Incremental Learning of System Log Formats

[Extended Abstract]

Kenny Q. Zhu
Shanghai Jiao Tong University
kzhu@cs.sjtu.edu.cn

Kathleen Fisher
AT&T Labs Research
kfisher@research.att.com

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

System logs come in a large and evolving variety of formats, many of which are semi-structured and/or non-standard. As a consequence, off-the-shelf tools for processing such logs often do not exist, forcing analysts to develop their own tools, which is costly and time-consuming. In this paper, we present an incremental algorithm that automatically infers the format of system log files. From the resulting format descriptions, we can generate a suite of data processing tools automatically. The system can handle large-scale data sources whose formats evolve over time. Furthermore, it allows analysts to modify inferred descriptions as desired and incorporates those changes in future revisions.

Categories and Subject Descriptors

D.4.9 [**Operating Systems**]: Systems Programs and Utilities; D.3.4 [**Programming Languages**]: Processors

General Terms

Languages, Algorithms

Keywords

Grammar induction, analysis of system logs, domain-specific languages, parsing, tool generation, ad hoc data, PADS

1. INTRODUCTION

At AT&T and elsewhere, system implementers and administrators manipulate a wide variety of system logs on a daily basis. Common tasks include data mining, querying, performing statistical analysis, detecting errors, and transforming the data to standard formats. Because many of these logs are in non-standard formats, there are often no ready-made tools to help process these logs. Often the data is hierarchical rather than relational in nature, making automatic loading into relational databases difficult. As a result, system engineers have to resort to writing one-off parsers, typically in Perl or Python, to ingest these data sources, a tedious, error-prone and costly process.

To facilitate working with such log files, we developed PADS[2], a high-level declarative specification language for describing the physical formats of ad hoc data. A PADS description for a data source precisely documents the format of the data, and the PADS system compiles such descriptions into a suite of useful processing tools including an XML-translator, a query engine, a statistical analyzer, and programmatic libraries and interfaces. Analysts can then either use these generated tools to manage the logs or write custom tools using the generated libraries.

A significant impediment to using PADS is the time and expertise required to write a PADS description for a new data source. If such a source is well-documented, writing a PADS description is straightforward and requires time proportional to the existing documentation. Often, however, such data sources are not well documented and the user must adopt an iterative process to produce a description: write a partial description, use this description to parse the data, flag segments of the data that do not match the description, refine the description to cover these cases, and repeat. This process is time-consuming, often requiring days for complex formats.

As a first step towards addressing this problem, we developed the LEARNPADS[1] system [3, 5], which automatically infers a PADS description from sample data, and thus eliminates the need for hand-written descriptions. The LEARNPADS system successfully produces correct descriptions for a range of small data sources, but it cannot handle larger ones because the system includes a memory-intensive algorithm designed to process the entire data source at once.

In this paper, we take the next step towards automatically inferring descriptions of system log files by adapting LEARNPADS to work *incrementally*. With this modification, the system takes as input an initial description and a new batch of data. It returns a modified description that extends the initial description and covers the new data as well. The initial description can be supplied by the user or the system can use the original LEARNPADS system to infer it. This iterative architecture also allows the user to take the output of the system, make revisions such as replacing generated field names like `IP_1` with more meaningful names like `src`, and then use the refined description as the basis for the next round of automatic revision.

In the rest of the paper, we give a brief overview of PADS and the original inference system (Section 2). We then describe the incremental inference algorithm (Section 3), discuss its implementation (Section 4), give some experimental results (Section 5), and finally

[1]A demo is available from the PADS website `www.padsproj.org`.

```
Punion client_t {
  Pip       ip;      // 207.136.97.49
  Phostname host;    // ks38.kms.com
};
Punion auth_id_t {
  Pchar unauthorized : unauthorized == '-';
  Pstring(:' ':) id;
};
Pstruct request_t {
   "GET ";    Ppath    path;
   " HTTP/";  Pfloat   http_ver;
   '"';
};
Precord Pstruct entry_t {
         client_t       client;
   ' ';  auth_id_t      remoteID;
   ' ';  auth_id_t      auth;
   " ["; Pdate          date;
   ':';  Ptime          time;
   "] "; request_t      request;
   ' ';  Pint           response;
   ' ';  Pint           length;
};
```

Figure 2: PADS/C description for the `wl` format

conclude (Section 6). Space considerations preclude discussion of related work; however, our earlier paper [3] contains an extensive discussion of other grammar induction systems.

