The Data Access Protocol: DAP Version 4.0

Volume 1: Data Model and Persistent Representation

|  |  |
| --- | --- |
| Date: | May 31, 2012 |
| Revised: | May 31, 2012 |
| Status: | Draft |
|  |  |
| Authors: | John Caron (Unidata) |
|  | Ethan Davis (Unidata) |
|  | David Fulker (Opendap) |
|  | James Gallagher (Opendap) |
|  | Dennis Heimbigner (Unidata) |
|  | Nathan Potter (Opendap) |
|  |  |
| Copyright: | 2012 University Corporation for Atmospheric Research |
|  | and Opendap.org |
|  |  |

Abstract

This document defines the Data Access Protocol (DAP) version 4.0 (referred to also as DAP4). This data transmission protocol is intended to supersede all previous versions of the DAP protocol. DAP4 is designed specifically for science data. The protocol relies on the widely used and stable standards, and is capable of representing a wide variety of scientific data types.

Distribution of this document is unlimited.

This document takes material from the DAP2 specification and the OPULS Wiki page.

|  |  |
| --- | --- |
| Changes: |  |
| 2012.05.24: | Initial Draft |
| 2012.05.27 | Added specification of chunk order |
| 2012.05.28 | Added specification and interpretation of simple queries |
| 2012.05.28 | Added discussion about nested sequences. |
| 2012.05.29 | Formatting changes |

Notes on decisions made or that need to be made.

1. Added Bit and Char to the list of Atomic types.
2. Added atomic type aliases: Boolean=Bit, Byte=UInt8
3. Variables are distinguished from Fields and array variables must be Variables.
4. Nested Attributes are not supported.
5. Opaque instances are variable length. If not, then we need to consider adding a Bytestring type.
6. Should we use term Cardinal type versus Atomic type.
7. Enumerations have a basetype that is one of the integer atomic types.
8. Persistent representation is based on

<http://docs.opendap.org/index.php/DAP4:_DAP4_On_the_Wire_Format>. Almost certainly will need modification after we try to implement it.

1. Added 2-byte packed representation to XDR.
2. Nested Sequences are supported using keys.

Contents

[1. Introduction 6](#_Toc326151482)

[2. Requirements 6](#_Toc326151483)

[3. Overall Operation 6](#_Toc326151484)

[4. Characterization of a Data Source 7](#_Toc326151485)

[4.1 Non-Data Bearing Declarations versus Data Bearing Declarations 8](#_Toc326151486)

[4.2 Groups 8](#_Toc326151487)

[4.3 Dimensions 8](#_Toc326151488)

[Semantic Limitations 9](#_Toc326151489)

[4.4 Enumeration Types 9](#_Toc326151490)

[4.5 Atomic Types 9](#_Toc326151491)

[4.5.1 Integer Types 9](#_Toc326151492)

[4.5.2 Floating-point Types 10](#_Toc326151493)

[4.5.3 String Types 10](#_Toc326151494)

[4.5.4 Enumeration Types 11](#_Toc326151495)

[4.5.5 Opaque Types 11](#_Toc326151496)

[4.5.6 A Note Regarding Implementation of the Atomic Types 11](#_Toc326151497)

[4.6 Container Types 12](#_Toc326151498)

[4.6.1 Structure 12](#_Toc326151499)

[4.6.2 Sequence 12](#_Toc326151500)

[4.6.3 Sequence Keys 12](#_Toc326151501)

[4.7 Variables 14](#_Toc326151502)

[4.7.1 Arrays 14](#_Toc326151503)

[4.7.2 Simple Variables 14](#_Toc326151504)

[4.7.3 Container Variables 15](#_Toc326151505)

[4.7.4 Maps 16](#_Toc326151506)

[4.8 Attributes and Arbitrary XML 17](#_Toc326151507)

[4.8.1 Attributes 17](#_Toc326151508)

[4.8.2 Arbitrary XML content 18](#_Toc326151509)

[4.8.3 Attribute and OtherXML Specification and Placement 18](#_Toc326151510)

[4.8.4 Fully Qualified Names 18](#_Toc326151511)

[4.9 Namespaces 18](#_Toc326151512)

[5. Data Representation 19](#_Toc326151513)

[5.1 Value Encoding 19](#_Toc326151514)

[5.2 Binary Format 20](#_Toc326151515)

[5.3 Chunk Header Format 20](#_Toc326151516)

[5.4 Chunk Order 22](#_Toc326151517)

[5.5 Data Format 22](#_Toc326151518)

[5.6 Instance Format 22](#_Toc326151519)

[5.6.1 Atomic Typed Instances 22](#_Toc326151520)

[5.6.2 Container Instances 25](#_Toc326151521)

[5.6.3 Sequence Chunk 25](#_Toc326151522)

[5.6.4 Variable Length Dimension Annex (V-Annex) 25](#_Toc326151523)

[5.6.5 String Annex 26](#_Toc326151524)

[5.7 Error Responses 26](#_Toc326151525)

[6. Constraints 27](#_Toc326151526)

[6.1 Minimal Query Language 27](#_Toc326151527)

[6.1.1 Simple Query Interpretation 28](#_Toc326151528)

[6.2 Extended Query Language 31](#_Toc326151529)

[6.2.1 Server Side Namespaces 31](#_Toc326151530)

[References 32](#_Toc326151531)

[Appendix A. DAP4 Lexical Elements 33](#_Toc326151532)

[A.1 DDX Lexical Element Syntax 33](#_Toc326151533)

[A.2 DDX Simple Query Lexical Elements 35](#_Toc326151534)

# Introduction

This specification defines the protocol referred to as the Data Access Protocol, version 4.0 (“DAP4”). In this document ‘DAP’ refers to DAP4 unless otherwise noted.

DAP is intended to be the successor to all previous versions of the DAP (specifically DAP version 2.0). The goal is to provide a very general data model capable of representing a wide variety of existing data sets. The DAP builds upon a number of existing data formats. Specifically, it is influenced by DAP version 2.0[], netCDF-4[], HDF5[], and CDM[].

The DAP is a protocol for access to data organized as variables, which are arrays of typed data. It is particularly suited to accesses by a client computer to data stored on remote (server) computers which are networked to the client computer. DAP was designed to hide the implementation of different collections of data. The assumption is that a wide variety of data sets using a wide variety of data formats can be translated to the DAP format for transmission from the server holding that dataset to a client computer for processing.

It is important to stress the discipline neutrality of the DAP and the relationship between this and adoption of the DAP in disciplines other than the Earth sciences. Because the DAP is agnostic as relates to discipline, it can be used across the very broad range of data types encountered in oceanography - biological, chemical, physical and geological. There is nothing that constrains the use of the DAP to the Earth sciences.

# Requirements

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [].

# Overall Operation

The DAP is a stateless protocol that governs clients making requests from servers, and servers issuing responses to those requests. This section provides an overview of the requests and responses (i.e. the messages) which DAP-compliant software MUST support. These messages are used to request information about a server and data made accessible by that server, as well as requesting data values themselves.

The DAP uses two responses to represent a data source. One response, the DDX returns metadata information describing the structure of a request for data. That is, it characterizes the variables, their datatypes, names and attributes. The second response, the DataDDX, returns both the metadata about the request, but also the data that was requested. The DDX and the metadata part of the DataDDX are represented using a specific XML[] representation. The syntax of that representation is defined in Section ?.

The DAP returns error information using an Error response. If a request for any of the three basic responses cannot be completed then an Error response is returned in its place.

The two responses (DDX and DataDDX) are complete in and of themselves so that, for example, the data response can be used by a client without ever requesting either of the two other responses. In many cases, client programs will request the DDX before requesting the DataDDX, but there is no requirement they do so and no server SHALL require that behavior on the part of clients.

Operationally, communication between a DAP client and a DAP server uses some underlying already existing protocol. It SHALL be the case that all clients and all servers support the use of at least HTTP as that underlying protocol. The use of other underlying protocols may be negotiated between the client and server using the negotiation sequence described in Section ?.

