Automating Large-Scale Data Quality Verification

Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann
Amazon Research
{sseb,langed,phschmid,celikelm,biessmann}@amazon.com

Andreas Grafberger*
University of Augsburg
andreas.grafberger@student.uni-augsburg.de

ABSTRACT

Modern companies and institutions rely on data to guide every single business process and decision. Missing or incorrect information seriously compromises any decision process downstream. Therefore, a crucial, but tedious task for everyone involved in data processing is to verify the quality of their data. We present a system for automating the verification of data quality at scale, which meets the requirements of production use cases. Our system provides a declarative API, which combines common quality constraints with user-defined validation code, and thereby enables 'unit tests' for data. We efficiently execute the resulting constraint validation workload by translating it to aggregation queries on Apache Spark. Our platform supports the incremental validation of data quality on growing datasets, and leverages machine learning, e.g., for enhancing constraint suggestions, for estimating the 'predictability' of a column, and for detecting anomalies in historic data quality time series. We discuss our design decisions, describe the resulting system architecture, and present an experimental evaluation on various datasets.

PVLDB Reference Format:
Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann and Andreas Grafberger. Automating Large-Scale Data Quality Verification. *PVLDB*, 11 (12): 1781-1794, 2018.
DOI: https://doi.org/10.14778/3229863.3229867

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 [7], to schedule deliveries, and more generally, to guide every single business process and decision. 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, especially in large organizations with complex systems that result in complex data dependencies. Furthermore, there is a trend across different industries towards more automation of business processes with machine learning (ML) techniques. These techniques are often highly sensitive on input data, as the deployed models rely on strong assumptions about the shape of their inputs [42], and subtle errors introduced by changes in data can be very hard to detect [34]. At the same time, there is ample evidence that the volume of data available for training is often a decisive factor for a model's performance [17, 44]. 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, or is constantly being scraped from web resources), and comes in many different formats. Many such data sources do not support integrity contraints and data quality checks, and often there is not even an accompanying schema available, as the data is consumed in a 'schema-on-read' manner, where a particular application takes care of the interpretation. Additionally, there is a growing demand for applications consuming semi-structured data such as text, videos and images.

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. As a concrete example, imagine an on-demand video platform, where the machines that stream videos to users write log files about the platform usage. These log files must regularly be ingested into a central data store to make the data available for further analysis, e.g., as training data for recommender systems. Analogous to all data produced in real-world scenarios, these log files might have data quality issues, e.g., due to bugs in program code or changes in the semantics of attributes. Such issues potentially result in failures of the ingestion process. Even if the ingestion process still works, the errors in the data might cause unexpected errors in downstream systems that consume the data. As a consequence, the team managing the daily ingestion process must validate the data quality of the log files, e.g., by checking for missing values, duplicate records, or anomalies in size.

We therefore postulate that there is a pressing need for increased *automation of data validation*. We present a system that we built for this task and that meets the demands of production use cases. The system is built on the following principles: at the heart of our system is *declarativity*; we want users to spend time on thinking 'how' their data should look like, and not have to worry too much about how to im-

*work done while at Amazon Research

Proceedings of the VLDB Endowment, Vol. 11, No. 12

DOI: https://doi.org/10.14778/3229863.3229867

plement the actual quality checks. Therefore, our system offers a declarative API that allows users to define checks on their data by composing a huge variety of available constraints. Additionally, data validation tools should provide high *flexibility* to their users. The users should be able to leverage external data and custom code for validation (e.g., call a REST service for some data and write a complex function that compares the result to some statistic computed on the data). Our vision is that users should be able to write 'unit-tests' for data (Section 3.1), analogous to established testing practices in software engineering. Furthermore, data validation systems have to acknowledge the fact that *data is being continuously produced*, therefore they should allow for the integration of growing datasets. Our proposed system explicitly supports the incremental computation of quality metrics on growing datasets (Section 3.2), and allows its users to run anomaly detection algorithms on the resulting historical time series of quality metrics (Section 3.4). The last principle to address is *scalability*: data validation systems should seamlessly scale to large datasets. To address this requirement, we designed our system to translate the required data metrics computations to aggregation queries, which can be efficiently executed at scale with a distributed dataflow engine such as *Apache Spark* [50].

The contributions of this paper are the following:

- We present a declarative API combining common quality constraints with user-defined validation code, which enables 'unit tests' for data (Section 3.1).
- We discuss how to efficiently execute the constraint validation by translating checks to aggregation queries with dedicated support for incremental computation on growing datasets (Sections 3.2 and 4).
- We give examples of how machine learning can be leveraged in data quality verification, e.g., for enhancing constraint suggestions, for estimating the predictability of a column, and for detecting anomalies in historic data quality timeseries (Sections 3.3 and 3.4).
- We present an experimental evaluation of our proposed approaches (Section 5).

2. DATA QUALITY DIMENSIONS

We first take a look at data quality literature to understand common dimensions of quality. The quality of data can refer to the *extension* of the data (i.e., data values), or to the *intension* of the data (i.e., the schema) [4]. We focus on extensional data quality in the following. We treat schema problems with the concept of completeness; for example, if there is no attribute value specified in the open schema of an entity, we consider it missing. There are multiple studies on measuring the extensional data quality [4, 30, 39]. In the following, we briefly describe the most commonly referred to dimensions.

Completeness refers to the degree to which an entity includes data required to describe a real-world object. In tables in relational database systems, completeness can be measured by the presence of null values, which is usually interpreted as a missing value. In other contexts, for instance in a product catalog, it is important to calculate completeness given the correct context, i.e., the schema of the product category. For example, the absence of a value for the attribute `shoe_size` is not relevant for products in the category *notebooks*. In this case, the attribute value is missing because the attribute is not applicable [37]. We are only interested in measuring completeness when an attribute is applicable.

Consistency is defined as the degree to which a set of semantic rules are violated. *Intra-relation constraints* define a range of admissible values, such as a specific data type, an interval for a numerical column, or a set of values for a categorical column. They can also involve rules over multiple columns, e.g., 'if `category` is *t-shirts*, then the range of `size` is {S, M, L}'. *Inter-relation constraints* may involve columns from multiple tables. For example, a column `customerId` may only include values from a given reference table of all customers.

Accuracy is the correctness of the data and can be measured in two dimensions: syntactic and semantic. Syntactic accuracy compares the representation of a value with a corresponding definition domain, whereas semantic accuracy compares a value with its real-world representation. For example, for a product attribute `color`, a value *blue* can be considered syntactically accurate in the given domain even if the correct value would be *red*, whereas a value *XL* would be considered neither semantically nor syntactically accurate.

3. APPROACH

We introduce our general machinery for automated large-scale data quality verification. We first present our declarative API, which allows users to specify constraints on their datasets, and detail how we translate these constraints into computations of metrics on the data, which allow us to subsequently evaluate the constraints (Section 3.1). Next, we discuss how to extend this approach to scenarios where we need to evaluate such constraints for incrementally growing datasets (e.g., in the case of ingestion pipelines in a data warehouse) in Section 3.2. Lastly, we describe extensions of our approach such as the suggestion of constraints (Section 3.3) and anomaly detection on historical data quality time series of a dataset (Section 3.4).

3.1 'Unit Tests' for Data

The general idea behind our system is to enable users to easily define 'unit tests' for their datasets in a declarative manner [10]. These 'unit-tests' consist of constraints on the data which can be combined with user-defined functions, e.g., custom code. Table 1 shows the constraints available to our users. We want them to focus on the definition of their checks and their validation code, but not on the computation of the metrics required for the constraints. Therefore, we design our system to translate the user-defined constraints into an efficiently executable metrics computation.

