Unit Testing Data with Deequ

Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat,
Philipp Schmidt, Stephan Seufert, Pierre Brunelle, Andrey Taptunov
Amazon Research
{sseb,biessman,langed,tammruka,phschmid,seufert,brunep,taptunov}@amazon.com

ABSTRACT

Modern companies and institutions rely on data to guide every single decision. Missing or incorrect information seriously compromises any decision process. We demonstrate *Deequ*, an Apache Spark-based library for automating the verification of data quality at scale. This library provides a declarative API, which combines common quality constraints with user-defined validation code, and thereby enables *unit tests for data*. *Deequ* is available as open source, meets the requirements of production use cases at Amazon, and scales to datasets with billions of records if the constraints to evaluate are chosen carefully.

Our demonstration walks attendees through a fictitious business use case of validating daily product reviews from a public dataset, and is executed in a proprietary interactive notebook environment. We show attendees how to define data unit tests from automatically suggested constraints and how to create customized tests. Additionally, we demonstrate how to apply *Deequ* to validate incrementally growing datasets, and give examples of how to configure anomaly detection algorithms on time series of data quality metrics to further automate the data validation.

ACM Reference Format:
Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat,, Philipp Schmidt, Stephan Seufert, Pierre Brunelle, Andrey Taptunov. 2019. Unit Testing Data with Deequ. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320210

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320210

1 INTRODUCTION

Data is at the center of modern enterprises and institutions. Online retailers, for example, rely on data to support customers making buying decisions, to forecast demand [2], to schedule deliveries. Missing or incorrect information seriously compromises any decision process downstream, ultimately damaging the overall effectiveness and efficiency of the organization. The quality of data has effects across teams and organizational boundaries. Furthermore, there is a trend across different industries towards more automation of business processes with machine learning (ML) techniques, which also introduces novel data validation problems [4, 5, 8]. In modern information infrastructures, data lives in many different places (e.g., in relational databases, in 'data lakes' on distributed file systems, behind REST APIs, etc.), and comes in many different formats. Many such data sources do not support integrity contraints, and often there is not even an accompanying schema available. Due to these circumstances, every team and system involved in data processing has to take care of data validation in some way, which often results in tedious and repetitive work.

In order to address these challenges, we recently proposed *Deequ*[1], an open-source library for automating the verification of data quality at scale [6] with Apache Spark. *Deequ* provides a declarative API, which combines common quality constraints with user-defined validation code, and thereby enables *unit tests for data.* It allows users to explicitly and declaratively state their expectations about the data which they consume or produce. Furthermore, *Deequ* allows users to automate the data validation process by integrating the tests into data pipelines. In case of violations, data can be quarantined and data engineers can be automatically notified. We demonstrate *Deequ* using an interactive notebook environment, and walk attendees through a retail use case of validating daily product reviews. We focus on the following tasks:

- Definition of customized data unit tests from automatically suggested constraints.
- Data validation on incrementally growing datasets.
- Configuration of anomaly detection algorithms on time series of data quality metrics.

[1]https://github.com/awslabs/deequ

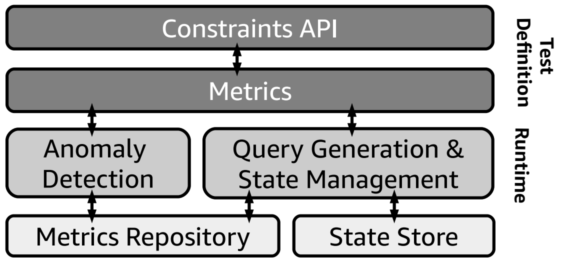


Figure 1: Architecture of *Deequ*.

2 OVERVIEW OF DEEQU

We give a brief overview over *Deequ*. Interested readers can find more details in our recent publications [6, 7] and on the project homepage at https://github.com/awslabs/deequ.

Execution of data validating tests. *Deequ*'s architecture is shown in Figure 1. Given a data unit test specified with its declarative API, *Deequ* inspects the constraints to validate, and identifies the data metrics required for evaluation. Next, it generates queries in SparkSQL [1] with custom designed aggregation functions in order to compute the statistics. For performance reasons, it applies multi-query optimization to enable scan-sharing for the aggregation queries in order to reduce the number of required passes over the input data. Once the data statistics are computed, *Deequ* invokes the user-defined validation functions contained in the test code and returns the evaluation results to the user. Additionally, it supports efficient constraint evaluation on changing data (e.g., incrementally growing logs) with a specialized implementation of incremental view maintenance for data quality metrics. This implementation stores mergeable sufficient statistics per data partition which enable the cheap computation of quality metrics from different combinations of partitions.

