CMPT 459/984 Course Project Final Report [Group 9]

Xin Tan
Simon Fraser University
Burnaby, BC, Canada
xta28@sfu.ca

Yuyu Lai
Simon Fraser University
Burnaby, BC, Canada
yuyu_lai@sfu.ca

ABSTRACT

Many companies and institutions rely on data to make informative, data-driven decisions. As a result, maintaining data quality is essential, since any missing or inaccurate data can detrimentally affect both business operations and decision-making processes downstream. Evidently, data quality verification is an integral part of the data science life-cycle. Our research aims to expand on the paper *Automating Large Scale Data Quality Verification*, in which the authors proposed a system called **Deequ** [11]. The system is built on top of Apache Spark for scalability, with the main functionality of allowing users to write unit tests for data while using machine learning to perform constraint suggestions and anomaly detection. Our research will take this approach a step further by evaluating the system on streaming data, **benchmarking** it against another state-of-the-art data quality verification tool, **DuckDQ**, and implementing KNN imputation based on the suggestions of Deequ. Our findings suggest that both systems have the ability to compute different settings of data (streaming and static). **For small to medium sized fixed datasets, we recommend using DuckDQ for its shorter runtime. For large, streaming datasets, we recommend Deequ for its ability to perform parallel computing and incremental computation for an overall view of the metrics across growing datasets.**

1 INTRODUCTION

Schelter et al. [11] emphasized the importance and impact necessity for quality data in modern companies and institutions. To attain quality data without it being tedious and repetitive, they proposed a declarative API that automates large-scale data quality verification. Based on this paper, they subsequently released a library on GitHub called Deequ.

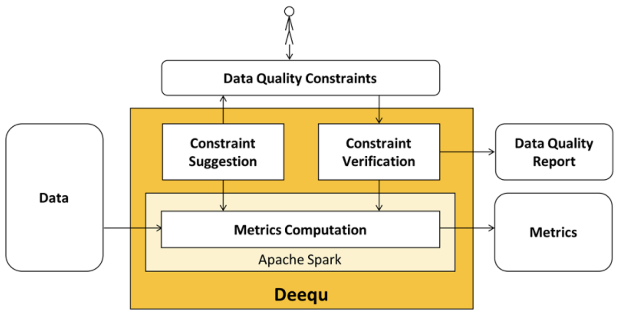


Figure 1: Deequ Architecture

To achieve scalability so that it can seamlessly scale to large datasets, they built the system to translate the required data metrics into aggregation queries, which can then be executed at scale with a distributed analytics engine such as Apache Spark [10]. In addition, the system provides flexibility by enabling integration with external data sources and custom validation code. Deequ's architecture is as follows: [10, 11].

Using Deequ, users are able to perform the following:

- Define unit-tests for data by combining common quality constraints with user-defined validation code.
- Leverage the machine learning component to enhance constraint suggestions and detect anomalies in time series data.

While the product presented by the author demonstrated its capabilities to streamline data validation processes effectively, we believe we can gain valuable insights on the system by benchmarking it against other state-of-the-art data quality verification systems. From our research, we found that Deequ has been compared to DuckDQ in terms of runtime. As a lightweight alternative to Deequ, DuckDQ avoids the overhead of having to start and orchestrate Apache Spark jobs. To confirm this claim, we ran a few experiments with varying criterion.

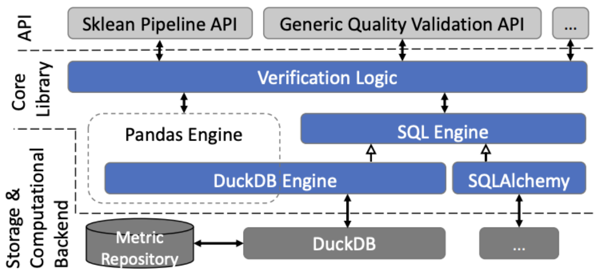


Figure 2: DuckDQ Architecture

The dataset we used is a JSON dataset called *'StockTicks.json'* which contained 1000 rows. Within the dataset, there are 8 columns in which some columns contained string values and some contained numerical values. The first area we evaluated was both systems' performance on fixed dataset. To perform this test, the dataset was uploaded onto DBFS FileStore. Then, the verification checks were ran on different constraints to evaluate the runtime.

Then, we evaluated on both systems' performance on two different data settings: fixed and streaming. Since the paper was only evaluated on fixed data, it was beneficial to observe its performance when applied on streaming data. To attain streaming data, we modified the setting so that we can access historical data while verifying the current using delta lake on Databricks.

Based on the suggestions provided by the system after applying it on the different data settings, we applied a data imputation method, KNN imputation, to achieve a more complete dataset. From our findings, we can make informed suggestions by comparing the results of each experiment and recommend which system would be the most suitable under certain use cases.

2 RELATED WORK

Unit Test for Data The authors of the paper proposed a declarative API for automating large scale data validation [11]. As of the publishing date of the paper, Deequ is one of the few tools available to perform large-scale data validation with high flexibility, with support for incremental computation on growing datasets, and scalability.

