Pollock: A Data Loading Benchmark

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

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

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

Lucas Reisener
Hasso Plattner Institute
University of Potsdam, Germany
lucas.reisener@student.hpi.de

Eugene Wu
Columbia University
New York, NY
ewu@cs.columbia.edu

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

ABSTRACT

Any system at play in a data-driven project has a fundamental requirement: the ability to load data. The de-facto standard format to distribute and consume raw data is csv. Yet, the plain text and flexible nature of this format make such files often difficult to parse and correctly load their content, requiring cumbersome data preparation steps. We propose a benchmark to assess the robustness of systems in loading data from non-standard csv formats and with structural inconsistencies. First, we formalize a model to describe the issues that affect real-world files and use it to derive a systematic "pollution" process to generate dialects for any given grammar. Our benchmark leverages the pollution framework for the csv format. To guide pollution, we have surveyed thousands of real-world, publicly available csv files, recording the problems we encountered. We demonstrate the applicability of our benchmark by testing and scoring 16 different systems: popular csv parsing frameworks, relational database tools, spreadsheet systems, and a data visualization tool.

PVLDB Reference Format:
Gerardo Vitagliano, Mazhar Hameed, Lan Jiang, Lucas Reisener, Eugene Wu, and Felix Naumann. Pollock: A Data Loading Benchmark. PVLDB, 16(8): 1870 - 1882, 2023. doi:10.14778/3594512.3594518

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HPI-Information-Systems/Pollock.

1 INGESTING RAW DATA

The ability to load the content of files is paramount for the success of any data-driven project. Typically, data files come in a variety of formats. To gain insights into the most common file formats used to share data, we analyzed 17 repositories of governmental data portals across six continents. The results of our survey are summarized in Table 1, reporting on the format of files contained in a total of 784 062 available datasets[1].

Table 1: Number of datasets by format in 17 governmental portals. One dataset can contain files with multiple formats.

Format	# datasets	% of total	Format	# datasets	% of total
HTML	326 446	41.63%	XML	128 452	16.38%
CSV	**245 594**	**31.32%**	ZIP	67 024	8.54%
PDF	151 053	19.26%	JSON	65 008	8.29%

Excluding formatting-oriented documents such as HTML and PDF, a vast majority of structured data is found in csv files. This format is a common choice due to the relational nature of most data, the ease of reading/writing plain-text files, and its support by many systems. Such a wide spread is also a curse, as the flexibility of the csv format is often coupled with non-standard files and a lack of appropriate metadata – a common scenario in real-world datasets. These issues are not foreign to academia: a survey by Mitlöhner et al. on 200k different open data csv files identified a wide range of dialects, files with multiple tables, the presence of comment lines, and multi-row headers [24]. Van den Burg et al. [35], in proposing a csv wrangler, observed that the Python csv parser failed to detect the correct dialect in 36% of non-standard csv from 8k files. Christodoulakis et al. [6], in applying their csv table detection algorithm on a set of more than 23k files collected from the Canada open data portal, identified 14 different encodings, five different delimiters, and up to 226 tables in a single file. Although research has addressed the tasks of CSV dialect parsing [9, 13, 35], csv line and cell classification [14, 20], or table extraction [6], it is unclear to which degree these results have been transferred into real-world systems, and how much of the data preparation burden is left to end-users. Our goal is therefore to benchmark the robustness of data loading in the presence of real-world errors and non-standard files. One of the fundamental obstacles to automated testing and benchmarking is the lack of a formal model to describe the structural issues of csv files and decouple them from semantic errors. A further challenge is the lack of annotated datasets of unprepared files with a clean ground truth. Creating such a dataset is an expensive task that requires a large amount of data, time, and domain expertise. To address these challenges, we contribute Pollock, a formal framework to classify file issues with respect to serialization grammars, and its application to generate systematically large-scale datasets of unprepared files with the corresponding clean ground truth.

Our framework formally defines the concepts of *content*, *structure*, and *format* of file grammars, and their *dialects*. We introduce

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594518

[1] The full results and the code to reproduce our analysis can be found on the project repository at https://github.com/HPI-Information-Systems/Pollock

the concept of "file pollution", a systematic transformation of a file that modifies its parsing grammar into a *structurally different* version of it. Based on this framework, we manually analyzed 3 712 real-world CSV files and identified and classified a broad set of non-standard features. We use our framework along with the survey results to systematically create polluted versions of an otherwise standard file. We then implemented the Pollock benchmark for CSV data loading, to assess the maturity of a system in handling polluted CSV data files[2]. This benchmark is designed for systems that operate at all stages of a data preparation pipeline, e.g., early parsing, data wrangling/cleaning, data visualization, or data storage. The specific contributions of our paper are:

(1) A general framework that identifies format, content, and structure, which is applicable to all data file grammars and can be used to systematically pollute them.
(2) A sample of 3 712 real-world CSV files annotated with respect to their pollutions, analyzed in a survey.
(3) A benchmark composed of an input standard CSV file and a set of 2 290 pollutions, based on the results of the survey.
(4) A set of weights based on the survey results to aggregate several metrics into a benchmark score.
(5) Experimental measurements to assess the loading capability of 16 different systems using our benchmark.

In Section 2, we describe our framework to characterize data parsing grammars and introduce the concepts of pollution and structural difference of grammars. Section 3 describes our data loading benchmark, derived from our survey results. Section 4 reports the results obtained by 16 different systems on our benchmark and discusses their shortcomings. In Section 5, we discuss the challenges of CSV files and related research efforts to address those issues. Section 6 concludes with a summary and directions for future work.

2 THE POLLUTION FRAMEWORK

In the scope of our work, data files are actual textual files, collections of files, or strings in memory encoded with given grammars. Regardless of their differences, all grammars used for file serialization have similar characteristics: they specify what content is allowed, what rules are used to parse content from the file, and how to format the content in a given representation.

We introduce a formal framework based on context-free grammars, which we use to serialize content into files and parse content out of files. We note that our framework is not only applicable to all possible dialects of the CSV grammar, but also to any other data serialization grammars that are context-free, e.g., JSON or XML.

2.1 Content, Structure, Format

A "data file" is a sequence of characters that expresses content in a given context-free grammar [4].

Definition 1 (Context-Free Grammar). A context-free grammar G is a set of terminal symbols $\mathcal{T}$, a set of non-terminal symbols $\mathcal{V}$, a start symbol $V_s \in \mathcal{V}$, and a set of rules $\mathcal{R} : \mathcal{V} \times (\mathcal{V} \cup \mathcal{T})$.

Since data files may also contain metadata, depending on the application, we use the more general term *file* and refer to the payload of a file as *content*. We refer to *serialization* as the act of producing a file that encodes a content C using the rules of a specific grammar G: $f = G(C)$, and *parsing* as the act of extracting content from a file: $C = G^{-1}(f)$.

Consider the sample CSV file f_0 of Figure 1 and its grammar G_0 to describe it. G_0 is a simplified version of the standard one defined in the RFC4180 document [29]. The content of the file is a set of records containing the values ("1,2,3", "4,5,6", ...) for the attributes with the headers "A,B,C". Other characters found in the file, e.g., commas and newlines, constitute the structure of the file: they serve parsing but do not belong to file content. Based on this intuition, we classify three types of symbols, and their corresponding rules: *content*, *structural*, and *format*.

Definition 2 (Content). In a grammar G, given a rule $R \in \mathcal{R}$ and terminal symbols $T_i, T_j \in \mathcal{T}$, the set of content symbols is $\mathbb{C} = \{C \in \mathcal{V} \mid \exists\, R : C \rightarrow T_i \mid T_j,\ T_i \neq T_j\}$. We call R a content rule.

Content rules are rules that may resolve to multiple terminal symbols[3]. Because of this, they describe the objects of serialization, or "what" is allowed in a given file. In the example of Figure 1, $R4$ and $R5$ are content rules.

Definition 3 (Structure). In a grammar G, given a rule $R \in \mathcal{R}$ and a terminal symbol $T \in \mathcal{T}$, the set of structural symbols is $\mathbb{S} = \{S \in \mathcal{V} \mid \exists!\, R : S \rightarrow T\}$. We call R a structural rule.

Structural rules are rules that resolve to a unique terminal symbol (or sequence). In simple words, they pose as markers to identify "where" to find content in a given file. In the example of Figure 1, $R6$ and $R7$ are structural rules.

Definition 4 (Format). In a grammar G, given a rule $R \in \mathcal{R}$ and non-terminal symbols $V_0, \ldots, V_n \in \mathcal{V}$, the set of format symbols is $\mathbb{F} = \{F \in \mathcal{V} \mid \exists R :\rightarrow V_0 \ldots V_n\}$. We call R a format rule.

All rules that do not directly resolve in a terminal symbol (or sequence) are *format* rules: they express "how" to combine content with structure in a given format. In the example of Figure 1, $R1$, $R2$, and $R3$ are format rules. We annotate format rules with grouping information. During parsing, this information is used to build the parse tree, specifying for every rule whether its right-hand side symbols are to be considered as an ordered list or as an unordered set. To express format rules with conciseness, we also introduce "symbol cardinality", a notation to specify the repetition of symbols.

