A Method and Tool for Automated Induction of Relations from Quantitative Performance Logs

Joshua Kimball(✉) and Calton Pu

Georgia Institute of Technology, Atlanta, GA 30332, USA
{jmkimball, calton.pu}@gatech.edu

Abstract. Operators use performance logs to manage large-scale web service infrastructures. Detecting, isolating and diagnosing fine-grained performance anomalies require integrating system performance measures across space and time. The diversity of these logs layouts impedes their efficient processing and hinders such analyses. Performance logs possess some unique features, which challenge current log parsing techniques. In addition, most current techniques stop at extraction leaving relational definition as a post-processing activity, which can be a substantial effort at web scale. To achieve scale, we introduce our *perftables* approach, which automatically interprets performance log data and transforms the text into structured relations. We interpret the signals provided by the layout using our template catalog to induce an appropriate relation. We evaluate our method on a large sample obtained from our experimental computer science infrastructure in addition to a sample drawn from the wild. We were able to successfully extract on average over 97% and 85% of the data respectively.

Keywords: Information integration · Data cleaning · Data extraction

1 Introduction

Our experimental computer science infrastructure, *elba*, generates huge volumes of data from large numbers of diverse experiments and systems topologies, which support our empirical-based method for understanding computer systems' more fundamental behavior. As we show later, we have run over 20,000 experiments on *elba* over the last three years generating over 100 TB of data spread across 400K various log files. To isolate and diagnose nuanced, fine-grained performance anomalies, we need to support a broad array of experimental configurations, since these bugs can materialize under a range of conditions. For example, experimental artifacts like logs can vary in number and layout *per experiment* making data extraction and subsequent analysis challenging to perform at scale. Recent approaches like DeepLog operate over arbitrary text and attempt to isolate "macro-level" system events like crashes [1]. Our automated relation induction approach, *perftables*, operates over the diverse performance monitoring outputs with the objective of isolating much more precise (shorter and transient) events.

D. Da Silva et al. (Eds.): CLOUD 2019, LNCS 11513, pp. 11–25, 2019.
https://doi.org/10.1007/978-3-030-23502-4_2

The layout diversity observed across these performance logs stems from our infrastructure's enormous experimental parameter space and its diversity of instrumentation. Resource monitoring is one particularly good illustration. Elba infrastructure currently features five resource monitors: iostat, systat (sar), collectl, oprofile and lockmon. Each execution (of a given monitor) can have very different output even though each of these programs accepts a fixed number of parameters. For example, toggling a runtime parameter to change the resources being monitored alters the layout of the monitor's log file. Assuming each resource monitoring decision is binary, there can be as many as 2^n possible layouts for a performance monitor capable of measuring up to n resources. (From this point forward, layout and format are used interchangeably.) Given this, the number of possible layouts is exponential in the number of resources being monitored. This makes a naïve approach of writing a parser for each unique format simply intractable. In our data set, we have found the number of distinct layouts to number in the hundreds (under the most conservative accounting). Data variety and volume at our scale impedes automated data extraction and subsequent data analysis, creating an enterprise data-lake-scale data management challenge for our infrastructure [2]. The longer data remains unprocessed, the more unwieldly its management becomes [3].

Previous Work. Approaches from previous work in automated information extraction has generally fallen into one of two categories: wrapper induction and supervised learning techniques [4, 5]. Wrapper induction techniques have been applied to documents with explicit record boundaries like HTML [4, 6, 7]. Supervised techniques have been applied to similar domains. Work on extracting relations from web lists also shares some parallels with work on information extraction [8].

Previous work from the systems and programming language communities on log dataset extraction also feature similar work [9]. Work from the systems community has generally relied on source code interposition techniques to decorate logging statements corresponding to specific string literals which are found in the output [10]. Work from the programming language community has primarily used examples to either synthesize transformations or automatically generate a transformation program from user provided transformation actions. RecordBreaker is one such example of this latter approach [9], and it along with Datamaran are the most similar approaches to ours [11].

Example 1.1. Most previous work assumes record boundaries have been established beforehand or can be easily established using repeated patterns found in explicit structures such as the HTML DOM tree. As Gao et al. explain, log files have no natural record boundaries or explicit mechanisms like HTML tags for determining them [11]. In addition, log files can have nested structures and variable length records, i.e. records which span a variable number of rows. Log files also include noise such as formatting concerns and various metadata as shown in Fig. 1.