Another tool that was compared to Deequ was DuckDQ, an Python library for data quality verification, which integrates with scikit-learn to facilitates the creation of ML pipelines [4]. The authors of the paper 'DuckDQ: Data Quality Assertions for Machine Learning Pipelines' outlined that Deequ was "explicitly designed for very large datasets, and require cloud infrastructure as well as manual integration into training and serving systems"[4]. They stated that the dependency on Apache Spark introduces unnecessary overhead on small and medium-sized data. Hence, they presented DuckDQ, a system which does not require a distributed environment and is easy to use. According to the paper, DuckDQ outperforms existing solutions by a factor of 3 to 40 in runtime. An experiment done by the group who created Deequ has also found that DuckDQ performs competitively to Deequ in terms of runtime.

The authors of DuckDQ also referenced Google's TFX [8]. TensorFlow Extended (TFX) is a TensorFlow-Based general purpose machine learning platform which allows for producing and deploying machine learning models while incorporating modules for analyzing and validating both the data and the model. The system being an end-to-end machine learning platform limits the use cases and flexibility of its applications outside of the platform.

Fadlallah et al.[5] proposed *BIGQA*, a flexible framework that supports data quality assessments by producing customized data quality reports while running efficiently on distributed frameworks [5]. It also allows for incremental data quality assessments to avoid reading the whole dataset in every iteration. However, it does not involve a form of predictability metrics or the usage of machine learning to learn relationships between columns. As well, Bicevskis et al. [1] proposed an approach to data quality evaluation which comprises of a graphical interface to expand the usage to non-IT professionals. However, their system focuses on platform independent(PIM) and platform specific (PSM) models and was not designed with the intention of scalability.

Galhardas et al. [6] present a language, an execution model and a set of algorithms for users to express data cleaning specifications. The authors go into detail the semantics of using this language and its different use cases on matching, transformations and user interactions.

Chandel et al. [2] highlights the use of declarative statements to discover data primitives on top of relational data sources. It emphasizes ease of use and integration with existing applications. The tool performs approximate selection predicates by allowing data deduplication and record linkage, however, they did not focus on the scalability aspect.

Yakout et al. [12] proposed a novel system named SCARE (Scalable Automatic Repairing) that uses machine learning and likelihood methods to learn the correlations from the ground-truth data to predict the replacement values. It uses horizontal data partitions to achieve scalability and data partitioning for parallel computing.

De et al. [3] introduces BayesWipe to modify data attributes in relational databases. The method uses both Bayesian generative and statistical error models to identify the data mistakes and repairs them by assigning probabilities to the replacement values. It also uses MapReduce to meet scalability demands.

Data Imputation In the paper *Comparison of Performance of Data Imputation Methods for Numeric Dataset*, the authors compared seven data imputation tools, mean imputation, median imputation, KNN imputation, predictive mean matching, Bayesian Linear Regression, Linear Regression, non- Bayesian, and random sample. Data imputation is the ability to estimate the plausible values to substitute the missing ones. By evaluating the methods on a data mining task and five different datasets, they found that KNN imputation outperforms other methods based on the Root Mean Square Error scores of each of the methods [7].

In another paper, the authors evaluated the performance of different imputation algorithms on four different categories of missing data: Structurally Missing Data (SMD), Missing Completely At Random (MCAR), Missing At Random (MAR) and Missing Not At Random (MNAR) [9]. Between the algorithms KNN, Random Forest, Mean, Amelia and Mice, they found that KNN and Random Forest outperformed the other algorithms consistently across all cases. Since Random Forest is more computationally intensive, they recommended that KNN would be the more suitable approach for missing data imputation.

BitFlip Based on our survey of all of the existing systems, it is apparent that there is lack in-depth benchmarking between data verification tools on different settings of data, specifically on fixed and streaming data. **Therefore, our research aims to provide valuable insight on the performance and areas of improvements for the two state-of-the-art data verification tools, Deequ and DuckDQ.** There also does not exist an end-to-end data verification system that showed the ability to perform data imputation based on the constraint suggestions. Although not required, it is a valuable extension to show the users possible areas and ways to improve their dataset. We will be using the runtimes of both Deequ and DuckDQ as baselines for our analysis.

3 METHOD & IMPLEMENTATION

3.1 Method

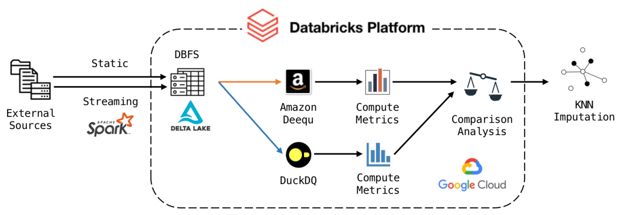
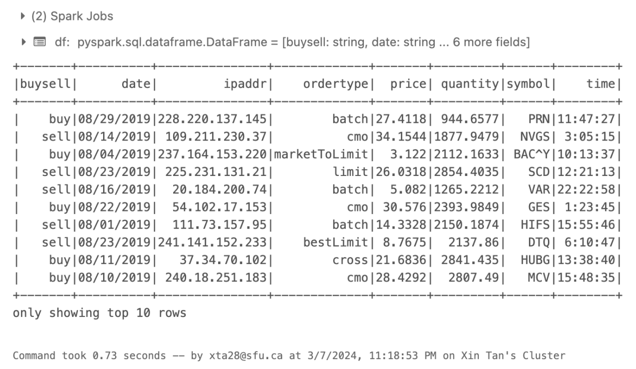


Figure 3: Project Architecture

Our solution is shown in Figure 3, which involves leveraging Apache Spark, Databricks Data Intelligence Platform and Google Cloud Platform (GCP) to construct a data processing pipeline capable of handling both streaming and static data scenarios.

