SURAGH: Syntactic Pattern Matching to Identify Ill-Formed Records

Mazhar Hameed
mazhar.hameed@hpi.de
Hasso Plattner Institute, University of Potsdam
Germany

Gerardo Vitagliano
gerardo.vitagliano@hpi.de
Hasso Plattner Institute, University of Potsdam
Germany

Lan Jiang
lan.jiang@hpi.de
Hasso Plattner Institute, University of Potsdam
Germany

Felix Naumann
felix.naumann@hpi.de
Hasso Plattner Institute, University of Potsdam
Germany

ABSTRACT

Every day, data are produced from different sources with varying structures for multiple purposes. With increasing heterogeneity, the ability to manage it accurately and according to user requirements has become more challenging for downstream applications. Among other challenges, detecting ill-formed records in files is a difficult problem: ill-formed records occur in raw data due to loosely defined schemata, incorrect formatting of values, record structure discrepancy, etc. They can lead to aborted loading processes, incorrectly parsed data, and can interfere with the training process of machine learning algorithms.

We propose SURAGH[1], a method to detect ill-formed records without relying on external information, such as data types, record structure, or schemata. It parses an input file and generates syntax-based patterns for column values using order-dependent abstractions. We combine these into patterns for entire rows and use them to identify ill-formed records.

1 DATA LOADING OBSTACLES

In today's growing technology market, the value that data can add in terms of improved analytics, decision-making, and knowledge building needs no introduction. However, parsing and storing raw data without standardized formats harbors challenges, such as invalid characters due to incorrect parsing, preambles or comments, or inconsistent formatting that causes problems in data manipulation operations. Data scientists and machine learning engineers spend much of their development time cleaning and preparing data [10, 21, 29]. Existing data cleaning [4, 5, 16, 23, 25] and data preparation [9, 12, 14, 27, 31] techniques address data pre-processing tasks. However, improving user experience in designing and carrying out an automated data pre-processing pipeline still has open challenges.

Comma-separated value (CSV) files are one of the most commonly used formats [19], widely available on most open data portals [18], and frequently used in scientific projects [3, 12, 17]. Not only can these CSV files be used for exploring, collecting, and integrating data, but their use is also significant in other research areas, such as knowledge-based design, building machine learning models, and information mining [2, 20, 26, 28].

[1]SURAGH is an Urdu word that means to investigate an event; to obtain clues about something.

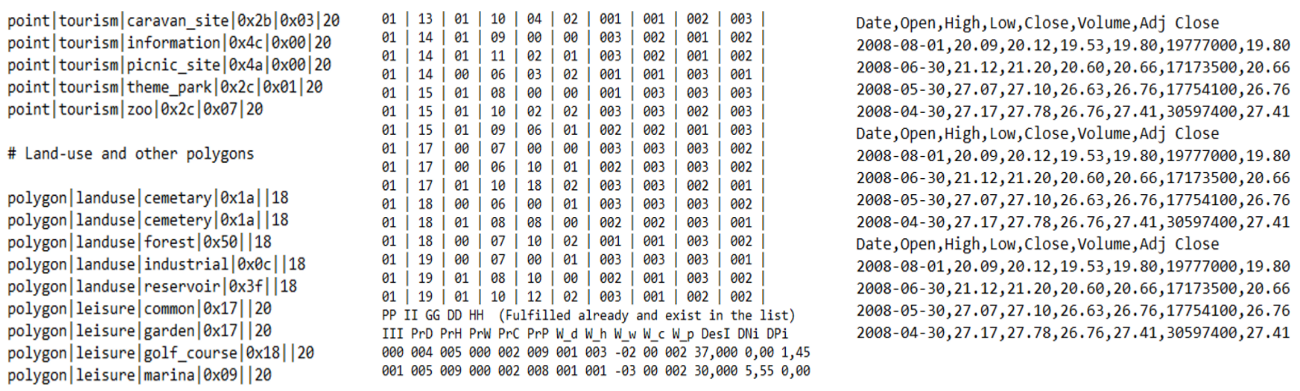
Despite the existing standard [11], data entry into CSV files is prone to errors because users and applications do not always adhere to this standard; moreover, the standard itself is rather loose. This behavior is also present in CSV files available on open data portals. Out of 2 066 files randomly selected from a government data portal[2], we were unable to directly load 418 (20.2%) of them into an RDBMS due to inconsistent records. We observe the same behavior with other tools, including business intelligence and data wrangling tools. We refer to such records as "ill-formed" (see Section 2.1 for a formal definition). These records do not adhere to the file's structural patterns and prevent data from being loaded, let alone being consumed by downstream applications. We expect that most readers have first-hand experience of a load-process aborting after many minutes due to some mistake, such as an incorrectly encoded value or a line too long towards the end of the input file. Another common experience is the need to "clean up" a file by removing preambles, footnotes, etc., before attempting to import the data into a database or other application. Our goal is to identify such problems in advance and alert users or machines about the problematic rows.

A record may be ill-formed due to inconsistencies that can exist at both column- and row-levels. *Row-level inconsistencies* occur when entire rows appear in the file but do not contain the expected data. For instance, such rows may contain metadata or comments, group headers, or are due to misplaced delimiter or end-of-line characters. Figure 1 shows several real-world examples. *Column-level inconsistencies* include inconsistent column values, such as incorrectly handled escape characters, string values in numeric columns, inconsistent formatting, a null or a missing value, etc. Figure 2 again shows several real-world examples.

While annotating our files, we identified common inconsistencies at both column and row-levels that cause well-formed records to become ill-formed records. At *column-level*, we observed multiple value formats, null values, out of range values, line breaks within a value, values non-compliant to a schema, values without quote characters but containing delimiters, non-escaped values, values with an incorrect escape character, etc. At *row-level* we observed comment rows, header rows, group header rows, aggregation rows, empty rows, footnote rows, misplaced delimiters, incorrect delimiters, inconsistent number of columns, and many other problems.

Eventually, such records can hinder or completely halt data loading and other data processing operations. Similarly, such

[2]https://www.data.gov/



(a) Comments in between data

```
point|tourism|caravan_site|0x2b|0x03|20
point|tourism|information|0x4c|0x00|20
point|tourism|picnic_site|0x4a|0x00|20
point|tourism|theme_park|0x2c|0x01|20
point|tourism|zoo|0x2c|0x07|20

# Land-use and other polygons

polygon|landuse|cemetary|0x1a||18
polygon|landuse|cemetery|0x1a||18
polygon|landuse|forest|0x50||18
polygon|landuse|industrial|0x0c||18
polygon|landuse|reservoir|0x3f||18
polygon|leisure|common|0x17||20
polygon|leisure|garden|0x17||20
polygon|leisure|golf_course|0x18||20
polygon|leisure|marina|0x09||20
```

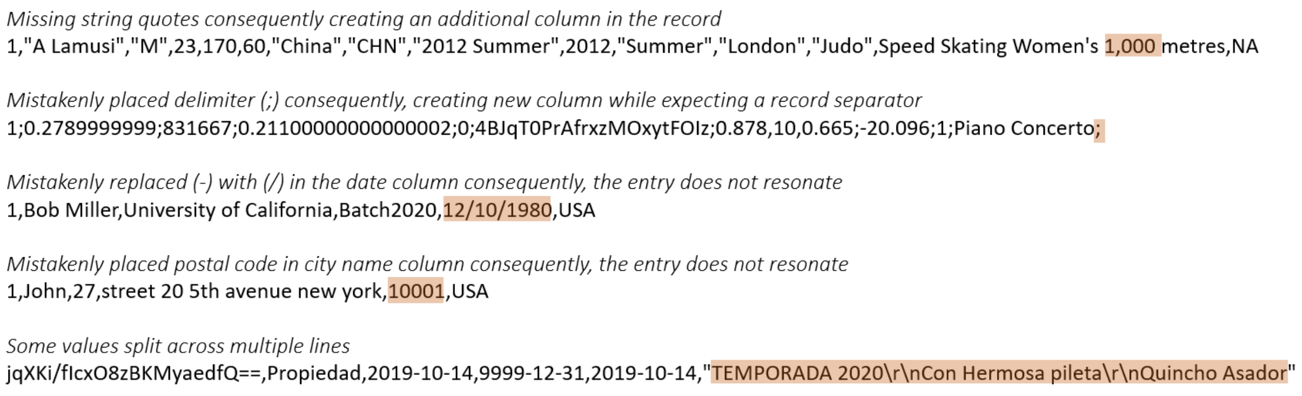
(b) Multi-delimited file

```
01 | 13 | 01 | 10 | 04 | 02 | 001 | 001 | 002 | 003 |
01 | 14 | 01 | 09 | 00 | 00 | 003 | 002 | 001 | 002 |
01 | 14 | 01 | 11 | 02 | 01 | 003 | 002 | 001 | 002 |
01 | 14 | 00 | 06 | 03 | 02 | 001 | 001 | 003 | 001 |
01 | 15 | 01 | 08 | 00 | 00 | 001 | 003 | 003 | 003 |
01 | 15 | 01 | 10 | 02 | 02 | 003 | 003 | 002 | 003 |
01 | 15 | 01 | 09 | 06 | 01 | 002 | 002 | 001 | 003 |
01 | 17 | 00 | 07 | 00 | 00 | 003 | 003 | 003 | 002 |
01 | 17 | 00 | 06 | 10 | 01 | 002 | 003 | 003 | 002 |
01 | 17 | 01 | 10 | 18 | 02 | 003 | 003 | 002 | 001 |
01 | 18 | 00 | 06 | 00 | 01 | 003 | 003 | 003 | 002 |
01 | 18 | 01 | 08 | 08 | 00 | 002 | 002 | 003 | 001 |
01 | 18 | 00 | 07 | 10 | 02 | 001 | 001 | 003 | 002 |
01 | 19 | 00 | 07 | 00 | 01 | 003 | 003 | 003 | 001 |
01 | 19 | 01 | 08 | 10 | 00 | 002 | 001 | 003 | 002 |
01 | 19 | 01 | 10 | 12 | 02 | 003 | 001 | 002 | 002 |
PP II GG DD HH  (Fulfilled already and exist in the list)
III PrD PrH PrW PrC PrP W_d W_h W_w W_c W_p DesI DNi DPi
000 004 005 000 002 009 001 003 -02 00 002 37,000 0,00 1,45
001 005 009 000 002 008 001 001 -03 00 002 30,000 5,55 0,00
```

