PADX : Querying Large-scale Ad Hoc Data with XQuery

Mary Fernández
Kathleen Fisher
AT&T 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 hierarchy
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

This paper describes our experience designing and implementing PADX, a system for querying large-scale ad hoc data sources with XQuery. PADX is the synthesis and extension of two existing systems: PADS and Galax. With PADX, an analyst writes a declarative data description of the physical layout of her ad hoc data, and the PADS compiler produces customizable libraries for parsing the data and for viewing it as XML. The resulting library is linked with an XQuery engine, permitting the analyst to view and query her ad hoc data sources using XQuery.

1 Introduction

Although enormous amounts of data exist in "well-behaved" formats such as XML and relational databases, massive amounts also exist in non-standard or *ad hoc* data formats. 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 [6]. It frequently comes with incomplete and/or out-of-date documentation, and there are almost always errors in the data.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, preferably last week? The accompanying documentation is outdated and missing important information, 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 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 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 (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 are not available to the analyst. However, with the appropriate tool support, many aspects of this process can be greatly simplified.

We have two tools, PADS [2, 8] and Galax [1, 7], each of which addresses aspects of the analyst's problem in isolation. The PADS system allows analysts to describe ad hoc data sources declaratively and then generates error-aware parsers and tools for manipulating the sources, including statistical profiling tools. Such support allows the analyst to produce a robust, error-aware parser quickly. The Galax system supports declarative querying of XML via XQuery. If Galax could be applied to ad hoc data, it would allow the analyst first to explore the data and then to produce answers to her questions.

In this work, we strive to integrate PADS and Galax to solve the analyst's data-management problems for the large ad hoc data sources that we have seen in practice. One approach would be to have PADS produce a tool for converting ad hoc data to XML and then ap-

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

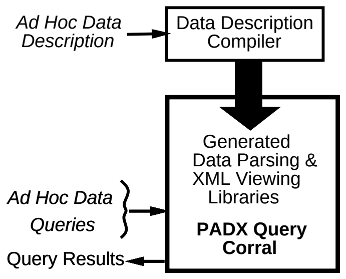

Figure 2. Data analyst's view of PADX

ply Galax to the resulting document. (In fact, PADS provides this ability.) However, the typical factor of eight space blow up in this conversion yields an unacceptable slowdown in performance. Consequently, we chose to design and implement PADX[1], a synthesis and extension of PADS and Galax. Figure 2 depicts PADX from the analyst's perspective. The analyst provides a PADS description of her ad hoc source, which is compiled into a library of components for parsing her data and for viewing and querying it as XML. The resulting libraries are linked together with the PADS and Galax runtime systems into one PADX query executable, called a "query corral.[2]" At query time, the analyst provides her ad hoc data sources and her query written in XQuery, and PADX produces the query's results.

Building PADX presented several problems. The first was semantic: We had to decide how to view ad hoc data as XML and how to express this view as a mapping from the PADS type system to XML Schema, the basis of XQuery's type system. A second problem involved systems design and engineering. Building PADX required evolving PADS and Galax in parallel, modifying the implementation of Galax to support an abstract data model so that Galax could view non-XML sources as XML, and augmenting PADS with the ability to generate concrete instances of this data model. Our solutions to these problems, which were necessary to build a working system, are described in Sections 3 and 4. A third problem involves the scale of data and efficiency of queries, in particular, how to efficiently evaluate complex queries over large sources. Section 5 describes how PADX currently handles large sources and the problems that we face with respect to data scale and query performance.

We begin with a more detailed account of a scenario that illustrates the data management tasks faced by AT&T data analysts and how PADX simplifies these tasks. We then crack open the PADX architecture, first describing PADS and Galax in isolation, and then describing our solutions to the problems described above. We conclude with related work and a discussion of open problems.

1.1 Data-management 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.

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

Figure 4 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_t`, appears at the bottom of the description (Line 42). In the next section, we use this example to give an overview 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 often want to 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 that data proceeds to a down-stream process or is loaded into a database.

With PADX, the analyst writes declarative XQuery expressions to query her ad hoc data source. Because XQuery is designed to manipulate semi-structured data, its expressiveness matches ad hoc data sources well. As a Turing-complete language, XQuery is powerful enough to express all the questions above. For example, Figure 5 contains an XQuery expression that produces all orders that started in October, 2004. In Section 4, we discuss in more detail why XQuery is an appropriate query language for ad hoc data. One benefit is that XQuery queries may be statically typed, which helps detect common errors at compile time. For example, static typing would raise an error if the path expression in Figure 5 referred to `ordesr` instead of `orders`, or if the analyst erroneously compared the timestamp value in `tstamp` to a string.

[1]Pronounced "paddocks", an enclosed area for exercising race horses.

[2]The equestrian metaphor is intentional: Getting these systems to work together is like corralling race horses!

