Structure Interpretation of Text Formats

SUMIT GULWANI, Microsoft, USA
VU LE, Microsoft, USA
ARJUN RADHAKRISHNA, Microsoft, USA
IVAN RADIČEK, Microsoft, Austria
MOHAMMAD RAZA, Microsoft, USA

Data repositories often consist of text files in a wide variety of standard formats, ad-hoc formats, as well as mixtures of formats where data in one format is embedded into a different format. It is therefore a significant challenge to parse these files into a structured tabular form, which is important to enable any downstream data processing.

We present Unravel, an extensible framework for structure interpretation of ad-hoc formats. Unravel can automatically, with no user input, extract tabular data from a diverse range of standard, ad-hoc and mixed format files. The framework is also easily extensible to add support for previously unseen formats, and also supports interactivity from the user in terms of examples to guide the system when specialized data extraction is desired. Our key insight is to allow arbitrary combination of extraction and parsing techniques through a concept called *partial structures*. Partial structures act as a common language through which the file structure can be shared and refined by different techniques. This makes Unravel more powerful than applying the individual techniques in parallel or sequentially. Further, with this rule-based extensible approach, we introduce the novel notion of *re-interpretation* where the variety of techniques supported by our system can be exploited to improve accuracy while optimizing for particular quality measures or restricted environments. On our benchmark of 617 text files gathered from a variety of sources, Unravel is able to extract the intended table in many more cases compared to state-of-the-art techniques.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → **Artificial intelligence**; • **Information systems** → **Extraction, transformation and loading**.

Additional Key Words and Phrases: program synthesis, data extraction, format diversity

ACM Reference Format:
Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ivan Radiček, and Mohammad Raza. 2020. Structure Interpretation of Text Formats. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 212 (November 2020), 29 pages. https://doi.org/10.1145/3428280

1 INTRODUCTION

The big data revolution has brought about abundance of data in a large variety of ad-hoc formats, which poses the significant challenge of getting this data into a clean and structured form that is amenable for analysis. Unfortunately, existing analytics tools such as spreadsheets, BI tools or data analysis libraries provide very limited automated support for such tasks. Users often need to author specialized one-off scripts for parsing each new format encountered, which requires programming expertise, as well as significant manual effort and time investment. In particular, we note that

Authors' addresses: Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Vu Le, Microsoft, USA, levu@microsoft.com; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Ivan Radiček, Microsoft, Austria, ivradice@microsoft.com; Mohammad Raza, Microsoft, USA, moraza@microsoft.com.

2475-1421/2020/11-ART212
https://doi.org/10.1145/3428280

in this work we are not addressing the problem of free-form data extraction from unstructured or semi-structured texts (which is more the domain of information extraction), but are instead focussing on the vast diversity of arbitrary formats seen in structured text files in which data is commonly stored. This diversity can be observed in the numerous standard document formats (e.g. CSV, JSON, XML, ... and many subtle variants and configurations thereof), arbitrary custom formats (e.g. log files from different systems), as well as mixtures of multiple formats being used in the same document (e.g. a log file or CSV file that may contain some fields in JSON format). Handling such diversity is a common stumbling block for many users, as it requires a specialized parsing strategy for every new variation in format that may be encountered.

In recent years, there has been much interest in the development of *program synthesis* techniques for data extraction, where the goal is to have a system that can intelligently infer the structure in a text file and synthesize a specialised parsing program for every new format encountered [Fisher et al. 2008; Gao et al. 2018; Kandel et al. 2011b; Le and Gulwani 2014; Raman and Hellerstein 2001; Raza and Gulwani 2017a; Zhu et al. 2019a]. While some of these techniques are more generally applicable than others, none of them directly address the problem of handling the diversity of formats encountered in practice. Existing approaches are limited mainly because of the use of specialized algorithms or domain-specific languages (DSLs) that are limited to a certain class of problems, and the gap lies in all of the variations and mixtures of formats that cannot be handled. For instance, given a log file in a mixed format where some fields are in JSON format, an existing synthesizer would attempt to synthesize a program in its own DSL that may fail to robustly capture all the nuances of the JSON format specification without prior knowledge of JSON, while a specialised JSON parser by itself will fail to parse the file at all as it does not fully conform to the specification. In this work, our fundamental philosophy to fill this gap is to embrace the diversity of parsing and synthesis techniques rather than choose any one particular synthesis algorithm or domain-specific language. This is based on the view that the diversity observed in data in practice should be reflected in an intelligent parsing system that attempts to handle such arbitrary data formats. In this spirit, we present a novel framework that permits the combination of arbitrary parsing strategies (which could be specialized parsers, or more general existing or novel synthesizers), and uses a divide-and-conquer approach whereby different sub-problems can be addressed using different parsing strategies in a hierarchical manner. In this respect, our general approach can be seen as a meta-level synthesis framework that permits the combination of different synthesizers to compositionally address different sub-problems, while we also present a concrete system based on this approach that incorporates various specialized synthesis and parsing strategies.

We formulate our approach as a structure inference system that is parameterized by *cooperating domain-specific inference systems*. Each such sub-system is modelled as an *interpretation rule* that can handle a different formatting aspect. Some rules are general and broadly applicable (e.g. splitting a string by a given delimiter), while others can be specific (e.g. parsing a JSON string). To allow these different types of rules to inter-operate and exchange information, we define *partial structures* that act as an interchange mechanism through which the rules exchange and combine formatting and extraction information.

Formally, the structure interpretation framework we develop is an abstract algebra that allows combinations of interpretation rules to extract relational data from files with diverse formats. Semantically, the interpretation rules operate over partial structures that represent an annotation of the given document along with some structural elements. The rule applications refine such a partial structure repeatedly with more annotations until the annotations together identify a single relational table. In addition, we define a partial order on the set of partial structures that intuitively corresponds to the granularity of the structure, and formulate the join and meet operations, which represent different ways of combining the information in partial structures.

However, just defining the interpretation rules and showing how to combine them is not sufficient: A concrete system must also address important algorithmic issues. Firstly, applying the rules in arbitrary combinations will quickly produce an intractable number of partial structures with impractical performance costs, so how do we control this complexity? Secondly, of all the partial structures that can be produced using different rule applications, which one do we pick? We describe concrete algorithms to address these issues, based on a branch-and-bound and local search techniques to only explore useful combinations of rule applications as well as ranking functions over partial structures. We present a concrete instantiation of a system based on this framework and describe its detailed implementation. This system incorporates a wide range of concrete interpretation rules, many of which are extraction systems described in prior work that have been easily incorporated as interpretation rules in our framework.

Our key technical contributions are the definition of the interpretation rule framework and partial structures, the identification of certain generic rules, and a system to efficiently compose rules and pick from various choices for exploration. We show in our evaluation how our system performs better than previous related approaches such as Datamaran [Gao et al. 2018] and PADS [Fisher et al. 2008]. Moreover, our rule-based framework provides a number of important features for usability in practice:

Extensibility. Existing data extraction techniques can be plugged into our framework as rules, and partial structures allow them to work seamlessly with each other, while also sharing information. Due to information sharing, the resulting system can handle mixed formats that the individual techniques cannot handle by themselves or in sequential combination. Extensibility allows us to leverage all the prior work on general automated extraction tools, as well as domain-specific tools for specialized formats.

Interactivity. While our system is able to extract data without user-provided examples, it also allows user-control to guide the structure interpretation by providing examples in a programming-by-example fashion [Cypher et al. 1993]. This allows us to operate in a mixed-mode fashion: to leverage the goodness of by-example systems to handle custom formats or particular user preferences, but without forcing the user to pay for the cost of always providing a large number of examples.

Re-interpretation. We introduce the notion of re-interpretation as the ability to utilise the different strengths of different component rules to optimize inference toward a particular quality measure (say efficiency). The system can first utilize all of the available rules (including less efficient ones) to infer the correct structure with high accuracy. This inferred structure can then be *re-interpreted* using only a subset of the rules mandated by the user. For instance, reading webpages by instantiating the complete document object model (DOM) is computationally expensive but can yield correct structure inference, which can then help to train a purely-text based parsing strategy that can be deployed more efficiently at scale. Re-interpretation can also help to improve the degree of interactivity in specialized systems that are restricted to niche languages: in our evaluation, we show how our rich set of automatic text-based parsing approaches helped to automatically infer specialized programs in the Microsoft Power BI M formula language [Microsoft 2020] in 44% of cases - which would otherwise have to be learned from examples.

In summary, we make the following concrete contributions in this work:

(1) We identify the practical problem of handling mixtures of formats. We describe examples of such scenarios (§3) and show that 50% of cases in our benchmark required multiple rule applications to handle mixed formats (§6).

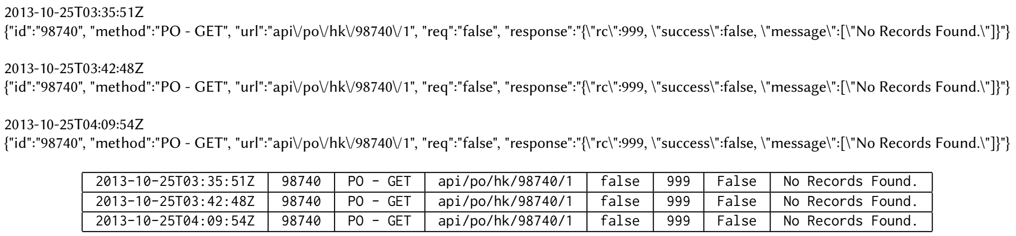
(2) We formulate the framework of structure interpretation to address the diversity of formats, based on multiple cooperating data-extraction logics represented by different interpretation rules operating over partial structures (§ 4). We present a concrete instantiation of the framework with a system that also addresses algorithmic issues to efficiently infer rule applications in the framework (§5). This framework by design allows for easy extensibility as new rules can be added with minimal effort to cover new specialized formats, as we show with some case studies.

(3) We show that the framework supports both automatic inference, as well as interactivity from the user in terms of providing examples (to handle specialized extraction tasks): in 90% of cases it is able to extract the intended table unguided by user-provided examples (although the system returns multiple tables from which the user needs to select the desired result; we discuss this further in §6), while using user-provided examples can address 62% of the remaining cases in our benchmark.

(4) We identify and evaluate the novel notion of re-interpretation that utilizes the variety of rules in the framework to make accurate inferences that can then be optimized towards particular quality measures or restricted environments. We show applications of this technique to improve a specialized by-example system, where 44% of cases can be learnt automatically without examples using re-interpretation.

We begin in the next section by discussing how Unravel works on a concrete example, and then continue describing motivating examples to illustrate the diversity of formats observed in practice and how such scenarios can be handled by our approach. In §4 we present the formal description of the structure interpretation framework of rules over partial structures. In §5, we then describe the concrete algorithms that we have implemented to perform efficient and accurate extractions within the framework. Finally, in §6, we present our experimental evaluation which demonstrates the effectiveness of our approach in handling a diverse set of benchmarks that no single previous approach could handle.

2 MOTIVATING EXAMPLE



```
2013-10-25T03:35:51Z
{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}

2013-10-25T03:42:48Z
{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}

2013-10-25T04:09:54Z
{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}
```

2013-10-25T03:35:51Z	98740	PO - GET	api/po/hk/98740/1	false	999	False	No Records Found.
2013-10-25T03:42:48Z	98740	PO - GET	api/po/hk/98740/1	false	999	False	No Records Found.
2013-10-25T04:09:54Z	98740	PO - GET	api/po/hk/98740/1	false	999	False	No Records Found.

Fig. 1. Log file with multi-line records, where some fields contain nested substructures in JSON format

In this section we discuss in detail how Unravel extracts a table from a concrete input file. Figure 1 shows an input file (top) and an extracted table (below) from this file. This file consists of records that span multiple lines and contains a mixture of hierarchical formats: the first field is a date-time value that occurs on the first line, while the other fields are formatted inside a JSON substructure on the following line, while one of the JSON fields (`"response"`) is itself another JSON fragment.

Intuitively, the high-level workflow of the algorithm is as follows: the algorithm applies rules on partial structures in the pool of interpretations (which initially consists only of the input file) until convergence, then extracts the candidate tables from the pool of interpretations, and finally ranks

2013-10-25T03:35:51Z {"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}
2013-10-25T03:42:48Z {"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}
2013-10-25T04:09:54Z {"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, \"message\":[\"No Records Found.\"]}"}

(a) Result of the step 1.

2013-10-25T03:35:51Z	{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, . . .
2013-10-25T03:42:48Z	{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, . . .
2013-10-25T04:09:54Z	{"id":"98740", "method":"PO - GET", "url":"api\/po\/hk\/98740\/1", "req":"false", "response":"{\"rc\":999, \"success\":false, . . .

(b) Result of the step 2.

98740	PO - GET	api/po/hk/98740/1	false	{"rc":999, "success":false, "message":["No Records Found."]}
98740	PO - GET	api/po/hk/98740/1	false	{"rc":999, "success":false, "message":["No Records Found."]}
98740	PO - GET	api/po/hk/98740/1	false	{"rc":999, "success":false, "message":["No Records Found."]}

(c) Result of the step 3.

