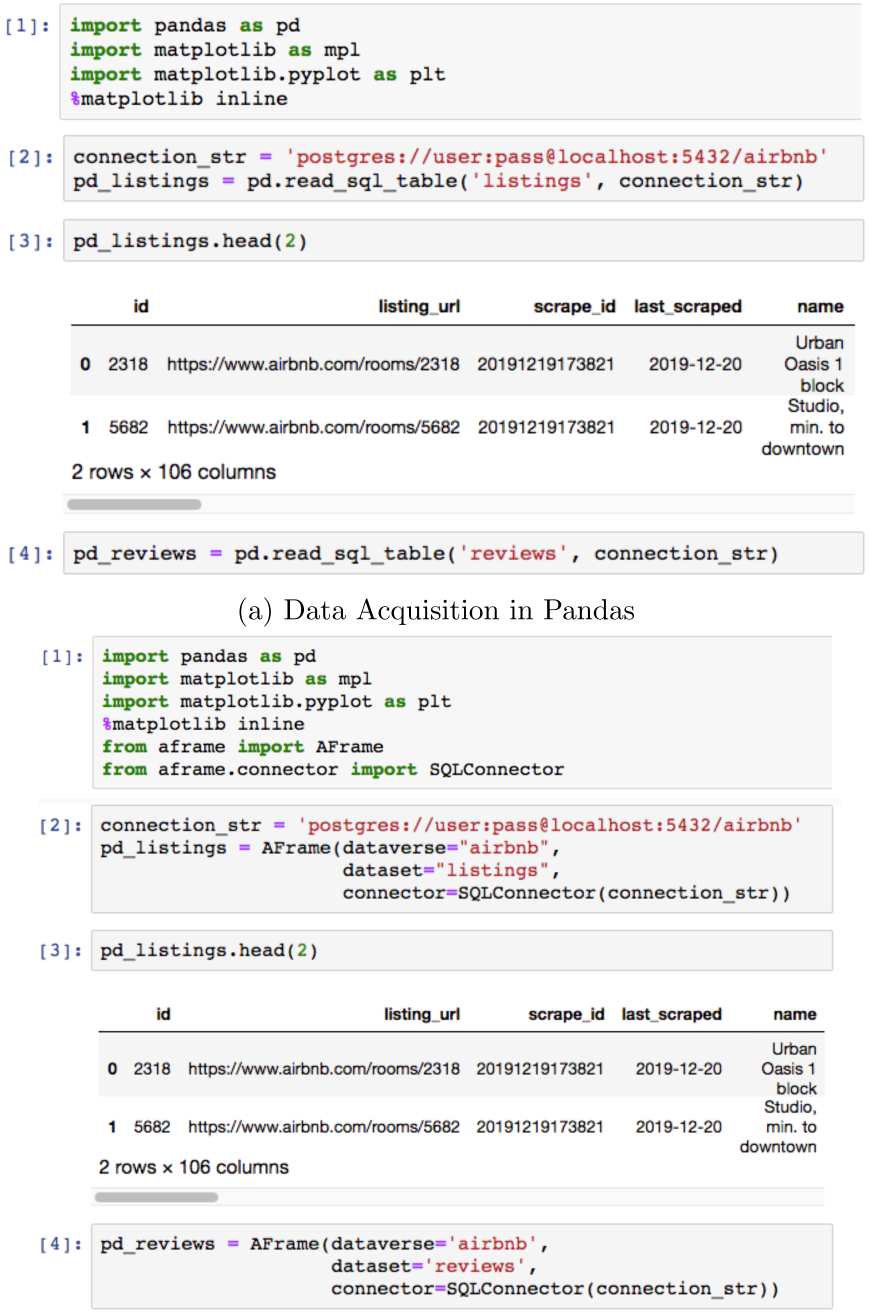
Figure 7.10 displays snippets from two notebooks. Figure 7.10a shows Airbnb's data loading using Pandas dataframes, and Figure 7.10b shows the same process using PolyFrame. Both PolyFrame and pandas made a connection to the same PostgreSQL instance running locally. Input lines 2 and 4 show the syntax of creating dataframe objects from the listings table and reviews table respectively. Pandas uses the `read_sql_table` method given a table name and a SQLAlchemy connection string. PolyFrame requires both database and table names with a SQLAlchemy connection string. Line 3 in each figure displays two sample records from the listings table.

It is worth pointing out that when a Pandas dataframe object is created the entire table is loaded into memory. In PolyFrame, only a connection is made to the database system to check for the existence of the indicated table and an initial query is generated but not executed. The data is thus not loaded into memory for PolyFrame's dataframe object initialization.

7.3.3 Data Preparation

In preparing the Airbnb data for analysis, several data transformations are applied. Figure 7.11 displays examples from the data preparation process that are applied to the dataframes. Figure 7.11a shows a subset of the data transformation steps in Pandas dataframe and Figure 7.11b shows the same steps applied in PolyFrame.

Line 9 converts the data type of attribute 'price' from string to a numerical data type using a string regular expression format to match the indicated pattern. It removes the dollar sign and gets rid of commas. Line 10 fills null values in selected attributes with zeroes. Line 11 renames an attribute. Lines 12 and 13 clean the reviews dataset and generate bookings data.



```
[1]: import pandas as pd
     import matplotlib as mpl
     import matplotlib.pyplot as plt
     %matplotlib inline

[2]: connection_str = 'postgres://user:pass@localhost:5432/airbnb'
     pd_listings = pd.read_sql_table('listings', connection_str)

[3]: pd_listings.head(2)
```

	id	listing_url	scrape_id	last_scraped	name
0	2318	https://www.airbnb.com/rooms/2318	20191219173821	2019-12-20	Urban Oasis 1 block
1	5682	https://www.airbnb.com/rooms/5682	20191219173821	2019-12-20	Studio, min. to downtown

2 rows × 106 columns

```
[4]: pd_reviews = pd.read_sql_table('reviews', connection_str)
```

(a) Data Acquisition in Pandas

```
[1]: import pandas as pd
     import matplotlib as mpl
     import matplotlib.pyplot as plt
     %matplotlib inline
     from aframe import AFrame
     from aframe.connector import SQLConnector

[2]: connection_str = 'postgres://user:pass@localhost:5432/airbnb'
     pd_listings = AFrame(dataverse="airbnb",
                          dataset="listings",
                          connector=SQLConnector(connection_str))

[3]: pd_listings.head(2)
```

	id	listing_url	scrape_id	last_scraped	name
0	2318	https://www.airbnb.com/rooms/2318	20191219173821	2019-12-20	Urban Oasis 1 block
1	5682	https://www.airbnb.com/rooms/5682	20191219173821	2019-12-20	Studio, min. to downtown

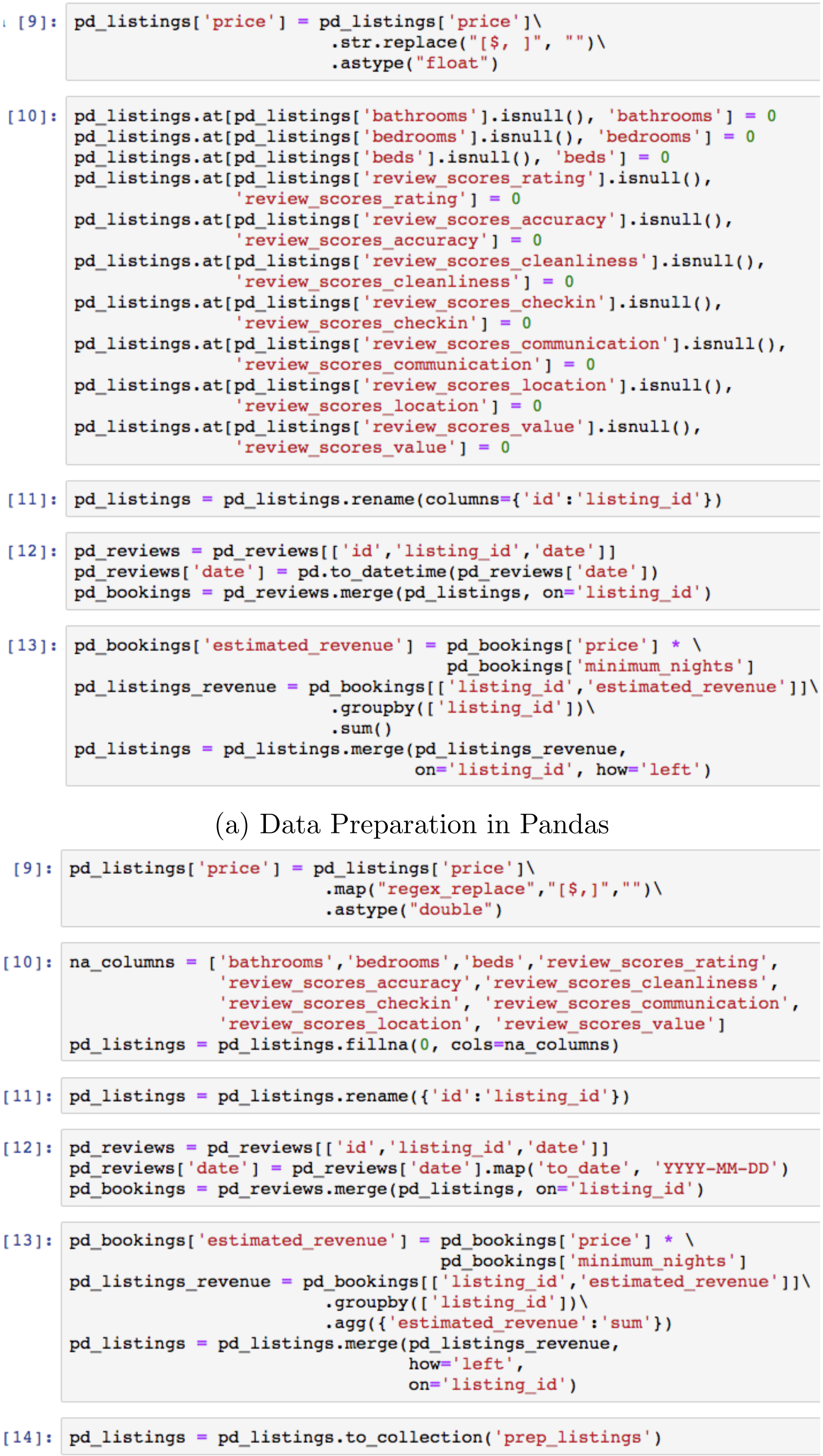
2 rows × 106 columns

```
[4]: pd_reviews = AFrame(dataverse='airbnb',
                         dataset='reviews',
                         connector=SQLConnector(connection_str))
```

(b) Data Acquisition in PolyFrame

Figure 7.10: Data Acquisition

As stated before, not all functions can be directly supported in PolyFrame. String manipulation functions are functions that depend on specific database features. Line 9 in Figure 7.11b shows the utilization of the map function to apply regular expression pattern matching by calling the database's built-in function (regex_replace) on the attribute 'price'.



```
[9]:  pd_listings['price'] = pd_listings['price']\
                              .str.replace("[$, ]", "")\
                              .astype("float")
```

```
[10]: pd_listings.at[pd_listings['bathrooms'].isnull(), 'bathrooms'] = 0
      pd_listings.at[pd_listings['bedrooms'].isnull(), 'bedrooms'] = 0
      pd_listings.at[pd_listings['beds'].isnull(), 'beds'] = 0
      pd_listings.at[pd_listings['review_scores_rating'].isnull(),
                     'review_scores_rating'] = 0
      pd_listings.at[pd_listings['review_scores_accuracy'].isnull(),
                     'review_scores_accuracy'] = 0
      pd_listings.at[pd_listings['review_scores_cleanliness'].isnull(),
                     'review_scores_cleanliness'] = 0
      pd_listings.at[pd_listings['review_scores_checkin'].isnull(),
                     'review_scores_checkin'] = 0
      pd_listings.at[pd_listings['review_scores_communication'].isnull(),
                     'review_scores_communication'] = 0
      pd_listings.at[pd_listings['review_scores_location'].isnull(),
                     'review_scores_location'] = 0
      pd_listings.at[pd_listings['review_scores_value'].isnull(),
                     'review_scores_value'] = 0
```

```
[11]: pd_listings = pd_listings.rename(columns={'id':'listing_id'})
```

```
[12]: pd_reviews = pd_reviews[['id','listing_id','date']]
      pd_reviews['date'] = pd.to_datetime(pd_reviews['date'])
      pd_bookings = pd_reviews.merge(pd_listings, on='listing_id')
```

```
[13]: pd_bookings['estimated_revenue'] = pd_bookings['price'] * \
                                         pd_bookings['minimum_nights']
      pd_listings_revenue = pd_bookings[['listing_id','estimated_revenue']]\
                              .groupby(['listing_id'])\
                              .sum()
      pd_listings = pd_listings.merge(pd_listings_revenue,
                                      on='listing_id', how='left')
```

