

Auto-Validate by-History: Auto-Program Data Quality Constraints to Validate Recurring Data Pipelines

Dezhan Tu*

University of California, Los Angeles

Yeye He, Weiwei Cui, Song Ge, Haidong Zhang,

Han Shi, Dongmei Zhang, Surajit Chaudhuri

Microsoft Research

ABSTRACT

Data pipelines are widely employed in modern enterprises to power a variety of Machine-Learning (ML) and Business-Intelligence (BI) applications. Crucially, these pipelines are *recurring* (e.g., daily or hourly) in production settings to keep data updated so that ML models can be re-trained regularly, and BI dashboards refreshed frequently. However, data quality (DQ) issues can often creep into recurring pipelines because of upstream schema and data drift over time. As modern enterprises operate thousands of recurring pipelines, today data engineers have to spend substantial efforts to *manually* monitor and resolve DQ issues, as part of their DataOps and MLOps practices.

Given the high human cost of managing large-scale pipeline operations, it is imperative that we can *automate* as much as possible. In this work, we propose AUTO-VALIDATE-BY-HISTORY (AVH) that can automatically detect DQ issues in recurring pipelines, leveraging rich statistics from historical executions. We formalize this as an optimization problem, and develop constant-factor approximation algorithms with provable precision guarantees. Extensive evaluations using 2000 production data pipelines at Microsoft demonstrate the effectiveness and efficiency of AVH.

1 INTRODUCTION

Data pipelines are the crucial infrastructure underpinning the modern data-driven economy. Today, data pipelines are ubiquitous in large technology companies such as Amazon, Google and Microsoft to power data-hungry businesses like search and advertisement [14, 61, 66, 73]. Pipelines are also increasingly used in traditional enterprises across a variety of ML/BI applications, in a growing trend to democratize data [7].

Production data pipelines are often *inter-dependent*, forming complex “webs”, where input tables used by downstream pipelines frequently depend on output tables from upstream pipelines.

Furthermore, these pipelines are often configured to *recur* on a regular basis (e.g., hourly or daily), to ensure data stay up-to-date for downstream use cases (e.g., fresh data enables ML models to be re-trained regularly, and BI dashboards refreshed continuously).

*Work done at Microsoft.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

KDD '23, August 6–10, 2023, Long Beach, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0103-0/23/08...\$15.00

<https://doi.org/10.1145/3580305.3599776>

Recurring Pipelines: Prone to Fail due to DQ. The *recurring* and *inter-dependent* nature of production pipelines make them vulnerable to failure due to data quality (DQ) issues, because over time unexpected DQ issues, such as data drift [54] and schema drift [14, 63], can creep in, causing cascading issues in pipelines.

Although DQ issues in data pipelines are widely documented in the literature (especially in industry settings [14, 61, 66, 73]), we describe a few common types of DQ issues from the literature, in order to make the discussion concrete and self-contained:

- **Schema drift:** A newly arrived batch of input data may have a schema change compared to previous input (e.g., missing columns or extra columns), which can result in incorrect behavior in data pipelines [14, 63].
- **Increasing nulls:** There is sometimes a sudden increase of null, empty strings, or special values (e.g., -1) in a column due to external factors – for instance, Google reports a DQ incident where null values in a column increase substantially in a short period of time, because the module that populates data in this column encountered an unusual number of RPC time-outs from a networking outage [54].
- **Change of units:** The unit of measurement for numeric values can change over time, when the logic that populates data evolves – for instance, Google reports a real DQ issue in their search ranking [54], where the program that populates the “age” field of web documents previously used the unit of “days” (e.g., a document that is 30 days old will have an “age” value of 30), which later got changed to “hours” (making the same document to be have the “age” value of 720). This leads to orders of magnitude larger “age” values, and incorrect behaviours downstream.
- **Change of value standards:** Value standards for string-valued data can change over time – for instance, Amazon reports a DQ issue where a “language-locale” column previously used lowercase values like “en-gb”, which later changed into uppercase “en-GB”, creating a mixed bag of inconsistent values in the same column, leading to incorrect behaviours in downstream applications [63].
- **Change of data volume:** The volume (e.g., row-count) for a new batch of data in a recurring pipeline can change significantly from previous batches, which can also be indicative of DQ issues.

This list of DQ issues is clearly not exhaustive as there are many other types of DQ issues documented in the literature [14, 61, 63].

When DQ issues arise in recurring pipelines, they tend to introduce *silent* failures (i.e., with no explicit exceptions thrown, or error messages generated). The silent nature of DQ issues makes them difficult to catch, but no less damaging. For example, when null values increase significantly, or the unit of measurement changes, downstream ML models will continue to operate but will likely churn out inaccurate predictions (e.g., Google reports a DQ issue in their production pipelines that causes a recommendation model

```

// define a data quality check
.addCheck(
  Check(CheckLevel.Error, "Review Check")
  .isUnique("review_id") // should not contain duplicates
  .isComplete("marketplace") // should never be NULL
  // contains only the listed values
  .isContainedIn("marketplace", Array("US", "UK", "DE", "JP", "FR"))
  .isNonNegative("year") // should not contain negative values

```

(a) Amazon's Deequ: each arrow points to a column-level constraint

```

# Add skew comparator for 'payment_type' feature.
payment_type = tfdv.get_feature(schema, 'payment_type')
payment_type.skew_comparator.infinity_norm.threshold = 0.01

# Add drift comparator for 'company' feature.
company = tfdv.get_feature(schema, 'company')
company.drift_comparator.infinity_norm.threshold = 0.001

skew_anomalies = tfdv.validate_statistics(train_stats, schema,
                                         previous_statistics=eval_stats,
                                         serving_statistics=serving_stats)

```

(b) Google's TFDV: each arrow points to a column-level constraint

Figure 1: Declarative Data-validation using manually-programmed constraints

in Google Store to produce sub-optimal results – fixing this single DQ issue improves their apps install rate by 2% [14]).

In general, “silent” DQ failures pollute downstream data products, which makes it more time-consuming for engineers to detect/debug/fix. Silent DQ failures in pipelines are therefore a major pain point in MLOps and DataOps practices [14, 61, 63, 66].

“Guardrails” for Pipelines: Data Validation. Technology companies with large-scale data pipeline operations are among the first to recognize the need for employing data-validation “guardrails” in recurring pipelines to catch DQ issues early as they arise. A number of data-validation tools have been developed, including Google’s TensorFlow Data Validation (TFDV) [20] and Amazon’s Deequ [60, 61].

These tools develop easy-to-use domain-specific languages (DSLs), so that engineers can write declarative DQ constraints that describe how “normal” data should look like in recurring data pipelines, such that unexpected deviation in the future can be flagged for review.

Figure 1(a) shows an example code snippet from Amazon Deequ. Using the DSL introduced in Deequ, one could declare the “review_id” column to be unique, the “marketplace” column to be complete (with no NULLs), etc.. These constraints are then used to validate future data arriving in the recurring pipeline.

Figure 1(b) shows a similar example from Google’s TFDV, which specifies that, when a new batch of input data arrives in a pipeline, the distributional distance of values in the “payment_type” column should be similar to the same column from previous batches (the code snippet therefore specifies that the L-infinity distance of the two should be no greater than 0.01).

Automate Data Validation: Leveraging History. While these DSL-based declarative data-validation solutions improve upon low-level assertions and can improve DQ in production pipelines as reported in [14, 61, 63], they require data engineers to manually write data constraints *one-column-at-a-time* like shown in Figure 1 (with a lone exception in [57], which employs off-the-shelf anomaly detection algorithms). This is clearly time-consuming and hard to scale – large organizations today operate thousands of pipelines, and hundreds of columns in each table, it is impractical for engineers to manually program DQ for each column.

We emphasize that writing DQ constraints is not just time-consuming, sometimes it is also genuinely difficult for humans to program DQ correctly, because users need to (1) have a deep understanding of the underlying data including how the data may evolve over time; and (2) be well-versed in complex statistical metrics (e.g., L-infinity vs. JS-divergence), before they can program DQ effectively. Consider the example of online user traffic, such data can fluctuate quickly over time (e.g., between different hours of the day and different days of the week), which is hard to anticipate and even harder to program using appropriate metrics and thresholds.

To address this common pain point, in this work we propose to “auto-program” DQ, by leveraging “history”. Our insight is that rich statistical information from past executions of the same pipeline (e.g., row-counts, unique-values, value-distributions, etc.) is readily available, which can serve as strong signals to reliably predict whether a new batch of data may have DQ issues or not.

To see why this is the case, consider a simplistic example where all K past executions of a recurring pipeline produce exactly 50 output rows (one row for each of the 50 US states). This row-count becomes a “statistical invariant” unique to this particular pipeline, which can serve as a good predictor for DQ issues in the future – deviations from the invariant in new executions (e.g. an output with only 10 rows or 0 row), would likely point to DQ issues.

Obviously, simple row-counts are not the best DQ predictor for all pipelines, as some pipeline can have row counts that can vary significantly. In such cases, DQ constraints based on other types of statistical metrics will likely be more effective.

Table 1 and Table 2 list common statistical metrics used to program DQ constraints (details of these metrics can be found in Appendix A). Tools like TFDV and Deequ already support many of these metrics today, but it is difficult for humans to manually select suitable metrics, and then guess what thresholds would work well. Our proposed AVH aims to automatically program suitable DQ from this large space of statistical primitives, so that the resulting DQ is tailored to the underlying pipeline, without human intervention.

Overall, our proposed AVH is designed to have the following properties that we believe are crucial in recurring pipelines:

- **Automated.** Instead of requiring humans to manually program DQ constraints column-at-a-time, AVH can auto-program rich DQ leveraging statistics from past executions.
- **Highly accurate.** AVH is specifically designed to achieve high accuracy, as frequent false-alarms would require constant human attention that can erode user confidence quickly. In AVH, we aim for very low False-Positive-Rate or FPR (e.g., 0.01%), which is also configurable if users so choose. AVH can then auto-program DQ guaranteed not to exceed that FPR, while still maximizing the expected “recall” (the number of DQ issues to catch).
- **Robust.** Unlike traditional ML methods that require a significant amount of training data, we exploit rich statistical properties (Chebyshev, Chantelli, CLT, etc.) of the underlying metrics, so that the predictions are robust even with limited historical data (e.g. only a few days of histories).
- **Explainable.** AVH produces explainable DQ constraints using standard statistical metrics (as opposed to black-box models), which makes it possible for human engineers to understand, review, and approve, if human interactions become necessary.

Contributions. We make the following contributions in this work.

- We propose a novel problem to auto-program pipeline DQ leveraging history, formalized as a principled optimization problem specifically optimizing both precision and recall.
- We develop algorithms that leverage the different statistical properties of the underlying metrics, to achieve a constant-factor approximation, while having provable precision guarantees.
- Our extensive evaluation on 2000 real production pipelines suggests that AVH substantially outperforms a variety of commercial solutions, as well as SOTA methods for anomaly detection from the machine learning and database literature.

2 RELATED WORK

Data validation. Data validation for pipelines is an emerging topic that has attracted significant interest from the industry, including recent efforts such as Google’s TensorFlow Data Validation (TFDV) [20], Amazon’s Deequ [60, 61], and LinkedIn’s Data Sentinel [67]. With the lone exception of [57], most existing work focuses on developing infrastructures and DSLs so that engineers can program DQ constraints in a declarative manner.

Anomaly detection. Anomaly detection has been widely studied in time-series and tabular settings [10–12, 29, 37, 47], and is clearly related. We compare with an extensive set of over 10 SOTA methods from the anomaly detection literature, and show AVH is substantially better in our problem setting of DQ in pipelines, because AVH uniquely exploits the statistical properties of underlying metrics, whereas standard anomaly detection methods would treat each metric as just another “feature dimension”. This enables AVH to have higher accuracy, and excel with even limited data (e.g., 7 days of historical data), as we will show in our experiments.

