# Benchmarking:
# Deequ vs. DuckDQ's Performance on Static and Streaming Data

**CMPT 984 Project Presentation**

Xin Tan & Yuyu Lai
Apr 11, 2024

amazon | science

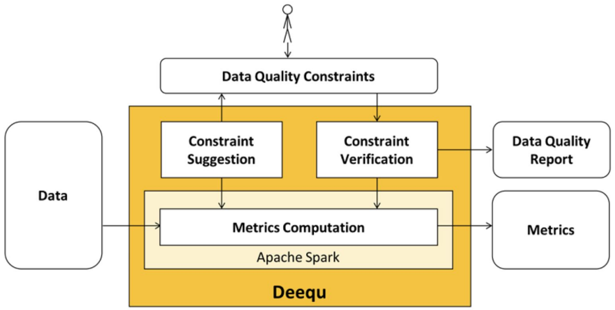
Google Cloud

Databricks

# Problem & Motivation

## Automating Data Quality Verification: Deequ

Library built on top of Apache Spark for defining "unit tests for data".



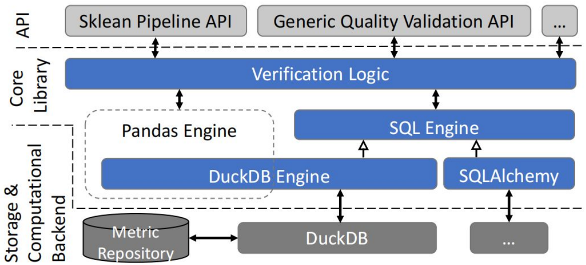

Open Source: [https://github.com/awslabs/deequ](https://github.com/awslabs/deequ)

- Metrics Computation
- Constraint Suggestion
- Constraint Verification
- Metrics Repository

[1] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification.

## Embedabble Data Quality Validation: DuckDQ

Python library that provides a fluent API for data quality checks.




Open Source: [https://github.com/tdoehmen/duckdq](https://github.com/tdoehmen/duckdq)

- Inspired by Deequ
- Excels at small to medium sized datasets
- Lightweight
- Outperforms existing solutions in runtime

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

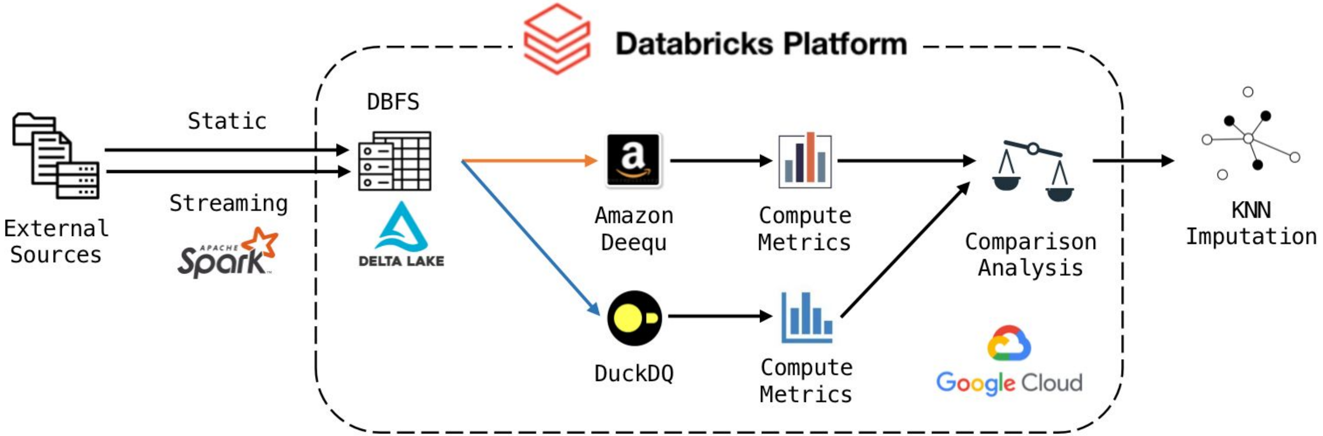
# Bit Flip

**Previous Work:**

- ❖ Deequ has only been evaluated on fixed data.
- ❖ No comparison with other state-of-the-art systems have been made.
- ❖ Lack of data imputation feature.

**Our Contribution:**

- ❖ Evaluate Deequ's performance on streaming data.
- ❖ Benchmark Deequ against DuckDQ.
- ❖ Based on the suggestions by Deequ, perform data imputation.



