Incremental Learning of Ad Hoc Data Formats

ABSTRACT

An ad hoc data source is any semi-structured, non-standard data sources. The format of such data sources is often evolving and frequently lacking documentation. As a consequence, 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.

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 fragments of two ad hoc data sources. The first example comes from Microsoft's distributed computing infrastructure Cosmos. The second example is excerpted from the file `/var/log/messages` on a CRAY supercomputer.

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 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 XXX[1] project [17] 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. XXX 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.

An impediment to using XXX is the need to learn the description language and the effort required to write XXX descriptions, which can be time consuming if the data format is large and complex. Evolving data formats present an additional challenge because the descriptions must be updated. For example, when analyzing Cisco router logs, we noticed that the format changes after the first 2GB, and then again after another few GBs. The sudden change in format is the result of router firmware upgrades.

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 XXX 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 LEARNXXX [6, 7] that automatically infers a XXX 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, requires 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.

In this paper, we propose a new algorithm that automatically infers descriptions of large scale or continuous ad hoc data sources *incrementally*. 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 LEARNXXX 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 take the description output at the end of an iteration, make changes to it (*e.g.*, renaming the automatically generated variable names), and insert the revised description back into the loop.

[1]Name substitution for blind review.

Cosmos event streams:

```
EventName=CosmosPerfCounter_1,CurrentTimeStamp=2010-04-10T02:07:54.950Z,Machine=msradb001,
name=\\Cosmos\\MessageSent,instanceName=Total,counterType=rate,currentValue=1,minValue=1,
maxValue=1,average=0,rate=1.18217E-006,runningTotalCount=0,runningTotal=0,timeIntervalMs=
845901093,CsProcessId=NONE
```

Messages.sdb:

```
Jun  4 10:42:56 nid00004 sshd[5405]: Accepted publickey for root from 193.168.0.1 port 43484 ssh2
Jun  4 10:57:49 nid00003 syslog-ng[3504]: syslog-ng version 1.6.8 starting
Jun  4 10:57:53 nid00003 sshd[4069]: Server listening on :: port 22.
```

Figure 1: Example ad hoc data sources

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.

We presented an earlier, 6-page abstract of this material in an informal workshop [8] that was reprinted unchanged and unrefereed in a SIG newsletter. Unlike the earlier paper, this paper presents a re-engineered algorithm that is capable of processing data sources more than two orders of magnitude larger, while at the same time producing outputs of higher quality. This paper also gives detailed empirical analysis of the correctness and scalability of the system.

In the rest of the paper, we give a brief overview of XXX and the original LEARNXXX inference algorithm (Section 2). We then describe the new incremental inference algorithm (Section 3) and give a comprehensive experimental evaluation of the system (Section 4). Finally, we compare this system with some related work (Section 5) and conclude the paper (Section 6).

2. REVIEW OF XXX AND LEARNXXX

The data in Figure 2 is a fragment of a simple web server log. We use this format, which we call `wl`, to illustrate the principal features of the XXX data description language. The data is made up of a sequence of records, separated by newlines. Each record contains a number of fields delimited by white spaces. 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.

XXX uses a type-based metaphor to describe ad hoc data. Each XXX 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. XXX/C is the variant of XXX that uses C as its host language, and therefore is syntactically similar to C. When compiled, the generated data structures and parsing code are in C.

```
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 3: XXX/C description for the `wl` format

Figure 3 shows a XXX/C specification that describes the records in Figure 2. 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 fragments of the record. 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 XXX/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 as 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 character. 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

```
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 2: A Fragment of a Simple Web Server Log `wl`

```
<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>
  <request> ...  </request>
  <response> ... </response>
  <length> ... </length>
</entry_t>
```

Figure 4: XML translator output from one record of `wl` **format**

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 3, XXX 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 XXX supports *dependencies*: earlier portions of the data can determine how to parse later portions.

