LearnPADS^{++}
Incremental Inference of Ad Hoc Data Formats

Kenny Q. Zhu[1], Kathleen Fisher[2], and David Walker[3]

[1] Shanghai Jiao Tong University
[2] Tufts University
[3] Princeton University

Abstract. An ad hoc data source is any semi-structured, non-standard data source. The format of such data sources is often evolving and frequently lacking documentation. Consequently, off-the-shelf tools for processing such data often do not exist, forcing analysts to develop their own tools, a costly and time-consuming process. In this paper, we present an incremental algorithm that automatically infers the format of large-scale data sources. From the resulting format descriptions, we can generate a suite of data processing tools automatically. The system can handle large-scale or streaming data sources whose formats evolve over time. Furthermore, it allows analysts to modify inferred descriptions as desired and incorporates those changes in future revisions. [4]

1 Introduction

Ad hoc data is any *non-standard*, *semi-structured* data source for which processing tools and libraries are not readily available. HTML, XML, and data in relational databases are not ad hoc because many tools exist to manage such data. Despite efforts to standardize data formats, ad hoc data persists in many domains ranging from computer system administration to financial transactions to health care to computational biology. Figure 1 shows an example of a piece of ad hoc data source.

People continue to produce and use ad hoc data because such formats are expedient and compact. Typical uses of these data sources include system fault

[4] This work was partially supported by NSFC Grants No. 61033002 and 61100050 and by NSF grant CCF-1016937. Any opinions, findings, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSFC or NSF. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

```
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
```

Fig. 1. A Fragment of a Simple Web Server Log `wl`

monitoring by tracking vital system health parameters in the system logs, intrusion detection by matching access patterns to intrusion models and data mining of scientific and financial data.

Despite the expediency of producing ad hoc data, these data formats become very difficult to deal with because of missing documentation, the lack of tools, and corruptions caused by repeated redesign and re-engineering over time. In the past, ad hoc data analysis usually involved writing a shell script or one-off wrapper program to parse each data format, a practice which is expensive, error-prone and brittle.

The PADS project [11] aims to solve the above problems. The central technology is a declarative, type-based, data description language that allows the user to specify the physical layout of data sources as well as semantic properties of the data. PADS specifications can be compiled into a suite of processing tools such as a statistical reporting tool, an XML converter and a query engine, and programming libraries including parser, printer and traversal functions. Figure 2 shows the PADS description for the `wl` data source, and Figure 3 demonstrates the XML translator output automatically generated from the PADS description.

```
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;
};
```

Fig. 2. PADS/C description for the `wl` format

```
<entry_t>
  <client>
    <ip>
      <elt><val>207</val></elt>
      <elt><val>136</val></elt>
      <elt><val>97</val></elt>
      <elt><val>49</val></elt>
      <length>4</length>
    </ip>
  </client>
  <remoteID>
    <unauthorized><val>-</val></unauthorized>
  </remoteID>
  <auth>
    <unauthorized><val>-</val></unauthorized>
  </auth>
  <date><val>2009-05-05</val></date>
  <time><val>16:37:20</val></time>
  <timezone><val>-0400</val></timezone>
  ...
</entry_t>
```

Fig. 3. Fragment of XML translator output from a `wl` record

The large scale as well as the streaming and evolving nature of many ad hoc sources led us to believe that a system which automatically *learns* a PADS description of a given data source and incrementally updates that description as the source evolves could significantly improve the productivity of ad hoc data users. As a first step, we developed an unsupervised algorithm LEARNPADS [7, 8] that automatically infers a PADS description of a data source by computing frequency statistics for the *tokens* in the data and using an information theoretic score to guide description optimization.

This algorithm, however, has three important limitations: first, it requires that all data fit into main memory and contains procedures that are quadratic to the size of data, and therefore cannot *scale* to very large sources; second, when the data format evolves over time, the description has to be learned from scratch; and finally, machine learned description, while optimized for both precision and conciseness at the same time, may not be very user-friendly in terms of readability.

In this paper, we propose a new algorithm that *incrementally* infers descriptions of large scale or evolving ad hoc data sources. [5] 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. The initial description may be supplied by the user or automatically generated using the original LEARNPADS system. This iterative architecture enables the learning of a very large data source by partitioning it into smaller batches and updating the description from one batch to the next. It also allows the user to modify the description output at the end of an iteration (*e.g.*, renaming the automatically generated variable names), and insert the revised description back into the loop.

