Oracle Binary XML Format

Draft 21 March 2007

Sivasankaran Chandrasekar
Sam Idicula
Tim Yu
Nipun Agarwal
Email: {firstname.lastname}@oracle.com

Status of this Memo

This document is written in the style of an IETF RFC, for potential submission to that standards body at some point in the future.

Abstract

This document specifies an optimized format for XML content. It is designed to exploit XML Schema information when available. This document also specifies the abstract protocol operations for managing the relevant metadata as well as transfer of encoded XML content.

This document is a product of Oracle Corporation's XML development groups in the Server Technologies division. Comments on this draft are welcomed, and should be addressed to Nipun.Agarwal@oracle.com. This document is a result of the Binary XML working group, and reflects the contribution of all the team members of this group, including Ravi Murthy, Eric Sedlar, Dmitry Lenkov, K Karun, Anjana Manian, Olga Peschanksy and Alex Yiu.

Change Log

Jan 2, 2009	Sam Idicula	Final draft
March 21, 2007	Sivasankaran Chandrasekar	Created public draft
May 20, 2005	Ravi Murthy	Internal draft

Table of Contents

1 INTRODUCTION

This document presents the Oracle Binary XML Format, also referred to as the Compact Schema Aware XML Format (CSX), a binary serialization of an XML infoset. This format significantly improves performance of XML processing, bringing it to the performance level of other existing technologies such as object-oriented or relational. This format is suitable for native use by the entire enterprise platform technology stack.

1.1 Terms

This draft uses the terms defined in XML Schema Parts 1 and 2 [XMLS1] and [XMLS2], the extensible markup language v1.0 [XML1], XQuery Data Model [XDM] and the document object model [DOM]. In addition, the following terms are defined:

CSX Processor

A CSX processor is capable of handling CSX format, providing the ability to transform it into XML Text [XML1] or make available DOM and SAX APIs onto the XML Infoset. Note that a CSX processor will often be used in scenarios transmitting XML over the network. In addition, it may provide support for permanent storage for XML Schemas and be contacted to ask for schema information or for pieces of instance documents. Some CSX processors may both originate or receive network protocol requests for CSX information.

Section

This is the unit of transfer for CSX data. Any XML document or fragment can be encoded a CSX section. In addition, a section can also represent a nested element within a XML document. In case of sections corresponding to nested elements, a section reference is used to provide a link from the parent section to the child section. The ability to create sections at the level of elements has many benefits including lazy DOM, faster ingestion by the database, etc.

Doc-ID

Each document (or XML infoset) that is encoded in CSX is optionally identified by a unique Doc-Id. It is either a 16-byte GUID or some opaque sequence of bytes e.g. URL.

Path-ID

A unique identifier (scoped to a CSX processor and typically 4-8 bytes) corresponding to a simple path expression (e.g. /a/b, /c/d/@foo) consisting of element (or attribute) names and child axes. Note that the element and attribute names are QNames. The path-id is typically used as a key to look-up storage and processing metadata for the current node.

Order key

A unique identifier to every element node in a document based on its position. The bytes comprising the (variable-sized) order key preserve document ordering i.e. byte comparison and byte sorting of order keys imply document ordering. The order key is based on the "dewey" numbering system and consists of a sequence of order-numbers – one per level – where each order-number provides document ordering of the node within its parent i.e. among its siblings.

E.g. assuming that order key is composed of integral order-numbers, the order key of 3.2.5 can represent the 5th child of the 2nd child of the 3rd child of the root. The order key is a "data" key and is typically used in conjunction with the path-id to locate and look-up the node data. See Appendix A for the algorithm used to encode dewey order keys. Note that the order key does not take into account, extras such as PIs and comments.

Node (section) reference

A reference to an external section of CSX data.

Collection (section) reference

A reference to a list of external sections corresponding to a set of sibling nodes (in document order).

Type-ID

A unique identifier for a global simple or complex type defined in an XML schema.

1.2 Notational Conventions

The augmented BNF used by this document to describe protocol elements is described in Section 2.1 of [RFC2616]. Because this augmented BNF uses the basic production rules provided in Section 2.2 of [RFC2616], those rules apply to this document as well.

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 [RFC2119].

Definitions of XML elements in this document use XML Schema.

2 REQUIREMENTS

CSX must be suitable for the following applications:

- MUST efficiently support conversion to and from XML text
- SHOULD be friendly to database operations such storage, updating, and indexing and make the format similar to a database relational row format for the same information model
- SHOULD be friendly to schema evolutions that maintain the validity of the document
- The size MUST be significantly smaller than XML text
- The CPU cost to load the infoset into memory MUST be significantly better than text XML parsing
- Fragment extraction MUST be possible using an index to an offset within the CSX format and reading forward sequentially
- MUST be datatype-aware: conversions to/from string formats may cause loss of precision, and involve performance overhead.
- MUST support partial loading of the XML infoset (which may be loaded on-demand)
- MUST support shredding the XML into smaller sections which can be independently processed and SHOULD minimize the overhead for a single section, since this will translate into per-row overhead when databases store one section per row
- SHOULD be possible to resolve XPaths that correspond to database column references with similar processing time to scanning a row in a database to find a particular column
- SHOULD support streaming evaluation of simple Xpaths
- MUST support XML with or without a schema
- SHOULD keep XML data in document order (by default). This is necessary to provide efficient support for applications which process XML data in document order.
- SHOULD minimize the cost to skip forward through stream when looking for an end-of-element token.

- SHOULD minimize CSX processor state needed to locate a fragment within a section.

3 CSX PROCESSOR

The following gives an overview of the functionality provided by a CSX processor. A CSX processor can:

- convert the CSX stream to and from XML 1.x text, DOM, or SAX events
- process data incrementally, and only the data needed (lazy processing)
- determine what schema is associated with a document (via schemaLocation tags, configuration files, an OASIS catalog, etc.), fetching it and optionally caching it

A CSX processor encodes XML into the CSX format based on the results of XML parsing, but also uses XML schemas to provide additional semantics allowing for more efficient processing. The use of schema information to encode requires that the document be at least partially validated with respect to the schema. Since schema validation is known to be an expensive operation, it may be desirable to do only partial validation in the case when the document is not known to be valid for other reasons. Hence the CSX stream encoded using a schema always implies at least partial validity (as defined below). In addition, if the data is known to be completely valid with respect to the schema, the CSX stream also stores this information.

Partial validity is defined to be full schema validity with the following exceptions:

- No validation of unique/key/keyref/ID/IDREF constraints
- No validation of pattern facets

In addition, a CSX processor may provide the following additional facilities:

- Schema management and identification
- Persistent storage of registered schemas
- Managing dependencies between schemas
- Finding schemas using a schema ID
- Schema evolution

CSX Processors are identified via a unique identifier derived using the GUID algorithm.

4 SCHEMA MANAGEMENT

4.1 SCHEMA IDENTIFICATION

Each schema is identified by an opaque binary identifier (schema ID) and a version number. The schema ID is scoped to the CSX processor and MUST remain constant even when the schema is evolved. The schema version number is distinct from the "version" tag defined by the W3C schema spec, as the schema version is required to be an integer. Furthermore, schemas managed by a CSX processor are required to be backwards compatible, so that any document that validates with a particular schema MUST validate with a later version of that schema. In other words, for any schema with ID=X and version=Y and a schema with ID=X and version=Z, and where $Z > Y$, all documents valid with respect to the first schema MUST validate with the later schema. The converse (where $Z < Y$) will not usually be true.

4.2 SCHEMA REGISTRATION

The client can register a schema with the CSX processor by providing the XML Schema document. The server registers the provided schema and returns the schema ID along with a fully annotated schema document.

The client can also evolve a previously registered schema by providing the new XML Schema document and its associated schema ID. The server responds with the new version number and the fully annotated schema document. The Doc-ID of the schema document is the same as the schema ID. The version of the schema document is represented by a system-level annotation csx:version as discussed below.

XML schema documents which are transferred between client and server are encoded in schema-less compact XML format i.e. using token (QName) IDs.

4.3 SCHEMA LOOKUP

A client can lookup the XML schema using its schema ID and version number. In this case the server returns the fully annotated schema corresponding to the specified parameters. If the client omits the version number, the server returns the latest version of the schema. The XML schema documents that are transferred between client and server are encoded in schema-less compact XML format i.e. using token (QName) IDs.

When a client needs to encode an input text XML document, it has to lookup the appropriate XML Schema. In general, the server may have specific rules to map instance documents to schemas, based on the namespace and name of root element, xsi:schemaLocation value (if provided) and the current user. A client can request the server for the entire set of schema mappings, or only those relevant to a given namespace and/or user. Prior to encoding the input XML document, the client uses the schema mapping information to determine the appropriate schemaID, fetches the annotated schema (if needed) and uses it to encode the document.

Each schema mapping consists of XML namespace, root-element-name, schema-location-value, user-name and the corresponding schemaID. Any of the identifiers can be left unspecified i.e. as an empty element and indicates a wildcard. If more than one mapping are applicable (due to wildcards), the client always chooses the more specific mapping (i.e. one without wildcards). Also, ties are broken using the precedence in the following order :- schema-location, namespace, element-name, user.

Example of schema mapping that maps a given namespace to a schemaID regardless of the root element name, schema location value or current user:

```
<csx:schemaMapping>
    <csx:schemaLocation/>
    <csx:namespace>http://www.example.com</csx:namespace>
    <csx:rootElement/>
    <csx:user/>
    <csx:schemaID>5678999</csx:schemaID>
</csx:schemaMapping>
```

The support for schema mappings on the server-side is optional i.e. the server can choose to return the exact schema (document) matching the client inputs. In this case, the client must assume that the match is exact i.e. the returned schema is applicable only for the current settings of schema-location, namespace, element-name and user.

