IBM Integration Bus V9 Best practices
Version 9 Release 0

# Reducing memory usage with parsers and message trees

By:
Craig Briscoe
David Crighton
Paul Faulkner
Vivek Grover
Stef Pugsley

V1.0

# Contents

# Chapter 1. Reduce your memory usage with parsers and message trees

Every message flow uses parsers, and therefore they are integral to all message flows. Parsers construct a message tree from incoming data, in an Input node for example, and then serialize a bitstream from the message tree, for example in an Output node. When a message flow processes larger messages, the message tree can grow and use more memory than the bitstream itself.

Parsers are used to own the incoming message data, construct a message tree from that incoming data, and then serialize a bitstream from a message tree. When a message flow processes larger messages, the message tree can grow and use more memory than the bitstream itself.

The information in this guide is intended as a guide to not only reduce the memory usage of your current message flows, but also offer examples and good practices for planning and creating future flows.

The best practice information in the following sections share common approaches to solving common problems based on real customer environments. They do not provide a "one size fits all" solution. They assume that you have a basic understanding of IBM® Integration Bus. As technology evolves and improved functionality is added to the product, new recommendations and advice might be added to the information in these documents.

# Chapter 2. Introduction

Messages are routed and transformed by message flows within IBM Integration Bus. As a message passes through the message flow, it is manipulated by message processing nodes that are often implemented in ESQL (Extended Structured Query Language). The message is stored in memory in the form of a message tree, which is created from the input message bitstream and serialized into an output bitstream by a parser.

When you process small messages, for example messages that are a few kilobytes in size, only a small amount of memory is needed to fully parse the message.

However, when you process large messages, the amount of memory and processing time that is required to store the message tree can quickly increase. The size of a message tree can vary substantially, and it is directly proportional to the size of the messages that are being processed: The larger the messages, the larger the message tree. The inefficient use of parsers, using the wrong parser, or even parsing when you do not need to, can all increase the amount of memory that is used by the integration node.

When the message size grows to megabytes, it becomes even more important to keep memory usage to a minimum. If all your memory is being used by processing large messages, then there might not be enough to maintain your expected level of service. Also, there might not be enough resources for other processes, such as deployment, to run smoothly.

The best practice information in the following sections share common approaches to solving common problems based on real customer environments. They do not provide a "one size fits all" solution. They assume that you have a basic understanding of IBM Integration Bus. As technology evolves and improved functionality is added to the product, new recommendations and advice might be added to the information in these documents.

# Chapter 3. Terms and phrases

Parsers, messages, and message trees are so interconnected, that it is difficult to discuss one aspect without using terms and phrases from another. Therefore, this section introduces key words, terms, and phrases that are used throughout this document.

*Table 1.*

| **Term** | **Meaning** | **Further reading** |
|---|---|---|
| Body folder | The *Body folder* is the last child of *Root*, and contains the message payload. | |
| Domain | Each parser is suited to a particular class of messages (such as fixed-length binary, delimited text, or XML) known as a message *domain*. Each message that is to be processed by a message flow must be associated with a domain. A domain determines the parser that is used when parsing and serializing the message. Each domain is suited to a particular class of messages, and some domains support several different classes of message. The domain for an input message is typically specified on the input node of the message flow. | |
| Folder | *Folders* are elements of the message tree under *Root*, such as Properties or Body. | |
| Fragmentation | *Fragmentation* means that the memory heap has enough free space on the current heap, but has no contiguous blocks that are large enough to satisfy the current request. | |
| Logical Tree | The logical tree structure is the internal (broker) representation of a message. It is also known as the *Message Assembly*. | For more information, see Logical tree structure in the main product documentation. |
| Message Tree | A message tree is a structure that is created, either by one or more parsers when an input message bit stream is received by a message flow, or by the action of a message flow node | For more information, see The message tree in the main product documentation. |

*Table 1. (continued)*

| **Term** | **Meaning** | **Further reading** |
|---|---|---|
| On-Demand Parsing | *On-demand parsing*, also referred to as *partial parsing*, is used to parse an input message bit stream only as far as is necessary to satisfy the current reference. | For more information, see Parsing on demand in the main product documentation. |
| Opaque Parsing | For XMLNSC messages, you can use *Opaque parsing*: A technique that allows the whole of an XML sub tree to be placed in the message tree as a single element. You can use opaque parsing where you do not need to access the elements of the subtree. | For more information, see "Opaque parsing" on page 22. |
| Partial Parsing | *Partial parsing*, also referred to as *on-demand parsing*, is used to parse an input message bit stream only as far as is necessary to satisfy the current reference. | For more information, see Parsing on demand in the main product documentation. |
| Root | The root of a message tree is called Root. The last element beneath the root of the message tree is always the message body. | For more information, see Message tree structure in the main product documentation. |
| Serialization | When an output message is created in a message flow, the message tree must be converted back into an actual message bitstream. This conversion is done by the parser and the process is referred to as *serialization* of the message tree. The creation of the output message is a simpler process than reading an incoming message. | |

# Chapter 4. Parsers

Every message flow uses parsers, and therefore they are integral to all message flows. Parsers construct a message tree from incoming data, in an Input node for example, and then serialize a bitstream from the message tree, for example in an Output node. When a message flow processes larger messages, the message tree can grow and use more memory than the bitstream itself.

Before the bitstream is processed, it must be interpreted by a parser in the message flow. The parser builds a representation of the message data that is called a *logical tree* or a *message tree* from the bitstream of the message data. This message tree can then be interpreted, transformed, and processed. The message tree has contents that are identical to the message, but the message tree is easier to manipulate in the message flow. The message flow nodes provide an interface to query, update, or create the content of the tree.

When an output message is created in a message flow, the message tree must be converted back into an actual message bitstream. This conversion is done by the parser. The process of creating the output message is referred to as *serialization* of the message tree. The creation of the output message is a simpler process than reading an incoming message: The whole message is written immediately after an output message is created.

A parser is typically created implicitly by node operations. However, they can also be created manually as part of language function/procedure calls using ESQL, Java™, PHP, or .NET.