(c) Header row repetition

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80
2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66
2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76
2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41
Date,Open,High,Low,Close,Volume,Adj Close
2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80
2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66
2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76
2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41
Date,Open,High,Low,Close,Volume,Adj Close
2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80
2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66
2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76
2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41
```

Figure 1: Examples of ill-formed records due to row-level inconsistencies



Missing string quotes consequently creating an additional column in the record

```
1,"A Lamusi","M",23,170,60,"China","CHN","2012 Summer",2012,"Summer","London","Judo",Speed Skating Women's 1,000 metres,NA
```

Mistakenly placed delimiter (;) consequently, creating new column while expecting a record separator

```
1;0.2789999999;831667;0.21100000000000002;0;4BJqT0PrAfrxzMOxytFOIz;0.878,10,0.665;-20.096;1;Piano Concerto;
```

Mistakenly replaced (-) with (/) in the date column consequently, the entry does not resonate

```
1,Bob Miller,University of California,Batch2020,12/10/1980,USA
```

Mistakenly placed postal code in city name column consequently, the entry does not resonate

```
1,John,27,street 20 5th avenue new york,10001,USA
```

Some values split across multiple lines

```
jqXKi/fIcxO8zBKMyaedfQ==,Propiedad,2019-10-14,9999-12-31,2019-10-14,"TEMPORADA 2020\r\nCon Hermosa pileta\r\nQuincho Asador"
```

Figure 2: Examples of ill-formed records due to column-level inconsistencies

records impose challenges for machine learning algorithms during data annotation, manipulation operations, and algorithm training phases. Manually identifying ill-formed records involves effort, domain expertise and is error-prone. We address this problem and propose an automatic solution, Suragh, to identify ill-formed records in a file by mapping column values to syntax-based patterns: syntactic patterns (see Definition 2.1). These patterns are easy to understand and represent data structures comprehensively. The syntactic patterns we introduce can assist the user in various tasks:

- **Data ingestion:** The core use-case of Suragh is to prevent frustrating experiences of raw data not loading into a host system.
- **Table detection:** Ill-formed records can cause ambiguity for table detection algorithms when detecting table boundaries. Using Suragh, we can remove these records and improve the accuracy of boundary detection. In addition, our approach can help users identify multiple vertically aligned tables in a file.
- **Quality assessment:** With the help of Suragh, users can pre-determine how messy the input files are and how much data preparation is needed.
- **Data standardization:** Syntactic patterns can serve as a basis for standardizing data into a uniform format, as they reflect typical problems in ill-formed records (outlier patterns) and desired structure (frequent patterns).

Our paper makes the following main contributions:

(1) A formalization to describe file schema, syntactic patterns, and ill/well-formedness of records.
(2) A set of 131 files from five open data sources, each annotated for ill-formed and well-formed rows for a total of 210 550 rows. The datasets are publicly available together with the annotations and code at the project page[3].
(3) A method, Suragh, that automatically recognizes ill-formed records by mapping column values to syntax-based patterns.
(4) A wide range of experiments conducted to validate Suragh and demonstrate the applicability of our approach.

The rest of the paper is organized as follows: Section 2 defines relevant concepts, provides a formal definition of ill/well-formed records, and describes the syntactic patterns grammar specified by EBNF rules [7]. Section 3 illustrates the workflow of the proposed algorithm and the role of pruning in different phases of the algorithm. Section 4 presents the experimental evaluation of Suragh. Section 5 discusses prominent research conducted in the field of pattern-based approaches, and Section 6 concludes our study and discusses future work.

[3]https://github.com/HPI-Information-Systems/SURAGH

2 PROBLEM DEFINITION

We first introduce the basic concepts involved in this work to then define the notion of syntactic patterns. We then define ill-formed and well-formed records, the grammar of syntactic patterns, and discuss the scope and challenges of the approach.

2.1 Ill-Formed and Well-Formed Records

The input to our approach is a *file*, which is composed of a number of records. A *record* is composed of horizontally aligned cells, where each cell belongs to a column and is separated by a delimiter. A *column* is composed of vertically aligned cells across records, where each cell contains a value (including the empty value). To distinguish ill-formed and well-formed records, we define a *pattern schema*: one or more ordered sequences of attributes, each attribute having a syntactic pattern (defined hereafter). A pattern schema can contain more than one ordered sequence because records even in well-structured tables might follow different patterns. For example, a column with user names might have a different number of tokens in each row.

We now define the central notion of a syntactic pattern:

Definition 2.1. Syntactic patterns are a sequence of symbols to represent characters of an input value. The production rules of Table 1 transform each input value to one or more weighted *syntactic patterns*, where weights reflect different levels of pattern abstraction.

For example, using our syntactic pattern grammar (presented in Section 2.2) two of the syntactic patterns for New York City zip codes (e.g., 10001) are $\langle D\rangle\langle D\rangle\langle D\rangle\langle D\rangle\langle D\rangle$ and $\langle SEQD\rangle$. We assign a higher weight to each abstraction of the pattern inversely proportional to its abstraction level (see Figure 3); thus, the abstraction "digit" $\langle D\rangle$ is given a higher weight than "sequence of digits" $\langle SEQD\rangle$. Suragh uses these weights to avoid generalization during its pattern selection and to prune highly abstracted patterns. However, $\langle SEQD\rangle$ and other higher-level abstractions are also crucial in the process when necessary, e.g., a column with an index number from 1 to 1 million; there, it is not trivial to find small set of patterns using low-level abstractions due to the many different cell values.

In general, we expect all the rows of a standard CSV file to conform to the same schema and thus contain values with the same syntactic patterns across columns. However, non-standard CSV files may include rows with different syntactic patterns, for example, if they contain multiple tables and therefore multiple schemata, or contain metadata rows, such as table titles, footnotes, etc.

Definition 2.2. A record *conforms to* a pattern schema if it has the same number of attributes as the schema, and all column values of the record conform to the corresponding column patterns of one of the attribute sequences of the pattern schema. We call such a record *well-formed* and *ill-formed* otherwise.