Definition 5 (Symbol cardinality). In a grammar G, given a rule $R \in \mathcal{R}$ containing a symbol $V \in \mathcal{V}$, symbol cardinality is the number of times V has to be repeated when applying rule R. Symbol cardinality is expressed by postfixing V with $\{m, n\}$, where $m, n \in \mathbb{N} \cup \{\infty\}$, signifying a repetition of a minimum of m to a maximum of n times. Brackets with a single number define a required cardinality of $m = n$. Lack of notation implies a cardinality of $m = n = 1$.

This notation can be used to express any grammar in Chomsky Normal Form (CNF) [5] (and therefore any CFG grammar) with more conciseness. As proof, suppose a format rule expressed as $R : F \rightarrow V_0 V_1\{1, m\} V_2$ with a given maximum cardinality $m \neq \infty$. In normal form, non-terminal rules need to be in the form $A \rightarrow BC$: the rule R has to be expanded with $m + 1$ additional

[2] Pollock, inspired by the abstract painter, stands for Polluted CSV benchmarK.

[3] For notational simplicity, we excluded sequences of symbols. Conceptually, sequences of terminal symbols are equivalent to individual terminal symbols.

rules: $R_0 : F \rightarrow F_m V_2$, $R_1 : F_m \rightarrow F_{m-1} V_1$,..., $R_m : F_1 \rightarrow F_0 V_1$, $R_{m+1} : F_0 \rightarrow V_0 V_1$. Formulating rules with symbols having an infinite cardinality in CNF is possible with the addition of an extra rule: for example, $R : F \rightarrow R_0 R_1\{0, \infty\}$ has to be expressed with the two rules $R_0 : F \rightarrow R_0 F_1$, $R_1 : F_1 \rightarrow R_1 | F_1 R_1$.

Being equivalent to grammars in CNF format, our framework can be applied to any grammar used for serialization and parsing data from files.

2.2 Grammar dialects

Data serialization grammars are often regulated by standard specifications [3, 29]. However, real-world files often do not comply with standards. For example, one of four csv files out of the 3 712 files we sampled in a real-world survey (Cf. Section 3.3) uses a field delimiter different from comma (as prescribed by the RFC4180 standard). Consider the four exemplary files shown in Figure 2: they are all obtained by serializing the same content C, a header row followed by two data rows. The content of the first file can be parsed with the RFC4180-compliant grammar G_0 of Figure 1. All other files require slightly different grammars G_1, G_2, G_3 to be correctly parsed. Referring to the framework introduced in Section 2.1, all three grammars have the same content and format of G_0, but:

(1) G_1 uses a different separator rule: $\widetilde{R}_6 : DEL \rightarrow$“;”
(2) G_2 allows rows with an extra delimiter: $\widetilde{R}_3 : \widetilde{row} \rightarrow row\, DEL$
(3) G_3 allows rows with different separators: $\widetilde{R}_7 : \widetilde{DEL} \rightarrow$ ’;’

Two context-free grammars are equivalent if they can serialize or parse the same sequences of tokens [25]. The grammars to parse the different files in Figure 2 are not strictly equivalent, because they differ in structural tokens or cardinalities. Still, they parse the same content from the four files. Regardless of their grammars, we define two files $f, \widetilde{f}$ as *content equivalent* if the content obtained parsing them with their respective grammars $G, \widetilde{G}$ is the same. Formally:

Definition 6 (Content equivalence). Two files f and $\widetilde{f}$ parsed with the grammars G and $\widetilde{G}$, respectively, are content equivalent if $C = G^{-1}(f) = \widetilde{G}^{-1}(\widetilde{f}) = \widetilde{C}$.

In other words, given the parse trees $C = G^{-1}(f)$ and $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$, f and $\widetilde{f}$ are content equivalent if there exists a homomorphism between format and content symbols, and for ordered format rules, all right hand side symbols are found in the same order.

Definition 7 (Structurally different grammars). A grammar G is structurally different from a grammar $\widetilde{G}$ if, given two content-equivalent files $f, \widetilde{f}$ with content C, the following hold: (1) $G \neq \widetilde{G}$, (2) $f = G(C) = G(\widetilde{G}^{-1}\widetilde{G}(C))$, and (3) $\widetilde{f} = \widetilde{G}(C) = \widetilde{G}(G^{-1}G(C))$.

Two grammars are (only) *structurally different* if they parse the same content from two different, yet content equivalent files.

Definition 8 (Grammar dialects). Given a grammar G, its *dialects* are all grammars $\widetilde{G}$ structurally different from G.

In the example of Figure 2, grammars G_1, G_2, and G_3 are all dialects of G_0 (which is in turn a dialect of the RFC4180 csv grammar).

2.3 File pollution

For a given standard grammar G, our goal is to benchmark how real-world systems load files serialized with different dialects of G. However, the set of dialects of a grammar G is infinite; and even for a single grammar G there are infinite possible files with different contents. Consider the three reasons why a file $\widetilde{f}$ can differ from f: (1) $\widetilde{f}$ is expressed in the same grammar as f, but serializes a different content, i.e., $\widetilde{f} = G(\widetilde{C})$; (2) $\widetilde{f}$ serializes the same content as f. but with a different grammar, i.e., $\widetilde{f} = \widetilde{G}(C)$; (3) $\widetilde{f}$ serializes a different content with a different grammar, i.e., $\widetilde{f} = \widetilde{G}(\widetilde{C})$.

To design a data loading benchmark, we are primarily interested in files that belong to (2), i.e., different files serializing the same content with different dialects of a grammar. However, to account for common issues in the wild (Cf. Section 3), we also consider restricted cases of (3), where the content C of a file f is a strict subset of the content $\widetilde{C}$ of a file $\widetilde{f}$ (whose grammar may also be structurally different). We call *file pollution* the transformation of a file $f = G(C)$ into a content equivalent file $\widetilde{f} = \widetilde{G}(C)$, where $\widetilde{G}$ is a dialect of G. We describe a simple procedure to construct file pollutions. Given a file f, rather than modifying its grammar G and then serializing a new file $\widetilde{f} = \widetilde{G}(C)$, we directly modify the parse tree $G^{-}1(f)$ in two ways: by changing structural symbols and the cardinalities of symbols in format rules. These changes guarantee that the content of the file $\widetilde{f}$ has been serialized with a structurally different grammar $\widetilde{G}$, without the need to construct $\widetilde{G}$ explicitly. Given a file f and its parse tree $C = G^{-1}(f)$, we can systematically enumerate all the possible file pollutions. A pollution can: (1) change any of the structural symbols S with a different symbol $\widetilde{S}$, or (2) increase or decrease the cardinality of a symbol V in a format rule.

Our formalization of pollution offers several advantages. First, structural differences that characterize a resulting dialect are well-defined; second, they can be chosen at design time as a parameter; third, as the pollution is a controlled transformation, a ground truth content is available to evaluate the results of loading. Of course, the space of pollutions is still large, so it is unclear how to sample relevant pollutions to concretely instantiate in a data loading benchmark. Given our framework, the problem of designing a relevant data loading benchmark can be formalized with the following problem statement:

Given a source file f that serializes content C with a grammar G, find pollutions that generate a set of files $\widetilde{f_0}, ..., \widetilde{f_k}$, such that the content C_i of every file f_i is equivalent to C (or a strict superset) and is serialized with a grammar $\widetilde{G_i}$, dialect of the original grammar G. To address this problem, the next section describes our survey of 3 712 real-world csv files and their pollutions.

3 THE POLLOCK BENCHMARK

In this section, we describe Pollock, a benchmark for csv data loading that results from the application of grammar-based pollution as introduced in Section 2. To formally define a benchmark using the pollution framework, we need to specify:

(1) A reference grammar G.
(2) A source file f that serializes a content C using the grammar G, serving as the basis for the pollution operations.
(3) For each of the format rules and structural rules in G, a set of pollutions to obtain different dialects $\widetilde{G}$.
(4) Metrics to measure how well a system loads polluted files.

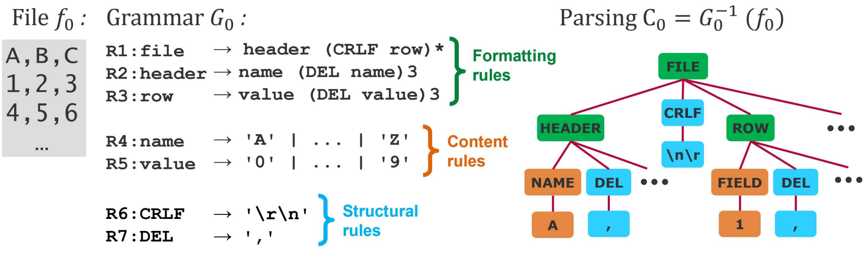
File f_0 :
A,B,C
1,2,3
4,5,6
...
Grammar G_0 :
R1:file → header (CRLF row)*
R2:header → name (DEL name)3
R3:row → value (DEL value)3
Formatting rules
R4:name → 'A' | ... | 'Z'
R5:value → '0' | ... | '9'
Content rules
R6:CRLF → '\r\n'
R7:DEL → ','
Structural rules
Parsing $C_0 = G_0^{-1}(f_0)$
FILE
HEADER
CRLF
ROW
\n\r
NAME
DEL
FIELD
DEL
A
,
1
,

Figure 1: A sample csv file f_0 and a grammar G_0 to parse it. Each node in the parse tree corresponds to a grammar rule.

