Automatic String Data Validation with Pattern Discovery

Ziyan Han[1], Xinwei Lin[1], Peng Di[2], Chuan Xiao[3,4], Makoto Onizuka[3], Jiuzhang Liu[5], Rui Mao[1⋆], and Jianbin Qin[1⋆]

[1] SICS, Shenzhen University, China
{hanzy@, linxinwei2020@email., mao@, qinjianbin@}szu.edu.cn
[2] Ant Group, China
dipeng.dp@antgroup.com
[3] Osaka University, Japan
{chuanx, onizuka}@ist.osaka-u.ac.jp
[4] Nagoya University, Japan
[5] Southwest University, China
xndx66030012@email.swu.edu.cn

Abstract. Periodic data insertions in enterprise data pipelines may propagate data quality issues downstream, potentially disrupting critical services. Although on-call engineers can investigate and fix such issues, identifying their root causes is often time-consuming. This paper presents AutoPattern, a self-validating data management system that automatically discovers patterns to validate semi-structured string data in enterprise pipelines. AutoPattern extracts patterns from historical data in a top-down manner, first inferring high-level structural skeletons to capture recursive and vertically aligned relationships, then performing fine-grained semantic refinement to balance generalization and specification. To address cold start and rapid data growth, we further introduce a data augmentation module and an incremental pattern update mechanism. Extensive experiments on public, synthetic, and industrial datasets demonstrate the effectiveness and efficiency of AutoPattern, with an average precision of 0.91 and a recall of 0.89, outperforming competitive baselines. A case study conducted on the industrial platform of Ant Group Inc., which hosts thousands of applications, further confirms its practicality in real-world production pipelines, where AutoPattern effectively captures meaningful patterns and assists engineers in rapid error localization.

Keywords: Data quality · Data validation · Pattern discovery.

1 Introduction

Software failures in production environments are inevitable and often difficult to diagnose. While a large body of work has focused on fault localization and automated debugging[13,46,25,45], not all failures are caused by software bugs. Faulty input data can also disrupt system operations, particularly in data-intensive

⋆ Corresponding author

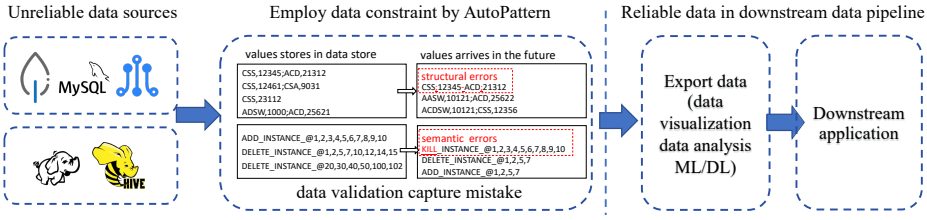


Fig. 1. Data pipelines with data validation.

scalable computing (DISC) systems such as Apache Spark [50,40,29,48,13]. Invalid input files are common in practice [29], with 11% of Stack Overflow issues related to data input and output (I/O) errors [44]. This problem is further exacerbated in enterprise data pipelines (e.g., SSIS [6], PowerBI [5]) and data warehouses [31], where data evolves continuously and periodic insertions are prevalent. Frequent updates may introduce errors from human mistakes or imperfect software processes, compromising data quality and system security. Such errors are difficult to detect and further complicate failure diagnosis, making data quality assurance essential yet labor-intensive and costly in data-driven systems [39].

Data cleaning vs. validation. Data cleaning aims to detect and repair erroneous data, while data validation is to prevent errors early in the pipeline. Commercial tools (e.g., Microsoft Excel [3], Trifacta [7], OpenRefine [4]) provide basic detection functions, and constraint-based methods (e.g., [35]) leverage dependencies for data cleaning. However, these approaches are designed for static datasets and struggle in dynamic environments where data and constraints evolve rapidly.

Data validation tools [14,38] define declarative constraints through domain-specific languages (DSLs) to detect errors early and aid debugging. However, hand-authoring constraints is time-consuming and labor-intensive. This motivates automatic validation approaches such as Auto-Validate [39] and profiling methods [32,43,36]. Yet these methods perform poorly on *nested* and *semi-structured* strings, where both hierarchical structure and character-level semantics are critical. Our study on enterprise production datasets shows that semi-structured strings comprise over 70% of values, while existing methods [12,37] often trigger false alarms. To the best of our knowledge, no prior work explicitly targets pattern-based validation of semi-structured strings. Prior work on *structure extraction* (e.g., CRAM [16], ListExtract [18], TEGRA [15]), *text segmentation* (e.g., Judie [9]), and *type description* (e.g. PADS [20] and LearnPADS [21]) primarily transforms semi-structured text into structured forms. FlashFill [26] synthesizes string transformation programs from input–output examples rather than validation patterns. These methods capture structural information, but are insufficient to detect character-level semantic errors. In contrast, our method infers structural skeletons and refines them at the character level, yielding validation patterns that are both precise and robust.

Example 1. As shown in the Fig. 1, enterprise data pipelines integrate data from various sources, where two common error types arise: (i) *structural errors*, which involve violations of expected delimiters, nesting, or alignment (e.g., "CSS,12345;ACD,21312"

→ "CSS;12345-ACD;21312"); and (ii) *semantic errors*, which occur when tokens violate domain-specific semantics (e.g., "KILL" instead of "DELETE"). These examples highlight the need to capture both structural and semantic patterns.

In this paper, we propose AutoPattern, a self-validating data management system for automatic validation of semi-structured string data. AutoPattern incorporates a two-stage pattern discovery module: (1) *skeleton extraction*, which infers high-level structural skeletons to capture recursive and vertically aligned relationships embedded in historical data; and (2) *fine-grained semantic refinement*, which performs entropy-guided, character-level refinement over the extracted skeletons via a generalization tree, effectively balancing generalization and specification. To further enhance adaptability, we introduce a data augmentation module to address cold-start issues, and an incremental pattern update mechanism to accommodate data growth. To summarize, the main contributions of this paper are as follows:

- We propose AutoPattern, a self-validating data management system that learns and updates patterns to validate semi-structured strings, enabling accurate detection and localization of structural and semantic errors in enterprise pipelines.
- We design a novel pattern-centric domain-specific language (DSL) for semi-structured strings that supports recursive and aligned structures (Section 3).
- We introduce a skeleton extraction algorithm that jointly leverages recursive and vertical splitting strategies to discover high-level structures (Section 4).
- We present a fine-grained semantic refinement algorithm that refines skeletons at the character level to balance generalization and specification (Section 5).
- We develop a data augmentation module and an incremental pattern update mechanism to handle limited historical data and evolving data streams (Section 6).
- We conducted extensive experiments on public, synthetic and industrial datasets to evaluate the effectiveness and efficiency of AutoPattern, and demonstrates its practicality through a case study on Ant Group's production platform (Section 7).