Constraint suggestion and anomaly detection. *Deequ* contains several advanced features which we will also showcase during the demonstration. The benefits of our library to users heavily depend on the richness and specificity of the constraints, which the users define. Therefore, we aim to make the adoption process as simple as possible and provide machinery to automatically suggest constraints and identify data types for datasets. *Deequ* applies scalable single-column profiling [3], and feeds the resulting profiles into rule-base constraint generators, which for example inspect the ratio of missing values or estimates of the cardinality of a column. In some cases, it might be difficult to define exact thresholds for constraints on certain data statistics, as these statistics might be subject to regular changes, e.g., due to weekly seasonality patterns. *Deequ* therefore allows its users to record a time series of data quality metrics and configure anomaly detection algorithms that supply sensible thresholds learnt from past data quality metrics for newly arriving data.

3 DEMO SCENARIO

We describe the scenario and setup of the demonstration, and give an overview over the individual steps through which we will walk the attendees.

Product reviews use case. We exemplify *Deequ* on a fictitious use case from the retail domain, for which we leverage a sample of the *Amazon Reviews Dataset*[2], a publicly available dataset of product reviews. We assume that our task is to ingest a batch of new product reviews every day. As part of this ingestion, we need to validate the data quality of the new batch, to make sure it can safely be consumed by downstream systems.

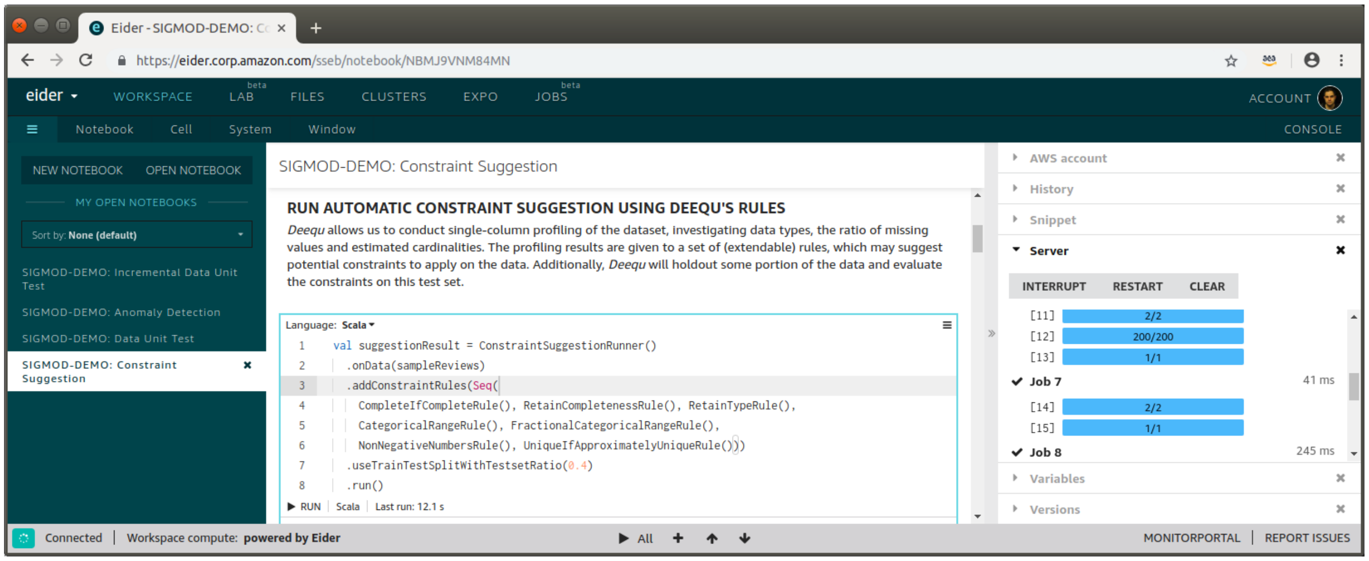
Setup. We use Apache Spark for our data pipeline and ingest the review data in the form of CSV files located in the distributed file system S3. We execute the demonstration in a custom proprietary notebook environment (shown in Figure 2), which allows us to run Scala-based Spark jobs as well as Python-based data analysis on data residing in S3. We will walk attendees through various example notebooks that implement different stages of our use case. The notebook environment allows us to interactively execute the code in front of the attendees and incorporate their feedback in the form of code changes.

