Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes

Jie Song*
University of Michigan
jiesongk@umich.edu

Yeye He
Microsoft Research
yeyehe@microsoft.com

ABSTRACT

Complex data pipelines are increasingly common in diverse applications such as BI reporting and ML modeling. These pipelines often recur regularly (e.g., daily or weekly), as BI reports need to be refreshed, and ML models need to be retrained. However, it is widely reported that in complex production pipelines, upstream data feeds can change in unexpected ways, causing downstream applications to break silently that are expensive to resolve.

Data validation has thus become an important topic, as evidenced by notable recent efforts from Google and Amazon, where the objective is to catch data quality issues early as they arise in the pipelines. Our experience on production data suggests, however, that on string-valued data, these existing approaches yield high false-positive rates and frequently require human intervention. In this work, we develop a corpus-driven approach to auto-validate *machine-generated data* by inferring suitable data-validation "patterns" that accurately describe the underlying data-domain, which minimizes false-positives while maximizing data quality issues caught. Evaluations using production data from real data lakes suggest that AUTO-VALIDATE is substantially more effective than existing methods. Part of this technology ships as an AUTO-TAG feature in MICROSOFT AZURE PURVIEW.

1 INTRODUCTION

Complex data pipelines and data flows are increasingly common in today's BI, ETL and ML systems (e.g., in Tableau [15], Power BI [13], Amazon Glue [2], Informatica [11], Azure ML [4], etc.). These pipelines typically *recur regularly* (e.g., daily), as BI reports need to be refreshed regularly [30, 61], data warehouses need to be updated frequently [64], and ML models need to be retrained continuously (to prevent model degradation) [21, 54].

However, it is widely recognized (e.g., [21, 39, 54, 59–61]) that in recurring production data pipelines, over time upstream data feeds can often change in unexpected ways. For instance, over time, data columns can be added to or removed in upstream data, creating *schema-drift* [21, 54, 61]. Similarly, *data-drift* [54] is also common, as the formatting standard of data values can change silently (e.g., from "en-us" to "en-US", as reported in [54]); and invalid values can also creep in (e.g., "en-99", for unknown locale).

```
// 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
```

Figure 1: Example code snippet to describe expected data-values, using declarative constraints in Amazon Deequ.

Such schema-drift and data-drift can lead to quality issues in downstream applications, which are both hard to detect (e.g., modest model degradation due to unseen data [54]), and hard to debug (finding root-causes in complex pipelines can be difficult). These silent failures require substantial human efforts to resolve, increasing the cost of running production data pipelines and decreasing the effectiveness of data-driven system [21, 54, 59, 61].

Data Validation by Declarative Constraints. Large tech companies operating large numbers of complex pipelines are the first to recognize the need to catch data quality issues early in the pipelines. In response, they pioneered a number of "data validation" tools, including Google's TensorFlow Data Validation (TFDV) [9, 21] and Amazon's Deequ [1, 61], etc. These tools develop easy-to-use *domain specific languages* (DSLs), for developers and data engineers to write declarative *data constraints* that describe how "normal" data should look like in the pipelines, so that any unexpected deviation introduced over time can be caught early as they arise.

Figure 1 shows an example code snippet[1] from Amazon Deequ [61] to specify data validation constraints on a table. In this case, it is declared that values in the column "review_id" need to be unique, column "marketplace" is complete (with no NULLs), values in "marketplace" need to be in a fixed dictionary {"US", "UK", "DE", "JP", "FR"}, etc. Such constraints will be used to validate against data arriving in the future, and raise alerts if violations are detected. Google's TFDV uses a similar declarative approach with a different syntax.

These approaches allow users to provide *high-level specifications* of data constraints, and is a significant improvement over *low-level* assertions used in ad-hoc validation scripts (which are are hard to program and maintain) [1, 9, 21, 61, 63].

Automate Data Validation using Inferred Constraints. While declarative data validation clearly improves over ad-hoc scripts, and is beneficial as reported by both Google and Amazon [21, 61], the need for data engineers to manually write data constraints *one-column-at-a-time* (e.g., in Figure 1) still makes it time-consuming and hard to scale. This is especially true for complex production data, where a single data feed can have hundreds of columns requiring substantial manual work. As a result, experts today can only afford to write validation rules for part of their data (e.g., important

---

*Work done at Microsoft Research.

[1]https://aws.amazon.com/blogs/big-data/test-data-quality-at-scale-with-deequ/

columns). Automating the generation of data validation constraints has become an important problem.

The main technical challenge here is that validation rules have to be inferred using data values *currently observed* from a pipeline, but the inferred rules have to apply to future data that *cannot yet be observed* (analogous to training-data/testing-data in ML settings where test-data cannot be observed). Ensuring that validation rules *generalize* to unseen future data is thus critical.

We should note that existing solutions already make efforts in this direction – both Google TFDV and Amazon Deequ can scan an existing data feed, and automatically suggest validation constraints for human engineers to inspect and approve. Our analysis suggests that the capabilities of these existing solutions are still rudimentary, especially on *string-valued data columns*. For instance, for the example C_1 in Figure 2(a) with date strings in `Mar 2019`, TFDV would infer a validation rule[2] requiring all future values in this column to come from a fixed dictionary with existing values in C_1: {" `Mar 01 2019`", ..." `Mar 30 2019`" }. This is too restrictive for data validation, as it can trigger false-alarms on future data (e.g. values like "`Apr 01 2019`"). Accordingly, TFDV documentation does not recommend suggested rules to be used directly, and "*strongly advised developers to review the inferred rule and refine it as needed*"[3].

When evaluated on production data (without human intervention), we find TFDV produces false-alarms on over 90% string-valued columns. Amazon's Deequ considers distributions of values but still falls short, triggering false-alarms on over 20% columns. Given that millions of data columns are processed daily in production systems, these false-positive rates can translate to millions of false-alarms, which is not adequate for automated applications.

Auto-Validate string-valued data using patterns. We in this work aim to infer accurate data-validation rules for *string-valued data*, using regex-like patterns. We perform an in-depth analysis of production data in real pipelines, from a large Map-Reduce-like system used at Microsoft [53, 71], which hosts over 100K production jobs each day, and powers popular products like Bing Search and Office. We crawl a sample of 7M columns from the data lake hosting these pipelines and find string-valued columns to be prevalent, accounting for 75% of columns sampled.

Figure 3 shows 7 example string-valued columns from the crawl. A key characteristic common to many such columns, is that these columns have *homogeneous* and *machine-generated* data values, which exhibit distinct "patterns" that encode specific semantic meanings. In our examples, these include knowledge-base entity-id [6] (used by Bing Search), online-ads delivery status (used by search ads), time-stamps in proprietary formats (used by search ads), etc. We note that these are not one-off patterns occurring just once or twice – each pattern in Figure 3 shows up in at least 5000 data columns in our crawl, suggesting that these are widely-used concepts. Similar data are likely common in other industries and domains (e.g., for pharmaceutical or financial companies [62]).

In this work, we focus on these string-valued columns that have homogeneous machine-generated data values. Specifically, if we could infer suitable patterns to describe the underlying "domain" of

the data (defined as the space of all valid values), we can use such patterns as validation rules against data in the future. For instance, the pattern "`<letter>{3} <digit>{2} <digit>{4}`" can be used to validate C_1 in Figure 2(a), and "`<digit>+/<digit>{2}/<digit>{4} <digit>+:<digit>{2}:<digit>{2} <letter>{2}`" for C_2 in Figure 2(b).

Our analysis on a random sample of 1000 columns from our production data lake suggests that around 67% string-valued columns have homogeneous machine-generated values (like shown in Figure 3), whose underlying data-domains are amenable to pattern-based representations (the focus of this work). The remaining 33% columns have *natural-language (NL) content* (e.g., company names, department names, etc.), for which pattern-based approaches would be less well-suited. We should emphasize that for large data-lakes and production data pipelines, machine-generated data will likely dominate (because machines are after all more productive at churning out large volumes of data than humans producing NL data), making our proposed technique widely applicable.

Conceptually, there are many ways to "generalize" a column of data into patterns. For a simple column of date-time strings like in Figure 5, and using a standard generalization hierarchy (Figure 4), one could generate over 3 billion patterns. For example, the first part (digit "`9`" for month) alone can be generalized in 7 different ways below, also shown in the top-right box of Figure 5. The cross-product of these options at each position creates over 3 billion possible patterns for this simple column.

The challenge for data-validation, is to select suitable patterns from this large space given only observed values in C, in anticipation of valid values from the same "domain" that may arrive in the future. The key is to not use overly restrictive patterns (as in data-profiling), which would yield many false-positive alerts. Conversely overly general patterns (like the trivial "`.*`") should also not be used, as they would fail to detect any data quality issues. In effect, we want to find patterns that balance two conflicting goals: (1) reduce false-alarms (or improve detection precision); and (2) catch as many issues as possible (or improve detection recall).

A corpus-driven approach to pattern-inference. Intuitively, the pattern-inference problem is hard if we only look at one input column C – for the date-time examples in Figure 2, we as humans know what patterns are suitable; but for the examples in Figure 3 drawn from proprietary domains, even humans may find it hard to determine suitable validation patterns, and would need to seek "additional evidence", such as other "similar-looking columns".

Following this intuition, we in this work formulate pattern-inference as optimization problems, by leveraging a large corpus of related tables **T** (e.g. data produced by production pipelines and dumped in the same enterprise data lake). Our inference algorithms leverage columns "similar" to C in **T**, to reason about patterns that may be ideal for data-validation.

Large-scale evaluations on production data suggest that Auto-Validate produces constraints that are substantially more accurate than existing methods. Part of this technology ships as an Auto-Tag feature for custom-classification in Microsoft Azure Purview [5].

2 AUTO-VALIDATE (BASIC VERSION)

We start by describing a basic-version of Auto-Validate for ease of illustration, before going into more involved algorithm variants that use vertical-cuts (Section 3) and horizontal-cuts (Section 4).