# Implementation Methods

**Dataset:** *'stockTicks.json' (1000 rows, 153KB)*

{"symbol":"BLDR","date":"08/01/2019","time":"6:35:51","price":3.5343,"quantity":3029.4181,"buysell":"sell","ordertype":"batch","ipaddr":"167.5.227.249"}

**Platform:** Databricks notebook workspace, Google Cloud Platform

Google Cloud

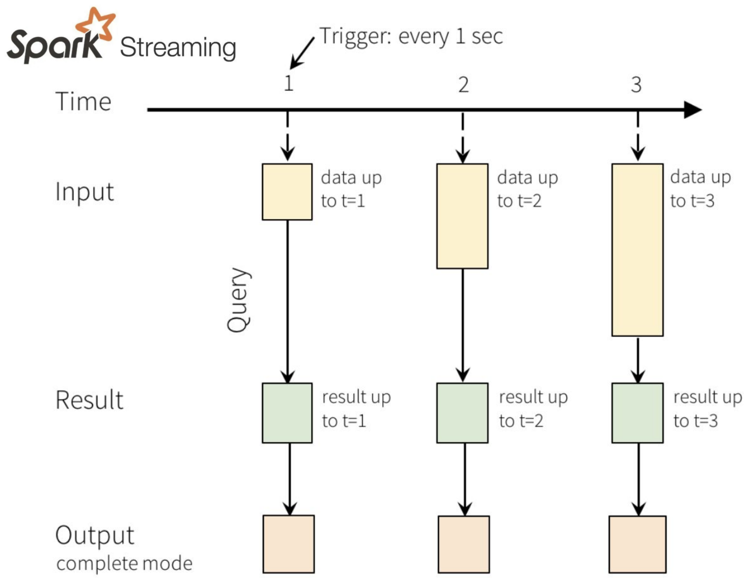
Databricks

Table 1. Experiment Setup

| | |
|---|---|
| **Compute Resources** | n2-highmem-4 (1 Driver, 32 GB Memory, 4 Cores) |
| **Runtime Version** | 13.3 LTS (includes Apache Spark 3.4.1, Scala 2.12) |
| **Static Data** | Uploaded file to DBFS |
| **Streaming Data** | Read and store as parquet, simulate streaming, write to Delta Tables |

# Streaming Data

**Simulate Streaming:** 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. **Streaming:** 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.

# Streaming Data

Cmd 15

Scala

```scala
%scala
// parse the schema for the source parquet
val schema = base_df.schema

// start the stream
spark.readStream
.schema(schema)
.format("parquet")
.option("maxFilesPerTrigger",1)
.load(data_path)
.writeStream.format("delta")
.option("failOnDataLoss", false)
.option("checkpointLocation", checkpoint_path)
.format("delta").table("trades_delta")
```

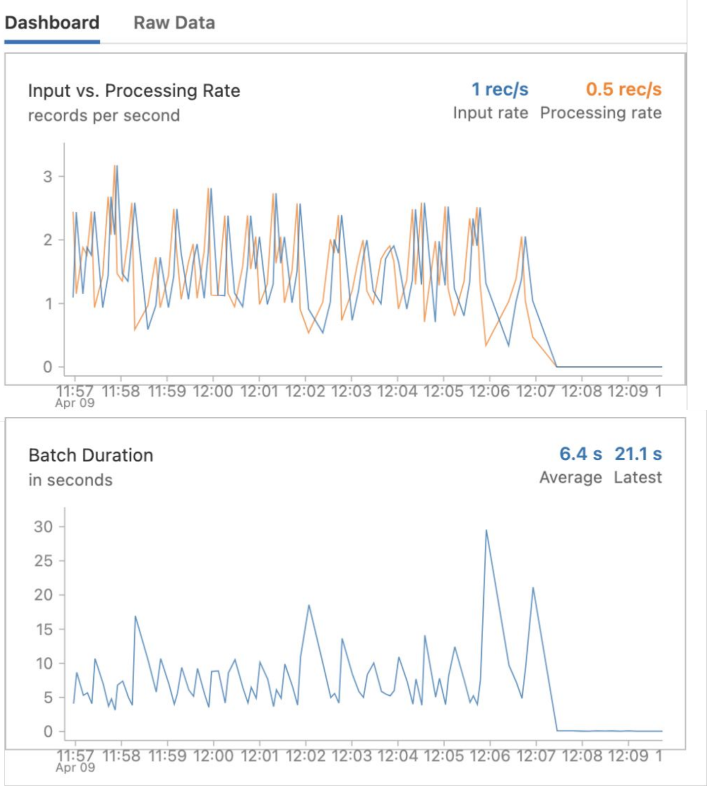
(1) Spark Jobs

Job 1794 View (Stages: 2/2)

