INTERNATIONAL STANDARD

ISO/IEC 15938-1

First edition
2002-07-01

AMENDMENT 2
2006-04-01

Information technology — Multimedia content description interface —

Part 1: **Systems**

AMENDMENT 2: Fast access extension

Technologies de l'information — Interface de description du contenu multimédia —

Partie 1: Systèmes

AMENDEMENT 2: Extension d'accès rapide

Reference number
ISO/IEC 15938-1:2002/Amd.2:2006(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 2 to ISO/IEC 15938-1:2002 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

Information technology — Multimedia content description interface —

Part 1: Systems

AMENDMENT 2: Fast access extension

This document preserves the sectioning of ISO/IEC 15938-1. The text and figures given below are additions and/or modifications to those corresponding sections in ISO/IEC 15938-1. All figures and tables shall be renumbered due to the addition of several figures and tables.

Add the following definitions to subclause 3.2 (keep alphabetical order), then renumber all definitions in subclause 3.2:

path index key
value representing the path to the element to be indexed/located, and the relative path to the fields to be keyed/searched.

value index key
set of encoded field values to be keyed/searched.

index stream
set of Index Access Units which together form the whole of the indexing data,

index decoder init
initialisation data for an index stream.

index access unit
index access unit header and associated structures forming a logical unit of access.

index access unit header
list of structures contained within this Index Access Unit

path index
structure allowing path index key to value index reference lookup.

value index
structure allowing value index key to value sub index reference lookup.

value sub-index
structure allowing value index key to BiM stream reference lookup.

node reference
reference from one node to another within a list or B-Tree structure.

data repository reference
reference to data entry within the binary or string data repository structures.

BiM stream reference
reference to a BiM encoded fragment within a BiM stream.

local access unit
default Access Unit associated with a Path Index.

local BiM stream reference
reference to a BiM encoded fragment contained within the local access unit.

remote BiM stream reference
reference to a BiM encoded fragment contained within the BiM stream, where the access unit is specified by an access unit ID.

value index reference
reference to a value index structure.

value sub-index reference
reference to a sub value index structure.

position codes reference
reference to a position codes entry within a position codes structure.

position code
location of an XML element within its parent element.

position codes
set of position code values for all elements within a context path.

BTree
binary decision tree, where each node can have multiple keys.

BTree order
specifies the maximum number of child nodes of a node in a BTree.

indexed element
XML element to which an index refers.

BiM stream reference format
specifies the format of the BiM stream reference.

value encoding
specifies how data has been encoded in value index key.

Add the following subclause 5.9.1:

5.9.1 General Description

Using the ISO/IEC 15938-1 index encoding, only fragments of the description that are of immediate interest to the terminal can be selectively acquired and combined with the current description tree. The terminal can search the index information to determine which fragments contain a node at a given location which has a related node with a given value within the description. Additionally the terminal may search for fragments containing nodes which have related node values falling within a given range.

The index information can also be compiled to allow the terminal to search for fragments containing nodes with multiple given related node locations, and respective values, within the description. This can allow the terminal to perform searches with multiple conditions, without needing to consolidate multiple result sets. As the indexing stream is optional a stream may consist of either

A DecoderInit and a description stream

A DecoderInit, an IndexDecoderInit, an index stream, and a description stream.

Before an index stream can be queried both the DecoderInit for the BiM stream to which the index stream belongs, and the IndexDecoderInit for the index stream must be acquired. However acquiring fragments from the description stream, without querying the index stream, only requires the DecoderInit to be acquired.

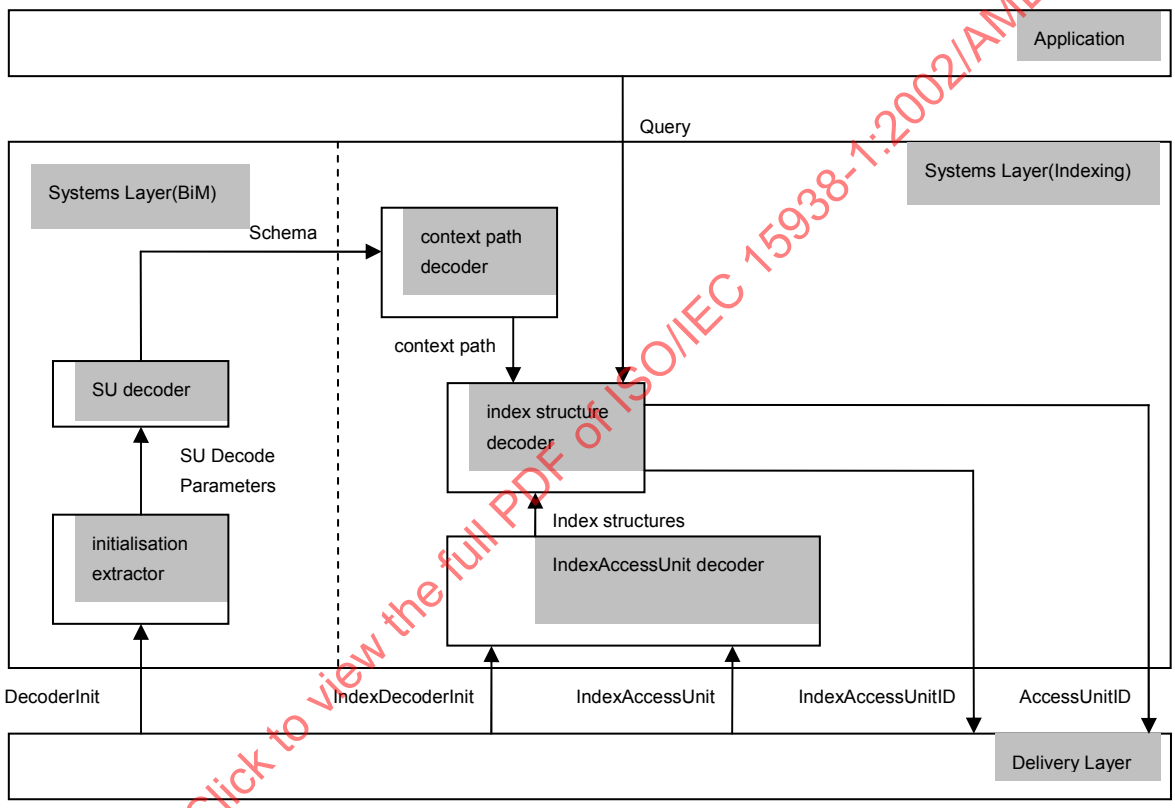

Figure Amd2.1 — Indexing Enabled Terminal Architecture Extension.

All components of Systems Layer(Indexing) section are non-normative

The Terminal Architecture for a BiM enabled terminal may be extended to support indexing, as shown in Figure Amd2.1. Figure Amd2.1 shows only the extensions to the Terminal Architecture, and not the complete architecture. The components shown in the Systems Layer(BiM) section of Figure Amd2.1 are the existing components of the standard Terminal Architecture.

Add the following subclause 5.9.2:

5.9.2 Options for multi criteria query

There are two main methods of querying for fragments when there are multiple criteria, multi-value indexing, and multi-stage indexing. These different methods are distinct and offer two complementary optimizations.

Multi-value indexing allows the whole data set to be indexed in a very compact manner. The size of the index stream and the number of comparisons of values is minimized, allowing the index to perform a multi criteria query using the smallest amount of index data and queries possible. Partitioning of indices with larger data

sets allows maximum IndexAccessUnit sizes to be imposed. This is useful when the underlying transport layer has an imposed, or preferred size for a unit of access, as might be the case with network packets, or transport layer data buffers. It is usual for several IndexAccessUnits to be required to complete a query, hence the IndexAccessUnit is not independent.

Multi-stage indexing allows the data set to be sectioned into multiple smaller index stream segments. This increases both the overall size of the index stream and the number of value comparisons by a moderate amount. However, multi-stage indexing can facilitate more efficient use of resources in client terminal devices where caching of stream index data is necessary but the size of the stream index data prohibits caching all of it. Caching is desirable where either the index stream is not always available or the acquisition time of Index Access Units from the index stream is significant. In a multi-stage index the data size of each independent segment of the index stream is reduced, allowing a whole segment of the index to be cached from the index stream into memory and searched independent of the Index Stream. This is often a better optimisation than attempting to cache a portion of an index arranged as a single segment.