Performance logs present some specific and unique challenges. First, performance logs output formats are impacted by two implicit factors: the computer architecture of the system components being monitored and the actual behavior of the system under study. This latter characteristic suggests layout is at least partially runtime dependent, thus the layout of a given performance log for a given execution is not known a priori.

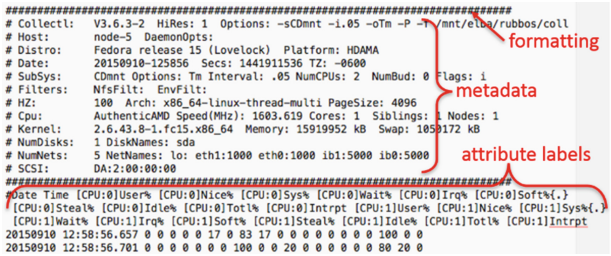

Fig. 1. collectl multi-core

Example 1.2. Figures 1 and 2 shows the performance output from the same performance monitor bootstrapped with the same monitoring parameters but running on different systems. Figure 1 displays the output for the multicore system while Fig. 2 depicts the single core system. Clearly, the output is significantly different holding all else constant. The impact of these implicit factors on the layout of the output underscores the need for an unsupervised approach.

```
# SINGLE CPU STATISTICS
#Time            Cpu  User Nice  Sys Wait IRQ  Soft Steal Idle
20:54:06.403      0     9    0    9    0    0    0      0   81
20:54:06.502      0    10    0   10    0    0    0      0   80
```

Fig. 2. collectl single core

Secondly, performance logs often contain multiple, related record types. In addition, record types can have degenerative sub-structures such as variable length attributes. This characteristic only adds to the complexity of comparing records once they are found. Prior work has assumed records are independent, so this work contains no mechanism for evaluating the relationships among records. This step is critical to realizing an end-to-end unsupervised approach. Information must be able to be extracted and directly represented in relations.

Example 2. Figure 4 shows a snippet of a log file containing process and context switch data from two sampling periods. First, notice that each active process appears on a separate line. Since each sampling period has a different number of active processes, each sampling period spans a variable number of lines. Also, the sampling period is the record in this case. Under a record type independence assumption, each region of data, i.e. the regions containing data about context switches and processes respectively, would be treated as their own record types. In this case, the two sampling periods not the data regions constitute the two record structures, which also happen to span a variable number of rows, i.e. variable length records.

While assuming independent record types is suitable for simple extraction, it is impractical at our scale. In our case, once the data has been extracted, it would still require significant transformation to get it into the correct relational form. This last example also demonstrates the need for an approach to identify record boundaries over

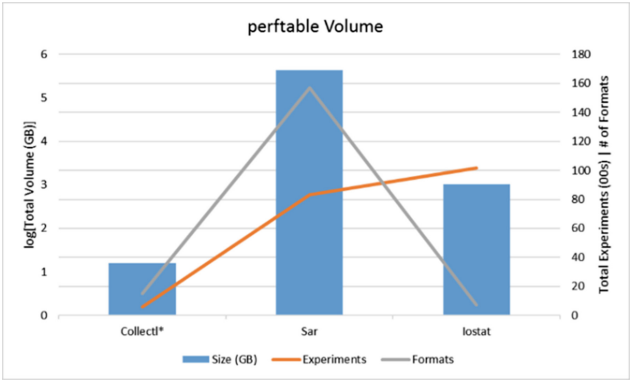

Fig. 3. perftables usage

a (potentially large) variable number of rows due to the impact of runtime factors can have on the layout.

Approach Overview. In this paper, we present *perftables*—our unsupervised algorithm for automatically inducing relations directly from performance monitoring log files. Our method goes beyond extraction as our unsupervised approach constructs tables directly from the observed data.

To accomplish this objective, we have defined a small set of pattern-based templates. We use a set of delimiters to first convert text into tokens. Then we transform each token sequence into a sequence of data type labels by applying a series of data type functions to each token. Next, we lazily match these sequences of data labels to one of our templates based on similarity. Once data has been matched to a template, the template can be used to extract the data and separate semantically meaningful metadata from "noise." To detect record boundaries, we induce a graph over the matching template instances. Finally, we construct relations from the template-matched data according to the record structure detected in the graph.