The main contributions of this paper are:

1. The design of a new system for generation of data descriptions and end-to-end ad hoc data processing tools from example data. The system is incremental and interactive, allowing it to process streaming data a chunk at a time, and allowing users to intercede to correct, adapt or modify intermediate results.
2. The engineering and optimization of algorithms that allow the system to handle large, industrial data sources of 30GB or more in a matter of a few hours.
3. The evaluation and analysis of the system on 16 different examples drawn from various industrial data sources.

In the rest of the paper, we describe the new incremental inference algorithm (Section 2) and give a comprehensive experimental evaluation of the system (Section 3). We then compare this system with some related work (Section 4) and finally conclude the paper (Section 5).

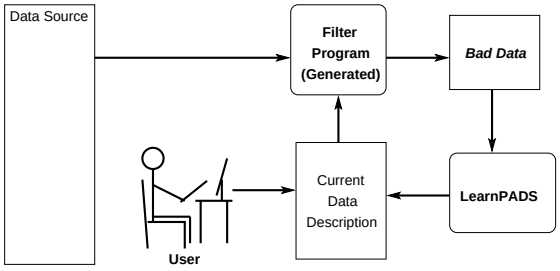

Fig. 4. An Overview of the Incremental Learning Framework

2 Main Algorithm

Our main algorithm can be characterized as a user-assisted bootstrapping processing, depicted in Figure 4. Given a candidate description `D`, the 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. Specifically, if a portion of a record fails to parse, that failure will be detected at a particular node in `D`. These failed portions are collected in an aggregation data structure `A` that mirrors the structure of `D`. When the algorithm accumulates M such records, where M is a parameter of the algorithm, it 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, and merge these new sub-descriptions into the transformed description to produce a new, 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. This is so because the updates are localized to only parts of `D` that incur parsing errors. At this point, the user can *optionally* get into the loop and makes modification to the description to create `D''`. The algorithm then makes `D''` the new candidate description and repeats the process until it has consumed all the input data. We call the main loop in Figure 4 the *incremental learning step*. The initial description `D` can either be supplied by a user or be inferred automatically by applying the original algorithm to N records selected from the data source, where N is another parameter.

In the following, we present the algorithm in more detail.

2.1 Preliminaries

Figure 5 defines the data structures for descriptions `D`, data representations `R`, and aggregate structures `A`. Some data types, such as the switched union, are omitted for the succinctness of the presentation. In these definitions, variable `re`

[5] A preliminary version of this paper appeared in an informal workshop [14].

```
Basic notation:
c           (a string character)
s1.s2       (concatenation of strings)
first(s)    (first character of s)
prefix(s)   (set of prefixes of s)
sprefix(s)  (set of strict prefixes of s)
len(s)      (length of s)

Descriptions:
Base ::= Pint | PstringME(re) | PstringFW(e)
D ::=
  Base              (Base token)
| Sync s            (Synchronizing token)
| Pair (x:D1, D2)   (Pair with dependency)
| Union (D1, D2)    (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 (R1, R2)
| Union1R R | Union2R R
| ArrayR (R list, SyncR list, SyncR)
| OptionR (R option)

Aggregation structure:
A :: =
  BaseA Base
| SyncA s
| PairA(A1, A2)
| UnionA(A1, Ar)
| ArrayA (A_elem, A_sep, A_term)
| OptionA A
| Opt A
| Learn [s]
```

Fig. 5. Preliminary data structures used in incremental inference

ranges over regular expressions, `e` over host language expressions, `s` and `t` over strings, and `i` over integers. For simplicity of presentation, we assume just three base types: integers, strings that match a regular expression and strings with a fixed width specified by an expression. Synchronizing tokens, or *sync tokens* for short, correspond to string literals in PADS descriptions. Such tokens, which are often white spaces or punctuation, serve as delimiters in the data and are useful for detecting errors. The binary dependent pairs `Pair (x:D1, D2)` are a simplification of PADS more general `Pstruct`s. The variable `x` refers to the data parsed by `D1` and may be used in `D2`. The union `Union (D1, D2)` provides a choice between descriptions `D1` and `D2`. An array description `Array(D, s, t)` has an element type described by `D`, a separator string `s` that appears between array elements, and a terminator string `t`. Finally, `Option D` indicates `D` is optional. To resolve ambiguities, unions are biased towards their first element, arrays are biased towards a longest match semantics and options are biased towards matching as opposed to not matching.