We now formally define our problem as follows: *Given an input file $F = \{r_1, r_2, \dots, r_m\}$ of m records, automatically generate its pattern schema and use it to classify each record as ill-formed or well-formed.*

Please note that we limit our solution to *syntactically* or structurally ill-formed records. Identifying *semantically* ill-formed records, e.g., rows with data errors, is beyond the goal of this research.

2.2 A Grammar for Data Rows

To construct syntactic patterns, we define a set of production rules to transform values and call them abstractions. Table 1 lists these abstractions with their representation and also shows the associated grammar. We specify the grammar using the extended Backus-Naur Form (EBNF) rules and notations [7]. The abstractions are of two types, namely (1) encoder and (2) aggregator. For a given value, the encoder abstractions map each character to a derived representation, while the aggregator abstractions that depend on encoder abstractions combine representations resulting from other encoder and aggregator abstractions based on a given rule. The application of these rules is order-dependent; for example, the "sequence of upper-case letters" $\langle SEQUL\rangle$ abstraction is an aggregator abstraction that is applicable only on representations of the "upper-case letter" $\langle UL\rangle$ abstraction. Figure 3 shows the dependency graph between abstractions, where abstractions without edges can be executed in any order. The abstractions shown in orange are encoders, while those shown in green are aggregators.

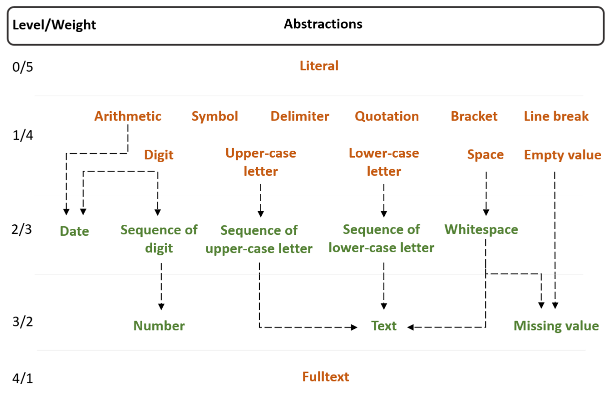


Figure 3: Abstractions dependency graph

For a given input file, Suragh generates, collects, and constructs syntactic patterns at several levels; syntactic value pattern, syntactic column pattern, and syntactic row pattern each serves a purpose in the process of detecting ill-formed records (see Section 3 for detailed definitions).

2.3 Scope and Challenges

Although CSV files are mainly comma-separated, we observe many CSV files with other delimiters, which are also in the scope of our work. In the context of our research, we do assume that a file contains structured data. Dealing with semi-structured and unstructured data is beyond the scope of this paper. Also, to limit search space size, our approach is designed for data files that contain ASCII values. The extension to larger character sets is conceptually easy but computationally expensive.

One main challenge we address is system efficiency. Our solution comprises several phases that require both time and memory optimization. We achieve efficiency by introducing pruning techniques to optimize the different phases of the algorithm (see Section 3).

3 THE SURAGH ALGORITHM

In this section, we describe the workflow of Suragh as depicted in Figure 4. Given an input file, Suragh returns a list of indices of ill- and well-formed records along with the dominant syntactic

Table 1: Abstractions with weight and associated grammar (EBNF notation)

Weight	Abstractions	Grammar
5	Literal	Represented by the character of the field itself
4	Delimiter⟨*DEL*⟩	"," | ";" | ":" | ?US-ASCII character 9? | "|"
	Upper-case letter ⟨*UL*⟩	"A" | "B" | "C" | "D" | "E "| "F" | ...... | "Z"
	Lower-case letter ⟨*LL*⟩	"a" | "b | "c" | "d" | "e" | "f" | .... | "z"
	Digit ⟨*D*⟩	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
	Space ⟨*S*⟩	?US-ASCII character 32?
	Quotation ⟨”⟩	’ " ’
	Arithmetic ⟨*ARITH*⟩	"*" | "+" | "-" | "/" | "%" | "=" | "<" | "> "
	Bracket ⟨*BRKT*⟩	"[" | "]" | "{" | "}" | "(" | ")"
	Symbol ⟨*SYM*⟩	"$" | "#" | "." | "?" | "@" | "\\" | "^" | " ‘ " | "~" | "_" | " ’ " | "&" | "!" | ’ " ’ | "," | ";" | ":" | ?US-ASCII character 9? | "|"
	Line break ⟨*LB*⟩	?US-ASCII character 10? | ?US-ASCII character 13?
	Empty value ⟨*EV*⟩	null
3	Sequence of upper-case letters ⟨*SEQUL*⟩	⟨*UL*⟩, {⟨*UL*⟩}
	Sequence of lower-case letters ⟨*SEQLL*⟩	⟨*LL*⟩, {⟨*LL*⟩}
	Sequence of digits ⟨*SEQD*⟩	⟨*D*⟩, {⟨*D*⟩}
	Whitespace ⟨*WS*⟩	⟨*S*⟩, {⟨*S*⟩}
	Date ⟨*DT*⟩	2*⟨*D*⟩,⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩, 4*⟨*D*⟩ | 2*⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩, 2 *⟨*D*⟩ | 4*⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩ | ⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩, ⟨*ARITH*⟩, 2 *⟨*D*⟩ | ⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩, ⟨*ARITH*⟩, 4*⟨*D*⟩ | ⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩,2 *⟨*D*⟩| ⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩, 4*⟨*D*⟩ | 4*⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩ | 4*⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩ | 4*⟨*D*⟩, ⟨*ARITH*⟩, ⟨*D*⟩, ⟨*ARITH*⟩, 2*⟨*D*⟩
2	Number ⟨*NUM*⟩	"+" | "-", ⟨*D*⟩ | ⟨*SEQD*⟩ OR ["+" | "-"], ⟨*D*⟩ | ⟨*SEQD*⟩, ("," | "."), ⟨*D*⟩ | ⟨*SEQD*⟩, { ("," | "."), ⟨*D*⟩ | ⟨*SEQD*⟩ }
	Text ⟨*TXT*⟩	⟨*UL*⟩ |⟨*SEQUL*⟩| ⟨*LL*⟩ | ⟨*SEQLL*⟩, (⟨*UL*⟩ | ⟨*SEQUL*⟩| ⟨*LL*⟩ | ⟨*SEQLL*⟩| ⟨*S*⟩| ⟨*WS*⟩ |" - "|" _" |" ’ "|" \\" | "/" |" . "|" &"), { ⟨*UL*⟩ | ⟨*SEQUL*⟩| ⟨*LL*⟩ | ⟨*SEQLL*⟩| ⟨*S*⟩| ⟨*WS*⟩ |" - "|" _" |" ’ "|" \\" | "/" |" . "|" &" }
	Missing value ⟨*MV*⟩	⟨*EV*⟩| | ⟨*S*⟩| | ⟨*WS*⟩| | ?US-ASCII character 9? | "Null" | "null" | "na" | "n/a" | "NA" | "N/A" | "NaN" |"nan" |" None "|" NONE "|’ "" ’
1	Fulltext ⟨*FTXT*⟩	Very long string e.g., cellValue.size() > 50

row patterns as its pattern schema. The workflow consists of four phases, and we discuss their functionalities in the following sections.

3.1 Pattern Modeling

In this phase, Suragh first leverages dialect detection to extract values and then applies our abstraction grammar to these values to generate syntactic value patterns. Then, value patterns are collected to form a set of syntactic column patterns.

3.1.1 Value extraction. To extract column values, we must first detect the dialect of the file. The dialect of a file specifies a set of characters that define the structure of a file. A CSV file dialect consists of a delimiter, a quote character, a quote escape character, and a record separator. Dialect detection is a well-known problem in academia [6, 8, 30] and industry: Suragh leverages the univocity parser[4] to identify a file's dialect and extract the individual column values. This dialect determines the scope of each column. We validated the detected dialects for all our files and found no incorrectly detected dialects.

[4]https://www.univocity.com/pages/about-parsers

3.1.2 Pattern generation. As mentioned, an ill-formed record contains inconsistencies at column- and/or row-level. Some of these inconsistencies result in different record structures, such as incorrect delimiter, while others result in inconsistent column values, such as values not compliant to a schema, making an ill-formed record different from a well-formed record. Extracting the intended pattern schema helps recognize these ill-formed records. For pattern schema detection, Suragh generates one or more syntactic value patterns for each individual cell value using the abstractions grammar.

Definition 3.1 (Syntactic value pattern). Given a column value, a syntactic value pattern vp is a sequence of literals or abstractions from that value. We represent abstractions with their acronyms, where each abstraction represents a single character or a group of characters. The set of syntactic value patterns for a column value is denoted with VP.

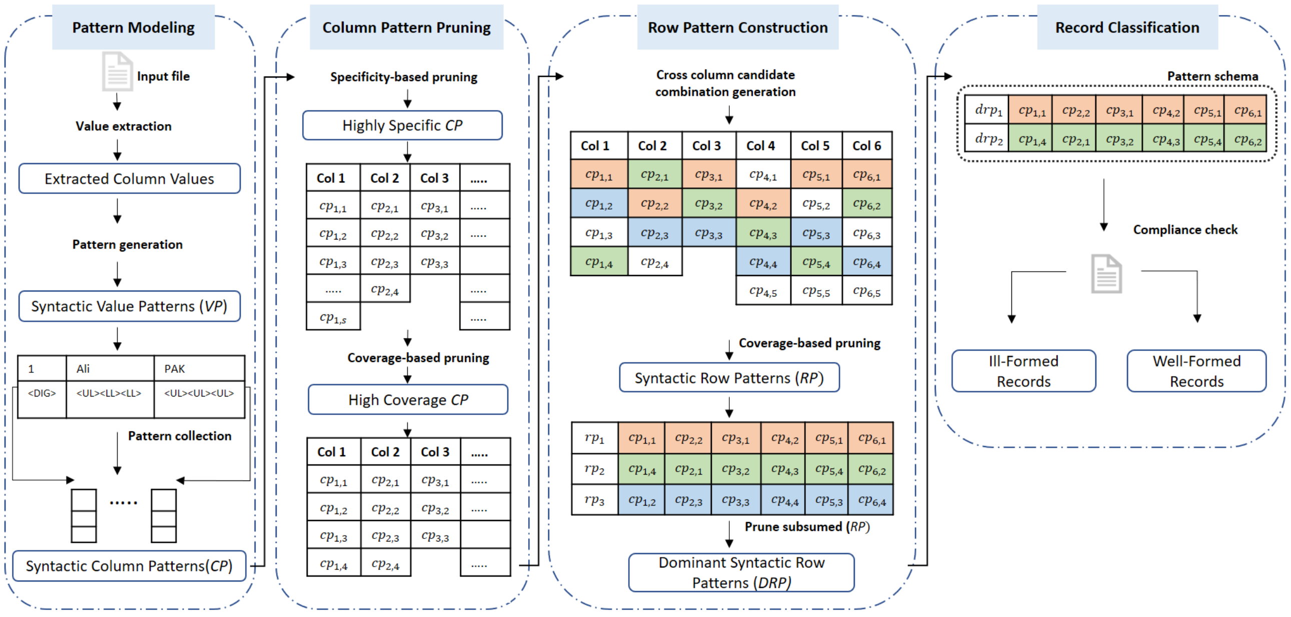
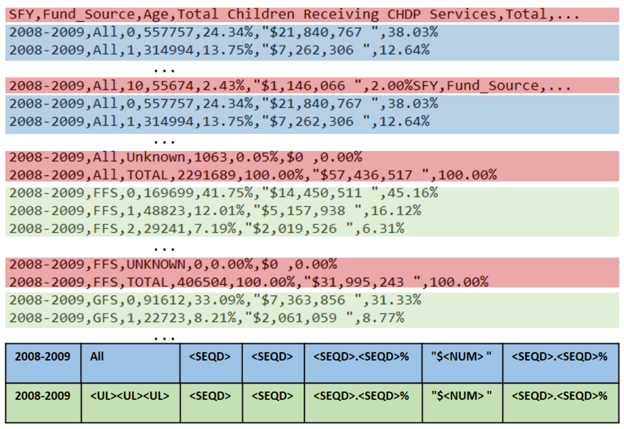


Figure 4: The workflow of Suragh

Algorithm 1: Syntactic value pattern generation