999	false	No Records Found.
999	false	No Records Found.
999	false	No Records Found.

(d) Result of the step 4.

Fig. 2. The steps of Unravel extracting data from a file in Figure 1.

the candidate tables. We use the following rules in this example (more detailed descriptions and further rules are given in §4.2):

- RegexSplit[r] that splits a string using a regular-expression r,
- DelimSplit[d] that splits a string using a constant (string) delimiter d, and
- Json that flattens the tree-structure represented by a JSON string.

Below we discuss the sequence of rules (and corresponding partial structures) applied to the file in Figure 1 required to obtain the resulting table; note that Unravel also explored other rule applications and as a result learns many different table extractions (more detailed discussion on the parameter inference is given in §4.2).

Step 1. In the beginning the only partial structure in the pool of interpretations is the whole file, shown in Figure 1 (top). Unravel first applies to the whole file the RegexSplit`[(^|\n)(?:\d)]` rule,[1] which splits the file into records on the lines that start with a number using a regular-expression; the resulting partial structure is shown in Figure 2a.

At this point the algorithm also explores other rule applications, as well as other instantiations for r in the RegexSplit[r] rule, and adds all of them to the pool of interpretations; more precisely, at this point there is no ranking between different rule applications. Moreover, the algorithm recursively continues to explore the resulting partial structures, which is not discussed here. For example, by applying the rule DelimSplit`[\n]`, the algorithm splits the file into records, such that each line is a separate record.

Step 2. Next, Unravel applies the DelimSplit`[\n]` rule to the records obtained in the previous step to split each record into two lines; the resulting structure is shown in Figure 2b.

Same as above, at this point the algorithm also explores other rule applications, as well as other instantiations for d in DelimSplit[d], adds all of them to the pool of interpretations, and continues to recursively explore the resulting partial structures. For example, the algorithm considers d = `{` and d = `:`.

[1] That is, the RegexSplit[r] rule instantiated with r = `(^|\n)(?:\d)`.

Step 3. Next, Unravel applies the Json rule to the last column of the partial structure from the previous step (Figure 2b), that is, it extracts table from the JSON-encoded string; the resulting partial structure is given in Figure 2c (note that the last column of the resulting partial structure, in Figure 2c, is another JSON-encoded string).

At this point the algorithm does not explore other rule applications, because it has *high confidence* that extracting a JSON structure from a JSON-encoded string is a right interpretation of the partial structure (we discuss this in more detail in §5.2).

Step 4. Next, Unravel again applies the Json rule to the last column of the partial structure from the last step (Figure 2c); the resulting partial structure is given in Figure 2d.

At this point, same as in the previous step, the algorithm does not explore other rule applications.

Step 5. Finally, Unravel combines the partial structures discussed in the previous steps and extracts the table in Figure 1 (bottom).

At this step the algorithm stops application of new rules, because no new rule application could be learned for the existing partial structures in the pool of interpretations. For example, the algorithm does not learn new rule applications for the first column in Figure 2d as this column consist of a numeric data-type.

The algorithm then extracts tables from partial structures (discussed in §4.2) and ranks them based on data regularity (discussed in § 5.2). For example, the tables resulting from applying RegexSplit`[(^|\n)(?:\d)]` in *Step 1* are going to be better ranked than the ones resulting from applying DelimSplit`[\n]`, as the algorithm is going to extract more regular data from the former than the latter.

3 A DIVERSITY OF FORMATS AND APPROACHES

```
123456 James Vasanth (00) 123-456 (00) 789-101 JamesBvasanth@mail.com Tech lead "NO 01, 23rd street, Pune"
5454582 Mary Maxwell (022) 3050-958 (021) 5661-070 maryAMaxwell@mail.com Professors "166 Abc Place Soway Master Taramani 510"
```

123456	James Vasanth	(00) 123-456	(00) 789-101	JamesBvasanth@mail.com	Tech lead	"NO 01, 23rd street, Pune"
5454582	Mary Maxwell	(022) 3050-958	(021) 5661-070	maryAMaxwell@mail.com	Professors	"166 Abc Place Soway Master Taramani 510"

```
12apple300
23grapes black500
100strawberry12300
```

12	apple	300
23	graps black	500
100	strawberry	12300

Fig. 3. Data formatted with contexual delimiters (above) and data formatted without any delimiters (below)

```
msgid ""
msgstr ""
"Project-Id-Version: Lingohub 1.0.1\n"
"Report-Msgid-Bugs-To: support@lingohub.com \n"

msgid "Let’s make the web multilingual."
msgstr "Machen wir das Internet mehrsprachig."

msgid "We connect developers and translators around the globe "
"on Lingohub for a fantastic localization experience."
msgstr "Wir verbinden Entwickler mit Übersetzern weltweit "
"auf Lingohub für ein fantastisches Lokalisierungs-Erlebnis."

msgid "%d page read."
msgid_plural "%d pages read."
msgstr[0] "Eine Seite gelesen wurde."
msgstr[1] "%d Seiten gelesen wurden."
```

Fig. 4. Translation file - requires record boundaries and key-value reasoning that spans over multiple lines.

In this section we illustrate in more detail the diversity of formats found in real-world scenarios, the range of existing techniques with different strengths and limitations, and how our approach

msgid	msgid_plural	msgstr	msgstr[0]	msgstr[1]
		Project-Id-Version: Lingohub 1.0.1\n Report-Msgid-Bugs-To: support@lingohub.com \n		
Let's make the web multilingual.		Machen wir das Internet mehrsprachig.		
We connect developers and translators around the globe on Lingohub for a fantastic localization experience.		Wir verbinden Entwickler mit Übersetzern weltweit auf Lingohub für ein fantastisches Lokalisierungs-Erlebnis.		
%d page read.	%d pages read.		Eine Seite gelesen wurde.	%d Seiten gelesen wurden.

Fig. 5. Translation example - the extracted table.

can address such scenarios. The goal of any intelligent parsing system is to be able to infer the structure that is inherent in a block of raw text-based data. Different approaches model or interpret the structure of data in specific ways and adopt specialized inference strategies based on these interpretations. For instance, some techniques take a *top-down* approach, where inference proceeds by first finding plausible record boundaries and then determining different fields within these records by matching common patterns between inferred records [Fisher et al. 2008; Le and Gulwani 2014]. Such approaches can require significant interaction by the user usually in the form of examples or specification of boundaries, and are also restricted by expressiveness of the domain-specific extraction languages that they work with. Other techniques have taken a *bottom-up* approach, where prominent fields or delimiters are inferred independently, and then alignment patterns between fields are inferred in order to determine record boundaries [Raza and Gulwani 2017a]. Such methods rely on there being strong alignment between fields, and do not work well when there is much variation in the records, e.g. Raza and Gulwani [2017a] cannot handle multi-line records or cases where there is significant missing data in records.

Template-based approaches assume the input data contains a fixed pattern of records (such as a regular expression template that contains blanks for field values), and attempt to directly infer entire templates that would capture records and fields at the same time [Gao et al. 2018]. These approaches do not work when there are no well-defined templates in the raw data, such as in Figure 3, where inference depends on patterns within the fields rather than the surrounding text. Some approaches favour less automated intelligence and rely more on interactivity from the user, such as specifying transformation steps in a visual UI [Kandel et al. 2011b; Raman and Hellerstein 2001]. In addition to these generic parsing approaches that have been presented in the literature, one can also view systems designed for each of the standard file formats and their individual dialects as specialized interpretations of data. Such systems also commonly require significant effort to be used correctly: e.g. the Python Pandas library's `read_csv` function has around 50 parameters that the user can specify to cater for different scenarios. Thus in general, we observe a wide range of techniques that differ over a range of different dimensions of quality, such as generality, level of interactivity, readability of inferred extraction logics, and efficiency of extraction.

Standard formats. There is a large number of standard document formats, including very popular ones such as CSV (comma-separated or delimited files), fixed-width files, XML, JSON, HTML, and numerous other less common ones. Though these may often be called "standard", there is often much subtlety and variation in correctly parsing files in such formats depending on different implementations. CSV is a very common format, but is often found in different dialects that require different parsing strategies. For instance, different delimiting characters may be used such as comma, space, semicolon or other arbitrary characters. Different quotation rules may be adopted to distinguish quote characters that occur inside text fields, such as escaping quotes using a backslash or with two consecutive quotes. There may be other subtle variations, such as how newline characters are handled inside text fields, or how blank lines are interpreted. Unlike delimited files, fixed-width files represent data fields at fixed character positions in a line. Though usually distinguished by padding whitespace characters, in many cases such fields may occur contiguously and therefore have ambiguous boundaries that need to be intelligently inferred. For richer file formats, the actual data of interest may occur within arbitrary patterns and in the presence of significant noise. For instance, webpages in HTML very often contain tabular information that does not exist in explicit HTML table or list tags, but formatted within arbitrary tag and class structures on the webpage, e.g. listings presented as tiles or banners in a page such as the list of movies shown in Figure 6. In general, noise in the data and subtle variations of format are common challenges encountered with many standard formats in practice.

In our framework, one may represent the range of different parsing strategies of standard formats as different interpretation rules. For example, different CSV parsing strategies, intelligent fixed-width inference strategies and automated web table extraction approaches can all be encoded as different interpretation rules. While this handling of standard formats is technically a simple application of existing techniques optimized for specific formats, it represents the base case of our system where we can permit a wide range of specialised rules to handle specialised cases well.

Custom formats. Often times, text-based data may come in purely custom or ad-hoc formats depending on the source. Figure 3 shows an example of data from a question in an Excel help forum[2], which shows the need for *contextual delimiters*: characters that may be used as delimiters in some regions of the data but can also be part of the data fields in others. In this case, the user needs to split the different fields using whitespace character as the delimiter, but whitespace also appears multiple times in some of the fields that contain multiple words, and hence not all occurrences of spaces can be treated as delimiters. In some other cases, there may not be any delimiting characters that separate different fields in the data. Figure 3 also shows such an example from another help forum question[3], where the data consists of numeric and non-numeric fields with no delimiting characters between them. While contextual and zero-length delimiters can be handled by a system such as Raza and Gulwani [2017a], that system must be given record boundaries and cannot infer these automatically. But in our approach, we have a range of rules to determine record boundaries, including skipping header lines and avoiding noisy lines in the data. Hence, in these cases, our system can infer a composition of two rules: first to split the initial file into records and then apply the text splitting rule based on Raza and Gulwani [2017a] to split the records into fields.

As another example, consider the fragment of the translation file shown in Figure 4 and the desired result in Figure 5. In this case the desired extraction requires a reasoning based on key-value pairs. Our system uses a combination of rules to automatically extract this table: (1) Split file into records, using the empty line as a record separator. (2) Each of the 5 columns is obtained by extracting a key-value pair, where values can span multiple consecutive lines (the keys are: `msgid`, `msgid_plural`, `msgstr`, `msgstr[0], msgstr[1]`).

Mixture of formats. Interestingly, in many cases we find that data is formatted using a mix of different syntactic constructs in the same file. For instance, log files generated by various systems often generate text that may include some fragments of data in existing formats. Figure 1 shows such a log file, which we have already discussed in §2. As discussed in §2, our system interprets the structure of this file through an inference of four distinct interpretation rules. On the other hand, standard JSON parsers cannot be applied to such files without pre-processing and extracting the relevant text fragments. Existing techniques, such as Fisher et al. [2008]; Gao et al. [2018]; Le and Gulwani [2014]; Raza and Gulwani [2017a], do not incorporate JSON understanding explicitly

[2] http://www.mrexcel.com/forum/excel-questions/991875-formula-split-words-cell.html
[3] http://www.mrexcel.com/forum/excel-questions/986709-split-text.html

in their design, so even if they could handle simple cases where a delimiter-based splitting may suffice, these techniques will be brittle and may easily fail with small variations in the JSON such as missing or differently ordered fields.

Similarly, in practice we find cases of CSV files where some fields are formatted as JSON or XML formats. While standard CSV parsers may apply on such files, they will not extract the fields embedded inside such substructures, for which a combination of parsers would be required. Our system can handle such cases by a combination of a standard CSV parser together with a standard JSON parser.

