[MS-XLDM]:
Spreadsheet Data Model File Format

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.
- **Copyrights.** This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL's, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.
- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.
- **Patents.** Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft Open Specification Promise or the Community Promise. If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- **Fictitious Names.** The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications do not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them. Certain Open Specifications are intended for use in conjunction with publicly available standard specifications and network programming art, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

Preliminary Documentation. This Open Specification provides documentation for past and current releases and/or for the pre-release (beta) version of this technology. This Open Specification is final

documentation for past or current releases as specifically noted in the document, as applicable; it is preliminary documentation for the pre-release (beta) versions. Microsoft will release final documentation in connection with the commercial release of the updated or new version of this technology. As the documentation may change between this preliminary version and the final version of this technology, there are risks in relying on preliminary documentation. To the extent that you incur additional development obligations or any other costs as a result of relying on this preliminary documentation, you do so at your own risk.

Revision Summary

Date	Revision History	Revision Class	Comments
01/20/2012	0.1	New	Released new document.

Table of Contents

1 Introduction 10
1.1 Glossary 10
1.2 References 11
1.2.1 Normative References 11
1.2.2 Informative References 11
1.3 Overview 12
1.4 Relationship to Protocols and Other Structures 12
1.5 Applicability Statement 12
1.6 Versioning and Localization 13
1.7 Vendor-Extensible Fields 13

2 Structures 14
2.1 Storage Format of the Stream 14
2.1.1 Spreadsheet Data Model Header 14
2.1.1.1 Byte Order Mark 15
2.1.1.2 Stream Storage Signature 15
2.1.1.3 BackupLogHeaderType 15
2.1.2 Files Section 16
2.1.2.1 Partitions 17
2.1.2.1.1 SdfPartitionType 17
2.1.2.2 File Stream Format 17
2.1.2.2.1 File End Markers 18
2.1.2.2.1.1 CRC Marker 18
2.1.2.3 Log File 19
2.1.2.3.1 SdfBackupLogType 19
2.1.2.3.1.1 SdfBackupLogCollationsType 20
2.1.2.3.1.2 SdfBackupLogLanguagesType 20
2.1.2.3.1.3 SdfFileGroupsType 20
2.1.2.3.1.3.1 SdfFileGroupType 21
2.1.2.3.1.3.1.1 SdfFileGroupClassEnum 22
2.1.2.3.1.3.2 SdfFileListType 22
2.1.2.3.1.3.3 SdfFileListBackupFileType 22
2.1.2.3.1.4 WriteEnum 23
2.1.2.4 CryptKey.bin File 23
2.1.2.4.1 CryptKey.bin File Format 24
2.1.2.4.1.1 CryptKey.bin Structures 24
2.1.2.4.1.1.1 CryptKeyHeader 24
2.1.2.4.1.1.2 Key BLOB 25
2.1.2.4.1.1.2.1 PUBLICKEYSTRUC 26
2.1.2.4.1.1.3 CryptKeyTrailer 27
2.1.2.4.2 Creating an Exponent-of-One Private Key 27
2.1.3 Virtual Directory 29
2.1.3.1 VirtualDirectoryType 29
2.1.3.2 VirtualDirectoryBackupFileType 29
2.2 File Name Generation 30
2.2.1 Top-Level Folder 30
2.2.2 Top-Level Folders 30
2.2.2.1 Cube Folder 30
2.2.2.1.1 Cube Folder Folders 31
2.2.2.1.1.1 Measure Group Folder 31

2.2.2.1.1.1.1 Measure Group Folder Folders 31
2.2.2.1.1.1.1.1 Partition Folder Files 31
2.2.2.1.1.1.2 Measure Group Folder Files 31
2.2.2.1.2 Cube Folder Files 32
2.2.2.1.2.1 Cube Information File 32
2.2.2.1.2.2 MDX Script Metadata File 32
2.2.2.1.2.3 Measure Group Metadata File 32
2.2.2.2 Data Source Folder 32
2.2.2.3 Dimension Folder 33
2.2.2.3.1 Metadata Files 33
2.2.2.3.1.1 Table Metadata Files 33
2.2.2.3.1.2 Table Information File 33
2.2.2.3.1.3 Table Relationship File 33
2.2.2.3.1.4 Column Hierarchy Files 34
2.2.2.3.1.5 User Hierarchy Metadata File 34
2.2.2.3.2 Data Files 34
2.2.2.3.2.1 Column Data Files 34
2.2.2.3.2.2 Table Relationship Index File 35
2.2.2.3.2.3 Column Hierarchy Position–to–Identifier File 35
2.2.2.3.2.4 Column Hierarchy Identifier–to–Position File 35
2.2.2.3.2.5 Column Hierarchy Hash Table 36
2.2.2.3.2.6 Column Hierarchy Dictionary 36
2.2.2.3.2.7 User Hierarchy Files 36
2.2.2.3.2.7.1 Child Count File 36
2.2.2.3.2.7.2 First Child Position File 37
2.2.2.3.2.7.3 Parent Position File 37
2.2.2.3.2.7.4 Multilevel Identifier File 37
2.3 Storage of Data Values 38
2.3.1 Column Data Storage 38
2.3.1.1 File Layout for Column Data Storage Files 39
2.3.1.1.1 General Layout of an .idf File 40
2.3.1.1.2 General Layout of an .idf File That Uses Hybrid Compression 41
2.3.1.1.3 Segment Size Limitations for .idf Files 42
2.3.2 Column Data Dictionary 42
2.3.2.1 File Layout for a Column Data Dictionary 43
2.3.2.1.1 XM_TYPE_LONG and XM_TYPE_REAL Data Dictionary Files 44
2.3.2.1.1.1 Required Hash Elements 45
2.3.2.1.1.2 Vector of Values 45
2.3.2.1.2 XM_TYPE_STRING Data Dictionary Files 46
2.3.2.1.2.1 BLOBs and Base64 Encoding 47
2.3.2.1.2.2 Required Hash Elements 47
2.3.2.1.2.3 Dictionary Page Layout 47
2.3.2.1.2.4 Dictionary String Store (Per Page) Information 49
2.3.2.1.2.4.1 Uncompressed Page Case 50
2.3.2.1.2.4.2 Compressed Page Case 51
2.3.2.1.2.4.3 Second Mark (End of Page Marker) 54
2.3.2.1.2.5 Dictionary Record Handles Vector 54
2.3.2.1.3 Dictionary Structures, Enumerations, and Constants 55
2.3.2.1.3.1 XM_TYPE Enumeration 55
2.3.2.1.3.2 Page Size Limitations for an XM_TYPE_STRING Hash Data Dictionary 55
2.3.2.1.3.3 Page Mask for an XM_TYPE_STRING Hash Data Dictionary 55
2.3.2.1.3.4 Huffman Character Set Mode 56

2.3.2.1.3.5 Record Handle Structures for an XM_TYPE_STRING Hash Data Dictionary 56
2.3.3 Column Data Hierarchy Hash Index 57
2.3.3.1 File Layout for Hash Index Files 57
2.3.3.1.1 Required Elements for All Files That Use Hashing 57
2.3.3.1.2 Required Elements for Hash Index Files 59
2.3.3.1.2.1 Records and Hash Statistics 59
2.3.3.1.2.2 Hash Bin Entries 62
2.3.3.1.2.3 Overflow Hash Entries 63
2.3.3.1.3 Hashing Algorithms 64
2.3.3.1.4 Hash Structures, Enumerations and Constants 64
2.3.3.1.4.1 XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT 64
2.3.3.1.4.2 Hash Algorithm Enumeration and Constant 65
2.3.3.1.4.3 Hash Bin Bucket Size Minimums 65
2.3.3.1.4.4 HashBin Structure 66
2.3.3.1.4.5 HashEntry Structure 67
2.3.3.1.4.6 XM_HASH_ENTRY_COUNT_PER_BIN 69
2.3.4 RowNumber Column 70
2.3.4.1 File Layout for the RowNumber Column 70
2.4 System-Generated Data Files 70
2.4.1 Column Data Position–to–Identifier Mapping 70
2.4.1.1 File Layout for Column Data Position–to–Identifier Mapping File 71
2.4.2 Column Data Identifier–to–Position Mapping 71
2.4.2.1 File Layout for Column Data Identifier–to–Position Mapping File 72
2.4.3 Relationship Index 72
2.4.3.1 File Layout for Relationship Index File 72
2.4.4 User Hierarchy System-Generated Files 73
2.4.4.1 User Hierarchy Child Count 74
2.4.4.1.1 File Layout for User Hierarchy Child Count 74
2.4.4.2 User Hierarchy First Child Position 75
2.4.4.2.1 File Layout for User Hierarchy First Child Position 75
2.4.4.3 User Hierarchy Multilevel Identifier 75
2.4.4.3.1 File Layout for User Hierarchy Multilevel Identifier 76
2.4.4.4 User Hierarchy Parent Position 76
2.4.4.4.1 File Layout for User Hierarchy Parent Position 77
2.5 Metadata Files 77
2.5.1 XMObject Document Node Element 77
2.5.1.1 XMObjectPropertiesType 78
2.5.1.2 XMObjectMembersType 78
2.5.1.3 XMObjectCollectionsType 79
2.5.1.4 XMObjectDataObjectsType 79
2.5.1.5 XMObjectMemberType 80
2.5.1.6 XMObjectCollectionType 80
2.5.1.7 XMObjectDataObjectType 81
2.5.1.8 XMObjectMemberNameEnum 81
2.5.1.9 XMObjectCollectionNameEnum 82
2.5.1.10 XMObjectClassNameEnum 83
2.5.2 XMObject Definitions by class Attribute 89
2.5.2.1 XMObject class="XMSimpleTable" 89
2.5.2.1.1 XMSimpleTablePropertiesType 89
2.5.2.1.2 XMSimpleTableMembersType 90
2.5.2.1.2.1 XMSimpleTableMemberType 90
2.5.2.1.2.2 XMSimpleTableMemberNameEnum 91

2.5.2.1.2.3 XMSimpleTableXMObjectMemberClassNameEnum 92
2.5.2.1.3 XMSimpleTableCollectionsType 93
2.5.2.1.3.1 XMSimpleTableCollectionType 93
2.5.2.1.3.2 XMSimpleTableCollectionNameEnum 94
2.5.2.1.3.3 XMSimpleTableXMObjectCollectionClassNameEnum 94
2.5.2.2 XMObject class="XMTableStats" 95
2.5.2.2.1 XMTableStatsPropertiesType 95
2.5.2.3 XMObject class="XMRawColumn" 96
2.5.2.3.1 XMRawColumnPropertiesType 96
2.5.2.3.2 XMRawColumnMembersType 98
2.5.2.3.2.1 XMRawColumnMemberType 99
2.5.2.3.2.2 XMRawColumnMemberNameEnum 99
2.5.2.3.2.3 XMRawColumnXMObjectMemberClassNameEnum 100
2.5.2.3.3 XMRawColumnCollectionsType 100
2.5.2.3.3.1 XMRawColumnCollectionType 100
2.5.2.3.4 XMRawColumnDataObjectsType 101
2.5.2.3.4.1 XMRawColumnDataObjectType 101
2.5.2.3.4.2 XMRawColumnXMObjectDataClassNameEnum 102
2.5.2.4 XMObject class="XMRelationship" 103
2.5.2.4.1 XMRelationshipPropertiesType 103
2.5.2.4.2 XMRelationshipDataObjectsType 103
2.5.2.4.3 XMRelationshipDataObjectType 104
2.5.2.4.4 XMRelationshipXMDataObjectXMObjectClassNameEnum 104
2.5.2.5 XMObject class="XMRelationshipIndexSparseDIDs" 105
2.5.2.5.1 XMRelationshipIndexSparseDIDsPropertiesType 105
2.5.2.6 XMObject class="XMRelationshipIndexDenseDIDs" 106
2.5.2.6.1 XMRelationshipIndexDenseDIDsPropertiesType 106
2.5.2.7 XMObject class="XMRelationshipIndex123DIDs" 107
2.5.2.8 XMObject class="XMColumnStats" 107
2.5.2.8.1 XMColumnStatsPropertiesType 107
2.5.2.9 XMObject class="XMHierarchy" 111
2.5.2.9.1 XMHierarchyPropertiesType 111
2.5.2.10 XMObject class="XMUserHierarchy" 113
2.5.2.10.1 XMUserHierarchyPropertiesType 113
2.5.2.11 XMObject class="XMHierarchyDataID2PositionHashIndex" 114
2.5.2.12 XMObject class="XMColumnSegment" 114
2.5.2.12.1 XMColumnSegmentPropertiesType 115
2.5.2.12.2 XMColumnSegmentMembersType 115
2.5.2.12.2.1 XMColumnSegmentMemberType 115
2.5.2.12.2.2 XMColumnSegmentMemberNameEnum 116
2.5.2.12.2.3 XMColumnSegmentXMObjectMemberClassNameEnum 117
2.5.2.13 XMObject class="XMPartition" 119
2.5.2.13.1 XMPartitionPropertiesType 119
2.5.2.14 XMObject class="XMMultiPartSegmentMap" 119
2.5.2.14.1 XMMultiPartSegmentMapPropertiesType 120
2.5.2.14.2 XMMultiPartSegmentMapCollectionsType 120
2.5.2.14.3 XMMultiPartSegmentMapCollectionType 121
2.5.2.14.3.1 XMMultiPartSegmentMapXMObjectCollectionClassNameEnum 121
2.5.2.15 XMObject class="XMSegment1Map" 122
2.5.2.15.1 XMSegment1MapPropertiesType 122
2.5.2.16 XMObject class="XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>" 122
2.5.2.16.1 XMSegmentEqualMapEx_PropertiesType 123

2.5.2.17 XMObject class="XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>" 123
2.5.2.18 XMObject class="XMValueDataDictionary<XM_Long>" 124
2.5.2.18.1 PropertiesValueDictionaryType 124
2.5.2.19 XMObject class="XMValueDataDictionary<XM_Real>" 125
2.5.2.20 XMObject class="XMHashDataDictionary<XM_Real>" 125
2.5.2.20.1 HashDictionaryAttributeGroup 126
2.5.2.20.2 PropertiesHashDictionaryRealType 126
2.5.2.21 XMObject class="XMHashDataDictionary<XM_Long>" 126
2.5.2.21.1 PropertiesHashDictionaryLongType 127
2.5.2.22 XMObject class="XMHashDataDictionary<XM_String>" 127
2.5.2.22.1 PropertiesHashDictionaryStringType 128
2.5.2.23 XMObject class="XMRENoSplitCompressionInfo<1>" 128
2.5.2.23.1 XMRENoSplitCompressionInfoPropertiesType 129
2.5.2.24 XMObject class="XMRENoSplitCompressionInfo<2>" 129
2.5.2.25 XMObject class="XMRENoSplitCompressionInfo<3> 129
2.5.2.26 XMObject class="XMRENoSplitCompressionInfo<4> 130
2.5.2.27 XMObject class="XMRENoSplitCompressionInfo<5> 130
2.5.2.28 XMObject class="XMRENoSplitCompressionInfo<6> 131
2.5.2.29 XMObject class="XMRENoSplitCompressionInfo<7> 131
2.5.2.30 XMObject class="XMRENoSplitCompressionInfo<8> 132
2.5.2.31 XMObject class="XMRENoSplitCompressionInfo<9> 132
2.5.2.32 XMObject class="XMRENoSplitCompressionInfo<10> 132
2.5.2.33 XMObject class="XMRENoSplitCompressionInfo<12> 133
2.5.2.34 XMObject class="XMRENoSplitCompressionInfo<16> 133
2.5.2.35 XMObject class="XMRENoSplitCompressionInfo<21> 134
2.5.2.36 XMObject class="XMRENoSplitCompressionInfo<32>" 134
2.5.2.37 XMObject class="XM123CompressionInfo" 135
2.5.2.38 XMRLECompressionInfo 135
2.5.2.38.1 XMRLECompressionInfoPropertiesType 135
2.5.2.39 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>" 136
2.5.2.39.1 XMHybridRLECompressionInfoMembersType 136
2.5.2.39.2 XMHybridRLECompressionInfoMemberType 137
2.5.2.39.3 XMHybridRLECompressionInfoMemberNameEnum 138
2.5.2.39.4 XMHybridRLECompressionInfoXMObjectClassNameEnum 138
2.5.2.40 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>" 140
2.5.2.41 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>" 140
2.5.2.42 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>" 141
2.5.2.43 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>" 141
2.5.2.44 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>" 142
2.5.2.45 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>" 142
2.5.2.46 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>" 143
2.5.2.47 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>" 143

2.5.2.48 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>" 144
2.5.2.49 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>" 144
2.5.2.50 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>" 145
2.5.2.51 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>" 145
2.5.2.52 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>" 146
2.5.2.53 XMObject class="XMHybridRLECompressionInfo<class XM123CompressionInfo>" 146
2.5.2.54 XMObject class="ColumnSegmentStats" 147
2.5.2.54.1 XMColumnSegmentStatsPropertiesType 147
2.5.2.55 XMObject class="XMRawColumnPartitionDataObject" 148
2.5.2.55.1 XMRawColumnPartitionDataObjectPropertiesType 148
2.5.3 Contents of the .tbl.xml Files 149
2.6 Model OLAP Files 149
2.6.1 Load Element Document Node 149
2.6.1.1 MajorObjectTabularModel 150
2.6.1.2 ObjectReferenceTabularModel 150
2.6.1.3 TabularModelElementsGroup Group 151
2.6.2 DataSourceTabularModel 151
2.6.3 DataSourceViewTabularModel 152
2.6.4 DatabaseTabularModel 152
2.6.5 CubeTabularModel 153
2.6.6 DimensionTabularModel 153
2.6.7 MeasureGroupTabularModel 154
2.6.8 PartitionTabularModel 155
2.6.9 MdxScriptTabularModel 155
2.6.10 OLAP Information Files 156
2.6.10.1 Partition Information File 156
2.6.10.1.1 PartitionInformationType 156
2.6.10.2 Dimension Information File 157
2.6.10.2.1 DimensionInformationType 157
2.6.10.2.1.1 DimensionInformationPropertiesType 158
2.6.10.2.1.1.1 DimensionInformationPropertyType 158
2.6.10.2.1.1.2 DimensionInformationMapDataSetType 158
2.6.10.3 Cube Information File 159
2.6.10.3.1 CubeInformationType 159
2.7 Compression 160
2.7.1 XMRENoSplit Compression Algorithms 160
2.7.1.1 XMRENoSplitCompressionInfo<1> 160
2.7.1.2 XMRENoSplitCompressionInfo<2> 162
2.7.1.3 XMRENoSplitCompressionInfo<3> 163
2.7.1.4 XMRENoSplitCompressionInfo<4> 165
2.7.1.5 XMRENoSplitCompressionInfo<5> 166
2.7.1.6 XMRENoSplitCompressionInfo<6> 168
2.7.1.7 XMRENoSplitCompressionInfo<7> 169
2.7.1.8 XMRENoSplitCompressionInfo<8> 171
2.7.1.9 XMRENoSplitCompressionInfo<9> 172
2.7.1.10 XMRENoSplitCompressionInfo<10> 174
2.7.1.11 XMRENoSplitCompressionInfo<12> 175

2.7.1.12 XMRENoSplitCompressionInfo<16> 177
2.7.1.13 XMRENoSplitCompressionInfo<21> 178
2.7.1.14 XMRENoSplitCompressionInfo<32> 179
2.7.2 XM123 Compression Algorithm 180
2.7.2.1 XM123CompressionInfo 181
2.7.3 XMHybridRLE Compression Algorithms 181
2.7.3.1 Conceptual Overview of RLE Entries and Bit-Packing Entries 182
2.7.3.2 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>> 185
2.7.3.3 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>> 186
2.7.3.4 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>> 187
2.7.3.5 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>> 188
2.7.3.6 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>> 189
2.7.3.7 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>> 191
2.7.3.8 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>> 192
2.7.3.9 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>> 193
2.7.3.10 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>> 193
2.7.3.11 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>> 194
2.7.3.12 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>> 196
2.7.3.13 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>> 197
2.7.3.14 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>> 197
2.7.3.15 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>> 198
2.7.3.16 XMHybridRLECompressionInfo<class XM123CompressionInfo> 199
2.7.4 Huffman Compression 199
2.7.4.1 Huffman Implementation Constraints 200
2.7.4.1.1 Classical Unbalanced Huffman Tree 200
2.7.4.1.2 Minimum and Maximum Codeword Sizes 200
2.7.4.1.3 Huffman Alphabet Size 201
2.7.4.1.4 Single and Multiple Character Set Modes 201
2.7.4.1.5 Huffman Information Provided in an XM_TYPE_STRING Dictionary 202
2.7.4.2 Conceptual Overview of a Huffman Tree 203
2.7.5 Xpress Compression 205

3 Structure Examples 206
3.1 tbl.xml Metadata File 206
3.2 Multiple-Segment Column Data .idf File 220
3.3 Dictionary File 222

4 Security 225
4.1 Security Considerations for Implementers 225
4.2 Index of Security Parameters 225

5 Appendix A: Compression Mask for XMRENoSplit Compression Algorithms 227

6 Appendix B: Product Behavior 243

7 Change Tracking 244

8 Index 245

1 Introduction

The Spreadsheet Data Model File Format defines a binary file format that is used to store a portion of a tabular data model, which represents tables, data, and relationships, within a containing spreadsheet file format.

Sections 1.7 and 2 of this specification are normative and contain RFC 2119 language. All other sections and examples in this specification are informative.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Augmented Backus-Naur Form (ABNF)
cyclic redundancy check (CRC)
GUID
language code identifier (LCID)
little-endian
universally unique identifier (UUID)

The following terms are defined in [MS-OFCGLOS]:

assembly
base64 encoding
binary large object (BLOB)
calculated column
hierarchy
measure group
multidimensional expression (MDX)
OLAP
OLAP cube
OLE DB
Online Analytical Processing (OLAP)
partition
table
XML schema definition (XSD)

The following terms are specific to this document:

hybrid compression: A type of data compression that uses a combination of run length encoding and bit-wise compression.

intrinsic hierarchy: A hierarchical data structure that is automatically formed from every single column of data in a spreadsheet and contains one node for every unique data value within each column.

segment map: A data structure that specifies which particular segment contains each individual range of data in a spreadsheet.

tabular data model: A representation of tables, data, and relationships. It must contain at least one table, and can contain definitions for relationships between the table's columns, hierarchical relationships between columns, or calculated columns. It can also contain data values, or connection information to retrieve data values from external locations.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

References to Microsoft Open Specification documents do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[MS-SPBEPO2] Microsoft Corporation, "SharePoint Back-End Protocols Overview Version 2".

[MS-SPFEPO2] Microsoft Corporation, "SharePoint Front-End Protocols Overview Version 2".

[MS-SSAS] Microsoft Corporation, "SQL Server Analysis Services Protocol Specification".

[MS-WUSP] Microsoft Corporation, "Windows Update Services: Client-Server Protocol Specification".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC5234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, http://www.rfc-editor.org/rfc/rfc5234.txt

[XMLSCHEMA1] Thompson, H.S., Ed., Beech, D., Ed., Maloney, M., Ed., and Mendelsohn, N., Ed., "XML Schema Part 1: Structures", W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XMLSCHEMA2] Biron, P.V., Ed. and Malhotra, A., Ed., "XML Schema Part 2: Datatypes", W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

1.2.2 Informative References

[MSDN-AnalysisServices] Microsoft Corporation, "Managing Backing Up and Restoring (Analysis Services)", http://msdn.microsoft.com/en-us/library/ms174874.aspx

[MSDN-CRYPTO] Microsoft Corporation, "Cryptography Reference", http://msdn.microsoft.com/en-us/library/aa380256.aspx

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary".

[MSKB228786] Microsoft Corporation, "How to export and import plain text session keys by using CryptoAPI", http://support.microsoft.com/kb/228786

[MS-OFCGLOS] Microsoft Corporation, "Microsoft Office Master Glossary".

[MS-OFFMACRO] Microsoft Corporation, "Office Macro-Enabled File Format Specification".

[MS-OFFMACRO2] Microsoft Corporation, "Office Macro-Enabled File Format Version 2 Structure Specification".

[MS-XLSB] Microsoft Corporation, "Excel Binary File Format (.xlsb) Structure Specification".

[MS-XLSX] Microsoft Corporation, "Excel Extensions to the Office Open XML SpreadsheetML File Format (.xlsx) Specification".

[XML10] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Third Edition)", February 2004, http://www.w3.org/TR/REC-xml

1.3 Overview

This file format, which is used to store a **tabular data model** file within a spreadsheet file, can contain one or more of the following types of metadata:

- A definition of a data source for data that is stored in a **table**. The data source can be a range of cells in a spreadsheet, one or more tables in a relational database, or a cube that is stored in an **Online Analytical Processing (OLAP)** database.
- The relationships between the included tables, if any.
- A user-defined, hierarchical relationship among the columns of a table.
- Any **calculated columns** that are created as a function of other, existing columns.

This file format can also include connection strings and passwords for accessing external data sources. Any data that is entered directly into the tabular data model—for example, data that is entered manually or by means of a cut-and-paste operation—can also be stored by this file format.

1.4 Relationship to Protocols and Other Structures

This file format is hosted within the structures that are defined in the following references:

- [MS-XLSX] describes a spreadsheet file format.
- [MS-XLSB] describes a spreadsheet file format.
- [MS-OFFMACRO] describes a spreadsheet file format.
- [MS-OFFMACRO2] describes a spreadsheet file format.

This file format is related to the protocols that are defined in the following references:

- [MS-SSAS] describes the protocol for the **OLAP** server on which the OLAP aspects of the metadata are derived. (The metadata that describes the data contained by this file format is based on both a tabular data model and OLAP.)
- [MSDN-AnalysisServices] describes backup and restore operations that produce a file with the .abf extension. This structure is an .abf file.

Portions of this structure are stored as XML, as described in [XML10].

1.5 Applicability Statement

This structure is used to persist a file within a containing file, as described in [MS-XLSX], [MS-XLSB], [MS-OFFMACRO], or [MS-OFFMACRO2]. This structure applies to the case where a user

creates a tabular data model within a session by using spreadsheet software that produces such a containing file.

1.6 Versioning and Localization

This document covers versioning issues in the following areas:

- **Structure Versions:** This document covers the following information:
  - The version of this structure is stored within the file. For more information, see section 2.1.2.3.1
  - Many of the XML elements are stamped with the provider version of the server that created an instance of this structure. For more information, see section 2.5.
- **Localization:** This document covers the following information:
  - All the string values that are stored in the structure are Unicode and hence support any language's Unicode characters.
  - OLAP metadata objects support the user specification of a language and a collation. For more information, see section 2.6.
  - This structure includes a collection of languages and a collection of collations. For more information, see section 2.1.2.3.1.

1.7 Vendor-Extensible Fields

The OLAP metadata objects have an **Annotations** collection, in which vendors can store vendor-specific information. For more details, see section 2.6.

2 Structures

2.1 Storage Format of the Stream

All of the files that are generated by an instance of the Spreadsheet Data Model are formed into a stream and stored within a spreadsheet file. The format of the storage container is also referred to as the Spreadsheet Data Model File Format and is described in the remainder of this section.

The Spreadsheet Data Model file consists of a header that is followed by a partition marker that is then followed by all the files in the directory, with each file separated by a marker. These files are then followed by a backup log and a virtual directory that contains the file list and related information. The three major sections—Spreadsheet Data Model header; stream of files, including the Spreadsheet Data Model backup log file at the end and the partition information at the beginning; and Spreadsheet Data Model virtual directory—are all Spreadsheet Data Model page aligned and MUST be padded with zeros (if necessary) to meet page alignment requirements.

A Spreadsheet Data Model file page MUST be 4096 bytes. This definition of the page size applies only to the Spreadsheet Data Model File Format, not to the formats of the files that are contained inside the Spreadsheet Data Model. File formats within the Spreadsheet Data Model might use a different definition of page size for their formats.

Most of the Spreadsheet Data Model file is saved by using XML metadata (Spreadsheet Data Model header, Spreadsheet Data Model backup log file, and Spreadsheet Data Model virtual directory), with the files themselves being streamed into the Spreadsheet Data Model file directly in their native format (binary or XML). However, some elements within the Spreadsheet Data Model header are binary or calculated values. Likewise, the **cyclic redundancy check (CRC)** file end marker involves the use of a CRC algorithm.

2.1.1 Spreadsheet Data Model Header

The Spreadsheet Data Model header is page aligned but never compressed—even if the Spreadsheet Data Model file as whole has been compressed. Therefore, the header is always one page (4096 bytes) in size and padded with zeros between the last header element and the end of the page.

The Spreadsheet Data Model header consists of several elements. These elements MUST be in the following order and conform exactly as defined.

The first element is the byte order mark (section 2.1.1.1), which MUST be 2 bytes. The byte order mark is also used prior to the beginning of the stream of files (section 2.1.2)—preceding the partition information)—as well as prior to the writing of the Spreadsheet Data Model backup log (section 2.1.2.3), which is the last file in the streamed files section. The byte order mark is not used before the virtual directory section (section 2.1.3).

The second element in the header is the stream storage signature (section 2.1.1.2). These first two elements (byte order mark and stream storage signature) are binary, not XML.

After these first two binary elements, the subsequent elements in the header are XML tags. These elements MUST be in the order that is specified for the **BackupLogHeaderType** complex type (section 2.1.1.3). These XML tags are followed by any padding with zeros that is necessary to fill the page to the page boundary at 4096 bytes.

There are no breaks or padding between any of the elements.

2.1.1.1 Byte Order Mark

The byte order mark indicates to the system the byte order of the file. It is the first element of both the header and, by extension, the entire Spreadsheet Data Model file. There MUST NOT be any breaks before or after this element. The byte order mask consists of 2 bytes. The first byte MUST be set to 0xFF. The second byte MUST be set to 0xFE.

The byte order mark MUST also be used to begin the files section and therefore precedes the partition marker that leads the files section. The byte order mark MUST also be used before the backup log, which is the last file in the files section. For more details about the files section, see section 2.1.2 and section 2.1.3. For more details about the backup log file, see section 2.1.1.5.1.

2.1.1.2 Stream Storage Signature

The stream storage signature indicates to the system that the file is a valid Spreadsheet Data Model file. The stream storage signature is a byte stream. The stream storage signature MUST come directly after the byte order mark and directly before the rest of the header without any breaks.

The stream storage signature MUST be set to the following ASCII string:

```
STREAM_STORAGE_SIGNATURE_)!@#$%^&*(
```

The stream storage signature MUST be encoded in Unicode.

2.1.1.3 BackupLogHeaderType

The **BackupLogHeaderType** complex type is the type of the **BackupLog** element, which is the XML element that contains the XML content of the backup log header (section 2.1.1).

The backup log header format begins with the byte order mark (section 2.1.1.1) and the stream storage signature (section 2.1.1.2) and is page aligned (see section 2.1 and section 2.1.1). The backup log is an XML document. Its document node is the **BackupLog** element.

```
<xs:complexType name="BackupLogHeaderType">
  <xs:sequence>
    <xs:element name="BackupRestoreSyncVersion" type="xs:int"/>
    <xs:element name="Fault" type="xs:boolean"/>
    <xs:element name="faultcode" type="xs:unsignedInt"/>
    <xs:element name="ErrorCode" type="xs:boolean"/>
    <xs:element name="EncryptionFlag" type="xs:boolean"/>
    <xs:element name="EncryptionKey" type="xs:int"/>
    <xs:element name="ApplyCompression" type="xs:boolean"/>
    <xs:element name="m_cbOffsetHeader" type="xs:unsignedLong"/>
    <xs:element name="DataSize" type="xs:unsignedLong"/>
    <xs:element name="Files" type="xs:unsignedInt"/>
    <xs:element name="ObjectID">
      <xs:simpleType>
       <xs:restriction base="xs:string">
         <xs:pattern value=
         "[0-9A-F]{8}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{12}" />
       </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="m_cbOffsetData" type="xs:unsignedLong"/>
  </xs:sequence>
```

```
</xs:complexType>
```

BackupRestoreSyncVersion: The internal version number of the software that has created this file. This value MUST be set to 140.

Fault: A Boolean value specifying that a CRC signature is not being used as an end-of-file marker (section 2.1.2.2.1). This value MUST be set to **false**. The CRC signature MUST be used. The CRC signature is a calculated value (section 2.1.2.2.1.1).

Faultcode: A value that is unused. The value MUST be an integer to avoid load errors, but the value itself does not matter.

ErrorCode: A Boolean value specifying that a CRC signature is being used as an end-of-file marker (section 2.1.2.2.1). This value MUST be set to **true**. The CRC signature MUST be used. The CRC signature is a calculated value (section 2.1.2.2.1.1).

EncryptionFlag: A Boolean value that specifies whether the Spreadsheet Data Model file is encrypted. The header MUST NOT be encrypted (section 2.1.1). This value MUST be set to **false**.

EncryptionKey: The version of encryption that is being used. This value MUST contain an integer to avoid load errors, but the value itself does not matter.

ApplyCompression: A Boolean value that specifies whether compression has been applied to the file. This value MUST be set to **true**. The header is the exception; it is never compressed. Individual files within the Spreadsheet Data Model file can also be compressed, regardless of whether the Spreadsheet Data Model file itself is compressed. The Spreadsheet Data Model file is compressed by using Xpress compression (section 2.7.5).

m_cbOffsetHeader: The byte offset of the beginning of the file list—that is, the byte offset of the virtual directory structure that contains the list of files in the directory. The offset value is calculated from the beginning of the Spreadsheet Data Model file. For example, if the offset is 28,672, the file list (the virtual directory) begins at byte 28,672 (hexadecimal 0x7000) in the file. The offset is Spreadsheet Data Model page aligned and therefore MUST be a multiple of the Spreadsheet Data Model file page size (section 2.1). For more information about the virtual directory that contains the file list, see section 2.1.3.

DataSize: The size, in bytes, of the file list (the virtual directory) in the Spreadsheet Data Model file. For example, if the file size is set to 3748, the entire virtual directory is 3748 bytes in size (section 2.1.3).

Files: The number of file entries in the file list (the virtual directory). For example, if this value is set to 5, five files exist in the virtual directory and five files are serially stored in the Spreadsheet Data Model file.

ObjectID: A value that is unused and MUST be ignored. This value MUST be a valid **universally unique identifier (UUID)**; otherwise, the file might not load.

m_cbOffsetData: A value that indicates the beginning of the stored files section and the end of the header section. The value is in bytes. For example, if the value is 4096, the beginning of the stored files section begins at byte 4096 (hexadecimal 0x1000). For more information about the header and the header size, see section 2.1.1.

2.1.2 Files Section

This section specifies the files in the file stream.

2.1.2.1 Partitions

A partitions marker exists between the Spreadsheet Data File header (including its padding) and the beginning of the actual files in the directory. The partitions marker is preceded by the byte order mark (section 2.1.1.1). The partitions marker is also treated like any other file in the files section and is terminated by a CRC marker (section 2.1.2.2.1.1).

The Partitions section is an XML document with a **Partitions** element as its document node. The **Partitions** element is of type **SdfPartitionsType**.

```
<xs:complexType name="SdfPartitionsType">
  <xs:sequence>
    <xs:element name="Partition" type="SdfPartitionType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Partition: A complex type element that specifies the properties of a partition.

2.1.2.1.1 SdfPartitionType

The **SdfPartitionType** complex type specifies the properties of a partition.

```
<xs:complexType name="SdfPartitionType">
  <xs:sequence>
    <xs:element name="ObjectPath" type="xs:string"/>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="DataSize" type="xs:long"/>
    <xs:element name="Location" type="xs:string"/>
    <xs:element name="DataSourceID" type="xs:string"/>
    <xs:element name="ConnectionString" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

ObjectPath: A value that is unused and MUST be ignored.

Name: A value that is unused and MUST be ignored.

DataSize: A value that is unused and MUST be ignored.

Location: A value that is unused and MUST be ignored.

DataSourceID: A value that is unused and MUST be ignored.

ConnectionString: A value that is unused and MUST be ignored.

2.1.2.2 File Stream Format

All files in the Spreadsheet Data Model file are stored in their native format, whether XML or binary. A CRC marker (section 2.1.2.2.1.1) delineates the end of one file and the beginning of the next file (if present).

The byte order mark (section 2.1.1.1) begins this files section, which is followed by the partitions marker (section 2.1.2.1), which is then all the files except for the backup log. At this point, there is another byte order mark that is followed by the backup log (section 2.1.2.3), which is the last file.

A CRC marker exists between the partitions marker (section 2.1.2.1) and the first file. As stated earlier in this section, all the files in this stream of files are terminated by a CRC marker. Finally, a CRC marker follows the last file, which is the backup log (section 2.1.2.3).

The entire files section MUST be Spreadsheet Data Model file page aligned (section 2.1). Many pages of files could exist, and all the files are streamed in without breaks, except for their CRC markers. However, following the last file in the stream (the backup log), padding with zeros MUST exist from the log's CRC marker to the end of the page boundary.

The virtual directory (section 2.1.3) begins at the start of the next page.

2.1.2.2.1 File End Markers

All the files in the Spreadsheet Data Model file are terminated by an end-of-file marker. A CRC marker is used to indicate the end of a file and, therefore, also the beginning of the next file (if present).

The CRC marker is a calculated value (section 2.1.2.2.1.1).

2.1.2.2.1.1 CRC Marker

The CRC marker provides a calculated signature value that indicates the end of one file and the beginning of the next file (if present). If the CRC marker is being used, the **ErrorCode** element of the header metadata (section 2.1.1.3) will be set to **true**.

CRC signatures are typically used to detect the alteration of data during transmission in communication systems but can also be used to detect the alteration of backup files, such as those in the Spreadsheet Data Model file.

The CRC is calculated according to the following pseudocode:

```
SET constant value CRC32_POLY to 0x04C11DB7
CREATE unsigned 32 bit integer array of 256 elements and name it crc32TableArray
CREATE unsigned 32 bit integer value, name it crcValue and SET it to 0xFFFFFFFF

CALL InitializationOfCRC32TableArray Function (as follows)
    FOR each element iValue in crc32TableArray
        FOR (cValue = ( iValue LEFT_BITSHIFT 24), jValue = 8), continue loop until jValue>0
            SET cValue to result of (cValue BITWISE_AND 0x80000000)
            IF cValue evaluates to TRUE (non-zero) THEN
                SET cValue to result of ( (cValue LEFT_BITSHIFT 1) BITWISE_EXCLUSIVEOR
CRC32_POLY)
            If cValue evaluates to FALSE (zero) THEN
                SET cValue to result of (cValue LEFT_BITSHIFT 1)
            SET crc32TableArray at index position ( iValue ) to cValue
            DECREMENT jValue by 1
        END FOR
    END FOR

CALL Calculation of crc32Value (after InitializationOfCRC32TableArray)(as follows)
    INPUT to function is an array of BYTES, called pBuffer, and also the buffer's length,
cLength
    FOR each byte in pBuffer up to its length
        SET tempIndex to result of ( (crcValue RIGHT_BITSHIFT 24) BITWISE_EXCLUSIVEOR (value
contained by the currently indexed byte in pBuffer))
        SET crcValue to result of ( (crcValue LEFT_BITSHIFT 8) BITWISE_EXCLUSIVEOR
(crc32TableArray at index position (tempIndex) ) )
```

```
END FOR
```

2.1.2.3 Log File

The log contains a list of all the files that are included in the instance of the Spreadsheet Data Model, except for the log itself, the virtual directory, and the partitions section. The log is the last file in the files section (see section 2.1.2.2). The log is an XML document. The document node is the **BackupLog** element.

2.1.2.3.1 SdfBackupLogType

The **SdfBackupLogType** is the type of the **BackupLog** document node element in the file list section of the Spreadsheet Data Model file. It contains a logging of the files that are included in the Spreadsheet Data Model instance.

```
<xs:complexType name="SdfBackupLogType">
  <xs:sequence>
    <xs:element name="BackupRestoreSyncVersion" type="xs:int"/>
    <xs:element name="ServerRoot" type="xs:string"/>
    <xs:element name="SvrEncryptPwdFlag" type="xs:boolean"/>
    <xs:element name="ServerEnableBinaryXML" type="xs:boolean"/>
    <xs:element name="ServerEnableCompression" type="xs:boolean"/>
    <xs:element name="CompressionFlag" type="xs:boolean"/>
    <xs:element name="EncryptionFlag" type="xs:boolean"/>
    <xs:element name="ObjectName" type="xs:string"/>
    <xs:element name="ObjectId" type="xs:string"/>
    <xs:element name="Write" type="WriteEnum"/>
    <xs:element name="OlapInfo" type="xs:boolean"/>
    <xs:element name="Collations" type="SdfBackupLogCollationsType"/>
    <xs:element name="Languages" type="SdfBackupLogLanguagesType"/>
    <xs:element name="FileGroups" type="SdfFileGroupsType"/>
  </xs:sequence>
</xs:complexType>
```

BackupRestoreSyncVersion: A value that MUST be set to 1153.

ServerRoot: The root folder in the originating file system from which the files in the Spreadsheet Data Model were copied.

SvrEncryptPwdFlag: A Boolean value that specifies whether the originating source application supports password encryption. The value MUST be set to **true**.

ServerEnableBinaryXML: A Boolean value that specifies whether the originating source application supports XML metadata in the Spreadsheet Data Model file in binary XML. The value MUST be set to **false**.

ServerEnableCompression: A value that MUST be set to **false**.

CompressionFlag: A value that MUST be set to **false**.

EncryptionFlag: A value that MUST be set to **false**.

ObjectName: The database name.

ObjectId: The OLAP database identifier value.

Write: An enumeration value that specifies the type of access allowed.

OlapInfo: A Boolean value that specifies whether the files came from a data model that is based on OLAP. The value MUST be one of the values that are described in the following table.

Value	Meaning
true	The data files came from a data model that is based on OLAP.
false	The data files came from either a tabular data model or a model that is stored on a server as specified in [MS-SPBEPO2] or [MS-SPFEPO2].

Collations: The name of the collation. The value MAY<1> be restricted to a string that is recognized as valid by the system.

Languages: The language.

FileGroups: The file groups that are contained in the Spreadsheet Data Model file.

2.1.2.3.1.1 SdfBackupLogCollationsType

The **SdfBackupLogCollationsType** complex type specifies a collection of collations that are used by the files included in this structure.

```
<xs:complexType name="SdfBackupLogCollationsType">
  <xs:sequence>
    <xs:element name="Collation" type="xs:string" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Collation: The name of the collation. The value MAY<2> be restricted to a string that is recognized as valid by the system.

2.1.2.3.1.2 SdfBackupLogLanguagesType

The **SdfBackupLogLanguagesType** complex type specifies a collection of languages that are used by the files included in the Spreadsheet Data Model structure.

```
<xs:complexType name="SdfBackupLogLanguagesType">
  <xs:sequence>
    <xs:element name="Language" type="xs:int" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Language: The **language code identifier (LCID)** for the backup log.

2.1.2.3.1.3 SdfFileGroupsType

The **SdfFileGroupsType** complex type specifies the list of files, first as a group and then individually.

```
<xs:complexType name="SdfFileGroupsType">
  <xs:sequence>
    <xs:element name="FileGroup" type="SdfFileGroupType"
```

```
                maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

FileGroup: A group of files with common, specified properties, followed by the list of files in the group and their individual properties.

2.1.2.3.1.3.1 SdfFileGroupType

The **SdfFileGroupType** complex type specifies the properties of a group of files as well as the files and the properties for the member files in that group.

```
<xs:complexType name="SdfFileGroupType">
  <xs:sequence>
    <xs:element name="Class" type="xs:int"/>
    <xs:element name="ID" type="xs:string"/>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="ObjectVersion" type="xs:int"/>
    <xs:element name="PersistLocation" type="xs:int"/>
    <xs:element name="PersistLocationPath" type="xs:string"/>
    <xs:element name="StorageLocationPath" type="xs:string"/>
    <xs:element name="ObjectID">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value=
        "[0-9A-F]{8}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{4}-[0-9A-F]{12}" />
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="FileList" type="SdfFileListType"/>
  </xs:sequence>
</xs:complexType>
```

Class: The OLAP object that the group of files belongs to.

ID: The identifier of the OLAP object.

Name: The name of the object.

ObjectVersion: An internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model. The value MUST be the same as that of the **ObjectVersion** property in the corresponding OLAP metadata objects' file. For information about the OLAP objects, each of which contains an **ObjectVersion** property, see section 2.6.

PersistLocation: The version number that will appear within the file name. For example, if the value is "10" for a dimension object, the file name will end in "10.dim.xml". This value MUST match that of the **PersistLocation** element of the OLAP database object, which is of type **DatabaseTabularModel** (see section 2.6.4).

PersistLocationPath: The folder in which the files of this file group are stored. This value MUST match that of the **DbStorageLocation** element of the OLAP database object, which is of type **DatabaseTabularModel** (section 2.6.4).

StorageLocationPath: A value that is unused and MUST be ignored.

ObjectID: The identifier of this object. This value matches that of the **ObjectID** property of the OLAP object.

FileList: The list of the files that are members of the group and the properties of those files.

2.1.2.3.1.3.1.1 SdfFileGroupClassEnum

The **SdfFileGroupClassEnum** simple type specifies the enumeration values for the **Class** element of the file group.

```
<xs:simpleType name="SdfFileGroupClassEnum">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="100002"/>
    <xs:maxInclusive value="100060"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **SdfFileGroupClassEnum** type.

Enumeration value	Description
100002	Database
100003	Data source
100006	Dimension
100010	Cube
100016	**Measure group**
100021	Partition
100053	Data source view
100060	**Multidimensional expression (MDX)** script

2.1.2.3.1.3.2 SdfFileListType

The **SdfFileListType** complex type specifies the list of files that are in a file group as well as the properties of those files.

```
<xs:complexType name="SdfFileListType">
  <xs:sequence>
    <xs:element name="BackupFile" type="SdFileListBackupFileType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

BackupFile: A complex type that specifies a file that is included in the Spreadsheet Data Model file.

2.1.2.3.1.3.3 SdfFileListBackupFileType

The **SdfFileListBackupFileType** complex type specifies the properties of a file that is included in the Spreadsheet Data Model file.

```
<xs:complexType name="SdFileListBackupFileType">
  <xs:sequence>
    <xs:element name="Path" type="xs:string"/>
    <xs:element name="StoragePath" type="xs:string"/>
    <xs:element name="LastWriteTime" type="xs:long"/>
    <xs:element name="Size" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```

Path: The path of the file in the original source file system.

StoragePath: The storage path of the file in this spreadsheet data file. This value is the key that is used to match information with information in the virtual directory. This value matches the value of the **Path** element for the same file in the virtual directory.

LastWriteTime: The time that the file was last written. The value is the number of nanoseconds that have elapsed since midnight on January 1, 1601.

Size: The actual size, in bytes, of this file within the storage. The value does not include the end-of-file marker, or CRC marker, if one is used.

2.1.2.3.1.4 WriteEnum

The **WriteEnum** simple type enumerates the allowed values for the name of the **Write** element in the **BackupLogType** type. The values specify the types of enabled access.

```
<xs:simpleType name="WriteEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="ReadWrite"/>
    <xs:enumeration value="ReadOnly"/>
    <xs:enumeration value="ReadOnlyExclusive"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **WriteEnum** type.

Enumeration value	Description
"ReadWrite"	Read-write access.
"ReadOnly"	Read-only access.
"ReadOnlyExclusive"	Read-only exclusive access. This enumeration value is in a different namespace–specifically, http://schemas.microsoft.com/analysisservices/2010/engine/200/200. When the element value is set to this enumeration value, the value of the **valuens** attribute ([MS-SSAS] section 2.2.4.2.1.3) on the element MUST be set to the namespace value.

2.1.2.4 CryptKey.bin File

The CryptKey.bin file contains a cryptographic key. The key is used to encrypt and decrypt the connection strings and password data that are found in the data source tabular model file (section 2.6.2), which is inside the Spreadsheet Data Model file (section 2.1). The CryptKey.bin file that is

stored in the Spreadsheet Data Model file is not the same as the CryptKey.bin file that is independently present on disk (if present at all).

The key inside the CryptKey.bin file is in its original form, even though it is also in the format of a key **binary large object (BLOB)** (section 2.1.2.4.1.1.2) of BLOB type **SIMPLEBLOB** (section 2.1.2.4.1.1.2.1). However, a session key (**SIMPLEBLOB** type) requires two keys to create a key BLOB. To leave the key BLOB in its original form, this situation is handled by using an exponent-of-one private key (section 2.1.2.4.2).

The CryptKey.bin file is composed of a **CryptKeyHeader** structure (section 2.1.2.4.1.1.1), the key BLOB itself (section 2.1.2.4.1.1.2), and a **CryptKeyTrailer** structure (section 2.1.2.4.1.1.3).

2.1.2.4.1 CryptKey.bin File Format

The CryptKey.bin file format consists of a **CryptKeyHeader** structure (section 2.1.2.4.1.1.1) followed by the key BLOB (section 2.1.2.4.1.1.2), which is a key data area containing a **BYTE** array that always contains a **PUBLICKEYSTRUC** BLOB header (section 2.1.2.4.1.1.2.1) and the original key as well as other information as specified by the type of key BLOB. The proper key BLOB can be generated by using an exponent-of-one key and the original key (see section 2.1.2.4.2). This is finally followed by a **CryptKeyTrailer** structure (section 2.1.2.4.1.1.3). These structures MUST be in this order and have no breaks between the areas or structures.

The file format MUST begin with the **CryptKeyHeader** structure, with no bytes preceding the structure. Likewise, no bytes can follow the **CryptKeyTrailer** structure in the file. No padding of any kind exists in the file, except for one case. Padding could exists between the key BLOB data and the beginning of the **CryptKeyTrailer** header. The **m_dwKeyDataSize** member of the **CryptKeyHeader** file MUST accurately account for this possible padding used by the key BLOB.

2.1.2.4.1.1 CryptKey.bin Structures

This section specifies the structures that are required by the CryptKey.bin file format.

2.1.2.4.1.1.1 CryptKeyHeader

The **CryptKeyHeader** structure stores configuration information for the CryptKey.bin file.

```
struct CryptKeyHeader
{
    GUID     m_Magic;
    DWORD    m_dwVersion;
    DWORD    m_dwHeaderSize;
    DWORD    m_dwKeyDataSize;
    DWORD    m_dwTrailerSize;
    DWORD    m_dwProvider;
    DWORD    m_dwAlgorithm;
    DWORD    m_dwFlags;
};
```

m_Magic: A **GUID** that identifies the CryptKey.bin file as a valid file. This value MUST be set to the value in the following table.

Name	Value
CryptKey Magic GUID	{0x5d21bc98, 0x8d2d, 0x4ee6, {0xa8, 0xe5, 0xd0, 0x38, 0xaa, 0xc9, 0x44, 0x41}}

Name	Value
	This value will resolve to the following GUID: 5d21bc98-8d2d-4ee6-a8e5-d038aac94441

m_dwVersion: The encryption key version. The value MUST be set to 0x00000004.

m_dwHeaderSize: The size, in bytes, of the **CryptKeyHeader** structure. The value MUST be set to 44 (hexadecimal 0x0000002C).

m_dwKeyDataSize: The size of the key BLOB, including the **PUBLICKEYSTRUC** structure as well as the key and any other required values.

m_dwTrailerSize: The size, in bytes, of the **CryptKeyTrailer** structure. The value MUST be set to 16 (hexadecimal 0x00000010).

m_dwProvider: The cryptographic provider that is used. Typically, the value is set to 0x00000001. The value MUST be set to one of the values in the following table.

Name	Value
MS_DEF_PROV "Microsoft Base Cryptographic Provider v1.0"	0x00000000
MS_ENHANCED_PROV "Microsoft Enhanced Cryptographic Provider v1.0"	0x00000001

m_dwAlgorithm: The cryptographic algorithm that is used. Typically, the value is set to 0x00000007. The value MUST be set to one of the values in the following table.

Name	Value
CALG_3DES	0x00000000
CALG_3DES	0x00000001
CALG_3DES	0x00000002
CALG_3DES	0x00000003
CALG_3DES	0x00000004
CALG_RC2	0x00000005
CALG_3DES_112	0x00000006
CALG_3DES	0x00000007

m_dwFlags: This value is unused and MUST be set to –1 (hexadecimal 0xFFFFFFFF).

2.1.2.4.1.1.2 Key BLOB

The key BLOB is composed of a **BYTE** array that contains a **PUBLICKEYSTRUC** BLOB header (section 2.1.2.4.1.1.2.1) as well as the encrypted key for the CryptKey.bin file. There also might be other elements of the key BLOB beyond the **PUBLICKEYSTRUC** and the encrypted key. Those elements depend on the type of BLOB.

The key BLOB is correctly generated by using the **CryptExportKey** function ([MSDN-CRYPTO]) with the key to be exported, an exponent-of-one private key as the public key, the *dwFlags* parameter set to zero, and the *dwBlobType* parameter set to **SIMPLEBLOB**. For more information about generating this key BLOB by using an exponent-of-one private key, see section 2.1.2.4.2.

2.1.2.4.1.1.2.1 PUBLICKEYSTRUC

The **PUBLICKEYSTRUC** structure, also known as the **BLOBHEADER** structure, indicates a key's BLOB type and the algorithm that the key uses. The information here pertains only to that needed by the CryptKey.bin file format. For more information about this structure and related information, see [MSDN-CRYPTO].

```
typedef struct _PUBLICKEYSTRUC
{
    BYTE      bType;
    BYTE      bVersion;
    WORD      reserved;
    ALG_ID    aiKeyAlg;
} BLOBHEADER,
PUBLICKEYSTRUC;
```

bType: The key BLOB type. The type that is used within CryptKey.bin is **SIMPLEBLOB**, so this member MUST be set to **SIMPLEBLOB**, as described in the following table.

Value	Meaning
SIMPLEBLOB (hexadecimal 0x1)	The key is a session key.

bVersion: The version number of the key BLOB format. The minimum value for this member is defined by the **CUR_BLOB_VERSION** identifier (which has a value of 2).

reserved: A member that is reserved and MUST be set to zero.

aiKeyAlg: One of the ALG_ID values that identifies the algorithm of the key contained by the key BLOB. The choice of algorithm MUST be the same as that specified in the **CryptKeyHeader** (section 2.1.2.4.1.1.1). However, the values that are used in this member are different than those that are used in the **CryptKeyHeader** and are described in the following table.

ALD_ID identifier	Value	Description
CALG_3DES	0x00006603	Triple DES encryption algorithm. For more information about restraints on the use of this type of key, see [MSDN-CRYPTO].
CALG_3DES_112	0x00006609	Two-key triple DES encryption with the effective key length equal to 112 bits.
CALG_RC2	0x00006602	RC2 block encryption algorithm. For more information about when this key can be used and its provider, see [MSDN-CRYPTO]. For the CryptKey.bin file, this algorithm is limited to an effective key length of 40 bits.

2.1.2.4.1.1.3 CryptKeyTrailer

The **CryptKeyTrailer** structure stores configuration information for the CryptKey.bin file.

```
struct CryptKeyTrailer
{
    GUID    m_Magic;
};
```

m_Magic: A GUID that identifies the CryptKey.bin file as a valid file. This value MUST be set to the value in the following table.

Name	Value
CryptKey Magic GUID	{0x5d21bc98, 0x8d2d, 0x4ee6, {0xa8, 0xe5, 0xd0, 0x38, 0xaa, 0xc9, 0x44, 0x41}} This value will resolve to the following GUID: 5d21bc98-8d2d-4ee6-a8e5-d038aac94441

2.1.2.4.2 Creating an Exponent-of-One Private Key

As the **SIMPLEBLOB** type indicates a session key, both a source (private) key and a destination (public) key are required to create a valid key BLOB for the CryptKey.bin file. In the CryptKey.bin case, doing so is accomplished by using an exponent-of-one private key. This type of key is also called a NULL key because although it is accepted by the **CryptExportKey** function ([MSDN-CRYPTO]), when it is used in that call as the public key, the resulting encryption and decryption do nothing to the private key to be exported. Therefore the private key to be exported is left in its original form.

To create the handle to the exponent-of-one private key, a valid key BLOB of type **SIMPLEBLOB** is required. This key BLOB is created such that the exponent of the key BLOB format is modified to an exponent of one. To obtain the handle of the exponent-of-one private key, the exponent-of-one key BLOB is used, along with the handle to the provider, in a call to the **CryptImportKey** function ([MSDN-CRYPTO]), as shown in the following pseudocode:

```
CALL CryptImportKey with parameters (Handle-To CryptProvider,
    ExponentOfOnePrivateKeyBLOB,
    Size-Of ExponentOfOnePrivateKeyBLOB,
    0,
    0,
    Address-Of handleToExponentOfOnePrivateKey)
```

The handle to the cryptographic provider is obtained through a call to the **CryptAcquireContext** function ([MSDN-CRYPTO]) and MUST use one of the allowed CryptKey.bin providers as the provider string. Furthermore, the provider type MUST be set to **PROV_RSA_FULL**.

The providers that are allowed are listed (as strings) in the following table.

Provider	String name
"Microsoft Base Cryptographic Provider v1.0"	MS_DEF_PROV
"Microsoft Enhanced Cryptographic Provider v1.0"	MS_ENHANCED_PROV

The key BLOB that is required by Cryptkey.bin can now be generated by calling **CryptExportKey** with the handle to the original key (the private key) to be exported, the handle of the exponent-of-one private key, the type set to **SIMPLEBLOB**, the flags value set to zero, a buffer to hold the returned key BLOB, and the length of the key BLOB. The call then returns the properly formatted key BLOB in the buffer parameter. The length of the key BLOB can be determined simply by making the same call to **CryptExportKey**, but with the buffer parameter (*BufferForExportedKeyBLOB*) set to zero.

The following pseudocode illustrates this call to create an exportable key BLOB:

```
CALL CryptExportKey with parameters (Handle-To KeyToBeExported,
    Handle-To ExponentOfOnePrivateKey,
    SIMPLEBLOB,
    0,
    Pointer-To BufferForExportedKeyBLOB,
    Pointer-To LengthOfExportedKeyBLOB)
```

This key BLOB, which is contained in *BufferForExportedKeyBLOB*, is then placed after the **CryptKeyHeader** (section 2.1.2.4.1.1.1) and before the **CryptKeyTrailer** (section 2.1.2.4.1.1.3) in CryptKey.bin.

The method of creating an exponent-of-one private key BLOB has been documented and is widely known. For more information, see [MSKB228786].

However, for convenience, the exponent-of-one private key BLOB is provided in the following table. The handle to this key BLOB is obtained by using **CryptImportKey**, as specified earlier in this section.

Name	Value
ExponentOfOnePrivateKeyBLOB	const BYTE array[] = { 0x07, 0x02, 0x00, 0x00, 0x00, 0xA4, 0x00, 0x00, 0x52, 0x53, 0x41, 0x32, 0x00, 0x02, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0xAB, 0xEF, 0xFA, 0xC6, 0x7D, 0xE8, 0xDE, 0xFB, 0x68, 0x38, 0x09, 0x92, 0xD9, 0x42, 0x7E, 0x6B, 0x89, 0x9E, 0x21, 0xD7, 0x52, 0x1C, 0x99, 0x3C, 0x17, 0x48, 0x4E, 0x3A, 0x44, 0x02, 0xF2, 0xFA, 0x74, 0x57, 0xDA, 0xE4, 0xD3, 0xC0, 0x35, 0x67, 0xFA, 0x6E, 0xDF, 0x78, 0x4C, 0x75, 0x35, 0x1C, 0xA0, 0x74, 0x49, 0xE3, 0x20, 0x13, 0x71, 0x35, 0x65, 0xDF, 0x12, 0x20, 0xF5, 0xF5, 0xF5, 0xC1, 0xED, 0x5C, 0x91, 0x36, 0x75, 0xB0, 0xA9, 0x9C, 0x04, 0xDB, 0x0C, 0x8C, 0xBF, 0x99, 0x75, 0x13, 0x7E, 0x87, 0x80, 0x4B, 0x71, 0x94, 0xB8, 0x00, 0xA0, 0x7D, 0xB7, 0x53, 0xDD, 0x20, 0x63, 0xEE, 0xF7, 0x83, 0x41, 0xFE, 0x16, 0xA7, 0x6E, 0xDF, 0x21, 0x7D, 0x76, 0xC0, 0x85, 0xD5, 0x65, 0x7F, 0x00, 0x23, 0x57, 0x45, 0x52, 0x02, 0x9D, 0xEA, 0x69, 0xAC, 0x1F, 0xFD, 0x3F, 0x8C, 0x4A, 0xD0, 0x01, 0x00, 0x01, 0x00, 0x64, 0xD5, 0xAA, 0xB1, 0xA6, 0x03, 0x18, 0x92, 0x03, 0xAA, 0x31, 0x2E, 0x48, 0x4B, 0x65, 0x20, 0x99, 0xCD, 0xC6, 0x0C, 0x15, 0x0C, 0xBF, 0x3E, 0xFF, 0x78, 0x95, 0x67, 0xB1, 0x74, 0x5B, 0x60, 0x01, 0x00,

Name	Value
	0x00, 0x00 };

2.1.3 Virtual Directory

Following the streamed-in files section (section 2.1.2) and any padding from that section to the Spreadsheet Data Model file page boundary (section 2.1), is the Spreadsheet Data Model virtual directory. The virtual directory lists all the files that are included in the instance of the Spreadsheet Data Model, except for the virtual directory itself.

No byte order mark (section 2.1.1.1) precedes the virtual directory. However, the virtual directory MUST be followed by padding with zeros to end of the page boundary, which is also the end of the Spreadsheet Data Model file.

The virtual directory is an XML document. The document node is the **VirtualDirectory** element.

2.1.3.1 VirtualDirectoryType

The **VirtualDirectoryType** complex type is the type of the **VirtualDirectory** element document node in the virtual directory section of the Spreadsheet Data Model file. **VirtualDirectoryType** contains the list of files.

```
<xs:complexType name="VirtualDirectoryType">
  <xs:sequence>
    <xs:element name="BackupFile" type="VirtualDirectoryBackupFileType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

BackupFile: A complex type containing the information about each file that is included in the Spreadsheet Data Model.