87af1d53-422e-4889-b5c2-fdefed208ada *Last updated: 3 days ago*

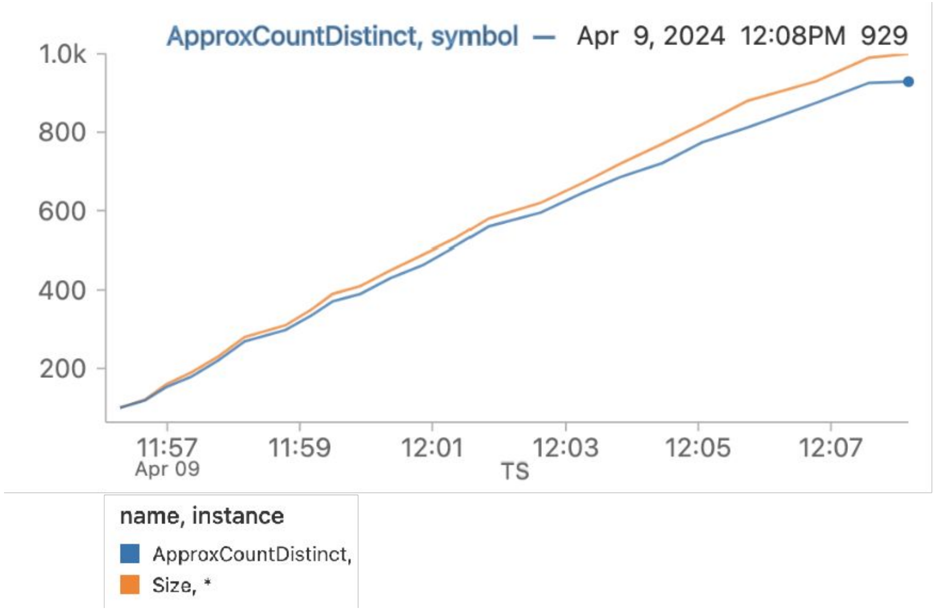
```
schema: org.apache.spark.sql.types.StructType = StructType(StructField(buysell,StringType,true),StructField(date,StringType,true),StructField(ipaddr,StringType,true),StructField(ordertype,StringType,true),StructField(price,DoubleType,true),StructField(quantity,DoubleType,true),StructField(symbol,StringType,true),StructField(time,StringType,true))
res7: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@24df4553
```