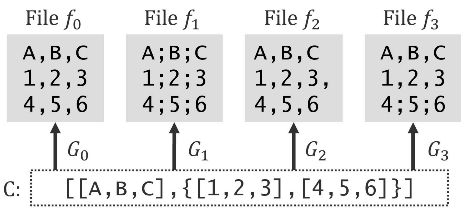

Figure 2: Four different files with equivalent content.

The core idea of Pollock is to systematically replicate real-world dialects, isolating one pollution at a time. To do so, we synthetically generate different polluted versions of a single input file f, denoted $\widetilde{f_1}, \widetilde{f_2}, ...\widetilde{f_n}$, each serialized with a different dialect of the standard grammar G. We ground the design of pollutions on a survey of 3 712 publicly available real-world files. With the results of the survey, we sample the space of possible pollutions and design a representative input file f to be polluted. We deliberately isolate pollutions to precisely benchmark their effect on data loading. We acknowledge that in practice, real-world files may deviate from the standard csv grammar with several pollutions interacting at once. Moreover, we also observed loading issues in files serialized with the standard RFC4180 grammar due to system-specific assumptions (e.g., a maximum length for cell values). To gain insights on loading real-world files, in Section 4 we also benchmark different systems with a random sample of the survey files, guaranteed to contain all pollutions at least once.

3.1 Survey setup

We aim to benchmark csv data loading, therefore the Pollock reference grammar G is the standard RFC4180 grammar for csv files [29]. Figure 3 presents a formulation of this grammar according to our framework. Cardinalities should be treated as constants for a given file: for example, rules F3 and F4 specify that the header and record rows all have the same number N of cells.

We surveyed a sample of 3 712 real-world files marked as csv: 2 274 csv files randomly sampled from the Mendeley Data portal [22] and 1 438 randomly sampled files from the open data portal of the United Kingdom government [8]. The first is a public repository of scientific projects, where researchers can share research artifacts, such as code, data, and experimental results. We crawled all 2 214 projects that, at the time of our survey, contained at least one file whose MIME type was “text/*”. Out of more than 34 000 files contained in these projects, we retained all 2 274 files with a “.csv” extension. The files selected from the UK government open data have been crawled from all datasets stored in the portal at the time of the experiments, retrieving a total of 17 851 files marked with the “text/csv” MIME type, out of which we randomly sampled 1 438 files. For all 3 712 survey files, we manually annotated whether their grammars follow the RFC standard and, if not, which rules differ in their dialect. The collected files with their annotations can be found online, together with all other benchmark artifacts[4].

```
Format rules:      F0: file = table CRLF{0,1}
                   F1: table = (header CRLF) {0,1} data
                   F2: data = record (CRLF record){0,∞}
                   F3: header = cell (COMMA cell){N,N}
                   F4: record = cell (COMMA cell){N,N}
                   F5: cell = QUOTE (escaped){0,∞} QUOTE
                   F6: cell = text text{0,∞}
                   F7: escaped = COMMA|ESCAPE QUOTE|CRLF|text
Content rules:     C0: text = 0x20-21|0x23-2B|0x2D-7E| ϵ
Structural rules:  S0: CRLF   = 0x0D 0x0A
                   S1: COMMA  = 0x2C
                   S2: QUOTE = 0x22
                   S3: ESCAPE = 0x22
```

Figure 3: RFC4180 standard grammar for csv files.

3.2 Input file design

To design the input file for our pollutions, we analyzed the 3 712 survey files regarding their general characteristics. Out of all files, 15 are empty, i.e., they have no content and a dimension of 0 bytes: in the following analysis, we exclude these files. The remaining files contain a total of 46 474 823 rows and 296 602 columns. The minimum number of rows per file is 1, with the maximum being 9 505 531 rows. The distribution of rows per file is highly skewed, with an average of 11 981.14 rows per file but a mode of 2 and a median of 84. Regarding columns, the minimum number of columns is also 1, with the maximum being 34 804. As for columns, the average is 76.46 columns per file, but the mode and median are both at 9 columns per file. To gain further insights into the data types of the columns, we automatically detected a data type for each one. We use the regular expression-based type detection proposed in the CleverCSV project [35], which classifies cells into one of twelve data types. To classify columns, we detect the type of each of the column cells and record the most frequently occurring type for each column. We further divide the string column type into three types: “short string” if all values in a column are under 100 characters, “long string” if any of the cells is longer than 100 characters, and “fixed length” if all values in a column have the same number of characters, e.g., code identifiers. Table 2 reports the statistics for each of the column data types. The table reports the number of columns for which CleverCSV was unable to detect a data type, roughly 2% of the total. We note the high number of empty columns in the surveyed files: overall, a total of 1 244 files contain at least one empty column. However, the high number of empty columns is caused by a tiny fraction of files that have an unusually large amount of trailing empty columns. For example, one of the files contains 19 non-empty columns, and 16 383 empty columns after the last non-empty one. We note that 119 044 columns, 97.58% of all

[4] https://github.com/HPI-Information-Systems/Pollock

Table 2: Column data types in survey files.

Data type	# col.	% total	Data type	# col.	% total
Number (digits)	129 531	43.672%	Datetime	165	0.056%
Empty	121 992	41.130%	Percentage	141	0.048%
String (long)	34 285	11.559%	Number (float)	130	0.044%
String (fixed)	1 466	0.494%	Email	103	0.035%
Date	730	0.246%	Time	94	0.032%
String (short)	694	0.234%	Unix path	4	0.001%
URL	261	0.088%	Undetected	6 706	2.261%

Table 3: Overview of Pollock pollutions with respect to the RFC4180 standard grammar.

Grammar rule	# Generated polluted files
F0: file= payload CRLF {0,1}	3
F1: table = header{0,1} data	7
F2: data = record (CRLF record){0,∞}	2
F3: header = cell (COMMA cell){N,N} CRLF	17
F4: record = cell (COMMA cell){N,N}	1 411
F5: cell = DQUOTE (quoted){0,∞} DQUOTE	756
S0: CRLF = 0x2C 0x0A	2
S1: COMMA = 0x2C	88
S2: DQUOTE = 0x22	1
S3: ESCAPE = 0x22	2

empty columns, are trailing empty columns in a file. These trailing empty columns affect 954 files (25.54% of the total files).

The survey files contain 111 340 columns (38.40% of the total) with at least one quoted cell. We analyzed the distribution of quoted cells inside these columns: in 37 833 columns, only less than 10% of their cells are quoted, and 66 275 columns have more than 90% of their cells quoted. This distribution is highly bimodal as the combined two cases cover 93.50% of the total quoted columns, and it reflects two different styles of handling quotation: in the former, only cells that require quotation in a column are quoted ("minimal" style); in the latter, all cells of a column are quoted regardless of need ("holistic" style). Considering the results of our survey, we design the source file as a csv file named "source.csv", with 9 columns and 84 lines – one header row and 83 data rows, for a total of 756 file cells. The file is available in the project repository. The number of rows and columns is chosen as the median of the survey files. The rows in the file represent products sold from an online shop at a given time. Overall, the nine columns represent the most frequent data types we encountered in our survey:

- **DATE** expressed as DD/MM/YYYY, with the column containing unambiguous values wrt. day and month (e.g., 28/01/2018).
- **TIME** represents a time of the day. The format used is HH:MM and the values increase the time from 00:00 in steps of 15 minutes.
- **PRODUCTID** contains a fixed-length alphanumeric code.
- **Qty** is a non-negative integer number.
- **Price** contains a currency value, expressed with the US dollar sign and a positive floating-point number with a full stop as a decimal delimiter and two significant digits.
- **ProductType** contains a short string (under 100 characters) in natural language. This column contains quoted cells and escaped characters and is quoted "minimal" style.
- **ProductDescription** contains a long string (above 100 characters) with a natural language description of the products. This column also contains quoted cells and escaped characters and is quoted "holistic" style.
- **URL** contains a sample URL and is quoted "holistic" style.
- **Comments** is a trailing empty column, simulating optional information regarding a given product.

Although the results of Table 2 show that numeric columns in the form of digits are more frequent than other data types and that many files contain numerous trailing empty columns, we design our file to contain one numeric column and one trailing empty column - in an effort of sampling a broader spectrum of data types. We also note that, while we run our experiments of Section 4 with this input file, the Pollock pollutions can be applied to any input file that follows the standard csv format.

3.3 Pollution design

Not all files of our survey can be parsed correctly using the standard csv grammar. Here we report, for each format and structural rule of the RFC grammar (Cf. Figure 1), all different variations of the rules required to parse the real-world files of the survey. For the scope of our benchmark, we include a single pollution type to cover each of these variations individually, even if the dialect of a single survey file might have several. A single pollution may have different possible parameters wrt. a file, e.g., a single-row pollution may apply to any row of a file. To generate polluted files, we first identify and isolate all pollution types affecting real-world files, and then we generate a benchmark file for each possible parameter, e.g., each row, column, or cell. In sum, our benchmark includes 2 290 polluted files. For every pollution type found in our survey, we report how many real-world files were affected and how (many) benchmark files ("Pollock files") represent this pollution. The list of pollution types and the number of benchmark files are summarized in Table 3. While theoretically possible, we acknowledge that our framework does not attempt to generate files with multiple pollutions. First, applying several pollutions on the same file requires a notion of dependency to avoid interactions where different pollutions cancel (or alter) each other's effect. Additionally, from a practical perspective, it would be time-consuming to run the benchmark as the number of files would increase exponentially. Studying the extension of our framework to create more complex pollutions is an interesting research problem discussed in Section 6.