The request consists of a HTTP GET request method using a Uniform Resource Identifier (URI)[3] that encodes information specific to the DAP (see Section ?). This GET request contains a HTTP protocol version number followed by a MIME-like message containing various headers that further describe the request. In practice, DAP clients typically use a third-party library implementation of HTTP/1.1 so the GET request, URI and HTTP version information are hidden from the client; it sees only the DAP Uniform Resource Locator (URL) and some of the request headers. The DAP server responds with a status line that includes the HTTP protocol version and an error or success code, followed by a MIME-like message containing information about the response and the response itself. The DAP response is the payload of the MIME-like HTTP response. Unless otherwise negotiated, the response payload is encoded in multpart-MIME format. This is further described in Section ?.

In addition to these data objects, a DAP server MAY provide additional “services” which clients may find useful. For example, many DAP-compliant servers provide HTML-formatted representations or ASCII representations of a data source’s structure and data. Such additional services are beyond the scope of this document.

# Characterization of a Data Source

The DAP characterizes a data source as a collection of variables, dimensions, and enumeration types. Each variable consists of a name, a type, a value, and a collection of Attributes. Dimensions have a name and a size. Enumerations list names and values of the enumeration constants. These elements may be grouped into collections using the concept of a “group” that has an identifier and defines a naming scope for the elements within it. Groups may contain other groups.

The distinction between information in a variable and in an Attribute is somewhat arbitrary. However, the intention is that Attributes hold information that aids in the interpretation of data held in a variable. Variables, on the other hand, hold the primary content of a data source.

Section ? provides a formal syntax for DAP DDX characterizations. It is defined using the RelaxNG standard [] for describing the context-free syntax of a class of XML documents, the DDX in this case. The following discussion closely follows that RelaxNG syntax specification. It should be noted that any syntax specification requires a specification of the lexical elements of the syntax. The XML specification [] provides most of the lexical context for the syntax, but there are certain places where additional lexical elements must be used. Section ? describes those additional lexical elements, and those elements are discussed at appropriate points in the following discussion.

Since the syntax is context-free, there are semantic limitations on what is legal in a DDX. These semantic limitations are defined at appropriate places in the following documentation. It should also be noted that if there are conflicts between what is described here and the RelaxNG syntax, then the syntax takes precedence.

## Non-Data Bearing Declarations versus Data Bearing Declarations

The declarations in a DDX can be grouped into two classes. One class is non-data bearing. That is, it provides syntactic or structural metadata about a dataset. The non-data bearing declarations are Groups, Dimensions, and Enumerations. Such declarations do not contain data values themselves. In many cases these declarations will not be explicitly represented in the original dataset. Instead, their existence and value(s) will be inferred based on various standards and conventions. The data bearing class of declarations are Variables and Attributes. These elements of the data model are used to house data values or semantic metadata read from the dataset (or, in the latter case) synthesized from the values and standards/conventions that the dataset is known to follow.

## Groups

A group is specified using this XML form.

|  |
| --- |
| <Group name=“name”>  …  </Group> |

A group defines a name space and contains other DAP elements. Specifically, it can contain groups, variables, dimensions, and enumerations. The fact that groups can be nested means that the set of groups in a DDX form a tree structure. For any given DDX, there exists a root group that is the root of this tree.

A nested set of groups defines a variety of name spaces and access to the contents of a group is specified using a notation of the form “/g1/g2/…/gn”. This is called a “path”. By convention “/” refers to the root group. Thus the path “/g1/g2/g3” indicates that one should start in the root group, move to group g1 within that root group, then to group g2 within group g1, and finally to group g3.

For comparison purposes, DAP groups correspond to netCDF-4 groups and not to the more complex HDF5 Group type.

Semantic Limitations

1. If declared, Groups must be named. This includes the root group, but that group the name is ignored for the purposes of fully qualified names.
2. A Group can contain any object, including a Group
3. Each Group declares a new lexical scope for the objects it contains.
4. A Group cannot be an Array, Structure or Sequence. That is, a Group cannot be used with type Objects with FQNs.

## Dimensions

A dimension declaration is specified using this XML form.

|  |
| --- |
| <Dimension name=“name” size=“size”/> |

A dimension declaration will be referenced elsewhere in the DDX by specifying its name. It should also be noted that anonymous dimensions also exist. They have a size but no name. Anonymous dimensions do not need to be declared. Additionally, as discussed in Section ?, a dimension may be of variable length, and such a variable length dimension is indicated using the notation “\*” for the size.

### Semantic Limitations

1. Dimension declarations are not associated with a data type.
2. Dimension sizes MUST be a 64-bit integer.

## Enumeration Types

An enumeration type defines a set of names with specific values: enumeration constants. As will be seen in Section ?, enumeration types may be used as the type for variables or attributes. The values that can be assigned to such typed objects must come from the set of enumeration constants.

An Enumeration type is declared using this XML form.

|  |
| --- |
| <Enumeration name=“name” basetype=”atomic type”>  <EnumConst name=“name” value=”value”/>  …  </Enumeration> |

Semantic Limitations

1. The optional “basetype” XML attribute defines the type for the value XML attribute of each enumeration constant. This basetype must be one of the integer types (see Section ?). If unspecified, then it defaults to the Atomic type “Int32”.

## Atomic Types

As their name suggests, atomic data types are conceptually indivisible. Atomic variables are used to store integers, real numbers, strings and URLs. There are five classes of atomic types, with each family containing one or more variations: integer types, floating-point types, string types, enumerations, and opaque types.

### Integer Types

The integer types are summarized in Table 1. The lexical structure for integer constants is defined in Appendix ?.

1. The DAP Integer Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Boolean | Single bit integer | [0, 1] |
| Bit | Synonym for Boolean |  |
| Int8 | Signed 8-bit integer | [-(2^7), (2^7) - 1] |
| UInt8 | Unsigned 8-bit integer | [0, (2^8) - 1] |
| Byte | Synonym for UInt8 |  |
| Int16 | Signed 16-bit integer | [-(2^15), (2^15) - 1] |
| UInt16 | Unsigned 16-bit integer | [0, (2^16) - 1] |
| Int32 | Signed 32-bit integer | [-(2^31), (2^31) - 1] |
| UInt32 | Unsigned 32-bit integer | [0, (2^32) - 1] |
| Int64 | Signed 64-bit integer | [-(2^63), (2^63) - 1] |
| UInt64 | Unsigned 64-bit integer | [0, (2^64) - 1] |

### Floating-point Types

The floating point data types are summarized in Table 2. The two floating point data types use IEEE 754[11] to represent values. The two types correspond to ANSI C’s float and double data types. The lexical structure for floating point constants is defined in Section ?.

1. The DAP Floating-Point Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Float32 | 32-bit Floating-point number | [±1.175494351 × 10^38,  ±3.402823466 × 10^38] |
| Float64 | 64-bit Floating-point number | [±2.2250738585072014 × 10^308, ±1.7976931348623157 × 10^308] |

### String Types

The three string data types are summarized in Table 3. Again, the lexical structure for these is defined in **Error! Reference source not found.**.

Strings are individually sized. This means that in constructor data types containing multiple instances of some String, such an array, successive instances of that String MAY be of different sizes.

Note that the Char type is defined to be 7-bit US-ASCII embedded in an 8-bit byte with a zero high order bit. This means that it can represent only a subset of UTF-8.

1. The String Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Char | 8-bit US-ASCII character with the high order bit zero. | [\x00, \x7f] |
| String | A variable length string of UTF-8 characters | As defined in [] |
| URI | A Uniform Resource Identifier | As defined in IETF RFC 2396[3] |

### Enumeration Types

An enumeration type specifies a set of named, 32-bit integer constants. See Section ?.

