PADX[1] : Querying Large-scale Ad Hoc Data with XQuery

Mary Fernández
Kathleen Fisher
AT& Labs Research
{mff,kfisher}@research.att.com

Robert Gruber [*]
Google
gruber@google.com

Yitzhak Mandelbaum
Princeton University
yitzhakm@cs.princeton.edu

Name & Use	Representation
Web server logs (CLF): Measure web workloads	Fixed-column ASCII records
AT&T provisioning data: Monitor service activation	Variable-width ASCII records
Call detail: Fraud detection	Fixed-width binary records
AT&T billing data: Monitor billing process	Various Cobol data formats
Netflow: Monitor network performance	Data-dependent number of fixed-width binary records
Newick: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierachy
Gene Ontology: Gene-gene correlations	Variable-width ASCII records in DAG-shaped hierarchy
CPT codes: Medical diagnoses	Floating point numbers
SnowMed: Medical clinic notes	keyword tags

Figure 1. Selected ad hoc data sources.

Abstract

1. Introduction

Although massive amounts of data exist in "well-behaved" formats such as XML or relational databases, enormous amounts of data exist in non-standard or *ad hoc* data formats as well. Figure 1 gives some sense of the range and pervasiveness of such data. 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 [2]. It often comes with incomplete and/or out-of-date documentation, and there are often errors in the data. Indeed, sometimes the errors are the most intersting parts, because such errors can indicate where something is going wrong in a monitored system.

[*] Work carried out while at AT&T Labs Research.

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, preferably last week? The accompanying documentation is incomplete and out of date, so she first has to experiment with the data to discover its structure.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 or 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 ended up 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 might be to figure out which $3's should be $4's in a PERL parser when a new column is added to the data. Errors in the data also pose a significant challenge for hand-coded parsers. If the data analyst is thorough in checking for errors, then the error checking code dominates the parser, making it even more difficult to understand the semantics of the data. If she is not, then erroneous data can escape undetected, potentially (silently!) corrupting downstream processing. Finally, during the initial data exploration and in answering the specified questions, the analyst had to code *how to compute* the questions rather than being able to express the queries in a declarative fashion. Of course, many of these pitfalls can be avoided with careful design and sufficient time, but such luxuries might not be available to the analyst. However, with the appropriate tool support, many aspects of this process can be greatly simplified.

We had tools designed to address aspects of the analyst's problem in isolation: PADS [**?**] and Galax [3]. PADS allows users to describe ad hoc data sources declaratively and then generates error-aware parsers and tools for manipulating the sources, including statistical profiling tools. Galax supports declarative querying of semi-structured data sources via XQuery. Such querying would allow the analyst to explore the data in the first place, produce the answers to her questions in the second, and potentially convert the data into XML format to faciliate further downstream processing with the vast array of XML-based tools. The challenge we faced was to integrate these tools in a way that could scale to the large ad hoc data sources we have seen in practice.

****Rest of this section still in outline form.**** To achieve this integration, we had to evolve PADS and Galax in parallel, modifying the implementation of Galax to support an abstract data model and then augmenting PADS with the ability to generate instances of this data model.

Systems were developed in parallel. Solving the above problem required some additional engineering: implementation of Galax's abstract data model on top of PADS parsing-read functions. Hence, we are going to talk about both systems followed by their synthesis.

Architecture of PADX. Supports both non-materialized and virtual querying of PADS data. Example: dot query. User gets to choose whether to query materialized or virtual XML data. Appropriate model depends on query work load and other processes in work flow. We support both (!)

Potential benefits: leverage speed of XQuery processor over native XML document. Potential costs: same as having using a materialized view of a database. "Staleness" of XML, multiple copies, extra time to convert to XML. Work-load/cost-based optimization problem. Tradeoffs between materializing XML document and querying it multiple times. Our architecture appropriate for certain problems. Data gets regenerated every week or so. Already Gigabytes of data. Could use PADS and Galax separately to same effect.

In the rest of this section, we discuss an example scenario in detail to illustrate the variety of data management tasks faced by AT&T data analysts.

1.1 Example Scenario

In the telecommunications industry, the term *provisioning* refers to the process of converting an order for phone service into the actual service. This process is complex, involving many interactions with other companies. To discover potential problems proactively, the Sirius project tracks AT&T's provisioning process by compiling weekly summaries of the state of certain types of phone service orders. These summaries, which are stored in flat ASCII text files, can contain more than 2.2GB of data per week.