2. PADS AND THE ORIGINAL LEARNPADS

We use a simple web server log format, which we call `wl`, to illustrate the principal features of the PADS data description language. Figure 1 shows a fragment of such data, which is comprised of a sequence of records, separated by newlines. Each record contains a number of fields delimited by white space. For example, the first record starts with an IP address, then has two dashes, a time stamp enclosed in square brackets, a quoted HTTP message, and finally two integers. The second record shows some variation: the IP address becomes a hostname and the second dash becomes an identifier.

PADS uses a type-based metaphor to describe ad hoc data. Each PADS type plays a dual role: it specifies a grammar by which to parse the data and a data-specific data structure in which to store the results of the parse. PADS/C is the variant of PADS that uses C as its host language. Hence, PADS/C types are drawn by analogy from C, and the generated data structures and parsing code are in C.

Figure 2 shows a PADS/C specification that describes each of the records in Figure 1. The specification consists of a series of declarations. Types must be declared before they are used, so the last declaration `entry_t` describes the entirety of a record, while the earlier declarations describe data fragments. Type `entry_t` is a **`Precord`**, meaning it comprises a full line in the input, and is a **`Pstruct`**, meaning it consists of a sequence of named fields, each with its own type. For convenience, **`Pstruct`**s can also contain anonymous literal fields, such as `" ["`, which denote constants in the input source. The generated representation for `entry_t` will be a C struct with one field for each of the named fields in the declaration. The type `client_t` is a **`Punion`**, meaning the described data matches *one* of its branches, by analogy with C unions. In particular, a `client_t` is either an IP address (`Pip`) or a host name (`Phostname`), where `Pip` and `Phostname` are PADS/C *base types* describing IP addresses and hostnames, respectively.

In general, base types describe atomic pieces of data such as integers (`Pint`) and floats (`Pfloat`), characters (`Pchar`) and strings (`Pstring(:' ':)`), dates (`Pdate`) and times (`Ptime`), paths (`Ppath`), *etc.* Strings represent an interesting case because in theory they could go on forever, so `Pstring` takes a parameter which specifies when the string stops: in this case, when it reaches a space. To account for more general stopping conditions, a programmer may use the base type `Pstring_ME`, which takes a regular expression as a parameter. With this type, the corresponding string is the longest that matches the regular expression. The first branch of the **`Punion`** `auth_id_t` illustrates the use of a *constraint*. It specifies that the `unauthorized` character must be equal to `'-'`. If the constraint fails to hold, the next branch of the union will be considered.

In addition to the features illustrated in Figure 2, PADS provides arrays, which describe sequences of data all of the same type; options, which describe data that may be present; and switched unions, which describe unions where a value earlier in the data determines which branch to take. Such unions illustrate that PADS supports *dependencies*: earlier portions of the data can determine how to parse later portions.

The goal of the LEARNPADS format inference engine is to infer PADS descriptions like the one in Figure 2 from raw data. From such a description, the PADS compiler can produce end-to-end processing tools fully automatically. A full description of the LEARNPADS algorithm appears in an earlier paper [3]. We give only a brief summary here.

LEARNPADS assumes that the input data is a sequence of newline-terminated records and that each record is an instance of the desired description. From such an input, it uses a a three-phase algorithm to produce a description. In the *tokenization* phase, LEARNPADS converts each input line into a sequences of tokens, where each token type is defined by a regular expression. Intuitively, these tokens correspond to PADS base types. In the *structure discovery* phase, LEARNPADS computes a frequency distribution for each token type and then uses that information to determine if the top-level structure of the data source is a base type, **`Pstruct`**, **`Parray`**, or **`Punion`**. Based on that determination, the algorithm partitions the data into new contexts and recursively analyzes each of those contexts, constructing the corresponding description as it recurses. This phase terminates with a candidate description. In the *format refinement* phase, the algorithm uses an information-theoretic scoring function to guide the application of rewriting rules. These rules seek to minimize the size of the description while improving its precision by performing structural transformations (such as merging adjacent **`Pstruct`**s), adding data dependencies, and constraining the range of various base types, *e.g.*, converting a general integer to a 32-bit integer.