3.2 Data

Our original data *'stockTicks.json'* is a static JSON file obtained from Mockaroo APIs, containing 1000 rows of transaction records for various stock symbols, with a mix of string and numerical values in 8 attributes such as date, price, buy/sell indicator, and IP address, etc, essential for analyzing market trends and trading behaviors.



```
(2) Spark Jobs
df:  pyspark.sql.dataframe.DataFrame = [buysell: string, date: string ... 6 more fields]

+-------+----------+---------------+-------------+-------+---------+------+--------+
|buysell|      date|         ipaddr|    ordertype|  price| quantity|symbol|    time|
+-------+----------+---------------+-------------+-------+---------+------+--------+
|    buy|08/29/2019|228.220.137.145|        batch|27.4118| 944.6577|   PRN|11:47:27|
|   sell|08/14/2019| 109.211.230.37|          cmo|34.1544|1877.9479|  NVGS| 3:05:15|
|    buy|08/04/2019|237.164.153.220|marketToLimit|  3.122|2112.1633| BAC^Y|10:13:37|
|   sell|08/23/2019| 225.231.131.21|        limit|26.0318|2854.4035|   SCD|12:21:13|
|   sell|08/16/2019|  20.184.200.74|        batch|  5.082|1265.2212|   VAR|22:22:58|
|    buy|08/22/2019|  54.102.17.153|          cmo| 30.576|2393.9849|   GES| 1:23:45|
|   sell|08/01/2019|  111.73.157.95|        batch|14.3328|2150.1874|  HIFS|15:55:46|
|   sell|08/23/2019|241.141.152.233|    bestLimit| 8.7675|  2137.86|   DTQ| 6:10:47|
|    buy|08/11/2019|   37.34.70.102|        cross|21.6836| 2841.435|  HUBG|13:38:40|
|    buy|08/10/2019| 240.18.251.183|          cmo|28.4292|  2807.49|   MCV|15:48:35|
+-------+----------+---------------+-------------+-------+---------+------+--------+
only showing top 10 rows

Command took 0.73 seconds -- by xta28@sfu.ca at 3/7/2024, 11:18:53 PM on Xin Tan's Cluster
```

Figure 4: 'The stockTicks.json' Data

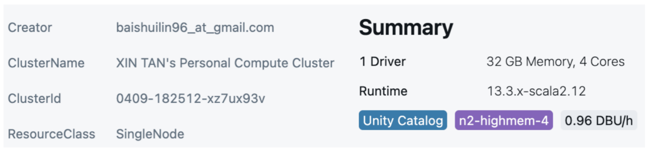
For static data, we upload it into the Databricks File System (DBFS) and load directly from there. We found that this method is the best way to avoid any corrupted data.

For streaming data, a simulation environment is created by streaming the same static source file to mimic a real-time data feed. This setup is instrumental in assessing Deequ's efficacy in continuous data verification against the traditional one-time validation for static datasets.

3.3 Quality Checks Implementation

We first successfully tested Deequ's performance for both static and streaming data source, then revised the codes to performed similar test on DuckDQ. So in this section, we will mainly explain how we conduct quality checks based on Deequ's example.

3.3.1 Cluster Configuration. At the beginning we managed our cloud resources by creating stack on the AWS CloudFormation platform. However, we moved to Google Cloud Platform (GCP) due to the budget limitation later on.

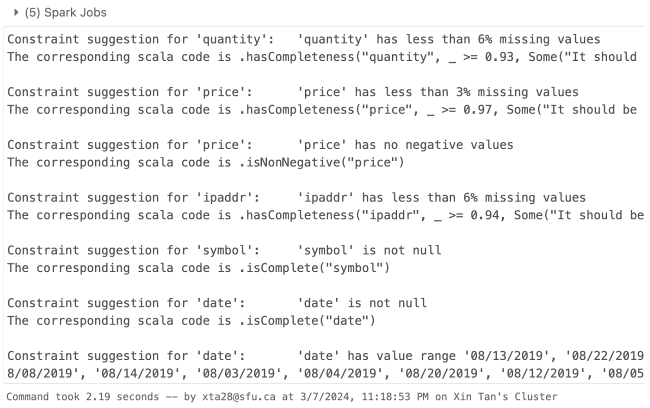


Creator	baishuilin96_at_gmail.com	**Summary**	
ClusterName	XIN TAN's Personal Compute Cluster	1 Driver	32 GB Memory, 4 Cores
ClusterId	0409-182512-xz7ux93v	Runtime	13.3.x-scala2.12
ResourceClass	SingleNode	Unity Catalog	n2-highmem-4 0.96 DBU/h

Figure 5: Compute Cluster

The experiments were carried out on a runtime version 13.3 LTS (includes Apache Spark 3.4.1, Scala 2.12), Node type is 'n2-highmem-4' (32 GB Memory, 4 Cores) to ensure efficient resource utilization.