These ASCII summaries store the summary 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 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, and prefixes this value with the string "no_ii" to indicate the number was generated. The event sequence represents the various states a service order goes through; it is represented as a new-line terminated, pipe separated list of state, timestamp pairs. There are over 400 distinct states that an order may go through during provisioning. It may be apparent from this description that English is a poor language for describing data formats!

The data analyst's first task is to write a parser for the Sirius data format. Like many ad hoc data sources, Sirius data can contain unexpected or corrupted values, so the parser must handle errors robustly to avoid corrupting the results of analyses. Today, parsers for ad hoc formats are often hand-crafted in PERL or C. Unfortunately, writing parsers this way is tedious and error prone, complicated by the lack of documentation, convoluted encodings designed to save space, and the need to produce efficient code. Moreover, the analyst's hard-won understanding of the data ends up embedded in parsing code, making long-term maintenance difficult for the original writers and sharing the knowledge with others nearly impossible.

With PADS, the analyst writes a declarative data description of the physical layout of their data. The language also permits analysts to describe expected semantic properties of their data so that deviations can be flagged as errors. The intent is to allow analysts to capture in a PADS description all that they know about a given data source.

Figure 3 gives the PADS description for the Sirius data format. In PADS descriptions, types are declared before they are used, so the type that describes the entire data source, `summary`, appears at the bottom of the description. In the next section, we use this example to describe several features of the PADS language. Here, we simply note that the data analyst writes this description, and the PADS compiler produces customizable C libraries and tools for parsing, manipulating, and summarizing the data. The fact that useful software artifacts are generated from PADS descriptions provides strong incentive for keeping the descriptions current, allowing them to serve as living documentation.

Analysts working with ad hoc data also often query their data. Questions posed by the Sirius analyst include "Select all orders starting within a certain time window," "Count the number of orders going through a particular state," 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 it proceeds to a down-stream process or is loaded into a database system.

With PADX, the synthesis of PADS and Galax, the analyst writes declarative XQuery expressions to query his ad hoc data source. Because XQuery is designed to manipulate semi-structured data, its expressiveness matches PADS data sources well. XQuery is a Turing-complete language and therefore powerful enough to express all the questions above. For example, Figure 4 contains an XQuery expression that produces all orders that started in October, 2004. In Section 4, we use this example to describe several features of XQuery and to illustrate why XQuery is an appropriate query language for ad hoc data. In particular, XQuery queries may be statically typed, which helps detects common errors at compile time. For example, static typing would raise an error if the path expression in Figure 4 referred to `ordesr` instead of `orders` or if the analyst erroneoulsy compared the timestamp `tstamp` to a string.

2. Using PADS to Access Ad Hoc Data [Kathleen]

In this section, we give a brief overview of PADS. More information may be found in [**?**, 1]. A PADS specification describes the physical layout and semantic properties of an ad hoc data source. The language provides a type-based model: basic types specify atomic data such as integers, strings, dates, *etc.*, while structured types describe compound data built from simpler pieces. The PADS library provides a collection of useful base types. Examples include 8-bit signed integers (`Pint8`), 32-bit unsigned integers (`Puint32`), IP addresses (`Pip`), dates (`Pdate`), and strings (`Pstring`). By themselves, these base types do not provide sufficient information for parsing because they do not indicate how the data is coded, *i.e.*, in ASCII, EBCDIC, or binary. To resolve this ambiguity, PADS uses the *ambient* coding. By default, the ambient coding is ASCII, but programmers can customize it as appropriate.

To describe more complex data, PADS provides a collection of structured types loosely based on C's type structure. In particular, PADS has **`Pstruct`**s, **`Punion`**s, and **`Parray`**s to describe record-like structures, alternatives, and sequences, respectively. **`Penum`**s describe a fixed collection of literals, while **`Popt`**s provide convenient syntax for optional data. Each of these types can have an associated predicate that indicates whether a value calculated from the physical specification is indeed a legal value for the type. For example, a predicate might require that two fields of a **`Pstruct`** are related or that the elements of a sequence are in increasing order. Programmers can specify such predicates using PADS expressions and functions, written using a C-like syntax. Finally, PADS

```
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 2. Tiny example of Sirius provisioning data.