Parsers are used within the IBM Integration Bus nodes to perform the following necessary tasks:

- For input nodes and get nodes, parsers are created to own the required portions of the incoming message data.
- For output nodes and reply nodes, parsers create output message data that is sent on the transport.
- For request nodes, parsers create the request message data that is sent on the transport, and they own the required portions of the incoming response.
- For transformation nodes, such as Compute or Mapping, parsers are associated with the output trees.
- For each message tree that represents the message, *Root* and *Properties* parsers are created to own and represent the content of the other parsers.

- For each of the other message trees (Environment, LocalEnvironment, and ExceptionList), a Root parser is created to represent the message tree.
- Each ESQL ROW variable (shared or otherwise) is represented by a Root parser.

Parser usage and the size of the message tree usually have the largest impact on memory and the performance of the message flow because the parsers are directly related to the size of the incoming message data.

## Selecting a parser

When you select a parser, your decision must be based on the characteristics of the messages that your applications exchange.

IBM Integration Bus provides a range of parsers to handle the following messaging standards in use:
- XMLNSC
- DFDL
- MIME
- JMSMap
- JMSStream
- BLOB
- SOAP
- IDoc

The following parsers are provided but deprecated:
- MRM-XML
- MRM-CWF
- MRM-TDS

The parsers parse AND SERIALIZE the messages into and out of the message flow, with different parsers having different performance costs.

Each parser can process either:
- The message body data of messages in a particular message domain, XML for example.
- particular message or transport headers, MQMD for example.

You must review the messages that your applications send through the IBM Integration Bus, and determine which message domain the message body data belongs to. The following table lists your application requirements, and the recommended parser to use:

*Table 2.*

| **Application requirements** | **Parser domain** |
|---|---|
| Use SOAP-based web services, including SOAP with Attachments (MIME) or MTOM | SOAP |
| Uses JSON format, as maybe used in RESTful web services | JSON |
| Your application data is in XML format other than SOAP | XMLNSC |
| Data comes from a C or COBOL application, or consists of fixed-format binary data | DFDL |
| Data consists of formatted text, perhaps with field content that is identified by tags, or separated by specific delimiters, or both | DFDL |
| Data is from a WebSphere Adapter such as the adapters for SAP, PeopleSoft, or Siebel | DataObject |
| Data is in SAP text IDoc format, such as those exported using the WebSphere MQ Link for R3 | DFDL |
| Data is in MIME format, other than SOAP, with attachments (for example, RosettaNet). You do not know, or do not have to know, the content of your application data | MIME |
| You do not know, or do not have to know, the content of your application data | BLOB |

It is recommended to use compact parsers such as XMLNSC for XML parsing and DFDL for non-XML parsing. The benefit of compact parsers is that they discard white space and comments in a message. Therefore those portions of the input message are not populated in the message tree, which keeps memory usage down.

While parsers are created implicitly by node operations, they can also be created manually as part of language function/procedure calls:

## Parser creation and memory usage

*Partial Parsing*, sometimes referred to as *on-demand parsing*, is the most efficient way to parse in IBM Integration Bus.

For more information relating specifically to partial parsing, see Partial parsing in “Parsing strategies” on page 18. The rest of this section discusses parsers and their memory usage in general.

Messages are parsed from the start of the message, and proceeds as far along the message as required to access the element that is being referred to in the flow though the processing logic (ESQL, Java, XPath, Graphical Data Mapper map, and so on). Depending on the field that is being accessed, it might not be necessary to parse the whole message. Only the portion of the message that is parsed is populated in the message tree. The rest is held as an unprocessed bitstream. The remainder might be parsed later in the flow if there is logic that requires it, or it might never be parsed if it is not required as part of the message flow processing.

When a message flow receives message data in one of its input nodes, parsers are created to handle that incoming data. Multiple parsers are created and chained to own each of the transport headers and then a body parser, such as XMLNSC, DFDL, or JSON, is created to own the message body. IBM Integration Bus also creates a Properties parser to represent the properties of the message tree, and a Root parser is created to own all these parser instances.

For example, when a message arrives on an MQInput node, you might have an incoming WebSphere® MQ message with the following message structure:

```
Root
 - Properties
   - ...
 - MQMD
   - ...
 - MQRFH2
   - ...
 - MQCIH
   - ...
 - XMLNSC
   - ...
```

This message tree has 6 parsers that represent it. For each main folder that is created from Root by a built-in node, a parser is created to own that folder.