Data cleaning. There is a large literature on data cleaning (e.g., surveyed in [31, 36, 43, 55, 56]), which we also compare with. Most existing work focuses on the static single-table setting [24, 26, 36, 38, 56, 66, 71, 72], where data errors need to be detected from one table snapshot. In comparison, we study how multiple historical table snapshots from recurring pipelines can be explicitly leveraged for data quality, which is a setting not traditionally considered in the data cleaning literature.

3 PRELIMINARY: DQ IN PIPELINES

In this section, we introduce necessary preliminaries for programming Data Quality (DQ) in the context of data pipelines.

As we discussed, data pipelines are ubiquitous today, yet DQ issues are common in recurring pipelines, giving rise to data-validation tools such as Google’s TFDV and Amazon’s Deequ. At its core, these methods validate DQ by checking input/output tables of recurring pipelines, against pre-specified DQ constraints. Most of these constraints are defined over a single column C at a time (Figure 1). Single-column DQ is also the type of DQ we focus in this work.

While TFDV and Deequ use syntactically different DSLs to program DQ constraints, the two are very similar in essence, as both can be described as constraints based on statistical metrics.

DQ constraints by statistical metrics. Most DQ primitives used for pipeline validation can be expressed as statistics-based constraints. Table 1 and Table 2 list common statistical metrics used in DQ (e.g., row-count, L-infinity, etc.). We denote this space of possible metrics by \mathbf{M} . This is obviously a large space that requires time and expertise from users to navigate and select appropriately.

We now define two types of DQ constraints using metric $M \in \mathbf{M}$, which we call *single-distribution* and *two-distributions DQ constraints*, respectively.

DEFINITION 1. A *single-distribution DQ constraint*, denoted by a quadruple $Q(M, C, \theta_l, \theta_u)$, is defined using a statistical metric $M \in \mathbf{M}$, over a target column of data C , with lower-bound threshold θ_l and upper-bound θ_u . The constraint $Q(M, C, \theta_l, \theta_u)$ specifies that C has to satisfy the inequality $\theta_l \leq M(C) \leq \theta_u$, or the metric $M(C)$ is expected to be between the range $[\theta_l, \theta_u]$ (otherwise the constraint Q is deemed as violated).

Single-distribution DQ can be instantiated using example metrics shown in Table 1 and Table 2. Such DQ constraints rely on a single data distribution of a column C , and can be validated using a newly-arrived batch data alone. We illustrate this using an example below.

EXAMPLE 1. In the example Deequ snippet shown in Figure 1(a), the “review_id” column is required to be unique, which can be expressed as a single-distribution DQ using the *unique_ratio* metric from Table 2, as $Q_1(\text{unique_ratio}, \text{review_id}, 1, 1)$, equivalent to $Q_1 : 1 \leq \text{unique_ratio}(\text{review_id}) \leq 1$ (where the upper-bound θ_u and lower-bound θ_l converge to the same value 1). If on the other hand, uniqueness is required to be high, say at least 95% (but not 100%), we can write as $Q_2(\text{unique_ratio}, \text{review_id}, 0.95, 1)$, or $Q_2 : 0.95 \leq \text{unique_ratio}(\text{review_id}) \leq 1$. Other examples in Figure 1(a) can be written as single-distribution DQ similarly.

Next, we introduce *two-distribution DQ constraints* that require comparisons between two distributions of a column.

DEFINITION 2. A *two-distribution DQ constraint*, denoted as $Q(M, C, C', \theta_l, \theta_u)$, is defined using a statistical metric $M \in \mathbf{M}$, that compares one batch “target” data in a column C , and a batch of “baseline” data C' , using lower-bound threshold θ_l and upper-bound threshold θ_u . Formally we write $Q(M, C, C', \theta_l, \theta_u) = \theta_l \leq M(C, C') \leq \theta_u$, which states that the metric $M(C, C')$ comparing C and C' is expected to be in the range $[\theta_l, \theta_u]$ (or Q is as violated otherwise).

Two-distribution DQ compares a target column against a baseline column, which can be the same column from two consecutive executions of the same pipeline, or two batches of training/testing data, etc. We illustrate this in the example below.

EXAMPLE 2. In the example TFDV snippet shown in Figure 1(b), the first constraint specifies that for the “payment_type” column, we expect two batches of the same data to differ by at most 0.01 using the L-infinity metric. This can be written as $Q_3 : 0 \leq L_{\text{inf}}(C, C') \leq 0.01$. The second constraint in Figure 1(b) defined on the “company” column, can be specified using two-distribution DQ similarly.

Conjunctive DQ program. Given a target column C , it is often necessary to validate C using multiple orthogonal metrics in \mathbf{M} (e.g., both row-counts and distribution-similarity need to be checked, among other things). In this work, we consider conjunctions of multiple DQ constraints, which we call a *conjunctive DQ program*. Note that the use of conjunction is intuitive, as we want all DQ to hold at the same time (prior work in TFDV and Deequ also implicitly employ conjunctions, as the example in Figure 1 shows).

DEFINITION 3. A *conjunctive DQ program*, defined over a given set of (single-distribution or two-distribution) DQ constraints S ,

Type	Metrics
Two-distribution	Earth Mover’s distance (EMD) [59], Jensen–Shannon divergence (JS_div) [28], Kullback–Leibler divergence (KL_div) [28], Two-sample Kolmogorov–Smirnov test (KS_dist) [50], Cohen’s d ($Cohen_d$) [27]
Single-distribution	$min, max, mean, median, sum, range, row_count, unique_ratio, complete_ratio$

Table 1: Statistical metrics used to generate DQ constraints for numerical data (details in Appendix A).

Type	Metrics
Two-distribution	L-1 distance[18], L-infinity distance[18], Cosine distance[34], Chi-squared test[30], Jensen–Shannon divergence (JS_div) [28], Kullback–Leibler divergence (KL_div) [28]
Single-distribution	$str_len, char_len, digit_len, punc_len, row_count, unique_ratio, complete_ratio, dist_val_count$

Table 2: Statistical metrics used to generate DQ constraints for categorical data (details in Appendix A).

denoted by $P(S)$, is defined as the conjunction of all $Q_i \in S$, written as $P(S) = \bigwedge_{Q_i \in S} Q_i$.

EXAMPLE 3. Continue with Example 2, let C denotes the target column “payment_type”. In addition to the aforementioned constraint $Q_3 : 0 \leq L_{inf}(C, C') \leq 0.01$, one may additionally require that this column to be at least 95% complete (with less than 5% of nulls), written as $Q_4 : 0.95 \leq complete_ratio(C) \leq 1$. Furthermore, we expect to see no more than 6 distinct values (with “cash”, “credit”, etc.) in this column, so we have $Q_5 : 0 \leq distinct_cnt(C) \leq 6$.

Putting these together and let $S = \{Q_3, Q_4, Q_5\}$, we can write a conjunctive program $P(S) = \bigwedge_{Q_i \in S} Q_i$ (or $Q_3 \wedge Q_4 \wedge Q_5$).

4 AUTO-VALIDATE-BY-HISTORY

While DQ programs are flexible and powerful, they are difficult to write manually. We now describe our AVH to auto-program DQ.

4.1 Problem Statement

For the scope of this work, we consider auto-generating conjunctive DQ programs for each column C in data pipelines (or only for a subset of important columns selected by users), using column-level single-distribution or two-distribution DQ constraints (Section 3).

For a given column C , our goal is to program suitable DQ by selecting from a large space of metrics M in Table 1 and Table 2. This space of M is clearly large and hard to program manually. Also note that while we list commonly-used metrics, the list is not meant to be exhaustive. In fact, AVH is designed to be *extensible*, so that new metrics (e.g., statistical distances relevant to other use cases) can be added into M in a way that is transparent to users.

For a given C and a set of possible M , this induces a large space of possible DQ constraints on C . We denote this space of possible single-distribution and two-distribution DQ as Q , defined as:

$$Q = \{Q(M, C, \theta_l, \theta_u) | M \in \mathbf{M}, \theta_l \in \mathbb{R}, \theta_u \in \mathbb{R}, \theta_l \leq \theta_u\} \cup \{Q(M, C, C', \theta_l, \theta_u) | M \in \mathbf{M}, \theta_l \in \mathbb{R}, \theta_u \in \mathbb{R}, \theta_l \leq \theta_u\} \quad (1)$$

We note that in production settings, it is crucial that auto-generated DQ programs are of high precision, with very few false-alarms (false-positive detection of DQ issues). This is because with thousands of recurring pipelines, even a low *False-Positive Rate (FPR)* can translate into a large number of false-positives, which is undesirable as they usually require human intervention. Because it is critical to ensure high precision, in AVH we explicitly aim for a very low level of FPR, which we denote by δ , e.g., $\delta = 0.1\%$.

Finally, because we are dealing with data from recurring data pipelines, we assume that the same data from K past executions of this pipeline is available, which we denote as $H = \{C_1, C_2, \dots, C_K\}$. These K previous batches of data are assumed to be free of DQ issues, which is reasonable because engineers usually manually check the first few pipeline runs after a pipeline is developed to

ensure it runs properly. DQ issues tend to creep in over time due to data drift and schema drift [14, 63].

AUTO-VALIDATE-BY-HISTORY (AVH). Given these considerations, we now formally define our problem as follows.

DEFINITION 4. AUTO-VALIDATE-BY-HISTORY. Given a target column C from a pipeline, and the same data from previous K executions $H = \{C_1, C_2, \dots, C_K\}$, a space of possible DQ constraints Q , and a target false-positive-rate (FPR) δ . Construct a conjunctive DQ program $P(S)$ with $S \subseteq Q$, such that the expected FPR of $P(S)$ is no greater than δ , while $P(S)$ can catch as many DQ issues as possible. We write AVH as the following optimization problem:

$$(AVH) \quad \max R(P(S)) \quad (2)$$

$$\text{s.t. } FPR(P(S)) \leq \delta \quad (3)$$

$$P(S) = \bigwedge_{Q_i \in S, S \subseteq Q} Q_i \quad (4)$$

Where $R(P(S))$ denotes the expected recall of a DQ program $P(S)$ that we want to maximize, and $FPR(P(S))$ denotes its expected FPR, which is required to be lower than a target threshold δ .

4.2 Construct DQ constraints

To solve AVH, in this section we will first describe how to construct a large space of DQ constraints Q (and estimate their FPR), which are pre-requisites before we can use Q to generate conjunctive programs for AVH (in Section 4.3).

Recall that to instantiate constraints like $Q_i(M, C, \theta_l, \theta_u)$ (from Definition 1), we need to pick a metric $M \in \mathbf{M}$, apply M on the given column C to compute $M(C)$, and constrain $M(C)$ using suitable upper/lower-bounds thresholds θ_u/θ_l .

Here we leverage the fact that a history $H = \{C_1, C_2, \dots, C_K\}$ of the same column C from past executions is available. If we apply M on H , we obtain $M(H) = \{M(C_1), M(C_2), \dots, M(C_K)\}$, which forms a statistical distribution¹. When we apply the same metric M on a newly arrived batch of data C , the resulting value $M(C)$ can then be seen as a data point drawn from the distribution $M(H)$. Let the estimated mean and variance of $M(H)$ be μ and σ^2 , respectively. We can construct a DQ constraint $Q(M, C, \theta_l, \theta_u)$, with the following probabilistic FRP guarantees.

PROPOSITION 1. For any metric $M \in \mathbf{M}$, and $\beta \in \mathbb{R}^+$, we can construct a DQ constraint $Q(M, C, \theta_l, \theta_u)$, with $\theta_l = \mu - \beta$, $\theta_u = \mu + \beta$. The expected FPR of the constructed Q on data without DQ issues, denoted by $E[FPR(Q)]$, satisfy the following inequality:

$$E[FPR(Q)] \leq \left(\frac{\sigma}{\beta}\right)^2 \quad (5)$$

¹Later, we will discuss exceptions to the assumption (e.g., non-stationary time-series).