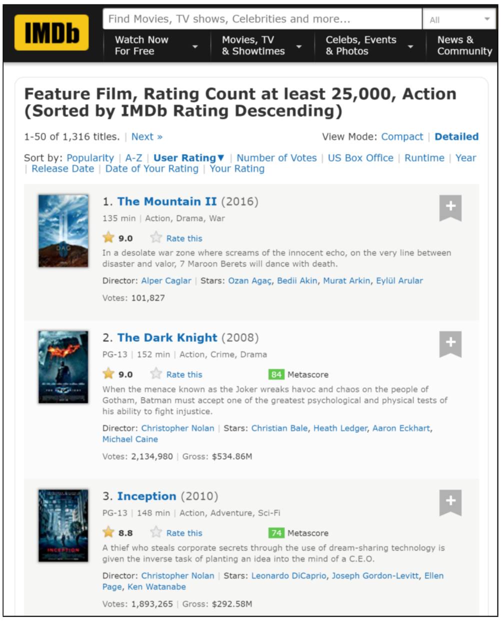
Another common scenario is when data fields require additional text extraction operations after the initial extraction. For instance, webpages in HTML format often contain plain text formatted in particular patterns, so that simply extracting the text content of nodes is not sufficient and one needs to perform further splitting or substring extraction. Figure 6 shows such a scenario of table extraction from a webpage containing information about movies, which is described in a YouTube video showing how to perform such extraction in the Power BI analytics tool [4]. In this case the user first performs extraction of data from the webpage using an HTML-based wrapper induction tool to extract text content of DOM nodes, and must then write a text substring extraction program to obtain the numeric values from the movie runtimes column (e.g. `"135"` from `"135 min"`). Such additional text processing is a common requirement in data ingestion scenarios. In this case, our system can apply a composition of web extraction and text extraction rules to obtain the final table. Firstly, it applies the automatic web table inference rule that uses the technique of Raza and Gulwani [2017a] to instantiate a webpage DOM and automatically infers DOM node selectors to obtain the initial table. This table contains the full text content of nodes, and our system subsequently applies text-splitting rules such as a delimiter-based splitting or the context-based splitting of Raza and Gulwani [2017a] to obtain the desired numeric values.

This scenario also illustrates the benefits of allowing examples-based interaction as this kind of text manipulation is more of a user preference: different users may want to extract different information from the columns (e.g. extract `"534.86"` from `"$534.86M"`, remove the parentheses around the movie years, or split movie genres `"Action, Drama, War"` into separate columns, etc). To indicate such preferences, users can provide examples of the desired output columns in a programming-by-example fashion and our system can infer the appropriate rule applications to adapt the level of extraction to satisfy the given examples.

This example also illustrates our notion of *re-interpretation*: the ability to utilise the different strengths of component systems to infer an extraction approach optimized toward a particular quality measure such as efficiency or readability. This scenario was inspired by a product team that was aiming to build a knowledge base from web data, which required applying such automatic extraction at large scale on thousands of webpages. The scale of the problem meant that they could not use a technique that instantiates the whole webpage DOM on every execution as this will be too costly. In such cases, our approach can use a DOM-based extraction rule to infer the correct extraction with high accuracy, but then re-interpret that extraction in a second phase to infer a purely text-based extraction program if one is expressible in the text parsing rules, and this more efficient program may then be applied for deployment at scale.

As another example scenario, often times sophisticated regular-expression based approaches infer extractions with higher accuracy as they may better describe the data, but the inferred structures can often be re-interpreted using simpler text manipulation operators such as simple substring operations based on position indexes. This is especially important when the user is working in restricted language environments where expressive operators such as regexes may not be supported.

[4] https://www.youtube.com/watch?v=HvPh2go8xJs



Feature Film, Rating Count at least 25,000, Action (Sorted by IMDb Rating Descending)

1-50 of 1,316 titles. | Next »   View Mode: Compact | Detailed

Sort by: Popularity | A-Z | User Rating▼ | Number of Votes | US Box Office | Runtime | Year | Release Date | Date of Your Rating | Your Rating

1. The Mountain II (2016)
135 min | Action, Drama, War
9.0 Rate this
In a desolate war zone where screams of the innocent echo, on the very line between disaster and valor, 7 Maroon Berets will dance with death.
Director: Alper Caglar | Stars: Ozan Agaç, Bedii Akin, Murat Arkin, Eylül Arular
Votes: 101,827

2. The Dark Knight (2008)
PG-13 | 152 min | Action, Crime, Drama
9.0 Rate this 84 Metascore
When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, Batman must accept one of the greatest psychological and physical tests of his ability to fight injustice.
Director: Christopher Nolan | Stars: Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine
Votes: 2,134,980 | Gross: $534.86M

3. Inception (2010)
PG-13 | 148 min | Action, Adventure, Sci-Fi
8.8 Rate this 74 Metascore
A thief who steals corporate secrets through the use of dream-sharing technology is given the inverse task of planting an idea into the mind of a C.E.O.
Director: Christopher Nolan | Stars: Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Ken Watanabe
Votes: 1,893,265 | Gross: $292.58M

The Mountain II	In a desolate war zone ...			135
The Dark Knight	When the menace known ...	$534.86M	84	152
Inception	A thief who steals ...	$292.58M	74	148

Fig. 6. IMDB webpage with tabular data not represented in HTML tables

In our evaluation, we explored one such scenario of the restricted M language for data manipulation in the Microsoft Power BI product. Though a by-example system exists for inferring M programs, we show how using re-interpretation we can fully automatically infer programs in the M language in 44% of cases without any examples.

4 A STRUCTURE INTERPRETATION FRAMEWORK

We describe our structure interpretation framework.

Text documents and extractions. We use the notation $\mathbb{T}$ to denote a text document and the term *extracted value* for a single atomic value that has been extracted from $\mathbb{T}$. Formally, an *extracted value* $\mathcal{V}$ is a substring $\mathbb{T}[i, j]$ of $\mathbb{T}$ between the indices i (inclusive) and j (exclusive). We say: (a) $\mathbb{T}[i, j] \leq \mathbb{T}[k, \ell]$ if $k \leq i \leq j \leq \ell$, i.e., if $\mathbb{T}[i, j]$ is contained in $\mathbb{T}[k, \ell]$, and (b) $\mathbb{T}[i, j] \circ \mathbb{T}[k, \ell] = \mathbb{T}[i, \ell]$, i.e., the smallest extracted value containing both $\mathbb{T}[i, j]$ and $\mathbb{T}[k, \ell]$.

The problem statement. Given an arbitrary text document $\mathbb{T}$ the *ad-hoc structure interpretation problem* is to extract a relational table R. We use the notation $R(m, n)$ for the n^{th} attribute (column) of the m^{th} tuple (row) of R. We require the following *well-formedness constraints* to hold on R:

(a) Each cell is an extracted value: $\exists i, j. R(m, n) = \mathbb{T}[i, j]$.
(b) The extracted values in R do not overlap, unless they are equal: any two $\mathbb{T}[i, j]$ and $\mathbb{T}[i', j']$ occurring in R, we have $(i = i' \wedge j = j') \vee j \leq i' \vee j' \leq i$.
(c) Extracted values are not shared across attributes: $R(m, n) = R(m', n') = \mathbb{T}[i, j] \implies n = n'$.

The above requirements allow for a large number of possible tables to be extracted from the input. Hence, we additionally require that solution produces the best table as per an *optimality criterion*. Our optimality criterion is based on the weighted sum of two metrics:

- *Coverage.* What fraction of the characters from the text file appear in the table? Formally, it is defined as $|\{k | \exists m, n, i, j : R(m, n) = \mathbb{T}[i, j] \wedge i \leq k < j\}| / |\mathbb{T}|$.
- *Description length.* This measure is the one used in the seminal PADS framework [Fisher et al. 2008]. In our context, it is defined as the number of bits required to transmit the table given the type signature of each column. Informally, if a column has a known type (say integers or dates), it can be transmitted with fewer bits; similarly, if a column contains categorical values or strings with a fixed pattern, fewer bits are required (see Fisher et al. [2008] for a full definition). Note that computing the exact value of this metric is essentially undecidable: in the general case, it is equivalent to computing Kolmogorov complexity. However, in §5, we use a proxy for this metric and explain how the proxy correlates to the description length.

The exact weights can be picked by the user based on the domain. The optimality criteria are a proxy for user intent, i.e., we assume that the user prefers to extract more data from a file as compared to less, and prefers "regular" data over irregular data.

Example 4.1. The condition (c) in the problem definition allows for the same $\mathbb{T}[i, j]$ to be shared across different rows in the same column. From our investigation of ad-hoc text documents, this is rather common. In the document below (left) and its corresponding table (right), the state codes (PA and CA) are shared across different rows.

```
PA
Allegheny,15120,Pittsburgh
Delaware,19063,Media
CA
Los Angeles,90210,Hollywood
Los Angeles,90211,Malibu
```

⟶

PA	Allegheny	15120	Pittsburgh
PA	Delaware	19063	Media
CA	Los Angeles	90210	Hollywood
CA	Los Angeles	90211	Malibu

4.1 Partial Structures as Interpretations

The main object of interest in our extraction procedures is a *partial structure*. Informally, a partial structure is an annotation of $\mathbb{T}$ with a number of labels known as *syntactic elements*.

Syntactic Elements. A *syntactic element*, denoted se, is an ordered collection $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_n$ of extracted values such that: (a) $\mathcal{V}_i$ and $\mathcal{V}_j$ do not overlap for $i \neq j$, and (b) each $\mathcal{V}_i$ occurs before $\mathcal{V}_j$ for $i < j$. Intuitively, a syntactic element is the result of one parsing step, i.e., a collection of extracted values that are obtained in the same manner.

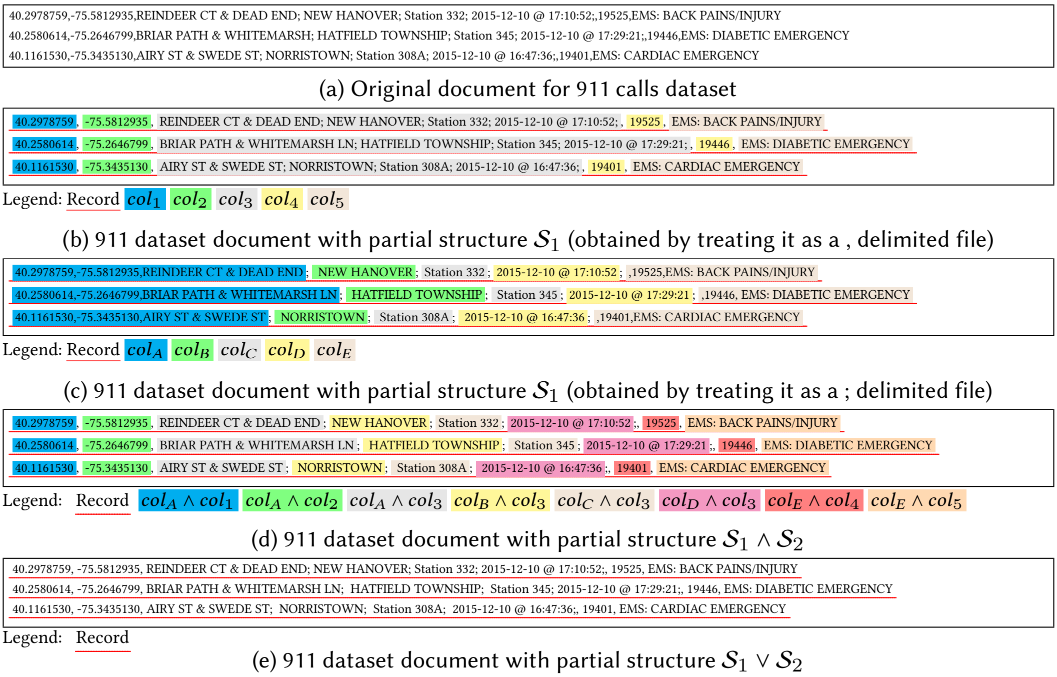
In relation to extracted tables, a single syntactic element commonly represents (a) records (e.g., Record in Figure 7b), (b) a single column (e.g., col_1 in Figure 7b), or (c) merging of several columns (e.g., col_3 in Figure 7b; here, the columns of the final extracted table that are merged are $\mathsf{col}_3 \wedge \mathsf{col}_A, \ldots, \mathsf{col}_3 \wedge \mathsf{col}_D$ in Figure 7d). We explain in Section 4.2 how to go from a collection of such varied syntactic elements to a single table.

Let $\mathsf{se}_1 = \mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_m$ and $\mathsf{se}_2 = \mathcal{V}'_0, \mathcal{V}'_1, \ldots, \mathcal{V}'_n$ be two syntactic elements. We define:

- se_1 is *contained in* se_2 (written as $\mathsf{se}_1 \leq \mathsf{se}_2$) if every value $\mathcal{V}_i \in \mathsf{se}_1$ is contained in some value $\mathcal{V}'_j \in \mathsf{se}_2$.
- *Composition* of se_1 and se_2 (written as $\mathsf{se}_1 \circ \mathsf{se}_2$) as the syntactic element $\mathcal{V}_0 \circ \mathcal{V}'_0, \ldots, \mathcal{V}_m \circ \mathcal{V}_n$ if $m = n$.
- The *meet* $\mathsf{se}_1 \wedge \mathsf{se}_2$ is defined when $m = n$ and each $\mathcal{V}_i, \mathcal{V}'_i$ pair overlap: the meet is $\mathcal{V}_0 \wedge \mathcal{V}'_0, \ldots, \mathcal{V}_n \wedge \mathcal{V}'_n$ where $\mathbb{T}[i, j] \wedge \mathbb{T}[k, \ell] = \mathbb{T}[\max(i, k), \min(j, \ell)]$.