(a) Data Preparation in Pandas

```
[9]:  pd_listings['price'] = pd_listings['price']\
                              .map("regex_replace","[$,]","")\
                              .astype("double")
```

```
[10]: na_columns = ['bathrooms','bedrooms','beds','review_scores_rating',
                    'review_scores_accuracy','review_scores_cleanliness',
                    'review_scores_checkin', 'review_scores_communication',
                    'review_scores_location', 'review_scores_value']
      pd_listings = pd_listings.fillna(0, cols=na_columns)
```

```
[11]: pd_listings = pd_listings.rename({'id':'listing_id'})
```

```
[12]: pd_reviews = pd_reviews[['id','listing_id','date']]
      pd_reviews['date'] = pd_reviews['date'].map('to_date', 'YYYY-MM-DD')
      pd_bookings = pd_reviews.merge(pd_listings, on='listing_id')
```

```
[13]: pd_bookings['estimated_revenue'] = pd_bookings['price'] * \
                                         pd_bookings['minimum_nights']
      pd_listings_revenue = pd_bookings[['listing_id','estimated_revenue']]\
                              .groupby(['listing_id'])\
                              .agg({'estimated_revenue':'sum'})
      pd_listings = pd_listings.merge(pd_listings_revenue,
                                      how='left',
                                      on='listing_id')
```

```
[14]: pd_listings = pd_listings.to_collection('prep_listings')
```

(b) Data Preparation in PolyFrame

Figure 7.11: Data Preparation

PolyFrame enables built-in function application through the dataframe's `map` and `apply` functions. Function parameters can be passed in as arguments.

In line 10 in Figure 7.11a the author applies Pandas' `at` function, which accesses data by row to find rows with null values and replace them with zeroes. The `at` function cannot be supported in PolyFrame because row indexes are not generally supported in database systems since most database collections are unordered. However, in Pandas there is another function, `fillna`, which can do the same thing in one function call. As a result, we use `fillna` instead in line 10 in Figure 7.11b given the target attributes as an argument.

PolyFrame and Pandas have the same syntax for line 11. In line 12, the attribute 'date' has been converted from the data type string into the datetime type in both Pandas dataframe and PolyFrame. However, datetime object conversion is a database-specific feature, so it is supported in PolyFrame through the 'map' function call as shown in line 12 in Figure 7.11b. In PostgreSQL, we call the `to_date` function and give it a date format.

In line 13, an estimated revenue for each booking is calculated using the number of minimum nights (contained in the listings dataset) multiplied by the price of each listing. This is only an estimate since the actual length of stay is not available in the datasets. The total estimated revenue for each listing is calculated by grouping the bookings with the same 'listing_id' and summing all of the estimated revenue from all of the bookings in the same group. The total estimated revenue is then appended back to the listings dataset. This is achieved by left-joining listings with the aggregated bookings on the 'listing_id' attribute.

In these example snippets, PolyFrame took advantage of lazy evaluation and did not send any queries to the database system for execution. This is because there are no actual results requested in this particular example. PolyFrame thus only generated nested SQL queries resulting from the data transformation functions applied to the dataframe object. On the other hand, Pandas executed the declared functions and materialized all of the results in memory at each step.

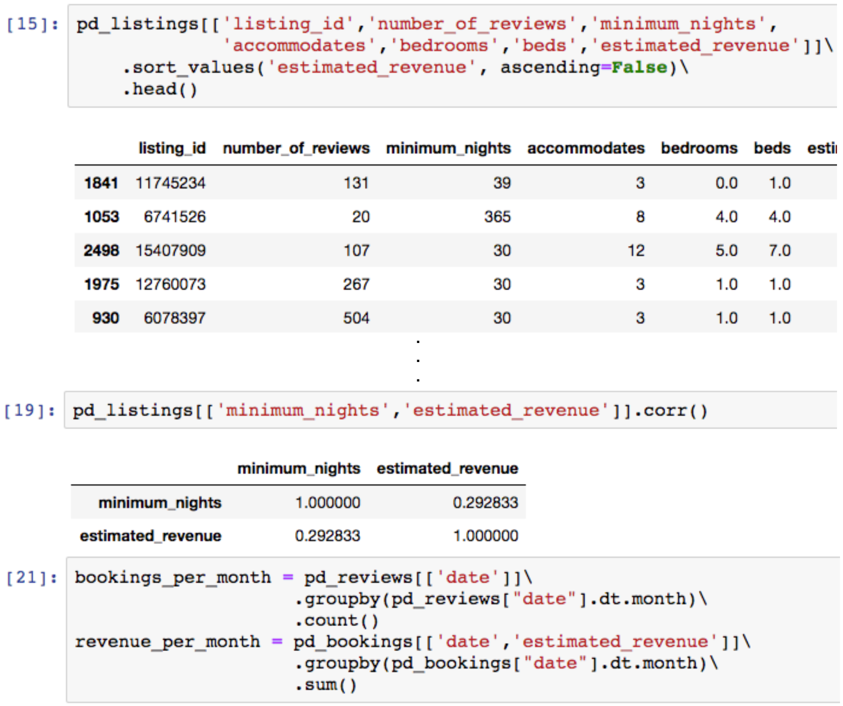
An important feature in PolyFrame is the ability to persist analysis results. Similar to how Pandas allows saving data as a file or a pickled object, PolyFrame allows data to be persisted as a new dataset (which can be temporary), a view, or a function to capture data transformation steps that can be applied to new records. Line 14 in Figure 7.11b is optional, but we included it here to show the syntax for PolyFrame's result persistence as a new dataset. We allow users to specify a new dataset name that they can then use in subsequent sessions if they would like to apply further analysis to previously prepared data.

7.3.4 Data Analysis

In analyzing the prepared data, the author of [1] groups the data and applies various aggregate functions along with a couple of machine learning models to extract important features that affect users' ratings. Figures 7.12 and 7.13 show a subset of functions applied during the data analysis stage. Figure 7.12a displays data analysis steps performed in Pandas dataframes and Figure 7.12b displays the same functions in PolyFrame. Figure 7.13 shows a part of the analysis that applies a machine learning model.

Line 15 in Figure 7.12 applies the `sort_values` function to the transformed listings dataset, and both PolyFrame and Pandas have the same syntax for this function.

As stated earlier, functions that cannot be implemented in PolyFrame will require data conversion to Pandas dataframes. In line 19 of Figure 7.12a the author applies the correlation function to compute the pairwise correlation between the 'minimum_nights' attribute and the 'estimated_revenue' attribute. This function is not supported in PolyFrame, so we converted the PolyFrame object to Pandas dataframe and applied the `corr` function to it to get the result in Figure 7.12b.



```
[15]: pd_listings[['listing_id','number_of_reviews','minimum_nights',
                   'accommodates','bedrooms','beds','estimated_revenue']]\
          .sort_values('estimated_revenue', ascending=False)\
          .head()
```

	listing_id	number_of_reviews	minimum_nights	accommodates	bedrooms	beds	esti
1841	11745234	131	39	3	0.0	1.0	
1053	6741526	20	365	8	4.0	4.0	
2498	15407909	107	30	12	5.0	7.0	
1975	12760073	267	30	3	1.0	1.0	
930	6078397	504	30	3	1.0	1.0	

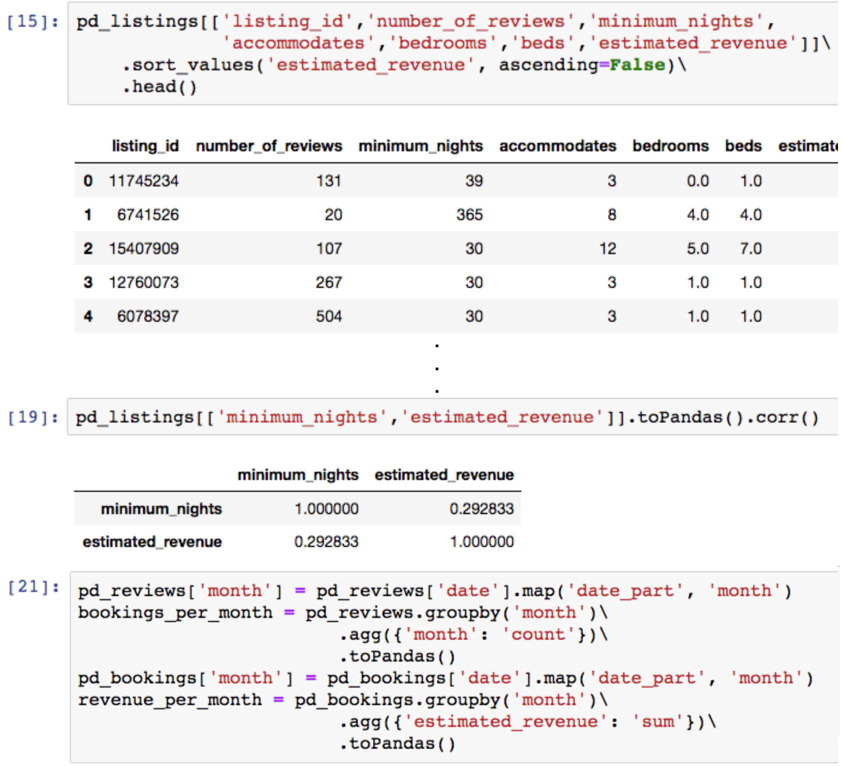
.
.
.

```
[19]: pd_listings[['minimum_nights','estimated_revenue']].corr()
```

	minimum_nights	estimated_revenue
minimum_nights	1.000000	0.292833
estimated_revenue	0.292833	1.000000

```
[21]: bookings_per_month = pd_reviews[['date']]\
                            .groupby(pd_reviews["date"].dt.month)\
                            .count()
      revenue_per_month = pd_bookings[['date','estimated_revenue']]\
                            .groupby(pd_bookings["date"].dt.month)\
                            .sum()
```

(a) Data Analysis in Pandas



```
[15]: pd_listings[['listing_id','number_of_reviews','minimum_nights',
                   'accommodates','bedrooms','beds','estimated_revenue']]\
          .sort_values('estimated_revenue', ascending=False)\
          .head()
```

	listing_id	number_of_reviews	minimum_nights	accommodates	bedrooms	beds	estimat
0	11745234	131	39	3	0.0	1.0	
1	6741526	20	365	8	4.0	4.0	
2	15407909	107	30	12	5.0	7.0	
3	12760073	267	30	3	1.0	1.0	
4	6078397	504	30	3	1.0	1.0	

.
.
.

```
[19]: pd_listings[['minimum_nights','estimated_revenue']].toPandas().corr()
```

