DQDF: Data-Quality-Aware Dataframes

Phanwadee Sinthong[*]
University of California, Irvine
California, USA
psinthon@uci.edu

Dhaval Patel
IBM TJ Watson Research Center
New York, USA
pateldha@us.ibm.com

Nianjun Zhou
IBM TJ Watson Research Center
New York, USA
jzhou@us.ibm.com

Shrey Shrivastava
IBM TJ Watson Research Center
New York, USA
shrey@us.ibm.com

Arun Iyengar
IBM TJ Watson Research Center
New York, USA
aruni@us.ibm.com

Anuradha Bhamidipaty
IBM TJ Watson Research Center
New York, USA
anubham@us.ibm.com

ABSTRACT

Data quality assessment is an essential process of any data analysis process including machine learning. The process is time-consuming as it involves multiple independent data quality checks that are performed iteratively at scale on evolving data resulting from exploratory data analysis (EDA). Existing solutions that provide computational optimizations for data quality assessment often separate the data structure from its data quality which then requires efforts from users to explicitly maintain state-like information. They demand a certain level of distributed system knowledge to ensure high-level pipeline optimizations from data analysts who should instead be focusing on analyzing the data. We, therefore, propose data-quality-aware dataframes, a data quality management system embedded as part of a data analyst's familiar data structure, such as a Python dataframe. The framework automatically detects changes in datasets' metadata and exploits the context of each of the quality checks to provide efficient data quality assessment on ever-changing data. We demonstrate in our experiment that our approach can reduce the overall data quality evaluation runtime by 40-80% in both local and distributed setups with less than 10% increase in memory usage.

PVLDB Reference Format:
Phanwadee Sinthong, Dhaval Patel, Nianjun Zhou, Shrey Shrivastava, Arun Iyengar, and Anuradha Bhamidipaty. DQDF: Data-Quality-Aware Dataframes. PVLDB, 15(4): 949-957, 2022.
doi:10.14778/3503585.3503602

1 INTRODUCTION

In this era of big data, interpreting large volumes of data for business advantages and decision making is becoming a common practise. Data-driven analytics and machine learning are directly affected by quality of the data. Anomalies in training data could significantly affect machine learning models' performance as models are sensitive to variations in data values. Errors in serving data could result in inability to understand the model or interpret its output.

[*]Work done while at IBM Research

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503602

The importance of validating data quality has become increasingly realized as various efforts [18, 19, 22, 27] have been put into creating automated data quality verification frameworks for evaluating and improving data quality. These new libraries introduce a learning curve as data quality information is disconnected from the data structure used in the analysis process; data scientists have to select and apply an appropriate data quality assessment library.

During exporatory Data Analysis (EDA), data is transformed and examined iteratively. Unnecessary retrieval of data in a distributed environment for validation is inefficient and could result in performance degradation. Existing solutions for data quality verification on evolving datasets such as [26] require explicit state information maintenance in order to operate efficiently on evolving datasets.

In order to optimize data quality validation on evolving data and require little to no effort from the users, we introduce Data-Quality-Aware Dataframes (DQDF), a data quality management system embedded in a data structure that is widely used by data scientists. The specific implementation of our approach which we describe in this paper is for Python applications. The analytics capabilities which we add apply data quality checks to the underlying data. It should be noted that the general approach could be applied to other types of calculations in other programming languages. Our library provides in-place data quality evaluation. The system utilizes metadata to facilitate data quality validation and records its results to optimize subsequent evaluations. To optimize the computation and reduce the overall run time, each of the data quality checks takes into account existing data quality information and data statistics that indicate changes in the data from the previous evaluation. Our contributions are the following.

- Provide state management and computational capabilities by enhancing a commonly used data construct, such as dataframes in Python.
- Embed data quality information as part of dataframes' metadata by introducing a new primitive, 'describe_quality()', for data quality evaluation.
- Provide top-level optimization for time series data validations by sharing commonly used complex dataframe operations' results via an embedded metadata catalog.

The rest of this paper is organized as follows: Section 2 discusses background and related work. Section 3 provides an overview of our system. Section 4 details a set of performance experiments and results from running DQDF. Section 5 describes future works.

2 BACKGROUND AND RELATED WORK

Different types of data quality assessments and definitions of data quality standards are heavily studied and well-documented [20, 21, 25]. In order to deliver an efficient and automated data quality validation on ever-changing data, we extended a pre-define list of quality checks (*validators*) from an automated data quality verification system, DQA [28]. It is important to note here that we use validators from DQA as examples to demonstrate the benefits of our data quality management system. Our library can be adjusted to work with other python-based data quality validation libraries. Here we briefly describe general dataframe properties, DQA, the different types of its validators, and related works.

2.1 Dataframe

Dataframe is a two-dimensional data structure with labeled axes. Dataframe is available through the pandas library [10] in Python. Due to the intuitive data model and extensive features, dataframe is a popular choice for data exploration [24]. In fact, the rich set of operations available in pandas dataframes has been cited as a reason that makes Python one of the most preferred language for data science [6, 11]. However, pandas' limitation is its ability to operate on datasets at scale. Pandas only operates on a single machine and fails to process data larger than memory. As a result, several efforts [2, 7, 8, 15, 16] have been put into creating scalable dataframe libraries (discussed next) to address pandas' limitations.