When a data source has a variable of type 'Enumeration' a DAP 4 server MUST represent that variable using an *integer type*, up to an including a 64-bit unsigned integer. However, in practice, these should use *Int32* variables when transporting the values unless an enumeration contains values too large for that type. This is true because DAP4 will use XDR to encode responses and thus Arrays of Enumerations will encode directly to single byes. If we use other types, like Int16, then they will expand to be 32-bit integers. On the other hand, a single Enumeration will expand to a 32-bit integer for encoding by XDR, but that cost is fairly small.

### Opaque Types

The XML format for declaring an Opaque type is as follows.

|  |
| --- |
| <Opaque … > |

The Opaque type is use to hold objects like JPEG images and other Binary Large Object (BLOB) data that have significant internal structure which might be understood by clients (e.g., an image display program) but that would be very cumbersome to describe using DAP's built-in types. Defining a variable of type “Opaque” does not communicate any information about its content, although an attribute could be used to do that.

Semantic Limitations

1. The content of an opaque object is completely un-interpreted by the DAP4 implementation. There is no attempt to re-order four-byte words to or from network byte order and there is no attempt to modify its actual length to conform to, for example, a four-byte boundary, although when transmitted on the wire, padding may be added.
2. The Opaque type is an Atomic Type, which might seem odd because instances of Opaque can be of different sizes. However, by thinking of Opaque as equivalent to a byte-string type, the analog with strings makes it clear that it should be an Atomic type.

### A Note Regarding Implementation of the Atomic Types

When implementing the DAP, it is important to match information in a data source or read from a DAP response to the local data type which best fits those data. In some cases an exact match may not be possible. For example Java lacks unsigned integer types [13]. Implementations faced with such limitations MUST ensure that clients will be able to retrieve the full range of values from the data source. As a practical consideration, this may be implemented by hiding the variable in question or returning an error.

## Container Types

The container types (also called constructor types) provide a way to build new data types by composing existing types. There are two container types: Structures and Sequences. A Structure type can contain both atomic and other Structure types subject to certain semantic limitations as listed in this document. In principle, there are no restrictions on the number of levels or types of nesting of the Structure types.

### Structure

A Structure groups fields so that the collection can be manipulated as a single item. A field is syntactically identical to a variable. The Structure’s fields MAY be of any type, including Structure types. The order of items in the Structure is significant only in relation to the persistent representation of that Structure.

Semantic Limitations

1. Structure Fields types may not be Sequences.

### Sequence

The Sequence type is syntactically similar to a Structure. It is intended to correspond to a relation in a relational database. The fields of the Sequence declaration correspond to the columns of a relation. An instance of a Sequence has some number of rows corresponding to the rows of some database relation. Two different Sequence declarations need not have the same number of rows, and depending on the constraints applied (Section ?) instances of a given Sequence declaration need not have the same number of rows.

A Sequence row can best be described as an instance of a Structure. Each row consists of the same set of fields (columns), but may contain different values.

Although a Sequence is primarily intended to represent a database relation, a server is free to represent other kinds of datasets as Sequences as long as it obeys the semantic constraints on Sequences and is prepared to implement the necessary constraint operations defined for Sequences (Section ?).

Semantic Limitations

1. Sequences MUST be defined at the top-level of a group. Sequence declarations MUST not be used to define fields of a Structure or other Sequence.
2. Sequence Fields types may not be Sequences.
3. With respect to queries (Section ?) and selection, only fields that are undimensioned and have an Atomic type can appear in such selections.

### Sequence Keys

In order to support certain the inter-linking of Sequences, it is necessary to define concepts of keys and foreign keys as fields of Sequences. This feature supports, among other things, nested sequences.

A Sequence may define Key fields or ForeignKey fields. Key fields are define using this XML format.

|  |
| --- |
| <Key type=”atomic type” name=”name” /> |

ForeignKey fields are defined using this XML format.

|  |
| --- |
| <ForeignKey key=”path to key field”/> |

Consider this example.

|  |
| --- |
| <Sequence name="SQ1">  <Key name="f1" type="Int32"/>  <Float32 name="fx"/>  ...  </Sequence>  <Sequence name="SQ2">  <ForeignKey key="/SQ1.f1"/>  <Float32 name="fy"/>  ...  </Sequence> |

A ForeignKey has the same semantics as is defined in traditional relational database theory. It specifically indicates how two Sequences can be combined (effectively using join) based on the <ForeignKey> element in one Sequence pointing to a <Key> element in another Sequence.

The <Key> element in the example in SQ1 is equivalent to the following field.

|  |
| --- |
| <Int32 name="f1"/> |

as far as its role as a field in the Sequence.

Other Sequences can refer to that key using a <ForeignKey> element. As far as fields go, defining a foreign key implicitly includes the key being referenced (f1 in our example) as a field in the Sequence.

So, the above example is equivalent to the following.

|  |
| --- |
| <Sequence name="SQ1">  <Int32 name="f1"/>  <Float32 name="fx"/>  ...  </Sequence>  <Sequence name="SQ2">  <Int32 name="f1"/>  <Float32 name="fy"/>  ...  </Sequence> |

Once we have keys and foreign keys, it is easy to represent two Sequences as if they were a nested Sequence. So, our example above could be presented to a user as the following equivalent nested Sequence.

|  |
| --- |
| <Sequence name="SQ1">  <Int32 name="f1"/>  <Sequence name="SQ2">  <Int32 name="fy"/>  </Sequence>  </Sequence>  </Sequence> |

Note that the common key field (f1) only appears in the outer Sequence because its existence with the same value is implicit in the nesting.

## Variables

Each variable in a data source MUST have a name, a type and one or more values. Using just this information and armed with an understanding of the definition of the DAP data types, a program can read any or all of the information from a data source.

The DAP variables come in several different types. There are several atomic types, the basic indivisible types representing integers, floating point numbers and the like, and two constructor types (also called container types) which are flexible collections of other variables. Constructor types may contain both atomic variable types as well as other constructor types.

The DAP variables describe the data when it is being transferred from the server to the client. It does not necessarily describe the format of the data inside the server or client. The DAP defines, for each data type described in this document, a persistent representation, which is the information actually communicated between DAP servers and DAP clients. The persistent representation consists of two parts: the declaration of the type and the encoding of its value(s). The data representation is presented in Section ?.

### Arrays

Most (but not all) types may be arrays. An Array is a one-dimensional indexed data structure similar to that defined by ANSI C. An Array’s member variable MAY be of any DAP data type. Array indexes MUST start at zero.

Multidimensional Arrays are defined as Arrays of Arrays. Multi-dimensional Arrays MUST be stored in row-major order (as is the case with ANSI C). Except for variable length dimensions, the size of each Array’s dimensions MUST be given. The total number of elements in an Array MUST NOT exceed 2^64-1. There is no prescribed limit on the number of dimensions an Array may have except that the foregoing limit on the total number of elements MUST NOT be exceeded. The number of elements in an Array is fixed as that given by the size(s) of its dimension(s), except when the array has a variable length dimension.

Semantic Limitations

1. Simple variables (see below) may be arrays.
2. Structure MAY be arrays.
3. Sequences MUST NOT be arrays.

### Simple Variables

A simple, dimensioned variable is declared using this XML form.

|  |
| --- |
| <Int32 name=”name”>  <Dimension name=”name”/>  …  <Dimension size=”integer”/>  …  <Dimension size=”\*”/>  </Int32> |

A simple variable is one whose type is one of the Atomic Types (see Section 1). The name of the Atomic Type (Int32 in this example) is used as the XML element name. Within the body of that element, it is possible to specify zero or more dimension references. A dimension reference may refer to a previously defined dimension declaration. It may also define an anonymous dimension with no name, but with a size. It may also define a variable length dimension using a size of “\*”.

Semantic Limitations

1. When declaring a variable, only one variable length dimension may be referenced, and that variable length dimension must be the last dimension listed.
2. Variables must be “top-level”, which means that they are declared immediately within groups. This is to distinguish them from “fields”, which look syntactically like variables, but are declared in Structures or Sequences.