A term `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 option is either the parse of its body or empty. 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 aggregation structure accumulates the set of currently unparseable data fragments whose form must be learned for inclusion in the grammar. The aggregation structure mirrors the structure of the description `D` with two additional nodes: an `Opt` node and a `Learn` node. The `Learn` nodes accumulate extra data

whose structure must be learned. The `Opt` nodes do the opposite: they mark where data were missing. An invariant of the aggregation structure is that newly inserted `Opt` nodes always wrap either a `BaseA` or a `SyncA` node.

2.2 Incremental Learning Step

Figure 6 gives pseudo-code for the *incremental learning step*. The input is the current description `D` and a batch of data records `xs`. The `init_aggregate` function initializes an empty aggregate according to description `D`. During parsing, the algorithm iteratively updates a list of possible aggregates `As`, seeded with the initial aggregate of `D`. For each data record `x`, the algorithm uses the `parse` function to produce a list `Rs` of possible parses. It then calls 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 prepending an element onto the front of a list.) Note that the potentially large number of parses and the growing list of aggregates in the inner loop are the performance bottleneck. We will show in Section 2.6 some strategies to alleviate this complexity.

```
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'
```

Fig. 6. Pseudo-code for the incremental learning step

When the system finishes parsing all the input data, the algorithm uses the `select_best` function to select the best aggregate from the list of candidate aggregates `As`. The `select_best` function counts the total number of `Opt` and `Learn` nodes in each of the aggregates, and returns the one with the smallest number. The idea is that the aggregate with the smallest number of added nodes is more likely to represent a description that is closest to the original description.

Finally, the `update_desc` function uses the structure of the best aggregate to update the previous description `D` to produce the new current description `D'`. The `update_desc` function works by doing two things. First, it converts the aggregate structure back to a PADS description with `Opt` nodes translated to `Poption` types. In addition, it invokes the LEARNPADS format inference algorithm to learn a sub-description for the data collected at each of the `Learn` nodes and replaces these `Learn` nodes with these new sub-descriptions. Second, it uses rewriting rules to improve the overall description.

2.3 Parsing

Our parser is a top-down recursive descent parser that performs error detection and recovery using synchronizing tokens. Figure 7 describes the most important elements of the parsing algorithm. For simplicity and brevity, we describe the algorithm abstractly using a relation of the form `(R,m) ∈ L(D,E,s,s')`. This relation may be read "using description `D` and operating within the environment `E`, parsing the input `I = s.s'` will consume input prefix `s` and leave `s'` as the residual input, returning the parse tree `R` and correctness metric `m`." The environment `E` is a mapping from variable names `x` to parse trees `R`. This environment stores the binding of variables to parse trees that the PADS dependent pair construct introduces. We use the symbol 'ϵ' to denote the empty environment.

```
Base:
(Int (atoi s), m) ∈ L(Pint,E,s,s')
  if re = (+|-)?[0-9]+
  and s ∈ L(re)
  and s'' ∈ prefix(s') and s.s'' ∉ L(re)
  and m = (0,1,0,len(s))
(Error, (1,0,0,0)) ∈ L(Pint,E,"",s'),
  if x ∈ prefix(s') then x ∉ L((+|-)?[0-9]+)
(Str s, m) ∈ L(PstringME(re),E,s,s'),
  if s ∈ L(re)
  and s'' ∈ prefix(s') and s.s'' ∉ L(re)
  and m = (0,1,0,len(s))
(Error, (1,0,0,0)) ∈ L(PstringME(re),E,"",s'),
  if x ∈ prefix(s') then x ∉ L(re)
(Str s, m) ∈ L(PstringFW(e),E,s,s')
  if E(e) = Int k and k >= 0
  and s = c1...ck and m = (0,1,0,k)
(Error, (1,0,0,0)) ∈ L(PstringFW(e),E,"",s')
  if E(e) ≠ Int k for any k > 0
(Error, (1,0,0,0)) ∈ L(PstringFW(e),E,"",s')
  if E(e) = Int k and k > 0 and len(s') < k

Sync:
(Good, (0,1,0,len(s))) ∈ L(Sync(s),E,s,s')
(Recovered s1, m) ∈ L(Sync(s2),E,s,s')
  if s = s1.s2
  and s3.s2 ∉ sprefix(s1.s2) for any s3
  and m = (1,0,len(s1),len(s2))
(Fail, (1,0,0,0)) ∈ L(Sync(s),E,"",s')
  if s ∉ prefix(s')

Pair:
(PairR (R1,R2), (m1 + m2))
        ∈ L(Pair(x:D1, D2),E,s1.s2,s')
  if  (R1, m1) ∈ L(D1,E,s1,s2.s')
  and (R2, m2) ∈ L(D2,E[x → R1],s2,s')

Union:
(Union1R R, m) ∈ L(Union(D1, D2),E,s,s')
  if (R, m) ∈ L(D1, E, s, s')
(Union2R R, m) ∈ L(Union(D1, D2),E,s,s')
  if (R, m) ∈ L(D2, E, s, s')

Main parse function:
parse(D, s) = {R | (R, m) ∈ L(D,ϵ,s,"")}
```