PROOF. We prove this proposition using Chebyshev’s inequality [16]. Chebyshev states that for a random variable X , $P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$, $\forall k \in \mathbb{R}^+$. For the random variable $M(C)$, let $k = \frac{\beta}{\sigma}$. Replacing k with $\frac{\beta}{\sigma}$ above, we get $P(|M(C) - \mu| \geq \beta) \leq (\frac{\sigma}{\beta})^2$. Note that this implies $P(|M(C) - \mu| \leq \beta) \geq 1 - (\frac{\sigma}{\beta})^2$, which can be rewritten as $P(-\beta \leq M(C) - \mu \leq \beta) \geq 1 - (\frac{\sigma}{\beta})^2$, or $P(\mu - \beta \leq M(C) \leq \mu + \beta) \geq 1 - (\frac{\sigma}{\beta})^2$.

Observe that $\mu - \beta \leq M(C) \leq \mu + \beta$ is exactly our $Q(C, M, \theta_l, \theta_u)$, where $\theta_l = \mu - \beta$, $\theta_u = \mu + \beta$. We thus get $P(Q \text{ holds on } C) \geq 1 - (\frac{\sigma}{\beta})^2$, which is equivalent to saying that the expected FPR of Q is no greater than $(\frac{\sigma}{\beta})^2$, or $E[FPR(Q)] \leq (\frac{\sigma}{\beta})^2$. \square

We use the following example to illustrate such a constructed DQ constraint and its estimated FPR.

EXAMPLE 4. Consider a metric $M = \text{complete_ratio}$ from Table 1 that computes the fraction of values in a column C that are “complete” (not-null), and a history of data from past executions $H = \{C_1, C_2, \dots, C_K\}$. Applying M on H , we obtain the *complete_ratio* on historical data as $M(H) = \{0.92, 0.90, \dots, 0.91\}$.

From the sample $M(H)$, we estimate its mean $\mu = 0.9$ and variance of $\sigma^2 = 0.0001$, respectively. Using Proposition 1, suppose we set $\beta = 0.05$, we get a $Q_6(C, \text{complete_ratio}, 0.85, 0.95)$ (or equivalently $0.85 \leq \text{complete_ratio}(C) \leq 0.95$), whose expected FPR has the following inequality: $E[FPR(Q_6)] \leq (\frac{0.01}{0.05})^2 = 0.04$.

Note that using different β allows us to instantiate different constraints with different levels of FPR. For example, setting $\beta = 0.1$ will induce a different $Q_7(\text{complete_ratio}, C, 0.8, 1)$ (or $0.8 \leq \text{complete_ratio}(C) \leq 1$), whose expected FPR is $E[FPR(Q_7)] \leq (\frac{0.01}{0.1})^2 = 0.01$. Note that this yields a lower FPR than that of Q_6 above, because Q_7 has a wider upper/lower-bound for *complete_ratio*.

Using Proposition 1 and different β values, we can instantiate an array of DQ constraints using the same M but different $[\theta_l, \theta_u]$ (thus different FPR guarantees). A DQ with a larger β allows a larger range of $M(C)$ values, which is less sensitive/effective in catching DQ issues, but is also “safer” with lower expected FPR.

Tighter bounds of FPR leveraging metric properties. The results in Proposition 1 apply to any metric $M \in \mathbf{M}$, and the corresponding bounds on FRP are loose as a result. We derive two tighter FRP bounds for specific types of statistical metrics below, by exploiting unique characteristics of these metrics.

PROPOSITION 2. For any metric $M \in \{\text{EMD}, \text{JS_div}, \text{KL_div}, \text{KS_dist}, \text{Cohen_d}, L_1, L_{\text{inf}}, \text{Cosine}, \text{Chi_squared}\}$, and any $\beta \in \mathbb{R}^+$, we can construct a DQ constraint $Q(M, C, \theta_l, \theta_u)$, with $\theta_l = 0$, $\theta_u = \mu + \beta$. The expected FPR of the constructed Q on data without DQ issues, denoted by $E[FPR(Q)]$, satisfy the following inequality:

$$E[FPR(Q)] \leq \frac{\sigma^2}{\beta^2 + \sigma^2} \quad (6)$$

This bound is derived using Cantelli’s inequality [16], a proof of which can be found in Appendix F.

PROPOSITION 3. For any metric $M \in \{\text{count}, \text{mean}, \text{str_len}, \text{char_len}, \text{digit_len}, \text{punc_len}, \text{complete_ratio}\}$, and any $\beta \in \mathbb{R}^+$, we can construct a DQ constraint $Q(M, C, \theta_l, \theta_u)$, with $\theta_l = \mu - \beta$,

$\theta_u = \mu + \beta$. The expected FPR of the constructed Q on data without DQ issues, denoted by $E[FPR(Q)]$, satisfy the following inequality:

$$E[FPR(Q)] \leq 1 - \frac{2}{\sqrt{\pi}} \int_0^{\frac{\beta}{\sqrt{2}\sigma}} e^{-t^2} dt \quad (7)$$

This bound is derived using Central Limit Theorem [16]. We show a proof of this in Appendix G.

We omit examples for Proposition 2 and Proposition 3, but DQ can be constructed similar to Example 4 with tighter bounds.

We note that these tighter bounds allow us to construct DQ constraints with better FPR guarantees, which help to meet the constraint in Equation (3) of AVH more effectively. The pseudo-code of this step can be found in Appendix D.

Time-series Differencing for Non-stationary Data. Our analysis so far assumes $M(H)$ to be a well-behaved distribution, generated from *stationary processes* [53], defined as processes with probability distributions that are *static* and do not change over time. While this is true for many real cases (e.g., Example 4), there are cases where $M(H)$ follows *non-stationary processes* [53], in which parameters of the underlying probability change over time.

EXAMPLE 5. Consider a recurring pipeline that processes one-day’s worth of user traffic data visiting a website. Because overall, the user traffic will grow over time, the volume of data processed by the pipeline will increase slightly every day. So for the metric $M = \text{row_count}$, we get a sequence of row-counts for the past K days as $M(H) = \{100K, 103K, 105K, 106K, \dots, 151K, 152K\}$. Note that $M(H)$ is *non-stationary* here, because the parameters of the underlying distribution (e.g., the mean of $M(C)$) change over time.

Modeling non-stationary $M(H)$ like above as stationary using a static distribution is clearly sub-optimal, which may lead to false-positives and false-negatives in DQ applications.

To account for non-stationary $M(H)$, we first determine whether a $M(H)$ is already stationary, using the Augmented Dickey–Fuller (ADF) test from the time-series literature [23]. If we reject the null-hypothesis in ADF that $M(H)$ is already stationary (e.g., Example 4), we proceed to construct DQ constraints as before. For cases where $M(H)$ is not stationary (e.g., Example 5), we repeatedly apply a technique known as time-series differencing [33] on $M(H)$ until it reaches stationarity. We illustrate this using a small example below, and defer details of the time-series differencing step to Appendix E.

EXAMPLE 6. Continue with Example 5, where $M(H) = \{100K, 103K, 105K, 106K, \dots, 151K, 153K\}$, and the metric $M = \text{row_count}$. The Augmented Dickey–Fuller (ADF) test will fail to reject the null hypothesis that $M(H)$ is non-stationary. Applying a first-order time-differencing step ([53]) with $t = 1$ will produce: $M'_{t=1}(H) = \{M(C_2) - M(C_1), M(C_3) - M(C_2), \dots, M(C_K) - M(C_{K-1})\} = \{3K, 2K, 1K, \dots, 2K\}$. This resulting $M'_{t=1}(H)$ passes the ADF test and is then used as a static distribution to generate Q .

We note that the differencing step also allows us to handle cyclic time-series $M(H)$ (e.g., weekly or hourly periodic patterns), by transforming $M(H)$ using first-order differencing with lags [33], which can then be handled like stationary processes as before.

4.3 Construct DQ Programs in AVH

After we construct constraints \mathbf{Q} and estimate their FPR bounds, we are ready solve AVH. Recall that in AVH, in addition to satisfying the hard constraint on FPR (Equation (3)), our objective (Equation (2)) is to maximize the expected “recall” of the constructed DQ program (the number of possible DQ issues to catch). In order to fully instantiate AVH, we still need to estimate the expected recall benefit of each DQ constraint $Q_i \in \mathbf{Q}$, which can guide us to select the most “beneficial” DQ program.

Estimate DQ recall using synthetic “training”. Clearly, we cannot foresee the exact DQ issues that may arise in the future in a particular pipeline, to precisely quantify the benefit of each $Q_i \in \mathbf{Q}$. However, there is a large literature that documents common types of DQ issues in pipelines (e.g., [14, 54, 61–63, 66, 67]), which include things like schema change, unit change, increased nulls, as discussed earlier. Our observation is that although it is hard to quantify the benefit of Q_i in a specific DQ incident, in the long run if future DQ issues are drawn from the set of common DQ problems, then we can still estimate the expected recall of a specific Q_i .

With that goal in mind, we carefully reviewed the DQ literature and cataloged a list of 10 common types of DQ issues in pipelines (schema change, unit change, increased nulls, etc.). We then vary parameters in each type of DQ to systematically capture different magnitudes of DQ deviations (e.g., different fractions of values are overwritten with nulls for “increased nulls”, different magnitudes of changes for “unit changes”, etc.), to construct a total of 60 procedures that can systematically inject DQ issues in a given column C by varying C . We denote this set of synthetically generated DQ issues on C as $\mathbf{D}(C)$. We give a full list of these common types of DQ issues and their parameters configurations in Appendix B.

Intuitively, this $\mathbf{D}(C)$ models a wide variety of data deviations that may happen in C due to DQ issues, which guides us to select salient $Q_i \in \mathbf{Q}$ that are unique “statistical invariants” specific to a pipeline, to best differentiate between the “normal” H , and the “bad cases” in $\mathbf{D}(C)$, for this pipeline. This synthetic $\mathbf{D}(C)$ in effect becomes “training data” in ML, by assisting us to estimate the recall benefit of Q_i in AVH. We give an example below to illustrate this.

EXAMPLE 7. We revisit the example from the Introduction, where a recurring pipeline produces exactly 50 output rows, with one row for each of the 50 US states, over all past K executions in the history. In such a pipeline, for the “state” column in the output, one distinguishing feature is that the column has exactly 50 distinct values, or $Q_8 : dist_val_cnt(state) = 50$ (which intuitively, is a “statistical invariant” only unique to this pipeline).

When we synthetically inject DQ issues into C (the “state” column) to produce $\mathbf{D}(C)$, we get variants of C , such as C with an increased number of nulls, C with values taken from a neighboring column (due to schema-change), etc. This constraint $Q_8 : dist_val_cnt(state) = 50$ will catch most of such variations in $\mathbf{D}(C)$, thus producing a high expected “recall” and making Q_8 a desirable constraint to use. Intuitively, Q_8 is a good constraint for C in this particular pipeline, because $dist_val_cnt = 50$ is a unique “statistical variant” specific to this column and pipeline, which has more discriminating power than other more generic constraints.

Formally, we define the expected recall of Q_i or $R(Q_i)$, as the set of issues it can detect in $\mathbf{D}(C)$, written as:

$$R(Q_i) = \{C' | C' \in \mathbf{D}(C), C' \text{ fails on } Q_i\} \quad (8)$$

Optimizing AVH with guarantees. Given a DQ program with a conjunction of constraints $P(S) = \bigwedge_{Q_i \in S} Q_i$ for some $S \subseteq \mathbf{Q}$, naturally the recall of two constraints $Q_i, Q_j \in S$ will overlap (with $R(Q_i) \cup R(Q_j) \neq \emptyset$). This leads to diminishing recall for similar DQ constraints in the same program, and requires us to leverage “complementary” constraints when generating DQ programs.

Given a conjunctive program $P(S)$ with $S \subseteq \mathbf{Q}$, we model the collective recall of S , as the union of individual $R(Q_i)$, or $\bigcup_{Q_i \in S} R(Q_i)$. This becomes a concrete instantiation of the objective function in Equation (2) of the AVH problem.