### Container Variables

As with simple variables, a container variable specifies a type as well as any dimension for that variable. The type, however, is either a Structure or a Sequence. These types are called container types.

#### Structures

The XML format for a Structure typed variable is as follows.

|  |
| --- |
| <Structure name=”name”>  {structure body}  …  <Dimension name=”name”/>  …  </Structure> |

The Structure contains within it a “structure body”, which is define in Section ?. The structure body may be followed with a list of dimension references indicating the dimensions of the Structure typed variable.

Semantic Limitations

1. Structures MAY be dimensioned.

#### Sequences

Sequences are syntactically similar to Structures, except the XML form is as follows.

|  |
| --- |
| <Sequence name=”name”>  {sequence body}  </Sequence> |

The fields (also called columns) of a sequence are divided into two kinds.

* Selectable – a selectable column is any Atomic-typed non-dimensioned field of a sequence. Only selectable columns can be used in a *whereclause* (Section ?).
* Blob – all non-selectable columns are called blob columns.

Semantic Limitations

1. Sequences MUST NOT be dimensioned.
2. Sequences MUST be declared as a variable. This means that they are top-level within a group and cannot be declared within other Structures or Sequences.

### Maps

A common dataset concept is that of a Grid. A Grid is an association of an N dimensional Array with N vectors. These N vectors are referred to as coordinate variables or maps.

[Need a good definition of coordinate variables.]

Using OGC coverage terminology, we have this.

1. The maps specify the ''Domain''
2. The array specifies the ''Range''
3. The Grid itself is a ''Coverage'' per OGC.
4. The Domain and Range are sampled functions

A map is defined using the following XML format.

|  |
| --- |
| <Map name=”FQN for some variable defined in the DDX” /> |

An example might look like this.

|  |
| --- |
| <Float32 name=”A”>  <Dimension name=”lat”/>  <Dimension name=”lat”/>  <Map name=”lat”/>  <Map name=”lon”/>  </Float32> |

Where the map variables are defined elsewhere like this.

|  |
| --- |
| <Float32 name=”lat”>  <Dimension name=”lat”/>  </Float32  <Float32 name=”lon”>  <Dimension name=”lon”/>  </Float32> |

The containing variable, A in the example, will be referred to as the “array variable”.

Semantic Limitations

1. Each map variable MUST have a rank no more than that of the array.
2. An array variable can have as many maps as desired.
3. Every named dimension mentioned in the map variables must appear in the set of dimensions of the array variable
4. The dimensions of the array variable may not contain duplicates so A[x,x] is disallowed.
5. Any map duplicates are ignored and the order of declaration of the maps is irrelevant.
6. A Map variable may not have a variable length dimension.
7. A Map may only nominate a top-level variable as a Map variable. That is, <MAP> may not refer to a field of a Structure or Sequence.

## Attributes and Arbitrary XML

### Attributes

Attributes are defined using the following XML format.

|  |
| --- |
| <Attribute name=”name” type=”atomic type name”>  <Namespace space=http://netcdf.ucar.edu/cf</Namespace>  <Value value=”value”/>  …  <Value value=”value”/>  </Attribute> |

In DAP4, Attributes (not to be confused with XML attributes) are tuples with four components:

Name

Type

Vector of values

One or more Namespaces (optional)

This differs slightly from DAP2 Attributes because the *namespace* feature has been added, although clients can choose to ignore it. For more about namespaces, refer to Section ?. The intent of including the namespace information is to simplify interactions with semantic web applications where certain formats or standards have formal definitions of attributes.

Attributes are typically used to associate semantic metadata with the variables in a data source. Attributes are similar to variables in their range of types and values, except that they are somewhat limited when compared to those for variables.

Attributes defined at the top-level within a group are also referred to as “global attributes”.

While the DAP does not require any particular Attributes, some may be required by various metadata conventions. The semantic metadata for a data source comprises the Attributes associated with that data source and its variables[14]. Thus, Attributes provide a mechanism by which semantic metadata may be represented without prescribing that a data source use a particular semantic metadata convention or standard.

If an attribute in a particular data source (e.g. an HDF5 file) is a multi-dimension Array, it is suggested that the Attribute be promoted to a variable and that a new Attribute be created for that variable which describes the promotion. This fits the paradigm of remote access better since the multi-dimensional array information would then be accessed with a constraint expression. Since constraint expressions can only be applied to variables, it makes sense to promote such data to a variable.

Semantic Limitations

1. DAP4 explicitly treats an attribute with one value as an attribute whose value is a one-element vector.
2. The following types are allowed for Attributes:
3. All of the Atomic types are allowed as the type for an attribute
4. String typed Attributes use UTF-8 encoding and Char typed attributes use US-ASCII encoding.
5. Attribute value constants MUST conform to the appropriate constant format for the given attribute type and as defined in Appendix ?.

### Arbitrary XML content

By supporting an explicit type to hold “arbitrary XML” markup, DAP4 provides a way for the protocol to transport information encoded in XML along with the attributes read from the dataset itself. This has proved very useful in work with semantic web software.

In an XML representation of DAP4, the name is optional, the XML element is *<OtherXML/>* and there are no *<value/>* elements because the “other xml” appears as the content of the *<OtherXML/>* element. The value of the attribute must be valid XML and must be distinct from the XML markup used to encode elements of the DAP4 data model (i.e., in a practical sense, the <OtherXML> must be in a namespace other than DAP4).

### Attribute and OtherXML Specification and Placement

Attribute and OtherXML declarations MAY occur within the body of the following XML elements: Group, Dimension, Variable, Field, Structure, Sequence.

### Fully Qualified Names

Every object in a DAP4 Dataset has a Fully Qualified Name (FQN). These names follow the common conventions of lexically-scoped identifiers. To write and FQN for some object O, locate the closest, top-level, enclosing object (P) for O. P may be the same as O. Start by creating the FQN for P by traversing a path through the Group tree to P. Concatenate the group names on that path and separating them with ‘/’. The root group is assumed to have no name, hence the FQN will begin with “/”. The FQN for P will end with the name of P. If O is a field nested in some set of (possibly nested) set of Structures, Sequences, or Enumerations, then collect a field pathname from P to O by concatenating the names on that path and separated by dots (“.”). The last name in the field pathname is the names of P. Prefix the field pathname with “.”. Concatenate the FQN with the field pathname for O to create the final FQN for O.

The forward slash character is never legal as a name. Cases where dots are used in names are accommodated by allowing dots to be escaped using a backslash (\).

## Namespaces

All elements of the DDX, Groups, Dimensions, Variables, and Attributes can contain an associated Namespace element. The namespace’s value is defined in the form of an XML style URI string defining the context for interpreting the element containing the namespace. Suppose, hypothetically, that we wanted to specify that an Attribute is to be interpreted as a CF convention []. One might specify this as follows.

|  |
| --- |
| <Attribute name=”latitude”>  <Namespace space=”http://cf.netcdf.unidata.ucar.edu”/>  …  </Attribute> |

Note that this is not to claim that this is how to specify a CF convention; this is purely hypothetical.

# Data Representation

Data can be an elusive concept. Data may exist in some storage format on some disk somewhere, on paper somewhere else, in active memory on some server, or transmitted along some wire between two computers. All these can still represent the same data. That is, there is an important distinction to be made between the data and its representation. The data consist of numbers: abstract entities that usually represent measurements of something, somewhere. Data also consist of the relationships between those numbers, as when one number defines a time at which some quantity was measured.

The abstract existence of data is in contrast to its concrete representation, which is how we manipulate and store it. Data can be stored as ASCII strings in a file on a disk, or as twos-complement integers in the memory of some computer, or as numbers printed on a page. It can be stored in netCDF, HDF5, GRIB, JGOFS, a relational database and any number of other digital storage forms.

The DAP specifies a particular representation of data, to be used in transmitting that data from one computer to another. This representation of some data is sometimes referred to as the persistent representation of that data, to distinguish it from the representations used in some computer’s memory. The DAP standard outlined in this document has nothing at all to say about how data is stored or represented on either the sending or the receiving computer. The DAP transmission format is completely independent of these details.