Multi-value indexing is intended to be used in situations where efficient index size is a priority and there are no significant restraints imposed by client terminal resources. There are two formats of multi-value indexing, composite value, and hierarchical single value indexing. In composite value indexing the values are stored as an N column table, where N is the number of related values. In hierarchical single value indexing the values are stored as an N level tree, with the tree's nodes containing 1 column tables (Lists). The composite value index is intended for use where there is unlikely to be multiple instances of the related node with the same value, whilst the hierarchical single value indexing is intended for use where there are many instances of the related node with the same value in the BiM fragments to be indexed. It is important to take care to choose the order of the related nodes carefully, as the related node with most common values placed at the highest level of the hierarchy will usually produce the smallest index stream and the minimum number of value compares when querying.

Multi-stage indexing is intended for use in situations where client terminals with limited resources must access a very large amount of BiM fragments by index, and for which there are a small number of search criteria common for most queries. The common criteria can be used to segment the index stream into a collection of index stream segments, each segment then being accessed and searched independently of other segments. This allows searches to start on an initial segment stored in a cache and then progress to the relevant follow-on segment which, if not cached, must be acquired from the index stream. The search is therefore completed with a minimum number of acquisitions and without requiring the whole multi-stage index to be cached.

Note that the method of caching and cache maintenance is implementation dependent, and not defined in this specification.

Note - Another situation where multi stage indexing can be used, is when consolidating multiple independent index streams. This may be the case if there are multiple providers of BiM fragments each with an associated index stream. Consolidation of the index streams can be achieved easily by modifying each index stream to be a second stage segment. A first stage index would then be created to associate a provider to a second stage segment within the index stream.

Change the following sentence at the end of subclause 7.1 as indicated:

Several other coding modes are initialised in the DecoderInit related to the features used by the binary description stream: the insertion of elements, the transmission of schema information, references to fragments and a fixed length context path.

And add the following sentence at the end of subclause 7.1:

The fixed length context path mechanism provides a simplified addressing of nodes for usage scenarios where only a limited number of nodes need to be addressed. This is done by a table that uniquely maps fixed length codes to full context paths.

In subclause 7.2.2, insert grey marked rows at the position indicated:

If (! NoAdvancedFeatures) {		
AdvancedFeatureFlags_Length	8+	vluimsbf8
/** FeatureFlags ****/**		
InsertFlag	1	bslbf
AdvancedOptimisedDecodersFlag	1	bslbf
AdditionalSchemaFlag	1	bslbf
AdditionalSchemaUpdatesOnlyFlag	1	bslbf
FragmentReferenceFlag	1	bslbf
MPCOnlyFlag	1	bslbf
HierarchyBasedSubstitutionCodingFlag	1	bslbf
ContextPathTableFlag	1	bslbf
ReservedBitsZero	FeatureFlags_Length*8-8	bslbf
/** FeatureFlags end **/		

If (**ContextPathTableFlag**) {		
ContextPathTable()		
}		
*/** FUUConfig - Advanced optimised decoder framework **/*		
If (AdvancedOptimisedDecodersFlag) {		

ContextPathTable {		
ContextPathTable_Length	8+	vluimsbf8
ContextPathCode_Length	8+	vluimsbf8
NumberOfContextPaths	8+	vluimsbf8
CompleteContextPathTable	1	bslbf
for(i=0;i<NumberOfContextPaths;i++){		
ContextPath_Length[i]	5+	vluimsbf5
ContextPath()[i]	ContextPath_Length[i]	
if(!CompleteContextPathTable){		
ContextPathCode[i]	ContextPathCode_Length	bslbf
}		
}		
nextByteBoundary()		
}		

In subclause 7.2.3, insert,

ContextPathTableFlag	Signals the presence of a context path table in the decoder init.
ContextPathTable_Length	Defines the number of bytes used for the indication of the ContextPathTable. Note – This length provides a simple framework to skip the table.
ContextPathCode_Length	Signals the length of the context path codes in number of bits.
NumberOfContextPaths	Signals the number of ContextPaths contained in the ContextPathTable.
CompleteContextPathTable	Signals if the ContextPathTable is complete and ordered according to the assignment of ContextPathCodes. If CompleteContextPathTable is set to '1' the ContextPathCodes are assigned in the order the ContextPaths are specified in the ContextPathTable starting from '1'. If CompleteContextPathTable is set to '0' the ContextPathCodes are assigned explicitly. The ContextPathCode '0' is reserved.
ContextPath_Length	Signals the number of bits used for the following ContextPath[i]()
ContextPath[i]()	Signals the ContextPath as specified in subclause 7.6.2 with the following restrictions: - ContextModeCode is set to '001' - PositionCode() is an empty bitfield
ContextPathCode[i]	Signals the ContextPathCode of .ContextPath[i]

In subclause 7.6.2, insert grey marked rows at the position indicated:

FragmentUpdateContext () {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(`NumberOfSchemas`))	uimsbf
ContextModeCode	3	bslbf
If (ContextModeCode=='101'){		
ContextPathCode	ContextPathCode_Length	bslbf
for (i=0; i < TBC_Counter(ContextPathCode); i++) {		
PositionCode()		
}		
}		
else {		
ContextPath()		
}		
}		

In subclause 7.6.4, insert grey marked rows at the position indicated:

Code	**Context Mode**
…	
101	Navigate in "Absolute addressing mode" from the selector node to the node specified by the Context Path signaled by the ContextPathCode.
110-111	Reserved

Add the following clause 10:

10 Indexing

10.1 Overview

The index is provided to support fast random access into a BiM stream. The index allows access to the FUU's, within the BiM stream, containing a desired XML node, either element or attribute, based on the specification of one or more related node, element or attribute, values. For instance, determining FUU's which contain "Car" elements, who's "Color" attribute is "Red". The search specification uses the context path of the desired XML node, and the relative context paths of the related nodes, this specification is termed the PathIndexKey. This allows a flexible PathIndexKey, which is capable of indexing complex type elements, simple type elements, and attributes according to one or more criteria.

This section gives a general overview of the structure and functionality of the index, and how it is accessed to arrive at a resultant set of FUU's matching a given search criteria.

The index is composed of two parts, the Path Index, and the Value Index. The Path Index allows the particular Value Index relating to the PathIndexKey, specified in the search criteria, to be located. The Value Index allows the set of BiM stream references which contain the desired XML node, with values for the PathIndexKey's related nodes meeting the search criteria. The indexing technologies specified here, allow a Value Index to be partitioned into smaller Value Sub Index structures. This partitioning allows a Value Index to contain an unlimited number of entries without increasing the resource requirements, in particular working memory, of the client.

The data structures in this specification allow for the Value Index to be created with the values for the PathIndexKey's related nodes, to be represented as a hierarchical index, or as a consolidated index. The hierarchical index can offer much smaller index structures, and faster searches, if the PathIndexKey's related nodes values contain significant repetition, as this allows repeated values to be grouped together and entered into the Value Index only once. If there is not significant repetition, then the Value Index can be represented in a consolidated index, which allows all of the values for the PathIndexKey's related nodes to be consolidated and represented within a single index structure. A search on the Value Index will result in a set of zero or more BiM stream references. The BiM Stream references locate the FUUs which contain the desired XML nodes with related node values satisfying the search criteria. The BiM Stream references can optionally specify the position of the desired XML node within each FUU, in addition to the FUU reference itself.

To demonstrate the searching process, consider the example where a simple, single criteria, search is being made for a "Picture" node, whose related node, "Subject", has a value of "Winter".