---

[2]This was initially tested on TFDV version 0.15.0, the latest version available in early 2020. We tested it again at the time of publication in March 2021 using the latest TFDV version 0.28.0, and observed the same behavior for this test case.
[3]https://www.tensorflow.org/tfx/data_validation/get_started

Figure 2: Two example data columns C_1 and C_2. The top portion of each column shows data in the current feed that can be observed, while the bottom portion shows values that will arrive in the future.

Figure 3: Example columns crawled from a production enterprise data lake. Each column has a distinctive data pattern in proprietary formats, encoding specific meanings (the first column is anonymized for review). These are all common patterns found in at least 5000 columns.

Figure 4: Example string generalization hierarchy.

Figure 5: Possible ways to generalize a column of date-time strings, using the hierarchy in Figure 4.

2.1 Preliminary: Pattern Language

Since we propose to validate data by patterns, we first briefly describe the pattern language used. We note that this is fairly standard (e.g., similar to ones used in [58]) and not the focus of this work.

Figure 4 shows a generalization hierarchy used, where leaf-nodes represent the English alphabet, and intermediate nodes (e.g., <digit>, <letter>) represent *tokens* that values can generalize into. A pattern is simply a sequence of (leaf or intermediate) tokens, and for a given value v, this hierarchy induces a space of all patterns consistent with v, denoted by $\mathbf{P}(v)$. For instance, for a value $v =$ "9:07", we can generate $\mathbf{P}(v) = \{$"<digit>:<digit>{2}", "<digit>+:<digit>{2}", "<digit>:<digit>+", "<num>:<digit>+", "9:<digit>{2}", ...$\}$, among many other options.

Given a column C for which patterns need to be generated, we define the space of *hypothesis-patterns*, denoted by $\mathbf{H}(C)$, as the set of patterns consistent with all values $v \in C$, or $\mathbf{H}(C) = \cap_{v \in C} \mathbf{P}(v) \setminus$ ".*" (note that we exclude the trivial ".*" pattern as it is trivially consistent with all C). Here we make an implicit assumption that values in C are homogeneous and drawn from the same domain, which is generally true for *machine-generated data* from production pipelines (like shown in Figure 3)[4]. Note that the assumption is

---

[4]Our empirical sample in an enterprise data lake suggests that 87.9% columns are homogeneous (defined as drawn from the same underlying domain), and 67.6% are homogeneous with machine-generated patterns.

used to only simplify our discussion of the basic-version of Auto-Validate (this section), and will later be relaxed in more advanced variants (Section 4).

We call $\mathbf{H}(C)$ hypothesis-patterns because as we will see, each $h(C) \in \mathbf{H}(C)$ will be tested like a "hypothesis" to determine if h is a good validation pattern. We use h and $h(C)$ interchangeably when the context is clear.

Example 1. Given the column C shown in Figure 5, and the hierarchy in Figure 4, the space of hypothesis patterns $\mathbf{H}(C)$ is shown at the top of Figure 5.

We use an in-house implementation to produce patterns based on the hierarchy, which can be considered as a variant of well-known pattern-profiling techniques like [58]. Briefly, our pattern-generation works in two steps. In the first step, it scans each cell value and emits *coarse-grained* tokens (e.g., `<num>` and `<letter>+`), without specializing into *fine-grained* tokens (e.g., `<digit>{1}` or `<letter>{2}`). For each value in Figure 5, for instance, this produces the same `<num>/<num>/<num> <num>:<num>:<num> <letter>+`. Each coarse-grained pattern is then checked for coverage so that only patterns meeting a coverage threshold are retained. In the second step, each coarse-grained pattern is specialized into fine-grained patterns (examples are shown in Figure 5), as long as the given threshold is met. More details of this can be found in a full version of this paper [42].

We emphasize that the proposed Auto-Validate framework is not tied to specific choices of hierarchy/pattern-languages – in fact, it is entirely orthogonal to such choices, as other variants of languages are equally applicable in this framework.

2.2 Intuition: Goodness of Patterns

The core challenge in Auto-Validate, is to determine whether a pattern $h(C) \in \mathbf{H}(C)$ is suitable for data-validation. Our idea here is to leverage a corpus of related tables $\mathbf{T}$ (e.g., drawn from the same enterprise data lake, which stores input/output data of all production pipelines). Intuitively, a pattern $h(C) \in \mathbf{H}(C)$ is a "good" validation pattern for C, if it accurately describes the underlying "domain" of C, defined as the space of all valid data values. Conversely, $h(C)$ is a "bad" validation pattern, if:

(1) $h(C)$ does not capture all valid values in the underlying "domain" of C;

(2) $h(C)$ does not produce sufficient matches in $\mathbf{T}$.

Both are intuitive requirements – (1) is desirable, because as we discussed, the danger in data-validation is to produce patterns that do not cover all valid values from the same domain, which leads to false alarms. Requirement (2) is also sensible because we want to see sufficient evidence in $\mathbf{T}$ before we can reliably declare a pattern $h(C)$ to be a common "domain" of interest (seeing a pattern once or twice is not sufficient).

We note that both of these two requirements can be reliably tested using $\mathbf{T}$ alone and without involving humans. Testing (2) is straightforward using standard pattern-matches. Testing (1) is also feasible, under the assumption that most data columns in $D \in \mathbf{T}$ contain homogeneous values drawn from the same underlying domain, which is generally true for "machine-generated" data from production pipelines. (Homogeneity is less likely to hold on other types of data, e.g., data manually typed in by human operators).

We highlight the fact that we are now dealing with two types of columns that can be a bit confusing – the first is the input column C for which we need to generate validation patterns h, and the second is columns $D \in \mathbf{T}$ from which we draw evidence to determine the goodness of h. To avoid confusion, we will follow the standard of referring to the first as the *query column*, denoted by C, and the second as *data columns*, denoted by D.

We now show how to test requirement (1) using $\mathbf{T}$ below.

Example 2. The left of Figure 6 shows a query column C for which validation patterns need to be generated. A few hypothesis-patterns in $\mathbf{H}(C)$ are listed in the middle. In this example, we intuitively know that $h_1(C)$ and $h_2(C)$ are "bad" patterns since they are too "narrow" and would lead to false-alarms, while $h_5(C)$ is a "good" validation pattern that suitably generalizes the underlying domain.

We show that such determinations can be inferred using $\mathbf{T}$. Specifically, $h_1(C)$ is a "bad" pattern based on $\mathbf{T}$, because we can find many columns like $D \in \mathbf{T}$ shown on the right of Figure 6 that are "impure" – these columns contain values that match $h_1(C)$, as well as values that do not (e.g., "9/12/2019 12:01:32 PM", where the "PM" part does not match $h_1(C)$). A large number of "impure" columns like this would indicate that $h_1(C)$ is not a good pattern to describe the underlying domain because if it were, we should not see many impure columns like $D \in \mathbf{T}$ given homogeneity.

Similarly, we can show that $h_2(C)$ is not a good validation pattern, for using $h_2(C)$ to describe the domain would again make many columns like D "impure" (values like "10:02:20 AM" are inconsistent with $h_2(C)$ since they have two-digit hours, whereas $h_2(C)$ uses a single `<digit>` for hours), suggesting $h_2(C)$ is also too narrow.

Using the ideal pattern $h_5(C)$ to describe the domain, on the other hand, would yield few "impure" columns in $\mathbf{T}$.

Intuitively, we can use the *impurity* of pattern h on data columns $D \in \mathbf{T}$, to infer whether h is a good validation pattern for the domain, defined as follows.

Definition 1. The *impurity* of a data column $D \in \mathbf{T}$, relative to a given hypothesis pattern h, is defined as:

$$\text{Imp}_D(h) = \frac{|\{v|v \in D, h \notin \mathbf{P}(v)\}|}{|\{v|v \in D\}|} \quad (1)$$

This definition is intuitive – we measure impurity as the fraction of values in D not matching h.

Example 3. In Figure 6, $\text{Imp}_D(h_1)$ can be calculated as $\frac{2}{12}$, since the last 2 values (with "PM") out of 12 do not match h_1.

Similarly, $\text{Imp}_D(h_2)$ can be calculated as $\frac{8}{12}$, since the last 8 values in D (with two-digit hours) do not match h_2.

Finally, $\text{Imp}_D(h_5)$ is $\frac{0}{12}$, since all values in D match h_5.

We note that if $h(C)$ is used to validate a data column D (in the same domain as C) arriving in the future, then $\text{Imp}_D(h)$ directly corresponds to expected false-positive-rate (FPR):

Definition 2. The expected *false-positive-rate* (FPR) of using pattern $h(C)$ to validate a data column D drawn from the same domain as C, denoted by $\text{FPR}_D(h)$, is defined as:

$$\text{FPR}_D(h) = \frac{\text{FP}_D(h)}{\text{TN}_D(h) + \text{FP}_D(h)} \quad (2)$$

Where $\text{FP}_D(h)$ and $\text{TN}_D(h)$ are the numbers of false-positive detection and true-negative detection of h on D, respectively. Note
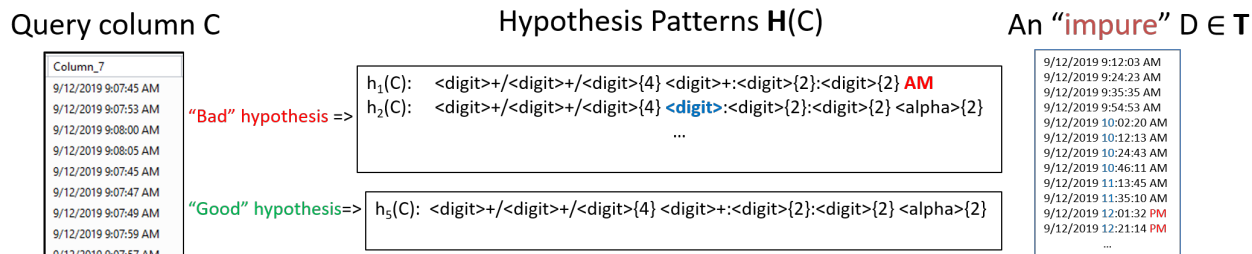
Figure 6: Leverage corpus T to determine "good" vs. "bad" hypothesis patterns, for a given query column C.

that since D is from the same domain as C, ensuring that $\text{TN}_D(h) + \text{FP}_D(h) = |D|$, $\text{FPR}_D(h)$ can be rewritten as:

$$\text{FPR}_D(h) = \frac{|\{v|v \in D, h \notin \mathbf{P}(v)\}|}{|\{v|v \in D\}|} = \text{Imp}_D(h) \quad (3)$$

Thus allowing us to estimate $\text{FPR}_D(h)$ using $\text{Imp}_D(h)$.