## Value Encoding

Atomic values in the persistent representation are encoded using a variant subset of the XDR standard []. The XDR format specifies the use of network byte order (big-endian) for encoding integers and floating point numbers. The encoding defined in this DAP standard uses what is termed “receiver-makes-it-right”. This means that a server may encode data in either big-endian or little-endian and provides a flag (Section ?) indicating the byte order that was used. The client code is then responsible for converting from the server specified order to the native byte-order of the client machine.

The DAP4 persistent encoding uses only the following XDR encodings.

1. 4-byte integer
2. 4-byte floating-point
3. 8-byte floating point
4. Counted packed strings
5. Counted packed bytes

In addition, the following new encodings are used.

1. Counted 2-byte integer arrays
2. 8-byte integer

The first of the new encodings supports counted, packed vectors of 2-byte integers analogous to the way that XDR currently transmits packed vectors of bytes. As with the byte case, any such packed array will be padded to make its total size be a multiple of 4 bytes.

## Binary Format

The reply returned by a Server must conform to the following binary format. This is independent of any additional headers and trailers required by the transport format, HTTP for example.

The complete persistent {response} consists of a sequence of *chunks* or *parts*. Each chunk is prefixed with the chunk header defined in the following Section. Following the chunk, there are N bytes of data, where N is defined in the chunk header.

The data of the first chunk contains the DDX encoded as a UTF-8 XML document.

The data of the second chunk contains a concatenated vector of *instances* representing the variables that are being returned in the reply to a request.

Additional chunks contain either Sequence records or Variable-Dimension vectors.

Each chunk, except the DDX, is followed immediately by a *string* *annex*, which contains the actual instances of variable-length objects: string and opaque (byte-string) constants referenced by the some other chunk.

Finally, the last chunk has no data and signals the end of a reply.

## Chunk Header Format

Each chunk is prefixed with the following binary header.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 8 | Integer | Response Unique ID |
| 8 | 8 | Integer | Length |
| 16 | 2 | Integer | MIME Type/Subtype |
| 18 | 2 | Integer | Unused |
| 20 | 2 | Short | Typecode |
| 18 | 2 | Short | Version |
| 24 | 2 | Integer | Per-Response Flags |
| 26 | 2 | Integer | Per-Chunk specific Flags |
| 30 | 8 | Integer | Chunk Unique ID |
| 38 | 4 | Integer | Piece Count (when a chunk is transmitted in several pieces) |
| 42 | 4 | Integer | Piece index (0 <= index < Piece count) |

All fields of the header MUST be encoded using network byte order (big-endian).

The fields of the header are defined as follows.

Response ID

All chunks that are part of the same response will have a common 64-bit unique id.

Length

The Length field will be of value N, where N is the number of bytes in the data payload.

MIME Type/Subtype

This field encodes, as an integer, the MIME type and subtype that define the encoding of the data part. This supports, for example, sending the DDX using formats other than XML – such as Json [] or Protobuf []. Where applicable, the character set is always assumed to be UTF-8.

The currently defined type encodings are as follows.

|  |  |
| --- | --- |
| MIME Type Name | Encoding |
| application/octet-stream | 1 |
| text/xml | 2 |

Typecode

The Typecode field hold an unsigned integer value equivalent to the following enumeration constants.

* ddx = 1
* dataset = 2
* sequence = 3
* v-annex = 4
* string-annex = 5
* error = 3
* stop = 4

Response Flags

The Response Flags field holds 16 1-bit flags. The value of this field must be the same for every chunk in the response. The current set of defined flags is as follows.

* Little-endian=1 – this flag indicates that the data payload of each chunk MUST use little-endian representation. If not set, then big-endian (network byte order) MUST be used.

Chunk Flags

The Chunk Flags field holds 16 1-bit flags. The value of this field MAY be different for each chunk within a response. The current set of defined flags is as follows.

* Invalid=1 – this flag indicates that, for whatever reason, this chunk should be considered invalid and that it will be followed by an Error chunk. The Length field of the invalid chunk MUST be valid so that locating the following error chunk can occur in the normal fashion.

Chunk ID

Each chunks within a response has a unique 64-bit integer identifier.

Piece Count and Piece ID

A given chunk may actually be transmitted in multiple pieces, each of which has a “piece id” that ranges from zero up to (but not including) the value in the Piece Count field. The complete chunk can be reconstructed by removing all but the first header and concatenating all of the pieces and computing a new Length field. All such chunks will have the same Chunk ID.

Following the binary header, there are N bytes of data, sometimes called the “payload”, where N is the value specified in the Length field in the header.

## Chunk Order

The order and occurrence of chunks in a reply MUST conform to the following grammar, where “reply” is the start symbol for the grammer.

|  |
| --- |
| reply: ERROR | ddx ERROR| ddx STOP ;  ddx: DDX dataset  dataset: /\*empty\*/ | DATASET annexes sequences ;  sequences: /\*empty\*/ | sequences sequence ;  sequence: SEQUENCE annexes ;  annexes: /\*empty\*/ | V-ANNEX | STRING-ANNEX  | V-ANNEX STRING-ANNEX ; |

The upper cased names refer to chunk types.

There are some things to note.

1. The first chunk MUST either be a DDX chunk and the last chunk must either be a STOP chunk or an ERROR Chunk.
2. An ERROR chunk MAY appear at any point after the DDX chunk and it terminates the reply.
3. The chunks, DDX, SEQUENCE, etc, MAY be represented as a sequence of pieces (Section ?) that are sent in order sorted in increasing order by piece id.

## Data Format

Broadly speaking the data payload of a chunk consists of a concatenated vector of data *instances*, typically one for each DAP4 variable returned by the server in response to a request. An instance, however, may contain other instances – the fields of a structure, for example – and those nested instances must conform to the same format as the top-level instances.

As currently defined, the data part is intended to include only information that cannot be deduced from the DDX. This means, for example, that arrays of, say, integers are not prefixed with any length information because it can be deduced from the DDX. This is one of a number of ways in which this format differs from the DAP version 2 encoding.

## Instance Format

Where applicable, all data in the instances is encoded as either little-endian or bit-endian as specified in the chunk header.

The specific representations are defined in subsequent Section.

### Atomic Typed Instances

#### Scalar Representation

For the atomic types, scalar instances are represented as follows (before any endian conversion).

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Representation |
| Boolean/Bit | Single bit integer | 1 bit zero extended to 32-bits |
| Int8 | Signed 8-bit integer | 8 bits sign extended to 32 bits |
| UInt8 | Unsigned 8-bit integer | 8 bits zero extended to 32 bits |
| Int16 | Signed 16-bit integer | 16 bits sign extended to 32-bits |
| UInt16 | Unsigned 16-bit integer | 16 bits zero extended to 32-bits |
| Int32 | Signed 32-bit integer | 32-bit |
| UInt32 | Unsigned 32-bit integer | 32-bit |
| Int64 | Signed 64-bit integer | 64-bit |
| UInt64 | Unsigned 64-bit integer | 64-bit |
| Float32 | 32-bit IEEE floating point | 32-bit |
| Float32 | 32-bit IEEE floating point | 64-bit |
| Char | US-ASCII character with zero high-order bit | Treat same as UInt8 |
| String | Varying length vector of UTF-8 characters | UInt64 offset into the following string annex followed by a UInt32 length. |
| Opaque | Varying length vector of Unsigned 8-bit bytes | UInt64 offset into the following string annex followed by an UInt32 length. |
| URL | Uniform Resource Locator in the form of a UTF-8 string. | Same as the String type |

In narrative form, all scalar instances of atomic types whose size is smaller than 32 bits are extended to a length of 32 bits. Unsigned types (including Boolean) are zero extended and signed types are sign extended.

All 32 bit quantities (after extension) are then represented as an unsigned 32-bit value. This means, for example, that the IEEE format of a float32 is ignored and it is treated as if it was a simple, unsigned 32-bit value.