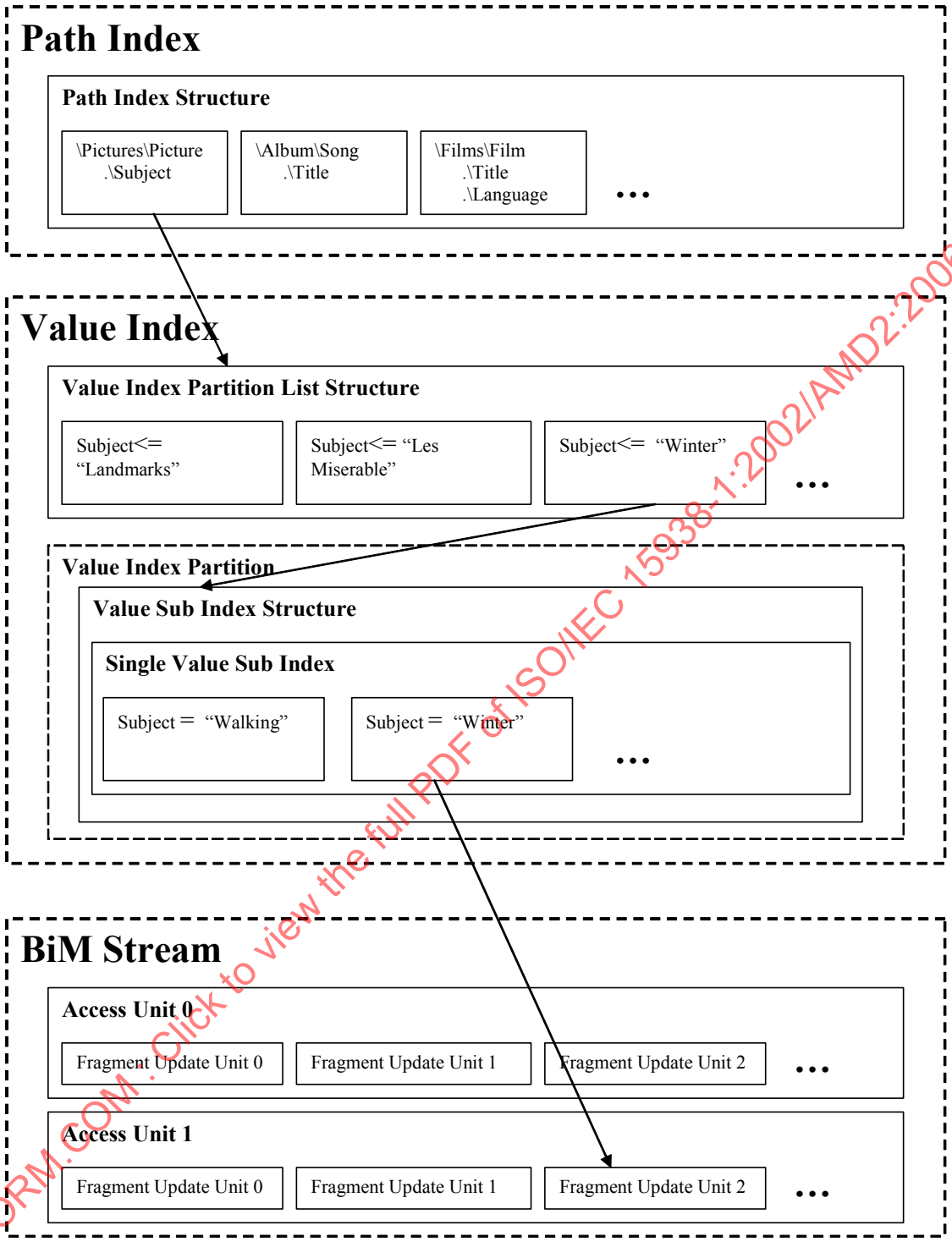

Figure Amd2.2 — Block diagram of the BiM Index structure

The Path Index is first searched to locate the relevant Value Index. This is achieved by scanning the Path Index structure. Once the Value Index has been determined, its Value Index Partition List structure is used to determine which of the Value Index partitions will contain the "Subject" being searched for, in this case "Winter". Now the Value Sub Index structure can be accessed, which in this case contains a Single Value Index, for the values of "Subject" in this partition. These values are then searched to determine the BiM Stream references which match the search criteria.

The next example explains how the Compound Value Index can be used to search for the “Film” element with two related elements “Title” and “Language”

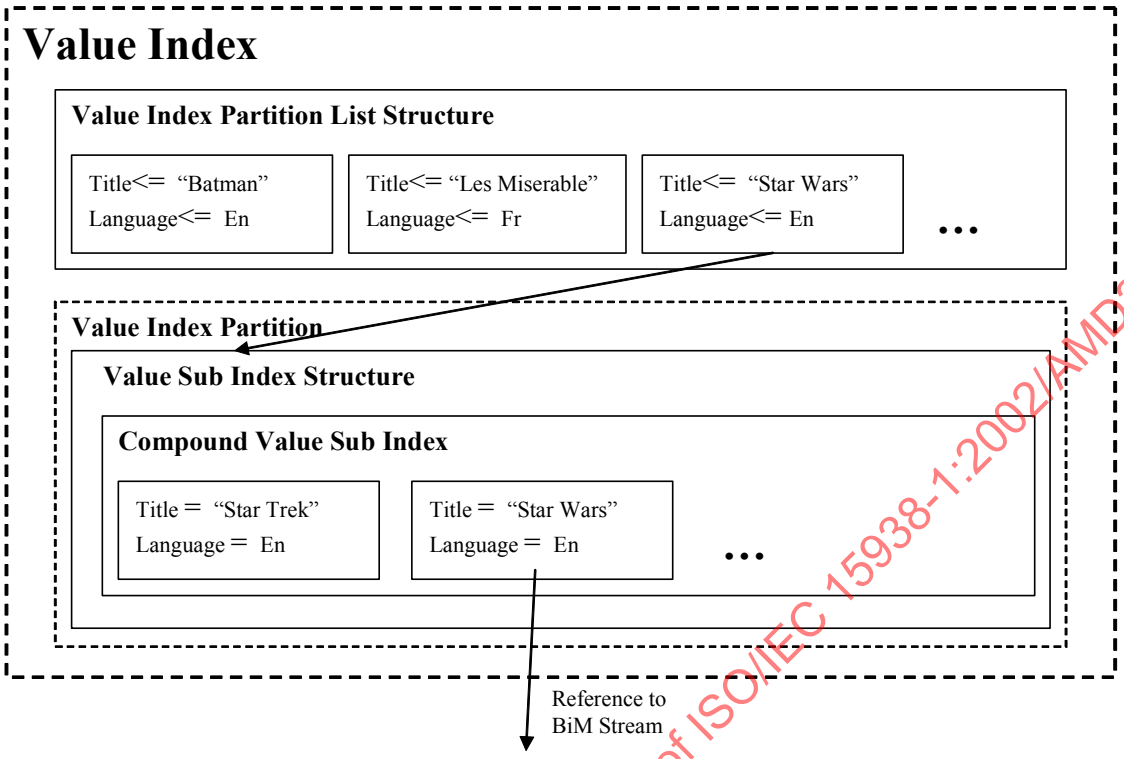

Figure Amd2.3 — Block diagram of the compound value index structure

The process is the same as in the first example, except that the Compound Value Index contains values for both related nodes.

The final example, is the same as the previous example, except that a hierarchy of Single Value Index entries has been used rather then the Compound Value Index.