2.1.3.2 VirtualDirectoryBackupFileType

The **VirtualDirectoryBackupFileType** complex type contains properties for each file that is included in the Spreadsheet Data Model instance.

```
<xs:complexType name="VirtualDirectoryBackupFileType">
  <xs:sequence>
    <xs:element name="Path" type="xs:string"/>
    <xs:element name="Size" type="xs:unsignedLong"/>
    <xs:element name="m_cbOffsetHeader" type="xs:unsignedLong"/>
    <xs:element name="Delete" type="xs:boolean"/>
    <xs:element name="CreatedTimestamp" type="xs:long"/>
    <xs:element name="Access" type="xs:long"/>
    <xs:element name="LastWriteTime" type="xs:long"/>
  </xs:sequence>
</xs:complexType>
```

Path: The path of the file within the virtual directory. The value is used as a key to match the file to the backup log, which contains an identical entry for the storage path of a file in the backup log.

Size: The actual size, in bytes, of this file within the storage. The size includes the end-of-file marker, the CRC marker, if one is used.

m_cbOffsetHeader: The offset, in bytes, within the storage to the file. The value includes the header.

Delete: A value that is unused and MUST be ignored.

CreatedTimestamp: The time that the file was created. The value is the number of nanoseconds that have elapsed since midnight on January 1, 1601.

Access: The time that the file was accessed. The value is the number of nanoseconds that have elapsed since midnight on January 1, 1601.

LastWriteTime: The time that the file was last written. The value is the number of nanoseconds that have elapsed since midnight on January 1, 1601.

2.2 File Name Generation

The metadata and data for each Spreadsheet Data Model is represented by a group of files that are automatically generated by the system and that are stored hierarchically in folders. The substrings within the generated names for the files have meaning. The method for the generation of the file names, by concatenation of the substrings that represent various properties of the file, is described in this section. The requirements for which files and folders are mandatory, and under what circumstances, are also described in this section. The file names are defined by using **Augmented Backus-Naur Form (ABNF)** notation, as specified in [RFC5234].

2.2.1 Top-Level Folder

The root folder for the stored files is the database folder.

The database folder name is generated as specified by the following ABNF rules:

```
Integer = *%x30-39
Char = %x41-5A / %x61-7A / %x30-39 / %x23-2E / "!" / "=" / "@" / "[" / "]" /
   "^" / "{" / "}" / "~"
UserID = *Char
DatabaseFolderName = UserID ".0" ".db"
```

UserID is an identifier that is assigned by the user to an object. The characters in **UserID** MAY<3> be normalized.

The database folder contains all of the remaining folders and files that are specified in the structure.

2.2.2 Top-Level Folders

The database folder contains the files and folders that are specified in the following subsections.

2.2.2.1 Cube Folder

Every tabular data model MUST have a cube folder. The cube folder name is generated as specified by the following ABNF rule:

```
CubeFolderName = UserID ".0.cub"
```

2.2.2.1.1 Cube Folder Folders

The cube folder MUST contain a measure group folder for each table that participates in the cube.

2.2.2.1.1.1 Measure Group Folder

Every cube folder MUST contain at least one measure group folder.

The measure group folder name is generated as specified by the following ABNF rules:

```
TableID = UserID
MeasureGroupFolderName = TableID "." Integer "." "det"
```

2.2.2.1.1.1.1 Measure Group Folder Folders

Each measure group folder MUST contain a partition folder.

The partition folder name is generated as specified by the following ABNF rule:

```
PartitionFolderName = TableID "." Integer "." "prt"
```

####### 2.2.2.1.1.1.1.1 Partition Folder Files

A partition information file MUST be generated in every partition folder.

The partition information file name is generated as specified by the following ABNF rule:

```
PartitionInfoFileName ::= "info" "." Integer "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the partition information file is specified in section 2.6.10.1.

2.2.2.1.1.1.2 Measure Group Folder Files

A partition metadata file MUST be generated in every measure group folder.

The partition metadata file name is generated as specified by the following ABNF rule:

```
PartitionFileName =  TableID "." Integer "." "prt" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the partition metadata file is specified in section 2.6.8.

2.2.2.1.2 Cube Folder Files

The following subsections specify name generation for the files that are contained in the cube folder.

2.2.2.1.2.1 Cube Information File

A cube information file MUST be generated in every cube folder.

The cube information file name is generated as specified by the following ABNF rule:

```
CubeInfoFileName = "info" "." Integer "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of cube information file is specified in section 2.6.10.3.

2.2.2.1.2.2 MDX Script Metadata File

A multidimensional expression (MDX) script metadata file MUST be generated in the cube folder.

The MDX script metadata file name is generated as specified by the following ABNF rule:

```
MdxScriptFileName = "MdxScript" "." "0" "." "scr" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the MDX script metadata file is specified in section 2.6.9.

2.2.2.1.2.3 Measure Group Metadata File

A measure group metadata file MUST be generated in the cube folder.

The measure group metadata file name is generated as specified by the following ABNF rule:

```
MeasureGroupFileName = TableID "." "0" "." "det" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the measure group metadata file is specified in section 2.6.7.

2.2.2.2 Data Source Folder

Each model MAY<4> contain a data source folder.

The data source folder name is generated as specified by the following ABNF rules:

```
DataSourceID = UserID
```

```
DataSourceFolderName = DataSourceID "." "0" "." "ds"
```

2.2.2.3 Dimension Folder

A model MUST contain one or more dimension folders. A model MUST have one dimension folder per table in the model instance.

The dimension folder name is generated as specified by the following ABNF rule:

```
DimensionFolderName = TableID "." "0" "." "dim"
```

TableID is the name assigned by the user to a table in an instance of the power pivot model.

2.2.2.3.1 Metadata Files

The following subsections specify file name generation for metadata files that are contained in the dimension folder.

2.2.2.3.1.1 Table Metadata Files

A table metadata file MUST be generated for each table that is part of the model.

The table metadata file name is generated as specified by the following ABNF rule:

```
TableMetadataFileName = TableID "." Integer "." "tbl" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the table metadata file is specified in section 2.5.3.

2.2.2.3.1.2 Table Information File

A table information file MUST be generated for each table that is part of the model.

The table information file name is generated as specified by the following ABNF rule:

```
TableInfoFileName = "info" "." Integer "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the table information file is specified in section 2.6.10.

2.2.2.3.1.3 Table Relationship File

If a table has a defined relationship, a table relationship file MUST be generated.

The table relationship file name is generated as specified by the following ABNF rule:

```
TableRelationshipFileName = "R$" TableID "." Integer "." "tbl" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the table relationship file is described in section 2.5.2.

2.2.2.3.1.4 Column Hierarchy Files

A column hierarchy file MUST be generated for each column in a table.

The column hierarchy file name is generated as specified by the following ABNF rules:

```
ColID = UserID
ColHierFileName = "H$" TableID "$" ColID "." Integer "." "tbl" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model. **TableIDUName** is the name assigned by a user to a column of a table in an instance of the power pivot model.

The content of the column hierarchy file is specified in section 2.5.1.

2.2.2.3.1.5 User Hierarchy Metadata File

If a table has a user-defined hierarchy, a user hierarchy metadata file MUST be generated.

The user hierarchy metadata file name is generated as specified by the following ABNF rules:

```
HierID = UserID
UserHierFileName = "U$" TableID "$" HierID "." Integer "." "tbl" "." "xml"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the user hierarchy metadata file is described in section 2.5.2.

2.2.2.3.2 Data Files

The following subsections specify file name generation for data files that are contained in the dimension folder.

2.2.2.3.2.1 Column Data Files

A data file MUST be generated for each column in a table.

The column data file name is generated as specified by the following ABNF rule:

```
ColDataFileName = Integer "." TableID "." ColID "." "0" "." "idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the column data file is specified in section 2.3.1.

2.2.2.3.2.2 Table Relationship Index File

A table relationship index file MUST be generated for a table if it has a defined relationship to another table in the model.

The table relationship index file name is generated as specified by the following ABNF rule:

```
TableRelationshipFileName = Integer "." "R$" TableID "." "INDEX" "." "0" "." "idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the table relationship index file is specified in section 2.4.

2.2.2.3.2.3 Column Hierarchy Position–to–Identifier File

A column hierarchy position–to–identifier file MUST be generated for each column in a table.

The column hierarchy position–to–identifier file name is generated as specified by the following ABNF rules:

```
ColHierPosToIDFileName = Integer "." "H$" TableName "$" ColName "." "POS_TO_ID" "." "0" "."
"idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the column hierarchy position–to–identifier file is specified in section 2.3.4.

2.2.2.3.2.4 Column Hierarchy Identifier–to–Position File

A column hierarchy identifier–to–position file MUST be generated for a column if a dictionary file is also generated. To determine when the metadata indicates that a dictionary has been generated, see section 2.5.

The column hierarchy identifier–to–position file name is generated as specified by the following ABNF rule:

```
ColHierIDToPosFileName = Integer "." "H$" TableID "$" ColID "." "ID_TO_POS" "." "0" "." "idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the column hierarchy identifier–to–position file is specified in section 2.1.3.5.

2.2.2.3.2.5 Column Hierarchy Hash Table

A column hierarchy hash table file can be generated for a column, depending on the data that is contained in the column. A column hierarchy hash table file MUST be generated if the metadata indicates that a hash data dictionary is used (see sections 2.5.2.20, 2.5.2.21, and 2.5.2.22).

The column hierarchy hash table file name is generated as specified by the following ABNF rule:

```
ColHierHashTableFileName = Integer "." "H$" TableID "$" ColID "." "hidx"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the column hierarchy hash table file is specified in section 2.3.3.

2.2.2.3.2.6 Column Hierarchy Dictionary

A column hierarchy dictionary file can be generated for a column, depending on the data that is contained in the column. A column hierarchy dictionary file MUST be generated if the metadata indicates that a value data dictionary is used (see sections 2.5.2.18 and 2.5.2.19).

The column hierarchy dictionary file name is generated as specified by the following ABNF rule:

```
ColHierDictionaryFileName = Integer "." TableID "." ColID "." "dictionary"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the column hierarchy dictionary file is specified in section 2.3.2.

2.2.2.3.2.7 User Hierarchy Files

User-defined hierarchies comprise an optional feature that can be defined for a table.

2.2.2.3.2.7.1 Child Count File

A user hierarchy child count file MUST be generated if a user-defined hierarchy is present.

The user hierarchy child count file name is generated as specified by the following ABNF rule:

```
UserHierChildCountFileName = Integer "." "U$" TableID "$" HierID "." "CHILD_COUNT" "." "0"
"." "idf"
```

A column hierarchy dictionary file can be generated for a column, depending on the data that is contained in the column. A column hierarchy dictionary file MUST be generated if the metadata indicates that a value data dictionary is used (see sections 2.5.2.18 and 2.5.2.19).

The column hierarchy dictionary file name is generated as specified by the following ABNF rule:

```
ColHierDictionaryFileName = Integer "." TableID "." ColID "." "dictionary"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the user hierarchy child count file is specified in section 2.4.4.1.

2.2.2.3.2.7.2 First Child Position File

A user hierarchy first child position file MUST be generated if a user-defined hierarchy is present.

The user hierarchy first child position file name is generated as specified by the following ABNF rule:

```
UserHierFirstChildPosFileName = Integer "." "U$" TableID "$" HierID "." "FIRST_CHILD_POS" "."
"0" "." "idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the user hierarchy first child position file is specified in section 2.4.4.2.

2.2.2.3.2.7.3 Parent Position File

A user hierarchy parent position file MUST be generated if is a user-defined hierarchy is present.

The user hierarchy parent position file name is generated as specified by the following ABNF rule:

```
UserHierParentPosFileName = Integer "." "U$" TableID "$" HierID "." "PARENT_POS" "." "0" "."
"idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the user hierarchy parent position file is specified in section 2.4.4.4.

2.2.2.3.2.7.4 Multilevel Identifier File

A user hierarchy multilevel identifier file MUST be generated if a user-defined hierarchy is present.

The user hierarchy multilevel identifier file name is generated as specified by the following ABNF rule:

```
UserHierMultiLevelIdFileName = Integer "." "U$" TableID "$" HierID "." "MULTI_LEVEL_ID" "."
"0" "." "idf"
```

Integer is an internal version number that is assigned by the system to each version of this object. This version number does not have to match the version number of other objects in the same model.

The content of the user hierarchy multilevel identifier file is specified in section 2.4.4.3.

2.3 Storage of Data Values

Data is stored in the system by column. Each table is separated into its constituent columns, and a separate set of files is generated to represent the data in each column. Every unique row value in a column is assigned a data identifier. The data identifier values comprise a contiguous range of integers, which can start at any value. Data identifiers are represented as signed 32-bit integers and, as such, are limited to that addressable range.

The data in each column is evaluated heuristically and is either hash encoded or value encoded. String data is always hash encoded. Nonstring data can be either hash encoded or value encoded. Except for the hash dictionary and the value dictionary, all data is represented by data identifier and not by value.

All the column data storage files are stored with compression. Other files might or might not use compression. Different compression methods are used, depending on the file's data. Different file types, or even different files of the same type, have different requirements regarding which compression formats are allowed. For more information about the types of compression that are available for use, see section 2.7.

All the file formats use **little-endian** format.

The file format layouts are platform independent. A file that is written on one supported platform—for example, a 32-bit machine, is readable on a different, supported platform, such as a 64-bit machine, and vice versa.

Each column has an associated file that describes the metadata for the column. The file format for column metadata storage is clear text XML. It is necessary to reference the XML metadata file to understand and decode the contents of the data files for the column. The identification of the proper XML file to use to decode a column data file is explained in section 2.2. The content of the column metadata XML file is explained in section 2.7.

2.3.1 Column Data Storage

A column data storage file for each column in the source data table MUST be present. A separate file is used to store the column data for each column. An example of a generated file name for a column data storage file for a table that has the identifier "Table1" and a column that has the identifier "Cat" is 4.Table1.Cat.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The system represents each unique data value in a column by an assigned data identifier for that unique value. In the tabular data model file format, the data identifier is always stored, but the data value is not stored. To decode the data identifiers into their actual values, the data dictionary (section 2.3.2) or value hash index (section 2.3.3) is used. The system MAY<5> use any method for assigning data identifiers to values.

The column data file contains an array of the data identifier values that represent the values contained in each row of the column in the source data. In this file, one data identifier is represented per row in the source data column. The order in which the data identifiers appear can vary from the order in which they appear in the rows of the source data table. Partial sorting of values MAY<6> be performed to optimize compression.

The column data storage file is compressed by one of several methods, although it is always compressed by using a XMHybridRLE compression method (except for the special case of the **RowNumber** column, which uses XM123 compression as part of a hybrid). For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the column data storage file. An example of a file name for the file that contains the metadata for the data storage file for a table that has the identifier "Table1" is Table1.1.tbl.xml. The metadata file contains the metadata for all the columns of the table. For example, the metadata for the column "Cat" in table "Table1" is found in the **Columns** collection of the **XMSimpleTable** object in the metadata file. In the **Columns** collection, a **Column** item exists for every column in the table. The **Column** item in the **Column** collection for which the value of the **name** attribute is "Cat" contains the metadata for the "Cat" column. For an explanation of how to interpret the metadata file, see section 2.5.

2.3.1.1 File Layout for Column Data Storage Files

All files with the .idf file name extension have the same file format layout. The meaning of the contents of the file depends on the type of file—for example, for a column data storage file, see section 2.3.1.

An .idf file is always compressed. The type of compression can vary. The compression can be XMHybridRLE compression (see section 2.7.3), XMRENoSplit compression (see section 2.7.1), or hybrid compression that uses XM123 compression if the file is a **RowNumber** column (see section 2.7.2 and section 2.7.3.16). An .idf file is also divided into segments. Each segment contains a contiguous slice of rows for the specific column. An .idf file always contains at least one segment and possibly more. Segments are not identical. They can vary in both size (that is, the number of rows in the segment) and the compression method that is used to compress the data in the segment. The XML metadata for each segment specifies the compression method that is used (see section 2.5).

The first 8 bytes of each segment indicates the segment size in units. Each unit is 8 bytes. Therefore, the next segment size multiplied by the unit size is the size of that segment when persisted to disk (in the file). For example, if the first 8 bytes a particular segment is set to 0x02, 2 units exist, which translates into a segment size of 16 bytes. Following the first 8 bytes are 16 more bytes that contain the actual segment data. The first 8 bytes of a segment are not included in the overall segment size value that it holds. It is possible for a segment to be zero in size. In that case, the initial 8 bytes indicates this fact—that is, the 8 bytes are set to zero to indicate a segment size of zero. For an example of the general layout of an .idf file, see section 2.3.1.1.1.

The segment size MUST accurately indicate the size of that segment in the .idf file; otherwise, a file validation error could occur. In particular, segments that use XMHybridRLE compression depend on the segment size to be accurate, and if it is not, data errors that occur when reading the file could cause data corruption or failure errors later.

The segment size refers to the size of the compressed segment when it is persisted to disk. This size differs from the size that a segment can be in memory. Because these in-memory segment sizes matter, they will be discussed here, as well.

The size of an in-memory segment can vary, and its size is measured in rows. All in-memory segments that belong to the same column MUST be of equal size. Additionally, all in-memory segments have a minimum size and a maximum size that are measured in rows. All in-memory segments MUST also have a row count that is a power of two.

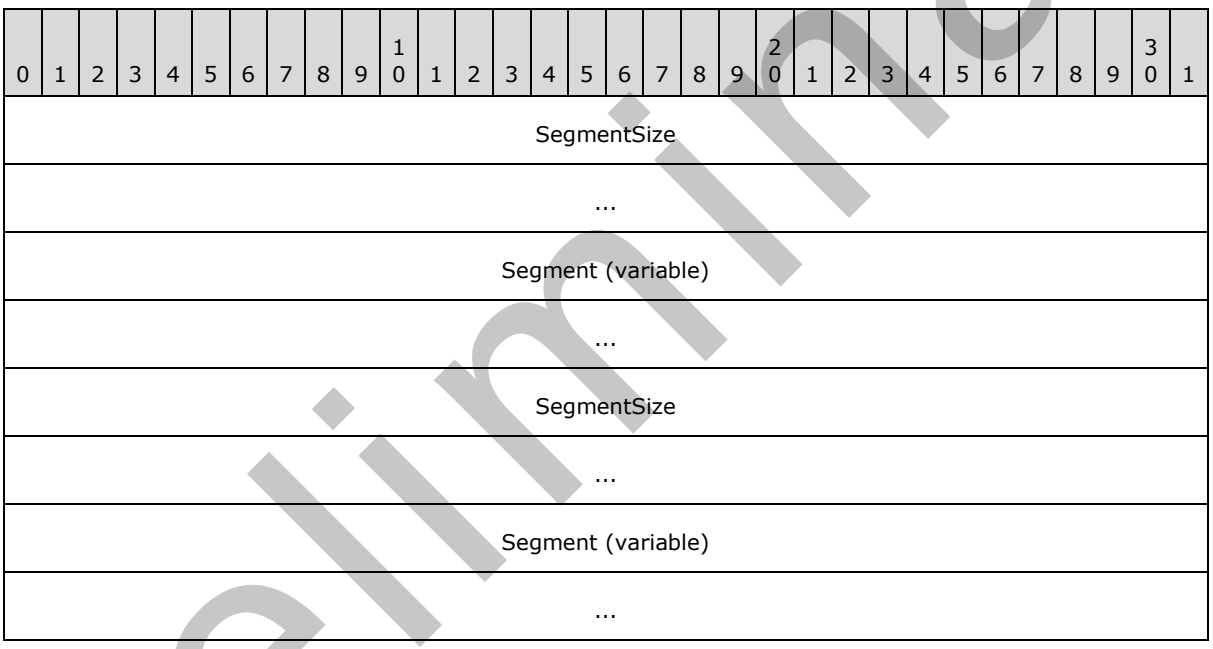
Two exceptions to these rules exist. First, the last segment does not have to comply with these restrictions. Second, segments that are compressed with a hybrid compression MUST treat both the primary segment and the subsegments as one unit, and the unit as a whole MUST meet these restrictions (except for the last primary segment and subsegment in that series of hybrid compressed segments). For more information about the segment size restrictions, including the exact values for the minimum and maximum rows that are required, see section 2.3.1.1.3.

Regarding the column data storage file type, a column data storage .idf file is always compressed by using XMHybridRLE compression. Therefore, a column data storage file has a minimum of two segments. The first segment (the primary segment) is for the RLE part of the compression, and the second segment (the subsegment) is for the XMRENoSplit bit packing part of the compression. However, the layout of the primary segment does not include any size information regarding its subsegment. The subsegment, like its primary segment, follows the basic layout of all segments. This fact means that the first 8 bytes of the subsegment contains the size of the subsegment, just as the first 8 bytes of the primary segment contains only the size information for the primary segment (and thus excludes any size information for the subsegment). For an example of the general layout of an .idf file that uses hybrid compression, see section 2.3.1.1.2. For more information about the hybrid compression of segments, see section 2.7.3.

Any unused trailing bytes within a segment are padded with zeros, but this padding is handled within the compression process. Extra padding with zeros could also exist at the end of the file, and this padding is not accounted for in any segment size value. However, unaccounted-for padding between segments cannot exist. The end-of-file padding is not included in the segment size calculation and is ignored if present.

2.3.1.1.1 General Layout of an .idf File

The following diagram shows the general layout of an .idf file. This layout applies to any .idf file.



SegmentSize (8 bytes): The size of the segment that immediately follows this field.

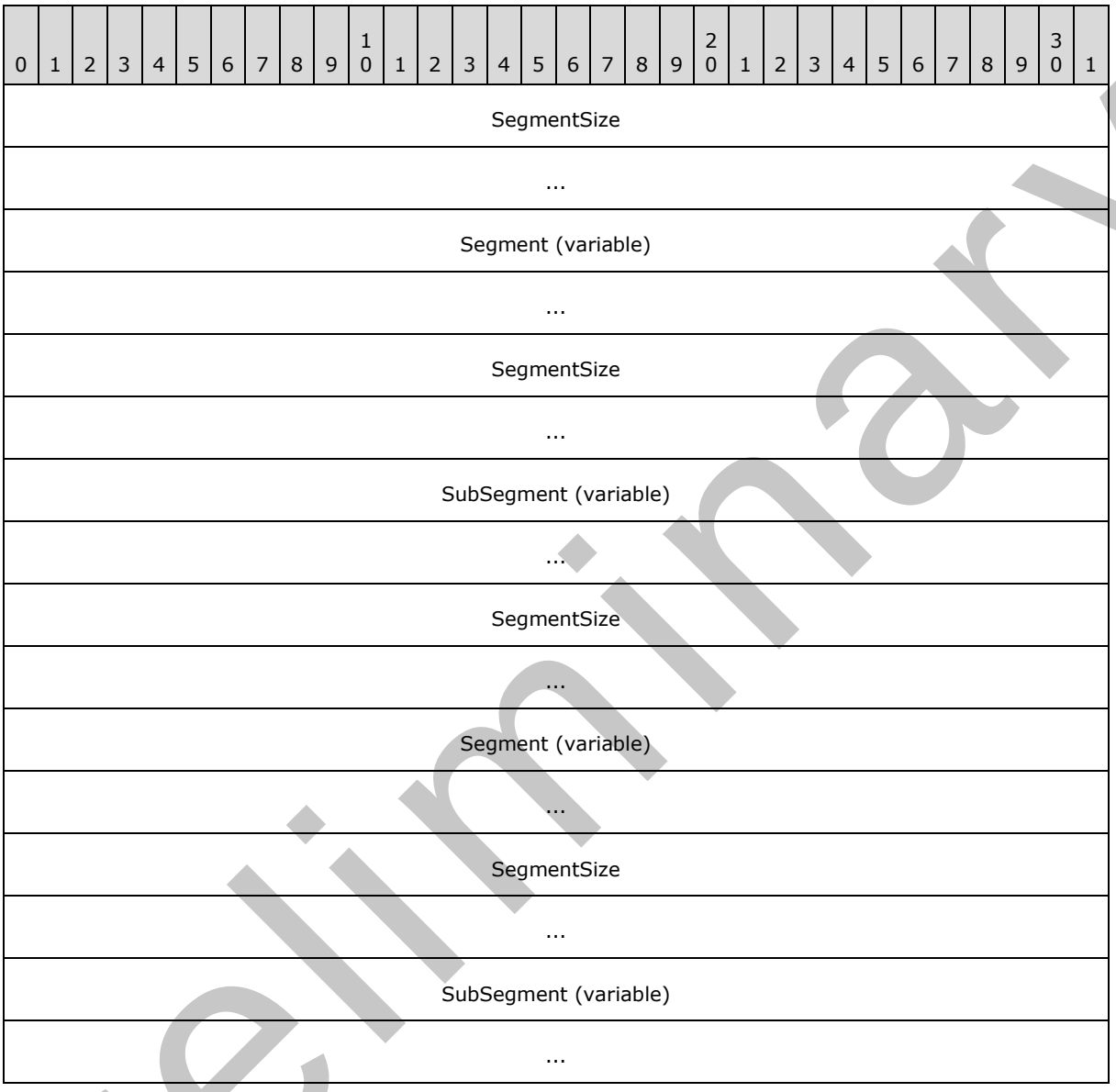
Segment (variable): A segment. The size is specified by the value of the preceding **SegmentSize** field.

SegmentSize (8 bytes): The size of the segment that immediately follows this field.

Segment (variable): A segment. The size is specified by the value of the preceding **SegmentSize** field.

2.3.1.1.2 General Layout of an .idf File That Uses Hybrid Compression

The following diagram provides an example layout of a column data storage .idf file that has two primary segments and two subsegments.



SegmentSize (8 bytes): The size of the primary segment that immediately follows this field.

Segment (variable): A primary segment. The size is specified by the value of the preceding **SegmentSize** field.

SegmentSize (8 bytes): The size of the subsegment that immediately follows this field.

SubSegment (variable): A subsegment that is associated with preceding primary segment. The size is specified by the value of the preceding **SegmentSize** field.

SegmentSize (8 bytes): The size of the primary segment that immediately follows this field.

Segment (variable): A primary segment. The size is specified by the value of the preceding **SegmentSize** field.

SegmentSize (8 bytes): The size of the subsegment that immediately follows this field.

SubSegment (variable): A subsegment that is associated with preceding primary segment. The size is specified by the value of the preceding **SegmentSize** field.

2.3.1.1.3 Segment Size Limitations for .idf Files

Segments in .idf files MUST follow certain size rules. First, all in-memory segments that belong to the same column (that is, the same .idf file when persisted to disk) MUST be of equal size. Second, all in-memory segments have a minimum size of 16,384 rows and a maximum size of 16,777,216 rows. Third, all in-memory segments MUST have a row count that is a power of two.

Only two exceptions to these segment size requirements exist.

First, the last segment of a partition does not need to be within the range of the minimum and maximum row counts, nor does it need to have a row count that is a power of two. The last segment can even be of zero size (the case of the last segment as an empty segment).

Second, when using hybrid compression, both a primary segment and a subsegment that is associated with the primary segment exist. These two segments MUST be considered one unit when applying these rules because the two segments represent data from the same column.

The case of the last segment as an empty segment can occur when an empty table (that is, a table with no rows) exists. The reason is that every column belonging to that table MUST have at least one segment, and every column is required to have a column data storage file (.idf file). Therefore, the first segment is also the last segment and can bypass the restrictions, and therefore be zero (empty). In other words, because the two segments are treated as one unit, both the primary (RLE) segment and the subsegment (bit packing subsegment) are zero (empty).

Note again that this limitation for segments is measured in rows, not in 8-byte units. The reason is that the size of a row is variable, because the particular column might be a column of floating point values, integers, strings, or BLOBs. However, if these row count requirements are adhered to, the compressed segments (which are persisted to the Spreadsheet Data Model file as streamed-in .idf files) will be correct and will not generate any errors or undefined behavior when the file is read.

2.3.2 Column Data Dictionary

Column data can have an associated data dictionary file generated. An example of a generated file name for a dictionary file for a table that has the identifier "Table1" and a column that has the identifier "Label" is 4.Table1.Label.0.dictionary. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The data dictionary contains information that is used to decode the value in the source data that a data identifier represents. The data dictionary file contains an array of unique values that appear in the source data. The file is ordered by data identifier so the first value in the file represents the lowest data identifier, the second value in the file represents the next-lowest data identifier, and so on. Dictionaries that contain only integer or floating point data are not compressed. Dictionaries that contain strings might have parts of the dictionary that are compressed. In the case of dictionaries that contain strings or BLOBs, although most parts of the dictionary file are not compressed, the specific strings or BLOBs might be compressed by using a Huffman compression. BLOBs are pre-

encoded by means of **base64 encoding** and are therefore treated as strings (and stored in a string dictionary). For a discussion of the type of Huffman compression that is used, see section 2.7.

It is necessary to reference the XML metadata file to determine whether a data dictionary file is present for a data column. This metadata is contained in the same file as the column data storage file metadata (see section 2.3.1). If a particular **XMRawColumn** object—specifically, the **XMRawColumn** object of the **Column** item in the **Columns** collection that has a name equal to the column name—has a **DataObject** in the **DataObjects** collection for which the value of the **class** attribute equals one of the values in the following list, the column MUST have a column data dictionary file generated.

- **XMHashDataDictionary<XM_Real>** (section 2.5.2.19)
- **XMHashDataDictionary<XM_Long>** (section 2.5.2.18)
- **XMHashDataDictionary<XM_String>** (section 2.5.2.22)

For an explanation of how to interpret the XML metadata file see section 2.5.

2.3.2.1 File Layout for a Column Data Dictionary

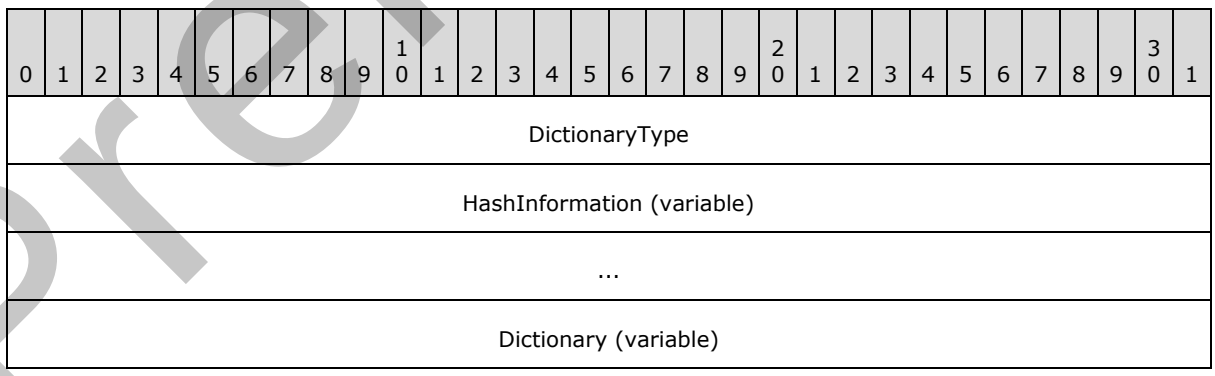
The column data dictionary file format layout varies depending on the type of dictionary that is persisted to the file. Dictionaries can be of type integer, real, or string, the latter of which includes BLOBs because they are precompressed by means of base64 encoding.

The first element in a dictionary file (that is, a file that has the .dictionary file name extension) is an enumeration value. Because the size of an enumeration value depends on the compiler, the number of bytes to be read (or written) is variable. For example, on a 32-bit system or application, a standard enumeration value is 4 bytes, unless it is specifically declared to use some other integer value. A standard enumeration value also defaults to 4 bytes on a 64-bit system or application, unless otherwise specified. The enumeration value used here defaults to 4 bytes.

The **XM_TYPE** enumeration (section 2.3.2.1.3.1) consists of four values, ranging from –1, which implies an invalid type, through the integer, real, and string types.

Depending on the dictionary type, the specifics of the file format layout vary. There are two basic cases: dictionaries of type integer (**XM_TYPE_LONG**) or real (**XM_TYPE_REAL**), and dictionaries of type string (**XM_TYPE_STRING**). For more information about the former, see section 2.3.2.1.1. For more information about the latter, see section 2.3.2.1.2.

The following diagram shows this first element (the dictionary type) in a dictionary file. The type is followed by any hash information (section 2.3.3.1.1), which is then followed by the main part of the dictionary.



...

DictionaryType (4 bytes): The dictionary type. The value can be **XM_TYPE_LONG**, **XM_TYPE_REAL**, **XM_TYPE_STRING**, or **XM_TYPE_INVALID**.

HashInformation (variable): The hash information that is required for all dictionaries. An **XM_TYPE_STRING** dictionary has the option of not including any hash information in certain situations. For more information, see section 2.3.2.1.2.2.

Dictionary (variable): The dictionary store. If the value of **DictionaryType** is **XM_TYPE_STRING**, this store consists of information, pages, and records. If the value of **DictionaryType** is **XM_TYPE_REAL** or **XM_TYPE_LONG**, this store consists of information plus numeric items and values.

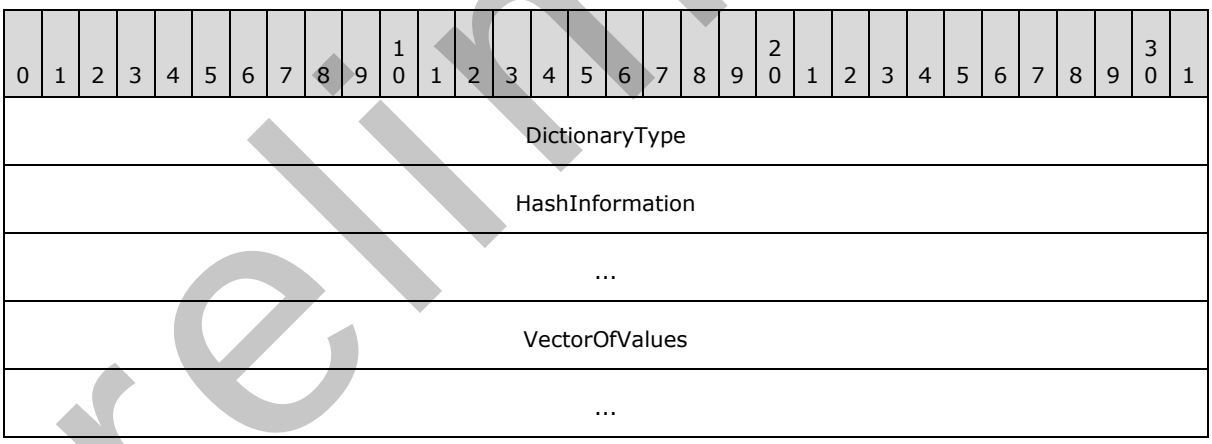
2.3.2.1.1 XM_TYPE_LONG and XM_TYPE_REAL Data Dictionary Files

For a dictionary of type **XM_TYPE_LONG**, if the **OperatingOn32** element (section 2.5.2.21.1) is set to **true**, the dictionary will use 4-byte integers. If this element is set to **false**, the dictionary will use 8-byte integers. This change has an effect on how values, potentially including any hash information, are stored and interpreted in the file.

Note that the sizes of the first required five elements of the hash (see section 2.3.2.1.1.1) are not affected by this information. The values that are contained by the hash bin and hash entry structure size elements (that is, two of the five elements) are affected, but the sizes are not.

The values that are contained by each of the required hash elements MUST be correct; otherwise, a file validation error could occur. For more information about the **XM_TYPE_LONG** and **XM_TYPE_REAL** hash data dictionary XML metadata values, see section 2.5.2.21 and section 2.5.2.20.

The following diagram shows the general layout of an **XM_TYPE_REAL** or **XM_TYPE_LONG** dictionary.



DictionaryType (4 bytes): The type of dictionary. The value MUST be **XM_TYPE_REAL** or **XM_TYPE_LONG**.

HashInformation (variable): The required hash elements for dictionary files.

VectorOfValues (variable): The set of real (64-bit) or integer (32-bit or 64-bit, depending on the range of values that are encountered) values. The values are not compressed.

2.3.2.1.1.1 Required Hash Elements

For integer and real dictionaries, the next five elements are hash elements (see section 2.3.3.1.1). For all dictionary files, the value of **cBins** MUST be set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** (section 2.3.3.1.4.1) to indicate that no further hash information is included.

The underlying system does not use any included hash information, other than the five required elements, in a dictionary. Therefore, there MUST NOT be any other hash information (other than those five elements) in a dictionary file, regardless of the type of dictionary.

2.3.2.1.1.2 Vector of Values

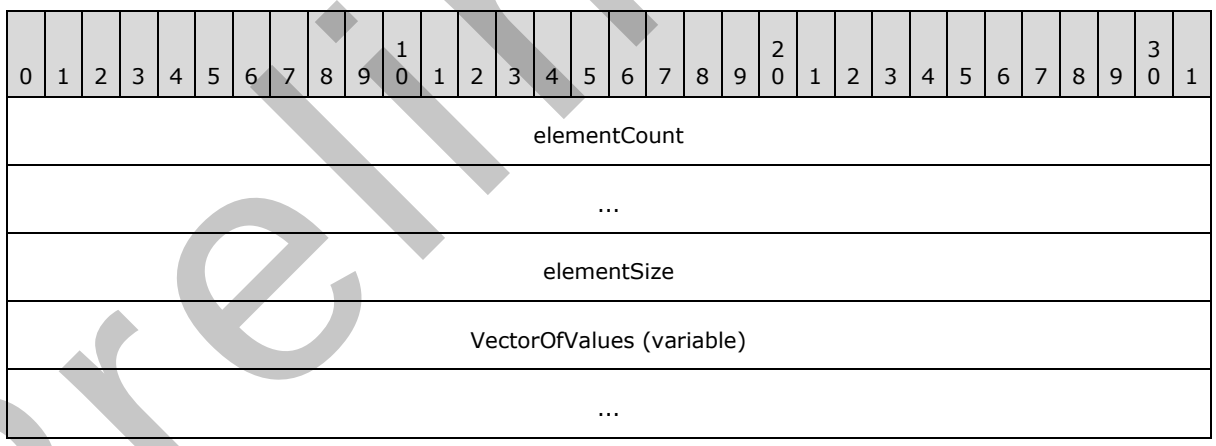
For **XM_TYPE_LONG** or **XM_TYPE_REAL** dictionaries, what now follows is a vector of either integer or double values. These values are the actual dictionary items, which are stored in a vector (or array), and are not compressed.

The number of dictionary items and the individual sizes of the dictionary items are encoded first. Therefore, at this point in the file, the next 8 bytes represent the number of elements in the vector. The following 4 bytes represent the size, in bytes, of each element in the vector. Therefore, the vector itself is of variable size—this size, in bytes, equals the number of elements multiplied by the element size.

The element size for a **XM_TYPE_LONG** dictionary also varies depending on whether the operating system is 32-bit or 64-bit because the element size reflects the size of a 32-bit integer or a 64-bit integer. For an **XM_TYPE_REAL** dictionary, the element size is the size of a double value.

The vector of values completes the format of a hash data dictionary of type **XM_TYPE_LONG** or **XM_TYPE_REAL**. Padding with zeros might exist at the end of the file, but such padding is ignored and not read.

The following diagram shows a general view of the layout of the vector of values.



elementCount (8 bytes): The number of elements in the **XM_TYPE_REAL** or **XM_TYPE_LONG** dictionary.

elementSize (4 bytes): The size of each element.

VectorOfValues (variable): The vector of real or integer values in the dictionary.

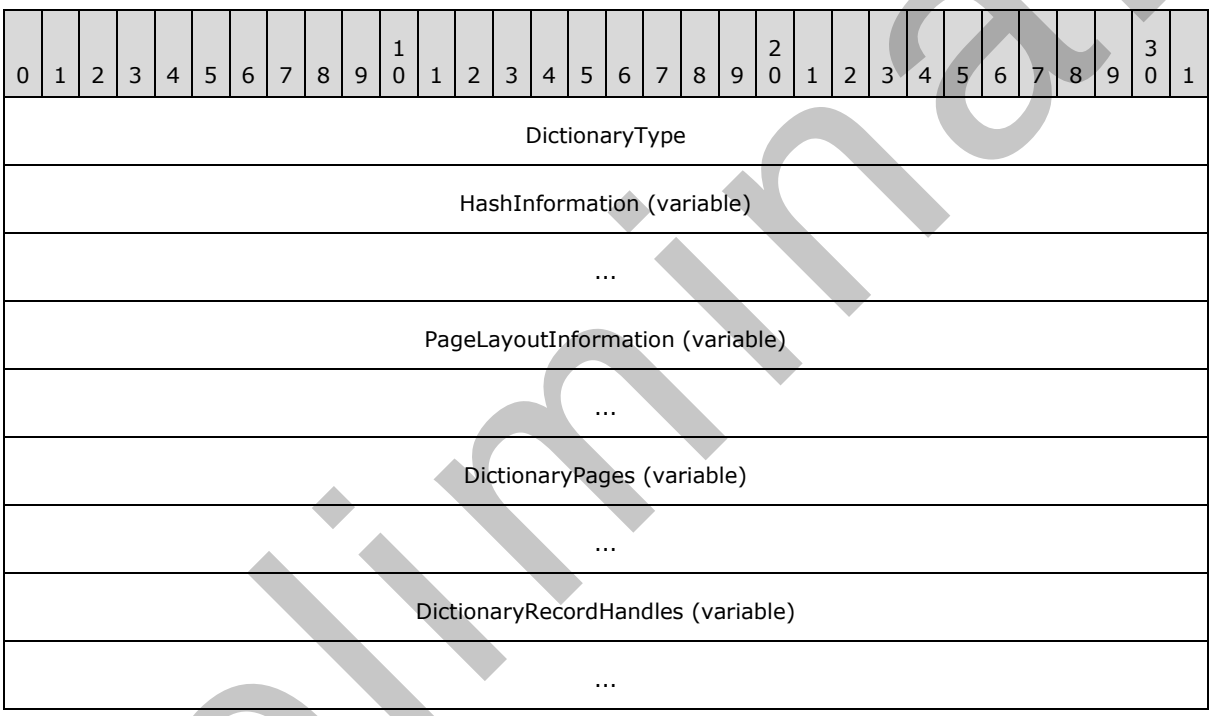
2.3.2.1.2 XM_TYPE_STRING Data Dictionary Files

The layout for dictionaries of type **XM_TYPE_STRING** is different than that for dictionaries of type **XM_TYPE_LONG** and **XM_TYPE_REAL**. After the **XM_TYPE** information is read (see section 2.3.2.1), an XML metadata flag is checked to determine whether any hash information is included in the file.

For dictionaries of type **XM_TYPE_LONG** or **XM_TYPE_REAL**, this hash information is always present, but for dictionaries of type **XM_TYPE_STRING**, the information is present only if the flag value 0x01 is set in the **DictionaryFlags** element (section 2.5.2.22.1) in the metadata for the dictionary.

For general information about **XM_TYPE_STRING** hash data dictionary metadata, see section 2.5.2.22.

The following diagram shows the general layout of the **XM_TYPE_STRING** dictionary.



DictionaryType (4 bytes): The type of the dictionary. The value MUST be **XM_TYPE_STRING**.

HashInformation (variable): The required hash elements.

PageLayoutInformation (variable): The information that pertains to the whole dictionary, excluding the hash information and the dictionary type that exist in the preceding fields. This field contains information such as whether compression is used (on at least one page), the string count, and the number of pages.

DictionaryPages (variable): One or more sets of information, each of which pertains to a single page. A set of information includes the string store.

DictionaryRecordHandles (variable): A vector of record handle structures, one per string.

2.3.2.1.1.2.1 BLOBs and Base64 Encoding

BLOBs are supported and stored in the **XM_TYPE_STRING** hash data dictionary format. The reason is that BLOBs are treated in the same manner as strings because they have already been encoded by using base64 encoding before storage into a dictionary file.

BLOBs stored in Spreadsheet Data Model files MUST be encoded by using base64 encoding prior to any other compression and storage. For information on the Spreadsheet Data Model file format, see section 2.1.

If BLOBs are being stored in an **XM_TYPE_STRING** hash data dictionary, the **XM_STRDICT_BLOB_STORAGE_PAGE** flag will be set in the **DictionaryFlags** element in the metadata for the dictionary (see section 2.5.2.22.1). This **XM_STRDICT_BLOB_STORAGE_PAGE** flag MUST be set if BLOBs are being stored in the dictionary.

Because they are strings (with only 64 character symbols used), BLOBs can also be compressed by using Huffman compression. So if compression is used on the string store, both the strings and the BLOBs will be compressed by using Huffman compression if they fall within all of the Huffman compression constraints.

For more information about **XM_TYPE_STRING** hash data dictionary metadata, including the dictionary flags that need to be set, see section 2.5.2.22. For more information about the Huffman compression that is used, see section 2.7.4.

2.3.2.1.1.2.2 Required Hash Elements

If the flag value 0x01 (section 2.5.2.22.1) is set, the file contains the five required hash elements, and the system will rebuild the hash table for the dictionary at run time—thus allowing fast lookups into the string dictionary, even if the dictionary has duplicate strings.

If this flag is not set, no hash information is contained in the file. This behavior is different than that for the **XM_TYPE_LONG** and **XM_TYPE_REAL** dictionaries (section 2.3.2.1.1.1).

If **XM_STRDICT_OPTION_ALLOW_LOOKUP** is not set, and therefore the five required hash elements are not included, no lookups will be allowed in the dictionary. Thus, it will be assumed that the dictionary contains only unique strings (so that no collisions will occur by having two strings that are identical but now cannot be correctly identified without hash information).

However, when the flag is set, only the first five (required) hash elements are present, and **cBins** MUST be set to the **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** value (section 2.3.3.1.4.1) to indicate that no further hash information is included.

The underlying system does not use any included hash information, other than the five required elements, in a dictionary. Therefore, there MUST NOT be any other hash information (other than those five elements) in a dictionary file, regardless of the type of dictionary

For information about the required hash elements, see section 2.3.3.1.1. For how this information is treated by the **XM_TYPE_LONG** and **XM_TYPE_REAL** hash data dictionaries, see section 2.3.2.1.1.1.

2.3.2.1.1.2.3 Dictionary Page Layout

XM_TYPE_STRING dictionaries (also referred to as *string dictionaries*) use a page system. This system is similar to the one in which column data storage files (.idf files) use segments (see section

2.3.2.1). However, the two systems are not identical. Dictionary pages ought not to be confused with operating-system pages.

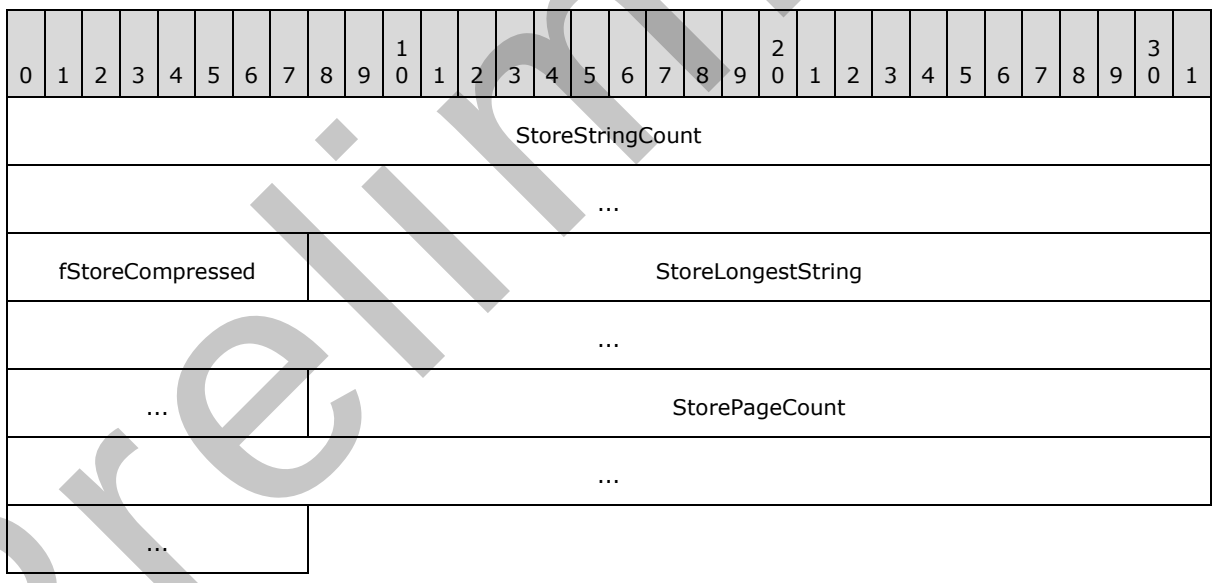
An **XM_TYPE_STRING** dictionary divides the strings into pages. Each page can be of variable size within certain limitations. For more information about dictionary page sizes and the page count limit, see section 2.3.2.1.3.2.

Each page can be compressed or not, independent of any other page. The compression that is used for string dictionary pages is a variation of Huffman compression, which is a widely known compression algorithm that is often used for compressing strings. The Huffman compression procedure that is used depends on whether the string is considered to originate from a single character set (such as ASCII-US or Unicode using just the ASCII-US set) or from multiple character sets. Not all character sets, even though they are Unicode, can be compressed via this Huffman implementation. For more information about the Huffman compression algorithm that is implemented, see section 2.7.4. For particular information regarding the implications of character set choice, see section 2.7.4.1.4.

Following any hash information in the file (see section 2.3.2.1.2.2), the subsequent fields consist of general information regarding the entire dictionary string store. Information that is specific to each page within the dictionary is discussed separately (section 2.3.2.1.2.4).

The next 8 bytes indicate the number of strings in the dictionary store. This value applies to the entire string store in the dictionary, so it includes all the pages in the string store. The following byte is a Boolean flag that indicates whether the store has compressed pages. If this value is set to **true**, at least one page in the dictionary's string store is compressed, but it does not necessarily mean that all the pages are compressed. The next 8 bytes indicate the length, in characters, of the longest string in the entire store. The final 8 bytes indicate the total number of pages in the dictionary's string store.

The following diagram shows the layout of the elements just discussed, beginning with the number of strings in the dictionary string store and ending with the total page count for the string store.



StoreStringCount (8 bytes): The number of strings in the entire dictionary store (that is, in all the pages).

fStoreCompressed (1 byte): Boolean. A flag that indicates whether at least one page in the store is compressed. If the value is **true**, at least one page in the store is compressed.

StoreLongestString (8 bytes): The longest string, in number of characters, in the entire store.

StorePageCount (8 bytes): The number of pages in the entire dictionary store.

2.3.2.1.2.4 Dictionary String Store (Per Page) Information

Each page has the same format, beginning with general information for the page and then including the actual stored strings, either compressed or uncompressed depending on whether the page is marked as compressed.

Therefore, for each page, the first 8 bytes contain the mask state information for the page. This mask indicates whether the page is compressed by using Huffman compression. (In contrast, the **fStoreCompressed** flag (section 2.3.2.1.2.3) simply indicates whether at least one page is compressed.) For more information about the values that this mask can hold, see section 2.3.2.1.3.3.

The mask state information is followed by a single byte containing a Boolean flag that indicates whether the page contains NULL values. The next 8 bytes contain the starting index that is used for locating the first record handle structure for the strings on this page. This index is zero-based because it refers to the vector of record handle structures for the dictionary. For more information about the **XM_TYPE_STRING** dictionary vector of record handles, see section 2.3.2.1.2.5.

Therefore, a starting index of zero refers to the first element in the record handle vector, and that indexed record handle structure is the first record handle structure for the page. As another example, a starting index of 1045 implies that index 1045 of the record handle vector contains the first record handle of the page.

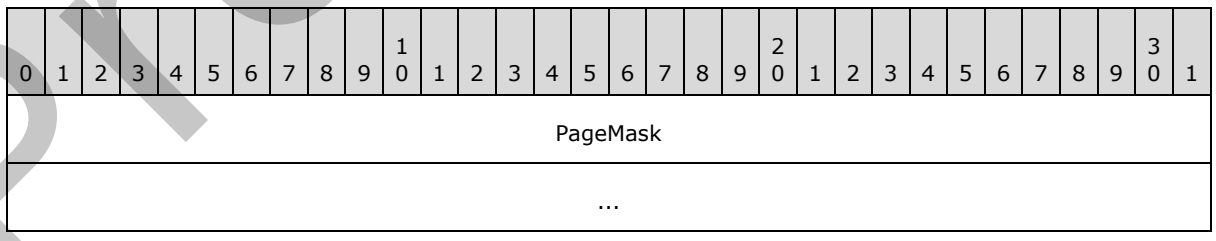
The next 8 bytes indicate the number of strings that are contained on this particular page. This number is followed by another single byte containing a Boolean flag that indicates whether this particular page is compressed. Both the mask state information and this Boolean flag MUST reflect the same state (compressed or not compressed).

The final 4 bytes contain a special mark that indicates the beginning of the page's string store. This mark MUST be set to the unsigned integer value 0xAABBCCDD (in decimal, 2,864,434,397).

After all the strings, another special mark indicates the end of the page's string store and the beginning of the next page (see section 2.3.2.1.2.4.3).

For more information about the strings stored on uncompressed pages, see section 2.3.2.1.2.4.1. For more information about the strings stored on compressed pages, see section 2.3.2.1.2.4.2. For more information about the second string store page marker, see section 2.3.2.1.2.4.3. For more information about the record handles vector information that is stored at the end of a dictionary file, see section 2.3.2.1.2.5.

The following diagram shows a general layout of the page components just discussed.



PageContainsNULLs	PageStartIndex	
...		
...	PageStringCount	
...		
...	PageCompressed	StringStoreBeginMark
...		StringStore (variable)
...		
StringStoreEndMark		

PageMask (8 bytes): The mask state information for this page.

PageContainsNULLs (1 byte): A Boolean flag that indicates whether the page contains NULL values. If the value is **true**, the page contains NULL values.

PageStartIndex (8 bytes): The start index of this page.

PageStringCount (8 bytes): The number of strings on this page.

PageCompressed (1 byte): A Boolean flag that indicates whether the page is compressed. If the value is **true**, the page is compressed.

StringStoreBeginMark (4 bytes): A special mark that indicates the beginning of the actual strings in the store for this page.

StringStore (variable): The string store (that is, the actual strings in the store).

StringStoreEndMark (4 bytes): The special mark that indicates the end of the string store for this page.

2.3.2.1.2.4.1 Uncompressed Page Case

When the page is not compressed, the next 8 bytes after the general per-page information (section 2.3.2.1.2.4) indicate the number of characters that can still be stored on the page. In other words, these 8 bytes indicate the remaining amount of room, in characters, that are available in the store. This value is based on the page size, the current number of characters in the store, and other page information that is taking up room on the page.

These characters are of type **_TCHAR** (section 2.1.1.2). The bytes are treated simply as a stream of characters of type **_TCHAR**, without any knowledge of the actual bytes used per character symbol or of the character set that is used. Internally, the system uses Unicode.

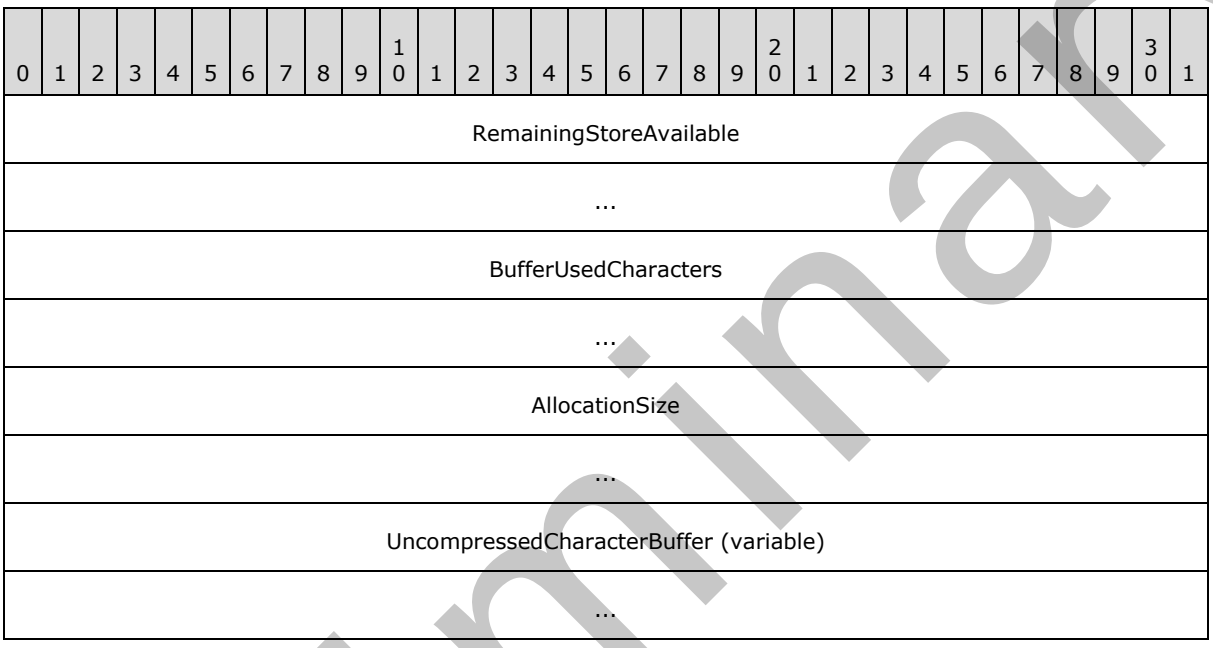
The next 8 bytes contain the number of used characters in the stored character buffer. This value represents the number of characters in the string store buffer that have already been stored. It also represents the next free offset that can be written to. The value MUST be accurate, and the implied size (number of characters multiplied by the size of one character) MUST NOT exceed the actual. allocated buffer size.

The next 8 bytes indicate the allocation size, in bytes, that is needed to hold the buffer of string character data. This value is followed by the set of uncompressed strings as one long byte buffer. The number of bytes in this byte buffer is defined by the allocation size. Each string in this character buffer MUST be terminated by the null character ('\0'). All size calculations need to take this termination character requirement into account.

After this buffer, the second mark (see section 2.3.2.1.2.4.3) is read. Then, a new page, if present, begins. This next page can again be either compressed or uncompressed.

Following the page-specific information for every page, a vector of record handles completes the dictionary file (see section 2.3.2.1.2.5).

The following diagram shows the layout of the uncompressed page elements just discussed.



RemainingStoreAvailable (8 bytes): The remaining number of characters that can be written to the character buffer.

BufferUsedCharacters (8 bytes): The number of characters that already exist in the character buffer. This value also indicates the beginning offset where additional characters can be written.

AllocationSize (8 bytes): The size of the character buffer.

UncompressedCharacterBuffer (variable): The character buffer. The size of this buffer is specified by **AllocationSize**. The buffer contains the uncompressed strings that are stored on this page. Each string in this character buffer MUST be terminated by the null character ('\0').

2.3.2.1.2.4.2 Compressed Page Case

If the page is compressed, the format is more complicated than that of the uncompressed page case. The first 4 bytes contain the total number of bits in the store—that is, the bit offset of the end of the last compressed string in the store. This value is an unsigned integer value. This value is needed to determine the bits of the last string in the store, because the compressed strings are

referenced by their bit offsets into the store. For all but the last string, subtracting the next string's offset from the current string's offset provides the current string's bit length.

The next 4 bytes identify the type of character set—in other words, the character set mode that is used. Strings can be of a single or of a multiple character set. If the character set is single, and if the single character set uses only two bytes per character (because of those two bytes, one will be the same for every character in the string), the Huffman encoding process can strip off the identifying character set information and store the one byte that identifies the character set separately from the actual stripped characters.

This procedure cannot be performed in the multiple character set case nor typically in the single character case when multibyte characters are used (in which more than one byte is used per actual character, not counting the character set identifier byte). In such situations, all the bytes MUST be stored and then used in the encoding and decoding process. Hence, the Huffman process needs to account for this difference.

The character set type identifier is an unsigned integer. For the values that identify the character set mode that is used for the page's string store, see section 2.3.2.1.3.4. For more information about the Huffman compression algorithm that is used, see section 2.7.4.

The next 8 bytes contain the allocation size, in bytes, of the actual compressed strings in the store.

If the character set mode is single, the next byte, which contains an unsigned value, indicates the character set that is used. Essentially, this byte would have been the upper byte of each character in the single character set string. If the character set mode is multiple, this byte MUST NOT be present in the file layout.

The next 4 bytes (**uiDecodeBits**) contain the maximum number of bits that are used to create a fast lookup table for Huffman decoding. This number is also referred to as the codeword length. The value is an unsigned integer. The valid range for **uiDecodeBits** is from 2 through 12. Although Huffman encoding supports codewords of 2 through 15 bits, the lookup table supports only 2 through 12 bits, and any codewords that use lengths greater than 12 (or possibly less in some cases) will be decoded through a Huffman tree traversal (rather than through the faster lookup table). This value does not reflect the actual size of the codewords that are used if the longest codeword is greater than 12, but this value MUST NOT be set any higher than 12. For more information about how this value is set and what its effects are, see section 2.7.4.1.2.

The next 128 bytes (**encodeArray**), which are read as a stream of unsigned 8-bit integer values, contain the encoded Huffman alphabet as an array. The system uses a Huffman alphabet of 256 characters, which is stored in an unsigned 8-bit integer array. This array contains the codeword length for each element in the alphabet.

The next 8 bytes (**ui64BufferSize**) contain an unsigned integer that indicates the size, in 8-bit integer units, of the buffer of the compressed strings. This value is expected to be the same as that of **AllocationSize** (section 2.3.2.1.2.4.1). Because the buffer consists of a stream of bytes, and the size of a byte is the same as the size of an unsigned 8-bit integer), this value is the same as the number of bytes that are needed to create that buffer.

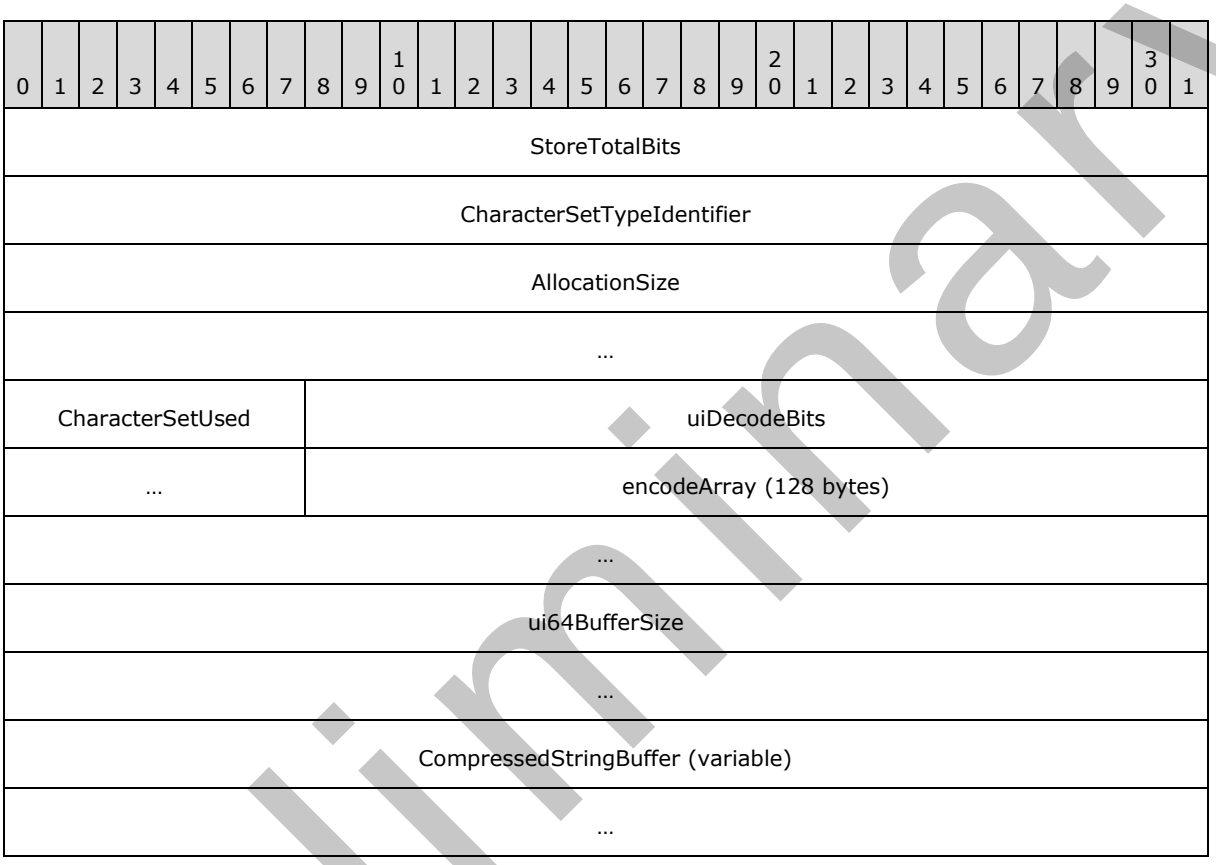
The next set of bytes contains the set of compressed strings for the compressed page. The number of bytes equal the value of **ui64BufferSize**. These strings are compressed by using a constrained version of classic Huffman compression, which requires the encoded array to correctly decode the strings. Therefore, the information here MUST NOT be altered between the writing of the file by the system and the next reading of the file; otherwise, the decompression of the strings might fail.

For more information about the Huffman compression algorithm that is used, the variables here that are related to the Huffman compression, and how to encode and decode by using this constrained Huffman implementation, see section 2.7.4.

Following the buffer, the second mark (see section 2.3.2.1.2.4.3) is read. Then, a new page, if present, begins. Again, the next page (if present) can be either compressed or uncompressed.

Following the page-specific information for every page, a vector of record handles completes the dictionary file (see section 2.3.2.1.2.5).

The following diagram shows the layout of the compressed page elements just discussed.



StoreTotalBits (4 bytes): The total number of bits in the store.

CharacterSetTypeIdentifier (4 bytes): A value that identifies whether the character set mode is single or multiple.

AllocationSize (8 bytes): The allocation size that is needed for the string store (that is, for the buffer).

CharacterSetUsed (1 byte): An identifier for the character set that is used. This value applies only to the single character set mode. This value is not present if the character set mode is multiple.

uiDecodeBits (4 bytes): The number of bits that are used in the lookup table for Huffman decoding.

encodeArray (128 bytes): The encoded Huffman alphabet array for decoding.

Ui64BufferSize (8 bytes): The size of the character buffer. This size is expected to be the same as the allocation size.

CompressedStringBuffer (variable): The buffer of compressed strings for this page.

2.3.2.1.2.4.3 Second Mark (End of Page Marker)

At the end of each page, 4 bytes contain a special mark that indicates the end of that page's string store. This mark MUST be set to the unsigned integer value 0xABCDABCD (in decimal, 2,882,382,797). For the general layout of a dictionary page (whether compressed or uncompressed), see section 2.3.2.1.2.4.

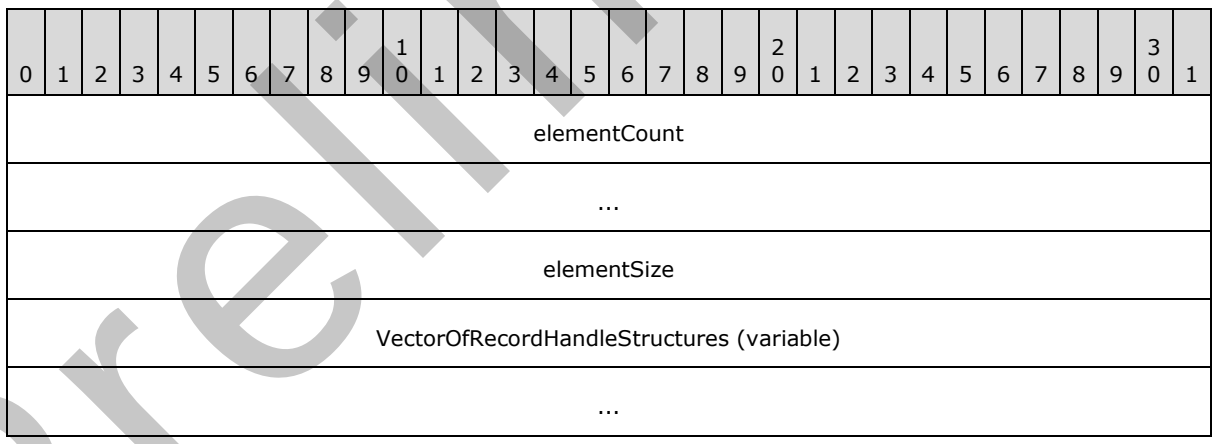
2.3.2.1.2.5 Dictionary Record Handles Vector

What follows the pages and their information in the string dictionary is a vector of record handle structures. Record handles have a one-to-one relationship with the strings in all the string stores. That is, each string has a corresponding record handle, and each record handle has a corresponding string. This correspondence exists regardless of whether the string is compressed. Each record handle contains a bit or a byte offset and a page identifier that indicates which page holds the string that is associated with that record handle. For more information, please section 2.3.2.1.3.5.

For decoding the vector, the next 8 bytes (**elementCount**) in the file represent the number of elements in the vector (or array). This value is followed by 4 bytes (**elementSize**) that indicate the size of each element in the vector. Each element in the vector is a record handle structure and. as such, has the size of a record handle structure (see section 2.3.2.1.3.5). Therefore, the vector of record handles is of variable size—specifically, **elementCount** multiplied by **elementSize**.

The vector of record handles is the last item in a hash data dictionary file. Extra padding with zeros might exist at the end of the file. If present, this information is ignored and not read by the system.

The following diagram shows a general view of the record handle elements just discussed, beginning with the number of records (**elementCount**) and ending with the vector of elements. It does not show the internal details of the record handle structure.



elementCount (8 bytes): The number of elements in the record handle vector.

elementSize (4 bytes): The size, in bytes, of each element (that is, of one record handle structure).

VectorOfRecordHandleStructures (variable): The vector of record handle structures.

2.3.2.1.3 Dictionary Structures, Enumerations, and Constants

This section contains information that is related to the data structures, enumerations, and other constants that are used by the hash data dictionary file format.

2.3.2.1.3.1 XM_TYPE Enumeration

The **XM_TYPE** enumeration is used to identify the type of hash data dictionary that is stored in the dictionary file. (Dictionary files have the .dictionary file name extension.)