The rest of this paper introduces preliminaries and the validation framework (Section 2), reviews related work (Section 8), and concludes the paper (Section 9).

2 Preliminaries and Framework

This section presents preliminaries [32] and the overall framework of AutoPattern.

Definition 1 (Atom pattern). *An* atom pattern *(or* atom*) is a basic matching function over strings. Formally, $\alpha : String \to \mathbb{N}$ maps a string l to the length of the longest prefix of l that satisfies its constraint; $\alpha(l) = 0$ indicates no match.*

Definition 2 (Pattern). *A* pattern *is a sequence of atoms. A string l matches a pattern p iff the atoms in p consecutively match non-empty substrings that cover l exactly.*

Definition 3 (Data profile). *Given a set of strings $\mathcal{L}$,* data profile *is a set of patterns P summarizing $\mathcal{L}$, i.e., for each $l \in \mathcal{L}$ there exists $p \in P$ such that l matches p.*

Definition 4 (Data validation). *Starting from an initial profile P_0 learned from historical data $\mathcal{L}_0$, validation iterates $P_0 \xrightarrow{F_0} P'_0 \xrightarrow{F_1} P'_1 \cdots$. At step i, P'_i validates the next batch $\Delta\mathcal{L}_{i+1}$ and updates patterns with optional feedback F_i as data evolves.*

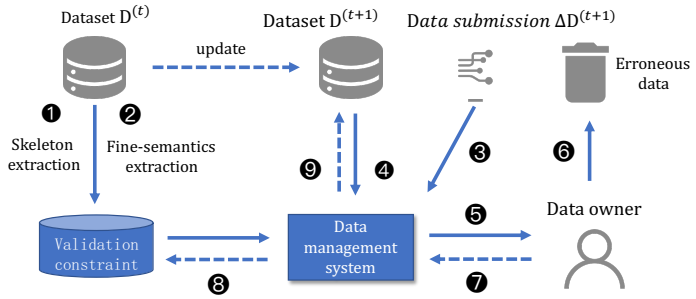


Fig. 2. The framework of AutoPattern.

2.1 Data Validation Framework

Fig. 2 illustrates the framework of our self-validating data management system. Given a snapshot $D^{(t)}$, the system first extracts a skeleton $H(D^{(t)})$ (❶), then refines it into fine-grained patterns $P_f(D^{(t)})$ (❷). When new data $\Delta D^{(t+1)}$ arrives, it checks compatibility with $P_f(D^{(t)})$ (❸). If compatible, $\Delta D^{(t+1)}$ passes validation (❹) and the dataset is updated. Otherwise, the system requests user confirmation (❺): if rejected, the data is discarded (❻); if confirmed (❼), the system updates both the constraints (❽) and the dataset (❾). In summary, constraints are derived from historical data and adaptively refined with user feedback for future validation.

3 Domain-Specific Language

We design a DSL to specify patterns over semi-structured strings, organized via a generalization tree [32,39] and supporting predefined atoms and user-defined extensions.

Definition 5 (Generalization Tree). *A* generalization tree G *is a hierarchy where leaf nodes are characters and internal nodes are atom patterns. The mapping* $m : char \to atom$ *assigns each character to its generalized atom in* G.

Definition 6 (Pattern Matching). *Given a pattern* $p = \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_k$ *and a string* l*, the* pattern matching *function* $\mathcal{M}_p : String \to \mathbb{N}$ *returns the length of the longest prefix of* l *matched by* p*. It is recursively defined as* $\mathcal{M}_p(l) = 0$ *if* $\alpha_1(l) = 0$*, and* $\mathcal{M}_p(l) = \alpha_1(l) + \mathcal{M}_{p'}(l[\alpha_1(l) :])$ *otherwise, where* $p' = \alpha_2 \circ \cdots \circ \alpha_k$*. If* $\mathcal{M}_p(l) = |l|$*,* p *matches* l *exactly. We adopt greedy matching to avoid backtracking.*

Definition 7 (Base Type and Pattern Structure). *The* base types *include below primitive tokens. A* pattern structure S *defined over these base types follows:*

$$b ::= \texttt{Int} \mid \texttt{Float} \mid \texttt{String} \mid \texttt{Symbol} \mid \texttt{customToken},$$
$$S ::= \texttt{Align}\{S_1, S_2, \ldots, S_n\} \mid \texttt{Recursive}\{S\}[sep] \mid \texttt{Recursive}\{S\}[sep, p]$$
$$\mid \texttt{Recursive}\{S\}[sep, term] \mid \texttt{Recursive}\{S\}[sep, term, p].$$

Here, $\texttt{Align}\{S_1, S_2, \ldots, S_n\}$ *requires* $S_1, \ldots, S_n$ *in order;* $\texttt{Recursive}\{S\}[sep]$ *repeats* S *any number of times, separated by* sep*; optional* p *specifies the repetition count; and optional* $term$ *specifies a termination symbol.*

Definition 8 (Syntax and Semantics of the DSL). *Given a generalization tree G and a dataset, i.e., a list of strings $\mathcal{L}$, the syntax of our DSL is defined as follows:*

$$\begin{aligned} \textit{recursive } r &::= \texttt{getRecursive}(G, \mathcal{L}), \\ \textit{align } v &::= \texttt{getAlign}(G, \mathcal{L}), \\ \textit{skeleton } h &::= r \mid v \mid \texttt{combine}(r, v, d), \\ \textit{fineGrained } f &::= \texttt{refine}(h, G, \mathcal{L}), \\ \textit{profile } pr &::= f. \end{aligned}$$

Here, `getRecursive`/`getAlign` extract recursive/alignment skeletons, `combine` merges them with depth d, and `refine` yields the final fine-grained profile pr.