4.4 SCHEMA ANNOTATIONS

Schema annotations refer to both attributes and elements in csx namespace "http://xmlns.oracle.com/2004/CSX" that appear within a XML schema document. Note that csx

elements can only appear within <xsd:appInfo> elements. There are two categories of schema annotations:

1. **User-level Annotations** are specified as hints by the user during schema registration and influence encoding type and sectioning criteria
2. **System-level Annotations** are added by the CSX processor during schema registration. These annotations cannot be explicitly specified or overridden by the user.

When a client looks up a schema from the CSX processor, it receives a fully annotated schema which contains all applicable user-level annotations (could be different from those specified by user during registration) and system-level annotations.

The following is the list of user-level schema annotations.

4.4.1 encodingType (enumeration of CSX encoding datatypes)

This attribute indicates the datatype used for encoding the node value of this element or attribute. It can be used within xsd:attribute, xsd:element (for elements based on simpleType or simpleContent only) and xsd:simpleType elements.

If the user does not specify encodingType, the CSX processor picks the default encoding type based on the XML Schema datatype.

4.4.2 sections (element of type csx:sectionType)

By default, the entire document is encoded as a single section. However, users can explicitly indicate the section roots. The csx:sections element can only appear within xsd:schema->xsd:annotation->xsd:appInfo context. The element <csx:sections> specifies the sectioning criteria in terms of path expressions identifying section roots. The path expression must conform to a subset of XPath 1.0 and is an absolute path consisting of element names or wildcard and child or descendant axes. Predicates or other axes such as ancestor are not allowed. The namespaces for the element names within the path should be explicitly provided using the appropriate prefix from the current scope. The absence of a prefix within the path indicates null namespace.

Examples of valid section paths are /PurchaseOrder/LineItems, //Address, /a/*/b, //*

Example of annotated schema :

```
<xsd:schema…>
  <xsd:annotation>
    <xsd:appInfo>
     <csx:sections>
          <csx:section>
               <csx:path>/PurchaseOrder/LineItems</csx:path>
          </csx:section>
          <csx:section>
               <csx:path>//Address</csx:path>
          </csx:section>
      </csx:sections>
    </xsd:appInfo>
  </xsd:annotation>
…
</xsd:schema>
```

The path specified as the section root is applied in the context of the element which starts the scope of the specified schema. For example, the PurchaseOrder element may be nested within

another document. The outer document itself could be non-schema based or conforming to an entirely different schema with <any> declaration. The section path /PurchaseOrder/LineItems is applied in the context of the PurchaseOrder element.

The specification of sectioning criteria is an ***ordered*** list. If a node satisfies more than one of the section root paths, then the first specification is used. Though this choice does not affect the mere decision to section, it does impact the further use of processor-specific auxiliary information that may be specified within the context of <csx:section> element.

The following is the list of all system-level schema annotations.

4.4.3 version (unsignedInt)

The version number of the schema document. This number is incremented by the CSX processor when the schema document is evolved. See section 4.5 for schema evolution considerations.

4.4.4 propertyID (unsignedInt)

Each element and attribute in a schema is assigned a propertyID to uniquely identify that property in a CSX processor. The real requirement is that a property ID be unique within a set of schemas related via include/import; however, since at any future point in time another schema may be created including any others in a CSX processor, processor-wide uniqueness is required.

4.4.5 typeID (unsignedInt)

Each global simple or complex type defined in the schema must be assigned a typeID to uniquely identify the type in a CSX processor. See above for the need for CSX processor level uniqueness. When the type is explicitly referenced using xsi:type attribute, the dynamic type is encoded in CSX using the typeID. Even in cases when the type information is implicitly present (for example, an instance of XQuery data model), the type id is encoded in CSX stream.

The pre-defined xsd and xdt types are assigned reserved type-ids. See section on token management.

4.4.6 kidList (element of type csx:kidListType)

This system annotation appears within the context of a complexType declaration. It lists the possible children of the current element identified by their propertyID, and their assigned kidNum. The kidNum is a local identifier (short integer) for the child property within the context of the parent element and is typically smaller than the propertyID.

If the sequential mode optimization can be applied to the current element, csx:kidList also carries sequential=”true” attribute. In this case, the order of <csx:kid> elements is the same as that implied by the schema.

Example of csx:kidList schema annotation:

```
<xsd:element name=”Employee”><xsd:complexType>
  <xsd:annotation><xsd:appInfo>
    <csx:kidList sequential=”true”>
      <csx:kid propertyID=”3456” kidNum=”1”/>
      <csx:kid propertyID=”3457” kidNum=”2”/>
    </csx:kidList>
  </xsd:appInfo></xsd:annotation>
  <xsd:sequence>
    <xsd:element name=”Name” type=”xsd:string” csx:propertyID=”3456”/>
    <xsd:element ref=”Salary”/>
```

```
    </xsd:sequence>
</xsd:complexType></xsd:element>
```

4.5 SCHEMA EVOLUTION

The compact XML format of a XML document is significantly influenced by the XML Schema to which it conforms. In fact, several pieces of information contained within the XML Schema are exploited to optimize the CSX format. However, this implies that a *correct* XML Schema be available in order to decode the CSX values. The evolution of XML Schemas are constrained such that CSX instances encoded using the old XML Schema can still be decoded correctly using the new XML Schema.

The following logical constraints are enforced for determing legal XML Schema evolutions :

- The new XML Schema should validate a superset of the XML instances that were valid against the old XML Schema.
- None of the schema annotations that impact the CSX format can be modified.

The following is the list of schema evolutions that may be supported by CSX processors.

- Defining new schemas that import or include the current XML Schemas. This may trigger an "implicit" schema evolution because it may:
    - Add a new derived type of a base type defined in some earlier schema. This implies that previous defined "substitutable" elements of the base type are now substitutable by elements of the subtype.
    - Add a new member to the substitutionGroup formed by an earlier element.
- Adding an optional attribute to a complextype or attributegroup.
- Adding an optional element (i.e. effective minOccurs after unrolling the model is 0) to a choice, all, sequence or group model. Note that adding a new element to the middle of a sequence model may turn off certain optimizations for new instances. However the old instances can still be decoded correctly using the system annotations in the schema.
- Increasing the value of maxOccurs
- Adding values to enumeration simpleType at the end of the enumeration
- Increasing the value of maxLength facet
- Expand the range of allowed values ::: short -> int, float -> double
- Adding mixed="true"
- Converting element from simpletype to complextype with simplecontent (needed to introduce an attribute)

5 DTD MANAGEMENT

DTDs continue to be used by current applications – primarily for their ability to handle entity declarations. Though XML Schema provides a better mechanism for expressing structural and datatype information about XML documents, it does not have a good mechanism for handling entities. Entities provide a "macro" functionality by which users can define names for character and other text/xml units, and use them via reference within their XML documents. Since the definitions of these entities are kept in a central place (viz. DTD), it makes it easier to manage and evolve them. Also, the readability of the document is enhanced. The CSX processor and format need to support the following requirements :

- Support for encoding XML instances with internal DTDs
- Support transfer of DTD files between CSX processors

- Support for encoding and decoding XML instances referring to external DTD files
- Support encoding entity references directly within CSX stream instead of expanding them
- Support explicit modes during fetch for requesting entity expansion

5.1 Internal DTD

The DOCTYPE declaration within a XML document is encoded using fine-grained opcodes corresponding to element declarations, attribute list declarations, parsed entity (both general and parameter entities), unparsed entity and notation declarations.

5.2 External DTD

External DTDs are also assigned unique DTD IDs and are managed similar to XML schemas. When DTD documents are transferred between client and server, they are also encoded in CSX using the DTD-related opcodes (see section 8.5.4). If a document references an external DTD, the corresponding DTD ID is encoded in the section header.

5.3 Entity References

Entity references are encoded using ENTREF opcode whose operand is the entity name in textual form. Example: &foobar; is encoded as [ENTREF][6][foobar].

6 Token Management

One of the major compression techniques in CSX is the use of a token table (i.e. symbol table) to minimize space needed for repeated items in XML. This is especially useful in encoding XML documents that do not have an associated schema. CSX supports tokens of the following types:

Token Type	Defined By	Size of token ID
Namespace URL tokens	CSX processor	4/8 bytes
Element/Attr (qualified) name tokens	CSX processor	4/8 bytes
Path tokens	CSX processor	4/8 bytes
Prefix mapping tokens	Section	2 bytes
Property	Schema	4 bytes
Type	Schema	4 bytes

The namespace, element/attribute and path tokens are managed at the level of a CSX processor i.e. the CSX processor assigns unique token IDs to these tokens. Prefix mapping tokens are managed within a section as described below.

The CSX processor level token mappings can be summarized as follows:

Namespace URL ←→ Namespace Token ID
Namespace Token ID, Local Name, Node type (Element or Attribute)
←→ QName Token ID
Simple (Navigational) Path ←→ Path Token ID

6.1 CSX processor Level Token Registration and Lookup

A client can register a set of tokens with the CSX processor and retrieve their token IDs. Alternatively, the client can generate (local) token IDs (for example, using the same hashing scheme as the server) and send the token definitions also as part of the encoded CSX stream. The server will implicitly register the new tokens (if needed), and send back their real token IDs. This is referred to as inlined token definitions (see below).

A client can retrieve all token definitions belonging to a particular namespace by providing the namespace ID.

6.2 Inlined Token Definitions

Token definitions may appear in a CSX stream, assigning integer token IDs to each of these items. Token IDs must be unique within a particular document (although server implementations may choose to keep them unique for all documents conforming to a schema).

6.3 Differences from propertyIDs

Property IDs are defined by the schema, and correspond to a particular element/attribute declaration. Note that a schema may have a declaration for the same QName multiple times, by using local element/attribute declarations. Property IDs do not share the token ID space with QName token IDs i.e. a CSX processor may assign the same token ID value to both a property defined within a schema as well as a QName token. However, property IDs are always encoded as negative values within the CSX stream. Hence the same set of CSX opcodes can have either property IDs or token IDs as operands. Property IDs may NEVER change during schema evolution. Property IDs defined by a schema are the same regardless of whether or not those declarations are included in another schema. There is one property ID corresponding to each element and attribute declaration in the schema.