```
Input  : c: input column
         A: the set of abstractions
         G: the abstractions dependency graph
Output : $L_{VP}$ = the list of $VP$ that represents values in $c$
1  $L_{VP} \leftarrow []$
2  foreach $string \in c$ do
3      $VP \leftarrow \{\}$
4      $A \leftarrow$ getAbstractions($string$)
5      $PS \leftarrow$ pruneAbstractionCombinations(powerSet(A),
         G)
6      foreach $set \in PS$ do
7          $vp \leftarrow string$
8          foreach $abstraction \in set$ do
9              $vp \leftarrow$ executeAbstractions($abstraction$ , $vp$)
10         end
11         if $vp \notin VP$ then
12             $VP \leftarrow VP \cup \{vp\}$
13         end
14     end
15     $L_{VP}$.add($VP$)
16 end
17 return $L_{VP}$
```

The process of generating syntactic value patterns is presented in Algorithm 1. Suragh uses abstractions to map each character in each value of a column and generates a set of syntactic value patterns. Formally, let $F = \{r_1, r_2, \ldots, r_n\}$ be the input file of n records. Let $C = \{c_1, c_2, \ldots, c_m\}$ be the set of m columns. For each value in c_i, the algorithm first accesses the relevant abstractions $\{a_1, a_2, \ldots, a_t\}$ based on input value characters (line 4). Then, the algorithm creates a power set of the selected abstractions to capture all possible combinations of abstractions and uses them to generate all possible syntactic value patterns (line 5). After creating a power set, the algorithm prunes the subsets that violate the encoder-aggregator abstraction dependencies shown in Figure 3. For example, the algorithm prunes a subset if it contains "date" $\langle DT \rangle$ abstraction before "digit" $\langle D \rangle$ in the execution order. For each remaining subset, it maps each column value character to its derived representation from the abstractions to generate the set of value patterns $VP = \{vp_1, vp_2, \ldots, vp_u\}$ (lines 6-14). It repeats the same process for all column values and populates a list that contains the set of value patterns for each column value in c_i (line 15). For example, Figure 6(a) lists the set of value patterns for column value "`All`" in the input file F in Figure 5[5].

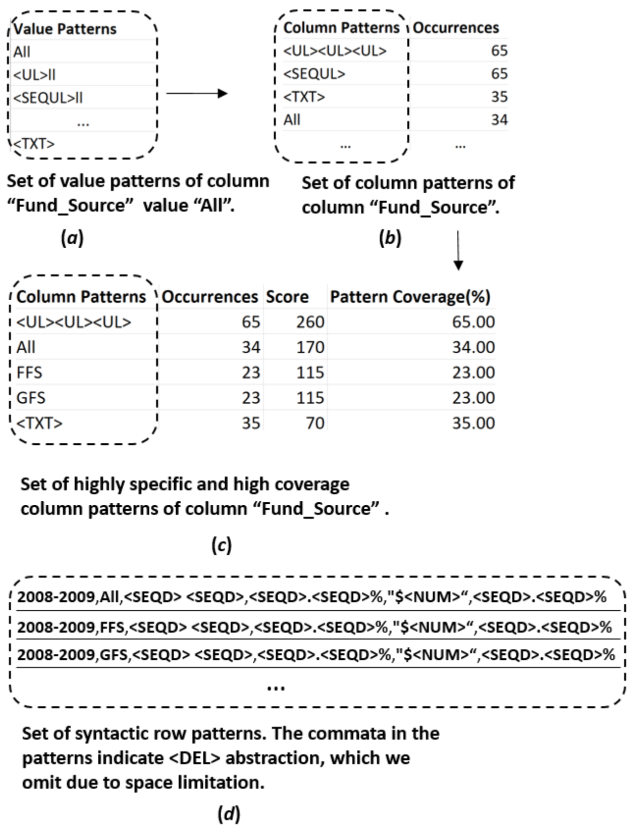


```
SFY,Fund_Source,Age,Total Children Receiving CHDP Services,Total,...
2008-2009,All,0,557757,24.34%,"$21,840,767 ",38.03%
2008-2009,All,1,314994,13.75%,"$7,262,306 ",12.64%
                  ...
2008-2009,All,10,55674,2.43%,"$1,146,066 ",2.00%SFY,Fund_Source,...
2008-2009,All,0,557757,24.34%,"$21,840,767 ",38.03%
2008-2009,All,1,314994,13.75%,"$7,262,306 ",12.64%
                  ...
2008-2009,All,Unknown,1063,0.05%,$0 ,0.00%
2008-2009,All,TOTAL,2291689,100.00%,"$57,436,517 ",100.00%
2008-2009,FFS,0,169699,41.75%,"$14,450,511 ",45.16%
2008-2009,FFS,1,48823,12.01%,"$5,157,938 ",16.12%
2008-2009,FFS,2,29241,7.19%,"$2,019,526 ",6.31%
                  ...
2008-2009,FFS,UNKNOWN,0,0.00%,$0 ,0.00%
2008-2009,FFS,TOTAL,406504,100.00%,"$31,995,243 ",100.00%
2008-2009,GFS,0,91612,33.09%,"$7,363,856 ",31.33%
2008-2009,GFS,1,22723,8.21%,"$2,061,059 ",8.77%
                  ...
```

2008-2009	All	<SEQD>	<SEQD>	<SEQD>.<SEQD>%	"$<NUM> "	<SEQD>.<SEQD>%
2008-2009	<UL><UL><UL>	<SEQD>	<SEQD>	<SEQD>.<SEQD>%	"$<NUM> "	<SEQD>.<SEQD>%

Figure 5: Selected lines of a 100-line file [5] with dominant syntactic row patterns (shaded blue and green) and ill-formed records (shaded red). The cell separators in the table indicate the $\langle DEL \rangle$ abstraction, which we omit due to space limitation.

[5]https://github.com/HPI-Information-Systems/SURAGH/blob/main/InputFile.csv



Value Patterns
All
<UL>ll
<SEQUL>ll
...
<TXT>

Set of value patterns of column "Fund_Source" value "All".

(*a*)

Column Patterns	Occurrences
<UL><UL><UL>	65
<SEQUL>	65
<TXT>	35
All	34
...	...

Set of column patterns of column "Fund_Source".

(*b*)

Column Patterns	Occurrences	Score	Pattern Coverage(%)
<UL><UL><UL>	65	260	65.00
All	34	170	34.00
FFS	23	115	23.00
GFS	23	115	23.00
<TXT>	35	70	35.00

Set of highly specific and high coverage column patterns of column "Fund_Source" .

(*c*)