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

```
 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.

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

Figure 5. Query applied to Sirius provisioning data.

2 Using PADS to Access Ad Hoc Data

In this section, we give a brief overview of PADS, focusing on its data description language and the portions of the libraries it generates that are relevant to PADX. More information about PADS is available [2, 8].

2.1 PADS: The language

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. A type may have an associated predicate that determines whether a parsed value is indeed a legal value for the type. For example, a predicate might require that one field of a **`Pstruct`** is bigger than another or that the elements of a sequence are sorted. Programmers can specify such predicates using PADS expressions and functions, written in a C-like syntax. Finally, PADS **`Ptypedef`**s allow programmers to define new types that add further constraints to existing types.

PADS types can be parameterized by values. This mechanism reduces the number of base types and permits 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(:'|':)` (*e.g.*, Line 29) describes a string terminated by a vertical bar. Parameters can be used with compound types to specify the size of an array or the appropriate branch of a union.

`Pstruct`s describe ordered sequences of data with unrelated types. In Figure 4, the type declaration for the **`Pstruct`** `order_t` (Lines 35–38) contains an order header (`order_header_t`) followed by the literal character **`'|'`**, followed by an event sequence (`event_seq_t`). PADS supports character, string, and regular expression literals.

`Punion`s describe alternatives in the data format. For example, the `dib_ramp_t` type (Lines 9–12) indicates that the `ramp` field in a Sirius record can be either a `Puint_64` or a string `"no_ii"` followed by a `Puint_64`. During parsing, the branches of a **`Punion`** are tried in order; the first branch that parses without error is taken.

The `order_header_t` type (Lines 13–27) contains several anonymous uses of the **`Popt`** type. This type is syntactic sugar for a stylized use of a **`Punion`** with two branches: the first with the indicated type, and the second with the "void" type, which always matches but never consumes any input.

PADS provides **`Parray`**s to describe varying-length sequences of data all with the same type. The `event_seq_t` type (Lines 32–34) uses a **`Parray`** to characterize the sequence of events an order goes through during processing. This declaration indicates that each element in the sequence has type `event_t`. It also specifies that the elements will be separated by vertical bars, and that the sequence will be terminated by an end-of-record marker (**`Peor`**). In general, PADS provides a rich collection of array-termination conditions: reaching a maximum size, finding a terminating literal (including end-of-record and end-of-source), or satisfying a user-supplied predicate over the already-parsed portion of the **`Parray`**.

Finally, the **`Precord`** (Line 35) and **`Psource`** (Line 42) annotations deserve comment. The first indicates that the annotated type constitutes a record, while the second means that the type constitutes the totality of a data source. The notion of a record varies depending upon the data encoding. ASCII data typically uses newline characters to delimit records, binary sources tend to have fixed-width records, while COBOL sources usually store the length of each record before the actual data. PADS supports each of these encodings of records and allows users to define their own encodings.

2.2 PADS: The generated library

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

- an in-memory representation,
- parsing and printing functions,
- a mask, which allows customization of generated functions, and
- a parse descriptor, which describes syntactic and semantic errors detected during parsing.

To give a feeling for the library that PADS generates, Figure 6 includes a fragment of the generated library for the Sirius `event_t` declaration.

The C declarations for the in-memory representation (Line 1–4), the mask (Line 5–9), and the parse descriptor (Line 10–17) all share the structure of the PADS type declaration. The mapping to C for each is straightforward: **`Pstruct`**s map to C structs with appropriately mapped fields, **`Punion`**s map to tagged unions coded as C structs with a tag field and an embedded union, **`Parray`**s map to a C struct with a length field and a dynamically allocated sequence, **`Penum`**s map to C enumerations, **`Popt`**s to tagged unions, and **`Ptypedef`**s to C typedefs. Masks include auxiliary fields to control behavior at the level of a structured type, and parse descriptors include fields to record the state of the parse, the number of detected errors, the error code of the first detected error, and the location of that error.

The parsing functions, *e.g.* `event_t_read` on Line 19, take a mask as an argument and returns an in-memory representation and a parse descriptor. The mask allows the user to specify which constraints the parser should check and which portions of the in-memory representation it should fill in. This control allows the description-writer to specify all known constraints about the data without worrying about the run-time cost of verifying potentially expensive constraints for time-critical applications.

Appropriate error-handling is as important as processing error-free data. The parse descriptor marks which portions of the data contain errors and specifies the detected errors. Depending upon the nature of the errors and the desired application, programmers can take the appropriate action: halt the program, discard parts of the data, or repair the errors. If the mask requests that a data item be verified and set, and if the parse descriptor indicates no error, then the in-memory representation satisfies the semantic constraints on the data.

Because we generate a parsing function for each type in a PADS description, we support multiple-entry point parsing, which accommodates larger-scale data. For a small file, a programmer can call the parsing function for the PADS type that describes the entire file (*e.g.* `summary_t_read`) to read the whole file with one call. For larger-scale data, programmers can sequence calls to parsing functions that read manageable portions of the file, *e.g.*, reading one record at a time in a loop. The parsing code generated for **`Parray`**s allows users to choose between reading the entire array at once or reading it one element at a time, again to support parsing and processing very large data sources. We return to the use of multiple-entry point parsing functions in Section 5.

3 Using XQuery and Galax

In this section, we give a brief overview of XML, XQuery, and Galax, focusing on Galax's data-model support for viewing non-XML data as XML. Given the subject of this workshop, we assume the reader is already familiar with XML, XQuery, and XML Schema.

XML [18] 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 standard format. This flexibility has made XML the "lingua franca" of data integration and exchange.

XQuery [20] 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 [21]. XQuery contains XPath 2.0 [19] as a proper sub-language, 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.

XQuery is a natural choice for querying ad hoc data. Like XML data, ad hoc data is semi-structured, and XQuery is tailored to such data. XQuery's static type system detects type errors at compile time, which is valuable when querying ad hoc sources: Long-running queries on large ad hoc sources do not raise dynamic type errors, and queries made obsolete by schema evolution are identified at compile time. XQuery is also ideal for specifying integrated views of multiple sources. Although here we focus on querying one ad hoc source at a time, XQuery supports simultaneous querying of multiple sources. Lastly, XQuery is practical: It will soon be a standard; numerous manuals already exist [5]; and it is widely