Declarative definition of data quality constraints. In our system, users define checks for their datasets, which either result in errors or warnings during execution, if the validation fails. This approach provides a high flexibility for users: they can write complex functions and leverage existing libraries for their validation code; they can use external data or even can call external services. In order to showcase our system, we introduce an exemplary use case on an on-demand video platform. Assume that the machines that stream videos to users write log files about the platform usage, with details such as the type of device used, the length of the session, the customer id or the location.

Table 1: Constraints available for composing user-defined data quality checks.

constraint	arguments	semantic
dimension *completeness*		
`isComplete`	column	check that there are no missing values in a column
`hasCompleteness`	column, udf	custom validation of the fraction of missing values in a column
dimension *consistency*		
`isUnique`	column	check that there are no duplicates in a column
`hasUniqueness`	column, udf	custom validation of the unique value ratio in a column
`hasDistinctness`	column, udf	custom validation of the unique row ratio in a column
`isInRange`	column, value range	validation of the fraction of values that are in a valid range
`hasConsistentType`	column	validation of the largest fraction of values that have the same type
`isNonNegative`	column	validation whether all values in a numeric column are non-negative
`isLessThan`	column pair	validation whether values in the 1s column are always less than in the 2nd column
`satisfies`	predicate	validation whether all rows match predicate
`satisfiesIf`	predicate pair	validation whether all rows matching 1st predicate also match 2nd predicate
`hasPredictability`	column, column(s), udf	user-defined validation of the predictability of a column
statistics (can be used to verify dimension *consistency*)		
`hasSize`	udf	custom validation of the number of records
`hasTypeConsistency`	column, udf	custom validation of the maximum fraction of values of the same data type
`hasCountDistinct`	column	custom validation of the number of distinct non-null values in a column
`hasApproxCountDistinct`	column, udf	custom validation of the approx. number of distinct non-null values
`hasMin`	column, udf	custom validation of a column's minimum value
`hasMax`	column, udf	custom validation of a column's maximum value
`hasMean`	column, udf	custom validation of a column's mean value
`hasStandardDeviation`	column, udf	custom validation of a column's standard deviation
`hasApproxQuantile`	column, quantile, udf	custom validation of a particular quantile of a column (approx.)
`hasEntropy`	column, udf	custom validation of a column's entropy
`hasMutualInformation`	column pair, udf	custom validation of a column pair's mutual information
`hasHistogramValues`	column, udf	custom validation of column histogram
`hasCorrelation`	column pair, udf	custom validation of a column pair's correlation
time		
`hasNoAnomalies`	metric, detector	validation of anomalies in time series of metric values

```
1  val numTitles = callRestService(...)
2  val maxExpectedPhoneRatio = computeRatio(...)
3
4  var checks = Array()
5
6  checks += Check(Level.Error)
7   .isComplete("customerId", "title",
8    "impressionStart", "impressionEnd",
9    "deviceType", "priority")
10  .isUnique("customerId", "countryResidence",
11   "deviceType", "title")
12  .hasCountDistinct("title", _ <= numTitles)
13  .hasHistogramValues("deviceType",
14    _.ratio("phone") <= maxExpectedPhoneRatio)
15
16 checks += Check(Level.Error)
17  .isNonNegative("count")
18  .isLessThan("impressionStart", "impressionEnd")
19  .isInRange("priority", ("hi", "lo"))
20
21 checks += Check(Level.Warning, on="delta")
22  .hasNoAnomalies(Size, OnlineNormal(stdDevs=3))
23 checks += Check(Level.Error, on="delta")
24  .hasNoAnomalies(Size, OnlineNormal(stdDevs=4))
25
26 checks += Check(Level.Warning)
27  .hasPredictability("countryResidence",
28    ("zipCode", "cityResidence"), precision=0.99)
29
30 Verification.run(data, checks)
```

Listing 1: Example for declarative data quality constraint definitions using our API.

These log files must regularly be ingested into a central data store to make the data available for further analysis, e.g., as training data for recommender systems. Analogous to all data produced in real-world scenarios, these log files might have data quality issues, e.g., due to bugs in program code, data loss, redeployments of services, or changes in semantics of data columns. Such issues might potentially result in several negative consequences, e.g., the ingestion process might fail and need to be manually restarted after communication with the data provider. Even if the ingestion process still works, the errors in the data might cause unexpected errors in downstream systems that consume the data. In many cases these errors might be hard to detect, e.g., they might cause regressions in the prediction quality of a machine learning model, which makes assumptions about the shape of particular features computed from the input data [34]. Therefore, the video streaming service could use our system to validate the data quality before starting the data ingestion process, by declaring a custom set of checks that should hold on the data. Listing 1 depicts a toy example of how such a declarative quality check for video stream logs could look like and highlights the combination of declarative constraint definitions with custom code. External data is fetched in the beginning and used throughout the quality check: an external REST service is called to determine the overall number of movies in the system and the expected ratio of smartphone watchers is computed (see lines 1 & 2). Then, a set of completeness and consistency checks is defined, e.g., we require the columns `customerId`, `title`, `impressionStart`,

Table 2: Computable metrics to base constraints on.

metric	semantic
dimension *completeness*	
`Completeness`	fraction of non-missing values in a column
dimension *consistency*	
`Size`	number of records
`Compliance`	ratio of columns matching predicate
`Uniqueness`	unique value ratio in a column
`Distinctness`	unique row ratio in a column
`ValueRange`	value range verification for a column
`DataType`	data type inference for a column
`Predictability`	predictability of values in a column
statistics (can be used to verify dimension *consistency*)	
`Minimum`	minimal value in a column
`Maximum`	maximal value in a column
`Mean`	mean value in a column
`StandardDeviation`	standard deviation of the value distribution in a column
`CountDistinct`	number of distinct values in a column
`ApproxCountDistinct`	number of distinct values in a column estimated by a hyperloglog sketch [21]
`ApproxQuantile`	approximate quantile of the value in a column [15]
`Correlation`	correlation between two columns
`Entropy`	entropy of the value distribution in a column
`Histogram`	histogram of an optionally binned column
`MutualInformation`	mutual information between two columns

`impressionEnd`, `deviceType` and `priority` to be complete (lines 7 to 9), and we dictate that the column combination `customerId`, `countryResidence`, `deviceType`, and `title` is unique in the data at hand (lines 10 and 11). We make sure that the number of distinct values in the `title` column is less than or equal to the overall number of movies in the system (line 12) and we check that the ratio of 'phone' devices meets our expectations by investigating a histogram of the `deviceType` column in lines 13 and 14. Subsequently, we issue another set of consistency checks that define the expected shape of the data (e.g., no negative values in the `count` column, a happens-before relationship between the viewing timestamps, and a set of valid values for the `priority` column, lines 16 to 19).

Next, we have two checks that rely on comparisons to previously computed metrics on former versions of the dataset (available from a central 'metrics database'): we advise the system to detect anomalies in the time series of sizes of records that have been added to the dataset over time and issue a warning if the is size more than three standard deviations away from the previous mean and throw an error if it is more than four standard deviations away (see lines 21 to 24). Finally, we define a predictability check for the `countryResidence` column which dictates that our system should be able to predict values in this column with a precision of 99% by inspecting the corresponding values in the `zipCode` and `cityResidence` columns.