3.3.1 F0: file format. The rule *F*0 of the grammar specifies that a file is composed of a table with an optional newline sequence crlf. In our survey, we encountered:

- 15 empty files, with no table
- 184 files with no trailing newline
- 3 508 files with one trailing newline
- 5 files with more than one trailing newline. All these files end with two newlines.

In the following analysis, we exclude the 15 empty files but retain one empty file among the benchmark files.

Pollock files: 1 empty file; 1 file without a trailing CRLF; 1 file with two trailing CRLF.

3.3.2 *F1: table format.*

A table of a standard CSV file is composed of a single optional header line and data. Our survey found:

- 2 751 files with one header line
- 470 files with no header
- 476 files with multiple header lines

Of the files with multiple header lines, 94 contain multirow table headers spanning two or three lines. The other 282 files contain multiple "preamble" lines: rows with comments or metadata separated from the true table header with at least one empty line, i.e., a line with only separators and no content. Finally, 188 files contain multiple tables. In these files, there are two (or more) sections with header and data that mark different sections of the file content, at times with preamble lines or multiple header lines.

Pollock files: 1 file without a header; 2 files with multiple header lines (2 and 3 lines); 1 file with a preamble line; 3 files with two tables: one where both have the same number of columns, one where the second has more columns than the first, and one where the second has fewer columns than the first.

3.3.3 *F2: data format.*

According to rule *F*2, CSV files contain data arranged in rows, each row containing a record. In our survey, we encountered:

- 3 files with no records but only a header row
- 4 files with only a single record
- 3 690 files with multiple records

Pollock files: 1 file with only the header row; 1 file with a header and a single data row.

3.3.4 *F3, F4: header and record format.*

The RFC4180 grammar requires that header and record rows have the same number of cells (cf. rules *F*3 and *F*4 in Figure 1). We did not encounter files where the header does not terminate with a newline sequence. Regarding the number of cells, in our datasets we encountered

- 2 657 files with a consistent number of cells
- 1 040 files with an inconsistent number of cells

The number of cells can be inconsistent for different reasons: 221 files have preamble header lines with a different number of separators, some files have multiple tables in them (see above), with different column counts, others have data records with schema drift, where missing or extra cells are present in a subset of the records.

Pollock files: 17 files with an inconsistent header: one with a missing column separator for each of the 8 header separators, 9 with an extra separator before each column; 1 411 files with an inconsistent row: 664 with a missing column separator for each of the 8 column separators in the 83 data rows, 747 with an extra column separator before each of the 9 columns in all data rows.

3.3.5 *F5: Cell format.*

A cell inside a row can contain any sequence of characters: however, if this sequence contains any of the tokens of the rules $S0, S1, S2$ of Figure 3, the cell has to be enclosed in quotation characters (cf. Rule *F*6). The "reserved" tokens correspond to the structural characters required to separate rows (CRLF), delimit columns inside rows (COMMA), and the quotation character itself (QUOTE). The quotation character must be escaped with an extra quotation character to disambiguate it from the end of the quoted cell. We encountered seven files with an incorrectly quoted cell where a quotation mark was not escaped. We note that other pollutions related to quoting and escaping are harder to identify under this scope without explicit domain knowledge, but they are possible to identify with respect to other format rules. For example, a cell containing a newline or extra separator character without a quote would lead to a record having a different number of columns, a problem we identified in the previous analysis of files under rule *F*4.

Pollock files: 756 files with incorrectly quoted cells, adding one unescaped quotation mark in each of the file cells.

3.3.6 *S0: newline sequence.*

The RFC4180 defines the newline sequence to be the combination of the carriage return (CR) and line feed (LF) characters. In our survey, we encountered:

- 1 999 files with the sequence of both CR and LF
- 1 691 files with the only LF character
- 7 files with only the CR character

Pollock files: 2 files with non-standard newline sequences in every row, one using CR-only and one using LF-only.

3.3.7 *S1: cell delimiter.*

The standard character to delimit cells of a record is COMMA (cf. *S*1 in Figure 1). Nonetheless, it is common for CSV files to have a different delimiter, e.g., due to different locale specifications for floating-point numbers. In the survey dataset:

- 2 754 files use a comma delimiter.
- 834 files use a semicolon delimiter.
- 101 files use a comma plus whitespace or tab character as the delimiter.
- 8 files use a tab or a sequence of white spaces as delimiters.

Among the files using comma as a delimiter, 12 files have some of their rows delimited with sequences of white spaces. These inconsistent rows typically contain metadata regarding the table contained in the file, such as preamble or footnote lines.

Pollock files: 4 files with non-standard delimiters in every row, one using semicolon, one using tab, one using whitespace, one using comma+whitespace; 84 files with an inconsistent delimiter in a single row, one using whitespace for each file row.

3.3.8 *S2: quotation character.*

The second structural token specified by the RFC4180 grammar is the quotation character QUOTE, defined to be the double quotation character (Cf. *S*2 in Figure 1). In our survey, we identified only two different characters used for this marker:

- 1 596 files do not have any quoted cell.
- 2 090 files use the double quote character.
- 11 files use the apostrophe character.

We note that in the survey files using a different quotation character, no quote is found inside a cell and requires escaping. However, a different quotation character should also be accompanied by a different escaping sequence—following the RFC rules, a doubling of the quotation character.

Pollock files: 1 file with non-standard quotation character in every row (using apostrophe, also escaping any apostrophe in the cells with an extra apostrophe).

3.3.9 S3: escape character. The last structural token in the csv grammar is ESCAPE, the character used to escape a quotation character when contained in the value of a quoted cell. The RFC standard defines it to be the same as the QUOTE character (cf. *S*3 in Figure 1). This sequence occurs rarely and often leads to errors or inconsistencies if parsers or files do not adhere to the standard. In our experiments, only 2 systems out of 16 can correctly load the whole content of files with a polluted escape quote, with the others either dropping the content of the cells following the escape character or of the whole row altogether.

Of the 2101 files with quoted cells,

- 1 849 files do not contain cells with escaped values.
- 250 files contain cells with values escaped according to the RFC standard.
- 2 files contain cells without any escape sequence.

Even if not observed in our survey, we also note that a common non-standard escaping strategy is to preclude double quotes inside a cell with a backslash symbol (Ux5C).

Pollock files: 1 file with a non-standard escape character in every cell (the backslash symbol); 1 file where quotations are not escaped.

3.4 Metrics design

Formally, given an input file f that encodes a content C with a standard grammar G, a single pollution obtains a different file $\widetilde{f} = \widetilde{G}(C)$, encoding the same content with a formally equivalent grammar. Once we obtain a polluted file, we aim at benchmarking how a given system under test (SUT) parses and loads it in memory.

Any SUT parses the content of its input files with a given grammar G_{SUT} which is generally unknown to the end-users. Also, every system has different in-memory representations of a content $C_{SUT} = G^{-1}_{SUT}(f)$. However, all tested systems that can load csv input files are also capable of exporting content in an output file encoded with the standard RFC4180 grammar $f_o = G_{RFC}(G^{-1}_{SUT}(f_i))$.

Although pollutions are not allowed to change content production rules, we note that some format pollutions effectively create polluted versions of the source file with different content. This behavior is possible with pollutions that edit format rules by deleting content, e.g., to simulate a file with no header row; or by adding content in the input file, e.g., to simulate multiple tables in files. It would not be fair to expect a system to export content not present in the input file due to deletions from pollutions. Similarly, a correct loading should also include any extra input data not in the original source file but introduced by pollutions.

To address this, in measuring systems' performances we cannot use the content of the original source file. However, the polluted content $\widetilde{C}$ can be parsed from the polluted file $\widetilde{f}$ using the polluted grammar $\widetilde{G}$, which is known by design at benchmarking time. Therefore, we compare if the content parsed with the RFC grammar from the output file of a SUT, $C_o = G^{-}1_{RFC}(f_o)$, is equivalent to the polluted content parsed from the input polluted file $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$.

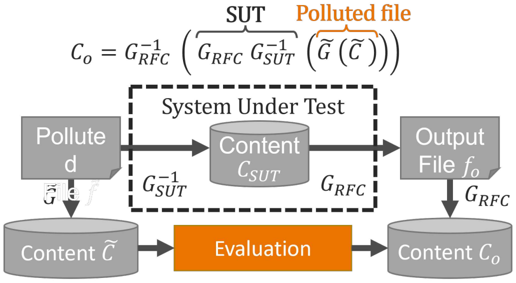
SUT
Polluted file
$C_o = G^{-1}_{RFC}\left(\overbrace{G_{RFC}\,G^{-1}_{SUT}}^{\text{SUT}}\left(\overbrace{\widetilde{G}\left(\widetilde{C}\right)}^{\text{Polluted file}}\right)\right)$
System Under Test
Polluted File $\widetilde{f}$
$\widetilde{G}$
Content C_{SUT}
G^{-1}_{SUT}
G_{RFC}
Output File f_o
G_{RFC}
Content $\widetilde{C}$
Evaluation
Content C_o

Figure 4: Summary of the benchmarking process.