EXAMPLE 4. Continue with Example 3, it can be verified that the expected FPR of using h as the validation pattern for D, directly corresponds to the impurity $\text{Imp}_D(h)$ – e.g., using h_1 to validate D has $\text{FPR}_D(h_1) = \text{Imp}_D(h_1) = \frac{2}{12}$; while using h_5 to validate D has $\text{FPR}_D(h_5) = \text{Imp}_D(h_5) = 0$, etc.

Based on $\text{FPR}_D(h)$ defined for one column $D \in \mathbf{T}$, we can in turn define the estimated FPR on the entire corpus $\mathbf{T}$, using all column $D \in \mathbf{T}$ where some value $v \in D$ matches h:

DEFINITION 3. Given a corpus $\mathbf{T}$, we estimate the FPR of pattern h on $\mathbf{T}$, denoted by $\text{FPR}_\mathbf{T}(h)$, as:

$$\text{FPR}_\mathbf{T}(h) = \underset{D \in \mathbf{T}, v \in D, h \in \mathbf{P}(v)}{\text{avg}} \text{FPR}_D(h) \quad (4)$$

EXAMPLE 5. Continue with the h_5 in Example 4. Suppose there are 5000 data columns $D \in \mathbf{T}$ where some value $v \in D$ matches h_5. Suppose 4800 columns out of the 5000 have $\text{FPR}_D(h_5) = 0$, and the remaining 200 columns have $\text{FPR}_D(h_5) = 1\%$. The overall $\text{FPR}_\mathbf{T}(h_5)$ can be calculated as $\frac{200*1\%}{5000} = 0.04\%$, using Equation (4).

We emphasize that a low FPR is critical for data-validation, because given a large number of columns in production systems, even a 1% FPR translates into many false-alarms that require human attention to resolve. FPR is thus a key objective we minimize.

2.3 Problem Statement

We now describe the basic version of AUTO-VALIDATE as follows. Given an input query column C and a background corpus $\mathbf{T}$, we need to produce a validation pattern $h(C)$, such that $h(C)$ is expected to have a low FPR but can still catch many quality issues. We formulate this as an optimization problem. Specifically, we introduce an FPR-Minimizing version of Data-Validation (FMDV), defined as:

$$(\text{FMDV}) \quad \min_{h \in \mathbf{H}(C)} \text{FPR}_\mathbf{T}(h) \quad (5)$$

$$\text{s.t. } \text{FPR}_\mathbf{T}(h) \leq r \quad (6)$$

$$\text{Cov}_\mathbf{T}(h) \geq m \quad (7)$$

Where $\text{FPR}_\mathbf{T}(h)$ is the expected FPR of h estimated using $\mathbf{T}$, which has to be lower than a given target threshold r (Equation (6)); and $\text{Cov}_\mathbf{T}(h)$ is the coverage of h, or the number of columns in $\mathbf{T}$ that match h, which has to be greater than a given threshold m (Equation (7)). Note that these two constraints directly map to the two "goodness" criteria in Section 2.2.

The validation pattern h we produce for C is then the minimizer of FMDV from the hypothesis space $\mathbf{H}(C)$ (Equation (5)). We note

that this can be interpreted as a conservative approach that aims to find a "safe" pattern $h(C)$ with minimum FPR.

EXAMPLE 6. Suppose we have FMDV with a target FPR rate r no greater than 0.1%, and a required coverage of at least $m = 100$. Given the 3 example hypothesis patterns in Figure 6, we can see that patterns h_1 and h_2 are not feasible solutions because of their high estimated FPR. Pattern h_5 has an estimated FPR of 0.04% and a coverage of 5000, which is a feasible solution to FMDV.

LEMMA 1. *In a simplified scenario where each column in $\mathbf{T}$ has values drawn randomly from exactly one underlying domain, and each domain in turn corresponds to one ground-truth pattern. Then given a query column C for which a validation pattern needs to be generated, and a corpus $\mathbf{T}$ with a large number (n) of columns generated from the same domain as C, with high probability (at least $1 - (\frac{1}{2})^n$), the optimal solution of FMDV for C (with $r = 0$) converges to the ground-truth pattern of C.*

Intuitively, this result holds under the simplifying assumptions, because when given a sufficiently large number of n columns in $\mathbf{T}$ from the same domain as C, any candidate patterns that "under-generalizes" the domain will be pruned away due to FPR violations, leaving the ground-truth pattern to be the optimal solution. More details of the result can be found in [42].

We also explored alternative formulations, such as coverage-minimizing data-validation (CMDV), where we minimize coverage $\text{Cov}_\mathbf{T}(h)$ in the objective function in place of $\text{FPR}_\mathbf{T}(h)$ as in FMDV. We omit details in the interest of space, but will report that the conservative FMDV is more effective in practice.

A related variant of FMDV, is to automatically "tag" data in data lakes with semantic labels based on inferred data patterns (with the key difference being that the false-positives in FMDV become false-negatives in data-tagging). We describe this dual version of FMDV in [34], which is the basis of an "auto-tag" feature that ships in Azure Purview [5].

2.4 System Architecture

While FMDV is simple overall, a naive implementation would require a full scan of $\mathbf{T}$ to compute $\text{FPR}_\mathbf{T}(h)$ and $\text{Cov}_\mathbf{T}(h)$ for each hypothesis h. This is both costly and slow considering the fact that $\mathbf{T}$ is typically large in enterprise data lakes (e.g., in terabytes), where a full-scan can take hours. This is especially problematic for user-in-the-loop validation like in Google's TFDV, where users are expected to review/modify suggested validation rules and interactive response time is critical.

We thus introduce an offline index with summary statistics of $\mathbf{T}$, so that at online time and given a new query column C, we
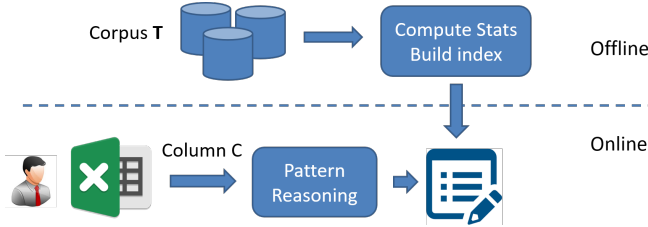
Figure 7: Architecture of the AUTO-VALIDATE System.

can evaluate hypothesis patterns h efficiently using only the index, without needing to scan **T** again.

The architecture of our system can be seen in Figure 7.

Offline stage. In the offline stage, we perform one full scan of **T**, enumerating all possible patterns for each $D \in \mathbf{T}$ as $\mathbf{P}(D) = \cup_{v \in D} \mathbf{P}(v)$, or the union of patterns for all $v \in D$ (where **P** is the generalization hierarchy in Section 2.1).

Note that the space of $\mathbf{P}(D)$ can be quite large, especially for "wide" columns with many tokens as discussed in Figure 5. To limit $\mathbf{P}(D)$, in the basic version we produce $\mathbf{P}(D)$ as $\cup_{v \in D, t(v) < \tau} \mathbf{P}(v)$, where $t(v)$ is the number of tokens in v (defined as the number of consecutive sequences of letters, digits, or symbols in v), and τ is some fixed constant (e.g., 8 or 13) to make $\mathbf{P}(D)$ tractable. We will describe how columns wider than τ can be safely skipped in offline indexing without affecting result quality, using a vertical-cut technique (in Section 3).

For each pattern $p \in \mathbf{P}(D)$, we pre-compute the local impurity score of p on D as $\mathrm{Imp}_D(p)$, since p may be hypothesized as a domain pattern for some query column C in the future, for which this D can provide a piece of local evidence.

Let $\mathbf{P}(\mathbf{T}) = \{p | p \in \mathbf{P}(D), D \in \mathbf{T}\}$ be the space of possible patterns in **T**. We pre-compute $\mathrm{FPR}_\mathbf{T}(p)$ for all possible $p \in \mathbf{P}(\mathbf{T})$ on the entire **T**, by aggregating local $\mathrm{Imp}_D(p)$ scores using Equation (3) and (4). The coverage scores $\mathrm{Cov}_\mathbf{T}(p)$ of all p can be pre-computed similarly as $\mathrm{Cov}_\mathbf{T}(p) = |\{D | D \in \mathbf{T}, v \in D, p \in \mathbf{P}(v)\}|$.

The result from the offline step is an index for lookup that maps each possible $p \in \mathbf{P}(\mathbf{T})$, to its pre-computed $\mathrm{FPR}_\mathbf{T}(p)$ and $\mathrm{Cov}_\mathbf{T}(p)$ values. We note that this index is many orders of magnitude smaller than the original **T** – e.g., a 1TB corpus **T** yields an index less than 1GB in our experiments.

Online stage. At online query time, for a given query column C, we enumerate $h \in \mathbf{H}(C)$, and use the offline index to perform a lookup to retrieve $\mathrm{FPR}_\mathbf{T}(p)$ and $\mathrm{Cov}_\mathbf{T}(p)$ scores for FMDV, without needing to scan the full **T** again. This indexing approach reduces our latency per query-column C from hours to tens of milliseconds, which makes it possible to do interactive, human-in-the-loop verification of suggested rules, as we will report in experiments.