3.3.2 Establishing Analysis Infrastructure. We first add the '2.0.6-spark-3.4' version Deequ package from Maven Central using `com.amazon.deequ`. Then utilize the previously saved data to initialize three Delta tables: `trades_delta`, `bad_records`, and `deequ_metrics`. Next, we perform several preparation for streaming anaylsis:



```
(5) Spark Jobs
Constraint suggestion for 'quantity':   'quantity' has less than 6% missing values
The corresponding scala code is .hasCompleteness("quantity", _ >= 0.93, Some("It should

Constraint suggestion for 'price':      'price' has less than 3% missing values
The corresponding scala code is .hasCompleteness("price", _ >= 0.97, Some("It should be

Constraint suggestion for 'price':      'price' has no negative values
The corresponding scala code is .isNonNegative("price")

Constraint suggestion for 'ipaddr':     'ipaddr' has less than 6% missing values
The corresponding scala code is .hasCompleteness("ipaddr", _ >= 0.94, Some("It should be

Constraint suggestion for 'symbol':     'symbol' is not null
The corresponding scala code is .isComplete("symbol")

Constraint suggestion for 'date':       'date' is not null
The corresponding scala code is .isComplete("date")

Constraint suggestion for 'date':       'date' has value range '08/13/2019', '08/22/2019
8/08/2019', '08/14/2019', '08/03/2019', '08/04/2019', '08/20/2019', '08/12/2019', '08/05
Command took 2.19 seconds -- by xta28@sfu.ca at 3/7/2024, 11:18:53 PM on Xin Tan's Cluster
```

Figure 6: Constraint Suggestion Results

(1) **Generating Quality Constraint Suggestions:** We employ Deequ's `ConstraintSuggestionRunner` to analyze historical data to automatically generate quality constraint suggestions. Deequ examines the provided data and produces constraints assuming future data should appear similar. These suggestions help us in understanding potential constraints for improving data quality. Below are some computable metrics to base constraints on:
   - `Size`: returns the number of records
   - `ApproxCountDistinct`: returns the approximate count of distinct values in a column
   - `Distinctness`: returns the fraction of (distinct values / total values) in a column
   - `Completeness`: returns the fraction of values that are non-null in a column
   - `Compliance`: returns the fraction of values in a column that meet a given constraint

(2) **Selecting and Setting Up Analyzers:** We then choose some constraints generated by Deequ for actual use, set up a series of analyzers, includes analyzers for data size (`Size()`), approximate count of distinct stock symbols (`ApproxCountDistinct("symbol")`), completeness of IP address, quantity, and price (`Completeness`), and compliance check for quantity (`Compliance("top quantity", "quantity >= 0")`).

3.3.3 Static Test. The quality checks for static data is quite easy and we simply follow what the authors described in the paper.

The only difference is that our main purpose here is to compare runtime in real-world user-defined data quality checks scenario. So after loading the 'stockTicks.json' into a dataframe, we established a heavy test by applying common constraints from all 3 dimensions (completeness, consistency and statistics) on all the 8 attributes, like *'.hasSize(_ == 1000)'*, *'.hasDistinctness(Seq("time"), _ >= 0.1)'*, *'.hasApproxQuantile("price", 0.5, _ <= 40)'*, *'.isComplete("symbol")'*, *'.isUnique("buysell")'*, *'.isNonNegative("quantity")'*, etc.

3.3.4 *Streaming Test: Streaming Simulation.*
For streaming, we want to make sure the results can be validated, so in the following steps, we simulate the streaming based on the same static dataset, enabling analysis in a streaming context without real-time data generation.

(1) **Read and Store as Parquet:** We use Spark to read and repartition the JSON data into 100 partitions, then write to a temporary folder in Parquet format.
(2) **Simulate Streaming:** We simulate a data stream from the Parquet files using `.readStream`, with `.option("maxFilesPerTrigger",1)` to limit file processing per trigger.
(3) **Write to Delta Tables:** Stream-read data is written to `trades_delta` Delta table for analysis. This simulates a streaming scenario, allowing Deequ's data quality checks on "streaming" data.

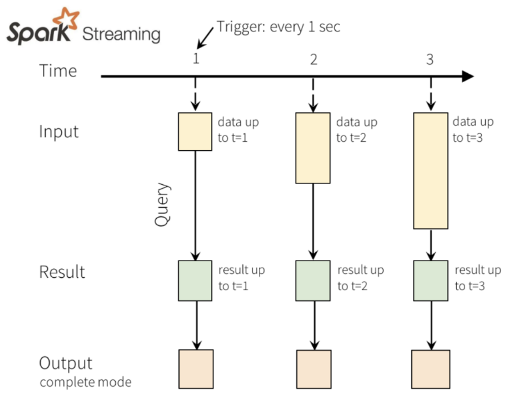


Figure 7: Spark Streaming

3.3.5 *Streaming Test: Reading from Delta Table.*
Delta Lake is the optimized storage layer on Databricks developed for tight integration with Structured Streaming, allowing users to easily use a single copy of data for both batch and streaming operations and providing incremental processing at scale.