All 64-bit values are treated as if they were 64-bit unsigned integers.

Instances of string, opaque, and URL types are represented as a 64 bit offset into the associated string annex, and followed by a 32 bit length indicating the number of bytes in the instance. The length field is actually redundant because (Section ?) it is repeated in the string annex.

Scalar Endian Conversion

When a reply is received and the “endian-ness” of the sender differs from that of the receiver, the values of the received data must be converted to the receiver’s endian-ness. The rules are as follows when the endian-ness differs.

For 32 bit quantities, the 4 bytes of the received data are swapped as follows.

1. byte 0 -> byte 3
2. byte 1 -> byte 2
3. byte 2 -> byte 1
4. byte 3 -> byte 0

For 64-bit quantities, the received 8 bytes are swapped as follows.

1. byte 0 -> byte 7
2. byte 1 -> byte 6
3. byte 2 -> byte 5
4. byte 3 -> byte 4
5. byte 4 -> byte 3
6. byte 5 -> byte 2
7. byte 6 -> byte 1
8. byte 7 -> byte 0

There is one exception: 64-bit floating point. In order to be consistent with XDR, it is necessary to treat the 64-bits of the value as two 32-bit values. Each value must be byte-swapped and then the order of the two 32-bit values must be interchanged.

#### Array Representation

In the following, the term “element count” is used. The element count is the cross product of all the dimension sizes for the array. If the last dimension is variable length, then it is not used in the cross product. So, for example, A[2][3][4] has an element count of 2x3x4 = 24. An array of rht form A[2][3][\*] has an element count of 2x3 = 6 and the variable length dimension is ignored.

For the atomic types, instances that are arrays of values instances are represented as follows (before any endian conversion).

1. Any type whose size is less than 32-bits is represented as a concatenation of N values, where N is the element count. This is called “packing” because the values are not converted to 32-bit values before transmission. The order of concatenation is defined by the dimensions taken in row-major order. All packed arrays are padded with arbitrary byte values so that the total length of the packed data is a multiple of 4 bytes (the same as XDR packed arrays).
2. Any type whose size is 32-bits is represented as the concatenation of N scalar values, where N is as defined in #1. The order of concatenation is defined by the dimensions taken in row-major order.
3. Any type whose size is 64-bits is represented as the concatenation of N scalar values, where N is as defined in #1. The order of concatenation is defined by the dimensions taken in row-major order.
4. String, opaque, and URL types are represented as a concatenation of N scalar values, where N is as defined in #1. The order of concatenation is defined by the dimensions taken in row-major order.

### Container Instances

#### Structures

A scalar structure instance is the concatenation of the instances for each field as described in this section. Thus, a structure represents an instance that in turn is composed from its field instances.

A dimensioned structure instance consists of the concatenation of the instances taken in row-major order.

#### Sequences

The sequence instance consists of a single, unsigned 64-bit integer which is the chunk id (Section ?) of the chunk containing the actual records of the sequence.

### Sequence Chunk

Each sequence instance consists of some set of *records*. The number of records in the sequence is only known at the time the response is sent. So, each Sequence instance in a dataset chunk has its own separate chunk to hold the records of the sequence.

The data in a sequence chunk begins with the following header.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 8 | Integer | Number of records in the chunk |

The header is followed by the concatenation of zero or more record instances. Each record instance is, much like a structure instance, the concatenation of the instances for each field as described in this section. Like a structure instance, a record instance represents in turn is composed from its field instances.

### Variable Length Dimension Annex (V-Annex)

Consider a variable or field that is dimensioned with a variable length dimension as its last dimension. Its instance is stored as a vector of instances of the following form.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 8 | Integer | Offset into the v-annex |

The number of such instances is equal to the cross-product of the dimensions of a field or variable excluding the variable-length dimension. Consider this example.

|  |
| --- |
| <Int32 name=”v”>  <Dimension size=”5”/>  <Dimension size=”3”/>  <Dimension size=”\*”/>  </Int32> |

The number of “pointers into the v-annex will be 5\*3= 15. The v-annex will then contain 15 vectors of type Int32 and each of arbitrary length.

The chunk header for the v-annex has the v-*annex* typecode.

A v-annex begins with the following header.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 8 | Integer | Chunk id of the associated data chunk |

Following that, there is a concatenation of instances of the following format.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 4 | Integer | typecode |
| 4 | 8 | Integer | Count of instances in this occurrence of the  variable-length dimension |
| 12 | sizeof(instance)  \* N | byte | Vector of N instances N is the  value of the Count field |

If an instance itself contains fields with variable length dimensions, then those variable length dimensions are themselves part of this same v-annex. If this v-annex contains references to strings or opaque instances, then those instances are in the string annex for the chunk to which this v-annex is associated.

### String Annex

Each dataset and sequence chunk MAY be associated with what is termed here a “string annex.

The chunk header for the annex has the *string-annex* typecode.

A string annex begins with the following header.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 8 | Integer | Chunk id of the associated data chunk |

Following that, there is a concatenation of instances of the following format.

|  |  |  |  |
| --- | --- | --- | --- |
| Offset (Bytes) | Size (Bytes) | Type | Description |
| 0 | 4 | Integer | Typecode |
| 4 | 8 | Integer | Length (N) in bytes of the string/opaque |
| 12 | N | byte | Vector of N bytes where N is the  value of the Count field |
| 12+N | P = 0-3 | byte | P padding bytes such that (12+N+P) % 4 = 0 |

Each such instance contains the information about a string value: its length and its content plus any necessary padding.

The string annex holds the content of all the strings and opaque constants referenced in that associated chunk part. If the string annex is empty, then it need not be transmitted.

## Error Responses

An unsuccessful DAP4 request will cause the server to return a DAP4 error response. The error response may be returned in lieu of the Dataset response or may occur as a chunk at any point in a response. If an error chunk is detected, then it also implies the end of the response. If in the process of sending a part, an error occurs that causes the content of the part to be considered invalid, the malformed part should, if feasible, set the invalid flag in the chunk header. The next received part will be an error response.

The data part of the Error chunk is an XML document whose syntax is defined in Appendix ?. The document MUST contain at least the following information.

1. An error code (an integer) – provides a short characterization of the error. The currently defined set of error codes is as follows. At least the following types of DAP4 errors MUST be supported.

|  |  |  |
| --- | --- | --- |
| Code | Characterization | Semantics |
| 1 | Internal Error | The error is internal to the Server. Some examples of this are: a programming bug/issue, out of memory, disk failure. |
| 2 | User Syntax Error | The query expression is syntactically malformed. The server should return a message in the error object that explains where in the constraint expression the problem was detected. |
| 3 | Not Found Error | The requested resource cannot be found. |

1. An error message – text providing a more detailed characterization of the error.

In addition, the error response MAY contain the following information.

* Positional information – where appropriate, this provides a "pointer" into some text document that shows where the error was detected. Possible documents include the DDX and the constraint expression. The positional information will include, as appropriate, a document reference, (optionally) a line number, and a character number either within the line or within the document as a whole.
* Context information – provide additional information that will help to isolate the error and its cause. A java stack trace would be an example of this.
* Other information – Any other arbitrary information encoded as XML that is thought useful by the error provider.

# Constraints

It is important to define a minimal request language – a query language – to select information from a dataset on a server and obtaining in response a DDX and data corresponding to that request. The query language must encompass three classes of computations.

* The query language MUST provide operators for selecting subsets of a DDX and a dataset.
* The query language …
* The query language …

## Minimal Query Language

This section defines the syntax and semantics of the minimal request language that MUST be supported by all implementations. The method by which a server is provided with a query is specified in Volume 2. But as a typical example, if such a query were to be embedded in a URL, then it is presumed that it is prefixed with a “?” and is appended to the end of the URL.