Generally, our method differs from previous work in its ability to handle logs with runtime dependent layouts. Due to the impact of runtime behavior on log output, multiple record types and variable length records are particularly prevalent in performance logs. For example, we have observed over 100 distinct layouts generated from several distinct runtime configurations across 3 monitoring programs. Specifically, *perftables* does not depend on pre-defining record boundaries. Moreover, it does not assume record boundaries appear over some constant, fixed number of lines. Its lazy approach obviates the need for such a hyperparameter.

Secondly, our method goes beyond extraction and induces relations directly from the log text data. Previous work has relegated schema definition and data transformation to manual post-extraction tasks—a significant burden at our infrastructure's scale. To analyze experimental data at our scale, we require an approach that can extract and transform unstructured log data into structured (relational) data with as little human supervision as possible. As Fig. 3 shows, we have used *perftables* to successfully extract with more than 98% accuracy over 250 GB of data from over 1 TB of log data.

In this paper, we demonstrate how our approach efficiently, accurately and automatically identifies and extracts relations from performance log files. Specifically, we have developed a small set of layout pattern-based templates, which support data extraction and attribute identification. Secondly, we have developed a set of algorithms to automatically identify record boundaries even in the presence of irregular and variable length records. We also show how our templates support automatically defining relations from matching data. Finally, we demonstrate the effectiveness (accuracy and efficiency) of our templates inside our environment and provide coverage for performance log data beyond our domain.

```
Linux 2.6.32-279.22.1.el6.x86_64 (Mysql2)   04/14/2014

09:42:47 PM   cswch/s
09:42:48 PM   2080.00

09:42:47 PM       PID  minflt/s  majflt/s     %user   %system   nswap/s   CPU
09:42:48 PM         1      0.00      0.00      0.00      0.00      0.00     0
....{95}....
09:42:48 PM     14615    294.00      0.00      0.00      0.00      0.00     0
....
09:45:18 PM   cswch/s
09:45:19 PM   1819.00

09:45:18 PM       PID  minflt/s  majflt/s     %user   %system   nswap/s   CPU
09:45:19 PM         1      0.00      0.00      0.00      0.00      0.00     0
....{90}....
09:45:19 PM     14631      0.00      0.00      0.00      0.00      0.00     0
```

Fig. 4. Long variable length records. Intermediate rows, indicated by braces, were removed for space considerations.

2 Terminology and Problem Statement

In this section, we will formally define our problem of unsupervised table extraction from performance log files.

Definition 2.1 (Data Type Sequence). By applying one or more delimiters to a string, it can be transformed into a sequence of tokens. This process is typically referred to as tokenization.

A best-fitting data type description can be estimated for each token by applying a data type function to each one. For example, if a token consists of the characters "123" then a data type description function might return "INT" to indicate integer as the best-fitting data type for this character sequence. By applying a data type function to every token in a sequence, a sequence of data type labels can be constructed. We refer to this sequence of labels as a data type sequence for short.

Definition 2.2 (Layout Template). A Layout Template is a regular expression for data type sequences. We say the data type sequence matches a layout template *iff* the regular expression of a layout template matches the string form of a data type sequence.

Definition 2.3 (Layout). A layout is a specific arrangement of data. Formats or layouts like those depicted in the previous Figures use formatting characters like whitespace and other special character delimiters like "#" or ":" and the order of metadata and data and their orientation to accomplish two objectives: partition data

from metadata and metadata from "noise" and express relationships among the data. For example, metadata which immediately precedes data can be assumed to describe the data that follows it. In short, a layout is a sequence such that order can be used to partition the sequence into data and metadata constituent parts.

Formally, a layout, L, consists of text that can be divided into rows separated by newline characters, i.e. "\n." A layout consisting of n rows is $<r_1, r_2, ..., r_n>$. Applying some tokenization function, f, to the i^{th} row r_i results in m tokens $<t_{i1}, t_{i2}, ..., t_{im}>$, and applying some function g to one or more successive t_{ij} determines its membership in M or D, the sets of metadata and data respectively.