Translating constraints to metrics computations. In the following, we detail how our system executes the actual data quality verification. The declarative definition of constraints (which are evaluated by the user code) relies on a particular set of data quality metrics that our system computes from the data at hand. The system inspects the checks and their constraints, and collects the metrics required to evaluate the checks. Table 2 lists all data quality metrics supported by our system. We directly address the data quality dimensions *completeness* and *consistency* listed in Section 2. Let D denote the dataset D with N records, on which we operate, and let c_v denote the cardinality of value the v in a particular column of dataset D. Furthermore, let V denote the set of unique values in a particular column of the dataset D. We calculate `Completeness` as the fraction of non-missing values in a column: $|\{d \in D \mid d(col) \neq \texttt{null}\}| / N$. For measuring *consistency*, we provide metrics on the number of unique values, the data types, the data set size, the value ranges, and a general predicate matching metric. The `Size` metric for example refers to the number of records N, while the `Compliance` metric denotes the ratio of records which satisfy a particular predicate: $|\{d \in D \mid p(d)\}| / N$. The metric `Uniqueness` refers to the unique value ratio [19] in a particular column: $|\{v \in V \mid c_v = 1\}| / |V|$, while `Distinctness` corresponds to the unique row ratio $|V| / N$ in the column. In addition, we implement standard summary statistics for numerical columns that can be used for defining additional semantic rules on datasets, such as `Minimum`, `Maximum`, `Mean`, `StandardDeviation`, `Histogram`, and `Entropy`, which we for example compute as $-\sum_v \frac{c_v}{N} \log \frac{c_v}{N}$. We also include standard statistics such as `Correlation` and `MutualInformation` for measuring the amount of association between two columns, where the latter is computed as: $\sum_{v_1} \sum_{v_2} \frac{c_{v_1 v_2}}{N} \log \frac{c_{v_1 v_2}}{c_{v_1} c_{v_2}}$. As some metrics are rather expensive to compute and might involve re-partitioning or sorting the data, our system provides approximations of metrics such as quantiles in the form of `ApproxQuantile` (computed via an efficient online algorithm [15]) or `ApproxCountDistinct` for estimating the number of distinct values with a hyperloglog sketch [21].

Lastly, we offer an implementation of `Predictability`. In an attempt to automate the verification of the correctness of values, we train a machine learning model that predicts a value for a target column t of a particular record from all k observed values $l_1, \ldots, l_k \in V_t$ in the target column, given the corresponding values $l_{i_1}, \ldots, l_{i_n}$ of input columns $i_1, \ldots, i_n$ for the particular record, e.g., using the maximum a posteriori decision rule: $\text{argmax}_k\, p(l_k | l_{i_1}, \ldots, l_{i_n})$. An example would be to predict the value of a 'color' column in a product table from text in the 'description' and 'name' columns. We train this model on a sample of observed values in the target column, and measure its prediction quality on the held-out rest of the data. We return the quality score, calculated using standard measures such as precision, recall or F_1-score of the predictions, as value of the metric.

After having inspected the checks and collected the metrics, the system triggers the efficient computation of the metrics (see Section 4 for details on how we physically execute these computations), invokes the user-defined validation code from the constraints, and evaluates the results.

Output. After execution of the data quality verification, our system reports which constraints succeeded and which failed, including information on the predicate applied to the metric into which the constraint was translated, and the value that made a constraint fail.

```
...
Success("isComplete(title)",
 Completeness("title") == 1.0)),
Success("isNonNegative(count)",
 Compliance("count >= 0") == 1.0)),
Failure("isUnique(customerId, countryResidence,
  deviceType, title)",
 Uniqueness("customerId", "countryResidence",
 "deviceType", "title") == 1.0, 0.9967),
Failure("isInRange(priority, ('hi', 'lo'))",
 Compliance("priority IN ('hi', 'lo')") == 1.0,
  0.833),
...
```

Listing 2: Exemplary output of data quality verification showing metrics, applied predicates and results.

Listing 2 shows an excerpt of a potential output for our example. We see that our `isComplete(title)` constraint has been translated to a predicate `Completeness(title) == 1.0` which held on the data. Analogously, our constraint `isNonNegative(count)` leads to the predicate `Compliance( "count >= 0") == 1.0` and also matched all records. On the contrary, our unique constraint has failed, as only 99.67% of records have been identified as unique, and the predicate which the system generated from the `isInRange` constraint only matched 83.3% of records.

3.2 Incremental Computation of Metrics for Growing Datasets

In real-world deployments, data is seldomly static; instead we typically encounter systems that continuously produce data (e.g., by interacting with users). Therefore it is of utter importance for data validation systems like ours to support scenarios where we continously ingest new batches of records for a particular dataset. In such cases, we need access to updated metrics for the whole dataset as well as for the new records and we must be able to update such metrics incrementally without having to access the previous data (see Figure 1 for details). In the following, we present our incremental metrics computation machinery built for this task.

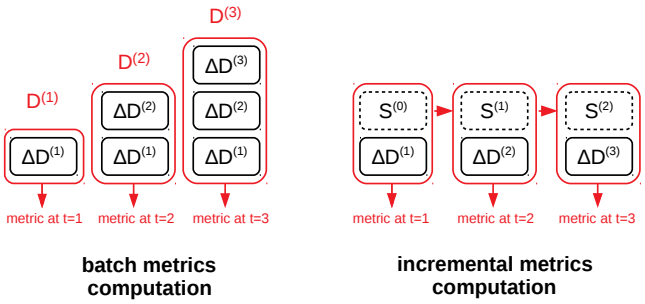

Figure 1: Instead of repeatedly running the batch computation on growing input data D, we support running an incremental computation that only needs to consume the latest dataset delta $\Delta D^{(t)}$ and a state S of the computation.

Computational model. Let $D^{(t)}$ denote the snapshot of dataset at time t and let $\Delta D^{(t)}$ denote the delta data at time t (the additional records) required to form $D^{(t+1)}$. Note that we restrict ourselves to append-only cases where a dataset at time t is simply the union of all previous deltas: $D^{(t)} = \bigcup_{k=1}^{t} \Delta D^{(k)}$. Instead of computing metrics for the growing dataset from all snapshots, incremental computation introduces a state S, a function f for updating the state from a delta and the previous state, and a function g for computing the actual metric from the state S such that $m^{(t)} = g(S^{(t)})$. Furthermore, we need an initial 'empty' state $S^{(0)}$. The benefit of incremental computation is that it allows us to compute the series of metrics for the dataset snapshots via a recursive computation that *only consumes the deltas*:

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

Reformulation of quality metrics. In the following, we present a set of reformulations of the existing metrics to enable incremental computation for them. For each metric, we show how to 'split' the computation of the metrics for the new dataset $D^{(t+1)}$ into the computation of sufficient statistics over the previous dataset D and the dataset delta ΔD (we drop the indexes here to improve readability). Once such a reformulation is given, we can conduct the computation for $D^{(t+1)}$ by loading the persisted sufficient statistics for D and updating these from values computed only on the newly added records ΔD.

Notation: Let N and ΔN denote the number of records in the datasets D and ΔD. Let V and ΔV denote all unique values in a particular column of the dataset D or ΔD. The set of unique values in the new dataset $V^{(t+1)}$ is simply the union $V \cup \Delta V$ of the sets of unique values from the previous dataset and the delta dataset. Furthermore, let c_v and Δc_v denote the cardinality of value v in a particular column of dataset D or ΔD.