The scoring function, which is based on the *minimum description length principle* [4], measures how well a description describes data by calculating the number of bits necessary to transmit both the description and the data *given the description*. We use the terms *type* and *data complexity* to refer to the number of bits necessary to encode the description and the data given the description, respectively. This function penalizes overly general descriptions, such as **`Pstring`**, which have a low type complexity but a very high data complexity. It also penalizes overly specific descriptions that painstakingly describe each character in the data. Such descriptions

```
207.136.97.49  - - [05/May/2009:16:37:20 -0400] "GET /README.txt HTTP/1.1" 404 216
ks38.kms.com - kim [10/May/2009:18:38:35 -0400] "GET /doc/prev.gif HTTP/1.1" 304 576
```

Figure 1: A fragment from a web server log in `wl` format

have a low data complexity but a high type complexity.

This algorithm produces good results for the small log files that we have experimented with, but it has two limitations: performance and adaptability. In terms of performance, the algorithm requires space quadratic in the input file size to perform the data dependency analysis, so it cannot be used on log files larger than the square of the size of usable memory. In terms of adaptability, the algorithm only considers its input data in constructing a description. Hence if tomorrow's log file has a new kind of record, the algorithm cannot modify the existing description; it must start from scratch.

3. THE INCREMENTAL ALGORITHM

To address these problems, we extended LEARNPADS to work incrementally. Given a candidate description `D`, the new algorithm uses `D` to parse the records in the data source. It discards records that parse successfully, since these records are already covered by `D`, but it collects records that fail to parse. When the algorithm accumulates M such records, where M is a parameter of the algorithm, it invokes the incremental learning step, described below, to produce a refined description `D'`. This refined description subsumes `D` and describes the M new records. In addition, the algorithm attempts to preserve as much of the structure of `D` as possible, so users supplying initial descriptions can recognize the resulting descriptions. The algorithm then takes `D'` to be the new candidate description and repeats the process until it has consumed all the input data. The initial description `D` can either be supplied by a user or it can be inferred automatically by applying the original algorithm to N records selected from the data source, where N is another parameter.

Intuitively, the incremental learning step works by attempting to parse each of the M records according to the current description `D`. It discards the portions of each record that parse correctly. If a portion fails to parse, that failure will be detected at a particular node in the description `D`. It collects these failed portions in an aggregation data structure `A` that mirrors the structure of `D`. After thus aggregating all the failures in the M records, the algorithm transforms `D` to accommodate the places where differences were found (*i.e.*, by introducing options where a piece of data was missing or unions where a new type of data was discovered). It then uses the original LEARNPADS algorithm to infer descriptions for the aggregated portions of bad data.

Figure 3 defines the data structures for descriptions `D`, data representations `R`, and aggregate structures `A`. In these definitions, variable `re` ranges over regular expressions, `s` and `t` over strings, and `i` over integers. A value with type `D` is the abstract syntax tree of PADS description: it is what we want to learn. For simplicity of presentation, we assume just two base types: integers and strings that match a regular expression. Synchronizing tokens, or *sync tokens* for short, correspond to string literals in PADS descriptions. Such tokens, which are often white space or punctuation, serve as delimiters in the data and are useful for detecting errors. We use binary pairs and unions to account for the **`Pstruct`**s and **`Punion`**s in PADS/C descriptions. An array has an element type described by `D`, a separator string s that appears between array elements, and a terminator string t. `Option D` indicates `D` is optional.

```
Descriptions:
Base ::= Pint | PstringME(re)
D ::=
  Base               (Base token)
| Sync s             (Synchronizing token)
| Pair (D_1, D_2)    (Pair)
| Union (D_1, D_2)   (Union)
| Array(D, s, t)     (Array)
| Option D           (Option)
Data representation
BaseR ::= Str s | Int i | Error
SyncR ::= Good | Fail | Recovered s
R ::=
  BaseR
| SyncR
| PairR (R_1, R_2)
| Union1R R | Union2R R
| ArrayR (R list, SyncR list, SyncR)
| OptionR R
Aggregation structure
A :: =
  BaseA Base
| SyncA s
| PairA(A_1, A_2)
| UnionA(A_l, A_r)
| ArrayA (A_elem, A_sep, A_term)
| OptionA A
| Opt A
| Learn [s]
```

Figure 3: Data structures used in incremental inference

A term with type `R` is a parse tree obtained from parsing data using a description `D`. Parsing a base type can result in a string, an integer or an error. Parsing a sync token `Sync s` can give three different results: `Good`, meaning the parser found `s` at the beginning of the input; `Fail`, meaning `s` is not a substring of the current input; or `Recovered s'`, meaning `s` is not found at the beginning of the input, but can be *recovered* after "skipping" string `s'`. The parse of a pair is a pair of representations, and the parse of a union is either the parse of the first branch or the parse of the second branch. The parse of an array includes a list of parses for the element type, a list of parses for the separator and a parse for the terminator which appears at the end of the array.