The property ID mappings can be summarized as follows.

Schema ID, Namespace ID, Local Name → Property ID (of global declaration)
Schema ID, Parent PropertyID, Namespace ID, Local Name → Property ID (of local declaration)
PropertyID → Namespace Token ID, Local Name, Schema Declaration

CSX may encode element and attributes using either QName or property ID tokens. However, property IDs are needed to exploit the schema information regarding structure (e.g. sequential mode), etc while encoding the CSX stream. Schema IDs of included or imported schemas have NO effect on the scoping of token IDs.

6.4 Prefix Mapping Tokens

The mapping of a prefix to a namespace is assigned an 2 byte prefix ID. The prefix ID is unique within a document. The definition of the prefix ID is always inlined in the encoded CSX stream (i.e. no out-of-band registration or lookup). The value of "0" for prefix ID is reserved and implies no prefix.

6.5 Reserved Tokens

The token IDs from 1 through 64 (in all token spaces) are reserved for use by tokens defined by this specification and subsequent revisions. Note that there are separate ID spaces for namespace URLs, element QNames, attribute QNames and prefix mappings.

Reserved Namespace URL Tokens:

Nmspc ID	URL value
1	http://www.w3.org/XML/1998/namespace
2	http://www.w3.org/XML/2000/xmlns/
3	http://www.w3.org/2001/XMLSchema-instance
4	http://www.w3.org/2001/XMLSchema
5	http://xmlns.oracle.com/2004/csx
6	http://xmlns.oracle.com/xdb
7	null namespace
8	http://www.w3.org/2001/XInlcude

Reserved Attribute QName Tokens:

Token ID	Namespace ID	Local name value
0x1-0xF	(reserved for internal attributes)	
0x10	1	space
0x11	1	lang
0x12	3	type
0x13	3	nil
0x14	3	schemaLocation
0x15	3	noNamespaceSchemaLocation
0x16	2	xmlns

Reserved Prefix IDs:

Prefix ID	Namespace ID	Prefix
1	1	xml
2	2	xmlns
3	3	xsi
4	4	xsd
5	4	xs
6	5	csx

Reserved TypeIDs

Type ID	Type Name
1	xs:string
2	xs:boolean
3	xs:decimal
4	xs:float
5	xs:double
6	xs:duration
7	xs:dateTime
8	xs:time
9	xs:date
10	xs:gYearMonth
11	xs:gYear
12	xs:gMonthDay
13	xs:gDay
14	xs:gMonth
15	xs:hexBinary
16	xs:base64Binary
17	xs:anyURI
18	xs:QName
19	xs:NOTATION
…	
	xdt:untypedAtomic
…	

7 DATATYPES

CSX uses the following list of encoding datatypes for node values. Each of the XML schema datatypes has a default encoding datatype. However, the user can override it to one of the other allowed datatypes by using the csx:encoding-type attribute within the appropriate element, attribute or simpleType declaration.

The actual encoding type information is also encoded within CSX. This allows schemas to be evolved by changing the datatype in a backward compatible fashion - such as changing from xsd:int to xsd:long, or xsd:dateTime to xsd:string. Since the encoding type is present within CSX data, such backward compatible schema changes have no effect on the existing CSX data.

Node values are always accompanied by a length. The first length byte also contains the encoding information as shown in the Format section.

List of CSX Encoding Datatypes

CSX Encoding Datatype	Description	Restrictions
string	UTF8 character data	None
binary	binary data	
boolean	one byte containing 1 or 0 for true/false	Though XML Schema allows "1" and "0" as lexical values, CSX is decoded always as "true" and "false".
int	signed two's-complement native integer format in big-endian ordering. (1/2/4/8 bytes)	
unsigned-int	Two's-complement native integer format in big-endian ordering (1/2/4/8 bytes)	
float	IEEE-754 floating point (4 or 8 bytes)	
oranum	CSX number format (maximum of 22 bytes) (See section 14.1)	Allows maximum precision of 38 digits and scale between -84 to 127. Allows numbers from 1.0×10^{-130} to (but not including) 1.0×10^{126}
oradate	CSX date format for Gregorian dates (7 bytes) (See section 14.2)	Does not allow fractional seconds. Does not allow timezone component.
orats	CSX timestamp format for Gregorian dates with timezone (See section 14.3)	Allows up to 9 digits in fractional seconds. Timezone component specified as "Z" will be decoded as "+00:00". Same for epochTZ.
epoch	number of seconds since Jan 1, 1970 (4 or 8 bytes signed binary)	Does not allow fractional seconds.
epochTZ	number of seconds since Jan 1, 1970 as above followed by 2 byte signed number of minutes from GMT (6 or 10 bytes signed binary)	Does not allow fractional seconds. Timezone component "Z" is decoded as "+00:00". Also see orats.
enum	Each potential value of the enumeration is assigned an	

	unsigned integer value, starting with 0, in the order they appear in the schema.	
qname	4/8 token ID. Followed by 2-byte prefix ID.	

The table below lists the default encoding datatype and the set of allowed encoding datatypes for all the XML schema datatypes. Some of the CSX encoding datatypes have restrictions in terms of the set of legal values that it can encode. In cases when the default mapping is not appropriate for the application, the user has to explicitly override the encoding type. Note that "string" is always allowed as the encoding type. Type mismatches or value overflows should be treated as encoding errors.

Mapping from XML Schema Datatypes to CSX Encoding Datatypes

XML Schema Datatype	Default Encoding Datatype	Allowed Encoding Datatypes
string	String	
boolean	Boolean	string
float	Float	string
double	Float	string
decimal	Oranum	string
duration	String	
dateTime	Orats	oradate, epoch, epochTZ, string
time	Orats	oradate, epoch, epochTZ, string
date	orats	oradate, epoch, epochTZ, string
gYearMonth	orats	oradate, epoch, epochTZ, string
gYear	orats	oradate, epoch, epochTZ, string
gMonthDay	orats	oradate, epoch, epochTZ, string
gDay	orats	oradate, epoch, epochTZ, string
gMonth	orats	oradate, epoch, epochTZ, string
hexBinary	binary	string
base64Binary	binary	string
anyURI	string	
QName	qname	string
NOTATION	string	
normalizedString	string	
token	string	
language	string	
IDREFS	string	
ENTITIES	string	
NMTOKEN	string	

NMTOKENS	string	
Name	string	
NCName	string	
ID	string	
IDREF	string	
ENTITY	string	
integer	oranum	string
nonPositiveInteger	oranum	string
negativeInteger	oranum	string
nonNegativeInteger	oranum	string
positiveInteger	oranum	string
long	int	oranum, string
int	int	oranum, string
short	int	oranum, string
byte	int	oranum, string
unsignedLong	unsigned-int	oranum, string
unsignedInt	unsigned-int	oranum, string
unsignedShort	unsigned-int	oranum, string
unsignedByte	unsigned-int	oranum, string

8 SECTION FORMAT

The CSX section format is a sequence of CSX instructions starting with the STRTSEC instruction, section header, followed by section data. The section data may be encoded piecewise in section *chunks* using the CHUNK instruction or all together in a single section stream. Token definitions as well as node data are defined via a stream of instructions. The stream of node data instructions roughly corresponds to a set of SAX events, serialized into binary format.

8.1 ARRAY MODE

The array mode is used whenever the CSX processor detects multiple sibling elements with the same qualified name. In array mode, the token ID (as explicitly specified in the section stream or implicitly derived from the kidNum in schema-sequential mode) for the previous element is reused for each subsequent data item encountered until an end-of-array opcode is encountered.

Example Instance

<Item>Item 1</Item>
<Item>Item 2</Item>
…
<Item>Item 100</Item>

Example CSX

CSX Opcode	Operands	Notes
PRPST4L1	4 byte token ID for “Item” + text value “Item 1”	
ARRBEG		Start of array mode
DATSTR	Item 2	Same token ID as previous element
…		
DATSTR	Item 100	
ARREND		End of array mode

8.2 SCHEMA-SEQUENTIAL MODE

The schema-sequential mode can be used to encode elements for which the corresponding schema declaration carries the sequential=”true” attribute on csx:kidList annotation. The schema declaration lists the (only) allowed order in which the immediate children of the element can occur. Each of the children is also assigned a unique kidNum by the schema.

The sequential attribute can be set to true whenever the CSX processor can determine that there is only one possible order in which the children of the element can occur. This case exists when the content model for a complexType element consists only of any number of <sequence> and <choice> groups with maxOccurs=1 (element level maxOccurs >1 is okay). The schema-sequential mode cannot be used if there is any <sequence> or <choice> group with maxOccurs > 1, or when there is an <all> group.

In schema-sequential mode, the CSX stream consists only of the data values and does not contain any token IDs. Hence, the CSX processor must keep track of the current kidNum to infer the node information for each piece of data. The current kidNum is increased after each child operand (at this level) is processed unless it is in array mode. Note that even in the schema-sequential mode, the child operands could carry explicit token ID information. For example, to handle substitionGroups where the declared element has been substituted by another element.

Example Schema

```
<xsd:element name=”Employee”><xsd:complexType>
  <xsd:annotation><xsd:appInfo>
    <csx:kidList sequential=”true”>
      <csx:kid propertyID=”3456” kidNum=”1”/>
      <csx:kid propertyID=”2133” kidNum=”2”/>
      <csx:kid propertyID=”2438” kidNum=”3”/>
    </csx:kidList>
  </xsd:appInfo></xsd:annotation>
  <xsd:sequence>
    <xsd:element name=”Name” type=”xsd:string” csx:propertyID=”3456”/>
    <xsd:element ref=”Salary”/>
    <xsd:element ref=”Comment” maxOccurs=”10”/>
  </xsd:sequence>
</xsd:complexType></xsd:element>
```

Assume that element <mgrComment> has been defined to be in the substitution group of <Comment>.