The following example shows the message tree in full:

```
['MQROOT' : 0x642f630]
  Properties = ( ['MQPROPERTYPARSER' : 0x6366008]
  MessageSet            = '' (CHARACTER)
  MessageType           = '' (CHARACTER)
  MessageFormat         = '' (CHARACTER)
  Encoding              = 546 (INTEGER)
  CodedCharSetId        = 437 (INTEGER)
  Transactional         = TRUE (BOOLEAN)
  Persistence           = FALSE (BOOLEAN)
  CreationTime          = GMTTIMESTAMP '2015-01-14 13:14:12.350' (GMTTIMESTAMP)
  ExpirationTime        = -1 (INTEGER)
  Priority              = 0 (INTEGER)
  ReplyIdentifier       = X'000000000000000000000000000000000000000000000000' (BLOB)
  ReplyProtocol         = 'MQ' (CHARACTER)
  Topic                 = NULL
  ContentType           = '' (CHARACTER)
  IdentitySourceType    = '' (CHARACTER)
  IdentitySourceToken   = '' (CHARACTER)
  IdentitySourcePassword = '' (CHARACTER)
  IdentitySourceIssuedBy = '' (CHARACTER)
  IdentityMappedType     = '' (CHARACTER)
  IdentityMappedToken    = '' (CHARACTER)
  IdentityMappedPassword = '' (CHARACTER)
  IdentityMappedIssuedBy = '' (CHARACTER)
  )
  MQMD       = ( ['MQHMD' : 0x63433c8]
  SourceQueue      = 'IN' (CHARACTER)
  Transactional    = TRUE (BOOLEAN)
  Encoding         = 546 (INTEGER)
  CodedCharSetId   = 437 (INTEGER)
  Format           = '        ' (CHARACTER)
  Version          = 2 (INTEGER)
  Report           = 0 (INTEGER)
  MsgType          = 8 (INTEGER)
  Expiry           = -1 (INTEGER)
  Feedback         = 0 (INTEGER)
  Priority         = 0 (INTEGER)
  Persistence      = 0 (INTEGER)
  MsgId            = X'414d512042524b36312020202020202006e06d4920001a04' (BLOB)
  CorrelId         = X'000000000000000000000000000000000000000000000000' (BLOB)
  BackoutCount     = 0 (INTEGER)
  ReplyToQ         = '                                                ' (CHARACTER)
  ReplyToQMgr      = 'V9IN                                           ' (CHARACTER)
  UserIdentifier   = 'vgrover    ' (CHARACTER)
  AccountingToken  = X'16010515000000060022a3f33997b361af17567f401000000000000000000000b' (BLOB)
  ApplIdentityData = '                                ' (CHARACTER)
  PutApplType      = 11 (INTEGER)
  PutApplName      = 'rfhutil.exe' (CHARACTER)
  PutDate          = DATE '2015-01-14' (DATE)
  PutTime          = GMTTIME '13:14:12.350' (GMTTIME)
  ApplOriginData   = '    ' (CHARACTER)
  GroupId          = X'000000000000000000000000000000000000000000000000' (BLOB)
  MsgSeqNumber     = 1 (INTEGER)
  Offset           = 0 (INTEGER)
  MsgFlags         = 0 (INTEGER)
  OriginalLength   = -1 (INTEGER)
  )
MQRFH2 {
  MQCHAR4  StrucId;
  MQLONG   Version;
  MQLONG   StrucLength;
  MQLONG   Encoding;
  MQLONG   CodedCharSetId;
  MQCHAR8  Format;
  MQLONG   Flags;
  MQLONG   NameValueCCSID;
  } MQRFH2;
 )
  MQCIH      = (
  Version            = 2
  Format             = '        '
  Encoding           = 2
  CodedCharSetId     = 437
  Flags              = 0
  ReturnCode         = 0
  CompCode           = 0
  Reason             = 0
  UOWControl         = 273
  GetWaitInterval    = -2
  LinkType           = 1
  OutputDataLength   = -1
  FacilityKeepTime   = 0
  ADSDescriptor      = 0
  ConversationalTask = 0
  TaskEndStatus      = 0
  Facility           = X'0000000000000000'
  Function           = '    '
  AbendCode          = '    '
  Authenticator      = '        '
  Reserved1          = '        '
  ReplyToFormat      = '        '
  RemoteSysId        = '    '
  RemoteTransId      = '    '
  TransactionId      = '    '
  FacilityLike       = '    '
  AttentionId        = '    '
  StartCode          = '    '
  CancelCode         = '    '
  NextTransactionId  = '    '
  Reserved2          = '        '
  Reserved3          = '        '
  CursorPosition     = 0
  ErrorOffset        = 0
  InputItem          = 0
  Reserved4          = 0
  )

  XMLNSC     = ( ['xmlnsc' : 0x63669b8]
  root = (
  Folders  = (
  Folder1 = (
  SubField = '1' (CHARACTER)
        )
        Folder2 = (
        SubField = '1' (CHARACTER)
        )
      )
      Fields   = (
      Field1 = 'field1Value' (CHARACTER)
      Field2 = 'field1Value' (CHARACTER)
      Field3 = 'field1Value' (CHARACTER)
      )
      Elements = (
      Field1  = 'field1Value' (CHARACTER)
      Folder1 = (
      SubField = '1' (CHARACTER)
        )
      Field2  = 'field1Value' (CHARACTER)
      Field3  = 'field1Value' (CHARACTER)
      Folder2 = (
      SubField = '1' (CHARACTER)
        )
     )
   )
)
```

If this message tree goes through a transformation node, then a new output message tree is created that might have none, some or all of the contents of the input message tree. This output tree is a separate message tree from the input message tree. As such, a new set of parsers are created to own the parser folders in the output tree. That is, there is not just one parser per domain, multiple instances can be created based on the message flow logic

So in the previous example, if this tree were copied with ESQL such as `SET OutputRoot = InputRoot;` then you would now have 12 parsers that were created by this message flow. It is clear to see that the more transformation nodes that are used, the more parsers are created to own the new output message trees. Therefore, when you consider copying message trees, the cost of the extra parsers must be considered.

The request nodes, such as the SOAPRequest nodes, are similar to a transformation node, in that a different message tree is propagated to the one that is received on the input node. The input message tree is used to form a request message for which a response to is received, usually from a remote server. The response message is then parsed using the parser parameters that are specified on the request node. This sequence means that the propagated response message creates a new set of parsers such as Root, Properties, and Header parsers, and a body parser. A message flow developer does not often have any choice on which request nodes are used in a message flow, and as such cannot make any improvements in this area.

The transformation nodes also allow a message flow developer to create parser instances manually. This creation can be done explicitly using method/function call:

- In ESQL the CREATE with the DOMAIN clause creates a parser of the named domain.
- In Java the MbElement **`create*UsingParser`** methods create a parser of the named parser name.
- In .Net the NbElement **`create*UsingParser`** methods create a parser of the named parser name.
- In the C Plugin interface the **`cniCreate*UsingParser`** methods create a parser of the named parser name.

The creation can also be done implicitly when constructing **`MbMessage`**, **`NbMessage`**, or **`CciMessage`** objects.

Although built-in transformation nodes construct one set of Output trees only, the Java, C, and .Net APIs allow many sets of output message objects to be created. Therefore it is important to understand the scope of these objects and associated parsers.