Definition 9 (Pattern-based Distance). *Given two strings l_i and l_j and a generalization tree G, the* pattern-based distance *is defined as $d(l_i, l_j) = GC(l_i, a^*) + GC(l_j, a^*)$, where a^* is the nearest common ancestor of the atom patterns of l_i and l_j in G, and $GC(l, a)$ is the cumulative cost of mapping l upward to ancestor a. If $l_i = l_j$, $d(l_i, l_j) = 0$; if any character in l_i or l_j cannot be mapped to G, $d(l_i, l_j) = \infty$.*

4 Skeleton Extraction

Problem statement. Given a string dataset $\mathcal{L}$ and a generalization tree G, *skeleton extraction* partitions each $l \in \mathcal{L}$ into $l = h_1(l) \circ h_2(l) \circ \cdots \circ h_k(l)$ and, for each segment index i, forms the segment set $H_i = \{h_i(l) \mid l \in \mathcal{L}\}$ from which a pattern p_i is learned. The goal is to jointly determines a partition and per-segment patterns $\{p_i\}_{i=1}^{k}$ by minimizing the overall pattern-based distance within segments: $H_{opt} = \arg\min_{\{h_1,\ldots,h_k\} s.t. \forall l \in \mathcal{L}: l = \circ_{i=1}^{k} h_i(l)} \sum_{1 \le i \le k} Dist(H_i)$, where $Dist(H_i) = \sum_{l_p < l_q} d(h_i(l_p), h_i(l_q))$ aggregates pairwise distances within H_i.

Theorem 1. *The optimal skeleton extraction problem is NP-hard.*

Proof. We prove the NP-hardness by a polynomial-time reduction from *Multiple Sequence Alignment (MSA)* problem under the sum-of-pairs (SP) cost, which is NP-hard [41]. Under the SP model, replacing the alignment objective with our pairwise pattern-based distance makes the optimal global segmentation correspond to an optimal MSA alignment. Hence, the problem is NP-hard.

Despite its hardness, we design the following recursive and vertical splitting algorithms that together capture complex skeletons in semi-structured strings.

4.1 Recursive Splitting Algorithm

Recursive splitting identifies repetitive sub-structures connected by delimiters. Given a string l, a *recursive splitting operator* r divides l into alternating segments and delimiters, $r(l) = \{Seg_1 Deli_1 Seg_2 Deli_2 ... Seg_t\}$, where each Seg_i is a contiguous segment and each $Deli_i$ is the delimiter. For a dataset $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, let $\mathcal{S}(l_i) = \{r_1(l_i), r_2(l_i), ..., r_m(l_i)\}$ denote all candidate recursive segmentations of l_i,

Algorithm 1: RecursiveSplit

Input : A set of strings $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, and the generalization tree G.
Output : The optimal recursive skeleton $\mathcal{H}$.

$C_{\text{skel}} \leftarrow \emptyset$;
$\mathcal{S}(l_1), \mathcal{S}(l_2), \ldots, \mathcal{S}(l_n) \leftarrow$ EnumerateRecursiveSegmentations($\mathcal{L}$);
foreach $l_i \in \mathcal{L}$ **do**
  foreach $r \in \mathcal{S}(l_i)$ **do**
    $Dist_{sum} \leftarrow 0$;
    foreach $l_j \in \mathcal{L}, j \neq i$ **do**
      $r^* \leftarrow r_{opt}(l_j, r(l_i))$ according to Eq.(1);
      $Dist_{sum} \leftarrow Dist_{sum} + Dist_r(r^*)$;
    $H_r \leftarrow$ ExtractSkeletonFromSegmentation(r, G);
    $C_{\text{skel}} \leftarrow C_{\text{skel}} \cup \{(H_r, Dist_{\text{sum}})\}$;
$\mathcal{H} \leftarrow$ SelectSkeletonByMinDistance(C_{skel});
return $\mathcal{H}$;

and $\mathcal{S}(\mathcal{L}) = \{\mathcal{S}(l_1), \mathcal{S}(l_2), \ldots, \mathcal{S}(l_n)\}$ denote the full candidate segmentation set. To evaluate structural consistency, we define the *recursive distance* of a segmentation $r(l)$ as $Dist_r(r(l)) = \sum_{1 \leq i < j \leq t} d(Seg_i, Seg_j)$, where $d(\cdot, \cdot)$ is the pattern-based distance. The *optimal segmentation* of a string l_j w.r.t. a reference $r(l_i)$ is defined as

$$r_{opt}(l_j, r(l_i)) = \underset{r(l_j) \in \mathcal{S}(l_j)}{\arg\min} \left(Dist_{\mathsf{align}}(r(l_i), r(l_j)) + Dist_r(r(l_j))\right). \tag{1}$$

where $Dist_{\mathsf{align}}(r(l_i), r(l_j))$ is the segment-alignment cost between $r(l_i)$ and $r(l_j)$, computed by summing pattern-based distances between corresponding segments.
Algorithm. Algorithm 1 first enumerates candidate recursive segmentations $\mathcal{S}(l_i)$ for each string. For each reference $r \in \mathcal{S}(l_i)$, it evaluates global consistency by comparing r against every other string l_j $(j \neq i)$: it selects $r^* = r_{opt}(l_j, r(l_i))$ via Eq.(1) and accumulates the pairwise cost $Dist_{sum}$. The reference r is then converted into a structural skeleton H_r and stored with its accumulated distance in C_{skel}. Finally, the skeleton with the minimum total distance is returned.
Complexity. Let $n = |\mathcal{L}|$, $m = |\mathcal{S}(l_i)|$, and k the average number of segments per candidate. The algorithm examines $O(nm)$ reference segmentations, each compared with $O(n)$ other strings; each comparison scans m candidates and computes $Dist_r(\cdot)$ in $O(k^2)$, leading to a worst complexity of $O(m^2n^2k^2)$. With precomputed recursive distances, selecting r_{opt} reduces to $O(1)$, yielding $O(mn^2k^2)$ overall.

4.2 Vertical Splitting Algorithm

Vertical splitting aligns tokens across multiple strings by positions. Given $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, let $minLength(\mathcal{L})$ denote the minimum string length in $\mathcal{L}$. For each $l \in \mathcal{L}$, a *vertical splitting operator* v divides l into alternating segments and delimiters, $v(l) = \{Seg_1^l Deli_1^l \ldots Seg_t^l Una^l\}$, where the unaligned tail Una^l satisfies $Length(l) = minLength(\mathcal{L}) + Length(Una^l)$. Let $\mathcal{V}(l) = \{v_1(l), v_2(l), \ldots, v_m(l)\}$

Algorithm 2: VERTICALSPLIT

Input : A set of strings $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, and the generalization tree G.
Output : The optimal vertical skeleton $\mathcal{H}$.