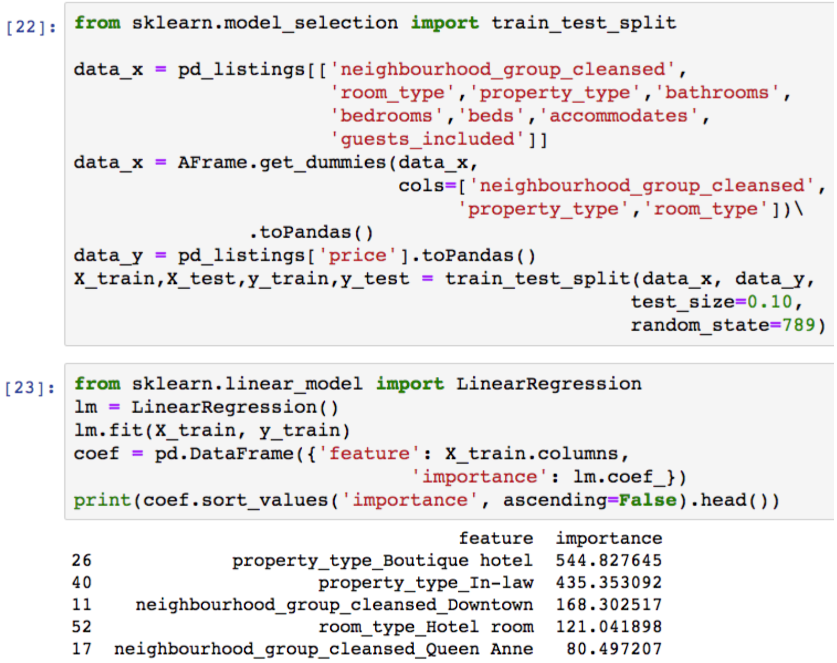
	minimum_nights	estimated_revenue
minimum_nights	1.000000	0.292833
estimated_revenue	0.292833	1.000000

```
[21]: pd_reviews['month'] = pd_reviews['date'].map('date_part', 'month')
      bookings_per_month = pd_reviews.groupby('month')\
                               .agg({'month': 'count'})\
                               .toPandas()
      pd_bookings['month'] = pd_bookings['date'].map('date_part', 'month')
      revenue_per_month = pd_bookings.groupby('month')\
                               .agg({'estimated_revenue': 'sum'})\
                               .toPandas()
```

(b) Data Analysis in PolyFrame

Figure 7.12: Data Analysis

In line 21, the author groups the data and applies aggregate functions. Prior to applying the `group_by` method, the 'date' attribute was converted to type datetime as shown earlier in line 12 in Figure 7.11. The month information is then extracted to be the grouping value . In PolyFrame, we utilize the `map` function to call PostgreSQL's built-in function `date_part` to extract the month from the datetime objects. The extracted month values were appended back as a new attribute called 'month'. At this point, we can then group the data on the 'month' attribute and apply the aggregate functions. Notice here that we also apply the `toPandas` method to initiate query execution and retrieve the values.



```
[22]: from sklearn.model_selection import train_test_split

      data_x = pd_listings[['neighbourhood_group_cleansed',
                            'room_type','property_type','bathrooms',
                            'bedrooms','beds','accommodates',
                            'guests_included']]
      data_x = AFrame.get_dummies(data_x,
                                  cols=['neighbourhood_group_cleansed',
                                        'property_type','room_type'])\
                     .toPandas()
      data_y = pd_listings['price'].toPandas()
      X_train,X_test,y_train,y_test = train_test_split(data_x, data_y,
                                                        test_size=0.10,
                                                        random_state=789)
```

```
[23]: from sklearn.linear_model import LinearRegression
      lm = LinearRegression()
      lm.fit(X_train, y_train)
      coef = pd.DataFrame({'feature': X_train.columns,
                           'importance': lm.coef_})
      print(coef.sort_values('importance', ascending=False).head())
```

```
                               feature  importance
26            property_type_Boutique hotel  544.827645
40                  property_type_In-law  435.353092
11  neighbourhood_group_cleansed_Downtown  168.302517
52                   room_type_Hotel room  121.041898
17 neighbourhood_group_cleansed_Queen Anne   80.497207
```

Figure 7.13: Applying Machine Learning Model

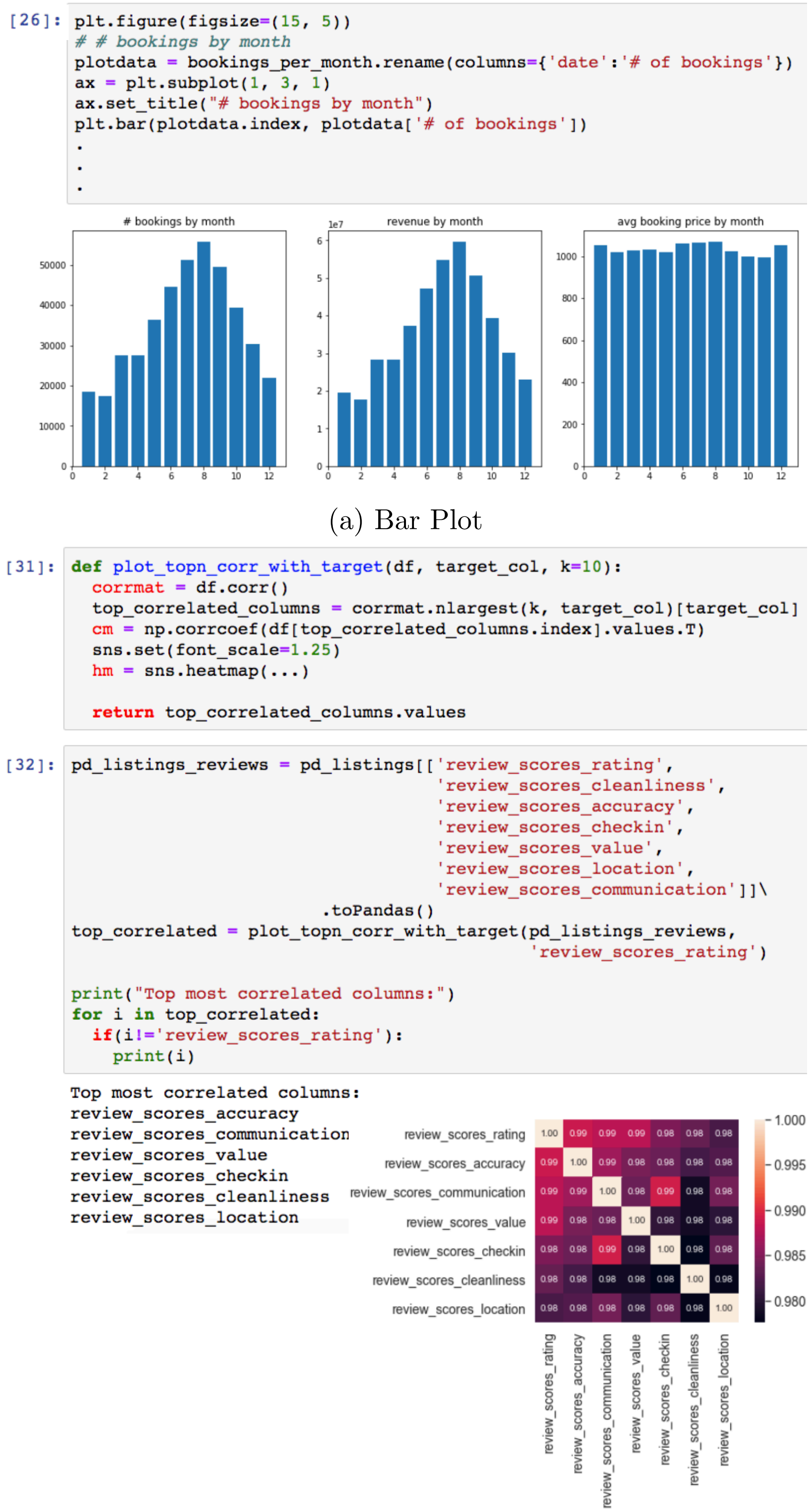
Figure 7.13 shows a snippet of the PolyFrame code for applying a linear regression model from Scikit-learn. First, the PolyFrame object is converted to Pandas dataframe. Line 22 shows two Pandas dataframe objects (data_x, data_y) being created by calling the 'toPandas' function on PolyFrame objects. The training data contains a subset of attributes projected from the original listings dataset and the target attribute is 'price'. After training the model, line 23 prints out the model's most important features that affect the listing price in sorted descending order of importance score.

During the data analysis stage, PolyFrame not only generated queries, but it also sent them to PostgreSQL for execution and returned results to the user. If the results from the data preparation stage had not been persisted as shown in line 14 in Figure 7.11b, all of the preparation functions applied earlier would also be part of the queries run in this analysis stage. Therefore, every operation that requires results would also have an overhead carried over from the data preparation stage. It is thus a good idea to persist the prepared data before proceeding with the analysis because the analysis process can be iterative and the overhead can be reduced significantly. In contrast, Pandas materializes results for each operation so the results from the data preparation stage are already available in memory prior to starting the analysis stage.

7.3.5 Data Visualization

In the data visualization stage, the analyzed data is visualized using various techniques such as bar plots, heatmaps, and coordinate maps. Figure 7.14 shows two plots from the notebook that apply matplotlib library functions to the previously analyzed dataframes. There are two main types of visualizations performed in this case study.

The first type is a summary visualization, which is performed on the summary of a dataset. Figure 7.14a shows three bar plots using data contained in the dataframes resulting from the data analysis step shown in Figure 7.12 line 21. The leftmost plot shows the number of bookings per month. The middle plot shows the total revenue per month. The last plot displays the average booking price per month calculated based on the previous two results. There are only twelve data points per plot, and those can be computed by the database system on any dataset regardless of its size. As a result, even when the visualization library does not work directly on PolyFrame, thus requiring Pandas materialization, the aggregated (materialized) results can fit comfortably in memory.



```
[26]: plt.figure(figsize=(15, 5))
      # # bookings by month
      plotdata = bookings_per_month.rename(columns={'date':'# of bookings'})
      ax = plt.subplot(1, 3, 1)
      ax.set_title("# bookings by month")
      plt.bar(plotdata.index, plotdata['# of bookings'])
      .
      .
      .
```

(a) Bar Plot

```
[31]: def plot_topn_corr_with_target(df, target_col, k=10):
        corrmat = df.corr()
        top_correlated_columns = corrmat.nlargest(k, target_col)[target_col]
        cm = np.corrcoef(df[top_correlated_columns.index].values.T)
        sns.set(font_scale=1.25)
        hm = sns.heatmap(...)

        return top_correlated_columns.values
```

```
[32]: pd_listings_reviews = pd_listings[['review_scores_rating',
                                          'review_scores_cleanliness',
                                          'review_scores_accuracy',
                                          'review_scores_checkin',
                                          'review_scores_value',
                                          'review_scores_location',
                                          'review_scores_communication']]\
                                .toPandas()
      top_correlated = plot_topn_corr_with_target(pd_listings_reviews,
                                                  'review_scores_rating')

      print("Top most correlated columns:")
      for i in top_correlated:
        if(i!='review_scores_rating'):
          print(i)
```

```
Top most correlated columns:
review_scores_accuracy
review_scores_communication
review_scores_value
review_scores_checkin
review_scores_cleanliness
review_scores_location
```

review_scores_rating 1.00 0.99 0.99 0.99 0.98 0.98 0.98
review_scores_accuracy 0.99 1.00 0.99 0.98 0.98 0.98 0.98
review_scores_communication 0.99 0.99 1.00 0.98 0.99 0.98 0.98
review_scores_value 0.99 0.98 0.98 1.00 0.98 0.98 0.98
review_scores_checkin 0.98 0.98 0.99 0.98 1.00 0.98 0.98
review_scores_cleanliness 0.98 0.98 0.98 0.98 0.98 1.00 0.98
review_scores_location 0.98 0.98 0.98 0.98 0.98 0.98 1.00
review_scores_rating
review_scores_accuracy
review_scores_communication
review_scores_value
review_scores_checkin
review_scores_cleanliness
review_scores_location
1.000
0.995
0.990
0.985
0.980

(b) Heat Map

Figure 7.14: Data Visualization

The other type of visualization is performed on a subset of the data. Figure 7.14b displays a code snippet that generates a heatmap. Line 31 creates a function that generates a heatmap given a dataframe and a target attribute. It computes and outputs the most correlated attributes from the dataframe and their correlation values to the target attribute. Line 32 projects 7 out of 106 attributes from the original listings dataset. All of the projected attributes only contain numerical values. In this case study, the target attribute is 'review_score_rating'.