An aggregate structure is the *accumulation* of parse trees; it collects the data that cannot be parsed and therefore must be re-learned. The aggregation structure mirrors the structure of the description `D` with two additional nodes: an `Opt` node, and a `Learn` node. An invariant is that an `Opt` node always wraps a `BaseA` or a `SyncA` node, where it indicates that the underlying base or sync token is missing in some of the parses being aggregated, and therefore that the wrapped token should be made optional. The `Learn` node accumulates the bad portions of the data that need to be learned. The newly learned sub-descriptions will be spliced into the original description to get the new description.

Figure 4 gives pseudo-code for the incremental learning step. The `init_aggregate` function initializes an empty aggregate according to description `d`. Then for each data record `x`, we use the `parse`

```
incremental_step(d, xs) =
  as = [init_aggregate(d)];
  foreach x in xs {
    rs = parse(d, x);
    as' = [];
    foreach r in rs {
      foreach a in as {
        a' = aggregate(a, r);
        as' = a :: as'
      }
    }
    as = as'
  }
  best_a = select_best(as);
  d' = update_desc(d, best_a);
  return d'
```

Figure 4: Pseudo-code for the incremental learning step

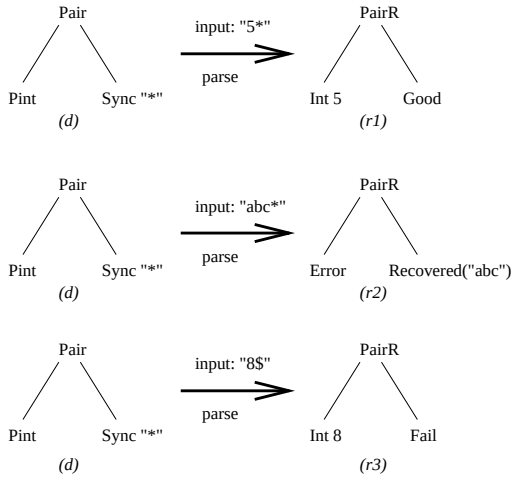

Figure 5: Result of parsing three input lines

function to produce a list `rs` of possible parses. We then call the `aggregate` function to merge each parse `r` in the current list of parses with each aggregate `a` in the current list of aggregates. We use '`::`' to denote consing an element onto the front of a list. When we finish parsing all the data lines and obtain a final list of aggregates `as`, we select the best aggregate according to some metric, and finally update the previous description `d` to produce the new description `d'` using the best aggregate.

To illustrate the parsing and aggregation phases of the algorithm, we introduce a simple example. Suppose we have a description d, comprised of a pair of an integer and a sync token "`*`", and we are given the following three lines of new input:

```
5*
abc*
8$
```

Figure 5 shows the three data representations that result from parsing the lines, which we call r_1, r_2 and r_3, respectively. Notice the first line parsed without errors, the second line contains an error for `Pint` and some unparsable data "`abc`", and the third contains a `Fail` node because the sync token `*` was missing. Figure 6 shows the aggregation of r_1 to r_3 starting from an empty aggregate. In general, `Error` and `Fail` nodes in the data representation trigger the creation of `Opt` nodes in the aggregate, while unparsable data is collected in `Learn` nodes.

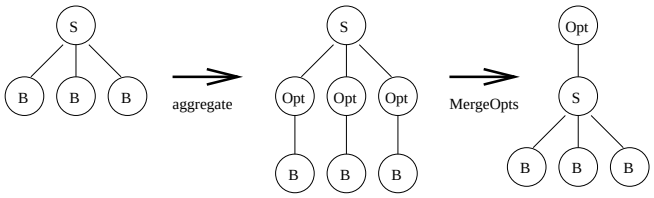

Figure 7: MergeOpts rewriting rule

4. IMPLEMENTATION

For purposes of presentation, we have described an idealized and unoptimized algorithm. Our actual implementation includes a number of refinements to improve the quality of the description and/or reduce the inference time. In this section, we discuss some of these refinements.