$C_{\text{skel}} \leftarrow \emptyset$;
$\mathcal{V}(l_1), \mathcal{V}(l_2), \ldots, \mathcal{V}(l_n) \leftarrow$ ENUMERATEVERTICALSEGMENTATIONS($\mathcal{L}$);
$\mathcal{V}_{\text{all}} \leftarrow \bigcup_{i=1}^{n} \mathcal{V}(l_i)$;
foreach $v \in \mathcal{V}_{all}$ **do**
  $Dist_{sum} \leftarrow 0$; $H_v \leftarrow \epsilon$;
  foreach $col \in$ BUILDALIGNEDCOLUMNS($v, \mathcal{L}$) **do**
    $Dist_{sum} \leftarrow Dist_{sum} + Dist_v(col)$;
    $H_v \leftarrow H_v \circ$ EXTRACTSKELETONFROMCOLUMN(col, G);
  $C_{\text{skel}} \leftarrow C_{\text{skel}} \cup \{(H_v, Dist_{sum})\}$;
$\mathcal{H} \leftarrow$ SELECTSKELETONBYMINDISTANCE(C_{skel});
return $\mathcal{H}$;

denote all candidate vertical segmentations of l, and $\mathcal{V}(\mathcal{L}) = \{\mathcal{V}(l_1), \mathcal{V}(l_2), \ldots, \mathcal{V}(l_n)\}$ their collection. The *vertical distance* between two strings l_i and l_j under v is

$$Dist_v(v(l_i), v(l_j)) = \sum_{1 \le k \le t} \big(d(Seg_k^{l_i}, Seg_k^{l_j}) + d(Deli_k^{l_i}, Deli_k^{l_j})\big) + \mathcal{C}_{unalign}(v(l_i), v(l_j)),$$

where $d(\cdot, \cdot)$ is the pattern-based distance and $\mathcal{C}_{unalign} \ge 0$ penalizes unalignment.
Algorithm. Algorithm 2 first enumerates all candidate vertical segmentations $\mathcal{V}(l_i)$ for each string l_i, then computes their union $\mathcal{V}_{\text{all}}$ to eliminate duplicate splits. For each candidate $v \in \mathcal{V}_{\text{all}}$, it aligns *all* strings into columns. For each column col, it accumulates the vertical distance $Dist_v(col) = \sum_{i<j} Dist_v\big(\text{frag}(l_i, col), \text{frag}(l_j, col)\big)$ and concatenates the extracted column pattern to form H_v. After all columns are processed, each skeleton H_v along with its total distance is recorded; finally, the skeleton with the minimum distance is returned.
Complexity. Let $n = |\mathcal{L}|$, $m = |\mathcal{V}_{\text{all}}|$, and k the average columns per candidate. Evaluating $O(mk)$ columns results in a worst complexity of $O(n^2mk)$. With cached computations, the per-column cost reduces to $O(n)$, yielding $O(nmk)$ overall.

4.3 Skeleton Generation and Selection

Skeleton generation. Real-world data are often nested and heterogeneous; therefore, we adopt a hierarchical process combining recursive and vertical splitting. For each candidate segmentation, base-type substructures (non-decomposable) are retained as atomic components, while others are further partitioned by RECURSIVESPLIT (Alg. 1) and VERTICALSPLIT (Alg. 2), then selecting the decomposition with lower pattern-based distance. The resulting sub-skeletons are concatenated into a global candidate, and at each step the locally optimal sub-skeleton (by pattern-based distance) is selected until all components reduce to base types.
Skeleton selection. When multiple candidates yield the same minimum distances, we rank them by two metrics and select those most suitable for validation: (i)

Algorithm 3: Fine-grained Semantic Refinement

Input : A set of candidate skeletons $\mathcal{H}$, a dataset of strings $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$, and the generalization tree G.
Output : A set of refined patterns $P = \{p_H \mid H \in \mathcal{H}\}$.

$P \leftarrow \emptyset$;
foreach $H \in \mathcal{H}$ **do**
  $\mathcal{L}_H \leftarrow \{l \in \mathcal{L} \mid \textsc{Alignable}(H, l)\}$; $p_H \leftarrow H$;
  for $i \leftarrow 1$ **to** $|H|$ **do**
    $\mathcal{S}_i \leftarrow \{\text{frag}(l, i) \mid l \in \mathcal{L}_H \wedge \textsc{AlignedAt}(H, l, i)\}$;
    $p_H \leftarrow \textsc{GreedyTraverse}(p_H, i, \mathcal{S}_i, G, \mathcal{L}_H)$;
  $P \leftarrow P \cup \{p_H\}$;
return P;

Algorithm 4: GreedyTraverse

Input : A pattern $p = \alpha_1 \circ \cdots \circ \alpha_{|H|}$, column index i, column multiset $\mathcal{S}_i$, the generalization tree G, and the dataset $\mathcal{L}_H$.
Output : The refined pattern p_{refine}.

$p_{\text{refine}} \leftarrow p$; *best* $\leftarrow \textsc{Cost}(p, \mathcal{L}_H)$; *improved* $\leftarrow$ true;
while *improved* **do**
  improved $\leftarrow$ false;
  foreach $c \in \textsc{GetChildren}(\alpha_i, G)$ **do**
    if $\forall s \in \mathcal{S}_i : \textsc{MatchAtom}(c, s)$ **then**
      $p^{cand} \leftarrow \textsc{ReplaceAt}(p_{\text{refine}}, i, c)$; $cCost \leftarrow \textsc{Cost}(p^{cand}, \mathcal{L}_H)$;
      if $cCost <$ *best* **then**
        $p_{\text{refine}} \leftarrow p^{cand}$; *best* $\leftarrow cCost$; *improved* $\leftarrow$ true;
return p_{refine};

coverage $Cov(H)$, the proportion of historical strings matching H (higher is better), and (ii) *structural clarity* $SCS(H)$, measuring how clearly H defines boundaries (e.g., delimiter consistency and column alignment). Candidates are first sorted by $Cov(H)$ and then by $SCS(H)$, and the top-k are selected as final patterns.

5 Fine-grained Semantic Refinement

Skeleton extraction captures structural patterns but may overlook character-level semantics, we therefore introduce a fine-grained semantic refinement method.