Example 3. In the performance monitoring domain, layout explicitly encodes or aligns the measurements to corresponding resources. It expresses relationships among data visually. In Fig. 2, each line expresses the relationship between time and a magnitude for each of the resources being measured. Specifically, at 20:54:06.403, the CPU utilization is 19%, i.e. 100% - Idle%. It also shows the components of this utilization: User and (Sys)tem. Since the values appear on the same line, the layout is expressing a co-occurrence between these components of utilization at time, 20:54:06.403. In the multi-core case in Fig. 1, we see each CPU core (and corresponding components of utilization) are represented as separate columns. Once again, the layout expresses a co-occurrence among these cores' measurements at time 12:58:56.657. In both Figures, the preceding labels describe the data, and more specifically, that a label at a specific position corresponds to data at the same position in a subsequent row. The presence of labels provides an additional important signal. Specifically, knowing labels exist and their location in a file provides information about the location of the data they describe. Moreover, labels immediately preceding values in a tabular-like orientation suggests order can be used to match values to labels—an important signal that could be used during processing. In this respect, these files exhibit some self-describing characteristics.

Definition 2.4 (Log Data). Consider a file F with m layouts $<l_1, l_2, ..., l_m>$. Given our layout definition, interpreting each of the file's layouts can help us separate data from metadata and segregate useful labels from other metadata. Our goal is to find a layout that most closely matches the observed data, so it can be used to extract a Table T from this data. This is a subjective goal as solutions will have a different number of tables, columns and records. We obviously want to maximize the amount of information that can be reliably extracted. Instead, we need to formulate the problem as an optimization task.

Problem. The task is to find the best fitting Layout Template or Templates given the text. Once we have matched a template to an observed layout, we can use the template to construct a table T containing some number of columns and a maximal number of rows from the matching data. So, our refined problem is to extract a table T from the given log data using the best fitting layout L so that the number of extracted tuples is maximized.

3 Model

In this section, we identify the layout patterns our model covers and its assumptions. Our model reduces layout patterns into sequences of coarse-grained data type labels. By casting each token to a best-fitting data type, we can begin to "see" the format patterns more explicitly. Our model also includes a collection of Layout Templates that are expressed as regular expressions over the same alphabet as the one for data type labels. Layout templates not only express data composition but also a specific ordering. These model components combined with a few other reasonable assumptions enable us to automatically extract relations from performance log data.

3.1 Visual Structural Cues and Layouts

Performance log files are often formatted to support human readability and comprehension. As such, humans can use visual cues provided by a file layout to easily separate data from metadata. Unfortunately, performance log text does not explicitly and consistently identify the regular structures that are visually obvious.

We can view the problem of automatically extracting data from performance log files as one of interpreting the file layout. Our task is to find a mechanism to convert the visual cues provided by the layout into something more explicit to support automated detection and extraction. As we will show, the arrangement or sequence of data types seems to sufficiently approximate the layout's visual cues.

We previously defined layout as a sequence of metadata and data in which order can be used to differentiate data from metadata. Figure 5 depicts some of the most common layout structures appearing in performance logs. Each "row" in the figure represents a line, and each "cell's" shading indicates whether it represents a data (grey) or metadata (lavender) element. Gaps among the cells indicate breaks or irregularities, i.e. NULLs. Each example in the figure can be described by their orientations of data to metadata: tabular, horizontal, vertical, or series of independent tabular structures. We use these graphical models as a basis for defining our collection of Layout Templates, which relate sequences of data types to data and metadata distinctions.

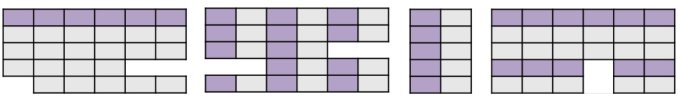

Fig. 5. Common layouts (Color figure online)

3.2 Layout Templates

To support our broader identification and segmentation tasks, we have defined a set of *data layout templates*, or *layout templates* for short, to codify each of the layouts pictured in Table 1. Specifically, our templates are defined using regular expressions over the same alphabet used for representing data type sequences (S, D, N). We name these patterns after the basic data type sequences they describe.

Table 1. Template definitions

Type	*Pattern*
Uniform (*U*)	U = {S+ | (D | N)+}
Alternating (*A*)	A = {D?(SN)+ | D?(NS)+}
Tag (*G*)	G = {S(D | N)+}
Tabular (*T*)	$U_{Si,}$ U_{N+j} where 0 <= i < j < n
Horizontal (*H*)	$A+_{i,}$ $A+_{i+1}$ where 0 <= i < n
Vertical (*V*)	$G+_{i}$, $G+_{i+1}$ where 0 <= i < n
Series	$(T_i \mid H_i \mid V_i)+$ where 0 <= i < n

Note: Series template describes sequences of other layout templates.

For example, Fig. 2 could be expressed as a sequence of four matching patterns: U_S, U_S, U_N, U_N. (A note on notation: an alphabetic subscript on a basic pattern refers to the specific branch taken in the pattern definition.)