4.1 Token families

So far, parsing a `Sync` token yields one of three results: `Good`, `Fail` or `Recovered`. In the actual implementation, a `Sync` token can be not only a constant string, but also a constant integer, an integer range or a combination thereof. Consider parsing the token `Sync (Str "GET")` when the current input starts with "POST." The `parse_base` function indicates the result should be `Fail`. In reality, the input "POST" is in the same *family* as "GET," *i.e.*, a word, and it may very well be that this `Sync` token should have been an enumeration of words rather than a single word. To handle such cases, we created a fourth type of parse node, `Partial`, to indicate that the input belongs to the same family as the expected token but does not match exactly, *i.e.*, it is *partially* correct. During aggregation, partial nodes cause the description to be specialized to include the additional values. In the above example, the aggregate function will change the description to `Sync (Enum [Word "GET", Word "POST"])`. Such partial nodes reduce the number of parsing errors and produce more compact and meaningful descriptions.

4.2 Rewriting rules

When the incremental learning algorithm produces a refined description from an aggregate, the algorithm applies rewriting rules to the new description to improve its quality and readability. Most of the rules are data-independent and inherited from LEARNPADS, such as removing degenerate lists and flattening nested structs and unions. We introduce one new *data dependent* rule called *MergeOpts* to optimize a type pattern that occurs frequently during incremental learning. Recall that the aggregate function introduces `Opt` nodes above a `BaseA` or `SyncA` node whenever the corresponding `Base` or `Sync` token in the description failed to parse. When faced with an entirely new form of data, the algorithm is likely to introduce a series of `Opt` nodes as each type in the original description fails in succession. The *MergeOpts* rule collapses these consecutive `Opt` nodes if they are correlated, *i.e.*, either they are all always present or all always absent. To verify this correlation, the algorithm maintains a table that records the branching decisions when parsing each data line. It uses this table to determine whether to merge adjacent `Opt` nodes during rewriting. Figure 7 illustrates the effect of this rule. In the figure, S denotes a struct and B a base token.

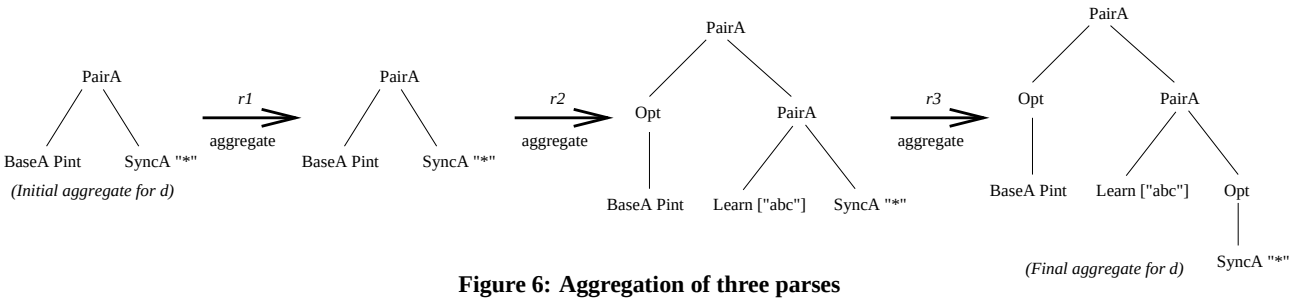

Figure 6: Aggregation of three parses

4.3 Performance

The pseudo-code in Figure 4 suggests the number of aggregates is of the order $O(m^n)$, where m is the maximum number of parses for a line of input and n is the number of lines to aggregate. Clearly, this algorithm will not scale unless m and n are bounded.

We have implemented several optimizations to limit the number of parses and aggregates. First, we do not return all possible parses when parsing a description component `D`. Instead, we rank the parses by a metric that measures their quality and return only the top k. The metric is a triple: $m = (e,\ s,\ c)$, where e is the number of errors, s is the number of characters skipped during `Sync` token recovery, and c is the number of characters correctly parsed. The metric is considered *perfect* if $e = 0$. Metric m_1 is better than m_2 if m_1 is perfect and m_2 is not, or if

$$\frac{c_1}{s_1 + c_1} > \frac{c_2}{s_2 + c_2}.$$

In practice, `parse` returns a list of *parse triples* $(r,\ m,\ j)$, where r is the data representation of the parse, m is the metric associated with r, and j is the position in the input after the parse. We define a `clean` function that first partitions the triples into groups that share the same *span, i.e.*, the substring of the input consumed by the parse. For each group, `clean` retains all perfect parses. If none exists, it retains the best parse in the group. We justify discarding the other triples because given a description d and a fixed span, we always prefer the parse with the best metric. This idea is similar to the dynamic programming techniques used in Earley Parsers [1]. Finally `clean` returns all the perfect triples plus up to the top k non-perfect triples. The `clean` function reduces the number of bad parses to a constant k while guaranteeing that if there is a perfect parse, it will be returned.