Command complete

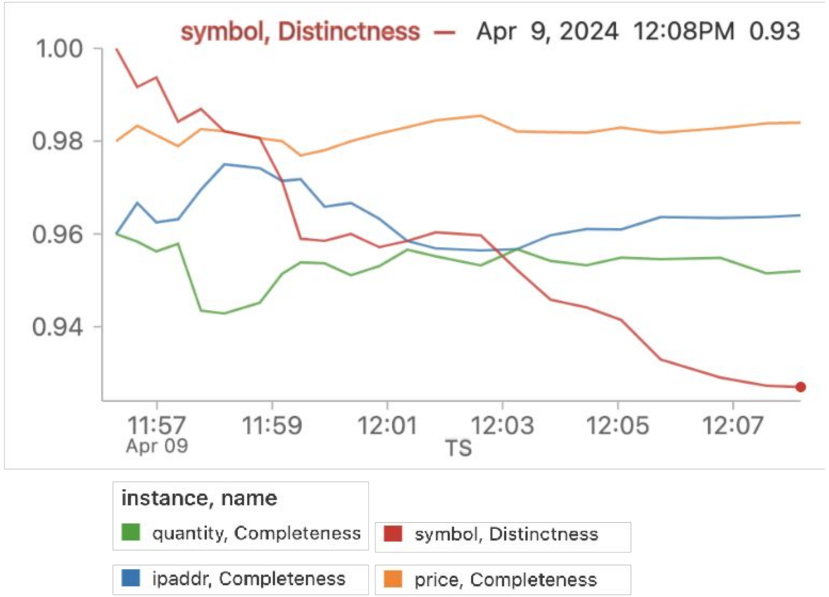


# Streaming Data

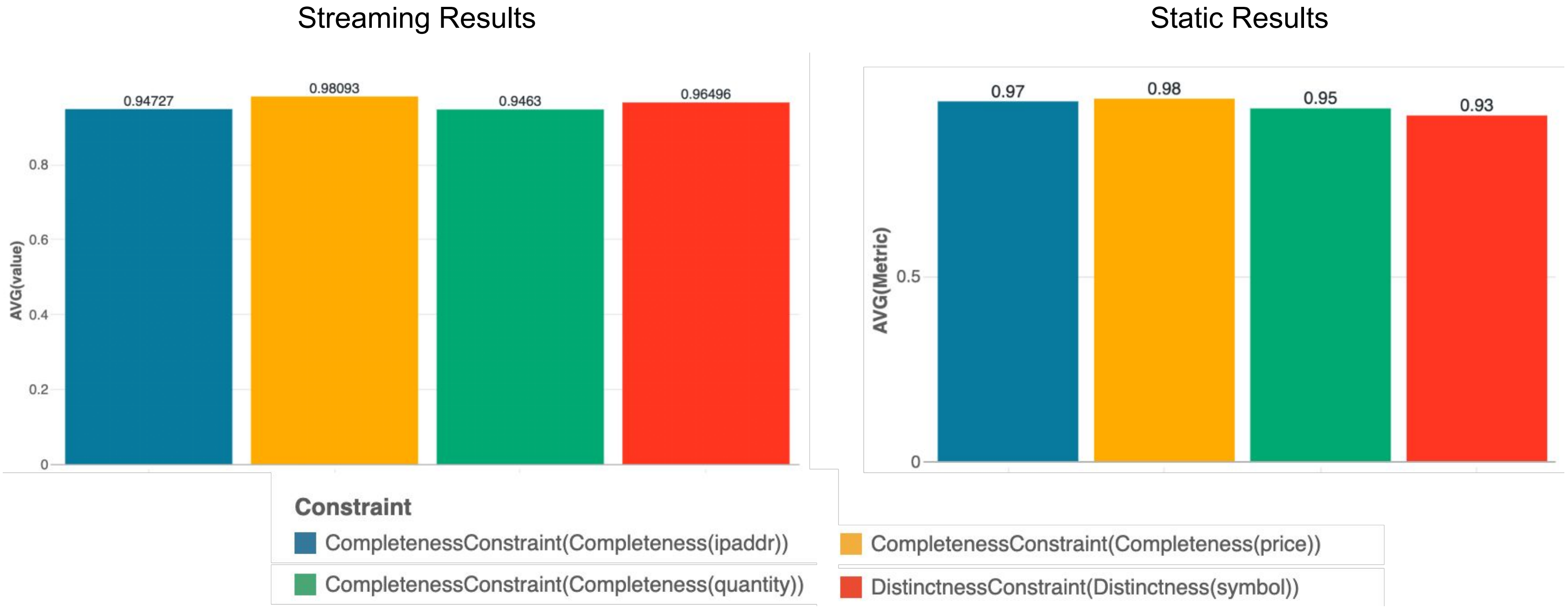
Historical Distinctness




Historical Completeness



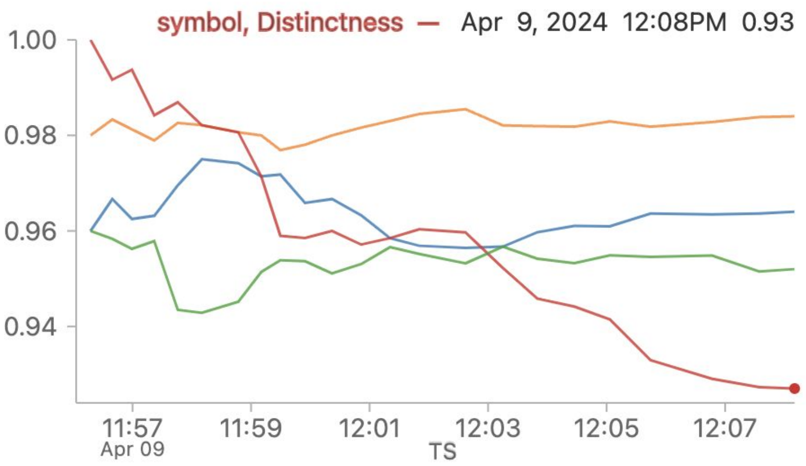
# Evaluation 1: Streaming vs. Static Results



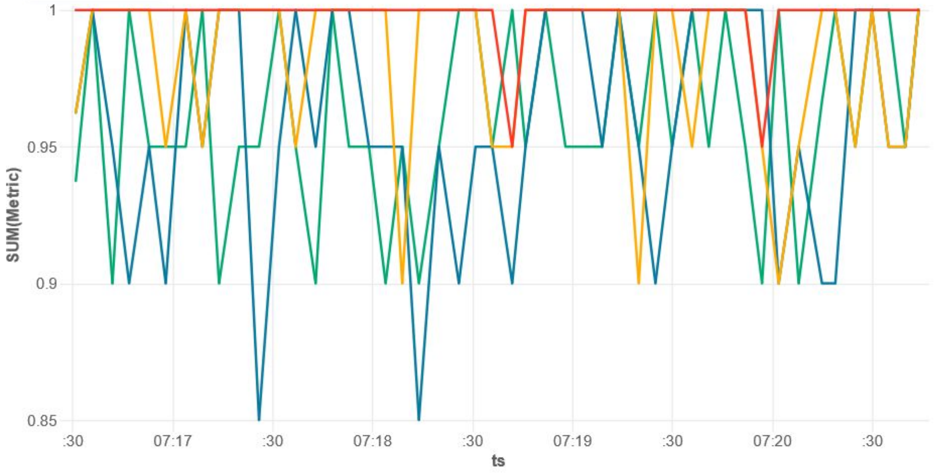
# Evaluation 2: Incremental Computation vs. Batch

amazon | science

**Deequ**




**DuckDQ:**




**Constraint**

- CompletenessConstraint(Completeness(ipaddr))
- CompletenessConstraint(Completeness(price))
- CompletenessConstraint(Completeness(quantity))
- DistinctnessConstraint(Distinctness(symbol))

# Evaluation 3: Performance over Static Data

Table 2. Runtime in Seconds

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

Table 3. Verification Results

| Constraint | 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 |

# KNN Imputation

To apply the findings of Deequ's data verification check:

❖ Experimented with KNN imputation.