2.2 Data Quality Advisor

Data Quality Advisor (DQA) [28] is a unified framework for data quality validation. The tool generates human-readable data quality reports and data quality pipelines to be integrated into the existing AI ecosystem. In DQA, a validator is an abstraction that forms the basis of data quality checking operations. The system supports generic checks and domain-specific sets of validators such as time series. A validator object has three main attributes: checker function, validity record, and execution backend. The checker function is the function to be executed on the datasets. Validators produce validity records after they perform the respective quality checks. Execution backend contains information about each check's preferred execution engine (e.g., SQL, Spark, etc.).

2.3 Related Work

To our knowledge, there is no previous work that combines the dataframe interface with data quality validation to deliver a seamless user experience and optimize the computation on repetitive evaluations of the data resulting from EDA operations. We will compare and contrast existing systems in terms of automated data verification systems and dataframe technology.

2.3.1 Automated data verification systems. Here we consider systems that are capable of automatically performing data quality assessments on distributed datasets.

Deequ: Deequ [27] is an automated data verification system built on top of Apache Spark. Deequ utilizes Spark to perform data quality checks on large datasets that are typically stored in distributed file systems. A new extension [26] to Deequ provides an optimization to reduce the system's overall runtime by utilizing state objects (that need to be maintained for each of Spark's data partitions) to incrementally perform data verification on ever-changing data and avoid re-evaluating the existing data. However, users are required to explicitly create these external state objects. Since Deequ is only available in Scala, it targets users that are already familiar with the Spark ecosystem because it relies on data having a consistent partitioning scheme in order to perform its incremental computation. Another optimization that is available in Deequ is a shared data scan which groups together aggregate functions (e.g., sum, average, min, etc.) that will be performed on data attributes and translating them into a single Spark SQL aggregation statement to reduce the number of passes through the data. However, the shared data scan benefit is only limited to aggregate functions.

TensorFlow Data Validation: TFDV [22] is a scalable data validation system for machine learning pipelines in production environments. TFDV automatically detects the data schema and allows users to make adjustments that capture data constraints and suit their expectations of the data characteristics (e.g., data type, domain). The system detects anomalies by comparing the data statistics against the system-generated schema definition. It provides alerts with context to help users make informed decisions to correct the errors. However, as the focus of TFDV is in automating data quality validation specifically for machine learning pipelines, it does not provide any optimizations for repetitive quality checks on ever-changing data resulting from data exploration.

2.3.2 Scalable Dataframe Systems. There are several data cleaning libraries [12, 17] that can operate on pandas dataframes but they only operate on a local work station. In recent years, there have been efforts to scale pandas dataframes to operate in a distributed environment. Here we consider some of these scalable dataframe systems that provide general pandas-like API and scale-out the operations onto large distributed datasets.

Spark/Koalas DataFrame: Apache Spark [1] is an open-source cluster computing framework that provides in-memory parallel computation on a cluster with scalability and fault tolerance. Spark provides Spark DataFrame [16], a module to interact with structured tabular data that incorporate relational data processing with functional programming. However, Spark DataFrame syntax is very different from the pandas dataframes' due to its lazy evaluation approach to optimize computations. As a result, Koalas [7] is introduced as a library to ease transitioning from pandas to Spark. Koalas provides a pandas-like API and translating the operations to be executed on Spark.

Dask: Dask[2] is a framework that scales Python data analysis libraries such as pandas, Scikit-learn[14], and NumPy[9] to operate in a distributed environment. Dask provides a pandas dataframe based implementation that scales to multiple machines by partitioning large datasets row-wise into multiple small pandas dataframes and processes them in parallel. Due to its row-based partitioning approach, Dask does not support the pandas accessing of rows and columns by location. By using pandas internally, Dask does not provide top-level optimization on a chain of operations.

3 DQDF SYSTEM ARCHITECTURE

Data quality validation is executed from scratch iteratively on evolving data. It is a time-consuming process and difficult to optimize as changes in the data might not be visible. Currently, an effort is required from the users to maintain state information to provide statistical information about the changes needed for optimizations. In order to deliver data quality assessments and analyses geared toward data scientists, we integrate data quality information as part of the dataframe metadata. This design allows for optimizations to take place right where data manipulations happen and requires no external state information maintenance. DQDF is able to operate on different types of data (e.g., tabular, timeseries) supported by dataframe constructs as it is a centralized metadata management system and it does not rely on a data partitioning scheme.

In order to demonstrate the feasibility of our design, we provide two different implementations; pandas-DQDF which extends pandas dataframe and Dask-DQDF which is a scalable version of our approach that extends Dask dataframe.

In this section, we describe the DQDF architecture, explain each of its components in detail, describe our optimizations for repetitive evaluations, and present the workflow.

3.1 System Overview

Figure 1 shows a system overview of DQDF. The core component of DQDF is the Catalog Generator. The Catalog Generator is composed of four main components. We use a term *dataframe catalog*, inspired by the notion of database catalog in relational databases where metadata and schemas of tables in database systems are stored.

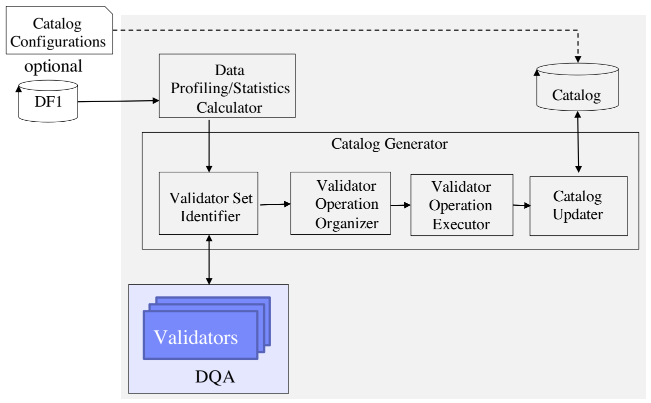


Figure 1: DQDF System Overview

First, *Validator Set Identifier* is a component that selects a predefined set of validators based on the type of the underlying data indicated by end users at the time of dataframe initialization. Users can add and drop any validators from the identified set at any point after the dataframe initialization. The validator set identifier also takes care of determining which validators need to be executed in subsequent data quality evaluations based on the changes in the underlying data received from the data profiler.