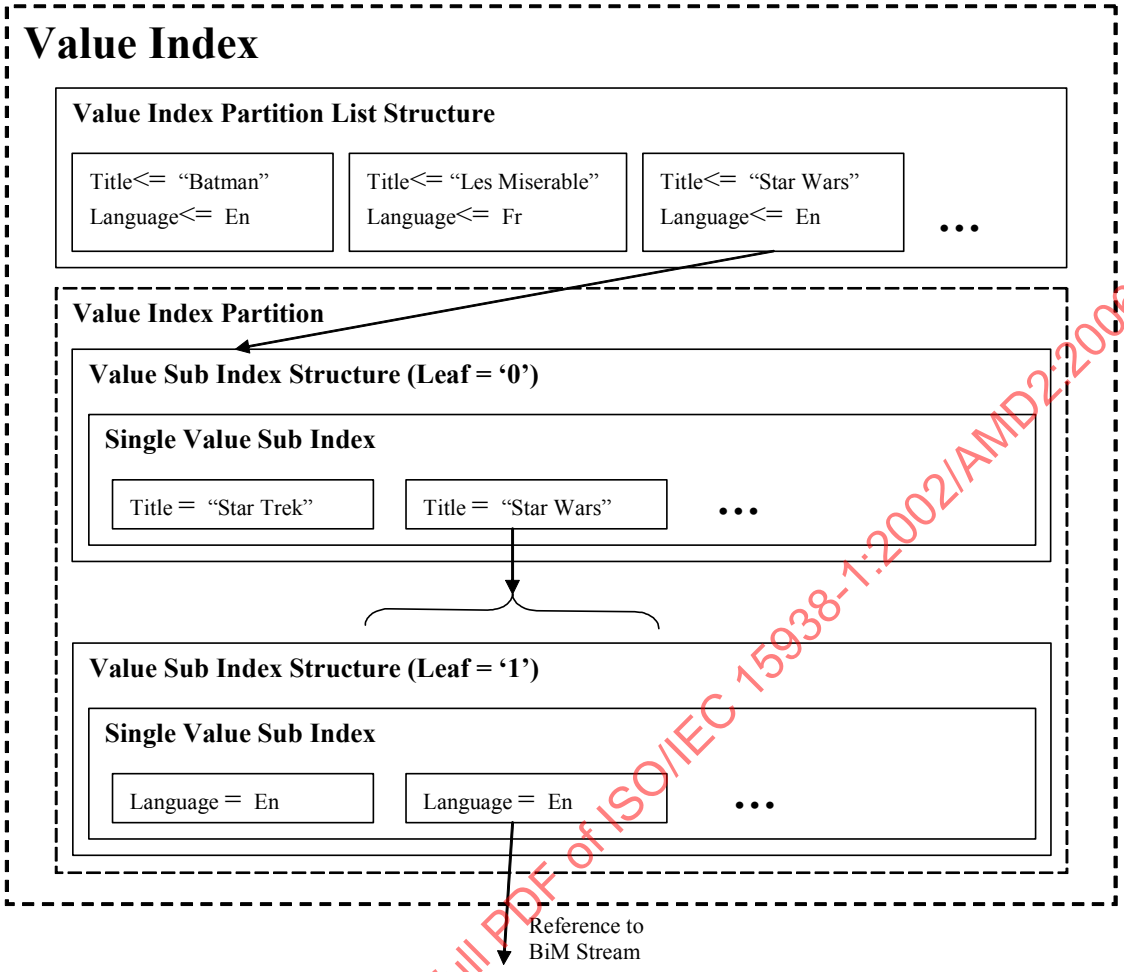

Figure Amd2.4 — Block diagram of the hierarchical single field index structure

This shows that the first related node, "Title" is searched first, but instead returning the BiM stream references, it returns a range to search in the child Sub Value Index. The child is then searched for the correct value for the "Language" element to determine the BiM stream references.

10.2 Characteristics of the delivery layer

The delivery layer is an abstraction that includes functionalities for the synchronization, framing and multiplexing of indexing streams with other data streams. Index streams may be delivered independently or together with the associated Description Stream. No specific delivery layer is specified or mandated by ISO/IEC 15938.

A delivery layer (DL) suitable for conveying ISO/IEC 15938 index streams shall have the following properties in addition to the properties defined for decoding of description streams:

— The DL shall provide a mechanism to communicate an index stream from its producer to the terminal.

— The DL shall provide a mechanism by which a random access point to the index stream can be identified.

— The DL shall provide a suitable random access mechanism allowing access to an IndexAccessUnit by use of a 16 bit IndexAccessUnit identifier.

— The DL shall provide a default 16 bit IndexAccessUnit identifier for each PathIndex in the index stream.

— The DL shall provide a mechanism by which a random access point to the description stream can be identified.

— The DL shall provide a suitable random access mechanism allowing access to an Access Unit by use of a 16 bit Access Unit identifier.

— The DL shall provide delineation of the index access units within the index stream, i.e., IndexAccessUnit boundaries shall be preserved end-to-end.

— The DL shall preserve the order of IndexAccessUnits on delivery to the terminal, if the producer of the index stream has established such an order.

— The DL shall provide either error-free index access units to the terminal or an indication that an error occurred.

— The DL shall provide a means to deliver the `DecoderInit` information (see subclauses 6.2 and 7.2) and the IndexDecoderInit information (see subclause 10.3 to the terminal before any index access unit decoding occurs.

— The DL shall provide signalling of the association of an index stream to a description stream.

— If an application requires index access units to be of equal or restricted lengths, it shall be the responsibility of the DL to provide that functionality transparently to the systems layer.

Note - The 16 bit Access Unit ID is independent of the 16 bit Index access Unit ID.

10.3 IndexDecoderInit

10.3.1 Overview

The Index`DecoderInit` specified in this subclause is used to configure parameters required for the decoding of the index access units. There is only one Index`DecoderInit` associated with one index stream.

Main components of the Index`DecoderInit` are an indication of the profile and level of the associated index stream.

Both the DecoderInit for the description stream and the IndexDecoderInit for the index stream must be acquired prior to decoding Index Access Units.

10.3.2 Syntax

IndexDecoderInit () {	**Number of bits**	**Mnemonic**
SystemsIndexProfileLevelIndication	8+	vluimsbf8
}		

10.3.3 Semantics

Name	*Definition*
SystemsIndexProfileLevelIndication	Indicates the profile and level as defined in ISO/IEC 15938-1 to which the description stream conforms. Table Amd2.1 lists the indices and the corresponding profile and level.

Table Amd2.1 — Index Table for SystemsIndexProfileLevelIndication

Index	*Systems Profile and Level*
0	no profile specified
1 – 127	Reserved for ISO Use

10.4 Index Access Unit

10.4.1 Overview

The Index Access Unit id used to collect multiple structures together into a logical unit. For instance all the data for structures belonging to a single Index Access Unit, must be contained in the Data Repository structure within the same Index Access Unit. The grouping of structure into an Index Access Unit would normally be determined by what is simplest and logical for the encoding process, but may also be limited by the underlying transport.

It is the responsibility of the transport layer to provide the retrieval of Index Access Units from their 16 bit Index Access unit id.

Before the Index Access Unit can be interpreted the Decoder Init and the Index Decoder Init must be acquired.

10.4.2 Syntax

IndexAccessUnit (){	No. of Bits	Mnemonic
IndexAccessUnitHeader(){		
num_structures	8	uimsbf
for(j = 0; j < num_structures; j++){		
structure_type	8	uimsbf
structure_id	8	uimsbf
structrue_ptr	24	uimsbf
structure_length	24	uimsbf
}		
}		
for(j = 0; j < **num_structures**; j++) {		
structure[j]()		
}		
}		

10.4.3 Semantics

Name	*Definition*
num_structures	number of structures within Index Access Unit.
structure_type	identifies type of structure, such as data repository. See subclause 10.4.4.
structure_id	stores index id/sub index id, context dependent on structure_type. see subclause 10.4.5.
structure_ptr	bytes offset from start of Index Access Unit.
structure_length	length in bytes of structure

10.4.4 structure_type assignments

Value	*Description*
0x00	Index Configuration Structure (see subclause 10.5)
0x01	Reserved
0x02	Data Repository Structure (see subclause 10.6)
0x03	Path Index Structure (see subclause 10.7)
0x04	Value Index Structure (see subclause 10.8)
0x05	Value Sub-Index Structure (see subclause 10.9)
0x06	BiMStreamReferences Structure (see subclause 10.12)
0x07	Reserved
0x08	Position Codes Structure (see subclause 10.6)
0x09-0xDF	Reserved
0xE0-0xFF	User Defined

10.4.5 structure_type and their matching valid structure_id

structure_type	*structure_id*	*Description*
0x00	0x00-0xFF	Used to identify Path Index Structure to which this configuration relates.
0x01	0x00-0xFF	User Defined

0x02	0x00	Data Repository Structure of type strings (see subclause 10.6.3)
0x02	0x01	Data Repository Structure of type binary data (see subclause 10.6.4)
0x02	0x02-0xFF	Reserved
0x03	0x00	Root Index. This is the Path Index to start a search of a stand alone or hierarchical index (see subclause 10.7)
0x03	0x01-0xFF	Used to identify hierarchical child index (see subclause 10.7)
0x04	0x00-0xFF	Used to identify a specific instance of a value index structure, within an Index Access Unit (see subclause 10.8)
0x05	0x00-0xFF	Used to identify a specific instance of a value sub-index structure, within an Index Access Unit (see subclause 10.9)
0x06	0x00-0xFF	Used to identify a specific instance of a BiMStreamReference structure (see subclause 10.12)
0x07	0x00-0xFF	Reserved
0x08	0x00-0xFF	Reserved
0x09-0xDF	0x00-0xFF	Reserved
0xE0-0xFF	0x00-0xFF	User Defined

Structure types whose structure_id is 'Reserved' shall set structure_id to 0xFF.

10.5 IndexConfiguration

10.5.1 Overview

This structure contains the configuration parameters associated with the index whose PathIndex structure resides within the same IndexAccessUnit and has the same structure_id.

If this structure is not present within the IndexAccessUnit, the following default values shall be used,

PathIndexKey_format 0x00

BTree_order 0x00

global_value_index_config_flag '0'

LocalAccessUnitID Defined by underlying transport layer

The remaining fields will not be referenced within the index, as global_value_index_config_flag is zero.