Furthermore, recall that we can upper-bound the FPR of each $Q_i \in S$, using Proposition 1-3. Given a program $P(S)$, assume a worst-case where the FRP of each $Q_i \in S$ is disjoint, we can then upper-bound the FPR of $P(S)$ (Equation (3)), as the sum of the FRP bounds of each Q_i , or $\sum_{Q_i \in S} FPR(Q_i)$.

Together, we rewrite the abstract AVH in Equation (2)-(4) as:

$$(AVH) \quad \max \left| \bigcup_{Q_i \in S} R(Q_i) \right| \quad (9)$$

$$\text{s.t.} \quad \sum_{Q_i \in S} FPR(Q_i) \leq \delta \quad (10)$$

$$S \subseteq \mathbf{Q} \quad (11)$$

Intuitively, we want to weigh the “cost” of selecting a constraint Q_i , which is its estimated $FPR(Q_i)$, against its “benefit”, which is its expected recall $R(Q_i)$. Furthermore, we need to account for the fact that constraints with overlapping recall benefits yield diminishing returns that is analogous to submodularity. We prove that AVH is in general intractable and hard to approximate in Appendix H.

Given that it is unlikely that we can solve AVH optimally in polynomial time, we propose an efficient algorithm that gives a constant-factor approximation of the best possible solution in terms of the objective value in Equation (9), while still guaranteed to satisfy the FPR requirement in Equation (10) in expectation. The pseudo-code of the procedure is shown in Algorithm 1.

Algorithm 1 takes as input a set of metrics \mathbf{M} , an FPR target δ , as well as a column C together with its history $H = \{C_1, C_2, \dots, C_K\}$. We start by constructing a large space of possible DQ constraints \mathbf{Q} (Line 1), using the given \mathbf{M} and H (Section 4.2).

Using this \mathbf{Q} , we then iterate to find a solution $S \subseteq \mathbf{Q}$, which is first initialized to empty. In each iteration, we select the best possible Q_s from remaining constraints in \mathbf{Q} that have not yet been selected (Line 4), based on a cost/benefit calculation, where the “benefit” of adding a constraint Q_i is its increment recall gain on top of the current solution set S , written as $|R(Q_i) \setminus \bigcup_{Q_j \in S} R(Q_j)|$, divided by its additional “cost” of adding Q_i , which is the increased FPR when adding $FPR(Q_i)$. The selected Q_s is then simply the constraint that maximizes this benefit-to-cost ratio, as shown in Line 4. We add this Q_s to the current solution S , update the current total FPR as well \mathbf{Q} , and iterate until we exhaust \mathbf{Q} .

Algorithm 1: Auto-Validate by-History (AVH)

input : Metrics M , a target-FPR δ , column C , and its history $H = \{C_1, C_2, \dots, C_K\}$
output : Conjunctive DQ Program $P(S)$

- 1 $Q \leftarrow \text{Construct-Constraints}(M, H)$
- 2 $S \leftarrow \emptyset, FPR \leftarrow 0$
- 3 **while** $FPR \leq \delta$ **do**
- 4 $Q_s = \arg \max_{Q_i \in Q} \left(\frac{|R(Q_i) \setminus \bigcup_{Q_j \in S} R(Q_j)|}{FPR(Q_i)} \right)$
- 5 **if** $FPR(Q_s) + FPR \leq \delta$ **then**
- 6 $S \leftarrow S \cup Q_s$
- 7 $FPR \leftarrow FPR + FPR(Q_s)$
- 8 $Q \leftarrow Q \setminus Q_s$
- 9 $Q_m = \arg \max_{Q_m \in Q} (|R(Q_m)|)$
- 10 **if** $|\bigcup_{Q_i \in S} R(Q_i)| < |R(Q_m)|$ **then**
- 11 | $S \leftarrow \{Q_m\}$
- 12 **return** $P(S)$

In the final step (Line 9), we compare the best possible singleton $Q_m \in Q$ that maximizes recall without violating the FPR requirement, with the current S from above. We pick the best between $\{Q_m\}$ and S based on their recall as our final solution to AVH.

We show that Algorithm 1 has the following properties (a proof of which can be found in Appendix I).

PROPOSITION 4. *Algorithm 1 is a $(\frac{1}{2} - \frac{1}{2e})$ -approximation algorithm for the AVH problem in Equation (9), meaning that the objective value produced by Algorithm 1 is at least $(\frac{1}{2} - \frac{1}{2e})OPT$, when OPT is the objective value of the optimal solution to AVH. Furthermore, Algorithm 1 produces a feasible solution in expectation, meaning that the expected FPR of its solution is guaranteed to satisfy Equation (10).*

5 EXPERIMENTS

We evaluate the effectiveness and efficiency of AVH, using real production pipelines. Our code will be shared at [3] after an internal review.

5.1 Evaluation Benchmarks

Benchmarks. We perform rigorous evaluations, using real and synthetic benchmarks derived from production pipelines.

- **REAL.** We construct a REAL benchmark using production pipelines from Microsoft’s internal big-data platform [73]. We perform a longitudinal study of the pipelines, by sampling 1000 numeric columns and 1000 categorical columns from these recurring pipelines, and trace them over 60 consecutive executions (which may recur daily or hourly). For each column C , this generates a sequence of history $\{C_1, C_2, \dots, C_{60}\}$, for a total of 2000 sequences.

We evaluate the precision/recall of each algorithm \mathcal{A} (AVH or otherwise) on the 2000 sequences, by constructing sliding windows of sub-sequences for back-in-time tests of \mathcal{A} ’s precision/recall (following similar practices in other time-series domains [17, 21]):

Precision. Given a sequence of past runs $H = \{C_1, C_2, \dots, C_K\}$, if an algorithm \mathcal{A} looks at H together with the real C_{K+1} that arrives next, and predicts C_{K+1} to have data-quality issues, then it is likely a false-positive detection, because the vast majority of production

pipeline runs are free of DQ issues (if there were anomalous runs, they would have been caught and fixed by engineers, given the importance of the production data). To validate that it is indeed the case in our test data, we manually inspected a sample of our production pipeline data and did not identify any DQ issues. (Details of the process can be found in Appendix L.)

For each full sequence $S = \{C_1, C_2, \dots, C_{60}\}$, we construct a total of 30 historical sliding windows (each with a length of 30), as $H_{30} = \{C_1, C_2, \dots, C_{30}\}$, $H_{31} = \{C_2, C_3, \dots, C_{31}\}$, etc. Then at time-step K (e.g., 30), and given the history $H_K = \{C_{K-29}, C_{K-28}, \dots, C_K\}$, we ask each algorithm \mathcal{A} to look at H_K and predict whether the next batch of real data C_{K+1} has a DQ issue or not, for a total of $(2000 \times 30) = 60K$ precision tests.

Recall. For recall, because there are few documented DQ incidents that we can use to test algorithms at scale, we systematically construct recall tests as follows. Given a sliding window of prefix $H = \{C_1, C_2, \dots, C_{30}\}$, we swap out the next batch of real data C_{31} , and replace it with a column C'_{31} that looks “similar” to C_{31} (e.g., with a similar set of values).

Specifically, we use C_{31} as the “seed query”, to retrieve top-20 columns most similar to C_{31} based on content similarity (Cosine), from the underlying data lake that hosts all production pipelines. Because C'_{31} will likely have subtle differences from the real C_{31} (e.g., value-distributions, row-counts, etc.), algorithm \mathcal{A} should ideally detect as many C'_{31} as DQ issues as possible (good recall), without triggering false alarms on the real C_{31} (good precision). Because we retrieve top-20 similar columns, this generates a total of $2000 \times 20 = 40K$ recall tests.²

- **SYNTHETIC.** In addition, we create a SYNTHETIC benchmark, where the precision tests are identical to REAL. For recall tests, instead of using real columns that are similar to C_{31} , we synthetically inject 10 common DQ issues reported in the literature into C_{31} (described in Appendix B). This allows us to systematically test against a range of DQ issues with different levels of deviations.

Evaluation metrics. For each algorithm \mathcal{A} , we report standard precision/recall results on the 60K precision tests and 40K recall tests described above. We use standard precision and recall, defined as $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, where TP, FP, and FN are True-Positive, False-Positive, and False-Negative, respectively.

5.2 Methods Compared

We compare with an extensive set of over 20 methods, including strong commercial solutions, as well as state-of-the-art algorithms from the literature of anomaly detection and data cleaning. We categorize these methods into groups, which we describe below.

Commercial solutions. We compare with the following commercial solutions that aim to automatically validate data pipelines. Google TFDV. We compare with Google’s Tensorflow Data Validation (TFDV) [20]. We install the latest version from Python pip, and use recommended settings in [8].

Amazon Deequ. We compare with Amazon’s Deequ [60, 61], using configurations suggested in their documentations [4].

²It should be noted that some of the C'_{31} columns we retrieve may be so similar to C_{31} that they become indistinguishable, making it impossible for any \mathcal{A} to detect such C'_{31} as DQ issues. This lowers the best-possible recall, but is fair to all algorithms.

Azure Anomaly Detector. Azure Anomaly Detector [1] is a cloud-based anomaly detection service for time-series data, utilizing state-of-the-art algorithms in the literature [58].

Azure ML Drift Detection. Azure ML has the ability to detect data drift over time [2]. We use data from the past K executions as the “baseline” and data from a new execution as the “target”.

Time-Series-based anomaly detection. There is a large body of literature on detecting outliers from time-series data. We use a recent benchmark study [64] to identify the following four best-performing methods, and use the same implementations provided in [9] on our statistical data for comparison purposes.

LSTM-AD [49] employs LSTM networks to learn and reconstruct time series. It uses the reconstruction error to detect anomalies.

Telemanom [40] also uses LSTM networks to reconstruct time-series telemetry, identifying anomalies by comparing expected and actual values and applying unsupervised thresholds.

Health-ESN [22] uses the classical Echo State Network (ESN) and is trained on normal data. Anomalies are detected when the error between the input and predicted output exceeds a certain threshold, which is determined through an information theoretic analysis.

COF [70] is a local density-based method that identifies time-series outliers, by detecting deviations from spherical density patterns.

Classical anomaly detection. We also compare with the following anomaly detection methods developed in tabular settings.

One-class SVM [65] is a popular ML method for anomaly detection, where only one class of training data is available. We train one-class SVM using historical data, and use it to make predictions.

Isolation Forest [46] is also a popular method for anomaly detection based on decision trees. We again train Isolation Forest using historical data, and then predict on newly-arrived data.

Local Outlier Factor (LOF) [15] is another one-class method for anomaly detection based on data density. We configure LOF in a way similar to other one-class methods above.

K-MeansAD [45] is also a classical anomaly detection method, which is based on the unsupervised K-Means clustering.

ECOD [44] identifies outliers by estimating the distribution of the input data and calculating the tail probability for each data point.

Average KNN (Avg-KNN) is another outlier detection method, and was used to automate data validation in a pipeline setting [57] that is similar to the scenario considered in our work.

Statistical tests. We compare with the following classical statistical tests used to detect outliers in distributions.

Kolmogorov–Smirnov (KS) is a classical statistical hypothesis test for homogeneity between two numeric distributions, and is used in prior work to detect data drift [54]. We vary its p-value thresholds to generate PR curves.

Chi-squared is a classical hypothesis test for homogeneity between two categorical distributions, and also used in prior work [54]. We vary its p-value thresholds like above.

Median Absolute Deviation (MAD) is a measure of statistical dispersion from robust statistics [39], and has been used to detect quantitative outliers [36]. We use MAD-deviation (Hampel X84, similar to z-scores) to produce predictions [36].

Database constraints. There is a large literature on using database constraints for data cleaning. We compare with these methods:

Functional Dependency (FD). FD is widely-used to detect data errors in tables [13, 35, 48, 52], by exploiting correlations between columns (e.g., salary \rightarrow tax-rate). Since not all columns can be “covered” by FD, to estimate its best possible recall, we detect all possible FDs from our 2000 test tables, and mark a test column C to be “covered” if there exists a detected FD that has C in its RHS. We report this as FD-UB (Functional Dependency Upper-bound).