3 AUTO-VALIDATE (VERTICAL CUTS)

So far we only discussed the basic-version of AUTO-VALIDATE. We now describe formulations that handle more complex and real situations: (1) *composite* structures in query columns (this section); and (2) "dirty" values in query columns that violate the homogeneity assumption (Section 4).

Figure 8 shows a real column with composite structures that are concatenated from *atomic domains*. Intuitively, we can see that it consists of at least 4 "sub-domains", a floating-number ("0.1"), followed by two time-stamps, followed by a status message ("OnBooking").

We observe that such columns are common in machine-generated data, where many are complex with over 10 sub-domains.

Such complex columns pose both quality and scalability challenges. In terms of quality, if we were to directly test complex patterns against **T** like in FMDV, we may not be able to see many exact matches in **T**, because the complex patterns in C may be concatenated in ad-hoc ways and are sparsely represented in **T**. As a result, we may fail to find a feasible solution to FMDV, because of the coverage requirement in Equation (7).

The second challenge is scalability. As column C become "wider" with more tokens/segments, the space of possible patterns $\mathbf{P}(C)$ that we will have to enumerate grows exponentially in the number of tokens. This is an issue both at online query time for query column C, as well as at offline indexing time to enumerate $\mathbf{P}(D)$ for data column $D \in \mathbf{T}$ – recall that the date-time example in Figure 5 already produces over 3 billion patterns; for the column in Figure 8 the space of patterns is simply impractically large to enumerate. In the offline indexing step in Section 2.4, we discussed that we intentionally omit wide columns with more than τ tokens to make pattern enumeration tractable. We will show here that our horizontal-cut algorithm can "compensate" such omission without affecting result quality.

A natural approach to complex query column C is to vertically "cut" it into sub-columns (like shown in Figure 8). Patterns for each sub-column can then be generated in turn using FMDV. The benefit of this divide-and-conquer approach is that each sub-domain is likely well-represented in **T**, allowing them to be reliably inferred from **T**. Furthermore, the cost of pattern-enumeration in offline-indexing becomes significantly smaller, as each sub-domain can be enumerated separately.

Specifically, we first use a lexer to tokenize each $v \in C$ into coarse-grained token-class (`<symbol>`, `<num>`, `<letter>`), by scanning each v from left to right and "growing" each token until a character of a different class is encountered. In the example of Figure 8, for each row, we would generate an identical token-sequence "`[<num>|<num>/<num>/<num> <num>:<num>:<num>| ... |<letter>]`".

We then perform multi-sequence alignment (MSA) [22] for all token sequences, before actually performing vertical-cuts. Recall that the goal of MSA is to find optimal alignment across all input sequences (using objective functions such as sum-of-pair scores [43]). Since MSA is NP-hard in general [43], we follow a standard approach to greedily align one additional sequence at a time. We note that for homogeneous machine-generated data, this often solves MSA optimally.

EXAMPLE 7. For the column in Figure 8, each value $v \in C$ has an identical token-sequence with 29 tokens: "`[<num>|<num>/<num>/<num> <num>:<num>:<num>| ... |<letter>]`", and the aligned sequence using MSA can be produced trivially as just a sequence with these 29 tokens (no gaps).

After alignment, characters in each $v \in C$ would map to the aligned sequence of length n (e.g., "0.1" maps to the first `<num>` in the aligned token sequence, "02" maps to the second `<num>`, etc.). Recall that our goal is to vertically split the n tokens into m segments

so that values mapped to each segment would correspond to a sub-domain as determined by FMDV. We define an m-segmentation of C as follows.

DEFINITION 4. Given a column C with n aligned tokens, define $C[s, e]$ as a segment of C that starts from token position s and ends at position e, with $s, e \in [n]$. An m-segmentation of C, is a list of m such segments that collectively cover C, defined as: $(C[s_1, e_1], C[s_2, e_2], \ldots C[s_m, e_m])$, where $s_1 = 1, e_m = n$, and $s_{i+1} = e_i + 1 \; \forall i \in [m]$. We write $C[s_i, e_i]$ as C_i for short, and an m-segmentation as $(C_i : i \in [m])$ for short.

Note that each C_i can be seen just like a regular column "cut" from the original C, with which we can invoke the FMDV. Our overall objective is to jointly find: (1) an m-segmentation $(C_i : i \in [m])$ from all possible segmentation, denoted by $S(C)$; and (2) furthermore find appropriate patterns $h_i \in H(C_i)$ for each segment C_i, such that the entire C can be validated with a low expected FPR. Also note that because we limit pattern enumeration in offline-indexing to columns of at most τ tokens, we would not find candidate patterns longer than τ in offline-index, and can thus naturally limit the span of each segment to at most τ ($e_i - s_i < \tau, \forall i$). We can write this as an optimization problem FMDV-V defined as follows:

$$\text{(FMDV-V)} \quad \min_{\substack{h_1 \ldots h_m \\ h_i \in H(C_i) \\ (C_i : i \in [m]) \in S(C)}} \sum_{h_i} \text{FPR}_T(h_i) \quad (8)$$

$$\text{s.t.} \sum_{h_i} \text{FPR}_T(h_i) \leq r \quad (9)$$

$$\text{Cov}_T(h_i) \geq m, \; \forall h_i \quad (10)$$

In FMDV-V, we are optimizing over all possible segmentation $(C_i : i \in [m]) \in S(C)$, and for each C_i all possible hypothesis patterns $h_i \in H(C_i)$. Our objective in Equation (8) is to minimize the sum of $\text{FPR}_T(h_i)$ for all h_i. We should note that this corresponds to a "pessimistic" approach that assumes non-conforming rows of each h_i are disjoint at a row-level, thus needing to sum them up in the objective function. (An alternative could "optimistically" assume non-conforming values in different segments to overlap at a row-level, in which case we can use max in place of the sum. We find this to be less effective and omit details in the interest of space).

We can show that the minimum FRP scores of m-segmentation optimized in Equation (8) have *optimal substructure* [27], which makes it amenable to dynamic programming (without exhaustive enumeration). Specifically, we can show that the following holds:

$$\text{minFRP}(C[s, e]) =$$

$$\min \left(\min_{h \in P(C[s,e])} \text{FPR}_T(h), \right. \quad (11)$$

$$\left. \min_{t \in [s, e-1]} \left(\text{minFPR}(C[s, t]) + \text{minFPR}(C[t + 1, e]) \right) \right)$$

Here, $\text{minFRP}(C[s, e])$ is the minimum FRP score with vertical splits as optimized in Equation (8), for a sub-column corresponding to the segment $C[s, e]$ from token-position s to e. Note that $C[1, n]$ corresponds to the original query column C, and $\text{minFRP}(C[1, n])$ is the score we want to optimize. Equation (11) shows how this can be

Figure 8: Example columns with composite domains.

computed incrementally. Specifically, $\text{minFRP}(C[s, e])$ is the minimum of (1) $\min_{h \in P(C[s,e])} \text{FPR}_T(h)$, which is the FRP score of treating $C[s, e]$ as one column (with no splits), solvable using FMDV; and (2) $\min_{t \in [s, e-1]} \left(\text{minFPR}(C[s, t]) + \text{minFPR}(C[t + 1, e]) \right)$, which is the best minFRP scores from all possible two-way splits that can be solved optimally in polynomial time using bottom-up dynamic-programming.

EXAMPLE 8. Continue with Example 7. The aligned sequence of column C in Figure 8 has 29 tokens, corresponding to $C[1, 29]$. The sub-sequence $C[4, 14]$ maps to identical values "02/18/2015 00:00:00" for all rows, which can be seen as a "sub-column" split from C.

Using Equation (11), $\text{minFRP}(C[4, 14])$ of this sub-column can be computed as the minimum of: (1) Not splitting of $C[4, 14]$, or $\min_{h \in P(C[4,14])} \text{FPR}_T(h)$, computed using FMDV; and (2) Further splitting $C[4, 14]$ into two parts, where the best score is computed as $\min_{t \in [4, 13]} \left(\text{minFPR}(C[4, t]) + \text{minFPR}(C[t + 1, 14]) \right)$, which can in turn be recursively computed. Suppose we compute (1) to have FPR of 0.004%, and the best of (2) to have FPR of 0.01% (say, splitting into "02/18/2015" and "00:00:00"). Overall, not splitting $C[4, 14]$ is the best option.

Each such $C[s, e]$ can be computed bottom-up, until reaching the top-level. The resulting split minimizes the overall FPR and is the solution to FMDV-V.

We would like to highlight that by using vertical-cuts, we can safely ignore wide columns with more than τ tokens during offline indexing (Section 2.4), without affecting result quality. As a concrete example, when we encounter a column like in Figure 8, brute-force enumeration would generate at least 4^{29} patterns, which is impractically large and thus omitted given a smaller token-length limit τ (e.g., 13) in offline indexing. However, when we encounter this column Figure 8 as a query column, we can still generate valid domain patterns with the help of vertical-cuts (like shown in Example 8), as long as each segment has at most τ tokens. Vertical-cuts thus greatly speed up offline-indexing as well as online-inference, without affecting result quality.

4 AUTO-VALIDATE (HORIZONTAL CUTS)

So far we assumed the query column C to be "clean" with homogeneous values drawn from the same domain (since they are generated by the same underlying program). While this is generally true, we observe that some columns can have ad-hoc special values (e.g., denoting null/empty-values) that do not follow the domain-pattern of C, as shown in Figure 9.

We note that examples like these are not uncommon – even for machine-generated data, there can be a branch of logic (think of

Figure 9: Example ad-hoc special values (in red boxes) not observing patterns of other values in the column.