Second, *Validator Operation Organizer* is a component that rearranges the identified validators and extract shared computations to pre-execute. Validators can then utilize the pre-executed results in place of the actual computations. As a result, operations that would otherwise get executed multiple times are reduced to only once per call to data quality evaluation.

After the commonly used functions are already executed, *Validator Opearation Executor* calls all the remaining validators' checker functions and collects validity reports from each validator. It will then delete the shared computations' results from the catalog.

Finally, the *Catalog Updater* will update the existing dataframe catalogs with a new list of reports and the current data statistics. Next, we will give an example of a dataframe catalog and describe our pre-defined validators.

3.2 Catalog

Our catalog is an abstract class that stores metadata. There are two types of catalogs in DQDF; one contains metadata about the dataframe called Dataframe catalog and the other that contains metadata specific to a validator called validator catalog.

3.2.1 Dataframe catalog. A catalog class contains statistical information about the underlying data, its validators, each validator's independent computations, and the data quality information. The dataframe catalog also contains a list of validator-specific information (validator catalog).

3.2.2 Validator catalog. This catalog type contains validation-specific information that DQDF uses to optimize the overall data quality computation. A validator catalog contains a modified validator's checker function, a validity report, and a *trigger function*. A validator catalog can also include a list of independent operations used as part of its checker function. A trigger function is specific to a validator. For example, a validator that only operates on a numerical column would have a trigger function that returns false if a string column has been dropped. However, a trigger function of a dataset-based validator will likely return true if there is any changes in the data.

3.3 Predefined Validators

Similar to other data quality verification systems, DQA also has several validators for each type of data (e.g., generic tabular data, time series data). The system also provides an interface for users to define their own validators. We extended these validators to take in DQDFs and re-design their checker functions to leverage the embedded dataframe catalog to reduce the amount of needed computation and efficiently utilize already pre-computed statistics.

Data cleaning is defined by [20] as a process to correct systematic problems in the raw data. Domain expertise and finding and rectifying specific observations in the data are the most successful data cleaning techniques. Incorrect values in the data can include duplicate, mistyped, and outlier data. When presented with expected results, domain experts can easily recognize the observations that are incorrect. Common techniques that allow effective data cleaning then involve displaying helpful statistics about the dataset. We have picked a subset of data quality checks that present data statistics and could potentially help data scientists identify outliers in the raw data such as null values, missing data, extreme values, and most common values, etc.,

After examining the list of validators for both general tabular data and time series data, we have identified repetitive operations that are performed on the data each time data quality validation

Table 1: General Tabular Data Validators

Validator	Operation Characteristic			Optimization for incremental calculation	Shared Computation	Description
	dataset-based	column-based	record-based			
check_na_columns			✓	Running sum of null values	-	Return null percentage per column
check_infinity_column			✓	Running sum of infinity values	-	Return infinity (np.inf) percentage per column
check_zero_ratio_column			✓	Running sum of zeros	-	Return zero value percentage per column
check_duplicate_rows	✓			-	-	Check for duplicate rows
check_duplicate_column_names	✓			-	-	Compare column names
check_duplicate_values	✓			-	-	Compare column values
check_constant_columns		✓		-	unique	List of columns with a constant value
check_columnwise_unique_values		✓		-	nunique	Number of unique values per column
check_most_occurring_values		✓		-	value_counts	List of n most occurring values per column
check_repeating_values_columns		✓		-	value_counts	List of non-unique values per column
check_non_repeating_values_columns		✓		-	value_counts	List of unique values per column
check_numeric_not_categorical_columns		✓		-	unique	Check if numerical columns are also categorical

Table 2: Time Series Data Validators

Validator	Target column	Optimization for incremental calculation	Shared Components			Description
			drop_duplicates (timestamp_col)	sort_index	sampling_freq	
min_frequency	Timestamp	Running counts of True and False	✓	✓		Check that sampling frequencies are above a threshold
min_duration	Timestamp	index.min, index.max	✓	✓		Determine if the dataset satisfies the minimum amount of data
periodicity	Timestamp	-	✓	✓	✓	Determine if the data is periodic
find_duplicates_present_in_time	Timestamp	-	✓	✓		Check duplicates values in timestamp column
find_if_uniform_sampling_rate	Timestamp	-	✓	✓	✓	Check if the data has a uniform sampling rate
find_outliers	Data	-	✓	✓	✓	Find outliers in data columns

happens. This is because originally in DQA each validator is isolated from one another and they operate on the same data in a sequence.

3.3.1 Generic Tabular Data. Checker functions for generic tabular data are shown in Table 1. We also identify their operation characteristics which play an important role in optimizing the computation. There are three types of operation characteristics; record-based, column-based, and dataset-based. These characteristics indicate the granularity level at which the checker functions operate. Record-based functions are functions that consider each individual record separate from other records. For example, the first record-based checker function on the list is `check_na_columns` which counts the number of null or missing values for each column. It returns the percentages of these values per column if present. The check passes if there are no null or missing values in all of the columns and fails otherwise. Column-based functions need to consider the entire column's values in order to determine the final output. For example, the `check_most_occurring_values` function returns the top n most occurring values per column. This function needs to consider each record's value in comparison with the other records' per column. The dataset-based functions are those that require access to the entire dataset in order to compute the results.