```
2008-2009,All,<SEQD> <SEQD>,<SEQD>.<SEQD>%,"$<NUM>“,<SEQD>.<SEQD>%
2008-2009,FFS,<SEQD> <SEQD>,<SEQD>.<SEQD>%,"$<NUM>“,<SEQD>.<SEQD>%
2008-2009,GFS,<SEQD> <SEQD>,<SEQD>.<SEQD>%,"$<NUM>“,<SEQD>.<SEQD>%
...
```

Set of syntactic row patterns. The commata in the patterns indicate <DEL> abstraction, which we omit due to space limitation.

(*d*)

Figure 6: Value patterns, column patterns, highly specific and high coverage column patterns, and row patterns.

3.1.3 Pattern collection. Suragh collects the syntactic value patterns generated in the previous step column by column and uses them as column patterns.

Definition 3.2 (Syntactic column pattern). A syntactic column pattern cp is a syntactic value pattern that represents one or more column values in a column. The set of syntactic column patterns for a column is denoted with CP.

After collecting value patterns, Suragh forms a set CP of syntactic column patterns $\{cp_1, cp_2, \ldots, cp_v\}$. This set represents a column in a file, and each pattern in this set represents one or more cell values in that column. For example, for the input file F in Figure 5, Figure 6(b) lists the set of column patterns for column "`Fund_Source`".

3.2 Column Pattern Pruning

During syntactic pattern generation, the algorithm aims to apply every combination of abstractions to generate all possible value patterns for representing column values, requiring many system resources and affecting performance. We propose a cell value detection module that scans each column value and uses only relevant abstractions (see Figure 7). For example, for the value "`All`" in the input file F in Figure 5, the relevant abstractions are $\{\langle UL\rangle, \langle LL\rangle, \langle SEQUL\rangle, \langle SEQLL\rangle$, and $\langle TXT\rangle\}$.

In addition, the dependency graph between the abstractions guides the process of reducing the number of abstraction combinations, which optimizes the time expense of the pattern generation process because the algorithm generates fewer patterns. However, not every generated pattern is suitable for the ill-formed record detection due to its generality and/or coverage.

To select the highly specific and high coverage subset of column patterns from the generated patterns, we make use of two pruning techniques: (1) specificity-based pruning and (2) coverage-based pruning. We preserve *Highly specific* and *high coverage* column patterns after applying the above pruning techniques on the given column pattern set CP.

3.2.1 Specificity-Based Pruning. Suragh generates syntactic patterns with all possible abstraction combinations based on the content of the input values. These patterns represent column values from the most specific to the most generic way, depending on the pattern's abstractions. *The most specific column pattern* contains the fewest and the lowest level abstractions. In contrast, *the most generic column pattern* includes maximum and the highest level abstractions.

Suragh aims to choose column patterns that are highly specific in a column. To do so, it applies specificity-based pruning and leverages the weighting of abstractions, and uses the relationship between encoder and aggregator abstractions to prune generic column patterns. The goals of this phase are:

- Preserve the most specific column patterns.
- Reduce the search space for optimization.

Weighing criteria: Each abstraction a_i has a weight w based on its abstraction level (see Figure 3): the higher the abstraction level, the lower the weight. Suragh uses these weights to obtain a score for each column pattern cp. This score helps Suragh prune column patterns by identifying the significance of these patterns in a column. To obtain a score for each column pattern, Suragh scans through the column, counts its number of occurrences oc, and multiplies it by the mean value of the utilized abstractions weight:

$$Score(cp) = oc(cp) \; * \; \frac{1}{t}\sum_{i=1}^{t} w(a_i(cp)) \qquad (1)$$

Consider, as an example, that the value "`male`" appeared in a column 80% of the time. To represent this value, three of the possible value patterns using the abstraction grammar are $\{$*male*, $\langle LL\rangle\langle LL\rangle\langle LL\rangle\langle LL\rangle$, and $\langle SEQLL\rangle\}$. As a user, we would like to see the value *male* itself as one of the top column patterns for this column due to its specificity and high coverage and because this pattern with the literal value itself represents the column very well. Thus, we assign the highest weight to the *literal* abstraction due to its specificity to prioritize the patterns with literals. The second-highest weighting applies to level 1 abstractions that help identify highly specific patterns when the topmost patterns are not predominantly literal values. For example, if a column contains different values distributed across the cells, e.g., a column containing the US state alpha codes, i.e., AL, AK, AZ, etc. In this case, despite the highest weighting, the literal column patterns would be ranked low due to their occurrences, so abstraction level 1 and higher abstractions would serve the purpose better. For example, two of the possible column patterns for the mentioned column are $\{\langle UL\rangle\langle UL\rangle$ and $\langle SEQUL\rangle\}$. According to our definition, we assign $\langle UL\rangle\langle UL\rangle$ a higher weight and $\langle SEQUL\rangle$ a lower weight based on their abstraction level.

Pruning generic column patterns: As mentioned earlier, not every generated pattern is suitable. For example, in Figure 6(b), the column patterns $\langle UL\rangle\langle UL\rangle$ and $\langle SEQUL\rangle$ occur the same number of times and represent the same set of values. Therefore, we prune ⟨*SEQUL*⟩ since it is just a higher abstraction of the same values in this case. Note that occurrences is not the only criterion to prune these patterns. We also check the dependency between the encoder and the aggregator abstractions of these patterns and the pattern score. In the above example, ⟨*SEQUL*⟩ is dependent on ⟨*UL*⟩, and due to its higher abstraction, it also has a lower score.

Suragh first sorts column patterns by their score, starts with the highest-scored column pattern, compares it to all subsequent column patterns by examining their occurrences, checks encoder-aggregator dependency on those that meet the occurrence criterion, and removes those with higher abstractions. In this way, Suragh removes most generic patterns, as they are just another representation of specific patterns. However, there are other cases where a column contains many different values, and it is not possible to represent values with low-level abstractions due to a maximum variation, such as columns with addresses or website URLs. Therefore, despite the lower weight, due to the number of occurrences of higher-level abstractions, the top patterns of the columns Suragh returns are generic.

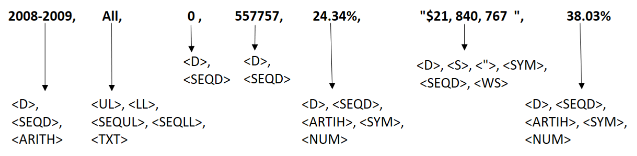

2008-2009,
All,
0 ,
557757,
24.34%,
"$21, 840, 767 ",
38.03%
<D>, <SEQD>
<D>, <SEQD>
<D>, <S>, <">, <SYM>, <SEQD>, <WS>
<D>, <SEQD>, <ARITH>
<UL>, <LL>, <SEQUL>, <SEQLL>, <TXT>
<D>, <SEQD>, <ARTIH>, <SYM>, <NUM>
<D>, <SEQD>, <ARTIH>, <SYM>, <NUM>

Figure 7: Cell value to abstraction mapping

3.2.2 Coverage-Based Pruning. In coverage-based pruning, we introduce a coverage threshold to prune column patterns at the column-level (column coverage threshold β) and column pattern combinations at the row-level (row coverage threshold γ, see Section 3.3.1). Here, *Column-level pattern coverage:* means how many column values are covered by a column pattern, and *Row-level pattern coverage:* implies how many records are covered by a combination of column patterns across all columns. In Section 4 we test combinations of seven different thresholds at both column- and row-level with a step size of 5, starting from 1% to 30% of the pattern coverage. Here, percentage implies the coverage of column patterns at column-level and column pattern combinations at row-level. The goals of this phase are:

- Preserve high coverage column patterns (column-level).
- Preserve high coverage column pattern combinations (row-level).
- Reduce the search space for optimization.

Column coverage threshold. Let β be the specified column coverage threshold; we prune all column patterns that provide column coverage less than or equal to the specified β. We use seven different thresholds for column pattern pruning. The goal is to select the set of high coverage column patterns using a global column coverage threshold.

For example, for the input file F in Figure 5, Figure 6(c) shows the highly specific and high coverage column patterns for column "`Fund_Source`" that we obtain after specificity- and coverage-based pruning, where β = 20%. For the column coverage threshold of 25%, the remaining column patterns for the mentioned column are {⟨*UL*⟩⟨*UL*⟩⟨*UL*⟩, *All*, and ⟨*TXT*⟩}.

3.3 Row Pattern Construction

In this phase of Suragh, we use the highly specific and high coverage column patterns from the column pattern pruning phase and first construct the set *RP* of syntactic row patterns $\{rp_1, rp_2, \ldots, rp_y\}$ that represents records. We then select the set *DRP* of dominant syntactic row patterns $\{drp_1, drp_2, \ldots, drp_z\}$ by pruning the set of syntactic row patterns to generate the pattern schema for the input file.

3.3.1 Coverage-Based Pruning. Similar to selecting column patterns with high coverage at the column-level, the goal here is to select the combinations of column patterns with high coverage when constructing row patterns, which leads us to select high coverage row patterns.

Row coverage threshold. Let γ be the specified row coverage threshold; we prune all cross column candidate combinations that provide row coverage less than or equal to the specified γ. We use the same thresholds as at the column-level to prune column patterns combinations during row pattern construction.

For example, for the input file F in Figure 5, a combination of column patterns {*2008-2009* and *FFS*} covers 23% rows for the columns {"`SFY`" and "`Fund_Source`"} (see Figure 6(c) for column "`Fund_Source`" column patterns). For the row coverage threshold of 25%, Suragh prunes this combination and stops the cross-column combination generation process for this candidate.

3.3.2 Cross column candidate combination generation. For row pattern construction, Suragh creates all combinations of column patterns across all columns while pruning based on coverage (γ). Here, *coverage* is the number of records corresponding to a combination of column patterns across all columns. Furthermore, the algorithm generates column pattern combinations for all columns and constructs all row patterns with high coverage.

Definition 3.3 (Syntactic row pattern). A syntactic row pattern rp is an ordered sequence of (highly specific and high coverage) column patterns that represents one or more records. The set of syntactic row patterns for a file is denoted with *RP*.

The process of constructing the set of syntactic row patterns is presented in Algorithm 2. Formally, let $\{c_1, c_2, \ldots, c_m\}$ be the set of columns and let $CP_i = \{cp_{i,1}, \ldots, cp_{i,v}\}$ be the corresponding set of highly specific and high coverage column patterns cps for a column c_i. Suragh first starts comparing CP_i and CP_{i+1}, and then progressively moves towards CP_m (lines 2-13). For each pair of columns, Suragh generates cross-column combinations by comparing each $cp \in CP_i$ with each $cp \in CP_{i+1}$ governed by coverage-based pruning (lines 4-11). *Intersection* (line 6) computes the record coverage of a specified cp combination, and the *coverage* check (line 7) allows cp combinations above the specified threshold γ. Then, *combinePattern* concatenates the cp combination, and stores it into a column combination set (line 8). This column combination set contains cp combinations across columns, and serves as new input after each iteration (line 12). The algorithm repeats the same process and creates all possible column pattern combinations until c_m and test them against the coverage threshold. To avoid the multiplicative nature of this approach, Suragh systematically grows row patterns by one column at a time in a branch & bound fashion. It prunes those intermediate patterns that already violate the coverage threshold γ and returns the set of row patterns (line 14). For example, for the input file F in Figure 5, Figure 6(d) lists the set of syntactic row patterns.

Algorithm 2: Cross column candidate combination generation:

```
Input  : $\{CP_1, CP_2, \ldots, CP_n\}$, row coverage threshold $\gamma$
         (see Section 3.2.2)
Output : $RP$ = set of syntactic row patterns
1  $RP \longleftarrow CP_1$
2  foreach $CP_i$ , $i > 1$ do
3      $CC \longleftarrow \{\}$
4      foreach $cp_r \in RP$ do
5          foreach $cp_c \in CP_i$ do
6              $PC \longleftarrow intersection(cp_r, cp_c)$
7              if $coverage(PC) > \gamma$ then
8                  $CC \longleftarrow CC \cup combinePattern(cp_r, cp_c)$
9              end
10         end
11     end
12     $RP \longleftarrow CC$
13 end
14 return $RP$
```

3.3.3 Prune subsumed syntactic row pattern. The set of syntactic row patterns that Suragh constructs may contain one or more row patterns, where one row pattern may represent records that are also represented by another row pattern, resulting in pattern redundancy. To avoid this redundancy, Suragh detects and removes the row patterns that are subsumed by any other row patterns.

Definition 3.4 (Pattern subsumption). A row pattern $rp_i \in RP$ is subsumed by another row pattern $rp_k \in RP$ if the set of all records it represents is a proper subset of the set of those represented by rp_k.

Definition 3.5 (Syntactic row pattern dominance). A dominant syntactic row pattern drp is a syntactic row pattern that is not subsumed by any other syntactic row pattern. The set of dominant syntactic row patterns for a file is denoted with DRP.

3.4 Record Classification

In the final phase of Suragh, we utilize the constructed set of dominant syntactic row patterns from the row pattern construction phase to generate a pattern schema for the input file and classify records based on this schema. First, Suragh accumulates dominant row patterns obtained from the row construction phase and generates pattern schema for the input file. Then, Suragh checks each record whether it conforms to the generated schema.

Suragh classifies non-conforming records as ill-formed whereas conforming records as well-formed. For example, Figure 5 shows the input file, the set of dominant syntactic row patterns that form its pattern schema, the records that comply with the schema, and the non-compliant (ill-formed) records.

4 EXPERIMENTS

First, this section describes the datasets used in our experiments, the dataset selection process, and the annotation criteria. We then present our experimental evaluation of Suragh, including the global threshold selection and its results for each dataset. Finally, we present a comparison between Suragh and state-of-the-art syntactic pattern approach and line classification approach and conclude the section with an error analysis.

4.1 Datasets and Annotation

This section lists the datasets used in our evaluations and specification of the annotated ill- and well-formed records.

4.1.1 Datasets. We conducted our experiments with datasets collected from five different open data sources: DataGov, Mendeley, GitHub, UKGov, NYCData. The statistics for each data source are summarized in Table 2. The files of each source have at least some column and/or row-level inconsistencies. To find files with inconsistencies, we first randomly selected files from each source and manually loaded each into an RDBMS. If the data loading process aborted, we kept the file as one of our test files. Later on, we utilized the same RDBMS along-with data wrangling and business intelligence tools to investigate where loading operations are interrupted and used this information to annotate records in our files. To render manual annotation of individual lines feasible, we further reduced the number of files by manually selecting one file from each group of similarly structured files. As a result, we have a unique set of diverse files for each source. Here, uniqueness implies that the files either do not have the same schema or, if they do have the same schema, they have different ill-formed records.

The UKGov and GitHub dataset files were taken from related work on dialect detection [30]. We randomly selected 1 000 CSV files from both of these datasets and manually checked each file by loading it into the RDBMS and selected the unique files with ill-formed records, which gives us 23 and 25 files from UKGov and GitHub, respectively. The numbers 23 and 25 are what were left after only unique files were selected. For the DataGov dataset, we crawled CSV files from www.data.gov. From the randomly selected 2 066 files, there were 418 files with ill-formed records that we observed by loading them into the RDBMS. We then used 40 manually selected unique files for our experiments. Similar to UKGov and GitHub, the number 40 is what was left after only unique files were selected. Our Mendeley dataset was created by selecting files from data.mendeley.com where files are grouped by research projects. We crawled all 2 214 projects and selected the lexicographical first 170, which already displayed a large variety of ill-formed records. These projects contain not only CSV files but also other formats, such as .XML, .XLSX, etc. We kept only projects with at least one CSV file and manually selected 33 unique files for our experiments, each file coming from a different project. Similar to other datasets, the number 33 is what was left after only unique files were selected. NYCData dataset includes a random subset of files downloaded from opendata.cityofnewyork.us and was kept unseen in the development of Suragh to ensure generalizability. Note that similar to other datasets, we have ensured file uniqueness for the unseen dataset as well.

Moreover, we utilized a command line tool[6] to check whether a file is compliant to RFC 4180. The purpose is to include both standardized and non-standardized files in our datasets and emphasize that also standard files can stop the data loading process, for instance, due to values not compliant to schema. Despite the inconsistent records, we manually checked each file and ensured that each selected file contained at least 50% non-empty values at both column and row-levels to avoid empty values and missing values ($\langle EV \rangle$, $\langle MV \rangle$) to dominate the pattern search.

The difference in the number of records per file is significant, as shown by the high standard deviation in Table 2, for two

[6]https://github.com/Clever/csvlint/

reasons: 1) A few of the files in our datasets are very large. 2) To ensure a unique set of files, we selected only one file from sets of similar files, resulting in a different number of records per file.

Table 2: The dataset overview, number of records (R), ill-formed (I), well-formed (W), each with the standard deviation in the brackets.

Source	# files	Avg # records	Avg # ill-f.	Avg # well-f.
DataGov	40	1143.2 (2060.6)	70.9 (165.9)	1072.3 (2014.1)
Mendeley	33	2688.8 (3664.3)	56.8 (109.1)	2632.0 (3636.1)
GitHub	25	791.9 (1094.6)	110.6 (300.8)	681.4 (963.5)
UKGov	23	1548.1 (3179.2)	98.7 (202.1)	1449.4 (3076.1)
NYCData	10	2068.7 (3159.4)	713.0 (1970.3)	1355.7 (1507.8)

4.1.2 Data annotation. We annotated each record in each file as ill- or well-formed and created a ground truth of 210 550 records across all datasets. As part of the annotation process for Suragh, we first checked each file in our dataset by loading it into three commercial tools from different categories: (i) an RDBMS, (ii) a data wrangling tool, and (iii) a business intelligence tool. We observed that all tools recognized the dialect for files that conformed to the RFC 4180 standard [11], but the RDBMS aborted the data loading process due to the ill-formed records in each file. We observed different behavior with the data wrangling and business intelligence tools for standardized files: in most cases, the tools successfully load our files correctly by simply mending the cell value, such as correcting the date format type based on the common pattern in a column. However, in some cases, this automatic correction led to incorrect results, such as assigning the most generic data type (string) to a column in the case of values that do not match a schema.

We then manually identified column-level and row-level inconsistencies in these ill-formed records (all encountered inconsistencies are listed in Section 1). Based on these findings, we manually determined the most specific syntactic patterns for each given column in the file and manually labeled them as part of the pattern schema. Finally, we annotated each record as well-formed if it conforms to the manually labeled pattern schema and ill-formed otherwise.

In addition, for most files that do not conform to the RFC standard, the RDBMS does not recognize the correct dialect and loads all the data into one column. For the few files where the RDBMS did recognize the dialect, it aborted the data loading process due to various issues, such as incorrect quotation marks, misplaced delimiters, inconsistent number of columns, etc. While the data wrangling and business intelligence tools, due to their built-in intelligence, do recognize the correct dialect in most cases, they may still load data incorrectly: for instance, they might assign the most generic data type (string) to each column or add new columns due to an incorrect splitting, or by simply deleting records they could not parse. A few cases where these tools do not recognize the correct dialect load all the data into one column, similar to the RDBMS.

For these non-standardized files, we manually checked each record and performed the same process as above to obtain its pattern schema and annotate ill- and well-formed records.

Note that for our study, we annotated the header row in all CSV files as ill-formed: We cannot assume that all tools have a built-in feature to distinguish header rows from data rows, or at least provide a user with a feature to mark the first row as a header row so that it does not become a part of the data.

4.2 Performance Evaluation

This section first presents the precision-recall metric and the performance evaluation of Suragh using the set of coverage-based thresholds at both column- and row-level. Then, it presents how we determined the global threshold setting and demonstrates the results for each dataset, including the unseen data (NYCData), which we used to test the generalizability of Suragh. Finally, this section presents the runtime analysis of Suragh.