Figure 4 summarizes our approach to benchmark a system's loading of a polluted file in a general, SUT-independent fashion. We load a polluted file $\widetilde{f}$ in a SUT, which parses it with an unknown grammar G_{SUT}. We then export it back using the standard RFC grammar and compare the contents parsed from the input files using the polluted grammar $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$, and the content parsed from the system output file $C_o = G^{-1}_{RFC}(G_{RFC}(G^{-1}_{SUT}(\widetilde{G}(C)))$. To compare contents independently of a specific internal system's representation, we normalize the output of individual cells to compare their values. The normalization parses dates and numbers and transforms all string characters in lowercase. For example, two cells containing the same date in two different formats are considered equivalent. To measure the equivalence of two parsed contents, we refer to the hierarchy induced by the format rules of the grammars (Cf. Section 2.3). Following the RFC standard grammar (Cf. Figure 3), we identify four content groups: a file is composed of (1) a table; a table is composed of (2) a header and (3) a set of records; records are composed of (4) cells. For the first level, we use a binary measure: *success (S).* If a file is loaded correctly without any application error, we assign a value of 1 to this score, otherwise a 0. However, even if a system successfully loads a polluted file, the resulting content may still differ from the expected content. A system may either miss some content while loading a file, e.g., excluding a polluted row, or include "spurious" content, e.g., by padding a row with unwanted cells. Therefore we also use precision to evaluate the loading "completeness", and recall to evaluate the loading "conciseness", combining them both into the F1 score. Given an input set of elements I and an output set of elements O, precision (P), recall (R), and F1 are defined as usual:

$$P = \frac{|(I \cap O)|}{|I|} \qquad R = \frac{|(I \cap O)|}{|O|} \qquad F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

To obtain a well-rounded score, we compute these metrics at the header, record, and cell level:

1. *Header precision (H_P), recall (H_R), F1 (H_{F_1})*: These metrics are computed on header cells, and are necessary because systems often have separate assumptions regarding header and data rows. They measure the effect of pollutions on file headers, e.g., if an extra header column is added because a single data row contains an extra cell.
2. *Record precision (R_P), recall (R_R), F1 (R_{F_1})*: These metrics are computed for each data record, defined as the string hash of its cell values. They capture whether individual records are loaded coherently, or their content is split, merged, or rearranged within different records, e.g. if due to a missing escape character two data rows get merged into one.
3. *Cell precision (C_P), recall (C_R), F1 (C_{F_1})*: These metrics are computed on individual data cells and are the most fine-grained. They identify data errors regardless of their position in the output file, e.g., if a value gets lost due to a missing delimiter.

The range of all scores is $[0, 1]$, with 1 representing a perfect data loading. In case of a data loading with a success of 0, meaning the system aborted loading due to some error, we assign a value of 0 to all remaining scores.

The nature of our benchmark is to isolate, whenever possible, different pollutions and test systems on loading files with each of them separately. As such, every system can be benchmarked with respect to a single pollution, and a single dimension. However, we also aim at providing a unified *Pollock score* for every SUT that measures its data loading performance across all different pollutions. To do so, we average all scores obtained across different polluted files, plus the scores obtained on the source file, and then sum them to obtain a single number per score. To provide an additional and more realistic score, we weigh the average by the occurrence of the pollution in the real world, as identified by the survey of Section 3.1. The weights are normalized, to sum up to 1. In the case of pollutions that replicate a single pollution systematically (e.g., for every row, cell or column), we scale the weights by the number of repetitions. For example, considering that in our survey 12 files had inconsistent row delimiters, and we repeat the pollution for each of the 84 rows of the source file, the metrics of each polluted file will weight 12/84 in the final average. Considering that every score has a range of $[0, 1]$ and that there are a total of 10 different scores for each polluted file, the maximum Pollock score obtainable by a system under test is 10.

4 BENCHMARKING RESULTS

To demonstrate the usage and usefulness of our benchmark, we experimented by applying it to a set of diverse real-world systems. By evaluating their data loading capabilities, we highlight their shortcomings and simultaneously assess the usefulness of Pollock. As listed in Table 4, we selected 16 systems of four tool-categories to highlight our benchmark's versatile nature and to analyze possible differences in the handling of file pollutions at different stages of a data preparation pipeline. We experimented with:

- Eight programming frameworks designed for csv parsing in programming languages
- Four relational database management systems
- Three systems designed for spreadsheet data analysis
- One business intelligence/data visualization tool

We chose programming frameworks for three popular programming languages: Python, R, and Java. For all Python modules, we used Python version 3.10.5. We benchmark the native "csv" module [36], referred to as PyCsv, which is used to read and write csv files; the Pandas module [28], designed for data analysis and manipulation; and CleverCSV [35], a module developed to specifically address loading "messy" csv files. For all R modules, we used R version 4.2.1. We benchmark the native "read csv" function [27], referred to as RCsv; and the specialized Hypoparsr algorithm [9], introduced in a research paper to perform "advanced" csv parsing. For Java modules, we used OpenJDK version 11. We benchmark the parsing libraries "Apache CSV Commons", referred to as CSVCommons [10], OpenCSV [32], and Univocity [2]. For those that allowed it, we resorted to automated parameter detection. In other cases, we manually specified suitable parsing parameters, if it was possible to do so. For database systems, we benchmarked four open-source RDBMS: MySQL [7], MariaDB [11], PostgreSQL [15], and SQLite [18]. Due to the nature of relational database systems, loading a file requires creating a table with the correct schema first. To do so, we specify all data types of such a table to be of TEXT or VARCHAR type, as our benchmark is concerned with file structure and not with semantic type detection of files. In Section 5, we note how benchmarking type detection relates to data loading. For spreadsheet systems, we benchmarked LibreOffice Calc [12], an open-source desktop system, referred to as Calc, a commercial desktop system referred to as SpreadDesktop[5], and an online tool referred to as SpreadWeb [5]. Lastly, we benchmarked a commercial data visualization tool, referred to as DataViz [5]. We note that we ran "best effort" experiments, using every applicable configuration option offered by each tool. Table 4 synthetically reports the loading configurations used for each tool. In the project repository[6], we share the results obtained by all systems and the scripts used to benchmark all non-commercial systems.

Table 4: Configurations of the benchmarked systems. "A" stands for automatic detection, "M" for manual specification, a missing entry marks the lack of a configurable option.

	Preamble lines	Multirow header	Missing header	Newline seq.	Delimiter	Quotation	Escape
CleverCSV 0.7.4 [35]				A	A	A	A
CSVCommons 1.9.0				M	M	M	M
Hypoparsr 0.1.0 [9]				A	A	A	A
OpenCSV 5.6	M				M	M	M
Pandas 1.4.3	M	A	A	M	A	M	M
PyCsv 3.10.5					A	A	A
RCsv 4.2.1	M		M	A	A	A	A
Univocity 2.9.1		A		A	A	A	A
MariaDB 10.9.3	M		M	M	M	M	M
MySQL 8.0.31	M		M	M	M	M	M
PostgreSQL 15.0			M		M	M	M
SQLite 3.39.0	M			M	M		
Calc 7.3.7					M	M	
SpreadDesktop	M		M		M	M	
SpreadWeb					M		
DataViz					M	M	

In the remainder of this section, we present an overview of the most interesting and surprising findings, based on the results of our benchmark, grouped by the rules of the grammar affected by the different pollutions we presented in Section 3. For space reasons, we report only F1 scores rather than precision and recall scores. For every subsection, we include takeaways for end-users, highlighting which problems require adequate preparation to correctly load the benchmark files, and for system developers, to identify opportunities for improvement.

[5] Anonymized due to licenses that forbid disclosing benchmarking results.
[6] https://github.com/HPI-Information-Systems/Pollock

Table 5: Systems with imperfect loading of the source file (RFC4180 compliant): success and F1-scores.

	S	H_{F1}	R_{F1}	C_{F1}	Loading time (ms)
Hypoparsr 0.1.0 [9]	1.00	0.00	0.11	0.63	3 277.11 ± 94.66
OpenCSV 5.6	1.00	1.00	0.98	0.99	12.72 ± 0.48
PyCsv 3.10.5	1.00	1.00	0.92	0.99	14.29 ± 3.08
DataViz	1.00	0.77	0.00	0.77	18 569.75 ± 592.11

4.1 Source file

Before analyzing the effect of pollutions on data loading, we assessed how systems handle the original source file. All systems are successful in opening this RFC4180 compliant file, but unfortunately not all of them load header, rows, and cells correctly. The results of these systems can be seen in Table 5. Among parsers, Hypoparsr is the only one unable to detect the header correctly, parsing it as a data row and appending a new header to the file, and also unable to detect the structure of rows containing cells with escaped commas and double quotes. PyCsv and OpenCSV both coincidentally fail in the same row, which contains the special symbol “\” and a delimiter: PyCsv considers the backslash symbol as an escape for the character that follows and ignores it in the resulting cell value; OpenCSV splits the cell into two at the delimiter character, even if the cell itself was properly enclosed in quotation marks. DataViz loads all records erroneously because all values of the *TIME* column, which represents an absolute time in the HH:MM format, are transformed into the values “30/12/1899 HH:MM:00” (HH:MM standing for the original values in the input file).

User takeaways: When loading otherwise standard files with a programming framework, be aware of special symbols usually reserved in the programming language, such as “\”: the framework may require extra escaping on top of what is required for the RFC standard. When loading files in more advanced systems, such as those developed for business intelligence, prepare the file such that its data types are compatible with the system.