Fig. 7. Definition of `parse` function (excerpts)

The *parse metric* `m` measures the quality of a parse. It is a 4-tuple: (e, g, s, c), where the e is the number of tokens with parse errors, g is the number of tokens parsed correctly, s is the number of characters skipped during `Sync` token recovery, and c is the number of characters correctly parsed. To sum two parse metrics, we sum their components: $(e_1, g_1, s_1, c_1) + (e_2, g_2, s_2, c_2) = (e_1 + e_2, g_1 + g_2, s_1 + s_2, c_1 + c_2)$. We compare parse metrics by comparing the ratios of correctly parsed characters against erroneous tokens and the estimated number of skipped tokens. We estimate the number of skipped tokens by computing the fraction of the number of skipped characters over the estimated token length. Hence, $(e_1, g_1, s_1, c_1) \geq (e_2, g_2, s_2, c_2)$ iff

$$\frac{c_1}{e_1 + \frac{s_1}{\max((s_1+c_1)/(e_1+g_1),1)}} \geq \frac{c_2}{e_2 + \frac{s_2}{\max((s_2+c_2)/(e_2+g_2),1)}}$$

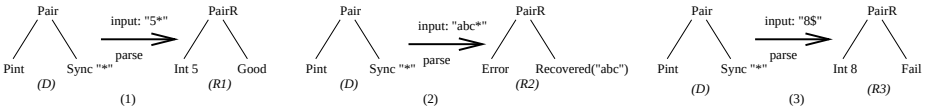

Fig. 8. Result of parsing three input lines

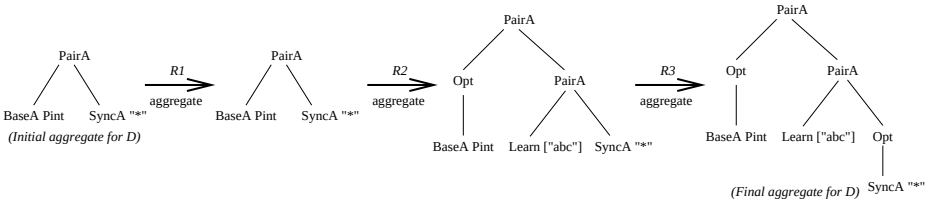

Fig. 9. Aggregation of three parses

2.4 An Example of Parsing and Aggregation

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*" and "abc*" and "8$". Figure 8 shows the three data representations that result from parsing the lines, which we call $R1$, $R2$ and $R3$, respectively. Notice the first line parsed without errors, the second line contains an error for `Pint` and some unparseable data "`abc`", and the third contains a `Fail` node because the sync token `*` was missing. Figure 9 shows the aggregation of $R1$ to $R3$ 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 unparseable data is collected in `Learn` nodes.

2.5 Description Rewriting

Once we have successfully parsed, aggregated and relearned a new chunk of data, we optimize the new description using rewriting rules. Our original non-incremental algorithm already had such an optimization phase; we have modified and tuned the algorithm for use in the incremental system.

Description rewriting optimizes an information-theoretic Minimum Description Length (MDL) score [9], which is defined over descriptions `D` as:

$$\mathrm{MDL}(\mathtt{D}) = \mathrm{TC}(\mathtt{D}) + w \times \mathrm{ADC}(x_1, \ldots, x_k \mid \mathtt{D}),$$

where $\mathrm{TC}(\mathtt{D})$ is called the *type complexity* of `D` and $\mathrm{ADC}(x_1, \ldots, x_k \mid \mathtt{D})$ is called the *atomic data complexity*. The type complexity is a measure of the size of the abstract syntax of `D`. The atomic data complexity of data records $x_1, \ldots, x_k$ relative to `D` is the number of bits required to transmit an *average* data record given the description `D`. The MDL score of `D` is the weighted sum of these two components. Our experiments indicate a weight w of approximately 10 is effective

in our domain. Given a rewriting rule that rewrites `D` to `D'`, the rule fires if and only if MDL(`D`) $\leq$ MDL(`D'`). Rewriting continues until no further rule can fire. Hence, our rewriting strategy is a greedy local search.