Even though we have to convert from PolyFrame to Pandas dataframes and the data has to be transferred from the database system into memory for data visualization, utilizing data slicing (projection) is still more space and runtime-efficient than loading an entire dataset into memory. PolyFrame is able to take advantage of the database system's query processor still to lower the amount of transferred data.

7.3.6 End-to-end Performance Comparison with Pandas

In addition to evaluating the usability of PolyFrame by examining its functionality support through different stages in an analysis use case, a performance comparison is also important when choosing between different frameworks. Our goal for PolyFrame is not to replace Pandas dataframe but to work along side it to provide much needed scalability and ease-of-use when operating on large datasets. As a result, PolyFrame-to-Pandas conversion can be applied for operations that cannot be completed when using PolyFrame alone, as we have seen. The goal of this experiment is to illustrate the performance efficiency of PolyFrame when applying to a real analysis use case. We use Pandas as our baseline comparison.

In order to present a reproducible evaluation environment, we set up our experiments on an Amazon m4.large EC2 machine with 8 GB of memory and 200 GB of SSD. The data used for this experiment is obtained from [11]. We use the same two datasets that were used in the case study notebook, but instead of using one city (seattle) we obtained data from multiple cities to evaluate the scalability of our framework. In this way we conducted the experiment over datasets with an increasing number of records. The number of records in the listings and reviews datasets are displayed in Tables 7.2 and 7.3, respectively, and are summarized below.

- **Listings:** The dataset has 106 attributes and it contains data from 1 - 60 different cities. The total csv file size of the dataset ranges from 30 MB - 3 GB. The listings dataset has attribute 'id' as its primary key.
- **Reviews:** The dataset has 6 attributes and it contains data ranging from 1 - 60 different cities. The total csv file size of the dataset ranges from 250 MB - 6.8 GB. The reviews dataset has attribute 'id' as its primary key and attribute 'listing_id' that refers to a particular property in the listings dataset.

Number of cities	**CSV file size**	**Number of records**
1	87 MB	36,905
10	489 MB	191,205
20	932 MB	389,227
30	1.1 GB	456,300
40	1.5 GB	632,277
50	2 GB	827,411
60	3.1 GB	988,451

Table 7.2: Listings Dataset Summary

Number of city	**CSV file size**	**Number of records**
1	271 MB	836,586
10	1.3 GB	4,194,878
20	2.4 GB	8,319,765
30	3 GB	9,881,517
40	4.1 GB	13,503,629
50	5.5 GB	17,863,880
60	6.8 GB	21,602,632

Table 7.3: Review Dataset Summary

We conducted the experiment on a single ec2 machine and on datasets with increasing number of records. Since the website provides downloadable data in a form of CSV files labeled by city names, we obtained the data by increasing the number of cities that we downloaded. As a result, the sizes of the datasets in each incremental evaluation are not equivalent, as each city contains different numbers of listings and reviews. We used a database system to ingest the data like to mimic a real-world data ingestion and storage scenario. Storing large amounts of data as files in a file system is not efficient because selecting even a small subset of the data would require a complete file scan. In addition, Pandas also supports creating dataframes from data stored in SQL database systems via SQLAlchemy [18]. We choose PostgreSQL v.12 as our database to ingest the data because PostgreSQL is a readily-available and popular single node database system that can be installed easily. We created Pandas dataframes by connecting it to the tables in PostgreSQL, so both PolyFrame and Pandas consumed data from the same database system.

In Pandas, users can persist analysis data as a new file, which can be inefficient and inconvenient when the dataset is large. In PolyFrame, analysis data can be persisted as a new dataset, a temporary dataset, or as a view. These result persistence modes are provided to meet flexible application requirements. Depending on the nature of the analysis, it can be beneficial to persist prepared data as a new data collection if the data will be analyzed using different methods. It can also be convenient and efficient to save the data preparation steps as a view, if the analysis is often done on live data, to get the most up-to-date results. To provide insights into the different persistence modes, we conducted variations of our experiment on three variants of PolyFrame. Regular PolyFrame operations performed on the fly without any result persistence are labeled 'PolyFrame'. 'PolyFrame-view' refers to persisting the prepared data as a view prior to starting the data analysis stage. 'PolyFrame-table' refers to persisting the prepared data as a new table prior to starting the data analysis stage. (This point in the end-to-end use case is where the variations occurred.)

7.3.7 Evaluation Results

Here we present the results from our end-to-end performance evaluation. First, we present the overall case study performance followed by the scalability result from each of the data analysis stages. We then discuss the results in detail.

Overall

We setup timing points in each of the analysis stages and executed the case study analysis in its entirety. The overall performance evaluation of both PolyFrame and Pandas is displayed in Figure 7.15. For this experiment we executed the stages on the datasets that contain information for 20 cities. This was the largest number of cities that can be analyze using Pandas dataframes on the hardware that we used.

In both data acquisition and preparation stages, Pandas suffered from eagerly evaluating all of the operations and materializing intermediate results in memory. On the other hand, PolyFrame benefited from lazy evaluation and was able to delay the execution of operations until results are required.

During the data analysis stage, Pandas performed better than PolyFrame because data from the data preparation stage was already available in memory, while PolyFrame had to send queries for execution and retrieve results. However, not all variants of PolyFrame have the same runtime. This is because persisting the data from the data preparation stage as a table (PolyFrame-table) prior to starting the analysis can significantly reduce the runtime when performing the analysis steps.

In the last step the materialized results from the analysis stage are plotted using different techniques to visualize the data summary. At this stage, all of PolyFrame's results are converted to Pandas dataframes for use by the visualization package. As a result, the runtimes of all variants of PolyFrame in the data visualization stage are comparable to that of Pandas.

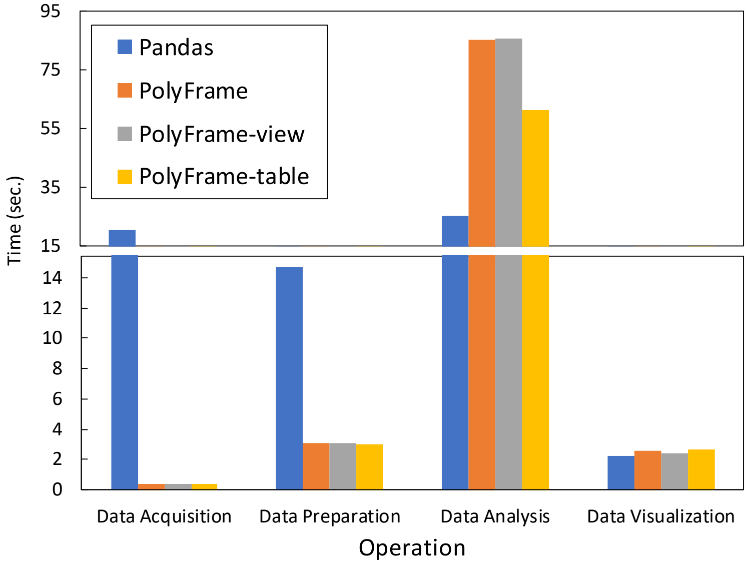


Figure 7.15: Overall Result

Scalability Evaluation

In order to evaluate PolyFrame's scalibility benefits, we also measured the total runtime of this end-to-end case study over datasets with increasing numbers of records. Figure 7.16 displays the end-to-end scalability timing results for both PolyFrame and Pandas. Pandas performed the best on the smallest dataset size. However, after increasing the number of records in the datasets, Pandas performed worse than all variants of PolyFrame. This is because the time required to load both datasets into memory and transform the data in the data preparation stage surpassed the time that it takes to perform the actual data analysis steps. On the 20-cities datasets, PolyFrame began having a better runtime than Pandas, and when PolyFrame's data was persisted (PolyFrame-table) before starting the analysis, the runtime difference between Pandas and PolyFrame is even more evident. On datasets that contain data from more than 20 cities, Pandas failed to join the two datasets, as it was not able to materialize the joined results in memory. As a result, we do not have Pandas evaluation runtimes on subsequent dataset sizes. In contrast, PolyFrame was able to finish the end-to-end analysis on all of the tested dataset sizes.

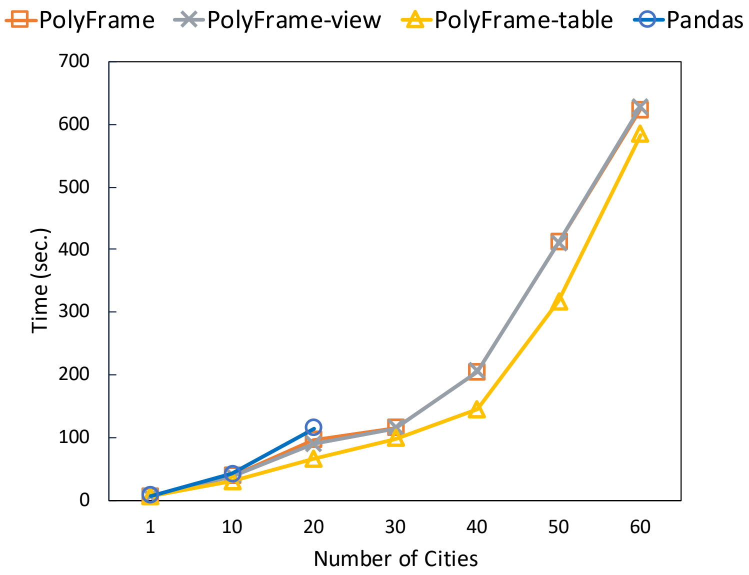


Figure 7.16: End-to-end Scalability Result

Next, we examine the scalability evaluation results in each of the stages in the analysis from data acquisition through to data visualization.

Data Acquisition : Figure 7.17 displays the runtime comparisons between PolyFrame and Pandas dataframes during the data acquisition stage of the analysis. In this experiment, both frameworks were connected to the same PostgreSQL instance. For this case study, both frameworks have to load two datasets: listings and reviews. Pandas performed worse than all variants of PolyFrame due to having to eagerly read the entire datasets into memory. PolyFrame only had to make a connection to the underlying database system to check for the existence of the indicated tables and then build (but not run) the initial query. As a result, PolyFrame's runtime is constant across all dataset sizes because it does not load any data into memory during object initialization, while Pandas' runtime increases when increasing the size of the datasets. Pandas also ran out of memory on moderate-sized data, e.g., when the experiment reached 40 cities with the combined data size of 5.6GB as indicated in Tables 7.2 and 7.3.