Developer takeaways: When parsing content of a csv file, the RFC4180 standard rules should take precedence over those of the language: the value of cells should be interpreted first as a byte string, and then parsed and/or escaped to a more refined data type. Data type parsing should inform users of its “confidence”, perhaps defaulting to the raw cell value when confidence is low.

4.2 File and table pollution

Our benchmark contains 12 files that are polluted at the file and table level, i.e., serialized with non-standard file, table, and data rules (Cf. Rules F0, F1, F2 in Table 3). The left columns of Table 6 report the results on these files: some systems fail to load them altogether. Notably, the Python parsers PyCsv and Pandas along with RCsv and SpreadDesktop and DataViz abort while loading an empty file, while all other systems correctly load it. Interestingly, when the input file shows two trailing newline sequences, PostgreSQL halts due to the presence of empty values in the “time” column – although this error was not thrown while loading the standard source file. After loading, no system can correctly recognize multiple header rows or preamble rows, even those that claim to perform automatic header detection. When no header is present, some systems load data rows with missing cells: e.g., SpreadDesktop, Calc, RCsv, and DataViz drop the empty column. When multiple tables are present, all systems that successfully load them either remove the extra column from the second table, if the first contains more; or add an extra column to the first table if the second contains more.

User takeaways: All systems show high sensitivity to proper “tabular” formatting of files. No matter the system of choice, and its promised automation level, perform the following preparations: condense header lines into one; remove preamble lines; split multiple tables into separate files.

Developer takeaways: Many systems still lack the support for non-standard headers, preamble lines, and multiple tables. For manual loading, we advise implementing interfaces to ignore or specify which rows are to be considered the header, and which ones are to be considered the data rows. The same interface can also be used to load a multitable file without the need to split it into separate files. For automated loading, the most common strategy is to give a higher weight to the first few rows, which then influences how the remainder of the file is parsed. Apart from integrating existing research algorithms to detect row classes, multiple tables, and headers [6, 20, 37], we recommend that developers update the existing algorithms with contextual information.

4.3 Structural characters and inconsistent rows

The remaining files of the benchmark are polluted with structural changes and inconsistent rows. These include file-wise pollutions, and row-wise pollutions. The former affects all rows, i.e., by changing one of the structural characters across the entire file (Cf. Rules F0, F1, F2 in Table 3). The latter affect individual rows, making them inconsistent with the rest of the file, either by having a different number of delimiters, or by having a different structural character. We apply these pollutions to every row/cell combination in the file. This repetition is necessary for fine-grained evaluation: in fact, while e.g., Hypoparsr, incorrectly loads all cells and rows after a misquoted value, other systems, e.g., OpenCSV or SpreadWeb, are more robust and only err on the affected cell/row.

The system with the worst performance is PostgreSQL: it is only successful in loading files where the header has an inconsistent number of delimiters, but if any of the data rows is inconsistent either with an extra or a missing delimiter, it halts the data loading operation. The other database systems are more robust to rows with inconsistent delimiters, loading the record but shifting all cells and/or trimming extra ones. Surprisingly, CSVCommons aborts the loading, but only for the file where the separator is missing from the last header column. For most systems, the headers are not affected by extra delimiters in a data row, except for DataViz, which always includes an extra header cell even if a single data row has an extra separator – leading to a H_{F1} score of 0.57. Observing the results of Table 6, this set of files proves to be the least successful across many csv parsing systems for different reasons: CSVCommons and OpenCSV fail to load any file with an extra quotation mark in one of the rows. As for RCsv, its behavior changes depending on the row affected by the quotation mark: if it is in one of the cells of the header row, it appends all the first data row to the cell but parses the other cells correctly and loads the file. If the extra quotation mark is found in one of the cells of the first four data rows, it halts loading with

Table 6: Pollock results (rounding down) of the 16 systems under test, grouped by pollution type.

	File and table pollution (12 files)				Inconsistent number of delimiters (1 428 files)				Structural character change (850 files)				Pollock score (2 289 +1) files		Average file-wise time (milliseconds)
	S	H_{F1}	R_{F1}	C_{F1}	S	H_{F1}	R_{F1}	C_{F1}	S	H_{F1}	R_{F1}	C_{F1}	Simple	Weighted	
CleverCSV 0.7.4 [35]	1.00	0.75	0.91	0.91	1.00	0.99	1.00	0.99	1.00	0.93	0.57	0.74	**9.19**	**9.45**	69.96 ± 0.13
CSVCommons 1.9.0	0.75	0.50	0.74	0.74	1.00	0.99	1.00	0.99	0.10	0.10	0.10	0.10	**6.64**	**9.25**	23.96 ± 7.64
Hypoparsr 0.1.0 [9]	1.00	0.35	0.30	0.53	1.00	0.07	0.07	0.44	1.00	0.26	0.16	0.69	**3.88**	**4.37**	6 040.15 ± 8.22
OpenCSV 5.6	1.00	0.75	0.90	0.91	1.00	0.99	0.98	0.99	0.10	0.10	0.10	0.10	**6.63**	**7.74**	18.50 ± 2.37
Pandas 1.4.3	0.91	0.67	0.85	0.85	1.00	0.99	0.98	0.99	0.99	0.99	0.97	0.98	**9.89**	**9.43**	1.39 ± 0.17
PyCsv 3.10.5	0.91	0.66	0.78	0.82	1.00	0.99	0.92	0.99	1.00	0.99	0.92	0.98	**9.72**	**9.43**	13.15 ± 0.13
RCsv 4.2.1	0.91	0.58	0.44	0.79	1.00	0.99	0.83	0.98	0.95	0.94	0.49	0.61	**7.79**	**6.40**	8.29 ± 0.61
Univocity 2.9.1	1.00	0.75	0.91	0.91	1.00	0.99	1.00	0.99	0.99	0.99	0.98	0.99	**9.93**	**7.93**	3.16 ± 0.19
MariaDB 10.9.3	1.00	0.75	0.98	0.90	1.00	1.00	0.98	0.88	1.00	0.99	0.97	0.88	**9.58**	**7.48**	20.96 ± 0.05
MySQL 8.0.31	1.00	0.75	0.98	0.90	1.00	1.00	0.98	0.88	1.00	0.99	0.97	0.88	**9.58**	**7.48**	63.96 ± 1.15
PostgreSQL 15.0	0.50	0.33	0.49	0.37	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.00	**0.13**	**6.96**	13.59 ± 0.28
SQLite 3.39.0	1.00	0.66	0.99	0.91	1.00	1.00	1.00	1.00	1.00	0.99	0.98	0.99	**9.95**	**9.37**	353.81 ± 22.54
Calc 7.3.7	1.00	0.74	0.91	0.90	1.00	0.99	1.00	0.99	1.00	0.98	0.98	0.98	**9.92**	**7.83**	2 646.06 ± 14.28
SpreadDesktop	0.91	0.74	0.83	0.74	1.00	0.99	1.00	0.99	0.99	0.98	0.98	0.98	**9.92**	**9.59**	28 776.18 ± 14.28
SpreadWeb	1.00	0.74	0.91	0.86	1.00	0.99	1.00	0.94	0.99	0.97	0.97	0.91	**9.72**	**9.43**	2 949.76 ± 16.29
DataViz	1.00	0.46	0.16	0.64	1.00	0.73	0.00	0.73	1.00	0.57	0.00	0.48	**5.00**	**5.15**	24 411.52 ± 292.67

an error reporting an inconsistent number of delimiters. Otherwise, loading is carried out successfully, but several rows are merged into one, hence the low C_{F1} score. Pandas is unsuccessful with a single file: the one where an extra delimiter is present in the last column of the last row. Curiously, SpreadWeb's only unsuccessful loading is with the file containing an extra quote in row 35. Univocity and SQLite are unable to load a file whose rows terminate with the only carriage return character – a pollution that does not affect any other system's loading capabilities. However, even if their loading does not abort, not all systems can manage inconsistent delimiters and extra quotation characters – apart from the aforementioned cell parsing issues of RCsv, OpenCSV, and Hypoparsr also have a low C_{F1}. These systems all merge the content of subsequent cells, often from multiple rows, if an inconsistent quote or delimiter is found in a given cell. We note that the more robust systems appear to be PyCsv, Pandas, SQLite, and the spreadsheet systems Calc, SpreadDesktop, and SpreadWeb. As observable by the high C_{F1} and R_{F1} scores, the majority of parsing errors are limited within the rows affected by the pollution, while the remaining rows are parsed correctly.

User takeaways: Most systems demonstrate resilience to inconsistent rows: even if the affected rows may have column shifts or lose values, the remaining data rows load correctly. The drawback of such behavior is the lack of feedback from the system's perspective, which may lead to undesirable downstream results. In some cases, these errors can be detected due to unexpected data types (e.g., strings in place of numeric values), but in others they may go unrecognized. We recommend that users pre-emptively check their files for consistency, or set the systems to strict levels of logging to catch these errors as early as possible.

Developer takeaways: We encourage systems (especially the more automated ones) to warn users when loading inconsistent files. We also call for a more advanced parsing to automatically detect "simple" errors, such as missing delimiters or quotation characters, and fix them – perhaps by taking into account the expected structure/-data type of a cell given its row and column context [17].

4.4 Overall Pollock score

The bolded columns of Table 6 report the Pollock score of the systems under test. We report two scores: one as a simple average, and one weighted by the occurrence of pollutions in our real-world survey – therefore depicting a more realistic scenario, as explained in Section 3.4. Different scoring schemes may serve different purposes: although end-users may be interested in the weighted score, to assess real-world performance, parser developers may want to more easily identify "hard" cases, to correct critical bugs.