The original learning system contains many MDL-based rewriting rules, for example, to flatten nested structs and unions and to refine ranged types. *BlobFinding* is an important new rewriting rule which takes a given sub-description `D` and uses a heuristic to determine if the type complexity of `D` is too high w.r.t. the amount of data it covers. If this is true, and there is an identifiable constant string or pattern `re` that immediately follows `D`, then we rewrite `D` to `Pstring_SE(:re:)`. This rule is tremendously helpful in controlling the size and complexity of learned descriptions. Without it, descriptions can grow in complexity to the point where parsing is slow and the algorithm fails to scale.

We also introduced a new *data dependent* rewriting rule called *MergeOpts* to optimize a pattern that occurs frequently in descriptions 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.

2.6 Optimizations

The pseudo-code in Figure 6 suggests the number of aggregates is of the order $O(p^n)$, where p 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 p and n are bounded. To deal with this problem, we have implemented several optimizations to limit the number of parses and aggregates.

The first key optimization culls parses based on the parse metric `m`. To be more precise, we instrument the implementation of the `parse` function to return 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 rather than just representations. We define a `clean` function that retains all perfect parses or, if no perfect parse exists, the best k non-perfect parses within the same *span*. This idea is similar to the dynamic programming techniques used in Earley Parsers [6].

A second optimization, the *parse cut-off* optimization, 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 may result in no possible parses for the top-level description, in which case we restart the process with this 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`. This table stores the result for parsing according to `D` at the specific position.

Table 1. The data sources

Name (Large)	Size	Lines	Description
redstorm	34.18 GB	219096168	Supercomputer log from Sandia National Lab
liberty	30.833 GB	265569231	Supercomputer log from Sandia National Lab
dalpiv.dat	15.41 GB	25867260	Yellow pages web server log
vshkap2.log	10.33 GB	89662433	Syslog format
cosmosLog_csm.exe.log	6.09 GB	22143288	Microsoft Cosmos service manager log
free_impression.dat	2.60 GB	27644006	Impression data of yellow pages for Free users
Name (Small)	**Size**	**Lines**	**Description**
free_clickthroughs.dat	24 MB	285332	Yellow pages click through stream data
thirdpartycontent.log	40 MB	281519	Third party content stream data
eventstream.current	500 MB	1579920	Event streams on Cosmos
strace_jaccn.dat	80 MB	896490	NERSC application traces
LA-UR-EVENTS.csv	30 MB	433490	Comma separated LANL disk replacement data
messages.sdb	520 MB	5047341	/var/log/messages from CRAY
HALO_have2impression.log	360 MB	210034	Server side impression records of iPhone apps
LA-UR-NODE-NOZ.TXT	32 MB	1630479	Space separated LANL disk replacement data
searchevents.dat	90 MB	2035348	Yellow pages search event log
4046.xls	7 MB	24193	DNA Microarray data

3 Experimental Results

To evaluate the performance of our prototype system and to understand the trade-offs in setting the various parameters in the algorithm, we ran a number of experiments using 16 data sources. These sources are divided into two groups: six *large files*, each more than 1GB, and ten *smaller files*, each under 1GB. Table 1 lists the names of these data sources, the file sizes, the number of lines, and brief descriptions. We conducted our experiments on a 2.4GHz machine with 24 GBs of memory and two 64-bit quad-core Intel Xeon Processors running Linux version 2.6.18. Our system is single-threaded, so we effectively used only one of the eight available cores.

Benchmark data sources. We are interested in two kinds of performance measures: *time to learn a description* and *quality of the learned description*. The time to learn can be further broken down into two components: time to learn the initial description and time to learn with an initial description in hand.

The quality of the description can be measured in three ways: the *MDL score* [9] of the description, the *edit distance* [4] between the learned description and a "gold description" written by human expert, and the *accuracy* of the learned description. The MDL score provides a fully automated way to quantify both the precision and the compactness of a description, with smaller MDL scores corresponding to better descriptions. However, while MDL is useful, it is best seen as a proxy measure, since humans may prefer a description with a higher MDL score if that description better captures the human being's intuitions.