Entropy-based cost function. We quantify the semantic richness of a pattern $p = \alpha_1 \circ \cdots \circ \alpha_k$ over a dataset $\mathcal{L}$ using *entropy* and further define a cost function:

$$Ent(p, \mathcal{L}) = -\sum_{1 \le i \le k} P_{\mathcal{L}}(\alpha_i) \log P_{\mathcal{L}}(\alpha_i) + \beta, \quad Cost(p, \mathcal{L}) = \sum_{1 \le i \le k} \lambda(\alpha_i)\, Ent(\alpha_i, \mathcal{L}),$$

where $P_{\mathcal{L}}(\alpha_i)$ is the empirical probability of atom α_i in $\mathcal{L}$, β is a small smoothing term, and $\lambda(\alpha_i) \ge 0$ weights atom importance. Let $\mathcal{P}(G)$ denote candidate patterns

derived from skeletons by downward traversal of the generalization tree G; thus, the optimal pattern is $p^* = \arg\min_{p' \in \mathcal{P}(G)} Cost(p', \mathcal{L})$.

Greedy Refinement Algorithm. Since the candidate space $\mathcal{P}(G)$ grows exponentially, we design a greedy column-wise refinement algorithm. For each skeleton $H \in \mathcal{H}$, Alg. 3 collects its alignable subset $\mathcal{L}_H$, builds fragment multisets $\mathcal{S}_i$ for each column i, and invokes Alg. 4 (GreedyTraverse) to refine the i-th atom of pattern p_H. GreedyTraverse refines one column at a time: given the current pattern p, column index i, fragment multiset $\mathcal{S}_i$, tree G, and dataset $\mathcal{L}_H$, it explores child atoms $c \in \textsc{GetChildren}(\alpha_i, G)$ and replaces $\alpha_i \to c$ only if (i) c matches all fragments in $\mathcal{S}_i$ and (ii) the global cost decreases, i.e., $\textsc{Cost}(p^{\text{cand}}, \mathcal{L}_H) < \textsc{Cost}(p, \mathcal{L}_H)$. It repeats until no further improvement, then proceeds to the next column, yielding patterns that balance generality and specificity while remaining robust for validation.

6 Adaptive Pattern Learning and Update

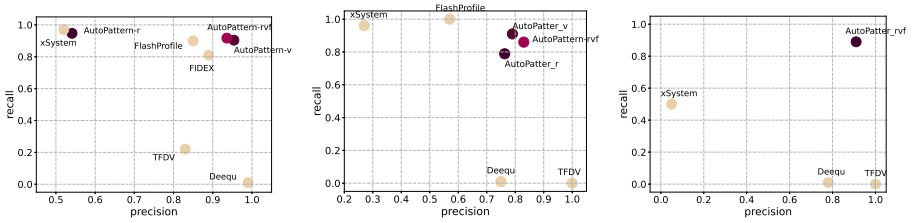
Data Augmentation. In practice, pattern learning often faces a *cold-start* issue: limited historical data fail to capture future variations, leading to correct new records being misclassified as false negatives. Corpus-based enrichment [39] relies heavily on corpus quality and domain coverage, which are hard to guarantee. We therefore integrate an automatic *data augmentation* module that cooperates with fine-grained refinement (Alg. 3) and activates when training data are insufficient. If the number of usable samples for a column/atom is below a threshold, it triggers *example generation* guided by the generalization tree G. For each atom α_i in the refined pattern set P, we identify its sibling set $\mathcal{C}(\alpha_i)$ (children of the same parent in G) and sample $\alpha_i' \in \mathcal{C}(\alpha_i)$ to replace α_i. Enumerating and combining such substitutions yields boundary-like synthetic instances that expand data diversity and enhance robustness. Conceptually, this process performs the inverse of pattern inference, synthesizing data from patterns by traversing G until reaching leaf nodes. These automatically generated examples expand the feature space of training data, with minimal human effort, reducing early false negatives and mitigating cold start.

Incremental Pattern Update. To maintain validation accuracy without reprocessing all history, the system incrementally updates patterns as new records arrive. If an incoming value satisfies the current pattern set P, we keep P unchanged; otherwise, we generalize the responsible atoms via a local upward search on G to the nearest ancestor that covers both historical and newly verified data. Formally, the incremental pattern update process can be expressed as: $P^{(t)} = f(\Delta D(t), P^{(t-1)})$, where $P^{(t-1)}$ denotes the current pattern profile, $\Delta D^{(t)}$ represents the newly submitted data, and $P^{(t)}$ is the updated pattern profile after adaptation.

7 Experiments

7.1 Experimental Setup

Datasets. We evaluated AutoPattern on three datasets: (1) Kaggle (214 keys), build from CSV files collected from Kaggle [1], treating each column as a key (task); (2)



(a) Kaggle dataset (b) Random dataset (c) Enterprise dataset

Fig. 3. Performance evaluation.

Random (1000 keys), synthesized by composing Kaggle-derived atomic columns with randomly sampled delimiters to form nested strings; and (3) Enterprise (1000 keys), sampled from an industrial platform with about 2 million key-value records, aggregating all values of each key into a per-key string set (average length ≈ 939). Each *key* denotes a homogeneous string field, on which we train and evaluate separately.

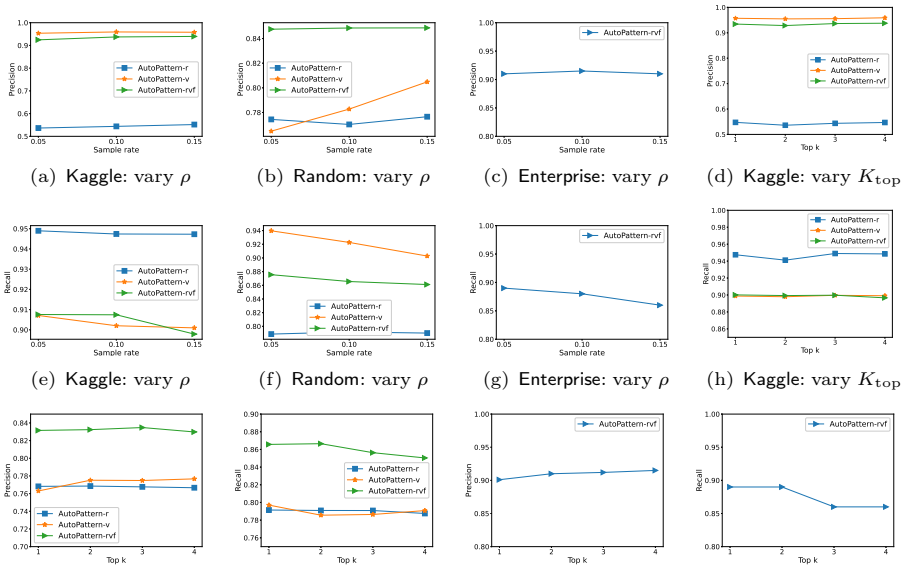
Methods. We evaluated three variants of the proposed method, including AutoPattern-r (recursive-only), AutoPattern-v (vertical-only), and AutoPattern-rvf (recursive +vertical with fine-grained semantic refinement). The first two variants perform only skeleton extraction without the refinement stage. We also compared with representative baselines, including Deequ [38], a Spark-based data-quality library with a declarative DSL; FIDEX [43], an example-driven filter synthesizer; TensorFlow Data Validation (TFDV) [14], which detect schema and statistical anomalies; xSystem [28], which extracts patterns via a branch-and-merge strategy; and FlashProfile [32], which synthesize regex-like patterns via program induction.