4.2.1 Precision-recall metric. As per Definition 2.2, a record in a file is ill-formed if it does not conform to the pattern schema of that file. We call a detected ill-formed record *true* if its corresponding record in the ground truth is labeled as ill-formed, and *false* if it does not. To show the effectiveness of our approach, we use the precision and recall metrics:

$$P = \frac{|\textit{true ill-formed records detected}\,|}{|\textit{true \& false ill-formed records detected}\,|} \tag{2}$$

$$R = \frac{|\textit{true ill-formed records detected}\,|}{|\textit{total ill-formed records}\,|} \tag{3}$$

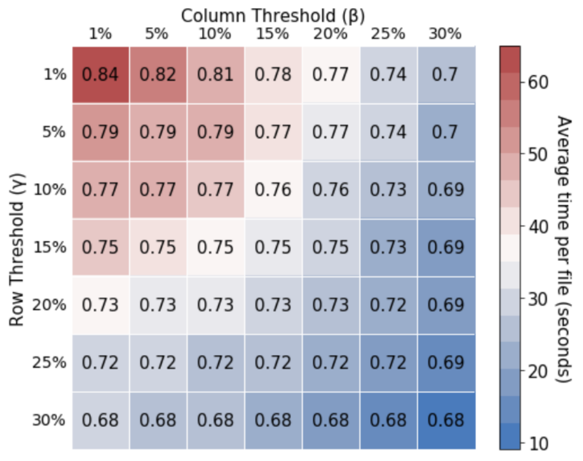
4.2.2 Threshold setting evaluation. To evaluate Suragh, we test combinations of seven different threshold values for both column- and row-level thresholds presented in Section 3.2.2, leading to 49 experiments for each dataset. The CSV files do not always have an equal number of ill- and well-formed records, so choosing experiments at the record-level across all files would bias the results toward large files. To avoid this bias, we conducted our experiments at the file-level. We compared the set of ill-formed records detected by Suragh with the set of true ill-formed records of ground truth for each CSV file. Figures 8a, 8b, and 8c show precision, recall, and F-1 scores, respectively, for each threshold-combinations where the color codes reflect the average runtimes associated with each threshold-combination. Each cell reports the average score across all files of all datasets (excluding the unseen dataset NYCData) for a combination of column- and row-level thresholds. As expected, we can observe that low thresholds lead to less pruning and, thus, to higher runtime.

For the first threshold, we use 1% since 0% pattern coverage is meaningless. We kept a maximum threshold of 30% for two reasons: 1) we achieved the highest recall by retaining only patterns with the highest coverage 2) precision decreased due to many false positives.

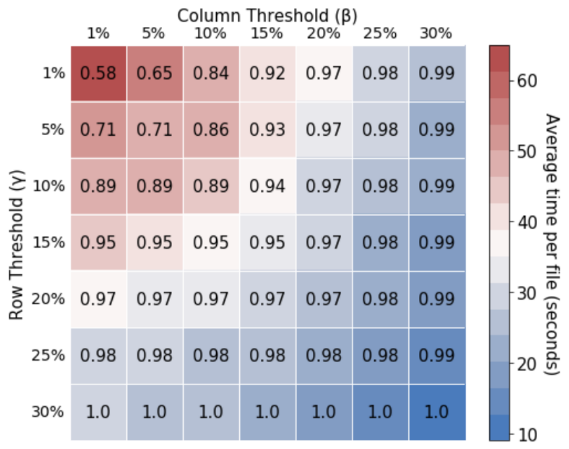
The goal of these experiments was to determine the global threshold setting (column- and row-level thresholds combination) and provide a user with a tool including built-in threshold setting.

The F-1 score shows an increasing trend at the beginning and a peak in the middle of the threshold setting, which then decreases as the threshold step size increases. The decreasing trend of the F-1 score from the middle of the threshold setting to both ends is due to the lowest recall at the minimum threshold setting and the lowest precision at the maximum threshold setting. The threshold setting (β = 20%, γ = 1%) is the one with the highest F-1 score, which we tested on unseen data and can thus recommend as a global setting.

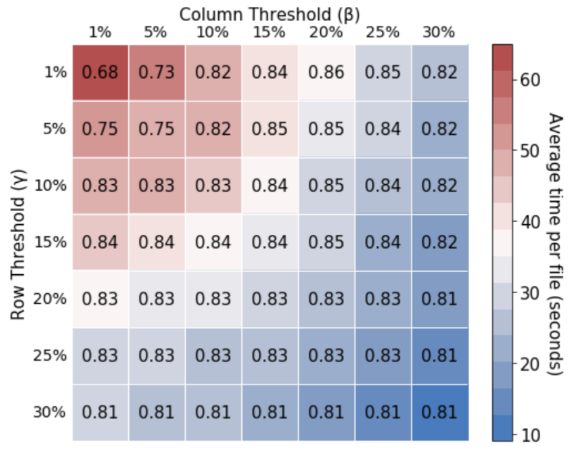
We observed that in our datasets, most column patterns covering $\leq$ 20% of cell values, and column pattern combinations covering $\leq$ 1% of records, refer to ill-formed records. Since we first apply the column-level threshold β, the larger step size of β perfectly balances with the smaller step size of the row threshold γ, as most of the patterns that might refer to ill-formed records



(a) Precision score



(b) Recall score



(c) F-1 score

Figure 8: Average precision, average recall, and average F-1 score across all datasets. The color code shows runtimes (same for all three plots).

are already pruned after applying the formal threshold. This behavior is reflected in Figures 8a and 8b where precision decreases by about 7%, but the recall score increases significantly by about 39% between $\beta = 1\%$ to 20% when $\gamma = 1\%$. Table 4 shows average precision, average recall, and F-1 score Suragh obtained for all datasets using the best threshold setting found (global threshold setting).

4.2.3 *Unseen dataset.* To test the generalizability of Suragh, we applied our approach on the unseen dataset NYCData. Table 4 shows the results achieved by Suragh using the global threshold setting as determined using the other four datasets. Similar to the performance on all the other datasets, Suragh achieved a recall of over 90% and a precision of 70%.

4.2.4 *System efficiency.* We achieve classification times of less than 9 milliseconds per row, on average with the global threshold setting. All experiments are performed on a 4-core Intel Core i7 2.3G CPU with 16GB RAM. Figure 9 shows the runtime of Suragh for the files in our dataset. While we observe a quadratic scalability overall, variance is quite high due to the quite different pattern complexity of individual files. The runtime increase is expected due to the increase in cross-column combinations in Algorithm 2, which depends on #columns and #rows; adding more columns leads to more cross-column combinations. Adding more rows can dominate the runtime as these combinations are integrated to construct row patterns, where #rows can influence the row coverage threshold, resulting in less pruning and thus more computation time.

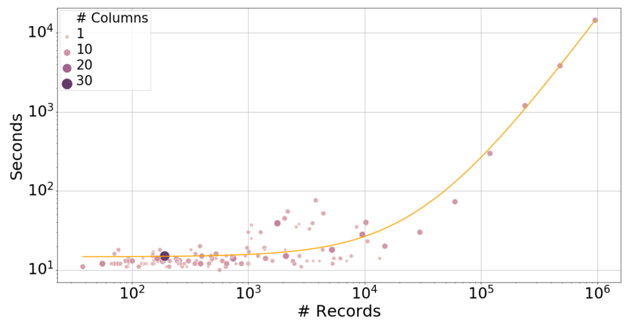


Figure 9: Ill-formed record detection efficiency, with a fitted quadratic curve. The last six files were obtained by extending existing files with duplicate records.

4.3 Comparative Analysis

There is no other approach dealing specifically with the detection of ill-formed records at the syntactic level. However, FAHES is a syntactic pattern-based approach to detect disguised missing values (DMVs) [23], with which we can simulate our task. The authors categorize DMVs into the following cases: (1) out of range data values, e.g., missing values disguised with a negative value for an attribute with positive values; (2) outliers, e.g., missing values disguised as very large values; (3) strings with repeated characters, e.g., replacing a phone number with 5555555555; (4) values with non-conforming data types, e.g., disguising the missing strings with numerical values; (5) valid values that are randomly distributed within the range of the data. The proposed approach uses a syntactic outlier detection module for Cases 1, 3, and 4 to detect those DMVs that are syntactic outliers or contain special patterns. The approach also uses a numerical outlier detection module for Cases 1 and 2, and follows missing-completely-at-random (MCAR) or missing-at-random (MAR) models [1, 15] for Case 5. For our experiments, we use the FAHES demo [24][7].

[7]https://github.com/daqcri/Fahes_Demo/

To compare FAHES with our approach, we consider records that, according to FAHES, contain disguised missing values as ill-formed, otherwise as well-formed. Since our approach is broader, the files in our datasets include not only the inconsistencies that FAHES deals with but many others, so not every file could be parsed by FAHES during our experiments. Of the files that FAHES successfully parsed, we excluded those with only header and missing values as inconsistencies because FAHES does not consider these cases. Table 3 shows the average precision, average recall, and F-1 score obtained by FAHES and Suragh with the global threshold setting. Note that we set the precision score to 1 when FAHES returned no DMVs.

Table 3: FAHES comparison overview

Source	# files	# files used	FAHES [23]			Suragh		
			P	R	F-1	P	R	F-1
DataGov	40	20	0.35	0.27	0.30	0.87	0.89	0.88
Mendeley	33	3	0.00	0.00	0.00	1.00	1.00	1.00
Github	25	11	0.70	0.22	0.33	0.80	0.92	0.86
UKGov	23	19	0.37	0.10	0.16	0.84	0.99	0.91
NYCData	10	8	0.63	0.06	0.11	0.81	0.93	0.87

Another system to which we compared our approach is the multi-class random forest classifier approach Strudel for CSV file line classification [12]. Strudel divides rows into six semantic classes: metadata, group, header, data, derived, and notes. The approach uses a set of features: content, context, and computational features to model the individual classes to classify rows in a CSV file. Strudel is a supervised learning approach, so for our experiments, we trained on the datasets available at the Strudel project page[8] and tested on the datasets listed in Section 4.1.1. We chose the Strudel datasets for training, because they contain a large set of files than the datasets we used for Suragh, and the authors tested their approach on out-of-domain datasets. Moreover, we used the same parameter settings as [12], except for the dialect detection tool, which we replaced with the univocity parser.

To compare Strudel with our approach, we consider the rows classified as *data* by Strudel to be well-formed and the remaining five classes as ill-formed. Table 4 shows the average precision, average recall, and F-1 score obtained by Strudel and Suragh with the global threshold setting. Strudel's recall is lower for two main reasons: (i) If there are errors in the data block, Strudel cannot detect them due to the scope of the project, as it does further distinguish erroneous data rows from correct data rows. (ii) In our file collection, some files have metadata and footnote rows with the same number of fields as data rows, making it difficult for Strudel to distinguish these non-data rows from data rows that authors also reported in their paper. Note that we set the precision score to 1 when Strudel returned all rows as data, thus not misclassifying any outlier-row.

Table 4: Strudel comparison overview

Source	# files	# records	Strudel [12]			Suragh		
			P	R	F-1	P	R	F-1
DataGov	40	45 728	0.98	0.21	0.34	0.80	0.93	0.86
Mendeley	33	88 729	0.97	0.42	0.58	0.80	0.98	0.88
Github	25	19 799	0.93	0.26	0.40	0.78	0.96	0.86
UKGov	23	35 607	0.90	0.22	0.35	0.71	0.99	0.83
NYCData	10	20 687	1.00	0.04	0.08	0.70	0.95	0.81

[8]https://hpi.de/naumann/s/strudel

4.4 Error Analysis

Not every ill-formed record follows a unique pattern in a file. Similarly, a well-formed record might follow a unique pattern, confusing Suragh to treat it as ill-formed. We present an analysis of selected detection errors made by Suragh.