From a description like the one in Figure 3, the XXX compiler can automatically produce a suite of data processing tools such as an XML translator that converts the raw data into XML and statistical reporter that computes the various statistics of each types in the data source. Figure 4 presents the output of such an XML translator when applied to the first data records in Figure 2. Note that every data field has been tagged with a corresponding XXX data type. Figure 5 shows a snippet of the statistical report on a `wl` data source.

```
************************************************
<top> : struct entry_t
************************************************
good vals: 3000  bad vals: 0  pcnt-bad: 0.0

[Describing each field of <top>]
================================================
<top>.client : union client_t
================================================
good vals: 3000  bad vals: 0  pcnt-bad: 0.0
  min ip 1  max host 2
  distribution:
  val: ip (1)   count: 1704 pcnt-of-good: 56.800
  val: host (2) count: 1296 pcnt-of-good: 43.200
                . . . . . . . . . . . . . . . .
  SUMMING       count: 3000 pcnt-of-good: 100.000
================================================
<top>.length.len : uint32
================================================
good vals: 2651  bad vals: 0  pcnt-bad: 0.000
  min 35  max 37947 avg 3896.320
  distribution:
    val: 3082 count: 80  pcnt-of-good: 3.018
    val:  178 count: 55  pcnt-of-good: 2.075
    val:  170 count: 54  pcnt-of-good: 2.037
    val:  518 count: 50  pcnt-of-good: 1.886
    val:   43 count: 49  pcnt-of-good: 1.848
    val: 9372 count: 49  pcnt-of-good: 1.848
    val: 1277 count: 45  pcnt-of-good: 1.697
    val: 1425 count: 45  pcnt-of-good: 1.697
    val:  536 count: 43  pcnt-of-good: 1.622
    val: 1027 count: 42  pcnt-of-good: 1.584
              . . . . . . . . . . . . . . . .
    SUMMING   count: 512 pcnt-of-good: 19.313
```

Figure 5: Fragment of statistical report of a `wl` **data source**

LEARNXXX. The goal of the LEARNXXX format inference engine is to infer XXX descriptions like the one in Figure 3 from raw data. The algorithm is designed to be *sound* in the sense that the generated description describes *all* of the training data, including data that a human might deem to be erroneous. We might have automatically deemed very low occurrence data erroneous and pruned the corresponding elements of the description, but we felt such decisions were best left in human hands. The XXX statistical reporter (automatically generated from the learned description, along with other XXX tools) can help with this task, if a user chooses, by reporting the distribution of data that match each portion of the generated description. A full description of the LEARNXXX algorithm appears in an earlier paper [6]. We give only a brief summary here.

LEARNXXX 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 three-phase algorithm to produce a description. In the *tokenization* phase, LEARNXXX converts each input line into a sequences of tokens, where each token type is defined by a regular expression. Intuitively, these tokens correspond to XXX base types. For example, the sequences of tokens (shown in brackets) converted from the two data lines in Figure 2 are:

```
[ip] [ ] [-] [ ] [-] [ ] [[] [date] [:]
   [time] []] [ ] ["] [str] [ ] [path] [ ]
   [str] [/] [float] ["] [ ] [int] [ ] [int]

[host] [ ] [-] [ ] [str] [ ] [[] [date] [:]
   [time] []] [ ] ["] [str] [ ] [path] [ ]
   [str] [/] [float] ["] [ ] [int] [ ] [int]
```

Note that there are data sources in which the major unit of repetition is something other than a single line of text. Our system allows the use of other record delimiters through a regular expression. Automatic inference of such a delimiter does not seem worthwhile because it normally does not take the user more than a few seconds to look at the data source and determine the appropriate delimiter.

In the *structure discovery* phase, LEARNXXX 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 and recursively analyzes each of those partitions, constructing the corresponding description as it recurses. This phase terminates with a candidate description.

The LEARNXXX system decides the current top-level structure must be a **`Punion`** only if the frequency distribution does not look like a base type, a **`Pstruct`** or a **`Parray`**. Having decided for a **`Punion`**, the algorithm must partition the records of the data into buckets; each bucket corresponds to one branch of the new **`Punion`**. The original LEARNXXX partitioned the records based on the first token of each record: all records starting with **`Pint`** into one bucket, **`Pdate`** into a second, *etc.* Although this approach is very effective for some formats, for others it fails to capture useful structure, because two semantically very different records may incidentally have the same type of token as their first token. And conversely records with similar meaning may not share the same first token. To address this lack, we added a second mechanism for partitioning records in a **`Punion`** context: edit distance. With this approach, two records are placed in the same bucket if the number of token "edits" (insertions and deletions) required to morph the token sequence of one record into the other is less than a threshold percentage of the length of the sequences. This second approach is also very successful in many, but not all formats. As a result, the current LEARNXXX system provides both options; the user selects the desired mechanism using a command-line switch.