```
typedef struct {     // In-memory representation
  order_header_t order_header;
  event_seq_t    events;
} event_t;

typedef struct {     // Mask
  Pbase_m            compoundLevel;   // Struct-level controls
  order_header_t_m   order_header;
  event_seq_t_m      events;
} event_t_m;

typedef struct {     // Parse descriptor
  Pflags_t   pstate;       // Normal, partial, or panicking
  Puint32    nerr;         // Number of detected errors
  PerrCode_t errCode;      // Error code of first detected error
  Ploc_t     loc;          // Location of first error
  order_header_t_pd order_header;     // Nested header information
  event_seq_t_pd    events; // Nested event sequence information
} event_t_pd;

/* Parsing and printing functions */
Perror_t event_t_read     (P_t *pads, event_t_m *m, event_t_pd *pd, event_t *rep);
ssize_t  event_t_write2io (P_t *pads, Sfio_t *io,   event_t_pd *pd, event_t *rep);
```

Figure 6. Fragment of the library generated for the `event_t` declaration from Sirius data description.

implemented in commercial databases.

Galax is a complete, extensible, and efficient implementation of XQuery 1.0 that supports XML 1.0 and XML Schema 1.0 and that was designed with database systems research in mind. Its architecture is modular and documented [15], which makes it possible for other researchers to experiment with a complete XQuery implementation. Its compiler produces evaluation plans in the first complete algebra for XQuery [13], which permits experimental comparison of query-compilation techniques. Lastly, its query optimizer produces efficient physical plans that employ traditional and novel join algorithms [13], which makes it possible to apply non-trivial queries to large XML sources. Lastly, its abstract data model permits experimenting with various physical representations of XML and non-XML data sources. Galax's abstract data model is the focus of the the rest of this section.

3.1 Galax's Abstract Data Model

Galax's abstract data model is an object-oriented realization of the XQuery Data Model. The XQuery Data Model [17] contains tree nodes, atomic values, and sequences of nodes and atomic values. A tree node corresponds to an entire XML document or to an individual element, attribute, comment, or processing-instruction. Algebraic operators in a query-evaluation plan produced by Galax's query compiler access documents by applying methods in the data model's object-oriented interface.

Figure 7 contains part of Galax's data model interface[3] for a node in the XQuery Data Model. Node accessors return information such as a node's name (`node_name`), the XML Schema type against which the node was validated (`type`), and the node's atomic-valued data if it was validated against an XML Schema simple type (`typed_value`). The `parent`, `child`, and `attribute` methods navigate the document and return a node sequence containing the respective parent, child, or attribute nodes of the given node.

[3]Galax is implemented in O'Caml, so these signatures are in O'Caml.

The first six methods in Figure 7 (Lines 5–11) access the physical representation of a document. Therefore, a concrete instance of the data model must provide their implementations. Galax provides default implementations for the four descendant and ancestor axes (Lines 13–16), which are defined recursively in terms of the child and parent methods. These defaults may be overridden in concrete data models that can provide more efficient implementations than the defaults. For example, some representations permit axes to be implemented by range queries over relational tables [11].

All the axis methods take an optional node-test argument, which is a boolean predicate on the names or types of nodes in the given axis. For example, the XQuery expression `descendant::order` returns nodes in the descendant axis with name `order`. Galax compiles this expression into a single axis/node-test operator that invokes the corresponding methods in the abstract data model, delegating evaluation of node tests to the concrete data model. Some implementations, like PADX, can provide fast access to nodes by their name. We describe PADX's concrete data model in Section 4.

One other important feature of Galax's abstract data model is that sequences are represented by *cursors* (also known as streams), non-functional lists that yield items lazily. Accessing the first item in a sequence does not require that the entire sequence be materialized, *i.e.*, evaluated eagerly. Galax's algebraic operators produce and consume cursors of values, which permits pipelined and short-circuited evaluation of query plans.

In addition to the concrete data model for PADX,which we describe in the next section, Galax has three other concrete data models: a DOM-like representation in main memory and two "shredded" representations, one in main memory and one in secondary storage for very large documents (*e.g.* $>$ 100MB). The shredded data model partitions a document into tables of elements, attributes, and values that can be indexed on node names and values [16].

4 Using PADX to Query Ad Hoc Data

Figure 8 depicts an internal view of the PADX architecture first shown in Figure 2. Pre-existing components (in grey boxes) include

```
type sequence = cursor
class virtual node :
object
  (* Selected XQuery Data Model accessors *)
  method virtual node_name   : unit -> atomicQName option
  method virtual type       : unit -> (schema * atomicQName)
  method virtual typed_value : unit -> atomicValue sequence