As a reminder, the score is obtained by summing up 10 different numbers in the $[0, 1]$ range. These numbers correspond to success and precision, recall, and f1 scores at the header, record, and cell levels. Therefore, the maximum score reachable by any system is 10. The last column of Table 6 reports the average file-wise loading-time of the benchmark files. The measurements were obtained by repeating our benchmark three times on a consumer machine equipped with an Intel i7 CPU with 2.20GHz and 16GB of RAM. We explicitly warn readers to not compare systems across different categories: for example, the conditions under which an RDBMS loads files into a table are much more restrictive and require more specification parameters from end-users (e.g., defining the expected table format), than, for example, automated frameworks.

User takeaways: Among specialized csv parsing modules, the ones with the highest scores, and the fastest are Pandas and Univocity. We attribute this result to these systems' development maturity: having been in use for a long time and by a large community, they include safeguards against many pollutions. Comparing different languages, all Python frameworks have good results, while Java frameworks have an overall worse loading performance.

Among databases, SQLite has the best benchmark score, but also the highest average loading time. Interestingly, this system also shows the highest difference between the simple and the weighted scoring schemes: for almost all pollutions with an inconsistent row or cell, its loading failed altogether with a success of 0 (as can be noted by the central columns of the table). Although these pollutions constitute a large number of files, they are also infrequent, which causes a higher weighted score.

Spreadsheet systems generally have good performance: in our observation, their spreadsheet nature made them solid in loading data from csv files. However, they are also amongst the ones with the highest loading times: their user-interfaces make it more cumbersome to load large datasets composed of several files, due to all the interactions needed to specify a correct loading. Finally, business intelligence tools, such as DataViz, are tailored towards cleaner, closer-to-standard data files. Its low score is influenced by an excess of "intelligent" pre-processing that fails with data not in line with the tool's expectations. Therefore, before using such tools we advise users to prepare their files, not only up to the csv standard but also regarding their data types.

Developer takeaways: The lowest scores and the highest loading times among programming frameworks are obtained by systems whose file loading already proved unreliable even for the standard source file itself, e.g. for Hypoparsr and OpenCSV. Within the RDBMS category, the lowest scores are caused by highly restrictive assumptions regarding file structures. For example, PostgreSQL's low success rate is due to halting loading even if a single record is unrecognized by the system. We advise RDBMS developers to be more flexible when loading csv files, and perhaps offer users the option to skip polluted records rather than the whole file, as other benchmarked systems do, e.g. Pandas.

Regarding user-oriented tools, such as spreadsheets and business intelligence systems, one direction of improvement is the inclusion of more sophisticated automated detection strategies. This would improve usability as well as loading time for files, without the need to manually specify parameters through user interfaces.

4.5 Real-world loading

To gain further insight into the loading of real-world files, we tested the different systems with a sample of 100 survey files, which were manually cleaned row by row to provide ground truth for the measurements. We provide the sample and cleaned versions of the files on the Pollock page[7]. The sample was chosen at random, ensuring that all pollutions were represented in at least one of the sampled files. The results of the experiment are reported in Table 7.

As can be seen from the generally lower scores obtained by all systems, real-world files are more challenging to load for several reasons. Considering the variety of errors, we refrain from providing an extensive rundown of all failures, but identify some of the key reasons that lead to imperfect loading. Automatic detection of parameters does not work well in files affected by multiple pollutions at once, such as Pandas delimiter detection failing to recognize a semicolon delimiter for files with inconsistent numbers of delimiters in rows. In other cases, systems have restricted assumptions regarding otherwise standard csv files, such as PostgreSQL failing to load files with duplicate or missing header names. Finally, some systems fail to scale with file dimensions, for example Calc failing to load more than 1m records in a file that contains more than 1.1m records, or MariaDB and MySQL failing to load files if the header name is above 64 characters.

Table 7: Results on a sample of 100 files from our survey.

	S	H_{F1}	R_{F1}	C_{F1}	Po.	Loading time (ms)
CleverCSV 0.7.4 [35]	1.00	0.70	0.96	0.95	**8.89**	840.55 ± 2.23
CSVCommons 1.9.0	0.46	0.26	0.43	0.42	**3.85**	297.81 ± 18.47
Hypoparsr 0.1.0 [9]	1.00	0.51	0.27	0.64	**5.43**	2 288.23 ± 15.67
OpenCSV 5.6	0.98	0.78	0.94	0.93	**9.01**	168.65 ± 5.92
Pandas 1.4.3	0.88	0.49	0.63	0.64	**6.28**	8.70 ± 0.26
PyCsv 3.10.5	0.98	0.67	0.88	0.87	**8.33**	176.82 ± 13.50
RCsv 4.2.1	0.97	0.24	0.52	0.58	**5.05**	25.14 ± 22.56
Univocity 2.9.1	0.95	0.4	0.61	0.63	**5.92**	60.38 ± 1.91
MariaDB 10.9.3	0.70	0.67	0.49	0.61	**6.13**	40.92 ± 9.96
MySQL 8.0.31	0.68	0.64	0.47	0.59	**5.89**	200.62 ± 17.90
PostgreSQL 15.0	0.54	0.51	0.53	0.53	**5.30**	12.00 ± 0.26
SQLite 3.39.0	1.00	0.65	0.73	0.90	**7.96**	342.02 ± 139.91
Calc 7.3.4	1.00	0.44	0.47	0.60	**5.60**	3 358.68 ± 460.75
SpreadDesktop	0.98	0.79	0.53	0.80	**7.41**	28 090.21 ± 51.80
SpreadWeb	0.98	0.68	0.60	0.81	**7.31**	4 846.62 ± 1265.19
DataViz	0.98	0.48	0.11	0.77	**5.15**	28 702.13 ± 294.54

User takeaways: When loading real-world files, several aspects need to be taken into consideration aside from strict adherence to the csv standard. Not every system is scalable for loading larger files (over 1M records, or 100 columns): programming frameworks offer the highest loading accuracy, whereas database systems are faster but require "cleaner" inputs. One possible strategy is to "sanitize" a polluted file by cleaning it through a programming framework before feeding it into more complicated systems, e.g., database systems or business intelligence tools.

Developer takeaways: Regarding automatic systems, we observed that the detection of dialect parameters is often unreliable for files with structural inconsistencies (varying numbers of row delimiters, multiple tables, etc.). We recommend that automated approaches take into account structural pollutions, either by addressing them separately in a preprocessing step, or by filtering for "outlier" rows during their automatic detections. Additionally, we urge developers to relax their assumptions regarding file structure and dimension: as data becomes larger and more ubiquitous, it is not uncommon to expect files with high record counts and column sizes.

5 RELATED WORK

In this work, we formalize a model to describe data serialization grammars, we surveyed thousands of real-world csv files, and we propose a benchmark to measure loading data from non-standard csv files. We identify three corresponding categories of related work: formal research to describe the csv grammar; research to categorize and parse non-standard csv files; grammar-based fuzzing and existing benchmarks for data preparation.

Research on csv standards and grammars: Although csv files have been in use for decades, all efforts to regulate their format have been reactive rather than proactive. The first and best-known document defining a standard is the RFC4180 [29] of 2005. This document already mentions the widespread use and lack of formal specification for csv files and is presented as a consolidation of the

[7] https://github.com/HPI-Information-Systems/Pollock

most common CSV features encountered in practice, rather than an unambiguous standard. Ten years later, the W3C consortium formalized a "non-normative" document to establish a JSON model for tabular data on the internet [34]. Their model extends RFC4180 with stricter specifications of tabular structures and, perhaps more importantly, includes metadata to describe the structure and dialect of the file content itself. Nevertheless, file distributors do not apply the W3C recommendations and hardly ever distribute CSV file metadata. Simultaneously, academia also started formal work on data serialization grammars: Arenas et al. designed a language focused on the metadata description of CSV files [1], while Martens et al. proposed SCULPT, a formal language to describe web tabular data [23]. The former is described as a language to navigate the content of "csv-like" files. Even though it is generally applicable to files with different dialects of CSV, it cannot be used to describe differences in the structure of their grammars. Rather, given the knowledge of a grammar, it provides a tool to reference and annotate content within a file. The latter language is more similar to our framework, and it inspired the definitions of content, structure, and format of a grammar. SCULPT is described as a "schema" language for tabular data: a set of parsing expressions, tokens, and production rules. Although similar to our framework, the focus of a SCULPT schema is to annotate the content of a tabular file, rather than the grammar used to produce it. In contrast, we propose a framework to annotate grammars and their dialects, and not file contents. Therefore, our framework is not tied to any specific grammar or file format, but is applicable to any data serialization/parsing grammar. Moreover, our framework is not only "descriptive", but it is also "generative", as the concept of file pollution can be applied to systematically generate files expressed in different dialects.

Research on CSV parsing: The CSV file format is a known source of data loading problems. For example, Mitlöhner et al. encountered such problems in their survey of publicly available CSV files [24], where, out of 141 738 parsed CSV files, 36 912 (26.04%) were parsed with errors. They report different sources of errors, such as non-standard dialects, incorrect file extensions, and multiple tables within a file. Unlike our survey in Section 3, they do not follow an explicit formal method to categorize file issues.