Example Instance

```
<Employee>
  <Name>John</Name>
  <Salary>10000</Salary>
  <Comment>first comment</Comment>
  <Comment>second comment</Comment>
  <MgrComment>manager comment</MgrComment>
</Employee>
```

Example CSX

CSX Opcode	Operands	Notes
PRPSTT4F	4 byte propertyID for "Employee" + Flag bit to indicate that children should be processed in schema-sequential mode	Start of the schema-sequential mode.
DATSTR	John	Current kidNum = 1
DATINT4	10000	Current kidNum = 2
DATSTR	first comment	Current kidNum = 3
ARRBEG		kidNum is not incremented in array mode
DATSTR	second comment	Current kidNum = 3
PRPT4L1	4 byte propertyID for "mgrComment" + text value "manager comment"	Current kidNum = 3. Opcode used to specify propertyID because the actual element is different from declared element.
ARREND		End of array mode
ENDPRP		

8.2.1 Partial Schema-sequential Mode

In case the schema does not completely restrict the order of the children, a CSX processor may use the partial schema-sequential mode. In this mode, some of the children are encoded using implicit kidNum whereas others may be encoded using an explicit kidNum or token ID. The schema declaration indicates the set of children for which the schema-sequential mode can be applied and the remaining children for which this mode cannot be applied. A opcode NOSEQ is used to indicate that the immediately following operand does not use the schema-sequential mode. In response, the CSX processor does not increment the current kidNum while processing the following operand.

The partial schema-sequential mode is useful in case the children of an element are mostly ordered but with a few exceptions. For example, a small <choice> group with maxOccurs > 1 nested within a large <sequence> group. Another usecase is for handling attributes because the order of attributes cannot be constrained by the schema. The partial schema-sequential mode is also useful in case of schema evolution (see section below).

Example Schema

```
<xsd:element name="Employee"><xsd:complexType>
  <xsd:annotation><xsd:appInfo>
    <csx:kidList sequential="true" maxSeqKidNum="3">
```

```
            <csx:kid propertyID=”3456” kidNum=”1”/>
            <csx:kid propertyID=”2133” kidNum=”2”/>
            <csx:kid propertyID=”2438” kidNum=”3”/>
            <csx:kid propertyID=”3457” kidNum=”4”/>
            <csx:kid propertyID=”3458” kidNum=”5”/>
          </csx:kidList>
        </xsd:appInfo></xsd:annotation>
        <xsd:sequence>
          <xsd:element name=”Name” type=”xsd:string” csx:propertyID=”3456”/>
          <xsd:element ref=”Salary”/>
          <xsd:element ref=”Comment” maxOccurs=”10”/>
        </xsd:sequence>
        <xsd:attribute name=”empID” type=”xsd:string” csx:propertyID=”3457”/>
        <xsd:attribute name=”age” type=”xsd:int” csx:propertyID=”3458”/>
    </xsd:complexType></xsd:element>
```

Example Instance

```
<Employee age=”40” empID=”4444”>
  <Name>John</Name>
  <Salary>10000</Salary>
  <Comment>first comment</Comment>
  </Employee>
```

Example CSX

CSX Opcode	Operands	Notes
PRPSTT4F	4 byte propertyID for “Employee” + Flag bit to indicate that children should be processed in schema-sequential mode	Start of the schema-sequential mode. Current kidNum is initialized to 0.
NOSEQ		Current kidnum is **not** incremented when processing next operand
PRPK1L1	Kidnum=5 + text value “40”	
NOSEQ		Current kidnum is **not** incremented when processing next operand
PRPK1L1	Kidnum=4 + text value “4444”	
DATSTR	John	Current kidNum = 1
DATINT4	10000	Current kidNum = 2
DATSTR	first comment	Current kidNum = 3
ENDPRP		

8.2.2 Impact of Schema Evolution

When a schema is evolved by adding new (optional) elements in the middle of the sequence, the newly added children are not encoded using schema-sequential mode. The kidNums of the earlier children are not changed, hence existing CSX data will continue to be valid per the new version of the schema. The new element is assigned a new kidnum, and the maxSeqKidNum value is set such

that the earlier children are encoded using schema-sequential mode. The new element has to be encoded with an explicit kidNum or token ID value i.e. partial schema-sequential mode.

If a schema is evolved to add a large number of new children, it may be better to turn off schema-sequential mode optimization entirely i.e. sequential=”false”. However, since the kidNums are listed in the annotated schema, the existing CSX instances can still be decoded correctly.

8.3 PREFIXES

CSX treats prefixes differently than other types of tokens to allow for more optimization. The following observations are used in encoding prefixes:

- the mapping between a prefix and a namespace URL is typically one-to-one in any given document, whereas across a collection of documents this is not true
- the number of prefixes used in a particular document is typically small, usually in single digits, so the benefits of sharing prefix tokens across documents is minimal

To preserve DOM fidelity, each node must include a prefix definition in addition to the namespace URL and localname. The namespace URL & localname are encoded using token IDs. If the prefix mappings are 1-1, and a prefix definition precedes the current instruction, the prefix may be looked up based on the namespace URL defined by the tokenID. If mappings are not 1-1, the instruction must contain an explicit prefix ID. A CSX processor MUST assume that prefix mappings are 1-1 until multiple prefix token definitions for the same namespace (in the same scope) are encountered. It is an error to encounter a CSX instruction without a prefix token ID if there are multiple prefix mappings for the same namespace URL (in the current scope). The prefix definitions are scoped to the element that appears immediately after it in the CSX stream.

8.4 FORMAT

The following items appear in CSX operands, as defined below:

- Prefix IDs: token for any prefix defined that is mapped to a namespace URL in the current document. Note that the same prefix may be mapped to different namespaces in different scopes in the document—in this case, the prefix will be defined multiple times.
- Namespace IDs: token ID corresponding to the namespace URL.
- Token IDs: token ID corresponding to the fully qualified element or attribute name. The (property) IDs defined by the schema are encoded as negative values whereas the token IDs scoped to the CSX processor are encoded as positive values.

The CSX format puts the following limits on XML documents and schemas:

- Namespace prefixes must be less than 256 bytes long
- Namespace URLs must be less than 65535 bytes
- Total number of elements/attributes declared in a document must be less than 2^63
- Total length of any single text node value must be less than 2^63 bytes
- Total length of any other node value must be less than 2^32 bytes
- Number of namespaces used in a document must be less than 32768
- Local names must be less than 65535 bytes long

The CSX encoder should also enforce the following limits.

- tokenID encoding should be used if kidnums exceed 2^16

8.4.1 Section Header

A section header optionally contains the following fields that collectively act as a unique identifier for the section:

- Processor GUID – identifier of the CSX processor to which the referenced tokens and schemas (if any) are scoped.
- Doc-ID – identifier of the document in which this section belongs.
- Path-ID – identifier for the simple path corresponding to the root node of the section. For example, the section for <LineItem> will contain the path-id for /PurchaseOrder/LineItem
- Order key – the ordering information corresponding to the root node of the section. For example, the section for the 3rd Address within the 2nd LineItem will contain order key of 2.3.

In case the section corresponds to a document, the CSX processor guid and doc-id fields are optional while the path-id and order key fields are omitted. The format of the section header is shown below.

Bytes	Description
1	CSX Version. Currently should have the value 1.
1	Flag Bit 0 is set if no inlined token definitions are present in this section Bit 1 is set if this section does not reference any schemas Bit 2 is set if a CSX processor GUID is present Bit 3 is set if document identifier is present Bit 4 is set if the path-id and order key is present Bit 5 is set if all the token ids in qname token definition opcodes are numbered sequentially starting from 1, and, all namespace token ids in namespace token definition opcodes are numbered sequentially starting from 1
1	Length of docID
n	Doc id – GUID or URL
1	Length of path id
n	Path – id
1	Length of order key
n	Order key
16 bytes	CSX processor GUID

Note that any opcode including property IDs are not valid until an schema start opcode at the current or outer level of nesting has appeared to specify the schema ID to which the property IDs are scoped.

Specifying the processor ID in the section header is often not necessary, as the CSX processor will know the CSX processor in scope from the enclosing context. For example, a CSX processor retrieving CSX data over a TCP connection from a CSX processor will generally assume that data comes scoped to that particular processor, making the processor GUID redundant. However, since finding schemas and tokens from a CSX processor is required to decode CSX, some applications (such as storage of CSX in a filesystem, message passing applications, etc.) use the processor GUID to indicate where to go to find the metadata required to decode the CSX data. The mechanisms for contacting the processor given that GUID are out of scope for this specification, but might include configuration files, network broadcast to find a processor based on GUID, etc.

A CSX section must have all tokens scoped to a single CSX processor. A CSX processor that needs to create a stream based on documents from multiple processors will have to translate tokens into a single processor space. This allows CSX processing to assume bidirectional token mapping, for example a particular namespace URL can have only one token to represent it.

The flag bits in the section header that assert that there are no inlined token definitions or that there are no references to schemas - are useful for optimizing certain kinds of processing. However, a CSX processor is not required to set these bits especially if it impacts performance due to unnecessary buffering. Typically, the database storage of CSX data involves stripping out the token definitions and other pre-processing steps. These flag bits can be set in such scenarios.

8.4.2 Basic Section Encoding

Basic section encoding is used when no buffering can be assumed by the CSX processor, or when the space overhead of chunk based encoding (see below) isn't justified by the processing benefits. In this mode, CSX processors will generally have to process the section instruction by instruction. Node data and token definition data are interspersed in the stream, and terminated by the ENDSEC instruction. A sample use case for basic section encoding might include the CSX data in a database section table (which might never have token data) stored in the stream, or a document encoded by a transient processor. The byte stream in basic section encoding will look as follows:

STRTSEC	Section Header	Node and Token Data	ENDSEC

8.4.3 Chunk Based Encoding

A section may contain the CHUNK instruction that allows CSX processors to *constrain* the occurrence of token definition instructions. The CHUNK instruction is immediately followed by a set of token definition instructions. The CSX processor guarantees that between the first non-token-definition instruction through the end of the chunk, there are no token definition instructions. This guarantee can speed up processing by allowing the receiver to process CSX data in that chunk without scanning it instruction by instruction. For example, if the receiver is the database, it can bulk append the node data in the section chunk into a target BLOB without interpreting every opcode in the CSX stream.