❖ KNN is shown to outperform other methods on numeric datasets.

```python
import pandas as pd
from sklearn.impute import KNNImputer
from sklearn.preprocessing import OneHotEncoder

numeric_columns = ['price', 'quantity']
numeric_df = df[numeric_columns]

# Perform KNN Imputation
imputer = KNNImputer(n_neighbors=5)
numeric_imputed = imputer.fit_transform(numeric_df)
```

**Dataset with missing values:**

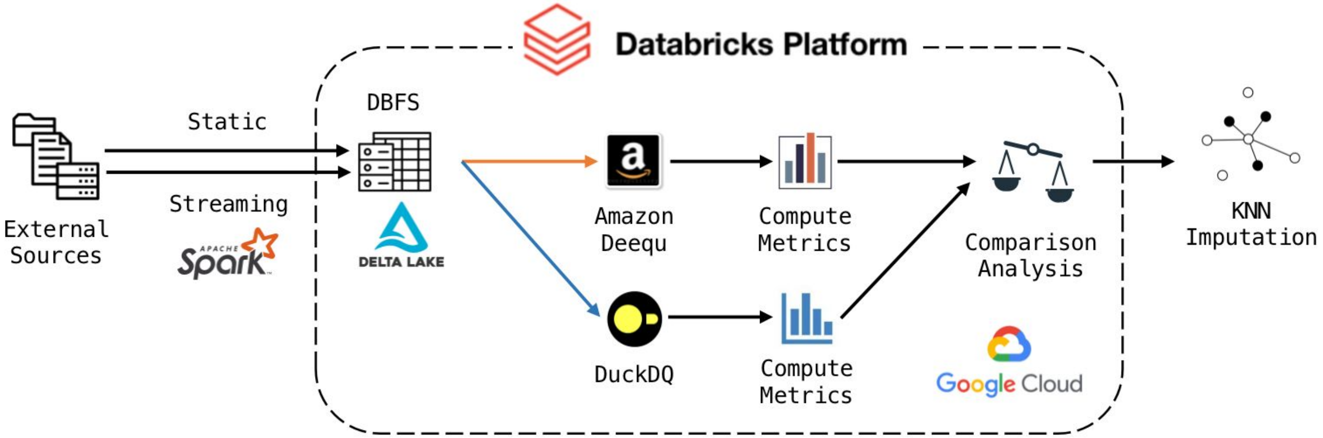
| | price | quantity |
|---|---|---|
| 0 | NaN | 3338.6100 |
| 53 | NaN | 1667.2239 |
| 66 | 16.2680 | NaN |
| 68 | NaN | 588.3304 |
| 81 | 23.7744 | NaN |
| ... | ... | ... |
| 909 | 21.8370 | NaN |
| 920 | 21.3060 | NaN |
| 923 | 27.3420 | NaN |
| 926 | 19.5546 | NaN |
| 940 | 18.4585 | NaN |