Over the years, research has tried to address the unique challenges of CSV files: Döhmen et al. proposed the robust parser HYPOPARSR [9]; van den Burg et al. focused on dialect detection for "messy" CSV files with CLEVERCSV [35]; other projects tried to tackle more complex issues, such as table recognition and cell classification in CSV files [6, 20, 37]. We evaluated the first two systems, proposed as CSV loading modules, with our benchmark.

Data preparation benchmarks: Grammar-based fuzzing comprises a set of techniques to generate random inputs that are likely to induce bugs in software, using grammars to ensure the validity of the inputs [19, 33]. Albeit these techniques have been traditionally used in security testing, there has been recent interest in applying fuzzing to benchmark the performance of data analytics workloads [39]. The pollution framework of Pollock can be seen as a particular instance of grammar-based fuzzing [38], specialized for the task of benchmark data loading. The need for such a benchmark has been recently acknowledged by researchers in discussing the state of real-world data preparation [21, 31]. In their survey, Hameed and Naumann compiled a set of common data preparation tasks and evaluated whether commercial tools for data preparation offered the respective functionalities [16]. Although the focus of their survey is on systems specifically designed for data preparation, they identified how significant data preprocessing was required on non-standard data files to enable loading data into said systems. For some individual tasks occurring in a data-driven pipeline, researchers have proposed specific benchmarks. Poess et al. designed TPC-DI, a benchmark for data integration [26]. Its core includes files in heterogeneous formats containing information to feed a target decision support system. The benchmark includes plain-text character delimited files in TXT and CSV format. Because its focus is on system throughput and performance at scale rather than robustness, these files follow the RFC4180 standard, and therefore TPC-DI is not fit to assess a system's data loading capability in face of pollution. Shah et al. focus on benchmarking the type-inference task in AutoML platforms [30]. Their work provides a reference labeled dataset of files usable for machine learning tasks, with a variety of commonly used data types. The benchmark evaluates the performance of an AutoML system by running the same machine learning model twice and comparing the results: once loading data with the correct data types (provided as ground-truth), and once loading data with automatic type inference. The task covered by that benchmark, aside from the specific focus on AutoML tools, constitutes a semantic counterpart to the pollutions of Pollock, as we do not measure whether data types are inferred correctly.

6 CONCLUSIONS

A complete and concise data loading stage is the prerequisite for any further data operation and can substantially reduce the *data preparation* burden of data scientists. We defined a framework to formally distinguish the concepts of content, structure, format, and grammar dialect and use it to systematically categorize issues in real-world CSV files. Second, we apply the notion of file pollution to design Pollock, a benchmark for data loading. After benchmarking 16 real-world systems, our results showed that unsuccessful data loading is often caused by a lack of flexibility in the systems' configurations. Currently, Pollock focuses on single-pollution files. As our experiments in Section 4.5 show, systems struggle more with multiple pollutions at once. Extending Pollock to pollute files with multiple pollutions is an interesting research challenge: it would require notions of pollution dependency and a more complex strategy to sample the search space of pollution combinations, which would be exponentially large. Such an extension could open the door to a "dynamic benchmark" that can tailor specific data preparation scenarios. With Pollock, we hope to stir research efforts in the data preparation area toward a more principled direction. We advocate that with the use of our benchmark, system designers have an objective metric to assess the data loading capabilities of their tools, as well as a means to identify unexpected and surprising behaviors, as we did in our experimental results.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. 2016. A Framework for Annotating CSV-like Data. *PVLDB* 9, 11 (2016), 876–887.

[2] Jeronimo Backes. 2022. *Univocity CSV Parser.* www.univocity.com (last accessed Apr 14th, 2022).

[3] T. Bray. 2017. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259. RFC Editor. https://www.rfc-editor.org/rfc/rfc8259.txt

[4] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (1956), 113–124.

[5] Noam Chomsky and Marcel P Schützenberger. 1959. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics.* Vol. 26. Elsevier, 118–161.

[6] Christina Christodoulakis, Eric 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.

[7] Oracle Corporation. 2022. *MySQL.* www.mysql.com (last accessed July 12th, 2022).

[8] UK Crown. 2021. *UK Open Data portal.* data.gov.uk (last accessed July 30th, 2021).

[9] 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).* ACM, 1–12.

[10] Apache Software Foundation. 2022. *Apache Commons CSV.* https://commons.apache.org/proper/commons-csv/ (last accessed Jul 12th, 2022).

[11] MariaDB Foundation. 2022. *MariaDB.* www.mariadb.org (last accessed July 12th, 2022).

[12] The Document Foundation. 2022. *LibreOffice Calc 7.3.4.* https://www.libreoffice.org/discover/calc/ (last accessed July 12th, 2022).

[13] 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.

[14] Majid Ghasemi Gol, Jay Pujara, and Pedro Szekely. 2019. Tabular cell classification using pre-trained cell embeddings. In *Proceedings of the International Conference on Data Mining (ICDM).* IEEE, 230–239.

[15] The PostgreSQL Global Development Group. 2022. *PostgreSQL.* https://www.postgresql.org (last accessed July 12th, 2022).

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

[17] Mazhar Hameed, Gerardo Vitagliano, Lan Jiang, and Felix Naumann. 2022. SURAGH: Syntactic Pattern Matching to Identify Ill-Formed Records. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* OpenProceedings.org.

[18] Richard D Hipp. 2022. SQLite. https://www.sqlite.org/index.html

[19] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *A-TEST@ESEC/SIGSOFT FSE.* ACM, 45–48.

[20] 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).* OpenProceedings.org, 193–204.

[21] Arun Kumar. 2021. Automation of Data Prep, ML, and Data Science: New Cure or Snake Oil?. In *Proceedings of the International Conference on Management of Data (SIGMOD).* ACM, 2878–2880.

[22] Mendeley Ltd. 2021. *Mendeley Data.* data.mendeley.com (last accessed July 30th, 2021).

[23] Wim Martens, Frank Neven, and Stijn Vansummeren. 2015. SCULPT: A Schema Language for Tabular Data on the Web. In *Proceedings of the 24th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, Florence Italy, 702–720.

[24] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. 2016. Characteristics of open data CSV files. In *Proceedings of the Image Analysis and Processing Conference (ICIAP).* 72–79.

[25] Marvin C. Paull and Stephen H. Unger. 1968. Structural Equivalence of Context-Free Grammars. *Journal of Computer and System Science (JCSS)* 2, 4 (1968), 427–463.

[26] Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, and Brian Caufield. 2014. TPC-DI: The First Industry Benchmark for Data Integration. *PVLDB* 7, 13 (2014), 1367–1378.

[27] R Core Team. 2022. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/(lastaccessedJuly2022)

[28] Jeff Reback, jbrockmendel, Wes McKinney, Joris Van den Bossche, Matthew Roeschke, Tom Augspurger, Simon Hawkins, Phillip Cloud, gfyoung, Sinhrks, Patrick Hoefler, Adam Klein, Terji Petersen, Jeff Tratner, Chang She, William Ayd, Shahar Naveh, JHM Darbyshire, Richard Shadrach, Marc Garcia, Jeremy Schendel, Andy Hayden, Daniel Saxton, Marco Edward Gorelli, Fangchen Li, Torsten Wörtwein, Matthew Zeitlin, Vytautas Jancauskas, Ali McMaster, and Thomas Li. 2022. *pandas-dev/pandas: Pandas 1.4.3.* https://doi.org/10.5281/zenodo.6702671

[29] Y. Shafranovich. 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files.* RFC 4180. RFC Editor. 1–8 pages. http://www.rfc-editor.org/rfc/rfc4180.txt

[30] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. In *Proceedings of the International Conference on Management of Data (SIGMOD).* ACM, 1584–1596.

[31] Lisa Singh, Amol Deshpande, Wenchao Zhou, Arindam Banerjee, Alex Bowers, Sorelle Friedler, H.V. Jagadish, George Karypis, Zoran Obradovic, Anil Vullikanti, and Wangda Zuo. 2019. NSF BIGDATA PI Meeting - Domain-Specific Research Directions and Data Sets. *SIGMOD Record* 47, 3 (2019), 32–35.

[32] Glen Smith, Scott Conway, Andrew Rucker Jones, Sean Sullivan, Kyle Miller, Tom Squires, Kyle Miller, Maciek Opala, and J.C. Romanda. 2022. *OpenCSV - Project page.* http://opencsv.sourceforge.net (last accessed Jul 12th, 2022).

[33] Ezekiel O. Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2022. Inputs From Hell. *IEEE Trans. Software Eng.* 48, 4 (2022), 1138–1153.

[34] Jeni Tennison, Gregg Kellogg, and Ivan Herman. 2015. *Model for Tabular Data and Metadata on the Web.* W3C Recommendation. W3C. https://www.w3.org/TR/2015/REC-tabular-data-model-20151217/.

[35] G. J. J. van den Burg, A. Nazábal, and C. Sutton. 2019. Wrangling Messy CSV Files by Detecting Row and Type Patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.

[36] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.

[37] Gerardo Vitagliano, Lan Jiang, and Felix Naumann. 2022. Detecting Layout Templates in Complex Multiregion Files. *PVLDB* 15, 3 (2022), 646–658.

[38] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. *The Fuzzing Book.* CISPA + Saarland University, Saarbrücken. https://publications.cispa.saarland/3120/

[39] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *ASE.* IEEE, 722–733.