As shown below, the CHUNK instruction has the chunk length operand, followed by a set of zero or more token definitions. These are followed by the CSX data instructions, etc., until the end of the chunk (as indicated by the chunk length value). This may be followed by another CHUNK instruction.

The CHUNK instruction also has a flag operand that indicates if any of the tokens previously defined within this section are being referenced within this chunk. If this flag bit is not set *and* there are no token definitions at the start of the chunk, server side processing of the chunk can be optimized to avoid opcode iteration for token ID remapping purposes.

		←----------	----------N----------→		
……	CHUNK(N)	Token Definitions	Node data (no token definitions)	CSX Instruction (could be CHUNK)	……..

8.5 Element Encoding

The CSX opcodes are designed to optimize both space and processing speed for the common cases. Since some of the optimized opcodes may not handle the largest possible operands and/or other options, processors can use the "generalized" opcodes in such cases. For example, there are optimized opcodes for the case of element or attribute with a single text node child. If an element contains multiple text node children and/or has interspersed comments, etc., it can be encoded

using the generalized element start (PRPST*), followed by opcodes corresponding to the children and ending with the ENDPRP opcode.

Opcodes are encoded using the following scheme as a single byte. Instructions may only have a single operand of variable length, and that operand must be the last one. In this case, the length of the variable length operand itself must be the first operand.

The node data values are always accompanied by 1/2/8 byte length. The first byte of the length indicates the encoding type as shown below. The 1-byte length format is sufficient to encode all types (and their lengths) except the following:

- string values greater than 64 bytes
- binary values greater than 32 bytes

In these cases, the length is represented by 2/8 bytes. The actual length is calculated by masking out the two high-order bits of the first byte. The two high-order bits of the first byte indicate the encoding type as follows:

- 0x00 – encoding type is string
- 0x01 – encoding type is binary
- 0x10, 0x11 – reserved for future use

Note: The opcodes listed below are identified by their mnemonic names. The actual byte value corresponding to each of these opcodes is given in section 16.

8.5.1 DAT Opcodes

This set of opcodes specifies the encoding type as well as the length of the data operand of the *current* node/item, e.g., in schema/sequential mode or XQuery Data model item. There are no other operands, and there is no prefix id. These "opcodes" can also appear as the 1-byte data length operand in other operations.

Operation	**Description**
DATSTRx	Data encoded as string. Length 1-64 bytes.
DATBINx	Data encoded as binary. Length 1-32 bytes.
DATNMx	Data encoded as oranum. Length 1-21 bytes.
DATINT1	Data encoded as 1 byte int.
DATINT2	Data encoded as 2 byte int.
DATINT4	Data encoded as 4 byte int.
DATINT8	Data encoded as 8 byte int.
DATUINT1	Data encoded as 1 byte unsigned-int.
DATUINT2	Data encoded as 2 byte unsigned-int
DATUINT4	Data encoded as 4 byte unsigned-int
DATUINT8	Data encoded as 8 byte unsigned-int
DATFLT4	Data encoded as 4 byte float
DATFLT8	Data encoded as 8 byte float
DATEPH4	Data encoded as 4 byte epoch
DATEPH8	Data encoded as 8 byte epoch
DATEPZ6	Data encoded as 6 byte epochTZ
DATEPZ10	Data encoded as 10 byte epochTZ
DATODT	Data encoded as oradate
DATOTS	Data encoded as orats without timezone
DATOTSZ	Data encoded as orats with timezone
DATBOL	Data encoded as boolean
DATQNM4	Data encoded as 4-byte token ID followed by 2-byte prefix ID
DATQNM8	Data encoded as 8-byte token ID followed by 2-byte prefix ID
DATENM1	Data encoded as 1 byte enum

DATENM2	Data encoded as 2 byte enum

8.5.2 General DATA opcodes

Operation	Description
DATAL2	A two-byte length operand follows indicating the length of data, followed by that many bytes of data for the current element.
DATAL8	A eight byte length operand follows indicating the length of data, followed by that many bytes of data for the current element
DATATL1	1 byte data length operand, 4-byte type-id followed by data.
DATATL2	2 byte data length operand, 4-byte type-id followed by data.
DATATL8	8 byte data length operand, 4-byte type-id followed by data.
DATEMPT	Schema-sequential/array mode only: the data for the current element is empty (e.g. <tag/>)
DATNULL	Schema-sequential mode only: the current element doesn't exist in this instance.

8.5.3 Schema Related Opcodes

Operation	Description
SCHSST1	Start schema scope. Contains a 1 byte length operand followed by a one byte version number. The third operand is a variable-length schema ID
SCHSST4	Start schema scope. Contains a 1 byte length operand followed by a four byte version number. The third operand is a variable-length schema ID
SCHSST4V	Start schema scope with full validity assertion. Contains a 1 byte length operand followed by a four byte version number. The third operand is a variable-length schema ID. The data encoded using this schema is asserted to be fully valid (as opposed to the default partial validity).
SCHSEND	End of schema scope

8.5.4 DTD Related Opcodes

Operation	Description
DTDSTR	Start of doctype declaration. Operands are 2-byte length total of all operand data, followed by name, publicID and systemID strings prefixed with their individual 2-byte lengths.
DTDELEM	DTD Element definition. Operands are 2-byte length total of all operand data, followed by name and content spec strings prefixed with their individual 2-byte lengths.
DTDALIST	DTD Attribute List definition. Operands are 2-byte length total of all operand data, followed by element name and attribute definition text strings prefixed with their individual 2-byte lengths.
DTDENT	DTD (General) Entity definition. Operands are 2-byte length total of all operand data, followed by name, value, publicID, systemID and notation name strings prefixed with their individual 2-byte lengths.

DTDPENT	DTD Parameter Entity definition. Operands are 2-byte length total of all operand data, followed by name, publicID and systemID strings prefixed with their individual 2-byte lengths.
DTDNOT	DTD Notation definition. Operands are 2-byte length total of all operand data, followed by name, publicID and systemID strings prefixed with their individual 2-byte lengths.
DTDEND	End of doctype declaration.
ENTREF	Entity Reference. First operand is 1 byte name length followed by entity name.
CHARREF	Character Reference. First operand is a 1 byte length followed by variable number of bytes comprising the character reference.

8.5.5 Document/Section related Opcodes

Operation	Description
DOC	Document node. 1st operand is a 1 byte length of the optional charset ID—this may be 0 if not present. 2nd operand is a 2 byte flag, as follows: Bit 0 is set if standalone is declared in prolog. Bit 1 is set if a prolog is present Bit 2 is set if the encoding is declared in the prolog Bit 3 is set if the XML version is in the image header (else version=1.0) Bit 4 is set if standalone = TRUE Bit 5 is set if the document preserves ignoreable whitespace Bits 8-15 are the XML version (low four bits indicate minor version), e.g. version 1.1 = 0x11. 3rd operand is the character set the data was *originally* encoded in.
STRTSEC	Start of section. This opcode is followed by the section header and the section data.
ENDSEC	End of section
CHUNK	Chunk instruction followed by 1byte flag and 4 byte chunk length. Flag bits are: 0x01 – Chunk references a token previously defined within this section.
REF	Section reference. 1st operand is a one-byte reference length followed by reference data

8.5.6 Text/CDATA/PI/Comment Opcodes

Operation	Description
TEXT1	A text node with a 1 byte length operand, followed by the data
TEXT2	A text node with a 2 byte length operand, followed by the data
TEXT8	A text node with a 8 byte length operand, followed by the data
CDATA1	A CDATA node with a 1 byte length operand, followed by the data
CDATA2	A CDATA node with a 2 byte length operand, followed by the data
CDATA8	A CDATA node with a 8 byte length operand, followed by the data
PI1L1	Processing Instruction. Operand 1 is a 1 byte length for both target and data and operand 2 is a 1 byte length just for the target, operand 3 is the target bytes and data bytes concatenated
PI2L4	Same as PI1L1 but the total length is 4 bytes and the target length is 2
CMT1	Comment. Operand 1 is a 1 byte length, and operand 2 is the value
CMT2	Comment. Operand 1 is a 2 byte length, and operand 2 is the value
CMT8	Comment. Operand 1 is a 8 byte length, and operand 2 is the value

8.5.7 Token Definition Opcodes

Operation	Description
DEFNM4L1	Define a namespace URL token. The first operand is the 1-byte URL length, the second operand is a 4-byte namespace token ID, and the third operand is the namespace URL.
DEFNM4L2	Same as DEFNM4L1 except with 2-byte URL length.
DEFNM8L1	Same as DEFNM4L1 except with 8 byte namespace token ID.
DEFNM8L2	Same as DEFNM4L1 except with 2-byte URL length and 8-byte namespace token ID.
DEFPFX1	Define a prefix, with the first operand being a one byte prefix length, the second operand being a 4 byte namespace token ID, the third operand a two byte prefix ID, and the 4th operand the prefix data
DEFPFX2	Same as DEFPFX1, but with a 8 byte namespace token ID
DEFQ4N4L1	Define a QName token. The first operand is a 1 byte name length, the second operand is a 1 byte type, followed by 4 byte token ID, 4 byte namespace token ID, and the local name. The values for type are: 0 (element QName) and 1 (attribute QName)
DEFQ4N4L2	Same as DEFQ4N4L1 but with 4-byte token ID, 4-byte namespace token ID and 2-byte name length
DEFQ4N8L1	Same as DEFQ4N4L1 but with a 4 byte token ID, 8-byte namespace token ID and 1 byte name length
DEFQ4N8L2	Same as DEFQ4N4L1 but with a 4 byte token ID, 8 byte namespace token ID and 2 byte name length
DEFQ8N4L1	Same as DEFQ4N4L1 but with a 8 byte token ID, 4 byte namespace token ID and 1 byte name length
DEFQ8N4L2	Same as DEFQ4N4L1 but with a 8 byte token ID, 4 byte namespace token ID and 2 byte name length
DEFQ8N8L1	Same as DEFQ4N4L1 but with a 8 byte token ID, 8 byte namespace token ID and 1 byte name length
DEFQ8N8L2	Same as DEFQ4N4L1 but with a 8 byte token ID, 8 byte namespace token ID and 2 byte name length