Order Dependency (OD) [42, 68, 69]. We discover OD using the same statistical information by ordering tables with statics in time.

Sequential Dependency (SD) [19, 32] generalizes OD, and we discover SD using the same statistical information over time.

Denial Constraints (DC). We use the approach in [25] to discover DC that generalizes FD and OD, and use them for validating data.

AUTO-VALIDATE-BY-HISTORY (AVH). This is our proposed AVH method as described in Section 4.

5.3 Experiment Results

Overall quality comparisons. Figure 2 and 3 show the average precision/recall of different methods on the REAL and SYNTHETIC benchmark, respectively. AVH is at the top-right corner with high precision/recall, outperforming other methods across all cases.

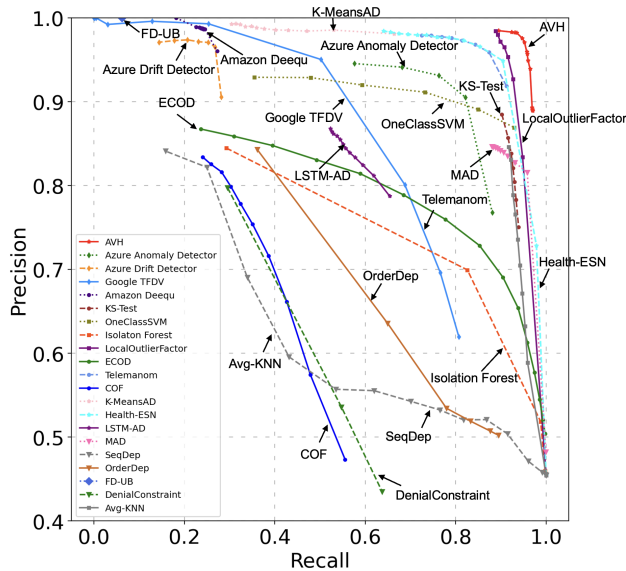
Anomaly detection methods, especially LOF and Health-ESP, are the best performing baselines. However, these methods use each statistical attribute just as a regular dimension in a data record, *while our AVH exploits different statistical properties of the underlying metrics* (Chebyshev, Chantelli, CLT, etc., in Proposition 1-3), which gives AVH an unique advantage over even the state-of-the-art anomaly detection methods, underscoring the importance of our approach in validating data from recurring data pipelines.

Commercial data-validation solutions like Amazon Deduq and Google TFDV have high precision but low recall, because they use predefined and static configurations (e.g., JS-Divergence and L-infinity are the default for TFDV), which lack the ability adapt to different pipelines, and thus the low recall. Similarly, statistical tests (KS/Chi-squared/MAD) use fixed predictors that also cannot adapt to different pipelines, and show sub-optimal performance.

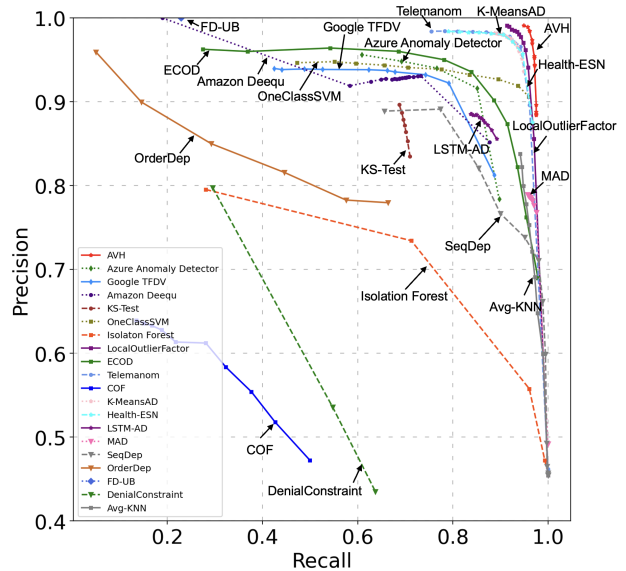
Constraint-based data cleaning methods from the database literature (e.g., FD, DC, OD, SD, etc.) are not competitive in our tests, because these methods are designed to handle single table snapshot, typically using manually designed constraints.

Figure 5 shows breakdown of AVH results in Figure 3 by different types of DQ issues in the SYNTHETIC benchmark. We can see that AVH is effective against most types of DQ issues (schema-change, distribution-change, data-volume-change, etc.). On numerical data, we see that it is the most difficult to detect “character-level perturbation” (randomly perturbing one digit character for another digit with small probabilities) and “character deletion” (randomly removing one digit character with small probabilities), which is not unexpected since such small changes may not always change the underlying numerical distributions. On categorical data, “character-level perturbation” is also the most difficult to detect, but AVH is effective against “character deletion” and “character insertion”.

Sensitivity and ablation studies. We perform extensive experiments to study the sensitivity of AVH (to the length of history, different types of data-errors, target FPR τ , etc.). We also perform an ablation study to understand the importance of AVH components.

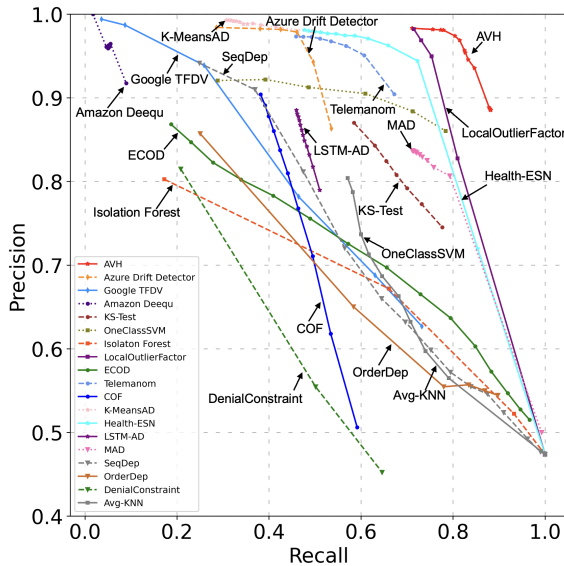


(a) Test on numerical data

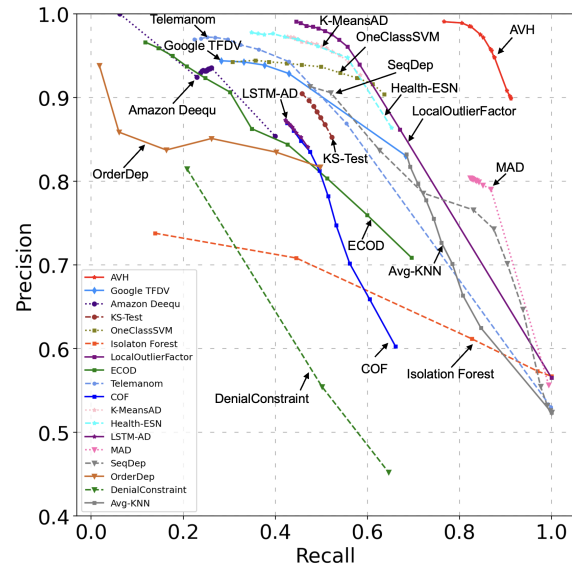


(b) Test on categorical data

Figure 2: Precision/Recall results on the REAL benchmark (2000 real pipelines).



(a) Test on numerical data



(b) Test on categorical data

Figure 3: Precision/Recall results on the Synthetic benchmark.

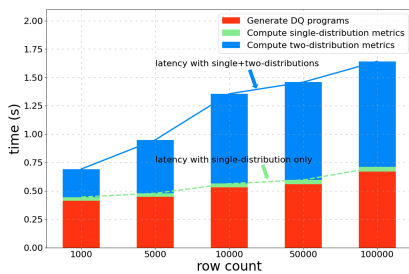


Figure 4: Efficiency analysis of AVH

In the interest of space, we present these additional experimental results in Appendix J and Appendix K, respectively.

Efficiency. Figure 4 shows the end-to-end latency of AVH to process a new batch of data. We vary the number of rows in a column C (x-axis), and report latency averaged over 100 runs. Recall that AVH can be used offline, since DQ constraints can be

auto-installed on recurring pipelines (without involving humans). Nevertheless, we want to make sure that the cost of AVH is small. Figure 4 confirms this is the case – the latency of AVH on 100K rows is 1.6 seconds on average, making this interactive. The figure further breaks down the time spends into three components: (1) computing single-distribution metrics (green), (2) two-distribution metrics (blue), and (3) DQ programs (red), where computing two-distribution metrics (blue) takes the most time, which is expected. Overall, we see that the overall latency grows linearly with an increasing number of rows, indicating good scalability.

6 CONCLUSIONS

In this work, we develop an AUTO-VALIDATE-BY-HISTORY (AVH) framework to automate data-validation in recurring pipelines. AVH

can automatically generate explainable DQ programs that are provably accurate, by leveraging the statistical properties of the underlying metrics. Extensive evaluations on production pipelines show the efficiency and effectiveness of AVH.

REFERENCES

- [1] [n. d.]. Azure Anomaly Detector. <https://azure.microsoft.com/en-us/services/cognitive-services/anomaly-detector/>.
- [2] [n. d.]. Azure ML Drift Detection Service. <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-monitor-datasets>.
- [3] [n. d.]. Code will be open-sourced after an interval review process. <https://github.com/River12/Auto-Validate-by-History>.
- [4] [n. d.]. Deequ: recommended settings. (Retrieved in 02/2022). <https://github.com/aws-labs/deequ/tree/master/src/main/scala/com/amazon/deequ/suggestions/rules>.
- [5] [n. d.]. Full version of the paper Auto-Validate-by-History.
- [6] [n. d.]. Normal distribution and error function. https://en.wikipedia.org/wiki/68%E2%80%939395%E2%80%9399.7_rule.
- [7] [n. d.]. Self-Service Data Preparation, Worldwide, 2016. <https://www.gartner.com/doc/3204817/forecast-snapshot-selfservice-data-preparation>.
- [8] [n. d.]. TFDV: recommended settings. (Retrieved in 02/2022). https://www.tensorflow.org/tfx/data_validation/get_started.
- [9] Retrieved in 2022. Time-series anomaly detection: GitHub code. <https://github.com/HPI-Information-Systems/timeeval-evaluation-paper>.
- [10] Charu C Aggarwal and Philip S Yu. 2001. Outlier detection for high dimensional data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 37–46.
- [11] Sabyasachi Basu and Martin Meckesheimer. 2007. Automatic outlier detection for time series: an application to sensor data. *Knowledge and Information Systems* 11, 2 (2007), 137–154.
- [12] Irad Ben-Gal. 2005. Outlier detection. In *Data mining and knowledge discovery handbook*. Springer.
- [13] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galullin. 2014. Sampling from repairs of conditional functional dependency violations. *The VLDB Journal* 23, 1 (2014), 103–128.
- [14] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *Conference on Systems and Machine Learning (SysML)*. <https://www.sysml.cc/doc/2019/167.pdf>.
- [15] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.
- [16] Michael George Bulmer. 1979. *Principles of statistics*. Courier Corporation.
- [17] Sean D Campbell. 2005. A review of backtesting and backtesting procedures. (2005).
- [18] Cyrus D Cantrell. 2000. *Modern mathematical methods for physicists and engineers*. Cambridge University Press.
- [19] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2015. Relaxed functional dependencies—a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering* 28, 1 (2015), 147–165.
- [20] Emily Caviness, Paul Suganthan GC, Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. Tensorflow data validation: Data analysis and validation in continuous ml pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2793–2796.
- [21] Chris Chatfield. 2000. *Time-series forecasting*. Chapman and Hall/CRC.
- [22] Qing Chen, Anguo Zhang, Tingwen Huang, Qianping He, and Yongduan Song. 2020. Imbalanced dataset-based echo state networks for anomaly detection. *Neural Computing and Applications* 32 (2020), 3685–3694.
- [23] Yin-Wong Cheung and Kon S Lai. 1995. Lag order and critical values of the augmented Dickey–Fuller test. *Journal of Business & Economic Statistics* 13, 3 (1995), 277–280.
- [24] Fei Chiang and Renée J Miller. 2008. Discovering data quality rules. *VLDB* 1, 1 (2008).
- [25] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *VLDB* 6, 13 (2013).
- [26] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. IEEE.
- [27] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge.
- [28] Dominik Maria Endres and Johannes E Schindelin. 2003. A new metric for probability distributions. *IEEE Transactions on Information theory* 49, 7 (2003), 1858–1860.
- [29] Zakia Ferdousi and Akira Maeda. 2006. Unsupervised outlier detection in time series data. In *22nd International Conference on Data Engineering Workshops (ICDEW’06)*. IEEE, x121–x121.
- [30] Stephen E Fienberg. 1979. The use of chi-squared statistics for categorical data problems. *Journal of the Royal Statistical Society: Series B (Methodological)* 41, 1 (1979), 54–64.
- [31] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian Saita. 2001. *Declarative data cleaning: Language, model, and algorithms*. Ph. D. Dissertation. INRIA.
- [32] Lukasz Golab, Howard Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. 2009. Sequential dependencies. *Proceedings of the VLDB Endowment* 2, 1 (2009), 574–585.
- [33] Clive WJ Granger and Roselyne Joyeux. 1980. An introduction to long-memory time series models and fractional differencing. *Journal of time series analysis* 1, 1 (1980), 15–29.
- [34] Vagisha Gupta, Shelly Sachdeva, and Neha Dohare. 2021. Deep similarity learning for disease prediction. *Trends in Deep Learning Methodologies* (2021), 183–206.
- [35] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.
- [36] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* 25 (2008), 1–42.
- [37] Victoria Hodge and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial intelligence review* (2004).
- [38] Zhipeng Huang and Yeye He. 2018. Auto-Detect: Data-Driven Error Detection in Tables. In *SIGMOD*.
- [39] Peter J Huber. 2011. Robust statistics. In *International encyclopedia of statistical science*. Springer, 1248–1251.
- [40] Kyle Hundman, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. 2018. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD international*