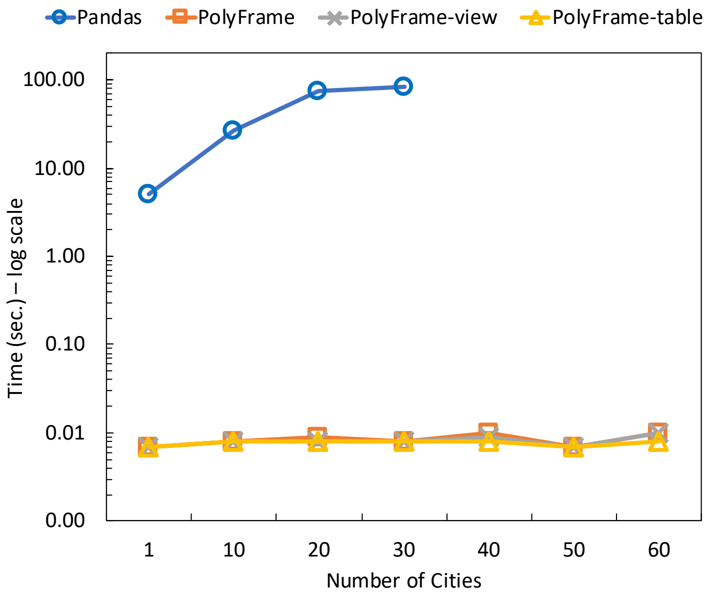


Figure 7.17: Data Acquisition Result

Data Preparation : During the data preparation stage, the case study author's notebook examines values in both datasets by calling aggregate functions on some of the attributes before performing multiple data transformation steps on both datasets. The example operations involve selecting a subset of attributes from the listings and reviews datasets, filling null values, formatting values in some of the selected attributes, and joining listings with reviews to create booking data and calculate total revenue.

Figure 7.18 shows the data preparation performance of PolyFrame and Pandas. Pandas failed to join the two datasets when we increased the number of cities to 30 and threw an out-of-memory error. PolyFrame was able to take advantage of lazy evaluation for most of the data transformation operations and only sent the queries to PostgreSQL for the operations where the user requests results. As shown in Figure 7.11, the data transformation steps listed there do not trigger query execution in PolyFrame. Only nested SQL queries were generated as a result of these operations. In addition, PolyFrame was able to take advantage of PostgreSQL's query optimizer and the ability to join data larger than the available memory. As a result, PolyFrame's performance was much better than Pandas' and it was able to complete the preparation steps on all of the dataset sizes tested.

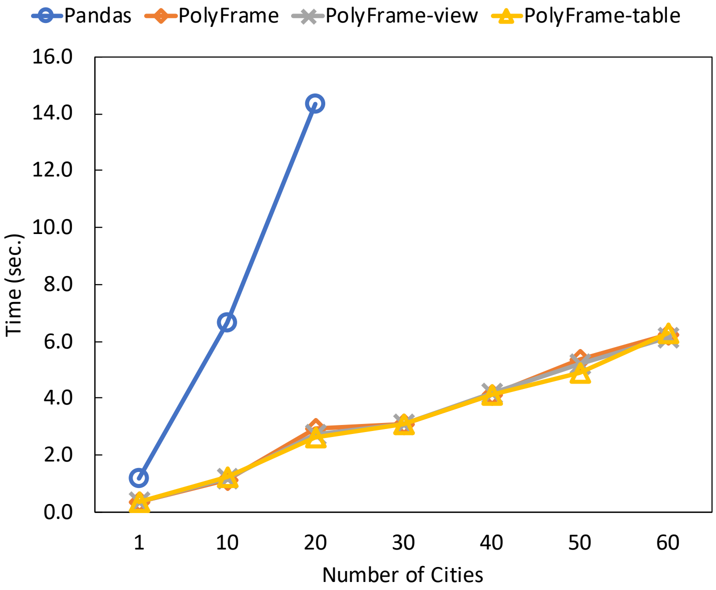


Figure 7.18: Data Preparation Result

Data Analysis : In the data analysis stage, several aggregate functions are performed on different grouping attributes to answer questions that the author listed as objectives. The operations performed during the analysis stage triggered query execution in PolyFrame and had to return results. In order to provide users with analysis flexibility, PolyFrame provides multiple result persistence options, as previously mentioned. Here we evaluated two alternative result persistence modes. We persisted the data resulting from the data preparation stage as a view (PolyFrame-view) and also as a new table (PolyFrame-table) prior to starting the analysis.

Figure 7.19 shows the runtime results during data analysis for Pandas and PolyFrame. Since Pandas was not able to complete the data preparation steps on datasets with more than 20 cities, it was not able to deliver results in the subsequent stages including the analysis stage.

Regular PolyFrame (no result persistence) and PolyFrame operating on a view (PolyFrame-view) have the same runtime in this experiment, and both are slower than Pandas on all of the dataset sizes that Pandas was able to operate on. However, if we persist the prepared data resulting from the data preparation steps as a new table (PolyFrame-table), the PolyFrame runtime results are comparable to Pandas. Also, even though PolyFrame's performance was initially slower than Pandas, it was able to complete the experiment on all of the dataset sizes, including a combined data size (60 cities) that is larger than the available memory.

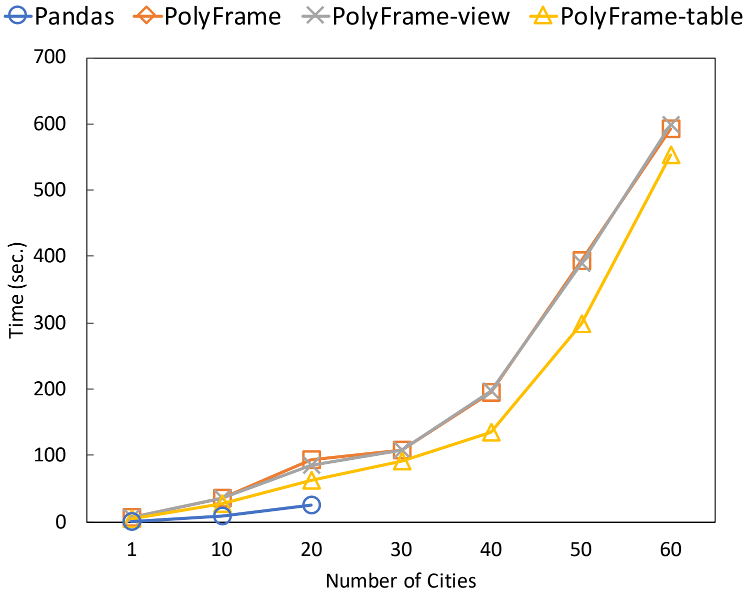


Figure 7.19: Data Analysis Result

Data Visualization : Results from the data analysis steps are visualized using various plots such as bar graphs, heat maps, and location maps. All of the visualization steps are performed after the PolyFrame objects have been converted to Pandas dataframes in the data analysis stage.

Figure 7.20 displays the experiment results comparing PolyFrame with Pandas dataframe during the data visualization portion of the case study. As mentioned before, Pandas was not able to complete the data preparation steps on the dataset containing more than 20 cities. As a result, there was no data to be analyzed or visualized in the later stages of the analysis. In contrast, even though the PolyFrame objects have to be converted to Pandas dataframes in order to apply the visualization functions, it was then able to complete the visualization stage successfully. This is because the visualization functions are either applied to an aggregated summary of the datasets or to a subset (slice) of the datasets' attributes. Since PolyFrame uses lazy evaluation it was able to utilize PostgreSQL's query processor to avoid complete table loading into memory and efficiently to compute data summaries even on source data that is larger than the available memory. It was also able to project only the attributes that were required for analysis and visualization.

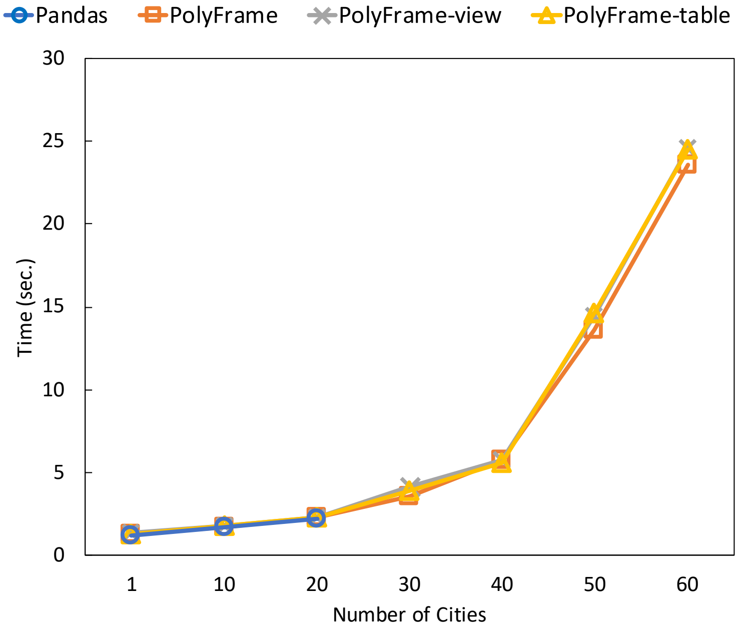


Figure 7.20: Data Visualization Result

7.3.8 Discussion of Experiments

We conducted these performance evaluation experiments on Pandas to get a performance baseline and to better understand the benefits and limitations of PolyFrame in comparison to Pandas. We measured the end-to-end EDA runtime needed to complete an exploratory data analysis and also separately measure the runtime in each of the analysis stages.

In the data acquisition and preparation processes, the benefit of PolyFrame's lazy evaluation is prominent. In the data acquisition stage, we did not have to wait for the data to be loaded entirely into memory prior to starting the analysis. The exact syntax of PolyFrame object creation is different from Pandas, but the database connection string provided is essentially the same. During data preparation, PolyFrame not only benefitted from lazy evaluation, but it was also able to take advantage of a query optimizer, allowing it to use an index to efficiently join the datasets and to avoid materializing intermediate results in memory. As a result, PolyFrame was able to operate on large datasets efficiently.

Exploratory data analysis is often iterative and sequential in terms of the stages in the project lifecycle. As a result, when a framework cannot complete one of the steps in the early stages, it cannot proceed to the subsequent stages. In this case study, Pandas failed to complete data preparation on moderate-sized data because it cannot accommodate the materialization of joined results in memory. This in turn prevented it from completing the data analysis and visualization because the analysis stage relies on the data having been transformed. In contrast, PolyFrame operating on PostgreSQL was able to take advantage of the database query optimizer avoided having to materialize a complete join result in memory. It was thus able to complete the analysis process on all data sizes. Even in the data visualization stage, in which PolyFrame objects have to be converted into Pandas dataframes, it was still possible for PolyFrame to complete the EDA use case on data larger than the available memory. This is because the visualization functions are applied to data summaries and a subset of the attributes, not to the whole dataset.

Although our experiments were only performed using a single machine for processing, the scalability results in each of the analysis stages have shown that PolyFrame is able to operate on data larger than the available memory.

7.4 Conclusion

In this chapter, we have demonstrated the practicality of using PolyFrame to perform two data analyses and examined its performance in each of the stages in the analysis life cycle. These case studies have helped us identify missing features that help simplify end-to-end data analysis on large datasets such as result persistence. PolyFrame's ability to communicate with a diverse set of database systems (e.g., RDBMS, NoSQL, and graph) while delivering the data scientists' familiar dataframe syntax allows it to accommodate big data analyses on various types of data efficiently without requiring extensive distributed system knowledge from end users.

We have compared PolyFrame's performance against the Pandas dataframe baseline. The results clearly show that lazy evaluation and a query optimizer can play an important role in the analysis of large datasets even in a local setting. Also, even in a case when Pandas data conversion is needed to apply a data visualization library, PolyFrame continued to efficiently deliver needed results and completed the analysis. This is because result visualization is often applied on data summaries that can be efficiently computed by database systems. Finally, while our focus here has been on PolyFrame, we expect that the benefits illustrated in this chapter also apply to other database-backed dataframe libraries that rely on database systems' data management capabilities.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this dissertation, we have presented a query-based approach to scale dataframe operations to large volumes of data by utilizing lazy evaluation and incrementally constructing database queries.

We began by briefly examining the benefits and drawbacks of existing approaches to scaling Pandas dataframes in Chapter 3.