**After imputation:**

| | 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 |

# Conclusion

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

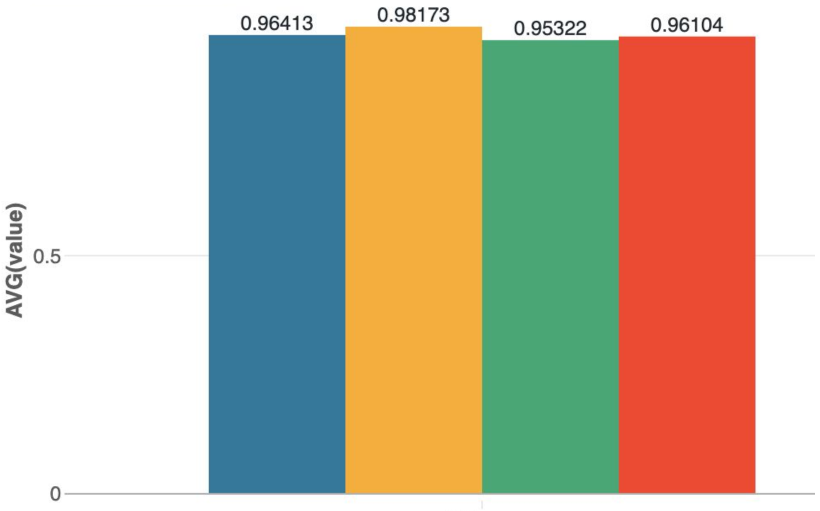


# Thank you!

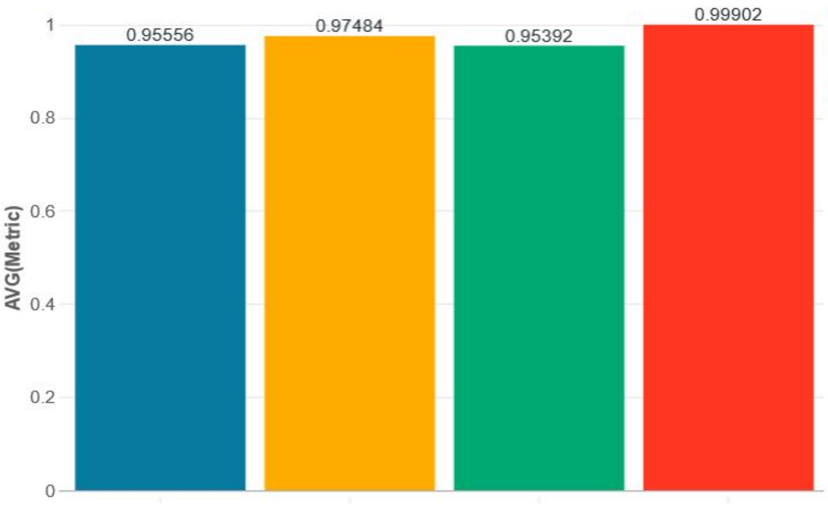
**Q & A**

# *Appendix - 1 Streaming Results*

**Deequ:**




**DuckDQ:**

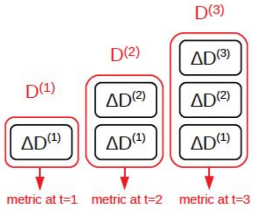



**Constraint**

- CompletenessConstraint(Completeness(ipaddr))
- CompletenessConstraint(Completeness(price))
- CompletenessConstraint(Completeness(quantity))
- DistinctnessConstraint(Distinctness(symbol))

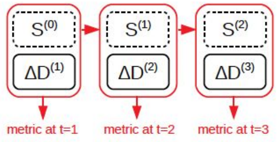
# Appendix - 2 Incremental Computation of Metrics for Growing Datasets

Instead of repeatedly running the batch computation on growing input data D, running an incremental computation that only needs to consume the latest dataset delta ΔD(t) and a state S of the computation.



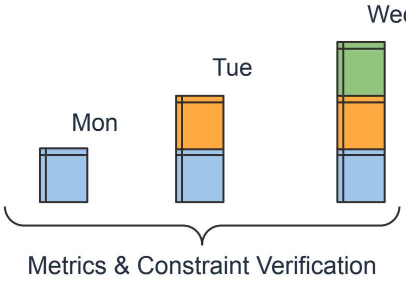

**batch metrics computation**

$$S^{(t)} = f(\Delta D^{(t)}, S^{(t-1)})$$

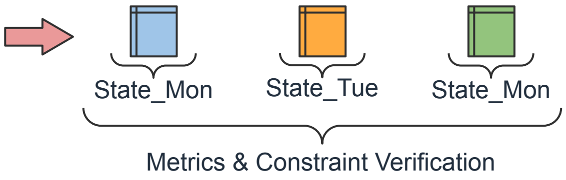



**incremental metrics computation**

***Example:*** logs with daily partitions




**Incremental:** Global constraint evaluation combines partition states; Computational load proportional to partition data size.




**Reformulate our quality metrics:**

- *Completeness:*

$$\frac{|\{v \in V \cup \Delta V \mid c_v + \Delta c_v = 1\}|}{|V \cup \Delta V|}$$

- *MutualInformation:*

$$\sum_{v_1}\sum_{v_2} \frac{c_{v_1 v_2} + \Delta c_{v_1 v_2}}{N + \Delta N} \log \frac{c_{v_1 v_2} + \Delta c_{v_1 v_2}}{(c_{v_1} + \Delta c_{v_1})\,(c_{v_2} + \Delta c_{v_2})}$$

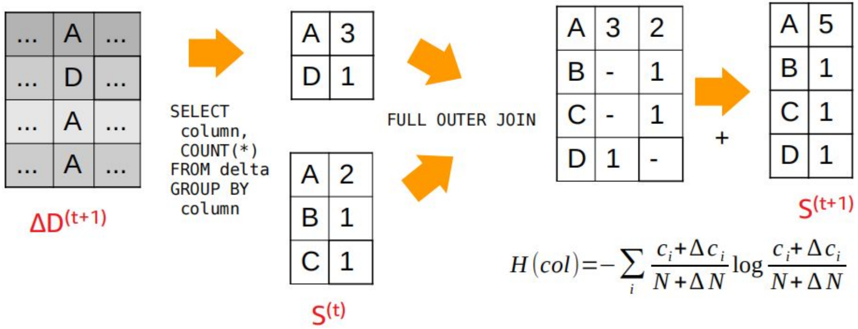
```
val completeness = Completeness("origin")

// Compute state of the changed partition
val newStateToday =
  completeness.computeStateFrom(newPartitionToday)

// Load states of non-changed partitions
val (stateSunday, stateMonday) = loadPreviousStates("…")

// Sum of the states of the individual partitions
val newTableState = stateSunday + stateMonday + newStateToday

// Compute the completeness of 'origin' in the whole table fro
state
val newTableCompleteness = completeness.computeMetricFrom(newT
```