3.3.2 Time Series Data. Time series validators in DQA are complex functions that aim at evaluating the unique characteristics of time series data such as order, frequency, and duration. Selected predefined time series validators are displayed in Table 2. Their checker functions consist of multiple operations. Instead of classifying these checker functions based on the operation characteristics, we identify their target columns because time series validators often compare values between ordered consecutive records. The target columns are either the timestamp column or other data columns. This distinction also allows us to leverage the optimizations we have implemented for the generic tabular validators and eliminate the need to re-execute these checker functions if the identified data changes do not affect their target columns.

3.4 Optimizations

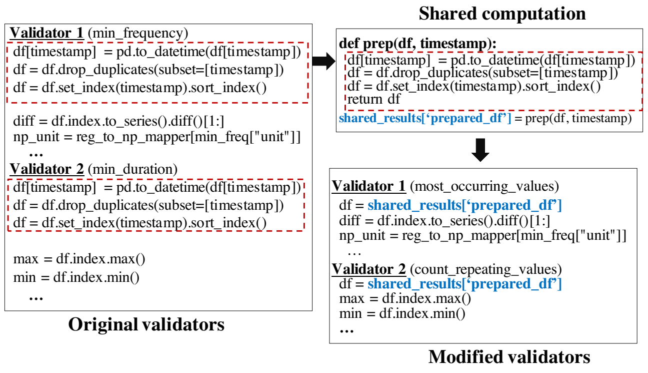
We have selected DQA's validators as an initial set of validators to apply our metadata-driven optimizations. DQA does not provide any optimizations on its own. Its validators are originally executed independently and sequentially before the library generates the data quality report. As such, we modified a selected set of DQA's validators so that they can take advantage of our in-place metadata management mechanism to avoid executing unnecessary checks on the data. Here we discuss our optimizations in detail.

3.4.1 *Shared computation.*

Carefully examining the repetitive operations from Table 1 and 2, we notice that some of them are computationally intensive which significantly affects the overall system's runtime. For tabular data, a commonly used solution is distributed shared data scan by grouping together aggregate functions to reduce the number of passes through the data. This works well for tabular data validators as the majority of them involve calculating aggregate values from the data, which can be done regardless of the record order. However, in the case of time series data analysis, the order of records needs to be maintained. Time series validators often compare each individual data points with their neighboring records ordered by the timestamp such as the 'find_duplicate_present_in_time' validator identifies the positions of records with duplicate timestamps. Shared data scan simply does not work for timeseries validators if the records are unordered because it is difficult to maintain row and column labels and express them using plain SQL syntax as clearly explained in [24]. An order will have to be enforced to enable such optimization. Fortunately, dataframes treat both rows and columns labels equally and multiple scalable dataframe libraries such as Dask supports these functionalities at scale. Since DQDF operates at the dataframe level we could extract common time series operations by utilizing the dataframe API.

As a result, we introduce a notion of *shared computation* through an abstract class that utilizes the dataframe API. It can be used to declare an operation or a set of operations for the system to precompute prior to starting the usual evaluation. Shared computation framework allows any type of operations beyond just aggregate functions to be shared across validators. The results of these operations will be available in the dataframe catalog only for the duration of a given quality evaluation. Therefore, validators that need to perform any of the declared shared computations can retrieve the computation results from the dataframe catalog and avoid redundant computations. Users can declare new shared computations for the system to pre-compute. An example of the shared computation for general tabular data can be found in Table 1. For this set of validators, the shared computations are the count of each distinct values and a list of all unique values per column. For time series validators, the shared components are displayed in Table 2.

After shared computations are extracted and executed, their results are available in the dataframe catalog. Figure 2 displays the modifications we made to the original DQA's validators in order to leverage shared computation results in place of the actual computation. To the left of this figure we can see a portion of the two original DQA's time series validators which are *min_frequency* and *min_duration*. These validators perform different data quality checks on the data but they all execute pandas' `to_datetime` function follows by `drop_duplicates` and `set_index` functions as part of their validations. These sequence of operations prepare the time series data by converting the timestamp coloumn to type datetime object, drop duplicate rows, and use the timestamp as an index column of the dataset. These data preparation steps could be time-consuming and could involve multiple cross-network communications in a distributed environment. Utilizing shared computation allows us to only execute these operations once and cache the results for other validators to utilize. To the right of the figure, we have extracted the three operations as a function to preserve the order and stored the results in a dictionary-like construct. Below the shared computation example are the modified DQA's validators that take in a shared computation result available in the dataframe catalog and use this result in place of their actual computation.



```
Validator 1 (min_frequency)
df[timestamp]  = pd.to_datetime(df[timestamp])
df = df.drop_duplicates(subset=[timestamp])
df = df.set_index(timestamp).sort_index()

diff = df.index.to_series().diff()[1:]
np_unit = reg_to_np_mapper[min_freq["unit"]]
   ...
Validator 2 (min_duration)
df[timestamp]  = pd.to_datetime(df[timestamp])
df = df.drop_duplicates(subset=[timestamp])
df = df.set_index(timestamp).sort_index()

max = df.index.max()
min = df.index.min()
   ...
```

```
def prep(df, timestamp):
 df[timestamp]  = pd.to_datetime(df[timestamp])
 df = df.drop_duplicates(subset=[timestamp])
 df = df.set_index(timestamp).sort_index()
 return df
shared_results['prepared_df'] = prep(df, timestamp)
```

```
Validator 1 (most_occurring_values)
  df = shared_results['prepared_df']
  diff = df.index.to_series().diff()[1:]
  np_unit = reg_to_np_mapper[min_freq["unit"]]
   ...
Validator 2 (count_repeating_values)
  df = shared_results['prepared_df']
  max = df.index.max()
  min = df.index.min()
  ...
```