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 (*e.g.* converting regular unions into switched unions), and constraining the range of 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* [13], 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 complexity* 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 are extremely verbose. Such descriptions 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 data sources larger than the square root of the size of usable memory. In terms of adaptability, the algorithm only learns a description from a fixed amount of data. If the data changes over time, the algorithm cannot modify the existing description; it must start from scratch. This prevents the user from adapting descriptions to manage evolving data sources.

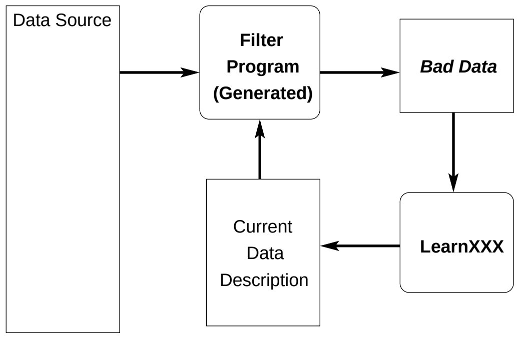

Figure 6: An Overview of the Incremental Learning Framework

3. MAIN ALGORITHM

To address these problems, we extended LEARNXXX to work incrementally. Figure 6 illustrates the overall process. 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. 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 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 LEARNXXX 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. 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 6 the *incremental learning step*. 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.

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

3.1 Preliminaries

Figure 7 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` 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 XXX 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)`

```
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(Al, Ar)
| ArrayA (A_elem, A_sep, A_term)
| OptionA A
| Opt A
| Learn [s]
```

Figure 7: Preliminary data structures used in incremental inference

are a simplification of XXX 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.

```
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 8: Pseudo-code for the incremental learning step

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 were data were missing. An invariant of the aggregation structure is that newly inserted Opt nodes always wrap either a BaseA or a SyncA node.

3.2 Incremental Learning Step

Figure 8 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 3.6 some strategies to alleviate this complexity.

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 the close 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 XXX description with Opt nodes translated to Poption types. In addition, it invokes the LEARNXXX 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.