# Appendix - 3 Example for an incremental update of the entropy of a column

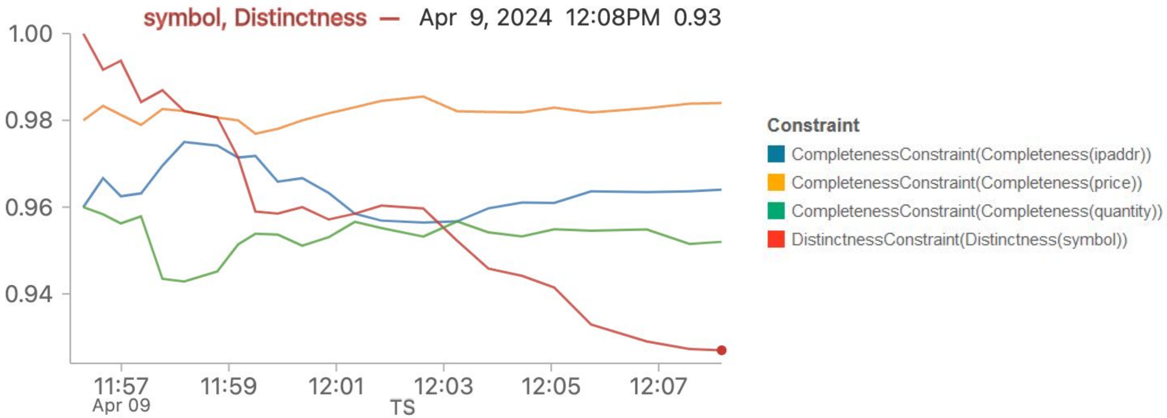
### 4.1 Incremental Computation

In the following, we detail how to make our system's analyzers 'state-aware' to enable them to conduct incremental computations. A corresponding base class in Scala is shown in Listing 3, where `M` denotes the type of metric to compute and `S` denotes the type of state required. Persistence and retrieval of the state are handled outside of the implementation by a so-called `StateProvider`. The method **`initialState`** produces an initial empty state, **`apply`** produces a state and the corresponding metric for an initial dataset, and **`update`** consumes the current state and a delta dataset, and produces the updated state, as well as the corresponding metrics, both for the dataset as a whole and for the delta, in the form of a tuple `(S, M, M)`. Furthermore, the method `applyOrUpdateFromPersistedState` executes the incremental computation and takes care of managing the involved states using `StateProviders`.

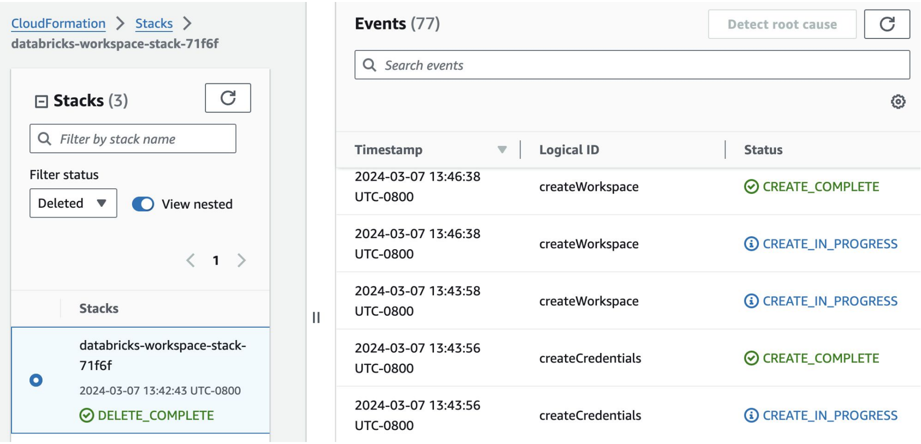
```scala
trait IncrementalAnalyzer[M, S]
  extends Analyzer[M] {

 def initialState(initialData: DataFrame): S

 def update(
   state: S,
   delta: DataFrame): (S, M, M)

 def updateFromPersistedState(
   stateProvider: Option[StateProvider],
   nextStateProvider: StateProvider,
   delta: DataFrame): (M, M)
}

trait StateProvider {

 def persistState[S](
   state: S,
   analyzer: IncrementalAnalyzer[M, S])

 def loadState[S](
   analyzer: IncrementalAnalyzer[M, S]): S
}
```