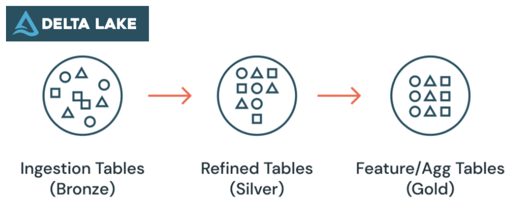


Figure 8: Delta Lake on Databricks

So we then read the data stream from the Delta table and utilizes the `foreachBatch` function to run Deequ analysis on each batch of data. This includes:

- Setting up stateStorage
- Conducting analysis on the current batch
- Performing unit validation: adding the batch to an error records table `bad_records` if validation fails
- Updating the metrics table `deequ_metrics` to reflect the latest aggregated metrics collected from all streaming data records

Figure 9 compares the number of records ingested to the number processed per second, as well as the time taken to process each batch of records in seconds, revealing the system's throughput and stability over time, providing insight into the efficiency and consistency of the data processing pipeline.

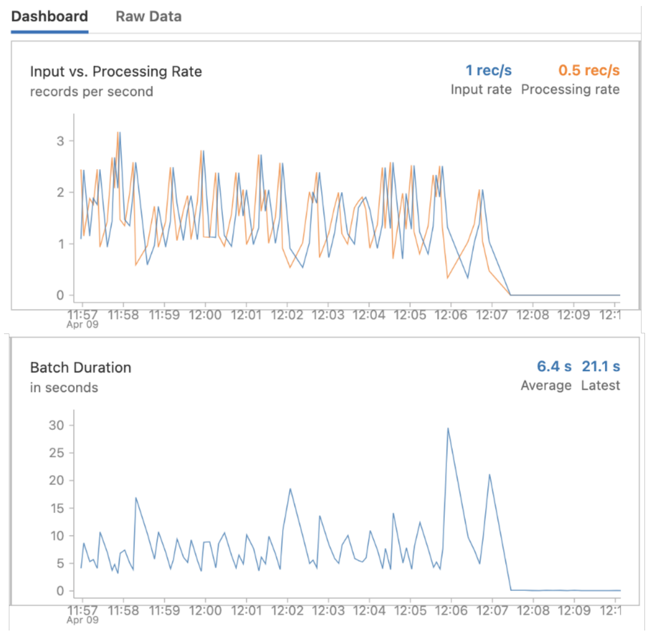
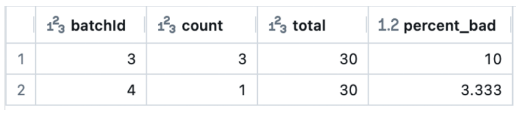


Figure 9: Streaming Job Performance

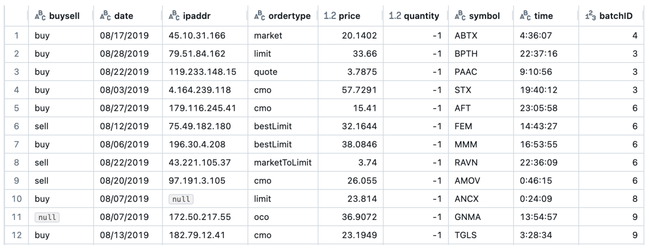
3.3.6 *Streaming Test: Visualizing Quality Metrics.*
Each update of the metrics will be written as a duplicate entry, we can parse this out to only take the latest view, or use this to create a time series view of the data quality.

We calculate the total number of records for each batch in the `bad_records` table to find the ones with the most severe data quality issues.



	batchId	count	total	percent_bad
1	3	3	30	10
2	4	1	30	3.333

Figure 10: Bad Records per Batch



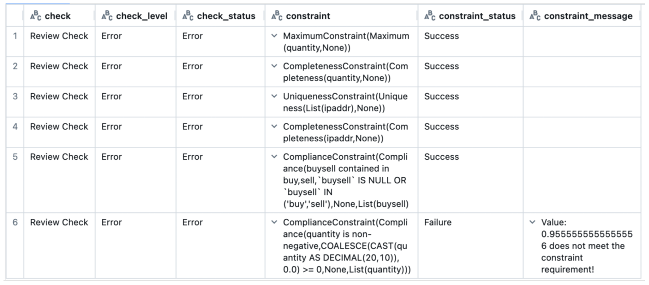
	buysell	date	ipaddr	ordertype	price	quantity	symbol	time	batchID
1	buy	08/17/2019	45.10.31.166	market	20.1402	-1	ABTX	4:36:07	4
2	buy	08/28/2019	79.51.84.162	limit	33.66	-1	BPTH	22:37:16	3
3	buy	08/22/2019	119.233.148.15	quote	3.7875	-1	PAAC	9:10:56	3
4	buy	08/03/2019	4.164.239.118	cmo	57.7291	-1	STX	19:40:12	3
5	buy	08/27/2019	179.116.245.41	cmo	15.41	-1	AFT	23:05:58	6
6	sell	08/12/2019	75.49.182.180	bestLimit	32.1644	-1	FEM	14:43:27	6
7	buy	08/06/2019	196.30.4.208	bestLimit	38.0846	-1	MMM	16:53:55	6
8	sell	08/22/2019	43.221.105.37	marketToLimit	3.74	-1	RAVN	22:36:09	6
9	sell	08/20/2019	97.191.3.105	cmo	26.055	-1	AMOV	0:46:15	6
10	buy	08/07/2019	null	limit	23.814	-1	ANCX	0:24:09	8
11	null	08/07/2019	172.50.217.55	oco	36.9072	-1	GNMA	13:54:57	9
12	buy	08/13/2019	182.79.12.41	cmo	23.1949	-1	TGLS	3:28:34	9

Figure 11: Real-time Bad Records

Meanwhile, the *real-time display* generated is for immediate monitoring and rapid identification of problematic records.