Our templates express varying degrees of restrictiveness. This follows the intuition that the more structure or regularity the data exhibits, the more specific the matching rules can be. Accordingly, our templates can be applied using the best fit principle.

3.3 Assumptions

Our method makes several assumptions about files and their layouts, which enable the application of our templates. While these assumptions might appear to be restrictive, we demonstrate in our evaluation that all log files in our sample, including those collected from the wild, respect these assumptions.

Layout Templates Coverage Assumption. This assumption makes explicit the set of files our approach covers. Our approach begins by assuming files observe a left-to-right, top-to-bottom orientation. Specifically, our method covers files that match our Series layout template. Stated differently, our method can process log files that can be expressed as an ordering of our Layout Templates.

For our method to achieve its ultimate objective to automatically recover relations, the order of data in files of interest needs to matter; order must have semantic meaning. This assumption originates from our definition of a Layout. Our coverage assumption not only restricts the potential layouts our method covers, but it also bounds the search space and limits the set of files from which we can automatically materialize relations.

Token Creation Assumption. This assumption concerns the process of applying some regularly occurring delimiter to split a text of interest into tokens. Specifically, we assume each character in a text is either used for formatting or as part of a data value. Under this assumption, a character used as a delimiter cannot also be part of a data value.

Said differently, characters used as delimiters should not split semantically meaningful data. For example, using a colon ":" as a delimiter on text containing datetime would split semantically meaningful tokens, since the corresponding date entity is now

represented as a series of independent tokens. Under this assumption, a colon character needs to either be a delimiter or part of a value for a given text.

While this assumption seems to be restrictive, we add flexibility by limiting the context under which the assumption must be true. Previous work has assumed delimiter characters need to be pre-identified or apply uniformly to a file. This has typically been referred to as tokenization or chunking. In our case, we assume the context for evaluation is the text between two consecutive newline characters, i.e. a line of a file.

In our domain, whitespace is frequently used for formatting and layout purposes. From this experience, we have found using whitespace characters as delimiters usually respects this assumption.

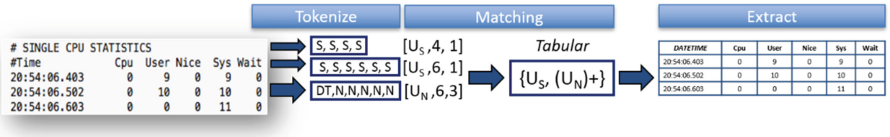

Fig. 6. perftables approach

4 perfTables Approach

Our approach consists of four steps: tokenizing the file, matching data type sequences to layout templates, identifying candidate relations and records and finally extracting relations. Our layout templates are projected onto a file after a file has been transformed into sequences of data type labels. We create this tokenized representation by applying user-provided (or a default set) of delimiters to the file. Next, we create sequences of data type labels by inferring the best fitting coarse-grained data type for each token. We match our Layout Templates to these data type label sequences using a backtracking approach. Once data type sequences have been matched to templates, we use information from the matched data to identify candidate relations and their constituent records. Finally, we this information and the matchings to form relations.

4.1 Token Creation

First, a file is broken up into tokens using either a set of user-provided or default delimiters. The default set consists of white space characters, pipe (|), comma and quotation marks.

Specifically, each line can be converted into a "row of tokens" by applying one or more of these delimiters to it. After the file has been tokenized, we now consider each line of the file to be a row. Specifically, a row r with m tokens is expressed: $r = <t_0, t_1, t_2, \ldots, t_{m-1}>$.

The default delimiters are used to bootstrap or initialize our method. Users can supply supplemental delimiters; however, we have found our default set to be reliable for performance log data.

Sequences of Data Type Labels. Each token in the row can be evaluated for fit among three coarse-grained data types: DATETIME, NUMBER and STRING. We represent each token in the row with a label corresponding to the best fitting coarse-grained data type: S for STRING, D for DATETIME and N for NUMBER. At the end of this encoding step, each row is represented by a sequence of S, D and N characters. We call these sequences of data type labels sequences for short.

The next step in our method involves analyzing these patterns for the implicit semantic clues expressed in the layout. For example, a row with the sequence S, N, S, N, S, N describes a sequence of alternating STRING and NUMBER data. This layout suggests data is located at the positions corresponding to the "N" labels, and its metadata is located at the "S" label positions. (Note: the preceding sequence can be expressed by the regular expression (SN)+ which also corresponds with our Alternating pattern definition.) The next step in our approach involves evaluating these sequences by matching these data type labels to our Layout Templates' regular expressions.