```
enum XM_TYPE
{
    XM_TYPE_INVALID    =    -1,
    XM_TYPE_LONG       =     0,
    XM_TYPE_REAL       =     1,
    XM_TYPE_STRING     =     2
};
```

The following table describes the available enumeration values.

Enumeration value	Meaning
XM_TYPE_INVALID	The data type is invalid.
XM_TYPE_LONG	The dictionary holds integers.
XM_TYPE_REAL	The dictionary holds real (floating point) values.
XM_TYPE_STRING	The dictionary holds strings. The strings might or might not be compressed and are stored per page in the dictionary.

2.3.2.1.3.2 Page Size Limitations for an XM_TYPE_STRING Hash Data Dictionary

Each page that is persisted to disk within an **XM_TYPE_STRING** dictionary file is of variable size, up to a page size limit of 4,294,967,296 bytes if uncompressed or 536,870,912 bytes if compressed. N minimum page size exists.

The maximum number of pages in a dictionary file is 524,288.

2.3.2.1.3.3 Page Mask for an XM_TYPE_STRING Hash Data Dictionary

The **XM_TYPE_STRING** page mask contains compression information for the page.

The mask information is on a per-page basis and indicates whether the strings on the page have been compressed by using Huffman compression. The default mask value indicates that the string store for the page is not compressed.

The following table contains the mask values.

Name	Value
XM_STRING_STORE_PAGE_OPTION_DEFAULT	0x000

Name	Value
XM_STRING_STORE_PAGE_OPTION_COMPRESSED	0x001

2.3.2.1.3.4 Huffman Character Set Mode

The character set mode indicates the character set characteristics of the dictionary strings on that dictionary page.

During the Huffman compression of strings, one of two possible modes can be used: a single character set mode or a multiple character set mode. The former means that only one character set is used for the strings being compressed for that page in the dictionary. The latter means that more than one character set is being used for the strings being compressed for that page in the dictionary.

Single character set and multiple character set modes can also be used for other situations, such as the use of multibyte character sets and the Huffman compression of BLOBs that use base64 encoding. For more information, see section 2.7.4.

The following table lists the character set mode values for Huffman compression.

Name	Value
XM_HUFFMAN_SINGLECHARSET	703121
XM_HUFFMAN_MULTICHARSET	703122

2.3.2.1.3.5 Record Handle Structures for an XM_TYPE_STRING Hash Data Dictionary

The record handle structure contains bit or byte offset information and page identifying information for a particular string in the dictionary.

The record handle has a one-to-one correspondence with a particular, unique string in an **XM_TYPE_STRING** hash data dictionary string store. One record handle exists for each string in a dictionary file.

The record handle structure that is used depends on whether the string is compressed. For compressed strings, a bit offset is used. For uncompressed strings, a byte offset is used. The size of the structure member is identical for both the bit and the byte offset. The page identifier member is identical in meaning and size in both structures.

The **XM_CompressedStringRecordHandle** structure stores the bit offset and page identifier for a compressed string that exists in the vector of record handle structures used by the **XM_TYPE_STRING** hash data dictionary.

```
struct XM_CompressedStringRecordHandle
{
    unsigned __int32    bitOffset;
    unsigned __int32    pageID;
};
```

bitOffset: The compressed string bit offset, starting with zero for the first string of each page. A bit offset is relative to its page, not to the entire dictionary.

pageID: The page identifier, which is a zero-based index (beginning at page zero and continuing through the total number of pages for the dictionary minus one).

The **XM_ConstantStringRecordHandle** stores the byte offset and page identifier for an uncompressed string that exists in the vector of record handle structures used by the **XM_TYPE_STRING** hash data dictionary.

```
struct XM_ConstantStringRecordHandle
{
    unsigned __int32   byteOffset;
    unsigned __int32   pageID;
};
```

byteOffset: The uncompressed string byte offset, starting with zero for the first string of each page. A byte offset is relative to its page, not to the entire dictionary.

pageID: The page identifier, which is a zero-based index (beginning at page zero and continuing through the total number of pages for the dictionary minus one).

2.3.3 Column Data Hierarchy Hash Index

Column data can have an associated value hash index file generated. An example of a generated file name for a column data hierarchy hash index file for a table that has the identifier "Table1" and a column that has the identifier "Cat" is 1.H$Table1$Cat.hidx. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The column data hierarchy hash index is a hash index file of the unique data identifier values that are present in the column. The file is neither ordered nor compressed.

It is necessary to reference the XML metadata file to determine whether a column data hierarchy hash index file is present. If a particular **XMRawColumn** object—specifically, the **XMRawColumn** object of the **Column** item in the **Columns** collection that has a name equal to the column name—has a **DataObject** in the **DataObjects** collection for which the class="XMValueDictionary<XM_Real>" (section 2.5.2.19) or for which the class="XMValueDictionary<XM_Long>" (section 2.5.2.18), the column MUST have a column data hierarchy hash index file generated. For an explanation of how to interpret the XML metadata file, see section 2.5.

2.3.3.1 File Layout for Hash Index Files

Unlike .idf files or **XM_TYPE_STRING** type dictionary files, hash index files—that is, files that have the .hidx file name extension—do not use any compression.

The description of the hash index file format layout (also referred to as the .hidx file format) is divided into sections. The first section (section 2.3.3.1.1) pertains to hash information that MUST be present in any file that uses a hash. Such files have either the .hidx or the .dictionary extension. These files include column data hierarchy hash index files (section 2.3.3), relationship hash index files (section 2.4.3), and dictionary files (section 2.3.2.1).

2.3.3.1.1 Required Elements for All Files That Use Hashing

Whenever a hash is used (except in the case of dictionaries of type **XM_TYPE_STRING** as specified in section 2.3.2.1.2), the first five elements of the hash table MUST be present. These elements are described in the remainder of this section.

The first 4 bytes represent an enumeration value that either identifies the hash algorithm used or equals the **XM_INVALID** value, which indicates that no hashing algorithm was specified in the persisted file. In most cases, there is no choice for the hashing algorithm, so the hashing algorithm does not need to be saved for future reference. However, this algorithm MUST be specified correctly; otherwise, an error could occur. For more information about the hash algorithm enumeration, see section 2.3.3.1.4.2.

Dictionary files need to include only the first five required elements, so no knowledge of the actual hashing algorithm (that is, the actual procedural code used) is needed. However, in hash index files (.hidx files), full hash information is required, so the actual hashing algorithm used to create this information MUST be the one that is specified in section 2.3.3.1.3. This hashing algorithm is the one that the system uses and expects. Also, as specified in the hash algorithm enumeration (section 2.3.3.1.4.2), the hash algorithm identifier is set to **XM_INVALID**.

The next 4 bytes contain the size, in bytes, of the **HashEntry** hash entry structure (section 2.3.3.1.4.5). Each hash entry structure is this size, which is referred to here as **HashEntrySize**. This size varies depending on the hash policies that are in place. For more information about the **HashEntry** structure and hash policies, see section 2.3.3.1.4.5.

The next 4 bytes specify the size, in bytes, of the **HashBin** hash bin structure. This size is referred to as **HashBinSize** for this file format description. The structure varies in size because it includes **HashEntry** structures. For more information about the **HashBin** structure, see section 2.3.3.1.4.4.

The next 4 bytes specify local entries or a local entry count, meaning the number of hash entries that are allowed per bin before an overflow occurs (section 2.3.3.1.4.6). If overflow hash entries (also called *collision entries*) exist, they are added sequentially to the end of the file.

The next 8 bytes specify the number of bins that are used in the hash. This number is referred to as **cBins** in this file format description. Depending on the data type, a minimum required value exists for the number of bins (also referred to as the minimum number of buckets for a hash). For more information about bin count minimums, see section 2.3.3.1.4.3. The value of **cBins** either reflects the number of bins or is set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** (section 2.3.3.1.4.1).

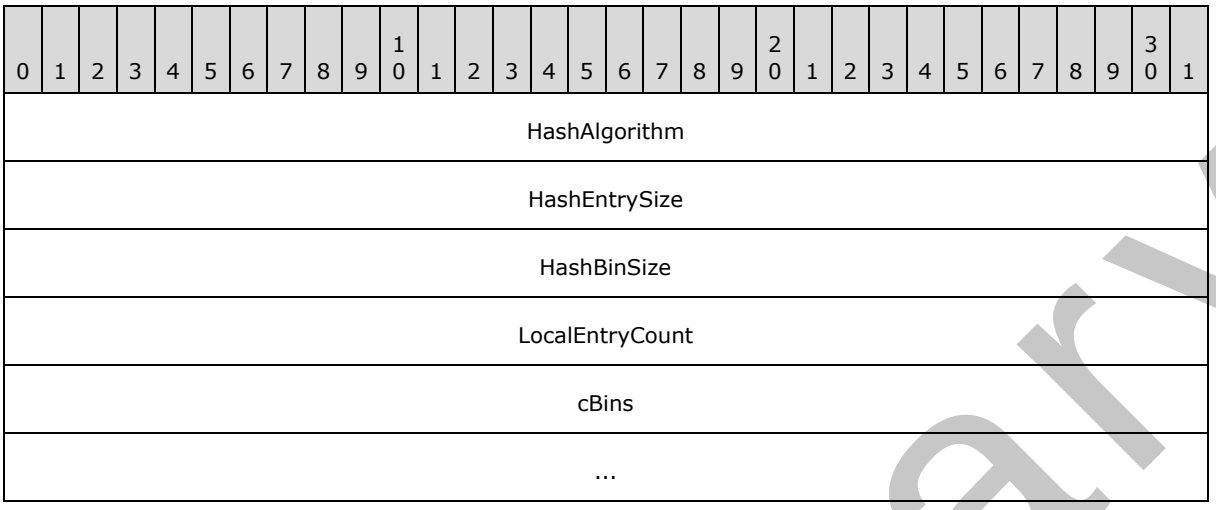
If **cBins** is set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT**, that signals that the hash table was not created, so processing of the hash in the file is stopped at this point. Dictionary files (section 2.3.2.1) MUST set this value to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** to indicate that they do not include any hash table information beyond these required elements. For more information about **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT**, see section 2.3.3.1.4.1.

It is important to emphasize that even if **cBins** is set to stop any further hash table processing, the **HashBin** and **HashEntry** structures MUST be the correct, expected size for the current hash policy in place and the type of hash in use, the hash algorithm MUST be identified as expected by the system (section 2.3.3.1.4.2), and the local entries value MUST equal the **XM_HASH_ENTRY_COUNT_PER_BIN** calculated value (section 2.3.3.1.4.6), which depends on the hash policies that are in place and the type of hash.

If any of these values is incorrect, a file error will likely occur. For more information about all of these structures, see section 2.3.3.1.4. For a discussion about the effects of hash policies, see section 2.3.3.1.4.5.

The following diagram shows the common bytes that are required in any file containing hash information (that is, .hidx and .dictionary files). Note that string dictionaries comprise the only dictionary file type that can fully exclude any hash information, but there are ramifications to doing

so (section 2.3.2.1.2.2.) For more information about the actual structure of **HashBin** or **HashEntry**, section 2.3.3.1.4.4 or section 2.3.3.1.4.5, respectively.



HashAlgorithm (4 bytes): The hash algorithm specified.

HashEntrySize (4 bytes): The size of the **HashEntry** structure.

HashBinSize (4 bytes): The size of the **HashBin** structure.

LocalEntryCount (4 bytes): The number of hash entries that are allowed per bin before an overflow (collision) occurs.

cBins (8 bytes): Either the number of bins used in the hash or **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT**. The latter value is used by dictionaries.

2.3.3.1.2 Required Elements for Hash Index Files

The following subsections contain required information for hash index files (.hidx files). Note that an .hidx file also requires the elements discussed in section 2.3.3.1.1.

2.3.3.1.2.1 Records and Hash Statistics

This section covers the number of records in the hash and any hash statistics that are included in the file. The next 8 bytes specify the number of records in the hash table. The next 8 bytes after that indicate the current mask to be used. This mask simply equals the number of bins minus one. The next byte contains a Boolean flag that indicates whether hash statistics were gathered and included in the file. If this value is **true**, hash statistics have been included in the file.

The hashing algorithm used to create the hash table is specified in section 2.3.3.1.3.

The rest of this section deals with the hash statistics. This data is present only if the flag that indicates the gathering of hash statistics is set to **true**. If the flag is set to **false**, the hash statistics data MUST NOT be included.

Because the hash statistics section is an optional section (indicated by the just-mentioned flag), some of the information in the section duplicates information that is found in previous elements. If hash statistics have been included, the next 8 bytes indicate the number of elements in the hash.

The next 8 bytes following that indicate the number of bins available in the hash, and the following 8 bytes indicate the number of bins available that were actually used in the hash.

The next 8 bytes represent the number of elements that were available via fast access. An element is a fast access element if it is in the actual hash bin and therefore not an overflow (or collision) for that hash bin. An overflow element requires a longer access time.

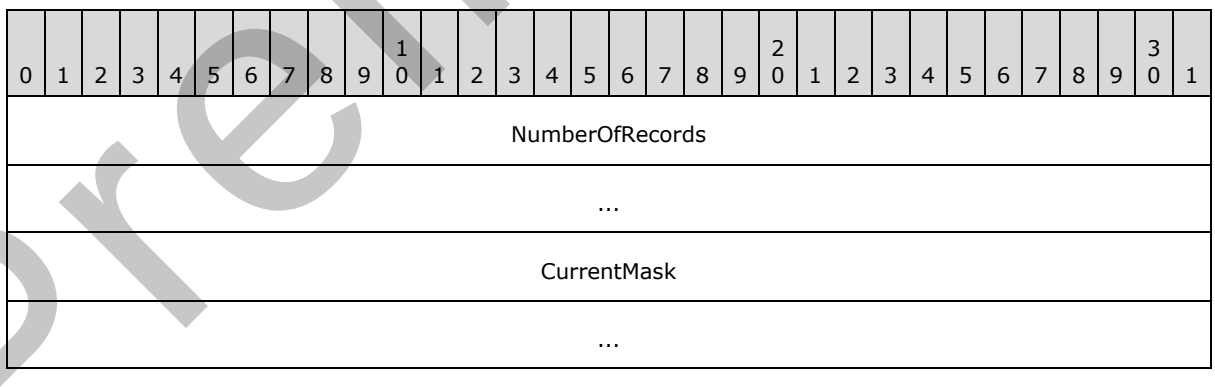
The next 8 bytes contain the local entry size for each bin—in other words, the number of hash entries that can be contained in one bin before an overflow occurs. This size is the same as the local entry count mentioned in the five required elements (section 2.3.3.1.1). The next 8 bytes contain the maximum probes, or maximum chain count, for one bin. This value indicates the largest number of entries for one of the bins, over the range of all the bins. If this value is greater than the maximum number of hash entries that can be contained in a bin before an overflow occurs, at least one of the bins has had an overflow and thus has extra hash entries (also referred to as *collision entries*). These extra entries are added to the end of the file.

The next set of bytes represent a histogram. Of these bytes, the first 8 (**elementCount**) represent the number of elements in the histogram. The next 4 bytes (**elementSize**) represent the size, in bytes, of each element in the vector (or array). Therefore, the histogram itself is of variable size—specifically, **elementCount** multiplied by **elementSize**, in bytes. The histogram is a vector of elements of size **elementSize**. If the value of an element in the histogram vector is not zero, that value signifies the number of hash bins containing a certain number of entries in the bin—where the certain number is the array index number of that element. For example, if the histogram vector element at index 3 contains a value of 5, five hash bins each have three hash entries in their bin.

The histogram can contain empty array elements because it MUST include all the previous array elements up to the array elements that have nonzero values. The histogram can also include empty array elements following the last nonzero array element. If the number of entries signifies an overflow, the histogram will still show the total number of hash entries in a bin, including the overflow elements. Note that despite the fact that this value includes the total number of hash entries, a hash bin in reality can contain only the maximum number of elements already specified as the value of the **LocalEntryCount** field (section 2.3.3.1.1), and any overflow entries are chained internally and persisted to the file at the end of the file.

For example, if the histogram array element at index 3 contains a value of 9, three hash bins have nine entries each in their respective bins. If the maximum local entry count (or local entry size) for each bin is six, the implication is that each of those bins had an overflow and have three collision entries each (because nine minus six equals three) at the end of the file.

The following diagram shows the hash structure elements just discussed, beginning with the number of records and ending with the hash statistics histogram element.



HashStats	NumberOfElements
...	
...	NumberOfBins
...	
...	NumberOfUsedBins
...	
...	FastAccessElements
...	
...	LocalsSizePerBin
...	
...	MaximumChain
...	
...	elementCount
...	
...	elementSize
...	
...	HistogramVector (variable)
...	

NumberOfRecords (8 bytes): The number of records in the hash table.

CurrentMask (8 bytes): The current mask to use.

HashStats (1 byte): A Boolean flag that specifies whether hash statistics have been included in the file. If the value is **true**, hash statistics have been included in the file. These hash statistics are the elements following this Boolean flag—from **NumberOfElements** through **HistogramVector**.

NumberOfElements (8 bytes): The number of elements in the hash.

NumberOfBins (8 bytes): The number of bins available in the hash.

NumberOfUsedBins (8 bytes): The number of bins that are actually used in the hash.

FastAccessElements (8 bytes): The number of elements with fast access.

LocalsSizePerBin (8 bytes): The number of hash entry structures that can be contained in one bin before an overflow occurs.

MaximumChain (8 bytes): The largest number of entries in a bin, over all the bins.

elementCount (8 bytes): The number of elements in the histogram vector.

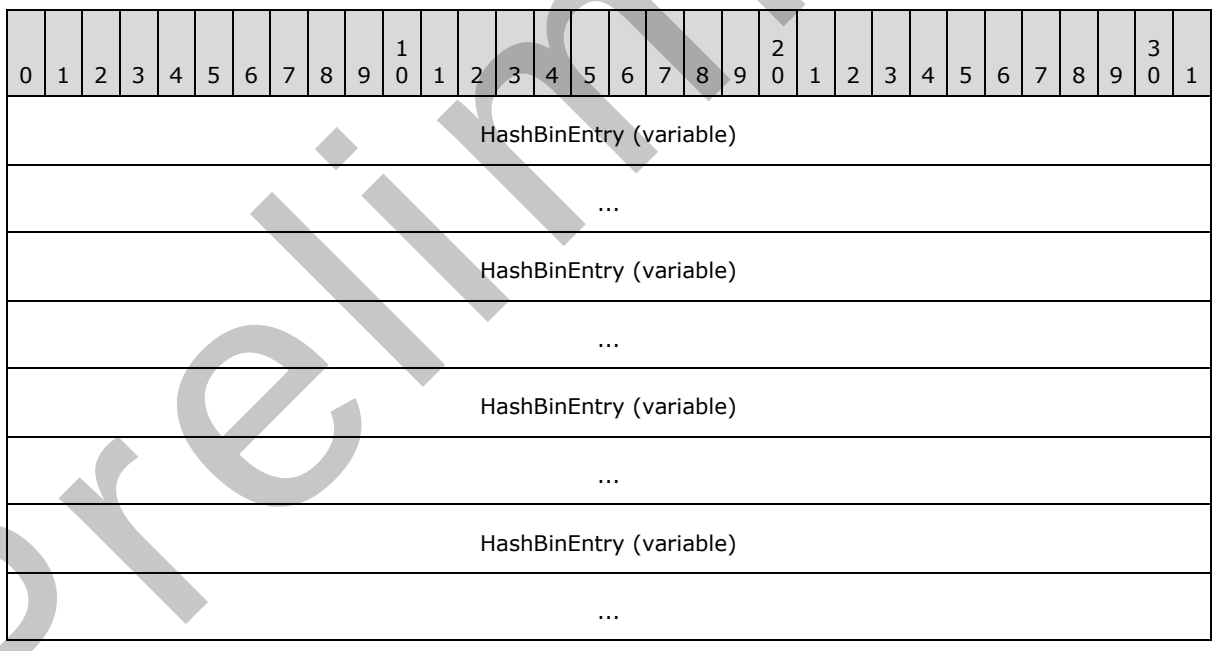
elementSize (4 bytes): The size, in bytes, of each element in the histogram vector.

HistogramVector (variable): The vector of elements, each of which has the size that is specified by **elementSize**.

2.3.3.1.2.2 Hash Bin Entries

The next section of the hash includes the hash bins that are created and used during the hashing process. The number of entries, which appear in sequential bin order, is specified by **cBins**. Each entry is the size (**HashBinSize**) of the hash bin structure (**HashBin** structure). Each hash bin structure also contains an array of hash entries (**HashEntry** structures), up to the maximum number that is allowed per bin, so no overflow entries will exist within this structure. The sizes of the hash bin structure and the hash entry structure vary depending on several factors, including the operating system that is used and the current hash policies that are in place. For more information about the **HashBin** structure and the **HashEntry** structure, see section 2.3.3.1.4.4 and section 2.3.3.1.4.5.

The following diagram shows a general view of the hash bin entries in a file. This particular example shows four **HashBin** entries that simply exist sequentially, one after another, in the file. The diagram does not show the details within each hash bin entry.



HashBinEntry (variable): A **HashBin** structure.

HashBinEntry (variable): A **HashBin** structure.

HashBinEntry (variable): A **HashBin** structure.

HashBinEntry (variable): A **HashBin** structure.

2.3.3.1.2.3 Overflow Hash Entries

The final section of the file includes the collisions, if any, from any overflow in the hash bins. The next 8 bytes contain the total count of the collisions. This value includes the collisions for all the hash bins. This value is followed by the collision entries. Each collision entry is a **HashEntry** structure and, as such, is the size (**HashEntrySize**) of a hash entry. For more information about the **HashEntry** structure, see section 2.3.3.1.4.5.

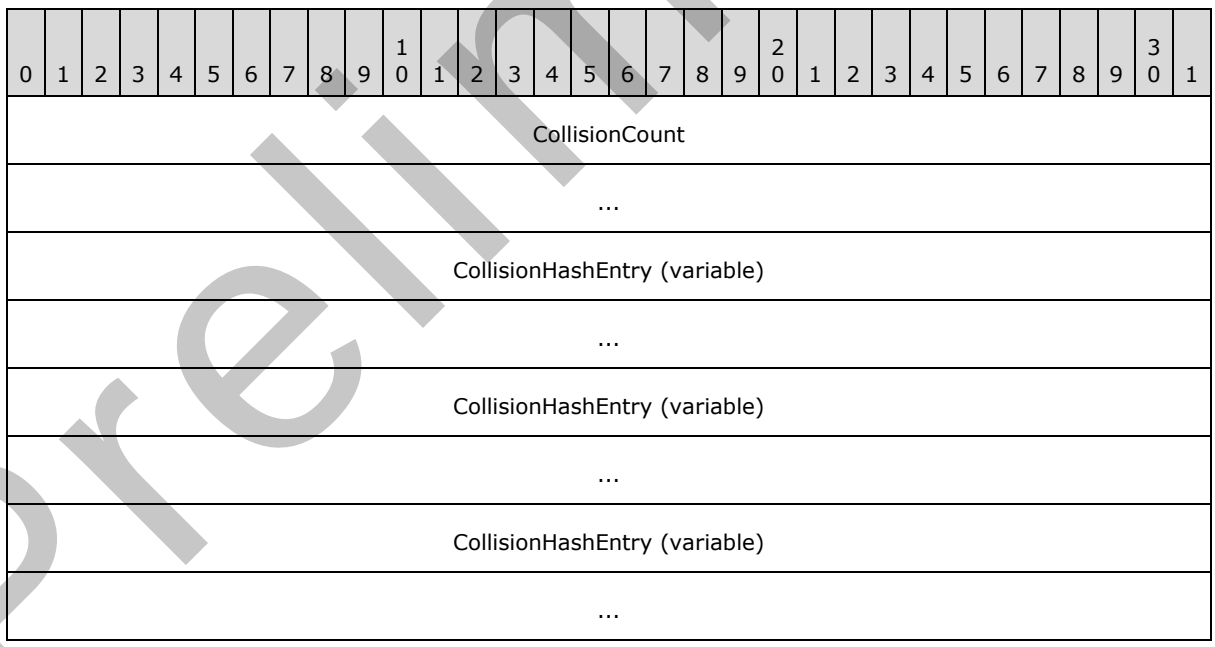
Collisions are added sequentially to the end of file in the same order as their associated hash bins. Zero or more collisions might exist for a hash table. However, the 8 bytes that specify the count of collisions MUST be present, even if it contains a value of zero to indicate that no collisions exist.

The collision entries are the last elements in a hash index file (.hidx file). Padding with zeros might exist at the end of the file. If present, this padding is ignored and not read by the system.

The following diagram shows a general view of the collisions count and the collision hash entries in the file.

This particular example shows four collision **HashEntry** entries that simply exist sequentially, one after another, in the file. The details within the **HashEntry** structures are not shown.

Each of these collision entries corresponds to a hash bin that has one or more collisions. However, which collision entry corresponds to which hash bin cannot be gleaned from just the sequential collision entries here. The correspondence can, however, be inferred by looking at the **m_Count** member of each of the **HashBin** structures (section 2.3.3.1.4.4) to see whether the value of **m_Count** exceeds **XM_HASH_ENTRY_COUNT_PER_BIN** (section 2.3.3.1.4.6), which equals the value of the local entry count in the five required hash elements (section 2.3.3.1.1).



CollisionHashEntry (variable)
...

CollisionCount (8 bytes): The total number of collisions over all the hash bins.

CollisionHashEntry (variable): The **HashEntry** structure for this collision entry.

CollisionHashEntry (variable): The **HashEntry** structure for this collision entry.

CollisionHashEntry (variable): The **HashEntry** structure for this collision entry.

Collision HashEntry (variable): The **HashEntry** structure for this collision entry.

2.3.3.1.3 Hashing Algorithms

The following hashing algorithm is used by hash index files (files with extension HIDX). This algorithm MUST be used in order to ensure proper hash table creation.

The hashing algorithm is shown in pseudocode.

```
INPUT keyValue
CREATE MagicConstant as an unsigned 32 bit integer
CREATE cHashBitsUsed as 64 bit integer
CREATE cBuckets as a double
SET cBuckets to number of bins, also called buckets, to be used in hash
SET MagicConstant to 0x12B9B6A5
SET cHashBitsUsed to ( CEILING (LOG(cBuckets)/LOG(2.0)) )
SET keyValue to (keyValue MultiplyBy MagicConstant)
SET keyValue to (keyValue RIGHT_BITSHIFT by (32 - cHashBitsUsed))
SET hashForKeyValue to keyValue
OUTPUT hashForKeyValue
```

The hash and key value variable types can be determined by looking at the **HashEntry** structure (section 2.3.3.1.4.5).

There is a minimum size for the number of bins used for proper hash table creation. The variable *cBuckets* MUST use at least the minimum bin value. For the bin minimum values, please see section 2.3.3.1.4.3.

2.3.3.1.4 Hash Structures, Enumerations and Constants

The following subsections contain information related to the data structures, enumerations, and other constants that are used by files that include hash information (that is, .hidx and .dictionary files).

2.3.3.1.4.1 XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT

The **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** constant indicates that the bin count is invalid.

When the value of **cBins** (section 2.3.3.1.1) is set to this constant, it means that no further hash information is included in the file. This value is used by column data dictionary files (.dictionary files). For more information about dictionary files, see section 2.3.2.

XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT is not used in hash index files (.hidx files) because by definition, these files are expected to have full hash information, with only the hash statistics being optional (section 2.3.3.1.2.1).

The **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** constant has a value of –1.

2.3.3.1.4.2 Hash Algorithm Enumeration and Constant

The **XMHashAlgorithm** enumeration indicates which **XM_TYPE_STRING** dictionary hash algorithm is in use. The **XM_INVALID** constant is used in all other cases.

In most hash algorithm scenarios, the hash algorithm value is set to **XM_INVALID**, which indicates that no hashing algorithm was specified in the persisted file. The algorithm is determined at run time based on the type of file and its data.

For hash index files (.hidx files) and column data dictionary files (.dictionary files) of type integer (**XM_TYPE_LONG**) or real (**XM_TYPE_REAL**), it is always the case that the hash algorithm value is set to **XM_INVALID**. For more information about the **XM_TYPE** enumeration, see section 2.3.2.1.3.1.

Name	Value
XM_INVALID	–1

However, for dictionaries of type **XM_TYPE_STRING**, the value is not **XM_INVALID** but MUST be one of values in the **XMHashAlgorithm** enumeration.

```
enum XMHashAlgorithm
{
    XM_HASH_ALGORITHM_SQL         =    0,
    XM_HASH_ALGORITHM_FAST_CI     =    1,
    XM_HASH_ALGORITHM_FAST_CS     =    2
};
```

The following table describes the enumeration values in **XMHashAlgorithm**.

Enumeration value	Meaning
XM_HASH_ALGORITHM_SQL	The hash algorithm is the default algorithm. It will work for all string situations but SHOULD NOT be chosen if one of the other two values in **XMHashAlgorithm** can be used.
XM_HASH_ALGORITHM_FAST_CI	This algorithm can be used if the locale identifier is 1033 and it is acceptable to ignore the case of the characters in the strings (that is, if the strings are case insensitive).
XM_HASH_ALGORITHM_FAST_CS	This algorithm can be used if either the locale identifier is 1033 and the characters are case sensitive or the strings are BLOBs that use base64 encoding.

2.3.3.1.4.3 Hash Bin Bucket Size Minimums

A minimum number of bins (or buckets) is required for hashes that are used in .hidx files.

In general, for all data other than string data, the minimum bin count is 16. For string data, there MUST be a minimum of 256 bins. Note that .hidx files contain integer data, not string data, as indicated by the **m_Key** and **m_Value** members in the **HashEntry** structure (section 2.3.3.1.4.5).

Name	Value
XM_HASH_MINIMUM_BIN_COUNT	16
XM_STRING_HASH_MINIMUM_BIN_COUNT	256

2.3.3.1.4.4 HashBin Structure

The **HashBin** structure contains information about the hash bins used.

The **HashBin** structure is system cache aligned and therefore varies in size according to the operating system and alignment requirements for that operating system. For alignment reasons, the chain pointer (the **m_rgChain** member) MUST be the first element in the structure. For 32-bit systems, padding is also added to the structure to ensure that the count-of-entries element is properly aligned.

The hash bin contains one or more hash entries. However, because of alignment issues and the variable size of the **HashEntry** structure (section 2.3.3.1.4.5), it is possible for the local hash entry array (the **m_rgLocalEntries** member) to contain no hash entries, where **XM_HASH_ENTRY_COUNT_PER_BIN** evaluates to zero. In such a case, all the hash entries are referenced through the chain pointer, **m_rgChain**. (The chain pointer is an array of **HashEntry** structures.)

```
DECLSPEC_ALIGN(SYSTEM_CACHE_ALIGNMENT_SIZE) struct HashBin
{
    HashEntry*          m_rgChain;
#ifndef _WIN64
    unsigned __int32    m_Padding;
#endif
    unsigned __int32    m_Count;
    HashEntry           m_rgLocalEntries[XM_HASH_ENTRY_COUNT_PER_BIN];
};
```

m_rgChain: The pointer for the chain of collisions, which is a chain of **HashEntry** structures that represent the overflow (collision) entries. The value is NULL when persisted to disk, and the collisions are added to the end of the file.

m_Padding: Padding for alignment purposes. This padding is included for 32-bit systems (the compiler constant **_WIN64** is not defined) and excluded for 64-bit systems (**_WIN64** is defined).

m_Count: The total number of **HashEntry** entries in the bin.

m_rgLocalEntries: An array of size **XM_HASH_ENTRY_COUNT_PER_BIN** that contains the locally stored **HashEntry** entries (not the overflow collision entries).

The **DECLSPEC_ALIGN** macro is defined as follows:

```
#if (_MSC_VER >= 1300) && !defined(MIDL_PASS)
#define DECLSPEC_ALIGN(x)   __declspec(align(x))
#else
#define DECLSPEC_ALIGN(x)
```

The **SYSTEM_CACHE_ALIGNMENT_SIZE** constant is defined as follows:

```
#if defined(_AMD64_) || defined(_X86_)
#define SYSTEM_CACHE_ALIGNMENT_SIZE 64
#else
#define SYSTEM_CACHE_ALIGNMENT_SIZE 128
#endif
```

For more information about the **HashEntry** structure, see section 2.3.3.1.4.5. For more information about **XM_HASH_ENTRY_COUNT_PER_BIN**, see section 2.3.3.1.4.6.

2.3.3.1.4.5 HashEntry Structure

The **HashEntry** structure contains information about the hash entries used.

In most cases, a hash entry structure contains a key/hash pair for the hash table. However, depending on the hash policies in use, the hash entry structure information can vary, including either more or less information. The hash key MUST be included. Other elements vary.

A hash policy defines which elements are required in the **HashEntry** structure. The hash policies vary according to internally defined variables that seek to balance load factors, storage issues, and processing speeds.

The hashes for **XM_TYPE_LONG**, **XM_TYPE_REAL**, and **XM_TYPE_STRING** hash data dictionaries are recalculated at run time, so only the basic, required elements for the hash are required in the file format (section 2.3.3.1.1), with **cBins** set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT** (section 2.3.3.1.4.1). Even so, hash policies play a role in determining the expected size of the **HashBin** (section 2.3.3.1.4.4) and **HashEntry** structures and the value of the local entry count (section 2.3.3.1.4.6) because they all use the **HashEntry** structure.

However, for files with the .hidx file name extension, where full hash information is required, all the hashing information is included and MUST be accurate. For hash index files (.hidx files), the hash value (**m_Hash**) is included in **HashEntry**, along with the required element, the key (**m_Key**). The hashing algorithm used is also specified and MUST be used to generate the hash information (section 2.3.3.1.3).

The following code shows the full HashEntry structure, which is defined by using the DECLSPEC_ALIGN(X) macro (section 2.3.3.1.4.4), where **X** = 1. The code includes comments that contain pseudocode to indicate which structure members will be included in the structure for various policy settings. Only **m_Key** is guaranteed to be in any HashEntry structure for any policy.

```
DECLSPEC_ALIGN(1) struct HashEntry
{
    // IF this policy is true (HASHPOLICY_INCLUDEHASH) THEN
    //    include member m_Hash

         unsigned __int32        m_Hash;

    // END IF

    // IF this policy is true (HASHPOLICY_INCLUDELENGTH) THEN
    //    include member m_Len

         unsigned __int32        m_Len;
```

```
    // END IF

          TKey                    m_Key; // Required member in all cases

    // IF this policy is true (HASHPOLICY_INCLUDEVALUE) THEN
    //     include member m_Value

          TValue                  m_Value;

    // END IF
};
```

m_Hash: The hashing value that is paired with the key value, **m_Key**. Including and using the hash instead of the value makes sense for simple types, where comparing the hashes has the same cost as comparing the values. It does not make sense for complex types or character data (strings).

m_Len: The length field. The meaning varies, depending on usage. For example, for strings, the length means the length of the string. Using this member increases the size of the structure and might result in more overflows (collisions).

m_Key: The key value that is associated with the hashing value, **m_Hash**. This member MUST be present in all cases. The data type for this value varies, depending on the type of the hash table being used, as shown in the following table.

File type	Data type for m_key (TKey resolution)
.hidx file	**Int32**
XM_TYPE_LONG dictionary (32-bit)	**Int32**
XM_TYPE_LONG dictionary (64-bit)	**Int64**
XM_TYPE_REAL dictionary	**DOUBLE**
XM_TYPE_STRING dictionary	**Int32**

m_Value: The actual value that is associated with the key, **m_Key**. The data type for this value varies depending on the type of hash table that is being used, as shown in the following table.

File type	Data type for m_Value (TValue resolution)
.hidx file	**Int32**
XM_TYPE_LONG dictionary (32-bit)	**Int32**
XM_TYPE_LONG dictionary (64-bit)	**Int32**
XM_TYPE_REAL dictionary	**Int32**
XM_TYPE_STRING dictionary	**Int32**

The following table summarizes the relationship between a hash policy setting and the files that are affected by that setting. Note that the constant string hash policy is only for **XM_TYPE_STRING** dictionaries. This policy can be used only when each string mapped via the hash is guaranteed to be

a unique string. **XM_TYPE_STRING** dictionaries can use this hash policy when the strings per page are guaranteed to be unique strings, with no duplicates on that page (section 2.3.2.1.2.2).

Hash policy setting	Affected files	Description
HASHPOLICY_INCLUDEHASH	Both .hidx and .dictionary files	This value (true) is the default for all hash policies. Note that for **XM_TYPE_STRING** dictionaries, the data identifier is the key, but the length of the string is not stored separately from the hash value. Instead, the high-order word of the string hash and the low-order word of the string's length are combined into the stored hash value. This applies to both Huffman compressed strings and strings that are not compressed. However, this does not apply if the constant string hash policy is being used, instead. In that case, the length is included separately.
HASHPOLICY_INCLUDELENGTH	XM_TYPE_STRING dictionaries (when full hash is included)	This value is true if the constant string hash policy is being used.
HASHPOLICY_INCLUDEVALUE	XM_TYPE_STRING dictionaries (when full hash is included)	This value is true if the constant string hash policy is being used.

2.3.3.1.4.6 XM_HASH_ENTRY_COUNT_PER_BIN

The **XM_HASH_ENTRY_COUNT_PER_BIN** value dictates how many **HashEntry** structures (section 2.3.3.1.4.5) can be contained in a **HashBin** structure (section 2.3.3.1.4.4) before an overflow occurs.

The **XM_HASH_ENTRY_COUNT_PER_BIN** value depends on the target architecture's cache line size as well as the **HashEntry** structure (section 2.3.3.1.4.5) being used, which varies in form and size depending on which hash policies are in effect.

For 32-bit and 64-bitapplications, how to calculate the **XM_HASH_ENTRY_COUNT_PER_BIN** value is shown in the following pseudocode:

```
IF application is 32 bit OR 64 bit THEN
   SET XM_HASH_ENTRY_COUNT_PER_BIN to ( (CacheLineSize MultiplyBy SYSTEM_CACHE_ALIGNMENT_SIZE
- (8 + (Size-Of (unsigned 32 bit integer))) / (Size-Of (HashEntry structure) )
END IF
```

For the definition of **SYSTEM_CACHE_ALIGNMENT_SIZE**, see section 2.3.3.1.4.4.

CacheLineSize is dependent on the hash policies that are in place (see section 2.3.3.1.4.5). However, in all cases in this document, **CacheLineSize** is equal to 1.

The hash entry structure (**HashEntry**) is variable and its size depends on the hash policies that are in place. For more information, see section 2.3.3.1.4.5.

2.3.4 RowNumber Column

Every table MUST have a RowNumber file generated for the table. An example of a generated file name for a RowNumber file for a table that has the identifier "Table1" is 4.Table1.RowNumber.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The RowNumber file is used to tell the system how many rows are contained in each data segment of the column data. This information is always encoded by using XMHybridRLECompressionInfo<class XM123CompressionInfo> compression (section 2.7.3.16).

It is necessary to reference the XML metadata file to understand and decode the contents of the RowNumber file. The metadata file for the RowNumber file is contained in the same file as the metadata for the column data (section 2.3.1). The **Column** item in the **Column** collection for which the value of the **name** attribute is "RowNumber" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.3.4.1 File Layout for the RowNumber Column

The **RowNumber** column is a special case for column data storage files. In addition to generating a column data storage file (.idf file) for every column in a source data table, an additional .idf file is generated to represent a column of row numbers. This **RowNumber** column is an internally generated column, but as a result, does have its own column data storage file (.idf file). The purpose of a RowNumber file is to associate a row number index with each row of each segment, per segment, for the entire span of the column. As such, a **RowNumber** column provides a row number index that can be used to select a particular row or set of rows across columns. However, because the system generates these files and knows their function, it does not mean that each actual row number is encoded in the .idf file. In actuality, a more compact way is used.

In fact, although **RowNumber** columns do use XMHybridRLE compression, they use only XM123CompressionInfo subsegment compression. This hybrid combination is unlike the typical XMHybridRLE and XMRENoSplit case. The .idf file format layout is the same, but the subsegment is just a placeholder, and the RLE segment for each segment is encoded in a special way that details the row number information, which the system already knows how to interpret. For a detailed discussion of the compression used for the **RowNumber** column and how to interpret the information correctly, see section 2.7.3 and section 2.7.3.16.

2.4 System-Generated Data Files

The system generates not only the files that represent the data but additional files, depending on the data. These additional files are described in this section.

2.4.1 Column Data Position–to–Identifier Mapping

Column data can have a position–to–identifier mapping file. An example of a position–to–identifier mapping file name for a table that has the identifier "Table1" and a column that has the identifier "Label" is 1.H$Table1$Label.POS_TO_ID.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The position–to–identifier mapping file contains an array of data identifier values. The order of the array is by the sorted order of the underlying values that the data identifiers represent. The first value in the array corresponds to the data identifier for which the underlying value that the data identifier represents is first in sorted order. For example, if the values in the column are sorted, and the first value after sorting has a data identifier of 5, the first value in the array is 5. The second array value corresponds to the data identifier of the second entry when the column values are sorted, and so on.

The position–to–identifier mapping file is compressed by one of several methods, although it is always be compressed by using an XMRENoSplit compression method and does not use XMHybridRLE compression. For a discussion of the types of compression available that are to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the column position–to–identifier mapping file. An example of a file name for the file that contains the metadata for the position–to–identifier mapping file for a column that has the identifier "net" in a table that has the identifier "Table1" is H$Table1$net.0.tbl.xml. The metadata for the position–to–identifier mapping column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "POS_TO_ID" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.1.1 File Layout for Column Data Position–to–Identifier Mapping File

The column data position–to–identifier mapping file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (section 2.3.1.1.) However, differences exist.

The position–to–identifier mapping file is a system-generated file and never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that, at a minimum, a position–to–identifier mapping file always has one segment and is never associated with a subsegment. The reason is that subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.4.2 Column Data Identifier–to–Position Mapping

Column data can have an identifier–to–position mapping file. A data identifier–to–position mapping file is generated only in the case where a value hash table has been generated. An identifier–to–position mapping file is not generated for a dictionary file. An example of a generated identifier–to–position mapping file name for a table that has the identifier "Table1" and a column that has the identifier "net" is 1.H$Table1$net.ID_TO_POS.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The identifier–to–position mapping file contains an array of position values that are zero-based numbers. These position values represent the positions within the sorted values of the underlying source data values that are represented by the data identifiers. The order of the array is by data identifier value, from lowest to highest. The first array value is the position within the set of source data values of the lowest-numbered data identifier, the second value is the position of the second-lowest data identifier value, and so on. For example, if the lowest-valued data identifier is sorted to the fifth position in the array of the source data values that are underling the data identifiers, the first value in this array will be 4 (because the array is zero-based). The second array item contains the position within the array of sorted source data values that is assigned to the second-lowest data identifier value, and so on.

The identifier–to–position mapping file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not use XMHybridRLE compression. For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the column identifier–to–position mapping file. An example of a file name for the file that contains the metadata for the data identifier–to–position mapping file for a column that has the identifier "net" in a table that has the identifier "Table1" is H$Table1$net.0.tbl.xml. The metadata for the data

identifier–to–position mapping column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "ID_TO_POS" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.2.1 File Layout for Column Data Identifier–to–Position Mapping File

The column data identifier–to–position mapping file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (section 2.3.1.1.) However, differences exist.

The identifier–to–position mapping file is a system-generated file and never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that, at a minimum, an identifier–to–position mapping file always has one segment and is never associated with a subsegment. The reason is that subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.4.3 Relationship Index

If a relationship between two tables in a tabular data model is defined, a relationship index is generated. Tabular data models require that the key column in one of the tables be unique (many-to-many relationships are not allowed). The relationship index file is generated for the table that is on the "many" side of the relationship. An example of a generated relationship index file name for the "many" table in the relationship is 73.R$Table1$c4047114-e5d3-4730-ab46-478baf7ae64f.INDEX.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The relationship index file contains an array of integers. One integer exists in this file for each unique value in the join column of the table on the "many" side of the relationship. The first integer that is present in the file corresponds to the first unique value that is encountered, starting with the first row, in the join column of the "many" table. The second integer corresponds to the second unique value that is encountered in the join column, and so on. The integer value is the row number in the other table of the relationship (the "one" table) to which the row is joined. Row numbering is zero-based. If a row cannot be joined because no value match exists, the value –1 will appear in the relationship index file.

The relationship index file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not XMHybridRLE compression. For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the relationship index file. An example of a file name for the file that contains the metadata for the relationship index file is R$Table1$c4047114-e5d3-4730-ab46-478baf7ae64f.73.tbl.xml. The metadata for the relationship index column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "INDEX" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.3.1 File Layout for Relationship Index File

A relationship index file can have one of two file format forms: that of either an .idf file or an .hidx file. The relationship index file typically uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (section 2.3.1.1). However, when the

relationship index file uses the .idf file format layout, differences from a column data storage .idf file exist.

The relationship index .idf file is a system-generated file and never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that a relationship index file always has one segment at a minimum and is never associated with a subsegment. The reason is that subsegments are associated only with XMHybridRLE compression. which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

Most of the time, the relationship index file uses the .idf file format. However, in some situations when sparse relationship information exists (which means that very large gaps exist between the column values with relationships), the relationship index file takes the form of an .hidx file (hash index file).

For example, if only two values exist for the column, one with a value of 2 and the other with a value of 5 billion, using an .idf file will generate rows for all the unused values between 2 and 5 billion. A hash table simply encodes the two key-value pairs of interest. The hash table encodes a key-value pair with the column value (the data identifier) as the hash key, and the row number as the hash value. For more information about the hash index file format file layout, see section 2.3.3.

2.4.4 User Hierarchy System-Generated Files

A tabular data model allows users to define hierarchies. A hierarchy is defined by its levels, where one column in the source data contains the value for a particular level. For example, a common hierarchy for geography has "Country/Region" at the top level, "State" at the next level down, and "City" at the level below "State". In this case, a column in the source data table exists that contains the value for each of the levels.

For each defined user hierarchy, the system generates four data files. These files consist of the child count file (section 2.4.4.1), the first child position file (section 2.4.4.2), the multilevel identifier file (section 2.4.4.3), and the parent position file (section 2.4.4.4). The integer values that appear in these files require understanding an in-memory data structure that is formed by the system. Note that this data structure is never materialized or seen by users.

A data structure is formed in which each combination of distinct values that actually exists at all of the levels is represented by rows in a table. The table starts at the highest level and descends through the levels. All row numbering that refers to the table is zero-based—that is, the first row is row 0.

The first *N* rows in the table, which are numbered from 0 through (*N* – 1), consist of all the distinct values at the highest level in the user hierarchy. The values are sorted by the collation for that column. For example, if the highest level in the user hierarchy is "Country/Region", and the countries that are present in the data are "United States" and "Canada", then row 0 in the table is for "Canada" (the first item in sort order), and row 1 in the table is for "United States".

The next *N* rows in the table represent all the valid combinations of values that exist at the highest level and at the next-highest level in the user hierarchy. For example, if the next-highest level is "Area"; the United States has areas named "Northwest", "Northeast", "Southwest", and "Southeast"; and Canada has areas named "East" and "West"; six rows in the table will exist to represent the second level, and they are rows 2 through 7. Within each level, the values appear in sorted order under their common parent level, and the parent levels remain in the originally sorted order.

Therefore, for the first two levels in the example table, the rows of the in-memory table will be as shown in the following table.

Row	Value
0	"Canada"
1	"United States"
2	"Canada-East"
3	"Canada-West"
4	"United States-Northeast"
5	"United States-Northwest"
6	"United States-Southeast"
7	"United States-Southwest"

The process just described is repeated recursively until valid combinations of items that exist at each level of the user hierarchy are represented in the table.

2.4.4.1 User Hierarchy Child Count

Every user hierarchy has a child count file. An example of a generated user hierarchy child count file name for a table that has the identifier "Table1" and a hierarchy that has the identifier "Geography" is 8.U$Table1$Geography.CHILD_COUNT.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The user hierarchy child count file contains an array of integers. One integer exists in this file for each row in the in-memory tabular structure (section 2.4.4) for the user hierarchy. The first integer that is present in the file corresponds to row 0 of the table, the second integer corresponds to row 1, and so on. The integer value represents the number of child items at the next level for the item in the table row. For example, the item "Canada", row 0, has 2 child items, and the item "United States", row 1, has 4 child items. So the first value in this example file is 2 (the number of child items of the item "Canada"), and the second value is 4 (the number of child items of the item "United States").

The user hierarchy child count file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not XMHybridRLE compression. For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the user hierarchy child count file. An example of a file name for the file that contains the metadata for the user hierarchy child count file is U$Table1$Geography.0.tbl.xml. The metadata for the child count column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "CHILD_COUNT" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.4.1.1 File Layout for User Hierarchy Child Count

The user hierarchy child count file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (section 2.3.1.1.) However, differences exist.

The user hierarchy child count file is a system-generated file that never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that a user hierarchy child count file always has one segment at a minimum and is never associated with a subsegment,

because subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.4.4.2 User Hierarchy First Child Position

Every user hierarchy has a first child position file. An example of a generated user hierarchy first child position file name for a table that has the identifier "Table1" and a hierarchy that has the identifier "Geography" is 8.U$Table1$Geography.FIRST_CHILD_POS.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The user hierarchy first child position file contains an array of integers. One integer exists in this file for each row in the in-memory tabular structure (section 2.4.4) for the user hierarchy. The first integer that is present in the file corresponds to row 0 of the table, the second integer corresponds to row 1, and so on. The integer value is the row number of the row in the in-memory tabular structure that contains the first child of the item on this row. In the example, the first row, row 0, "Canada", has its first child at row 2, "Canada-East", so the first value in the file is 2. The second row of the table, row 1, "United States", has its first child in row 4, "United States-Northeast", so the second value in the file in this example is 4.

The user hierarchy first child position file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not use XMHybridRLE compression. For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the user hierarchy first child position file. An example of a file name for the file that contains the metadata for the user hierarchy first child position file is U$Table1$Geography.0.tbl.xml. The metadata for the first child position column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "FIRST_CHILD_POS" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.4.2.1 File Layout for User Hierarchy First Child Position

The user hierarchy first child position file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (section 2.3.1.1.) However, differences exist.

The user hierarchy first child position file is a system-generated file and never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that a user hierarchy first child position file always has one segment at a minimum and is never associated with a subsegment, because subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.4.4.3 User Hierarchy Multilevel Identifier

Every user hierarchy has a multilevel identifier file. An example of a generated user hierarchy multilevel identifier file name for a table that has the identifier "Table1" and a hierarchy that has the identifier "Geography" is 8.U$Table1$Geography.MULTI_LEVEL_ID.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The user hierarchy multilevel identifier file contains an array of integers. One integer exists in this file for each row in the in-memory tabular structure, which is described in section 2.4.4, for the user

hierarchy. The first integer that is present in the file corresponds to Row 0 of the table, the second integer corresponds to Row 1, and so on.

The integer value in the file is the data identifier value that represents the data value at the lowest level represented on the table row. In the example, Row 0 is "Canada" so the integer value is the data identifier for "Canada" in the "Country/Region" column. Row 1 is "United States" so the next integer value in this file is the data identifier for "United States" in the "Country/Region" column. Row 2 is for "Canada-East" so the item at the lowest level represented in that table row is "East", and the third integer is the data identifier value for the item "East" in the "Area" column.

The user hierarchy multilevel identifier file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not use XMHybridRLE compression. For a discussion of the types of compression that are available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the user hierarchy multilevel identifier file. An example of a file name for the file that contains the metadata for the user hierarchy multilevel identifier file is U$Table1$Geography.0.tbl.xml. The metadata for the multilevel identifier column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "MULTI_LEVEL_ID" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.4.3.1 File Layout for User Hierarchy Multilevel Identifier

The user hierarchy multilevel identifier file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (see section 2.3.1.1.) However, differences exist.

The user hierarchy multilevel identifier file is a system-generated file that never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that a user hierarchy multilevel identifier file always has one segment at a minimum and is never associated with a subsegment, because subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.4.4.4 User Hierarchy Parent Position

Every user hierarchy has a parent position file. An example of a generated user hierarchy parent position file name for a table that has the identifier "Table1" and a hierarchy that has the identifier "Geography" is 8.U$Table1$Geography.PARENT_POS.0.idf. For an explanation of the interpretation of the substrings within the file name, see section 2.2.

The user hierarchy parent position file contains an array of integers. One integer exists in this file for each row in the in-memory tabular structure, which is described in section 2.4.4, for the user hierarchy. The first integer that is present in the file corresponds to Row 0 of the table, the second integer corresponds to Row 1, and so on.

The integer value in the file is the row number in the table that contains the parent item of the item in the table row. In the example, the first two rows are items at the highest level of the user hierarchy. Therefore, neither has a parent item and this fact is represented by the value –1. So the first two values in the file in this example are each –1. The next row in the table has the item "Canada-East". The parent item of "Canada-East" is "Canada". "Canada" is at Row 0 in the table, so in this example, the third value in the file is zero.

The user hierarchy parent position file is compressed by one of several methods, although it is always compressed by using an XMRENoSplit compression method and does not use XMHybridRLE compression. For a discussion of the types of compression available to be used, see section 2.7.

It is necessary to reference the XML metadata file to understand and decode the contents of the user hierarchy parent position file. An example of a file name for the file that contains the metadata for the user hierarchy parent position file is U$Table1$Geography.0.tbl.xml. The metadata for the parent position column is found in the **Columns** collection of the **XMSimpleTable** object in the file. In the **Columns** collection, the **Column** item that has the name "PARENT_POS" contains the metadata for this file. For an explanation of how to interpret the metadata file, see section 2.5.

2.4.4.4.1 File Layout for User Hierarchy Parent Position

The user hierarchy parent position file uses the same .idf file layout as the column data storage .idf file, including the use of segments and segment layout (see section 2.3.1.1.) However, differences exist.

The user hierarchy parent position file is a system-generated file that never uses XMHybridRLE compression but only XMRENoSplit compression. This fact also means that a user hierarchy parent position file always has one segment at a minimum and is never associated with a subsegment, because subsegments are associated only with XMHybridRLE compression, which in turn is used only by column data storage .idf files and the special case of the **RowNumber** column data .idf file.

2.5 Metadata Files

The system stores metadata in XML files. The following sections describe the XML files by using **XML schema definition (XSD)** fragments. For the general XSD fragment for any **XMObject** element, see section 2.5.1. For the XSD fragment for each **XMObject** element based on its specific **class** attribute value, see section 2.5.2. All the types that are referenced in the text in the metadata sections use XML Schema types, as specified in [XMLSCHEMA1] and [XMLSCHEMA2].

2.5.1 XMObject Document Node Element

The table metadata file, the table relationship metadata file, and the column data metadata files all have an **XMObject** element as the document root node. The **XMObject** element is a general element that contains an object that is used to represent objects of many different classes in the XML metadata files.

The general XSD fragment for the **XMObject** element is defined in this section. Because the **XMObject** element is used to represent many different types of objects, its XSD fragment contains a reference to `xs:any` and is therefore very general. Additional sections are provided that provide a more-specific complex type definition for the **XMObject** element when its **class** attribute contains a known value. For the XSD fragment for each **XMObject** element when the class is known, see section 2.5.2.