- conference on knowledge discovery & data mining. 387–395.
- [41] Samir Khuller, Anna Moss, and Joseph Seffi Naor. 1999. The budgeted maximum coverage problem. *Inform. Process. Lett.* 70, 1 (1999).
- [42] Philipp Langer and Felix Naumann. 2016. Efficient order dependency detection. *The VLDB Journal* 25, 2 (2016), 223–241.
- [43] Peng Li, Xiang Cheng, Xu Chu, Yeye He, and Surajit Chaudhuri. 2021. Auto-FuzzyJoin: auto-program fuzzy similarity joins without labeled examples. In *Proceedings of the 2021 International Conference on Management of Data*. 1064–1076.
- [44] Zheng Li, Yue Zhao, Xiyang Hu, Nicola Botta, Cezar Ionescu, and George Chen. 2022. Ecod: Unsupervised outlier detection using empirical cumulative distribution functions. *IEEE Transactions on Knowledge and Data Engineering* (2022).
- [45] Moisés F Lima, Bruno B Zarpelao, Lucas DH Sampaio, Joel JPC Rodrigues, Taufik Abrao, and Mario Lemes Proença. 2010. Anomaly detection using baseline and k-means clustering. In *SoftCOM 2010, 18th International Conference on Software, Telecommunications and Computer Networks*. IEEE, 305–309.
- [46] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth IEEE international conference on data mining*. IEEE, 413–422.
- [47] Greta M Ljung. 1993. On outlier detection in time series. *Journal of the Royal Statistical Society: Series B (Methodological)* 55, 2 (1993), 559–567.
- [48] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*. 865–882.
- [49] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovkesh Vig, Puneet Agarwal, and Gautam Shroff. 2016. LSTM-based encoder-decoder for multi-sensor anomaly detection. *arXiv preprint arXiv:1607.00148* (2016).
- [50] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.
- [51] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming* 14, 1 (1978), 265–294.
- [52] Thorsten Papenbrock and Felix Naumann. 2016. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*. 821–833.
- [53] Kun Il Park and M Park. 2018. *Fundamentals of probability and stochastic processes with applications to communications*. Springer.
- [54] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1723–1726.
- [55] Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.
- [56] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter’s wheel: An interactive data cleaning system. In *VLDB*, Vol. 1.
- [57] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion. In *EDBT*. 61–72.
- [58] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. 2019. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3009–3017.
- [59] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 1998. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*. IEEE, 59–66.
- [60] Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat, Philipp Schmidt, Stephan Seufert, Pierre Brunelle, and Andrey Taptunov. 2019. Unit Testing Data with Deequ. In *Proceedings of the 2019 International Conference on Management of Data*. 1993–1996.
- [61] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. 2018. Deequ-data quality validation for machine learning pipelines. In *Machine Learning Systems Workshop at the Conference on Neural Information Processing Systems (NeurIPS)*.
- [62] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. 2019. Differential Data Quality Verification on Partitioned Data. In *ICDE*. IEEE, 1940–1945.
- [63] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.
- [64] Sebastian Schmidl, Phillip Wenig, and Thorsten Papenbrock. 2022. Anomaly detection in time series: a comprehensive evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1779–1797.
- [65] Bernhard Schölkopf, Robert C Williamson, Alex Smola, John Shawe-Taylor, and John Platt. 1999. Support vector method for novelty detection. *Advances in neural information processing systems* 12 (1999).
- [66] Jie Song and Yeye He. 2021. Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes. In *Proceedings of the 2021 International Conference on Management of Data*. 1678–1691.
- [67] Arun Swami, Sriram Vasudevan, and Joojay Huyn. 2020. Data Sentinel: A Declarative Production-Scale Data Validation Platform. In *ICDE*. IEEE, 1579–1590.
- [68] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2016. Effective and complete discovery of order dependencies via set-based axiomatization. *arXiv preprint arXiv:1608.06169* (2016).
- [69] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Expressiveness and complexity of order dependencies. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1858–1869.
- [70] Jian Tang, Zhixiang Chen, Ada Wai-Chee Fu, and David W Cheung. [n. d.]. Enhancing effectiveness of outlier detections for low density patterns. In *PAKDD 2002*. Springer.

- [71] Pei Wang and Yeye He. 2019. Uni-Detect: A Unified Approach to Automated Error Detection in Tables. In *Proceedings of the 2019 International Conference on Management of Data*. 811–828.
- [72] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. SCODED: Statistical Constraint Oriented Data Error Detection. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 845–860.
- [73] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (2012), 611–636.

Type	Metric	Description
Two-distribution	<i>EMD</i>	Earth Mover’s distance (Wasserstein metric) [59] between two numeric distributions
	<i>JS_div</i>	Jensen–Shannon divergence [28] between two numeric distributions
	<i>KL_div</i>	Kullback–Leibler divergence (relative entropy) [28] between two numeric distributions
	<i>KS_dist</i>	Two-sample Kolmogorov–Smirnov test [50] between two numeric distribution (using p-value)
	<i>Cohen_d</i>	Cohen’s d [27] that quantify the effect size between two numeric distributions
Single-distribution	<i>min</i>	the minimum value observed from a numeric column
	<i>max</i>	the maximum value observed from a numeric column
	<i>mean</i>	the arithmetic mean observed from a numeric column
	<i>median</i>	the median value observed from a numeric column
	<i>sum</i>	the sum of values observed from a numeric column
	<i>range</i>	the difference between max and min observed from a numeric column
	<i>row_count</i>	the number of rows observed from a numeric column
	<i>unique_ratio</i>	the fraction of unique values observed from a numeric column
	<i>complete_ratio</i>	the fraction of complete (non-null) values observed from a numeric column

Table 3: Statistical metrics used to generate DQ constraints for numerical data

Metric Type	Metric	Description
Two-distribution	L_1	L-1 distance [18] between two categorical distribution
	L_{inf}	L-infinity distance [18] between two categorical distributions
	<i>Cosine</i>	Cosine distance [34] between two categorical distributions
	<i>Chisquared</i>	Chi-squared test [30] using p-value between two categorical distributions
	<i>JS_div</i>	Jensen–Shannon divergence [28] between two categorical distributions
	<i>KL_div</i>	Kullback–Leibler divergence (relative entropy) [28] between two categorical distribution
	<i>Pat_L1</i>	L-1 distance between the pattern profiles extracted from two categorical distributions
	<i>Pat_Linf</i>	L-infinity distance between the pattern profiles extracted from two categorical distributions
	<i>Pat_Cosine</i>	Cosine distance between the pattern profiles extracted from two categorical distributions
	<i>Pat_Chisquare</i>	Chi-squared p-value between the pattern profiles extracted from two categorical distributions
	<i>Pat_JS_div</i>	Jensen–Shannon divergence of the pattern profiles extracted from two categorical distributions
	<i>Pat_kl_div</i>	Kullback–Leibler divergence of the pattern profiles extracted from two categorical distributions
Single-distribution	<i>str_len</i>	the average length of strings observed in a categorical column
	<i>char_len</i>	the average string length for values observed in a categorical column
	<i>digit_len</i>	the average number of digits in values observed from a categorical column
	<i>punc_len</i>	the average number of punctuation in values observed from a categorical column
	<i>row_count</i>	the number of rows observed from a categorical column
	<i>unique_ratio</i>	the fraction of unique values observed from a categorical column
	<i>complete_ratio</i>	the fraction of complete (non-null) values observed from a categorical column
	<i>dist_val_count</i>	the number of distinct values observed from a categorical column

Table 4: Statistical metrics used to generate DQ constraints for categorical data.

A DETAILS OF STATISTICAL METRICS

Table 3 and Table 4 give detailed descriptions of the statistical metrics used in AVH, which corresponds to simplified versions in Table 1 and Table 2, respectively.

B SYNTHETIC “TRAINING” DATA

We carefully reviewed the DQ literature and cataloged a list of 10 common types of DQ issues in pipelines, so that we can systematically synthesize data deviations that are due to DQ issues, which would help us to select the most salient “features” or DQ constraints that are sensitive in detecting common DQ deviations.

We enumerate the list of 10 different types of DQ issues below, as well as the parameters we use (to control deviations with different magnitudes). By injecting varying amounts of DQ issues into a given column C , we generates a total of 60 variations C' for each C (e.g., different fractions of values in C are replaced with nulls for the type of DQ issue “increased nulls”). Collectively, we denote this set of synthetically generated DQ issues on C as $D(C)$.

DQ Issue Type 1: Schema change. We replace $p\%$ (with $p = 1, 10, 100$) of values in a target column C for which we want to inject DQ variation, using values randomly sampled from a neighboring column of the same type. This is to simulate a “schema change”, where some fraction of values in a different column are either partially mis-aligned (e.g., due to a missing delimiter or bad parsing logic), or completely mis-aligned (e.g., due to extra or missing columns upstream introduced over time). Note that $p = 100$ corresponds to a complete schema-change, otherwise it is a partial schema change.

DQ Issue Type 2: Change of unit. To simulate a change in the unit of measurement, which is a common DQ issue (e.g., reported by Google in [14, 54] like discussed earlier), we synthetically multiply values in a numeric column C by $x10$, $x100$ and $x1000$.

DQ Issue Type 3: Casing change. To simulate possible change of code-standards (e.g., lowercase country-code to uppercase, as reported by Amazon [61]), we synthetically change $p\%$ fraction of values (with $p = 1, 10, 100$) in C , from lowercase to uppercase, and vice versa.

DQ Issue Type 4: Increased nulls. Since it is a sudden increase of null values such as NULL/empty-string/0 is common DQ issue, we sample $p\%$ values in a C (with $p = 1, 50, 100$), and replace them with empty-strings in the case of categorical attribute, and 0s in the case of numerical attribute.

DQ Issue Type 5: Change of data volume. Since a sudden increase/decrease of row counts can also be indicative of DQ issues [54], we up-sample values in C by a factor of $\times 2, \times 10$, or down-sample C with only 50%, 10% of the values.