8.5.8 Property (Element/Attribute) Opcodes

Operation	Description
PRPK1L1	Element/Attribute with a single text node value. The following operands are present: 1-byte data length, 1-byte kidnum, data
PRPK1L2	Same as PRPK1L1 but with 2-byte data length
PRPK2L1	Same as PRPK1L1 but with 2-byte kidnum and 1-byte data length
PRPK2L2	Same as PRPK1L1 but with 2-byte kidnum and 2-byte data length
PRPT1L1	Element/Attribute with a single text node value. The following operands are present: 1-byte data length, 1-byte token ID, data
PRPT1L2	Same as PRPT1L1 but with 2-byte data length
PRPT2L1	Same as PRPT1L1 but with 2-byte token ID
PRPT2L2	Same as PRPT2L1 but with 2-byte data length
PRPT4L1	

	Same as PRPT2L1 but with 4-byte token ID
PRPT4L2	Same as PRPT2L1 but with 4-byte token ID and 2-byte data length
PRPT8L1	Same as PRPT2L1 but with 8-byte token ID and 1-byte data length
PRPT8L2	Same as PRPT2L1 but with 8-byte token ID and 2-byte data length
PRPSTK1	Start element/attribute, with 1 byte kidnum as the only operand
PRPSTK2	Same as PRPSTK1 with 2 byte kidnum as the only operand
PRPSTT1	Same as PRPSTK1 with 1 byte tokenID as the only operand
PRPSTT2	Same as PRPSTK1 with 2 byte tokenID as the only operand
PRPSTT4	Same as PRPSTK1 with 4 byte token ID as the only operand
PRPSTT8	Same as PRPSTK1 with 8 byte token ID as the only operand
PRPSTK1F	Start element/attribute, with 1 byte kidnum as the 1st operand, and a 1 byte flag as the second operand. This opcode may not be used with flags requiring additional metadata.
PRPSTK2F	Same as PRPSTK1F with 2 byte kidnum as the first operand
PRPSTT1F	Same as PRPSTK1F with 1 byte token ID as the first operand
PRPSTT2F	Same as PRPSTK1F with 2 byte token ID as the first operand
PRPSTT4F	Same as PRPSTK1F with 4 byte token ID as the first operand
PRPSTT8F	Same as PRPSTK1F with 8 byte token ID as the first operand
PRPSTK1V	Start element/attribute, with a length byte as the 1st operand followed by 1 byte kidnum, a 1 byte flag, and a variable length metadata field (as indicated by the element flags). The metadata may not be longer than 64 bytes
PRPSTK2V	Same as PRPSTK1V with 2 byte kidnum as the first operand
PRPSTT1V	Same as PRPSTK1V with 1 byte token ID as the first operand
PRPSTT2V	Same as PRPSTK1V with 2 byte token ID as the first operand
PRPSTT4V	Same as PRPSTK1V with 4 byte token ID as the first operand
PRPSTT8V	Same as PRPSTK1V with 8 byte token ID as the first operand
PRPSTT4B1F	Same as PRPSTT4F but has the element/attribute token ID followed by a 1 byte ID for the built-in type and a 1 byte flag
PRPSTT4Y4B1F	Same as PRPSTT4F but has the element/attribute token ID followed by a 4 byte token ID for the top level type QName, 1 byte ID for the built-in type and a 1 byte flag
PRPSTT4Y4F	Same as PRPSTT4F but has the element/attribute token ID followed by a 4 byte token ID for the top level type QName and a 1 byte flag.
PRPSTT4Y8F	Same as PRPSTT4F but has the element/attribute token ID followed by a 8 byte token ID for the top level type QName and a 1 byte flag.
PRPSTT4Y8B1F	Same as PRPSTT4F but has the element/attribute token ID followed by a 8 byte token ID for the top level type QName, 1 byte ID for the built-in type and a 1 byte flag.
PRPSTT8B1F	Same as PRPSTT8F but has the element/attribute token ID followed by a 1 byte ID for the built-in type and a 1 byte flag
PRPSTT8Y4B1F	Same as PRPSTT8F but has the element/attribute token ID followed by a 4 byte token ID for the top level type QName, 1 byte ID for the built-in type and a 1 byte flag
PRPSTT8Y4F	Same as PRPSTT8F but has the element/attribute token ID followed by a 4 byte token ID for the top level type QName and a 1 byte flag.
PRPSTT8Y8F	Same as PRPSTT8F but has the element/attribute token ID followed by a 8 byte token ID for the top level type QName and a 1 byte flag.

PRPSTT8Y8B1F	Same as PRPSTT8F but has the element/attribute token ID followed by a 8 byte token ID for the top level type QName, 1 byte ID for the built-in type and a 1 byte flag.
ELMSTART	Schema/sequential mode: start *current* element
ELMSTSSEQ	Schema/sequential mode: start current element whose children are *also* in schema-sequential mode.

Element/Attribute Start Flags for schema based encoding
Some of the element/attribute start opcodes contain flags, which are described below. Some of the flags indicate the usage of the space in the variable-width metadata segment. The fields referenced by the flags must appear in the order of the flags in the list below (if the appropriate flag indicates they are present)

0x01	This element's children should be processed in schema-sequential mode
0x02	This element is not of the declared type i.e. an xsi:type attribute appears on explicitly on this element. The first 4 bytes of metadata are the type ID.
0x04	Implicit type information present. The first 4 bytes of metadata are the type ID.
0x08	Prefix ID present—two bytes of prefix ID are present after any type ID.
0x10	This element is nillable.

Element/Attribute Start Flags for non-schema based encoding
Some of the element/attribute start opcodes contain flags relating to type information even though non-schema based encoding might be in force. The following flags are used in conjunction with the PRPSTT{4/8}Y{4/8}B1F opcodes.

0x1	This element is nillable.
0x2	This element does not have a named type (anonymous type).
0x4	Prefix ID present – two bytes of prefix ID are present after this flag.

8.5.9 Miscellaneous Opcodes

Operation	**Description**
ARRBEG	Begin array mode for the last element processed. If no element has been processed yet, this opcode is ignored. If array mode is already in force for this level of element nesting, the opcode is ignored.
ARREND	End of array
ENDPRP	End of element or attribute
NOSEQ	Not sequential. The immediately following property is not encoded using schema-sequential mode optimization.
NOP	No operation. Useful for filler when data values shrink
NOPARR	No operation, with a 4 byte operand indicating the total number of bytes to skip
NMSPC	Namespace node. The special xmlns:prefix=URL attribute declaration. The only operand is a 2 byte prefix ID
NSP4	Namespace node. The first operand is prefix length and the second operand is 4 byte namespace ID followed by actual prefix value. Normally NMSPC is preferred over NSP4 but to support piece-wise DML operations, when prefix ID is not available, NSP4 can be used.
NSP8	Same as NSP4 but the namespace ID is 8 bytes.
ARRSTK1V	Array Mode Start, with a length byte as the 1st operand followed by 1 byte kidnum, a 1 byte flag, and a variable length metadata field (as

	indicated by the element flags). The metadata may not be longer than 64 bytes. This indicates beginning of array mode with the metadata associated with this opcode as the context.
ARRSTK2V	Same as ARRSTK1V with 2 byte kidnum as the first operand.
ARRSTT4V	Same as ARRSTK1V with 4 byte token id as the first operand.
ARRSTT8V	Same as ARRSTK1V with 8 byte token id as the first operand.
PRTDATA	Partial data. 4 byte length operand followed by data.
PRTDATAT	Partial data. 4 byte length operand, 4 byte type-id followed by data
PRTTEXT	Partial Text, 4 byte length operand followed by text.
PRTCDATA	Partial Cdata, 4 byte length operand followed by Cdata Value.
PRTPI	Partial Processing Instruction. 4 byte length operand followed by partial PI Value.
PRTCMT	Partial Comment. 4 byte length operand followed by partial comment.
SPACE1	Ignorable whitespace Node with a 1 byte length.
SPACE2	Ignorable whitespace Node with 2 byte length.
SPACE8	Ignorable whitespace Node with 8 byte length.

Undefined opcodes are reserved for future expansion.

Encoding Notes

The opcode set is not perfectly symmetric—for example, some of the specialized opcodes providing for larger sizes of some fields require larger sizes for other fields. The most general way to encode elements is to use the variable width metadata opcodes—various size limitations may force encoding to use the generalized opcodes.

The optimizations defined in the schema (for sequential & array modes) are at the discretion of the encoding engine to use—for example, just because the schema specifies that sequential mode is allowable, the instance document may not use that optimization.

The data within an attribute or element can itself be a **list** of atomic values. This is encoded as a sequence of DAT* opcodes within PRPST* and ENDOP opcodes.

Large node values can be encoded using a sequence of PRTDATA opcodes ending with one of the DAT* opcodes. This scheme avoids the need to specify the total length of the node data at the start of the opcode, thus improving streamability. Only string and binary encoded data can be split up using PRTDATA* opcodes. As previously discussed, the first two bits of first length byte indicate the encoding type.

Schema Scoping

The SCHSST opcode is used to define the scope of property IDs and type IDs appearing after it in the CSX stream. The scope is managed in a stack fashion and is in effect until the corresponding SCHSEND opcode that reverts to the previously defined schema scope.

8.5.10 Whitespace opcodes: SPACE1, SPACE2 and SPACE8

SPACEx opcodes are used to encode ignoreable whitespace such as whitespace present between end tag of an element and begin tag of next. SPACE1, SPACE2 and SPACE8 opcodes take one operand that is 1, 2 or 8 bytes in length. The 3 MS bits in the length byte(s) are used to denote the type of whitespace - (000 - space, 001 - tab, 010 - linefeed, 011 - carriage return) and the rest of the length operand indicates the occurence count of the whitespace.