Example 4.2. In Figure 7b-7d, the collection of sub-strings labelled `Record`, $\texttt{col}_\texttt{i}$ (for $1 \leq i \leq 5$), and $\texttt{col}_\texttt{x}$ (for $x \in A, \ldots, E$) are valid syntactic elements. We have that $\texttt{col}_1 \circ \ldots \texttt{col}_5 = \texttt{col}_\texttt{A} \circ \ldots \texttt{col}_\texttt{E} = \texttt{Record}$, and that $\texttt{Record} \wedge \texttt{col}_\texttt{i} = \texttt{col}_\texttt{i}$. Figure 7d illustrates additional $\wedge$ operations.

Partial Structures. A *partial structure* $\mathcal{S}$ is a set of syntactic elements $\{\mathsf{se}_0, \ldots, \mathsf{se}_n\}$ where for all se_i and se_j: either se_i and se_j are non-overlapping, or $\mathsf{se}_i \leq \mathsf{se}_j \vee \mathsf{se}_j \leq \mathsf{se}_i$ holds. We use the symbol



40.2978759,-75.5812935,REINDEER CT & DEAD END; NEW HANOVER; Station 332; 2015-12-10 @ 17:10:52;,19525,EMS: BACK PAINS/INJURY
40.2580614,-75.2646799,BRIAR PATH & WHITEMARSH; HATFIELD TOWNSHIP; Station 345; 2015-12-10 @ 17:29:21;,19446,EMS: DIABETIC EMERGENCY
40.1161530,-75.3435130,AIRY ST & SWEDE ST; NORRISTOWN; Station 308A; 2015-12-10 @ 16:47:36;,19401,EMS: CARDIAC EMERGENCY

(a) Original document for 911 calls dataset

40.2978759, -75.5812935, REINDEER CT & DEAD END; NEW HANOVER; Station 332; 2015-12-10 @ 17:10:52; , 19525, EMS: BACK PAINS/INJURY
40.2580614, -75.2646799, BRIAR PATH & WHITEMARSH LN; HATFIELD TOWNSHIP; Station 345; 2015-12-10 @ 17:29:21; , 19446, EMS: DIABETIC EMERGENCY
40.1161530, -75.3435130, AIRY ST & SWEDE ST; NORRISTOWN; Station 308A; 2015-12-10 @ 16:47:36; , 19401, EMS: CARDIAC EMERGENCY

Legend: Record col_1 col_2 col_3 col_4 col_5

(b) 911 dataset document with partial structure $\mathcal{S}_1$ (obtained by treating it as a , delimited file)

40.2978759,-75.5812935,REINDEER CT & DEAD END; NEW HANOVER; Station 332; 2015-12-10 @ 17:10:52; ,19525,EMS: BACK PAINS/INJURY
40.2580614,-75.2646799,BRIAR PATH & WHITEMARSH LN; HATFIELD TOWNSHIP; Station 345; 2015-12-10 @ 17:29:21; ,19446, EMS: DIABETIC EMERGENCY
40.1161530,-75.3435130,AIRY ST & SWEDE ST; NORRISTOWN; Station 308A; 2015-12-10 @ 16:47:36; ,19401,EMS: CARDIAC EMERGENCY

Legend: Record col_A col_B col_C col_D col_E

(c) 911 dataset document with partial structure $\mathcal{S}_1$ (obtained by treating it as a ; delimited file)

40.2978759, -75.5812935, REINDEER CT & DEAD END ; NEW HANOVER ; Station 332 ; 2015-12-10 @ 17:10:52 ;, 19525, EMS: BACK PAINS/INJURY
40.2580614, -75.2646799, BRIAR PATH & WHITEMARSH LN ; HATFIELD TOWNSHIP ; Station 345 ; 2015-12-10 @ 17:29:21 ;, 19446, EMS: DIABETIC EMERGENCY
40.1161530, -75.3435130, AIRY ST & SWEDE ST ; NORRISTOWN ; Station 308A ; 2015-12-10 @ 16:47:36 ;, 19401, EMS: CARDIAC EMERGENCY

Legend: Record $col_A \wedge col_1$ $col_A \wedge col_2$ $col_A \wedge col_3$ $col_B \wedge col_3$ $col_C \wedge col_3$ $col_D \wedge col_3$ $col_E \wedge col_4$ $col_E \wedge col_5$

(d) 911 dataset document with partial structure $\mathcal{S}_1 \wedge \mathcal{S}_2$

40.2978759, -75.5812935, REINDEER CT & DEAD END; NEW HANOVER; Station 332; 2015-12-10 @ 17:10:52;, 19525, EMS: BACK PAINS/INJURY
40.2580614, -75.2646799, BRIAR PATH & WHITEMARSH LN; HATFIELD TOWNSHIP; Station 345; 2015-12-10 @ 17:29:21;, 19446, EMS: DIABETIC EMERGENCY
40.1161530, -75.3435130, AIRY ST & SWEDE ST; NORRISTOWN; Station 308A; 2015-12-10 @ 16:47:36;, 19401, EMS: CARDIAC EMERGENCY

Legend: Record

(e) 911 dataset document with partial structure $\mathcal{S}_1 \vee \mathcal{S}_2$

Fig. 7. 911 dataset document with multiple partial structures

$\mathbb{S}$ to denote the set of all partial structures. Intuitively, each partial structure is an "interpretation" of the text document.

We say that $\mathcal{S}_1$ *refines* $\mathcal{S}_2$ (written as $\mathcal{S}_1 \leq \mathcal{S}_2$) if for every syntactic element $\mathsf{se} \in \mathcal{S}_2$ there exists a sequence of elements $\mathsf{se}_1, \mathsf{se}_2, \ldots, \mathsf{se}_k$ in $\mathcal{S}_1$ such that $\mathsf{se}_1 \circ \mathsf{se}_2 \circ \ldots \mathsf{se}_k = \mathsf{se}$. Intuitively, $\leq$ defines a partial order on $\mathbb{S}$ that corresponds to the granularity of the interpretation: every element of the $\mathcal{S}_2$ interpretation can be obtained by combining $\mathcal{S}_1$ elements.

In the poset $\langle \mathbb{S}, \leq \rangle$, we can define meet and join operations in the standard way. Formally, given $\mathcal{S}_1$ and $\mathcal{S}_2$, the *meet* $\mathcal{S}_1 \wedge \mathcal{S}_2$ satisfies the following: (a) $(\mathcal{S}_1 \wedge \mathcal{S}_2) \leq \mathcal{S}_1$ and $(\mathcal{S}_2 \wedge \mathcal{S}_2) \leq \mathcal{S}_1$, and (b) for all $\mathcal{S}$ such that $\mathcal{S} \leq \mathcal{S}_1$ and $\mathcal{S} \leq \mathcal{S}_2$, we have that $\mathcal{S} \leq (\mathcal{S}_1 \wedge \mathcal{S}_2)$ Similarly, we can define the dual operation join $\mathcal{S}_1 \vee \mathcal{S}_2$. The poset of partial structures is closely related to the interval domain used in program analysis [Cousot and Cousot 1977]. Intuitively, each extracted value is an interval and syntactic elements and partial structures are collections of intervals. However, note that $\langle \mathbb{S}, \leq \rangle$ is not a lattice, i.e., unique meets and joins may not exist for each $\mathcal{S}_1$ and $\mathcal{S}_2$.

It is easy to produce effective definitions (i.e., a finite procedure) of the operations over $\mathbb{S}$ using component-wise operations $\leq$, $\wedge$ and $\circ$ over syntactic elements. However, we elide these procedures for the sake of space.

Example 4.3. Figure 7 illustrates the concept of meet and join of partial structures. As can be seen, $\mathcal{S}_1 \wedge \mathcal{S}_2$ imposes the interpretation of both $\mathcal{S}_1$ and $\mathcal{S}_2$ on to the document, while $\mathcal{S}_1 \vee \mathcal{S}_2$ imposes the maximal structure common to both the partial structures.

4.2 Structure Interpretation and Extraction

We are now ready to explain the core mechanics of UNRAVEL.

Examples. While the Unravel system can predictively extract tables from text files, it can also be guided by user provided examples. Formally, a set of *examples* E consists of constraints of the form $(m, n) \mapsto \mathcal{V}$; each $(m, n) \mapsto \mathcal{V}$ constrains the n^{th} column of the m^{th} row to have the extracted value $\mathcal{V}$. This definition covers most forms of examples that are used in automated data extraction and transformation literature (for example, see [Le and Gulwani 2014; Martins et al. 2019; Wang et al. 2019]).

We say that: (a) partial structure $\mathcal{S}$ is *consistent* with the examples E if it does not "split" the file in the middle of an example. Formally, for all $(m, n) \mapsto \mathcal{V} \in E$ (where $\mathcal{V} = \mathbb{T}[i, j]$), no extracted value $\mathbb{T}[k, \ell]$ in any syntactic element of $\mathcal{S}$ has $i < k < j$ or $i < \ell < j$. (b) relational table R *satisfies* E if for all $(m, n) \mapsto \mathcal{V} \in E$, we have $R(m, n) = \mathcal{V}$.

Interpretation rules. In practice, a developer of the Unravel system would specify *interpretation rules* that operate on syntactic elements to interpret documents. Formally, an *interpretation rule* $\mathcal{R}$ is a (partial) function from $\mathsf{SE}^p \not\mapsto \mathsf{SE}^q$, that takes as input a p-tuple of syntactic elements and optionally produces a q-tuple of syntactic elements, for some $p, q \in \mathbb{N}$. Given a rule $\mathcal{R}$ and a partial structure $\mathcal{S}$, we say that applying $\mathcal{R}$ on $\mathcal{S}$ produces $\mathcal{S}' \leq \mathcal{S}$ (written as $\mathcal{S} \rightarrow_{\mathcal{R}} \mathcal{S}'$) if: $\exists \mathsf{se}_1, \dots, \mathsf{se}_p \in \mathcal{S}. \mathcal{R}(\mathsf{se}_1, \dots, \mathsf{se}_p) = (\mathsf{se}'_1, \dots, \mathsf{se}'_q)$ and $\mathcal{S}' = \mathcal{S} \cup \{\mathsf{se}'_1, \dots, \mathsf{se}'_q\}$.

We first discuss on how interpretation rules can be built from simpler functions of some specific type signatures. The type signatures we consider correspond to "string splitting" (or "file to records"), "records to table", and "file to table" operations. The reason we consider these specific types of functions is that most existing extraction systems operate in one of these modes, thereby letting us convert other existing extraction systems to rules.

- *String splitting*: Given a function $f : \mathsf{str} \rightarrow \mathsf{str}^n$ that splits a string into n different fragments, we can construct two rules that both take a single syntactic element se as input. The first rule (called flat-map version) applies f on each value in se and combines all the output fragments into one output element. The second rule (called map version) applies f on each value on se and returns n elements $\mathsf{se}_0, \dots, \mathsf{se}_n$ where each se_i contains the i^{th} component of the f output on the values of se.
- *Records to Table*: Given a function $f : \mathsf{str}^* \rightarrow \mathsf{str}^{*n}$ that maps a list of strings (records) into n different lists of strings (columns), we can generate the rule that simply applies f on a single syntactic element and produces n different elements.
- *File to Table*: Given a function $f : \mathsf{str} \rightarrow (\mathsf{str}^m)^n$ that maps a string to a table of m rows and n columns, we can flat-map f on a syntactic element to produce n different syntactic elements. Each output syntactic element corresponds to the combined values in one column of all produced tables.

Converting all these three types of functions into a single formalism (interpretation rules) lets us apply any of them at any depth, i.e., the file to records rule need not be applied on the full file, but may be applied at a later stage.

We list a *subset* of rules that are used in the implementation of the Unravel system; for each rule we also describe how are the parameters inferred.

- DelimSplit$[d]$: These pair of rules (flat-map and map versions) split the input string into substrings separated by a constant string d.
  To infer the parameter value d instantiations we (a) pick from a list of common delimiters (e.g., `,`, `;`, `|`), and (b) identify common substrings that appear in the strings of se ; then we filter-out ones that are not producing the same number of splits across different strings. [5]

[5] The algorithm actually allows some noise, that is, that on some small number of rows there are different number of splits. We discuss handling of noise in more detail below.

For example, in Figure 7b and Figure 7c, the elements $col_A, \ldots, col_E$ and $col_1, \ldots, col_5$ result from applying DelimSplit[`,`] and DelimSplit[`;`] respectively, on Record. Similarly, Figure 2b is obtained from Figure 2a by applying DelimSplit[`\n`]. In this example the algorithm also infers $d =$ `{` and $d =$ `:`. All of the instantiations mentioned here are picked because they are common delimiters or occur regularly in strings, and produce the same number of splits across different strings.