10.5.2 Syntax

IndexConfiguration() {	No. of Bits	Mnemonic
PathIndexKey_format	8	uimsbf
BTree_order	8	uimsbf
overlapping_Partitions	1	bslbf
CompoundValueSubIndices	1	bslbf
partition_list	1	bslbf
reserved	4	bslbf
global_value_index_config_flag	1	bslbf
BiMStreamReference_format	8	uimsbf
LocalAccessUnitID	16	uimsbf
}		

10.5.3 Semantics

Name	*Definition*
PathIndexKey_format	Specifies the format used for the PathIndexKey entries. See table below.
BTree_order	The order of the BTree, defined as the number of node references per node. If the order is 1, then the BTree is equivalent to an ordered list, as it only has a right hand branch. A value of 0x0 signals an unordered list.
overlapping_Partitions	Indicates that a range of ValueIndexKeys found within a ValueIndexPartition may overlap those within another ValueIndexPartition structure.
CompoundValueSubIndices	Indicates that the single layer encoding format has been used within the ValueSubIndex structures. Sublause 10.10
partition_list	If 1, indicates that there is a partition list of SubValueIndices, If 0, There is only one SubValueIndex, which is contained inline within the ValueIndex structure.
reserved	Bits reserved for future use, set to '1'.
global_value_index_config_flag	If 1, indicates that the following global overlapping_SubValueIndices, single_layer_SubValueIndices, and BiMStreamReference_format values should be used for all value index structures within this index stream.
BiMStreamReference_format	Specifies the format of the BiMStreamReference. (see subclause 10.8.3)
LocalAccessUnitID	The Access Unit id of the Local Access Unit to which Local BiMStreamReferences refer.

PathIndexKey_format	**Format**
0x00	PathIndexKey_literal (see subclause 10.7.5)
0x01	PathIndexKey_context_path (see subclause 10.7.7)
0x02-0xFF	Reserved

10.6 Data Repository

10.6.1 Overview

The Data Repository forms the base structure, used to hold string data and binary data. All references to the data repository are local. i.e. from within the same Index Access Unit. The type of data, which the data repository carries, is indicated by the structures associated structure_id.

10.6.2 Syntax

DataRepository() {	**No. of Bits**	**Mnemonic**
if(structure_id == 0x00) {		
string_repository()		
}		
else if(structure_id == 0x01) {		
binary_repository()		
}		
else {		
Reserved		
}		
}		

10.6.3 string_repository

10.6.3.1 Overview

The string repository is used to hold all strings used by structures within the same Index Access Unit.

There shall only ever be one string repository per Index Access Unit. References to this repository are always local (that is, from the same Index Access Unit). Support is provided for identifying the string encoding system, to enable the use of non ASCII base character sets. The use of length fields or termination values are dependent on the string encoding used.

10.6.3.2 Syntax

string_repository() {	**No. of Bits**	**Mnemonic**
encoding_type	8	uimsbf
for(i=0; i<strings_count; i++) {		
for(j=0; j<string(i).length; j++) {		
string_character	8+	bslbf
}		
string_terminator	8+	bslbf
}		
}		

10.6.3.3 Semantics

Name	*Definition*
encoding_type	An 8 bit field used to define the character encoding system, according to section 10.6.3.4.

10.6.3.4 Character Encoding and their termination values

encoding_type	*Description*	*Termination Value*
0x00	7 bit ASCII (ISO/IEC 10646-1 [1])	0x00
0x01	UTF-8	0x00
0x02	UTF-16	0x0000
0x03	GB2312	0x0000
0x04	EUC-KS	0x0000
0x05	EUC-JP	0x0000
0x06	Shift_JIS	0x0000
0x07-0xDF	Reserved	Undefined
0xE0-0xFF	User Defined	User Defined

10.6.4 binary_repository

10.6.4.1 Overview

The encoding of data in the binary repository is defined at the point of reference. Each item of data must either have a length explicitly encoded within it, or a length implicitly understood by the decoder (i.e. fixed length). No provision is made to define the data length within the binary data repository structure.

All entries shall be byte aligned.

There shall only ever be one binary data repository within a single Index Access Unit.

10.6.4.2 Syntax

binary_repository() {	**No. of Bits**	**Mnemonic**
for(i=0; i<value_count; i++) {		
for(j=0; j< length; j++) {		
value_byte	8	bslbf
}		
}		
}		

10.6.4.3 Semantics

Name	*Definition*
`value_byte`	A byte of binary value data

10.7 PathIndex

10.7.1 Overview

A path index structure capable of supporting, unordered lists, ordered lists, and b-trees is desirable, as each is optimal for different applications. However it is not desirable to implement multiple path index structure handlers in every decoder, and so a multipurpose structure is defined. This structure can be parsed by the same structure decoder whether it is list or b-tree, with minimal additional overhead in the decoder.

10.7.2 Syntax

PathIndex() {	**No. of Bits**	**Mnemonic**
for(int j=0; j<num_nodes; j++) {		
PathIndexNode[k]()		
}		
}		

10.7.3 PathIndexNode

10.7.3.1 Overview

The PathIndexNode represents the encoding of a single node within the PathIndex. Each PathIndexNode may contain one or more PathIndexKeys, depending on the BTree order.

10.7.3.2 Syntax

PathIndexNode() {	**No. of Bits**	**Mnemonic**
node_reference	8+	vluimsbf8
if(BTree_order > 1) {		
number_of_entries	ceil($\log^2$ BTree_order-1)	bslbf
}		
for(int k=0; k<number_of_entries; k++) {		
PathIndexKey[k] ()		
nextByteBoundary()		
if(BTree_order > 1) {		
node_reference[k]	8+	vluimsbf8
}		
ValueIndex_reference [k]()		
}		
}		

10.7.3.3 Semantics

Name	*Definition*
node_reference	A byte offset to the next sibling or child node within the path index. If the PathIndexKey to be located is less than that of the nodes PathIndexKey, then the preceding node_reference provides a link to the next level that should be searched within the B-Tree. If the index you are trying to locate is greater than the last index_key within the node then the last node reference provides a link to the next level that should be searched. If the node_reference is set to 0x00, then the bottom of the B-Tree has been reached and so the item can not be found within the index list.
number_of_entries	defines the number of keys within this index node. As the index node cannot have 0 keys the value is the number of keys -1.
PathIndexKey	This is the key to be compared against the PathIndexKey to be located. Entries will always be in increasing order.

10.7.4 PathIndexKey

10.7.4.1 Overview

The path index key is used by the client to identify and locate a Value Index for a query it wishes to perform. The path index key identifies the paths of nodes which have been indexed, and the paths of the values used to index the node.

10.7.4.2 Syntax

PathIndexKey () {	**No. of Bits**	**Mnemonic**
if(**PathIndexKey_format** == 0x00) {		
PathIndexKey_litteral ()		
} else if (**PathIndexKey_format** == 0x01) {		
PathIndexKey_context_path ()		
} else {		
undefined		
}		
}		

10.7.5 PathIndexKey_literal

10.7.5.1 Overview

The use of literals to identify indexed nodes and value nodes allows a value index to be located by the use of well known literals. This allows low end clients which have only predetermined searching capabilities fixed within their software to locate index paths via a simple 16 bit number. The PathIndexKey_literal also allows flexibility for client and server to use an application specific alternative as a key within the path index.

10.7.5.2 Syntax

PathIndexKey_literal () {	**No. of Bits**	**Mnemonic**
PathIndexKey_literal_value ()		
num_value_nodes	8	uimsbf
for(k = 0; k < **num_value_nodes**; k++) {		
PathIndexKey_literal_value ()		
value_encoding	16	uimsbf
}		
}		

10.7.5.3 Semantics

Name	*Definition*
num_value_nodes	The number of value nodes.
value_encoding	Signals the method of encoding used for the index key value. (subclause 10.7.5.4)

10.7.5.4 Value_encoding

10.7.5.4.1 Interpretation

value_encoding	**value encoding interpretation**
0x0000 – 0x00FF	Field is a 16 bit offset in bytes from the start of the string repository structure.
0x0100 – 0x01FF	Field contains an inline 2-byte value.
0x0200 – 0x0201 0x0300 0x0401	Field contains an inline 4-byte value.
0x0204 - 0x0206	Field contains an inline 1-byte value.
0x0202 - 0x0203	Field is a 16 bit offset in bytes from the start of the binary data repository.
0x0302 0x0400	Field contains an inline 8-byte value.
0x0204 – 0x02FF 0x0402 – 0x04FF	Undefined.
0x0500 – 0xFFFF	Reserved for future use.