Automated constraint suggestion. We assume that our input data comes in the form of CSV files without machine-readable schema information other than column names. In such cases, our tests should validate the consistency of data types, value ranges and absence of missing values in the data. *Deequ* aims to automate the definition of such simple constraints as much as possible. For that, we load a sample of the data (e.g., the reviews from a particular day) and ask our library to generate suggestions for constraints using a set of predefined rules. A rule might for example compare the number of rows in the sample to a sketch-based estimate of the cardinality of a column, and suggest a unique constraint if these are close. *Deequ* additionally allows us to hold out a random portion of the data, on which it will evaluate the suggested constraints. *Deequ* outputs suggestions for all the columns of the dataset, both in human-readable form and in the form of Scala code that can be copied and pasted. Additionally, it provides the information whether the suggested constraint was satisfied on the held-out data, e.g.:

```
'review_headline' has less than 1% missing values
(satisfied on holdout set)
Code: .hasCompleteness("review_headline", _ >= 0.99)

'product_id' is unique
(failed on holdout set, uniqueness was 0.918)
Code: .isUnique("product_id")
```

[2]https://s3.amazonaws.com/amazon-reviews-pds/readme.html



SIGMOD-DEMO: Constraint Suggestion

RUN AUTOMATIC CONSTRAINT SUGGESTION USING DEEQU'S RULES

Deequ allows us to conduct single-column profiling of the dataset, investigating data types, the ratio of missing values and estimated cardinalities. The profiling results are given to a set of (extendable) rules, which may suggest potential constraints to apply on the data. Additionally, *Deequ* will holdout some portion of the data and evaluate the constraints on this test set.

Language: Scala

```
val suggestionResult = ConstraintSuggestionRunner()
  .onData(sampleReviews)
  .addConstraintRules(Seq(
    CompleteIfCompleteRule(), RetainCompletenessRule(), RetainTypeRule(),
    CategoricalRangeRule(), FractionalCategoricalRangeRule(),
    NonNegativeNumbersRule(), UniqueIfApproximatelyUniqueRule()))
  .useTrainTestSplitWithTestsetRatio(0.4)
  .run()
```

RUN | Scala | Last run: 12.1 s

```
[11] 2/2
[12] 200/200
[13] 1/1
✓ Job 7                41 ms
[14] 2/2
[15] 1/1
✓ Job 8               245 ms
```

Figure 2: The demo will be executed in an interactive notebook environment using Apache Spark, Python data visualization tools and a sample of the public Amazon reviews dataset.

Definition of the data unit test. Based on the outputs from the constraint suggestion, we can now start to define our data unit test to run on the review data. The constraint suggestions are often leveraged for a data exploration, and users later decide whether to adopt them or not (usually the information whether the constraints held on the test data or not is crucial here). This typically results in a set of simple integrity constraints that concentrate on completeness, data types and value ranges. In our case, we would for example require that the column `customer_id` is never null and contains integer values only, that the column `product_category` only contains strings from a predefined set of values and that at least 99% of values in the `vine` column have a particular value:

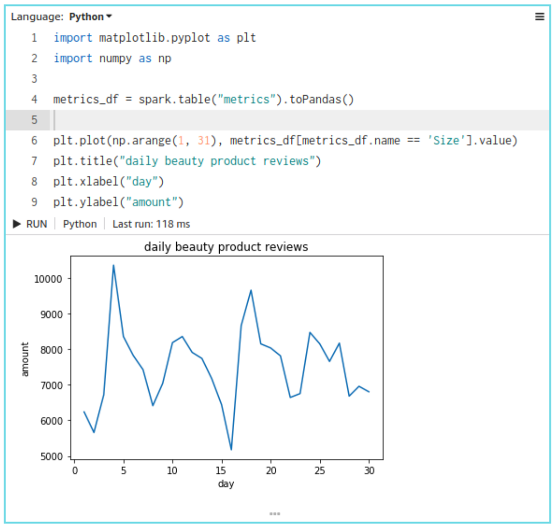
```
Check(Level.Error)
  .isComplete("customer_id")
  .hasDataType("customer_id", Integral)
  .isComplete("product_category")
  .isContainedIn("product_category",
    Array("Beauty", "Shoes", "Jewelry"))
  .isContainedIn("vine", Array("N"), _ >= 0.99)
  .isNonNegative("total_votes")
...
```