4.4.1 False positive cases. Our global threshold settings achieved an average precision of 76% across all datasets. To investigate the false positive cases for all datasets, we manually checked each file. We found that Suragh does not accurately classify a record when a correct pattern is less frequent. Consider a column that stores employee names comprising mostly two tokens (first-name and last-name). Thus, the top patterns for this column will be for two words. If an employee has a name with three tokens, this record might be marked as ill-formed and thus be a false positive.

Another case of a less frequent pattern is a column containing values with long decimal numbers, such as 0.0347414562 and $7.6389273017e-005$. Data-driven systems load them correctly by identifying their type; therefore, we include both patterns as well-formed in our annotations. However, as very few values are represented in the scientific notation "e", Suragh prunes this pattern, resulting in false positives.

4.4.2 False negative cases. Our global threshold settings were able to achieve an average recall of 96% across all datasets. One case of false negatives is that columns containing only non-textual values make it difficult to distinguish the header row from the data row, e.g., numeric headers, such as year, date, etc. Thus, Suragh fails to classify the header row as ill-formed, resulting in a false negative.

Another false negative case is when ill-formed record value patterns simply occur frequently – the file contains many problematic rows. This is often the case for rows with many missing values. Note that we only consider empty/null values records as inconsistent if the file contains non-uniform empty/null values, e.g., a column that contains empty/null values at arbitrary locations. Therefore, the global threshold setting considers these patterns as well-formed, which prevents Suragh from classifying these records as correctly.

5 RELATED WORK

Detecting ill-formed records in CSV files is a novel research problem. However, to improve the user experience, some related work has developed pattern-based approaches to implement automated data pre-processing pipelines.

The first step in retrieving data from CSV files is to determine the dialect of a file, particularly the delimiter and quote characters. Van den Burg et al. detect the dialect of CSV files by leveraging row and type patterns [30]. The authors use pattern scores to favor frequent and long row patterns to find the most suitable dialect. In our approach, we used the more readily available and flexible univocity dialect detection system.

Discovering the structure of CSV files is a challenging task that requires distinguishing amongst rows of different types, e.g., data, metadata, header, group header, aggregation, and footnote rows. To address the row classification problem, Jiang et al. propose the Strudel approach [12], which classifies rows based on three types of features: content, context, and computational features. We compared Suragh with this approach in Section 4.3.

Qahtan et al. suggest a syntactic pattern approach to detect disguised missing values as an outlier [23], which we compared with our proposed technique in Section 4.3. The authors also introduced so-called pattern functional dependencies, a new type of integrity constraint for data cleaning by making use of syntactic patterns [22].

Jin et al. introduced the transformation-by-pattern (TBP) paradigm for data transformation using a pattern-based approach that uses a source pattern, a target pattern, and a transformation program [13]. Their way of representing the data patterns is similar to our abstraction representation, so our automatically detected patterns might serve as input to this approach in discovering TBP programs.

6 CONCLUSION AND FUTURE WORK

Raw data contain ill-formed records, which have inconsistencies at both column- and row-levels, preventing data from being (correctly) loaded into downstream applications. Identifying such records in advance reduces errors and user effort in the event of a data loading problem.

We introduced Suragh– a method to automatically detect ill-formed records and helps expert users by generating a pattern schema for a file so that they can pre-determine how messy the data is and how much data preparation is needed. Suragh takes an input file from a user and then generates, collects, and constructs syntactic patterns at several levels that help classify ill-formed records in a file. In addition, we used pruning techniques and selected highly specific and high coverage syntactic patterns by introducing weighing criteria and coverage thresholds to optimize our solution.

To evaluate our approach, we collected real-world datasets that exhibit data errors found in practice at the syntactic level. To annotate our data, we performed an extensive empirical study by loading each file of our datasets into three tools: 1) an RDBMS, 2) a data wrangling tool and 3) a business intelligence tool, to investigate whether and where the load-process is interrupted. We annotated 210 550 records in files that had at least one loading problem. Finally, we conducted an extensive set of experiments on datasets: DataGov, Mendeley, GitHub, and UKGov and achieved an average precision of 77% and an average recall of 97% in identifying ill-formed records. We also tested the generalizability of our approach on an unseen dataset NYCData and achieved 70% precision and 95% recall.

We plan to extend our system with an interface that allows users to interact with it and choose between the sets of dominant syntactic row patterns and even increase or decrease their size by changing the threshold settings. Such an interactive tool would help users keep or remove ill-formed records, depending on their preferences.

After detecting ill-formed records in a file, the next possible direction would be to transform these records into a uniform format and help users prepare data. Our system shall transform these records by looking at the frequent patterns in a file using the dominant syntactic row patterns.

ACKNOWLEDGMENTS

This research was funded by the HPI research school on Data Science and Engineering.

REFERENCES

[1] Paul D Allison. 2001. *Missing data.* Sage publications, Thousand Oaks, CA.

[2] Sara Bonfitto, Luca Cappelletti, Fabrizio Trovato, Giorgio Valentini, and Marco Mesiti. 2021. Semi-automatic column type inference for CSV table understanding. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 535–549.

[3] Christina Christodoulakis, Eric B Munson, Moshe Gabel, Angela Demke Brown, and Renée J Miller. 2020. Pytheas: pattern-based table discovery in CSV files. *PVLDB* 13, 11 (2020), 2075–2089.

[4] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1247–1261.

[5] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 541–552.

[6] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-hypothesis CSV parsing. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 1–12.

[7] EBNF 1996. ISO/IEC 14977:1996(E), Extended BNF. https://www.iso.org/standard/26153.html. Accessed: 2021-05-25.

[8] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 883–899.

[9] Mazhar Hameed and Felix Naumann. 2020. Data Preparation: A Survey of Commercial Tools. *SIGMOD Record* 49, 3 (2020), 18–29.

[10] Joseph M Hellerstein, Jeffrey Heer, and Sean Kandel. 2018. Self-Service Data Preparation: Research to Practice. *IEEE Data Engineering Bulletin* 41, 2 (2018), 23–34.

[11] IETF 2005. RFC 4180. https://tools.ietf.org/html/rfc4180. Accessed: 2021-03-12.

[12] Lan Jiang, Gerardo Vitagliano, and Felix Naumann. 2021. Structure Detection in Verbose CSV Files. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 193–204.

[13] Zhongjun Jin, Yeye He, and Surajit Chauduri. 2020. Auto-transform: learning-to-transform by patterns. *PVLDB* 13, 11 (2020), 2368–2381.

[14] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive data cleaning for statistical modeling. *PVLDB* 9, 12 (2016), 948–959.

[15] Roderick JA Little and Donald B Rubin. 2019. *Statistical analysis with missing data*. Vol. 793. John Wiley & Sons.

[16] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective error correction via a unified context representation and transfer learning. *PVLDB* 13, 11 (2020), 1948–1961.

[17] 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 International Conference on Management of Data (SIGMOD)*. 865–882.

[18] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. 2016. Characteristics of open data CSV files. In *Proceedings of the International Conference on Open and Big Data (OBD)*. IEEE, 72–79.

[19] Sebastian Neumaier, Axel Polleres, Simon Steyskal, and Jürgen Umbrich. 2017. Data integration for open data on the web. In *Reasoning Web International Summer School*. Springer, 1–28.

[20] Dan Olteanu. 2020. The Relational Data Borg is Learning. *PVLDB* 13, 12 (2020), 3502–3515.

[21] Gil Press. 2016. Cleaning Data: Most Time-Consuming, Least Enjoyable Data Science Task. *Forbes* (March 2016).

[22] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern functional dependencies for data cleaning. *PVLDB* 13, 5 (2020), 684–697.

[23] Abdulhakim A Qahtan, Ahmed Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. 2018. FAHES: A robust disguised missing values detector. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*. 2100–2109.

[24] Abdulhakim Ali Qahtan, Ahmed K Elmagarmid, Mourad Ouzzani, and Nan Tang. 2018. FAHES: Detecting Disguised Missing Values. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1609–1612.

[25] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.

[26] Yoones A Sekhavat, Francesco Di Paolo, Denilson Barbosa, and Paolo Merialdo. 2014. Knowledge base augmentation using tabular data. In *Proceedings of the Workshop on Linked Data on the Web (LDOW)*.

[27] Vraj Shah and Arun Kumar. 2019. The ML data prep zoo: Towards semi-automatic data preparation for ML. In *Proceedings of the International Workshop on Data Management for End-to-End Machine Learning*. 1–4.

[28] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *PVLDB* 13, 5 (2020), 616–628.

[29] Ignacio G Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[30] Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.

[31] Hadley Wickham. 2014. Tidy data. *Journal of statistical software* 59, 10 (2014), 1–23.