Figure 2: Shared Computation Example

3.4.2 *Incremental computation.*

Incremental computation for repetitive workload has been intensively studied and successfully implemented in [26] for partitioned data by utilizing manually created state-like objects accompanying each data partition. As such, our focus is not on re-implementing such optimization but we describe here how our catalog implementation and trigger functions allow such optimization to take place efficiently on both partitioned and non-partitioned data without requiring an external state object.

In order to enable incremental computation on changing data, prior knowledge about the validated data is needed to be maintained. Similar systems often require users to create state objects manually for each data partition to enable such optimization. While this design works well for developers and data engineers, it could be troublesome for data scientists. In DQDF, we embed validator-specific statistics into the data structure allowing the incremental computation of data quality to take place automatically without requiring distributed system knowledge from users.

Trigger functions allow the incremental computation process to be even more efficient by making it context-aware. Trigger functions utilize the metadata catalog to eliminate the need to execute the computation in the case that the change in metadata does not affect certain validators.

3.5 Workflow

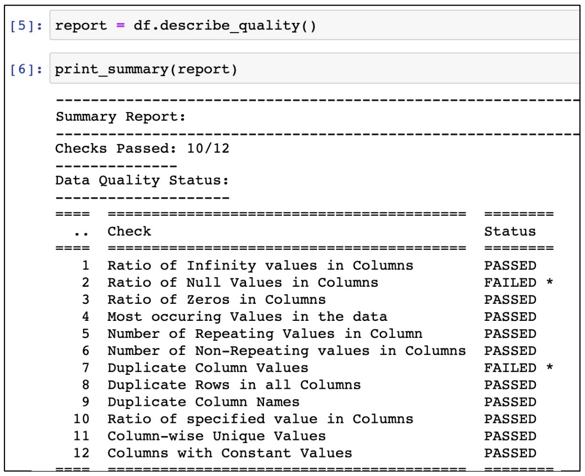
Figure 3 shows a workflow after a user invokes the 'describe_quality' method. This method is part of the DQDF library which can be called on a dataframe object. If this is the first time the method is invoked or the value is null, a new dataframe catalog will be created and initialized with a list of predefined validators based on the type of data indicated. Then the system will compute and record the current data statistics including row count, column names, and column types. If a dataframe catalog exists, the current dataframe catalog will be sent to a catalog generator. The generator compares the recorded data statistics with the current statistics. It will then identify a list of affected validators that need to be re-run. After a list of affected validators has been assembled, each validator's checker function will be called on the underlying data. Each validator will then produce a validity record containing the result of the evaluation and a recommendation for users to make corrections to the data in case the check fails. The generator will update the dataframe catalog with the current information.

```
Algorithm 1: describe_quality
  Input: Dataframe
  Output: List of validation output
  if catalog is empty then
      initialize a catalog;
      $validations \leftarrow$ initialize validations;
  else
      compute change in dataset metadata;
      $validations \leftarrow$ get needed validations;
      populate catalog with necessary information;
  end
  $outputs \leftarrow \{\}$
  foreach $v \in validations$ do
      $result \leftarrow$ execute validation $v$;
      $outputs$.insert($result$)
  end
  update catalog;
  return outputs;
```

Figure 3: Algorithm for describe_quality

3.6 User Model

Here we showcase the interface of our library in a Jupyter notebook environment. Users would be importing in our library as if they would the regular Pandas dataframe. Figure 4 shows how users can run 12 pre-selected set of validations and receive a data quality report. This is done by calling the describe_quality method on a dataframe object (df). After manipulating and cleaning the data, users can call the same method to get a new report which gets updated automatically. Each validation check is executed in an optimized manner according to the change in the dataset metadata. This design does not requires users to maintain a state object explicitly which simplifies their interactions. DQA provides a pretty-print method called 'print_summary' for result visualization. The quality report contains the validations' results and recommendation. Figure 5 shows a process required to add a custom validator. Line 14 shows a custom function that takes a dataframe object and checks for any negative values in each of the numerical columns. DQA requires that each checker function returns a ValidationOutput. Line 15 adds the custom validator to DQDF by supplying the name, a checker function, and select one of our provided validator catalog (we provide 3 levels of support; record-based, column-based, and dataset-based). Line 16 prints all of the validations and the 'check_negative_columns' has been added at the end of the list.