- RegexSplit[$r_1, \ldots, r_n$]: This pair of rules (flat-map and map versions) split the input string at the matches of r_i in sequence.
  To infer the instantiations for parameters $r_1, \ldots, r_n$ the algorithm uses bottom-up predictive synthesis a la Raza and Gulwani [2017a]. Informally, the algorithm generates a large number of regular expressions that match some part of the input string and then selects sequences that produce consistent splits.
  For example, as discussed in § 2, Figure 2a is obtained from Figure 1 (top) by applying the rule RegexSplit[`(^|\n)(?:\d)`].
- FixedWidth[$\ell_1, \ldots, \ell_n$]: This rule is a record to table style rule, that splits records of equal length into n consecutive columns that have lengths $\ell_1, \ldots, \ell_n$.
  To infer the instantiations for parameters $\ell_1, \ldots, \ell_n$ the algorithm analyses the input strings and finds "natural boundaries" between tokens; a position is considered a natural boundary if across all input strings: (a) it is a whitespace followed by a non-whitespace; (b) separates two data-types.
- Tree flattening rules: The rules Json, Xml, and Web work in string to table mode. Each of them parses a string into a tree structure (JSON object, XML tree, and DOM tree, respectively), and then uses appropriate selectors (jq, xpath, and css selector respectively) to produce multiple columns.
  The process of choosing the right selectors to extract interesting data from these trees is an active and on-going area in research (see Arasu and Garcia-Molina [2003]; Crescenzi et al. [2001]; Gulhane et al. [2011]; Kushmerick [1997]; Nielandt et al. [2016]; Raza and Gulwani [2017b]), as well as in software library development (see Cognos Analytics [2019]; Data Miner [2019]; Json Normalize [2019]).
  These rules can do both one-shot parsing of the appropriate format and parse JSON/XML/HTML strings that are embedded in other formats once they are extracted as a single syntactic element (see, for example, Figure 1 and in general discussion in § 2).
- KeyValue[d_p, d_k]: This rule is based on a records to table function: the function treats each record as a list of key-value pairs, and each column in the table corresponds to the value of a single key. The rule is parametrized by a pair delimiter d_p and key delimiter d_k where different key-value pairs in each $\mathcal{V}_i$ are separated by d_p, and the key and value in each pair is separated by d_k.
  To infer the instantiations for parameters d_p and d_k the algorithm iterates the predefined list of possible parameter values and chooses the ones that produce consistent outputs. The algorithm allows that keys are in different order across different records and also allows that some keys are missing in some rows. See discussion around noise and missing fields in § 6.
  For example, in the file in Figure 5, the pair delimiter is a newline, and the key delimiter is a space (and there are also missing fields).
- Skip[k] and SkipLast[k]: These rules do not correspond to any of the three forms listed above. Instead, they take a syntactic element $\mathsf{se} = \mathcal{V}_0, \ldots, \mathcal{V}_n$ and produces the syntactic element without the first (resp. last) k extracted values $\mathcal{V}_i$. These rules are usually used in Unravel to skip headers and footers in files or columns.

To infer the parameter k we analyse various features of $\mathcal{V}_0, \ldots, \mathcal{V}_n$ (e.g., the matched regular expression) and check whether some prefix (resp. suffix) has a distingushing set of features that the rest of the syntactic elements does not have.
For example, if $\mathcal{V}_5, \ldots, \mathcal{V}_n$ are all strings that represent dates in a particular format, but $\mathcal{V}_0, \ldots, \mathcal{V}_4$ are not, we can infer $k = 5$ for the Skip rule.

- PatternFilter$[\phi_1, \ldots, \phi_n]$: This rule partitions a syntactic element se into a number of disjoint syntactic elements se_i such that the union of all se_i is se. Each se_i is defined by predicate parameter ϕ_i: se_i consists of those strings of se that satisfy ϕ_i.
  The predicates $\phi_1, \ldots, \phi_n$ are inferred using simple techniques such as finding common prefixes/suffixes to produce StartsWith/EndsWith predicates, as well as more complex techniques for regular expression learning [Padhi et al. 2018] to produce RegexMatch predicates.
- Black-box rules: We have also included a number of rules that correspond to other published data-extraction techniques (for example, FlashExtract [Le and Gulwani 2014] and ColumnSplit [Raza and Gulwani 2017a]).
  The black-box rules illustrate the "glueing" power of the partial structure and syntactic element framework: these techniques work with each other and with other native rules seamlessly. Further, the integration is not shallow, i.e., structure information gained from one technique can be used by the others and vice-versa.

We point out that each rule leverages its own parameter inference technique. However, for predictive synthesis, the main bias for generating extractions without examples is the regularity detected in the structure of the outputs generated by the different options. Any such predictive synthesis and intelligent parameter inference technique will fail in certain cases; that is, there are going to be cases when the technique will be unable to infer any parameter or it will infer a wrong set of parameters. We discuss some such cases in our evaluation in §6.

Extraction Rules. The final step of the Unravel procedure extracts tables from partial structures. This is done by using a number of simple rewrite rules (known as extraction rules) that combine syntactic elements from partial structures into tables. Formally, an *extraction rule* $\mathcal{E} : \mathbb{S} \times 2^{\mathsf{Tables}} \rightarrow 2^{\mathsf{Tables}}$ takes as input a partial structure and a possibly empty set of already extracted tables and combines them in order to produce a set of new tables. We informally describe the extraction rules used in Unravel.

- *Base case.* Every syntactic element in the partial structure is a table with a single column.
- *Simple combinations.* Given two tables with no overlapping extracted values and with the same number of rows, we can produce a table with the same number of rows that includes the columns from both tables. Similarly, given two tables with the same number of columns and no overlapping extracted values, we can concatenate them to produce a table with the same number of columns, but with rows from both the tables.
- *Syntactic joins.* We can join two tables with no overlapping extracted values syntactically based on "closest preceding row" and "closest following row" relations between two tables. Given tables T_1 and T_2, the join table $T_1 \bowtie_{cpr} T_2$ associates each row in T_1 with the row in T_2 that precedes it and is syntactically closest to it. The join $T_1 \bowtie_{cfr} T_2$ can be similarly defined.

4.3 Example and Discussion

We provide a fully worked out example of Unravel and then discuss some modifications to and features of the framework.

Example 4.4. Here, we illustrate all the steps in the full extraction process from the file discussed in Example 4.1.

- The starting point is always a partial structure containing a single syntactic element which contains only one string, i.e., the full file (say se_F).
- We apply the rule DELIMSPLIT$[\backslash n]$ to produce a syntactic element se_L that contains the lines of the file. The produced partial structure is $\{\mathsf{se}_F, \mathsf{se}_L\}$.
- We apply the rule PATTERNFILTER$[\mathtt{[A-Z]}*]$ on se_L to produce two syntactic elements—one containing only the lines with state codes (`PA` and `CA`), and the other with the lines containing city details. Call these $\mathsf{se}_{\mathsf{State}}$ and $\mathsf{se}_{\mathsf{Details}}$.
- Now, we use the delimiter rule DELIMSPLIT$[,]$ on the element $\mathsf{se}_{\mathsf{Details}}$ to produce the syntactic elements $\mathsf{se}_{\mathsf{County}}$ (Allegheny, Delaware, Los Angeles), $\mathsf{se}_{\mathsf{Zip}}$, and $\mathsf{se}_{\mathsf{City}}$.
- At this point, the partial structure contains all information needed to interpret the file and produce the desired table.
- We apply the table generation rules:
  (1) Using the simple combination rule we can produce a single details table containing the columns $\mathsf{se}_{\mathsf{County}}$, $\mathsf{se}_{\mathsf{Zip}}$, and $\mathsf{se}_{\mathsf{City}}$. The syntactic element $\mathsf{se}_{\mathsf{State}}$ gives us a single column state table.
  (2) Using the "closest preceding row" rule to join the details table with the state table produces the required table as shown in Example 4.1.

Decoding. The definition of extracted values in terms of sub-strings glosses over the issue of *decoding functions.* For example, the substring `"Sugar, Spice, and \"Everything Nice\""` in a CSV is to be interpreted as `Sugar, Spice, and "Everything Nice"`. Here, the extracted value is given by a substring that processed through a decoding function that "unquotes and unescapes". Other formats that need decoding include Base64, HTML/XML character codes, etc.

Decoding and recursive decoding are common in text files. The framework can natively handle these by defining extracted values as alternating substring and decoding operations $[i_0, j_0] \circ d_0 \circ \ldots \circ [i_n, j_n] \circ d_n$; This value is equal to s_{n+1} where $s_0 = \mathbb{T}$ and $s_{k+1} = d_k(s_k[i_k, j_k])$. While we excluded decoding for the sake of notational and conceptual simplicity, the same formalism can be reproduced with this new definition of extracted values.

Noise and malformed documents. An additional complication which we have not mentioned up to now is that real-world text documents are often malformed containing noise in the form of missing values, spurious lines, etc. For example, for DELIMSPLIT, we have found many practical cases where the number of delimiters is not the same in each substring of a syntactic element (e.g., when some records in a CSV file have different number of columns than others). In UNRAVEL, we can modify most rules to handle noise up-to a threshold by dropping/modifying syntactic element values whenever they do not match an expected format. For instance, our implementation of DELIMSPLIT can either drop strings which have a different number of delimiters, or fill in missing fragments with empty strings.

Example 4.5 (Noise). Consider a variant of the file presented in Figure 7 which has the following two lines prepended to the original file.

```
A list of 911 calls in the Philedelphia region.

Test data obtained from Kaggle; Usable under ODbl, Downloaded on 2017-05-31, Original size: 1500KB, m rows; n cols; 0 empty cells, handled correctly.
```

These header lines are noise, i.e., not a part of the tabular data in the file. This noise is handled in the following two ways.

In-Rule Noise Handling. In the first line, neither DELIMSPLIT$[;]$ nor DELIMSPLIT$[,]$ can extract data that is consistent with the other lines, as the line does not have commas or semi-colons. The *in-rule* noise handling of the DELIMSPLIT kicks in: if a small fraction of rows do not produce the same number of columns as the rest, we ignore those rows. With this change, we produce syntactic elements without the first row. Note that this in-rule noise handling has to be optionally written by each rule author.

In-structure Noise Handling. Now, in the second line we have that (using notation from Figure 7):

- col_1, col_2, col_3, col_4, and col_5 are equal to `Test data obtained from Kaggle; Usable under ODbl`, `Downloaded on 2017-05-31`, `Original size: 1500KB`, `m rows; n cols; 0 empty cells`, and `handled correctly`.
- col_A, col_B, col_C, col_D, and col_E are equal to `Test data obtained from Kaggle`, `Usable under ODbl, Downloaded on 2017-05-31, Original size: 1500KB, m rows`, `n cols, 0 empty cells`, and `handled correctly`.

However, syntactic element $\mathsf{col}_A \wedge \mathsf{col}_2$ is not well-defined under strict semantics, as it is not defined on the second line. Now, *in-structure* noise handling of the Unravel handles this case as follows: if a small fraction of values in $\mathsf{SE}_1 \wedge \mathsf{SE}_2$ are not defined, we define the noisy version of $\mathsf{SE}_1 \wedge_{\mathsf{Noise}} \mathsf{SE}_2$ to ignore these elements. On applying this rule, all elements of the second line are ignored, and we are left with only the data from line 3 onwards, which is the desired result. In-structure noise handling is common to all rules, and is built in to the formalism.

Minimizing number of examples. Unravel potentially requires fewer examples than alternative example driven extraction frameworks (say FlashExtract [Le and Gulwani 2014]) in many cases. This is due to the ability to combine interpretation rules with examples, i.e., Unravel is searching over higher-level constructs (interpretation rules) than most by-example systems. For example, consider a text file that contains lines of the form `June^Dry^Hot` and `November^Rainy^Cool`. Here, Unravel can work with one example (say $(1,1) \mapsto$ `June`) while FlashExtract requires three (say $(1,1) \mapsto$ `June`, $(1,2) \mapsto$ `Dry`, and $(1,3) \mapsto$ `Hot`). The reason behind this can be explained as follows:

- Given $(1,1) \mapsto$ `June`, FlashExtract aims to find a program that can extract values like `June` from each line. It may come up with the program *Extract from beginning of line upto* `^`. This is not sufficient to extract the other columns.
- In the same scenario, Unravel aims to find an interpretation rule which produces `June` from the first line. It produces the rule DelimSplit[`^`], which is sufficient to extract the other columns too.

5 THE UNRAVEL PROCEDURE

We presented the mechanics of the Unravel framework in §4. However, just defining the rules is not sufficient:

- Applying the presented rules in arbitrary combinations will quickly produce an unwieldy number of partial structures. How do we procedurally apply rules to produce reasonable partial structures?
- Given all the partial structures that can be produced using the rules, which one do we pick, and how do we extract the appropriate table from it?

We address these issues by first providing a non-deterministic generic procedure, and then resolving the non-determinism using locally optimal search and pruning.

5.1 A Generic Procedure

Algorithm 1 presents a generic non-deterministic procedure for extracting tables from a text document. Intuitively, the procedure starts from the least-refined partial structure (2) and repeatedly (lines 5-13) (a) applies interpretation rules, and (b) combines structures with meet and join operations to produce additional partial structures—here, each partial structure produced must be consistent with the examples. Then, the procedure picks one partial structure to perform the extractions (line 15) and repeatedly applies extraction rules on this structure to produce tables (lines 16-19). One of the tables produced is then selected and returned (lines 21-22)—again this selection is to be consistent with any examples provided. The procedure depicted is highly non-deterministic and the quality of the results are highly dependent on the non-deterministic Select$*$ choice functions.