  (* Required axes *)
  method virtual parent      : node_test option -> node option
  method virtual child       : node_test option -> node sequence
  method virtual attribute   : node_test option -> node sequence

  (* Other axes *)
  method descendant_or_self  : node_test option -> node sequence
  method descendant          : node_test option -> node sequence
  method ancestor_or_self    : node_test option -> node sequence
  method ancestor            : node_test option -> node sequence

... Other accessors in XQuery Data Model ...
```

Figure 7. Signatures for methods in Galax's abstract node interface

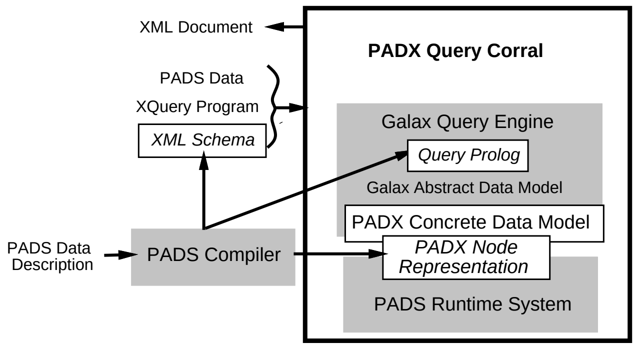

Figure 8. Internal view of PADX Architecture

```
 1. <xs:simpleType name="base_Puint32">
 2.  <xs:restriction base="xs:unsignedInt"/>
 3. </xs:simpleType>
 4. <xs:complexType name="val_Puint32">
 5.  <xs:choice>
 6.   <xs:element name="rep" type="p:base_Puint32"/>
 7.   <xs:element name="pd"  type="p:Pbase_pd"/>
 8.  </xs:choice>
 9. </xs:complexType>
10. <xs:complexType name="Pbase_pd">
11.  <xs:sequence>
12.   <xs:element name="pstate"  type="p:Pflags_t"/>
13.   <xs:element name="errCode" type="p:PerrCode_t"/>
14.   <xs:element name="loc"     type="p:Ploc_t"/>
15.  </xs:sequence>
16. </xs:complexType>
```

Figure 9. Fragment of XML Schema for PADS base types.

the PADS compiler, the Galax query engine, and the PADS runtime system. In this section, we focus on the new components (in white boxes) and describe the compiler and run-time support for viewing PADS data as XML. From a PADS description, the compiler generates an XML Schema description that specifies the virtual XML view of the corresponding PADS data, an XQuery prolog that imports the generated schema and that associates the input data with the correct schema type, and a type-specific library that provides the virtual XML view of PADS values necessary to implement PADX's concrete data model.

Note that a query corral is *customized* for a particular PADS description, in particular, its concrete data model only supports views of data sources that match the PADS description. To maintain the correct correspondence between a description, XML Schema, queries, and data, the query corral explicitly contains the generated query prolog, which imports the XML Schema that corresponds to the underlying type-specific library. This guarantees that the user's XQuery program is statically typed, compiled, and optimized with respect to the correct XML Schema and that the underlying data model is an instance of this XML Schema. At runtime, the query corral takes an XQuery program and a PADS data source and produces the query result in XML. We discuss the problem of producing native PADS values in Section 6.

4.1 Viewing PADS data as XML

The mapping from a PADS description to an XML Schema is straight-forward. The interesting aspect of this mapping is that both PADS values that are error free and those containing errors are accessible in the XML view. We begin with the mapping of PADS base types.

A default XML Schema, `pads.xsd`, contains the schema types that represent the PADS base types shared by all PADS descriptions. Figure 9 contains a fragment of this schema. Every PADS base type is mapped to the schema simple type that most closely subsumes the value space of the given PADS base type. For example, the `Puint32` base type maps to the schema type `xs:unsignedInt` (Lines 1–3). Recall that all parsed PADS values have an in-memory representation and a parse descriptor, which records the state of the parse, the error code for detected errors, and the location of those errors. The XML view of a parsed value is a choice of the in-memory representation (`rep`), if no error occurred, or of the parse descriptor (`pd`), if an error occurred (Lines 4–8). This light-weight view exposes the parse descriptor only when an error occurs. The parse-descriptor type for all base types is represented by the schema type `Pbase_pd` (Line 10–14).

The fragment of the XML Schema in Figure 10 corresponds to the