We designed the constraint suggestion rules in a conservative way with the goal to produce a low number of false positives (in order to reduce false alarms if users decide to only apply the automatically suggested constraints). Therefore, users with a deeper understanding of the data typically start with the suggested constraints and add custom handcrafted constraints. Typically these constraints define thresholds on aggregates over the data, and often leverage sketch data structures that can be computed quickly at scale. In our example, we define some advanced constraints, e.g., we require that there are no credit card numbers present in reviews, and we define the expected range of the (approximate) median and mean of the `star_rating` column:

```
Check(Level.Error)
 .containsCreditCardNumber("review_body", NEVER)
 .hasApproxQuantile("star_rating", 0.5, _ >= 4.0)
 .hasMean("star_rating", { meanRating =>
   meanRating > 3.5 && meanRating < 5.0 })
```

Finally, we show custom validation logic that is enabled by our decision to allow for arbitrary user-defined validation code. We assume that the reviews are a denormalized copy from another data source (which is often the case in complex industry data pipelines), and that we can ask this data source about “ground truth” properties of the data. We imagine that there is an external `ProductService` (e.g., invokable via a REST API) which allows us to retrieve the number of distinct available products in a certain product category on a given day. Due to the flexibility of *Deequ*, we can write code that contacts this service during the validation and verifies that the statistics observed in the data (e.g., the approximate cardinality of the `product_id` column) are consistent with the ground truth returned by the service.

```
Check(Level.Error)
 .hasApproxCountDistinct("product_id", { count =>
   val expected = ProductService
    .numAvailableProducts(day, "Beauty")
   count <= 1.15 * expected })
 .where("product_category = 'Beauty'")
```



```
import matplotlib.pyplot as plt
import numpy as np

metrics_df = spark.table("metrics").toPandas()

plt.plot(np.arange(1, 31), metrics_df[metrics_df.name == 'Size'].value)
plt.title("daily beauty product reviews")
plt.xlabel("day")
plt.ylabel("amount")
```

Figure 3: Visualization of a time series of data quality metrics exhibiting weekly seasonality.

After having defined the data unit tests, we will execute them on different parts of the reviews dataset and show to attendees how to inspect the validation results in the form of Spark dataframes that summarize data metrics and constraint results.

Anomaly detection on data quality metrics. Most of the constraints presented so far comprise of thresholds on statistics of the data. There might be cases however, where it is difficult to define such thresholds because the distribution of certain statistics is not stationary, but subject to seasonality patterns. Figure 3 highlights such a case, when we compute the time series of the number of reviews for beauty products in our data, which exhibits a weekly seasonality pattern. We demonstrate how to handle such cases with *Deequ*. Users can record the time series of data statistics and leverage several predefined anomaly detection algorithms [9] to decide whether the metric observed in a new batch is to be considered anomalous or not.

```
Check(Level.Warning)
  .usingRepository(repository)
  .isNonAnomalous(Size(),
    algorithm = HoltWinters(seasonality = Weekly))
  .isNonAnomalous(Mean("total_votes"),
    algorithm = OnlineNormal())
```

We showcase this technique by training an anomaly detection model and using it to catch manually introduced errors in new data.

Handling growing datasets. In the final part of the demo, we show how to configure *Deequ* to compute and validate data metrics repeatedly on growing datasets. In our example, the reviews dataset grows by daily batches and we do not want to have to re-scan all the data each day for evaluating constraints on the dataset as a whole. In that case, our library provides a simple abstraction that allows users to record sufficient statistics for data partitions that can be merged with statistics ofnewly ingested data to avoid having to re-read historic data during the re-execution of tests.

4 INTERACTIVITY

Our demonstration will be executed in a notebook environment, which allows for a high amount of interaction with attendees, as they will be able to suggest changes to the code which we can evaluate "on the fly". Attendees can for example ask us to try different rules for constraint suggestion, they can propose custom constraints to evaluate on the data, and they can suggest different errors to introduce in the data and find out whether the data unit tests catch these errors.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. *SIGMOD* (2015), 1383–1394.

[2] Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, and Yuyang Wang. 2017. Probabilistic demand forecasting at scale. *PVLDB* 10, 12 (2017), 1694–1705.

[3] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe* (2008).

[4] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. *SIGMOD* (2017), 1723–1726.

[5] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, Gyuri Szarvas, et al. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Engineering Bulletin* (2018).

[6] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018).

[7] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2019. Differential Data Quality Verification on Partitioned Data. *ICDE* (2019).

[8] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *NeurIPS* (2015), 2503–2511.

[9] Peter R Winters. 1960. Forecasting sales by exponentially weighted moving averages. *Management science* 6, 3 (1960), 324–342.