```
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,"")}
```

Figure 9: Definition of `parse` function (excerpts)

3.3 Parsing

Our parser is a top-down recursive descent parser that performs error detection and recovery using synchronizing tokens. Figure 9 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 XXX dependent pair construct introduces. We use the symbol 'ϵ' to denote the empty environment.

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:

$$\begin{aligned} (e_1, g_1, s_1, c_1) &\geq (e_2, g_2, s_2, c_2) \text{ iff} \\ \frac{c_1}{e_1 + \frac{s_1}{\max((s_1+c_1)/(e_1+g_1),1)}} &\geq \frac{c_2}{e_2 + \frac{s_2}{\max((s_2+c_2)/(e_2+g_2),1)}} \end{aligned}$$

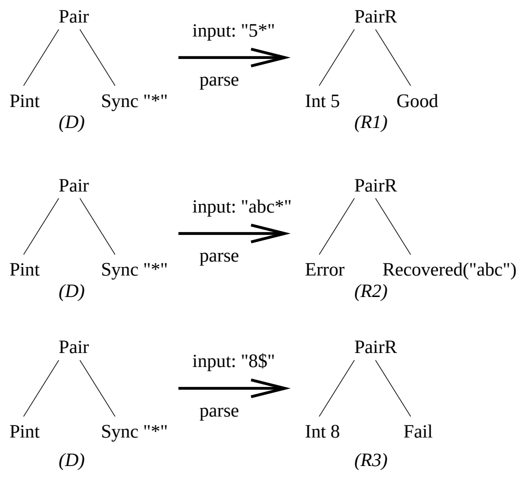

Figure 10: Result of parsing three input lines

3.4 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 "$\star$", and we are given the following three lines of new input: "5*" and "abc*" and "8$". Figure 10 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 $\star$ was missing. Figure 11 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.

3.5 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 is based on optimizing an information-theoretic Minimum Description Length (MDL) score [13], which is defined over descriptions `D` as:

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

where TC(`D`) is called the *type complexity* of `D` and ADC($x_1, \ldots, x_k$ | `D`) is called the *atomic data complexity*. The type complexity is a measure of the size of the abstract syntax of `D`.

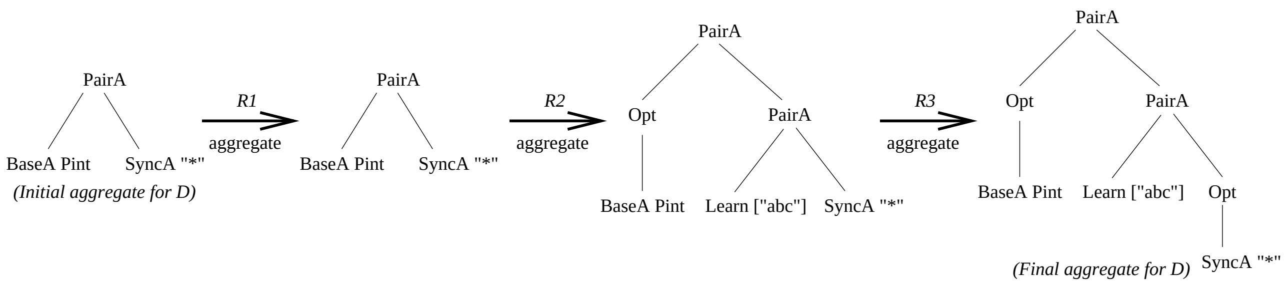

Figure 11: Aggregation of three parses

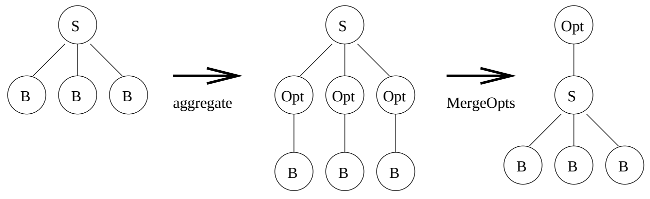

Figure 12: MergeOpts rewriting rule

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. This rule takes a given sub-description `D` and uses a heuristic to determine if the type complexity of `D` is too high relative to the amount of data it covers. The heuristic is that if the height of `D` is larger than `minBlobHeight` (which is empirically determined), and there is an identifiable constant string or pattern `re` that immediately follows `D`, then we rewrite `D` to `Pstring_SE(:re:)`. Description `Pstring_SE` is a XXX base type that is similar to `Pstring`, except it uses a stopping regular pattern as opposed to a stopping character to indicate termination. 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. 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 12 illustrates the effect of this rule. In the figure, S denotes a struct and B a base token.

3.6 Optimizations

The pseudo-code in Figure 8 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. To deal with this problem, we have implemented several optimizations to limit the number of parses and aggregates.

One 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 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 exist, it retains the best k non-perfect parses 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 [10].

To understand the impact that different values of k can have, consider $k = 1$, which causes the system to discard all erroneous parses except the one with the best parse metric. This strategy significantly reduces the cost of the parsing and aggregation operations, but we have found in practice that it usually results in sub-optimal overall parse structures. For example, suppose the description in Figure 13 is the current top-level description `D` when learning the `wl` format. Recall that constants in descriptions, such as `' '` and `'-'`, are used as sync tokens during parsing. Now, suppose we use `D` to parse the second record in Figure 2. `D` parses the data without error until it reaches the second dash in `D`. The description then expects a sync token '-', but instead sees a string `amnesty`. According to the semantics of sync tokens, given in Figure 9, there can be two possible parses:

```
(Recovered "kim [10/May/2009:18:38:35 ", (1, 26, 1))
```
or:
```
(Fail, (1, 0, 0))
```

In the first parse, the parser skips 26 characters before finding a