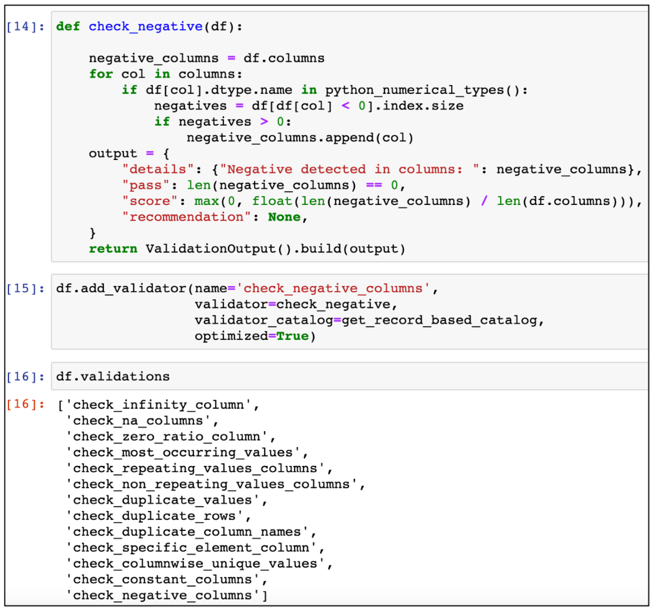


```
[5]: report = df.describe_quality()

[6]: print_summary(report)

     ---------------------------------------------------------------
     Summary Report:
     ---------------------------------------------------------------
     Checks Passed: 10/12
     --------------
     Data Quality Status:
     --------------------
     ====  ========================================  ========
       ..  Check                                     Status
     ====  ========================================  ========
        1  Ratio of Infinity values in Columns       PASSED
        2  Ratio of Null Values in Columns           FAILED *
        3  Ratio of Zeros in Columns                 PASSED
        4  Most occuring Values in the data          PASSED
        5  Number of Repeating Values in Column      PASSED
        6  Number of Non-Repeating values in Columns PASSED
        7  Duplicate Column Values                   FAILED *
        8  Duplicate Rows in all Columns             PASSED
        9  Duplicate Column Names                    PASSED
       10  Ratio of specified value in Columns       PASSED
       11  Column-wise Unique Values                 PASSED
       12  Columns with Constant Values              PASSED
     ====  ========================================  ========
```

Figure 4: Running Data Quality Validation



```
[14]: def check_negative(df):

          negative_columns = df.columns
          for col in columns:
              if df[col].dtype.name in python_numerical_types():
                  negatives = df[df[col] < 0].index.size
                  if negatives > 0:
                      negative_columns.append(col)
          output = {
              "details": {"Negative detected in columns: ": negative_columns},
              "pass": len(negative_columns) == 0,
              "score": max(0, float(len(negative_columns) / len(df.columns))),
              "recommendation": None,
          }
          return ValidationOutput().build(output)

[15]: df.add_validator(name='check_negative_columns',
                       validator=check_negative,
                       validator_catalog=get_record_based_catalog,
                       optimized=True)

[16]: df.validations

[16]: ['check_infinity_column',
       'check_na_columns',
       'check_zero_ratio_column',
       'check_most_occurring_values',
       'check_repeating_values_columns',
       'check_non_repeating_values_columns',
       'check_duplicate_values',
       'check_duplicate_rows',
       'check_duplicate_column_names',
       'check_specific_element_column',
       'check_columnwise_unique_values',
       'check_constant_columns',
       'check_negative_columns']
```

Figure 5: Adding a Custom Validator

4 EXPERIMENTAL EVALUATION

In order to demonstrate the generality of our approach and ensure the scalability of our design, we implemented the DQDF architecture by extending the pandas dataframe library to operate on a single workstation and extending the Dask dataframe library to operate on a multi-node cluster. It is important to note here that we conducted this experiment as a demonstration of the scalability of our new architecture rather than to compare the performance of pandas and Dask. Important points to consider when choosing between Dask and pandas are well documented on the official Dask documentation [3]. The environment setups for both single node and multi-node experiments are described below.

4.1 Experimental Setup

We extended Pandas and Dask dataframe implementations using their provided subclassing interface by embedding a dataframe catalog object as part of the library. Our implementation provides a method for users to invoke a data quality evaluation. As such, we compare the system's overall performance by invoking the quality evaluation method (which enables our optimizations) against running all of the validators on a dataframe without any optimizations.

We performed our evaluations on machines with 4 processing cores, 8 GB RAM, 100 GB hard drive, and each running CentOS 7. The tested operations are adding and deleting rows and columns.

Table 3: Experimental Setup and Result Summary

Category/ Operation	Pandas-DQDF (Single Node)		Dask-DQDF (4-Node)	
	Tabular	Time Series	Tabular	Time Series
Experimental Setup				
Data sizes	0.5GB - 2.5GB	20MB - 100MB	2GB - 10GB	100MB - 500MB
# of data quality checks	12	6	12	6
Average Percentage of Runtime Reduction				
Increase # of rows	44%	30.50%	41.80%	46.40%
Decrease # of rows	45.18%	29.50%	44.20%	46.80%
Increase # of columns	50.30%	81.04% across 4 runs (31.2% on the 1st run)	47.60%	85.23% across 4 runs (46.45% on the 1st run)
Decrease # of columns	83.7% across 4 runs (33.8% on the 1st run)	81.2% across 4 runs (30.15% on the 1st run)	74.3% across 4 runs (39.2% on the 1st run)	85.1% across 4 runs (46.9% on the 1st run)

4.1.1 ***Single Node Setup.*** We used the Wisconsin benchmark data [23] for our general tabular data evaluation. This dataset allows us to precisely control the selectivity percentages, to generate data with uniform value distributions, and to broadly represent data for general analysis use cases. We used a data generator to generate our experimental datasets with data sizes ranging from 500 MB to 2.5 GB. For time series data, we used a time series data generator to generate the data with increasing timestamps to represent sensor or IoT data. Time series data sizes ranged from 20 MB to 100 MB. We picked time series data sizes that were smaller than tabular data because our time series validators perform computationally intensive operations that result in much longer runtimes.