```
 1. <xs:schema targetNamespace="file:/example/sirius.p"
 2.            xmlns="file:/example/sirius.p"
 3.            xmlns:xs="http://www.w3.org/2001/XMLSchema"
 4.            xmlns:p="http://www.padsproj.org/pads.xsd">
 5. <xs:import namespace = "http://www.padsproj.org/pads.xsd".../>
 6. ...
 7. <xs:complexType name="order_header_t">
 8.  <xs:sequence>
 9.   <xs:element name="order_num"     type="p:val_Puint32"/>
10.   <xs:element name="att_order_num" type="p:val_Puint32"/>
11.   <xs:element name="ord_version"   type="p:val_Puint32"/>
12.   <!-- More local element declarations -->
13.   <xs:element name="pd"            type="p:PStruct_pd" minOccurs="0"/>
14.  </xs:sequence>
15. </xs:complexType>
16. <!-- More complex type declarations -->
17. <xs:complexType name="orders_t">
18.  <xs:sequence>
19.   <xs:element name="elt"    type="order_t" maxOccurs="unbounded"/>
20.   <xs:element name="length" type="p:Puint32"/>
21.   <xs:element name="pd"     type="p:Parray_pd" minOccurs="0"/>
22.  </xs:sequence>
23. </xs:complexType>
     ...
24. <xs:element name="Psource" type="summary_t"/>
25. </xs:schema>
```

Figure 10. Fragment of XML Schema for Sirius PADS description.

description in Figure 4. Note that the schema imports the schema for PADS base types (Line 5). Each compound type is mapped to a complex schema type with a particular content model. A **Pstruct** is mapped to a complex type that contains a sequence of local elements, each of which corresponds to one field in the **Pstruct**. For example, the **Pstruct** order_header_t is mapped to the complex type order_header_t (Lines 7–15), which contains an element declaration for the field order_num, among others. A **Punion** is mapped to a complex type that contains a choice of elements, each of which corresponds to one field in the **Punion**.

Each complex type also includes an optional pd element that corresponds to the type's parse descriptor (Lines 13 and 21). All parse-descriptor types contain the parse state, error code, and location. The parse-descriptor for compound types contain additional information, *e.g.*, Pstruct_pd contains the number of nested errors and Parray_pd contains the index of the array item in which the first error occurred. The pd element is absent if no errors occurred during parsing, but if present, permits an analyst to easily identify the kind and location of errors in the source data. For example, the following XQuery expression returns the locations of all orders that contain at least one error: $pads/Psource/orders/elt/pd/loc.

The schema types for some compound types contain additional fields from the PADS in-memory representation, *e.g.*, arrays have a length (Line 20). Note that Parray types do not associate a name with each individual array item, so in the corresponding schema type, the default element elt encapsulates each array item.

The PADS compiler generates a query prolog that specifies the environment in which all XQuery programs are typed and evaluated. Figure 11 contains the query prolog for the schema in Figure 10. The import schema declaration on Line 1 imports the schema in Figure 10. This declaration puts all global element and type declarations in scope for the query. The variable declaration on Line 2 specifies that the value of the variable $pads is provided externally and that its type is a document whose top-level element is of type Psource, defined on Line 24 in Figure 10. This declaration guarantees that the query is statically typed with respect to the correct input type.

At run time, the user can specify the input data as a command-line argument or by calling the XQuery fn:doc function on a PADS source, *e.g.* pads:/example/sirius.data.

4.2 PADX Concrete Data Model

In Figure 8, the interface between Galax and PADS consists of two modules: the generic PADX concrete data model, which implements the Galax abstract data model, and a compiler-generated module, in which each PADS type has a corresponding, type-specific node representation providing the XML view of values of that type. We note that the generic concrete data model is implemented in O'Caml and the compiler-generated module is implemented in C, but to simplify exposition, we present the compiler-generated module in O'Caml syntax.

Figure 13 contains a fragment of the PADX concrete data model for a node. This object provides a thin wrapper around the type-specific node representation, padx_node_rep, whose interface is in Figure 12. A node representation contains references to a PADS value's in-memory representation and parse descriptor. The node representation interface returns the XML view of the PADS value, including the value's element name, its typed value, and parent. The kth_child and kth_child_by_name methods return all of the PADS value's children in order and those with a given name in order, respectively.

For some methods in Figure 13 (Lines 4–5), the concrete data model simply invokes the corresponding type-specific methods. One exception is the child axis method (Lines 7–17), which we describe in detail as it illustrates how the XML view of a PADS source is materialized lazily. The child method takes an optional name-

```
import schema default element namespace "file:/example/sirius.p";
declare variable $pads as document-node(Psource) external;
```

Figure 11. PADX generated query prolog

```
class virtual padx_node_rep :
  object
    (* Private data includes parsed value’s rep & pd *)
    method node_name   : unit -> string
    method typed_value : unit -> item
    method parent       : unit -> padx_node_rep option
    method kth_child   : int -> padx_node_rep option
    method kth_child_by_name : int -> string -> padx_node_rep option
  end
```

Figure 12. The PADX node representation