```
Precord Pstruct entry_t {
        Phostname       host;
   ' ';
   '-';  /* the first dash */
   ' ';
   '-';         /* the second dash */
   " ["; Pdate          date;
   ':';  Ptime          time;
   "] "; request_t      request;
   ' ';  Pint           response;
   ' ';  Pint           length;
};
```

Figure 13: Example description D

'-' token. In the second, the parser returns failure and consumes no characters from the input. According to our ranking function on parse metrics, the first parse is better and therefore the second one is discarded. Unfortunately, this strategy causes the parsing of the remaining types up to "] " to fail, thereby giving a bad overall parse. If we keep the second parse and continue with it, the next sync token " [" will be able to recover after skipping `kim` and succeed in parsing the rest of the types, hence giving a better overall parse. Therefore, in the implementation, we have chosen k to be small (for speed) but larger than one (for quality).

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.

4. 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.

We are interested in two kinds of performance measures:

1. *time to learn a description*
2. *quality of the learned description*

The time to learn can be further broken down into two components: time to learn the initial description, and the time to incrementally learn a description for the rest of the data.

The quality of the description can be measured in three ways: the *MDL score* [13] of the description, the *edit distance* [5] 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 *normalized edit distance* score:

$$normal\_dist(D) = \frac{edit\_dist(D, D_{gold})}{|D_{gold}|}$$

where $|D|$ denotes the total number of nodes in D. We have empirically determined that a normalized 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 this measure adds information to what we gain from the MDL score alone.

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

4.1 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 4.1 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 XXX time is the time it takes the generated XXX 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 XXX type `Pstring(:Peor:)`, 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.

4.2 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 14 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

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 applications
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

Table 1: The data sources

Data	MDL	Dist	Accuracy	Learn Time (secs)	`parse` time (secs)	`XXX` time (secs)	`wc` time (secs)	`Pstring` 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%	55548	24736	9791	191	719
vshkap2.log	10063.71	1.750	100%	23337	14651	2163	57	174

Table 2: Large data sources

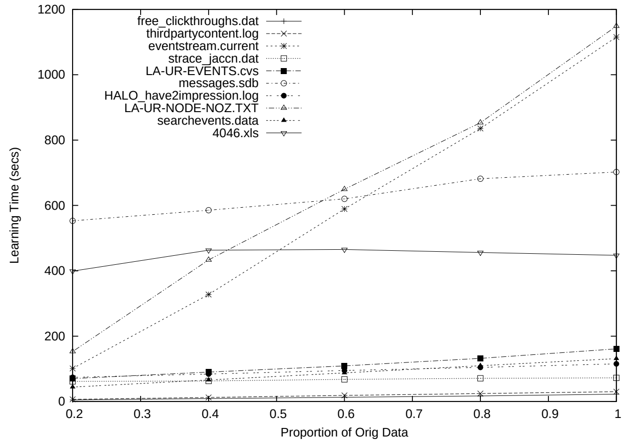

Figure 14: Learning time vs percentage of data sources

40% of the data, the *BlobFinding* rule simplifies the description to one that parses much more quickly and correctly parses the rest of the data.

4.3 Initial and incremental batch size

To understand the interplay of parameters N and M, we did the following experiment. 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 normalized distance score. All the learned descriptions parse the original data without error and therefore achieve 100% accuracy. Because of space constraints, we show only the results for `eventstream.current` and `messages.sdb` in Table 4.3 and Table 4.3, respectively. The results for the remaining files are available on the web (`http://202.120.38.146/incremental/incremental-learning.html`). Each table 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. A human expert was given only the first 500 records of the data and was asked to write a `XXX` description that correctly parses the 500 records. The timings in the manual row are the time it took the expert to produce the initial description (which was estimated to be an hour), and the time to learn the entire data source beginning with that description. We highlight the best result in each table. For example, 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 the rest of this section, we summarize some of the findings from analyzing these results.

