PADS: An End-to-end System for Processing Ad Hoc Data

Mark Daly
Princeton University
mdaly@princeton.edu

Yitzhak Mandelbaum and David Walker
Princeton University
yitzhakm@cs.princeton.edu
dpw@cs.princeton.edu

Mary Fernández and Kathleen Fisher
AT&T Labs Research
mff@research.att.com
kfisher@research.att.com

1. INTRODUCTION

Although enormous amounts of data exist in "well-behaved" formats such as relational tables and XML, massive amounts also exist in non-standard or *ad hoc* data formats. Ad hoc formats arise in diverse domains such as telecommunications, bioinformations, and finance. Ad hoc data comes in many forms: ASCII, binary, EBCDIC, and mixed formats. It can be fixed-width, fixed-column, variable-width, or even tree-structured. It is often quite large, including some data sources that generate over a gigabit per second [3]. It frequently comes with incomplete or out-of-date documentation, and there are almost always errors in the data. Sometimes these errors are the most interesting aspect of the data, *e.g.*, in log files where errors indicate that something is going wrong in the associated system.

The lack of standard tools for processing ad hoc data forces analysts to roll their own tools, leading to scenarios such as the following. An analyst receives a new ad hoc data source containing potentially interesting information and a list of pressing questions about that data. Could she please provide the answers to the questions as quickly as possible? The accompanying documentation is outdated and incomplete, so she first has to experiment with the data to discover its structure. Eventually, she understands the data well enough to hand-code a parser, usually in C or PERL. Pressed for time, she interleaves code to compute the answers to the supplied questions with the parser. As soon as the answers are computed, she gets a new data source and a new set of questions to answer.

Through her heroic efforts, the data analyst answered the necessary questions, but the approach is deficient in many respects. The analyst's hard-won understanding of the data is embedded in a hand-written parser, where it is difficult for others to benefit from her understanding. The parser is likely to be brittle with respect to changes in the input sources. Consider, for example, how tricky it is to figure out which \$3's should be \$4's in a PERL parser when a new column appears in the data. Errors in the data also pose a significant challenge in hand-coded parsers. If the data analyst thoroughly checks for errors, then the error checking code dominates the parser, making it even more difficult to understand the semantics of the data format. If she is not thorough, then erroneous data can escape undetected, potentially corrupting downstream data processing. Finally, in answering the specified questions, the analyst had to code *how to compute* the questions rather than expressing the queries in a declarative fashion. Many of these pitfalls can be avoided with careful design and sufficient time, but an analyst rarely has such luxuries. However, with the appropriate tool support, many aspects of this process can be greatly simplified.

The PADS system [5] allows analysts to describe ad hoc data sources declaratively. The descriptions take the form of types, based on a dependent type theory [6]. PADS base types describe simple objects such as strings, numbers, dates, and IP addresses. Records and arrays specify sequences of elements in a data source, and unions and enums specify alternatives. Any of these structured types may be parameterized, and users may write arbitrary semantic constraints as well. The PADS language is both expressive and concise. For example, 92 pages of the OPRA standard for options-market transactions is captured by a 450-line PADS description.

The PADS compiler produces a customizable library for parsing a given ad hoc data source. A suite of tools built around this library includes statistical data-profiling tools, such as histograms [**?**], accummulators and clustering algorithms [8]. Also included is an instance of the Galax query engine [1] that permits ad hoc sources described in PADS to be viewed as XML and to be queried with XQuery [4]. Lastly, an interactive front-end helps users produce PADS descriptions and invoke tools without having to learn the details of the PADS language or tool interfaces. An open-source implementation of PADS is available for download [2].

2. USING PADS

In our demonstration, we will present the following scenario, in which an AT&T data analyst interactively creates a PADS description for a new data source, uses PADS tools to learn about the distribution of values and errors in her data, and writes and executes simple queries to perform basic analysis tasks. We will also have available other PADS applications from other application domains.

Our analyst's task is to process *provisioning* data. In the telecommunications industry, provisioning refers to the complex process of converting an order for phone service into the actual service. In practice, AT&T's Sirius project discovers provisioning problems proactively by compiling weekly summaries of the state of phone service orders. These summaries, which are stored in flat ASCII text files, can contain more than 2.2GB of data per week. Figure 1 contains sample Sirius data.

Provisioning summaries store the processing date and one record per order. Each order record contains a header followed by a nested sequence of events. The header has 13 pipe separated fields: the order number, AT&T's internal order number, the order version, four different telephone numbers associated with the order, the zip code, a billing identifier, the order type, a measure of the complexity of

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

```
0|15/Oct/2004:18:46:51
9152|9152|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|10|16/Oct/2004:10:02:10
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|LOC_CRTE|1001476800|LOC_OS_10|17/Oct/2004:08:14:21
```

Figure 1: Tiny example of Sirius provisioning data.