DQ Issue Type 6: Change of data distributions. To simulate a sudden change of data distributions [14], we sorted all values in C first, then pick the first or last $p\%$ values as a biased sample and replace C , with $p = 10, 50$.

DQ Issue Type 7: Misspelled values by character perturbation. Typos and misspellings is another type of common DQ issue (e.g., “Missisipii” and “Mississippi”), frequently introduced by humans when manually entering data. To simulate this type of DQ issue, we randomly perturb $p\%$ of characters in C to a different character of the same type (e.g., $[0-9] \rightarrow [0-9]$, and $[a-z] \rightarrow [a-z]$), with $p = 1, 10, 100$.

DQ Issue Type 8: Extraneous values by character insertion. Sometimes certain values in a column C may be associated with extraneous characters that are not expected in clean data. To simulate this, for each value in C , we insert randomly generated characters with probability $p\%$, where $p = 10, 50$.

DQ Issue Type 9: Partially missing values by character deletion. Sometimes certain values in a column C may get partially truncated, due to issues in upstream logic. We simulate this by deleting characters for values in C with probability $p\%$, where $p = 10, 50$.

DQ Issue Type 10: Extra white-spaces by padding. We randomly insert leading or trailing whitespace for $p\%$ of values, where $p = 10, 50, 100$.

While we are clearly not the first people to report these aforementioned DQ issues, we are the first to systematically catalog them and synthetically generate such DQ variations, and are the first to use them as “training data” that guides a DQ algorithm to select the most salient DQ features specific to the characteristics of a column C . We release our generation procedures in [5], which can be used for future research.

C PATTERN GENERATION

In addition to use raw values from columns and compare their distributional similarity (e.g., using $L_1, L_{inf}, Cosine$, etc.), sometimes values in a column follows a specific pattern, for example, timestamp values like “2022-03-01 (Monday)”, currency values like “\$19.99”, zip-codes like “98052-1202”, etc. For such values, comparing distributions for raw values that are drawn from a large underlying domain induced by patterns (e.g., time-stamp), typically yields very small overlap/similarity because of the large space of possible values in the underlying domain. (This is in contrast to small categorical domains with a small number of possible values, where distributional similarity is usually high and more meaningful).

In AVH, we observe that the pattern strings for such pattern-induced domains is an orthogonal representation of values in a column, which gives another way to “describe” the column and

Algorithm 2: Pattern Generation

Input : A categorical column C
Output : The column pattern C'

```

1 foreach  $value \in C$  do
2   foreach  $same\ pattern\ of\ consecutive\ chars \in value$  do
3     if  $char \in [0-9]$  then
4       | Replace consecutive chars with a symbol  $\backslash d$ 
5     if  $char \in [A-Z]$  or  $[a-z]$  then
6       | Replace consecutive chars with a symbol  $\backslash l$ 
7     if  $char \in [punctuation]$  then
8       | Replace consecutive chars with a symbol  $-$ 
9    $C' \leftarrow C$ 
10 return  $C'$ 

```

Algorithm 3: Construct DQ constraints Q

Input : Metrics M , history $H = \{C_1, C_2, \dots\}$ of col C
Output : Constructed DQ constraints Q

```

1  $Q \leftarrow \emptyset$ 
2 foreach  $M \in M$  do
3    $M(H) \leftarrow \{M(C_1), M(C_2), \dots, M(C_K)\}$ 
4    $M(H) \leftarrow process-stationary(M(H)) // Algorithm\ 4$ 
5    $\mu \leftarrow mean\ of\ M(H), \sigma^2 \leftarrow variance\ of\ M(H)$ 
6   foreach  $\beta \in [\sigma, n\sigma]$ , increasing with a step-size  $s$ , do
7     |  $Q_i \leftarrow Q(M, C, \mu - \beta, \mu + \beta)$ 
8     |  $FPR(Q_i) \leftarrow calc-FPR(M, \beta) // Equation\ (5)-(7)$ 
9     |  $Q \leftarrow Q \cup Q_i$ 
10 return  $Q$ 

```

detect possible DQ deviations. For example, timestamp values like “2022-03-01 (Monday)” can be generalized to a pattern “ $\backslash d \backslash d \backslash d \backslash d \backslash d \backslash d \backslash d \backslash l \backslash l \backslash l \backslash l \backslash l \backslash l$ ”, currency values like “\$19.99” can be generalized to “ $\$ \backslash d \backslash d \backslash d$ ”, etc. Assuming that the format of the data is changed due to upstream DQ issue, e.g., currency values become mixed where some values have no currency-signs, or time-stamps becomes mixed with multiple formats of time-stamps, a distributional similarity of the pattern strings above provides a powerful way to “describe” the expected pattern distribution in a column, which makes it possible to catch DQ issues in columns whose underlying domains are pattern-related.

For the metrics that have a prefix “Pat_” in Table 2, we first generate pattern-strings for each value $v \in C$, by converting each character in v to a wildcard character following a standard $[0-9] \rightarrow \backslash d$ (for digits), $[a-zA-Z] \rightarrow \backslash l$ (for letters), and replace all punctuation as “-”. We then compute the same distributional similarity (e.g., $L_1, L_{inf}, Cosine$, etc.), just as regular distributional similarity metrics for raw string values. (Note that AVH is robust to a large space of DQ constraints, and can intelligently select the most salient features, such that for columns where pattern-based DQ is not a good DQ description, such pattern-based DQ constraints will not be selected automatically.)

D CONSTRUCT CONSTRAINTS: PSEUDOCODE

We show the pseudo code to construct DQ constraints in Algorithm 3. This procedure directly corresponds to Section 4.2

Algorithm 4: Time-series differencing for stationary

Input : $M(H) = \{M(C_1), M(C_2), \dots, M(C_K)\}$
Output : Processed $M'(H)$ that is stationary

```
1  $is\_stationary \leftarrow ADF(M(H));$  // Perform ADF test
2 if  $is\_stationary$  then
3 |   return  $M(H)$ 
4 else
5 |    $M'(H) \leftarrow \text{time-series-differencing}(M(H))$  // using
   |   first-order and seasonal differencing
6 |   return  $M'(H)$ 
```

E TIME-SERIES DIFFERENCING

Algorithm 4 gives an overview of the time-series differencing step, which we will expand and explain in this section. Details of this step can be found in Algorithm 5.

Algorithm 5: Time-series differencing for stationary (Details)

Input : $M(H) = \{M(C_1), M(C_2), \dots, M(C_K)\}$
Output : Processed stationary $M'(H)$

```
1  $is\_stationary \leftarrow ADF(M(H));$  // Perform ADF test
2 if  $is\_stationary$  then
3 |   return  $M(H)$ 
4 else
5 |   // Perform lag transformation without log transformation
   |   foreach  $lag \in [1, K - 1]$  do
6 |   |    $M'(H) \leftarrow lag\_transform(M(H), lag)$ 
7 |   |    $is\_stationary \leftarrow ADF(M'(H))$ 
8 |   |   if  $is\_stationary$  then
9 |   |   |   return  $M'(H)$ 
10 |   // Perform lag transformation with log transformation
   |   foreach  $lag \in [1, K - 1]$  do
11 |   |    $M'(H) \leftarrow lag\_transform\_with\_log(M(H), lag)$ 
12 |   |    $is\_stationary \leftarrow ADF(M'(H))$ 
13 |   |   if  $is\_stationary$  then
14 |   |   |   return  $M'(H)$ 
15 |   return None
```

Recall that time-series differencing aims to make time series stationary with static underlying parameters. We start by performing the ADF test to determine the stationarity of $M(H)$, and return $M(H)$ if it is already stationary. If it is not, we then perform lag-based transforms [33]. Given a sequence of $M(H) = \{M(C_1), M(C_2), \dots, M(C_K)\}$, a lag-based transform with $lag = l$ is defined as

$$M(H)^{lag=l} = \{M(C_{l+1}) - M(C_1), M(C_{l+2}) - M(C_2), \dots, M(C_{l+K}) - M(C_l)\}$$

Which performs a difference step for two events that are l time-steps away. Observe that such a differencing step handles cyclic data with periodic patterns (e.g., weekly user traffic data can be differenced away with $lag = 7$), as shown in Example 6 earlier.

For each $lag \in [1, K - 1]$, if the resulting $M(H)^{lag}$ is already stationary (passes the ADF test), we return the corresponding $M(H)^{lag}$ for the next stage for AVH to auto-program DQ (and remember the lag parameter to pre-process data arriving in the future).

If none of the lag parameter leads to a stationary time-series, we additionally perform a log transform on $M(H)$, which can better handle time-series with values that are orders of magnitude different. We repeat the same process as lag-only transforms like above, until we find a stationary time-series or we return None (in which case, the sequence $M(H)$ associated with this metric M will be ignored by downstream AVH due to its non-stationary nature. Also note that it is possible to perform additional second-order or third-order differencing, which we omit here).

F PROOF OF PROPOSITION 2

Proof Sketch: We prove this proposition using Cantelli's inequality [16]. Cantelli's inequality states that for a random variable X , there is a class of one-sided inequality in the form of $P(X - \mu \geq k\sigma) \leq \frac{1}{1+k^2}, \forall k \in \mathbb{R}^+$.

For metrics $M \in \{EMD, JS_div, KL_div, KS_dist, Cohen_d, L_1, L_{inf}, Cosine, Chisquared\}$, which are "distance-like" metrics, DQ constraints can be one-sided only, to guard against deviations with distances larger than usual, e.g., newly-arrived data whose distance from previous batches of data are substantially larger than is typically expected. (On the other hand, if the distance of new data and previous batches of data are smaller than usual, this shows more homogeneity and is typically not a source of concern).

For this reason, we can apply Cantelli's inequality for metrics $M \in \{EMD, JS_div, KL_div, KS_dist, Cohen_d, L_1, L_{inf}, Cosine, Chisquared\}$, with one-sided DQ. For such a metric M , let $M(C)$ be our random variable. Let $k = \frac{\beta}{\sigma}$. Replacing k with $\frac{\beta}{\sigma}$ above, we get $P(M(C) - \mu \geq \beta) \leq \frac{\sigma^2}{\sigma^2 + \beta^2}$. Note that $P(M(C) - \mu \leq \beta)$ is exactly our one-sided DQ for metrics with distance-like properties. We thus get $P(Q \text{ violated on } C) \leq \frac{\sigma^2}{\sigma^2 + \beta^2}$, which is equivalent to saying that the expected FPR of Q is no greater than $\frac{\sigma^2}{\sigma^2 + \beta^2}$. \square

G PROOF OF PROPOSITION 3

Proof Sketch: We prove this proposition using Central Limit Theorem (CLT) [16]. Recall CLT states that when independent random variables are summed up and normalized, it tends toward normal distribution. Metrics $M \in \{count, mean, str_len, char_len, digit_len, punc_len, complete_ratio\}$, can all be viewed as the sum of independent random variables (for example, $str_len, char_len, digit_len$ etc. are straightforward sum of these functions applied on individual cells; $count$ are the 0/1 sum for a random variable indicating tuple presence/not-presence, etc.). Such sums are then averaged over all cells in the same column C , which would tend to normal distributions per CLT. We can thus apply the tail bound of normal distributions, making it possible to apply tail bounds of normal distributions.

For $M \in \{count, mean, str_len, char_len, digit_len, punc_len, complete_ratio\}$, let $M(C)$ be our random variable. From tail bounds of normal distributions, we know $P(-k\sigma \leq M(C) - \mu \leq k\sigma) = erf(\frac{k}{\sqrt{2}})$ [6], where $erf(x)$ is the Gauss error function. Let $k = \frac{\beta}{\sigma}$. Replacing k with $\frac{\beta}{\sigma}$ above, we get $P(-\beta \leq M(C) - \mu \leq \beta) = erf(\frac{\beta}{\sqrt{2}\sigma}) = \frac{2}{\sqrt{\pi}} \int_0^{\frac{\beta}{\sqrt{2}\sigma}} e^{-t^2} dt$. Note that $P(-\beta \leq M(C) - \mu \leq$

β) is exactly $P(Q$ satisfied on C), thus we get $E[FPR(Q)] = 1 - \frac{2}{\sqrt{\pi}} \int_0^{\frac{\beta}{\sqrt{2}\sigma}} e^{-t^2} dt$. \square

H HARDNESS OF THE AVH PROBLEM

PROPOSITION 5. *The AVH problem in Equation (9)-Equation (11) is NP-hard. Furthermore, it cannot be approximated within a factor of $(1 - \frac{1}{e})$ under standard assumptions.*

Proof Sketch: We show the hardness using a reduction from the Maximum Coverage problem [51]. Recall that in Maximum Coverage, we are given a set of sets S , and the objective is to find a subset $S' \subseteq S$ such that the union of the elements covered by S' , $|\bigcup_{S_i \in S'} S_i|$, is maximized, subject to a cardinality constraint $|S'| \leq K$.