Chapter 4 explained our design of AFrame, a scalable data analytic library that provides a Pandas-like interface on top of a big data management system. We have outlined the architecture of AFrame and described its incremental query formation process in detail through each of the stages in a data analysis lifecycle. We have demonstrated through our experimental evaluation that AFrame's lazy evaluation is beneficial in allowing it to take advantage of AsterixDB's query optimizer and indexes to efficiently operate on data at scale.

Chapter 5 outlined our dataframe benchmark that we used to evaluate AFrame performance.

We have shown that our benchmark can be used in both local and distributed settings. We compared AFrame with multiple dataframe libraries. The evaluation results showed that AFrame can operate competitively in both settings. We have also demonstrated that query optimizations can be crucial when dealing with data at scale.

In Chapter 6, we have described the language-independent version of AFrame, PolyFrame. PolyFrame retargets AFrame's incremental query formation process to operate on other query-based database systems. We explained its two-level query rewriting process and how the implementation supports additional composable query languages through a configuration template and regular expression pattern matching. We compared PolyFrame versus Spark reading from the same database systems through a set of analytical benchmark operations. The experimental results highlighted the benefits of deferring to database systems for query processing over utilizing an external compute engine.

In Chapter 7, we illustrated the usability and efficacy of using PolyFrame to perform two end-to-end data analyses. We explained the different result persistence modes in PolyFrame and how it could play an important role in reducing the system's runtime during data analysis. By comparing PolyFrame's performance against the Pandas dataframe baseline, we have highlighted the general benefits of database-backed dataframes during data preparation, analysis, and visualization.

8.2 Future Work

PolyFrame still has a lot of room for improvement. In its current state, PolyFrame has shown promising results in allowing data scientists to interact with large datasets without requiring extensive distributed systems knowledge. Future studies in partial result caching could allow database-backed dataframes to avoid recomputation on frequently used operations especially during exploratory data analyses where operations are often performed iteratively on the same datasets.

In order to support more of the large number of functions available in Pandas dataframe, the important notion of row index in Pandas dataframe needs to be addressed. An important research question being explored in other similar dataframe libraries is how to support Pandas' notion of index-based row labeling efficiently in a distributed environment. There is not yet an efficient solution to enable row-indexing on unordered data. 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.

Last but not least, recall that Petersohn et al. [54] identified three types of functions in the Pandas dataframe library - relational algebra, linear algebra, and spreadsheet functions. PolyFrame's focus has been on support for the relational algebra portion of the library, as it relies on general features supported by database systems under the hood. An interesting longer term question would be how one could perhaps expand the PolyFrame system to cover additional areas of Pandas' functionality.

Bibliography

[1] Airbnb in Seattle. https://towardsdatascience.com/airbnb-in-seattle-data-analysis-8222207579d7.

[2] Apache Arrow and the 10 Things I Hate About pandas. https://wesmckinney.com/blog/apache-arrow-pandas-internals/.

[3] Apache asterixdb. https://asterixdb.apache.org/.

[4] Apache Hive. https://hive.apache.org/.

[5] Apache Parquet. https://parquet.apache.org/.

[6] Apache Spark. http://spark.apache.org/.

[7] Apache Tez. http://tez.apache.org/.

[8] CrowdFlower. http://www.crowdflower.com/.

[9] Dask. http://dask.org/.

[10] GraySort benchmark. http://sortbenchmark.org/.

[11] Inside airbnb. http://insideairbnb.com/get-the-data.html.

[12] Kaggle. http://www.kaggle.com/crowdflower/twitter-airline-sentiment/.

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

[14] Numpy. http://numpy.org/.

[15] Pandas. http://pandas.pydata.org/.

[16] Pandas on Ray. https://rise.cs.berkeley.edu/blog/pandas-on-ray-early-lessons/.

[17] Police department incident reports. http://data.sfgov.org/.

[18] SQLAlchemy. https://www.sqlalchemy.org/.

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

[20] IBIS. `https://ibis-project.org/`, 2021.

[21] Koalas. `http://koalas.readthedocs.io`, 2021.

[22] Mongodb. `http://mongodb.com/`, 2021.

[23] Mongodb aggregation. `https://docs.mongodb.com/manual/aggregation/`, 2021.

[24] Neo4j. `http://neo4j.com/`, 2021.

[25] PostgreSQL. `http://www.postgresql.org/`, 2021.

[26] Spark data sources. `https://spark.apache.org/docs/latest/sql-data-sources.html`, 2021.

[27] Vertica. `https://www.vertica.com`, 2021.

[28] 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.

[29] S. Alsubaiee et al. AsterixDB: a scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[30] M. Armbrust et al. Scaling Spark in the real world: performance and usability. *PVLDB*, 8(12):1840–1843, 2015.

[31] M. Armbrust et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[32] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[33] M. J. Carey. AsterixDB mid-flight: A case study in building systems in academia. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1–12. IEEE, 2019.

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

[35] D. Chamberlin. SQL++ for SQL Users: A Tutorial. September 2018. *Available via Amazon.com*, 2018.

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

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

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

[39] S. Hagedorn, S. Kläbe, and K.-U. Sattler. Putting pandas in a box. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[40] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, 2013.

[41] S. Jahangiri. shivajah/JSON-Wisconsin-Data-Generator, Dec. 2020.

[42] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, et al. Magpie: Python at speed and scale using cloud backends. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[43] 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.

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

[45] 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.

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

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

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

[49] 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.

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

[51] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Springer Publishing Company, Incorporated, 4th edition, 2019.

[52] J. Partner, A. Vukotic, and N. Watt. *Neo4j in action.* Manning Publications Co., 2013.

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

[54] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):2033–2046, 2020.

[55] D. A. Schmidt. *The structure of typed programming languages.* MIT press, 1994.

[56] P. Sinthong and M. J. Carey. AFrame: Extending DataFrames for large-scale modern data analysis. In *IEEE International Conference on Big Data (Big Data)*, pages 359–371, 2019.

[57] 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.

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

Appendix A

AsterixDB DDL

A.1 AsterixDB Twitter Feed

```
CREATE FEED TwitterFeed with {
   "adapter-name" : "push_twitter",
   "type-name" : "Tweet",
   "format" : "twitter-status",
   "consumer.key" : "***",
   "access.token" : "***",
   "access.token.secret" : "***"
 };
CONNECT FEED TwitterFeed TO LiveTweets;
START FEED TwitterFeed;
```

Figure A.1: Create a Twitter feed to collect tweets.

A.2 AsterixDB Dataframe Benchmark DDL

```
CREATE TYPE ClosedType AS CLOSED{
    unique1: int64,
    unique2: int64,
    unique3: int64,
    two: int64,
    four: int64,
    ten: int64,
    twenty: int64,
    onePercent: int64,
    tenPercent: int64,
    twentyPercent: int64,
    fiftyPercent: int64,
    evenOnePercent: int64,
    oddOnePercent: int64,
    stringu1: string,
    stringu2: string,
    string4: string
};

CREATE TYPE OpenType AS {unique2: int64};

CREATE DATASET ClosedData(ClosedType)
PRIMARY KEY unique2
WITH {"storage-block-compression":
    {"scheme": "snappy"}};

CREATE DATASET OpenData(OpenType)
PRIMARY KEY unique2
WITH {"storage-block-compression":
    {"scheme": "snappy"}};
```

Figure A.2: Create datatypes and datasets to use for benchmarking

Appendix B

PolyFrame Translated Queries and Rewrite Rules

B.1 PolyFrame Translated Queries

Pandas Dataframe Expression

```
df[df[‘lang’] == ‘en’][[‘name’, ‘address’]].head(10)
```

SQL++ Translation

```
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;
```

SQL Translation

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

MongoDB Query Language Translation

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

Cypher Translation

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

B.2 Sample Language-specific Rewrite Rules for Cypher

```
;Below are query explanations
;q1: select all records from a collection
;q2: project an attribute
;q3: return boolean statement (e.g., id > 5)
```

```
;q4: return total count of records

;q5: sort records based on an attribute in descending order

;q6: sort records based on an attribute in ascending order

;q7: group records based on an attribute ($grp_by_attribute)

;q8: return an aggregate value ($agg_func) of each group

;q9: return all attributes with one new added attribute

;q10: return distinct values of an attribute

;q11: drop an attribute

;q12: inner join with another collection

;q13: left outer join with another collection

;q14: return an aggregate value of an attribute

;q15: select all records from a view

;q16: return one record from each group

;q17: return records from groups that have only one record

;q18: return an unnest attribute along with other existing attributes


[QUERIES]

q1 = MATCH(t: $collection)

q2 = $subquery

    WITH t{$attribute_alias}

q3 = $subquery

    WITH t WHERE $statement

q4 = $subquery

    RETURN COUNT(*) AS t

q5 = $subquery

    WITH t ORDER BY $sort_desc_attr DESC

q6 = $subquery

    WITH t ORDER BY $sort_asc_attr

q7 =

q8 = $subquery

    WITH {$grp_by_attribute, $agg_value} AS t

q9 = $subquery

    WITH t{.*, $attribute_alias}
```

```
q10 = $subquery
     WITH DISTINCT($attribute) AS t
     RETURN t
q11 = $subquery
     WITH apoc.map.removeKeys(t {.*}, [$attribute_remove]) AS t
q12 = $subquery
     MATCH (t),($r_alias:$other)
     WHERE t.$left_on = $r_alias.$right_on
     WITH t{.*, $r_alias}
q13 = $subquery
     OPTIONAL MATCH (t),($r_alias:$other)
     WHERE t.$left_on = $r_alias.$right_on
     WITH t{.*, $r_alias}
q14 = $subquery
    WITH {$agg_value} AS t
q15 =
q16 = $subquery
     WITH {$grp_by_attribute} as grp_key , collect(t) as t
     WITH t[0] AS t
q17 = $subquery
     WITH {$grp_by_attribute} as grp_key , collect(t) as t
     WHERE length(t) = 1
     WITH t[0] AS t
q18 = $subquery
     UNWIND t.$attribute AS $alias
     WITH t{.*, $alias:$alias}


[ATTRIBUTE ALIAS]
single_attribute = t.$attribute
attribute_remove = ‘$attribute’
attribute_alias = ‘$alias‘: $attribute
attribute_project = ‘$attribute‘:t.$attribute
rename = ‘$new_attribute‘: $old_attribute
```

```
agg_value = ‘$agg_func_$attribute‘: $func
attribute_separator = $left, $right
sort_asc_attr = t.$attribute
sort_desc_attr = t.$attribute
grp_by_attribute = ‘$attribute‘: t.$attribute
grp_value = ‘$attribute‘
str_format = "$value"

[ARITHMETIC STATEMENTS]
add = $left + $right
sub = $left - $right
mul = $left * $right
div = $left / $right
mod = $left %% $right
pow = $left ^ $right

[LOGICAL STATEMENTS]
and = $left AND $right
or = $left OR $right
not = NOT $left

[COMPARISON STATEMENTS]
eq = $left = $right
ne = $left != $right
gt = $left > $right
lt = $left < $right
ge = $left >= $right
le = $left <= $right
isna = $left IS NULL
notna = $left IS NOT NULL
isin = $left IN [$right]

[TYPE CONVERSION]
```

```
to_str = apoc.convert.toInteger($statement)

to_int = apoc.convert.toInteger($statement)

to_float = apoc.convert.toFloat($statement)


[LIMIT]

limit = $subquery

       RETURN t

       LIMIT $num

return_all = $subquery

            RETURN t

sample_size = 1000


[ESCAPE CHARACTERS]

escape = [‘"]


[FUNCTIONS]

min = min(t.$attribute)

max = max(t.$attribute)

avg = avg(t.$attribute)

std = stDevP(t.$attribute)

count = count

sum = sum(t.$attribute)

abs = abs(t.$attribute)

fillna = CASE $attribute WHEN NULL THEN $value ELSE $attribute END

replace = CASE $statement WHEN TRUE THEN $to_replace ELSE $attribute END

function_format = $function($attribute)

function_arg_format = $function($attribute, $argument)


[SAVE RESULTS]

to_collection = $subquery

              CREATE (n:$collection)

              SET n = t

to_view =
drop_collection = MATCH(t: $collection)
                DELETE t
```