The number of records `Size` is the most straightforward metric to rewrite, as the size of the new dataset $D^{(t+1)}$ is simply the sum $N + \Delta N$ of the size N of the previous dataset D plus the size ΔN of the delta dataset ΔD. For an incremental version of `Compliance`, we need to maintain two intermediate results, namely the absolute number of records $|\{d \in D \mid p(d)\}|$ that previously matched the predicate as well as the size N of the previous dataset D. Then we can compute the compliance for the new dataset $D^{(t+1)}$ from these retained values and the number of records $|\{d \in \Delta D \mid p(d)\}|$ that matched the predicate in the delta as well as the size ΔN of the delta:

$$\frac{|\{d \in D \mid p(d)\}| + |\{d \in \Delta D \mid p(d)\}|}{N + \Delta N}$$

We can reformulate `Completeness` as compliance with an 'is not null' predicate. The incremental computation of `Uniqueness` requires us to know the cardinalities c_v of the value v in the previous dataset D as well as the set of distinct values V. We need to inspect the sum of the cardinalities $c_v + \Delta c_v$ for each value v in the previous dataset and the delta:

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

We also compute incremental `Distinctness` along these lines by comparing the number of distinct values in the data $|V \cup \Delta V|$ to the size of the data $N + \Delta N$:

$$\frac{|V \cup \Delta V|}{N + \Delta N}$$

Incremental computation of `Entropy` requires us to estimate the probability $p(v)$ of a particular value v occurring in the column from the value's cardinality c_v in the previous data, its cardinality Δc_v in the delta and the sizes N and ΔN of the previous dataset and the delta:

$$-\sum_v \frac{c_v + \Delta c_v}{N + \Delta N} \log \frac{c_v + \Delta c_v}{N + \Delta N}$$

The incremental computation of `MutualInformation` requires us to maintain histograms about the cardinalities c_{v_1} of the first column, c_{v_2} of the second column, as well as cooccurrence counts $c_{v_1 v_2}$ for all pairwise occurrences, and merge these with the corresponding counts $\Delta c_{v_1 v_2}$ for the delta dataset:

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

In order to compute our `Predictability` metric, we evaluate the prediction quality of a multinomial naive bayes model (trained on features extracted from the user-specified input columns) for the target column. The parameters are typically estimated using a smoothed version of the maximum likelihood estimate: $\text{argmax}_k \sum_i f_i \log \frac{N_{ki}+\alpha_i}{N_k+\alpha}$. Here, N_{ki} denotes the number of times feature i occurs in class k, N_k stands for the overall number of feature occurrences in class k, a uniform prior used for the sake of simplicity, α_i is the smoothing term per feature and α the sum of the smoothing terms. For updating a classification model from a previous dataset D to $D^{(t+1)}$, we need to know $N_{ki}^{(t+1)}$ and $N_k^{(t+1)}$, which we can easily compute by adding the counts ΔN_{ki} and ΔN_k from the delta ΔD to the counts N_{ki} and N_k for the previous version $D^{(t)}$ of the dataset:

$$\text{argmax}_k \sum_i f_i \log \frac{N_{ki} + \Delta N_{ki} + \alpha_i}{N_k + \Delta N_k + \alpha}$$

The data structures which we use for the `ApproxQuantile` and `ApproxCountDistinct` metrics naturally support incremental computations and therefore do not require special care from our side.

3.3 Constraint Suggestion

The benefits of our system to users heavily depend on the richness and specificity of the checks and constraints, which the users define and for which our system will regularly compute data quality metrics. As a consequence, it is very important for a system like ours to make the adoption process as simple as possible. Therefore we provide machinery to automatically suggest constraints and identify data types for datasets (even if no schema is available). Such suggestion functionality can then be integrated into ingestion pipelines and can also be used during exploratory data analysis. The starting point for our constraint suggestion machinery is a dataset where individual columns are known and have names, but no further schema information such as data types or constraints is available. A classical example for such a dataset would be a CSV file living in a distributed filesystem. Our system assists the user in identifying data types of columns and suggests potential constraints to users, which they can use as a foundation to design declarative checks for the dataset at hand.

Heuristics on summary statistics. Our constraint suggestion functionality is built on a heuristics-based approach employing single-column profiling [1]. While more complex data profiling would certainly be helpful, we are required to be able to consume terabyte-sized tables with several billions of rows, and therefore have to restrict ourselves to simple statistics. As already explained, the user provides a single table dataset with no type information and no schema information except column names as input. Furthermore, the user can optionally specify a set of columns to inspect (and a sample size to use during the suggestion phase) to speedup the process. Our system then executes single column profiling in three passes over the data. In the first pass, we compute the data size, run data type detection on each column, and compute the completeness as well as the approximate number of distinct values via hyperloglog sketches [21, 18] for each column of interest. The profiling tasks in the second pass operate on the columns which we identified to have numeric types. For every such column, we compute summary statistics such as the minimum, maximum, mean, standard deviation, and approximate quartiles [15]. In a third pass, we compute the frequency distribution of values for columns with a cardinality below a user-specified threshold (in order to bound the required memory). Afterwards, our system recommends constraints for the dataset at hand, based on heuristics leveraging the profiling results. In the following, we list a selection of the heuristic rules which we apply:

- If a column is complete in the sample at hand, we suggest an `isComplete` (not null) constraint.
- If a column is incomplete in the sample at hand, we suggest a `hasCompleteness` constraint. We model the fact whether a value is present or not as a Bernoulli-distributed random variable, estimate a confidence interval for the corresponding probability, and return the start value of the interval as lower bound for the completeness in the data.
- If the detected type of the column is different from 'string', we suggest a `hasConsistentType` constraint for the detected type.
- For key discovery, we investigate an approximation of the 'unique row ratio' [12]: if the ratio of dataset size to the approximated number of distinct values in that column is within the approximation error bound of the used hyperloglog sketch, we suggest an `isUnique` constraint.
- If a column is numeric and its observed values fall into a certain range, we suggest a `Compliance` constraint with a predicate that matches only values within that range (e.g., a range of only positive values if the observed minimum is 0).
- If the number of distinct values in a column is below a particular threshold, we interpret the column as categorical and suggest an `isInRange` constraint that checks whether future values are contained in the set of already observed values.

Note that we see constraint suggestion as a 'human-in-the-loop' process with a low computational budget and therefore rely on the end user to select from and validate our suggestions which might not necessarily hold for future data (or even the sample at hand in the case of unique constraint suggestions).

Learning semantics of column and table names. We notice that the actual names of columns often contain inherent semantics that allow humans to intuitively infer column types and constraints. Examples for such names are 'id', which is commonly used for an artifical primary key column of type string or int, 'is_deleted', which probably refers to boolean column, or 'price_per_unit' which indicates a numeric column. We therefore train a machine learning model to predict constraints solemnly based on the name of the table and column, as well as its type. The training data for this model is extracted from the schemas of tables from open source projects. Our system integrates this model by leveraging its predictions to enhance (and potentially correct) the suggestions made by our heuristics. In the case where our heuristic rules suggest an `isUnique` constraint, we consult the classifier's probabilistic prediction to decide whether to follow the suggestion or not.

3.4 Anomaly Detection

Anomaly detection in our system operates on historic time series of data quality metrics (e.g., the ratio of missing values for different versions of a dataset). We pose no restriction on the anomaly detection algorithm to apply and our system ships with a handful of standard algorithms. Examples are an algorithm that simply checks for user-defined thresholds. The algorithm from our example called `OnlineNormal` computes a running mean and variance estimate and compares the series values to a user-defined bound on the number of standard deviations they are allowed to be different from the mean. An additional method allows users to specify the degree of differencing applied prior to running the anomaly detection. This gives users the possibility to apply a simple technique in order to stationarize the to-be analysed time series [22]. Data quality metrics such as the number of missing values in a continously produced dataset might be subject to seasonality or trends (e.g., the loss only occurs at certain times when the system is under heavy load). In these cases, asserting the correct behaviour may not be feasible with user-supplied thresholds. To this end, we allow users to plug in their own algorithms for anomaly detection and time series prediction.

4. IMPLEMENTATION

We implement our data validation library on top of the distributed dataflow engine *Apache Spark* [50], using AWS infrastructure for storage. Note that our library does not depend on any functionality exclusive to Spark, and would be easily extendable to leverage different runtimes, as long as they support SQL queries, user-defined aggregation functions and simple machine learning models[1]. We decided for Spark due to the fact that a Scala/JVM environment makes it very easy for users to write custom verification code and interact with external libraries and systems. Figure 2 gives an overview of the applied architecture. Our system operates on `DataFrames`, a relational abstraction for a partitioned (and often denormalized) table. The user-facing API consists of so-called `Checks` and `Constraints`, which allow our users to declaratively define on which statistics of the data their verification code should be run. When executing the checks, our library inspects the contained constraints

[1]A system with support for materialized views would even allow us to simplify our incremental computation machinery.