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

$N \backslash M$	25	100	400	1600	6400
	3.37	3.00	3.00	3.00	3.00
Manual	10199.04	9763.39	9763.39	9763.39	9742.43
(3600)	442.18	827.66	897.30	910.00	868.01
	5.05	5.05	5.05	5.05	5.05
500	16119.06	15184.53	15184.53	15184.53	15184.53
(0.30)	927.84	1286.60	1117.60	1147.43	1118.18
	5.05	5.05	5.05	5.05	5.05
1000	16118.88	15184.35	15184.35	15184.35	15184.35
(0.89)	962.74	1088.59	1086.97	1113.33	1115.03
	5.05	5.05	5.05	**5.05**	5.05
2000	16118.52	15183.99	15183.99	**15183.99**	15183.99
(2.24)	934.40	1117.25	1120.37	**1100.55**	1114.62
	5.37	5.37	5.37	5.37	5.37
4000	16577.07	15642.54	15642.54	15642.54	15642.54
(9.34)	1393.08	1089.47	1198.06	1127.42	1091.39
	5.37	5.37	5.37	5.37	5.37
8000	16575.84	15641.31	15641.31	15641.31	15641.31
(36.63)	1404.10	1115.77	1115.64	1186.75	1250.11
	5.37	5.37	5.37	5.37	5.37
16000	16573.39	15638.87	15638.87	15638.87	15638.87
(164.38)	1545.75	1457.71	1443.14	1252.26	1331.52

Table 3: N vs. M - eventstream.current

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

Table 4: N vs. M - messages.sdb

from large initial batches is often good enough to cover most of the remaining data, and thus no incremental updates are needed.

After studying the results of varying the initial batch size N, the main conclusion that we draw is that the end results of our algorithm are sensitive to the quality of the initial description, and that the quality of the initial description 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. Hence, given this fact, an effective way to use the system is to have it infer an initial candidate grammar automatically, based on the first batch of data, to examine the initial candidate manually, make any necessary adjustments, and then to invoke the system on the rest of the data.

To determine the effects of the weight w in the MDL score on the quality of the learning descriptions, we pick four smaller sources and learn descriptions for them with w set at 1, 5, 10, 100 and 1000, while keeping the other parameters the same. The results in Figure 15 show that setting w to a very small or very large number either results in bad descriptions with high edit distance score, or timeouts at run time (description was too complex to parse in reasonable time). Therefore, we choose $w = 10$ for practicality.

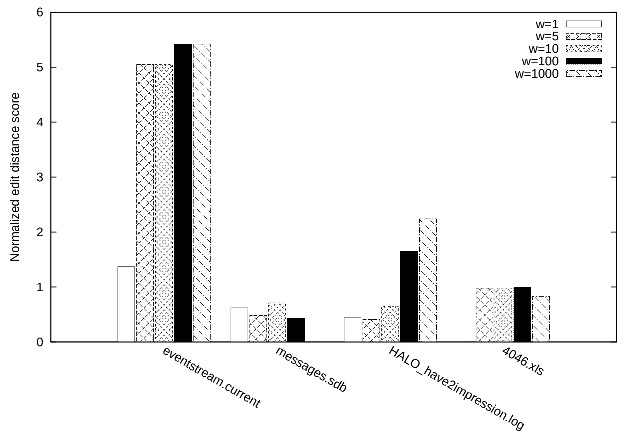

Figure 15: Effects of w on description quality

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 16 and Figure 17. 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 (`http://202.120.38.146/incremental/incremental-learning.html`).

5. RELATED WORK

There is a long history of research in *grammar induction*, the process of discovering grammars from example data. Vidal [21] and De La Higuera [14] both give surveys of research in the area. One way research in this area may be categorized is by analyzing what kinds of inputs the various algorithms require. Some algorithms require positive and negative examples to discover a grammar, some algorithms require manual labeling of example data, some algorithms require answers to various kinds of queries. Our system only requires positive examples (negative examples are not available in practice), does not ask users to answer queries, and does not require labeling.

Given that we work only with positive examples, there are still a number of possible approaches we could take. Our work on the core learning algorithm borrows key ideas from two different places. First, the structure discovery phase of our algorithm, which generates an initial candidate grammar from a set of examples, is a variant of Arasu and Garcia-Molina's algorithm for inferring the structure of web pages for the purpose of information extraction [2]. Second, our system makes use of various grammar rewriting algorithms that seek to optimize an information-theoretic Minimum Description Length (MDL) score. MDL rewriting has been used many times before; we recommend Grünwald's book [13] as a starting point for reading about this topic. Given this background of basic algorithms, the key contribution of the work is more applied: we have borrowed basic algorithms to get started, modified them