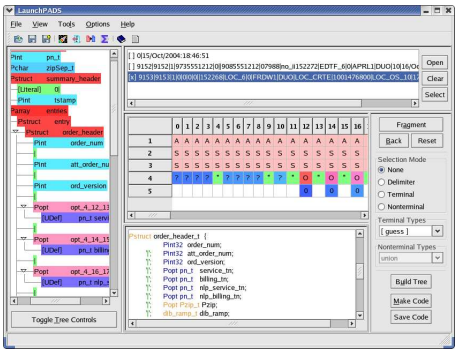

Figure 2: LAUNCHPADS Interactive User Interface.

the order, an unused field, and the source of the order data. Many of these fields are optional, in which case nothing appears between the pipe characters. The billing identifier may not be available at the time of processing, in which case the system generates a unique identifier prefixed with the string "no_ii" to indicate the number was generated. The event sequence represents the subset of 400 possible states a service order goes through; it is represented as a newline terminated, pipe separated list of state, timestamp pairs. It may be apparent from this description that English is a poor language for describing data formats!

The analyst's first task is to write a parser for the Sirius data format. Like many ad hoc data sources, Sirius data may contain unexpected or corrupted values, so the parser must handle errors robustly to avoid corrupting the results of analyses. With PADS, the analyst writes a declarative data description of the physical layout of her data. If the analyst is new to PADS, she can use the LAUNCHPADS interactive interface shown in Figure 2 to help her create a PADS description. She begins by loading her sample data into the *dataview* (top-right frame) and then selects a fragment of data to describe in the *gridview* (middle frame). In the gridview, the analyst iteratively refines the description of the selected data. In this example, she has selected the header part of an order record and is defining its composite structure, which includes three phone-number fields. This refinement step terminates when the analyst has associated a base type, such as string, phone number, date, *etc.*, with every value in the sample data. Once all selected values have an associated base type, LAUNCHPADS generates the *treeview* (left-hand frame). The treeview depicts the abstract syntax of a PADS description. In this view, the analyst can refine the description by creating, removing, and renaming the generated types. She may also add semantic constraints that specify relations between one part of the data and another, for example that one field in a type is some function of another field in a different type.

When the analyst is satisfied with the description in the treeview, she can test her description on a larger fragment of sample data. To do this, LAUNCHPADS generates syntactically correct PADS code, which is shown in Figure 4 and invokes the PADS compiler to produce a parsing library from the generated description. Description-independent tools are linked with the description-dependent library and made available to the analyst through menus in the LAUNCHPADS interface.

The analyst can test her description by applying the accummulator tool to a larger sample of data. For each type in a PADS description, accumulators report the number of good values, the number of bad values, and the distribution of legal values. In the LAUNCHPADS interface, records identified by the accumulator as containing errors are displayed in the data view. The analyst can then determine whether the errors are due to genuine errors in the data or due to incomplete or out-of-date documentation, in which case she can refine the description to improve its coverage.

This phase helps the analyst learn the layout and the meaning of the data, determine the completeness of the format's documentation, identify different representations for "data not available", and learn the distribution of values for particular fields, *etc.* When finished with this phase, the analyst may be ready to ask some basic queries such as "Select all orders starting within a certain time window," and "What is the average time required to go from a particular event state to another particular event state". Such queries are useful for rapid information discovery and for vetting errors and anomalies in data before the data proceeds to a down-stream process or is loaded into a database.

With PADS, the analyst uses XQuery to query her ad hoc data source. Because XQuery is designed for semi-structured data, its expressiveness matches ad hoc data sources well. For example, the analyst can write the expression below to produce all orders that started in October, 2004.

```
$pads/Psource/orders/elt[events/elt[1]
  [tstamp/rep >= xs:date("2004-10-01")
and tstamp/rep < xs:date("2004-11-01")]]
```

Existing PADS tools may not solve all the analysts problems, in which case, she may write her own PADS applications that call directly the PADS-generated parsing or tool libraries. Most importantly, her effort has produced a reusable description that she can share with other analysts. The fact that useful software artifacts are generated from the descriptions provides strong incentive for keeping the descriptions current.

3. ARCHITECTURE

The PADS system, depicted in Figure 3, consists of its description language, compiler, run-time system, and pre-defined tool suite. From a description, the compiler generates a library of description-dependent parsing functions. The generated library is linked with a core run-time library and description-independent tool programs. Currently, the statistical profiling tool provides accumulator functions and functions that employ randomized and approximate techniques to create histogram, wavelet [7], and quantile [8] summaries. The XML tool produces a canonical XML view of a PADS source [4] and implements the data model required by the Galax XQuery engine (not shown here). The format template allows the user to pretty print the data into a delimited format suitable for loading in a relational database. The format program allows users to override how elements of each type are displayed and to omit certain fields entirely from the data source. Lastly, a user may write a custom application in C to implement their own analyses.