10.7.5.4.2 Respective Sizes

value_encoding	**Description**	**Encoding**	**Size in bits**
0x0000	string type	Null-terminated string	variable (8+)
0x0001 – 0x00FF	Reserved for custom string types		
0x0100	signed short	two's complement – Big Endian	16

0x0101	unsigned short	unsigned binary – Big Endian	16
0x0102 – 0x01FF	Reserved for custom 2 byte types		16
0x0200	signed long	two's complement – Big Endian	32
0x0201	unsigned long	unsigned binary – Big Endian	32
0x0202	variable length signed integer	One bit represents sign (0: positive, 1:negative), followed by abs(value) using vluimsbf5	variable (6+)
0x0203	variable length unsigned integer	vluismbf8	variable (8+)
0x0204	boolean	0:False 1:True	8
0x0205	signed byte	Two's complement	8
0x0206	unsigned byte	unsigned binary	8
0x0207 – 0x02FF	Reserved for custom integer types		
0x0300	signed float	IEEE standard 754-1985 – Big Endian	32
0x0301	reserved		
0x0302	signed double	IEEE standard 754-1985 – Big Endian	64
0x0303 – 0x03FF	reserved		
0x0400	dateTime	Modified Julian Date and Milliseconds (as defined in subclause 10.5.4.4)	64
0x0401	date	Modified Julian Date (as defined in subclause 10.5.4.5)	32
0x0402 – 0x04FF	Reserved for custom binary formats.		
0x0500 – 0xFFFF	Reserved for future use		

10.7.5.5 dateTime Codec

The XML Schema primitive is used widely, and so a specific codec has been designed for representing date time fields.

Times shall be based on GMT, with no provision provided for maintaining the local time offset information. Any requirements to localise time values shall be performed by the receiving terminal.

The dateTime primitive is represented as an 8-byte unsigned integer number (Big-Endian), Days are represented using the first 4 bytes using Modified Julian Date. Time is represented using the last 4 bytes expressed as the number of elapsed milliseconds since 00:00:00 hours.

The origin for the Modified Julian Date shall be Midnight 17th November 1858.

Example dates:

Date	Modified Julian Date
1st April 1980	44 330
30th January 2000	51 573
1st March 2001	51 969

Example dateTimes:

dateTime value	Encoded value
1980-04-01T02:00:00Z	0x0000AD2A006DDD00
2000-01-30T12:10:01Z	0x0000C975029C59A8
2001-03-01T00:00:00Z	0x0000CB0100000000

10.7.5.6 date Codec

The XML Schema primitive simple type date describes a date within the Gregorian calendar. Within the XML the date takes the form of a string as defined by ISO/IEC 8601.

The XML Schema date primitive shall be represented as a 4-byte unsigned integer (Big-Endian). It shall contain the number of days using the Modified Julian Date format, as described in subclause 10.7.5.5.

10.7.6 PathIndexKey_literal_value

10.7.6.1 Overview

Literal values for the PathIndexKey are used where the encoder and decoder have additional knowledge about the context paths used within the index.

An example is where indexed nodes are always aligned with fragments, and the fragments use a limited set of context paths. In this instance the context paths used for the fragments, and hence the indexed node, are assigned a number which is known to the encoder and the decoder. This number can then be used in place of the context path.

Another instance where literal keys are useful is where a fragment is to be indexed based on data not contained in the source instance document. For instance an index of fragments which have been changed in the last day could be generated.

The literal index key places a requirement for both the encoder and decoder to understand the meaning of the literal key. Otherwise the decoder will not be able to use the index. Any index based on an unknown literal key does not prevent the decoder from using other value Indices specified within the same path index.

10.7.6.2 Syntax

PathIndexKey_literal_value () {	**No. of Bits**	**Mnemonic**
literal_type	16	uimsbf
if(**literal_type** < 0x8000) {		
UserDefined_literal	(Lower 15 bits of **literal_type**)	
} else if (**literal_type** < 0xFF00) {		
UserDefined_inlined	8*(**literal_type** & 0xFF)	
} else if (**literal_type** < 0xFFFD) {		
reserved	16	uimsbf
} else if (**literal_type** == 0xFFFE) {		
context_path_length	8+	vluimsbf8
context_path	variable	uimsbf
} else if (**literal_type** == 0xFFFF) {		
indexed_node_xpath_ptr	16	uimsbf
}		
}		

10.7.6.3 Semantics

Name	*Definition*
literal_type	The type of literal value encoding
UserDefined_literal	15 bit literal value of user defined significance. Value is lower 15 bits of **literal_type**.
UserDefined_inlined	variable length inlined used defined data
context_path_length	The length of the index root context path in bits.
context_path	This is a variable length field, which identifies the index root element context path. This is encoded as a context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002, with position codes normalized to 1.
indexed_node_xpath_ptr	Pointer to a W3C XPath expression within the string repository

10.7.7 PathIndexKey_context_path

10.7.7.1 Overview

The PathIndexKey_context_path allows a client device to determine and use value indices without prior knowledge of what the Path Index is likely to contain. This provides a very flexible index to be generated by a server, and still be decoded by a client.

In contrast with the literal path index key, the context path does not require any external definitions to be known by the encoder or decoder, other than the XML schema.

10.7.7.2 Syntax

PathIndexKey_context_path () {	**No. of Bits**	**Mnemonic**
indexed_node_context_path_length	8+	vluimsbf8
indexed_node_context_path	variable	uimsbf
do {		
valuenode_indicator	1	bslbf
if(**valuenode_indicator** == '1') {		
valuenode_context_path_length	8+	vluimsbf8
valuenode_context_path	variable	uimsbf
value_encoding	16	uimsbf
}		
} while(**valuenode_indicator** == '1')		
if(num_valuenodes() == 0)		
{		
value_encoding	16	uimsbf
}		
}		

10.7.7.3 Semantics

Name	*Definition*
indexed_node_context_path_length	The length of the index root context path in bits.
indexed_node_context_path	This is a variable length field, which identifies the index root element context path. This is encoded as a context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002. If position code information is present within the context path, it shall be ignored.
valuenode_indicator	A '1' indicates another value node follows A '0' indicates no more value nodes follow (End of list)
valuenode_context_path_length	The length valuenode_context_path in bits.
valuenode_context_path	This is a variable length field, which identifies the context path of the value node. This is encoded as a relative context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002. If position code information is present within the context path, it shall be ignored.
value_encoding	Signals the method of encoding used for the index key value. (subclause 10.7.5.4)
num_valuenodes()	Return the number of value nodes defined in the preceding key list

Note: If no value nodes are defined in the key list, then the indexed node must be an element of a simple type, or an attribute, and the value of this node is used as the key value. If value nodes are defined in the key list, then the indexed nodes must be elements of simple type, or attributes, and the values of these nodes are used as the key values, In the case where value nodes are defined in the key list, the indexed node may be any element or attribute, and the value of the indexed node is not used within the key.

10.7.8 ValueIndex_reference

10.7.8.1 Overview

The ValueIndex_reference is used to specify the Index Access Unit and structure_id of the referenced ValueIndex structure.

10.7.8.2 Syntax

ValueIndex_reference () {	**No. of Bits**	**Mnemonic**
IndexAccessUnit_identifier	16	uimsbf
ValueIndex_identifier	8	uimsbf
}		

10.7.8.3 Semantics

Name	*Definition*
IndexAccessUnit_identifier	The ID of the Index Access Unit containing the referenced Value Index structure.
ValueIndex_identifier	The ID of the Value Index structure within the Index Access Unit. This is carried in the structure_id field of the Index Access Unit header.

10.8 ValueIndex

10.8.1 Overview

The ValueIndex structure is the top level of an index. It provides a list of all ValueSubIndexReference fields and the ranges of ValueIndexKeys that they contain. When considering a classic indexing system it is normal for there not to be any overlaps in the range of ValueIndexKeys to be found within a given set of sub indexes. This is to minimise the amount of searching required to find a particular value.

Having overlapping ValueSubIndex structures can lead to sequential searching of ValueSubIndex structures, introducing an associated decrease in performance. However in some circumstances it may be desirable to allow this, to simplify index compilation or transmission.

In the case of overlapping ValueSubIndexReferences they shall be declared within the index structure in order of search priority. Where the first declared ValueSubIndexReferences, which may contain the set of required ValueIndexKeys, has the highest priority.

10.8.2 Value Ordering

The ordering of index entries within an index is dependent on a field's primitive XML schema simple type. In the case of strings the order may be dependent on the selected language, and not necessarily in alphanumeric order.

Table Amd2.2 — Defined index order for primitive simple types

Simple Type	Ordering
string	All strings shall be ordered in increasing Lexicographical order. Lexicographical ordering is language dependent, and may not be alphanumeric.
anyURI	Increasing alphanumeric order.
boolean	‘False’ precedes ‘True’
NMTOKEN	Increasing binary representation order
gYear	Increasing numeric value
integer	Increasing numeric value with negative values first
date	Increasing date value
nonNegativeInteger	Increasing numeric (binary) value
positiveInteger	Increasing numeric (binary) value

dateTime	Increasing dateTime (binary) value
duration	Increasing duration (binary)
float	Increasing numeric value (negative values first)
double	Increasing numeric value (negative values first)

Given high_ValueIndexKey, (a_1, a_2, ..., a_n) and (b_1, b_2, ..., b_n), of two arbitrary ValueSubIndices among the ValueSubIndices list, the sorting of ValueSubIndices is determined as follows:

(a_1, a_2, ..., a_n) is larger than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every j($0 \le j \le i-1$), $a_j = b_j$ and $a_i > b_i$.