```
<xs:element name="XMObject" type="XMObjectType"/>

<xs:complexType name="XMObjectType">
  <xs:all>
    <xs:element name="Properties" type="XMObjectPropertiesType" minOccurs="0"/>
    <xs:element name="Members" type="XMObjectMembersType" minOccurs="0"/>
    <xs:element name="Collections" type="XMObjectCollectionsType" minOccurs="0"/>
    <xs:element name="DataObjects" type="XMObjectDataObjectsType" minOccurs="0"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"/>
  <xs:attribute name="name" type="xs:string"/>
```

```
    <xs:attribute name="ProviderVersion" type="xs:int"/>
  </xs:complexType>
```

Properties: An element that contains content described as `xs:any`. Depending on the class of the **XMObject** element, the actual content allowed is constrained, as specified in section 2.5.2.

Members: A collection of **Member** items of type **XMObjectMemberType** for the **XMObject** element. The **Member** items allowed for a specific **XMObject** element are constrained, as specified in section 2.5.2, depending on the value of the **class** attribute.

Collections: A collection of **Collection** items of type **XMObjectCollectionType** for the **XMObject** element. The **Collection** items allowed for a specific **XMObject** element are constrained, as specified in section 2.5.2, depending on the value of the **class** attribute.

DataObjects: A collection of **DataObject** items of type **XMDataObject** for the **XMObject** element. The **DataObject** items allowed in the collection for a particular **XMObject** element are constrained, as specified in section 2.5.2, depending on the value of the **class** attribute.

class: An enumeration value that specifies the class of the **XMObject** element.

name: A string that specifies the object name.

ProviderVersion: An integer that specifies the version of the provider that wrote the object.

2.5.1.1 XMObjectPropertiesType

The **XMObjectPropertiesType** complex type holds the properties for an instance of an **XMObject** object.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectPropertiesType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectPropertiesType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectPropertiesType">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

xs:any: Any element as content.

2.5.1.2 XMObjectMembersType

The **XMObjectMembersType** complex type holds a collection of **Member** items for an instance of an **XMObject** object. Each **Member** item represents a property of the **XMObject** instance, but a **Member** item can contain complex content, whereas **XMObjectPropertiesType** (section 2.5.1.1) holds elements of a simple type.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectMembersType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type

XMObjectMembersType is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectMembersType">
  <xs:sequence>
    <xs:element name="Member" type="XMObjectMemberType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Member: A property of the **XMObject** instance that can contain complex content. In the general case, any **Member** can be contained in the collection of **Member** objects. However, the content of specific instances of the **XMObjectMembersType** type is constrained depending on the value of the **class** attribute of the containing **XMObject** element.

2.5.1.3 XMObjectCollectionsType

The **XMObjectCollectionsType** complex type holds collections of complex properties that pertain to the parent **XMObject** instance. Each **Collection** item represents a property of the **XMObject** instance. The collection can contain multiple instances of the same **Collection** item, and each instance can contain complex content. An example is a table, which can contain multiple column items in a collection.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectCollectionsType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectCollectionsType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectCollectionsType">
  <xs:sequence>
    <xs:element name="Collection" type="XMObjectCollectionType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Collection: A property of the **XMObject** instance, which can contain multiple instances of the same property, and in which each instance can contain complex content. In the general case, any **Collection** can be contained in the collection of **Collection** objects. However, the content of specific instances of the **XMObjectCollectionsType** type is constrained depending on the value of the **class** attribute of the containing **XMObject** element.

2.5.1.4 XMObjectDataObjectsType

The **XMObjectDataObjectsType** complex type holds data objects for the parent **XMObject** instance. Each **DataObject** object in the collection represents a data object of the parent **XMObject** instance.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectDataObjectsType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectDataObjectsType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectDataObjectsType">
```

```
    <xs:sequence>
      <xs:element name="DataObject" type="XMObjectDataObjectType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

DataObject: A data object that is related to the **XMObject** instance. In the general case, any **DataObject** can be contained in the collection of **DataObjects**. However, the content of specific instances of the **XMObjectDataObjectsType** type are constrained depending on the value of the **class** attribute of the containing **XMObject** element.

2.5.1.5 XMObjectMemberType

The **XMObjectMemberType** complex type holds properties for one **Member** item of a parent **XMObject** instance. The properties for the member are held within another instance of an **XMObject** element that is nested within the **Member** element.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectMemberType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectMemberType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectMemberType">
  <xs:all>
    <xs:element name="Name" type="XMObjectMemberNameEnum" />
    <xs:element name="XMObject" type="XMObjectType" minOccurs="0"/>
  </xs:all>
</xs:complexType>
```

Name: An enumeration value that specifies the name of the **Member** item.

XMObject: A nested instance of an **XMObject** element. The **XMObject** element contains the properties, members, collections, and data objects for the **Member** instance that has the name specified by the **Name** element.

2.5.1.6 XMObjectCollectionType

The **XMObjectCollectionType** complex type holds the data for one collection item of a parent **XMObject** instance. The properties for each item in the collection are held within another instance of an **XMObject** element that is nested within the **Collections** element.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectCollectionType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectCollectionType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectCollectionType">
  <xs:sequence>
    <xs:element name="Name" type="XMObjectCollectionNameEnum" />
    <xs:element name="XMObject" type="mstns:XMObjectType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
```

```
</xs:complexType>
```

Name: An enumeration value that specifies the name of the **Collection** item.

XMObject: A nested instance of an **XMObject** element. The **XMObject** element contains the properties, members, collections, and data objects for the **Collection** object that has the name specified by the **Name** element.

2.5.1.7 XMObjectDataObjectType

The **XMObjectDataObjectType** complex type holds the data for one data object item in the collection of data objects for the parent **XMObject** instance. The properties for each item in the collection are held within another instance of an **XMObject** element that is nested within the **DataObject** element.

The XSD fragment presented in this section is general and covers the definition of all element instances of type **XMObjectDataObjectType**. However, when the **class** attribute value of the containing **XMObject** element is known, the content of an element of type **XMObjectDataObjectType** is more constrained than indicated by the definition contained in this section. For the constrained definitions, see section 2.5.2.

```
<xs:complexType name="XMObjectDataObjectType">
  <xs:all>
    <xs:element name="XMObject" type="XMObjectType"/>
  </xs:all>
</xs:complexType>
```

XMObject: A nested instance of an **XMObject** element, which contains the properties, members, collections, and data objects for a data object item in the collection of data objects.

2.5.1.8 XMObjectMemberNameEnum

The **XMObjectMemberNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Members** collection of an **XMObject** object.

```
<xs:simpleType name="XMObjectMemberNameEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="SegmentMap"/>
    <xs:enumeration value="TableStats"/>
    <xs:enumeration value="ColumnStats"/>
    <xs:enumeration value="SubSegment"/>
    <xs:enumeration value="CompressionInfo"/>
    <xs:enumeration value="ColumnSegmentStats"/>
    <xs:enumeration value="IntrinsicHierarchy"/>
    <xs:enumeration value="SubCompression"/>
    <xs:enumeration value="RLECompression"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMObjectMemberNameEnum** type.

Enumeration value	Description
"SegmentMap"	The member contains a **segment map**.
"TableStats"	The member contains table statistics.
"ColumnStats"	The member contains column statistics.
"SubSegment"	The member contains information for a subsegment.
"CompressionInfo"	The member contains information about compression.
"ColumnSegmentStats"	The member contains column segment statistics.
"IntrinsicHierarchy"	The member contains information about an **intrinsic hierarchy**.
"SubCompression"	The member contains information about subcompression.
"RLECompression"	The member contains information about RLE compression.

2.5.1.9 XMObjectCollectionNameEnum

The **XMObjectCollectionNameEnum** simple type enumerates the allowed values for the name of a **Collection** item in the **Collections** collection of an **XMObject** object.

```
<xs:simpleType name="XMObjectCollectionNameEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Columns"/>
    <xs:enumeration value="Segments"/>
    <xs:enumeration value="Partitions"/>
    <xs:enumeration value="Relationships"/>
    <xs:enumeration value="UserHierarchies"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMObjectCollectionNameEnum** type.

Enumeration value	Description
"Columns"	The **Collection** item contains a collection of **Column** items. Each **Column** item represents a column in the source data table.
"Segments"	The **Collection** item contains a collection of **Segment** items. Each **Segment** item represents a segment of the rows of a column in the source data.
"Partitions"	The **Collection** item contains a collection of **Partition** items. Only one partition is supported.
"Relationships"	The **Collection** item contains a collection of **Relationship** items. Each **Relationship** item describes a join relationship.
"UserHierarchies"	The **Collection** item contains a collection of user-defined hierarchies. Each **UserHierarchy** item describes a user-defined hierarchy.

2.5.1.10 XMObjectClassNameEnum

The **XMObjectClassNameEnum** simple type enumerates the allowed values for the **class** attribute of an **XMObject** element. The content allowed in an instance of an **XMObject** element depends on the value of this attribute.

```
<xs:simpleType name="XMObjectClassNameEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="XMSimpleTable"/>
    <xs:enumeration value="XMRawColumn"/>
    <xs:enumeration value="XMRelationship"/>
    <xs:enumeration value="XMRelationshipIndexSparseDIDs"/>
    <xs:enumeration value="XMRelationshipIndexDenseDIDs"/>
    <xs:enumeration value="XMHierarchy"/>
    <xs:enumeration value="XMUserHierarchy"/>
    <xs:enumeration value="XMHierarchyDataID2PositionHashIndex"/>
    <xs:enumeration value="XMColumnSegment"/>
    <xs:enumeration value="XMPartition"/>
    <xs:enumeration value="XMMultiPartSegmentMap"/>
    <xs:enumeration value="XMSegment1Map"/>
    <xs:enumeration value=
     "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_FastInstantiation>"/>
    <xs:enumeration value=
     "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_ComplexInstantiation>"/>
    <xs:enumeration value="XMTableStats"/>
    <xs:enumeration value="XMColumnSegmentStats"/>
    <xs:enumeration value="XMColumnStats"/>
    <xs:enumeration value="XMValueDataDictionary&lt;XM_Long>"/>
    <xs:enumeration value="XMValueDataDictionary&lt;XM_Real>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_Real>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_Long>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_String>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XMVariantPtr>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;1>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;2>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;3>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;4>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;5>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;6>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;7>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;8>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;9>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;10>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;12>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;16>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;21>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;32>"/>
    <xs:enumeration value="XM123CompressionInfo"/>
    <xs:enumeration value=
     "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;1>>"
                  />
    <xs:enumeration value=
     "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"
                  />
    <xs:enumeration value=
     "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;3>>"
                  />
    <xs:enumeration value=
     "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;4>>"
```

```
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;5>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;6>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;7>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;8>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;9>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;10>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;12>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;16>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;21>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;32>>"
                    />
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XM123CompressionInfo>"/>
      <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMREGeneralCompressionInfo>"/>
      <xs:enumeration value="XMRawColumnPartitionDataObject"/>
      <xs:enumeration value="XMRLECompressionInfo"/>
      <xs:enumeration value="XMRLEGeneralCompressionInfo"/>
      <xs:enumeration value="XMColumnSegmentDataObject"/>
      <xs:enumeration value="XMRelationshipIndex123DIDs"/>
    </xs:restriction>
  </xs:simpleType>
```

The following table describes the enumeration values in the **XMObjectClassNameEnum** type.

Enumeration value	Description
"XMSimpleTable"	The object specifies the metadata for a table.
"XMRawColumn"	The object specifies the metadata for a column.
"XMRelationship"	The object specifies the metadata for a relationship between two tables.
"XMRelationshipIndexSparseDIDs"	The object specifies the

Enumeration value	Description
	metadata for a relationship index in which the data identifiers are sparse.
"XMRelationshipDenseDIDs"	The object specifies the metadata for a relationship in which the data identifiers are dense.
"XMHierarchy"	The object specifies the metadata for a hierarchy.
"XMUserHierarchy"	The object specifies the metadata for a user hierarchy.
"XMHierarchyDataID2PositionHashIndex"	The object specifies the metadata for a hash index of data identifier–to–position mapping.
"XMColumnSegment"	The object specifies the metadata for a column segment.
"XMPartition"	The object specifies the metadata for a partition.
"XMultiPartSegmentMap"	The object specifies the metadata for a segment map associated with partitions.
"XMSegment1Map"	The object specifies the metadata for a segment map for a column with a single segment.
"XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation"	The object specifies the metadata for a segment map of equally sized segments (except that the size of the last segment can differ from that of the others). Note that fast instantiation is for predetermined segment sizes.
"XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation"	The object specifies the metadata for a segment map of equally sized segments (except that the size of the last segment can differ from that of the others). Note that complex instantiation occurs when the segment size is determined at run time.
"XMTableStats"	The object specifies the metadata for table statistics.
"XMColumnStats"	The object specifies the

Enumeration value	Description
	metadata for column statistics.
"XMColumnSegmentStats"	The object specifies the metadata for column segment statistics.
"XMValueDataDictionary<XM_Long>"	The object specifies the metadata for a value dictionary for values of type **long**.
"XMValueDataDictionary<XM_Real>"	The object specifies the metadata for a value dictionary for values of type **real**.
"XMHashDataDictionary<XM_Real>"	The object specifies the metadata for a hash dictionary for values of type **real**.
"XMHashDataDictionary<XM_Long>"	The object specifies the metadata for a hash dictionary for values of type **long**.
"XMHashDataDictionary<XM_String>"	The object specifies the metadata for a hash dictionary for values of type **string**.
"XMRENoSplitCompressionInfo<1>"	The object specifies the metadata for NoSplit compression of 1 bit in length.
"XMRENoSplitCompressionInfo<2>"	The object specifies the metadata for NoSplit compression of 2 bits in length.
"XMRENoSplitCompressionInfo<3>"	The object specifies the metadata for NoSplit compression of 3 bits in length.
"XMRENoSplitCompressionInfo<4>"	The object specifies the metadata for NoSplit compression of 4 bits in length.
"XMRENoSplitCompressionInfo<5>"	The object specifies the metadata for NoSplit compression of 5 bits in length.
"XMRENoSplitCompressionInfo<6>"	The object specifies the metadata for NoSplit compression of 6 bits in length.
"XMRENoSplitCompressionInfo<7>"	The object specifies the metadata for NoSplit compression of 7 bits in length.
"XMRENoSplitCompressionInfo<8>"	The object specifies the metadata for NoSplit compression of 8 bits in length.
"XMRENoSplitCompressionInfo<9>"	The object specifies the metadata for NoSplit

Enumeration value	Description
	compression of 9 bits in length.
"XMRENoSplitCompressionInfo<10>"	The object specifies the metadata for NoSplit compression of 10 bits in length.
"XMRENoSplitCompressionInfo<12>"	The object specifies the metadata for NoSplit compression of 12 bits in length.
"XMRENoSplitCompressionInfo<16>"	The object specifies the metadata for NoSplit compression of 16 bits in length.
"XMRENoSplitCompressionInfo<21>"	The object specifies the metadata for NoSplit compression of 21 bits in length.
"XMRENoSplitCompressionInfo<32>"	The object specifies the metadata for NoSplit compression of 32 bits in length.
"XM123CompressionInfo"	The object specifies the metadata for 123 compression.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>"	The object specifies the metadata for hybrid NoSplit compression of 1 bit in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>"	The object specifies the metadata for hybrid NoSplit compression of 2 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>"	The object specifies the metadata for hybrid NoSplit compression of 3 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>"	The object specifies the metadata for hybrid NoSplit compression of 4 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>"	The object specifies the metadata for hybrid NoSplit compression of 5 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>"	The object specifies the metadata for hybrid NoSplit compression of 6 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>"	The object specifies the metadata for hybrid NoSplit compression of 7 bits in length.
"XMHybridRLECompressionInfo<class	The object specifies the metadata for hybrid NoSplit

Enumeration value	Description
XMRENoSplitCompressionInfo<8>>"	compression of 8 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>"	The object specifies the metadata for hybrid NoSplit compression of 9 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>"	The object specifies the metadata for hybrid NoSplit compression of 10 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>"	The object specifies the metadata for hybrid NoSplit compression of 12 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>"	The object specifies the metadata for hybrid NoSplit compression of 16 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>"	The object specifies the metadata for hybrid NoSplit compression of 21 bits in length.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>"	The object specifies the metadata for hybrid NoSplit compression of 32 bits in length.
"XMHybridRLECompressionInfo<class XMRE123CompressionInfo>"	The object specifies the metadata for hybrid 123 compression.
"XMHybridRLECompressionInfo<class XMREGeneralCompressionInfo>"	The object specifies the metadata for hybrid General compression.
"XMRawColumnPartitionDataObject"	The object specifies the metadata for a partition.
"XMRLECompressionInfo"	The object specifies the metadata for RLE compression.
"XMColumnSegmentDataObject"	The object specifies the metadata for a data object for a column segment.
"XMRLEGeneralCompressionInfo"	The object specifies the metadata for general RLE compression.

2.5.2 XMObject Definitions by class Attribute

The general definition of the **XMObject** element is specified in section 2.5.1. This section contains more-specific definitions for the content of an **XMObject** element, according to each available value of the **class** attribute on the **XMObject** element.

2.5.2.1 XMObject class="XMSimpleTable"

When the **class** attribute value for the **XMObject** element is "XMSimpleTable", the **XMObject** element contains the metadata for a table object, and the type of the **XMObject** element is **XMSimpleTableXMObjectType**.

```
<xs:complexType name="XMSimpleTableXMObjectType">
  <xs:all>
    <xs:element name="Properties" type="XMSimpleTablePropertiesType"/>
    <xs:element name="Members" type="XMSimpleTableMembersType"/>
    <xs:element name="Collections" type="XMSimpleTableCollectionsType"/>
  </xs:all>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMSimpleTable">
  </xs:attribute>
</xs:complexType>
```

Properties: The property values for the **XMSimpleTable** object.

Members: A collection of **Member** complex type items, each of which contains a complex property for the **XMSimpleTable** object.

Collections: A collection of **Collection** complex type items, each of which contains a complex property for the **XMSimpleTable** object. The **Collection** complex property can be repeated multiple times.

name: The name of the **XMSimpleTable** object instance.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.1.1 XMSimpleTablePropertiesType

The **XMSimpleTablePropertiesType** complex type holds the specific properties that are allowed when the **XMObject** element is of class "XMSimpleTable".

```
<xs:complexType name="XMSimpleTablePropertiesType">
  <xs:all>
    <xs:element name="Version" type="xs:int"/>
    <xs:element name="Settings">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="4367"/>
```

```
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="RIViolationCount" type="xs:long"/>
    </xs:all>
  </xs:complexType>
```

Version: The internal version number for this data. This version number is not required to match the version numbers of other objects within the same table or column.

Settings: A mask that describes settings for the table. The bits that are listed in the following table can be set.

Bit	Meaning
0x1	The table contains normal data.
0x2	The table contains an index.
0x3	The table has temporary content.
0x4	The table has an intrinsic hierarchy.
0x5	The table has a user hierarchy.
0x100	The table has been processed.
0x1000	The table uses an automatic NULL for an unknown member.

RIViolationCount: The number of relational integrity violations in the **XMSimpleTable** object.

2.5.2.1.2 XMSimpleTableMembersType

The **XMSimpleTableMembersType** complex type holds a collection of **Member** items, each of which contains a property for the parent **XMSimpleTable** object.

```
<xs:complexType name="XMSimpleTableMembersType">
  <xs:sequence>
    <xs:element name="Member" type="XMSimpleTableMemberType"
                minOccurs="2" maxOccurs="2"/>
  </xs:sequence>
</xs:complexType>
```

Member: A complex type element that contains a property for the parent **XMObject** element of class "XMSimpleTable". The value of the **Name** element for the two instances of this element in the **Members** collection MUST have one instance of each enumeration value from the **XMSimpleTableMemberNameEnum** type (section 2.5.2.1.2.2).

2.5.2.1.2.1 XMSimpleTableMemberType

The **XMSimpleTableMemberType** complex type holds a **Member** item, which contains a property of the parent **XMSimpleTable** object.

```
<xs:complexType name="XMSimpleTableMemberType">
  <xs:sequence>
```

```
      <xs:element name="Name" type="XMSimpleTableMemberNameEnum"
                  minOccurs="0"/>
      <xs:element name="XMObject">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="XMObjectTypeBase">
              <xs:attribute name="class"
                  type="XMSimpleTableXMObjectMemberClassNameEnum"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Name: The name of the **Member** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value.

class: An enumeration value that specifies the class name of this **XMObject** element instance. When the **Name** element of the **Member** item has a particular value, the **XMObject** element of the **Member** item MUST have a specific value for the **class** attribute. The following table lists the constraints between the values of **Name** and **class**.

Value of Name element	Required value of class attribute
"SegmentMap"	One of the following: ▪ "XMMultiPartSegmentMap" ▪ "XMSegment1Map" ▪ "XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>"
"TableStats"	"XMTableStats"

2.5.2.1.2.2 XMSimpleTableMemberNameEnum

The **XMSimpleTableMemberNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Members** collection of an **XMSimpleTable** object.

```
<xs:simpleType name="XMSimpleTableMemberNameEnum">
  <xs:restriction base="XMObjectMemberNameEnum">
    <xs:enumeration value="SegmentMap"/>
    <xs:enumeration value="TableStats"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMSimpleTableMemberNameEnum** type.

Enumeration value	Description
"SegmentMap"	The **Member** item is a segment map.
"TableStats"	The **Member** item contains table statistics.

2.5.2.1.2.3 XMSimpleTableXMObjectMemberClassNameEnum

The **XMSimpleTableXMObjectMemberClassNameEnum** simple type enumerates the allowed values for the class name of the **XMObject** element that is contained in a **Member** item in the **Members** collection of an **XMSimpleTable** object.

```
<xs:simpleType name="XMSimpleTableXMObjectMemberClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMMultiPartSegmentMap"/>
    <xs:enumeration value="XMSegment1Map"/>
    <xs:enumeration value=
       "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_ComplexInstantiation>"/>
    <xs:enumeration value=
       "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_FastInstantiation>"/>
    <xs:enumeration value="XMTableStats"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMSimpleTableXMObjectMemberClassNameEnum** type.

Enumeration value	Description
"XMMultiPartSegmentMap"	The object contains a multipart segment map, which is a segment map that is built on top of other segment maps.
"XMSegment1Map"	The object contains a segment map for a column that has just one segment.
"XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>"	The object contains a segment map for equally sized segments. Complex instantiation occurs when the actual segment size is determined at run time.
"XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>"	The object contains a segment map for equally sized segments. Fast instantiation occurs when the actual segment size is determined at creation time.
"XMTableStats"	The object contains table statistics.

2.5.2.1.3 XMSimpleTableCollectionsType

The **XMSimpleTableCollectionsType** complex type holds a collection of **Collection** items, each of which contains a property for the parent **XMSimpleTable** object. **Collection** items can be repeated multiple times within the collection.

```
<xs:complexType name="XMSimpleTableCollectionsType">
  <xs:sequence>
    <xs:element name="Collection" type="XMSimpleTableCollectionType"
                minOccurs="4" maxOccurs="4"/>
  </xs:sequence>
</xs:complexType>
```

Collection: A complex type element that contains a **Collection** item, which contains a property for the parent **XMObject** element of class **XMSimpleTable**. **Collection** items can be repeated within the collection.

The value of the **Name** element for the four instances of this element in the **Collections** collection MUST have one instance of each enumeration value from the **XMSimpleTableCollectionNameEnum** type (section 2.5.2.1.3.2).

2.5.2.1.3.1 XMSimpleTableCollectionType

The **XMSimpleTableCollectionType** complex type holds a **Collection** item, which contains a property of the parent **XMSimpleTable** object.

```
<xs:complexType name="XMSimpleTableCollectionType">
  <xs:sequence>
    <xs:element name="Name" type="XMSimpleTableCollectionNameEnum"/>
    <xs:element name="XMObject" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="XMObjectTypeBase">
            <xs:attribute name="class"
             type="XMSimpleTableXMObjectCollectionClassNameEnum"/>
            <xs:attribute name="name" type="xs:string"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Name: The name of the **Collection** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content that is allowed depends on the value of the **class** attribute of the **XMObject** element instance.

The following attributes are added by extension for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element. When the **Name** element of the **Collection** item has a particular value, the **XMObject** element of the **Collection** item MUST have a specific value for the **class** attribute. The following table lists the constraints between the values of **Name** and **class**.

Value of Name element	Required value of class attribute
"Partitions"	"XMPartition"
"Columns"	"XMRawColumn"
"Relationships"	"XMRelationship"
"UserHierarchies"	"XMUserHierarchy"

name: The name of the **Collection** item.

2.5.2.1.3.2 XMSimpleTableCollectionNameEnum

The **XMSimpleTableCollectionNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Collections** collection of an **XMSimpleTable** object.

```
<xs:simpleType name="XMSimpleTableCollectionNameEnum">
  <xs:restriction base="XMObjectCollectionNameEnum">
    <xs:enumeration value="Partitions"/>
    <xs:enumeration value="Columns"/>
    <xs:enumeration value="Relationships"/>
    <xs:enumeration value="UserHierarchies"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMSimpleTableCollectionNameEnum** type.

Enumeration value	Description
"Partitions"	**Collection** is a collection of items that pertain to each partition.
"Columns"	**Collection** is a collection of items that pertain to each column.
"Relationships"	**Collection** is a collection of items that pertain to relationships for the table.
"UserHierarchies"	**Collection** is a collection of items that pertain to user-defined hierarchies.

2.5.2.1.3.3 XMSimpleTableXMObjectCollectionClassNameEnum

The **XMSimpleTableXMObjectCollectionClassNameEnum** simple type enumerates the allowed values for the class name of the **XMObject** element that is contained in a **Collection** item in the **Collections** collection of an **XMSimpleTable** object.

```
<xs:simpleType name="XMSimpleTableXMObjectCollectionClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMPartition"/>
    <xs:enumeration value="XMRawColumn"/>
    <xs:enumeration value="XMRelationship"/>
    <xs:enumeration value="XMUserHierarchy"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMSimpleTableXMObjectCollectionClassNameEnum** type.

Enumeration value	Description
"XMPartition"	The **XMObject** element contains a partition.
"XMRawColumn"	The **XMObject** element contains information about a column.
"XMRelationship"	The **XMObject** element contains information about a table relationship.
"XMUserHierarchy"	The **XMObject** element contains information about user-defined hierarchies.

2.5.2.2 XMObject class="XMTableStats"

When the **class** attribute value for the **XMObject** element is "XMTableStats", the **XMObject** element contains statistical information about the table, and the type of the **XMObject** element is **XMTableStatsXMObjectType**.

```
<xs:complexType name="XMTableStatsXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMTableStatsPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMTableStats"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.2.1 XMTableStatsPropertiesType

The **XMTableStatsPropertiesType** simple type contains the specific properties that are allowed when the **XMObject** element is of class "XMTableStats".

```
<xs:complexType name="XMTableStatsPropertiesType">
  <xs:all>
    <xs:element name="SegmentSize" type="xs:long"/>
    <xs:element name="Usage">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="2"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

SegmentSize: The number of rows in the segment.

Usage: An enumeration value for the usage of the table. One of the values in the following table can be set.

Value	Meaning
0	The table usage is unknown.
1	The table is a dimension table.
2	The table is a fact table.

2.5.2.3 XMObject class="XMRawColumn"

When the **class** attribute value for the **XMObject** element is "XMRawColumn", the **XMObject** element contains the metadata for a raw data column, and the type of the **XMObject** element is **XMRawColumnXMObjectType**.

```
<xs:complexType name="XMRawColumnXMObjectType">
  <xs:all>
    <xs:element name="Properties" type="XMRawColumnPropertiesType"/>
    <xs:element name="Members" type="XMRawColumnMembersType"/>
    <xs:element name="Collections" type="XMRawColumnCollectionsType"/>
    <xs:element name="DataObjects" type="XMRawColumnDataObjectsType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRawColumn"/>
</xs:complexType>
```

Properties: A complex type that specifies the property values for the **XMRawColumn** object.

Members: A collection of **Member** complex type items, each of which contains a complex property for the **XMRawColumn** object.

Collections: A collection of **Collection** complex type items, each of which contains a complex property for the **XMRawColumn** object. The **Collection** complex property can be repeated multiple times.

DataObjects: A collection of **DataObject** complex type items, each of which contains an object with information about the column's data.

name: The name of the **XMRawColumn** object instance.

class: An enumeration value that specifies the class name of this **XMObject** element.

ProviderVersion: The provider version.

2.5.2.3.1 XMRawColumnPropertiesType

The **XMRawColumnPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMRawColumn".

```
<xs:complexType name="XMRawColumnPropertiesType">
  <xs:all>
    <xs:element name="Settings">
```

```
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="7994"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="ColumnFlags">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="63"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="Collation" type="xs:string"/>
    <xs:element name="OrderByColumn" type="xs:string"/>
    <xs:element name="Locale" type="xs:long"/>
    <xs:element name="BinaryCharacters" type="xs:unsignedInt"/>
  </xs:all>
</xs:complexType>
```

Settings: The settings for the column. The low order 5 bits (values 0x0 through 0x11) contain the column type. The remainder of the values can be combined with the column type.

The following table describes the values for the column type.

Value	Meaning
0	The column has no settings.
0x1	The column contains basic data.
0x2	The column contains calculated data.
0x3	The column contains a relationship line number.

The following table describes the remainder of the values.

Value	Meaning
0x5	The column contains data identifier–to–position mapping for a hierarchy.
0x7	The column contains position–to–data identifier mapping for a hierarchy.
0x8	The column contains the position of the parent item within the parent item's own level in a hierarchy.
0x9	The column contains the position of the child item within the child item's own level in a hierarchy.
0x10	The column contains a data identifier within a merged multilevel user hierarchy.
0x11	The column contains a count of the child items within a merged multilevel hierarchy.
0x100	NULLs are converted to zeros or spaces.

Value	Meaning
0x200	The column is trimmed on the left.
0x400	The column is trimmed on the right.
0x800	The column is a calculated column.
0x1000	The column needs defragmentation.

ColumnFlags: The flags for the properties of the column. One or more of the values in the following table can be set. The value 0x8 MUST be set.

Value	Meaning
0	No flags are set.
0x1	The column cannot be NULL.
0x2	A constraint exists specifying that all the values in the column MUST be unique.
0x4	The column is the primary key for the table.
0x8	The column has an intrinsic hierarchy.
0x10	The column contains row numbers.
0x20	The column has an unsorted hierarchy.

Collation: The name of the collation. The value MAY<7> be restricted to a string that is recognized as valid by the system.

OrderByColumn: The column by which to order the hierarchy.

Locale: An LCID that specifies the locale.

BinaryCharacters: The maximum number of characters in a string that uses base64 encoding and that represents a BLOB.

2.5.2.3.2 XMRawColumnMembersType

The **XMRawColumnMembersType** complex type holds a collection of **Member** items, each of which contains a property for the parent **XMSimpleTable** object.

```
<xs:complexType name="XMSimpleTableMembersType">
  <xs:sequence>
    <xs:element name="Member" type="XMSimpleTableMemberType"
                minOccurs="2" maxOccurs="2"/>
  </xs:sequence>
</xs:complexType>
```

Member: A property for the parent **XMObject** element of class **XMRawColumn**. The value of the **Name** element for the two instances of this element in the **Members** collection MUST have one instance of each enumeration value from the **XMRawColumnMemberNameEnum** type (section 2.5.2.3.2.2).

2.5.2.3.2.1 XMRawColumnMemberType

The **XMRawColumnMemberType** complex type holds a **Member** item that is a property of the parent **XMRawTable** object.

```
<xs:complexType name="XMRawColumnMemberType">
  <xs:sequence>
    <xs:element name="Name" type="XMRawColumnMemberNameEnum"
                minOccurs="0"/>
    <xs:element name="XMObject">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="XMObjectTypeBase">
            <xs:attribute name="class"
                type="XMRawColumnXMObjectMemberClassNameEnum"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Name: The name of the **Member** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value

The following attribute is added by extension for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element instance. When the **Name** element of the **Member** item has a particular value, the **XMObject** element of the **Member** item MUST have a specific value for the **class** attribute. The following table lists the constraints between the values of **Name** and **class**.

Value of Name element	Required value of class attribute
"IntrinsicHierarchy"	"XMHierarchy"
"TableStats"	"XMTableStats"

2.5.2.3.2.2 XMRawColumnMemberNameEnum

The **XMRawColumnMemberNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Members** collection of an **XMRawColumn** object.

```
<xs:simpleType name="XMRawColumnMemberNameEnum">
  <xs:restriction base="XMObjectMemberNameEnum">
    <xs:enumeration value="IntrinsicHierarchy"/>
    <xs:enumeration value="ColumnStats"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMRawColumnMemberNameEnum** type.

Enumeration value	Description
"IntrinsicHierarchy"	The **Member** item represents the intrinsic hierarchy of a column.
"ColumnStats"	The **Member** item contains column statistics.

2.5.2.3.2.3 XMRawColumnXMObjectMemberClassNameEnum

The **XMRawColumnXMObjectMemberClassNameEnum** simple type enumerates the allowed values for the class name of the **XMObject** element that is contained in a **Member** item in the **Members** collection of an **XMRawColumn** object.

```
<xs:simpleType name="XMRawColumnXMObjectMemberClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMHierarchy"/>
    <xs:enumeration value="XMColumnStats"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMRawColumnXMObjectMemberClassNameEnum** type.

Enumeration value	Description
"XMHierarchy"	The **XMObject** element specifies information about the hierarchy.
"XMColumnStats"	The **XMObject** element specifies statistics about the column.

2.5.2.3.3 XMRawColumnCollectionsType

The **XMRawColumnCollectionsType** complex type holds a collection of **Collection** items, each of which contains a property for the parent **XMRawColumns** object. **Collection** items can be repeated multiple times within the collection.

```
<xs:complexType name="XMRawColumnCollectionsType">
  <xs:sequence>
    <xs:element name="Collection" type="XMRawColumnCollectionType"
                minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

Collection: A **Collection** item that is a property for the parent **XMObject** element of class **XMRawColumn**. **Collection** items can be repeated within the collection.

2.5.2.3.3.1 XMRawColumnCollectionType

The **XMRawColumnCollectionType** complex type holds a **Collection** item that is a property of the parent **XMObject** object.

```
<xs:complexType name="XMRawColumnCollectionType">
```

```
    <xs:sequence>
      <xs:element name="Name" type="XMObjectCollectionNameEnum"
                  fixed="Segments"/>
      <xs:element name="XMObject" type="XMColumnSegmentXMObjectType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

Name: The name of the **Collection** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The value of the **class** attribute of this **XMObject** element MUST equal "XMColumnSegment".

2.5.2.3.4 XMRawColumnDataObjectsType

The **XMRawColumnDataObjectsType** complex type holds data objects for the parent **XMRawColumn** object instance.

```
<xs:complexType name="XMRawColumnDataObjectsType">
  <xs:sequence>
    <xs:element name="DataObject" type="XMRawColumnDataObjectType"
                minOccurs="2" maxOccurs="2"/>
  </xs:sequence>
</xs:complexType>
```

DataObject: A data object that holds information related to the data in the raw column. The two instances of the **DataObject** element in the **DataObjects** collection MUST abide by the following rules:

- The value of the **class** attribute for one of the **DataObject** elements MUST be "XMRawColumnPartitionDataObject".
- The value of the **class** attribute for one of the **DataObject** elements MUST be one of the following:
  - "XMValueDataDictionary<XM_Long>"
  - "XMValueDataDictionary<XM_Real>"
  - "XMHashDataDictionary<XM_Long>"
  - "XMHashDataDictionary<XM_Real>"
  - "XMHashDataDictionary<XM_String>"

2.5.2.3.4.1 XMRawColumnDataObjectType

The **XMRawColumnDataObjectType** complex type holds the data for one data object item in the collection of data objects for the parent **XMObject** element.

```
<xs:complexType name="XMRawColumnDataObjectType">
  <xs:sequence>
    <xs:element name="XMObject">
      <xs:complexType>
        <xs:complexContent>
```

```
            <xs:extension base="XMObjectTypeBase">
              <xs:attribute name="class"
                  type="XMRawColumnXMObjectDataObjectClassNameEnum"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content that is allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value.

class: An enumeration value that specifies the class name of this **XMObject** element instance.

2.5.2.3.4.2 XMRawColumnXMObjectDataClassNameEnum

The **XMRawColumnXMObjectDataObjectClassNameEnum** simple type enumerates the allowed values for the **class** attribute of the **XMObject** element that is contained in a **DataObject** item in the **DataObjects** collection of an **XMRawColumn** object.

```
<xs:simpleType name="XMRawColumnXMObjectDataObjectClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMValueDataDictionary&lt;XM_Long>"/>
    <xs:enumeration value="XMValueDataDictionary&lt;XM_Real>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_Real>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_Long>"/>
    <xs:enumeration value="XMHashDataDictionary&lt;XM_String>"/>
    <xs:enumeration value="XMRawColumnPartitionDataObject"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMRawColumnXMObjectDataObjectClassNameEnum** type.

Enumeration value	Description
"XMValueDataDictionary<XM_Long>"	The column has a value data dictionary of type **long**.
"XMValueDataDictionary<XM_Real>"	The column has a value data dictionary of type **real**.
"XMHashDataDictionary<XM_Long>"	The column has a hash data dictionary of type **long**.
"XMHashDataDictionary<XM_Real>"	The column has a hash data dictionary of type **real**.
"XMHashDataDictionary<XM_String>"	The column has a hash data dictionary of type **string**.
"XMRawColumnPartitionDataObject"	The **XMObject** specifies information about the partition for the raw column.

2.5.2.4 XMObject class="XMRelationship"

When the **class** attribute value for the **XMObject** element is "XMRelationship", the **XMObject** element contains the metadata for a relationship description, and the type of the **XMObject** element is **XMRelationshipXMObjectType**.

```
<xs:complexType name="XMRelationshipXMObjectType">
  <xs:all>
    <xs:element name="Properties" type="XMRelationshipPropertiesType"/>
    <xs:element name="DataObjects" type="XMRelationshipDataObjectsType"/>
  </xs:all>
<xs:attribute name="ProviderVersion" type="xs:int"/>
<xs:attribute name="name" type="xs:string"/>
<xs:attribute name="class" type="XMObjectClassNameEnum"
              fixed="XMRelationship"/>
</xs:complexType>
```

Properties: The property values for the **XMRawColumn** object.

DataObjects: A collection of **DataObject** complex type items, each of which contains an object that has information about the column's data.

ProviderVersion: The provider version.

name: The name of the **XMRelationshipXMObject** instance.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.4.1 XMRelationshipPropertiesType

The **XMRelationshipPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class " XMRelationship".

```
<xs:complexType name="XMRelationshipPropertiesType">
  <xs:all>
    <xs:element name="PrimaryTable" type="xs:string"/>
    <xs:element name="PrimaryColumn" type="xs:string"/>
    <xs:element name="ForeignColumn" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

PrimaryTable: The name of the table in which the primary key column exists.

PrimaryColumn: The primary key column.

ForeignColumn: The foreign key column.

2.5.2.4.2 XMRelationshipDataObjectsType

The **XMRelationshipsDataObjectsType** complex holds data objects for the parent **XMRelationship** object instance.

```
<xs:complexType name="XMRelationshipDataObjectsType">
  <xs:all>
    <xs:element name="DataObject" type="XMRelationshipDataObjectType"/>
```

```
    </xs:all>
  </xs:complexType>
```

DataObject: A data object item that holds information related to the data for the relationship.

2.5.2.4.3 XMRelationshipDataObjectType

The **XMRelationshipObjectType** complex type holds the data for one data object item in the collection of data objects for the parent **XMObject** element.

```
<xs:complexType name="XMRelationshipDataObjectType">
  <xs:all>
    <xs:element name="XMObject">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="XMObjectTypeBase">
            <xs:attribute name="class"
             type="XMRelationshipXMDataObjectXMObjectClassNameEnum"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value.

class: An enumeration value that specifies the class name of this **XMObject** element instance.

2.5.2.4.4 XMRelationshipXMDataObjectXMObjectClassNameEnum

The **XMRelationshipXMDataObjectClassNameEnum** simple type enumerates the allowed values for the **class** attribute of the **XMObject** element that is contained in a **DataObject** item in the **DataObjects** collection of an **XMRelationship** object.

```
<xs:simpleType name="XMRelationshipXMDataObjectXMObjectClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMRelationshipIndexDenseDIDs"/>
    <xs:enumeration value="XMRelationshipIndexSparseDIDs"/>
    <xs:enumeration value="XMRelationshipIndex123DIDs"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMRelationshipXMDataObjectClassNameEnum** type.

Enumeration value	Description
"XMRelationshipIndexDenseDIDs"	The object is a relationship index with dense data identifiers.

Enumeration value	Description
"XMRelationshipIndexSparseDIDs"	The object is a relationship index with sparse data identifiers.
"XMRelationshipIndex123DIDs"	The object is a relationship index that is used only for the **RowNumber** column (section 2.3.4).

2.5.2.5 XMObject class="XMRelationshipIndexSparseDIDs"

When the **class** attribute value for the **XMObject** element is "XMRelationshipIndexSparseDIDs", the **XMObject** element contains the metadata for relationship index where the data identifiers are sparse, and the type of the **XMObject** element is **XMRelationshipIndexSparseDIDsXMObjectType**.

```
<xs:complexType name="XMRelationshipIndexSparseDIDsXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMRelationshipIndexSparseDIDsPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRelationshipIndexSparseDIDs"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.5.1 XMRelationshipIndexSparseDIDsPropertiesType

The **XMRelationshipIndexSparseDIDsPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMRelationshipIndexSparseDIDs".

```
<xs:complexType name="XMRelationshipIndexSparseDIDsPropertiesType">
  <xs:all>
    <xs:element name="Flags">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="1"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

Flags: The flags for the relationship index. The only value that can be set is 0x01, which means that the index potentially has relational integrity violations.

2.5.2.6 XMObject class="XMRelationshipIndexDenseDIDs"

When the **class** attribute value for the **XMObject** element is "XMRelationshipIndexDenseDIDs", the **XMObject** element contains the metadata for a relationship index where the data identifiers are dense, and the type of the **XMObject** element is **XMRelationshipIndexDenseDIDsXMObjectType**.

```
<xs:complexType name="XMRelationshipIndexDenseDIDsXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMRelationshipIndexDenseDIDsPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRelationshipIndexDenseDIDs"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.6.1 XMRelationshipIndexDenseDIDsPropertiesType

The **XMRelationshipIndexDenseDIDsPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMRelationshipIndexDenseDIDs".

```
<xs:complexType name="XMRelationshipIndexDenseDIDsPropertiesType">
  <xs:all>
    <xs:element name="Records" type="xs:long"/>
    <xs:element name="TableName" type="xs:string"/>
    <xs:element name="Flags">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="1"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

Records: The count of records.

TableName: The table name. This value MUST be the same as the value of the **ID** element (section 2.6.6) for the dimension that corresponds to this table.

Flags: The flags for the relationship index. The only value that can be set is 0x01, which means that the index potentially has relational integrity violations.

2.5.2.7 XMObject class="XMRelationshipIndex123DIDs"

When the **class** attribute value for the **XMObject** element is "XMRelationshipIndex123DIDs", the **XMObject** element contains metadata for a relationship index that is used only for the **RowNumber** column when that column is on the primary side of the relationship, and the type of the **XMObject** element is **XMRelationshipIndex123DIDsXMObjectType**. For more details about the **RowNumber** column, see section 2.3.4.

```
<xs:complexType name="XMRelationshipIndex123DIDsXMObjectType">
  <xs:all>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRelationshipIndex123DIDs"/>
</xs:complexType>
```

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.8 XMObject class="XMColumnStats"

When the **class** attribute value for the **XMObject** element is "XMColumnStats", the **XMObject** element contains statistical information about the column, and the type of the **XMObject** element is **XMColumnStatsXMObjectType**.

```
<xs:complexType name="XMColumnStatsXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMColumnStatsPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMColumnStats"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.8.1 XMColumnStatsPropertiesType

The **XMColumnStatsPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMColumnStats".

```
<xs:complexType name="XMColumnStatsPropertiesType">
  <xs:all>
    <xs:element name="DistinctStates" type="xs:int"/>
    <xs:element name="MinDataID" type="xs:int"/>
    <xs:element name="MaxDataID" type="xs:int"/>
    <xs:element name="OriginalMinSegmentDataID" type="xs:int"/>
    <xs:element name="RLESortOrder" type="xs:long"/>
    <xs:element name="RowCount" type="xs:long"/>
```

```
      <xs:element name="HasNulls" type="xs:boolean"/>
      <xs:element name="RLERuns" type="xs:long"/>
      <xs:element name="OthersRLERuns" type="xs:long"/>
      <xs:element name="Usage">
        <xs:simpleType>
          <xs:restriction base="xs:long">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="3"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="DBType">
        <xs:simpleType>
          <xs:restriction base="xs:short">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="29"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="XMType">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="3"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="CompressionType">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="2"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="CompressionParam" type="xs:long"/>
      <xs:element name="EncodingHint">
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="2"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="AggCounter" type="xs:long"/>
      <xs:element name="WhereCounter" type="xs:long"/>
      <xs:element name="OrderByCounter" type="xs:long"/>
    </xs:all>
  </xs:complexType>
```

DistinctStates: The number of distinct values, including NULL, in the column.

MinDataID: The minimum data identifier for the column.

MaxDataID: The maximum data identifier for the column.

OriginalMinSegmentDataID: The minimum data identifier for a segment.

RLESortOrder: A value that is unused, MUST be –1, and MUST be ignored.

RowCount: The number of rows in the column segment.

HasNulls: A value that specifies whether the column has NULL values.

RLERuns: An estimate of the number of RLE runs.

OthersRLERuns: The number of RLE runs that are not solid runs. A solid run is a run of consecutive, identical values that cannot be compressed by RLE techniques.

Usage: An enumeration value that specifies the column usage. The values in the following table are used.

Value	Meaning
0	The column is the primary key for the table.
1	The column is the foreign key for the table.
2	The column contains BLOBs.
3	The column is a regular one.

DBType: An enumeration value that specifies the **OLE DB** type of the column. The values in the following table are used.

Value	OLE DB type
0	Null
1	Empty
2	Boolean
3	I2
4	I2
5	I4
6	I8
7	UI1
8	UI2
9	UI4
10	UI8
11	Real4
12	Real8
13	Currency
14	Decimal

Value	OLE DB type
15	Numeric
16	Date
17	DatabaseDate
18	DatabaseTimestamp
19	FileTime
20	String
21	WideString
22	BSTR
23	GUID
24	Bytes
25	Hchapter
26	Variant
27	Error
28	Unknown
29	Varnumeric

XMType: An enumeration value. The values in the following table are used.

Value	Meaning
0	XM_Long
1	XM_Real
2	XM_String

CompressionType: An enumeration value that specifies the type of compression. The values in the following table are used.

Value	Type of compression
0	Automatic (that is, determined by the system)
1	NoSplit (see section 2.7.1)

CompressionParam: Either the bit length for NoSplit compression or the bookmark distance for RLE compression.

EncodingHint: An enumeration value that specifies the type of encoding. The values in the following table are used.

Value	Type of encoding
0	Automatic (that is, determined by the system)
1	Hash encoding
2	Value encoding

AggCounter: A value that is unused and MUST be ignored.

WhereCounter: A value that is unused and MUST be ignored.

OrderByCounter: A value that is unused and MUST be ignored.

2.5.2.9 XMObject class="XMHierarchy"

When the **class** attribute value for the **XMObject** element is "XMHierarchy", the **XMObject** element contains metadata about the **hierarchy** that represents the column, and the type of the **XMObject** element is **XMHierarchyXMObjectType**.

```
<xs:complexType name="XMHierarchyXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMHierarchyPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMHierarchy"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

name: The name of the **XMHierarchy** object instance.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.9.1 XMHierarchyPropertiesType

The **XMHierarchyPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMHierarchy".

```
<xs:complexType name="XMHierarchyPropertiesType">
  <xs:all>
    <xs:element name="SortOrder" type="xs:int"/>
    <xs:element name="IsProcessed" type="xs:boolean"/>
    <xs:element name="TypeMaterialization" type="xs:int"/>
    <xs:element name="ColumnPosition2DataID" type="xs:long"/>
    <xs:element name="ColumnDataID2Position" type="xs:long"/>
    <xs:element name="DistinctDataIDs" type="xs:long"/>
    <xs:element name="TableStore" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

SortOrder: An integer value that specifies the sort order. The values in the following table are valid.

Value	Sort order
0	Ascending
2	Unsorted

IsProcessed: A Boolean value that specifies whether the hierarchy has been processed.

TypeMaterialization: An integer value that specifies the type of materialization. The values in the following table are valid.

Value	Type of materialization
–1	The type of materialization is unspecified. This value MUST NOT be used if **IsProcessed** is not equal to FALSE.
0	The hierarchy gets materialized as a table of two columns: one for position–to–data identifier mapping, and another for data identifier–to–position mapping.
1	The hierarchy gets materialized as a column for position–to–data identifier mapping and a hash for data identifier–to–position mapping.
2	No materialization occurs because the column is empty.
3	No materialization occurs because an identity column is present.

ColumnPosition2DataID: A long value that specifies whether a column position–to–data identifier index is used. The values in the following table are valid.

Value	Meaning
0	A column position–to–data identifier index is used.
–1	A column position–to–data identifier index is not used.

ColumnDataID2Position: A long value that specifies whether a data identifier–to–column position index is used. The values in the following table are valid.

Value	Meaning
1	A data identifier–to–column position index is used.
–1	A data identifier–to–column position index is not used.

DistinctDataIDs: The number of distinct data identifiers.

TableStore: The root name of the hierarchy metadata file that is generated by the system. This value includes neither the version number that is part of the file name nor the "tbl.xml" that appears at the end of the file name.

2.5.2.10 XMObject class="XMUserHierarchy"

When the **class** attribute value for the **XMObject** element is "XMUserHierarchy", the **XMObject** element contains metadata about user-defined hierarchy, and the type of the **XMObject** element is **XMUserHierarchyXMObjectType**.

```
<xs:complexType name="XMUserHierarchyXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMUserHierarchyPropertiesType"/>
  </xs:all>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMUserHierarchy"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

name: The name of the **XMuserHierarchy** object instance.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.10.1 XMUserHierarchyPropertiesType

The **XMUserHierarchyPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMUserHierarchy".

```
<xs:complexType name="XMUserHierarchyPropertiesType">
  <xs:all>
    <xs:element name="IsProcessed" type="xs:boolean"/>
    <xs:element name="TableStore" type="xs:string"/>
    <xs:element name="TableName" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

IsProcessed: A Boolean value that specifies whether the user hierarchy has been processed.

TableStore: A string value that specifies name and unique value combinations for a user hierarchy. The string is constructed as a series of interpretable parts, with a dollar sign ($) as the separator. The names of the columns in the user hierarchy, from the top down, are included, and after each level's name is a zero-based starting number of unique values that exist at the levels numbered higher than the current level. The names that are used MUST match the **ID** element of the **Level** element of the **Hierarchy** element that is contained in the dimension (see section 2.6.6).

For example, assume that a user hierarchy with 4 levels exists. Assume that from the top down, the levels are Country, State, City, and Customer. In the data for this hierarchy, 2 unique values exist for Country, 4 unique values exist for Country-State combinations, and 7 unique values exist for Country-State-City combinations. The string would then be "$Country$0$State$2$City$6$Customer$13$".

In referring to levels, the "ALL" level is not included; the references are to the highest user-defined level, which is one level below the ALL level. The first component of the string is the dollar sign ($).

Then comes the column name that represents the highest level in the user hierarchy, which is "Country". Each substring component continues to be separated by a dollar sign ($). The next component of the string is the number of unique combinations that are higher than this level. That value is "0" because no levels exist that are higher than this one. The next level name is "State". Because 2 countries exist, 2 unique values exist at levels above State, so the value "2" appears in the string. The next level name is "City". Because 4 unique combinations of Country-State and 2 unique countries exist, a total of 6 combinations exist at levels above the City level. Therefore, "6" is added to the string. The next and lowest level is "Customer". Because 7 unique Country-State-City values exist, and there already were 6 unique level values, 13 unique Country-State-City values now exist above the Customer level. Hence, the value "13" is added to the string value. The string value ends with a dollar sign ($).

TableName: The root name of the user hierarchy metadata file that is generated by the system. This value includes neither the version number that is part of the file name nor the "tbl.xml" that appears at the end of the file name.

2.5.2.11 XMObject class="XMHierarchyDataID2PositionHashIndex"

When the **class** attribute value for the **XMObject** element is "XMHierarchyDataID2PositionHashIndex", the **XMObject** element contains metadata for the hierarchy data identifier–to–position hash index mapping, and the type of the **XMObject** element is **XMHierarchyDataID2PositionHashIndexXMObjectType**.

```
<xs:complexType name="XMHierarchyDataID2PositionHashIndexXMObjectType">
  <xs:all/>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMHierarchyDataID2PositionHashIndex"/>
</xs:complexType>
```

ProviderVersion: The provider version.

name: The name of the **XMHierarchyDataID2PositionHashIndex** object instance.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.12 XMObject class="XMColumnSegment"

When the **class** attribute value for the **XMObject** element is "XMColumnSegment", the **XMObject** element contains metadata for a column segment, and the type of the **XMObject** element is **XMColumnSegmentXMObjectType**.

```
<xs:complexType name="XMColumnSegmentXMObjectType">
  <xs:all>
    <xs:element name="Properties" type="XMColumnSegmentPropertiesType"/>
    <xs:element name="Members" type="XMColumnSegmentMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMColumnSegment">
  </xs:attribute>
</xs:complexType>
```

Properties: The property values for the **XMSimpleTable** object.

Members: A collection of **Member** complex type items, each of which contains a complex property for the **XMSimpleTable** object.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.12.1 XMColumnSegmentPropertiesType

The **XMColumnSegmentPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMColumnSegment".

```
<xs:complexType name="XMColumnSegmentPropertiesType">
  <xs:all>
    <xs:element name="Records" type="xs:long"/>
    <xs:element name="Mask">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="2"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

Records: The number of records in the column segment.

Mask: A value that is unused and MUST be ignored.

2.5.2.12.2 XMColumnSegmentMembersType

The **XMColumnSegmentMembersType** complex type holds a collection of **Member** items, each of which contains a property for the parent **XMColumnSegment** object.

```
<xs:complexType name="XMColumnSegmentMembersType">
  <xs:sequence>
    <xs:element name="Member" type="XMColumnSegmentMemberType"
                minOccurs="3" maxOccurs="3"/>
  </xs:sequence>
</xs:complexType>
```

Member: A complex type element that contains a complex **Member** item, which contains a property for the parent **XMObject** element. The value of the **Name** element for the three instances of this element in the **Members** collection MUST have one instance of each enumeration value from the **XMColumnSegmentMemberNameEnum** type (section 2.5.2.12.2.2).

2.5.2.12.2.1 XMColumnSegmentMemberType

The **XMColumnSegmentMemberType** complex type holds a **Member** item that is a property of the parent **XMColumnSegment** object.

```
<xs:complexType name="XMColumnSegmentMemberType">
  <xs:sequence>
    <xs:element name="Name" type="XMColumnSegmentMemberNameEnum"
```

```
                minOccurs="0"/>
      <xs:element name="XMObject">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="XMObjectTypeBase">
              <xs:attribute name="class"
                  type="XMColumnSegmentXMObjectMemberClassNameEnum"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

Name: The name of the **Member** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value.

class: An enumeration value that specifies the class name of this **XMObject** element instance. When the **Name** element of the **Member** item has a particular value, the **XMObject** element of the **Member** item MUST have a specific value for the **class** attribute. The following table lists the constraints between the values of **Name** and **class**.

Value of Name element	Required value of class attribute
"SubSegment"	"XMColumnSegment"
"CompressionInfo"	"XMHybridRLECompressionInfo class<XMRENoSplitCompressionInfo<*n*>>", where *n* = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 21, 32, or XM123CompressionInfo
"ColumnSegmentStats"	"XMColumnSegmentStats"

2.5.2.12.2.2 XMColumnSegmentMemberNameEnum

The **XMColumnSegmentMemberNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Members** collection of an **XMColumnSegment** object.

```
<xs:simpleType name="XMColumnSegmentMemberNameEnum">
  <xs:restriction base="XMObjectMemberNameEnum">
    <xs:enumeration value="SubSegment"/>
    <xs:enumeration value="CompressionInfo"/>
    <xs:enumeration value="ColumnSegmentStats"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMColumnSegmentMemberNameEnum** type.

Enumeration value	Description
"SubSegment"	The **Member** item specifies information about the subsegment.
"CompressionInfo"	The **Member** item specifies information about the compression for the column segment.
"ColumnSegmentStats"	The **Member** item specifies statistical information for the column segment.

2.5.2.12.2.3 XMColumnSegmentXMObjectMemberClassNameEnum

The **XMColumnSegmentXMObjectMemberClassNameEnum** simple type enumerates the allowed values for the **class** attribute of the **XMObject** element that is contained in a **Member** item for a member of a **XMColumnSegment** object.

```
<xs:simpleType name="XMColumnSegmentXMObjectMemberClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMColumnSegment"/>
    <xs:enumeration value="XMColumnSegmentStats"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;1>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;3>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;4>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;5>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;6>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;7>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;8>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;9>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;10>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;12>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;16>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;21>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;32>>"/>
    <xs:enumeration value=
       "XMHybridRLECompressionInfo&lt;class XM123CompressionInfo>"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMColumnSegmentXMObjectMemberClassNameEnum** type.

Enumeration value	Description
"XMColumnSegment"	The object specifies information about the column segment.
"XMColumnSegmentStats"	The object specifies column segment statistics.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<1>. For more details, see section 2.7.3.2.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<2>. For more details, see section 2.7.3.3.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<3>. For more details, see section 2.7.3.4.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<4>. For more details, see section 2.7.3.5.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<5>. For more details, see section 2.7.3.6.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<6>. For more details, see section 2.7.3.7.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<7>. For more details, see section 2.7.3.8.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<8>. For more details, see section 2.7.3.9.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<9>. For more details, see section 2.7.3.10.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<10>. For more details, see section 2.7.3.11.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<12>. For more details, see section 2.7.3.12.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<16>. For more details, see section 2.7.3.13.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<21>. For more details, see section 2.7.3.14.
"XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>"	The column is compressed by means of hybrid RLE compression with XMRENoSplitCompression<32>. For

Enumeration value	Description
	more details, see section 2.7.3.15.
"XMHybridRLECompressionInfo<class XM123CompressionInfo>"	The column is compressed by means of hybrid RLE compression with XM123Compression. For more details, see section 2.7.3.16.

2.5.2.13 XMObject class="XMPartition"

When the **class** attribute value for the **XMObject** element is "XMPartition", the **XMObject** element contains metadata about the **partition (2)**, and the type of the **XMObject** element is **XMPartitionXMObjectType**.

```
<xs:complexType name="XMPartitionXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMPartitionPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMPartition"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

name: The name of the partition. This name is the same as the name of the source data table.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.13.1 XMPartitionPropertiesType

The **XMPartitionPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMPartition".

```
<xs:complexType name="XMPartitionPropertiesType">
  <xs:all>
    <xs:element name="IsProcessed" type="xs:boolean"/>
    <xs:element name="Partition" type="xs:int"/>
  </xs:all>
</xs:complexType>
```

IsProcessed: A Boolean value that specifies whether the partition has been processed.

Partition: An incremental identifier for the partition.

2.5.2.14 XMObject class="XMMultiPartSegmentMap"

When the **class** attribute value for the **XMObject** element is "XMMultiPartSegmentMap", the **XMObject** element contains mapping information for data segments, and the type of the **XMObject** element is **XMMultiPartSegmentMapXMObjectType**.

```
<xs:complexType name="XMMultiPartSegmentMapXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMMultiPartSegmentMapPropertiesType"/>
    <xs:element name="Collections" type="XMMultiPartSegmentMapCollectionsType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMMultiPartSegmentMap"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

Collections: A collection of **Collection** complex type items, each of which contains a complex property for the **XMMultiPartSegmentMap** object. The **Collection** complex property can be repeated multiple times.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of the **XMObject** element.

2.5.2.14.1 XMMultiPartSegmentMapPropertiesType

The **XMMultiSegmentMapPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMMultiPartSegmentMap".

```
<xs:complexType name="XMMultiPartSegmentMapPropertiesType">
  <xs:all>
    <xs:element name="FirstPartitionRecordCount" type="xs:long"/>
    <xs:element name="FirstPartitionSegmentCount" type="xs:long"/>
  </xs:all>
</xs:complexType>
```

FirstPartitionRecordCount: A value that is unused and MUST be ignored.

FirstPartitionSegmentCount: A value that is unused and MUST be ignored.

2.5.2.14.2 XMMultiPartSegmentMapCollectionsType

The **XMMultiPartSegmentMapCollectionsType** complex type contains the collection properties that are allowed when the **XMObject** element is of class "XMMultiPartSegmentMap".

```
<xs:complexType name="XMMultiPartSegmentMapCollectionsType">
  <xs:sequence>
    <xs:element name="Collection"
                type="XMMultiPartSegmentMapCollectionType"/>
  </xs:sequence>
</xs:complexType>
```

Collection: A collection of **Collection** complex type items, each of which contains a complex property for the **XMMultiPartSegmentMap** object. Each **Collection** item can be repeated multiple times.