4.1.2 ***Multi-node Setup.*** In order to demonstrate the scalability of our design, we extended Dask dataframe to produce a parallel variant of DQDF allowing it to operate in a distributed environment. For the multi-node experiment, we set up Dask on a cluster with four processing machines. Each machine has the same specifications as the machine used for the single node evaluation. For the benchmark data, we setup HDFS 3.3.0, allowing all workers access to the data. We used the Wisconsin benchmark dataset with the data sizes ranging from 2.5 GB to 10 GB. For the time series data use case, we generated data of sizes ranged from 100 MB to 500 MB.

We summarize the setups and experimental results in Table 3 before highlighting some of the experimental scenarios in detail in the next subsection. We use 'pandas DQDF' and 'Dask DQDF' for our extended pandas dataframe and extended Dask dataframe implementations respectively. We evaluated DQDF using 12 tabular data validators and 6 time series data validators.

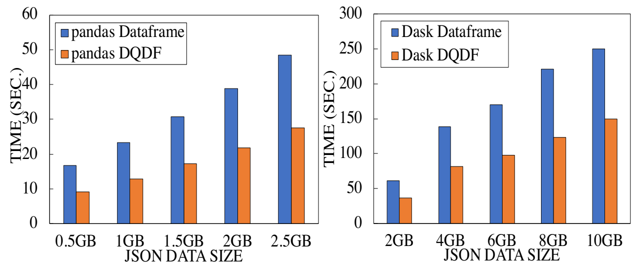
4.2 General Tabular Data Results

We evaluated the overall data quality validation runtimes on four different scenarios (increasing/decreasing number of rows/columns). Due to space limitations, we only show some of the results here.

For general tabular data validation, DQDF benefits from 6 shared computations and 3 incremental computations. As a result, 9 out of 12 quality checks have been optimized, leaving it with only three quality checks without any optimization.

Figure 6 displays results for both single node and cluster evaluations on increasing number of records. We started with reading in a JSON file to create a dataframe and evaluated its data quality. We then read in another four files, appended their content to the previously evaluated dataframe (one file at a time) before evaluating the resulting dataframe's data quality. For this experiment, Pandas DQDF was able to reduce the quality evaluation runtime by 44% on average. Dask DQDF was able to reduce the runtime for datasets with an increasing number of records by 41.8% on average. In the case of decreasing the number of records, we observe the same trend in the opposite direction (decreasing runtime).

In the case of decreasing number of columns, the results of both single-node and cluster evaluations are shown in Figure 7. DQDF can benefit from eliminating unnecessary validations because all validators have already evaluated the entire dataset. Almost all of the validators that have passed can be eliminated on subsequent evaluations. As a result, the first time we ran the validation, the run time is significantly higher than the subsequent runs.

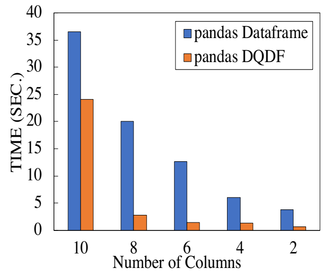


(a) Single node **(b) Four-node cluster**

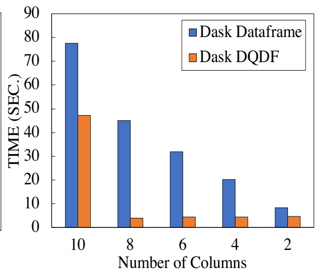
Figure 6: Tabular data: increasing number of rows

4.3 Time Series Data Results

Time series validators perform complex and computationally intensive operations such as getting sampling frequency, rate, and duration from the dataset. Five out of six validators only examine each record's timestamp to detect anomalies without touching the



(a) Single node



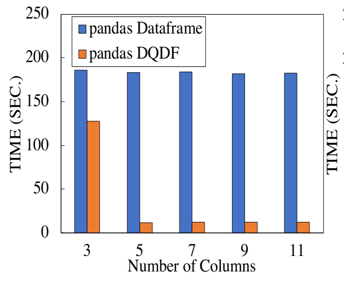
(b) Four-node cluster

Figure 7: Tabular data: decreasing number of columns

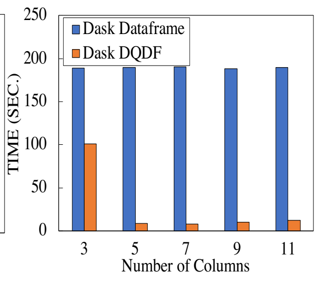
other data columns. As a result, our datasets contain one timestamp column and two numerical columns for row-based evaluations. For column-based evaluations, we increased the number of numerical data columns from 3 up to 11.

All six time series validators we utilized in our experiments benefited from shared computations as they rely on data being sorted and duplicate values have been dropped. For the case of increasing the number of records, two out of the six validators also benefited from incremental computations.

Results on a dataset with an increasing number of columns are shown in Figure 8 for both a single node and a four-node cluster. As previously mentioned, all but one of our time series validators operate on the timestamp column. Increasing the number of data columns does not trigger evaluation of validators that only consider the timestamp column. As a result, the first time we validated the data quality, the DQDF's runtime was significantly higher than its subsequent runs that benefited from eliminating unnecessary execution of timestamp-based validators. Data quality validation on regular pandas and Dask dataframes resulted in repetitive validations of the timestamp column.