Consider a scenario where multiple bitstream portions are retrieved from an external source and need to be mapped into an output message as shown in the following ESQL example:

```
-- Bitstream portions are in Environment.Variables.MsgData[]
DECLARE outRef REFERENCE TO OutputRoot;
CREATE LASTCHILD OF outRef AS outRef DOMAIN('XMLNSC') NAME 'XMLNSC';
CREATE LASTCHILD OF outRef AS outRef NAME 'TestCase';
DECLARE envRef REFERENCE TO Environment.Variables.MsgData;
DECLARE parseOptions INTEGER BITOR(RootBitStream, ValidateNone);
WHILE LASTMOVE(envRef) DO
  CREATE LASTCHILD OF Environment DOMAIN('DFDL') PARSE(envRef OPTIONS parseOptions TYPE '{}:CSV_DFDL');
  CREATE LASTCHILD OF outRef NAME 'Record';
  SET outRef = Environment.DFDL;
  MOVE envRef NEXTSIBLING NAME 'MsgData';
END WHILE;
```

In this example, the `CREATE` with the `DOMAIN` clause creates a DFDL parser in each iteration of the WHILE loop. Each of these DFDL parsers is a new DFDL parser to the one on the previous iteration. Therefore, if this loop iterates 2000 times, then 2000 DFDL parsers are created. More parsers means more memory usage, so resources are rapidly consumed with this approach. The following sections describe ways to mitigate the memory problems that are associated with this kind of scenario.

## Parser considerations: Managing memory usage

There are a number of techniques that can be used to keep memory usage to a minimum and these are described in the following sections:

### The DELETE statement

Creating many parsers has a detrimental impact on the memory that is used because all the parsers are in memory at the same time. The DELETE statement detaches and deletes a portion of a message tree, allowing its memory to be reused.

Looking at the scenario that is described in the previous section “Parser creation and memory usage” on page 9, of the multiple bitstream portions that are retrieved from an external source and mapped into an output message, a message flow developer can clean up the resources that are created during the loop with the DELETE statement. The following example demonstrates this behavior by adding a DELETE statement into the previous example:

```
-- Bitstream portions are in Environment.Variables.MsgData[]
DECLARE outRef REFERENCE TO OutputRoot;
CREATE LASTCHILD OF outRef AS outRef DOMAIN('XMLNSC') NAME 'XMLNSC';
CREATE LASTCHILD OF outRef AS outRef NAME 'TestCase';
DECLARE envRef REFERENCE TO Environment.Variables.MsgData;
DECLARE parseOptions INTEGER BITOR(RootBitStream, ValidateNone);
WHILE LASTMOVE(envRef) DO
  CREATE LASTCHILD OF Environment DOMAIN('DFDL') PARSE(envRef OPTIONS parseOptions TYPE '{}:CSV_DFDL');
  CREATE LASTCHILD OF outRef NAME 'Record';
  SET outRef = Environment.DFDL;
  DELETE FIELD Environment.DFDL;
  MOVE envRef NEXTSIBLING NAME 'MsgData';
END WHILE;
```

The following image shows the message tree before and after the DELETE statement is used:

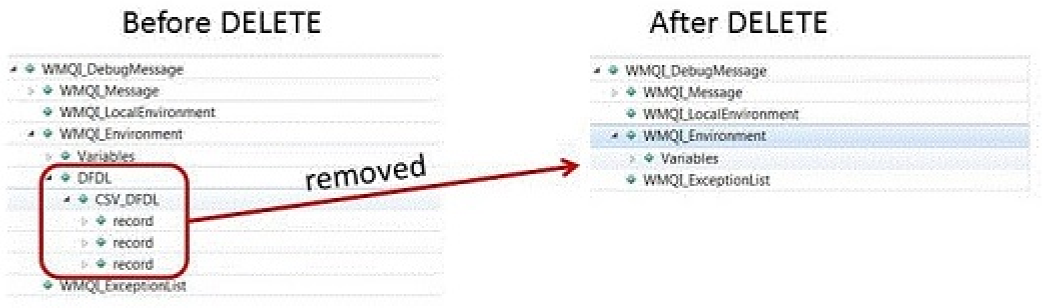


*Figure 1.*

The recommendations in this section do not apply if your parsers are being created in the OutputRoot trees (or the equivalent message objects in Java), and the trees are propagated. For more information, see "The Splitter pattern" on page 58.

### MQSI_FREE_MASTER_PARSERS

When you consider message flow processing from the message data point of view, you might implement a number of common patterns.

The five patterns are as follows:

1. **`Routing`**: Receive input message and route the content unchanged (The headers might be updated).
2. **`Augmentation`**: Receive input message, update the content, and write the changed content to a different destination.
3. **`Transformation`**: Receive input message, and then map all or some of its content to a different domain.
4. **`Splitter`**: A large input message is received, and is then broken into many smaller output records that are then output.
5. **`Collector`**: Many small records are collected into a much larger output message.

In any of these types of scenarios, remember the following points:

- The input, output, or both might be large.
- The input data, output data, or both might be validated.

How the logic is implemented for #1 and #5 has a significant impact on both performance and memory usage. The six external aspects of message processing that affect how much memory a message flow consumes are as follows:

1. The size of the incoming bitstream.

2. The number of message tree fields that are created in the message tree that represents the incoming bitstream.
3. How often the message tree is copied in the message flow.
4. The number of message tree fields in any output, request, or reply message trees.
5. Whether any output, request, or reply message trees are serialized, and the size of these trees.
6. Whether other parsers are created to handle portions of a message tree or bitstream in the formation of an output message tree.

To summarize: Multiple instances of parsers can be created. In language nodes, the instances might be created to excess. If a large message is parsed or created without using the appropriate techniques, then many message tree fields can be created.

If extraneous parsers, fields, or both are created, then a valid question to ask is:

When are these parsers cleaned up?

When a parser is created, it is owned by the creating thread, and cannot be used by any other thread. This situation means that if a message flow has extra instances, then each instance has its own set of parsers. Similarly, if a message flow has multiple input nodes, then a thread serves each input node, which means that there are multiple message flow threads. If a message flow does not use appropriate techniques for managing parser and tree creation, then extra threads amplify memory usage issues.

Because an input node controls the processing of input message data, it also owns the pool of parsers that are used by the message flow. Generally, any parser that is created within the message flow is assigned to this master parser pool. If a parser is created in a SHARED ROW variable, then this parser is owned by the execution group not a message flow thread.

For any nodes that create new output trees (transformation or request nodes), they request the parsers from this thread level master pool of parsers. The parser is returned to the pool for reuse when the following events happen:
1. The message flow execution unwinds such that node is no longer on the stack. For example: All downstream nodes were executed and the next path is propagated to, or an exception is thrown such that a previous Try-Catch node Catch terminal is propagated to.
2. `ESQL PROPAGATE` or MbOutputTerminal.propagate(MbMessageAssembly, true) is returned from, and all conditions were met to free the parsers for reuse.