2.5.2.14.3 XMMultiPartSegmentMapCollectionType

The **XMMultiPartSegmentMapCollectionType** complex type holds a **Collection** item that is a property of the parent **XMMultiPartSegmentMap** object.

```
<xs:complexType name="XMMultiPartSegmentMapCollectionType">
  <xs:sequence>
    <xs:element name="Name" type="XMObjectCollectionNameEnum"
                fixed="Partitions"/>
    <xs:element name="XMObject" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="XMObjectTypeBase">
            <xs:attribute name="class" type=
                "XMMultiPartSegmentMapXMObjectCollectionClassNameEnum"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Name: The name of the **Collection** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element.

2.5.2.14.3.1 XMMultiPartSegmentMapXMObjectCollectionClassNameEnum

The **XMMultiPartSegmentMapXMObjectCollectionClassNameEnum** simple type enumerates the allowed values for the class name of the **XMObject** element that is contained in a **Collection** item in the **Collections** collection of an **XMMultiPartSegmentMap** object.

```
<xs:simpleType name=
              "XMMultiPartSegmentMapXMObjectCollectionClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMSegment1Map"/>
    <xs:enumeration value=
        "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_FastInstantiation>"/>
    <xs:enumeration value=
        "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_ComplexInstantiation>"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMMultiPartSegmentMapXMObjectCollectionClassNameEnum** type.

Enumeration value	Description
"XMSegment1Map"	A segment map for a column that has a single segment.
"XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>"	A segment map of equally sized segments (except that the size of the last segment can differ from that of the others). Note that complex

Enumeration value	Description
	instantiation occurs when the segment size is determined at run time.
"XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>"	A segment map of equally sized segments (except that the size of the last segment can differ from that of the others). Note that fast instantiation is for predetermined segment sizes.

2.5.2.15 XMObject class="XMSegment1Map"

When the **class** attribute value for the **XMObject** element is "XMSegment1Map", the **XMObject** element contains mapping information for the first data segment, and the type of the **XMObject** element is **XMSegment1MapXMObjectType**.

```
<xs:complexType name="XMSegment1MapXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMSegment1MapPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMSegment1Map"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of the **XMObject** element.

2.5.2.15.1 XMSegment1MapPropertiesType

The **XMSegment1MapPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMSegment1Map".

```
<xs:complexType name="XMSegment1MapPropertiesType">
  <xs:all>
    <xs:element name="Records" type="xs:long"/>
  </xs:all>
</xs:complexType>
```

Records: The number of records in the segment map.

2.5.2.16 XMObject class="XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>"

When the **class** attribute value for the **XMObject** element is "XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>", the **XMObject** element contains metadata for a segment map of equally sized segments (except that size of the last

segment can differ from that of the others), and the type of the **XMObject** element is **XMSegmentEqualMapEx_XMSegmentEqualMap_FastInstantiationXMObjectType**. Fast instantiation means that the segment size is predetermined and does not need to be determined at run time.

```
<xs:complexType name=
"XMSegmentEqualMapEx_XMSegmentEqualMap_FastInstantiationXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMSegmentEqualMapEx_PropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
  fixed="XMSegmentEqualMapEx&lt;XMSegmentEqualMap_FastInstantiation>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.16.1 XMSegmentEqualMapEx_PropertiesType

The **XMSegmentEqualMapEx_PropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of either class "XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>" or class "XMSegmentEqualMapEx<XMSegmentEqualMap_FastInstantiation>"

```
<xs:complexType name="XMSegmentEqualMapEx_PropertiesType">
  <xs:all>
    <xs:element name="Segments" type="xs:long"/>
    <xs:element name="Records" type="xs:long"/>
    <xs:element name="RecordsPerSegment" type="xs:long"/>
  </xs:all>
</xs:complexType>
```

Segments: The number of segments.

Records: The total number of records in all of the segments.

RecordsPerSegment: The number of records per segment.

2.5.2.17 XMObject class="XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>"

When the **class** attribute value for the **XMObject** element is "XMSegmentEqualMapEx<XMSegmentEqualMap_ComplexInstantiation>", the **XMObject** element contains metadata for a segment map of equally sized segments (except that size of the last segment can differ from that of the others), and the type of the **XMObject** element is **XMSegmentEqualMapEx_XMSegmentEqualMap_ComplexInstantiationXMObjectType**. Complex instantiation means that the segment size is determined at run time.

```
<xs:complexType name=
```

```
"XMSegmentEqualMapEx_XMSegmentEqualMap_ComplexInstantiationXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMSegmentEqualMapEx_PropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
  fixed="XMSegmentEqualMapEx&lt;XMSegmentEqualMap_ComplexInstantiation>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.18 XMObject class="XMValueDataDictionary<XM_Long>"

When the **class** attribute value for the **XMObject** element is "XMValueDataDictionary<XM_Long>", the **XMObject** element contains metadata for a value dictionary of **long** values, and the type of the **XMObject** element is **XMObject_ValueDictionaryLongType**.

```
<xs:complexType name="XMObject_ValueDictionaryLongType">
  <xs:all>
    <xs:element name="Properties"
                type="PropertiesValueDictionaryType"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMValueDataDictionary&lt;XM_Long>"/>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

ProviderVersion: The provider version.

2.5.2.18.1 PropertiesValueDictionaryType

The **PropertiesValueDictionaryType** complex type contains the specific properties that are allowed when the **XMObject** element is of either class "XMValueDictionary<XM_Real>" or "XMValueDictionary<XM_Long>".

```
<xs:complexType name="PropertiesValueDictionaryType">
  <xs:all>
    <xs:element name="DataVersion" type="xs:int"/>
    <xs:element name="BaseId" type="xs:long"/>
    <xs:element name="Magnitude" type="xs:double"/>
  </xs:all>
</xs:complexType>
```

DataVersion: The internal version number for this data. This version number is not required to match the version numbers of other objects within the same table or column.

BaseId: A value that is part of the calculation used to map from a data identifier to a data value. To perform such a mapping, add this value to the data identifier, and then multiply by the value of the **Magnitude** element.

Magnitude: A value that is part of the calculation used to map from a data identifier to a data value. To perform such a mapping, add the value of the **BaseId** element to the data identifier, and then multiply by this value.

2.5.2.19 XMObject class="XMValueDataDictionary<XM_Real>"

When the **class** attribute value for the **XMObject** element is "XMValueDataDictionary<XM_Real>", the **XMObject** element contains metadata for a value dictionary of **real** values, and the type of the **XMObject** element is **XMObject_ValueDictionaryRealType**.

```
<xs:complexType name="XMObject_ValueDictionaryRealType">
  <xs:all>
    <xs:element name="Properties"
                type="PropertiesValueDictionaryType"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMValueDataDictionary&lt;XM_Real>"/>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

Class: An enumeration value that specifies the class name of this **XMObject** element.

ProviderVersion: The provider version.

2.5.2.20 XMObject class="XMHashDataDictionary<XM_Real>"

When the **class** attribute value for the **XMObject** element is "XMHashDataDictionary<XM_Real>", the **XMObject** element contains metadata for a hash dictionary of **real** values, and the type of the **XMObject** element is **XMObject_HashDictionaryRealType**.

```
<xs:complexType name="XMObject_HashDictionaryRealType">
  <xs:all>
    <xs:element name="Properties"
                type="PropertiesHashDictionaryRealType"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMHashDataDictionary&lt;XM_Real>"/>
  <xs:attributeGroup ref="HashDictionaryAttributeGroup"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

HashDictionaryAttributeGroup: An attribute group that specifies the common attributes for all hash dictionary **XMObject** objects.

2.5.2.20.1 HashDictionaryAttributeGroup

The **HashDictionaryAttributeGroup** attribute group contains the attributes that are common to the **XMObject** objects for hash dictionaries.

```
<xs:attributeGroup name="HashDictionaryAttributeGroup">
  <xs:attribute name="name">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value=
"\d*\.Table-[a-zA-Z0-9$^&amp;&amp;apos;@{},=!$#()%~_\[\]\.\-\+]*\.dictionary"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
</xs:attributeGroup>
```

name: The name of the dictionary file. For information about the interpretation of the components of this attribute, see section 2.2.2.3.2.6.

ProviderVersion: The provider version.

2.5.2.20.2 PropertiesHashDictionaryRealType

The **PropertiesHashDictionaryRealType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMHashDictionary<XM_Real>". This type is also the base type for extension for other hash dictionary **XMObject** types.

```
<xs:complexType name="PropertiesHashDictionaryRealType">
  <xs:sequence>
    <xs:element name="DataVersion" type="xs:int"/>
    <xs:element name="LastId" type="xs:int"/>
    <xs:element name="Nullable" type="xs:boolean"/>
    <xs:element name="Unique" type="xs:boolean"/>
  </xs:sequence>
</xs:complexType>
```

DataVersion: The internal version number for this data. This version number is not required to match the version numbers of other objects within the same table or column.

LastId: The last data identifier value for this hash dictionary.

Nullable: A Boolean value that specifies whether this hash dictionary can contain NULL values.

Unique: A Boolean value that specifies whether all the values in this hash dictionary are unique.

2.5.2.21 XMObject class="XMHashDataDictionary<XM_Long>"

When the **class** attribute value for the **XMObject** element is "XMHashDataDictionary<XM_Long>", the **XMObject** element contains metadata for a hash dictionary of **long** values, and the type of the **XMObject** element is **XMObject_HashDictionaryLongType**.

```
<xs:complexType name="XMObject_HashDictionaryLongType">
  <xs:all>
    <xs:element name="Properties"
```

```
                type="PropertiesHashDictionaryLongType"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMHashDataDictionary&lt;XM_Long&gt;"/>
  <xs:attributeGroup ref="HashDictionaryAttributeGroup"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

HashDictionaryAttributeGroup: An attribute group that specifies the common attributes for all hash dictionary **XMObject** objects.

2.5.2.21.1 PropertiesHashDictionaryLongType

The **PropertiesHashDictionaryLongType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMHashDictionary<XM_Long>".

```
<xs:complexType name="PropertiesHashDictionaryLongType">
  <xs:complexContent>
    <xs:extension base="PropertiesHashDictionaryRealType">
      <xs:sequence>
        <xs:element name="OperatingOn32" type="xs:boolean"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

OperatingOn32: A Boolean value that indicates whether the dictionary encoded values are 32-bit values.

The preceding description documents only the extended element in the **PropertiesHashDictionaryLongType** type. For a description of the elements in the base type, see section 2.5.2.20.2.

2.5.2.22 XMObject class="XMHashDataDictionary<XM_String>"

When the **class** attribute value for the **XMObject** element is "XMHashDataDictionary<XM_String>", the **XMObject** element contains metadata for a hash dictionary of **string** values, and the type of the **XMObject** element is **XMObject_HashDictionaryStringType**.

```
<xs:complexType name="XMObject_HashDictionaryStringType">
  <xs:all>
    <xs:element name="Properties"
                type="PropertiesHashDictionaryStringType"/>
  </xs:all>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMHashDataDictionary&lt;XM_String&gt;"/>
  <xs:attributeGroup ref="HashDictionaryAttributeGroup"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

HashDictionaryAttributeGroup: An attribute group that specifies the common attributes for all hash dictionary **XMObject** objects.

2.5.2.22.1 PropertiesHashDictionaryStringType

The **PropertiesHashDictionaryStringType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMHashDictionary<XM_String>".

```
<xs:complexType name="PropertiesHashDictionaryStringType">
  <xs:complexContent>
    <xs:extension base="mstns:PropertiesHashDictionaryRealType">
      <xs:sequence>
        <xs:element name="DictionaryFlags">
          <xs:simpleType>
            <xs:restriction base="xs:long">
              <xs:minInclusive value="0"/>
              <xs:maxInclusive value="3"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

DictionaryFlags: A bitmask value in which zero or more of the flags that are described in the following table can be set.

Flag value	Meaning
0x01	Lookup is allowed. This flag MUST be set.
0x02	Storage is compressed.

2.5.2.23 XMObject class="XMRENoSplitCompressionInfo<1>"

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<1>", the object is compressed with XMRENoSplitCompression<1> compression (see section 2.7.1.1), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo1Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo1Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;1>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

ProviderVersion: The provider version.

2.5.2.23.1 XMRENoSplitCompressionInfoPropertiesType

The **XMRENoSplitCompressionInfoPropertiesType** complex type contains the properties for compression with XMRENoSplitCompression or XM123Compression.

```
<xs:complexType name="XMRENoSplitCompressionInfoPropertiesType">
  <xs:all>
    <xs:element name="Min" type="xs:int"/>
  </xs:all>
</xs:complexType>
```

Min: The minimum value of the input values that are contained in a compression instance.

2.5.2.24 XMObject class="XMRENoSplitCompressionInfo<2>"

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<2>", the object is compressed with XMRENoSplitCompression<2> compression (see section 2.7.1.2), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo2Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo2Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;2>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

class: An enumeration value that specifies the class name of this **XMObject** element.

ProviderVersion: The provider version.

2.5.2.25 XMObject class="XMRENoSplitCompressionInfo<3>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<3>", the object is compressed with XMRENoSplitCompression<3> compression (see section 2.7.1.3), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo3Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo3Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
```

```
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;3>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.26 XMObject class="XMRENoSplitCompressionInfo<4>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<4>", the object is compressed with XMRENoSplitCompression<4> compression (see section 2.7.1.4), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo4Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo4Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;4>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.27 XMObject class="XMRENoSplitCompressionInfo<5>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<5>", the object is compressed with XMRENoSplitCompression<5> compression (see section 2.7.1.5), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo5Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo5Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;5>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.28 XMObject class="XMRENoSplitCompressionInfo<6>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<6>", the object is compressed with XMRENoSplitCompression<6> compression (see section 2.7.1.6), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo6Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo6Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;6>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.29 XMObject class="XMRENoSplitCompressionInfo<7>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<7>", the object is compressed with XMRENoSplitCompression<7> compression (see section 2.7.1.7), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo7Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo7Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;7>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.30 XMObject class="XMRENoSplitCompressionInfo<8>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<8>", the object is compressed with XMRENoSplitCompression<8> compression (see section 2.7.1.8), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo8Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo8Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;8>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.31 XMObject class="XMRENoSplitCompressionInfo<9>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<9>", the object is compressed with XMRENoSplitCompression<9> compression (see section 2.7.1.9), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo9Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo9Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;9>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.32 XMObject class="XMRENoSplitCompressionInfo<10>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<10>", the object is compressed with XMRENoSplitCompression<10> compression (see section 2.7.1.10), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo10Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo10Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;10>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.33 XMObject class="XMRENoSplitCompressionInfo<12>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<12>", the object is compressed with XMRENoSplitCompression<12> compression (see section 2.7.1.11), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo12Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo12Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;12>"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.34 XMObject class="XMRENoSplitCompressionInfo<16>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<16>", the object is compressed with XMRENoSplitCompression<16> compression (see section 2.7.1.12), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo16Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo16Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
```

```
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;16&gt;"/>
  </xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.35 XMObject class="XMRENoSplitCompressionInfo<21>

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<21>", the object is compressed with XMRENoSplitCompression<21> compression (see section 2.7.1.13), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo21Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo21Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;21&gt;"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.36 XMObject class="XMRENoSplitCompressionInfo<32>"

When the **class** attribute value for the **XMObject** element is "XMRENoSplitCompressionInfo<32>", the object is compressed with XMRENoSplitCompression<32> compression (see section 2.7.1.14), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRENoSplitCompressionInfo32Type**.

```
<xs:complexType name="XMRENoSplitCompressionInfo32Type">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class"
                type="XMObjectClassNameEnum"
                fixed="XMRENoSplitCompressionInfo&lt;32&gt;"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.37 XMObject class="XM123CompressionInfo"

When the **class** attribute value for the **XMObject** element is "XM123CompressionInfo", the object is compressed with XM123Compression compression (see section 2.7.2.1), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XM123CompressionInfoXMObjectType**.

```
<xs:complexType name="XM123CompressionInfoXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMRENoSplitCompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XM123CompressionInfo"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.38 XMRLECompressionInfo

When the **class** attribute value for the **XMObject** element is "XMRLECompressionInfo", the object is compressed with RLE Compression, the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMRLECompressionInfoXMObjectType**.

```
<xs:complexType name="XMRLECompressionInfoXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMRLECompressionInfoPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRLECompressionInfo"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.38.1 XMRLECompressionInfoPropertiesType

The **XMRLECompressionInfoPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMRLECompressionInfo".

```
<xs:complexType name="XMRLECompressionInfoPropertiesType">
```

```
  <xs:all>
    <xs:element name="BookmarkBits" type="xs:long"/>
    <xs:element name="StorageAllocSize" type="xs:long"/>
    <xs:element name="StorageUsedSize" type="xs:long"/>
    <xs:element name="SegmentNeedsResizing" type="xs:boolean"/>
  </xs:all>
</xs:complexType>
```

BookmarkBits: The distance between RLE bookmarks. This value is used to perform a left bit shift operation on 1, which yields the number of RLE records that are encoded between bookmarks. For example, if the value is 5, the operation `1 << 5` is performed. The result, 32, would be the number of RLE records between bookmarks.

StorageAllocSize: The allocated storage size, in 4-byte units, for the segment.

StorageUsedSize: The used storage size, in 4-byte units, for the segment.

SegmentNeedsResizing: A Boolean value that specifies whether an operation has occurred that requires the segment to be resized. This value MUST be **false**.

2.5.2.39 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>", the object is compressed with **hybrid compression** and uses XMRENoSplitCompression<1> compression (section 2.7.3.2), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo1Type**.

```
<xs:complexType name="XMREHybridCompressionInfo1Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;1>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.39.1 XMHybridRLECompressionInfoMembersType

The **XMHybridRLECompressionInfoMembersType** complex type holds a collection of **Member** items, each of which contains a property for the parent **XMObject** object.

```
<xs:complexType name="XMHybridRLECompressionInfoMembersType">
  <xs:sequence>
```

```
    <xs:element name="Member" type="XMHybridRLECompressionInfoMemberType"
                minOccurs="2" maxOccurs="2"/>
  </xs:sequence>
</xs:complexType>
```

Member: A complex type element that contains a property for the parent **XMObject** element. The value of the **Name** element for the two instances of this element in the **Members** collection MUST have one instance of each enumeration value from the **XMHybridRLECompressionInfoMemberNameEnum** type (section 2.5.2.39.3).

2.5.2.39.2 XMHybridRLECompressionInfoMemberType

The **XMHybridRLECompressionInfoMemberType** complex type holds a **Member** item that contains information about a complex property of the parent **XMObject** object.

```
<xs:complexType name="XMHybridRLECompressionInfoMemberType">
  <xs:sequence>
    <xs:element name="Name"
                type="XMHybridRLECompressionInfoMemberNameEnum"/>
    <xs:element name="XMObject">
      <xs:complexType>
        <xs:complexContent>
          <xs:extension base="XMObjectTypeBase">
            <xs:attribute name="class"
            type="XMHybridRLECompressionInfoXMObjectMemberClassNameEnum"/>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Name: The name of the **Member** object.

XMObject: A complex type that contains a nested instance of an **XMObject** element. The type of the element is an extension of **XMObjectTypeBase**. However, the actual content allowed in an instance is constrained and depends on the value of the **class** attribute of the **XMObject** element. The content of the **XMObject** element MUST follow the constraints depending on its **class** attribute value.

class: An enumeration value that specifies the class name of this **XMObject** element instance. When the **Name** element of the **Member** item has a particular value, the **XMObject** element of the **Member** item is constrained and MUST have a specific value for the **class** attribute. The following table lists the constraints between the values of **Name** and **class**

Value of Name element	Required value of class attribute
"RLECompression"	"XMRLECompressionInfo"
"SubCompression"	The **class** attribute of the **XMObject** element MUST be either an instance of the "XM123CompressionInfo" class or an instance of the "XMNoSplitCompressionInfo<*n*>" class, where *n* is the same value as that in the class of the hybrid compression. For example, if the containing **XMObject** element is of class "XMHybridRLECompressionInfo<class

Value of Name element	Required value of class attribute
	XMRENoSplitCompressionInfo<7>>", *n*=7 and the **class** of this **XMObject** element MUST be "XMRENoSplitCompressionInfo<7>", where *n*=7, as well.

2.5.2.39.3 XMHybridRLECompressionInfoMemberNameEnum

The **XMHybridCompressionInfoMemberNameEnum** simple type enumerates the allowed values for the name of a **Member** item in the **Members** collection of an **XMObject** object for the hybrid compression classes.

```
<xs:simpleType name="XMHybridRLECompressionInfoMemberNameEnum">
  <xs:restriction base="xs:string">
    <xs:enumeration value="RLECompression"/>
    <xs:enumeration value="SubCompression"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMHybridCompressionInfoMemberNameEnum** type.

Enumeration value	Description
"RLECompression"	The **Member** item contains information about RLE compression.
"SubCompression"	The **Member** item contains information about subcompression.

2.5.2.39.4 XMHybridRLECompressionInfoXMObjectClassNameEnum

The **XMHybridRLECompressionInfoXMObjectMemberClassNameEnum** simple type enumerates the allowed values for the class name of the **XMObject** element that is contained in a **Member** item in the **Members** collection of an **XMSimpleTable** object.

```
<xs:simpleType name="XMHybridRLECompressionInfoXMObjectMemberClassNameEnum">
  <xs:restriction base="XMObjectClassNameEnum">
    <xs:enumeration value="XMRLECompressionInfo"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;1>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;2>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;3>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;4>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;5>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;6>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;7>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;8>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;9>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;10>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;12>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;16>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;21>"/>
    <xs:enumeration value="XMRENoSplitCompressionInfo&lt;32>"/>
    <xs:enumeration value="XM123CompressionInfo"/>
  </xs:restriction>
</xs:simpleType>
```

The following table describes the enumeration values in the **XMHybridRLECompressionInfoXMObjectMemberClassNameEnum** type.

Enumeration value	Description
"RLECompressionInfo"	The **XMObject** object contains information about RLE compression.
"XMRENoSplitCompressionInfo<1>"	The **XMObject** object describes XMRENoSplitCompressionInfo<1> compression. For more information, see section 2.7.1.1.
"XMRENoSplitCompressionInfo<2>"	The **XMObject** object describes XMRENoSplitCompressionInfo<2> compression. For more information, see section 2.7.1.2.
"XMRENoSplitCompressionInfo<3>"	The **XMObject** object describes XMRENoSplitCompressionInfo<3> compression. For more information, see section 2.7.1.3.
"XMRENoSplitCompressionInfo<4>"	The **XMObject** object describes XMRENoSplitCompressionInfo<4> compression. For more information, see section 2.7.1.4.
"XMRENoSplitCompressionInfo<5>"	The **XMObject** object describes XMRENoSplitCompressionInfo<5> compression. For more information, see section 2.7.1.5.
"XMRENoSplitCompressionInfo<6>"	The **XMObject** object describes XMRENoSplitCompressionInfo<6> compression. For more information, see section 2.7.1.6.
"XMRENoSplitCompressionInfo<7>"	The **XMObject** object describes XMRENoSplitCompressionInfo<7> compression. For more information, see section 2.7.1.7.
"XMRENoSplitCompressionInfo<8>"	The **XMObject** object describes XMRENoSplitCompressionInfo<8> compression. For more information, see section 2.7.1.8.
"XMRENoSplitCompressionInfo<9>"	The **XMObject** object describes XMRENoSplitCompressionInfo<9> compression. For more information, see section 2.7.1.9.
"XMRENoSplitCompressionInfo<10>"	The **XMObject** object describes XMRENoSplitCompressionInfo<10> compression . For more information, see section 2.7.1.10.
"XMRENoSplitCompressionInfo<12>"	The **XMObject** object describes XMRENoSplitCompressionInfo<12> compression. For more information, see section 2.7.1.11.
"XMRENoSplitCompressionInfo<16>"	The **XMObject** object describes XMRENoSplitCompressionInfo<16> compression. For more information, see section 2.7.1.12.
"XMRENoSplitCompressionInfo<21>"	The **XMObject** object describes XMRENoSplitCompressionInfo<21> compression. For more information, see section 2.7.1.13.
"XMRENoSplitCompressionInfo<32>"	The **XMObject** object describes XMRENoSplitCompressionInfo<32> compression. For more information, see section 2.7.1.14.
"XM123CompressionInfo"	The **XMObject** object describes XM123CompressionInfo compression. For more information, see section 2.7.2.

2.5.2.40 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<2> compression (see section 2.7.3.3), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo2Type**.

```
<xs:complexType name="XMREHybridCompressionInfo2Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.41 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<3> compression (see section 2.7.3.4), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo3Type**.

```
<xs:complexType name="XMREHybridCompressionInfo3Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.42 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<4> compression (see section 2.7.3.5), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo4Type**.

```
<xs:complexType name="XMREHybridCompressionInfo4Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;4>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.43 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<5> compression (see section 2.7.3.6), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo5Type**.

```
<xs:complexType name="XMREHybridCompressionInfo5Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;5>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.44 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<6> compression (see section 2.7.3.7), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo6Type**.

```
<xs:complexType name="XMREHybridCompressionInfo6Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;6>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.45 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<7> compression (see section 2.7.3.8), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo7Type**.

```
<xs:complexType name="XMREHybridCompressionInfo7Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;7>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.46 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<8> compression (see section 2.7.3.9), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo8Type**.

```
<xs:complexType name="XMREHybridCompressionInfo8Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;8>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.47 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<9> compression (see section 2.7.3.10), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo9Type**.

```
<xs:complexType name="XMREHybridCompressionInfo9Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;9>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.48 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<10> compression (see section 2.7.3.11), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo10Type**.

```
<xs:complexType name="XMREHybridCompressionInfo10Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;10>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.49 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<12> compression (see section 2.7.3.12), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo12Type**.

```
<xs:complexType name="XMREHybridCompressionInfo12Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;12>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.50 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<16> compression (see section 2.7.3.13), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo16Type**.

```
<xs:complexType name="XMREHybridCompressionInfo16Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;16>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.51 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<21> compression (see section 2.7.3.14), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo21Type**.

```
 <xs:complexType name="XMREHybridCompressionInfo21Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;21>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.52 XMObject class="XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>", the object is compressed with hybrid compression and uses XMRENoSplitCompression<32> compression (see section 2.7.3.15), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfo32Type**.

```
<xs:complexType name="XMREHybridCompressionInfo32Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;32>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.53 XMObject class="XMHybridRLECompressionInfo<class XM123CompressionInfo>"

When the **class** attribute value for the **XMObject** element is "XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo< XM123CompressionInfo>>", the object is compressed with hybrid compression and uses XM123 compression (see section 2.7.3.16), the **XMObject** element contains the metadata for the compression, and the type of the **XMObject** element is **XMHybridCompressionInfoXM123Type**.

```
<xs:complexType name="XMREHybridCompressionInfoXM123Type">
  <xs:all>
    <xs:element name="Members"
                type="XMHybridRLECompressionInfoMembersType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;XM123CompressionInfo>>"/>
</xs:complexType>
```

Members: A collection of **Member** items, each of which contains a complex property for the **XMObject** element.

ProviderVersion: The provider version.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.54 XMObject class="ColumnSegmentStats"

When the **class** attribute value for the **XMObject** element is "XMColumnSegmentStats", the **XMObject** element contains statistical information for a column segment, and the type of the **XMObject** element is **XMColumnSegmentStatsXMObjectType**.

```
<xs:complexType name="XMColumnSegmentStatsXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMColumnSegmentStatsPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMColumnSegmentStats"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

name: The name of the **XMRawColumnPartitionDataObject** object.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.54.1 XMColumnSegmentStatsPropertiesType

The **XMColumnSegmentStatsPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMColumnSegmentStats".

```
<xs:complexType name="XMColumnSegmentStatsPropertiesType">
  <xs:all>
    <xs:element name="DistinctStates" type="xs:long"/>
    <xs:element name="MinDataID" type="xs:int"/>
    <xs:element name="MaxDataID" type="xs:int"/>
    <xs:element name="OriginalMinSegmentDataID" type="xs:int"/>
    <xs:element name="RLESortOrder" type="xs:long"/>
    <xs:element name="RowCount" type="xs:long"/>
    <xs:element name="HasNulls" type="xs:boolean"/>
    <xs:element name="RLERuns" type="xs:long"/>
    <xs:element name="OthersRLERuns" type="xs:long"/>
  </xs:all>
</xs:complexType>
```

DistinctStates: The number of distinct values, including NULL, in the column segment.

MinDataID: The minimum data identifier for the column segment.

MaxDataID: An integer value that specifies the maximum data identifier for the column segment.

OriginalMinSegmentDataID: The minimum data identifier for a segment.

RLESortOrder: A value that is unused, MUST be–1, and MUST be ignored.

RowCount: The count of rows in this segment.

HasNulls: A Boolean value that specifies whether the segment has NULL values.

RLERuns: The number of RLE runs.

OthersRLERuns: The number of RLE runs that are not solid runs. A solid run is a run of consecutive, identical values that can be compressed by RLE techniques.

2.5.2.55 XMObject class="XMRawColumnPartitionDataObject"

When the **class** attribute value for the **XMObject** element is "XMRawColumnPartitionDataObject", the **XMObject** element contains information about the partition for the data object, and the type of the **XMObject** element is **XMRawColumnPartitionDataObjectXMObjectType**.

```
<xs:complexType name="XMRawColumnPartitionDataObjectXMObjectType">
  <xs:all>
    <xs:element name="Properties"
                type="XMRawColumnPartitionDataObjectPropertiesType"/>
  </xs:all>
  <xs:attribute name="ProviderVersion" type="xs:int"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="class" type="XMObjectClassNameEnum"
                fixed="XMRawColumnPartitionDataObject"/>
</xs:complexType>
```

Properties: A collection of properties for the **XMObject** element.

ProviderVersion: The provider version.

name: The name of the **XMRawColumnPartitionDataObject** object.

class: An enumeration value that specifies the class name of this **XMObject** element.

2.5.2.55.1 XMRawColumnPartitionDataObjectPropertiesType

The **XMRawColumnPartitionDataObjectPropertiesType** complex type contains the specific properties that are allowed when the **XMObject** element is of class "XMRawColumnPartitionDataObject".

```
<xs:complexType name="XMRawColumnPartitionDataObjectPropertiesType">
  <xs:all>
    <xs:element name="DataVersion" type="xs:int"/>
    <xs:element name="Partition" type="xs:int"/>
    <xs:element name="SegmentCount" type="xs:int"/>
  </xs:all>
</xs:complexType>
```

DataVersion: The internal version number for this data. This version number is not required to match the version numbers of other objects within the same table or column.

Partition: An incremental number that identifies for the partition.

SegmentCount: The count of segments in the partition.

2.5.3 Contents of the .tbl.xml Files

Each file that contains metadata for a table (.tbl.xml file) contains an **XMSimpleTable** object. These table metadata files differ according to which columns exist in the **Columns** collection of the **XMSimpleTable** object. The following table specifies which columns exist for each type of table metadata file.

Type of table metadata file	Example file name	Columns collection content
Hierarchy file	10.H$Table-Diet$wt.0.tbl.xml	One column collection item for each of the system-generated hierarchy indexes: ID_TO_POS and POS_TO_ID.
User hierarchy file	10.U$Hierarchy1.0.tbl.xml	One column collection item for each of the system-generated user hierarchy columns: CHILD_COUNT, FIRST_CHILD_POS, and MULTI_LEVEL_ID, PARENT_POS.
Relationship file	R$Table-Diet$ec91bf00-f577-4c86-b7f8-8c5dcd44a2ac.64.tbl.xml	One column collection item for the system-generated relationship file column: INDEX.
Table file	Table-Diet.31.tbl.xml	One column collection item for each column in the source data table.

2.6 Model OLAP Files

The tables contained in one model are represented as an **OLAP cube**. The OLAP cube metadata for the model is contained in a set of XML files. These files are derived from metadata complex type definitions, as specified in [MS-SSAS] section 2.2.4.2.2, and modified as specified in the following subsections.

2.6.1 Load Element Document Node

Every model OLAP file has a **Load** element as the document node of the XML document. The **Load** element serves as the document node for the remainder of the OLAP object metadata description. The content of the **Load** element is analogous to the content of the **Create** command on the OLAP server, as specified in [MS-SSAS] section 3.1.4.3.2.1.1.3.

```
<xs:element name="Load" type="LoadElementType"/>

<xs:complexType name="LoadElementType">
  <xs:all>
    <xs:element name="ParentObject" type="ObjectReferenceTabularModel"/>
    <xs:element name="ObjectDefinition" type="MajorObjectTabularModel"/>
  </xs:all>
</xs:complexType>
```

ParentObject: A reference, as specified in section 2.6.1.2, to the parent of the object that is defined by the **ObjectDefinition** element.

ObjectDefinition: A complex type element, as specified in section 2.6.1.1, that contains the definition of an OLAP object.

2.6.1.1 MajorObjectTabularModel

The **MajorObjectTabularModel** complex type defines an OLAP major object. This type is analogous to the **MajorObject** type, as specified in [MS-SSAS] section 2.2.4.2.2.1, but contains fewer objects that are available to be defined within the **xs:choice** element. Additionally, the types of the elements extend the types as specified in [MS-SSAS] to add elements for tabular models.

```
<xs:complexType name="MajorObjectTabularModel">
  <xs:choice>
    <xs:element name="Cube" type="CubeTabularModel"/>
    <xs:element name="Database" type="DatabaseTabularModel"/>
    <xs:element name="DataSource" type="DataSourceTabularModel"/>
    <xs:element name="DataSourceView" type="DataSourceViewTabularModel"/>
    <xs:element name="Dimension" type="DimensionTabularModel"/>
    <xs:element name="MdxScript" type="MdxScriptTabularModel"/>
    <xs:element name="MeasureGroup" type="MeasureGroupTabularModel"/>
    <xs:element name="Partition" type="PartitionTabularModel"/>
  </xs:choice>
</xs:complexType>
```

Cube: An element of type **CubeTabularModel** (section 2.6.5), which is an extension of the **Cube** type ([MS-SSAS] section 2.2.4.2.2.9).

Database: An element of type **DatabaseTabularModel** (section 2.6.4), which is an extension of the **Database** type ([MS-SSAS] section 2.2.4.2.2.5).

DataSource: An element of type **DataSourceTabularModel** (section 2.6.2), which is an extension of the **DataSource** type ([MS-SSAS] section 2.2.4.2.2.6).

DataSourceView: An element of type **DataSourceViewTabularModel** (section 2.6.3), which is an extension of the **DataSourceView** type ([MS-SSAS] section 2.2.4.2.2.7).

Dimension: An element of type **DimensionTabularModel** (section 2.6.6), which is an extension of the **Dimension** type ([MS-SSAS] section 2.2.4.2.2.8).

MdxScript: An element of type **MdxScriptTabularModel** (section 2.6.9), which is an extension of the **MdxScript** type ([MS-SSAS] section 2.2.4.2.2.10).

MeasureGroup: An element of type **MeasureGroupTabularModel** (section 2.6.7), which is an extension of the **MeasureGroup** type ([MS-SSAS] section 2.2.4.2.2.11).

Partition: An element of type **PartitionTabularModel** (section 2.6.8), which is an extension of the **Partition** type ([MS-SSAS] section 2.2.4.2.2.13).

2.6.1.2 ObjectReferenceTabularModel

The **ObjectReferenceTabularModel** complex type specifies the parent object of the object that is being described. This type is a subset of the **ObjectReference** type as specified in [MS-SSAS] section 3.1.4.3.2.1.1.1.

```
<xs:complexType name="ObjectReferenceTabularModel">
  <xs:all>
    <xs:element name="DatabaseID" type="xs:string" minOccurs="0"/>
    <xs:element name="CubeID" type="xs:string" minOccurs="0"/>
  </xs:all>
```

```
</xs:complexType>
```

DatabaseID: An element as specified in [MS-SSAS] section 3.1.4.3.2.1.1.1.

CubeID: An element as specified in [MS-SSAS] section 3.1.4.3.2.1.1.1.

2.6.1.3 TabularModelElementsGroup Group

The **TabularModelElementsGroup** group contains a group of elements that have been added to many of the OLAP major object types, for the case of the tabular model.

```
<xs:group name="TabularModelElementsGroup">
  <xs:sequence>
    <xs:element name="Ordinal" type="xs:int"/>
    <xs:element name="PersistLocation" type="xs:int"/>
    <xs:element name="System" type="xs:boolean"/>
    <xs:element name="DataFileList" type="xs:string"/>
  </xs:sequence>
</xs:group>
```

Ordinal: The position of this object within the collection of objects of this type.

PersistLocation: The version number that will appear within the file name. For example, if the value is "10" for a dimension object, the file name will end in "10.dim.xml".

System: A Boolean value that MUST be set to **false**.

DataFileList: A semicolon-separated list of all the data files materialized for this object.

2.6.2 DataSourceTabularModel

The **DataSourceTabularModel** complex type is extended from the **DataSource** base type, as specified in [MS-SSAS] section 2.2.4.2.2.6. The **DataSource** base type is an abstract type and has two types derived from it, both of which can be used in the data source definition. They are the **OlapDataSource** type, as specified in [MS-SSAS] section 2.2.4.2.2.6.2, and the **RelationalDataSource** type, as specified in [MS-SSAS] section 2.2.4.2.2.6.1.

The data source definition is contained in a data source definition XML file. An example of a generated data source definition XML file name is PushedDataSource-F052E9FD-98DA-441C-A0C0-B84DA82E5F25.0.ds.xml.

Every tabular model MUST have a data source defined.

```
<xs:complexType name="DataSourceTabularModel">
  <xs:complexContent>
    <xs:extension base="DataSource">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
        <xs:element name="PermissionFileList" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

PermissionFileList: A semicolon-separated list of the files included in the tabular model metadata that define user permissions for the data source.

2.6.3 DataSourceViewTabularModel

The **DataSourceViewTabularModel** complex type is extended from the **DataSourceView** base type, as specified in [MS-SSAS] section 2.2.4.2.2.7.

The data source view definition is contained in a data source view definition XML file. An example of a generated data source view definition XML file name is Sandbox.0.dsv.xml.

Every tabular model MUST have a data source view defined.

```
<xs:complexType name="DataSourceViewTabularModel">
  <xs:complexContent>
    <xs:extension base="DataSourceView">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

2.6.4 DatabaseTabularModel

The **DatabaseTabularModel** complex type is extended from the **Database** base type, as specified in [MS-SSAS] section 2.2.4.2.2.5.

The database definition is contained in a database definition XML file. An example of a generated database definition XML file name is ImportDiet2.db.xml.

Every tabular model MUST have a database defined.

```
<xs:complexType name="DatabaseTabularModel">
  <xs:complexContent>
    <xs:extension base="Database">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

2.6.5 CubeTabularModel

The **CubeTabularModel** complex type is extended from the **Cube** base type, as specified in [MS-SSAS] section 2.2.4.2.2.9.

The OLAP cube definition is contained in a cube definition XML file. An example of a generated cube definition XML file name is Model.33.cub.xml.

Every tabular model MUST have a cube defined.

When the cube is defined for a tabular model, the following rules MUST be followed:

- Each table that is included in the tabular model MUST be defined in the cube's **Dimensions** collection as a **Dimension** of type **CubeDimension**, as specified in [MS-SSAS] section 2.2.4.2.2.9.1.
- Each column in each table MUST be defined in the cube's **Attributes** collection as an **Attribute** of type **CubeAttribute**, as specified in [MS-SSAS] section 2.2.4.2.2.9.2.

```
<xs:complexType name="CubeTabularModel">
  <xs:complexContent>
    <xs:extension base="Cube">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
        <xs:element name="PermissionFileList" type="xs:string"/>
        <xs:element name="MeasureGroupFileList" type="xs:string"/>
        <xs:element name="PerspectiveFileList" type="xs:string"/>
        <xs:element name="AssemblyFileList" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

PermissionFileList: A semicolon-separated list of the files included in the tabular model metadata that define user permissions for the model.

MeasureGroupFileList: A semicolon-separated list of the files included in the tabular model metadata that define the measure groups for the model.

PerspectiveFileList: A semicolon-separated list of the files included in the tabular model metadata that define perspectives for the model.

AssemblyFileList: A semicolon-separated list of the files included in the tabular model metadata that define **assemblies** for the model.

2.6.6 DimensionTabularModel

The **DimensionTabularModel** type is extended from the **Dimension** base type, as specified in [MS-SSAS] section 2.2.4.2.2.8.

The dimension definition is contained in a dimension definition XML file. An example of a generated dimension definition XML file name is Table-Diet.64.dim.xml.

Every tabular model MUST have a dimension defined.

A dimension MUST be defined for every table that is included in the tabular model.

The dimension MUST follow the following rules:

- An **Attribute** element of type **DimensionAttribute**, as specified in [MS-SSAS] section 2.2.4.2.2.8.1, MUST be defined for every column in the table.
- If the dimension represents a table that is a primary table in a table relationship, a **Relationships** collection of type **Relationships**, as specified in [MS-SSAS] section 2.2.4.2.2.8, MUST contain a **Relationship** element of type **Relationship**, as specified in [MS-SSAS] section 2.2.4.2.2.8.3, for each relationship defined in the tabular model.

```
<xs:complexType name="DimensionTabularModel">
  <xs:complexContent>
    <xs:extension base="Dimension">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
        <xs:element name="PermissionFileList" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

PermissionFileList: A semicolon-separated list of the files included in the tabular model metadata that define user permissions for the dimension.

2.6.7 MeasureGroupTabularModel

The **MeasureGroupTabularModel** type is from the **MeasureGroup** base type, as specified in [MS-SSAS] section 2.2.4.2.2.11.

The measure group definition is contained in a measure group definition XML file. An example of a generated measure group definition XML file name is Table-Diet.64.det.xml.

Every tabular model MUST have at least one measure group defined.

A measure group MUST be defined for every table in the tabular model.

Measure group definitions MUST follow the following rules:

- The measure group MUST contain a **Dimension** element of type **DegenerateMeasureGroupDimension**, as specified in [MS-SSAS] section 2.2.4.2.2.11.1.4.
- **DegenerateMeasureGroupDimension** MUST have an **Attribute** element of type **MeasureGroupDimensionAttribute**, as specified in [MS-SSAS] section 2.2.4.2.2.11.2, defined for every column in the table.
- If the table is a primary table in a table relationship, the measure group MUST have a **Dimension** element of type **ReferenceMeasureGroupDimension**, as specified in [MS-SSAS] section 2.2.4.2.2.11.1.3.
- **ReferenceMeasureGroupDimension** MUST have an **Attribute** element of type **MeasureGroupDimensionAttribute**, as specified in [MS-SSAS] section 2.2.4.2.2.11.2, defined for every column in the related table.

```
<xs:complexType name="MeasureGroupTabularModel">
  <xs:complexContent>
    <xs:extension base="MeasureGroup">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
        <xs:element name="AggregationDesignFileList" type="xs:string"/>
        <xs:element name="PartitionFileList" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

AggregationDesignFileList: A semicolon-separated list of the files included in the tabular model metadata that define AggregationDesign objects for the model.

PartitionFileList: A semicolon-separated list of the files included in the tabular model metadata that define partitions for the model.

2.6.8 PartitionTabularModel

The **PartitionTabularModel** type is extended from the **Partition** base type, as specified in [MS-SSAS] section 2.2.4.2.2.13.

The partition (2) definition is contained in a partition definition XML file. An example of a generated partition definition XML file name is Table-LatLong.1.prt.xml.

Every tabular model MUST have at least one partition defined.

A partition MUST be defined for every table in the tabular model.

```
<xs:complexType name="PartitionTabularModel">
  <xs:complexContent>
    <xs:extension base="Partition">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

2.6.9 MdxScriptTabularModel

The **MdxScriptTabularModel** complex type extends the **MdxScript** base type, as specified in [MS-SSAS] section 2.2.4.2.2.10.

The multidimensional expression (MDX) script definition is contained in an MDX script definition XML file. An example of a generated MDX script definition XML file name is MdxScript.0.scr.xml.

Every tabular model MUST have an MDX script defined.

Every MDX script MUST contain a command that defines at least one measure. The measure MAY<8> be defined with the following command:

```
<Command>
  <Text>
    CALCULATE;
    CREATE MEMBER CURRENTCUBE.Measures.[__Count of Models] AS 1;
    ALTER CUBE CURRENTCUBE UPDATE DIMENSION Measures,
    Default_Member = [__Count of Models];
  </Text>
</Command>
```

The **MdxScriptTabularModel** complex type is defined as follows:

```
<xs:complexType name="MdxScriptTabularModel">
  <xs:complexContent>
    <xs:extension base="MdxScript">
      <xs:sequence>
        <xs:group ref="TabularModelElementsGroup"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

TabularModelElementsGroup: A group of elements that is added to base OLAP types for the tabular model derivations from those types. For more details, see section 2.6.1.3.

2.6.10 OLAP Information Files

In addition to the standard OLAP metadata information that is contained in the **Dimension** object (see section 2.6.6) and the **Partition** object (see section 2.6.8), an information file is generated that contains additional metadata information for those objects.

2.6.10.1 Partition Information File

The additional metadata information for the **Partition** object is contained in a partition information XML file. An example of a generated partition information XML file name is Info.33.xml.

The document node in the partition information file contains a **Partition** element.

```
<xs:element name="Partition" type="PartitionInformationType"/>
```

Partition: A complex type element that specifies additional metadata information for the partition.

2.6.10.1.1 PartitionInformationType

The **PartitionInformationType** complex type holds additional metadata information about the partition, beyond the metadata information that is contained in the OLAP **Partition** object (section 2.6.8).

```
<xs:complexType name="PartitionInformationType">
  <xs:sequence>
    <xs:element name="DataVersion" type="xs:int"/>
```

```
    <xs:element name="RigidAggVersion" type="xs:int"/>
    <xs:element name="FlexAggVersion" type="xs:int"/>
    <xs:element name="DataIndexVersion" type="xs:int"/>
    <xs:element name="RigidIndexVersion" type="xs:int"/>
    <xs:element name="FlexIndexVersion" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
```

DataVersion: The internal version number for this object. This version number is not required to match the version numbers of other objects within the same table or column.

RigidAggVersion: A value that is unused and MUST be ignored.

FlexAggVersion: A value that is unused and MUST be ignored.

DataIndexVersion: A value that is unused and MUST be ignored.

RigidIndexVersion: A value that is unused and MUST be ignored.

FlexIndexVersion: A value that is unused and MUST be ignored.

2.6.10.2 Dimension Information File

The additional metadata information for the **Dimension** object is contained in a dimension information XML file. An example of a generated Dimension information XML file name is Info.33.xml.

The document node in the dimension information file contains a **Dimension** element.

```
<xs:element name="Dimension" type="DimensionInformationType"/>
```

Dimension: A complex type element that specifies additional metadata information for the dimension.

2.6.10.2.1 DimensionInformationType

The **DimensionInformationType** complex type holds additional metadata information about the dimension, beyond the metadata information contained in the OLAP **Dimension** object (section 2.6.6).

```
<xs:complexType name="DimensionInformationType">
  <xs:sequence>
  <xs:element name="DataVersion" type="xs:int"/>
  <xs:element name="IndexVersion" type="xs:int"/>
  <xs:element name="DecodeStoreVersion" type="xs:int"/>
  <xs:element name="LevelStoreVersion" type="xs:int"/>
  <xs:element name="Properties"
              type="DimensionInformationPropertiesType"/>
  </xs:sequence>
</xs:complexType>
```

DataVersion: The internal version number for this object. This version number is not required to match the version numbers of other objects within the same table or column.

IndexVersion: A value that is unused and MUST be ignored.

DecodeStoreVersion: A value that is unused and MUST be ignored.

LevelStoreVersion: A value that is unused and MUST be ignored.

Properties: A complex type element that specifies additional properties for the dimension.

2.6.10.2.1.1 DimensionInformationPropertiesType

The **DimensionInformationPropertiesType** complex type holds a collection of properties for the dimension.

```
<xs:complexType name="DimensionInformationPropertiesType">
  <xs:sequence>
    <xs:element name="Property" type="DimensionInformationPropertyType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Property: A complex type that specifies a single property in the properties collection for dimension information.

2.6.10.2.1.1.1 DimensionInformationPropertyType

The **DimensionInformationPropertyType** complex type specifies the information for one property instance for the dimension information object.

```
<xs:complexType name="DimensionInformationPropertyType">
  <xs:sequence>
    <xs:element name="ParentChild" type="xs:boolean"/>
    <xs:element name="Depth" type="xs:int"/>
    <xs:element name="Balanced" type="xs:boolean"/>
    <xs:element name="HasHoles" type="xs:boolean"/>
    <xs:element name="MapDataset"
                type="DimensionInformationPropertyMapDatasetType"/>
  </xs:sequence>
</xs:complexType>
```

ParentChild: A value that is unused and MUST be ignored.

Depth: A value that is unused and MUST be ignored.

Balanced: A value that is unused and MUST be ignored.

HasHoles: A value that is unused and MUST be ignored.

MapDataset: A complex type that specifies additional mapping information for dimension information properties.

2.6.10.2.1.1.2 DimensionInformationMapDataSetType

The **DimensionInformationMapDatasetType** specifies the property information for a dataset map.

```
<xs:complexType name="DimensionInformationPropertyMapDatasetType">
  <xs:sequence>
    <xs:element name="m_cbOffsetHeader" type="xs:long"/>
    <xs:element name="m_cbOffsetData" type="xs:long"/>
    <xs:element name="m_cRecord" type="xs:long"/>
    <xs:element name="m_cSegment" type="xs:long"/>
    <xs:element name="m_mskFormat" type="xs:long"/>
    <xs:element name="m_cbHeader" type="xs:long"/>
    <xs:element name="m_cPath" type="xs:long"/>
    <xs:element name="m_cData" type="xs:long"/>
    <xs:element name="m_cSegmentIndex" type="xs:long"/>
    <xs:element name="MapDataIndices" type="xs:long"/>
    <xs:element name="MinMaxValues" type="xs:long"/>
  </xs:sequence>
</xs:complexType>
```

m_cbOffsetHeader: A value that is unused and MUST be ignored.

m_cbOffsetData: A value that is unused and MUST be ignored.

m_cRecord: A value that is unused and MUST be ignored.

m_cSegment: A value that is unused and MUST be ignored.

m_mskFormat: A value that is unused and MUST be ignored.

m_cbHeader: A value that is unused and MUST be ignored.

m_cPath: A value that is unused and MUST be ignored.

m_cData: A value that is unused and MUST be ignored.

m_cSegmentIndes: A value that is unused and MUST be ignored.

MapDataIndicies: A value that is unused and MUST be ignored.

MinMaxValues: A value that is unused and MUST be ignored.

2.6.10.3 Cube Information File

The additional metadata information for the **Cube** object is contained in a cube information XML file. An example of a generated cube information XML file name is Info.21.xml.

The document node in the cube information file contains a **Cube** element.

```
<xs:element name="Cube" type="CubeInformationType"/>
```

Cube: A complex type element that specifies additional metadata information for the cube.

2.6.10.3.1 CubeInformationType

The **CubeInformationType** complex type is empty.

```
<xs:complexType name="CubeInformationType">
  <xs:sequence>
  </xs:sequence>
```

```
</xs:complexType>
```

CubeInformationType contains no elements.

2.7 Compression

All data structures except for XML files are compressed. The compression algorithms are described in the following subsections. If the decompression algorithm is not simply a reversal of the compression algorithm, the decompression algorithm is also explained.

2.7.1 XMRENoSplit Compression Algorithms

XMRENoSplit compression algorithms use range encoding, in which the algorithms use a minimum offset plus only the necessary number of bits to encode the entire range of data values, after the range's reduction by that minimum offset. This procedure reduces the numeric size of the values to be stored and, therefore, the number of bits that are required to hold each value. After this range reduction, a form of bit packing compression is used for which the data to be stored is compacted into a specified number of bits.

The storage area for the compressed value MUST be zeroed out prior to compressing the value and storing it in that area.

2.7.1.1 XMRENoSplitCompressionInfo<1>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<1>**, as specified in section 2.5.2.23.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.23). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 1.

The following pseudocode illustrates this phase:

```
SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit
```

In Phase 2, the compressed storage (*compressedStorage*) is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 1) is multiplied by the storage data

size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((1) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

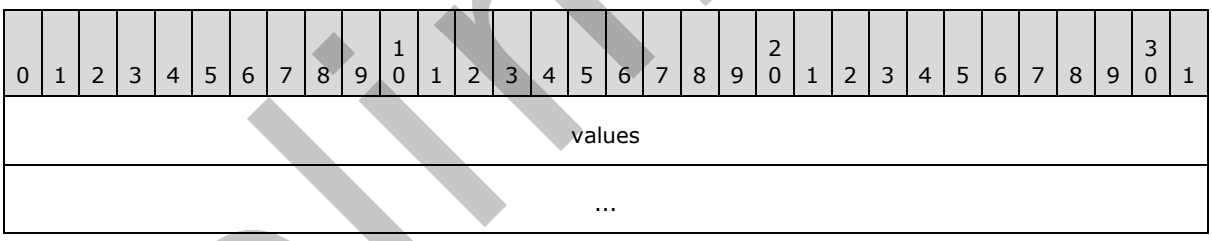
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 63, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 63))

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 1 bit, in sequence and ordered low to high. In the sequence, the first value occupies Bit 0, the second value occupies Bit 1, and so on.

The *startBit* offset followed the sequence of Bit 0, 1, 2, and so on, up to Bit 63 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 64 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.1.2 XMRENoSplitCompressionInfo<2>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<2>**, as specified in section 2.5.2.24.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.24). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 2.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 2) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((2) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

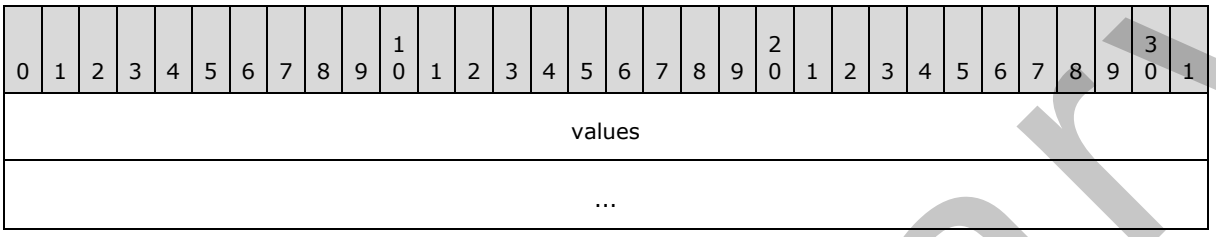
Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the

value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 62, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 62))

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 2 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 1, the second value occupies Bits 2 through 3, and so on.

The *startBit* offset followed the sequence of Bit 0, 2, 4, and so on, up to Bit 62 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 32 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.1.3 XMRENoSplitCompressionInfo<3>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<3>**, as specified in section 2.5.2.25.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.25). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 3.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 3) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((3) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

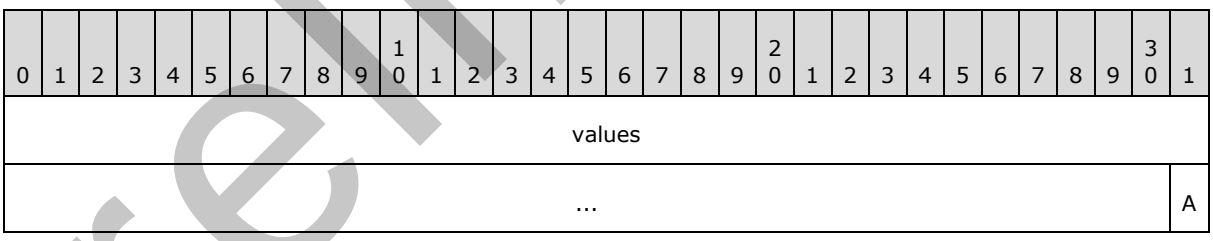
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 61, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 61))

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 3 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 2, the second value occupies Bits 3 through 5, and so on.

The *startBit* offset followed the sequence of Bit 0, 3, 6, and so on, up to Bit 60 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various

masking effects. For this compression algorithm, at most 21 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.1.4 XMRENoSplitCompressionInfo<4>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<4>**, as specified in section 2.5.2.26.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.26). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 4.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 4) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((4) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the

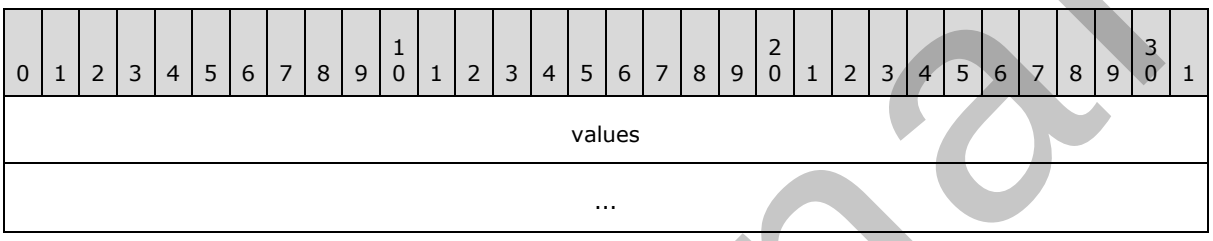
second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 60, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 60))

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 4 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 3, the second value occupies Bits 4 through 7, and so on.

The *startBit* offset followed the sequence of Bit 0, 4, 8, and so on, up to Bit 60 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 16 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.1.5 XMRENoSplitCompressionInfo<5>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<5>**, as specified in section 2.5.2.27.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.27). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the

resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 5.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 5) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((5) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

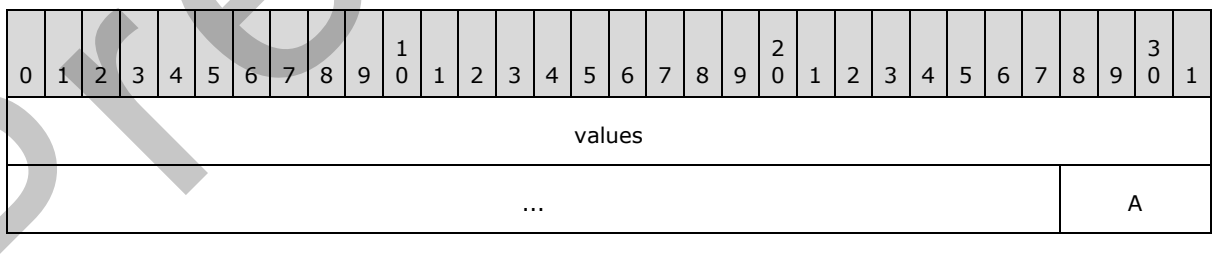
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 59, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 59))

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 5 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 4, the second value occupies Bits 5 through 9, and so on.

The *startBit* offset followed the sequence of Bit 0, 5, 10, and so on, up to Bit 55 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bit 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 12 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value

A (4 bits): The padding.

2.7.1.6 XMRENoSplitCompressionInfo<6>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<6>**, as specified in section 2.5.2.28.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.28). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 6.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 6) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((6) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

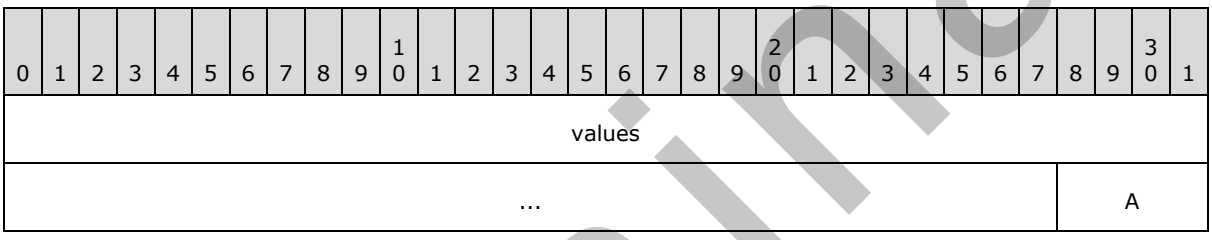
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 58, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 58))

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 6 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 5, the second value occupies Bits 6 through 11, and so on.

The *startBit* offset followed the sequence of Bit 0, 6, 12, and so on, up to Bit 54 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bit 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 10 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value

A (4 bits): The padding.

2.7.1.7 XMRENoSplitCompressionInfo<7>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<7>**, as specified in section 2.5.2.29.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more

details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.29). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 7.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 7) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((7) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

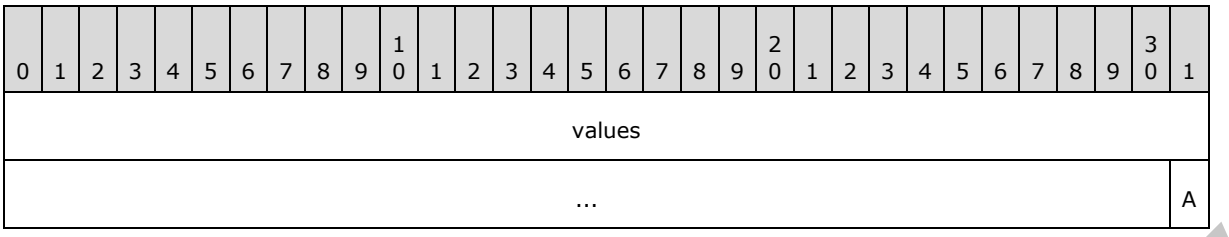
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 57, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 57))

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 7 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 6, the second value occupies Bits 7 through 13, and so on.

The *startBit* offset followed the sequence of Bit 0, 7, 14, and so on, up to Bit 56 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most nine values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.1.8 XMRENoSplitCompressionInfo<8>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<8>**, as specified in section 2.5.2.30.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

The minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.30). The result of this subtraction (*idVal* - *Min*) is then copied into the storage at that particular bit offset. The offset is a multiple of 8.

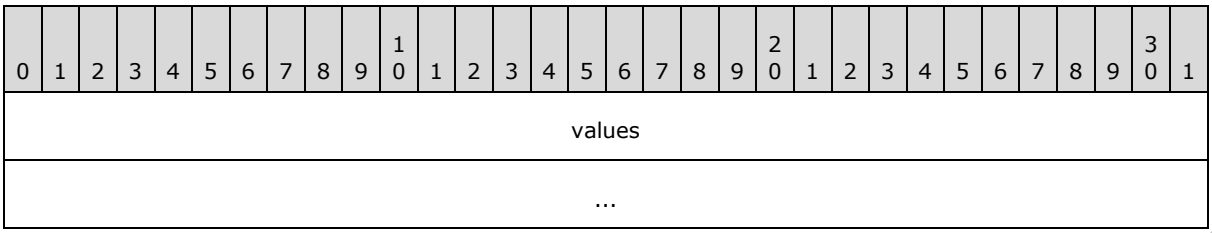
The following pseudocode illustrates this process:

SET compressedStorage at offset = (idVal - Min) WHERE offset is multiple of 8

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompressing values from this method consists of a simple reversal of the compression process.

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 1 byte, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 7, the second value occupies Bits 8 through 15, and so on.

The *startBit* offset followed the sequence of Bit 0, 8, 16, and so on, up to Bit 56 as the data was compressed into the 64-bit storage area. For this compression algorithm, at most eight values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

No padding exists in the compressed storage area because the number of necessary bits is a multiple of 8. However, if the entire storage area is not used, the remaining storage area is set to zero. For example, if only three values are placed into the compressed storage area, all the remaining bits, from Bit 24 through 63, will be zero.

2.7.1.9 XMRENoSplitCompressionInfo<9>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<9>**, as specified in section 2.5.2.31.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.31). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 9.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 9) is multiplied by the storage data

size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((9) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

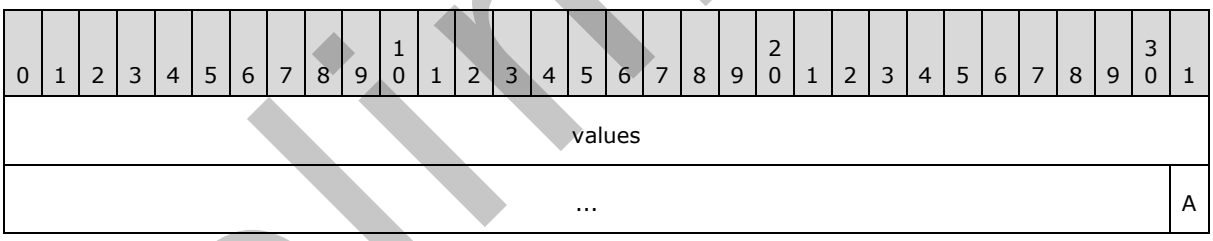
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 55, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 55))

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 9 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 8, the second value occupies Bits 9 through 17, and so on.

The *startBit* offset followed the sequence of Bit 0, 3, 6, and so on, up to Bit 54 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits, are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most seven values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.1.10 XMRENoSplitCompressionInfo<10>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<10>**, as specified in section 2.5.2.32.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.32). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 10.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 10) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((10) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

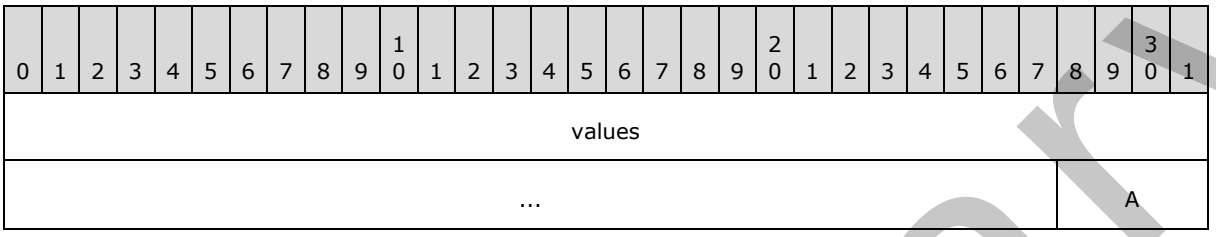
Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the

value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 54, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 54))

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 10 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 9, the second value occupies Bits 10 through 19, and so on.

The *startBit* offset followed the sequence of Bit 0, 10, 20, and so on, up to Bit 50 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bit 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most six values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value

A (4 bits): The padding.

2.7.1.11 XMRENoSplitCompressionInfo<12>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<12>**, as specified in section 2.5.2.33.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.33). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 12.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 12) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((12) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

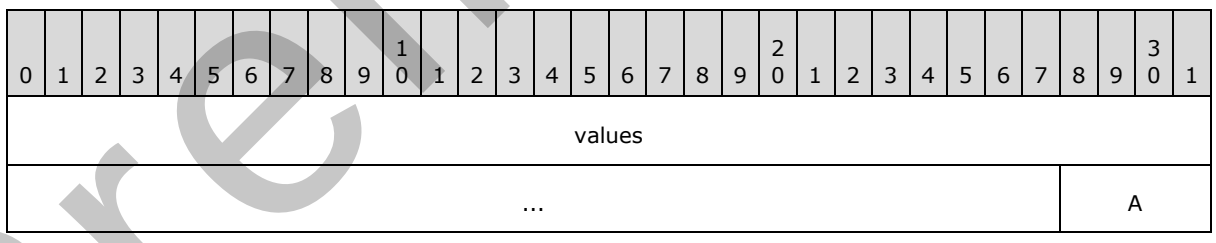
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 52, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 52))

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 12 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 11, the second value occupies Bits 12 through 23, and so on.

The *startBit* offset followed the sequence of Bit 0, 12, 24, and so on, up to Bit 48 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bit 60 through 63),

any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most five values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value

A (4 bits): The padding.

2.7.1.12 XMRENoSplitCompressionInfo<16>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<16>**, as specified in section 2.5.2.34.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

The minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.34). The result of this subtraction (*idVal* - *Min*) is then copied into the storage at that particular bit offset. The offset is a multiple of 16.

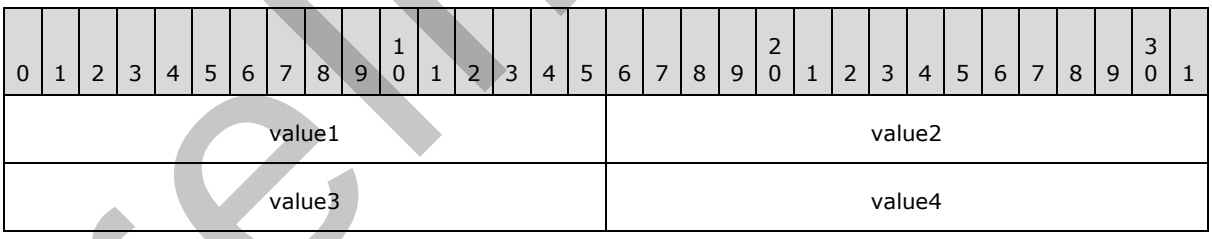
The following pseudocode illustrates this process:

SET compressedStorage at offset = (idVal - Min) WHERE offset is multiple of 16

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompressing values from this method consists of a simple reversal of the compression process.

The following diagram shows the compressed data values.



value1 (2 bytes): The first value in the sequence.

value2 (2 bytes): The second value in the sequence.

value3 (2 bytes): The third value in the sequence.

value4 (2 bytes): The fourth value in the sequence.

In the sequence, **value1** occupies Bits 0 through 15, **value2** occupies Bits 16 through 31, and so on. The *startBit* offset followed the sequence of Bit 0, 8, 16, and so on, up to Bit 56 as the data was compressed into the 64-bit storage area. For this compression algorithm, at most four values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value. No padding exists in the compressed storage area because the number of necessary bits is a multiple of 16. However, if the entire storage area is not used, the remaining storage area is set to zero. For example, if only three values are placed into the compressed storage area, all the remaining bits, from Bit 48 through 63, will be zero.

2.7.1.13 XMRENoSplitCompressionInfo<21>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<21>**, as specified in section 2.5.2.35.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into three phases

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.35). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 21.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the compressed storage, *compressedStorage*, is combined through a bitwise AND operation with a masking value, resulting in a masked off *compressedStorage* that is named *maskedStorage*. The masking value, named *maskArrayValue*, is found in the masking array named *maskArray*, as specified in section 5. The index into this array to obtain the correct masking value is calculated in the following manner: the bit count (in this case, 21) is multiplied by the storage data size (64 bits), and then a bitwise OR operation is performed on the multiplication result and the current offset into the storage area (*startBit*).

The following pseudocode illustrates this phase:

SET maskArrayValue = maskArray at index [((21) MULTIPLY (64)) BITWISE_OR (startBit)]

SET maskedStorage = (compressedStorage) BITWISE_AND (maskArrayValue)

In Phase 3, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the masked and compressed storage (*maskedStorage*). This operation generates the final result, named *maskedStorageWithValue*.

The following pseudocode illustrates this phase:

SET maskedStorageWithValue = (maskedStorage) BITWISE_OR (shiftedVal)

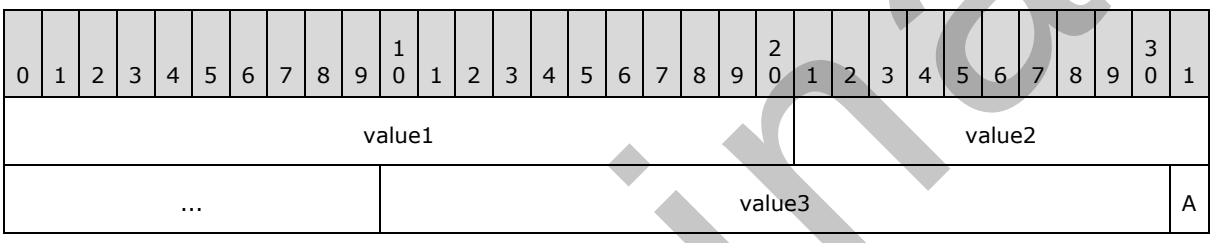
Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompression does not require the use of the mask. One way to decompress a value that has been compressed with this compression method is to right bit shift the masked storage containing the value by the current offset into the storage, perform a bitwise AND operation on the result of the value 0xFFFFFFFFFFFFFFFF right bit shifted by 43, and then add *Min*.

Using the same definitions as earlier, the following pseudocode illustrates one way to decompress and retrieve the original compressed value from the storage:

SET idVal = Min + ((maskedStorageWithValue RIGHT_BITSHIFT startBit) BITWISE_AND (0xFFFFFFFFFFFFFFFF RIGHT_BITSHIFT 43))

The following diagram shows the compressed data values.



value1 (21 bits): The first value in the sequence.

value2 (21 bits): The second value in the sequence.

value3 (21 bits): The third value in the sequence.

In the sequence, **value1** occupies Bits 0 through 20, **value2** occupies Bits 21 through 41, and so on. The *startBit* offset followed the sequence of Bit 0, 21, and 42 as the data was compressed into the 64-bit storage area. For this compression algorithm, at most three values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects.

A (1 bit): The padding.

2.7.1.14 XMRENoSplitCompressionInfo<32>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMRENoSplitCompressionInfo<32>**, as specified in section 2.5.2.36.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For

more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

The minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.36). The result of this subtraction (*idVal* - *Min*) is then copied into the storage at the particular bit offset. The offset is a multiple of 32.

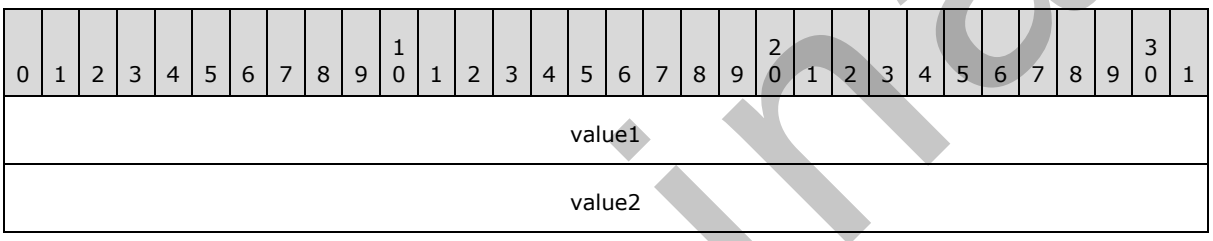
The following pseudocode illustrates this process:

SET compressedStorage at offset = (idVal - Min) WHERE offset is multiple of 32

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

Decompressing values from this method consists of a simple reversal of the compression process.

The following diagram shows the compressed data values.



value1 (4 bytes): The first value in the sequence.

value2 (4 bytes): The second value in the sequence.

In the sequence, **value1** occupies Bits 0 through 31, and **value2** occupies Bits 32 through 63. The *startBit* offset followed the sequence of Bit 0 and then Bit 32 as the data was compressed into the 64-bit storage area. For this compression algorithm, at most two values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value. No padding exists in the compressed storage area because the number of necessary bits is a multiple of 32. However, if the entire storage area is not used, the remaining storage area is set to zero. For example, if only one value is placed into the compressed storage area, all the remaining bits, from Bit 32 through Bit 63, will be zero.

2.7.2 XM123 Compression Algorithm

XM123 compression is used only in the special situation where the internally generated **RowNumber** column (section 2.3.4) is serialized to disk. This column follows the same file format as any column data storage file (section 2.3.1).

The storage area for the compressed value MUST be zeroed out prior to compressing the value and storing it in that area.

2.7.2.1 XM123CompressionInfo

This compression is never used standalone, even though it could be referenced in the XML metadata in what might appear at first look to be a standalone manner within the XML metadata, see section 2.5.2.37. It is always used as part of an XMHybridRLE compression and only for a particular special case. Please see section 2.7.3.16.

2.7.3 XMHybridRLE Compression Algorithms

XMHybridRLE compression algorithms use two forms of compression in combination: run length encoding (RLE) and bit packing. As a result, these compression algorithms use two segments to represent all the values. The segments are referred to here as the *RLE segment*, or *primary segment*, and the *subsegment*, or *bit-packing subsegment*. The RLE segment contains a possible mix of RLE entries and bit-packing entries, the latter of which refer to bit-packed values that follow in the subsegment.

The first type of compression that is used in this hybrid compression is RLE. RLE is used only on appropriate data items—those that repeat often enough to make RLE an efficient compression choice. In RLE compression, two 4-byte values are used that together comprise the first type of entry in the RLE segment—the RLE entry. The first value of the RLE entry is the data value. The second value is the repeat count, which is the number of times that the data value repeats in a continuous sequence. The repeat count MUST be equal to or greater than 64. For an RLE run to be generated by the system, at least 64 consecutive items (that is, 64 identical records) MUST exist in that run of data. Otherwise, the individual items will be bit packed by using the chosen bit-packing algorithm in the related subsegment.

The second type of entry in the RLE segment is the bit-packing entry. The bit-packing entry also contains two 4-byte values. The first value is a negated 1-based offset into the subsegment data, and the second value is the count of the number of values that follow in the subsegment. The offset value is represented as the negative of itself to clearly distinguish it from the RLE entries. The bit-packed values exist in a separate bit-packing subsegment that follows the RLE segment.

The first bit-packing entry uses –1 as the first value (to indicate that it is the first data value in the subsegment) and then has the count of bit-packing items as the second value. Any subsequent bit-packing entries also have a negative offset value as the first value. This negative offset value is calculated by taking the previous negative bit-packing entry offset value, and subtracting from it the bit-packing entry count of that same entry.

Any number of RLE entries and bit-packing entries can exist in the RLE segment, and in any order. It is also possible for the RLE segment to have no RLE entries or no bit-packing entries in the RLE segment. In very rare cases, it is possible for both the segment and subsegment to be empty. For more information about segment minimum and maximum row sizes, see section 2.3.1.1.3.

The bit-packed compression values follow the RLE segment. The bit-packed values are in the same order that they are referred to here but within their own, bit-packed subsegment. All the bit-packed values that are referenced (as bit-packing entries) within the RLE segment use the same type of bit-packing compression because only one type of bit-packing compression is allowed per segment. In other words, after the RLE segment, the bit-packing subsegment that is associated with the RLE segment uses a single bit-packing algorithm, such as that for **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>** in which each bit-packed entry is compressed by using 3 bits. For more information about the column data storage file format, see section 2.3.1.

For a diagrammed example of RLE entries mixed with bit-packing entries, see section 2.7.3.1.

After the end of all the data entries (both the RLE entries and the bit-packing entries) in the RLE segment, any remaining storage area allocated for the RLE segment is zeroed out. The size of this padding depends on the amount of storage allocated and the amount used. These amounts can be found in the metadata information for this compression (see section 2.5.2.38.1).

Following the end of this segment (including any padding at the end of the segment), the next segment begins. This segment is called the *subsegment of the XMHybridRLE compression* and contains the selected bit-packing compression sequence. For more information about the column data storage file format, which is the file format that uses this hybrid compression dual segment layout, see section 2.3.1.

The second part of the hybrid compression, the subsegment, involves using a bit-packing compression algorithm. This part is used for data items that are not suitable for RLE compression, and it uses a form of range encoding bit-packing compression in which the data to be stored (after that data has been offset by a minimum value) is compacted into a specified number of bits. The bit size that is chosen determines the bit-packing algorithm that is used to compress the data. The values that are bit packed in this segment conform to the sequence that is specified in the RLE segment.

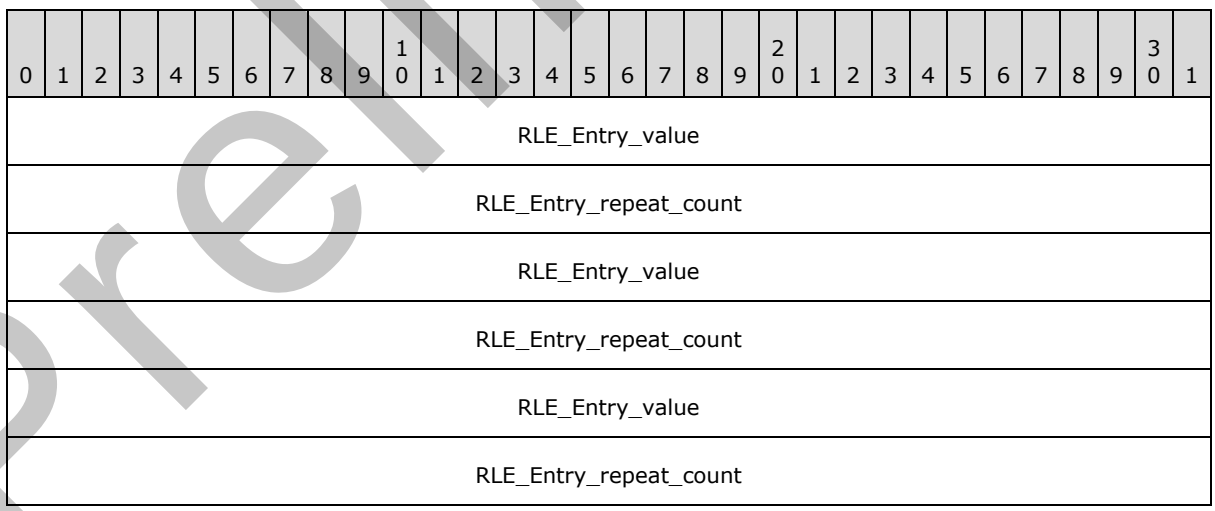
Despite any name similarity, the range encoding bit-packing algorithm that is used in the hybrid compression is not necessarily the same as the range encoding bit-packing algorithm that is used in the nonhybrid case.

The storage area for the compressed value, whether RLE or bit packed, MUST be zeroed out prior to compressing the value and storing it in that area.

2.7.3.1 Conceptual Overview of RLE Entries and Bit-Packing Entries

The following bit diagram shows a sequence of RLE entries and bit-packing entries as described in section 2.7.3. This conceptual overview is long to clearly show the pattern of negative offset values used in the bit-packing entries. The sequence includes several RLE entries and an occasional bit-packing entry.

As mentioned in section 2.7.3, any mix of RLE entries and bit-packing entries can exist. So in this conceptual overview, the number of RLE entries and the number of bit-packing entries is arbitrary. The count of values that is shown in each bit-packing entry can also be larger or smaller than that shown in this example. The count of values that is shown for an RLE entry MUST be greater than or equal to the required minimum of 64.



Bit_Packing_Entry_offset
Bit_Packing_Entry_count
RLE_Entry_value
RLE_Entry_repeat_count
RLE_Entry_value
RLE_Entry_repeat_count
Bit_Packing_Entry_offset
Bit_Packing_Entry_count
RLE_Entry_value
RLE_Entry_repeat_count
Bit_Packing_Entry_offset
Bit_Packing_Entry_count
RLE_Entry_value
RLE_Entry_repeat_count
RLE_Entry_value
RLE_Entry_repeat_count
Bit_Packing_Entry_offset
Bit_Packing_Entry_count
padding (variable)
...

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

Bit_Packing_Entry_offset (4 bytes): The offset into the subsegment. This offset is –1. Note that the first offset equals the first data value in the subsegment.

Bit_Packing_Entry_count (4 bytes): The number of values that will be using the subsegment bit-packing algorithm. This count is 5.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

Bit_Packing_Entry_offset (4 bytes): The offset into the subsegment. This offset is –6. Note that –6 = –1 – 5 (the values from the previous bit-packing entry).

Bit_Packing_Entry_count (4 bytes): The number of values that will be using the subsegment bit-packing algorithm. This count is 20.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

Bit_Packing_Entry_offset (4 bytes): The offset into the subsegment. This offset is –26. Note that –26 = –6 – 20 (the values from the previous bit-packing entry).

Bit_Packing_Entry_count (4 bytes): The number of values that will be using the subsegment bit-packing algorithm. This count is 10.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

RLE_Entry_value (4 bytes): A value compressed with RLE.

RLE_Entry_repeat_count (4 bytes): The number of times that the preceding value appears sequentially in this RLE run.

Bit_Packing_Entry_offset (4 bytes): The offset into the subsegment. This offset is –36. Note that –36 = –26 – 10 (the values from the previous bit-packing entry).

Bit_Packing_Entry_count (4 bytes): The number of values that will be using the subsegment bit-packing algorithm. This count is 2.

padding (variable): The padding, which is unused but MUST be filled with zeros. The size, in bytes, is a multiple of 4.

2.7.3.2 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<1>>**, as specified in section 2.5.2.39. The RLE portion of this compression follows the format that is described in section 2.7.3. The bit packing sub-segment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.39). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 1.

The following pseudocode illustrates this phase:

```
SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit
```

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

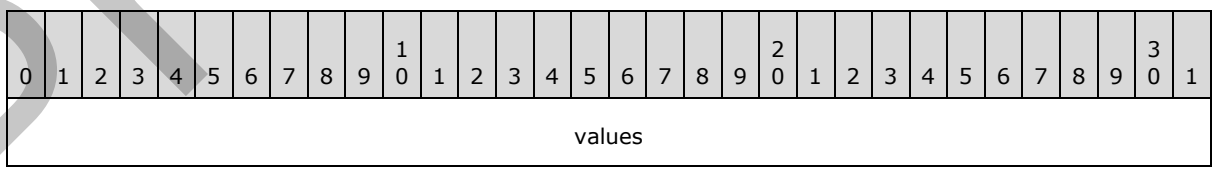
The following pseudocode illustrates this phase:

```
SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)
```

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.1.

The following diagram shows the compressed data values.



…

values (8 bytes): The set of values, each occupying 1 bit, in sequence and ordered low to high. In the sequence, the first value occupies Bit 0, the second value occupies Bit 1, and so on.

The *startBit* offset followed the sequence of Bit 0, 1, 2, and so on, up to Bit 63 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 64 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.3.3 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<2>>**, as specified in section 2.5.2.40. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.40). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 2.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

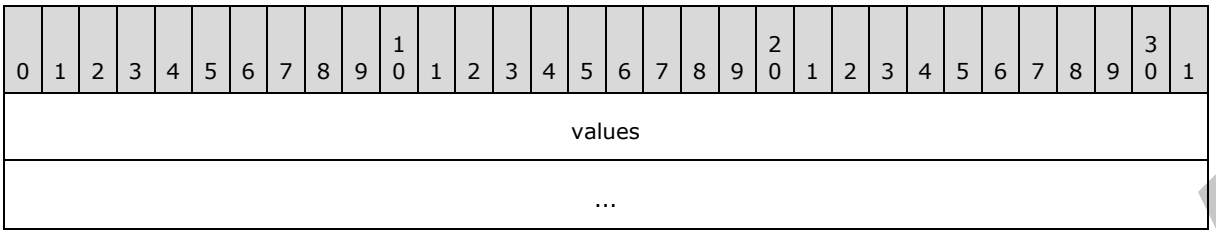
In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.2.

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 2 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 1, the second value occupies Bits 2 through 3, and so on.

The *startBit* offset followed the sequence of Bit 0, 2, 4, and so on, up to Bit 62 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 32 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.3.4 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<3>>**, as specified in section 2.5.2.41. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.41). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 3.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

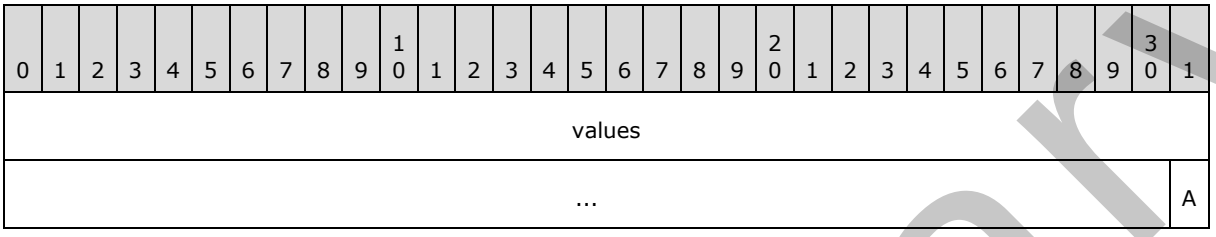
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.3.

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 3 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 2, the second value occupies Bits 3 through 5, and so on.

The *startBit* offset followed the sequence of Bit 0, 3, 6, and so on, up to Bit 60 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 21 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.3.5 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<4>>**, as specified in section 2.5.2.42. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.42). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the

resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 4.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

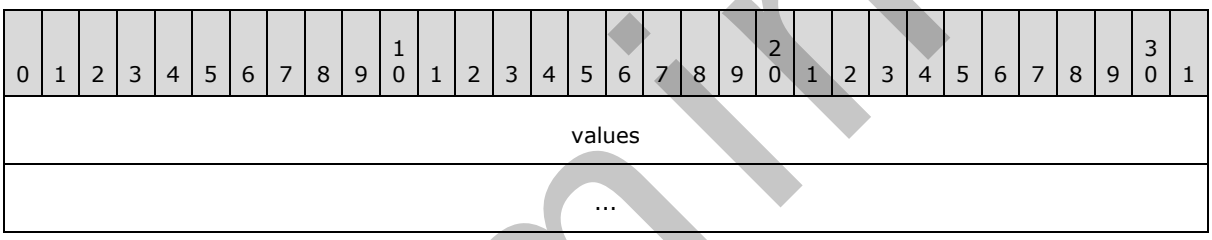
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.4.

The following diagram shows the compressed data values.



values (8 bytes): The set of values, each occupying 4 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 3, the second value occupies Bits 4 through 7, and so on.

The *startBit* offset followed the sequence of Bit 0, 4, 8, and so on, up to Bit 60 as the data was compressed into the 64-bit storage area. Any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 16 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

2.7.3.6 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<5>>**, as specified in section 2.5.2.43. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each

compressedStorage data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.43). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 5.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

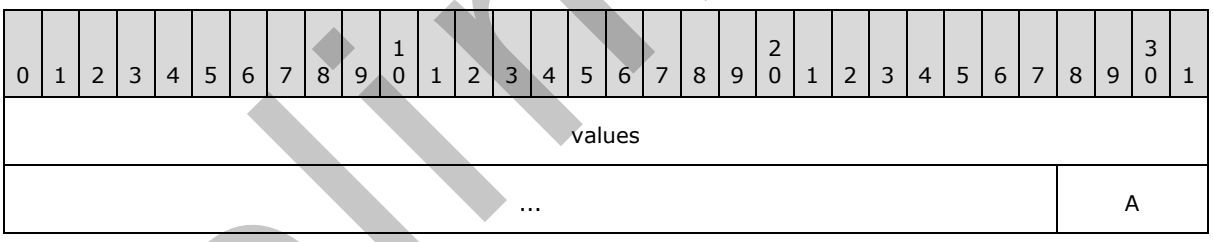
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.5.

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 5 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 4, the second value occupies Bits 5 through 9, and so on.

The *startBit* offset followed the sequence of Bit 0, 5, 10, and so on, up to Bit 55 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bits 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 12 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (4 bits): The padding.

2.7.3.7 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<6>>**, as specified in section 2.5.2.44. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.44). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 6.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

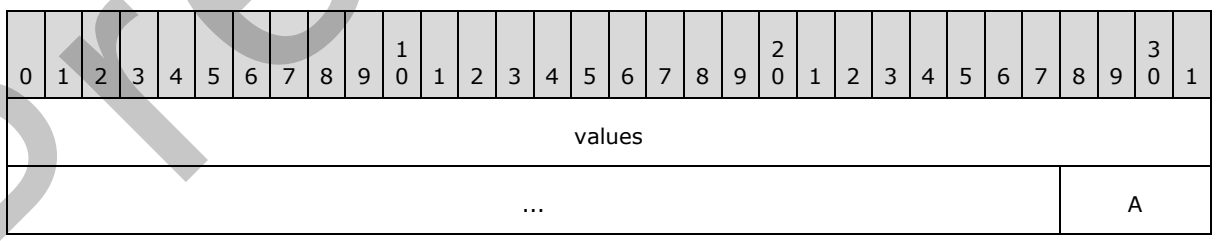
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.6.

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 6 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 5, the second value occupies Bits 6 through 11, and so on.

The *startBit* offset followed the sequence of Bit 0, 6, 12, and so on, up to Bit 54 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bits 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 10 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (4 bits): The padding.

2.7.3.8 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<7>>**, as specified in section 2.5.2.45. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.45). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 7.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

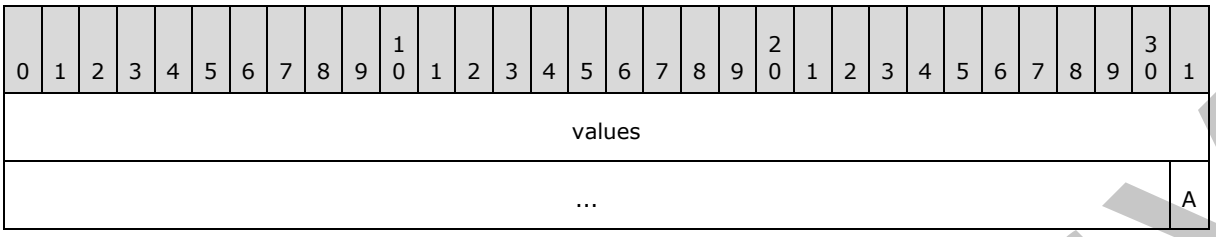
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.7.

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 7 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 6, the second value occupies Bits 7 through 13, and so on.

The *startBit* offset followed the sequence of Bit 0, 7, 14, and so on, up to Bit 56 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 9 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.3.9 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<8>>**, as specified in section 2.5.2.46. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is identical to the 8-bit compression format that is used in XMRENoSplit compression, as specified in section 2.7.1.8. For metadata information, see section 2.5.2.46.

2.7.3.10 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<9>>**, as specified in section 2.5.2.47. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.47). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 9.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

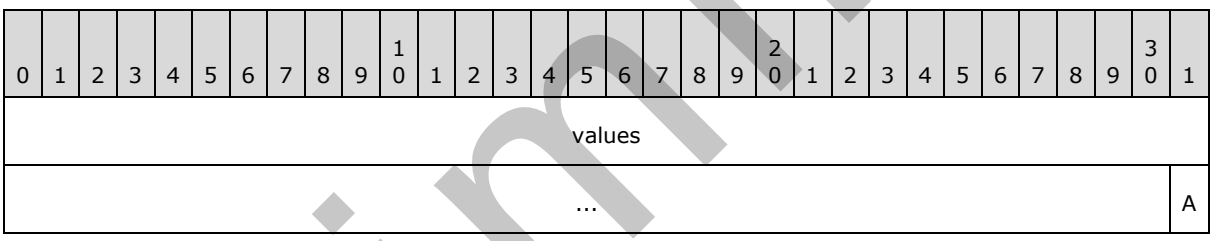
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.9.

The following diagram shows the compressed data values.



values (63 bits): The set of values, each occupying 9 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 8, the second value occupies Bits 9 through 17, and so on.

The *startBit* offset followed the sequence of Bit 0, 9, 18, and so on, up to Bit 54 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 7 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.3.11 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<10>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class**

XMRENoSplitCompressionInfo<10>>, as specified in section 2.5.2.48. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.48). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 10.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

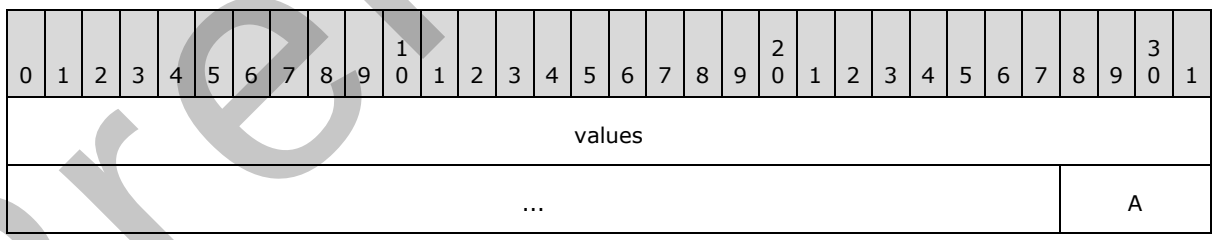
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.10.

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 10 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 9, the second value occupies Bits 10 through 19, and so on.

The *startBit* offset followed the sequence of Bit 0, 10, 20, and so on, up to Bit 50 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bits 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 6 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (4 bits): The padding.

2.7.3.12 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<12>>**, as specified in section 2.5.2.49. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.49). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 12.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

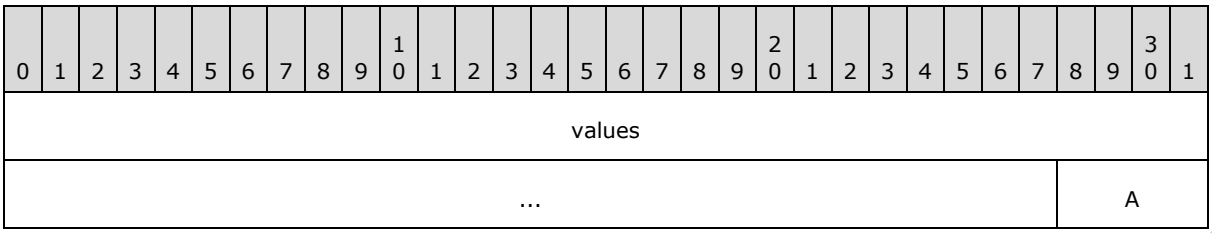
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.11.

The following diagram shows the compressed data values.



values (60 bits): The set of values, each occupying 12 bits, in sequence and ordered low to high. In the sequence, the first value occupies Bits 0 through 11, the second value occupies Bits 12 through 23, and so on.

The *startBit* offset followed the sequence of Bit 0, 12, 24, and so on, up to Bit 48 as the data was compressed into the 64-bit storage area. In addition to the end bits (Bits 60 through 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most 5 values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (4 bits): The padding.

2.7.3.13 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<16>>**, as specified in section 2.5.2.50. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is identical to the 16-bit compression format that is used in XMRENoSplit compression, as specified in section 2.7.1.12. For metadata information, see section 2.5.2.50.

2.7.3.14 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<21>>**, as specified in section 2.5.2.51. The RLE portion of this compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is as follows.

Assume that each value to be compressed is named *idVal*. Each *idVal* value will be compressed into the compressed storage in the following manner: The maximum size of the value to be compressed MUST NOT exceed 32 bits. Assume that the resulting compressed storage area data value is named *compressedStorage*. The entire compressed storage area is a multiple of 64 bits, so each *compressedStorage* data value (the section of storage that is being written to) is 64 bits in size. For more details about the actual values that are compressed by using this type of compression, see section 2.2.2.3.2.

To simply this explanation, the process is divided into two phases.

In Phase 1, the minimum value (*Min*) is subtracted from the value to be compressed. The value of *Min* can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.51). The result of this subtraction (*idVal* - *Min*) is then left bit shifted by the current bit offset into the storage. Assume that this offset is named *startBit*, and the

resulting value after the bit shifting and subtraction is named *shiftedVal*. The offset (*startBit*) is a multiple of 21.

The following pseudocode illustrates this phase:

SET shiftedVal = (idVal - Min) LEFT_BITSHIFT startBit

In Phase 2, the data value is placed into the compressed storage. A bitwise OR operation is performed on the value (*shiftedVal*) and the compressed storage (*compressedStorage*). This operation generates the final result, named *storageWithValue*.

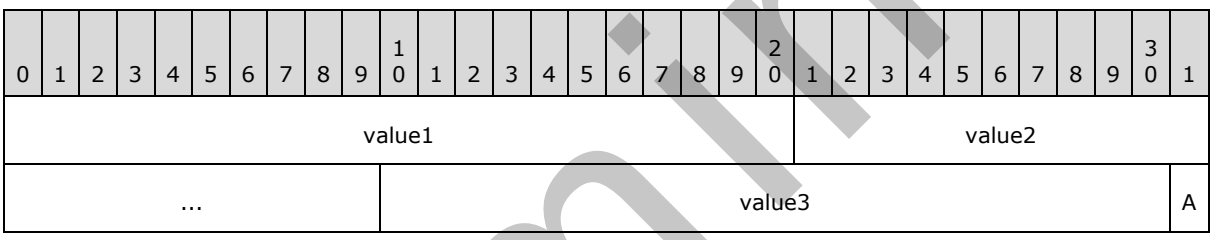
The following pseudocode illustrates this phase:

SET storageWithValue = (compressedStorage) BITWISE_OR (shiftedVal)

Within the value, the order of the value's bits is little-endian. Within the compressed storage, however, individual values are placed low to high, with the first value occupying the lowest bits, the second value occupying the next-lowest bits, and so on. All the file formats use little-endian values. It is only the way individual values are ordered within the file that is being emphasized here, because that specific ordering is required.

The same decompression method that is used in XMRENoSplit compression can be used here. For more information, see section 2.7.1.13.

The following diagram shows the compressed data values.



value1 (21 bits): The first value in the sequence.

value2 (21 bits): The second value in the sequence.

value3 (21 bits): The third value in the sequence.

In the sequence, **value1** occupies Bits 0 through 20, **value2** occupies Bits 21 through 41, and so on. The *startBit* offset followed the sequence of Bit 0, 21, and 42 as the data was compressed into the 64-bit storage area. In addition to the end bit (Bit 63), any unused bits are thus only padding, and the value of the padding depends on the result of the various masking effects. For this compression algorithm, at most three values exist in the 64-bit compressed storage value, and the process will begin again at Offset 0 with the next *compressedStorage* value.

A (1 bit): The padding.

2.7.3.15 XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XMRENoSplitCompressionInfo<32>>**, as specified in section 2.5.2.52. The RLE portion of this

compression follows the format described in section 2.7.3. The bit-packing subsegment portion of the compression is identical to the 32-bit compression format that is used in XMRENoSplit compression, as specified in section 2.7.1.14. For metadata information, see section 2.5.2.52.

2.7.3.16 XMHybridRLECompressionInfo<class XM123CompressionInfo>

This section specifies the compression algorithm that is used when the compression metadata specifies the compression to be of type **XMHybridRLECompressionInfo<class XM123CompressionInfo>**, as specified in section 2.5.2.53. The RLE portion of this compression follows the format described in section 2.7.3. However, only the bit-packing entry is used in the RLE portion, except that in this case, the value that is associated with the bit-packing entry (which typically contains the subsegment offset and the count of items that will follow in the bit-packing compression subsegment) is used differently here. The offset is set to –1, as expected for the first bit-packing entry; however, the count now represents the number of rows that exist in the segment. The rest of the RLE segment is unused and padded with zeros. The bit-packing subsegment portion of the compression is as follows.

Nothing is stored by using bit packing in this situation, except for 8 bytes that are set to zero to indicate that the subsegment itself, which contains the bit-packing compression, is zero bytes in size. For more information about the file format layout, including how the sizes of different sections of the file are indicated, see section 2.3.1.1.

The minimum value (*Min*) is not used in the actual compression. It is used when recreating the correct offset value for the row number. The *Min* value can be found in the XML metadata file that is associated with the data file containing the compressed data (see section 2.5.2.53). The *Min* value that is stored in the XML metadata is actually an offset. Together, the number of rows for the segment and the *Min* value for the segment allow the creation of the correct overall row number for the entire column, which could be composed of multiple segments. The *Min* value is a 32-bit value.

The reason is that this compression is used for the internally created **RowNumber** column (section 2.3.4), and that column is zero indexed through the total number of rows in the column.

For example, the first *Min* value that is stored, for the first segment, is zero. Therefore, the range of rows for that segment is from *Min* through *Min* plus the total number of rows for that segment minus one. The minus one is because of zero indexing. For a total of five rows in the first segment, the indexing ranges from *Min* through *Min* + 4, where the offset is zero. For the next segment, the offset is the next index beyond the last value in the previous segment. Again, if five rows exist in the next segment, the index will range from *Min* + 5 through *Min* + 9, where the offset (the value that is stored) is 5. Moving from one segment to the next, the row numbers thus continue sequentially without any breaks.

For this compression, the maximum that can be indexed is 2 billion (hexadecimal 0x77359400). This number includes the *Min* value. In other words, the *Min* value plus the last row number is, at most, 2 billion.

2.7.4 Huffman Compression

Huffman compression is the only type of compression that is used when strings are stored and compressed in a dictionary file. String data is not stored in the column data storage files but in a dictionary file—specifically, in an **XM_TYPE_STRING** dictionary file. (For more information about **XM_TYPE_STRING** dictionaries, see section 2.3.2.1.2. For more information about the per-page string store information in particular, see section 2.3.2.1.2.4.) The strings might be compressed. BLOBs are also stored in **XM_TYPE_STRING** dictionaries after they have been encoded by means of base64 encoding. Therefore, stored BLOBs are treated as single character set strings and can be successfully compressed by using Huffman compression, as well.

If any of the strings are compressed, the compression used is always classic Huffman encoding. Although Huffman encoding has many forms, the Huffman tree and algorithm—especially of classic Huffman encoding—is well known and will not be discussed in detail here.

In short, in Huffman encoding, the algorithm determines the probability that a particular symbol from the assigned Huffman alphabet will be encountered in the stream of symbols being encoded. Using this statistical knowledge of the frequencies of different symbols in the alphabet being used, a binary tree of $2 \times N - 1$ nodes can be constructed, where N represents the number of symbols used in the text being encoded. (Note that N can be the size of the Huffman alphabet or less than the size of the Huffman alphabet being used.) Symbols that have the greatest frequency of occurrence in the text to be encoded are assigned higher positions in the binary tree than symbols that have a lower frequency of occurrence. The binary tree that is constructed does not need to be a balanced binary tree.

2.7.4.1 Huffman Implementation Constraints

In the implementation of Huffman encoding that is used here, further constraints and guidelines exist and are explained in the following subsections.

2.7.4.1.1 Classical Unbalanced Huffman Tree

As stated in section 2.7.4, in this implementation, Huffman tree creation employs a traditional, classical approach that uses the frequency of occurrence of a symbol in the text to be encoded to build the binary tree hierarchy. Additionally, this method does not require that the tree be balanced.

Therefore, encodings MUST follow a classic Huffman approach, but the tree that is generated does not need to be a balanced tree.

2.7.4.1.2 Minimum and Maximum Codeword Sizes

This implementation of Huffman encoding does not support the encoding of symbols that are either zero bits or 1 bit in length, but it does limit the maximum size of codewords. Therefore, it is important that for this Huffman implementation, codeword lengths be limited to a range that has a minimum and a maximum value.

The length of each codeword MUST be greater than or equal to 2 bits and less than or equal to 15 bits.

However, the decode bits value (**uiDecodeBits**) that is specified in the dictionary file MUST NOT exceed 12—even when more than 12 bits are used for codewords—otherwise, an error will occur. (For more information about **uiDecodeBits**, see section 2.3.2.1.2.5.)

The system creates lookup tables that handle codewords of a size only up to 12 bits. Any codewords that are longer than 12 bits in length require traditional Huffman tree traversal techniques for decoding. Therefore, when setting the decode bits value (**uiDecodeBits**) in the **XM_TYPE_STRING** hash data dictionary file (section 2.3.2.1.2), consider the following guidelines for deciding what value to set as the **uiDecodeBits** size.

First, if the longest codeword is 8 bits or less, it is suggested that **uiDecodeBits** be set to that longest codeword value.

Second, if the longest codeword is greater than 12 bits, it is suggested that **uiDecodeBits** be set to 12. Setting **uiDecodeBits** to any value greater than 12 will result in an error.

Third, if the longest codeword is greater than or equal to 9 but less than 12 bits, it is suggested that **uiDecodeBits** be set to the codeword size that includes 99 percent of all the compressed bits in the encoded text (meaning the encoded text for that Huffman encoding on that dictionary page).

The total number of compressed bits can be determined by summing over all codewords and using the value equal to the frequency of each codeword multiplied by the number of bits in that codeword, as shown in the following pseudocode:

FOR all Codewords

SET totalCompressedBits = totalCompressedBits + (codewordFrequency MULTIPLY numberOfBitsInCodeword)

END FOR

The smallest codeword bit size that corresponds to 99 percent coverage is the value to use for **uiDecodeBits** to help ensure that almost all of the bits are found via the fast lookup tables and that only a few, infrequently used bytes are left out of the table.

2.7.4.1.3 Huffman Alphabet Size

The Huffman alphabet size that is used in this implementation is limited to 256 symbols or less and is byte oriented. This means that this Huffman encoding algorithm encodes bytes rather than actual characters. The algorithm has no real knowledge of the languages being used.

Some languages—those that fit into 128-bit character sets, such as ASCII—have a one-to-one relationship between a character and an encoded byte. Other languages require multiple bytes to encode a character. In both cases, the characters are simply encoded byte by byte in the Huffman tree without regard to how any of these single or multiple bytes relate to each other. In other words, the Huffman implementation has no knowledge of the meaning of the individual bytes and simply encodes each byte according to its frequency of occurrence in the string byte stream.

The encoded Huffman alphabet array that is contained within the dictionary page for those compressed strings reflects the byte's numeric value. This numeric value is used as the index into the array, and the content of the array index item is the codeword size that is used for that index (byte) value. A byte value is 8 bits and can therefore be represented as a numeric value from 0 through 255. If the codeword size that is used is zero, the meaning is that the index (byte) value was not used in the tree.

The actual frequencies used to generate the Huffman tree (and therefore the resulting codeword assignments) is not persisted to the dictionary file. Even so, the encoded array of bytes and their corresponding codeword sizes, and the knowledge that the minimum codeword size is required to be 2 bits, can be used to reconstruct the original, classic Huffman tree.

It is critical that the encoded Huffman alphabet array be correct and adhere to the criteria detailed here for this Huffman implementation. The array MUST encode the codeword lengths for each used alphabet character (byte value) as well as set unused elements to zero. Those codeword lengths MUST be from 2 through 15 bits, and the classic Huffman tree approach MUST be used when deciding symbol-to-codeword assignment. For an example of a Huffman tree implementation that uses these criteria, see section 2.7.4.2.

2.7.4.1.4 Single and Multiple Character Set Modes

This Huffman encoding implementation supports both single character set encoding and multiple character set encoding. For single character set encoding, where the entire text that is encoded uses one character set, the upper byte is not encoded but is instead stored separately. For more

information about setting single versus multiple character set mode, see section 2.3.2.1.2.4.2. During the decoding process, this character set value MUST be added back to each decoded symbol to recreate a valid character in the byte stream.

By contrast, in multiple character set encoding, where the text to be encoded contains two or more character sets that are intermixed in some way, both the upper and the lower byte MUST be encoded during Huffman encoding, but the decoding process does not require any character set byte to be added back to the stream. Therefore, encoding and decoding processes need to take these differences into account when deconstructing and reconstructing the characters.

A multiple byte character set does not necessarily use only the multiple character set mode. A multiple byte character set can use a single character set mode. It depends on whether the specified bytes (the first byte, the third, the fifth, and so on) in the byte stream are the same.

For example, in the Unicode version of ASCII, the first byte is the same as the third byte, which is the same as the fifth, and so on. This is because those bytes represent the character set and are each the upper byte of a 2-byte character. A single character set mode works fine in this situation. Those bytes, each alternating with the byte that actually contains the character code, can safely be pulled out of the byte stream and added back to each character byte later, during the decoding process.

However, if the character set uses multiple bytes per character, not counting the byte that designates the character set (the language), the first byte might not be the same as the third byte or the fifth byte. In such a case, using a multiple character set mode helps to ensure that each byte is encoded in the stream and that none are removed.

Even so, it still depends on the individual characters within that language. It is possible that the strings being compressed, even though they are multiple-byte characters, still have duplicate alternative bytes in the byte stream, mimicking a single byte character set. This can happen if only a few strings with only a few characters are being encoded. In this case, the single character set mode can be used and will not cause any data corruption or loss.

2.7.4.1.5 Huffman Information Provided in an XM_TYPE_STRING Dictionary

The **XM_TYPE_STRING** hash data dictionary file format (section 2.3.2.1.2) supports having multiple pages, either compressed or uncompressed. Each compressed page contains its own Huffman alphabet encoding array with the codeword sizes of the symbols, the character set mode, the character set value (in the case of single character set mode), the size of the longest codeword used for the internally generated lookup table, and the actual encoded bit stream for that page's strings. For more information about compressed pages, see section 2.3.2.1.2.4.2. A discussion about how these items are used will now ensue.

As already mentioned, the **XM_TYPE_STRING** dictionary file contains the information that is needed to decode the Huffman encoded strings for that page of that dictionary. This includes the decode bits value (**uiDecodeBits**), which represents a maximum codeword length that is used in the faster, internally generated lookup table. It does not mean that only codewords up to that maximum length were used in the Huffman tree. For example, if **uiDecodeBits** is 4, that does not mean that only codewords of a size up to 4 (that is, 2, 3, and 4) exist, but that typically no codewords are generated that are more than 4 bits in length. (For more information about **uiDecodeBits**, see section 2.3.2.1.2.5.)

For **uiDecodeBits** values less than or equal to 8, typically only codewords of that length or less were used, but for values greater than 8, the chances of having codewords greater than the **uiDecodeBits** value are more likely to occur. Codewords can actually go up to 15 bits in length. For more information, see section 2.7.4.1.2.

The dictionary also includes the encoded Huffman alphabet used. Again, this is an array of values, where the index of the array represents the symbol value (the numeric value of the byte being encoded), and the contents of that indexed item is the length of the codeword used for that symbol. In the dictionary file, this array is compacted down to use only 128 bytes, placing data in both the upper and lower bytes. Therefore, by treating the upper and lower bytes as individual elements, a Huffman alphabet array of 256 bytes can be generated. For more information about the layout of the stored variables just discussed, see section 2.3.2.1.2.4.2.

From this codeword length array, the knowledge of a classic Huffman tree, and the information detailed here regarding the constraints and guidelines that are used in the encoding process, a Huffman tree can be reconstructed. After that is accomplished, the codeword (bit sequence) for each byte symbol can be determined by walking the tree to that particular symbol.

At this point, the codewords for each byte symbol are used to decode the compressed strings (the bit stream). If the stream is using a single character set, the character set value (a byte value) is added back into the odd bytes (the first byte, the third, and so on) of the stream, and the actual byte stream of characters is then reconstructed from the encoded bit stream. For an example of this process, see section 2.7.4.2.

Note that the codewords (bit sequences) are encoded without breaks in the bit stream. Therefore, if one string takes 15 bits to fully encode (as a series of codewords that together total 15 bits), and the next string takes 10 bits, the resulting bit stream of encoded text will be 25 bits. No padding exists between strings.

The vector of record handle structures that is contained at the end of the dictionary file provides the page number and bit offset for each compressed string in the dictionary (or the page number and byte offset for uncompressed strings). Using those record handle structures, the continuous bit stream of compressed strings can be correctly divided before decoding them into a byte stream (see section 2.3.2.1.2.5).

2.7.4.2 Conceptual Overview of a Huffman Tree

To demonstrate how Huffman encoding is used, this section provides a conceptual overview. The field names **uiDecodeBits** and **encodeArray** come from the **XM_TYPE_STRING** hash data dictionary (section 2.3.2.1.2.5).

Assume that a data column containing two strings representing gender exists. The strings are "Female" and "Male" and appear to use the ASCII character set. Internally, they actually use Unicode because Unicode contains the ASCII character set. The strings are stored in a dictionary and are Huffman encoded. The dictionary page that contains the encoded strings has the character set (a value of zero, which indicates English) and indicates that the strings are using single character set Huffman mode. The page also indicates that the value of the lookup table decode bits (**uiDecodeBits**) is 3, which means that the characters in the strings were most likely encoded with 3-bit codewords and 2-bit codewords (see section 2.7.4.1.2). Single-bit codewords are not allowed.

The dictionary also has an encoded array (**encodeArray**) of the Huffman alphabet used. The indexes of this array correspond to the Huffman alphabet symbols (byte symbols), and the content of each indexed element indicates the codeword length that is used for that symbol (byte) in the encoded string. The symbols themselves are bytes, not characters, and vary in value from 0 through 255, which is also why the size of the Huffman alphabet array is fixed at 256 elements.

In this conceptual overview, almost all the array elements are equal to zero (that is, the codeword size equals zero), which means that those byte symbols are not used, except for the elements at the following indexes:

- The element at index 70 contains a value of 3.

- The element at index 77 contains a value of 3.
- The element at index 97 contains a value of 3.
- The element at index 101 contains a value of 2.
- The element at index 108 contains a value of 2.
- The element at index 109 contains a value of 3.

Note that the preceding index values (70, 77, 97, 101, 108, and 109) correspond to the ASCII numeric character values for the letters, 'F', 'M', 'a', 'e', 'l', and 'm'.

Because the strings are using English (ASCII), this behavior is expected because a one-to-one relationship exists between the byte value and the character value in ASCII (or in Unicode using this character set). This would not be the case if the character set used needed more than one byte per character.

The dictionary also has the encoded strings as a bit stream. Note that this is a bit stream, not a byte stream. In this case, the encoded bit stream will be decoded as the string "FemaleMale", which will then be a continuous byte (not bit) stream of decoded characters. Remember that the character set upper byte will be added back to each of these bytes. The bit stream itself is 25 bits in length with the following sequence '1000011111001001011100100'.

The following diagram shows a classic Huffman tree that is created by using the frequency of occurrence for each character. Frequencies (from the original string stream "FemaleMale") are shown in the diagram.

However, this tree also shows that symbols with 2-bit codewords are at one level (the top level that is allowed), symbols with 3-bit codewords are at the next level down, and so on through all the codeword values. In this case, only 2-bit and 3-bit codeword levels exist. The symbol byte values (101 and 108 on the 2-bit level, and the next four symbol byte values on the 3-bit level) are placed according to a classic Huffman implementation, where the lowest value goes to the leftmost node, the next-highest value goes to the right of that node, and so on across the level.

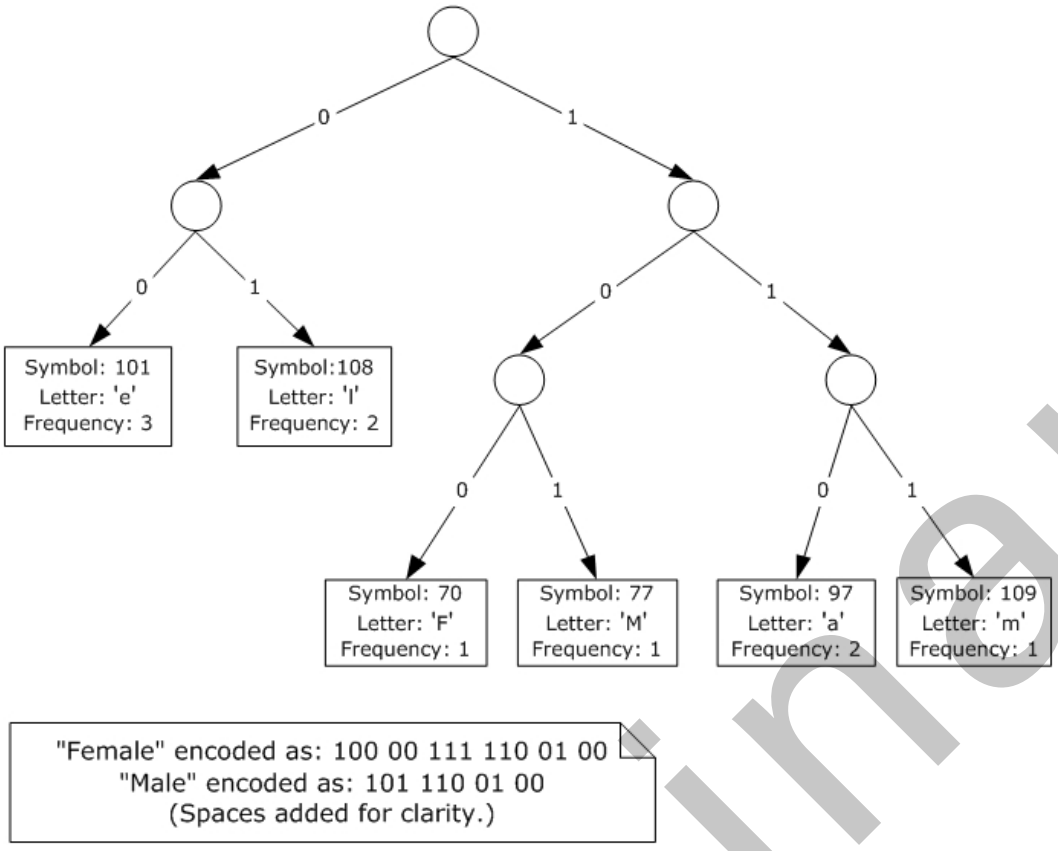


Figure 1: Huffman tree example

In the preceding diagram, walking down the tree—either left (0) or right (1) to each leaf node—shows how the codewords for each character are generated. Therefore, after constructing a Huffman tree, the resulting codewords can be used to decode the bit stream that contains the encoded strings. The bit stream is purely a continuous stream of bits. Errors could result if this bit stream is treated as a series of integers, words, or other types because doing so could induce format errors.

2.7.5 Xpress Compression

The entire database can be persisted to disk as a Spreadsheet Data Model file (section 2.1). This file will be compressed by using Xpress compression, as specified in [MS-WUSP] section 2.1.1. This compression is separate from and in addition to any compression that occurs within any individual file contained in the Spreadsheet Data Model file.

3 Structure Examples

3.1 tbl.xml Metadata File

The following example shows the content of a tbl.xml file, as specified in section 2.5, for a table. As required, this tbl.xml file has an **XMObject** element as its document node (section 2.5.1), with the value of the **class** attribute as "XMSimpleTable" (section 2.5.2.1). This metadata file contains the metadata for the example multiple-segment .idf column data file that is described in section 3.2.