(a) Single node



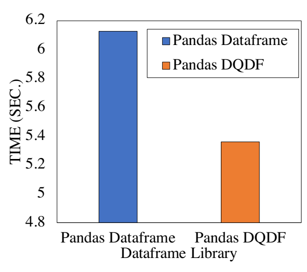
(b) Four-node cluster

Figure 8: Time series data: increasing number of columns

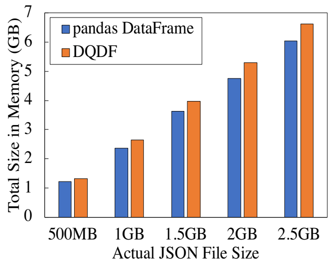
4.4 Case Study and Memory Footprint

In order to assess the usability of our library, we conducted a case study of applying our library to an existing exploratory data analysis [4] on a housing dataset [13]. The goal of this analysis is to predict housing price fluctuations. The notebook contains multiple repetitive data cleaning operations as the author tries to correct erroneous records by filling null values, dropping unwanted columns, and computing statistics. We modeled our study after this notebook and used DQDF in place of Pandas Dataframes to produce the same outputs as the original notebook. We also declared custom validators to take advantage of commonly computed values. The notebooks can be found at [5]. We include the end-to-end runtime comparison of applying Pandas and DQDF in Figure 9a. On average, using Pandas dataframe took 12.52% longer to finish because DQDF was able to eliminate repetitive execution of operations such as null value detection. Even though the benefits of DQDF will be more significant as the size of the data grows, this case study shows that the optimizations also apply to static data.

In order to provide visibility into our design and ensure efficiency, we also conducted data structure size comparison. The total memory usages of DQDFs include the sizes of their catalogs along with their recorded quality results. The comparison results of DQDF and the regular pandas dataframes' sizes are displayed in Figure 9b. DQDFs acquire up to 9.38% more memory space than regular pandas dataframes on average. The reason for such a small percentage is because results of the shared computations are only available during the data quality evaluation and they are not maintained across evaluation sessions.



(a) Case Study Comparison



(b) Memory Usage Comparison

Figure 9: Space and Runtime Comparisons

5 CONCLUSION AND FUTURE WORK

In this work, we have presented a system that integrates data quality information as part of a data structure's metadata allowing data scientists access to relevant information embedded into data structures that they are familiar with. We have demonstrated the practicality of our approach by implementing two different prototypes by extending the widely used Python dataframe libraries, pandas and Dask, to operate on a local machine as well as across machines in a distributed environment. Our decision to embed a data quality management system as part of dataframe libraries allows us to optimize repetitive data quality evaluation on changing data efficiently without requiring efforts from users to maintain additional state information. We demonstrated in our experiments that the system leveraging shared and incremental computations can significantly reduce the overall validation runtime by 40% and up to 80% depending on the type of operations.

Moving forward, we would like to optimize the shared computation extraction process to automatically detect common computations across multiple validators. Our implementations can be extended to other scalable data structure libraries to provide users with options that better suit their analysis requirements.

REFERENCES

[1] 2021. Apache Spark. http://spark.apache.org/.

[2] 2021. Dask. http://dask.org/.

[3] 2021. Dask Common Uses. https://docs.dask.org/en/latest/dataframe.html#common-uses-and-anti-uses/.

[4] 2021. Data Cleaning in Python. https://towardsdatascience.com/data-cleaning-in-python-the-ultimate-guide-2020-c63b88bf0a0d.

[5] 2021. DQDF Case Study. https://github.com/psinthong/DQDF_Case_Study.

[6] 2021. The incredible growth of Python. https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

[7] 2021. Koalas. http://koalas.readthedocs.io.

[8] 2021. Modin. https://modin.readthedocs.io/en/latest/.

[9] 2021. Numpy. https://numpy.org/.

[10] 2021. Pandas. http://pandas.pydata.org/.

[11] 2021. Pandas Makes Python Better. https://towardsdatascience.com/pandas-makes-python-better-ec6cc1e30233/.

[12] 2021. PandasSchema. https://tmiguelt.github.io/PandasSchema/.

[13] 2021. Sberbank Russian Housing Market. https://www.kaggle.com/c/sberbank-russian-housing-market/overview/description.

[14] 2021. Scikit-learn. https://scikit-learn.org/stable//.

[15] 2021. Vaex. https://github.com/vaexio/vaex.

[16] Michael Armbrust et al. 2015. Scaling Spark in the real world: Performance and usability. *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1840–1843.

[17] Niels Bantilan. 2020. pandera: Statistical Data Validation of Pandas Dataframes. In *Proceedings of the Python in Science Conference (SciPy)*. 116 – 124.

[18] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1387–1395.

[19] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Infrastructure for Machine Learning. In *SysML Conference.*

[20] Jason Brownlee. 2020. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.* Machine Learning Mastery.

[21] Li Cai and Yangyong Zhu. 2015. The challenges of data quality and data quality assessment in the big data era. *Data science journal* 14 (2015).

[22] Emily Caveness, Paul Suganthan GC, Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. TensorFlow Data Validation: Data Analysis and Validation in Continuous ML Pipelines. In *ACM International Conference on Management of Data (SIGMOD)*. 2793–2796.

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

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

[25] Leo L Pipino, Yang W Lee, and Richard Y Wang. 2002. Data quality assessment. *Commun. ACM* 45, 4 (2002), 211–218.

[26] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. 2019. Differential Data Quality Verification on Partitioned Data. In *IEEE International Conference on Data Engineering (ICDE)*. 1940–1945.

[27] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 1781–1794.

[28] Shrey Shrivastava, Dhaval Patel, Anuradha Bhamidipaty, Wesley M Gifford, Stuart A Siegel, Venkata Sitaramagiridharganesh Ganapavarapu, and Jayant R Kalagnanam. 2019. DQA: Scalable, Automated and Interactive Data Quality Advisor. In *IEEE International Conference on Big Data (Big Data)*. 2913–2922.