Finally, we use the `Verification Suite` to run defined data quality checks (e.g., the maximum value of `quantity` and verifying that it is non-negative, the completeness of `quantity` and `ipaddr`, the uniqueness of `ipaddr`, and the validity of the `buysell` field), the results of these checks are then converted into a Spark DataFrame for display.

In Figure 12, each row corresponds to a different check, indicating whether it succeeded or failed, along with a descriptive message. For example, the last row shows a failure in the non-negativity check of the quantity field, suggesting that there might be negative values present which violate the set constraint.



	check	check_level	check_status	constraint	constraint_status	constraint_message
1	Review Check	Error	Error	MaximumConstraint(Maximum(quantity,None))	Success	
2	Review Check	Error	Error	CompletenessConstraint(Completeness(quantity,None))	Success	
3	Review Check	Error	Error	UniquenessConstraint(Uniqueness(List(ipaddr),None))	Success	
4	Review Check	Error	Error	CompletenessConstraint(Completeness(ipaddr,None))	Success	
5	Review Check	Error	Error	ComplianceConstraint(Compliance(buysell contained in buy,sell,`buysell` IS NULL OR `buysell` IN ('buy','sell'),None,List(buysell)	Success	
6	Review Check	Error	Error	ComplianceConstraint(Compliance(quantity is non-negative,COALESCE(CAST(quantity AS DECIMAL(20,10)), 0.0) >= 0,None,List(quantity)))	Failure	Value: 0.9555555555555556 does not meet the constraint requirement!

Figure 12: Check Results Summary

The pie chart below illustrates the distribution of constraint validation results from the data quality checks, it shows that the majority of checks have passed. However, the 'Failure' segment suggests that there is still a portion of data that does not comply with certain quality standards, which requires further investigation to identify the underlying issues.

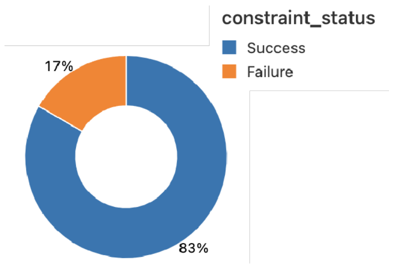
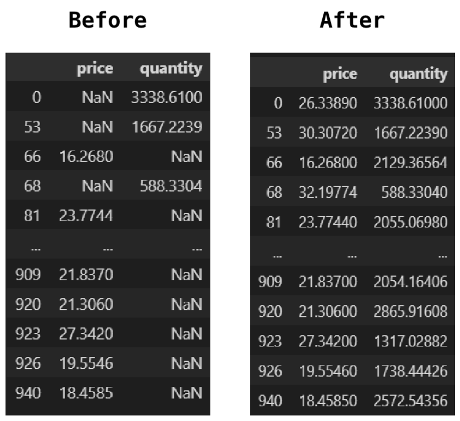


Figure 13: Unit Validation Status

3.4 Data Imputation using KNN

Based on the suggestions by Deequ on the completeness of the data, we performed data imputation to replace the missing values with substitute values. From the survey of data imputation method on numerical columns, it seems the best performing one is KNN. KNN Imputation uses the K-nearest neighbours algorithm to replace the missing values with the mean value of the closest neighbours found in the dataset.

Before running imputation, Deequ computed the completeness of the Price column to be 98.2% and the Quantity column to be 95.4%. Based on this information, we ran Scikit-learn's KNNImputer function to perform KNNImputation and filled in the missing values. The before and after imputation view of the dataset can be seen below.



Before

	price	quantity
0	NaN	3338.6100
53	NaN	1667.2239
66	16.2680	NaN
68	NaN	588.3304
81	23.7744	NaN
...	...	...
909	21.8370	NaN
920	21.3060	NaN
923	27.3420	NaN
926	19.5546	NaN
940	18.4585	NaN

After

	price	quantity
0	26.33890	3338.61000
53	30.30720	1667.22390
66	16.26800	2129.36564
68	32.19774	588.33040
81	23.77440	2055.06980
...	...	...
909	21.83700	2054.16406
920	21.30600	2865.91608
923	27.34200	1317.02882
926	19.55460	1738.44426
940	18.45850	2572.54356

Figure 14: KNN Imputation on Price and Quantity Columns with Missing Values

The reason for this implementation is to demonstrate how data verification systems such as Deequ can take their constraint suggestions a step further by resolving those constraints.

4 EVALUATION

We evaluated our system design based on different criterion for the different data settings:

(1) Static Test
- **Runtime:** The individual runtime for executing each of the five distinct constraints separately and the cumulative runtime for executing all constraints concurrently were analyzed.

(2) Streaming Test
- **Accuracy:** How closely the value of those metrics are compared to the true completeness and distinctness percentages of the dataset.
- **Dynamic Performance:** The computation of the completeness and distinctness metrics of Deequ and DuckDQ over multiple batches.

4.1 Static Test Evaluation

DuckDQ was designed as a lightweight version of Deequ without the overhead of running Apache Spark. It boasts a much shorter runtime without any overhead of computation or job provisioning in comparison to Deequ.

So we run each of the systems on a varying amount of constraints to test. For Completeness, Uniqueness and Distinctness, we ran them on all eight columns. For isApproxQuantile and NonNegative, we only ran them on the numerical columns. In total, we ran 28 constraint computes on each systems.