```
<XMObject xmlns="http://schemas.microsoft.com/analysisservices/imbi"
  xmlns:imbi200=
  "http://schemas.microsoft.com/analysisservices/2010/imbi/200"
  xmlns:imbi200_200=
  "http://schemas.microsoft.com/analysisservices/2010/imbi/200/200"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  class="XMSimpleTable"
  name="Table_1_51adc096-9274-4394-b47d-a2fcabfbc1de"
  ProviderVersion="0">
  <Properties>
    <Version xsi:type="xsd:int">1</Version>
    <Settings xsi:type="xsd:long">4353</Settings>
    <RIViolationCount xsi:type="xsd:long">0</RIViolationCount>
  </Properties>
  <Members>
    <Member>
      <Name>SegmentMap</Name>
      <XMObject class="XMMultiPartSegmentMap" ProviderVersion="0">
        <Properties>
          <FirstPartitionRecordCount xsi:type="xsd:long">0
          </FirstPartitionRecordCount>
          <FirstPartitionSegmentCount xsi:type="xsd:long">0
          </FirstPartitionSegmentCount>
        </Properties>
        <Collections>
          <Collection>
            <Name>Partitions</Name>
            <XMObject class=
             "XMSegmentEqualMapEx&lt;XMSegmentEqualMap_FastInstantiation>"
               ProviderVersion="0">
              <Properties>
                <Segments xsi:type="xsd:long">3</Segments>
                <Records xsi:type="xsd:long">2101256</Records>
                <RecordsPerSegment xsi:type="xsd:long">1048576
                </RecordsPerSegment>
              </Properties>
            </XMObject>
          </Collection>
        </Collections>
      </XMObject>
    </Member>
    <Member>
      <Name>TableStats</Name>
      <XMObject class="XMTableStats" ProviderVersion="0">
        <Properties>
          <SegmentSize xsi:type="xsd:long">0</SegmentSize>
          <Usage xsi:type="xsd:int">0</Usage>
        </Properties>
```

```
      </XMObject>
    </Member>
  </Members>
  <Collections>
    <Collection>
      <Name>Partitions</Name>
      <XMObject class="XMPartition"
                name="Table_1_51adc096-9274-4394-b47d-a2fcabfbc1de"
                ProviderVersion="0">
        <Properties>
          <IsProcessed xsi:type="xsd:boolean">true</IsProcessed>
          <Partition xsi:type="xsd:int">0</Partition>
        </Properties>
      </XMObject>
    </Collection>
    <Collection>
      <Name>Columns</Name>
      <XMObject class="XMRawColumn" name="RowNumber" ProviderVersion="1">
        <Properties>
          <Settings xsi:type="xsd:long">1025</Settings>
          <ColumnFlags xsi:type="xsd:long">31</ColumnFlags>
          <Collation/>
          <OrderByColumn/>
          <Locale xsi:type="xsd:long">1033</Locale>
          <BinaryCharacters xsi:type="xsd:unsignedInt">0</BinaryCharacters>
        </Properties>
        <Members>
          <Member>
            <Name>IntrinsicHierarchy</Name>
            <XMObject class="XMHierarchy"
                      name="[Hierarchy for column RowNumber]"
                      ProviderVersion="0">
              <Properties>
    <SortOrder xsi:type="xsd:int">0</SortOrder>
    <IsProcessed xsi:type="xsd:boolean">true</IsProcessed>
    <TypeMaterialization xsi:type="xsd:int">3</TypeMaterialization>
    <ColumnPosition2DataID xsi:type="xsd:long">-1</ColumnPosition2DataID>
    <ColumnDataID2Position xsi:type="xsd:long">-1</ColumnDataID2Position>
    <DistinctDataIDs xsi:type="xsd:long">2101256</DistinctDataIDs>
          <TableStore/>
            </Properties>
          </XMObject>
        </Member>
        <Member>
          <Name>ColumnStats</Name>
          <XMObject class="XMColumnStats" ProviderVersion="0">
            <Properties>
              <DistinctStates xsi:type="xsd:int">2101256</DistinctStates>
              <MinDataID xsi:type="xsd:int">3</MinDataID>
              <MaxDataID xsi:type="xsd:int">2101258</MaxDataID>
              <OriginalMinSegmentDataID xsi:type="xsd:int">2
              </OriginalMinSegmentDataID>
              <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
              <RowCount xsi:type="xsd:long">2101256</RowCount>
              <HasNulls xsi:type="xsd:boolean">false</HasNulls>
              <RLERuns xsi:type="xsd:long">0</RLERuns>
              <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
              <Usage xsi:type="xsd:int">3</Usage>
              <DBType xsi:type="xsd:short">3</DBType>
```

```
          <XMType xsi:type="xsd:int">0</XMType>
          <CompressionType xsi:type="xsd:int">0</CompressionType>
          <CompressionParam xsi:type="xsd:long">0</CompressionParam>
          <EncodingHint xsi:type="xsd:int">1</EncodingHint>
          <AggCounter xsi:type="xsd:long">0</AggCounter>
          <WhereCounter xsi:type="xsd:long">0</WhereCounter>
          <OrderByCounter xsi:type="xsd:long">0</OrderByCounter>
        </Properties>
      </XMObject>
    </Member>
  </Members>
  <Collections>
    <Collection>
      <Name>Segments</Name>
      <XMObject class="XMColumnSegment" ProviderVersion="0">
        <Properties>
          <Records xsi:type="xsd:long">1048576</Records>
          <Mask xsi:type="xsd:long">0</Mask>
        </Properties>
        <Members>
          <Member>
            <Name>SubSegment</Name>
            <XMObject class="XMColumnSegment"
                      ProviderVersion="0">
              <Properties>
                <Records xsi:type="xsd:long">1048576</Records>
                <Mask xsi:type="xsd:long">0</Mask>
              </Properties>
              <Members>
                <Member>
                  <Name>CompressionInfo</Name>
                  <XMObject class="XM123CompressionInfo"
                            ProviderVersion="0">
                    <Properties>
                      <Min xsi:type="xsd:int">3</Min>
                    </Properties>
                  </XMObject>
                </Member>
                <Member>
                  <Name>ColumnSegmentStats</Name>
                  <XMObject class="XMColumnSegmentStats"
                            ProviderVersion="0">
                    <Properties>
          <DistinctStates xsi:type="xsd:long">0</DistinctStates>
          <MinDataID xsi:type="xsd:int">2</MinDataID>
          <MaxDataID xsi:type="xsd:int">2</MaxDataID>
          <OriginalMinSegmentDataID xsi:type="xsd:int">2
          </OriginalMinSegmentDataID>
          <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
          <RowCount xsi:type="xsd:long">0</RowCount>
          <HasNulls xsi:type="xsd:boolean">false</HasNulls>
          <RLERuns xsi:type="xsd:long">0</RLERuns>
          <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
                    </Properties>
                  </XMObject>
                </Member>
              </Members>
            </XMObject>
          </Member>
```

```
      <Member>
        <Name>CompressionInfo</Name>
        <XMObject class=
"XMHybridRLECompressionInfo&lt;class XM123CompressionInfo>"
ProviderVersion="0">
          <Members>
            <Member>
              <Name>RLECompression</Name>
              <XMObject class="XMRLECompressionInfo"
                        ProviderVersion="0">
                <Properties>
        <BookmarkBits xsi:type="xsd:long">24</BookmarkBits>
        <StorageAllocSize xsi:type="xsd:long">32
        </StorageAllocSize>
        <StorageUsedSize xsi:type="xsd:long">2</StorageUsedSize>
        <SegmentNeedsResizing xsi:type="xsd:boolean">false
        </SegmentNeedsResizing>
                </Properties>
              </XMObject>
            </Member>
            <Member>
              <Name>SubCompression</Name>
              <XMObject class="XM123CompressionInfo"
                        ProviderVersion="0">
                <Properties>
                  <Min xsi:type="xsd:int">3</Min>
                </Properties>
              </XMObject>
            </Member>
          </Members>
        </XMObject>
      </Member>
      <Member>
        <Name>ColumnSegmentStats</Name>
        <XMObject class="XMColumnSegmentStats"
                  ProviderVersion="0">
          <Properties>
            <DistinctStates xsi:type="xsd:long">0</DistinctStates>
            <MinDataID xsi:type="xsd:int">3</MinDataID>
            <MaxDataID xsi:type="xsd:int">1048578</MaxDataID>
            <OriginalMinSegmentDataID xsi:type="xsd:int">2
            </OriginalMinSegmentDataID>
            <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
            <RowCount xsi:type="xsd:long">1048576</RowCount>
            <HasNulls xsi:type="xsd:boolean">false</HasNulls>
            <RLERuns xsi:type="xsd:long">0</RLERuns>
            <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
          </Properties>
        </XMObject>
      </Member>
    </Members>
  </XMObject>
  <XMObject class="XMColumnSegment" ProviderVersion="0">
    <Properties>
      <Records xsi:type="xsd:long">1048576</Records>
      <Mask xsi:type="xsd:long">0</Mask>
    </Properties>
    <Members>
      <Member>
```

```
          <Name>SubSegment</Name>
          <XMObject class="XMColumnSegment" ProviderVersion="0">
            <Properties>
              <Records xsi:type="xsd:long">1048576</Records>
              <Mask xsi:type="xsd:long">0</Mask>
            </Properties>
            <Members>
              <Member>
                <Name>CompressionInfo</Name>
                <XMObject class="XM123CompressionInfo"
                          ProviderVersion="0">
                  <Properties>
                    <Min xsi:type="xsd:int">1048579</Min>
                  </Properties>
                </XMObject>
              </Member>
              <Member>
                <Name>ColumnSegmentStats</Name>
                <XMObject class="XMColumnSegmentStats"
                          ProviderVersion="0">
                  <Properties>
                    <DistinctStates xsi:type="xsd:long">0
                    </DistinctStates>
                    <MinDataID xsi:type="xsd:int">2</MinDataID>
                    <MaxDataID xsi:type="xsd:int">2</MaxDataID>
                    <OriginalMinSegmentDataID xsi:type="xsd:int">2
                    </OriginalMinSegmentDataID>
                    <RLESortOrder xsi:type="xsd:long">-1
                    </RLESortOrder>
                    <RowCount xsi:type="xsd:long">0</RowCount>
                    <HasNulls xsi:type="xsd:boolean">false
                    </HasNulls>
                    <RLERuns xsi:type="xsd:long">0</RLERuns>
                    <OthersRLERuns xsi:type="xsd:long">0
                    </OthersRLERuns>
                  </Properties>
                </XMObject>
              </Member>
            </Members>
          </XMObject>
        </Member>
        <Member>
          <Name>CompressionInfo</Name>
          <XMObject class=
"XMHybridRLECompressionInfo&lt;class XM123CompressionInfo>"
                    ProviderVersion="0">
            <Members>
              <Member>
                <Name>RLECompression</Name>
                <XMObject class="XMRLECompressionInfo"
                          ProviderVersion="0">
                  <Properties>
                    <BookmarkBits xsi:type="xsd:long">24
                    </BookmarkBits>
                    <StorageAllocSize xsi:type="xsd:long">32
                    </StorageAllocSize>
                    <StorageUsedSize xsi:type="xsd:long">2
                    </StorageUsedSize>
                    <SegmentNeedsResizing xsi:type="xsd:boolean">
```

```
                    false</SegmentNeedsResizing>
                  </Properties>
                </XMObject>
              </Member>
              <Member>
                <Name>SubCompression</Name>
                <XMObject class="XM123CompressionInfo"
                          ProviderVersion="0">
                  <Properties>
                    <Min xsi:type="xsd:int">1048579</Min>
                  </Properties>
                </XMObject>
              </Member>
            </Members>
          </XMObject>
        </Member>
        <Member>
          <Name>ColumnSegmentStats</Name>
          <XMObject class="XMColumnSegmentStats" ProviderVersion="0">
            <Properties>
              <DistinctStates xsi:type="xsd:long">0
              </DistinctStates>
              <MinDataID xsi:type="xsd:int">1048579</MinDataID>
              <MaxDataID xsi:type="xsd:int">2097154</MaxDataID>
              <OriginalMinSegmentDataID xsi:type="xsd:int">2
              </OriginalMinSegmentDataID>
              <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
              <RowCount xsi:type="xsd:long">1048576</RowCount>
              <HasNulls xsi:type="xsd:boolean">false</HasNulls>
              <RLERuns xsi:type="xsd:long">0</RLERuns>
              <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
            </Properties>
          </XMObject>
        </Member>
      </Members>
    </XMObject>
    <XMObject class="XMColumnSegment" ProviderVersion="0">
      <Properties>
        <Records xsi:type="xsd:long">4104</Records>
        <Mask xsi:type="xsd:long">0</Mask>
      </Properties>
      <Members>
        <Member>
          <Name>SubSegment</Name>
          <XMObject class="XMColumnSegment"
                    ProviderVersion="0">
            <Properties>
              <Records xsi:type="xsd:long">4104</Records>
              <Mask xsi:type="xsd:long">0</Mask>
            </Properties>
            <Members>
              <Member>
                <Name>CompressionInfo</Name>
                <XMObject class="XM123CompressionInfo"
                          ProviderVersion="0">
                  <Properties>
                    <Min xsi:type="xsd:int">2097155</Min>
                  </Properties>
                </XMObject>
```

```
          </Member>
          <Member>
            <Name>ColumnSegmentStats</Name>
            <XMObject class="XMColumnSegmentStats"
                      ProviderVersion="0">
              <Properties>
                <DistinctStates xsi:type="xsd:long">0
                </DistinctStates>
                <MinDataID xsi:type="xsd:int">2</MinDataID>
                <MaxDataID xsi:type="xsd:int">2</MaxDataID>
                <OriginalMinSegmentDataID xsi:type="xsd:int">2
                </OriginalMinSegmentDataID>
                <RLESortOrder xsi:type="xsd:long">-1
                </RLESortOrder>
                <RowCount xsi:type="xsd:long">0</RowCount>
                <HasNulls xsi:type="xsd:boolean">false
                </HasNulls>
                <RLERuns xsi:type="xsd:long">0</RLERuns>
                <OthersRLERuns xsi:type="xsd:long">0
                </OthersRLERuns>
              </Properties>
            </XMObject>
          </Member>
        </Members>
      </XMObject>
    </Member>
    <Member>
      <Name>CompressionInfo</Name>
      <XMObject class=
"XMHybridRLECompressionInfo&lt;class XM123CompressionInfo>"
                ProviderVersion="0">
        <Members>
          <Member>
            <Name>RLECompression</Name>
            <XMObject class="XMRLECompressionInfo"
                      ProviderVersion="0">
              <Properties>
                <BookmarkBits xsi:type="xsd:long">24
                </BookmarkBits>
                <StorageAllocSize xsi:type="xsd:long">32
                </StorageAllocSize>
                <StorageUsedSize xsi:type="xsd:long">2
                </StorageUsedSize>
                <SegmentNeedsResizing xsi:type="xsd:boolean">
                  false</SegmentNeedsResizing>
              </Properties>
            </XMObject>
          </Member>
          <Member>
            <Name>SubCompression</Name>
            <XMObject class="XM123CompressionInfo"
                      ProviderVersion="0">
              <Properties>
                <Min xsi:type="xsd:int">2097155</Min>
              </Properties>
            </XMObject>
          </Member>
        </Members>
      </XMObject>
```

```
            </Member>
            <Member>
              <Name>ColumnSegmentStats</Name>
              <XMObject class="XMColumnSegmentStats"
                        ProviderVersion="0">
                <Properties>
                  <DistinctStates xsi:type="xsd:long">0
                  </DistinctStates>
                  <MinDataID xsi:type="xsd:int">2097155</MinDataID>
                  <MaxDataID xsi:type="xsd:int">2101258</MaxDataID>
                  <OriginalMinSegmentDataID xsi:type="xsd:int">2
                  </OriginalMinSegmentDataID>
                  <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
                  <RowCount xsi:type="xsd:long">4104</RowCount>
                  <HasNulls xsi:type="xsd:boolean">false</HasNulls>
                  <RLERuns xsi:type="xsd:long">0</RLERuns>
                  <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
                </Properties>
              </XMObject>
            </Member>
          </Members>
        </XMObject>
      </Collection>
    </Collections>
    <DataObjects>
      <DataObject>
        <XMObject class="XMValueDataDictionary&lt;XM_Long>"
                  ProviderVersion="0">
          <Properties>
            <DataVersion xsi:type="xsd:int">1</DataVersion>
            <BaseId xsi:type="xsd:long">-3</BaseId>
            <Magnitude xsi:type="xsd:double">1.</Magnitude>
          </Properties>
        </XMObject>
      </DataObject>
      <DataObject>
        <XMObject class="XMRawColumnPartitionDataObject" name=
         "1.Table_1_51adc096-9274-4394-b47d-a2fcabfbc1de.RowNumber.0.idf"
                  ProviderVersion="0">
          <Properties>
            <DataVersion xsi:type="xsd:int">1</DataVersion>
            <Partition xsi:type="xsd:int">0</Partition>
            <SegmentCount xsi:type="xsd:int">3</SegmentCount>
          </Properties>
        </XMObject>
      </DataObject>
    </DataObjects>
  </XMObject>
  <XMObject class="XMRawColumn" name="Column_1" ProviderVersion="1">
    <Properties>
      <Settings xsi:type="xsd:long">1025</Settings>
      <ColumnFlags xsi:type="xsd:long">8</ColumnFlags>
      <Collation/>
      <OrderByColumn/>
      <Locale xsi:type="xsd:long">1033</Locale>
      <BinaryCharacters xsi:type="xsd:unsignedInt">0</BinaryCharacters>
    </Properties>
    <Members>
      <Member>
```

```
            <Name>IntrinsicHierarchy</Name>
            <XMObject class="XMHierarchy" name="[Hierarchy for column Column_1]"
ProviderVersion="0">
              <Properties>
                <SortOrder xsi:type="xsd:int">0</SortOrder>
                <IsProcessed xsi:type="xsd:boolean">true</IsProcessed>
                <TypeMaterialization xsi:type="xsd:int">0</TypeMaterialization>
                <ColumnPosition2DataID xsi:type="xsd:long">0</ColumnPosition2DataID>
                <ColumnDataID2Position xsi:type="xsd:long">1</ColumnDataID2Position>
                <DistinctDataIDs xsi:type="xsd:long">8</DistinctDataIDs>
                <TableStore>H$Table_1_51adc096-9274-4394-b47d-
a2fcabfbc1de$Column_1</TableStore>
              </Properties>
            </XMObject>
          </Member>
          <Member>
            <Name>ColumnStats</Name>
            <XMObject class="XMColumnStats" ProviderVersion="0">
              <Properties>
                <DistinctStates xsi:type="xsd:int">0</DistinctStates>
                <MinDataID xsi:type="xsd:int">3</MinDataID>
                <MaxDataID xsi:type="xsd:int">10</MaxDataID>
                <OriginalMinSegmentDataID xsi:type="xsd:int">2</OriginalMinSegmentDataID>
                <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
                <RowCount xsi:type="xsd:long">2101256</RowCount>
                <HasNulls xsi:type="xsd:boolean">false</HasNulls>
                <RLERuns xsi:type="xsd:long">12</RLERuns>
                <OthersRLERuns xsi:type="xsd:long">1</OthersRLERuns>
                <Usage xsi:type="xsd:int">3</Usage>
                <DBType xsi:type="xsd:short">20</DBType>
                <XMType xsi:type="xsd:int">0</XMType>
                <CompressionType xsi:type="xsd:int">0</CompressionType>
                <CompressionParam xsi:type="xsd:long">0</CompressionParam>
                <EncodingHint xsi:type="xsd:int">0</EncodingHint>
                <AggCounter xsi:type="xsd:long">0</AggCounter>
                <WhereCounter xsi:type="xsd:long">0</WhereCounter>
                <OrderByCounter xsi:type="xsd:long">0</OrderByCounter>
              </Properties>
            </XMObject>
          </Member>
        </Members>
        <Collections>
          <Collection>
            <Name>Segments</Name>
            <XMObject class="XMColumnSegment" ProviderVersion="0">
              <Properties>
                <Records xsi:type="xsd:long">1048576</Records>
                <Mask xsi:type="xsd:long">1</Mask>
              </Properties>
              <Members>
                <Member>
                  <Name>SubSegment</Name>
                  <XMObject class="XMColumnSegment" ProviderVersion="0">
                    <Properties>
                      <Records xsi:type="xsd:long">0</Records>
                      <Mask xsi:type="xsd:long">0</Mask>
                    </Properties>
                    <Members>
                      <Member>
```

```
                    <Name>CompressionInfo</Name>
                    <XMObject class="XMRENoSplitCompressionInfo&lt;2>"
                              ProviderVersion="0">
                      <Properties>
                        <Min xsi:type="xsd:int">3</Min>
                      </Properties>
                    </XMObject>
                  </Member>
                  <Member>
                    <Name>ColumnSegmentStats</Name>
                    <XMObject class="XMColumnSegmentStats"
                              ProviderVersion="0">
                      <Properties>
                        <DistinctStates xsi:type="xsd:long">0
                        </DistinctStates>
                        <MinDataID xsi:type="xsd:int">2</MinDataID>
                        <MaxDataID xsi:type="xsd:int">2</MaxDataID>
                        <OriginalMinSegmentDataID xsi:type="xsd:int">2
                        </OriginalMinSegmentDataID>
                        <RLESortOrder xsi:type="xsd:long">-1
                        </RLESortOrder>
                        <RowCount xsi:type="xsd:long">0</RowCount>
                        <HasNulls xsi:type="xsd:boolean">false
                        </HasNulls>
                        <RLERuns xsi:type="xsd:long">0</RLERuns>
                        <OthersRLERuns xsi:type="xsd:long">0
                        </OthersRLERuns>
                      </Properties>
                    </XMObject>
                  </Member>
                </Members>
              </XMObject>
            </Member>
            <Member>
              <Name>CompressionInfo</Name>
              <XMObject class=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"
                        ProviderVersion="0">
                <Members>
                  <Member>
                    <Name>RLECompression</Name>
                    <XMObject class="XMRLECompressionInfo" ProviderVersion="0">
                      <Properties>
                        <BookmarkBits xsi:type="xsd:long">14
                        </BookmarkBits>
                        <StorageAllocSize xsi:type="xsd:long">32
                        </StorageAllocSize>
                        <StorageUsedSize xsi:type="xsd:long">10
                        </StorageUsedSize>
                        <SegmentNeedsResizing xsi:type="xsd:boolean">
                          false</SegmentNeedsResizing>
                      </Properties>
                    </XMObject>
                  </Member>
                  <Member>
                    <Name>SubCompression</Name>
                    <XMObject class="XMRENoSplitCompressionInfo&lt;2>"
                              ProviderVersion="0">
                      <Properties>
```

```
                    <Min xsi:type="xsd:int">3</Min>
                  </Properties>
                </XMObject>
              </Member>
            </Members>
          </XMObject>
        </Member>
        <Member>
          <Name>ColumnSegmentStats</Name>
          <XMObject class="XMColumnSegmentStats" ProviderVersion="0">
            <Properties>
              <DistinctStates xsi:type="xsd:long">0</DistinctStates>
              <MinDataID xsi:type="xsd:int">3</MinDataID>
              <MaxDataID xsi:type="xsd:int">6</MaxDataID>
              <OriginalMinSegmentDataID xsi:type="xsd:int">2
              </OriginalMinSegmentDataID>
              <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
              <RowCount xsi:type="xsd:long">1048576</RowCount>
              <HasNulls xsi:type="xsd:boolean">false</HasNulls>
              <RLERuns xsi:type="xsd:long">4</RLERuns>
              <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
            </Properties>
          </XMObject>
        </Member>
      </Members>
    </XMObject>
    <XMObject class="XMColumnSegment" ProviderVersion="0">
      <Properties>
        <Records xsi:type="xsd:long">1048576</Records>
        <Mask xsi:type="xsd:long">1</Mask>
      </Properties>
      <Members>
        <Member>
          <Name>SubSegment</Name>
          <XMObject class="XMColumnSegment"
                    ProviderVersion="0">
            <Properties>
              <Records xsi:type="xsd:long">0</Records>
              <Mask xsi:type="xsd:long">0</Mask>
            </Properties>
            <Members>
              <Member>
                <Name>CompressionInfo</Name>
                <XMObject class="XMRENoSplitCompressionInfo&lt;2>"
                          ProviderVersion="0">
                  <Properties>
                    <Min xsi:type="xsd:int">3</Min>
                  </Properties>
                </XMObject>
              </Member>
              <Member>
                <Name>ColumnSegmentStats</Name>
                <XMObject class="XMColumnSegmentStats"
                          ProviderVersion="0">
                  <Properties>
                    <DistinctStates xsi:type="xsd:long">0
                    </DistinctStates>
                    <MinDataID xsi:type="xsd:int">2</MinDataID>
                    <MaxDataID xsi:type="xsd:int">2</MaxDataID>
```

```
                        <OriginalMinSegmentDataID xsi:type="xsd:int">2
                        </OriginalMinSegmentDataID>
                        <RLESortOrder xsi:type="xsd:long">-1
                        </RLESortOrder>
                        <RowCount xsi:type="xsd:long">0</RowCount>
                        <HasNulls xsi:type="xsd:boolean">false
                        </HasNulls>
                        <RLERuns xsi:type="xsd:long">0</RLERuns>
                        <OthersRLERuns xsi:type="xsd:long">0
                        </OthersRLERuns>
                      </Properties>
                    </XMObject>
                  </Member>
                </Members>
              </XMObject>
            </Member>
            <Member>
              <Name>CompressionInfo</Name>
              <XMObject class=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;2>>"
                        ProviderVersion="0">
                <Members>
                  <Member>
                    <Name>RLECompression</Name>
                    <XMObject class="XMRLECompressionInfo"
                              ProviderVersion="0">
                      <Properties>
                        <BookmarkBits xsi:type="xsd:long">14
                        </BookmarkBits>
                        <StorageAllocSize xsi:type="xsd:long">32
                        </StorageAllocSize>
                        <StorageUsedSize xsi:type="xsd:long">10
                        </StorageUsedSize>
                        <SegmentNeedsResizing xsi:type="xsd:boolean">
                          false</SegmentNeedsResizing>
                      </Properties>
                    </XMObject>
                  </Member>
                  <Member>
                    <Name>SubCompression</Name>
                    <XMObject class="XMRENoSplitCompressionInfo&lt;2>"
                              ProviderVersion="0">
                      <Properties>
                        <Min xsi:type="xsd:int">3</Min>
                      </Properties>
                    </XMObject>
                  </Member>
                </Members>
              </XMObject>
            </Member>
            <Member>
              <Name>ColumnSegmentStats</Name>
              <XMObject class="XMColumnSegmentStats"
                        ProviderVersion="0">
                <Properties>
                  <DistinctStates xsi:type="xsd:long">0
                  </DistinctStates>
                  <MinDataID xsi:type="xsd:int">3</MinDataID>
                  <MaxDataID xsi:type="xsd:int">6</MaxDataID>
```

```
            <OriginalMinSegmentDataID xsi:type="xsd:int">2
            </OriginalMinSegmentDataID>
            <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
            <RowCount xsi:type="xsd:long">1048576</RowCount>
            <HasNulls xsi:type="xsd:boolean">false</HasNulls>
            <RLERuns xsi:type="xsd:long">4</RLERuns>
            <OthersRLERuns xsi:type="xsd:long">0</OthersRLERuns>
          </Properties>
        </XMObject>
      </Member>
    </Members>
  </XMObject>
  <XMObject class="XMColumnSegment" ProviderVersion="0">
    <Properties>
      <Records xsi:type="xsd:long">4104</Records>
      <Mask xsi:type="xsd:long">1</Mask>
    </Properties>
    <Members>
      <Member>
        <Name>SubSegment</Name>
        <XMObject class="XMColumnSegment" ProviderVersion="0">
          <Properties>
            <Records xsi:type="xsd:long">8</Records>
            <Mask xsi:type="xsd:long">0</Mask>
          </Properties>
          <Members>
            <Member>
              <Name>CompressionInfo</Name>
              <XMObject class="XMRENoSplitCompressionInfo&lt;3>"
                        ProviderVersion="0">
                <Properties>
                  <Min xsi:type="xsd:int">3</Min>
                </Properties>
              </XMObject>
            </Member>
            <Member>
              <Name>ColumnSegmentStats</Name>
              <XMObject class="XMColumnSegmentStats"
                        ProviderVersion="0">
                <Properties>
                  <DistinctStates xsi:type="xsd:long">0
                  </DistinctStates>
                  <MinDataID xsi:type="xsd:int">2</MinDataID>
                  <MaxDataID xsi:type="xsd:int">2</MaxDataID>
                  <OriginalMinSegmentDataID xsi:type="xsd:int">
                    2</OriginalMinSegmentDataID>
                  <RLESortOrder xsi:type="xsd:long">-1
                  </RLESortOrder>
                  <RowCount xsi:type="xsd:long">0</RowCount>
                  <HasNulls xsi:type="xsd:boolean">false
                  </HasNulls>
                  <RLERuns xsi:type="xsd:long">0</RLERuns>
                  <OthersRLERuns xsi:type="xsd:long">0
                  </OthersRLERuns>
                </Properties>
              </XMObject>
            </Member>
          </Members>
        </XMObject>
```

```
            </Member>
            <Member>
              <Name>CompressionInfo</Name>
              <XMObject class=
"XMHybridRLECompressionInfo&lt;class XMRENoSplitCompressionInfo&lt;3>>"
                        ProviderVersion="0">
                <Members>
                  <Member>
                    <Name>RLECompression</Name>
                    <XMObject class="XMRLECompressionInfo"
                              ProviderVersion="0">
                      <Properties>
                        <BookmarkBits xsi:type="xsd:long">6
                        </BookmarkBits>
                        <StorageAllocSize xsi:type="xsd:long">32
                        </StorageAllocSize>
                        <StorageUsedSize xsi:type="xsd:long">12
                        </StorageUsedSize>
                        <SegmentNeedsResizing xsi:type="xsd:boolean">
                          false</SegmentNeedsResizing>
                      </Properties>
                    </XMObject>
                  </Member>
                  <Member>
                    <Name>SubCompression</Name>
                    <XMObject class="XMRENoSplitCompressionInfo&lt;3>"
                              ProviderVersion="0">
                      <Properties>
                        <Min xsi:type="xsd:int">3</Min>
                      </Properties>
                    </XMObject>
                  </Member>
                </Members>
              </XMObject>
            </Member>
            <Member>
              <Name>ColumnSegmentStats</Name>
              <XMObject class="XMColumnSegmentStats" ProviderVersion="0">
                <Properties>
                  <DistinctStates xsi:type="xsd:long">0</DistinctStates>
                  <MinDataID xsi:type="xsd:int">3</MinDataID>
                  <MaxDataID xsi:type="xsd:int">10</MaxDataID>
                  <OriginalMinSegmentDataID xsi:type="xsd:int">2
                  </OriginalMinSegmentDataID>
                  <RLESortOrder xsi:type="xsd:long">-1</RLESortOrder>
                  <RowCount xsi:type="xsd:long">4104</RowCount>
                  <HasNulls xsi:type="xsd:boolean">false</HasNulls>
                  <RLERuns xsi:type="xsd:long">4</RLERuns>
                  <OthersRLERuns xsi:type="xsd:long">1</OthersRLERuns>
                </Properties>
              </XMObject>
            </Member>
          </Members>
        </XMObject>
      </Collection>
    </Collections>
    <DataObjects>
      <DataObject>
        <XMObject class="XMHashDataDictionary&lt;XM_Long>" name=
```

```
      "1.Table_1_51adc096-9274-4394-b47d-a2fcabfbc1de.Column_1.dictionary"
                      ProviderVersion="0">
              <Properties>
                <DataVersion xsi:type="xsd:int">1</DataVersion>
                <LastId xsi:type="xsd:int">10</LastId>
                <Nullable xsi:type="xsd:boolean">false</Nullable>
                <Unique xsi:type="xsd:boolean">false</Unique>
                <OperatingOn32 xsi:type="xsd:boolean">true</OperatingOn32>
              </Properties>
            </XMObject>
          </DataObject>
          <DataObject>
            <XMObject class="XMRawColumnPartitionDataObject" name=
            "1.Table_1_51adc096-9274-4394-b47d-a2fcabfbc1de.Column_1.0.idf"
                      ProviderVersion="0">
              <Properties>
                <DataVersion xsi:type="xsd:int">1</DataVersion>
                <Partition xsi:type="xsd:int">0</Partition>
                <SegmentCount xsi:type="xsd:int">3</SegmentCount>
              </Properties>
            </XMObject>
          </DataObject>
        </DataObjects>
      </XMObject>
    </Collection>
    <Collection>
      <Name>Relationships</Name>
    </Collection>
    <Collection>
      <Name>UserHierarchies</Name>
    </Collection>
  </Collections>
</XMObject>
```

3.2 Multiple-Segment Column Data .idf File

This example shows a hexadecimal dump of the column data .idf file (section 2.3.1.1) that corresponds to the metadata contained in section 3.1.

In the metadata, the **XMRawColumn** object has a **Segments** collection with three **XMColumnSegment** objects, so three segments have been created for this column. Six segments actually exist in total because each segment has a subsegment. Because this is a column data .idf file, the compression that is used is required to be XMHybridRLE compression (section 2.7.3). This means that each segment has a subsegment member, and the class of the **SubSegment XMObject** element provides the compression that is used for that segment. The **SubCompression** member of the **SubSegment** object provides the minimum data identifier value in the segment. (For information about interpreting the XML metadata, see section 2.5). In the metadata for this column, the compression and minimum data identifier are as follows:

- Segment 1 has XMHybridRLE using XMRENoSplit compression 2-bit (section 2.7.3.3) with a minimum data identifier of 3.
- Segment 2 has the same compression characteristics as Segment 1.
- Segment 3 has XMHybridRLE using XMRENoSplit compression 3-bit (section 2.7.3.4) with minimum data identifier of 3.

The preceding information enables the interpretation of binary contents of the column data .idf file.

The segment size indicator (section 2.3.1.1) exists at Byte 0x00 for the first segment, which is the primary (RLE) segment for the hybrid compression (section 2.7.3). Because this is a hybrid compression, a bit-packing subsegment also exists, even if that subsegment is empty.

The segment size indicator is an 8-byte value, so Bytes 0x00 through 0x07 represent the 8-byte segment size indicator. In this example, it contains the value 0x10 (decimal 16), which indicates that 16 units exist in the segment. This value does not include the segment size indicator. Each unit is 8 bytes, so 128 bytes exist in the first (primary) segment. Therefore, this first segment runs from Byte 0x08 through Byte 0x87. This segment contains a few RLE entries.

Following this primary (RLE) segment is the subsegment. The subsegment also has a segment size indicator. This indicator exists in Bytes 0x88 through 0x8F. The value is 0x01 (decimal 1). A subsegment, even if no bit-packed elements exist, needs to have at least one unit in the segment. In Bytes 0x90 through 0x97, as indicated by the segment size, there is only one unit and it is zero.

The second segment (again, as required, composed of both the primary RLE segment and the bit-packing subsegment) begins with a segment size indicator for the primary segment (Bytes 0x98 through 0x9F), which is followed by the primary RLE segment (Bytes 0xA0 through 0x11F), which is then followed by the segment size indicator for the subsegment (Bytes 0x120 through 0x127), which is finally followed by the subsegment (Bytes 0x128 through 0x12F). Again, this subsegment is empty and therefore has the minimum required size of 1 unit.

The third segment (composed of both the primary RLE segment and the bit-packing subsegment) begins with its primary segment size indicator (Bytes 0x130 through 0x137), which is followed by the primary segment (Bytes 0x138 through 0x1B7), which is then followed by the subsegment segment size indicator (Bytes 0x1B8 through 0x1BF), which is finally followed by the subsegment (Bytes 0x1C0 through 0x1C7).

Unlike the previous combinations of RLE segments and bit-packing subsegments, this third set has bit-packed values as well as RLE-compressed values. It will therefore be described in more detail.

The primary segment contains RLE entries and one bit-packing entry (section 2.7.3). The RLE entries consist of two 4-byte values: a data value and a repeat value. The bit-packing entries are also composed of two 4-byte values. The first value is a negated 1-based offset into the subsegment data, and the second value is the count of the number of values that will follow in the subsegment.

The RLE entries are as follows and are referenced by their start bytes:

- Byte 0x138, Byte 0x13C: (data value = 0x00000003, repeat value = 0x00000400)
- Byte 0x140, Byte 0x144: (data value = 0x00000004, repeat value = 0x00000400)
- Byte 0x148, Byte 0x14C: (data value = 0x00000005, repeat value = 0x00000400)
- Byte 0x150, Byte 0x154: (data value = 0x00000006, repeat value = 0x00000400)

So the RLE entries are (3, 1024), (4, 1024), (5, 1024) and (6, 1024).

The bit-packing entry begins at Byte 0x158 and is easy to recognize because it is –1 (hexadecimal 0xFFFF). Following the negated 1-based offset is the count at Byte 0x15C. The count value is 0x08 (decimal 8). So, the subsegment contains 8 bit-packed values. In this case, these values are compressed by using hybrid bit-packing compression XMRENoSplit 3-bit (section 2.7.3.4), so each value is represented by 3 bits. The eight values therefore require 24 bits (3 bytes) in total. Bytes 0x1C0 through 0x1C2 contain the eight compressed values.

Each compressed value needs to be decompressed and have the minimum value added back in, so these bytes decompress to the sequence 7, 8, 9, 10, 9, 10, 9, 10.

The **ColumnSegmentStats** (see section 3.1 and section 2.5.2.54.1) for this third segment (combination segment) indicates that 4104 rows (**RowCount**) ought to exist. Adding the RLE counts and the 8 bit-packed values results in 4 × 1024 + 8, which equals 4104 data values that correspond to the number of rows.

The metadata also shows that four RLE runs (**RLERuns**) and one other RLE run that is not a solid run (**OthersRLERuns**) ought to exist. In this segment, four RLE entries and one bit-packing entry (which is the other RLE run just mentioned) exist.

The value of **MinDataID** is 3, and the value of **MaxDataID** is 10, which correspond to the value of 3 in the first RLE entry and the 10 values in the decompressed bit-packed values (with a minimum of 3 added back during the decompression).

The actual run of data values (from both the RLE segment and the bit-packing subsegment) includes the values 3, 4, 5, 6, 7, 8, 9, and 10.

```
00000000  10 00 00 00 00 00 00 00-03 00 00 00 00 00 04 00  ................
00000010  04 00 00 00 00 00 04 00-05 00 00 00 00 00 04 00  ................
00000020  06 00 00 00 00 00 04 00-00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00-01 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00-10 00 00 00 00 00 00 00  ................
000000A0  03 00 00 00 00 00 04 00-04 00 00 00 00 00 04 00  ................
000000B0  05 00 00 00 00 00 04 00-06 00 00 00 00 00 04 00  ................
000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000120  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000130  10 00 00 00 00 00 00 00-03 00 00 00 00 04 00 00  ................
00000140  04 00 00 00 00 04 00 00-05 00 00 00 00 04 00 00  ................
00000150  06 00 00 00 00 04 00 00-FF FF FF FF 08 00 00 00  ................
00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001B0  00 00 00 00 00 00 00 00-01 00 00 00 00 00 00 00  ................
000001C0  AC EF FB 00 00 00 00 00-                        ........
```

3.3 Dictionary File

This example shows a hexadecimal dump that illustrates the contents of the dictionary file (section 2.3.2.1) corresponding to the data in the column that is shown in section 3.2.

In the metadata, the **XMRawColumn** object has a **DataObjects** collection. Within this collection is a **DataObject XMObject** of class "XMHashDataDictionary<XM_Long>" (section 2.5.2.21). Therefore, the dictionary is an integer dictionary. The **XM_TYPE_LONG** dictionary contains

integers. However, **XM_TYPE_LONG** dictionaries can contain either 32-bit integers or 64-bit integers (section 2.3.2.1.1). The properties for this object show that the **OperatingOn32** attribute (section 2.5.2.21.1) is set to **true**. The dictionary therefore contains 32-bit integer values. The metadata properties also show that the dictionary contains neither NULL values (**Nullable** is **false**) nor unique values (**Unique** is **false**). Because the values are not guaranteed to be unique, hash information is required to ensure that all the values in the dictionary can be retrieved.

In the binary file, the first 4 bytes (Bytes 0x00 through 0x03) indicate the dictionary type, **XM_TYPE**. The value here is zero. In the **XM_TYPE** enumeration (section 2.3.2.1.3.1), zero corresponds to the **XM_TYPE_LONG** dictionary. This confirms what the metadata has already shown.

XM_TYPE_LONG dictionaries are required to have five hash elements (section 2.3.2.1.1.1). These elements identify the hashing algorithm (section 2.3.3.1.4.2), the sizes of the **HashBin** (section 2.3.3.1.4.4) and **HashEntry** (section 2.3.3.1.4.5) structures, the local entry count (section 2.3.3.1.4.6), and the number of bins used in the hash. The number of bins value is set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT**, which is –1 (section 2.3.3.1.4.1).

Looking at the bytes shows that the hash algorithm (Bytes 0x04 through 0x07) is set to 0xFFFF (decimal –1). This is as expected because –1 corresponds to the value of **XM_INVALID**, the required setting for **XM_TYPE_LONG** dictionaries.

Moving to the next element (Bytes 08 through 0x0B) the **HashEntry** size is set to 8, which means that the **HashEntry** structure is 8 bytes in size. Next (Bytes 0x0C through 0x0F) shows that the **HashBin** size is set to 0x40 (decimal 64), which means that the **HashBin** structure is 64 bytes in size.

The next set of bytes (Bytes 0x10 through 0x13) refers to the local entry count, which is the number of hash entries that a hash bin can contain before an overflow (or collision) occurs. The value is 6, so the hash bin (or bucket) can contain 6 **HashEntry** structures within the **HashBin** structure before it needs to start adding collision entries to the chain pointer. For more information, see section 2.3.3.1.4.4.

The last of the required hash elements is the number of bins. This value is set to **XM_HASH_BIN_VECTOR_INVALID_BIN_COUNT**. The next 8 bytes (Bytes 0x14 through 0x1B) represent the bins value. The value is 0xFFFFFFFF (decimal –1). So, the bins value has been properly set and indicates that no more hash information is included in the dictionary.

The rest of the bytes are related to the dictionary itself, not to the type or the hash. **XM_TYPE_LONG** dictionaries contain just the type information, the required hash elements, and vector of dictionary values. The latter are the actual dictionary items stored in a vector (or array) and are not compressed. So, the next information is the element count and the element size for that vector. The element count is 8 bytes in length. The element size is a 4-byte value. The element count (Bytes 0x1C through 0x23) is 8, so 8 elements exist in the vector. The element size (Bytes 0x24 through 0x27) is 4, which means that each element in the vector is 4 bytes in size.

The vector of values can now be parsed. The values are as follows and are referenced by their start bytes:

- Byte 0x28: value = 0x00000001
- Byte 0x2C: value = 0x00000002
- Byte 0x30: value = 0x00000003
- Byte 0x34: value = 0x00000004

- Byte 0x38: value = 0x0000270F (decimal 9999)
- Byte 0x3C: value = 0x0000270E (decimal 9998)
- Byte 0x40: value = 0x0000270D (decimal 9997)
- Byte 0x44: value = 0x0000270C (decimal 9996)

The vector of integer dictionary values therefore consists of the sequence 1, 2, 3, 4, 9999, 9998, 9997, 9996.

From this sequence, it is clear that no NULL values exist, as expected, but also that the dictionary values are unique. No duplicates exist.

```
00000000  00 00 00 00 FF FF FF FF-08 00 00 00 40 00 00 00  ............@...
00000010  06 00 00 00 FF FF FF FF-FF FF FF FF 08 00 00 00  ................
00000020  00 00 00 00 04 00 00 00-01 00 00 00 02 00 00 00  ................
00000030  03 00 00 00 04 00 00 00-0F 27 00 00 0E 27 00 00  .........'...'..
00000040  0D 27 00 00 0C 27 00 00-                         .'...'..
```

4 Security

4.1 Security Considerations for Implementers

The Spreadsheet Data Model file is compressed as a whole by means of Xpress compression (section 2.7.5). However, neither the entire Spreadsheet Data Model file nor any file contained within it is encrypted. Therefore, to prevent data tampering in general, taking reasonable precautions to prevent unauthorized access to any of the created Spreadsheet Data Model files is suggested.

Additionally, the file formats of the individual files contained within the Spreadsheet Data Model file are sensitive to data tampering. Neither the XML files nor the hash index files (.hidx files), which are also binary, are compressed. In fact, because all the compression algorithms that are used by the files are documented, the data inside the binary files is just as exposed as the metadata in the XML files. Even minor binary changes—whether benign or malicious—or malformed XML data can cause file format read errors, data corruption or alteration, and possibly undefined system behavior.

For example, various files within the Spreadsheet Data Model file use the .idf extension and have the same file format (section 2.3.1.1). This file format, in particular, is highly dependent on the segment size indicators being accurate. Inaccuracies could result in load errors or undefined behavior.

As another example, various files and structures, including the overall file format layout of the Spreadsheet Data Model itself (section 2.1), follow particular memory alignment rules. These alignment rules, as well as the specified byte sizes of different elements within the file formats, are designed to be independent of the operating system environment. This design could result in the padding of structures or file format elements. As a result, file and structure sizes can vary. If the files or structures do not correctly adhere to these alignment rules, load errors or undefined behavior can result.

In addition to the protection of data sources and data integrity, strict adherence to this specification is thus crucial to prevent read or run-time errors.

The actual data that is saved in the Spreadsheet Data Model can change over time. This changeability refers not just to the original data that is processed and saved in various ways within the Spreadsheet Data Model file but also to how that data is processed by the system. The result is that the number and type of files being saved, as well as the particular data that is saved in each file, can change. Again, such changes need to be expected, and proper security procedures for file protection are recommended to differentiate between a valid file that has changed and an invalid file that has been tampered with.

The CryptKey.bin file (section 2.1.2.4) contains the key BLOB that is needed to decrypt and encrypt the password and connection strings for the Spreadsheet Data Model. This key BLOB is in its original form. Because the key is exposed in this manner, care needs to be taken to ensure that a strong key—in other words, one that is not easily broken—is used.

4.2 Index of Security Parameters

Security parameter	Section
The **ConnectionString** property	2.1.2.1.1
The **SvrEncryptPwdFlag** property	2.1.2.3.1
The CryptKey.bin file	2.1.2.4

Security parameter	Section
The **QueryImpersonationInfo** property, as described in [MS-SSAS] section 2.2.4.2.2.6	2.6.2
The **ConnectionStringSecurity** property, as described in [MS-SSAS] section 2.2.4.2.2.6	2.6.2
The **ConnectionString** property, as described in [MS-SSAS] section 2.2.4.2.2.6	2.6.2
The **QueryImpersonationInfo** property, as described in [MS-SSAS] section 2.2.4.2.2.6	2.6.2
The **DataSourceImpersonationInfo** property, as described in [MS-SSAS] section 2.2.4.2.2.5	2.6.4

5 Appendix A: Compression Mask for XMRENoSplit Compression Algorithms

The following single-dimension array contains hexadecimal values that are used in XMRENoSplit compression algorithms:

```
const maskArray[]=
{
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFD, 0xFFFFFFFFFFFFFFFB, 0xFFFFFFFFFFFFFFF7,
0xFFFFFFFFFFFFFFEF, 0xFFFFFFFFFFFFFFDF, 0xFFFFFFFFFFFFFFBF, 0xFFFFFFFFFFFFFF7F,
0xFFFFFFFFFFFFFEFF, 0xFFFFFFFFFFFFFDFF, 0xFFFFFFFFFFFFFBFF, 0xFFFFFFFFFFFFF7FF,
0xFFFFFFFFFFFFEFFF, 0xFFFFFFFFFFFFDFFF, 0xFFFFFFFFFFFFBFFF, 0xFFFFFFFFFFFF7FFF,
0xFFFFFFFFFFFEFFFF, 0xFFFFFFFFFFFDFFFF, 0xFFFFFFFFFFFBFFFF, 0xFFFFFFFFFFF7FFFF,
0xFFFFFFFFFFEFFFFF, 0xFFFFFFFFFFDFFFFF, 0xFFFFFFFFFFBFFFFF, 0xFFFFFFFFFF7FFFFF,
0xFFFFFFFFFEFFFFFF, 0xFFFFFFFFFDFFFFFF, 0xFFFFFFFFFBFFFFFF, 0xFFFFFFFFF7FFFFFF,
0xFFFFFFFFEFFFFFFF, 0xFFFFFFFFDFFFFFFF, 0xFFFFFFFFBFFFFFFF, 0xFFFFFFFF7FFFFFFF,
0xFFFFFFFEFFFFFFFF, 0xFFFFFFFDFFFFFFFF, 0xFFFFFFFBFFFFFFFF, 0xFFFFFFF7FFFFFFFF,
0xFFFFFFEFFFFFFFFF, 0xFFFFFFDFFFFFFFFF, 0xFFFFFFBFFFFFFFFF, 0xFFFFFF7FFFFFFFFF,
0xFFFFFEFFFFFFFFFF, 0xFFFFFDFFFFFFFFFF, 0xFFFFFBFFFFFFFFFF, 0xFFFFF7FFFFFFFFFF,
0xFFFFEFFFFFFFFFFF, 0xFFFFDFFFFFFFFFFF, 0xFFFFBFFFFFFFFFFF, 0xFFFF7FFFFFFFFFFF,
0xFFFEFFFFFFFFFFFF, 0xFFFDFFFFFFFFFFFF, 0xFFFBFFFFFFFFFFFF, 0xFFF7FFFFFFFFFFFF,
0xFFEFFFFFFFFFFFFF, 0xFFDFFFFFFFFFFFFF, 0xFFBFFFFFFFFFFFFF, 0xFF7FFFFFFFFFFFFF,
0xFEFFFFFFFFFFFFFF, 0xFDFFFFFFFFFFFFFF, 0xFBFFFFFFFFFFFFFF, 0xF7FFFFFFFFFFFFFF,
0xEFFFFFFFFFFFFFFF, 0xDFFFFFFFFFFFFFFF, 0xBFFFFFFFFFFFFFFF, 0x7FFFFFFFFFFFFFFF,
0xFFFFFFFFFFFFFFFC, 0xFFFFFFFFFFFFFFF9, 0xFFFFFFFFFFFFFFF3, 0xFFFFFFFFFFFFFFE7,
0xFFFFFFFFFFFFFFCF, 0xFFFFFFFFFFFFFF9F, 0xFFFFFFFFFFFFFF3F, 0xFFFFFFFFFFFFFE7F,
0xFFFFFFFFFFFFFCFF, 0xFFFFFFFFFFFFF9FF, 0xFFFFFFFFFFFFF3FF, 0xFFFFFFFFFFFFE7FF,
0xFFFFFFFFFFFFCFFF, 0xFFFFFFFFFFFF9FFF, 0xFFFFFFFFFFFF3FFF, 0xFFFFFFFFFFFE7FFF,
0xFFFFFFFFFFFCFFFF, 0xFFFFFFFFFFF9FFFF, 0xFFFFFFFFFFF3FFFF, 0xFFFFFFFFFFE7FFFF,
0xFFFFFFFFFFCFFFFF, 0xFFFFFFFFFF9FFFFF, 0xFFFFFFFFFF3FFFFF, 0xFFFFFFFFFE7FFFFF,
0xFFFFFFFFFCFFFFFF, 0xFFFFFFFFF9FFFFFF, 0xFFFFFFFFF3FFFFFF, 0xFFFFFFFFE7FFFFFF,
0xFFFFFFFFCFFFFFFF, 0xFFFFFFFF9FFFFFFF, 0xFFFFFFFF3FFFFFFF, 0xFFFFFFFE7FFFFFFF,
0xFFFFFFFCFFFFFFFF, 0xFFFFFFF9FFFFFFFF, 0xFFFFFFF3FFFFFFFF, 0xFFFFFFE7FFFFFFFF,
0xFFFFFFCFFFFFFFFF, 0xFFFFFF9FFFFFFFFF, 0xFFFFFF3FFFFFFFFF, 0xFFFFFE7FFFFFFFFF,
0xFFFFFCFFFFFFFFFF, 0xFFFFF9FFFFFFFFFF, 0xFFFFF3FFFFFFFFFF, 0xFFFFE7FFFFFFFFFF,
0xFFFFCFFFFFFFFFFF, 0xFFFF9FFFFFFFFFFF, 0xFFFF3FFFFFFFFFFF, 0xFFFE7FFFFFFFFFFF,
0xFFFCFFFFFFFFFFFF, 0xFFF9FFFFFFFFFFFF, 0xFFF3FFFFFFFFFFFF, 0xFFE7FFFFFFFFFFFF,
0xFFCFFFFFFFFFFFFF, 0xFF9FFFFFFFFFFFFF, 0xFF3FFFFFFFFFFFFF, 0xFE7FFFFFFFFFFFFF,
0xFCFFFFFFFFFFFFFF, 0xF9FFFFFFFFFFFFFF, 0xF3FFFFFFFFFFFFFF, 0xE7FFFFFFFFFFFFFF,
0xCFFFFFFFFFFFFFFF, 0x9FFFFFFFFFFFFFFF, 0x3FFFFFFFFFFFFFFF, 0x0000000000000000,
0xFFFFFFFFFFFFFFF8, 0xFFFFFFFFFFFFFFF1, 0xFFFFFFFFFFFFFFE3, 0xFFFFFFFFFFFFFFC7,
0xFFFFFFFFFFFFFF8F, 0xFFFFFFFFFFFFFF1F, 0xFFFFFFFFFFFFFE3F, 0xFFFFFFFFFFFFFC7F,
0xFFFFFFFFFFFFF8FF, 0xFFFFFFFFFFFFF1FF, 0xFFFFFFFFFFFFE3FF, 0xFFFFFFFFFFFFC7FF,
0xFFFFFFFFFFFF8FFF, 0xFFFFFFFFFFFF1FFF, 0xFFFFFFFFFFFE3FFF, 0xFFFFFFFFFFFC7FFF,
0xFFFFFFFFFFF8FFFF, 0xFFFFFFFFFFF1FFFF, 0xFFFFFFFFFFE3FFFF, 0xFFFFFFFFFFC7FFFF,
0xFFFFFFFFFF8FFFFF, 0xFFFFFFFFFF1FFFFF, 0xFFFFFFFFFE3FFFFF, 0xFFFFFFFFFC7FFFFF,
```

```
0xFFFFFFFFF8FFFFFF, 0xFFFFFFFFF1FFFFFF, 0xFFFFFFFFE3FFFFFF, 0xFFFFFFFFC7FFFFFF,
0xFFFFFFFF8FFFFFFF, 0xFFFFFFFF1FFFFFFF, 0xFFFFFFFE3FFFFFFF, 0xFFFFFFFC7FFFFFFF,
0xFFFFFFF8FFFFFFFF, 0xFFFFFFF1FFFFFFFF, 0xFFFFFFE3FFFFFFFF, 0xFFFFFFC7FFFFFFFF,
0xFFFFFF8FFFFFFFFF, 0xFFFFFF1FFFFFFFFF, 0xFFFFE3FFFFFFFFFF, 0xFFFFC7FFFFFFFFFF,
0xFFFFF8FFFFFFFFFF, 0xFFFFF1FFFFFFFFFF, 0xFFFE3FFFFFFFFFFF, 0xFFFC7FFFFFFFFFFF,
0xFFFF8FFFFFFFFFFF, 0xFFFF1FFFFFFFFFFF, 0xFFE3FFFFFFFFFFFF, 0xFFC7FFFFFFFFFFFF,
0xFFF8FFFFFFFFFFFF, 0xFFF1FFFFFFFFFFFF, 0xFE3FFFFFFFFFFFFF, 0xFC7FFFFFFFFFFFFF,
0xFF8FFFFFFFFFFFFF, 0xFF1FFFFFFFFFFFFF, 0xF3FFFFFFFFFFFFFF, 0xC7FFFFFFFFFFFFFF,
0xF8FFFFFFFFFFFFFF, 0xF1FFFFFFFFFFFFFF, 0xE3FFFFFFFFFFFFFF, 0x87FFFFFFFFFFFFFF,
0x8FFFFFFFFFFFFFFF, 0x1FFFFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFFF0, 0xFFFFFFFFFFFFFFE1, 0xFFFFFFFFFFFFFFC3, 0xFFFFFFFFFFFFFF87,
0xFFFFFFFFFFFFFF0F, 0xFFFFFFFFFFFFFE1F, 0xFFFFFFFFFFFFFC3F, 0xFFFFFFFFFFFFF87F,
0xFFFFFFFFFFFFF0FF, 0xFFFFFFFFFFFFE1FF, 0xFFFFFFFFFFFFC3FF, 0xFFFFFFFFFFFF87FF,
0xFFFFFFFFFFFF0FFF, 0xFFFFFFFFFFFE1FFF, 0xFFFFFFFFFFFC3FFF, 0xFFFFFFFFFFF87FFF,
0xFFFFFFFFFFF0FFFF, 0xFFFFFFFFFFE1FFFF, 0xFFFFFFFFFFC3FFFF, 0xFFFFFFFFFF87FFFF,
0xFFFFFFFFFF0FFFFF, 0xFFFFFFFFFE1FFFFF, 0xFFFFFFFFFC3FFFFF, 0xFFFFFFFFF87FFFFF,
0xFFFFFFFFF0FFFFFF, 0xFFFFFFFFE1FFFFFF, 0xFFFFFFFFC3FFFFFF, 0xFFFFFFFF87FFFFFF,
0xFFFFFFFF0FFFFFFF, 0xFFFFFFFE1FFFFFFF, 0xFFFFFFFC3FFFFFFF, 0xFFFFFFF87FFFFFFF,
0xFFFFFFF0FFFFFFFF, 0xFFFFFFE1FFFFFFFF, 0xFFFFFFC3FFFFFFFF, 0xFFFFFF87FFFFFFFF,
0xFFFFFF0FFFFFFFFF, 0xFFFFFE1FFFFFFFFF, 0xFFFFC3FFFFFFFFFF, 0xFFFF87FFFFFFFFFF,
0xFFFF0FFFFFFFFFFF, 0xFFFE1FFFFFFFFFFF, 0xFFFC3FFFFFFFFFFF, 0xFFF87FFFFFFFFFFF,
0xFFF0FFFFFFFFFFFF, 0xFFE1FFFFFFFFFFFF, 0xFFC3FFFFFFFFFFFF, 0xFF87FFFFFFFFFFFF,
0xFF0FFFFFFFFFFFFF, 0xFE1FFFFFFFFFFFFF, 0xFC3FFFFFFFFFFFFF, 0xF87FFFFFFFFFFFFF,
0xF0FFFFFFFFFFFFFF, 0xE1FFFFFFFFFFFFFF, 0xC3FFFFFFFFFFFFFF, 0x87FFFFFFFFFFFFFF,
0x0FFFFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFFE0, 0xFFFFFFFFFFFFFFC1, 0xFFFFFFFFFFFFFF83, 0xFFFFFFFFFFFFFF07,
0xFFFFFFFFFFFFFE0F, 0xFFFFFFFFFFFFFC1F, 0xFFFFFFFFFFFFF83F, 0xFFFFFFFFFFFFF07F,
0xFFFFFFFFFFFFE0FF, 0xFFFFFFFFFFFFC1FF, 0xFFFFFFFFFFFF83FF, 0xFFFFFFFFFFFF07FF,
0xFFFFFFFFFFE0FFFF, 0xFFFFFFFFFFC1FFFF, 0xFFFFFFFFFF83FFFF, 0xFFFFFFFFFF07FFFF,
0xFFFFFFFFFE0FFFFF, 0xFFFFFFFFFC1FFFFF, 0xFFFFFFFFF83FFFFF, 0xFFFFFFFFF07FFFFF,
0xFFFFFFFFE0FFFFFF, 0xFFFFFFFFC1FFFFFF, 0xFFFFFFFF83FFFFFF, 0xFFFFFFFF07FFFFFF,
0xFFFFFFFE0FFFFFFF, 0xFFFFFFFC1FFFFFFF, 0xFFFFFFF83FFFFFFF, 0xFFFFFFF07FFFFFFF,
0xFFFFFFE0FFFFFFFF, 0xFFFFFFC1FFFFFFFF, 0xFFFFFF83FFFFFFFF, 0xFFFFFF07FFFFFFFF,
0xFFFFE0FFFFFFFFFF, 0xFFFFC1FFFFFFFFFF, 0xFFFF83FFFFFFFFFF, 0xFFFF07FFFFFFFFFF,
0xFFFE0FFFFFFFFFFF, 0xFFFC1FFFFFFFFFFF, 0xFFF83FFFFFFFFFFF, 0xFFF07FFFFFFFFFFF,
0xFFE0FFFFFFFFFFFF, 0xFFC1FFFFFFFFFFFF, 0xFF83FFFFFFFFFFFF, 0xFF07FFFFFFFFFFFF,
0xFE0FFFFFFFFFFFFF, 0xFC1FFFFFFFFFFFFF, 0xF83FFFFFFFFFFFFF, 0xF07FFFFFFFFFFFFF,
0xE0FFFFFFFFFFFFFF, 0xC1FFFFFFFFFFFFFF, 0x83FFFFFFFFFFFFFF, 0x07FFFFFFFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFFC0, 0xFFFFFFFFFFFFFF81, 0xFFFFFFFFFFFFFF03, 0xFFFFFFFFFFFFFE07,
0xFFFFFFFFFFFFFC0F, 0xFFFFFFFFFFFFF81F, 0xFFFFFFFFFFFFF03F, 0xFFFFFFFFFFFFE07F,
0xFFFFFFFFFFFFC0FF, 0xFFFFFFFFFFFF81FF, 0xFFFFFFFFFFFF03FF, 0xFFFFFFFFFFFE07FF,
0xFFFFFFFFFFFC0FFF, 0xFFFFFFFFFFF81FFF, 0xFFFFFFFFFFF03FFF, 0xFFFFFFFFFFE07FFF,
0xFFFFFFFFFFC0FFFF, 0xFFFFFFFFFF81FFFF, 0xFFFFFFFFFF03FFFF, 0xFFFFFFFFFE07FFFF,
0xFFFFFFFFFC0FFFFF, 0xFFFFFFFFF81FFFFF, 0xFFFFFFFFF03FFFFF, 0xFFFFFFFFE07FFFFF,
0xFFFFFFFFC0FFFFFF, 0xFFFFFFFF81FFFFFF, 0xFFFFFFFF03FFFFFF, 0xFFFFFFFE07FFFFFF,
0xFFFFFFFC0FFFFFFF, 0xFFFFFFF81FFFFFFF, 0xFFFFFFF03FFFFFFF, 0xFFFFFFE07FFFFFFF,
0xFFFFFFC0FFFFFFFF, 0xFFFFFF81FFFFFFFF, 0xFFFFFF03FFFFFFFF, 0xFFFFFE07FFFFFFFF,
0xFFFFC0FFFFFFFFFF, 0xFFFF81FFFFFFFFFF, 0xFFFF03FFFFFFFFFF, 0xFFFE07FFFFFFFFFF,
0xFFC0FFFFFFFFFFFF, 0xFF81FFFFFFFFFFFF, 0xFF03FFFFFFFFFFFF, 0xFE07FFFFFFFFFFFF,
0xFC0FFFFFFFFFFFFF, 0xF81FFFFFFFFFFFFF, 0xF03FFFFFFFFFFFFF, 0xE07FFFFFFFFFFFFF,
0xC0FFFFFFFFFFFFFF, 0x81FFFFFFFFFFFFFF, 0x03FFFFFFFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFF80, 0xFFFFFFFFFFFFFF01, 0xFFFFFFFFFFFFFE03, 0xFFFFFFFFFFFFFC07,
0xFFFFFFFFFFFFF80F, 0xFFFFFFFFFFFFF01F, 0xFFFFFFFFFFFFE03F, 0xFFFFFFFFFFFFC07F,
0xFFFFFFFFFFFF80FF, 0xFFFFFFFFFFFF01FF, 0xFFFFFFFFFFFE03FF, 0xFFFFFFFFFFFC07FF,
0xFFFFFFFFFFF80FFF, 0xFFFFFFFFFFF01FFF, 0xFFFFFFFFFFE03FFF, 0xFFFFFFFFFFC07FFF,
0xFFFFFFFFFF80FFFF, 0xFFFFFFFFFF01FFFF, 0xFFFFFFFFFE03FFFF, 0xFFFFFFFFFC07FFFF,
0xFFFFFFFF80FFFFFF, 0xFFFFFFFF01FFFFFF, 0xFFFFFFFE03FFFFFF, 0xFFFFFFFC07FFFFFF,
0xFFFFFFF80FFFFFFF, 0xFFFFFFF01FFFFFFF, 0xFFFFFFE03FFFFFFF, 0xFFFFFFC07FFFFFFF,
```