3. **DELETE FIELD** (or equivalent or language APIs) was issued and all fields for the parser were deleted. This action is likely to happen when the delete is issued on a parsers root element. The following image shows the extra parsers that are to be deleted:

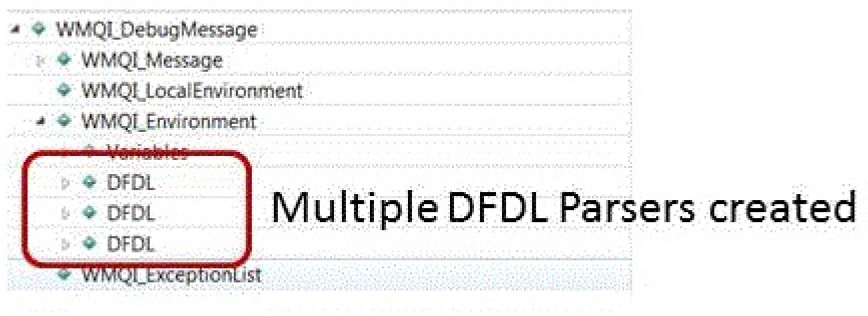

WMQI_DebugMessage
WMQI_Message
WMQI_LocalEnvironment
WMQI_Environment
Variables
DFDL
DFDL
DFDL
WMQI_ExceptionList

*Figure 2.*

When a parser is freed for reuse, none of its resources are deleted. It is returned to the pool of parsers and keeps all of its message tree field allocation that is cached for reuse. When the message flow completes the processing of the input message, by default it does not delete any parser resources. Instead, it resets all the parsers that were created, ready for reuse, and maintains all the message tree fields within them. The logic here is that when a message flow executes its main path repeatedly, it is likely to process similar size messages using the same nodes and order. Therefore, for performance reasons, the same parsers are maintained with the same resources. However, this behavior does have the consequence that if a message flow creates many parsers, or creates many message tree fields in one or more parsers, then these parsers are maintained for the life of the message flow.

When this situation causes memory exhaustion issues, the MQSI_FREE_MASTER_PARSERS environment variable can be used. When this variable is set, it gives the following behavior:

1. When a message flow finishes processing an input message, all the parsers and their resources are deleted.
2. The memory that is occupied by the parser is returned to the heap, NOT to the operating system. That is, the user does not see a reduction in the memory that is occupied by the DataFlowEngine process. However, the blocks of memory can be reused for other executions that require memory.
3. This alternative processing does not occur mid-flow. It deletes the parsers only when the input node finishes with the input message.

From IBM Integration Bus V9.0.0.0 onwards, if you implement good operating practice for managing parser creation and message trees, then you do not need to use the MQSI_FREE_MASTER_PARSERS environment variable. You do not need to use the variable because a message flow does not need to create many parsers or allocate many message tree fields.

If best practices are implemented to ensure that extraneous parsers and fields are not created, you must be aware that only the user aspects of the data are affected. When parsing or serializing messages, the parse mechanics allocate memory to process the bitstream or tree, and these allocations are unaffected by any parser or tree management techniques.

The environment variable MQSI_FREE_MASTER_PARSERS can be exported in the profile of the broker service ID. When MQSI_FREE_MASTER_PARSERS is set, the parsers are freed after every message instead of caching them to be reused. The following steps can be followed on the broker machine:

1. Export MQSI_FREE_MASTER_PARSERS=YES
2. Restart the broker

## Parsing strategies

You can use several efficient parsing strategies during the message flow development to reduce memory usage when you parse and serialize messages. This section describes *Partial parsing*, *Opaque parsing*, and how to avoid unnecessary parsing.

These strategies are as follows:

- "Identifying the message type quickly"
- "Partial parsing" on page 21
- "Opaque parsing" on page 22
- "Avoiding unnecessary parsing" on page 25

These strategies are described in detail in the following sections.

### Identifying the message type quickly

It is important to be able to correctly recognize the correct message format and type as quickly as possible. In message flows that process multiple types of message, this identification can be a problem. What often happens is that the message needs to be parsed multiple times to ensure that you have the correct format. That extra parsing to determine the message type needs to be avoided to reduce memory usage.

The message flow in Figure 3 on page 19 shows a number of Filter nodes (*Filter1* & *Filter2*), several subflows (*subflow1*, *subflow2*, and so on) each containing more nodes. The message flow is complex and is implemented with a long critical path. As such, messages are parsed multiple times.