To address this concern, we use edit distance to measure how close the learned description is to something a human being might write. This metric counts the number of edits necessary to convert the learned description into a "gold description" written by a human being, where an edit can be either an insertion or deletion of a node in the description. More precisely, the distance measure is a *relative edit distance* score: $rel\_dist(D) = edit\_dist(D, D_{gold})/|D_{gold}|$, where

Table 2. Large data sources

Data	MDL	Dist	Accuracy	Learn time (secs)	`parse` time (secs)	PADS time (secs)	`wc` time (secs)	Blob time (secs)
cosmosLog_csm.exe.log	21301.34	0.805	100%	1040	1225	430	34	89
dalpiv.dat	45785.72	0.865	100%	4012	2196	767	82	278
free_impressions.dat	6062.39	0.89	100%	2701	4032	493	15	46
liberty	8790.85	0.722	100%	21144	20851	8036	175	677
redstorm	13837.73	0.707	100%	35548	24736	9791	191	719
vshkap2.log	10063.71	1.750	100%	23337	14651	2163	57	174

$|D|$ denotes the total number of nodes in D. We have empirically determined that a relative edit distance of less than 1 indicates a relatively good description. Of course, the edit distance measure may also be imperfect as there can be a number of different but equally "good" ways to craft a gold description. Nevertheless, we have found it a useful measure.

Finally, our system would not be very useful if the learned description was not correct. Therefore, we also use an accuracy measure, which reports the percentage of original data source that the learned description parses without errors.

Large data sources. Our first experiment learns a description for each of the six large data sources in the benchmark. We set the initial batch size N to be 2000 and the incremental batch size M to be 100. Table 2 reports the MDL and distance scores, the accuracy, and the total learning time. In addition, it report various times to parse the data. The `parse` time is the time it takes the algorithm's `parse` function to parse the source data using the learned description. The PADS time is the time it takes the generated PADS parser to parse the same data. To put these parsing times in perspective, we list the time to count the total number of lines using the Unix `wc -l` command and the time to parse the data using the simple PADS type `Pstring(:Peor:)`, a.k.a *blob*, which parses each line as a newline-terminated string. The result shows that the incremental learning algorithm can learn the format of a 30GB file in a few hours. Importantly, the learned descriptions are all correct with respect to their original raw data.

Scaling performance. In the next experiment, we evaluate how the algorithm scales with increasing data size by running the system on increasingly large fractions of each of the small data files, starting with 20% and ending with 100%. For a given data source, we empirically determined which values of the batch-size parameters N and M give the best result when learning the entire source, and then used those values for this experiment. Figure 10 plots the resulting total learning time versus the percentage of the data file used in learning. The graph shows the algorithm enjoys near linear scale-up for all sources except `4046.xls`, which flattens after 40% of data. The *BlobFinding* rule is the cause of this anomaly: learning the initial description takes a relatively long time, but after the algorithm sees the first 40% of the data, the *BlobFinding* rule simplifies the description to one that parses much more quickly.