```
0xFFFFFF80FFFFFFFF, 0xFFFFFF01FFFFFFFF, 0xFFFFFE03FFFFFFFF, 0xFFFFFC07FFFFFFFF,
0xFFFFF80FFFFFFFFF, 0xFFFFF01FFFFFFFFF, 0xFFFFE03FFFFFFFFF, 0xFFFFC07FFFFFFFFF,
0xFFFF80FFFFFFFFFF, 0xFFFF01FFFFFFFFFF, 0xFFFE03FFFFFFFFFF, 0xFFFC07FFFFFFFFFF,
0xFFF80FFFFFFFFFFF, 0xFFF01FFFFFFFFFFF, 0xFFE03FFFFFFFFFFF, 0xFFC07FFFFFFFFFFF,
0xFF80FFFFFFFFFFFF, 0xFF01FFFFFFFFFFFF, 0xFE03FFFFFFFFFFFF, 0xFC07FFFFFFFFFFFF,
0xF80FFFFFFFFFFFFF, 0xF01FFFFFFFFFFFFF, 0xE03FFFFFFFFFFFFF, 0xC07FFFFFFFFFFFFF,
0x80FFFFFFFFFFFFFF, 0x01FFFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFF00, 0xFFFFFFFFFFFFFE01, 0xFFFFFFFFFFFFFC03, 0xFFFFFFFFFFFFF807,
0xFFFFFFFFFFFFF00F, 0xFFFFFFFFFFFFE01F, 0xFFFFFFFFFFFFC03F, 0xFFFFFFFFFFFF807F,
0xFFFFFFFFFFFF00FF, 0xFFFFFFFFFFFE01FF, 0xFFFFFFFFFFFC03FF, 0xFFFFFFFFFFF807FF,
0xFFFFFFFFFFF00FFF, 0xFFFFFFFFFFE01FFF, 0xFFFFFFFFFFC03FFF, 0xFFFFFFFFFF807FFF,
0xFFFFFFFFFF00FFFF, 0xFFFFFFFFFE01FFFF, 0xFFFFFFFFFC03FFFF, 0xFFFFFFFFF807FFFF,
0xFFFFFFFFF00FFFFF, 0xFFFFFFFFE01FFFFF, 0xFFFFFFFFC03FFFFF, 0xFFFFFFF807FFFFF,
0xFFFFFFFF00FFFFFF, 0xFFFFFFFE01FFFFFF, 0xFFFFFFFC03FFFFFF, 0xFFFFFF807FFFFFF,
0xFFFFFFF00FFFFFFF, 0xFFFFFFE01FFFFFFF, 0xFFFFFFC03FFFFFFF, 0xFFFFF807FFFFFFF,
0xFFFFFF00FFFFFFFF, 0xFFFFFE01FFFFFFFF, 0xFFFFFC03FFFFFFFF, 0xFFFF807FFFFFFFF,
0xFFFFF00FFFFFFFFF, 0xFFFFE01FFFFFFFFF, 0xFFFFC03FFFFFFFFF, 0xFFF807FFFFFFFFF,
0xFFFF00FFFFFFFFFF, 0xFFFE01FFFFFFFFFF, 0xFFFC03FFFFFFFFFF, 0xFF807FFFFFFFFFF,
0xFFF00FFFFFFFFFFF, 0xFFE01FFFFFFFFFFF, 0xFFC03FFFFFFFFFFF, 0xF807FFFFFFFFFFF,
0xFF00FFFFFFFFFFFF, 0xFE01FFFFFFFFFFFF, 0xFC03FFFFFFFFFFFF, 0x807FFFFFFFFFFFF,
0xF00FFFFFFFFFFFFF, 0xE01FFFFFFFFFFFFF, 0xC03FFFFFFFFFFFFF, 0x807FFFFFFFFFFFFF,
0x00FFFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFE00, 0xFFFFFFFFFFFFFC01, 0xFFFFFFFFFFFFF803, 0xFFFFFFFFFFFFF007,
0xFFFFFFFFFFFFE00F, 0xFFFFFFFFFFFFC01F, 0xFFFFFFFFFFFF803F, 0xFFFFFFFFFFFF007F,
0xFFFFFFFFFFFE00FF, 0xFFFFFFFFFFFC01FF, 0xFFFFFFFFFFF803FF, 0xFFFFFFFFFFF007FF,
0xFFFFFFFFFFE00FFF, 0xFFFFFFFFFFC01FFF, 0xFFFFFFFFFF803FFF, 0xFFFFFFFFFF007FFF,
0xFFFFFFFFFE00FFFF, 0xFFFFFFFFFC01FFFF, 0xFFFFFFFFF803FFFF, 0xFFFFFFFFF007FFFF,
0xFFFFFFFFE00FFFFF, 0xFFFFFFFFC01FFFFF, 0xFFFFFFFF803FFFFF, 0xFFFFFFFF007FFFFF,
0xFFFFFFFE00FFFFFF, 0xFFFFFFFC01FFFFFF, 0xFFFFFFF803FFFFFF, 0xFFFFFFF007FFFFFF,
0xFFFFFFE00FFFFFFF, 0xFFFFFFC01FFFFFFF, 0xFFFFFF803FFFFFFF, 0xFFFFFF007FFFFFFF,
0xFFFFE00FFFFFFFFF, 0xFFFFC01FFFFFFFFF, 0xFFFF803FFFFFFFFF, 0xFFFF007FFFFFFFFF,
0xFFFE00FFFFFFFFFF, 0xFFFC01FFFFFFFFFF, 0xFFF803FFFFFFFFFF, 0xFFF007FFFFFFFFFF,
0xFFE00FFFFFFFFFFF, 0xFFC01FFFFFFFFFFF, 0xFF803FFFFFFFFFFF, 0xFF007FFFFFFFFFFF,
0xFE00FFFFFFFFFFFF, 0xFC01FFFFFFFFFFFF, 0xF803FFFFFFFFFFFF, 0xF007FFFFFFFFFFFF,
0xE00FFFFFFFFFFFFF, 0xC01FFFFFFFFFFFFF, 0x803FFFFFFFFFFFFF, 0x007FFFFFFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFFC00, 0xFFFFFFFFFFFFF801, 0xFFFFFFFFFFFFF003, 0xFFFFFFFFFFFFE007,
0xFFFFFFFFFFFFC00F, 0xFFFFFFFFFFFF801F, 0xFFFFFFFFFFFF003F, 0xFFFFFFFFFFFE007F,
0xFFFFFFFFFFFC00FF, 0xFFFFFFFFFFF801FF, 0xFFFFFFFFFFF003FF, 0xFFFFFFFFFFE007FF,
0xFFFFFFFFFFC00FFF, 0xFFFFFFFFFF801FFF, 0xFFFFFFFFFF003FFF, 0xFFFFFFFFFE007FFF,
0xFFFFFFFFFC00FFFF, 0xFFFFFFFFF801FFFF, 0xFFFFFFFF003FFFF, 0xFFFFFFFE007FFFF,
0xFFFFFFFC00FFFFFF, 0xFFFFFFF801FFFFFF, 0xFFFFFFF003FFFFFF, 0xFFFFFFE007FFFFFF,
0xFFFFFFC00FFFFFFF, 0xFFFFFF801FFFFFFF, 0xFFFFFF003FFFFFFF, 0xFFFFFE007FFFFFFF,
0xFFFFFC00FFFFFFFF, 0xFFFFF801FFFFFFFF, 0xFFFFF003FFFFFFFF, 0xFFFFE007FFFFFFFF,
0xFFFFC00FFFFFFFFF, 0xFFFF801FFFFFFFFF, 0xFFFF003FFFFFFFFF, 0xFFFE007FFFFFFFFF,
0xFFFC00FFFFFFFFFF, 0xFFF801FFFFFFFFFF, 0xFFF003FFFFFFFFFF, 0xFFE007FFFFFFFFFF,
0xFFC00FFFFFFFFFFF, 0xFF801FFFFFFFFFFF, 0xFF003FFFFFFFFFFF, 0xFE007FFFFFFFFFFF,
0xFC00FFFFFFFFFFFF, 0xF801FFFFFFFFFFFF, 0xF003FFFFFFFFFFFF, 0xE007FFFFFFFFFFFF,
0xC00FFFFFFFFFFFFF, 0x801FFFFFFFFFFFFF, 0x003FFFFFFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFF800, 0xFFFFFFFFFFFFF001, 0xFFFFFFFFFFFFE003, 0xFFFFFFFFFFFFC007,
0xFFFFFFFFFFFF800F, 0xFFFFFFFFFFFF001F, 0xFFFFFFFFFFFE003F, 0xFFFFFFFFFFFC007F,
0xFFFFFFFFFFF800FF, 0xFFFFFFFFFFF001FF, 0xFFFFFFFFFFE003FF, 0xFFFFFFFFFFC007FF,
0xFFFFFFFFFF800FFF, 0xFFFFFFFFFF001FFF, 0xFFFFFFFFFE003FFF, 0xFFFFFFFFFC007FFF,
0xFFFFFFFF800FFFFF, 0xFFFFFFFF001FFFFF, 0xFFFFFFFE003FFFFF, 0xFFFFFFFC007FFFFF,
0xFFFFFFF800FFFFFF, 0xFFFFFFF001FFFFFF, 0xFFFFFFE003FFFFFF, 0xFFFFFFC007FFFFFF,
0xFFFFFF800FFFFFFF, 0xFFFFFF001FFFFFFF, 0xFFFFFE003FFFFFFF, 0xFFFFFC007FFFFFFF,
0xFFFFF800FFFFFFFF, 0xFFFFF001FFFFFFFF, 0xFFFFE003FFFFFFFF, 0xFFFFC007FFFFFFFF,
```

```
0xFFF800FFFFFFFFFF, 0xFFF001FFFFFFFFFF, 0xFFE003FFFFFFFFFF, 0xFFC007FFFFFFFFFF,
0xFF800FFFFFFFFFFF, 0xFF001FFFFFFFFFFF, 0xFE003FFFFFFFFFFF, 0xFC007FFFFFFFFFFF,
0xF800FFFFFFFFFFFF, 0xF001FFFFFFFFFFFF, 0xE003FFFFFFFFFFFF, 0xC007FFFFFFFFFFFF,
0x800FFFFFFFFFFFFF, 0x001FFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFF000, 0xFFFFFFFFFFFFE001, 0xFFFFFFFFFFFFC003, 0xFFFFFFFFFFFF8007,
0xFFFFFFFFFFFF000F, 0xFFFFFFFFFFFE001F, 0xFFFFFFFFFFFC003F, 0xFFFFFFFFFFF8007F,
0xFFFFFFFFFFF000FF, 0xFFFFFFFFFFE001FF, 0xFFFFFFFFFFC003FF, 0xFFFFFFFFFF8007FF,
0xFFFFFFFFFF000FFF, 0xFFFFFFFFFE001FFF, 0xFFFFFFFFFC003FFF, 0xFFFFFFFFF8007FFF,
0xFFFFFFFF000FFFFF, 0xFFFFFFFE001FFFFF, 0xFFFFFFFC003FFFFF, 0xFFFFFFF8007FFFFF,
0xFFFFFFF000FFFFFF, 0xFFFFFFE001FFFFFF, 0xFFFFFFC003FFFFFF, 0xFFFFFF8007FFFFFF,
0xFFFFFF000FFFFFFF, 0xFFFFFE001FFFFFFF, 0xFFFFFC003FFFFFFF, 0xFFFFF8007FFFFFFF,
0xFFFFF000FFFFFFFF, 0xFFFFE001FFFFFFFF, 0xFFFFC003FFFFFFFF, 0xFFFF8007FFFFFFFF,
0xFFFF000FFFFFFFFF, 0xFFFE001FFFFFFFFF, 0xFFFC003FFFFFFFFF, 0xFFF8007FFFFFFFFF,
0xFFF000FFFFFFFFFF, 0xFFE001FFFFFFFFFF, 0xFFC003FFFFFFFFFF, 0xFF8007FFFFFFFFFF,
0xFF000FFFFFFFFFFF, 0xFE001FFFFFFFFFFF, 0xFC003FFFFFFFFFFF, 0xF8007FFFFFFFFFFF,
0xF000FFFFFFFFFFFF, 0xE001FFFFFFFFFFFF, 0xC003FFFFFFFFFFFF, 0x8007FFFFFFFFFFFF,
0x000FFFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFE000, 0xFFFFFFFFFFFFC001, 0xFFFFFFFFFFFF8003, 0xFFFFFFFFFFFF0007,
0xFFFFFFFFFFFE000F, 0xFFFFFFFFFFFC001F, 0xFFFFFFFFFFF8003F, 0xFFFFFFFFFFF0007F,
0xFFFFFFFFFFE000FF, 0xFFFFFFFFFFC001FF, 0xFFFFFFFFFF8003FF, 0xFFFFFFFFFF0007FF,
0xFFFFFFFFFE000FFF, 0xFFFFFFFFFC001FFF, 0xFFFFFFFFF8003FFF, 0xFFFFFFFFF0007FFF,
0xFFFFFFFE000FFFFF, 0xFFFFFFFC001FFFFF, 0xFFFFFFF8003FFFFF, 0xFFFFFFF0007FFFFF,
0xFFFFFFE000FFFFFF, 0xFFFFFFC001FFFFFF, 0xFFFFFF8003FFFFFF, 0xFFFFFF0007FFFFFF,
0xFFFFFE000FFFFFFF, 0xFFFFC001FFFFFFFF, 0xFFFF8003FFFFFFFF, 0xFFFF0007FFFFFFFF,
0xFFFE000FFFFFFFFF, 0xFFFC001FFFFFFFFF, 0xFFF8003FFFFFFFFF, 0xFFF0007FFFFFFFFF,
0xFFE000FFFFFFFFFF, 0xFFC001FFFFFFFFFF, 0xFF8003FFFFFFFFFF, 0xFF0007FFFFFFFFFF,
0xFE000FFFFFFFFFFF, 0xFC001FFFFFFFFFFF, 0xF8003FFFFFFFFFFF, 0xF0007FFFFFFFFFFF,
0xE000FFFFFFFFFFFF, 0xC001FFFFFFFFFFFF, 0x8003FFFFFFFFFFFF, 0x0007FFFFFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFFC000, 0xFFFFFFFFFFFF8001, 0xFFFFFFFFFFFF0003, 0xFFFFFFFFFFFE0007,
0xFFFFFFFFFFFC000F, 0xFFFFFFFFFFF8001F, 0xFFFFFFFFFFF0003F, 0xFFFFFFFFFFE0007F,
0xFFFFFFFFFFC000FF, 0xFFFFFFFFFF8001FF, 0xFFFFFFFFFF0003FF, 0xFFFFFFFFFE0007FF,
0xFFFFFFFFFC000FFF, 0xFFFFFFFFF8001FFF, 0xFFFFFFFFF0003FFF, 0xFFFFFFFFE0007FFF,
0xFFFFFFFFC000FFFF, 0xFFFFFFFF8001FFFF, 0xFFFFFFFF0003FFFF, 0xFFFFFFFE0007FFFF,
0xFFFFFFFC000FFFFF, 0xFFFFFFF8001FFFFF, 0xFFFFFFF0003FFFFF, 0xFFFFFFE0007FFFFF,
0xFFFFFFC000FFFFFF, 0xFFFFFF8001FFFFFF, 0xFFFFFF0003FFFFFF, 0xFFFFFE0007FFFFFF,
0xFFFFFC000FFFFFFF, 0xFFFFF8001FFFFFFF, 0xFFFFF0003FFFFFFF, 0xFFFFE0007FFFFFFF,
0xFFFFC000FFFFFFFF, 0xFFFF8001FFFFFFFF, 0xFFFF0003FFFFFFFF, 0xFFFE0007FFFFFFFF,
0xFFFC000FFFFFFFFF, 0xFFF8001FFFFFFFFF, 0xFFF0003FFFFFFFFF, 0xFFE0007FFFFFFFFF,
0xFFC000FFFFFFFFFF, 0xFF8001FFFFFFFFFF, 0xFF0003FFFFFFFFFF, 0xFE0007FFFFFFFFFF,
0xFC000FFFFFFFFFFF, 0xF8001FFFFFFFFFFF, 0xF0003FFFFFFFFFFF, 0xE0007FFFFFFFFFFF,
0xC000FFFFFFFFFFFF, 0x8001FFFFFFFFFFFF, 0x0003FFFFFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFF8000, 0xFFFFFFFFFFFF0001, 0xFFFFFFFFFFFE0003, 0xFFFFFFFFFFFC0007,
0xFFFFFFFFFFF8000F, 0xFFFFFFFFFFF0001F, 0xFFFFFFFFFFE0003F, 0xFFFFFFFFFFC0007F,
0xFFFFFFFFFF8000FF, 0xFFFFFFFFFF0001FF, 0xFFFFFFFFFE0003FF, 0xFFFFFFFFFC0007FF,
0xFFFFFFFFF8000FFF, 0xFFFFFFFFF0001FFF, 0xFFFFFFFFE0003FFF, 0xFFFFFFFFC0007FFF,
0xFFFFFFFF8000FFFF, 0xFFFFFFFF0001FFFF, 0xFFFFFFFE0003FFFF, 0xFFFFFFFC0007FFFF,
0xFFFFFFF8000FFFFF, 0xFFFFFFF0001FFFFF, 0xFFFFFFE0003FFFFF, 0xFFFFFFC0007FFFFF,
0xFFFFFF8000FFFFFF, 0xFFFFFF0001FFFFFF, 0xFFFFFE0003FFFFFF, 0xFFFFFC0007FFFFFF,
0xFFFFF8000FFFFFFF, 0xFFFFF0001FFFFFFF, 0xFFFFE0003FFFFFFF, 0xFFFFC0007FFFFFFF,
0xFFFF8000FFFFFFFF, 0xFFFF0001FFFFFFFF, 0xFFFE0003FFFFFFFF, 0xFFFC0007FFFFFFFF,
0xFFF8000FFFFFFFFF, 0xFFF0001FFFFFFFFF, 0xFFE0003FFFFFFFFF, 0xFFC0007FFFFFFFFF,
0xFF8000FFFFFFFFFF, 0xFF0001FFFFFFFFFF, 0xFE0003FFFFFFFFFF, 0xFC0007FFFFFFFFFF,
0xF8000FFFFFFFFFFF, 0xF0001FFFFFFFFFFF, 0xE0003FFFFFFFFFFF, 0xC0007FFFFFFFFFFF,
```

```
0x8000FFFFFFFFFFFF, 0x0001FFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFF0000, 0xFFFFFFFFFFFE0001, 0xFFFFFFFFFFFC0003, 0xFFFFFFFFFFF80007,
0xFFFFFFFFFFF0000F, 0xFFFFFFFFFFE0001F, 0xFFFFFFFFFFC0003F, 0xFFFFFFFFFF80007F,
0xFFFFFFFFFF0000FF, 0xFFFFFFFFFE0001FF, 0xFFFFFFFFFC0003FF, 0xFFFFFFFFF80007FF,
0xFFFFFFFFF0000FFF, 0xFFFFFFFFE0001FFF, 0xFFFFFFFFC0003FFF, 0xFFFFFFFF80007FFF,
0xFFFFFFFF0000FFFF, 0xFFFFFFFE0001FFFF, 0xFFFFFFFC0003FFFF, 0xFFFFFFF80007FFFF,
0xFFFFFFF0000FFFFF, 0xFFFFFFE0001FFFFF, 0xFFFFFFC0003FFFFF, 0xFFFFFF80007FFFFF,
0xFFFFFF0000FFFFFF, 0xFFFFFE0001FFFFFF, 0xFFFFFC0003FFFFFF, 0xFFFFF80007FFFFFF,
0xFFFFF0000FFFFFFF, 0xFFFFE0001FFFFFFF, 0xFFFFC0003FFFFFFF, 0xFFFF80007FFFFFFF,
0xFFFF0000FFFFFFFF, 0xFFFE0001FFFFFFFF, 0xFFFC0003FFFFFFFF, 0xFFF80007FFFFFFFF,
0xFFF0000FFFFFFFFF, 0xFFE0001FFFFFFFFF, 0xFFC0003FFFFFFFFF, 0xFF80007FFFFFFFFF,
0xFF0000FFFFFFFFFF, 0xFE0001FFFFFFFFFF, 0xFC0003FFFFFFFFFF, 0xF80007FFFFFFFFFF,
0xF0000FFFFFFFFFFF, 0xE0001FFFFFFFFFFF, 0xC0003FFFFFFFFFFF, 0x80007FFFFFFFFFFF,
0x0000FFFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFE0000, 0xFFFFFFFFFFFC0001, 0xFFFFFFFFFFF80003, 0xFFFFFFFFFFF00007,
0xFFFFFFFFFFE0000F, 0xFFFFFFFFFFC0001F, 0xFFFFFFFFFF80003F, 0xFFFFFFFFFF00007F,
0xFFFFFFFFFE0000FF, 0xFFFFFFFFFC0001FF, 0xFFFFFFFFF80003FF, 0xFFFFFFFFF00007FF,
0xFFFFFFFFE0000FFF, 0xFFFFFFFFC0001FFF, 0xFFFFFFFF80003FFF, 0xFFFFFFFF00007FFF,
0xFFFFFFFE0000FFFF, 0xFFFFFFFC0001FFFF, 0xFFFFFFF80003FFFF, 0xFFFFFFF00007FFFF,
0xFFFFFFE0000FFFFF, 0xFFFFFFC0001FFFFF, 0xFFFFFF80003FFFFF, 0xFFFFFF00007FFFFF,
0xFFFFFE0000FFFFFF, 0xFFFFFC0001FFFFFF, 0xFFFFF80003FFFFFF, 0xFFFFF00007FFFFFF,
0xFFFFE0000FFFFFFF, 0xFFFFC0001FFFFFFF, 0xFFFF80003FFFFFFF, 0xFFFF00007FFFFFFF,
0xFFE0000FFFFFFFFF, 0xFFC0001FFFFFFFFF, 0xFF80003FFFFFFFFF, 0xFF00007FFFFFFFFF,
0xFE0000FFFFFFFFFF, 0xFC0001FFFFFFFFFF, 0xF80003FFFFFFFFFF, 0xF00007FFFFFFFFFF,
0xE0000FFFFFFFFFFF, 0xC0001FFFFFFFFFFF, 0x80003FFFFFFFFFFF, 0x00007FFFFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFC0000, 0xFFFFFFFFFFF80001, 0xFFFFFFFFFFF00003, 0xFFFFFFFFFFE00007,
0xFFFFFFFFFFC0000F, 0xFFFFFFFFFF80001F, 0xFFFFFFFFFF00003F, 0xFFFFFFFFFE00007F,
0xFFFFFFFFFC0000FF, 0xFFFFFFFFF80001FF, 0xFFFFFFFFF00003FF, 0xFFFFFFFFE00007FF,
0xFFFFFFFFC0000FFF, 0xFFFFFFFF80001FFF, 0xFFFFFFFF00003FFF, 0xFFFFFFFE00007FFF,
0xFFFFFFFC0000FFFF, 0xFFFFFFF80001FFFF, 0xFFFFFFF00003FFFF, 0xFFFFFFE00007FFFF,
0xFFFFFFC0000FFFFF, 0xFFFFFF80001FFFFF, 0xFFFFFF00003FFFFF, 0xFFFFFE00007FFFFF,
0xFFFFFC0000FFFFFF, 0xFFFFF80001FFFFFF, 0xFFFFF00003FFFFFF, 0xFFFFE00007FFFFFF,
0xFFFFC0000FFFFFFF, 0xFFFF80001FFFFFFF, 0xFFFF00003FFFFFFF, 0xFFFE00007FFFFFFF,
0xFFC0000FFFFFFFFF, 0xFF80001FFFFFFFFF, 0xFF00003FFFFFFFFF, 0xFE00007FFFFFFFFF,
0xFC0000FFFFFFFFFF, 0xF80001FFFFFFFFFF, 0xF00003FFFFFFFFFF, 0xE00007FFFFFFFFFF,
0xC0000FFFFFFFFFFF, 0x80001FFFFFFFFFFF, 0x00003FFFFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFFF80000, 0xFFFFFFFFFFFF00001, 0xFFFFFFFFFFFE00003, 0xFFFFFFFFFFFC00007,
0xFFFFFFFFFFF80000F, 0xFFFFFFFFFFF00001F, 0xFFFFFFFFFFE00003F, 0xFFFFFFFFFFC00007F,
0xFFFFFFFFFF80000FF, 0xFFFFFFFFFF00001FF, 0xFFFFFFFFFE00003FF, 0xFFFFFFFFC00007FF,
0xFFFFFFFF80000FFF, 0xFFFFFFFF00001FFF, 0xFFFFFFFE00003FFF, 0xFFFFFFFC00007FFF,
0xFFFFFFF80000FFFF, 0xFFFFFFF00001FFFF, 0xFFFFFFE00003FFFF, 0xFFFFFFC00007FFFF,
0xFFFFFF80000FFFFF, 0xFFFFFF00001FFFFF, 0xFFFFFE00003FFFFF, 0xFFFFFC00007FFFFF,
0xFFFFF80000FFFFFF, 0xFFFFF00001FFFFFF, 0xFFFFE00003FFFFFF, 0xFFFFC00007FFFFFF,
0xFFF80000FFFFFFFF, 0xFFF00001FFFFFFFF, 0xFFE00003FFFFFFFF, 0xFFC00007FFFFFFFF,
0xFF80000FFFFFFFFF, 0xFF00001FFFFFFFFF, 0xFE00003FFFFFFFFF, 0xFC00007FFFFFFFFF,
0xF80000FFFFFFFFFF, 0xF00001FFFFFFFFFF, 0xE00003FFFFFFFFFF, 0xC00007FFFFFFFFFF,
0x80000FFFFFFFFFFF, 0x00001FFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFF00000, 0xFFFFFFFFFFE00001, 0xFFFFFFFFFFC00003, 0xFFFFFFFFFF800007,
0xFFFFFFFFFF00000F, 0xFFFFFFFFFE00001F, 0xFFFFFFFFFC00003F, 0xFFFFFFFFF800007F,
0xFFFFFFFFF00000FF, 0xFFFFFFFFE00001FF, 0xFFFFFFFFC00003FF, 0xFFFFFFFF800007FF,
0xFFFFFFFF00000FFF, 0xFFFFFFFE00001FFF, 0xFFFFFFFC00003FFF, 0xFFFFFFF800007FFF,
0xFFFFFFF00000FFFF, 0xFFFFFFE00001FFFF, 0xFFFFFFC00003FFFF, 0xFFFFFF800007FFFF,
0xFFFFFF00000FFFFF, 0xFFFFFE00001FFFFF, 0xFFFFFC00003FFFFF, 0xFFFFF800007FFFFF,
0xFFFFF00000FFFFFF, 0xFFFFE00001FFFFFF, 0xFFFFC00003FFFFFF, 0xFFFF800007FFFFFF,
0xFFFF00000FFFFFFF, 0xFFFE00001FFFFFFF, 0xFFFC00003FFFFFFF, 0xFFF800007FFFFFFF,
0xFFF00000FFFFFFFF, 0xFFE00001FFFFFFFF, 0xFFC00003FFFFFFFF, 0xFF800007FFFFFFFF,
0xFF00000FFFFFFFFF, 0xFE00001FFFFFFFFF, 0xFC00003FFFFFFFFF, 0xF800007FFFFFFFFF,
0xF00000FFFFFFFFFF, 0xE00001FFFFFFFFFF, 0xC00003FFFFFFFFFF, 0x800007FFFFFFFFFF,
0x00000FFFFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFE00000, 0xFFFFFFFFFFC00001, 0xFFFFFFFFFF800003, 0xFFFFFFFFFF000007,
0xFFFFFFFFFE00000F, 0xFFFFFFFFFC00001F, 0xFFFFFFFFF800003F, 0xFFFFFFFFF000007F,
0xFFFFFFFFE00000FF, 0xFFFFFFFFC00001FF, 0xFFFFFFFF800003FF, 0xFFFFFFFF000007FF,
0xFFFFFFFE00000FFF, 0xFFFFFFFC00001FFF, 0xFFFFFFF800003FFF, 0xFFFFFFF000007FFF,
0xFFFFFFE00000FFFF, 0xFFFFFFC00001FFFF, 0xFFFFFF800003FFFF, 0xFFFFFF000007FFFF,
0xFFFFFE00000FFFFF, 0xFFFFFC00001FFFFF, 0xFFFFF800003FFFFF, 0xFFFFF000007FFFFF,
0xFFFFE00000FFFFFF, 0xFFFFC00001FFFFFF, 0xFFFF800003FFFFFF, 0xFFFF000007FFFFFF,
0xFFFE00000FFFFFFF, 0xFFFC00001FFFFFFF, 0xFFF800003FFFFFFF, 0xFFF000007FFFFFFF,
0xFFE00000FFFFFFFF, 0xFFC00001FFFFFFFF, 0xFF800003FFFFFFFF, 0xFF000007FFFFFFFF,
0xFE00000FFFFFFFFF, 0xFC00001FFFFFFFFF, 0xF800003FFFFFFFFF, 0xF000007FFFFFFFFF,
0xE00000FFFFFFFFFF, 0xC00001FFFFFFFFFF, 0x800003FFFFFFFFFF, 0x000007FFFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFFC00000, 0xFFFFFFFFFF800001, 0xFFFFFFFFFF000003, 0xFFFFFFFFFE000007,
0xFFFFFFFFFC00000F, 0xFFFFFFFFF800001F, 0xFFFFFFFFF000003F, 0xFFFFFFFFE000007F,
0xFFFFFFFFC00000FF, 0xFFFFFFFF800001FF, 0xFFFFFFFF000003FF, 0xFFFFFFFE000007FF,
0xFFFFFFFC00000FFF, 0xFFFFFFF800001FFF, 0xFFFFFFF000003FFF, 0xFFFFFFE000007FFF,
0xFFFFFFC00000FFFF, 0xFFFFFF800001FFFF, 0xFFFFFF000003FFFF, 0xFFFFFE000007FFFF,
0xFFFFFC00000FFFFF, 0xFFFFF800001FFFFF, 0xFFFFF000003FFFFF, 0xFFFFE000007FFFFF,
0xFFFFC00000FFFFFF, 0xFFFF800001FFFFFF, 0xFFFF000003FFFFFF, 0xFFFE000007FFFFFF,
0xFFFC00000FFFFFFF, 0xFFF800001FFFFFFF, 0xFFF000003FFFFFFF, 0xFFE000007FFFFFFF,
0xFFC00000FFFFFFFF, 0xFF800001FFFFFFFF, 0xFF000003FFFFFFFF, 0xFE000007FFFFFFFF,
0xFC00000FFFFFFFFF, 0xF800001FFFFFFFFF, 0xF000003FFFFFFFFF, 0xE000007FFFFFFFFF,
0xC00000FFFFFFFFFF, 0x800001FFFFFFFFFF, 0x000003FFFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFF800000, 0xFFFFFFFFFF000001, 0xFFFFFFFFFE000003, 0xFFFFFFFFFC000007,
0xFFFFFFFFF800000F, 0xFFFFFFFFF000001F, 0xFFFFFFFFE000003F, 0xFFFFFFFFC000007F,
0xFFFFFFFF800000FF, 0xFFFFFFFF000001FF, 0xFFFFFFFE000003FF, 0xFFFFFFFC000007FF,
0xFFFFFFF800000FFF, 0xFFFFFFF000001FFF, 0xFFFFFFE000003FFF, 0xFFFFFFC000007FFF,
0xFFFFFF800000FFFF, 0xFFFFFF000001FFFF, 0xFFFFFE000003FFFF, 0xFFFFFC000007FFFF,
0xFFFFF800000FFFFF, 0xFFFFF000001FFFFF, 0xFFFFE000003FFFFF, 0xFFFFC000007FFFFF,
0xFFFF800000FFFFFF, 0xFFFF000001FFFFFF, 0xFFFE000003FFFFFF, 0xFFFC000007FFFFFF,
0xFFF800000FFFFFFF, 0xFFF000001FFFFFFF, 0xFFE000003FFFFFFF, 0xFFC000007FFFFFFF,
0xFF800000FFFFFFFF, 0xFF000001FFFFFFFF, 0xFE000003FFFFFFFF, 0xFC000007FFFFFFFF,
0xF800000FFFFFFFFF, 0xF000001FFFFFFFFF, 0xE000003FFFFFFFFF, 0xC000007FFFFFFFFF,
0x800000FFFFFFFFFF, 0x000001FFFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0xFFFFFFFFFF000000, 0xFFFFFFFFFE000001, 0xFFFFFFFFFC000003, 0xFFFFFFFF8000007,
0xFFFFFFFFF000000F, 0xFFFFFFFFE000001F, 0xFFFFFFFFC000003F, 0xFFFFFFF8000007F,
0xFFFFFFFF000000FF, 0xFFFFFFFE000001FF, 0xFFFFFFFC000003FF, 0xFFFFFF8000007FF,
0xFFFFFFF000000FFF, 0xFFFFFFE000001FFF, 0xFFFFFFC000003FFF, 0xFFFFF8000007FFF,
0xFFFFFF000000FFFF, 0xFFFFFE000001FFFF, 0xFFFFFC000003FFFF, 0xFFFF8000007FFFF,
0xFFFFF000000FFFFF, 0xFFFFE000001FFFFF, 0xFFFC000003FFFFF, 0xFFF8000007FFFFF,
0xFFF000000FFFFFF, 0xFFE000001FFFFFF, 0xFFC000003FFFFFF, 0xFF8000007FFFFFF,
0xFF000000FFFFFFF, 0xFE000001FFFFFFF, 0xFC000003FFFFFFF, 0xF8000007FFFFFFF,
0xF000000FFFFFFFF, 0xE000001FFFFFFFF, 0xC000003FFFFFFFF, 0x8000007FFFFFFFF,
0x000000FFFFFFFFF, 0x000000000000000, 0x000000000000000, 0x000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFE000000, 0xFFFFFFFFFC000001, 0xFFFFFFFFF8000003, 0xFFFFFFFFF0000007,
0xFFFFFFFFE000000F, 0xFFFFFFFFC000001F, 0xFFFFFFFF8000003F, 0xFFFFFFFF0000007F,
0xFFFFFFFE000000FF, 0xFFFFFFFC000001FF, 0xFFFFFFF8000003FF, 0xFFFFFFF0000007FF,
0xFFFFFFE000000FFF, 0xFFFFFFC000001FFF, 0xFFFFFF8000003FFF, 0xFFFFFF0000007FFF,
0xFFFFFE000000FFFF, 0xFFFFFC000001FFFF, 0xFFFFF8000003FFFF, 0xFFFFF0000007FFFF,
0xFFFFE000000FFFFF, 0xFFFFC000001FFFFF, 0xFFFF8000003FFFFF, 0xFFFF0000007FFFFF,
0xFFFE000000FFFFFF, 0xFFFC000001FFFFFF, 0xFFF8000003FFFFFF, 0xFFF0000007FFFFFF,
0xFFE000000FFFFFFF, 0xFFC000001FFFFFFF, 0xFF8000003FFFFFFF, 0xFF0000007FFFFFFF,
0xFE000000FFFFFFFF, 0xFC000001FFFFFFFF, 0xF8000003FFFFFFFF, 0xF0000007FFFFFFFF,
0xE000000FFFFFFFFF, 0xC000001FFFFFFFFF, 0x8000003FFFFFFFFF, 0x0000007FFFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFFC000000, 0xFFFFFFFFF8000001, 0xFFFFFFFFF0000003, 0xFFFFFFFFE0000007,
0xFFFFFFFFC000000F, 0xFFFFFFFF8000001F, 0xFFFFFFFF0000003F, 0xFFFFFFFE0000007F,
0xFFFFFFFC000000FF, 0xFFFFFFF8000001FF, 0xFFFFFFF0000003FF, 0xFFFFFFE0000007FF,
0xFFFFFFC000000FFF, 0xFFFFFF8000001FFF, 0xFFFFFF0000003FFF, 0xFFFFFE0000007FFF,
0xFFFFFC000000FFFF, 0xFFFFF8000001FFFF, 0xFFFFF0000003FFFF, 0xFFFFE0000007FFFF,
0xFFFFC000000FFFFF, 0xFFFF8000001FFFFF, 0xFFFF0000003FFFFF, 0xFFFE0000007FFFFF,
0xFFFC000000FFFFFF, 0xFFF8000001FFFFFF, 0xFFF0000003FFFFFF, 0xFFE0000007FFFFFF,
0xFFC000000FFFFFFF, 0xFF8000001FFFFFFF, 0xFF0000003FFFFFFF, 0xFE0000007FFFFFFF,
0xFC000000FFFFFFFF, 0xF8000001FFFFFFFF, 0xF0000003FFFFFFFF, 0xE0000007FFFFFFFF,
0xC000000FFFFFFFFF, 0x8000001FFFFFFFFF, 0x0000003FFFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFF8000000, 0xFFFFFFFFF0000001, 0xFFFFFFFFE0000003, 0xFFFFFFFFC0000007,
0xFFFFFFFF8000000F, 0xFFFFFFFF0000001F, 0xFFFFFFFE0000003F, 0xFFFFFFFC0000007F,
0xFFFFFFF8000000FF, 0xFFFFFFF0000001FF, 0xFFFFFFE0000003FF, 0xFFFFFFC0000007FF,
0xFFFFFF8000000FFF, 0xFFFFFF0000001FFF, 0xFFFFFE0000003FFF, 0xFFFFFC0000007FFF,
0xFFFFF8000000FFFF, 0xFFFFF0000001FFFF, 0xFFFFE0000003FFFF, 0xFFFFC0000007FFFF,
0xFFFF8000000FFFFF, 0xFFFF0000001FFFFF, 0xFFFE0000003FFFFF, 0xFFFC0000007FFFFF,
0xFFF8000000FFFFFF, 0xFFF0000001FFFFFF, 0xFFE0000003FFFFFF, 0xFFC0000007FFFFFF,
0xFF8000000FFFFFFF, 0xFF0000001FFFFFFF, 0xFE0000003FFFFFFF, 0xFC0000007FFFFFFF,
0xF8000000FFFFFFFF, 0xF0000001FFFFFFFF, 0xE0000003FFFFFFFF, 0xC0000007FFFFFFFF,
0x8000000FFFFFFFFF, 0x0000001FFFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFF0000000, 0xFFFFFFFFE0000001, 0xFFFFFFFFC0000003, 0xFFFFFFFF80000007,
0xFFFFFFFF0000000F, 0xFFFFFFFE0000001F, 0xFFFFFFFC0000003F, 0xFFFFFFF80000007F,
```

```
0xFFFFFFF0000000FF, 0xFFFFFFE0000001FF, 0xFFFFFFC0000003FF, 0xFFFFFF80000007FF,
0xFFFFFF0000000FFF, 0xFFFFFE0000001FFF, 0xFFFFFC0000003FFF, 0xFFFFF80000007FFF,
0xFFFFF0000000FFFF, 0xFFFFE0000001FFFF, 0xFFFFC0000003FFFF, 0xFFFF80000007FFFF,
0xFFFF0000000FFFFF, 0xFFFE0000001FFFFF, 0xFFFC0000003FFFFF, 0xFFF80000007FFFFF,
0xFFF0000000FFFFFF, 0xFFE0000001FFFFFF, 0xFFC0000003FFFFFF, 0xFF80000007FFFFFF,
0xFF0000000FFFFFFF, 0xFE0000001FFFFFFF, 0xFC0000003FFFFFFF, 0xF80000007FFFFFFF,
0xF0000000FFFFFFFF, 0xE0000001FFFFFFFF, 0xC0000003FFFFFFFF, 0x80000007FFFFFFFF,
0x0000000FFFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFE0000000, 0xFFFFFFFFC0000001, 0xFFFFFFFF80000003, 0xFFFFFFFF00000007,
0xFFFFFFFE0000000F, 0xFFFFFFFC0000001F, 0xFFFFFFF80000003F, 0xFFFFFFF00000007F,
0xFFFFFFE0000000FF, 0xFFFFFFC0000001FF, 0xFFFFFF80000003FF, 0xFFFFFF00000007FF,
0xFFFFFE0000000FFF, 0xFFFFFC0000001FFF, 0xFFFFF80000003FFF, 0xFFFFF00000007FFF,
0xFFFFE0000000FFFF, 0xFFFFC0000001FFFF, 0xFFFF80000003FFFF, 0xFFFF00000007FFFF,
0xFFFE0000000FFFFF, 0xFFFC0000001FFFFF, 0xFFF80000003FFFFF, 0xFFF00000007FFFFF,
0xFFE0000000FFFFFF, 0xFFC0000001FFFFFF, 0xFF80000003FFFFFF, 0xFF00000007FFFFFF,
0xFE0000000FFFFFFF, 0xFC0000001FFFFFFF, 0xF80000003FFFFFFF, 0xF00000007FFFFFFF,
0xE0000000FFFFFFFF, 0xC0000001FFFFFFFF, 0x80000003FFFFFFFF, 0x00000007FFFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFFC0000000, 0xFFFFFFFF80000001, 0xFFFFFFFF00000003, 0xFFFFFFFE00000007,
0xFFFFFFFC0000000F, 0xFFFFFFF80000001F, 0xFFFFFFF00000003F, 0xFFFFFFE00000007F,
0xFFFFFFC0000000FF, 0xFFFFFF80000001FF, 0xFFFFFF00000003FF, 0xFFFFFE00000007FF,
0xFFFFFC0000000FFF, 0xFFFFF80000001FFF, 0xFFFFF00000003FFF, 0xFFFFE00000007FFF,
0xFFFFC0000000FFFF, 0xFFFF80000001FFFF, 0xFFFF00000003FFFF, 0xFFFE00000007FFFF,
0xFFFC0000000FFFFF, 0xFFF80000001FFFFF, 0xFFF00000003FFFFF, 0xFFE00000007FFFFF,
0xFFC0000000FFFFFF, 0xFF80000001FFFFFF, 0xFF00000003FFFFFF, 0xFE00000007FFFFFF,
0xFC0000000FFFFFFF, 0xF80000001FFFFFFF, 0xF00000003FFFFFFF, 0xE00000007FFFFFFF,
0xC0000000FFFFFFFF, 0x80000001FFFFFFFF, 0x00000003FFFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFF80000000, 0xFFFFFFFF00000001, 0xFFFFFFFE00000003, 0xFFFFFFFC00000007,
0xFFFFFFF80000000F, 0xFFFFFFF00000001F, 0xFFFFFFE00000003F, 0xFFFFFFC00000007F,
0xFFFFFF80000000FF, 0xFFFFFF00000001FF, 0xFFFFFE00000003FF, 0xFFFFFC00000007FF,
0xFFFFF80000000FFF, 0xFFFFF00000001FFF, 0xFFFFE00000003FFF, 0xFFFFC00000007FFF,
0xFFFF80000000FFFF, 0xFFFF00000001FFFF, 0xFFFE00000003FFFF, 0xFFFC00000007FFFF,
0xFFF80000000FFFFF, 0xFFF00000001FFFFF, 0xFFE00000003FFFFF, 0xFFC00000007FFFFF,
0xFF80000000FFFFFF, 0xFF00000001FFFFFF, 0xFE00000003FFFFFF, 0xFC00000007FFFFFF,
0xF80000000FFFFFFF, 0xF00000001FFFFFFF, 0xE00000003FFFFFFF, 0xC00000007FFFFFFF,
0x80000000FFFFFFFF, 0x00000001FFFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFF00000000, 0xFFFFFFFE00000001, 0xFFFFFFFC00000003, 0xFFFFFFF800000007,
0xFFFFFFF00000000F, 0xFFFFFFE00000001F, 0xFFFFFFC00000003F, 0xFFFFFF800000007F,
0xFFFFFF00000000FF, 0xFFFFFE00000001FF, 0xFFFFFC00000003FF, 0xFFFFF800000007FF,
0xFFFFF00000000FFF, 0xFFFFE00000001FFF, 0xFFFFC00000003FFF, 0xFFFF800000007FFF,
```

```
0xFFFF00000000FFFF, 0xFFFE00000001FFFF, 0xFFFC00000003FFFF, 0xFFF800000007FFFF,
0xFFF00000000FFFFF, 0xFFE00000001FFFFF, 0xFFC00000003FFFFF, 0xFF800000007FFFFF,
0xFF00000000FFFFFF, 0xFE00000001FFFFFF, 0xFC00000003FFFFFF, 0xF800000007FFFFFF,
0xF00000000FFFFFFF, 0xE00000001FFFFFFF, 0xC00000003FFFFFFF, 0x800000007FFFFFFF,
0x00000000FFFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFE00000000, 0xFFFFFFFC00000001, 0xFFFFFFF800000003, 0xFFFFFFF000000007,
0xFFFFFFE00000000F, 0xFFFFFFC00000001F, 0xFFFFFF800000003F, 0xFFFFFF000000007F,
0xFFFFFE00000000FF, 0xFFFFFC00000001FF, 0xFFFFF800000003FF, 0xFFFFF000000007FF,
0xFFFFE00000000FFF, 0xFFFFC00000001FFF, 0xFFFF800000003FFF, 0xFFFF000000007FFF,
0xFFFE00000000FFFF, 0xFFFC00000001FFFF, 0xFFF800000003FFFF, 0xFFF000000007FFFF,
0xFFE00000000FFFFF, 0xFFC00000001FFFFF, 0xFF800000003FFFFF, 0xFF000000007FFFFF,
0xFE00000000FFFFFF, 0xFC00000001FFFFFF, 0xF800000003FFFFFF, 0xF000000007FFFFFF,
0xE00000000FFFFFFF, 0xC00000001FFFFFFF, 0x800000003FFFFFFF, 0x000000007FFFFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFFC00000000, 0xFFFFFFF800000001, 0xFFFFFFF000000003, 0xFFFFFFE000000007,
0xFFFFFFC00000000F, 0xFFFFFF800000001F, 0xFFFFFF000000003F, 0xFFFFFE000000007F,
0xFFFFFC00000000FF, 0xFFFFF800000001FF, 0xFFFFF000000003FF, 0xFFFFE000000007FF,
0xFFFFC00000000FFF, 0xFFFF800000001FFF, 0xFFFF000000003FFF, 0xFFFE000000007FFF,
0xFFFC00000000FFFF, 0xFFF800000001FFFF, 0xFFF000000003FFFF, 0xFFE000000007FFFF,
0xFFC00000000FFFFF, 0xFF800000001FFFFF, 0xFF000000003FFFFF, 0xFE000000007FFFFF,
0xFC00000000FFFFFF, 0xF800000001FFFFFF, 0xF000000003FFFFFF, 0xE000000007FFFFFF,
0xC00000000FFFFFFF, 0x800000001FFFFFFF, 0x000000003FFFFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFF800000000, 0xFFFFFFF000000001, 0xFFFFFFE000000003, 0xFFFFFFC000000007,
0xFFFFFF800000000F, 0xFFFFFF000000001F, 0xFFFFFE000000003F, 0xFFFFFC000000007F,
0xFFFFF800000000FF, 0xFFFFF000000001FF, 0xFFFFE000000003FF, 0xFFFFC000000007FF,
0xFFFF800000000FFF, 0xFFFF000000001FFF, 0xFFFE000000003FFF, 0xFFFC000000007FFF,
0xFFF800000000FFFF, 0xFFF000000001FFFF, 0xFFE000000003FFFF, 0xFFC000000007FFFF,
0xFF800000000FFFFF, 0xFF000000001FFFFF, 0xFE000000003FFFFF, 0xFC000000007FFFFF,
0xF800000000FFFFFF, 0xF000000001FFFFFF, 0xE000000003FFFFFF, 0xC000000007FFFFFF,
0x800000000FFFFFFF, 0x000000001FFFFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFFF0000000000, 0xFFFFFFE000000001, 0xFFFFFFC000000003, 0xFFFFFF8000000007,
0xFFFFFF000000000F, 0xFFFFFE000000001F, 0xFFFFFC000000003F, 0xFFFFF8000000007F,
0xFFFFF000000000FF, 0xFFFFE000000001FF, 0xFFFFC000000003FF, 0xFFFF8000000007FF,
0xFFFF000000000FFF, 0xFFFE000000001FFF, 0xFFFC000000003FFF, 0xFFF8000000007FFF,
0xFFF000000000FFFF, 0xFFE000000001FFFF, 0xFFC000000003FFFF, 0xFF8000000007FFFF,
0xFF000000000FFFFF, 0xFE000000001FFFFF, 0xFC000000003FFFFF, 0xF8000000007FFFFF,
```

```
    0xF000000000FFFFFF, 0xE000000001FFFFFF, 0xC000000003FFFFFF, 0x8000000007FFFFFF,
    0x000000000FFFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0xFFFFFFE000000000, 0xFFFFFFC000000001, 0xFFFFFF8000000003, 0xFFFFFF0000000007,
    0xFFFFFE000000000F, 0xFFFFFC000000001F, 0xFFFFF8000000003F, 0xFFFFF0000000007F,
    0xFFFFE000000000FF, 0xFFFFC000000001FF, 0xFFFF8000000003FF, 0xFFFF0000000007FF,
    0xFFFE000000000FFF, 0xFFFC000000001FFF, 0xFFF8000000003FFF, 0xFFF0000000007FFF,
    0xFFE000000000FFFF, 0xFFC000000001FFFF, 0xFF8000000003FFFF, 0xFF0000000007FFFF,
    0xFE000000000FFFFF, 0xFC000000001FFFFF, 0xF8000000003FFFFF, 0xF0000000007FFFFF,
    0xE000000000FFFFFF, 0xC000000001FFFFFF, 0x8000000003FFFFFF, 0x0000000007FFFFFF,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0xFFFFFFC000000000, 0xFFFFFF8000000001, 0xFFFFFF0000000003, 0xFFFFFE0000000007,
    0xFFFFFC000000000F, 0xFFFFF8000000001F, 0xFFFFF0000000003F, 0xFFFFE0000000007F,
    0xFFFFC000000000FF, 0xFFFF8000000001FF, 0xFFFF0000000003FF, 0xFFFE0000000007FF,
    0xFFFC000000000FFF, 0xFFF8000000001FFF, 0xFFF0000000003FFF, 0xFFE0000000007FFF,
    0xFFC000000000FFFF, 0xFF8000000001FFFF, 0xFF0000000003FFFF, 0xFE0000000007FFFF,
    0xFC000000000FFFFF, 0xF8000000001FFFFF, 0xF0000000003FFFFF, 0xE0000000007FFFFF,
    0xC000000000FFFFFF, 0x8000000001FFFFFF, 0x0000000003FFFFFF, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0xFFFFFF8000000000, 0xFFFFFF0000000001, 0xFFFFFE0000000003, 0xFFFFFC0000000007,
    0xFFFFF8000000000F, 0xFFFFF0000000001F, 0xFFFFE0000000003F, 0xFFFFC0000000007F,
    0xFFFF8000000000FF, 0xFFFF0000000001FF, 0xFFFE0000000003FF, 0xFFFC0000000007FF,
    0xFFF8000000000FFF, 0xFFF0000000001FFF, 0xFFE0000000003FFF, 0xFFC0000000007FFF,
    0xFF8000000000FFFF, 0xFF0000000001FFFF, 0xFE0000000003FFFF, 0xFC0000000007FFFF,
    0xF8000000000FFFFF, 0xF0000000001FFFFF, 0xE0000000003FFFFF, 0xC0000000007FFFFF,
    0x8000000000FFFFFF, 0x0000000001FFFFFF, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0xFFFFFF0000000000, 0xFFFFFE0000000001, 0xFFFFFC0000000003, 0xFFFFF80000000007,
    0xFFFFF0000000000F, 0xFFFFE0000000001F, 0xFFFFC0000000003F, 0xFFFF80000000007F,
    0xFFFF0000000000FF, 0xFFFE0000000001FF, 0xFFFC0000000003FF, 0xFFF80000000007FF,
    0xFFF0000000000FFF, 0xFFE0000000001FFF, 0xFFC0000000003FFF, 0xFF80000000007FFF,
    0xFF0000000000FFFF, 0xFE0000000001FFFF, 0xFC0000000003FFFF, 0xF80000000007FFFF,
    0xF0000000000FFFFF, 0xE0000000001FFFFF, 0xC0000000003FFFFF, 0x80000000007FFFFF,
    0x0000000000FFFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
    0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFE0000000000, 0xFFFFFC0000000001, 0xFFFFF80000000003, 0xFFFFF00000000007,
0xFFFFE0000000000F, 0xFFFFC0000000001F, 0xFFFF80000000003F, 0xFFFF00000000007F,
0xFFFE0000000000FF, 0xFFFC0000000001FF, 0xFFF80000000003FF, 0xFFF00000000007FF,
0xFFE0000000000FFF, 0xFFC0000000001FFF, 0xFF80000000003FFF, 0xFF00000000007FFF,
0xFE0000000000FFFF, 0xFC0000000001FFFF, 0xF80000000003FFFF, 0xF00000000007FFFF,
0xE0000000000FFFFF, 0xC0000000001FFFFF, 0x80000000003FFFFF, 0x00000000007FFFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFFC0000000000, 0xFFFFF80000000001, 0xFFFFF00000000003, 0xFFFFE00000000007,
0xFFFFC0000000000F, 0xFFFF80000000001F, 0xFFFF00000000003F, 0xFFFE00000000007F,
0xFFFC0000000000FF, 0xFFF80000000001FF, 0xFFF00000000003FF, 0xFFE00000000007FF,
0xFFC0000000000FFF, 0xFF80000000001FFF, 0xFF00000000003FFF, 0xFE00000000007FFF,
0xFC0000000000FFFF, 0xF80000000001FFFF, 0xF00000000003FFFF, 0xE00000000007FFFF,
0xC0000000000FFFFF, 0x80000000001FFFFF, 0x00000000003FFFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFF80000000000, 0xFFFFF00000000001, 0xFFFFE00000000003, 0xFFFFC00000000007,
0xFFFF80000000000F, 0xFFFF00000000001F, 0xFFFE00000000003F, 0xFFFC00000000007F,
0xFFF80000000000FF, 0xFFF00000000001FF, 0xFFE00000000003FF, 0xFFC00000000007FF,
0xFF80000000000FFF, 0xFF00000000001FFF, 0xFE00000000003FFF, 0xFC00000000007FFF,
0xF80000000000FFFF, 0xF00000000001FFFF, 0xE00000000003FFFF, 0xC00000000007FFFF,
0x80000000000FFFFF, 0x00000000001FFFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFF00000000000, 0xFFFFE00000000001, 0xFFFFC00000000003, 0xFFFF800000000007,
0xFFFF00000000000F, 0xFFFE00000000001F, 0xFFFC00000000003F, 0xFFF800000000007F,
0xFFF00000000000FF, 0xFFE00000000001FF, 0xFFC00000000003FF, 0xFF800000000007FF,
0xFF00000000000FFF, 0xFE00000000001FFF, 0xFC00000000003FFF, 0xF800000000007FFF,
0xF00000000000FFFF, 0xE00000000001FFFF, 0xC00000000003FFFF, 0x800000000007FFFF,
0x00000000000FFFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFE00000000000, 0xFFFFC00000000001, 0xFFFF800000000003, 0xFFFF000000000007,
0xFFFE00000000000F, 0xFFFC00000000001F, 0xFFF800000000003F, 0xFFF000000000007F,
0xFFE00000000000FF, 0xFFC00000000001FF, 0xFF800000000003FF, 0xFF000000000007FF,
0xFE00000000000FFF, 0xFC00000000001FFF, 0xF800000000003FFF, 0xF000000000007FFF,
0xE00000000000FFFF, 0xC00000000001FFFF, 0x800000000003FFFF, 0x000000000007FFFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFFC00000000000, 0xFFFF800000000001, 0xFFFF000000000003, 0xFFFE000000000007,
0xFFFC00000000000F, 0xFFF800000000001F, 0xFFF000000000003F, 0xFFE000000000007F,
0xFFC00000000000FF, 0xFF800000000001FF, 0xFF000000000003FF, 0xFE000000000007FF,
0xFC00000000000FFF, 0xF800000000001FFF, 0xF000000000003FFF, 0xE000000000007FFF,
0xC00000000000FFFF, 0x800000000001FFFF, 0x000000000003FFFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFF800000000000, 0xFFFF000000000001, 0xFFFE000000000003, 0xFFFC000000000007,
0xFFF800000000000F, 0xFFF000000000001F, 0xFFE000000000003F, 0xFFC000000000007F,
0xFF800000000000FF, 0xFF000000000001FF, 0xFE000000000003FF, 0xFC000000000007FF,
0xF800000000000FFF, 0xF000000000001FFF, 0xE000000000003FFF, 0xC000000000007FFF,
0x800000000000FFFF, 0x000000000001FFFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFF000000000000, 0xFFFE000000000001, 0xFFFC000000000003, 0xFFF8000000000007,
0xFFF000000000000F, 0xFFE000000000001F, 0xFFC000000000003F, 0xFF8000000000007F,
0xFF000000000000FF, 0xFE000000000001FF, 0xFC000000000003FF, 0xF8000000000007FF,
0xF000000000000FFF, 0xE000000000001FFF, 0xC000000000003FFF, 0x8000000000007FFF,
0x000000000000FFFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFE000000000000, 0xFFFC000000000001, 0xFFF8000000000003, 0xFFF0000000000007,
0xFFE000000000000F, 0xFFC000000000001F, 0xFF8000000000003F, 0xFF0000000000007F,
0xFE000000000000FF, 0xFC000000000001FF, 0xF8000000000003FF, 0xF0000000000007FF,
0xE000000000000FFF, 0xC000000000001FFF, 0x8000000000003FFF, 0x0000000000007FFF,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFFC000000000000, 0xFFF8000000000001, 0xFFF0000000000003, 0xFFE0000000000007,
0xFFC000000000000F, 0xFF8000000000001F, 0xFF0000000000003F, 0xFE0000000000007F,
0xFC000000000000FF, 0xF8000000000001FF, 0xF0000000000003FF, 0xE0000000000007FF,
0xC000000000000FFF, 0x8000000000001FFF, 0x0000000000003FFF, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFF8000000000000, 0xFFF0000000000001, 0xFFE0000000000003, 0xFFC0000000000007,
0xFF8000000000000F, 0xFF0000000000001F, 0xFE0000000000003F, 0xFC0000000000007F,
0xF8000000000000FF, 0xF0000000000001FF, 0xE0000000000003FF, 0xC0000000000007FF,
0x8000000000000FFF, 0x0000000000001FFF, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFFF0000000000000, 0xFFE0000000000001, 0xFFC0000000000003, 0xFF80000000000007,
0xFF0000000000000F, 0xFE0000000000001F, 0xFC0000000000003F, 0xF80000000000007F,
0xF000000000000FF, 0xE000000000001FF, 0xC000000000003FF, 0x8000000000007FF,
0x0000000000000FFF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0xFFE0000000000000, 0xFFC0000000000001, 0xFF80000000000003, 0xFF00000000000007,
  0xFE0000000000000F, 0xFC0000000000001F, 0xF80000000000003F, 0xF00000000000007F,
  0xE0000000000000FF, 0xC0000000000001FF, 0x80000000000003FF, 0x00000000000007FF,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0xFFC0000000000000, 0xFF80000000000001, 0xFF00000000000003, 0xFE00000000000007,
  0xFC0000000000000F, 0xF80000000000001F, 0xF00000000000003F, 0xE00000000000007F,
  0xC0000000000000FF, 0x80000000000001FF, 0x00000000000003FF, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0xFF80000000000000, 0xFF00000000000001, 0xFE00000000000003, 0xFC00000000000007,
  0xF80000000000000F, 0xF00000000000001F, 0xE00000000000003F, 0xC00000000000007F,
  0x80000000000000FF, 0x00000000000001FF, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0xFF00000000000000, 0xFE00000000000001, 0xFC00000000000003, 0xF800000000000007,
  0xF00000000000000F, 0xE00000000000001F, 0xC00000000000003F, 0x800000000000007F,
  0x00000000000000FF, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
0xFE00000000000000, 0xFC00000000000001, 0xF800000000000003, 0xF000000000000007,
0xE00000000000000F, 0xC00000000000001F, 0x800000000000003F, 0x000000000000007F,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xFC00000000000000, 0xF800000000000001, 0xF000000000000003, 0xE000000000000007,
0xC00000000000000F, 0x800000000000001F, 0x000000000000003F, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xF800000000000000, 0xF000000000000001, 0xE000000000000003, 0xC000000000000007,
0x800000000000000F, 0x000000000000001F, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xF000000000000000, 0xE000000000000001, 0xC000000000000003, 0x8000000000000007,
0x000000000000000F, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
0xE000000000000000, 0xC000000000000001, 0x8000000000000003, 0x0000000000000007,
0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
```

```
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0xC000000000000000, 0x8000000000000001, 0x0000000000000003, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x8000000000000000, 0x0000000000000001, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
  0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000
  };
```

6 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft® Excel® 15 Technical Preview
- Microsoft® SQL Server® 2008 R2
- Microsoft® SQL Server® 2012

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 2.1.2.3.1: Excel 15 Technical Preview restricts the collation name to a list of strings that can be obtained by executing the following query against SQL Server 2008 R2:

```
SELECT * FROM FN:HELPCOLLATIONS() WHERE NAME NOT LIKE 'SQL%'
```

<2> Section 2.1.2.3.1.1: Excel 15 Technical Preview accepts only strings that can be obtained by executing the following query against SQL Server 2008 R2:

```
SELECT * FROM FN:HELPCOLLATIONS() WHERE NAME NOT LIKE 'SQL%'
```

<3> Section 2.2.1: Excel 15 Technical Preview normalizes **UserID** as described in [MS-SSAS] section 2.2.4.2.1.2.

<4> Section 2.2.2.2: Excel 15 Technical Preview generates a data source folder for the model. The folder is empty.

<5> Section 2.3.1: Excel 15 Technical Preview assigns data identifiers in the order in which each unique value is encountered in the source data.

<6> Section 2.3.1: Excel 15 Technical Preview uses partial sorting to optimize compression.

<7> Section 2.5.2.3.1: Excel 15 Technical Preview restricts the collation name to a list of strings that can be obtained by executing the following query against SQL Server 2008 R2:

```
SELECT * FROM FN:HELPCOLLATIONS() WHERE NAME NOT LIKE 'SQL%'
```

<8> Section 2.6.9: Excel 15 Technical Preview inserts the text for the command as stated in the document.

7 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

8 Index

A

Applicability 12

C

Change tracking 244

D

Dictionary File example 222

E

Examples
  Dictionary File 222
  Multiple-Segment Column Data .idf File 220
  tbl.xml Metadata File 206

F

Fields - vendor-extensible 13

G

Glossary 10

I

Implementer - security considerations 225
Index of security parameters 225
Informative references 11
Introduction 10

L

Localization 13

M

Multiple-Segment Column Data .idf File example 220

N

Normative references 11

O

Overview (synopsis) 12

P

Parameters - security index 225
Product behavior 243

R

References 11
  informative 11
  normative 11
Relationship to protocols and other structures 12

S

Security
  implementer considerations 225
  parameter index 225

T

tbl.xml Metadata File example 206
Tracking changes 244

V

Vendor-extensible fields 13
Versioning 13