Metrics. For each key, We randomly split its homogeneous string set into 10% for training and 90% for testing. Patterns were learned from the training split and evaluated on two test groups: (i) an clean in-domain subset (10%) from the test split, where any violated value counts as a false positive; and (ii) the remaining subset (80%) with injected errors to simulate incoming noisy values. Errors include structural perturbations (e.g., delimiteredits) and character-level edits (e.g., insertions, deletions). Precision is the ratio of truly correct test values accepted by the validator, and recall is the ratio of injected incorrect values correctly rejected.

Configuration. Experiments were run on a machine with an Intel(R) Xeon(R) Gold 6326 (2.90 GHz), 128 GB RAM, running Ubuntu 20.04. All methods were evaluated on Kaggle and Random, while only xSystem, Deequ, and TFDV were tested on Enterprise due to industrial runtime constraints.

7.2 Performance Evaluation

Fig. 3 reports the average precision and recall of all methods on the three benchmarks. AutoPattern-rvf achieves an average precision of 0.91 and a recall of 0.89, consistently outperforming all baselines. On Kaggle, AutoPattern-v attains slightly higher precision, while AutoPattern-rvf yields higher recall, indicating a precision–recall trade-off introduced by fine-grained refinement. On Random, which contains nested structures,



(a) Kaggle: vary ρ (b) Random: vary ρ (c) Enterprise: vary ρ (d) Kaggle: vary K_{top}

(e) Kaggle: vary ρ (f) Random: vary ρ (g) Enterprise: vary ρ (h) Kaggle: vary K_{top}

(i) Random: vary K_{top} (j) Random: vary K_{top} (k) Enterprise: vary K_{top} (l) Enterprise: vary K_{top}

Fig. 4. Sensitivity of performance to sample rate ρ and K_{top}.

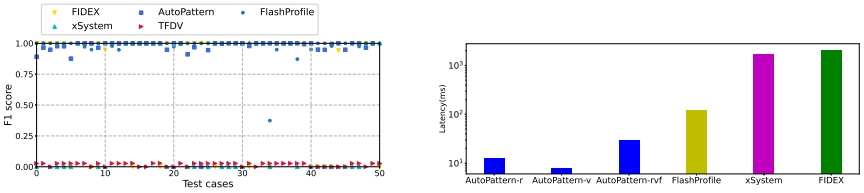
AutoPattern-rvf improves both precision and recall over AutoPattern-r and AutoPattern-v, as using a single splitting strategy yields incomplete structural coverage. On Enterprise, where strings are long, AutoPattern-rvf remains robust, while several baselines degrade or time out. Among baselines, FlashProfile performs well on Kaggle but degrades on Random due to limited inference; FIDEX times out on Enterprise due to DAG enumeration; and xSystem shows weaker results across datasets. Overall, these findings demonstrate the necessity of recursive and vertical skeleton extraction combined with fine-grained semantic refinement for validating semi-structured strings.

Sensitivity. We examined the impact of two parameters on AutoPattern's validation performance: (i) *sample rate* ρ, i.e., the fraction of per-key data used for training; and (ii) K_{top}, i.e., the number of learned patterns used for validation. As shown in Fig. 4, increasing sample rate and #Top-k generally improves precision (better coverage of historical variants) but slightly reduces recall (more patterns admit more erroneous variants), reflecting a natural precision–recall trade-off. Nevertheless, AutoPattern remains robust, exhibiting only mild sensitivity to these parameters.

Case-by-case analysis. We randomly sampled 50 cases (keys) and reported per-key F1 scores to compare methods at the individual case level. As shown in Fig. 5(a), pattern-based methods perform strongly, and our AutoPattern achieves consistently high F1 score across sampled cases.

7.3 Latency and User Effort

Fig. 5(b) reports the average end-to-end latency for pattern generation. In interactive settings, low latency is essential to maintain user experience. AutoPattern generates



(a) Kaggle: Case-by-case results (b) Average latency

Fig. 5. Performance and efficiency evaluation.

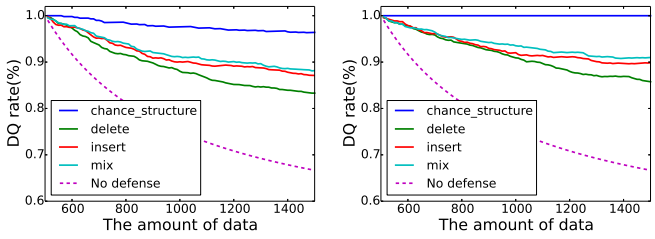
Table 1. Data augmentation evaluation

	threshold=0.8		threshold=0.9	
method	intervene(%)	train(%)	intervene(%)	train(%)
AutoPattern-sort$^{0-0.2}$	3.6	25.3	3.9	28.7
AutoPattern-random$^{0-0.2}$	0.0	1.1	0.0	1.5
dataAugment$^{0-0.2}$	3.0	21.0	3.0	22.4
AutoPattern-sort$^{0.2-0.8}$	8.6	33.3	8.6	36.0
AutoPattern-random$^{0.2-0.8}$	18.3	7.0	25.1	7.0
dataAugment$^{0.2-0.8}$	6.0	13.0	7.2	15.5
AutoPattern-sort$^{0.8-1.0}$	27.3	14.0	29.5	15.7
AutoPattern-random$^{0.8-1.0}$	7.8	1.3	9.1	2.1
dataAugment$^{0.8-1.0}$	15.1	11.1	17.5	13.3

patterns within a few seconds on average, keeping round-trip time acceptable even across multiple interaction cycles. To estimate manual effort, we invited 3 programmers (each with $\geq$3 years of data-engineering experience) to author patterns. After a short familiarization phase (several hours), they complete most pattern-writing tasks within a few minutes per key, though performance varied by individual and task complexity. In contrast, manual authoring requires iterative previewing and validation, whereas AutoPattern substantially reduces effort and supports rapid iteration.