The syntax of the minimal query language, also referred to as the “simple query” language, is as follows. [Note: this minimal language does not support server-side functions]

|  |
| --- |
| simplequery: /\*empty\*/  | constraintlist ;  constraintlist: constraintlist “,” constraint ;  constraint: arraysubset | relational\_query ; |

### Simple Query Interpretation

Each constraint in the list of constraints comprising a simple query is of one of two types: Array Subsetting and Relational queries. [Note: is a combination of both types possible?]

#### Array Subsetting

The grammar for an array subsetting constraint is as follows. It is a variant of the fully qualified name (Appendix ?).

|  |
| --- |
| arraysubset: grouppath | ‘/’ structpath ;  grouppath: groupname | grouppath groupname ;  groupname: ‘/’ NAME ;  structpath: grouppath range rangepath ;  rangepath: /\*empty\*/ | rangepath ‘.’ NAME range ;  range: /\*empty\*/ | range dimrange ;  dimrange: ‘[‘ INTEGER ‘]’  |‘[‘ INTEGER ‘:’ INTEGER ‘]’  |‘[‘ INTEGER ‘:’ INTEGER ‘:’ INTEGER ‘]’; |

Interpretation

Consider the following Array.

|  |
| --- |
| Int32 A[d1][d2]…[dn] |

where all of the dimension sizes di are integers. Consider the following array subset query.

|  |
| --- |
| A[start1:stride1:endn]… [startn:striden:endn] |

Where

|  |
| --- |
| for i=1 .. n, starti < di & endi < di & starti < endi & starti >= 0 & stridei >= 1 & endi >= 0 |

The query selects the elements A[i1][i2]…[in] from A where ii is in the set {starti+stridei\*j} where

|  |
| --- |
| j=0 .. k such that starti+stridei\*k <= endi and starti+stridei\*(k+1) > endi |

#### Relational Query

Relational queries are operations on one or more Sequence objects. The relational query format is intended to map directly to restricted form of the SQL query language [].

Specifically, it is intended to map to an SQL query of this form.

|  |
| --- |
| Select <columns>  From <relations>  Where <predicate> |

For DAP, a column is a reference to a field in a Sequence and a relation is a reference to a Sequence in a DDX. In light of this, the grammar for a relational query is as follows.

|  |
| --- |
| relquery: selectclause | selectclause ‘|’ whereclause ;  selectclause: ‘[’ columnpathlist‘]’;  columnpathlist: columnpath | columnpathlist columnpath ;  whereclause: predicate ;  predicate: conjunction | predicate ‘|’ conjunction ;  conjunction: negation | conjunction ‘&’ negation ;  negation: term | ‘!’ term ;  term: ‘(‘ predicate ‘)’ | comparison | BOOLEAN ;  comparison: primary operator primary ;  operator: ‘==’ | ‘!=’ | ‘>=’ | ‘<=’| ‘<’ | ‘>’ ;  primary: INTEGER | FLOAT | STRING | COLUMNPATH ; |

Note that the “where” keyword is replaced by the bar character (“|”) and should not be confused with the alternative case symbol in the grammar.

The set of sequences (relations) of interest can be inferred from the set of sequences referenced in the *columnpath*s. Each *columnpath* is the same as same form as an fully qualified name (FQN) as shown in Appendix Section ?.

Semantic Limitations

1. The *sequence-set* for the query are all those Sequences mentioned in the *columnpathlist*.
2. The *columnpaths* in the *whereclause* are restricted to refer to *selectable* (Section ?) columns from the *sequence-set*.

Interpretation

A simple query is intended to map directly to a specific form of SQL statement. The semantics of the query is then the same as the semantics of the SQL statement as defined in the SQL specification [].

Specifically, the translation is as follows. Consider the following simple query.

|  |
| --- |
| [C1,C2,…Cn]|P |

The Ci are paths to columns in one or more Sequences. Let {S} be the set of sequences referenced in any of the Ci. Let P be a predicate as defined above. Let {Q} be the set of Sequences mentioned in any column path in the predicate. It MUST be the case that {S} is a subset of {S}.

Given this simple query, it is translated to an SQL statement of the following form.

|  |
| --- |
| Select <columns>  From <relations>  Where <predicate> |

The pieces of this SQL statement are defined as follows. The <columns> is the set {C1,C2,…Cn}, <relations> is the set {Q}, and <predicate> is the same as P with operators translated as needed to make P be a proper SQL predicate.

A Note on Nested Relations

Supporting some equivalent of nested sequences is desirable because it provides a natural representation for certain datasets such as trajectories of trajectories.

The key concept is that a column (field) of a Sequence may be itself a Sequence. Consider this example.

|  |
| --- |
| <Sequence name="SQ1">  <Int32 name="f1"/>  <Float32 name="fx"/>  <Sequence name="SQ2">  <Float32 name="fy"/>  ...  </Sequence>  </Sequence> |

There is an implicit join column that connects an instance of SQ1 with an instance of SQ2. Asking for field such as SQ1.SQ2.fy in effect joins SQ1 and SQ2 on some hidden column(s) common to both. The specific field fy is obtained by then projecting out column SQ2.fy from the join. Without some kind of syntactic addition, there is actually no way to explicitly get the join column(s) of SQ1 or SQ2.

If a request for all of SQ1 were to be sent to a server, the server would have to return a sequence of records for SQ1. Each such record would in turn contain a pointer to the corresponding sequence of SQ2 records that “match” the implicit join field of SQ1.

How is the equivalent of this to be supported in the model specified in this document? The answer is that the nested relations are flattened into two relations and <Key> and <ForeignKey> elements are used to control the join of these flattened relations.

Consider this alternate example representing a flattened version of the previous, nested example.

|  |
| --- |
| <Sequence name="SQ1">  <Key name="f1" type="Int32"/>  <Float32 name="fx"/>  </Sequence>  <Sequence name="SQ2">  <ForeignKey key="/SQ1.f1"/>  <Float32 name="fy"/>  </Sequence> |

In this representation, making a request for all of SQ1 would return a single sequence that was the join of SQ1 and SQ2 on the explicit key field. This would look like this.

|  |
| --- |
| [SQ1.f1, SQ1.fx, SQ2.fy] | SQ1.f1 == SQ2.f1 |

The set of records returned contain the same information as the nested sequence data structure with some extra redundancy because of duplicated field values across records: SQ1.fx is the redundant information in this example.

## Extended Query Language

An extended query language is defined based on nested functions combined with single assignment variables. A semantically nonsensical, but grammatical example would look something like this.

|  |
| --- |
| svc("cmd");$x=f("string17",g(h(12))),f2($x,p[0:3:10]) |

Extended queries subsume simple queries. The grammar for an extended query is, notionally, as follows, where *simplequery* and constraint are as defined above.

|  |
| --- |
| Extended\_query: simplequery | complexquery ;  constraintlist: constraintlist “,” constraint ;  complexquery: programs  | complexquery ';' program ;  program: term  | program ',' term ;  term: expression  | localvariable '=' expression  expression: functioncall  | constraint  | localvariable  | STRING  functioncall: FUNCTIONNAME '(' argumentlist ')'  argumentlist: expression | argumentlist expression |

In narrative form, a query is a semi-colon separated list of “programs”. A program is a comma separated list of “terms”. A term is either an “expression” or an expression assigned to a local variable. An expression is either a function call or a variable reference or a constraint or a string. Each function has an arbitrary number of argument expressions separated by commas. Local variables are single-assignment, which means they can appear on the left side of a term only once within a query.

### Server Side Namespaces

Functions reside on the server. In order to support an orderly extension of the set of server-side functions, it is necessary to create a namespace mechanism for the server side similar to that provided by groups on the client side. That is, there will be standard pre-defined functions, server-specific functions, and even dynamically defined functions existing on the server. These functions must be contained in some tree of nested namespaces. This means that each function has a fully qualified name (similar to the Group-based fully qualified names of Section ?). As with Groups, we assume that function fully qualified names are defined as function-names separated by a special separator character. The chosen separator in this specification is the dot (“.”) character. For convenience, function-names that have no leading “.” are assumed to be in the pre-defined root namespace. That is, “name” is equivalent to “.name”.