B.3 Sample Language-specific Rewrite Rules for MongoDB

```
[QUERIES]
q1 = { "$match": {} }
q2 = $subquery,
    { "$project": { $attribute_alias } }
q3 = $subquery,
    { "$match": { "$expr": { $statement } } }
q4 = $subquery,
    { "$count": "count" }
q5 = $subquery,
    { "$sort": { $sort_desc_attr } }
q6 = $subquery,
    { "$sort": { $sort_asc_attr } }
q7 = $subquery,
   { "$group": { "_id": { $grp_by_attribute } } }
q8 = $subquery,
   { "$group": { "_id": { $grp_by_attribute }, $agg_value } },
   { "$addFields": { $grp_value } }
q9 = $subquery,
   { "$set": { $attribute_alias } }
q10 = $subquery,
   { "$group" : { "_id" : "$$attribute" } }
q11 = $subquery,
   { "$project": { $attribute_remove } }
q12 = $subquery,
     { "$lookup" : { "from" : "$other", "as" : "$other", "let": {"left": "$$left_on"},
       "pipeline": [ $right_query, { "$match": { "$expr":
                     { "$eq": ["$$right_on", "$$left"] } } } ] } },
     { "$unwind" : { "path" : "$$other", "preserveNullAndEmptyArrays" : false } },
     { "$replaceRoot" : { "newRoot" : { "$mergeObjects" : [ "$$other", "$$ROOT" ] } } },
     { "$project": { "$other": 0 } }
q13 = $subquery,
     { "$lookup" : { "from" : "$other", "as" : "$other", "let": {"left": "$$left_on"},
       "pipeline": [ $right_query, { "$match": { "$expr":
                     { "$eq": ["$$right_on", "$$left"] } } } ] } },
     { "$unwind" : { "path" : "$$other", "preserveNullAndEmptyArrays" : true } },
     { "$replaceRoot" : { "newRoot" : { "$mergeObjects" : [ "$$other", "$$ROOT" ] } } },
     { "$project": { "$other": 0 } }
q14 = $subquery,
     { "$group": { "_id": {}, $agg_value } },
     { "$project": {"_id": 0 } }
q15 = { "$project": { "_id": 0 } }
q16 = $subquery,
     { "$group": { "_id": { $grp_by_attribute }, "first": {"$first": "$$ROOT"} } },
     { "$replaceRoot": { "newRoot": "$first" } }
q17 = $subquery,
     { "$group": { "_id": { $grp_by_attribute }, "cnt": {"$sum": 1},
       "first": {"$first": "$$ROOT"} } },
     { "$match": { "$expr": { "$eq": ["$cnt", 1] } } },
     { "$replaceRoot": { "newRoot": "$first" } }
q18 = $subquery,
     { "$set": { "$attribute_alias": "$$attribute" } },
     { "$unwind": { "path": "$$attribute_alias", "preserveNullAndEmptyArrays": true } }

[ATTRIBUTES]
single_attribute = $attribute
attribute_alias = "$alias": { $attribute }
attribute_remove = "$attribute": 0
attribute_project = "$attribute": 1
```

```
sort_asc_attr = "$attribute": 1
sort_desc_attr = "$attribute": -1
rename = "$new_attribute": "$$old_attribute"
agg_value = "$agg_func_$attribute": { $func }
attribute_separator = $left, $right
grp_by_attribute = "$attribute": "$$attribute"
grp_value = "$attribute": "$_id.$attribute"
str_format = "$value"

[ARITHMETIC STATEMENTS]
add = "$add": [ "$$left", $right ]
sub = "$subtract": [ "$$left", $right ]
mul = "$multiply": [ "$$left", $right ]
div = "$divide": [ "$$left", $right ]
mod = "$mod": [ "$$left", $right ]
pow = "$pow": [ "$$left", $right ]
norm_div = $left

[LOGICAL STATEMENTS]
and = "$and": [ { $left }, { $right } ]
or = "$or": [ { $left }, { $right } ]
not = "$not": [ { $left } ]

[COMPARISON STATEMENTS]
eq = "$eq": ["$$left", $right]
ne = "$ne": ["$$left", $right]
gt = "$gt": ["$$left", $right]
lt = "$lt": ["$$left", $right]
ge = "$gte": ["$$left", $right]
le = "$lte": ["$$left", $right]
isna = "$lt": ["$$left", null]
notna = "$gt": ["$$left", null]
isin = "$in": ["$$left",[$right]]
```

```
[TYPE CONVERSION]
to_int32 = "$toInt": { $statement }
to_int64 = "$toLong": { $statement }
to_str = "$toString": { $statement }
to_double = "$toDouble": { $statement }

[LIMIT]
limit = $subquery,
       { "$project": { "_id": 0 } },
       { "$limit" : $num }
return_all = $subquery
sample_size = 1000

[FUNCTIONS]
min = "$min": "$$attribute"
max = "$max": "$$attribute"
avg = "$avg": "$$attribute"
std = "$stdDevSamp": "$$attribute"
abs = "abs": "$$attribute"
count = "$sum": 1
sum = "$sum": "$$attribute"
fillna = "$ifNull": [ "$$attribute", $value ]
replace = "$cond": { "if": { $statement }, "then": $to_replace, "else": "$$attribute" }
function_format = "$$function": "$$attribute"
function_arg_format = "$$function": {"input": "$$attribute", $argument}
kwarg = "$$key": $value

[SAVE RESULTS]
to_collection = $subquery,
   { "$out": "$collection" }
to_view = CREATE FUNCTION $namespace.$collection()
         {$subquery};
```

B.4 Sample Language-specific Rewrite Rules for SQL

```
[QUERIES]
q1 = SELECT * FROM $collection
q2 = SELECT $attribute_alias FROM ($subquery) t
q3 = SELECT * FROM ($subquery) t WHERE $statement
q4 = SELECT COUNT(*) FROM ($subquery) t
q5 = SELECT * FROM ($subquery) t ORDER BY $sort_desc_attr DESC
q6 = SELECT * FROM ($subquery) t ORDER BY $sort_asc_attr ASC
q7 = SELECT * FROM ($subquery) t GROUP BY $grp_by_attribute
q8 = SELECT $grp_by_attribute, $agg_value FROM ($subquery) t GROUP BY $grp_by_attribute
q9 = SELECT t.*, $attribute_alias FROM ($subquery) t
q10 = SELECT DISTINCT $attribute FROM ($subquery) t
q11 =
q12 = SELECT * FROM ($subquery) AS l INNER JOIN ($right_query) AS r ON
       l.$left_on = r.$right_on
q13 = SELECT * FROM ($subquery) AS l LEFT OUTER JOIN ($right_query) AS r ON
       l.$left_on = r.$right_on
q14 = SELECT $agg_value FROM ($subquery) t
q15 = SELECT * FROM $view
q16 = SELECT (t).* FROM (SELECT t, ROW_NUMBER() OVER (PARTITION BY $grp_by_attribute)
       AS row_number FROM ($subquery) t) t WHERE row_number = 1
q17 = SELECT (t).* FROM (SELECT t, COUNT(*) OVER (PARTITION BY $grp_by_attribute) AS cnt
                         FROM ($subquery) t) t WHERE cnt = 1
q18 = SELECT t.*, n.i AS $alias FROM ($subquery) t LEFT JOIN LATERAL UNNEST(referredTopics)
       AS n(i) ON true

[ATTRIBUTE ALIAS]
single_attribute = $attribute
attribute_remove = '$attribute'
attribute_project = "$attribute"
attribute_alias = $attribute AS "$alias"
attribute_name = "$attribute"
```

```
rename = $old_attribute AS "$new_attribute"
agg_value = $func AS "$agg_func_$attribute"
attribute_separator = $left, $right
sort_asc_attr = $attribute
sort_desc_attr = $attribute
grp_by_attribute = "$attribute"
grp_value = $attribute
str_format = ’$value’

[ARITHMETIC STATEMENTS]
add = $left + $right
sub = $left - $right
mul = $left * $right
div = $left / $right
mod = $left %% $right
pow = $left ^ $right

[LOGICAL STATEMENTS]
and = $left AND $right
or = $left OR $right
not = NOT $left

[COMPARISON STATEMENTS]
eq = $left = $right
ne = $left != $right
gt = $left > $right
lt = $left < $right
ge = $left >= $right
le = $left <= $right
isna = $left IS NULL
notna = $left IS NOT NULL
isin = $left IN ($right)
```

```
[TYPE CONVERSION]

to_int32 = (($statement) :: INTEGER)

to_int64 = to_bigint($statement)

to_double = (($statement) :: DOUBLE PRECISION)



[LIMIT]

limit = $subquery LIMIT $num

return_all = $subquery

sample_size = 1000


[ESCAPE CHARACTERS]

escape = "


[FUNCTIONS]

min = MIN("$attribute")

max = MAX("$attribute")

avg = AVG("$attribute")

std = STDDEV("$attribute")

count = COUNT("$attribute")

sum = SUM($attribute)

var = VAR_SAMP("$attribute")

abs = ABS($attribute)

fillna = CASE $attribute IS NULL WHEN True THEN $value ELSE $attribute END

replace = CASE WHEN $statement THEN $to_replace ELSE $attribute END

function_format = $function($attribute)

function_arg_format = $function($attribute, $argument)

kwarg = $key=$value
```

B.5 Sample Language-specific Rewrite Rules for SQL++

```
[QUERIES]

q1 = SELECT VALUE t FROM $namespace.$collection t

q2 = SELECT $attribute_alias FROM ($subquery) t

q3 = SELECT VALUE t FROM ($subquery) t WHERE $statement

q4 = SELECT VALUE COUNT(*) FROM ($subquery) t

q5 = SELECT VALUE t FROM ($subquery) t ORDER BY $sort_desc_attr DESC

q6 = SELECT VALUE t FROM ($subquery) t ORDER BY $sort_asc_attr ASC

q7 = SELECT * FROM ($subquery) t GROUP BY $grp_by_attribute

q8 = SELECT $grp_by_attribute, $agg_value FROM ($subquery) t GROUP BY $grp_by_attribute

q9 = SELECT t.*, $attribute_alias FROM ($subquery) t

q10 = SELECT DISTINCT ‘$attribute‘ FROM ($subquery) t

q11 = SELECT VALUE OBJECT_REMOVE(t, $attribute_remove) FROM ($subquery) t

q12 = SELECT $l_alias.*, $r_alias.* FROM ($subquery) AS $l_alias INNER JOIN ($right_query)

       AS $r_alias ON $l_alias.$left_on = $r_alias.$right_on

q13 = SELECT $l_alias.*, $r_alias.* FROM ($subquery) AS $l_alias LEFT OUTER JOIN

       ($right_query) AS $r_alias ON $l_alias.$left_on = $r_alias.$right_on

q14 = SELECT $agg_value FROM ($subquery) t

q15 = SELECT VALUE t FROM $namespace.$view t

q16 = SELECT VALUE grp[0].t FROM ($subquery) t GROUP BY $grp_by_attribute GROUP AS grp

q17 = SELECT VALUE grp[0].t FROM ($subquery) t GROUP BY $grp_by_attribute GROUP AS grp

       HAVING COUNT(t)=1

q18 = SELECT t.*, n AS $alias FROM ($subquery) t LEFT OUTER UNNEST t.$attribute AS n


[ATTRIBUTE ALIAS]

single_attribute = $attribute

attribute_remove = ’$attribute’

attribute_project = ‘$attribute‘

attribute_alias = $attribute AS ‘$alias‘

attribute_name = ‘$attribute‘

rename = ‘$old_attribute‘ AS ‘$new_attribute‘

agg_value = $func AS $agg_func_$attribute
```

```
attribute_separator = $left, $right

sort_asc_attr = $attribute

sort_desc_attr = $attribute

grp_by_attribute = $attribute

grp_value = $attribute

str_format = "$value"


[ARITHMETIC STATEMENTS]

add = $left + $right

sub = $left - $right

mul = $left * $right

div = $left / $right

mod = $left %% $right

pow = $left ^ $right


[LOGICAL STATEMENTS]

and = $left AND $right

or = $left OR $right

not = NOT $left


[COMPARISON STATEMENTS]

eq = $left = $right

ne = $left != $right

gt = $left > $right

lt = $left < $right

ge = $left >= $right

le = $left <= $right

isna = $left IS UNKNOWN

notna = $left IS KNOWN

isin = $left IN [$right]


[TYPE CONVERSION]

to_int32 = to_number($statement)
```

```
to_int64 = to_bigint($statement)
to_double = to_double($statement)



[LIMIT]
limit = $subquery LIMIT $num
return_all = $subquery
sample_size = 1000

[ESCAPE CHARACTERS]
escape = [‘]

[FUNCTIONS]
min = MIN($attribute)
max = MAX($attribute)
avg = AVG($attribute)
std = STDDEV($attribute)
count = COUNT($attribute)
sum = SUM($attribute)
var = VARIANCE($attribute)
abs = ABS($attribute)
fillna = CASE WHEN $attribute IS UNKNOWN THEN $value ELSE $attribute END
replace = CASE WHEN $statement THEN $to_replace ELSE $attribute END
function_format = $function(‘$attribute‘)
function_arg_format = $function(‘$attribute‘, $argument)
kwarg = $key=$value

[SAVE RESULTS]
to_collection = CREATE TYPE $namespace.TempType IF NOT EXISTS AS OPEN{ _uuid: uuid};
       CREATE DATASET $namespace.$collection(TempType) PRIMARY KEY _uuid autogenerated;
       INSERT INTO $namespace.$collection SELECT VALUE ($subquery);
to_view = CREATE VIEW $namespace.$collection AS $subquery;
```