7.4 Data Augmentation Evaluation

We evaluated how lightweight user feedback (as data augmentation) benefits AutoPattern in streaming insert-and-validate scenarios and quantify the required human intervention. Keys were grouped into three buckets by *uniqueness ratio* (fraction of unique values): $[0, 0.2)$, $[0.2, 0.8)$, and $(0.8, 1]$, with 10 keys sampled per bucket. We tested two initialization strategies: AutoPattern-sort (use the first 10 sorted values for training) and AutoPattern-random (use 10 random values), and compared them with dataAugment, which incorporates user feedback during the first three rounds of pattern generation. Table 1 reports, for each method and uniqueness bucket, the percentage of rounds requiring human intervention (intervene %) and training data (train %) needed to reach precision thresholds of 0.8 and 0.9. Data augmentation consistently reduces intervene % and train %, especially for mid-to-high uniqueness buckets (0.2–0.8 and 0.8–1.0). When uniqueness is very low (< 0.2), values are nearly enumerations and feedback offers limited benefit. Overall, early feedback expands the feature space and reduces manual effort during streaming validation.



(a) Gro_data dataset (b) Market dataset

Fig. 6. Data quality evaluation under streaming inserts.

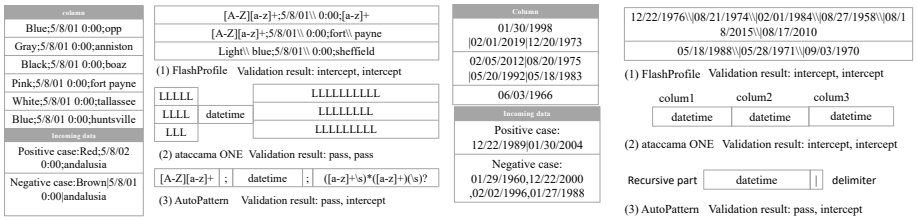
7.5 Data Quality Evaluation

We evaluated the impact of validation on long-term data quality (DQ) under streaming inserts. In unrestricted pipelines, dirty values accumulate over time and degrade DQ. DQ was measured per batch as the fraction of clean historical values among all observed values. In the simulation, we randomly sampled 500 verified clean rows as initial data and simulated streaming inserts in batches of size 10, each containing 5 clean (sampled from the initial data) and 5 corresponding corrupted values by (i) *structural perturbations* (e.g., delimiter changes), (ii) *character deletions*, and (iii) *character insertions*, as well as a mixed setting combining all types.

Figs. 6(a) and 6(b) plot DQ over batches under different error types, comparing an unprotected pipeline (no validation) with AutoPattern. Without validation, DQ drops quickly as dirty data accumulates, whereas AutoPattern maintains substantially higher DQ across all settings. Structural perturbations are the best detected due to explicit skeleton extraction, while character-level edits are also effectively blocked by the learned patterns. Overall, AutoPattern significantly slows DQ degradation, demonstrating practical benefits for ensuring data quality.

7.6 Case Study

To demonstrate real-world utility, we integrated AutoPattern into an industrial configuration platform at Ant Group Inc, which monitors about 2 million configuration changes with full history retained in a centralized store. We compared patterns learned by AutoPattern with two representative tools, FlashProfile [32] and Ataccama One [2], and their validation outcomes. As shown in Figs. 7(a) and 7(b), FlashProfile captures surface regularities but ignores internal nesting. Ataccama One identifies column splits and assigns predefined types (e.g., `datetime`), but often falls back to enumeration for unknown segments. In contrast, AutoPattern infers both structural skeletons and fine-grained semantics, generalizing to unseen positives while rejecting subtle negatives. Deployed for several months, AutoPattern has effectively intercepted configuration errors, improving incident detection and reducing diagnosis time. Compared with manual rule authoring, which required frequent rewrites under pattern drift, AutoPattern automates pattern discovery, reducing maintenance effort while maintaining high-quality validation on evolving data.



(a) Vertical skeleton data validation (b) Recursive skeleton data validation

Fig. 7. Case study at Ant Group Inc.

8 Related work

Structure extraction. Structure extraction transforms unstructured or semi-structured text into explicit structural forms. Existing methods primarily focus on specific formats such as log, JSON and XML [22,17,30]. Another line of work learns structural properties for data transformation [2,36].

Data validation. Recent systems employ machine learning to data validation, including Google's TFDV [14], Amazon's Deequ [38], and Auto-Validation [39]. Other studies address validation [34,12] and error detection in relational tables [35,11,42]. Beyond pattern-based validation, several approaches infer rich semantic types for validation [49,47,27], e.g., [47] synthesize type-detection logic from source code.

Data profiling. Data profiling has been extensively studied. Abedjan et al. [8] survey profiling techniques for relational database, covering statistical profiling, pattern profiling, and machine learning-based methods. Influential works on pattern profiling include inducing regular expressions from data [24,33,10] and learning domain-specific languages. Representative systems include Potter's Wheel [36], which ranks candidate patterns using the MDL principle, and FIDEX [43], which learns patterns from user-provided positive and negative examples to retrieve desired data. Additional studies on pattern profiling include [19,23].

9 Conclusion

We proposed AutoPattern, a self-validating data management system for complex semi-structured string data. Its main contributions include: (1) a domain-specific language supporting recursive and aligned structures; (2) a skeleton extraction algorithm combining recursive and vertical splitting; (3) a fine-grained semantic refinement method; and (4) adaptive mechanisms for data augmentation and incremental pattern updates. Extensive experiments on public, synthetic, and industrial datasets demonstrated the effectiveness and efficiency of AutoPattern.

Acknowledgment

This work was supported by NSFC 62472289, 62532007, and Guangdong Province Key Laboratory of Popular High Performance Computers 2017B030314073.