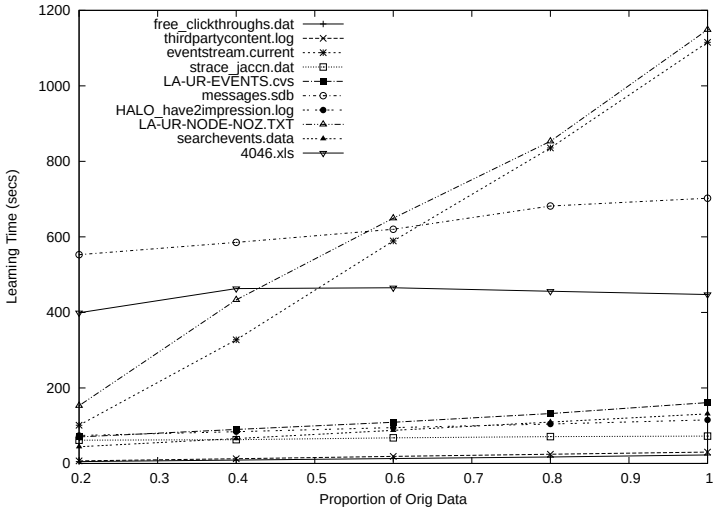

Fig. 10. Learning time vs. percentage of data sources

Initial and incremental batch size. We study the interplay of parameters N and M next. For each of the 10 small files, we repeatedly doubled N from 500 to 32000. For each N, we repeatedly quadrupled M from 25 to 6400. For each resulting pair of N and M, we ran the learning system on each data file and recorded the learning time, the MDL score and the relative distance score. All the learned descriptions parse the original data without error and therefore achieve 100% accuracy. We show only the results for `messages.sdb` in Table 3, while the remaining results are available on the web [1]. Table 3 represents a two-dimensional array, in which the N increases downward and the M increases to the right. Each table cell contains three numbers: the distance score, the MDL score and the total learning time in seconds. The number in parenthesis in the first column is the time to learn the initial batch in seconds, which is the same across all M's. As a baseline, we add a "Manual" row in which the time for the expert to produce an initial description is estimated to be 1 hour and the subsequent learning starts from that description. We highlight the best result in the table. The best description for `message.sdb` is learned with $N = 16000$ and $M = 400$ which are the parameters used for the scaling test of this source.

In general, as M goes up, the total learning time increases. With smaller batch sizes, the system updates descriptions more frequently, often simplifying them. These simplified descriptions parse more efficiently and hence require less time. When N is large, this phenomenon is not as prominent because the initial description learned from large initial batches is often good enough to cover most of the remaining data, and thus no incremental updates are needed.

Our main conclusion is that the end results of our algorithm are sensitive to the quality of the initial description, and that the quality of the initial de-

Table 3. N vs. M - messages.sdb

$N \backslash M$	25	100	400	1600	6400
Manual	0.62 8316.77	0.62 8355.71	0.62 8313.52	0.52 8297.47	0.52 8297.04
(3600)	337.44	438.29	292.88	295.68	292.05
500	0.67 8098.46	0.67 8098.46	0.67 8098.46	0.67 8098.46	0.67 8098.46
(2.13)	123.88	127.76	130.17	125.52	124.45
1000	1.10 9346.35	1.10 8443.28	1.24 8549.67	1.24 8544.63	1.24 8541.95
(5.75)	432.61	418.64	425.35	442.23	444.56
2000	2.48 10881.17	2.48 10881.17	2.48 10881.17	2.48 10881.17	2.48 10881.17
(6.55)	3935.54	3640.04	3983.46	3695.27	3643.84
4000	0.57 7936.66	0.57 7936.66	0.57 7936.66	0.57 7936.66	0.57 7936.66
(16.26)	868.20	881.52	885.64	910.99	925.19
8000	0.48 7932.71	0.48 7932.71	0.48 7932.71	0.48 7932.71	0.48 7932.71
(74.20)	245.05	242.79	249.90	244.78	248.62
16000	0.57 7995.88	0.48 7932.65	**0.48** **7932.65**	0.48 7932.65	0.48 7932.65
(585.03)	717.57	758.57	**696.82**	760.00	698.15

scription is dependent upon the initial batch of data. This is to be expected since our rewriting system is an incomplete, greedy local search, and therefore is sensitive to the initial candidate grammar it starts with. But given that the user can examine the intermediate descriptions during any iteration, necessary adjustments can be made to influence the final description.

To illustrate the quality of learned description and the difference between it and the gold description, we show the gold description and the best learned description of `messages.sdb` in Figure 11 and Figure 12. The learned description maintains a top-level structure almost identical to the gold description, except the gold description has slightly more refined details about the `message_t` type, which was represented by `Popt Struct_6113` and the blob at the end. The gold and learned descriptions for the other files are available on the web [1].

4 Related Work

There is a long history of research in *grammar induction*, the process of discovering grammars from example data. Vidal [13] and De La Higuera [10] both give surveys of research in the area. Readers are also referred to the extensive survey in this area from our previous paper [7].

The adaptations of our algorithm to incremental processing are partly inspired by traditional compiler error detection and correction techniques. In particular, the idea of using synchronizing tokens as a means for accumulating chunks of unknown/unparseable data has long been used in parsers from programming languages (see Appel's text [2] for an introduction to such techniques). This heuristic appears to work well in our domain of systems logs as these logs are usually structured around punctuation symbols (commas, semi-colons, ver-

```
Pstruct proc_id_t {
        '[';
        Puint32 id;
        ']';
};

Pstruct daemon_t {
        Pstring_SE (:"/[:\[]/":) name;
        Popt proc_id_t v_proc_id;
        ':';
};

Pstruct msg_body_t {
        daemon_t v_daemon_pri;
        Pwhite v_space;
        Pstring_SE(:Peor:) v_msg;
};

Punion message_t  {
        msg_body_t v_normal_msg;
        Pstring_SE(:Peor:) v_other_msg;
};

Precord Pstruct entry_t {
        Pdate  v_date;
        ' ';
        Ptime v_time;
        ' ';
        Pstring(:' ':) v_id;
        ' ';
        message_t v_message;
};
Psource Parray entries_t {
        entry_t[];
};
```