[Note, I choose dot as the separator because I do not want confusion with the client-side namespace; but not really happy with dot as the separator.]

References

1. DAP4 Lexical Elements
   1. DDX Lexical Element Syntax

This section describes the lexical elements that occur in the DAP4 DDX.

Within the RelaxNG DAP4 grammar (Section ?) there are markers for occurrences of primitive type such as integers, floats, or strings (ignoring case).

|  |
| --- |
| <attribute name="namespace">  <datatype="string"/>  </attribute> |

The markers typically look like this when defining an attribute that can occur in the DAP4 DDX.

The "<data type="string"/>" specifies the lexical class for the values that this attribute can have. In this case, the namespace attribute is defined to have a string value. Similar notation is used for values occurring as text within an xml element.

The lexical specification later in this section defines the legal lexical structure for such lexical items. Specifically, it defines the format of the following lexical items.

1. Constants, namely: string, float, integer, character, opaque, and Boolean.
2. Identifiers
3. Fully qualified names (also referred to as FQN) (see Section ?).

The specification is written using the extended Posix regular expression notation [] with some additions.

1. Names are assigned to regular expressions using the notation “name = <regularexpression>”
2. Named expressions can be used in subsequent regular expressions by using the notation “{name}”. Such occurrences are equivalent to textually substituting the expression associated with name for the “{name}” occurrence. This is similar to the way a macro operates.

Note that a regular expression name must be defined before any use to avoid circular definitions.

Notes:

1. The definition of {UTF8} is deferred to the next section.
2. Comments are indicated using the "//" notation. Standard xml escape formats (&x#DDD; or &<name>;) are assumed to be allowed anywhere.
   * 1. Basic character set definitions

|  |
| --- |
| CONTROLS = [\x00-\x1F] // ASCII control characters  WHITESPACE = [ \r\n\t\f]+  HEXCHAR = [0-9a-zA-Z]  // ASCII printable characters  ASCII = [0-9a-zA-Z !"#$%&'()\*+,-./:;<=>?@[\\\]\\^\_`|{}~] |

* + 1. Ascii characters that may appear unescaped in Identifiers

This is assumed to be basically all ASCII printable characters except these characters: '.' '/' '"' ''' and '&'. Occurrences of these characters are assumed to be representable using the standard xml &<name>; notation (e.g. &amp;). In this expression, backslash is interpreted as an escape character.

|  |
| --- |
| IDASCII=[0 9a zA Z!#$%()\*+:;<=>?@\[\]\\^\_`|{}~] |

* + 1. The Numeric Constant Classes: integer and float

|  |
| --- |
| INTEGER = {INT}|{UINT}|{HEXINT}  INT = [+-][0-9][0-9]\*{INTTYPE}?  UINT = [0-9][0-9]\*{INTTYPE}?  HEXINT = {HEXSTRING}{INTTYPE}?  INTTYPE = ([BbSsLl]|"ll"|"LL")  HEXSTRING = (0[xX]{HEXCHAR}{HEXCHAR}\*)  FLOAT = ({MANTISSA}{EXPONENT}?)|{NANINF}  EXPONENT = ([eE][+-]?[0-9]+)  MANTISSA = [+-]?[0-9]\*\.[0-9]\*  NANINF = (-?inf|nan|NaN) |

* + 1. The Boolean Constant Class

|  |
| --- |
| BOOLEAN = [01]|true|True|TRUE|false|False|FALSE |

* + 1. The String Constant Class

|  |
| --- |
| STRING = ([^"\\&]|{XMLESCAPE})\*  CHAR = ([^'\\&]|{XMLESCAPE}) |

* + 1. The Opaque Constant Class

|  |
| --- |
| OPAQUE = 0x([0-9A-Fa-f] [0-9A-Fa-f])+ |

* + 1. The Identifier Class

|  |
| --- |
| ID = {IDCHAR}{IDCHAR}\*  IDCHAR = ({IDASCII}|{XMLESCAPE}|{UTF8})  XMLESCAPE = [&][#][0-9]+; |

* + 1. The Atomic Type Class

|  |
| --- |
| ATOMICTYPE = Boolean | Bit | Char | Byte  | Int8 | UInt8 | Int16 | UInt16  | Int32 | UInt32 | Int64 | UInt64  | Float32 | Float64  | String | URL  | Enumeration | Opaque |

This list should be consistent with the atomic types in the grammar.

* + 1. The Fully Qualified Name Class

|  |
| --- |
| FQN = ([/]{ID})+([.]{ID})\* |

This should be consistent with the definition in Section ?.

* + 1. Lexical Class Precedence

Note that the above lexical element classes are not disjoint. The type element “<datatype=…/>” should be sufficient to interpret the type within the DDX.

* + 1. UTF-8

The UTF-8 specification, <http://www.w3.org/2005/03/23-lex-U>, defines several ways to validate a UTF-8 string of characters.

The full (most correct) validating version of UTF8 character set is as follows.

|  |
| --- |
| UTF8 = ([\xC2-\xDF][\x80-\xBF])  | (\xE0[\xA0-\xBF][\x80-\xBF])  | ([\xE1-\xEC][\x80-\xBF][\x80-\xBF])  | (\xED[\x80-\x9F][\x80-\xBF])  | ([\xEE-\xEF][\x80-\xBF][\x80-\xBF])  | (\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])  | ([\xF1-\xF3][\x80-\xBF][\x80-\xBF][\x80-\xBF])  | (\xF4[\x80-\x8F][\x80-\xBF][\x80-\xBF]) |

The lines of the above expression cover the UTF-8 characters as follows:

1. non-overlong 2-byte
2. excluding overlongs
3. straight 3-byte
4. excluding surrogates
5. straight 3-byte
6. planes 1-3
7. planes 4-15
8. plane 16

Note that ASCII and control characters are not included.

The above reference also defines some alternative regular expressions.

There is what is termed the partially-relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6][\x80-\xBF])  | ([\xE0-\xEF][\x80-\xBF][\x80-\xBF])  | ([\xF0-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]) |

Second, there is what is termed the most-relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6]...)|([\xE0-\xEF)...)|([\xF0 \xF7]...) |

Any conforming DAP4 implementation MUST use at least the most-relaxed expression for validating UTF-8 character strings, but MAY use either the partially-relaxed for full validatation expression.

* 1. DDX Simple Query Lexical Elements

The simple query syntax (Section ?) requires defining the several lexical elements. The lexical elements BOOLEAN, INTEGER, FLOAT, and STRING are defined as in Appendix Section ?.

The remaining item, COLUMNPATH is similar to an FQN, but with additional escaping requirements and disallowing of the XML “&..;” notation.

|  |
| --- |
| QASCII = [0 9a zA Z#$%\*+:?@^\_`{}~]  QESCAPE = [!./"'|;&=\[\]<>()\\]  QIDCHAR = {QASCII}|[\\]{QESCAPE}|{UTF8}  QID = {QIDCHAR}+  COLUMNPATH = ([/]{QID})+([.]{QID})\* |

1. DAP4 Error Syntax

|  |
| --- |
| <grammar xmlns="http://relaxng.org/ns/structure/1.0" |
| xmlns:doc="http://www.example.com/annotation" |
| datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" |
| ns="http://xml.opendap.org/ns/DAP/4.0#" |
| > |
| <start> |
| <ref name="error"/> |
| </start> |
| <define name="error"> |
| <element name="Error"> |
| <attribute name="errorcode"><data type="integer"/></attribute> |
| <element name = "Message"><text/></Message> |
| <optional> |
| <interleave> |
| <element name = "Position"><text/></Message> |
| <element name = "Context"><text/></Message> |
| <element name = "OtherInformation"><text/></Message> |
| </interleave> |
| </optional> |
| </element> |
| </define> |

1. DAP4 DDX Syntax