`try-except`) that handles special cases (e.g., nulls), and produces ad-hoc values not following the pattern of "normal" values. We henceforth refer to these as *non-conforming values*.

Recall that in FMDV, by assuming that the query column C has homogeneous values, we select patterns from $\mathbf{H}(C) = \cap_{v \in C} \mathbf{P}(v)$, or the intersection of patterns for all values $v \in C$. Such an assumption does not hold for columns in Figure 9, yielding an empty $\mathbf{H}(C)$ and no feasible solution to FMDV.

We thus consider a variant of AUTO-VALIDATE with "horizontal cuts", meaning that we can "cut off" a small fraction of non-conforming values in C (which is identical to making patterns tolerate a small fraction of non-conforming values). We use a *tolerance parameter* θ to control the maximum allowed fraction of non-conforming values that can be cut (e.g., 1%, 5%, etc.), which allows for a conscious trade-off between precision and recall.

This optimization problem, termed FMDV-H, is defined as

$$(\text{FMDV-H}) \quad \min_h \text{FPR}_\mathbf{T}(h) \tag{12}$$

$$\text{s.t. } h \in \cup_{v \in C} \mathbf{P}(v) \setminus \text{``*''} \tag{13}$$

$$\text{FPR}_\mathbf{T}(h) \leq r \tag{14}$$

$$\text{Cov}_\mathbf{T}(h) \geq m \tag{15}$$

$$|\{v | v \in C, h \in \mathbf{P}(v)\}| \geq (1 - \theta)|C| \tag{16}$$

Like before, our objective function is to minimize FPR in Equation (12). Equation (13) shows that the hypothesis pattern h is drawn from the union of possible patterns for all $v \in C$. The FPR and Coverage requirements in Equation (14) and Equation (15) are the same as the basic FMDV. Finally, Equation (16) requires that the selected h has to match at least $(1 - \theta)$ fraction of values in C, where θ is the tolerance parameter above (the remaining θ fraction are non-conforming values).

EXAMPLE 9. Consider the leftmost column C in Figure 9. Using FMDV-H, we determine the pattern $h = $ "`<digit>+,<digit>+,<digit>+,<digit>+,<digit>+`" to be a valid solution, as it meets the FPR and coverage requirements based on $\mathbf{T}$. Furthermore h is consistent with 99% of values in C (the remaining 1% non-conforming value is the "-" marked in a red box), thus also satisfying Equation (16).

For arbitrary θ and generalization hierarchy, we can show that deciding whether there is a feasible solution to FMDV-H is NP-hard (let alone minimizing FPR), using a reduction from independent set [45].

THEOREM 2. *The decision version of FMDV-H is NP-hard.*

While FMDV-H is hard in general, in practice because the patterns of non-conforming values often do not intersect with those of "normal" values (as in Example 9), they create easier instances of FMDV-H. Leveraging this observation, in this work we optimize

FMDV-H greedily, by discarding values whose patterns do not intersect with most others in C. We can then find the optimal pattern for the remaining conforming values in FMDV-H using FMDV.

Distributional test of non-conforming values. Given a pattern h inferred from the "training data" C using FMDV-H, and given the data C' arriving in the future, our last task is to determine whether the fraction of non-conforming values in C' has changed significantly from C.

Specifically, at "training" time, we can calculate the fraction of values in C not conforming to h, denoted as $\theta_C(h) = \frac{|\{v | v \in C, h \notin \mathbf{P}(v)\}|}{|C|}$. At "testing" time, we also compute the fraction of values in C' not matching h, denoted as $\theta_{C'}(h)$.

A naive approach to validate C' is to trigger alarms if $\theta_{C'}(h)$ is greater than $\theta_C(h)$. This, however, is prone to false-positives. Imagine a scenario where we compute the non-conforming ratio $\theta_C(h)$ on training data C to be 0.1%. Suppose on C' we find $\theta_{C'}(h)$ to be 0.11%. Raising alarms would likely be false-positives. However, if $\theta_{C'}(h)$ is substantially higher, say at 5%, intuitively it becomes an issue that we should report. (The special case where no value in C' matches h has the extreme $\theta_{C'}(h) = 100\%$).

To reason about it formally, we model the process of drawing a conforming value vs. a non-conforming value in C and C', as sampling from two binomial distributions. We then perform a form of statistical hypothesis test called *two-sample homogeneity test*, to determine whether the fraction of non-conforming values has changed significantly, using $\theta_C(h)$, $\theta_{C'}(h)$, and their respective sample sizes $|C|$ and $|C'|$.

Here, the null-hypothesis H_0 states that the two underlying binomial distributions are the same; whereas H_1 states that the reverse is true. We use *Fischer's exact test* [18] and *Pearson's Chi-Squared test* [44] with Yates correction for this purpose, both of which are suitable tests in this setting. We report C' as an issue if the divergence from C is so strong such that the null hypothesis can be rejected (e.g., $\theta_C(h) = 0.1\%$ and $\theta_{C'}(h) = 5\%$).

We omit details of these two statistical tests in the interest of space. In practice we find both to perform well, with little difference in terms of validation quality.

5 EXPERIMENTS

We implement our offline indexing algorithm in a Map-Reduce-like system used internally at Microsoft [23, 71]. The offline job processes 7M columns with over 1TB data in under 3 hours.

5.1 Benchmark Evaluation

We build benchmarks using real data, to evaluate the quality of data-validation rules generated by different algorithms.

Data set. We evaluate algorithms using two real corpora:

• `Enterprise`: We crawled data produced by production pipelines from an enterprise data lake at Microsoft. Over 100K daily production pipelines (for Bing, Office, etc.) read/write data in the lake [71].

• `Government`: We crawled data files in the health domain from NationalArchives.gov.uk, following a procedure suggested in [20] (e.g., using queries like "hospitals"). We used the crawler from the authors of [20] (code is available from [7]). These data files correspond to data in a different domain (government).[5]

---

[5] This government benchmark is released at https://github.com/jiesongk/auto-validate.

Corpus	total # of data files	total # of data cols	avg col value cnt (standard deviation)	avg col distinct value cnt (standard deviation)
Enterprise (T_E)	507K	7.2M	8945 (17778)	1543 (7219)
Government (T_G)	29K	628K	305 (331)	46 (119)

Table 1: Characteristics of data corpora used.

We refer to these two corpora as T_E and T_G, respectively. The statistics of the two corpora can be found in Table 1.

Evaluation methodology. We randomly sample 1000 columns from T_E and T_G, respectively, to produce two benchmark sets of query-columns, denoted by B_E and B_G.[6]

Given a benchmark **B** with 1000 columns, $\mathbf{B} = \{C_i | i \in [1, 1000]\}$, we programmatically evaluate the precision and recall of data-validation rules generated on **B** as follows. For each column $C_i \in \mathbf{B}$, we use the first 10% of values in C_i as the "training data" that arrive first, denoted by C_i^{train}, from which validation-rules need to be inferred. The remaining 90% of values are used as "testing data" C_i^{test}, which arrive in the future but will be tested against the inferred validation-rules.

Each algorithm A can observe C_i^{train} and "learn" data-validation rule $A(C_i^{\text{train}})$. The inferred rule $A(C_i^{\text{train}})$ will be "tested" on two groups of "testing data": (1) C_i^{test}; and (2) $\{C_j | C_j \in \mathbf{B}, j \neq i\}$.

For (1), we know that when $A(C_i^{\text{train}})$ is used to validate against C_i^{test}, no errors should be detected because C_i^{test} and C_i^{train} were drawn from the same column $C_i \in \mathbf{B}$. If it so happens that $A(C_i^{\text{train}})$ incorrectly reports any error on C_i^{test}, it is likely a false-positive.

For (2), when $A(C_i^{\text{train}})$ is used to validate $\{C_j | C_j \in \mathbf{B}, j \neq i\}$, we know each C_j ($j \neq i$) is likely from a different domain as C_i, and should be flagged by $A(C_i^{\text{train}})$. (This simulates schema-drift and schema-alignment errors that are common, because upstream data can add or remove columns). This allows us to evaluate the "recall" of each $A(C_i^{\text{train}})$.

More formally, we define the *precision* of algorithm A on test case C_i, denoted by $P_A(C_i)$, as 1 if no value in C_i^{test} is incorrectly flagged by $A(C_i^{\text{train}})$ as an error, and 0 otherwise. The overall precision on benchmark **B** is the average across all cases: $P_A(\mathbf{B}) = \text{avg}_{C_i \in \mathbf{B}} P_A(C_i)$.

The *recall* of algorithm A on case C_i, denoted by $R_A(C_i)$, is defined as the fraction of cases in $\{C_j | C_j \in \mathbf{B}, j \neq i\}$ that $A(C_i^{\text{train}})$ can correctly report as errors:

$$R_A(C_i) = \frac{|\{C_j | \text{Validate}(C_j, A(C_i^{\text{train}})) = \text{fail}, C_j \in \mathbf{B}, j \neq i\}|}{|\{C_j | C_j \in \mathbf{B}, j \neq i\}|} \quad (17)$$

Since high precision is critical for data-validation, if algorithm A produces false-alarms on case C_i, we squash its recall $R_A(C_i)$ to 0. The overall recall across all cases in **B** is then: $R_A(\mathbf{B}) = \text{avg}_{C_i \in \mathbf{B}} R_A(C_i)$.

In addition to programmatic evaluation, we also manually labeled each case in B_E with an ideal ground-truth validation-pattern. We use these ground-truth labels, to both accurately report precision/recall on B_E, and to confirm the validity of our programmatic evaluation methodology.

5.2 Methods Compared

We compare the following algorithms using each benchmark **B**, by reporting precision/recall numbers $P_A(\mathbf{B})$ and $R_A(\mathbf{B})$.

Auto-Validate. This is our proposed approach. To understand the behavior of variants of FMDV, we report FMDV, FMDV-V,

FMDV-H, as well as a variant FMDV-VH which combines vertical and horizontal cuts. For FMDV-H and FMDV-VH, we use two-tailed Fischer's exact test with a significance level of 0.01.