The encoder may optionally preserve the ignoreable whitespace. If it does, it may set the doc flag bit 5 described under DOC opcode. An example usage could be that, this bit may be consulted at

the time of printing the document to decide whether to do pretty printing or print only the space encoded in the document using SPACEx opcodes.

8.6 Encoding Instances of XQuery Data Model (non-normative)

CSX can be used to encode instances of the XQuery Data Model (XDM). Each XDM instance is a sequence of zero or more items. Each item is either a node or atomic value. A node can be one of the following types:- document, element, attribute, namespace, text, processing-instruction or comment. An atomic value is a value of atomic type – which is a primitive simple type or a type derived from another atomic simple type. A ***possible*** encoding of XDM using CSX opcodes is shown below. In this case, a CSX stream can contain any of these opcodes at the top-level.

XDM Item	CSX Opcode
Document node	DOC
Element node	PRP*, ELM*
Attribute node	PRP*
Namespace node	NMSPC
Text node	TEXT*
Processing instruction	PI*
Comment node	CMT*
Atomic value	DATA*
Sequence/EndSequence	SEQ/ENDSEQ

8.7 Section Reference Format

A section reference is an identifier for an external section of CSX. This is used in conjunction with the information contained in the (current) section header, to obtain the data for the external section. A **node** section reference consists of the following fields :

- Child path-id – the identifier corresponding to the full path leading up to the child node. Example, a section reference to <po:LineItem> contains the id for the path ”/po:PurchaseOrder/po:LineItem”.
- Child Order key – the order key of the child node *relative to its parent section*. For example, the 2nd LineItem with <PurchaseOrder> is identified by order key “2”. Also, the 3rd <Address> within the 2nd <LineItem> is identified by “3” which is the order key relative to the parent <LineItem> section. However, if <LineItem> were not sectioned, the order key within the <Address> section reference would be 2.3.

A **collection** section reference is a reference to a set of sibling nodes in document order. The sibling nodes may correspond to different QNames. The collection reference consists of the following fields:

- A set of child path-ids
- The first and last child order key

The format of the section reference is shown below:

Bytes	Description
1	Flag byte Bit 0 is set if this is a collection reference
1	Number of child path-ids – only for collection reference
1	Length of child path-id
n	child path-id

[repeat for collection reference]

1	Length of child order key (first order key for collection reference)
n	child order key (first order key for collection reference)
1	Length of last child order key – only for collection reference
n	last child order key

The section reference fields along with the (doc-id, path-id, order key) from the parent header can be used to locate and fetch the section data.

9 COMMUNICATION API

Communication between different CSX processors requires a basic set of operations to be supported by all CSX processors. These standard operations provide low-level functionality that are available to other CSX processors and, possibly, to user applications. The operations can be expressed as a locally available API or as parts of an HTTP based protocol or some other protocol. This section defines the (abstract) communication APIs.

CSX Processors can be of two kinds – Clients and Servers (Providers). The servers consume XML documents in the CSX format in Push mode and produce them in Pull mode. The clients produce XML documents in the CSX format in Push mode and consume them in Pull mode by invoking server's Push API (consumer) and Pull API (producer) correspondingly.

The APIs can be categorized as follows:

1. Pull-oriented APIs
    a. Pull API for schemas, DTDs and token sets (namespaces)
    b. Pull API for documents
2. Push-oriented APIs
    a. Accept API for schemas, DTDs and token sets
    b. Accept API for documents
3. Others
    a. Register API for schemas, DTDs and token sets

9.1 Data types

CSXStream – represents CSX format section. Binary stream of bytes.

DocumentIDType – opaque, variable length document identifier

SectionIDType – section identifier

VocabularyIDType – opaque, variable length vocabulary identifier

VocabularyKind – {schema, DTD, token set(namespace)}

9.2 Pull-Oriented APIs

9.2.1 Pull API for Schemas, DTDs and token sets

9.2.1.1 getVocab

This function is used to retrieve a fully annotated schema, DTD or all token definitions in a given namespace.

vocabID : represents a schema ID or a DTD ID or a namespace ID.
vocabKind : one of schema, DTD or namespace

public CSXStream getVocab(VocabularyIDType vocabID, VocabularyKind vocabKind);

9.2.1.2 lookupVocab

This function is used to lookup the schema corresponding to the specified inputs. The returned CSX stream is a schema mapping document or the actual annotated schema corresponding to the inputs.

public CSXStream lookupVocab(String schemaLocation, String namespace, String elementName, String user)

9.2.2 Pull API for Documents

9.2.2.1 getSection

This function retrieves the CSX data corresponding to the section identified by the sectionID and documentID parameters.

The section ID parameter is optional. Document ID is also optional. However, the document ID as well as the section ID might be required in some usage scenarios.

There are several parameters that influence the format of the returned section data. These are :

- inline_depth – Specifies the depth up to which the section references are to be expanded. The permitted values are :
    - -1 : all section references, regardless of their depth should be expanded.
    - 0 : none of the section references should be expanded.
    - n > 0 : section references up to depth n should be expanded.

  The default value of this parameter is 0.
- keep_references – Specifies that section references should be retained in the CSX data *even* if they are expanded. If this parameter is set to FALSE, then the expanded section references are omitted from the returned CSX.

  The default value of this parameter is FALSE.
- expand_entities – Specifies that all entity references should be expanded.

  The default value of this parameter is FALSE.
- non_schema_csx – Specifies that the returned CSX data should be encoded in a non-schema format i.e without using schema property IDs, etc.

  The default value of this parameter is FALSE.

This function returns the CSX stream corresponding to the specified section.

public CSXStream getSection(SectionIDType sectionID, DocumentIDType docID, int inline_depth, boolean keep_references, boolean expand_entities, boolean non_schema_csx);

9.3 Push-Oriented APIs

9.3.1 Accept API for Schemas, DTDs, and token sets

CSX processor ID, vocabulary ID (schema or DTD ID or token set ID), and vocabulary kind are transferred as parts of CSX Stream. All of them are mandatory. The timeout parameter (number of seconds) is a hint to the CSX processor to indicate how long the vocabulary is to be cached/persisted.

Parameters

Vocab – fully annotated schema, or DTD, or token definition set

public void acceptVocab(CSXStream vocab, int timeout);

9.3.2 Accept API for documents

CSX processor ID, vocabulary ID (schema or DTD ID or token definitions), and document ID are transferred as parts of CSX Stream. CSX processor ID is optional. However, it might be required for some protocols. It is also required when a corresponding schema, DTD, or token set where transferred in Push mode with accept interface.

Document ID is optional. When it is omitted, the document ID is assigned by the provider and returned. Some providers may ignore the document ID transferred with CSX Stream, assign their own document ID, and return it.

Vocabulary ID (schema ID, DTD ID, or token set ID) is optional. Some documents may have no vocabulary associated with them. Some documents may have two or more schemas associated with them. It is assumed that the vocabulary kind is determined by the vocabulary ID.

Parameters:

section – CSX Stream section

Returns – Document ID (the same as in the stream if it is accepted)

public DocumentIDType acceptDocument(CSXStream section);

9.4 Other APIs

9.4.1 Register API for Schemas, DTDs, and token sets

CSX processor ID and vocabulary ID (schema or DTD ID or token set ID) are not required and might be ignored by the registering CSX processor. For a schema, if its ID is provided, and the registering CSX processor have already registered a schema with this ID, then the input schema itself is (must) considered to be a new version of the previously registered schema with this ID. In this case, all schema evolution constraints apply.

The vocabulary kind is required and transferred as part of CSX Stream. For a token set its namespace may optionally be provided as a part of the CSX stream. Annotations in the returned stream must include the vocabulary ID. For a token set its ID is represented as namespace. If it was not provided in the stream for the registration it is assigned during the registration.

Parameters

Vocab – schema (possibly with user annotations), or DTD, or token set with no token definitions

Returns – fully annotated schema, or DTD, or token definition set

public CSXStream registerVocab(CSXStream vocab);

10 HTTP BASED COMMUNICATION

This section describes the HTTP methods to communicate with a CSX processor. The CSX processor must have an HTTP URL to provide the CSX services. The (binary) CSX data transferred to and from the processor is identified by a new mime type: application/vnd.oracle-csx.

The pull-oriented APIs are implemented by HTTP GET operation. The transfer control parameters such as section inlining, etc., are specified using new HTTP headers. The section ID, document ID, schema ID, etc., are specified as query parameters. The source URL is constructed by appending the processor URL with the method name.

The push-oriented APIs and the schema registration API are implemented by HTTP POST operation. The input CSX stream is the POST message body. Any other arguments (not CSX) are expressed as parameters in the URL. The response can also contain a CSX stream e.g. registerVocab().

10.1 X-CSX-Processor Header

Any HTTP entity body that is transferred with the CSX MIME type MUST include a X-CSX-Processor header with the processor URL. For example:

```
>> Request <<

  GET /foo.xml HTTP/1.1
  Host: www.example.com

>> Response <<

  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  [ CSX byte stream ]
```

10.2 X-CSX-Accept Header

The following parameters that control the CSX transfer mode are specified as HTTP headers. These headers can be used in any HTTP request for CSX data. In case of other operations and/or non-CSX data, these headers are ignored.