We show a polynomial time reduction from Maximum Coverage to AVH as follows. For any instance of Maximum Coverage with $S = \{S_i\}$, we construct the an AVH problem by converting each S_i into a DQ constraint Q_i , whose $FPR(Q_i)$ is unit cost 1, and recall $R(Q_i)$ is exactly the set of elements in S_i . If we could solve the corresponding AVH problem in polynomial-time, we would have solved the Maximum Coverage, thus contracting the hardness of Maximum Coverage.

Also note that through the construction above, the objective value of Maximum Coverage is identical to that of AVH. Thus we can use the inapproximation results from Maximum Coverage [51], to show that AVH cannot be approximated within a factor of $(1 - \frac{1}{e})$. \square

I PROOF OF PROPOSITION 4

Proof Sketch: We show that Algorithm 1 is a $(\frac{1}{2} - \frac{1}{2e})$ approximation algorithm for the AVH problem, which follows from the Budgeted Maximum Coverage problem [41]. Recall that in Budgeted Maximum Coverage problem, we are given a set of sets $S = \{S_i\}$, where each set S_i has a cost $c(S_i)$, and each element in sets has a weight $w(e_j)$, the objective is to find a subset $S' \subseteq S$ such that the weight of all elements covered by S' is maximized, subject to a budget constraint $\sum_{S_i \in S'} c(S_i) \leq B$. We show that for any instance of our AVH problem, it can be converted to Budgeted Maximum Coverage as follows. We convert each Q_i into a set S_i , and let the cost $c(S_i)$ be $FPR(Q_i)$. Furthermore, we convert the set of recall items into elements in Budgeted Maximum Coverage, and set the weight of each element to unit weight. Finally, we let the elements covered by S_i in Budgeted Maximum Coverage to be exactly the $R(Q_i)$ in AVH. The approximation ratio in Proposition 4 follows directly from the Theorem 3 of [41] now.

We note that there are an alternative algorithm with better approximation ratio $(1 - \frac{1}{e})$ [41], which however is of complexity $|\mathcal{Q}|^3$, where $|\mathcal{Q}|$ is the number of DQ constraints constructed from Algorithm 3. Because $|\mathcal{Q}|$ is at least in the hundreds, making the alternative very expensive in practice and not used in our system.

We also show that the solution S from our Algorithm 1 is a feasible solution of AVH, whose expected FPR is lower than δ . In order to see this, recall that we construct DQ $Q_i \in \mathcal{Q}$ and estimate each Q_i 's worst case $FPR(Q_i)$ following Proposition 1, 2, 3. Algorithm 1 ensures that $\sum_{Q_i \in S} FPR(Q_i) \leq \delta$. For the conjunctive

program $P(S)$ induced by S , the FPR of $P(S)$ follows the inequality $FPR(P(S)) \leq \sum_{Q_i \in S} FPR(Q_i)$, because the false-positives from $P(S)$, is produced by a union of the false-positives from each $Q_i \in S$. Combining this with $\sum_{Q_i \in S} FPR(Q_i) \leq \delta$, we get $FPR(P(S)) \leq \delta$. \square

J SENSITIVITY ANALYSIS

We perform extensive experiments to understand the sensitivity of our method.

Sensitivity to history length. Since AVH leverages a history of past pipeline executions, where the number of past executions likely has an impact on accuracy. Figure 6 shows the accuracy results for numerical and categorical data respectively, when {7, 14, 21, 28} days of historical data are available. Overall, having 28-day history leads to the best precision, though with 14 and 7-day histories AVH also produces competitive results. We highlight that unlike traditional ML methods that typically require more than tens of data points (Figure 2 suggests that even 30-day history is not sufficient for ML methods), AVH exploits the unique statistical properties of the underlying metrics (e.g., Chebyshev and CLT), and can work well even with limited data, which is a unique characteristic of AVH.

Sensitivity to target precision δ . Figure 7 shows the relationship between the target FPR δ parameter used in AVH (Equation (3)), and the real FPR observed on AVH results (we note that 1-FPR corresponds to the precision metric). On both numerical and categorical data, the real FPR increases slightly when a larger target FPR δ is used, showing the effectiveness of this knob δ in AVH. Also note that the real FPR is consistently lower than the target-FPR, likely due to the conservative nature of the statistical guarantees we leverage (Chebyshev and Cantelli's inequalities we use in Proposition 1 and 2 give worst-case guarantees).

K ABLATION STUDIES

We perform additional ablation studies, to understand the importance of different components used in AVH.

Effect of using single/two-distribution metrics. Recall that in AVH, we exploit both single-distribution and two-distribution metrics (in Table 1 and Table 2) to construct DQ programs. A key difference of the two types of metrics, is that computing two-distribution metrics (e.g., L_{inf} and L_1) would require both the current column C_K and its previous snapshot C_{K-1} (e.g., in $L_1(C_K, C_{K-1})$). This requires raw data from the previous run C_{K-1} to be kept around, which can be costly in production big-data systems. In contrast, single-distribution metrics (e.g., row_count and $unique_ratio$) can be computed on C_K and C_{K-1} separately, and we only need to keep the corresponding metrics from C_{K-1} without needing to keep the raw C_{K-1} , which makes single-distribution metrics a lot more efficient and inexpensive to use in AVH.

In Figure 8, we compare the full AVH (with both single- and two-distribution metrics), with AVH using only single-distribution metrics. Encouragingly, the latter variant produces comparable quality with the full AVH, likely because the large space of single-distribution metrics is already rich and expressive enough. This suggests that we can deploy AVH inexpensively without using two-distribution metrics, while still reaping most of the benefits.

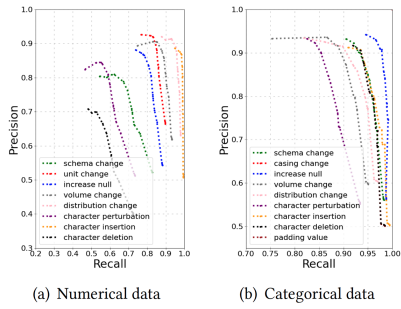


Figure 5: Results by DQ types

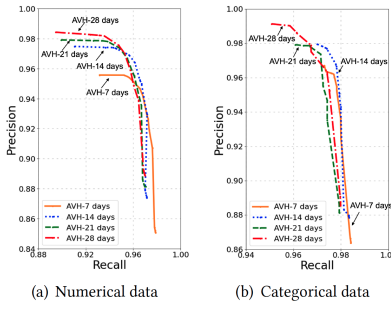


Figure 6: Sensitivity to history length

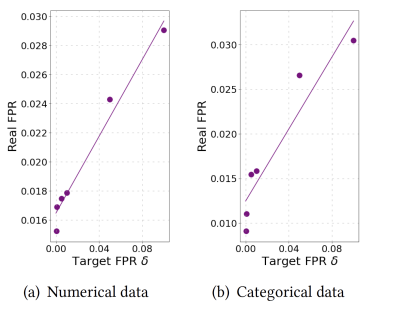


Figure 7: Sensitivity to target FPR δ

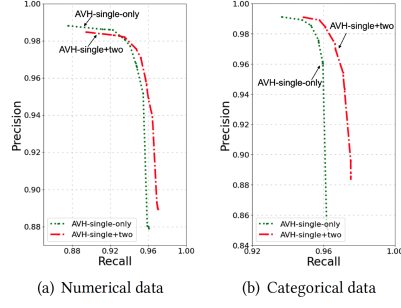


Figure 8: Effect of two-dist. metrics

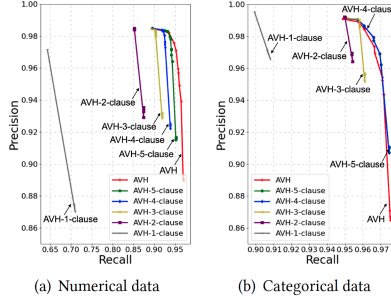


Figure 9: Effect of clause count limit

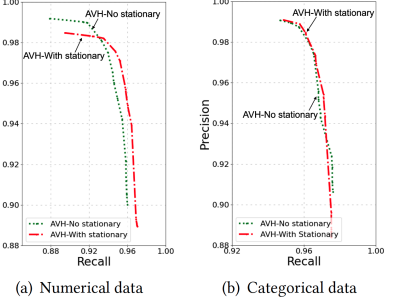


Figure 10: Using stationary processing

Effect of limiting the number of DQ clauses. We also study the number of DQ clauses that AVH generates, because intuitively, the more clauses it generates, the more expressive the DQ programs become, at the cost of human explainability/interpretability (there is a setting in AVH that engineers can review and approve auto-suggested DQ programs). We report that on numerical data in the Real benchmark, the median/mean of the number of clauses AVH generates is 3 and 2.62, respectively; on categorical data the median/mean is 2 and 2.15, respectively. We believe this shows that the programs AVH generates are not only effective but also simple/understandable. In Figure 9 we impose an artificial limit on the number of clauses that AVH can generate (in case better readability is required). We observe a drop in performance when only 1 or 2 clauses are allowed, but the performance hit becomes less significant if we allow 3 clauses.

Effect of stationary processing. Since we use stationarity test and stationary processing for statistics that are time-series (Section 4.2), in Figure 10 we study its effect on overall quality. We can see that for numerical data, stationary processing produces a noticeable improvement, which however is less significant on categorical data.

L MANUAL REVIEW OF PIPELINE DATA

Based on our conversations with data engineers and data owners, the data tables collected from production data pipelines used in our Real benchmark (describe in Section 5.1) are production-quality and likely free of DQ issues, because these files are of high business impact with many downstream dependencies, such that if they had any DQ issues they would have already been flagged and fixed by data engineers.

In order to be sure, we randomly sampled 50 categorical and 50 numerical data columns, and manually inspected these sample data

in the context of their original data tables, across 60 snapshots, to confirm the quality of the benchmark data. We did not find any DQ issues based on our manual inspection. We also perform a hypothesis test based on the manual analysis, with H_0 stating that over 3% of data has DQ issues. Our inspection above rejects the null hypothesis (p-level=0.05), indicating that it is highly unlikely that the benchmark data has DQ issues, which is consistent with the assessment from data owners, and confirms the quality of the data used in the Real benchmark.

During our conversations with data engineers, we were pointed to three known DQ incidents, which we collect and use as test cases to study AVH’s coverage. AVH was able to detect all such known DQ cases based on historical data. Figure 11 shows such an example that is intuitive to see. Here each file is an output table (in csv format) produced by a daily recurring pipeline. As can be seen in the figure, for the file produced on “2019-01-19”, the file size (and thus row-count) is much larger than the days before and after “2019-01-19” (21KB vs. 4KB). While small in scale, we believe this study on user-provided data further confirms the effectiveness of AVH.

AllRes_AnchorParity_20190109.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190110.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190111.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190112.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190113.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190114.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190115.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190116.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190117.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190118.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190119.tsv	6/16/2021 1:03 PM	TSV File	21 KB
AllRes_AnchorParity_20190120.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190121.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190122.tsv	6/16/2021 1:03 PM	TSV File	4 KB
AllRes_AnchorParity_20190123.tsv	6/16/2021 1:03 PM	TSV File	4 KB

Figure 11: Example of a real DQ issue flagged by AVH.