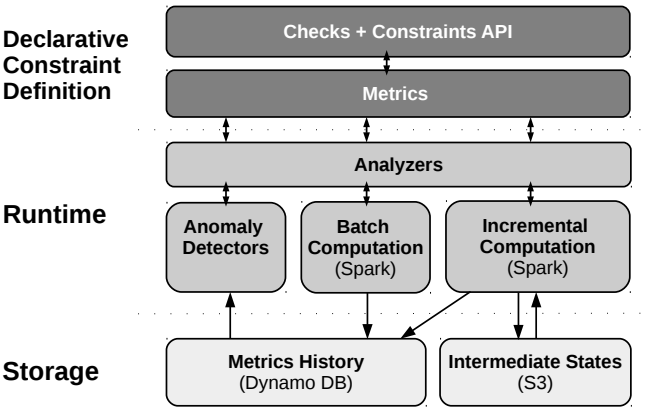

Figure 2: System architecture: users declaratively define checks and constraints to combine with their verification code. The system identifies the required metrics and efficiently computes them in the runtime layer. The history of metrics and intermediate states of incremental computations are maintained in AWS storage services.

and identifies the required `Metrics` that must be computed on the data in order to run the user-defined verification code of the constraints. For each metric, our library chooses a so-called `Analyzer` (which can be seen as a physical operator) capable of computing the particular metric on the data. The selected `Analyzers` are given to an `AnalysisRunner` in our runtime layer which schedules the execution of the metrics computation. This runner applies a set of simple optimizations to the computation of multiple metrics. For all metrics that do not require re-partitioning the data, the runner collects their required aggregation functions and executes them in a single generated *SparkSQL* query over the data to benefit from scan-sharing. In our example from Section 3.1, such metrics would be the `Size` of the dataset, the `Completeness` of six columns, as well as the `Compliance` for the three `satisfies` constraints. All these metrics will be computed simultaneously in a single pass over the data. The resulting metrics are finally stored in a document database (DynamoDB) for later retrieval (and usage by anomaly detection algorithms). The runtime for incremental computations stores the states of incremental analyzers in a distributed filesystem (S3).

For the predictability estimation, we have to train a machine learning model on the user-specified input columns and evaluate how well it predicts values in the target column. We developed a pluggable architecture, where we featurize the input columns by concatenating their string representations, and tokenize and hash them via Spark's `Tokenizer` and `HashingTF` transformers. Afterwards, any classification algorithm from *SparkML* [27] can be used to learn a prediction model; in our experiments we used Sparks Naive Bayes [36] implementation as it offers a scalable lower bound on prediction accuracy, does not require hyperparameter optimization, and is simple to train incrementally. We apply the trained classification model to predict values on a held-out fraction of the data and report the prediction quality (e.g., measured using precision) as predictability value.

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 `StateProvider`s.

```
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
}
```

Listing 3: Interface for incremental analyzers.

State management. In order to execute the incremental computation, a user needs to configure state providers to allow for state management. Typically, the state for a particular snapshot of the dataset resides in a directory on S3 and we provide a corresponding provider implementation. Given these, we call `applyOrUpdateFromPersistedState` which will compute the metric and persist the state. To compute the updated metric for the next snapshot of the dataset, we need two `StateProvider`s, one which provides the state for the old snapshot, and another one which will receive the updated state computed from the old state and the delta. Note that this internal API is typically hidden from the users, which are advised to program our system using the declarative checks API from Section 3.1. In the following we discuss additional implementation details. When incrementally computing metrics that require a re-partitioning of the data (e.g., entropy and uniqueness that require us to group the data by the respective column), we implement the incremental scheme as follows. The *state* S is composed of a histogram over the data (the result of `delta.select(columns).groupBy(columns).count()`). The *update function* merges

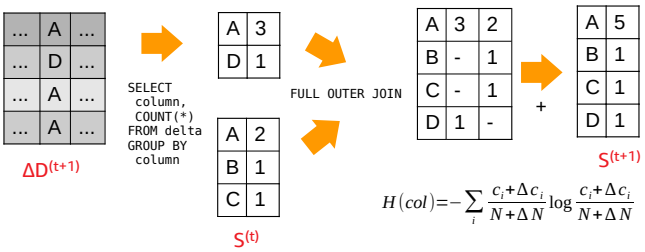

Figure 3: Example for an incremental update of the entropy of a column: the frequencies of values in the delta records $\Delta D^{(t+1)}$ are computed via a grouping query and merged with the previous state $S^{(t)}$ via a full outer join. After adding up the counts, we have the updated state $S^{(t+1)}$, from which the entropy of the column in the updated dataset $D^{(t+1)}$ can be computed.

the previous histogram for the current data with the histogram of the delta via an outer join on the grouping columns and computes the corresponding counts for the delta and the whole dataset. The analyzer then computes the metric from the state by an aggregation over the histogram. We showcase an example of incrementally updating the entropy of column in Figure 3.

Optimizations. During the computation of multiple metrics, we apply a set of manually enforced query optimizations: (a) we cache the result of the `count` operation on dataframes, as many metrics require the size of the delta for example; (b) we apply *scan sharing* for aggregations: we run all aggregations that rely on the same grouping (or no grouping) of the data in the same pass over the data.

4.2 Efficiently Suggesting Constraints

The major design objective in our constraint suggestion component is to keep the computation of required summary statistics cheap so that it can be executed during an ingestion pipeline for large datasets. It is therefore crucial to keep the number of passes over the data independent of the number of columns in the dataframe at hand. We assume our input to be a dataframe with named columns with unknown types (initially of type 'string' during ingestion, e.g., when reading CSV files). For the computation of the summary statistics required for our heuristics from Section 3.3, we only use aggregations that do not require a re-partitioning of the table, and we only conduct two passes over the data, where the aggregations share scans. Furthermore, we have an estimate of the number of distinct values per column of interesting columns after the first pass, which allows us to control the memory for sketches and histograms (e.g., by only computing the full value distribution for columns with low cardinality) used in the second pass.

As discussed in Section 3.3, we leverage a machine learning model for deciding upon unique constraint suggestion. This model's inputs are the table name, as well as column name and type. As training data for this model, we extract a dataset of 2,453 `(table_name, column_name, type, is_unique)` tuples from the database schemas of several open source projects such as *mediawiki*, *wordpress* and *oscommerce*. On this schema data, we train a logistic regression

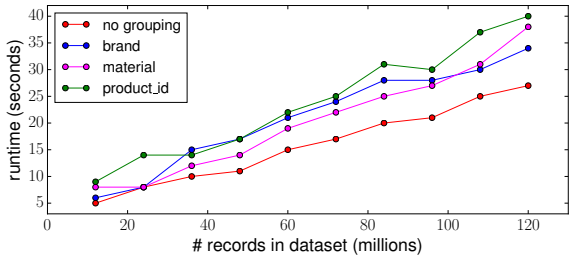

Figure 4: Linearly increasing runtime for different batch metrics computations on a growing product dataset with up to 120 million records.

model using hashed character n-grams of the names and a one-hot-encoding of the type as features. We leverage the `SGDClassifier` combined with the `HashingVectorizer` from *scikit-learn* [32], and tune the model's hyperparameters (feature vector dimensionality, regularization, size of n-grams) using 5-fold cross validation. We achieve an AUC score of 0.859 for the ROC curve, using a logistic loss function with L1 regularization and a regularization factor of 0.001 on n-grams of up to size 5 from the input data hashed to 10^8 dimensional feature vectors. We leverage the probabilistic prediction of this model (giving us a hint on whether the naming of the column indicates a unique constraint) as a score for our rule-based unique constraint suggestion and only issue the suggestion to the user if the model assigns a probability greather than 50%.