```
Precord Pstruct summary_header_t {
  "0|";
  Punixtime tstamp;
};

Pstruct no_ramp_t {
  "no_ii";
  Puint64 id;
};

Punion dib_ramp_t {
  Pint64     ramp;
  no_ramp_t  genRamp;
};

Pstruct order_header_t {
       Puint32            order_num;
 '|';  Puint32            att_order_num;
 '|';  Puint32            ord_version;
 '|';  Popt pn_t          service_tn;
 '|';  Popt pn_t          billing_tn;
 '|';  Popt pn_t          nlp_service_tn;
 '|';  Popt pn_t          nlp_billing_tn;
 '|';  Popt Pzip          zip_code;
 '|';  dib_ramp_t         ramp;
 '|';  Pstring(:'|':)     order_type;
 '|';  Puint32            order_details;
 '|';  Pstring(:'|':)     unused;
 '|';  Pstring(:'|':)     stream;
};

Pstruct event_t {
       Pstring(:'|':)    state;
  '|'; Punixtime         tstamp;
};

Parray event_seq_t {
  event_t[] : Psep('|') && Pterm(Peor);
};

Precord Pstruct order_t {
       order_header_t  order_header;
  '|'; event_seq_t     events;
};

Parray orders_t {
  order_t[];
};

Psource Pstruct summary{
  summary_header_t  summary_header;
  orders_t          orders;
};
```

Figure 3. PADS description for Sirius provisioning data.