Deequ [59]. Deequ is a pioneering library from Amazon for declarative data validation. It is capable of suggesting data validation rules based on training data. We compare with two relevant rules in Deequ for string-valued data (version 1.0.2): the CategoricalRangeRule (refer to as Deequ-Cat) and FractionalCategoricalRangeRule (referred to as Deequ-Fra) [1], which learn fixed dictionaries and require future test data to be in the dictionaries, either completely (Deequ-Cat) or partially (Deequ-Fra).

TensorFlow Data Validation (TFDV) [9]. TFDV is another pioneering data validation library for machine learning pipelines in TensorFlow. It can also infer dictionary-based validation rules (similar to Deequ-Cat). We install TFDV via Python pip (version 0.15.0) and invoke it directly from Python.

Potter's Wheel (PWheel) [58]. This is an influential pattern-profiling method, which finds the best pattern for a data column based on minimal description length (MDL).

SQL Server Integration Services (SSIS) [14]. SQL Server has a data-profiling feature in SSIS. We invoke it programmatically to produce regex patterns for each column.

XSystem [40]. This recent approach develops a flexible branch and merges strategy to pattern profiling. We use the authors' implementation on GitHub [17] to produce patterns.

FlashProfile [50]. FlashProfile is a recent approach to pattern profiling, which clusters similar values by a distance score. We use the authors' implementation in NuGet [8] to produce patterns.

Grok Patterns (Grok) [10]. Grok has a collection of 60+ manually-curated regex patterns, widely used in log parsing and by vendors like AWS Glue ETL [3], to recognize common data-types (e.g., timestamp, ip-address, etc.). For data-validation, we use all values in C_i^{train} to determine whether there is a match with a known Grok pattern (e.g., ip-address). This approach is likely high-precision but low recall because only common data-types are curated.

Schema-Matching [56]. Because Auto-Validate leverages related tables in **T** as additional evidence to derive validation patterns, we also compare with vanilla schema-matching that "broaden" the training examples using related tables in **T**. Specifically, we compare with two instance-based techniques Schema-Matching-Instance-1 (SM-I-1) and Schema-Matching-Instance-10 (SM-I-10), which use any column in **T** that overlaps with more than 1 or 10 instances of C_i^{train}, respectively, as additional training-examples. We also compare with two pattern-based Schema-Matching-Pattern-Majority (SM-P-M) and Schema-Matching-Pattern-Plurality (SM-P-P), which use as training-data columns in **T** whose majority-pattern and plurality-patterns match those of C_i^{train}, respectively. We invoke PWheel on the resulting data, since it is the best-performing pattern-profiling technique in our experiments.

Functional-dependency-upper-bound (FD-UB). Functional dependency (FD) is an orthogonal approach that leverages multi-column dependency for data quality [57]. While FDs inferred from individual table instances often may not hold in a semantic sense [19, 52], we nevertheless evaluate the fraction of benchmark columns that are part of any FD from their original tables, which would be

---

[6]We use the first 1000 values of each column in B_E and the first 100 values of each column in B_G to control column size variations.

a recall upper-bound for FD-based approaches. For simplicity, in this analysis, we assume a perfect precision for FD-based methods.

Auto-Detect-upper-bound (AD-UB) [37]. Auto-detect is a recent approach that detects incompatibility through two common patterns that are rarely co-occurring. For a pair of values (v_1, v_2) to be recognized as incompatible in Auto-detect, both of v_1 and v_2 need to correspond to common patterns, which limits its coverage. Like in FD-UB, we evaluate the recall upper-bound of Auto-detect (assuming a perfect precision).

5.3 Experimental Results

Accuracy. Figure 10(a) shows average precision/recall of all methods using the enterprise benchmark B_E with 1000 randomly sampled test cases. Since no pattern-based methods (FMDV, PWheel, SSIS, etc.) can generate meaningful patterns (except the trivial ".*") on 429 out of the 1000 cases, we report results on the remaining 571 cases in Figure 10(a), where pattern-based methods are applicable.

It can be seen from Figure 10(a) that variants of FMDV are substantially better than other methods in both precision and recall, with FMDV-VH being the best at 0.96 precision and 0.88 recall on average. FMDV-VH is better than FMDV-H, which is in turn better than FMDV-V and FMDV, showing the benefit of using vertical-cuts and horizontal-cuts, respectively.

Among all the baselines, we find PWheel and SM-I-1 (which uses schema-matching with 1-instance overlap) to be the most competitive, indicating that patterns are indeed applicable to validating string-valued data, but need to be carefully selected to be effective.

Our experiments on FD-UB confirm its orthogonal nature to single-column constraints considered by AUTO-VALIDATE– the upper-bound recall of FD-UB only "covers" around 25% of cases handled by AUTO-VALIDATE (assuming discovered FDs have perfect precision).

The two data-validation methods TFDV and Deequ do not perform well on string-valued data, partly because their current focus is on numeric-data, and both use relatively simple logic for string data (e.g., dictionaries), which often leads to false-positives.

Similar results can be observed on the government benchmark B_G, which is shown in Figure 10(b). (To avoid clutter, we omit methods that are not competitive in B_E in this figure). This benchmark is more challenging because we have a smaller corpus and less clean data (e.g., many are manually-edited csv files), which leads to lower precision/recall for all methods. Nevertheless, FMDV methods still produce the best quality, showing the robustness of the proposed approaches on challenging data corpus.

Evaluation Method	precision	recall
Programmatic evaluation	0.961	0.880
Hand curated ground-truth	0.963	0.915

Table 2: Quality results on B_E, using programmatic evaluation vs. manually-cleaned ground-truth.

For all comparisons in Figure 10 discussed above, we perform statistical tests using the F-scores between FMDV-VH and all other methods on 1000 columns (where the null-hypothesis being the F-score of FMDV-VH is no better than a given baseline). We report that the p-values of all comparisons range from 0.001 to 0.007, indicating that we should reject the null-hypothesis and the observed advantages are likely significant.

Manually-labeled ground-truth. While our programmatic evaluation provides a good proxy of the ground-truth without needing any labeling effort, we also perform a manual evaluation to verify the validity of the programmatic evaluation.

Specifically, we manually label the ground-truth validation patterns for 1000 test cases in B_E, and perform two types of adjustments: (1) To accurately report precision, we manually remove values in the test-set of each column that should not belong to the column (e.g., occasionally column-headers are parsed incorrectly as data values), to ensure that we do not unfairly punish methods that identify correct patterns; and (2) To accurately report recall, we identify ground-truth patterns of each test query column, so that in recall evaluation if another column is drawn from the same domain with the identical pattern, we do not count it as a recall-loss.

We note that both adjustments *improve* the precision/recall, because our programmatic evaluation *under-estimates* the true precision/recall. Table 2 compares the quality results using programmatic evaluation and manual ground-truth, which confirms the validity of the programmatic evaluation.

Figure 11 shows a case-by-case analysis of F1 results on B_E with competitive methods, using 100 randomly sampled cases, sorted by their results on FMDV-VH to make comparison easy. We can see that FMDV dominates other methods. An error-analysis on the failed cases shows that these are mainly attributable to advanced pattern-constructs such as flexibly-formatted URLs, and unions of distinct patterns, which are not supported by our current profiler.

Sensitivity. Figure 12 shows a detailed sensitivity analysis for all FMDV variants using average precision/recall. Figure 12(a) shows the result with a varying FRP target r, from the most strict 0 to a more lax 0.1. As we can see, r values directly translate to a precision/recall trade-off and is an effective knob for different precision targets. FMDV-VH is not sensitive for $r \geq 0.02$.

Figure 12(b) shows the effect of varying the coverage target m from 0 to 100. We see that the precision/recall of our algorithms is not sensitive to m in most cases, because the test columns we sample randomly are likely to have popular patterns matching thousands of columns. We recommend a large m, e.g., $m \geq 100$.

Figure 12(c) shows the impact of varying τ, which is the max number of tokens in a column for it to be indexed (Section 2.4). As can be seen, algorithms using vertical-cuts (FMDV-V and FMDV-VH) are insensitive to smaller τ, while algorithms without vertical-cuts (FMDV and FMDV-H) suffer substantial recall loss with a small $\tau = 8$, which shows the benefit of using vertical-cuts. We recommend using FMDV-VH with a small τ (e.g., 8) for efficiency.

Figure 12(d) shows the sensitivity to θ. We can see that the algorithm is not sensitive to θ, as long as it is not too small.

Efficiency. Figure 13 shows the average latency (in milliseconds) to process one query column C in our benchmark. We emphasize that low latency is critical for human-in-the-loop scenarios (e.g., TFDV), where users are expected to verify suggested constraints.

On the right of the figure, we can see the average latency for PWheel, FalshProfile, and XSystem, respectively (we use authors' original implementations for the last two methods [8, 17]). It can be seen that these existing pattern-profiling techniques all require on average 6-7 seconds per column. Given that tables often have tens of columns, the end-to-end latency can be slow.
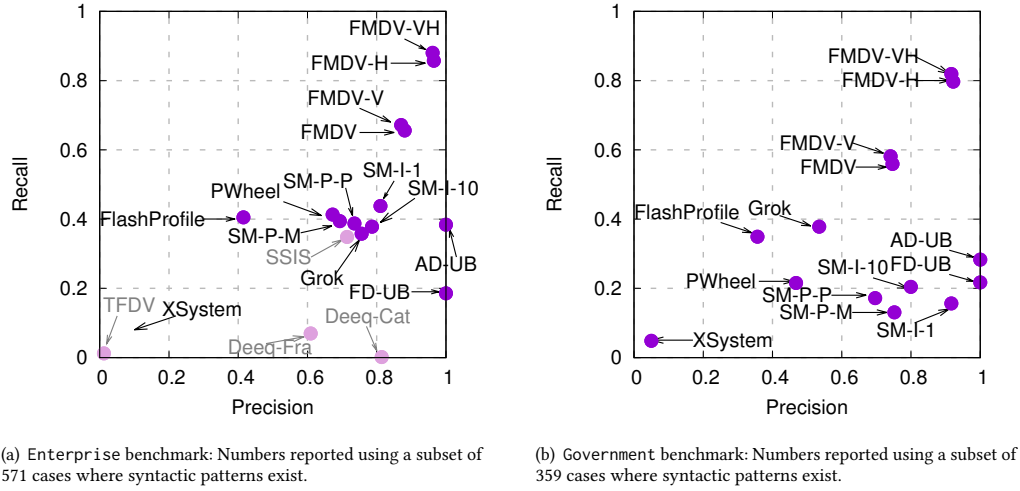
(a) `Enterprise` benchmark: Numbers reported using a subset of 571 cases where syntactic patterns exist.