5. EXPERIMENTAL EVALUATION

In the following, we run a set of scalability experiments for our batch metrics computation, apply our predictability estimation to a product dataset (Section 5.1), and investigate the benefits of applying incremental computation to growing datasets (Section 5.2). Finally, we evaluate our constraint suggestion functionality on two external datasets (Section 5.3) and showcase a simple anomaly detection use-case in Section 5.4.

For our Spark-based experiments, we leverage a dataset representing a sample from an internal product catalog, which consists of approximately 120 million records, where each record describes a product with several hundred attributes; the data size is roughly 50GB in parquet format. We mimic a use case with a growing append-only dataset, and randomly partition our product data into 10 'deltas' of about 12 million records for that. Additionally, we use two external datasets for evaluation. The first dataset[2] consists of comments from May 2015 in several discussion boards on the social news aggregation website *reddit.com*. The second dataset contains information about 5,180 twitter users, which we extracted from the publicly available twitter sample stream.

The Spark-based experiments leverage a cluster on Elastic MapReduce with 5 workers (c3.4xlarge instances) running Apache Spark 2.0.2 and HDFS 2.7.3.

[2] `https://www.kaggle.com/reddit/reddit-comments-may-2015`

5.1 Batch Computation

In this first set of experiments, we evaluate how well our metrics computation scales to large datasets and showcase the efficiency of our machine learning-based predictability estimation.

Scalability of metrics computations. In order to evaluate the scalability of our system's batch metrics computation, we compute a set of quality metrics on the resulting growing product dataset, which we read from S3. Figure 4 shows the results, where each point represents the runtime on a particular version of the growing dataset. The plot labeled 'no grouping' refers to the results for computing a set of six metrics (size of the data and completeness of five columns) which do not require us to re-partition the data. Therefore these metrics can be computed by aggregations in a single pass over the data. The remaining lines refer to the computation of metrics such as entropy and uniqueness on the columns `brand`, `material` and `product_id`, which require us to repartition the data (e.g., grouping it by the column for which we want to compute these metrics). Due to the inherent re-partitioning, these metrics are typically more costly to compute and their cost is related to the cardinality of the respective column. Nevertheless, all four evaluated workloads exhibit a runtime linearly growing with the dataset size, which is expected as our system internally generates simple aggregation queries with custom aggregation functions to be run by *SparkSQL* [3].

Predictability estimation with naive bayes. We showcase our predictability estimation functionality on a set of 845,000 fashion products from 10 popular brands which we extracted from the larger product dataset. We set the task of predicting the value of the `brand` column from other columns, such as `name`, `description`, `size`, `bulletpoints`, `manufacturer` and combinations of those. We run the corresponding experiments with Spark on a single c4.8xlarge instance. We take different samples from the dataset (100k records, 200k records, 400k records, 600k records, full dataset). On each of these, we train a naive bayes model with hashed input columns as features for predicting the `brand` column on 80% of the sample. Finally, we calculate the weighted F_1 score for the predictions on the remaining 20%. We repeat this for different input columns and differently sized samples of the data. We find that the `name` column alone is already a very good predictor for `brand`, as it results in a weighted F_1 score of over 97% on all samples of the dataset. The best results are achieved when leveraging a combination of the `name` column with the `bulletpoints`, `description` and `manufacturer` columns, where we reach F_1 scores of 97.3%, 98.8%, 99.4%, 99.5% for the 100k , 200k, 400k, 600k samples of the data, and of 99.5% for the full dataset. Based on these results (which can be computed with an `AnalysisRunner` from Section 4), a user could configure a constraint for future data to get notified once the predictability drops below the observed values, e.g.:

```
Check(Level.Warning)
 .hasPredictability("brand", ("name",
   "bulletpoints", "description",
   "manufacturer"), f1=0.97)
```

We additionally record the runtime of the model training for different featurizations. We find that the runtime scales linearly for growing data and mostly depends on the length

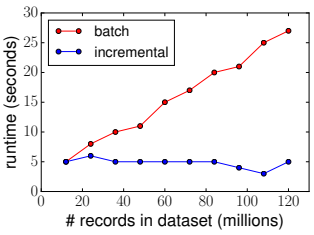

Figure 5: Runtimes for size and completeness on `product_id`, `material`, `color`, `brand`.

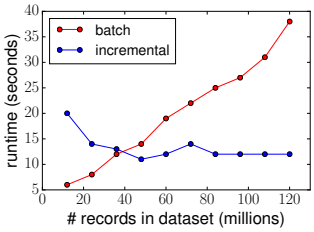

Figure 6: Runtimes for uniqueness and entropy on `material`.

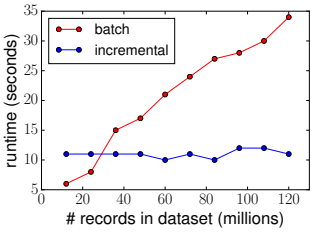

Figure 7: Runtimes for uniqueness and entropy on `brand`.

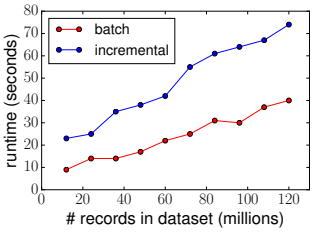

Figure 8: Runtimes for uniqueness and entropy on `product_id`.

of the strings in the input columns (e.g., training a model on the `description` column alone with lots of text takes longer than training on the `name` column combined with the `bulletpoints` column). This is expected as naive bayes conducts a single pass through the data, and sums up the feature vectors per class, which result from tokenizing and hashing the input columns. Note that the training is very efficient; it takes less than ten seconds in all cases, even on the full dataset.

5.2 Benefits of Incremental Computation

We revisit our scalability experiment on growing data to validate our assumptions about the benefits of incremental computation. We compare batch analysis, which always has to consume the dataset as a whole (the union of all the deltas seen so far) against our incremental approach which maintains a state and always operates on this state and the current delta only.

Figure 5 shows the results for again computing the metrics that do not require us to re-partition the data (we referred to this experiment as 'non-grouping' previously). These metrics can be computed in a single pass over the data, and the actual state for the incremental computation is tiny here, as only one or two numbers per metric have to be maintained. While the runtime of the batch analysis grows linearly with the dataset size, the runtime remains constant in the incremental case, as it only depends on the size of the delta (which is constant in our setup). Next, we revisit the computation of metrics such as entropy and uniqueness which require us to repartition the data. These metrics are typically more costly to compute and the incremental computation is also more difficult, as we have to internally maintain a histogram of the frequencies per value in the column (the result of the grouping operation). We first compute these metrics on the columns `material` and `brand` which have a rather low cardinality. The results are shown in Figure 6 and Figure 7. We see that the incremental approach has a substantive overhead in this case (persisting and joining the maintained histogram), however its runtime stays roughly constant and it outperforms the batch analysis after three or four deltas. Figure 8 shows the resulting runtimes for computing entropy and uniqueness for the `product_id` column. This column is special in this dataset as it consists of unique values only. Due to this characteristic, the runtime of the incremental approach shows the same growth behavior as the runtime of the batch analysis (linearly growing with data size), as every delta introduces a set of new values, and the histogram which the incremental computation maintains is basically just a copy of the original column. The overhead of maintaining this histogram is also what makes the incremental computation always perform worse. While this is a drawback, the incremental approach still has the advantage of not requiring access to the full dataset during computation time, which greatly simplifies ingestion pipelines.