Theorem 5.1. *For all sets of inputs, the table returned by Algorithm 1 satisfies the three well-formedness requirements of the* ad-hoc structure interpretation problem*.*

Algorithm 1 Structure Interpretation of Text Files

```
Require: Text document $\mathbb{T}$
Require: Set of interpretation rules $\mathbb{R}$
Require: Set of extraction rules $\mathbb{E}$
Require: Set of examples Examples
Ensure: Relational table $\mathsf{Table}^*$
 1: se ← $\langle \mathbb{T}[0, |\mathbb{T}|] \rangle$    ▷ Syn. element with full document
 2: $\mathcal{S}_0$ ← {se}    ▷ Least-refined partial structure
 3:
 4: Structs ← {$\mathcal{S}_0$}
 5: while * do    ▷ Produce more structures
 6:     if * then
 7:         $\mathcal{S}$ ← SelectStruct(Structs)
 8:         ($\mathcal{S} \rightarrow_{\mathcal{R}} \mathcal{S}'$) ← SelectRuleApplication($\mathbb{R}$, $\mathcal{S}$, Examples)
 9:         Structs ← Structs ∪ {$\mathcal{S}'$}
10:     else
11:         $\mathcal{S}_1$ ← SelectStruct(Structs)
12:         $\mathcal{S}_2$ ← SelectStruct(Structs)
13:         Structs ← Structs ∪ {$\mathcal{S}_1 \vee \mathcal{S}_2, \mathcal{S}_1 \wedge \mathcal{S}_2$}
14:
15: $\mathcal{S}^*$ ← SelectInterpretation(Structs)
16: Tables ← ∅
17: while * do    ▷ Extract tables
18:     $\mathcal{E}$ ← SelectExtractionRule($\mathbb{E}$, Tables, $\mathcal{S}^*$)
19:     Tables ← Tables ∪ $\mathcal{E}(\mathcal{S}^*$, Tables)
20:
21: Table$^*$ ← SelectTable(Tables, Examples)
22: return Table$^*$
```

Optimizations. We mention three general optimizations over the algorithm that we use in our implementation:

- First, instead of separate interpretation and extraction steps in Algorithm 1, we store a set of tables that can be extracted from each partial structure $\mathcal{S}$ along with it. For each rule application $\mathcal{S} \rightarrow_{\mathcal{R}} \mathcal{S}'$, the tables of $\mathcal{S}'$ can be computed from tables of $\mathcal{S}$ and the new syntactic elements produced by $\mathcal{R}$.
- Second, since we are interested in tables (as opposed to deeply nested tree structures), we ignore partial structures that contain deeply nested syntactic elements, i.e., if a partial structure $\mathcal{S}$ contains a chain of elements $\mathsf{se}_0 \lneq \mathsf{se}_1 \ldots \lneq \mathsf{se}_n$ for a sufficiently large n.
- Third, in certain cases, we combine rules that frequently applied one after the other into a single one. For example, we often apply DelimSplit$[d_1]$ and DelimSplit$[d_2]$ one after the other, with d_1 being record separator, and d_2 being the column separator. We introduce a new rule DelimFile$[d_1, d_2]$ that first splits by d_1 and then d_2 to directly obtain a table.

5.2 Instantiating Algorithm 1

We present a fully instantiated table extraction technique that uses a search and prune strategy to narrow down the non-determinism in Algorithm 1. The pruning and search are guided by ranking scores that are based on *interpretation confidence* and *interpretation regularity* (§4).

Interpretation confidence. Each rule has an associated interpretation confidence: this a measure of how likely it is that the rule can interpret elements that are not intended to be interpreted in that manner. Intuitively, rules that interpret more complex structures have higher scores: for example, Json and Xml are ranked higher than DelimSplit. Note that this score is fixed for each rule, and does not depend on the exact data or rule instantiation.

In the implementation, each rule author has to provide this confidence score for the rule. For the built in rules, we use a simple $0 - 1$ approach with most rules getting a score of 0, and rules that handle specialized data formats (in our case, XML, HTML, JSON, XML) getting a score of 1. We believe such a coarse metric is sufficient for most cases that arise in practice. Intuitively, an interpretation confidence of 1 for the Xml rule represents that it is very unlikely that a string not intended to be XML data could potentially be interpretable as XML. On the other hand, the DelimSplit has a score of 0 as it is more likely that a file not intended to be a delimited file could accidentally be interpreted as a delimited file (say, for example, by having exactly 1 comma in each line).

Example 5.2 (Interpretation Confidence). Consider a syntactic element se with substrings that resemble `{ "captain": "Picard", "officer": "Riker", "ship": { "name": "USS Enterprise", "number": "NCC-1701-E" } }`. Now, se can be interpreted by either Json or DelimSplit[,] rules: that is, we either treat the strings as JSON objects or as comma-separated values. Intuitively, the rule Json has a higher score and takes precedence: if a syntactic element is interpretable as valid JSON, it is likely to be the intended interpretation. If a string is parsable as JSON, it is almost always the intended interpretation.

Interpretation Regularity. This metric acts as a proxy for the description length metric mentioned in §4. The guiding principle here is *a syntactic element in which all values are similar can be described in fewer bits.* For example, a rule application that produces a syntactic element containing only numbers can be described in fewer bits (e.g., using a binary encoding of numbers) an arbitrary strings. Similarly, strings that can be represented using most data types have a shorted description length. The interpretation regularity score for a partial structure is the sum of individual scores for each syntactic element elements in it. Our interpretation regularity score is based on several factors:

- Data-type detection: If all the strings in a syntactic element can be interpreted as a known data-type (in the implementation, dates, numbers, boolean, categorical, guid, IP addresses, phone numbers, and zip codes), it is given a higher score.
- String regularity: If all the strings in a syntactic element are matched using a string regular expression, it is given a higher score. Here, the score is dependant of the specificity of the regular expression.

While other regularity factors exist, we found that the above two were sufficient for all benchmark cases.

Example 5.3 (Interpretation Regularity). Consider a syntactic element consisting of strings that resemble `2019-01-31, Antoine, 45, ant@ex.ca.edu` and `2019-10-23, Antony, 52, tony.sop@comp.com`. Now, this element can be interpreted using either of the rules DelimSplit[−] or DelimSplit[,]. The first one produces elements of the form `2019`, `01`, and `31, Antoine, 45, ant@ex.ca`, while the second produces elements of the form `2019-01-31`, `Antony`, `45` and `tony.sop@comp.com`.

We score the second extraction higher as the extraction has known types, i.e., dates and email addresses.

Given the above ranking scores (for interpretation confidence and interpretation regularity score), we instantiate the non-determinism Algorithm 1. For SelectRuleApplication on a given $\mathcal{S}$, we pick the rule application with the highest combined (a) Interpretation confidence score for $\mathcal{R}$, (b) Interpretation regularity score for $\mathcal{S}'$, and (c) Coverage (as defined in § 4). For SelectStruct, SelectInterpretation, and SelectTable, we pick the structure or table with the highest optimality score. This gives us a "best-first" style exploration of search space, with each *interpretation path* leading to one table in Tables out of which we select the optimal one.

This instantiation of Algorithm 1 does not guarantee that we return the globally optimal table: we make locally optimal choices and use pruning by interpretation confidence. However, in practice, we usually reach either the globally optimal table, or close to it. To compensate for this potential non-optimality and to study the effects, in the implementation, we produce top k ranked tables instead of the top 1 by modifying SelectInterpretation and SelectTable to produce k tables. Further, since the optimality criteria is only a proxy for user intent, sometimes the globally optimal table is not the user intended (or ground truth) one.

5.3 Faster Extractions: Re-interpreting Structures

Algorithm 1 does not guarantee that the table is extracted using the most efficiently operationalizable sequence of rules. While this does not matter for one-shot tasks, it is an important factor when extracting data from a large number of similarly formatted files. Our approach to this issue is *re-interpretation*: For each rule application $\mathcal{S} \to_{\mathcal{R}} \mathcal{S}'$ in the interpretation path, we attempt to find an alternative $\mathcal{R}'$ such that: (a) $\mathcal{S} \to_{\mathcal{R}'} \mathcal{S}'$, and (b) $\mathcal{R}'$ is more efficient to operationalize than $\mathcal{R}$. Re-interpretation takes advantage of the fact that the partial structure framework acts as a shared language between different interpretation rules. Furthermore, since the required syntactic elements are known for re-interpretation, the interpretation using $\mathcal{R}'$ can be *example driven* with the examples coming from $\mathcal{S}'$ (as shown in the following example).

Example 5.4. Consider the file shown in Figure 1: the first step in interpreting this file is to identify the records. By default, Unravel uses the regex split rule to say *records start on lines that match the regular expression* `\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}Z` *(i.e., timestamps in the ISO-8601 format)*. Using this interpretation of records is not ideal for performance: regular expression matching is an expensive operation. However, a significantly more efficient rule would be *records are exactly* 3 *lines each.*

Generating this second interpretation by default is a fairly expensive due to the lack of guidance: the algorithm would guess different alternatives for the parameter 3, and explore all the resulting structures to eventually find the best table. However, on re-interpretation, the records already produced by the first rule application act as guiding examples. Given the concrete records as examples, inferring the parameter 3 is trivial.

5.4 Interacting with Unravel

Algorithm 1 does not explicitly call out the points where a user may intervene or provide more information, i.e., it only depicts the fully predictive version of Unravel. In practice, the user may interact with Unravel at 3 different points to provide additional guidance (see Example 5.5).

- *Examples.* At each iteration of the while loop (lines 5-13), the user may provide additional examples to Unravel. These examples are (a) used to eliminate all partial structures $\mathbb{S}$ that are inconsistent with the new examples, and (b) used to aid parameter inference during rule application in line 8.
- *Rule selection.* On lines 7 and 8, a user may intervene to pick both the structure $\mathcal{S}$, as well as the rule $\mathcal{R}$ to apply. This feature is useful in resolving local ambiguity: for example,

disambiguating a file to be fixed width as opposed comma separated. In many cases, this disambiguation could also be done through examples rather than naming the rule that is to be applied. However, each option is more convenient in different cases, and studying the usability aspects is left as future work.

- *Table selection.* On line 21, a user may pick the particular table Table^* that is to be returned from the set of candidate tables Tables.

Example 5.5. Section 3 illustrates the typical case where providing examples is useful. In Figure 6, a user may provide the example `2016` for the column representing the year of release. This example hints to the algorithm that the syntactic element `(2016)`, `(2008)`, `(2010)` should be interpreted further using additional rules to extract the required column. Note that these examples may be provided either at the beginning of the extraction process, or at any point during the execution of the while loop (lines 5-13).

```
1,423    6,714   54,599,320
2,589    8,129  122,439,034
3,590   49,249  345,902,901
```

Consider the text file above, that can be interpreted both as a comma-separated file with 5 columns, as well as a fixed width file with 3 columns. Here, Unravel first applies DelimSplit[\n] to split the file into individual lines. At this point, a user may inform the tool (at line 8) that the FixedWidth rule is to be applied next. Using this hint, Unravel is able to correctly identify the parameters for the FixedWidth rule and produce the correct interpretation, i.e., a fixed width file with 3 columns. Note that the user may also have provided the example `1,423` which would also allow the procedure to distinguish between the two options.

6 EVALUATION

We describe our evaluation of the Unravel framework.

Benchmark data. We collected text files for the evaluation from:

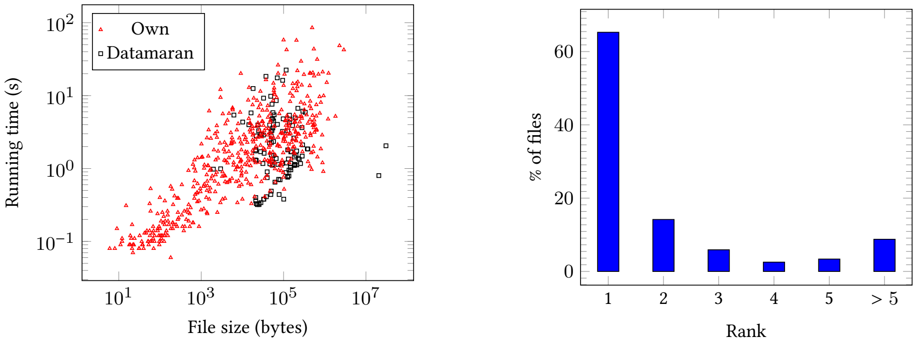
(1) 114 files[6] from benchmarks used for the evaluation of Datamaran in Gao et al. [2018], out of which 15 come from the PADS project [Fisher et al. 2008]; and
(2) 503 files from own benchmarks consisting of: (a) files from engineering teams that develop BI and data processing software, and (b) various online sources (e.g., help forums, web pages, YouTube tutorial videos).

Aims. We evaluate the following questions: (1) How does Unravel handle the benchmark w.r.t. the correctness of the extracted tables, the performance of the extraction, and what are the properties of the extracted tables? (2) How does Unravel compare with the state-of-the-art approach Datamaran? (3) What is the effect of the interactivity in Unravel? (4) How difficult it is to extend Unravel with a new rule and what are the effects of the extension? (5) What are the practical effects of re-interpretation?