```
(: Return orders started in October 2004 :)
$pads/Psource/orders/elt[events/elt[1]
  [tstamp >= xs:dateTime("2004-10-01:00:00:00")
and tstamp < xs:dateTime("2004-11-01:00:00:00")]]
```

Figure 4. Query applied to Sirius provisioning data.

`Ptypedef`s can be used to define new types that add further constraints to existing types.

PADS types can be parameterized by values. This mechanism serves both to reduce the number of base types and to permit the format and properties of later portions of the data to depend upon earlier portions. For example, the base type `Puint16_FW(:3:)` specifies an unsigned two byte integer physically represented by exactly three characters, while the type `Pstring(:' ':)` describes a string terminated by a space. Parameters can be used with compound types to specify the size of an array or which branch of a union should be taken.

We will use the example in Figure 3 to illustrate various features of the PADS language. In PADS descriptions, types are declared before they are used, so the type that describes the totality of the data source appears at the bottom of the description.

`Pstruct`s describe fixed sequences of data with unrelated types. In the Sirius example, the type declaration for `order__t` illustrates a simple **`Pstruct`**. It starts with an `order_header` followed by a timestamp `tstamp`, separated by the literal character `'|'`. PADS supports character, string, and regular expression literals, which are interpreted with the ambient character encoding.

2.1 outline

Focus on expressiveness of data description language.

Generated library : rep, pd, and per-type parsing functions and other type specific tools (but we're talking about those here).

Error-aware data processing.

Many of PADS features are not described here b/c they are not germane to understand this work. See PLDI and POPL papers.

3. Using XQuery and Galax

XML [**?**] is a flexible format that can represent many classes of data: structured documents with large fragments of marked-up text; homogeneous records such as those in relational databases; and heterogeneous records with varied structure and content such as those in ad hoc data sources. XML makes it possible for applications to handle all these classes of data simultaneously and to exchange such data in a simple, extensible, and standard format. This flexibility has made XML the "lingua franca" of data interoperability.

XQuery [4] is a typed, functional query language for XML that supports user-defined functions and modules for structuring large queries. Its type system is based on XML Schema. XQuery contains XPath 2.0 [**?**] as a proper sublanguage, which supports navigation, selection, and extraction of fragments of XML documents. XQuery also includes expressions to construct new XML values, and to integrate or join values from multiple documents. Unique among industry standards, XQuery also has a formal semantics, which makes it particularly interesting to database researchers.

As XQuery was designed for querying semi-structured XML data, it is a natural choice for querying semi-structured ad hoc data. XQuery's user syntax easily handles irregularly sturctured data. For example, the path expressions in Figure 4 are well-defined for orders with zero or more events, and the predicates are well-defined for events with zero or more timestamps. As noted in Section 1.1, XQuery's static type system can detect common errors at compile time. Such type safety is particularly valuable for long-running queries on large ad hoc sources and for data sources whose schemata evolve. XQuery is also ideal for specifying integrated

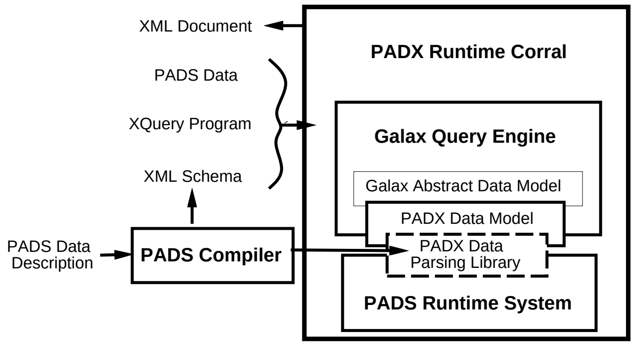

Figure 6. PADX Architecture

views of multiple sources. Although here we focus on querying one ad hoc source at a time, XQuery supports our goal of simultaneously querying multiple PADS and other XML sources. Lastly, XQuery is practical: Numerous language manuals already exist; it is widely implemented in commercial databases; and it will be coming an enduring standard.

Galax is a complete, extensible, and efficient implementation of XQuery 1.0. It includes support for all of XML 1.0, for most of XML Schema 1.0, which is the foundation of the XQuery type system, and for all of XQuery 1.0. Galax was designed with research in mind: Its architecture is modular and documented [**?**]; its query compiler produces evaluation plans in the first complete algebra for XQuery [**?**]; and its optimizer detects joins and grouping constructs in algebraic plans and produces efficient physical plans that employ traditional and novel join algorithms [**?**].

Galax provides an abstract data model. which makes it possible for Galax to evaluate queries simultaneously over native and virtual XML sources that implement the data model.

Algebraic operators "bottom out" by invoking methods in the abstract

3.1 Galax's Abstract Data Model

[**?**]

Abstract object-oriented data model that permits querying of virtual XML sources.

Tree data model. Data model accessors (axis::node-tests) that can/should be implemented efficiently by the underlying source are pushed into the OO tree data model. Default implementations for sources that don't do anything clever.

Motivation for abstract data model: Supporting PADS and secondary storage system occurred at same time. Data and information integration.

Implementations provided: Main-memory DOM-like rep; Main-memory shredded (HashTable) rep; Secondary-storage shredded rep; PADS.

Part of PADX implementation are generic functions that implement data model accessors. PADS compiler generates type-specific functions for walking virtual XML tree. Relate back to type-specific library functions mentioned in last section.

4. Using PADX to Query Ad Hoc Data

Put the pieces all together. Sythesis of the two systems here. (Symbiotic)

4.1 Virtual XML view of PADS data

Embedding of PADS types in XML Schema. One-to-one mapping from PADS compound types to XML Schema complex types. One-to-one mapping from PADS base types to XML Schema simple types. Field in compound types are realized as local elements in XML Schema.

All the compound types are annotated with an optional parse-descriptor (absent if no errors occured). Allows users to query error structures, which may be most important data. Other types annotated with corresponding fields from PADS rep, e.g., arrays have a length.

Extra level of indirection in representation of arrays—wrap each item in an element.

Extra level of indirection for base types: must contain the value of the base type and an optional parse-descriptor, if an error has occurred.

We don't take complete advantage of XML Schema, e.g., Penum types could be modeled by XML Schema enumeration simple types, but currently unsupported.

Generated XML Schema.

"Error-aware" mapping from PADS type system to isomorphic XML Schema.

```
<xs:complexType name="val_Puint32">
  <xs:choice>
   <xs:element name="val" type="p:Puint32"/>
   <xs:element name="pd" type="p:Pbase_pd"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="Pbase_pd">
 <xs:sequence>
   <xs:element name="pstate"  type="p:Pflags_t"/>
   <xs:element name="errCode" type="p:PerrCode_t"/>
   <xs:element name="loc"     type="p:Ploc_t"/>
 </xs:sequence>
</xs:complexType>
```

Example of query that uses generated schema.