References

1. Kaggle, https://www.kaggle.com/
2. Ataccama one (2017), https://www.ataccama.com/
3. Excel (2023), https://www.microsoft.com/en-us/microsoft-365/excel
4. OpenRefine (2023), https://openrefine.org/docs
5. Power BI: Data Flow (2023), https://docs.microsoft.com/en-us/power-bi/transform-model/service-dataflows-create-use
6. SSIS (2023), https://docs.microsoft.com/en-us/sql/integration-services/control-flow/data-profiling-task?view=sql-server-ver15
7. Trifacta (2023), https://www.trifacta.com/
8. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. The VLDB Journal **24**(4), 557–581 (2015)
9. Agichtein, E., Ganti, V.: Mining reference tables for automatic text segmentation. In: KDD. pp. 20–29 (2004)
10. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987)
11. Berti-Équille, L., Harmouch, H., Naumann, F., Novelli, N., Saravanan, T.: Discovery of genuine functional dependencies from relational data with missing values. PVLDB **11**(8), 880–892 (2018)
12. Biessmann, F., Golebiowski, J., Rukat, T., Lange, D., Schmidt, P.: Automated data validation in machine learning systems. IEEE Data Engineering Bulletin (2021)
13. Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E., Zeller, A.: Where is the bug and how is it fixed? an experiment with practitioners. In: ESEC/FSE. pp. 117–128 (2017)
14. Breck, E., Polyzotis, N., Roy, S., Whang, S., Zinkevich, M.: Data validation for machine learning. In: MLSys (2019)
15. Chu, X., He, Y., Chakrabarti, K., Ganjam, K.: Tegra: Table extraction by global record alignment. In: SIGMOD. pp. 1713–1728 (2015)
16. Cortez, E., Oliveira, D., da Silva, A.S., de Moura, E.S., Laender, A.H.: Joint unsupervised structure discovery and information extraction. In: SIGMOD. pp. 541–552 (2011)
17. Du, M., Li, F.: Spell: Streaming parsing of system event logs. In: ICDM. pp. 859–864 (2016)
18. Elmeleegy, H., Madhavan, J., Halevy, A.: Harvesting relational tables from lists on the web. PVLDB **2**(1), 1078–1089 (2009)
19. Fernau, H.: Algorithms for learning regular expressions from positive data. Information and Computation **207**(4), 521–541 (2009)
20. Fisher, K., Gruber, R.: Pads: a domain-specific language for processing ad hoc data. ACM Sigplan Notices **40**(6), 295–304 (2005)
21. Fisher, K., Walker, D.: The pads project: an overview. In: ICDT. pp. 11–17 (2011)
22. Gao, Y., Huang, S., Parameswaran, A.: Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In: SIGMOD. pp. 943–958 (2018)
23. Golab, L., Karloff, H., Korn, F., Srivastava, D.: Data auditor: Exploring data quality and semantics using pattern tableaux. PVLDB **3**(1-2), 1641–1644 (2010)
24. Gold, E.M.: Complexity of automaton identification from given data. Information and control **37**(3), 302–320 (1978)
25. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: ICSE. pp. 3–13 (2012)
26. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)

27. Hulsebos, M., Hu, K., Bakker, M., Zgraggen, E., Satyanarayan, A., Kraska, T., Demiralp, Ç., Hidalgo, C.: Sherlock: A deep learning approach to semantic data type detection. In: KDD. pp. 1500–1508 (2019)
28. Ilyas, A., da Trindade, J.M., Fernandez, R.C., Madden, S.: Extracting syntactical patterns from databases. In: ICDE. pp. 41–52 (2018)
29. Kirschner, L., Soremekun, E.O., Zeller, A.: Debugging inputs. In: ICSE. pp. 75–86 (2020)
30. Klettke, M., Störl, U., Scherzinger, S.: Schema extraction and structural outlier detection for json-based nosql data stores. BTW (2015)
31. Kozielski, S., Wrembel, R.: New Trends in Data Warehousing and Data Analysis. Springer (2009)
32. Padhi, S., Jain, P., Perelman, D., Polozov, O., Gulwani, S., Millstein, T.: Flashprofile: a framework for synthesizing data profiles. PACMPL **2**(OOPSLA), 1–28 (2018)
33. Pitt, L., Warmuth, M.K.: The minimum consistent dfa problem cannot be approximated within any polynomial. Journal of the ACM **40**(1), 95–142 (1993)
34. Polyzotis, N., Roy, S., Whang, S.E., Zinkevich, M.: Data management challenges in production machine learning. In: SIGMOD. pp. 1723–1726 (2017)
35. Qahtan, A., Tang, N., Ouzzani, M., Cao, Y., Stonebraker, M.: Pattern functional dependencies for data cleaning. PVLDB **13**(5), 684–697 (2020)
36. Raman, V., Hellerstein, J.M.: Potter's wheel: An interactive data cleaning system. In: PVLDB. pp. 381–390 (2001)
37. Schelter, S., Biessmann, F., Lange, D., Rukat, T., Schmidt, P., Seufert, S., Brunelle, P., Taptunov, A.: Unit testing data with deequ. In: SIGMOD. pp. 1993–1996 (2019)
38. Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., Grafberger, A.: Automating large-scale data quality verification. PVLDB **11**(12), 1781–1794 (2018)
39. Song, J., He, Y.: Auto-Validate: Unsupervised data validation using data-domain patterns inferred from data lakes. In: SIGMOD. pp. 1678–1691 (2021)
40. Teoh, J., Gulzar, M.A., Xu, G.H., Kim, M.: Perfdebug: Performance debugging of computation skew in dataflow systems. In: SoCC. pp. 465–476 (2019)
41. Wang, L., Jiang, T.: On the complexity of multiple sequence alignment. Journal of Computational Biology **1**(4), 337–348 (1994)
42. Wang, P., He, Y.: Uni-detect: A unified approach to automated error detection in tables. In: SIGMOD. pp. 811–828 (2019)
43. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. ACM SIGPLAN Notices **51**(10), 195–213 (2016)
44. Wang, Z., Chen, T.P., Zhang, H., Wang, S.: An empirical study on the challenges that developers encounter when developing apache spark applications. J. Syst. Softw. **194**, 111488 (2022)
45. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.: Context-aware patch generation for better automated program repair. In: ICSE. pp. 1–11 (2018)
46. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Software Eng. **42**(8), 707–740 (2016)
47. Yan, C., He, Y.: Synthesizing type-detection logic for rich semantic data types using open-source code. In: SIGMOD. pp. 35–50 (2018)
48. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. **28**(2), 183–200 (2002)
49. Zhang, D., Suhara, Y., Li, J., Hulsebos, M., Demiralp, Ç., Tan, W.C.: Sato: Contextual semantic type detection in tables. PVLDB **13**(11), 1835–1848 (2020)
50. Zhang, Q., Wang, J., Gulzar, M.A., Padhye, R., Kim, M.: Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In: ASE. pp. 722–733 (2020)