4.2 Model

During this step, we interpret the sequences and match them to our templates to identify candidate tables. The objective of this step is to identify those rows that "belong together."

Matching Sequences to Layout Templates. After tokenizing the file, we try to match sequences of data type labels to the best fitting layout template. Not knowing a file's layout *a priori* motivates the need for a lazy, adaptive approach to matching. Accordingly, these sequences are lazily evaluated according to their topological order.

Backtracking. We match sequences to templates using a backtracking algorithm. This approach optimizes the best-fitting template through a process of elimination. We evaluate sequences according to their topological order. Each data type label sequence's string form is matched to each of the regular expressions accompanying each template definition. Only matching templates are preserved until only one remains. The process restarts once a sequence invalidates the remaining template, but not before the remaining template and its span of matching rows is added to an array of template, row span tuples. Once all rows' sequences have been matched to templates, a *table candidate* can be induced from the constituent rows corresponding to each matching template, row span tuple.

4.3 Extracting Relations

Besides helping to isolate common patterns, our Layout Templates provide another important function. They provide some of a matching file's missing semantic information. Specifically, they use the location and position of matching data to impart relational model semantics. For example, based on data type, composition and position, a piece of matching text might be used as attribute labels. These mapping rules also support data alignment, i.e. determining which labels (if they exist) correspond to which data. In this respect, our templates provide a convenient abstraction for aligning matching data to the constructs of the relational model.

Each template specifies how matching data can be separated into attributes and attribute labels. For example, some of our definitions use sequence or a common token index to align labels and corresponding attribute data.

Candidate Attribute Labels. Each template includes rules for identifying the location of *label candidates*. Each label candidate must be a string, but each string is not necessarily a label candidate. The mapping rules accompanying each template make this noise, label or data distinction. For example, in Fig. 6, the rows with similar size, {Us, 6, 1} and {Un, 6, 3}, were paired, and the row matching a uniform string pattern can be conveniently used as semantically appropriate attribute labels in defining a relation for the data matching {Un, 6, 3}.

Given this mapping between our templates and relational model constructs, we can automatically infer a schema (one or more relations) directly from this log data. For now, we assume each instance of a template matching data is independent. We show in our Evaluation how our approach accommodates situations when this is not true.

5 Evaluation

We explore our method's performance primarily along two dimensions: accuracy and processing time. We assess its performance across two different datasets. The first data set originates from our substantial experimental systems infrastructure. We use it to conduct a broad array of systems experiments to assess experimental systems performance. To support this work, we need to collect an enormous amount of performance data.

5.1 Elba Dataset Characteristics

As Fig. 3 shows, we have used *perftables* to process tens of thousands of experimental systems' performance data on the order of hundreds of GBs. Our experimental systems infrastructure primarily relies on three monitoring software programs: collectl, sar (systat) and iostat. As we briefly discussed earlier, these monitors can generate a large variety of layouts. Figure 7 shows the log diversity generated by these three monitors in our environment.

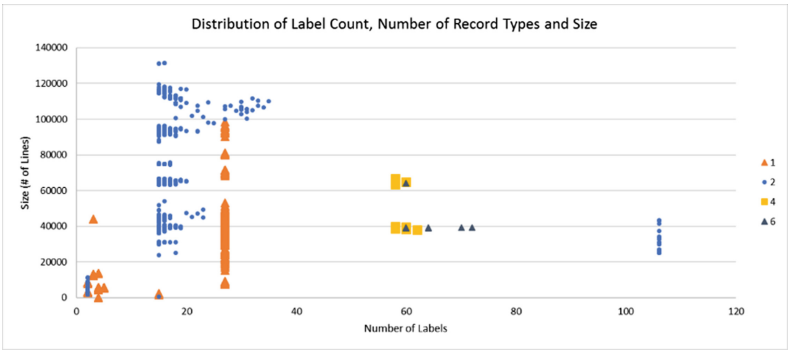

Fig. 7. Data volume and variety for three Elba performance monitors

Gao et al. developed a convenient categorization for describing layout variety. We adapt their categorization as follows: we differentiate interleaved and non-interleaved records precisely. In the following graphs, non-interleaved record structures are represented by "1," and in the interleaved case, we enumerate the number of record structures present in files to illustrate the variety of layouts more explicitly. We don't explicitly separate files with single lines from those with multiple lines in the interleaved case (Fig. 8).