Correctness and performance. To determine whether Unravel correctly extracts the data from some file, we first need a ground-truth for each input file. For some files in the benchmarks the ground-truth was already defined (e.g., for data on help forums and from the product teams). For files where such ground-truth was missing we have manually determined a reasonable tabular structure as a ground-truth.[7]

[6] We ignore 11 files from Gao et al. [2018] as they do not contain any tabular structure (as also noted by the authors).

[7] This is not always easy, as a file might have different interpretations based on application; in these cases we picked the table that made most sense.



(a) Running time.

(b) Ranking distribution.

Fig. 8. Evaluation Details.

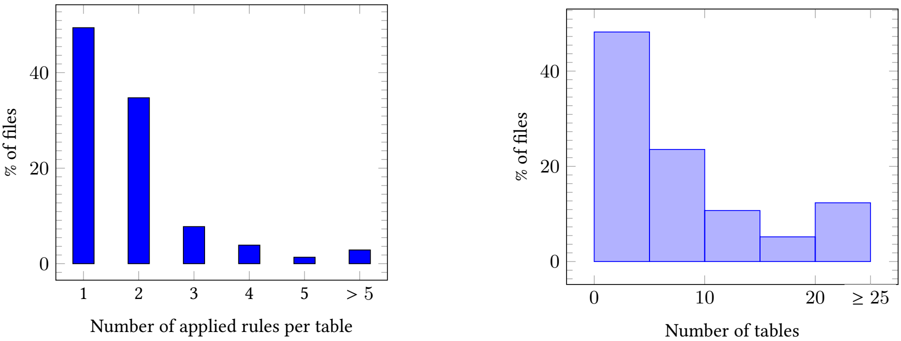
We consider a file correctly handled when one of the generated tables (Tables in line 21 of Algorithm 1) *exactly* matches the desired (ground-truth) output, i.e., we do not allow over- or under-splitting. However, the desired table need not be top-ranked for a file to be considered correctly handled (we discuss ranking below).

Using this criteria, Unravel *correctly handles* 593 *(*96%*) of the files* in the benchmark. The cases that the tool does not handle correctly were mainly due to two reasons: (a) the files contained rare esoteric formats that could not be handled by any generic rule (e.g., FASTQ format for biological sequences), or (b) the automated inference of parameters for a standard rule failed. The right solution for (a) is to take advantage of the extensibility and to plugin a domain-specific parser as a black-box rule (we did not do this in the experiments to avoid tweaking the tool after-the-fact). For (b), the solution is to improve parameter inference for existing rules with more intelligence. Examples for cases in (b) are: (1) when the header part of the file is much larger then the data part, the inference for Skip$[k]$ will not infer the large enough k, (2) when there are too many noisy rows in the (CSV-like) delimited files the inference for DelimSplit$[d]$ will not infer the correct delimiter d, (3) when the regular expression required for data extraction is too complicated (i.e., too expressive for the inference algorithm) the inference for RegexSplit$[r_1, \ldots, r_n]$ will fail to infer the correct sequence of regular-expressions $r_1, \ldots, r_n$.

Figure 8a shows the running time of Unravel on the files in the benchmark against the size of the files (and categorized by the benchmark source). *For* 84% *of the files the running time is under* 5s*, while the average running time is* 3.4s; *this is within the usability requirements for BI and data-analysis tools.*

Figure 8b shows the distribution of the ranking of the intended (desired) table. Around 91% *of files have the intended table with rank* ≤ 5. In most cases when the top-ranked table is not the desired one, the cause is *over-splitting* (our ranking scores tend to prefer more columns). In our experience with mass-market tools that use predictive analysis for data extraction, it is common and acceptable for users to browse the top-k results; this is because any AI-infused software is going to be imperfect in some cases, and browsing through the top-k results to potentially find an intended result is better than having to do the task manually. Further, while our ranking method works well enough for the given benchmark, we believe that this is an orthogonal problem to the one explored in this paper and that there is a lot of opportunity to research more sophisticated ranking methods.

Properties of extracted tables. In Figure 9a we show the the number of applied rules per desired (ground-truth) table. For 50.5% of the files the desired table is constructed by *applying more than*



(a) Distribution of the applied rules.

(b) Tables per file.

Fig. 9. Properties of the generated tables.

one rule; *this shows the need for a tool that can infer multiple cooperating data-extraction logics for a single file.* However, these are the numbers for only a single interpretation, i.e., a user might extract additional data using examples, in which cases Unravel will apply further rules (as we discuss below).

In Figure 9b we show the number of learned (generated) tables per file on our benchmark, that is, the number of tables that the tools finds after the confidence-based pruning, but before ranking.

In 35.2% of the cases the desired table *contains noise or has missing fields.* By manually inspecting 20 of those cases we distinguish the following distinct categories: (a) missing keys in key-value pairs (see the example in Figure 4 and Figure 5; 3 cases), (b) missing fields in HTML tables (see the example in Figure 6; 3 cases), (c) missing fields or wrong number of columns in delimited (CSV-like) files (similar to the example in Example 4.5; 8 cases), and (d) records that do not completely conform to the dominant record pattern (for example extracted by the RegexSplit rule; 6 cases).

Finally, in 63% of the cases the desired table contains *multi-line or noise records*, and in 4% of the cases the desired table contains *multiple record types.* These cases are out of reach for the tools that do not learn record boundaries or handle multiple record types (e.g., ColumnSplit[Raza and Gulwani 2017a]).

Comparison with Datamaran. Next, we compare Unravel to Datamaran; we perform this comparison because Datamaran is the most similar tool to Unravel, and their benchmark is available and contains various interesting text files.[8]

We point out that we do not perform direct comparison of the tools because the only available Datamaran version has some bugs (as confirmed with the authors via email communication), and we were unable to reproduce the results reported in Gao et al. [2018] or obtain meaningful results on our benchmark (see the discussion below). Therefore, here we discuss the results for Unravel that we have obtained on the Datamaran benchmark w.r.t. the results reported in Gao et al. [2018], and we briefly summarize our experience with running Datamaran on our benchmark.

On the Datamaran benchmark Unravel produces correct output for 103 (90%) files from Gao et al. [2018]. Gao et al. [2018] report that Datamaran correctly handles 96% of the files from the same benchmark. This shows that Unravel *produces comparable results* on the benchmark that mostly contains log-like files, for which Datamaran is fine-tuned.

It was not possible to directly and automatically compare the outputs of Datamaran and Unravel on our own benchmark, because Datamaran in almost all cases generated the output

[8] Although Datamaran is intended to mainly handle log files their benchmark includes a number of non-log files; additionally, it also includes files from the PADS [Fisher et al. 2008] project.

```
WARNING [01/01/14 : 22:32:19] [class: baseClass] Overflow
ERROR [04/07/14 : 20:04:13] [unknown] Uncaught exception
INFO [01/01/14 : 22:32:19] [fnc: login] Successful login
```

(a) Sample log file.

WARNING	01/01/14	22:32:19	class: baseClass	Overflow
ERROR	04/07/14	20:04:13	unknown	Uncaught exception
INFO	01/01/14	2:32:19	fnc: login	Successful login

(b) Unravel output.

Description	C	R	T	S	F
Web pages	0	10	0	0	0
CSV files	3	1	0	4	2
Fixed-width files	4	0	0	1	5
Key-value files	2	7	0	0	1
Log-like files	7	0	1	2	0
Other	4	0	0	0	6

(c) Manual inspection results (C - correct, R - wrong rows, T - invalid record types, S - skipped/filtered records, F - wrong fields).

WARNING	01	01	14	:	22:32:19]	class:	baseClass	Overflow
ERROR	04	07	14	:	20:04:13]	unknown]	Uncaught	exception
INFO	01	01	14	:	2:32:19]	fnc:]	login	Successful login

(d) Datamaran output.

2013	10	·	{	·	url	api	po	hk	98740	1	·	response	{	·	message	No	Records	Found	}	}
2013	10	·	{	·	url	api	po	hk	98740	1	·	response	{	·	message	No	Records	Found	}	}
2013	10	·	{	·	url	api	po	hk	98740	1	·	response	{	·	message	No	Records	Found	}	}

(e) Datamaran's (partial) output for the file in Figure 1.

Fig. 10. Details of the comparison with Datamaran.

that was different from Unravel's output. We illustrate this on two examples. Figure 10a shows a sample log file, and Figure 10b and Figure 10d show tables extracted by Unravel and Datamaran, respectively. Although Datamaran extracts all the data, it performs several over-splits, among them on the last three columns which then end up misaligned (which is clearly wrong). Next, Figure 10e shows partial [9] table extracted by Datamaran for the file shown in Figure 1. The outputs again do not match, but we consider both our ground-truth and the Datamaran's output reasonably correct interpretations of the input; we point out that although Datamaran performs several over-splits (on time and URL fields), extracts some delimiters (e.g., {) and JSON keys (e.g., `url`), we consider this extraction correct because it extracts all fields and they are correctly aligned. [10]

Hence, to provide fair comparison we manually inspected the output of Datamaran on 60 files in our own benchmark, across different categories (see Figure 10c). Based on the discussion above, we considered Datamaran's output correct when it contains a reasonable tabular interpretation of the input file (note that this is necessarily subjective): all rows are extracted and all fields are correctly aligned (we still allow minor over- and under- splits, as discussed on the example above).

As expected, Datamaran handles very regular data (e.g. log files) quite well, however, it does not understand semantic data (e.g. HTML structure, CSV quoting semantics, key-value pairs). The results summary is given in Figure 10c; we categorize the results as follows: (1) correct (C in the summary table), (2) the records are not correctly identified (mostly due to inability to understand semantic data; R in the table), (3) similar records are identified as different record types (T in the table), (4) records that do not conform to the most common pattern are skipped (i.e, they are treated as noise; S in the table), and (5) the fields are not correctly extracted from the records (similar as Figure 10d above; F in the table). We believe that for a non-buggy version of Datamaran the

[9] We omit some field, denoted by ·, for presentation sake.

[10] Although Datamaran would not be able to extract correct data from this file if some JSON keys were missing, or their order changed.

results would be similar on non-log files due to conceptual differences, while we expect that the results should be better for log files.

Interactivity. Unravel handles 554 (90%) of all the files in the benchmark *completely predictively*, that is, in those cases no user guidance is needed. Of the remaining 63 files, Unravel can handle 39 (62%) in *by-example mode*. In those cases the result is obtained using the following:

- a prefix (of up to 3 rows) of a desired table,
- a (single) example of how to further split an already extracted column, and
- a sample of different records (one sample for each record type), for record partitioning.

To further investigate the mixture of the interactive and predictive mode, we have provided column-extraction examples on 11 web-extraction files from the benchmark. In these cases Unravel extracts tabular data from the web-page predictively, and the user has to only specify further column extraction examples (based on her needs).

For example, in Figure 6, the user needs to provide the example `$534.86M` ⟶ `534.86`, and Unravel learns how to extract the whole column from that. Overall, we have extracted additional data from 1-4 columns per table, and in all but one case it was sufficient to provide *only a single example* for each extracted column (in one case we needed two examples).

We point out that unlike purely by-example approaches (e.g., FlashExtract [Le and Gulwani 2014]), Unravel can in many cases also work without any examples. Further, we are not aware of other approaches that can handle the *mixture of predictive and by-example modes* to extract data.

Extensibility. We next describe the steps required to add a new rule to Unravel, on a concrete example of the rule to predictively extract table data from web pages. We also describe how adding this rule affected results in our benchmark.

Our web-extraction rule Web is based on a predictive system described in Raza and Gulwani [2017b]. Hence, Web is parametrized by the program P, whose input is a HTML DOM tree and output is a table. To add this new rule to Unravel we have to provide: (a) semantics of the rule, that is, we need a procedure to learn the parameter P based on the input string, and a procedure that, given the parameter P, runs the input string and produces the output table; (b) a procedure that generates the confidence based on the input string and the learned parameter P (as discussed earlier, the confidence is simply a zero-one parameter).

Raza and Gulwani [2017b] define operational semantics for P and a procedure to learn it predictively, hence we can use this system as a black-box to provide semantics for the Web rule, we only need to parse the input string into the HTML DOM tree (there are numerous libraries for this). Assigning confidence is also quite easy: we assign *high confidence to any learned parameter P*, since this means that the input can be parsed as a valid HTML and we can extract a table from it, and hence we can be quite confident that this is a correct interpretation. In our Unravel implementation this was *easy to implement in around 50 lines of code.*

Finally, we discuss the effect of adding the Web rule to the framework on the evaluation. Without the Web rule, the framework cannot handle 14 files that contain HTML content (it does not have effect on the files without HTML content). The average running time remains the same on non-HTML files, but increases 2.7x on HTML files, when the Web rule is not present. This happens because the Web rule has high-confidence on HTML files, and hence Unravel converges faster and finds the desired interpretation.