(a_1, a_2, ..., a_n) is smaller than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every j($0 \le j \le i-1$), $a_j = b_j$ and $a_i < b_i$.

(a_1, a_2, ..., a_n) is equal to (b_1, b_2, ..., b_n) if and only if for every i($1 \le i \le n$), $a_i = b_i$.

Specifically, within the ValueIndexPartitionList() structure, if there is no overlapping between ValueSubIndices, for all j between 0 and ValueSubIndex_count-1 (high_ValueIndexKey[j,0], …, high_ValueIndexKey[j,k]) is smaller than (high_ValueIndexKey[j+1,0], …, high_ValueIndexKey[j+1,k])

"j" is the ValueSubIndex identifier
"k" is the value node identifier

This function high_ValueIndexKey[j,k] takes its value according to the loop defined in the ValueIndexPartitionList() table.

10.8.3 Syntax

ValueIndexPartitionList() {	**No. of Bits**	**Mnemonic**
if (!**global_value_index_config_flag**){		
overlapping_Partitions	1	bslbf
CompoundValueSubIndices	1	bslbf
partition_list	1	bslbf
reserved	5	bslbf
BiMStreamReference_format	8	uimsbf
}		
if(**partition_list** == '1') {		
for (j=0; j<**ValueSubIndex_count**, j++) {		
for(k=0; k<**num_valuenodes**; k++) {		

if (**overlapping_Partitions** == '1') {		
low_ValueIndexKey[j][k]	field encoding dependent	uimsbf
}		
high_ValueIndexKey[j][k]	field encoding dependent	uimsbf
}		
ValueSubIndexReference[j]()		
}		
} else {		
ValueSubIndex()		
}		
}		

10.8.3 Semantics

Name	*Definition*
overlapping_Partitions	When set to '1', indicates that one or more of the value sub indices which form this value index, overlap with respect to the range of values found within the sub index. Where sub indices overlap, the sub indices are declared in descending order of search priority. When set to '0', indicates that the sub indices do not overlap, and the declared sub indices are ordered in ascending order.
CompoundValueSubIndices	indicates the data structures used within the corresponding ValueSubIndex structures to represent keys with multiple values. When set to '1' it indicates that all values for a given index entry are declared together in a single CompoundValueSubIndex structure. When set to '0' it indicates that each value of a key is contained within a separate SingleValueSubIndex structure.
partition_list	If 1, indicates that there is a partition list of SubValueIndices, If 0, There is only one SubValueIndex, which is contained inline within the ValueIndex structure.
BiMStreamReference_format	Identifies the format and interpretation of the BiMStreamReference field which is used within the ValueSubIndex (leaf field). See Table Amd2.3 — BiMStreamReference formats
low_ValueIndexKey	The lowest value that can be referenced by an entry in a given value index partition. The lowest value signalled in low_ValueIndexKey may not be the lowest ValueIndexKey actually present in the given value index partition, it merely indicates that the referenced value index partition structure may contain entries with ValueIndexKeys in the given range. The size and type of encoding used and the interpretation of the low_ValueIndexKey are defined by the value_encoding within the PathIndex structure.
high_ValueIndexKey	The highest ValueIndexKey that can be referenced by the given value index partition. The highest value signalled in high_ValueSubIndex may not be the highest value actually present in the given fragment, it merely indicates that the referenced value index partition structure may contain entries with ValueIndexKeys in the given range. The size and type of encoding used and the interpretation of the high_ValueIndexKey are defined by value_encoding within the PathIndex structure.

Table Amd2.3 — BiMStreamReference formats

Value	Meaning
0x00	local_BiMStreamReference (see subclause 10.14.3)
0x01	remote_BiMStreamReference (see subclause 10.14.2)
0x02	local_BiMStreamReference_with_position (see subclause 10.14.5)
0x03	remote_BiMStreamReference_with_position (see subclause 10.14.4)
0x04	PathIndexReference (see subclause 10.14.6)
0x05 – 0x3F	reserved
0x40 – 0x7F	User Private
0x80	local_BiMStreamReference_vl (see subclause 10.14.8)
0x81	remote_BiMStreamReference_vl (see subclause 10.14.6)
0x82	local_BiMStreamReference_with_position_vl (see subclause 10.14.10)
0x83	remote_BiMStreamReference_with_position_vl (see subclause 10.14.9)
0x84	PathIndexReference_vl (see subclause 10.15.5)
0x85 – 0xDF	reserved
0xE0 – 0xFF	User Private

It should be noted that the high_ValueIndexKey for all but the first value node may be lower than the previous high_ValueIndexKey sub index entry. This is caused when there is a difference in the value of the parent value node.

For example if we have an index keyed on channel & event time nodes, we could have a set of sub indexes with the following ranges:

Sub index 1 – channel high_ValueIndexKey= ‘3’, event time high_ValueIndexKey= ‘12:00’

Sub index 2 – channel high_ValueIndexKey= ‘4’ event time high_ValueIndexKey= ‘09:00’

Where the index uses CompoundValueSubIndeces, the ordering of the high_ValueIndexKey shall match that defined for the index within the PathIndex structure.

When defining the range of values that a particular value index partition shall cover, sufficient space should be left to enable the addition of further index entries without impacting other value index partitions. For example if a ValueSubIndex can hold a maximum of say 64K entries, it is recommended that the range of current entries should equal around half to two thirds the space. This leaves plenty of room for additional entries without having to changing the way in which the value index is split into value index partitions.

10.8.5 ValueSubIndexReference

10.8.5.1 Overview

The ValueSubIndex_reference is used to specify the Index Access Unit and structure_id of the referenced ValueSubIndex structure.

10.8.5.2 Syntax

ValueSubIndexReference () {	**No. of Bits**	**Mnemonic**
ValueSubIndex_IndexAccessUnitID	16	uimsbf
ValueSubIndex_identifier	8	uimsbf
}		

10.8.5.3 Semantics

Name	*Definition*
ValueSubIndex_IndexAccessUnitID	The id of the Index Access Unit carrying the first ValueSubIndex of the described value index partition.
ValueSubIndex_identifier	This field identifies the ValueSubIndex structure instance containing the described value sub index. This value is carried within the structure_id field of the Index Access Unit header.

10.9 ValueSubIndex

10.9.1 Overview

A value sub index provides references to fragments, which contain values within the range specified for this value sub index. The structure supports indexes with both single and multiple value keys. In the case of indices with multiple value keys, the syntax provides two methods:

- SingleLayer CompoundValues - All values define together within a single ValueSubIndex.
- MultiLayer SingleValues- Each ValueSubIndex indexes a single value of a key.

10.9.1.1 SingleLayer CompoundValue SubIndex

Single Layer Structures provide a simple mechanism for describing multiple value key indices. As each entry in the structure can be decoded one by one in a straightforward manner, this structure would be preferred in a situation where the received index data need to be reorganised in the receiver before its use. Note that the index data can be restructured inside the receiver according to its own storage method and query processing policy. For example, a receiver may want to reorganise one of the received indices in its own B-tree index.

In addition, the Single Layer Structure provides an efficient mechanism for representing multiple value indexes, where there is typically a one to one mapping e.g. <surname, givenname>.

10.9.1.2 Multi Layer SingleValue SubIndex

Multi Layer Structures provide an efficient mechanism for describing multiple value indexes with common single value indexes. This is achieved with the use of multiple SingleValueSubIndex structures, where each structure is used to describe one layer of a multiple value index, (layer is equal to a key field of a multi field index).

Each index entry within the SingleValueSubIndex, point to further SingleValueSubIndex structures (except for the leaf field), which contain index entries of the next layer.