```
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[];
};
```

Figure 16: Gold description of messages.sdb

so they work effectively on an important, understudied domain (ad hoc system logs), proven it is possible to scale the algorithms up to the point that they may be applied to massive industrial data sets, and empirically evaluated the results.

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 [1] 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, vertical 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 has been developed in the past. For example, Parekh and Honavar [18] 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, many others) is work by Chidlovskii *et al.* [9], which seeks to learn wrappers (*i.e.,* data extraction functions) by using a modified edit distance algorithm. Our algorithm also uses edit distance

```
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[];
};
```

Figure 17: Best learned description of messages.sdb

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 Garcina-Molina's recursive descent algorithm mentioned above. Chidlovskii'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 in terms of their scale: Chidlovskii's algorithms take up to 30 seconds on up to 30 kilobytes of data; our algorithms take hundreds of times longer on data a million times larger.

Other related information-extraction efforts are those that attempt to identify tabular data either from free-form text [16, 19] or from web pages [15]. These approaches typically use hand-labeled examples to train machine learning systems to identify the tables. They then use heuristics specific to tabular data to extract the tuples contained within those tables.

Many researchers have studied the problem of learning a schema such as a DTD or XSchema from a collection of XML documents [3, 4, 11, 12]. At a high level, this task is similar to the kind of format inference we are attempting to do, but the details differ because, as mentioned above in reference to Chidlovskii's work, XML has different characteristics from ad hoc data: XML documents have a well-nested tree shape, with obvious delimiters defining the structure and the XML tags help with tokenization. As a result of these differences, XML inference algorithms cannot be used "off-the-shelf" for understanding the structure of ad hoc data. They must be modified, tuned and empirically evaluated on this new task. Having made this point, one of the more closely related XML schema inference systems is XTRACT [12]. It operates in three phases: generalization, factoring and MDL optimization. The first phase plays a role similar to our structure discovery phase in that it generates a collection of candidate structures from a series of XML examples. This generalization phase searches for patterns in XML data; it is tuned using the authors' knowledge of common DTD structures. Factoring decreases the size of generated candidate DTDs; some of the factoring rules resemble our rewriting rules. Finally, they tackle the MDL optimization problem by mapping the problem into an instance of the NP-complete Facility Location Problem, which they solve using a quadratic approximation algorithm. Our MDL-guided rewriting problem considers a more general set of rewriting rules and hence we cannot reuse this particular technique.

Finally, Potter's Wheel [20] is a system that attempts to help users find and purge errors from relational data sources. It does so through the use of a spread-sheet style interface, but in the background, a grammar inference algorithm infers the structure of the input data, which may be "ad hoc," somewhat like ours. This inference algorithm operates by enumerating all possible sequences of base types that appear in the training data. Like our work, Potter's Wheel is interested in large-scale data processing problems and is designed to process data incrementally and interactively. Since Potter's Wheel is aimed at processing relational data, they only infer `struct` types as opposed to enumerations, arrays, switches or unions.

6. CONCLUSION

Ad hoc data files such as systems logs 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 learning the structure of large industrial ad hoc data files. The system not only produces documentation for the files in the form of a XXX description, but it can also be used to generate end-to-end data processing tools such as a statistical analysis or an XML translator. We have shown that the system can scale to the point where it can process and learn the structure of logs on the order of up to 30GB or more in a few hours.

7. REFERENCES

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *SIGMOD*, pages 337–348, 2003.

[3] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

[4] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009, 2007.

[5] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[6] For blind review.

[7] For blind review.

[8] For blind review.

[9] 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.

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

[11] Henning Fernau. Learning XML grammars. In *MLDM*, pages 73–87, 2001.

[12] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *SIGMOD*, pages 165–176, 2000.

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

[14] Colin De La Higuera. Current trends in grammatical inference. *Lecture Notes in Computer Science*, 1876:28–31, 2001.

[15] Kristina Lerman, Lise Getoor, Steven Minton, and Craig Knoblock. Using the structure of web sites for automatic segmentation of tables. In *SIGMOD*, pages 119–130, New York, NY, USA, 2004.

[16] Hwee Tou Ng, Chung Yong Lim, and Jessica Li Teng Koo. Learning to recognize tables in free text. In *ACL*, pages 443–450, Morristown, NJ, USA, 1999.

[17] For blind review.

[18] 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.

[19] David Pinto, Andrew McCallum, Xing Wei, and W. Bruce Croft. Table extraction using conditional random fields. In *SIGIR*, pages 235–242, New York, NY, USA, 2003.

[20] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381 – 390, 2001.

[21] Enrique Vidal. Grammatical inference: An introduction survey. In *ICGI*, pages 1–4, 1994.