Re-interpretation. We evaluate *re-interpretation* capability of Unravel by implementing a new rule in our system, based on a table extraction system inside Microsoft Power BI. This system has the ability to generate *M code* [Microsoft 2020] to perform table extraction from a text file, based on the user-provided output examples. Hence, we perform an experiment to check whether we can use the predictively learned output from Unravel to learn a new interpretation, using the new by-example rule, from which we can then derive executable M code to perform table extraction.

Out of 554 files in our benchmark that Unravel handles predictively we were able to learn a new interpretation (by example) in 244 (44%) cases; that is, in this concrete case this shows that in 44% cases Unravel can help to generate M code for users *completely predictively*, where earlier they would need to provide examples. Further, this shows an interesting synergy between two systems - one that predictively uses various composable rules to learn the tabular structure, but for which it might be difficult to generate executable-code, and another monolithic by-example system that generates executable code, but for which it might be difficult to develop predictive learning.

7 RELATED WORK

Automatic Extraction. The two closest work to Unravel are PADS [Daly et al. 2006; Fisher and Gruber 2005; Fisher and Walker 2011; Fisher et al. 2008] and Datamaran [Gao et al. 2018]. The PADS project simplifies ad hoc parsing among several dimensions: introducing a DSL for describing formats [Fisher and Gruber 2005], inferring such formats automatically [Fisher et al. 2008], and defining a markup language which users use to annotate for more effective text structure inference [Xi and Walker 2010]. Datamaran also parses log files automatically but unlike PADS, it does not require the record boundary to be given. The key difference is that Unravel is an *extensible* framework that allows easy plug-and-play of rules to parse mixed file formats. Additionally, it also allows users to provide examples to express their intent more precisely or to create custom output.

Known-format Extraction. Vaarandi and Pihelgas [2015] introduced LogCluster, an algorithm to cluster data and mine line pattern in log files. Du and Li [2016] presented an algorithm based on longest common subsequence approach to perform online parsing of streaming data. LogMine is a distributed system that efficiently generate patterns for logs [Hamooni et al. 2016]. Zhu et al. [2019b] perform a study on popular parsing methods, implementing them and comparing their performance. Commercial solutions such as Splunk [Splunk 2019], ELK [ELK 2019], LogEntries [Logentries 2019] are able to parse popular log formats to enable further advanced analytics. The above approaches assume that the log file is structured and do not work on mixed formats.

Webpages contain valuable information that are usually locked in the HTML format. Wrapper induction learns procedures to extract data from webpages [Kushmerick 1997]. It can be supervised, where wrappers based on query languages such as CSS or XPath are learned from examples [Gulhane et al. 2011; Nielandt et al. 2016; Raza and Gulwani 2017b]. Unsupervised wrapper induction leverages the idea that some elements in the HTML DOM are repeated hence can potentially be parts of tabular data [Arasu and Garcia-Molina 2003; Crescenzi et al. 2001]. Unlike HTML, the text structure is our case is implicit and more diverse.

JSON is also popular data format; for example, the libraries `pandas.read_json` [Read Json 2019] and `pandas.io.json.json_normalize` [Json Normalize 2019] are common tools to convert JSON into tables. On the other hand, Unravel is more general and the above tools can be used as black-box rules in Unravel.

Program Synthesis. Abstract interpretation-based synthesis systems leverage the abstraction over the programs to refine the specification and reduce the search space [Peleg et al. 2018; Vechev et al. 2010]. In contrast, the abstraction in Unravel is over the data (i.e., file structure). Our rules are more akin to code/programs, which iteratively build an abstraction over the underlying data.

Various prior work applied programming by examples to extract data [Le and Gulwani 2014; Miller 2002; Raza and Gulwani 2017b]. FlashExtract [Le and Gulwani 2014] allows users to parse semi-structured files by giving examples. LAPIS [Miller 2002] lets users highlight text regions using constraints such as string literals, lightweight built-in functions, and relations. Unlike these tools, Unravel may work without any examples. However, Unravel may use these tools as black boxes in which users can enter the interactive mode to further refine the structure using examples. Wrangle [Guo et al. 2011; Kandel et al. 2011a] is an interactive tool that proactively suggests users possible transformations, but it is limited in extraction capability and cannot handle mixed formats.

Program by demonstration (PBD) is another great way to extract data. Helena [Chasins and Bodik 2017] and Roussillon [Chasins et al. 2018] allow users to synthesize extraction programs by demonstrating their extraction tasks. While we can handle many cases without any examples, for the cases where we do require examples, incorporating such PBD systems that provide such a demonstration trace may give our system more information about which component rules to prefer and further guide our algorithm to quickly converge to the correct solution with fewer examples. This will be beneficial for settings where the user can provide a trace of demonstration rather than just input-output examples.

8 CONCLUSION AND FUTURE DIRECTIONS

We identify and motivate the problem of data extraction from text files that contain *mixture of formats* (both standard and ad-hoc). To tackle this problem, we introduce Unravel, a framework for mixed-format data extraction that can operate in *predictive* (without any user examples) as well as *interactive* modes. The framework is based on the *partial structure* formalism that allows non-trivial combination of *multiple cooperating data-extraction logics*, making Unravel highly *extensible*. We evaluate Unravel on a diverse set of benchmarks and show that it can handle cases that no single previous approach could handle.

Unravel's ability to extract data from complex, hierarchical, semi-structured text files opens up multiple avenues for future work. We intend to explore the use of Unravel's predictive extraction to program synthesis based data wrangling techniques by allowing them to automatically handle diverse semi-structured text files as inputs. Additionally, the success of the partial structure framework suggests the use of similar structural domains in other programming-by-example domains such as string and tree transformations: for example, can we use rules to identify structure common to multiple example inputs and produce programs in terms of this common syntactic structure? Another interesting possibility is to explore the use of machine learning models to guide the rule selection and table ranking. This has the potential to reduce the amount of user interaction necessary for an extraction. Another interesting direction is to explore how can Unravel help with debugging of data extraction (e.g., as explored earlier by Mayer et al. [2015]). For example, since Unravel, besides the resulting tables, also generates a list of rules to extract the data (i.e., a program), this information could be potentially used to help the user spot what has been missed by visually decorating the input file. Additionally, the differences in visual decorations that stem from executing multiple programs could be used to proactively highlight ambiguities to the user.

On the theoretical side, we also like to study additional algebraic structure of partial structures, and work on extending them with structured syntactic elements to allow extraction of tree data. The partial structures framework is one formalism to allow multiple program synthesis tools to interact and share information in the data extraction domain. A question now arises whether similar frameworks are possible for other program synthesis domains. In regard to the implementation, we are in the process of releasing Unravel for general commercial use. We will evaluate the technique's capability in real world scenarios in terms of the range of files automatically handled, as well as explore the additional user interaction modalities.

ACKNOWLEDGMENTS

The authors would like to thank Hila Peleg and the anonymous reviewers for their valuable and constructive feedback on earlier versions of this paper.

REFERENCES

Arvind Arasu and Hector Garcia-Molina. 2003. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/872757.872799

Sarah Chasins and Rastislav Bodik. 2017. Skip Blocks: Reusing Execution History to Accelerate Web Scripts. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 51 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133875

Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

Cognos Analytics 2019. Cognos Analytics: How XML files are flattened. https://www.ibm.com/support/knowledgecenter/en/SSEP7J_10.2.2/com.ibm.swg.ba.cognos.dg_rtm_wb.10.2.2.doc/c_howxmlfilesareflattenednd09ab.html. Accessed: 2019-11-20.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. 2001. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 109–118. http://dl.acm.org/citation.cfm?id=645927.672370

Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch what I do: programming by demonstration.* MIT Press, Cambridge, MA, USA. http://portal.acm.org/citation.cfm?id=168080

Mark Daly, Yitzhak Mandelbaum, David Walker, Mary Fernández, Kathleen Fisher, Robert Gruber, and Xuan Zheng. 2006. PADS: An End-to-end System for Processing Ad Hoc Data. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. ACM, New York, NY, USA, 727–729. https://doi.org/10.1145/1142473.1142568

Data Miner 2019. Data Miner: Extract data from any website with 1 click. https://data-miner.io/. Accessed: 2019-11-20.

M. Du and F. Li. 2016. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 859–864. https://doi.org/10.1109/ICDM.2016.0103

ELK 2019. ELK. https://www.elastic.co/what-is/elk-stack. Accessed: 2019-11-20.

Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 295–304. https://doi.org/10.1145/1065010.1065046

Kathleen Fisher and David Walker. 2011. The PADS Project: An Overview. In *Proceedings of the 14th International Conference on Database Theory* (Uppsala, Sweden) *(ICDT '11)*. ACM, New York, NY, USA, 11–17. https://doi.org/10.1145/1938551.1938556

Kathleen Fisher, David Walker, Kenny Qili Zhu, and Peter White. 2008. From dirt to shovels: fully automatic tool generation from ad hoc data.. In *POPL*, George C. Necula and Philip Wadler (Eds.). ACM, 421–434. http://dblp.uni-trier.de/db/conf/popl/popl2008.html#FisherWZW08

Yihan Gao, Silu Huang, and Aditya G. Parameswaran. 2018. Navigating the Data Lake with DATAMARAN: Automatically Extracting Structure from Log Datasets.. In *SIGMOD Conference*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 943–958. http://dblp.uni-trier.de/db/conf/sigmod/sigmod2018.html#GaoHP18

Pankaj Gulhane, Amit Madaan, Rupesh Mehta, Jeyashankher Ramamirtham, Rajeev Rastogi, Sandeep Satpal, Srinivasan H. Sengamedu, Ashwin Tengli, and Charu Tiwari. 2011. Web-scale Information Extraction with Vertex. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 1209–1220. https://doi.org/10.1109/ICDE.2011.5767842

Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive Wrangling: Mixed-initiative End-user Programming of Data Transformation Scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/2047196.2047205

Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. ACM, New York, NY, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

Json Normalize 2019. pandas.io.json.json_normalize. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.io.json.json_normalize.html/. Accessed: 2019-11-20.

Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011a. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. ACM, New York, NY, USA, 3363–3372. https://doi.org/10.1145/1978942.1979444

Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011b. Wrangler: interactive visual specification of data transformation scripts.. In *CHI*, Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare (Eds.). ACM, 3363–3372. http://dblp.uni-trier.de/db/conf/chi/chi2011.html#KandelPHH11

Nicholas Kushmerick. 1997. *Wrapper Induction for Information Extraction.* Ph.D. Dissertation. Seattle, WA, USA. AAI9819266.

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples.. In *PLDI*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 55. http://dblp.uni-trier.de/db/conf/pldi/pldi2014.html#LeG14

Logentries 2019. Logentries. https://logentries.com/. Accessed: 2019-11-20.

Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *PVLDB* 12, 12 (2019), 1914–1917. https://doi.org/10.14778/3352063.3352098

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *28th ACM User Interface Software and Technology Symposium (UIST 2015)* (28th acm user interface software and technology symposium (uist 2015) ed.). ACM – Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/user-interaction-models-for-disambiguation-in-programming-by-example/

Microsoft. 2020. Power Query M formula language. (2020). https://docs.microsoft.com/en-us/powerquery-m/

Robert C. Miller. 2002. *Lightweight Structure in Text.* Ph.D. Dissertation.

Joachim Nielandt, Antoon Bronselaer, and Guy de Tré. 2016. Predicate Enrichment of Aligned XPaths for Wrapper Induction. *Expert Syst. Appl.* 51, C (June 2016), 259–275. https://doi.org/10.1016/j.eswa.2015.12.040

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276520

Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 382–405.

Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's Wheel : An Interactive Data Cleaning System. *VLDB* (2001). http://www.vldb.org/conf/2001/P381.pdf

Mohammad Raza and Sumit Gulwani. 2017a. Automated Data Extraction Using Predictive Program Synthesis.. In *AAAI*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 882–890. http://dblp.uni-trier.de/db/conf/aaai/aaai2017.html#RazaG17

Mohammad Raza and Sumit Gulwani. 2017b. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 882–890. http://dl.acm.org/citation.cfm?id=3298239.3298368

Read Json 2019. pandas.read_json. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html. Accessed: 2019-11-20.

Splunk 2019. Splunk. https://www.splunk.com/. Accessed: 2019-11-20.

Risto Vaarandi and Mauno Pihelgas. 2015. LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM) (CNSM '15)*. IEEE Computer Society, Washington, DC, USA, 1–7. https://doi.org/10.1109/CNSM.2015.7367331

Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-Guided Synthesis of Synchronization. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/1706299.1706338

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.* 286–300. https://doi.org/10.1145/3314221.3314588

Qian Xi and David Walker. 2010. A Context-free Markup Language for Semi-structured Text. *SIGPLAN Not.* 45, 6 (June 2010), 221–232. https://doi.org/10.1145/1809028.1806622

Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019a. Tools and benchmarks for automated log parsing.. In *ICSE (SEIP)*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 121–130. http://dblp.uni-trier.de/db/conf/icse/seip2019.html#ZhuHLHXZL19

Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019b. Tools and Benchmarks for Automated Log Parsing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, Piscataway, NJ, USA, 121–130. https://doi.org/10.1109/ICSE-SEIP.2019.00021