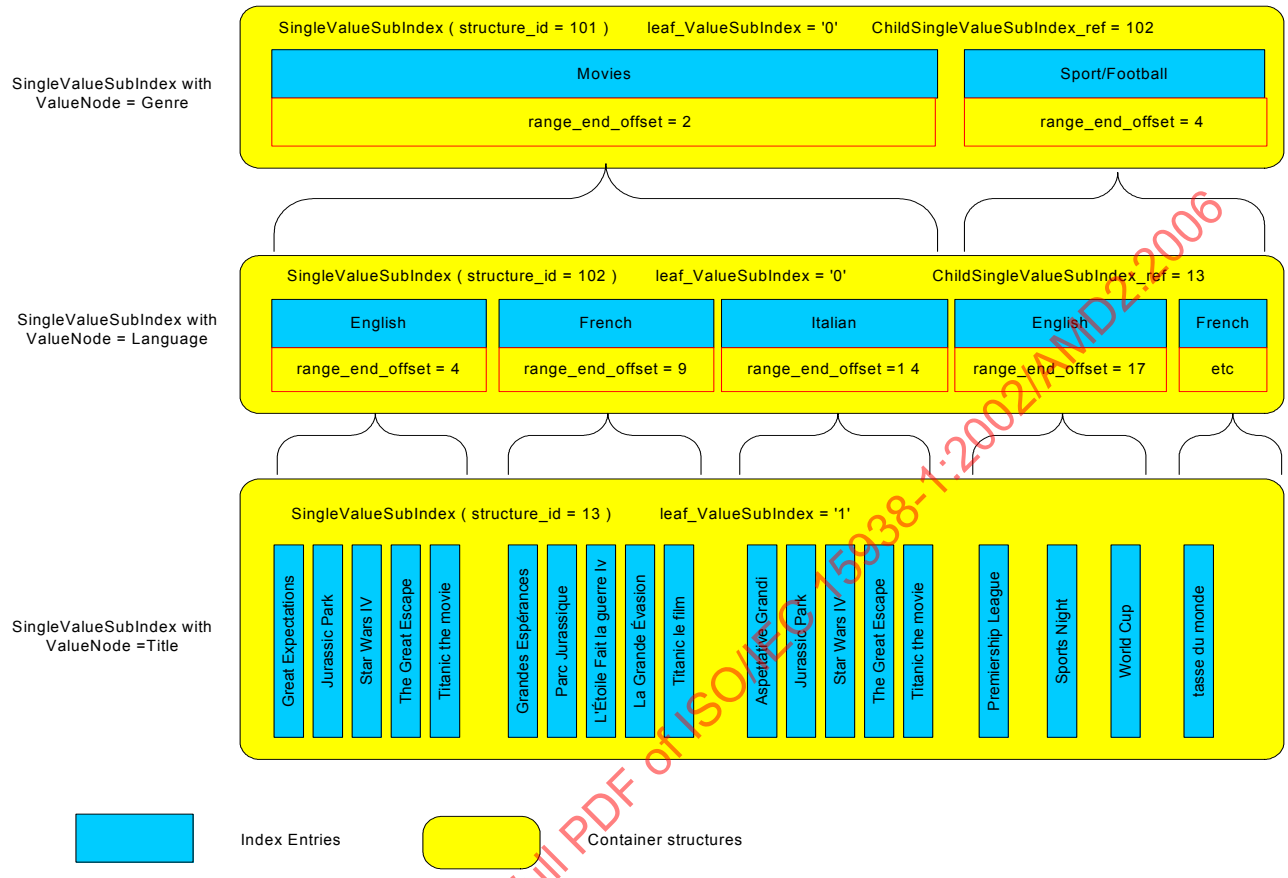

Figure Amd2.5 — Example ValueSubIndex structure (using multi layer syntax) for an index with 3 key fields (Genre, Language, & Title)

The ValueSubIndex structure is formed of two parts:

- ValueSubIndex_header.
- ValueSubIndex_entries.

The ValueSubIndex_header defines how the ValueSubIndex_entries sub structure should be interpreted, and indirectly defines the size of each index entry.

All entries within the ValueSubIndex_entries sub structure are ordered in ascending order. All entries are also of a fixed size, which enables the sub structure to be efficiently searched using a binary search algorithm.

The number of entries within the structure is not explicitly defined, but can be inferred as follows:

num_entries = (structure_length - sizeof(ValueSubIndex_header))/sizeof(ValueSubIndex_entry)

It should be noted that the syntax used within SingleValueSubIndex structures is not always common across all sub indices. Therefore the header of each SingleValueSubIndex should be parsed to infer the syntax used within a given instance.

10.9.2 Syntax

ValueSubIndex() {	No. of Bits	Mnemonic
ValueSubIndex_header()		
if(**CompoundValueSubIndices** == '0') {		
SingleValueSubIndex ()		
} else {		
CompoundValueSubIndex ()		
}		
}		

10.9.3 Semantics

Name	*Definition*
CompoundValueSubIndices	This value is obtained from the value index which referenced this value sub-index.

10.9.4 ValueSubIndex_header

10.9.4.1 Overview

Given ValueIndexKeys, (a_1, a_2, ..., a_n) and (b_1, b_2, ..., b_n), of two CompoundValueIndex_entries, the order between the two entries is determined as follows:

(a_1, a_2, ..., a_n) is larger than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every j($0 \le j \le i-1$), $a_j = b_j$ and $a_i > b_i$.

(a_1, a_2, ..., a_n) is smaller than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every j($0 \le j \le i-1$), $a_j = b_j$ and $a_i < b_i$.

(a_1, a_2, ..., a_n) is equal to (b_1, b_2, ..., b_n) if and only if for every i($1 \le i \le n$), $a_i = b_i$.

Specifically, within the ValueSubIndex() structure, for all j between 0 and num_entries-1 (ValueIndexKey[j,0], …, ValueIndexKey[j,k]) is smaller than (ValueIndexKey[j+1,0], …, ValueIndexKey[j+1,k])

10.9.4.2 Syntax

ValueSubIndex_header () {	No. of Bits	Mnemonic
leaf_ValueSubIndex	1	bslbf
multiple_BiMStreamReferences	1	bslbf
reserved	6	bslbf
}		

10.9.4.3 Semantics

Name	*Definition*
leaf_ValueSubIndex	This shall be set to '1' when the ValueSubIndex carries the leaf field of an index (last indexed field). Which indicates that the structure contains references to fragments, and not to further ValueSubIndex structures. This field is only used within multi layer value sub indexes. When a single layer sub index is being described this flag shall be ignored.
multiple_BiMStreamReferences	A flag which when set to '1' indicates that there are potentially multiple referenced fragments which have the same set of ValueIndexKeys. This provides a more bandwidth efficient mechanism, when multiple fragments have the same set of ValueIndexKeys. The actual BiMStreamReferences are carried in a separate structure within an Index Access Unit, and an offset is used to reference the set of relevant BiMStreamReferences within the structure. When the flag is set to '0' it indicates that BiMStreamReference are defined inline.

10.10 CompoundValueSubIndex

10.10.1 Overview

The CompoundValueSubIndex allows a set of BiMStreamReferences to be addressed as a set of one or more ValueIndexKeys. The compound value sub-index groups the values together in a flat index. This is beneficial if the different data values are uncorrelated.

10.10.2 Syntax

CompoundValueSubIndex () {	**No. of Bits**	**Mnemonic**
ValueSubIndex_entries {		
for (j=0; j<num_entries; j++) {		
for(f=0; f<num_fields; f++) {		
ValueIndexKey	value encoding dependent	**uimsbf**
}		
if(multiple_BiMStreamReferences == '1') {		
BiMStreamReference_end_offset	**16**	**uimsbf**
}		
else {		
BiMStreamReference()		
}		
}		
}		
}		

10.10.3 Semantics

Name	*Definition*
ValueIndexKey	The value of the ValueIndexKey of the referenced fragment. The size and meaning of this field depends on the value of the value_encoding member of the relevant PathIndex structure. The values of the ValueIndexKey must be within the range given for this value index partition.
BiMStreamReference_end_offset	When the multiple_BiMStreamReferences flag is set to '1' in the ValueSubIndex_header this field is used to indicate the inclusive end offset within the BiMStreamReferences structure where the set of valid references can be found. The format of these BiMStreamReferences is defined by the BiMStream_reference_format declared within the PathIndex structure.

The BiMStreamReference_start_offset is implicit from the previous entry within the ValueSubIndex, as follows.

- If it's the first entry within the ValueSubINdex_entries then BiMStreamReference_start_offset shall equal 0.
- If it's not the first entry, the previous entries BiMStreamReference_end_offset + 1 shall be used as the current entries inclusive BiMStreamReference_start_offset.

```
if (current index != 0) {
    BiMStreamReference_start_offset = value sub index_entries[current index-1].
BiMStreamReference_end_offset + 1;
 }else {
    BiMStreamReference_start_offset = 0;
}
```

It should be noted that for fixed size BiMStreamReference formats these references are based on BiMStreamReference entries and not byte offsets. The actual byte offset within the BiMStreamReferences structure is calculated as follows:

if(MostSignificantBit(BiMStreamReference_format) == '0')

{

byte_offset = BiMStreamReference_end_offset * sizeof(BiMStreamReference());

}

else

{

byte_offset = BiMStreamReference_end_offset;

}