A second optimization, which we call *parse cut-off*, terminates a candidate parse when parsing a struct with multiple fields f_1, f_2, ..., f_n if the algorithm encounters a threshold number of errors in succession. This technique may result in no possible parses for the top-level description. In this case, we restart the process with the parse cut-off optimization turned off. A third optimization is memoization. The program keeps a global memo table indexed by the pair of a description `D` and the beginning position for parsing `D` which stores the result for parsing `D` at the specific position. Finally, we bound the total number of aggregates the algorithm can produce by selecting the top k aggregates with the fewest number of `Opt` and `Learn` nodes.

5. EVALUATION

To evaluate the incremental algorithm, we ran it and the original LEARNPADS system on 10 different kinds of system logs of various sizes. We conducted the experiments on a PowerBook G4 with a 1.67GHz PowerPC CPU and 2GB memory running Mac OS X 10.4. Table 1 summarizes the results. The second column lists the number of lines and the size of each log. The time columns give the total running time in seconds, and the TC columns give the type complexity of the final description. In general, a lower type complexity means a more compact description. For all benchmarks, the initial learn size N is 500 lines and the incremental learn size M is 100 lines. A "-" indicates the original system failed to produce a description within thirty minutes. Table 1 shows the incremental algorithm learns descriptions that are slightly less compact than the original but in a much shorter time.

Formats	K Lines/KB	original		incremental	
		Time	TC	Time	TC
interface	1.2/185	48.5	0.7	2.9	1.1
asl.log	1.5/552	31.9	0.9	13.5	1.5
error_log	4.5/409	93.1	0.1	0.9	0.1
access_log	8.2/551	130.5	0.3	2.8	0.3
coblitz	9.4/2561	-	-	31.9	2.9
pws	17.4/3432	-	-	133	5.7
ai.big	57.4/5608	-	-	26.2	0.5
exlog	260.8/76720	-	-	610	3.0
redirect	302.6/102404	-	-	1852	17.1
getbig	550.4/92192	-	-	668	8.8

Table 1: Exec. times (secs) and Type Complexities (KB)

To measure the correctness of the inferred descriptions, we generated parsers from each description and used them to parse the data. All formats parsed with zero errors except for the `pws` format, a form of Apache server log, which contains a number of errors. These errors arise because PADS uses greedy matching to parse unions. We are developing a smarter parser implementation to resolve this problem.

The second experiment measures the execution time of learning descriptions for a series of web server logs ranging in size from 200k to one million lines. This data source is private to AT&T, so we ran the experiments on an AT&T internal server which runs GNU/Linux and has a 1.60GHz Intel Xeon CPU with 8GB of memory. Figure 8 suggests the incremental algorithm scales linearly with the number of lines. In particular, the algorithm learns a description for a million-line web log in under 10 minutes. The inferred description yields a parser that correctly parses all lines in the log.

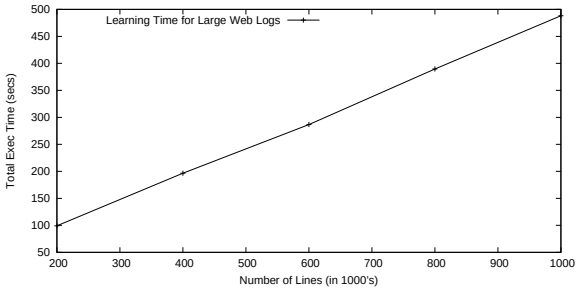

Figure 8: Scaling of increment algorithm

6. CONCLUSION

We have presented an incremental algorithm for inferring system log formats. We experimentally verified that this algorithm can produce quality descriptions within minutes when run on files with hundreds of thousands of lines. Our experience suggests that the quality of the final description is very sensitive to the quality of the initial description. Hence, we intend to work in the future on improving the original algorithm to produce better initial descriptions.

Acknowledgments

This material is based upon work supported by the NSF under grants 0612147 and 0615062, and a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Google.

7. REFERENCES

[1] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[2] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304, June 2005.

[3] K. Fisher, D. Walker, K. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL*, January 2008.

[4] P. D. Grünwald. *The Minimum Description Length Principle*. MIT Press, May 2007.

[5] Q. Xi, K. Fisher, D. Walker, and K. Q. Zhu. Ad hoc data and the token ambiguity problem. In *PADL'09*, 2009.