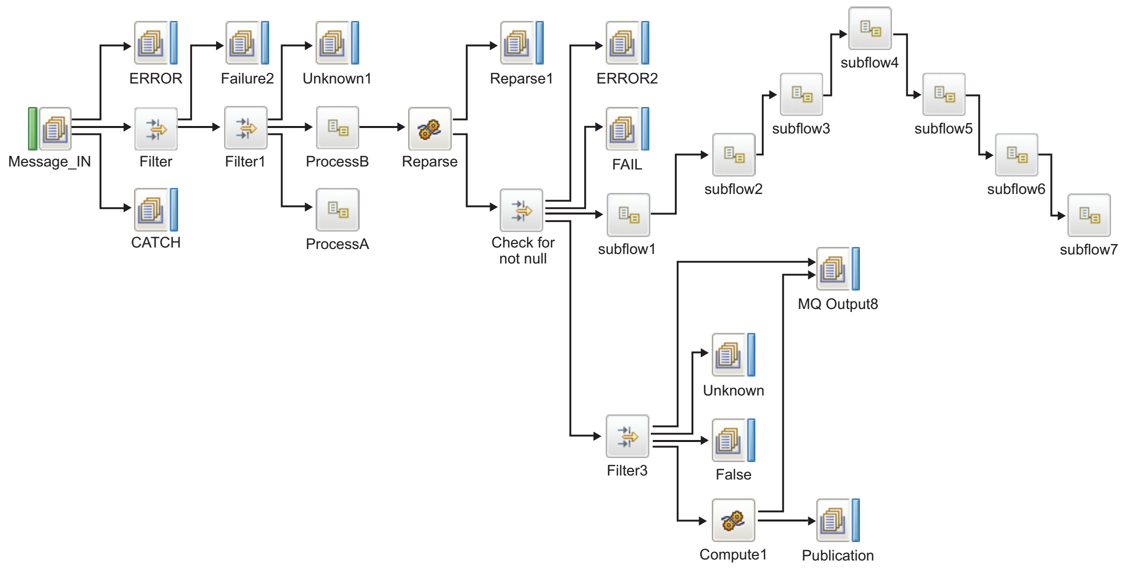


*Figure 3.*

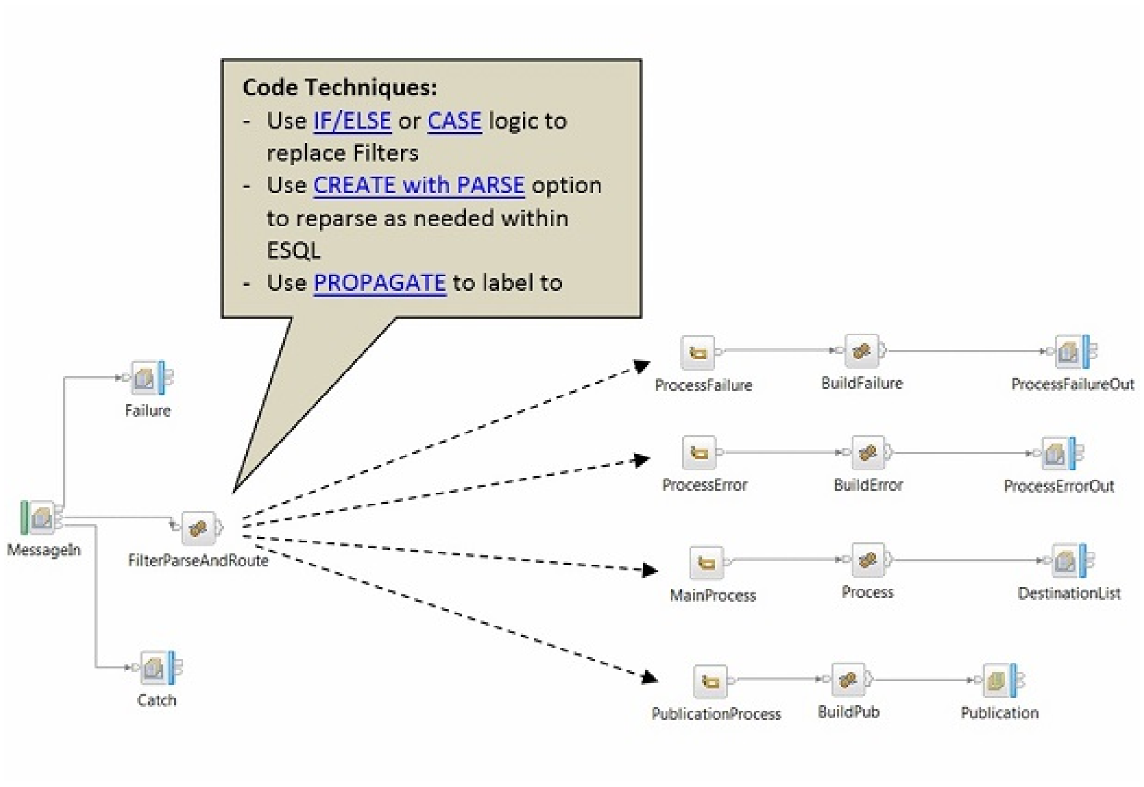


*Figure 4.*

In this example, the use of functions and procedures, ESQL parsing techniques, and dynamic routing of the flow are combined into the minimum number of nodes (excludes error handling subflow nodes). The logic for each of the paths is coded as a function or procedure, and called from the main procedure in the Compute node. This method also avoids the multiple parsing of messages that is executed in the multiple Filter nodes and subflows. This method significantly reduces the performance cost due to fewer nodes, and ultimately leads to less parsing, tree copying, and so on, for the most optimized solution.

**Tip:** You must remember; when you section a large, complex flow into multiple smaller message flows allows the individual message flows to release memory after each large step of processing. That is, after each smaller flow finishes its processing. So, you need to ensure that a balanced approach is adopted between these strategies to get the optimum memory and performance usage.

## Partial parsing

A message is parsed only when necessary to resolve the reference to a particular part of its content. An input message can be of any length, and parsing the entire message for only a specific part of content is not usually required. *Partial parsing* (also referred to as *On-demand parsing*) improves the performance of message flows, and reduces the amount of parsed data that is stored in memory. Partial parsing is used to parse an input message bit stream only as far as is necessary to satisfy the current reference.

To use Partial parsing, you must set the **Parse timing** property on the input node to `On Demand`

All the parsers that are provided with IBM Integration Bus support partial parsing. The amount of parsing that must be performed depends on which fields in a message need to be accessed, and the position of those fields in the message. In the next two diagrams, one has the fields ordered A to Z (Figure 5) and the other with them ordered Z to A (Figure 6). Depending on which field is needed, one of the cases is more efficient than the other. If you need to access field Z, then the first case would be best. Where you have influence over message design ensure that information that is needed for routing for example is placed at the start of the message and not at the end of the message.