(b) `Government` benchmark: Numbers reported using a subset of 359 cases where syntactic patterns exist.

Figure 10: Recall vs. Precision comparisons of all methods, on `Enterprise` and `Government` benchmarks.
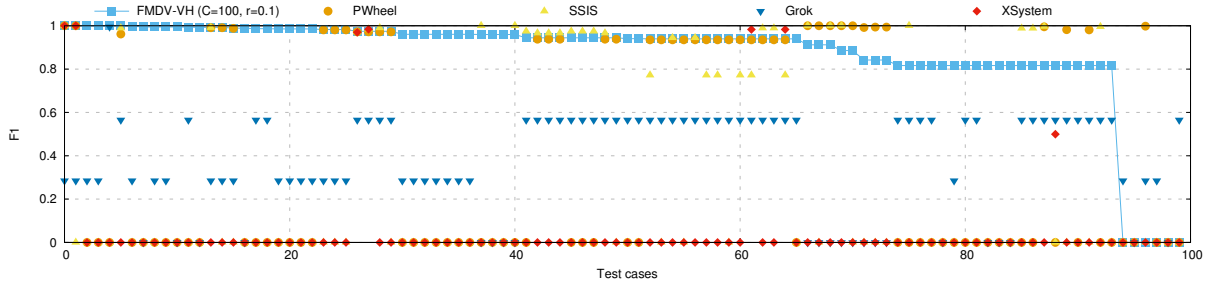
Figure 11: Case-by-case results reported using F-1 scores, on 100 randomly sampled cases.
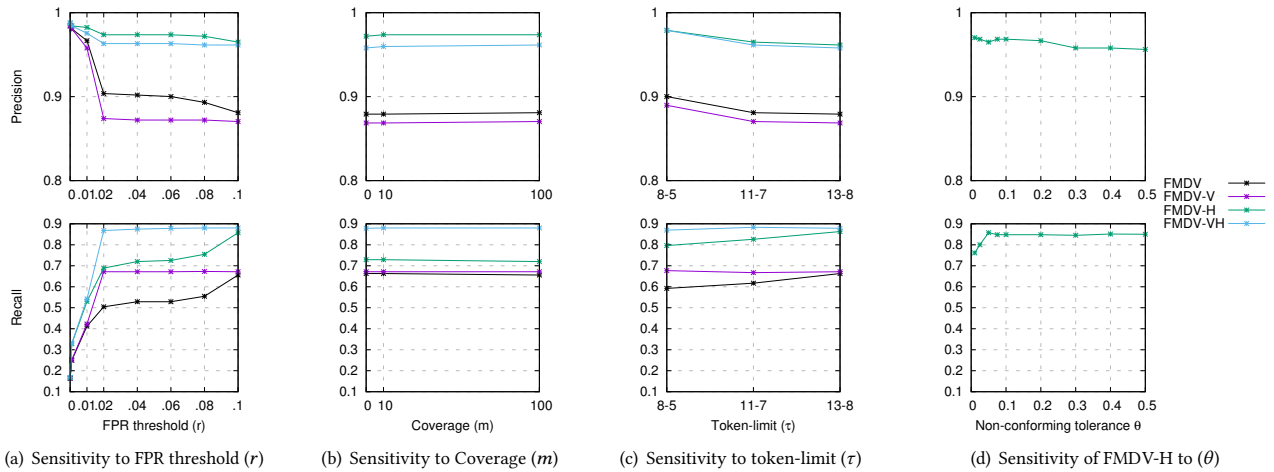

(a) Sensitivity to FPR threshold (r)   (b) Sensitivity to Coverage (m)   (c) Sensitivity to token-limit (τ)   (d) Sensitivity of FMDV-H to (θ)

Figure 12: Sensitivity analysis of FMDV algorithms on the Enterprise benchmark.

In comparison, we observe that despite using a more involved algorithm and a large corpus $\mathbf{T}$, all FMDV variants are two orders of magnitude faster, where the most expensive FMDV-VH takes only 0.082 seconds per column. This demonstrates the benefit of the offline indexing step in Section 2.4, which distills $\mathbf{T}$ (in terabytes) down to a small index with summary statistics (with less than one gigabyte), and pushes expensive reasoning to offline, allowing for fast online response time. As an additional reference point, if we

do not use offline indexes, the "FMDV (no-index)" method has to scan $\mathbf{T}$ for each query and is many orders of magnitude slower.

For the offline indexing step, we report that the end-to-end latency of our job (on a cluster with 10 virtual-nodes) ranges from around 1 hour (with $\tau = 8$), to around 3 hours (with $\tau = 13$). We believe this shows that our algorithm is viable even on small-scale clusters, despite using a large number of patterns.
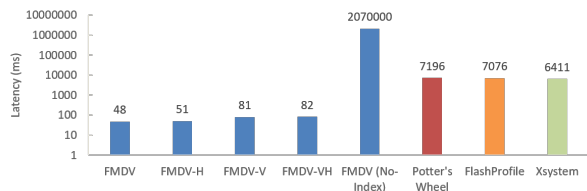
Figure 13: Latency in milliseconds (per query column).

Comparison	avg-time (sec)	avg-precision	avg-recall
Programmer #1	145	0.65	0.638
Programmer #2	123	0.45	0.431
Programmer #3	84	0.3	0.266
FMDV-VH	**0.08**	**1.0**	**0.978**

Table 3: User study comparing 3 programmers with FMDV-VH on 20 test columns, by time spent and pattern quality.

User study. To evaluate the human benefit of suggesting data-validation patterns, we perform a user study by recruiting 5 programmers (all with at least 5 years of programming experience), to write data-validation patterns for 20 sampled test columns in benchmark B_E. This corresponds to the scenario of developers manually writing validation patterns without using the proposed algorithm.

We report that 2 out of the 5 users fail completely on the task (their regex are either ill-formed or fail to match given examples). Table 3 shows the results for the remaining 3 users. We observe that on average, they spend 117 seconds to write a regex for one test column (for it often requires many trials-and-errors). This is in sharp contrast to 0.08 seconds required by our algorithm. Programmers' average precision is 0.47, also substantially lower than the algorithm's precision of 1.0 on hold-out test data.

Case studies using Kaggle. In order to use publicly available data sets to assess the benefits of data validation against schema-drift, we conduct a case study using 11 tasks sampled from Kaggle [12], where each task has at least 2 string-valued categorical attributes. These 11 tasks include 7 for classification: `Titanic`, `AirBnb`, `BNPParibas`, `RedHat`, `SFCrime`, `WestNile`, `WalmartTrips`; and 4 for regression: `HousePrice`, `HomeDepot`, `Caterpillar`, and `WalmartSales`.

To simulate schema-drift [21, 54, 61], for each Kaggle task, we keep the training data unchanged, but swap the position of the categorical attributes in the testing data (e.g., if a data set has two attributes `date` and `time` at column-position 1 and 2, respectively, after simulated schema-drift the two attributes will be at column-position 2 and 1, respectively). This creates a small schema-mismatch between training/testing data that is hard to detect but common in production pipelines like explained in [21, 54].

Figure 14 shows the quality results with and without validation on these Kaggle tasks. For all tasks, we use a popular GBDT method called XGBoost [16] with default parameters. We report R2 for regression tasks and average-precision for classification tasks.

The left (blue) bar of each task, labeled as `No-SchemaDrift`, shows the prediction-quality scores, on the original datasets and without schema-drift. We normalize these scores as 100%. The middle (green) bar, `SchemaDrift-without-Validation`, shows the quality scores with schema-drift, measured relative to the original quality scores. We observe a drop up to 78% (`WalmartTrips`) in normalized scores. Lastly, the right (red) bar, `SchemaDrift-with-Validation` shows the quality when data-validation is used, which correctly detect schema-drift in 8 out of 11 tasks (all except `WestNile`, `HomeDepot`
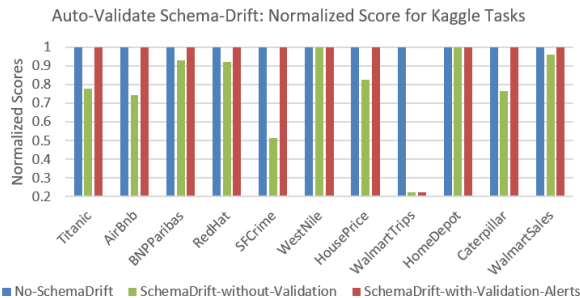

Figure 14: Impact of schema-drift on Kaggle tasks, with and without data-validation. FMDV detects drift in 8/11 cases.

and `WalmartTrips`, with no false-positives). Addressing such validation alerts would significantly boost the resulting quality scores. s While it is generally known that schema-drift hurts ML quality [9, 21], our analysis quantifies its impact on publicly available data and confirms the importance of data-validation.

6 RELATED WORKS

Data Validation. Notable recent efforts on data validation include Google's TensorFlow Data Validation (TFDV) [9, 21] and Amazon's Deequ [1, 61]. These offerings allow developers to write declarative data quality constraints to describe how "normal" data should look like, which are described in detail in the introduction.