Appendix C

Benchmark Translated Queries

C.1 Benchmark Timing Points

- **Pandas Timing**

```
# DataFrame creation time
df = pd.read_json(file_path)
# Expression-only time
df.head()
```

- **Spark Timing**

```
# DataFrame creation time
df = spark.read.format("mongo")
     .option("uri", "mongodb://x.x.x.x")
     ...
     .load()
# Expression-only time
df.head(5)
```

- **PolyFrame Timing**

```
# DataFrame creation time
df = AFrame(namespace, collection, DBConnector)
# Expression-only time
df.head()
```

C.2 Benchmark Translated SQL++ Queries

```
1.  SELECT VALUE COUNT(*) FROM data;
2.  SELECT two, four
    FROM (SELECT VALUE t FROM data t) t
    LIMIT 5;
3.  SELECT VALUE COUNT(*)
    FROM (SELECT VALUE t
          FROM (SELECT VALUE t FROM data t) t
          WHERE ten = x
            AND twentyPercent = y
            AND two = z) t;
4.  SELECT oddOnePercent,
           COUNT(oddOnePercent) AS cnt
    FROM (SELECT VALUE t FROM data) t
    GROUP BY oddOnePercent;
5.  SELECT VALUE UPPER(stringu1)
    FROM (SELECT VALUE t FROM data) t
    LIMIT 5;
6.  SELECT MAX(unique1)
    FROM (SELECT unique1
          FROM (SELECT VALUE t FROM data) t) t;
7.  SELECT MIN(unique1)
    FROM (SELECT unique1
          FROM (SELECT VALUE t FROM data) t) t;
```

```
8.  SELECT twenty, MAX(four) AS max_four
    FROM (SELECT VALUE t FROM data) t
    GROUP BY twenty;
9.  SELECT VALUE t
    FROM (SELECT VALUE t FROM data) t
    ORDER BY unique1 DESC
    LIMIT 5;
10. SELECT VALUE t
    FROM (SELECT VALUE t FROM data) t
    WHERE ten = x
    LIMIT 5;
11. SELECT VALUE COUNT(*)
    FROM (SELECT VALUE t
        FROM (SELECT VALUE t FROM data) t
        WHERE onePercent >= x
            AND onePercent <= y) t;
12. SELECT VALUE COUNT(*)
    FROM (SELECT l,r
              FROM leftData l JOIN rightData r
              ON l.unique1 = r.unique1) t;
13. SELECT VALUE COUNT(*)
    FROM (SELECT VALUE t
        FROM (SELECT VALUE t FROM data) t
        WHERE tenPercent IS UNKNOWN) t;
x,y,z = random values within range
```

C.3 Benchmark Translated SQL Queries

```
1.  SELECT COUNT(*) FROM (SELECT * FROM data) t;
2.  SELECT "two", "four"
    FROM (SELECT * FROM data) t LIMIT 5;
```

```
3.  SELECT COUNT(*)
    FROM (SELECT *
          FROM (SELECT * FROM data) t
          WHERE "ten" = x
            AND "twentyPercent" = y
            AND "two" = z) t;
4.  SELECT "oddOnePercent",
            COUNT("oddOnePercent") AS cnt
    FROM (SELECT * FROM data) t
    GROUP BY "oddOnePercent";
5.  SELECT upper("stringu1")
    FROM (SELECT stringu1
          FROM (SELECT * FROM data) t) t
    LIMIT 5;
6.  SELECT MAX("unique1")
    FROM (SELECT unique1
          FROM (SELECT * FROM data) t) t;
7.  SELECT MIN("unique1")
    FROM (SELECT unique1
          FROM (SELECT * FROM data) t) t;
8.  SELECT "twenty", MAX("four")
        FROM (SELECT * FROM data) t
        GROUP BY "twenty";
9.  SELECT *
    FROM (SELECT * FROM data) t
    ORDER BY unique1 DESC
    LIMIT 5;
10. SELECT *
    FROM (SELECT * FROM data) t
    WHERE "ten" = x
    LIMIT 5
```

```
11. SELECT COUNT(*)
    FROM (SELECT *
         FROM (SELECT * FROM data t) t
         WHERE "onePercent" >= x
            AND "onePercent" <= y) t;
12. SELECT COUNT(*)
    FROM (SELECT l.*,r.*
          FROM (SELECT * FROM left) l
          INNER JOIN (SELECT * FROM right) r
          ON l.unique1 = r.unique1) t;

13. SELECT COUNT(*)
    FROM (SELECT *
        FROM (SELECT * FROM data) t
        WHERE "tenPercent" IS NULL) t;

x,y,z = variables representing random values within range
```

C.4 Benchmark Translated Cypher Queries

```
1.  MATCH(t: data)
    RETURN COUNT(*) AS t
2.  MATCH(t: data)
    WITH t{‘two‘:t.two, ‘four‘:t.four}
    RETURN t
3.  MATCH(t: data)
    WITH t WHERE t.ten = x
            AND t.twentyPercent = y
            AND t.two = z
    RETURN COUNT(*) AS t
```

```
4.  MATCH(t: data)
    WITH {‘oddOnePercent‘: t.oddOnePercent,
        ‘count‘: count(t.oddOnePercent)} AS t
    RETURN t
5.  MATCH(t: data)
    WITH t{‘stringu1‘:t.stringu1}
    WITH t{‘upper(t.stringu1)‘:upper(t.stringu1)}
    RETURN t
    LIMIT 5
6.  MATCH(t: data)
    WITH t{‘unique1‘:t.unique1}
    WITH {‘max_unique1‘: max(t.unique1)} AS t
    RETURN t
7.  MATCH(t: data)
    WITH t{‘unique1‘:t.unique1}
    WITH {‘min_unique1‘: min(t.unique1)} AS t
8.  MATCH(t: data)
    WITH {‘twenty‘: t.twenty,
          ‘max_four‘: max(t.four)} AS t
    RETURN t
9.  MATCH(t: data)
    WITH t ORDER BY t.unique1 DESC
    RETURN t
    LIMIT 5
10. MATCH(t: data)
    WITH t WHERE t.ten = x
    RETURN t
    LIMIT 5
11. MATCH(t: data)
    WITH t WHERE t.onePercent >= x
            AND t.onePercent <= y
    RETURN COUNT(*) AS t
```

```
12. MATCH(t: data)
    MATCH (t),(r:wisconsin2)
    WHERE t.unique1 = r.unique1
    WITH t{.*, r}
    RETURN COUNT(*) AS t
13. MATCH(t: data)
    WITH t WHERE t.tenPercent IS NULL
    RETURN COUNT(*) AS t
x,y,z = random values within range
```

C.5 Benchmark Translated MongoDB Queries

```
1.  namespace.collection.aggregate([
    {"$match":{}} ,
    {"$count":"count"}
    ])
2.  namespace.collection.aggregate([
    {"$match":{}},
    {"$project":{"two":1,"four":1}},
    {"$project":{ "_id":0}},
    {"$limit":5}
    ])
3.  namespace.collection.aggregate([
    {"$match":{}},
    {"$match":{"$expr":{"$and":[{"$and":[
                    {"$eq":["$ten",x]},
                    {"$eq":["$twentyPercent", y]}]}]},
                    {"$eq":["$two",z]}]}}},
    {"$count":"count"}])
```

```
4.  namespace.collection.aggregate([
    {"$match":{}},
    {"$group":{"_id":{ "oddOnePercent":"$oddOnePercent" },
               "count_oddOnePercent":{"$sum":1}}},
    {"$addFields": { "oddOnePercent": "$_id.oddOnePercent"} },
    {"$project": {"_id": 0 } }
    ])
5.  namespace.collection.aggregate([
    {"$match":{}},
    {"$project":{"stringu1":1}},
    {"$project":{"stringu1:{"$toUpper":"$stringu1"}}},
    {"$project":{"_id":0}},
    {"$limit":5}
    ])
6.  namespace.collection.aggregate([
    {"$match":{}},
    {"$project":{"unique1":1}},
    {"$group":{"_id":{},"max":{"$max":"$unique1"}}},
    {"$project":{"_id":0}}
    ])
7.  namespace.collection.aggregate([
    {"$match":{}},
    {"$project":{"unique1":1}},
    {"$group":{"_id":{},"min":{"$min":"$unique1"}}},
    {"$project":{"_id":0}}
    ])
8.  namespace.collection.aggregate([
    {"$match":{}},
    {"$group":{"_id":{"twenty":"$twenty"}, "max":{"$max":"$four"}}},
    {"$addFields":{"twenty":"$_id.twenty"}},
    {"$project":{"_id":0}}
    ])
```

```
9.  namespace.collection.aggregate([
    {"$match":{}},
    {"$sort":{"unique1":-1}},
    {"$project":{"_id":0}},
    {"$limit":5}
    ])
10. namespace.collection.aggregate([
    {"$match":{}},
    {"$match":{"$expr":{"$eq":["$ten",x]}}},
    {"$project":{"_id":0}},
    {"$limit":5}
    ])
11. namespace.collection.aggregate([
    {"$match":{}},
    {"$match":{"$expr":{"$and":[{"$gte":["$onePercent",x]},
                                {"$lte":["$onePercent",y]}]}}},
    {"$count":"count"}
    ])
12. namespace.collection.aggregate([
    {"$lookup":{"from":"collection2", "as":"collection2",
            "let":{"left":"$unique1"},
            "pipeline":[{"$match":{}},
                {"$match":{"$expr":{"$eq":["$unique1","$$left"]}}}]}},
    {"$unwind":{"path":"$collection2","preserveNullAndEmptyArrays":false}},
    {"$count":"count"}
    ])
13. namespace.collection.aggregate([
   {"$match":{}},
   {"$match":{"$expr":{"$lt":["$tenPercent",null]}}},
   {"$count":"count"}
   ])
x,y,z = variables representing random values within range
```