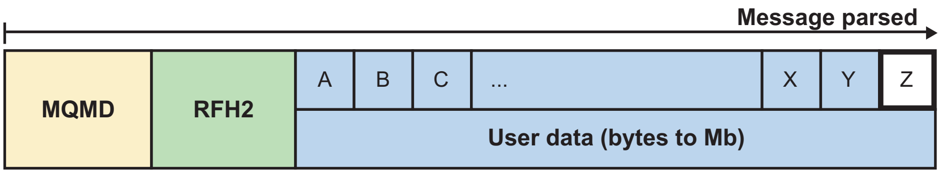


*Figure 5.*

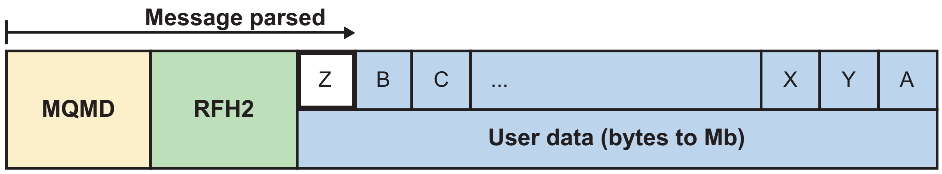


*Figure 6.*

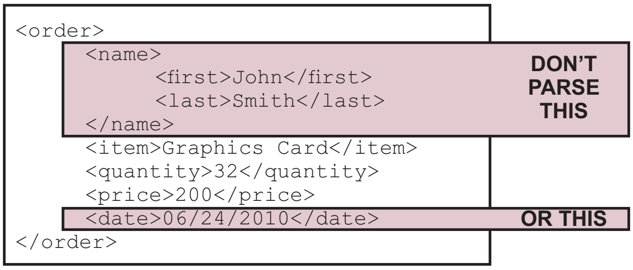
When you use ESQL and Mapping nodes, the field references are typically explicit. That is, you have references such as InputRoot.Body.A. IBM Integration Bus parses only as far as the required message field to satisfy that reference. The parser stops at the first instance. When you use the XPath query language, the situation is different. By default, an XPath expression searches for all instances of an element in the message, which implicitly means that a full parse of the message takes place. If you know that there is only one element in a message, then there is the chance to optimize the XPath query, for example, to retrieve only the first instance. For example, `/aaa[1]` if you want just the first instance of the search argument.

### Opaque parsing

For XMLNSC messages, you can use Opaque parsing: A technique that allows the whole of an XML sub tree to be placed in the message tree as a single element.

Opaque parsing is supported for the XMLNS and XMLNSC domains only.

Use the XMLNSC domain in new message flows if you want to use opaque parsing. The XMLNS domain is deprecated, and offers a more limited opaque parsing facility than the XMLNSC domain. The XMLNS domain is provided only to support legacy message flows.



```
<order>
    <name>
          <first>John</first>
          <last>Smith</last>
    </name>
    <item>Graphics Card</item>
    <quantity>32</quantity>
    <price>200</price>
    <date>06/24/2010</date>
</order>
```

*Figure 7.*

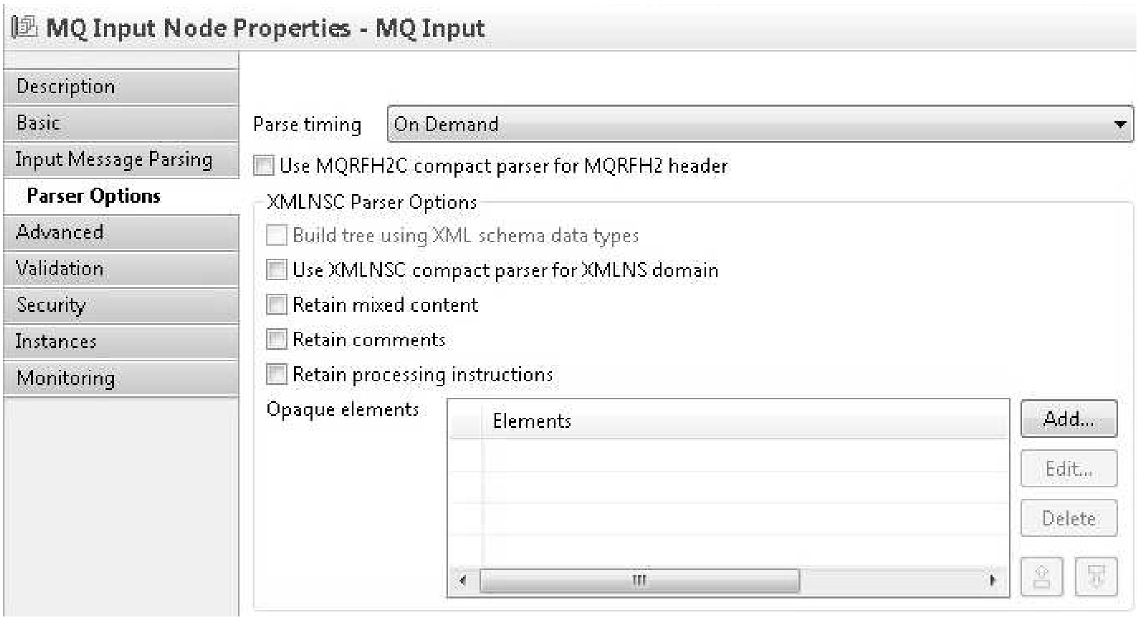
The entry in the message tree is the bitstream of the original input message. This technique has two benefits:

1. It reduces the size of the message tree because the XML subtree is not expanded into the individual elements.
2. The cost of parsing is reduced because less of the input message is expanded as individual elements and added to the message tree.

You can use opaque parsing where you do not need to access the elements of the subtree. For example, you need to copy a portion of the input tree to the output message but might not care about the contents in this particular message flow. You accept the content in the subfolder and have no need to validate or process it in any way.

Specifying elements for opaque parsing

You must specify elements for opaque parsing in the **Parser Options** section of the Input node of the message flow, as shown in Figure 8 on page 23:



MQ Input Node Properties - MQ Input

- Description
- Basic
- Input Message Parsing
- **Parser Options**
- Advanced
- Validation
- Security
- Instances
- Monitoring

Parse timing: On Demand

- [ ] Use MQRFH2C compact parser for MQRFH2 header

XMLNSC Parser Options

- [ ] Build tree using XML schema data types
- [ ] Use XMLNSC compact parser for XMLNS domain
- [ ] Retain mixed content
- [ ] Retain comments
- [ ] Retain processing instructions

Opaque elements:

| Elements |
|---|

Figure 8.

To specify elements for opaque parsing, add the element names to the **Opaque elements** table. Ensure that message validation is not enabled, otherwise it automatically disables opaque parsing. Opaque parsing does not make sense for validation, because the whole message must be parsed and validated.