```
class pads_node (nr : padx_node_rep) =
object
  inherit Galax.node
  method node_name   () = nr#node_name()
  method typed_value () = nr#typed_value()
  (* ... Other data model accessors ... *)
  method child name_test =
    let k = ref 0 in
    match name_test with
    | None ->
      let lazy_child () =
       (incr k;
        match nr#kth_child !k with
        | Some cnr ->  Some(new pads_node(cnr))
        | None -> None)
      in Cursor.cursor_of_function lazy_child
    | Some (NameTest name) ->
        (* Same as above, but call nr#kth_child_named *)
  (* ... Other axes ... *)
end
```

Figure 13. Fragment of the PADX concrete data model

test argument. We describe the case when the name-test is absent, which corresponds to the common expression `child::*`. The `child` method creates a mutable counter `k` (Line 8), which contains the index of the last child accessed, and a continuation function `lazy_child` (Lines 11–16), which is invoked each time the `child` cursor is poked. On each invocation, `lazy_child` increments the counter and delegates to the `kth_child` method of the type-specific node representation. For some PADS types, accessing the virtual k^th child does not require reading or parsing data, *e.g.*, if the virtual child is part of a complete PADS record. For other PADS types, *e.g.*, **`Parray`**s that contain file records, accessing the virtual k^th child may require reading and parsing data. The `kth_child` method provides a uniform interface to all types and delegates the problem of when to read and parse data to the underlying type-specific node representation.

To illustrate type-specific compilation, we give the compiler-generated node representation of an `order_header_t` value in Figure 14. The object takes the name of the field that contains the `order_header_t` value, which corresponds to the XML node name, and the in-memory representation and parse descriptor of the value. The `kth_child` method (Lines 9–15) takes an index and returns the node representation of the field at that index. For example, the first child (Line 11) corresponds to the field `order_num`, which contains a **`Puint32`** value. The `kth_child_by_name` method (Lines 16–21) provides constant-time lookup of a child with a particular name: It looks up the index of the name in the associative map `name_map` and then delegates to `kth_child`. Note that this XML view of an `order_header_t` value corresponds to the schema type `order_header_t` in Figure 10.

To summarize, the PADX concrete data model completely implements the Galax data model, making it possible to evaluate any XQuery program over a PADS data source. Due to limited space, we have omitted some details, such as how PADX guarantees that each virtual node has a unique, immutable identity, as is required by the Galax abstract data model. The data model's most important features are that it provides lazy access to virtual XML nodes in the PADS source, it delegates navigation to type-specific node representations, and it separates navigation of the virtual nodes from data loading, which is discussed next.

4.3 Loading PADS data

The PADX abstract data model provides Galax with a random-access view of a PADS data source. In particular, any virtual node may be accessed in any order at any time during query evaluation regardless of its physical location in the PADS data. This abstraction permits the PADX concrete data model to decide when and how to read and parse, or *load*, a data source.

PADX has three strategies for loading data, each of which use the multiple-entry parsing functions generated by the PADS compiler. The *bulk* strategy loads a complete PADS source before query evaluation begins, populating all the in-memory representations and parse descriptors. With all data pre-fetched, bulk loading is the simplest strategy to implement random access. However, because each PADS value has a lot of associated meta-data, bulk loading incurs a high memory cost and is only feasible for smaller data sources.

The *on-demand, random-access* strategy loads PADS data when Galax accesses virtual nodes via the abstract data model. The strategy maintains a fixed size buffer for loaded values and when the buffer is filled, expels values in LIFO order. The default units loaded are any PADS types annotated with **`Precord`**, which indicates that the type denotes an atomic physical unit in the ambient coding. This default works well in practice, because many PADS sources contain a header, one (or more) very large array(s) of records, and a trailer. This strategy loads all the data before the record array(s) and then loads each array item on demand, expelling old records when the buffer is filled. A small amount of meta-data is preserved for each expelled record, so that the virtual node containing that data can be reconstructed on subsequent accesses.

The *on-demand, sequential* strategy is a restriction of the on-demand, random-access strategy. It loads data on demand, but its fixed-size buffer stores only one record at a time, and it supports strictly sequential access to records, *i.e.*, accessing records out of order is prohibited. Given that the Galax abstract data model requires random access, it is not obvious when this strategy can be used, even though it has the smallest memory footprint of all three and therefore could scale to very large sources. It turns out that many common XQuery queries can be evaluated in *one* sequential scan over the input document, and in these cases, the sequential strategy is both semantically correct and time and space efficient. We give examples of "one-scan" queries and their performance in Section 5.

4.4 Ways to use PADX

Our focus so far has been on describing PADX's internal architecture to demonstrate the feasibility of viewing and querying ad hoc data sources as though they were XML. We expect this use of PADX to be convenient, because it supports rapid querying of transient data and does not require an analyst to convert the data into another format or load it into a database before being able to ask simple queries. PADX can be used in other ways. For example, an analyst might prefer to materialize a PADS source in XML and query her data using a high-performance, commercial XML query engine. To do this, the analyst simply runs the query "`$pads`", which returns the entire source materialized in XML, and then provides the resulting XML document to the query engine. Another use is to transform the PADX view of a PADS source into the XML view required by a database by some down-stream application. Such transformations can be easily expressed in XQuery and can be statically type checked against the PADX and target XML schemata.

We note, however, that the size of an ad hoc data source is significantly smaller than its representation in XML. For our two example PADS sources, the ratio of the size of the original PADS data to its size in XML using the mapping described in Section 4.1 ranges from 1:7 to 1:8. Of course, this size increase depends on the PADS types and field names in the PADS description, but even a reasonable choice of names like those in Figure 4 results in a significant size increase. We mention this size increase to give the reader some sense of the relative scale of data sources that PADX can query compared to those supported by native XML query engines.

5 Performance

Query performance in PADX depends on the efficiency of the underlying concrete data model; therefore its performance must be well understood before we can understand the performance of particular query plans. We focus on the performance of the concrete data model and measure the cost of accessing data via the PADS type-specific parsing functions, the PADX type-specific node representations, and the generic PADX concrete data model. At the end of this section, we give preliminary measurements on query performance.