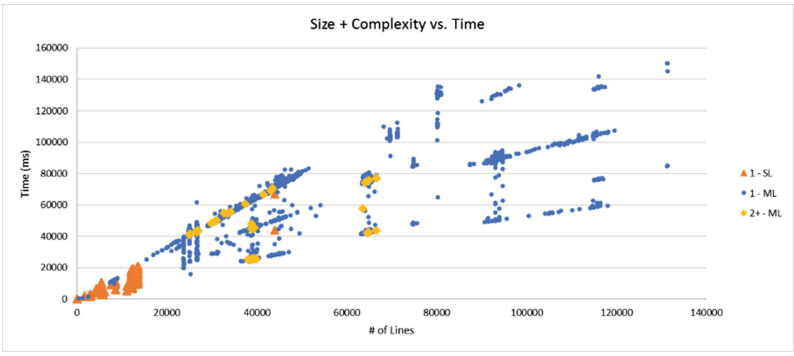

Fig. 8. Variety and size on performance (Elba)

Runtime and Accuracy. Figure 9 shows *perftables* performance by varying size and variety. The sub-linear trend highlights the effectiveness of our lazy approach.

We consider variety in terms of the number of repeated record structures that appear in a file. Even files with multiple record structures, *perftables* performs in sub-linear time.

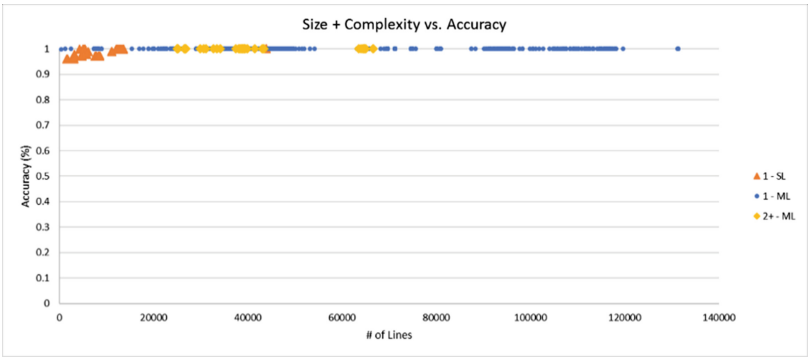

Fig. 9. Variety and size on accuracy (Elba)

5.2 GitHub Dataset Characteristics

The second dataset comes from the wild via a popular public source code repository, GitHub. We eliminated those files from the sample that did not originate from performance monitoring programs. This small, quasi-random sample helps us assess the validity of our assumptions by testing the generalizability of our approach. We retrieved this dataset by querying Github for keywords such as "log," "nagios," and "top." The latter two terms refer to two popular open source resource monitoring tools. Given their wide spread use, we thought they should be included in our sample. Table 2 details our GitHub sample's characteristics. While our quasi-random sample was small, it covers 7 unique monitors not currently deployed in our infrastructure, including: top (profile and process), vmstat (vmware), oprofile, nagios, logstat (kvm) and a bespoke CPU/network monitoring tool.

Table 2. GitHub sample characteristics

Record types	# of samples	Average size (# of lines)
One (single line)	5	3500
One (multi-line)	2	3000
Two or more	1	4000

Github Accuracy. In the following figure, we demonstrate the ability of our method to extend beyond the monitors and their corresponding outputs defined in our environment. On average, we were able to correctly extract over 90% of the data obtained from GitHub into relations. We were only able to extract approximately 75% of the nagios log data due to our approach treating network message labels as attribute labels instead of elements of an enumeration. Despite this result, repairing this error can be accomplished with some simple post-processing (Fig. 10).