Error Detection. There is an influential line of work on error-detection, including methods that focus on *multi-column* dependencies, such as FDs and denial constraints (e.g., [19, 25, 26, 29, 33, 41, 46, 68]), where most techniques would require some amount of human supervision (e.g., manually specified/verified constraints). AUTO-VALIDATE is *unsupervised* and focuses on *single-column* constraints, which naturally complements existing work.

Recent approaches propose learning-based error-detection methods (e.g., [36, 37, 47, 48, 55, 65, 69]), some of which are unsupervised [37, 47, 65] and similar in spirit to AUTO-VALIDATE.

Pattern Profiling. There is also a long line of work on pattern profiling, including pioneering methods such as Potter's wheel [58] that leverages the MDL principle, and other related techniques [31, 32, 40, 49]. While both data validation and profiling produce patterns, data profiling only aims to produce patterns to "cover" given values in a column C, without considering valid values from the same domain but not in C, and is thus prone to false-positives when used for data validation (for data that arrive in the future).

Other forms of validation. While pattern-based validation is a natural fit for machine-generated data; alternative forms of validation may be suitable for other types of data. For example, for natural-language data drawn from a fixed vocabulary (e.g., countries, airport-codes), dictionary-based validation learned from examples (e.g., set expansion [35, 51, 66]) is applicable. For complex types, semantic-type validation [38, 67, 70] can be better suited.

7 CONCLUSIONS AND FUTURE WORK

Observing the need to automate data-validation in production pipelines, we propose a corpus-driven approach to this problem. Possible directions of future work include extending beyond "machine-generated data" to consider natural-language-like data, and extending the same validation principle also to numeric data.

REFERENCES

[1] Retrieved in 2020. Amazon Deequ Library for Data Validation. https://github.com/awslabs/deequ.

[2] Retrieved in 2020. Amazon Glue Code Gen and Workflow. https://aws.amazon.com/blogs/big-data/simplify-data-pipelines-with-aws-glue-automatic-code-generation-and-workflows/.

[3] Retrieved in 2020. AWS Glue custom classifers. https://docs.aws.amazon.com/glue/latest/dg/custom-classifier.html.

[4] Retrieved in 2020. Azure ML: Data Pipelines. https://docs.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines.

[5] Retrieved in 2020. Azure Purview for data governance. https://azure.microsoft.com/en-us/services/purview/.

[6] Retrieved in 2020. Bing Entity Search. https://azure.microsoft.com/en-us/services/cognitive-services/bing-entity-search-api/.

[7] Retrieved in 2020. Data Crawler for NationalArchives.gov.uk. https://github.com/alex-bogatu/DataSpiders.

[8] Retrieved in 2020. FlashProfile package. https://www.nuget.org/packages/Microsoft.ProgramSynthesis.Extraction.Text/.

[9] Retrieved in 2020. Google TensorFlow Data Validation. https://www.tensorflow.org/tfx/guide/tfdv.

[10] Retrieved in 2020. Grok Patterns. https://github.com/elastic/elasticsearch/blob/master/libs/grok/src/main/resources/patterns/grok-patterns.

[11] Retrieved in 2020. Informatica Rev. https://www.informatica.com/products/data-quality/rev.html.

[12] Retrieved in 2020. Kaggle. https://www.kaggle.com/.

[13] Retrieved in 2020. Power BI: Data Flow. https://docs.microsoft.com/en-us/power-bi/transform-model/service-dataflows-create-use.

[14] Retrieved in 2020. SSIS: Data Profiling. https://docs.microsoft.com/en-us/sql/integration-services/control-flow/data-profiling-task?view=sql-server-ver15.

[15] Retrieved in 2020. Tableau: Flow. https://help.tableau.com/current/prep/en-us/prep_build_flow.htm.

[16] Retrieved in 2020. XGBoost. https://xgboost.readthedocs.io/en/latest/.

[17] Retrieved in 2020. XSystem Code. https://bitbucket.org/andrewiilyas/xsystem-old/src/outlier-detection/.

[18] Alan Agresti et al. 1992. A survey of exact inference for contingency tables. *Statistical science* 7, 1 (1992), 131–153.

[19] Laure Berti-Equille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. 2018. Discovery of genuine functional dependencies from relational data with missing values. *VLDB* (2018).

[20] Alex Bogatu, Alvaro AA Fernandes, Norman W Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE*. IEEE, 709–720.

[21] 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.*

[22] Humberto Carrillo and David Lipman. 1988. The multiple sequence alignment problem in biology. *SIAM journal on applied mathematics* 48, 5 (1988), 1073–1082.

[23] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008).

[24] Gang Chen, Keyu Yang, Lu Chen, Yunjun Gao, Baihua Zheng, and Chun Chen. 2016. Metric similarity joins using MapReduce. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2016), 656–669.

[25] Fei Chiang and Renée J Miller. 2008. Discovering data quality rules. *VLDB* 1, 1 (2008).

[26] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *VLDB* 6, 13 (2013).

[27] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[28] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1059–1070.

[29] Tamraparni Dasu, Theodore Johnson, Shanmugauelayut Muthukrishnan, and Vladislav Shkapenyuk. 2002. Mining database structure; or, how to build a data quality browser. In *SIGMOD*.

[30] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. 2009. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology.* 1–11.

[31] Kathleen Fisher and Robert Gruber. 2005. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices* 40, 6 (2005), 295–304.

[32] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. 2008. From dirt to shovels: fully automatic tool generation from ad hoc data. *ACM SIGPLAN Notices* 43, 1 (2008), 421–434.

[33] Lukasz Golab, Howard Karloff, Flip Korn, and Divesh Srivastava. 2010. Data auditor: Exploring data quality and semantics using pattern tableaux. *VLDB* 3, 1-2 (2010).

[34] Yeye He, Jie Song, Yue Wang, Surajit Chaudhuri, Vishal Anil, Blake Lassiter, Yaron Goland, and Gaurav Malhotra. [n.d.]. Auto-Tag: Tagging-Data-By-Example in Data Lakes using Pre-training and Inferred Domain Patterns.

[35] Yeye He and Dong Xin. 2011. Seisa: set expansion by iterative similarity aggregation. In *Proceedings of the 20th international conference on World wide web.* 427–436.

[36] 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.

[37] Zhipeng Huang and Yeye He. 2018. Auto-Detect: Data-Driven Error Detection in Tables. In *SIGMOD*.

[38] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César Hidalgo. 2019. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* 1500–1508.

[39] Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data

sets. In *NIPS MLSys Workshop*.

[40] Andrew Ilyas, Joana MF da Trindade, Raul Castro Fernandez, and Samuel Madden. 2018. Extracting syntactical patterns from databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 41–52.

[41] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*.

[42] Yeye He Jie Song. 2021. Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes (Full version). https://arxiv.org/abs/2104.04659. (2021).

[43] Winfried Just. 2001. Computational complexity of multiple sequence alignment with SP-score. *Journal of computational biology* 8, 6 (2001), 615–623.

[44] Gopal K Kanji. 2006. *100 statistical tests*. Sage.

[45] Richard M Karp and Avi Wigderson. 1985. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM (JACM)* 32, 4 (1985).

[46] Jyrki Kivinen and Heikki Mannila. 1995. Approximate inference of functional dependencies from relations. *Theoretical Computer Science* 149, 1 (1995).

[47] Zifan Liu, Zhechun Zhou, and Theodoros Rekatsinas. 2020. Picket: Self-supervised Data Diagnostics for ML Pipelines. *arXiv preprint arXiv:2006.04730* (2020).

[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] Felix Naumann. 2014. Data profiling revisited. *ACM SIGMOD Record* 42, 4 (2014), 40–49.

[50] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[51] Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale distributional similarity and entity set expansion. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*. 938–947.

[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] Hiren Patel, Alekh Jindal, and Clemens Szyperski. 2019. Big Data Processing at Microsoft: Hyper Scale, Massive Complexity, and Minimal Cost. In *Proceedings of the ACM Symposium on Cloud Computing*. 490–490.

[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] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2019. ANMAT: automatic knowledge discovery and error detection through pattern functional dependencies. In *Proceedings of the 2019 International Conference on Management of Data*. 1977–1980.

[56] Erhard Rahm and Philip A Bernstein. 2001. A survey of approaches to automatic schema matching. *the VLDB Journal* 10, 4 (2001), 334–350.

[57] Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.

[58] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*, Vol. 1.

[59] Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat, Phillipp 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.

[60] 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.

[61] 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.

[62] Michael Stonebraker and Ihab F Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.

[63] Arun Swami, Sriram Vasudevan, and Joojay Huyn. 2020. Data Sentinel: A Declarative Production-Scale Data Validation Platform. In *ICDE*. IEEE, 1579–1590.

[64] Panos Vassiliadis and Alkis Simitsis. 2009. Near real time ETL. In *New trends in data warehousing and data analysis*. Springer.

[65] 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.

[66] Richard C Wang and William W Cohen. 2007. Language-independent set expansion of named entities using the web. In *Seventh IEEE international conference on data mining (ICDM 2007)*. IEEE, 342–350.

[67] Cong Yan and Yeye He. 2018. Auto-Type: Synthesizing type-detection logic for rich semantic data types using open-source code. In *Proceedings of the 2018 International Conference on Management of Data*. ACM.

[68] Jing Nathan Yan, Oliver Schulte, Jiannan Wang, and Reynold Cheng. [n.d.]. CODED: Column-oriented data error detection with statistical constraints.

[69] 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.

[70] Dan Zhang, Yoshihiko Suhara, Jinfeng Li, Madelon Hulsebos, Çağatay Demiralp, and Wang-Chiew Tan. 2019. Sato: Contextual semantic type detection in tables. *arXiv preprint arXiv:1911.06311* (2019).

[71] 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.