```
class order_header_t_node_rep
     (field_name : string)
     (rep : order_header_t)
     (pd  : order_header_t_pd) =
object
  inherit padx_node_rep
  method name() = field_name
  ...
  method kth_child idx =
    match idx with
    |  1 -> Some(new val_Puint32_node_rep("order_num", rep.order_num, pd.order_num_pd))
    |  2 -> Some(new val_Puint32_node_rep("att_order_num", rep.att_order_num, pd.att_order_num_pd))
    | ...
    | 14 -> Some(new Pstruct_pd_node_rep("pd", pd))
    | _  -> None

  (* Chidren's name map *)
  let name_map = Associative_array.create [("order_num", 1); ("att_order_num", 2); ...; ("pd", 14)]
  method kth_child_by_name child_name =
    match Associative_array.lookup name_map child_name with
    | None -> Cursor.empty_cursor()
    | Some idx -> kth_child idx
end
```

Figure 14. Fragment of compiler-generated node representation for `order_header_t`

	Data size (MB)				
Source	1	5	10	20	50
Sirius	0.25	0.23	0.23	0.22	10.64
Web server	0.70	0.67	0.67	1.18	6.14

Table 1. Bulk strategy: load time per byte in μs

Source	Data size	PADS read	PADX node rep	PADX generic DM
Sirius	5MB	0.07	0.27	0.61
	10MB	0.06	0.26	0.56
	50MB	0.06	0.25	0.56
Web server	5MB	0.54	0.78	1.63
	10MB	0.53	0.74	1.61
	50MB	0.53	0.74	1.58

Table 2. Sequential strategy: load time per byte in μs

We measured data model and query performance for two PADS sources, Sirius and the Web server logs in Figure 1, on data sources of 1 to 50MB in size. Our measurements were taken on an 1.67GHz Intel Pentium M with 500MB real memory running Linux Redhat 9.0. Each test was run five times, the high and low times were dropped, and the mean of the remaining three times is the reported time.

5.1 Concrete Data Model

We first measured the time to bulk load data sources of 5, 10, 20, and 50MB by calling the PADS parsing functions, *i.e.*, the lowest level in the PADX data model. Table 1 gives the load time per byte in microseconds. For smaller sources, load time is constant, but eventually increases. For Sirius, the increasing load time is observed at 50MB and for the Web server data at 20MB. We note that for a PADS source, the memory overhead of a PADS parsed value can be four to sixteen times the size of the raw data, depending on the value's type. In the cases where non-linear load time occurs, the processes' physical memory usage is close to or exceeds real memory, CPU utilization plummets, and the process begins to thrash. These measurements indicate that the bulk strategy is only feasible for smaller data sources.

Next, we measured load time using the on-demand sequential stategy on sources of 5, 10, and 50 MB. We were particularly interested in the overhead introduced at each level in the concrete data model. Table 2 gives the load time per byte in microseconds (μs) for three levels: reading the source by calling the PADS parsing functions directly, a depth-first walk of the virtual XML document by calling the PADX node-representation functions, and a depth-first walk of the virtual XML document by calling the PADX generic data model. Recall that the node-rep functions are in C and the generic data model is in O'Caml.

We observe that the load time per byte at each level is near constant for increasing source size, but that each level incurs a substantial cost compared to the lower levels. For the Sirius source, the PADX node-representation is four times slower than the native PADS parsing functions, but for the Web-server source, the PADX node representation is only 44% slower. Understanding the source of this difference requires further experiments with other sources.

For both sources, the generic concrete data model (in O'Caml) is twice as slow as the node representation (in C). The interface from the generic data model to the node representation crosses the O'Caml-C boundary and uses data marshalling functions generated by the O'Caml IDL tool. We have noticed similar per-byte read costs in the Galax secondary storage system [16], whose data-model architecture is similar to that of PADX.

We also measured the time to load using the on-demand, random-access strategy. In general, it was 10–15% slower than the on-demand, sequential strategy.

These measurements indicate that the on-demand, sequential strategy scales with increasing data size, and that there is a constant overhead incurred at each level in the data model. Ideally, we would like the cost of accessing data via the generic concrete data model to be close to the PADS read cost, but this will require more engineering effort.

Data size (MB)	1	5	10	20	50
Time (seconds)	1.0	4.8	10.7	24.0	90.0

Table 3. PADX query evaluation time in seconds

5.2 Querying

Ultimately, PADX's query performance depends on Galax, because the Galax compiler produces and executes the query plans. Currently, Galax's query compiler includes a variety of logical optimizations for detecting joins and re-grouping constructs in XQuery expressions. Another important optimization is detecting when a query can be evaluated in one scan over the input document. Path expressions that contain only descendant axes and no branches are one example of the kind of queries that can be evaluated in one scan. For example, the following query, which returns the locations of all records containing some error in a Sirius source, can be evaluated in one scan:

```
$pads/Psource/orders/elt/pd/loc
```

Detecting and evaluating one-scan queries (also known as streamable queries) is necessary in XML environments in which the XML data is an infinite or bursty stream. Several query processors already exist in which streamable queries are evaluated directly over a stream of tokens produced by SAX-style parsers [9, 14].

Streamable queries are important for PADX, because the resulting plans can be evaluated on large PADS sources that are loaded on-demand and sequentially. Table 3 contains the time in seconds to evaluate the query above when applied to PADS data sources into which we injected errors randomly in the file (12 errors per 1MB). The query plan produced by Galax is not perfectly pipelined, thus the execution time is super linear.

To understand the costs and benefits of other evaluation strategies, we materialized the 1MB PADS source in Table 3, which yielded a 7.4MB XML document. We then used Galax to execute the above query, using the same query execution plan, and applied it to the 7.4MB XML document loaded into the main-memory data model. The execution time was 13.1s of which 12.9 was spent in document parsing. To amortize the cost of document parsing time, we often store documents in Galax's secondary storage system. To compare with this strategy, we stored the 7.4MB XML document in Galax's secondary storage system, which required 166MB of disk space. We then ran the above query on the stored document. The execution time was 2.9s, almost three times slower than PADX applied to the PADS data directly. For comparison with an independent query processor, we evaluated the above query using Saxon [12], a popular XSLT and XQuery engine, applied to the 7.4MB document and it executed in 6.3s.

In summary, our initial impressions are that evaluating streamable XQuery expressions directly on a PADS source is feasible, efficient, and convenient.

6 Discussion

The PADX 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 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 PADX description and related queries.

A PADX query corrall is an example of partially compiled query engine, because its concrete data model is customized for a particular data format, but its queries are interpreted over an abstract data model that delegates to the concrete model. This architecture places PADX on the continuum between query architectures that provide fully interpreted query plans applied to generic data models to architectures that provide fully compiled query plans applied to customized data model instances [10]. The latter architectures provide very high performance on large scale data. PADX has some of the benefits of such architectures but does not have the overhead of a complete database system.

Others share our interest in declarative descriptions of ad hoc data formats. Currently, the Global Grid Forum is working on a standard data-format description language for describing ad hoc data formats, called DFDL [3, 4]. Like PADS, DFDL has a rich collection of base types and supports a variety of ambient codings. Unlike PADS, DFDL does not support semantic constraints on types nor dependent types, *e.g.*, it is not possible to specify that the length of an array is determined by some field in the data. DFDL is an annotated subset of XML Schema, which means that the XML view of the ad hoc data is implicit in a DFDL description. DFDL is still being specified, so no DFDL-aware parsers or data analyzers exist yet. We expect that bi-directional translation between PADS and DFDL to be straightforward. Such a translation would make it possible for DFDL users to use PADX to query their ad hoc data sources.

The steps in a data-management workflow that PADX addresses typically precede the steps that require a high-performance database system, *e.g.*, asking complex OLAP queries applied to long-lived, archived data. Commercial database products do provide support for parsing data in external formats so the data can be imported into their database systems, but they typically support a limited number of formats, *e.g.*, COBOL copybooks, no declarative description of the original format is exposed to the user for their own use, and they have fixed methods for coping with erroneous data. For these reasons, PADX is complementary to database systems.

We continue to focus on improving the usability and scalability of PADX. Currently, PADX is not compositional, because the result of evaluating a query is in native XML, not in a PADS format. Given an arbitrary XQuery expression over a PADX source, an open problem is being able to infer a reasonable PADS format for the result and produce the results in this format. We have already mentioned the important problem of detecting when a query can be evaluated in a single scan over an input document and of producing a fully pipelined execution plan. Interestingly, this problem is important in XML environments in which the XML data is an infinite or bursty stream. We are working on improving Galax's ability to detect one-scan queries and to produce query plans that are indeed fully pipelined and that use limited memory.

7 References

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

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

[3] Data format description language (DFDL) a Proposal, Working Draft, Global Grid Forum. `https://forge.gridforum.org/projects/dfdl-wg/document/DFDL_Proposal/en/%2`, Aug 2005. Global Grid Forum.

[4] M. Beckerle and M. Westhead. GGF DFDL primer. `http://www.ggf.org/Meetings/GGF11/Documents/DFDL_Primer_v2.pdf`,

May 2004. Global Grid Forum.

[5] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

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

[7] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[8] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, June 2005.

[9] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB J.*, 13(3):294–315, 2004.

[10] R. Greer. Daytona and the fourth generation language cymbal. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 1999.

[11] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its axis steps. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 524–535, Berlin, Germany, Sept. 2003.

[12] M. Kay. SAXON 8.0. SAXONICA.com. `http://www.saxonica.com/`.

[13] C. Ré, J. Siméon, and M. Fernández. A complete and efficient algebraic compiler for XQuery. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, April 2006.

[14] K. Rose and L. Villard. Phantom XML. In *XML Conference and Exhibition*, 2005.

[15] J. Siméon and M. F. Fernández. Build your own XQuery processor. EDBT Summer School, Tutorial on Galax architecture, Sept 2004. `http://www.galaxquery.org/slides/edbt-summer-school2004.pdf`.

[16] A. Vyas, M. F. Fernández, and J. Siméon. The simplest XML storage manager ever. In *XIME-P 2004*, pages 37–42, Paris, France, June 2004.

[17] W3C. XQuery 1.0 and XPath 2.0 data model, Oct. 2005. `http://www.w3.org/TR/query-datamodel/`.

[18] Extensible markup language (XML) 1.0. W3C Recommendation, Feb. 2004. `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[19] XPath 2.0. W3C Working Draft, Oct. 2005. `http://www.w3.org/TR/xpath20`.

[20] XQuery 1.0: An XML query language. W3C Working Draft, Oct. 2005. `http://www.w3.org/TR/xquery/`.

[21] XML schema part 1: Structures. W3C Recommendation, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.