5.3 Constraint Suggestion

We evaluate our constraint suggestion component on a sample of 50,000 records from the reddit dataset as well as on the twitter users dataset. In each case, we take a random sample of 10% of the records, have our system suggest constraints based on the sampled data and compute the coverage of these constraints on the remaining 90% of the datasets. The suggested constraints as well as their coverage is shown in Table 3. The reddit dataset is a very easy case, as all columns are complete and only have types string and integer. This simple structure is reflected by the fact that all suggested constraints suggested hold on the test set. The experiment on the twitter users dataset is more interesting, as we have columns with missing values such as `location` and columns with a small set of discrete values such as `lang`. The completeness of the `location` column in the sample is 0.28076 and the suggested constraint `hasCompleteness >= 0.28` holds on the test data, e.g., the ratio of missing values does not increase. The system correctly suggests an `isUnique` constraint for the columns `id` and `screen_name` both of which are actually primary keys for the data. However, the system also suggests two constraints which do not hold for the data. The first one is the range of values for the `lang` column. Here we identified ten different values which only account for more than 99% of records in the test data, but miss rare languages such as turkish or hungarian. A failing constraint on that column can nevertheless be helpful; we found by manual investigation that the data in this column is not correctly normalized, e.g., there are different capitalizations of the same language value such as 'en-gb' and 'en-GB'. In the second case, the system errouneously suggests an `isUnique` constraint for the `statuses_count` column, due to the fact that there are many different values for this column in the sample at hand and we only known an approximation of the number of distinct values; the uniqueness value of this column is only 64% percent in the test data. The second error is corrected, however, once we leverage the predictions of our classifier for unique constraints: while the classifier assigns a high probability of 81% that our suggested unique constraint on the `id` column is valid, it only assigns a 2% probability to the

Table 3: Constraint suggestion and type prediction for the reddit comments and twitter users dataset. Constraints are suggested based on a 10% sample of the data, and their coverage is computed on the remaining 90% of the data. We leverage a machine learning model trained on column names to decide upon potential unique constraints.

dataset	column	suggested constraints	coverage	classifier score
reddit-comments	id	`isComplete`,	1.0	-
		`isUnique`	1.0	0.83
	created_utc	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`	1.0	-
	subreddit	`isComplete`	1.0	-
	author	`isComplete`	1.0	-
	ups	`isComplete`, `hasConsistentType(integral)`	1.0	-
	downs	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`	1.0	-
	score	`isComplete`, `hasConsistentType(integral)`	1.0	-
	edited	`isComplete`, `isNonNegative`	1.0	-
	controversiality	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`,	1.0	-
		`isInRange(0, 1)`	1.0	-
	text	`isComplete`	1.0	-
twitter-users	id	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`,	1.0	-
		`isUnique`	1.0	0.81
	screen_name	`isComplete`,	1.0	-
		`isUnique`	1.0	0.01
	lang	`isComplete`,	1.0	-
		`isInRange('en', 'pt', ...)`	0.991	-
	location	`hasCompleteness >= 0.28`	1.0	-
	followers_count	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`	1.0	-
	statuses_count	`isComplete`, `hasConsistentType(integral)`, `isNonNegative`,	1.0	-
		`isUnique`	0.636	0.02
	verified	`isComplete`, `hasConsistentType(boolean)`	1.0	-
	geo_enabled	`isComplete`, `hasConsistentType(boolean)`	1.0	-

`statuses_count` column being unique; therefore, we decide against the suggested constraint. Unfortunately, the classifier produces a false negative for the `screen_name` column, which is indeed unique for our sample at hand, however this would also be not immediately obvious to humans (as different users are allowed to have the same screen name on many social networking platforms), and we prefer conservative and robust suggestions (e.g., rather having false negatives than false positives), which build trust in our users.

5.4 Anomaly Detection

In order to showcase our anomaly detection functionality, we apply it to a fictitious use case on the reddit dataset. Assume that we want to leverage the discussion data for an information extraction task such as question answering. A potential data quality problem would now be that the data could contain large amounts of spam and trolling posts, which would negatively influence our machine learning model that we aim to train on the data. If we regularly ingest data from reddit, we would like to be alarmed if there are signs of increased spamming or trolling activity. The reddit data contains a `controversiality` field for each post, and the series of the ratio of such posts in a board per day might be a good signal for detecting potential trolling. To leverage our anomaly detection functionality for this task, we inspect the historic time series of the `Mean(controversiality)` metric per discussion board (`subreddit`) which we would need to compute during ingestions. In our declarative API, the corresponding check looks as follows:

```
Check(Level.Warning, groupBy="subreddit")
 .hasNoAnomalies("controversiality", Mean,
   OnlineNormal(upperDeviationFactor=3))
```

This indicates that we want to be warned if the mean controversiality on a particular day in a discussion board is more than three standard deviations higher than the previous mean. Figure 9 illustrates the result of this analysis for a selection of discussion boards from the reddit dataset. We see that there are discussion boards with relatively low variance in the controversiality over time such as *anime* and *askreddit*. However, there are also boards with spikes in controversiality such as *cringepics* and *chicagobulls* which get marked by our anomaly detection approach. Manual investigation of these spikes revealed that they are strongly correlated to the number of deleted users on the particular day, which indicates that they indeed result from trolling behavior.

6. LEARNINGS

We report on learnings on different levels that we gained from users of our data validation system. On an *organisational level*, there are many benefits in using a common data quality library. Such a common library helps establish a shared vocabulary across teams for discussing data quality and also establishes best practices on how to measure data quality, leading to a common way to monitor the metrics of datasets. It is also a big advantage if producers as well as consumers of datasets leverage the same system for verifying data quality, as they can re-use checks and constraints from each other, e.g., the producer can choose to adapt checks from downstream consumers earlier in the data processing pipeline.

On a *technical level*, users highlighted the fact that our data quality library runs on Spark, which they experienced as a fast, scalable way to do data processing, partly due to the optimizations that our platform is applying. Our sys-

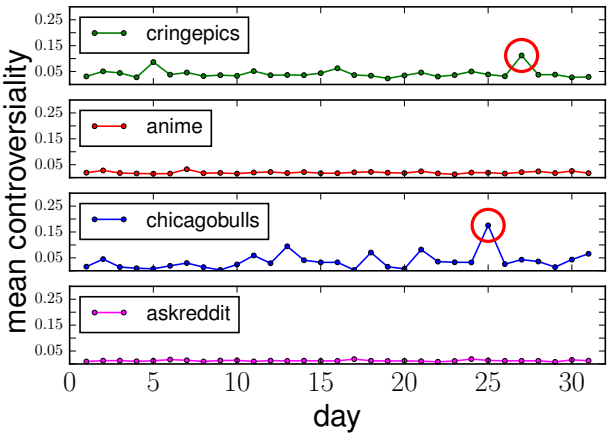

Figure 9: Anomalies detected in the time series of the `Mean(controversiality)` metric for different boards in the reddit dataset, which indicate trolling behavior potentially decreasing data quality.

tem helped reduce manual and ad-hoc analysis on their data, e.g., sampling and eyeballing the results to identify possible problems such as incomplete fields, outliers and derivations from the expected number of rows. Instead, such checks can now be run in an automated way as a part of ingestion pipelines. Additionally, data producers can leverage our system to halt their data publishing pipelines when they encounter cases of data anomalies. By that, they can ensure that downstream data processing, which often includes training ML models, is only working with vetted data.

7. RELATED WORK

Data cleaning has been an active research area for decades, see recent surveys [1, 10, 38] for an overview.