Fig. 11. Gold description of messages.sdb

```
Pstruct Struct_6113 {
        Pstring(:':':)  v_blob_5869;
        ':';
};

Precord Pstruct Struct_5671 {
        Pdate  v_date_1;
        ' ';
        Ptime  v_time_6;
        ' ';
        Pstring (:' ':) v_string_33;
        ' ';
        Popt Struct_6113 v_opt_6096;
        Pstring_SE(:Peor:)  v_blob_6095;
};

Psource Parray entries_t {
        Struct_5671[];
};
```

Fig. 12. Best learned description of messages.sdb

tical bars, parens, newlines, *etc.*) that act as field-terminators and hence work well as synchronizing tokens.

Other incremental algorithms for learning grammars from example data have been developed in the past. For example, Parekh and Honavar [12] have developed and proven correct an incremental interactive algorithm for inferring regular grammars from positive examples and membership queries. This algorithm works quite differently than ours: it operates over automata and it uses membership queries, which ours does not. More broadly speaking, Parekh and Honavar and many other related algorithms provide beautiful theoretical guarantees. In contrast, we have focused on implementation, empirical evaluation and scaling to support massive data sets.

Another place in which grammar induction is used is in information extraction from web pages. One example (amongst many others) is work by Chidlovskii *et al.* [5], which seeks to learn wrappers (*i.e.,* data extraction functions) by using a modified edit distance algorithm. Our algorithm also uses edit distance in its guts to measure similarity between chunks of data. However, the edit distance metric we use is just one element of a larger induction algorithm related to Arasu and Garcia-Molina's recursive descent algorithm [3]. Chidlovskii *et al.*'s

algorithm is also incremental – it integrates one new record of data at a time into a grammar. Our algorithm integrates batches of new data at a time. One reason we chose a batch-oriented approach is that processing data in batches helps disambiguate between various possibilities for both token definitions and tree structure. The tagged tree-structure of XML or HTML documents eliminates many of the ambiguities that appear in log files where the separators or tags are not known *a priori*. Our ad hoc data sets also appear different from the web-based data studied by Chidlovskii et al. in terms of their scale: our data is about a million times larger.

5 Conclusion

Ad hoc data sources are extremely difficult to manage because of their large size, evolving format, and lack of documentation. In this paper, we have presented the design, implementation and evaluation of a system for incrementally learning the structure of large or stream ad hoc data files. The output of the system is a data description in PADS language which can further generate end-to-end data processing tools. The system allows the users to get into the iterative learning process and make the description more accurate and readable.

References

1. LearnPADS^{++}. `http://www.padsproj.org/incremental-learning.html`.
2. Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
3. Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *SIGMOD*, pages 337–348, 2003.
4. Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.
5. Boris Chidlovskii, Jon Ragetli, and Maarten de Rijke. Wrapper generation via grammar induction. In *European Conference on Machine Learning*, Lecture Notes in Computer Science, pages 96–108. Springer Berlin, 2000.
6. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
7. 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.
8. Kathleen Fisher, David Walker, and Kenny Q. Zhu. LearnPADS: Automatic tool generation from ad hoc data. In *SIGMOD*, 2008.
9. P. D. Grünwald. *The Minimum Description Length Principle*. MIT Press, May 2007.
10. Colin De La Higuera. Current trends in grammatical inference. *Lecture Notes in Computer Science*, 1876:28–31, 2001.
11. PADS project. `http://www.padsproj.org/`, 2009.
12. Rajesh Parekh and Vasant Honavar. *Grammatical Interference: Learning Syntax from Sentences*, volume 1147, chapter An incremental interactive algorithm for regular grammar inference, pages 238–249. Springer Berlin, 1996.
13. Enrique Vidal. Grammatical inference: An introduction survey. In *ICGI*, pages 1–4, 1994.
14. Kenny Q. Zhu, Kathleen Fisher, and David Walker. Incremental learning of system log formats. In *ACM SOSP Workshop on the Analysis of System Logs*, 2009.