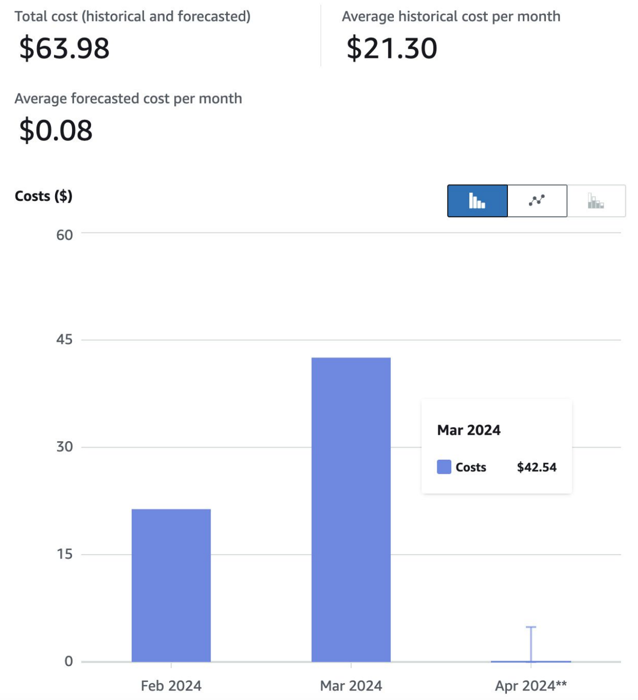


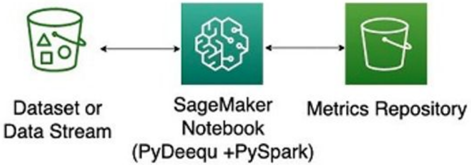

$$H(col) = -\sum_i \frac{c_i + \Delta c_i}{N + \Delta N} \log \frac{c_i + \Delta c_i}{N + \Delta N}$$



# Appendix - 2 Budget Control (AWS sucks)



# Appendix - 4 Delta Lake

1. An optimized storage layer that provides the foundation for tables on Databricks.

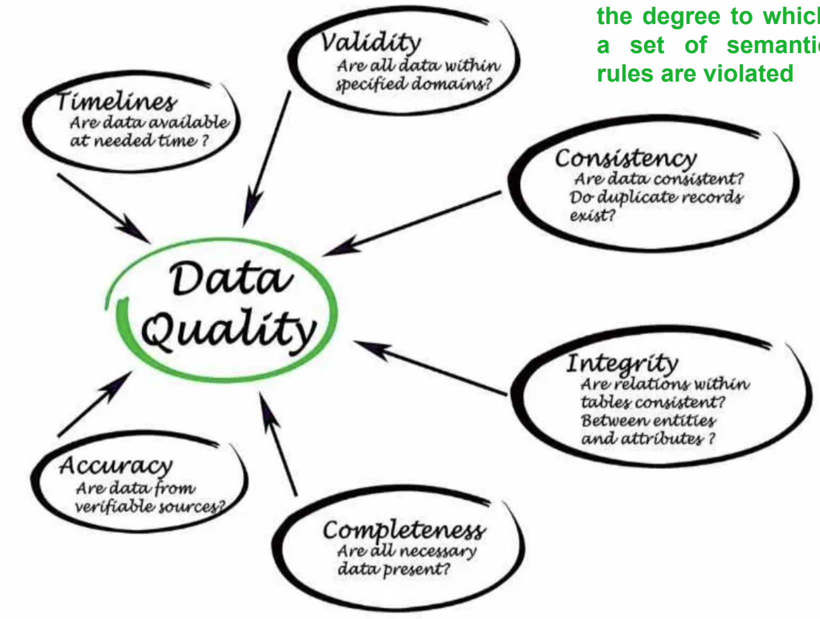
2. It extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling.

3. Delta Lake is fully compatible with Apache Spark APIs, and was 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.

# Appendix - 5 What Is the Common Dimensions of Data Quality?

Challenges and derived requirements for various data domain:

| Data Domain | Data Challenges | Derived Requirements |
|---|---|---|
| Company data | Duplicates<br>Irrelevant data<br>Missing reference data | Consistent Representation<br>Relevance<br>Value-Added |
| People data | Incomplete data<br>Outdated data<br>Root cause analysis<br>Tracking issues | Accuracy<br>Value-Added<br>Timeliness<br>Relevance |
| Service/Asset data | Missing data<br>Incomplete data<br>Incorrect values<br>Duplicates | Accessibility<br>Value-added<br>Completeness<br>Interpretability |
| Supply Chain data | Design errors<br>Duplicates<br>Master data issues | Completeness<br>Timeliness<br>Accuracy |




the degree to which a set of semantic rules are violated

how much data is needed to describe a real-world object

[2] Silvola, Risto et al. “Data quality assessment and improvement.” Int. J. Bus. Inf. Syst. 22 (2016): 62-81.