Declarative data quality verification The idea to allow declarative definitions of data quality standards is well-established. Ilyas et al. provide an overview on standard consistency definitions [23]. In fact, every DBMS supports standard *integrity constraints* such as key constraints or nullable fields. A relevant extension are *denial constraints*, which is a first order logic formalism that allows the coverage of more business rules across two tuples [11]. A popular paradigm for defining dependencies between columns are *functional dependencies* and *conditional functional dependencies* [6]; there exist fast, approximate algorithms for discovering them [31]. In contrast to this line of work, our predictability metric relies on ML to learn relationships between columns and uses empirical tests on hold-out datasets for validation. We conjecture that, due to their inherent robustness to outliers, ML methods more suitable for our use case to automatically detect changes in data quality on many large datasets. Galhardas et al. propose a declarative language *AJAX* for data cleaning as an extension of SQL as well as an execution model for executing data cleaning programs [14]. We similarly optimize the validation of our declarative data quality constraints to minimize computational effort. We combine a larger set of constraints into a unified framework, but we do not support the automatic execution of data repair methods.

ML for data cleaning Multiple researchers have suggested to use ML for cleaning data. While traditional methods can be used to generate candidates for fixing incorrect data (e.g., violations of functional dependencies), active learning methods can be used to select and prioritize human effort [49]. *ActiveClean* similarly uses active learning for prioritization, but at the same time it learns and updates a convex loss model [24]. *HoloClean* generates a probabilistic model over a dataset that combines integrity constraints and external data sources to generate data repair suggestions [35]. *BoostClean* automatically selects an ensemble of error detection and repair combinations using statistical boosting [25].

Data validation in ML applications The challenges involved in building complex machine learning applications in the real-world have recently been highlighted by many researchers, e.g., with respect to managing the resulting models [26], software engineering [42, 8], pipeline abstractions [2, 43, 46], and learnings from real-world systems [34, 7]. A new focus is being put on data management questions with respect to large-scale machine learning. Examples include managing the metadata of artifacts produced during machine learning workloads, including schemas and summary statistics of the datasets used for training and testing [47, 48, 40, 29, 28, 20, 33, 41], the discovery and organization of enterprise datasets [16], and machine learning specific sanity checks [45]. As a consequence, modern machine learning platforms begin to have explicit data validation components [7, 5, 9].

8. CONCLUSION

We presented a system for automating data quality verification tasks, which scales to large datasets and meets the requirements of production use cases. The system provides a declarative API to its users, which combines common quality constraints with custom validation code, and thereby enables 'unit tests' for data. We discussed how to efficiently execute the constraint validation by translating checks to scalable metrics computations, and elaborated on reformulations of the metrics to enable incremental computations on growing datasets. Additionally, we provided examples for the use of machine learning techniques in data quality verification, e.g. for enhancing constraint suggestions, for estimating the predictability of a column and for detecting anomalies in historic data quality timeseries.

In the future, we want to extend our machine learning-based constraint suggestion by leveraging more metadata as well as historical data about constraints defined with our API. Moreover, we will investigate the benefits of fitting well-known distributions to numeric columns to be able to understand this data in more detail and suggest more fine-grained constraints. Another direction is to provide users with more comprehensive error messages for failing checks and allow them easy access to records that made a particular constraint fail. Furthermore, we will apply seasonal ARIMA [22] and neural network-based time series forecasting [13] in order to enhance our anomaly detection functionality to also be able to handle seasonal and intermittent time series. Finally, we will we streaming explore validating data quality in streaming scenarios, which should be a natural extension of our discussed incremental use case.

9. REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

[2] P. Andrews, A. Kalro, H. Mehanna, and A. Sidorov. Productionizing Machine Learning Pipelines at Scale. *Machine Learning Systems workshop at ICML*, 2016.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. *SIGMOD*, 1383–1394, 2015.

[4] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino. Methodologies for data quality assessment and improvement. *ACM Computing Surveys*, 41(3):16, 2009.

[5] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *KDD*, 1387–1395, 2017.

[6] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. *ICDE*, 746–755, 2007.

[7] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic demand forecasting at scale. *PVLDB*, 10(12):1694–1705, 2017.

[8] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. *Big Data*, 1123–1132, 2017.

[9] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data Infrastructure for Machine Learning. *SysML*, 2018.

[10] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. *SIGMOD*, 2201–2206, 2016.

[11] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[12] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. *SIGMOD*, 240–251, 2002.

[13] V. Flunkert, D. Salinas, and J. Gasthaus. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *CoRR*, abs/1704.04110, 2017.

[14] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. *VLDB*, 371–380, 2001.

[15] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. *SIGMOD Record*, 30:58–66, 2001.

[16] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. GOODS: Organizing Google's Datasets. *SIGMOD*, 795–806, 2016.

[17] A. Y. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *Intelligent Systems*, 24(2):8–12, 2009.

[18] H. Harmouch and F. Naumann. Cardinality estimation: An experimental survey. *PVLDB*, 11(4):499–512, 2017.

[19] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[20] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, et al. Ground: A data context service. *CIDR*, 2017.

[21] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. *EDBT*, 683–692, 2013.

[22] R. J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2014.

[23] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4), 281–393, 2015.

[24] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu. Activeclean: An interactive data cleaning framework for modern machine learning. *SIGMOD*, 2117–2120, 2016.

[25] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu. Boostclean: Automated error detection and repair for machine learning. *CoRR*, abs/1711.01299, 2017.

[26] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 44(4):17–22, 2016.

[27] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in Apache Spark. *JMLR*, 17(1):1235–1241, 2016.

[28] H. Miao, A. Li, L. S. Davis, and A. Deshpande. On model discovery for hosted data science projects. *Workshop on Data Management for End-to-End Machine Learning at SIGMOD*, 6, 2017.

[29] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. *ICDE*, 571–582, 2017.

[30] F. Naumann. *Quality-driven Query Answering for Integrated Information Systems*. Springer, 2002.

[31] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. *SIGMOD*, 821–833, 2016.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, 2011.

[33] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB*, 10(12):1841–1844, 2017.

[34] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. *SIGMOD*, 1723–1726, 2017.

[35] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[36] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger. Tackling the poor assumptions of naive bayes text classifiers. *ICML*, 616–623, 2003.

[37] T. Rukat, D. Lange, and C. Archambeau. An interpretable latent variable model for attribute applicability in the amazon catalogue. *Interpretable ML Symposium at NIPS*, 2017.

[38] S. Sadiq, J. Freire, R. J. Miller, T. Dasu, I. F. Ilyas, F. Naumann, D. Srivastava, X. L. Dong, S. Link, and X. Zhou. Data quality the role of empiricism. *SIGMOD Record*, 46(4):35–43, 2018.

[39] M. Scannapieco and T. Catarci. Data quality under a computer science perspective. *Archivi & Computer*, 2, 1–15, 2002.

[40] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. *Machine Learning Systems workshop at NIPS*, 2017.

[41] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert. Declarative Metadata Management: A Missing Piece in End-to-End Machine Learning. *SysML*, 2018.

[42] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison. Hidden Technical Debt in Machine Learning Systems. *NIPS*, 2503–2511, 2015.

[43] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 535–546, 2017.

[44] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. *ICCV*, 843–852, 2017.

[45] M. Terry, D. Sculley, and N. Hynes. The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets. *Machine Learning Systems Workshop at NIPS*, 2017.

[46] T. van der Weide, D. Papadopoulos, O. Smirnov, M. Zielinski, and T. van Kasteren. Versioning for end-to-end machine learning pipelines. Workshop on Data Management for End-to-End Machine Learning at SIGMOD, 2, 2017.

[47] J. Vanschoren, J. N. Van Rijn, B. Bischl, and L. Torgo. OpenML: networked science in machine learning. *KDD*, 49–60, 2014.

[48] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A System for Machine Learning Model Management. *Workshop on Human-In-the-Loop Data Analytics at SIGMOD*, 14, 2016.

[49] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 95, 2010.