Table 1: Runtime Comparison

Constraints	Deequ	DuckDQ
Completeness	0.49	0.16
Uniqueness	3.6	0.21
Distinctness	3.34	0.21
isApproxQuantile	0.68	0.15
NonNegative	1.03	0.18
Full Run	5.89	0.22

Based on the table above, we can see that DuckDQ's claims can be validated, their runtimes are consistently signficantly shorter than Deequ. We also compared the constraint results between the two systems and found that most of them are the same, as they should be. However, interestingly, there were four conflicting results:

Table 2: Conflicting Results

Constraint Computes	DuckDQ	Deequ
DistinctnessConstraint(Distinctness(buysell))	0.003	0.002
UniquenessConstraint(Uniqueness(ipaddr))	0.957	1
UniquenessConstraint(Uniqueness(price))	0.966	0.983
UniquenessConstraint(Uniqueness(quantity))	0.909	0.952

We found that both systems produce inconsistent Uniqueness and Distinctness results, with the results from DuckDQ being closer to the ground truth that was computed locally using Python. Based on the runtimes and the metrics, we would recommend using DuckDQ to perform data quality verification on small to medium sized datasets.

4.2 Streaming Test Evaluation

In streaming test, for quality checks, we only pick 2 typical constraints *Completeness* and *Distinctness* over 4 instances: ipaddr, quantity, price and symbol.

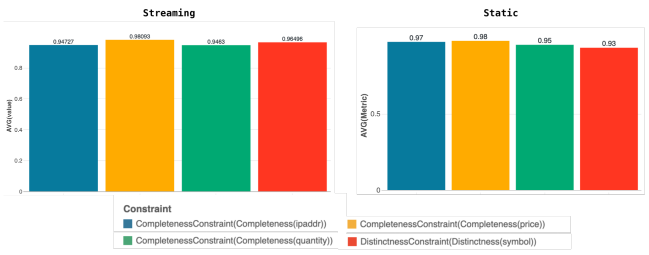


Figure 15: Streaming and Static Results Comparison

4.2.1 Accuracy. From the results here we can say that Deequ and DuckDQ are reliable for streaming data. The average results of both the tools across batches stay similar to the static data, it's reasonable they should not be exactly the same since the batches might varies a bit each time we run streaming.

4.2.2 Dynamic Performance. When we run streaming test using Deequ, in Figure 16, each line's progression illustrates an upward trend, suggesting that as the size growing, the approximate count of distinct values in the *symbol* column grows too.

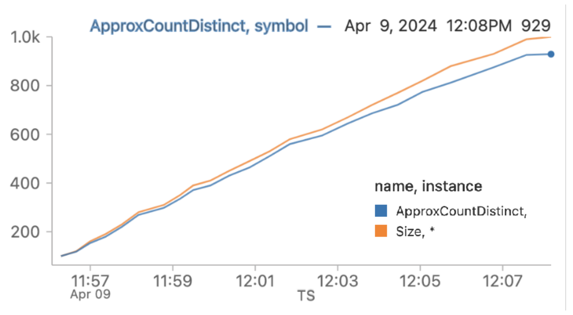


Figure 16: Trends in Data Volume and Uniqueness

And the Distinctness (returns the fraction) of *symbol* keeps going down on the right, while the 3 completeness values fluctuate at the start and stabilize in the end, but all within [0.94, 1].

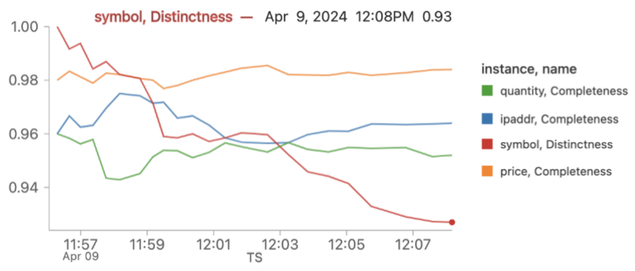
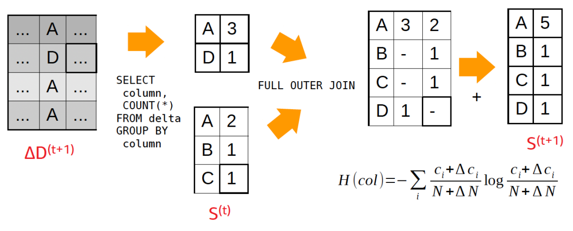


Figure 17: Historical Constraint Values of Deequ

We noticed that Deequ aggregates its metric scores over the batches. This corresponds to the original paper where the authors outlined the benefits of incremental computation on scalability.



$$H(col) = -\sum_i \frac{c_i + \Delta c_i}{N + \Delta N} \log \frac{c_i + \Delta c_i}{N + \Delta N}$$

Figure 18: Deequ's Example Update of Column:

The way they achieved this is by internally maintaining a histogram of the frequencies per value in a column and when each delta table introduces new values, the histogram updates itself. This behaviour is consistent with what we observed in our streaming data outcome in the figure above.