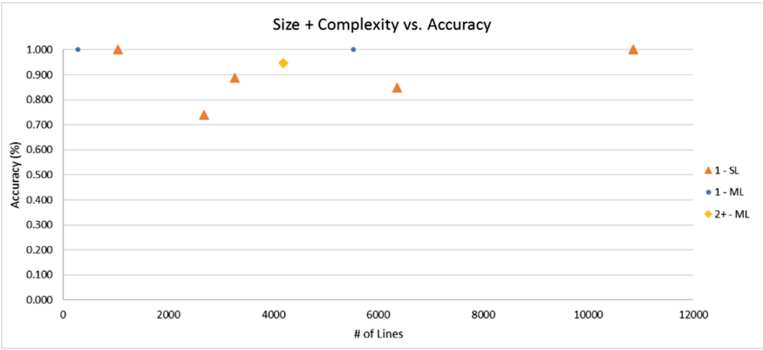

Fig. 10. Variety and size on accuracy (GitHub)

6 Related Work

Previous work has formulated similar log data extraction problems as uncovering some unobserved "template" (structure) from the observed text (data). Some previous work has relied on machine learning techniques to divine such structure from the data [12, 13]. There are two general limitations with such approaches. First, they are dependent on the availability of data, so in this sense, these methods are biased based on the composition of the corpus [8, 14]. Secondly, using supervised machine learning methods usually means manually curating labels for a training data set, and this data set should contain enough variety to account for potential bias. Unfortunately, to the best of our knowledge such a data set is not publicly available, and given the enormous parameter space, it is not likely to be sufficiently varied if one did exist.

Instead of trying to "learn" structure (bottom-up) as others have done, we project structure onto the text (top-down) using a collection of pattern templates. These templates correspond to frequently occurring layouts. Previous work in automated data extraction has used some regularities among HTML tags to separate data from its presentation [15]. For example, groups of specific HTML tags signal more regular substructures like tables and lists. In this respect, our types function like these groups of HTML tags [5].

7 Conclusion

In this paper, we introduced our approach, *perftables*, for automatically inducing relations from the log data generated by commonly used performance monitoring tools. The reasons for extracting this data into a relational form are many: facilitates integrating event and resource data across space (which node/ component) and time (when did the event happen or when was the measurement made), supports automated analysis techniques like machine learning and ultimately enables researchers to glean patterns across a vast volume of experiments occurring over many years.

We demonstrated that we can successfully extract over 98% of our experimental infrastructure's performance data into relations despite the presence of variable length records and multiple record types. Finally, we have shown that our approach extends beyond the array of performance monitors present in our infrastructure by collecting a sample of other performance monitoring logs from the wild.

References

1. Du, M., Li, F., Zheng, G., Srikumar, V.: DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017)
2. Rivera, J., Meulen, R.: Gartner says beware of the data lake fallacy. Gartner (2014). http://www.gartner.com/newsroom/id/2809117
3. Stein, B., Morrison, A.: The enterprise data lake: better integration and deeper analytics. PwC Technol. Forecast. Rethink. Integr. **1**, 18 (2014)

4. Agichtein, E., Gravano, L.: Snowball: extracting relations from large plain-text collections. In: Proceedings of the Fifth ACM Conference on Digital Libraries (2000)
5. Arasu, A., Garcia-Molina, H.: Extracting structured data from web pages. In: SIGMOD Conference (2003)
6. Liu, L., Pu, C., Han, W.: XWRAP: an XML-enabled wrapper construction system for web information sources. In: Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073) (2000)
7. Han, W., Buttler, D., Pu, C.: Wrapping web data into XML. ACM SIGMOD Rec. **30**, 33–38 (2001)
8. Cafarella, M.J., Halevy, A., Wang, D.Z., Wu, E., Zhang, Y.: WebTables: exploring the power of tables on the web. Proc. VLDB Endow. **1**, 538–549 (2008)
9. Fisher, K., Walker, D., Zhu, K.Q., White, P.: From dirt to shovels - fully automatic tool generation from ad hoc data. In: POPL (2008)
10. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: an online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS) (2017)
11. Gao, Y., Huang S., Parameswaran, A.G.: Navigating the data lake with DATAMARAN - automatically extracting structure from log datasets. In: SIGMOD Conference (2018)
12. Chu, X., He, Y., Chakrabarti, K., Ganjam, K.: TEGRA. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, New York, NY, USA (2015)
13. Cortez, E., Oliveira, D., Silva, A.S., Moura, E.S., Laender, A.H.F: Joint unsupervised structure discovery and information extraction. In: SIGMOD Conference (2011)
14. Elmeleegy, H., Madhavan, J., Halevy, A.Y.: Harvesting relational tables from lists on the web. In: PVLDB (2009)
15. Senellart, P., Mittal, A., Muschick, D., Gilleron, R., Tommasi, M.: Automatic wrapper induction from hidden-web sources with domain knowledge. In: Proceedings of the 10th ACM Workshop on Web Information and Data Management (2008)