CSX-Accept-Header := “X-CSX-Accept: “ CSX-Accept-option (“,”CSX-Accept-option)*
CSX-Accept-option := (inline_depth | keep_references | expand_entities | non_schema_csx)
inline_depth := “inline-depth=”number-value
keep_references := “keep-references=”[“true”|”false”]
expand-entities := “expand-entities=”[“true”| “false”]
non_schema_csx := “non-schema-csx=”[“true”|”false”]

```
>> Request <<

  GET /foo.xml HTTP/1.1
  Host: www.example.com
  X-CSX-Accept: inline-depth=-1, expand-entities=true

>> Response <<

  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  [ CSX byte stream ]
```

10.3 PULL API for Schemas, DTDs and token sets

public CSXStream getVocab(VocabularyIDType vocabID, VocabularyKind vocabkind);

```
>> Request <<

  GET
http://www.example.com/sys/repos/getVocab?vocabID=5648&vocabKind=1
HTTP/1.1
  Host: www.example.com

>> Response <<

  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  Content-Length: xxx

  [CSX byte stream corresponding to the annotated schema/DTD/token set]
```

public CSXStream lookupVocab(String schemaLocation, String namespace, String elementName, String user)

```
>> Request <<

  GET
http://www.example.com/sys/repos/lookupVocab?schemaLocation=foo.xsd&nam
espace=http://my.com/foo&elementName=root&user=SCOTT HTTP/1.1
  Host: www.example.com

>> Response <<

  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  Content-Length: xxx

  [CSX byte stream corresponding to the schema mapping document or the
annotated schema]
```

10.4 Pull API for documents

public CSXStream getSection(SectionIDType sectionID, DocumentIDType docID, int inline_depth, Boolean keep_references, Boolean expand_entities, Boolean non_schema_csx);

The sectionID and docID are encoded as base-64 strings and specified as query parameters to the URL.

```
>> Request <<

  GET
http://www.example.com/sys/repos/getSection?sectionID=5648&docID=12
HTTP/1.1
  Host: www.example.com
  X-CSX-Accept: inline-sections="true", inline-depth="2"

>> Response <<
```

```
  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  Content-Length: xxx

  [CSX byte stream corresponding to the section]
```

10.5 Register API for Schemas, DTDs, and token sets

public CSXStream registerVocab(CSXStream vocab);

```
>> Request <<

  POST http://www.example.com/sys/repos/registerVocab HTTP/1.1
  Host: www.example.com
  Content-Type: application/vnd.oracle-csx
  Content-Length: xxx

  [CSX byte stream corresponding to schema/DTD/token set to be
registered]

>> Response <<

  HTTP/1.1 200 OK
  Content-Type: application/vnd.oracle-csx
  X-CSX-Processor: http://www.example.com/sys/repos
  Content-Length: xxx

  [CSX byte stream corresponding to the annotated schema/DTD/tokenset]
```

10.6 Accept API for Schemas, DTDs, and token sets

public void acceptVocab(CSXStream vocab);

```
>> Request <<

  POST http://www.example.com/sys/repos/acceptVocab HTTP/1.1
  Host: www.example.com
  Content-Type: application/vnd.oracle-csx
  Content-Length: xxx

  [CSX byte stream corresponding to schema/DTD/token set to be
accepted]

>> Response <<

  HTTP/1.1 204 No Content
```

10.7 Accept API for documents

public DocumentIDType acceptDocument(CSXStream section);

The HTTP PUT method is used to upload a document to a CSX processor.

>> **Request <<**

```
PUT http://www.example.com/myfolder/mydoc.csx HTTP/1.1
Host: www.example.com
Content-Type: application/vnd.oracle-csx
Content-Length: xxx

[CSX byte stream corresponding to the document to be accepted]
```

>> **Response <<**

```
HTTP/1.1 200 OK
```

11 References

[XML1] XML 1.0 Specification.
[XMLS1] XML Schema Part1 – Structures.
[XMLS2] XML Schema Part 2 – Datatypes.
[XDM] XQuery 1.0 and XPath 2.0 Data Model.
[DOM] Document Object Model 2.0
[UTF-8] UTF-8 encoding. http://www.ietf.org/rfc/rfc2279.txt

12 Appendix A – CSX State (Non-Normative)

One of the major issues in CSX design is the amount of state that must be built up while processing a CSX stream. To handle XPath processing for use in queries, indexes, and XInclude, it is desirable for indexes and other external reference mechanisms to be able to use byte offsets to point into the stream without requiring processing of the entire stream from the beginning. Therefore, the CSX model must not require that state information be maintained for a large chunks of the section, and also requires that any state necessary to start processing at an arbitrary instruction be serializable into a small, fixed set of bytes. That state, along with the byte offset and stream reference, forms a *CSX Locator*, which can be used by indexes and other external references to start CSX processing. There is a tradeoff between this requirement and overall stream compaction. For example, XML text has a very efficient mechanism for representing namespace information, by scoping it recursively to nested nodes. However, this mechanism means that the entire stream must be processed to figure out what namespace definitions apply to any particular element.

CSX uses the start of a complex element definition as the point at which stateless processing may commence—allowing more jumping points than XML text, which must be processed from the start of the document, but not requiring that state information be pushed to every node, which would be too wasteful of space. Then, the state needed to start processing at any arbitrary node is only the state built up from the start of the parent element. An index may be defined to only start at complex element starts, in which case a full locator need not be stored, or may be more flexible, and require full locator storage.

An implementation of a locator might contain the current schema ID, a byte offset into the underlying CSX stream, and bits indicating whether sequential mode was in effect, and if so, an index and parent property ID, or if array mode is in effect, and if so, the last token ID processed, etc.

13 Appendix B – Encoding Dewey Order Keys

An order key is an ordered list of numbers. In the general case, the components of the order key are floating point numbers (due to incremental inserts of fragments in the middle of documents). Each order key component is individually encoded as shown below. The byte sequences corresponding to the individual order key components are finally concatenated together to obtain the byte sequence for the order key.

The scheme described below allows unlimited precision and scale up to 2^20. Further, it is designed to yield 1-byte encodings for integral values 0 through 111.

13.1 Step 1: Obtain bit sequence corresponding to the floating point number N

13.1.1 Case: N < 112

```
N = (integral part).(fractional part)
          is encoded as
[integral part(7 bits), fractional part(variable length)]
```

13.1.2 Case: N >= 112

```
N = 1.(mantissa) * (2 ^ (exponent + 6))
          is encoded as
[111, exponent (4 bits), mantissa (variable length)]
```

13.2 Step 2: Encode bit sequence into a sequence of bytes

The bit sequence is split into chunks of 7 bits. Each chunk is followed by a continuation bit. The continuation bit is set to 0 to indicate the last byte in the sequence (i.e. all trailing bits are zero). The continuation bit is set to 1 to indicate that there is a following byte in the sequence. This encoding scheme results in byte-comparable and byte-sortable sequences.

```
[data (7 bits), continuation bit (0: end, 1: continue) ]...
```

Additionally, for Case N>=112, bit sequences are always encoded into 2 or more bytes i.e. in case the mantissa is 0, a continuation byte 0 is appended. For example, see encoding for 128 and 256 below.

13.3 Examples (Integers)

Number	Byte sequence	Comments
0	[0000000 0]	
1	[0000001 0]	
2	[0000010 0]	
…	…	
111	[1101111 0]	
112	[111 0000 1] [1100000 0]	1.(11)*2^(0+6)

113	[111 0000 1] [1100010 0]	1.(110001)*2^(0+6)
114	[111 0000 1] [1100100 0]	1.(110010)*2^(0+6)
…	…	
128	[111 0001 1] [0000000 0]	1.(0)*2^(1+6)
129	[111 0001 1] [0000001 0]	1.(0000001)*2(1+6)
…	…	
256	[111 0010 1] [0000000 0]	1.(0)*2^(2+6)
…		

13.4 Examples (Floating Points)

Number	Byte sequence	Comments
0.5	[0000000 1] [1000000 0]	(0).(1)
3.25	[0000011 1] [0100000 0]	(11).(01)
112.5	[111 0000 1] [1100001 0]	1.(1100001)*2^(0+6)

13.5 Notes

In the first byte of an order key component, the following values are reserved for internal use.

- 111xxxx0 – Note that since case>=112 is always encoded in 2 or more bytes, these values are not valid for encoding order keys.
- 11111111 – Note that the largest allowed exponent is (14+6) = 20. Hence, 0xFF is not a valid first byte.

14 SCHEMA DATATYPES FORMAT

14.1 CSX Number Format

CSX Numbers are represented as an array of one-byte digits preceded by an exponent. There can be a maximum of 20 digits.

<exponent, digit1, digit2, …, digit20>

14.1.1 Digits

The CSX number format uses base 100 digits, that is, each digit represents a number between 0 and 99. In addition, 1 is added to every digit, so the values are between 1 and 100. The most significant digit is represented as digit1. Digits for negative numbers are stored differently than

those for positive numbers. Instead of adding 1, the digit is subtracted from 101. An ending tag of 102 is appended to all negative numbers.

14.1.2 Exponent

The exponent byte consists of three parts: the sign bit, the offset, and the exponent. The sign bit is the first bit of the number, and is set if the number is positive or zero. The sign bit has a numeric value of 128. The offset is always 65. The exponent is a number in -65..62, and is the exponent for the number in scientific notation base 100. That is, if the exponent byte is 200 = 128+65+7, the exponent is 7, so the number should be raised to 100^7 = 10^14. If a number is negative, its exponent byte is calculated exactly the same way, but it is inverted. For example, the number -1000 (minus 1000) is -10 * 100^1, so its exponent byte would be ~(128+65+1) = ~(194) = 255-194 = 61. Note that for all negative numbers, the sign bit will not be set.

14.2 CSX Date Format

The CSX date format uses a maximum of 7 bytes to represent date and time. The byte format is as follows:

<First 2 digits of year, Last 2 digits of year, month, day, hour, minute, seconds>

14.3 CSX Timestamp Format

The CSX timestamp format can have a maximum of 13 bytes to represent a timestamp including optional timezone information. The byte format is as follows:

2 bytes for year, 1 each for month, day, hour, minute, second. 4 for fractional seconds, 1 each for timezone hour and minute.

15 Appendix - Possible future enhancements to CSX

This section lists some of the possible future enhancements to CSX.

1. Flexible sectioning rules. Add ability to specify sectioning rule at the level of <xsd:element> instead of only at the <xsd:schema>. Also, add the capability to indicate sectioning rules at a type level rather than at the element level.
2. Tokenize entity definitions – and entity-ids within entity references.