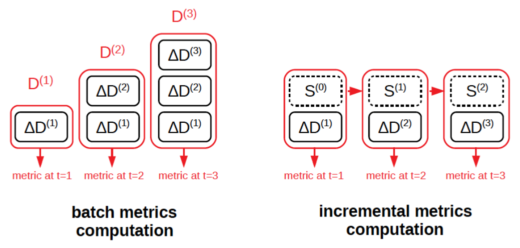


Figure 19: Batch vs. Incremental Computation of Metrics

In contrast to Deequ, DuckDQ does not maintain any information about its previous computations. As seen in the figure below, the metrics are calculated by batch, therefore we can see that there is no patterns of trends overtime for the corresponding metrics.

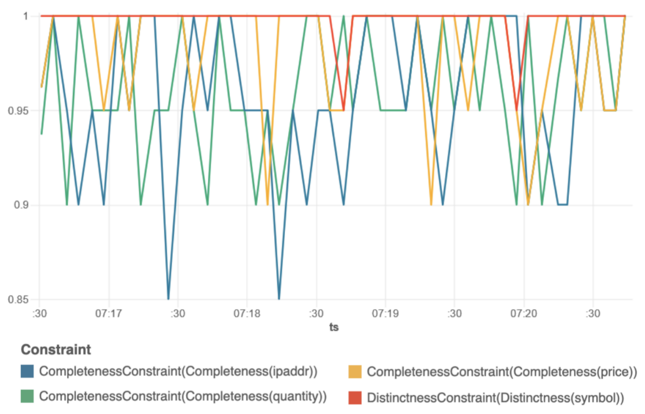


Figure 20: Historical Constraint Values of DuckDQ

In most cases, it is more beneficial to see the overall, aggregated trend of the data verification metrics. Therefore, in this evaluation we concluded that the incremental computation functionality of Deequ is more practical and informative.

5 DISCUSSIONS

The limitations of our work include not comparing enough constraints since we focused on comparing the completeness and distinctness of the dataset. The dataset we used also only contains 1000 rows due to concerns of the resources required to run more computation intensive workloads and the cost of that on Google Cloud Platform.

Our future work would be to test as many constraints as possible across the two systems to fully realize their performance and limitations. We would also test on varying sizes and types of data to evaluate their performance and scalability as well.

6 CONCLUSION

In our project, we benchmarked Deequ against DuckDQ on different data settings. We found that the accuracy of for both systems are relatively the same. However, DuckDQ runs significantly faster on small datasets. Deequ aggregates the results on streaming data due to its incremental computation feature while DuckDQ computes the results by batch with no information retained across batches. Therefore, our final conclusion would be:

- Capacity: Both Deequ and DuckDQ has the ability to handle multiple types of data.
- For static and small to medium size data: DuckDQ is better due to its runtime.
- For streaming data: Deequ is better because it incorporates incremental computation so that the overall metrics of streaming data could be observed.

REFERENCES

[1] Janis Bicevskis, Zane Bicevska, Anastasija Nikiforova, and Ivo Oditis. 2018. *An Approach to Data Quality Evaluation.* https://doi.org/10.1109/SNAMS.2018.8554915

[2] Amit Chandel, Oktie Hassanzadeh, Nick Koudas, Mohammad Sadoghi, and Divesh Srivastava. 2007. Benchmarking declarative approximate selection predicates. *Computing Research Repository - CORR*, 353–364. https://doi.org/10.1145/1247480.1247521

[3] Sushovan De, Yuheng Hu, Venkata Vamsikrishna Meduri, Yi Chen, and Subbarao Kambhampati. 2016. BayesWipe: A Scalable Probabilistic Framework for Improving Data Quality. *J. Data and Information Quality* 8, 1, Article 5 (oct 2016), 30 pages. https://doi.org/10.1145/2992787

[4] Till Doehmen, Mark Raasveldt, Hannes Muehleisen, and Sebastian Schelter. 2021. *DuckDQ: Data Quality Assertions for Machine Learning Pipelines.*

[5] HADI FADLALLAH, RIMA KILANY, HOUSSEIN DHAYNE, RAMI EL HADDAD, RAFIQUL HAQUE, YEHIA TAHER, and ALI JABER. 2023. *BIGQA: Declarative Big Data uality Assessment.*

[6] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. 2001. Declarative Data Cleaning: Language, Model, and Algorithms. In *Very Large Data Bases Conference.* https://api.semanticscholar.org/CorpusID:8296909

[7] Anil Jadhav, Dhanya Pramod, and Krishnan Ramanathan. 2019. *Comparison of Performance of Data Imputation Methods for Numeric Dataset.* https://doi.org/10.1080/08839514.2019.1637138

[8] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. 2017. *TFX: A TensorFlow-Based Production-Scale Machine Learning Platform.*

[9] Ben Omega Petrazzini, Hugo Naya, Fernando Lopez-Bello, Gustavo Vazquez, and Lucía Spangenberg. 2021. *Evaluation of different approaches for missing data imputation on features associated to genomic data.*

[10] Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat, Phillipp Schmidt, Stephan Seufert, Pierre Brunelle, and Andrey Taptunov. 2019. *Unit Testing Data with Deequ.* Retrieved 2024-02-18 from https://doi.org/10.1145/3299869.3320210

[11] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. *Automating large-scale data quality verification.* Retrieved 2024-02-18 from https://doi.org/10.14778/3229863.3229867

[12] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. 2013. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 553–564. https://doi.org/10.1145/2463676.2463706