```
declare namespace p = "http://www.padsproj.org/pads.xsd";
import schema default element namespace "file:sirius.p"
  schemaLocation "file:/somewhere/sirius.xsd";
$pads/p:Psource/orders/elt[events/elt[1]
  [tstamp >= xs:dateTime("2004-10-01:00:00:00") and
   tstamp < xs:date("2004-11-01:00:00:00") ]]
```

4.2 Physical Data Model

Implementation of Galax's Abstract Tree Model.

Minimum necessary to implement Galax DM:

1. Generic implementations of the DM accessors: axis::node-test(), children(), attributes(), name(), etc.

2. On PADX-side, we have a virtual handle for each node in the XML tree–we call that a node rep. Node rep contains pads handle (maintains state for PADS parser); type-specific vtable of DM accessors; other stuff...

Give example of vtable for event_t and possibly code for kthChildByName.

When to actually read from PADS data?

Options:

1. Bulk read: Materialize entire PADS representation, populate all of the PADS reps. Then PADX DM lazily invoked the DM accessors over this data. Problem: if data is big, it's all sitting in memory, even if the query only touches a fragment of the virtual XML tree.

2. Smart read:

Many common queries permit sequential, streamed access to underlying XML source. Give an example.

Smart node rep, preserves meta-data about previously read records, but re-uses memory for reading next item. This rep permits multiple scans of input (semantic problem is that DM must preserve node identity), but slowly.

```
method virtual node_name  : unit -> atomicQName option
method virtual parent     : node_test option -> node option
method virtual children   : node_test option -> node cursor
method descendant_or_self : node_test option -> node cursor
```

Figure 5. Signatures for methods in Galax's abstract data model

```
<xs:schema targetNamespace="file:sirius.p"
           xmlns="file:sirius.p"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:p="http://www.padsproj.org/pads.xsd">
<xs:import namespace = "http://www.padsproj.org/pads.xsd".../>
...
<xs:complexType name="order_header_t">
 <xs:sequence>
  <xs:element name="order_num" type="p:val_Puint32"/>
  <xs:element name="att_order_num" type="p:val_Puint32"/>
  <xs:element name="ord_version" type="p:val_Puint32"/>
  <!-- More local element declarations -->
  <xs:element name="pd" type="p:PStruct_pd" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>
<!-- More complex type declarations -->
<xs:complexType name="orders_t">
 <xs:sequence>
  <xs:element name="elt" type="order_t" maxOccurs="unbounded"/>
  <xs:element name="length" type="p:Puint32"/>
  <xs:element name="pd" type="p:Parray_pd" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>
...
</xs:schema>
```

Figure 7. Fragment of XML Schema for Sirius PADS description.

Heuristic: records are a good level of granularity to read. Each smart node corresponds to one record. When next smart node is accessed, a little meta-data is preserved: the node rep and the records location in the file (so we can re-read it if necessary).

3. Linear read: same as smart but does not preserve meta-data. Does not permit multiple scans of data source.

Put in PADX signatures for constructing a new node and accessing kthChild.

5. Performance

5.1 Materialization and Loading

Hypothesis: bulk loading should not scale for increasing document size (limits of main memory). Show that smart/linear does scale.

5.2 Querying

Give examples of queries that analyst cares about.

Example of query that can be evaluated in single scan over data source, but is currently not

Database person would balk at this point! Why aren't you just loading this data into a real database, building indices and getting good query performance? B/c data is ephmeral, queries are ephmeral, but analyst/programmer should profit from disciplined access/querying of their data. Don't abandon them to Perl.

6. Related Work

DFDL. Contivo.

7. Future Work and Discussion

Open problem: give result of XQuery and its corresponding type, serialize result back into PADS rep. How are the syntactic constraints for the new values expressed? Tool could pick default delimiters automatically.

Open problem: given an arbitrary PADS type, permit skipping and reading at arbitrary positions within the data source.

Big open problem: Given arbitrary XQuery expression, determine whether it can be evaluated in single scan over data.

"Build your own XQuery processor", EDBT Summer School, Sept 2004. Presentation on Galax architecture. `http://www.galaxquery.or`

"A Complete and Efficient Algebraic Compiler for XQuery" Christopher Ré, Jérôme Siméon, Mary Fernández. International Conference on Data Engineering (ICDE), to appear, April, 2006.

References

[1] PADS user manual. `http://www.research.att.com/projects/pads/inde` To appear.

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

[3] M. F. Fern´andez, J. Sim´eon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080. ACM, 2003.

[4] H. Katz, editor. *XQuery from the experts*. Addison Wesley, 2004.