We describe a few key language features to illustrate the language's expressiveness and completeness. The language provides a type-based model: basic types specify atomic data, while structured

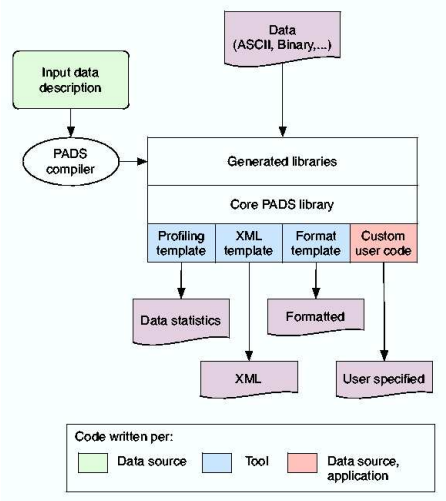

Figure 3: PADS Architecture

types describe compound data built from simpler pieces. Figure 4 contains the PADS description for the Sirius data format. Types are declared before they are used, so the type that describes the entire data source (`summary_t`) appears last in the description.

The PADS library provides a large collection of useful base types such as 8-bit signed integers, 32-bit unsigned integers, IP addresses, dates, and strings, which may be physically coded in *e.g.* ASCII, EBCDIC, or binary. To describe more complex data, PADS provides a collection of structured types loosely based on C's type structure. A **`Pstruct`** describes an ordered sequence of data with unrelated types. In Figure 4, the type declaration for the **`Pstruct`** `order_t` (Lines 35–38) contains an order header followed by the literal character **`'|'`**, followed by an event sequence. PADS supports character, string, and regular expression literals. A **`Punion`** describes alternatives in the data format (Lines 9–12), and a **`Popt`** type specifies optional data (Lines 17–21). A **`Parray`** describes varying-length sequences of data all with the same type. The type on Lines 32–34 contains the sequence of order events and indicates that each element in the sequence has type `event_t`. It also specifies that the elements are separated by vertical bars, and that the sequence is terminated by an end-of-record marker. PADS provides a rich collection of array-termination conditions: reaching a maximum size, finding a terminating literal, or satisfying a predicate.

From a description, the PADS compiler generates a C library for parsing and manipulating the associated data source. From each type in a PADS description, the compiler generates (1) an in-memory representation of the type, (2) parsing and printing functions, and (3) a parse descriptor, which records the state of the parse, the number of detected errors, and the code and location of the first error detected in a value of that type. Because a distinct parsing function is generated for each type in a PADS description, PADS supports multiple-entry point parsing, which accommodates efficient processing of very large-scale data [5]. Parse descriptors enable *error-aware* processing of a data source. Depending upon the nature of the errors and the desired application, users can take the appropriate action: halt the program, discard parts of the data, or repair the errors. This flexibility makes it possible to continue processing of very large sources even when errors are encountered.

The PADS system solves important data-management tasks: it supports declarative description of ad hoc data formats, its descriptions serve as living documentation, and it permits exploration of

```
 1. Precord Pstruct summary_header_t {
 2.  "0|";
 3.  Punixtime tstamp;
 4. };
 5. Pstruct no_ramp_t {
 6.  "no_ii";
 7.  Puint64 id;
 8. };
 9. Punion dib_ramp_t {
10.   Pint64     ramp;
11.   no_ramp_t  genRamp;
12. };
13. Pstruct order_header_t {
14.         Puint32             order_num;
15.   '|';  Puint32             att_order_num;
16.   '|';  Puint32             ord_version;
17.   '|';  Popt pn_t           service_tn;
18.   '|';  Popt pn_t           billing_tn;
19.   '|';  Popt pn_t           nlp_service_tn;
20.   '|';  Popt pn_t           nlp_billing_tn;
21.   '|';  Popt Pzip           zip_code;
22.   '|';  dib_ramp_t          ramp;
23.   '|';  Pstring(:'|':)      order_type;
24.   '|';  Puint32             order_details;
25.   '|';  Pstring(:'|':)      unused;
26.   '|';  Pstring(:'|':)      stream;
27. };
28. Pstruct event_t {
29.         Pstring(:'|':)    state;
30.   '|'; Punixtime          tstamp;
31. };
32. Parray event_seq_t {
33.   event_t[] : Psep('|') && Pterm(Peor);
34. };
35. Precord Pstruct order_t {
36.         order_header_t  order_header;
37.   '|'; event_seq_t      events;
38. };
39. Parray orders_t {
40.   order_t[];
41. };
42. Psource Pstruct summary_t{
43.   summary_header_t  summary_header;
44.   orders_t          orders;
45. };
```

Figure 4: PADS description for Sirius provisioning data.

ad hoc data and vetting of erroneous data using a standard query language. The resulting PADS descriptions and queries are robust to changes that may occur in the data format, making it possible for more than one person to profitably use and understand a PADS description and related queries.

Additional Authors. Robert Gruber (`gruber@google.com`), Google, while at AT&T Labs. Xuan Zheng (`xuanzh@eecs.umich.edu`), Univ. of Michigan, supported by AT&T Labs and NSF DMS 0354600.

4. REFERENCES

[1] Galax user manual. `http://www.galaxquery.org`.

[2] PADS user manual. `http://www.padsproj.org/`.

[3] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*. ACM, 2002.

[4] M. Fernández, K. Fisher, R. Gruber, and Y. Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. In *PLAN-X*, 2006.

[5] K. Fisher and R. Gruber. PADS: A domain-specifi c language for processing ad hoc data. In *PLDI*, 2005.

[6] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL*, 2006.

[7] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *STOC*, 2002.

[8] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, 2002.