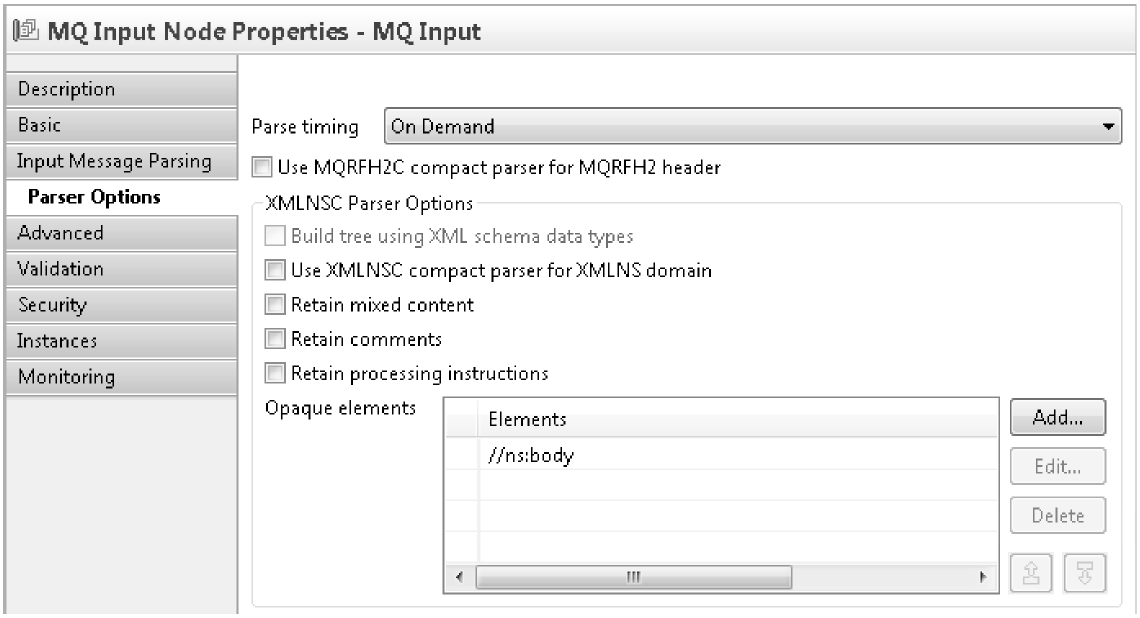
**Tip:** Opaque parsing for the named elements occurs automatically when the message is parsed. It is not possible to use the CREATE statement to opaquely parse a message in the XMLNSC domain; only node options can be used to add opaque parsing.

Opaque parsing in action

With typical on-demand parsing, if you need to access fields in the header and trailer sections of the message then the whole message must be parsed. In this example, you have a flow that needs to access the <*version*> and <*type*> fields only. The following message structure shows those structures, and is abbreviated for clarity:

```
<tns:Inventory xmlns:tns="http://www.example.org/NewXMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.example.org/NewXMLSchema Inventory.xsd ">
 <tns:header>  <tns:version>V100</tns:version>
 </tns:header>
 <tns:body>
  <tns:field1>tns:field1</tns:field1>
  <tns:field2>tns:field2</tns:field2>
.....
  <tns:field1000>tns:field2</tns:field1000>
</tns:body>
<tns:trailer>
  <tns:type>tns:type</tns:type>
 </tns:trailer>
</tns:Inventory>
```

Using opaque parsing, you can eliminate the need to parse the body section of the payload. You need to set the parent of the elements that you do not want to parse in the **Parser Options** section of the Input node of the message flow, as shown in Figure 9.



Parse timing: On Demand

- [ ] Use MQRFH2C compact parser for MQRFH2 header

XMLNSC Parser Options

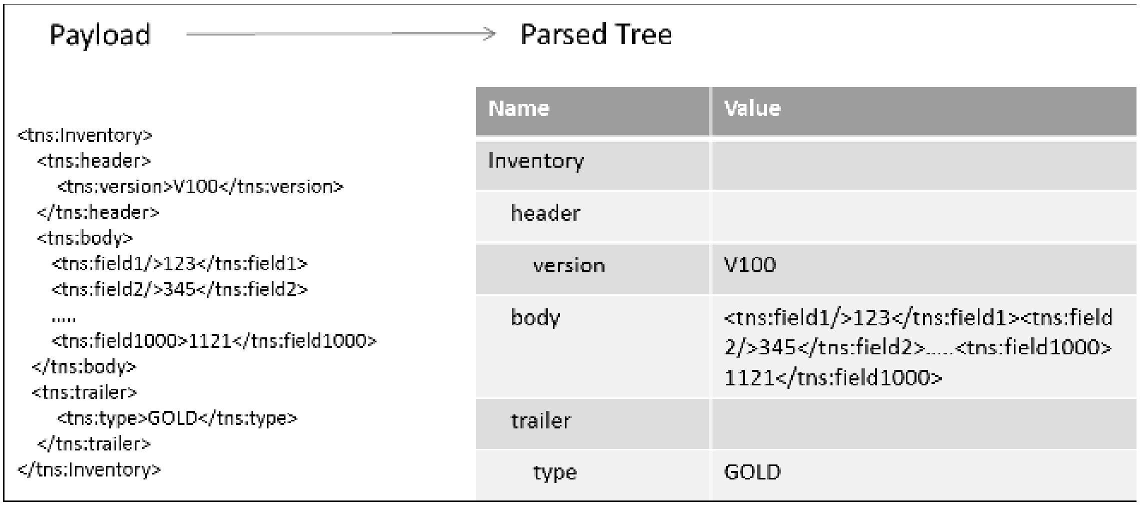
- [ ] Build tree using XML schema data types
- [ ] Use XMLNSC compact parser for XMLNS domain
- [ ] Retain mixed content
- [ ] Retain comments
- [ ] Retain processing instructions

Opaque elements

| Elements |
|---|
| //ns:body |

*Figure 9.*

All elements that are defined in the **Opaque elements** list are treated as a single string when parsed. This parsing behavior is shown in Figure 10 on page 25.



Payload → Parsed Tree

```
<tns:Inventory>
  <tns:header>
    <tns:version>V100</tns:version>
  </tns:header>
  <tns:body>
   <tns:field1/>123</tns:field1>
   <tns:field2/>345</tns:field2>
   .....
   <tns:field1000>1121</tns:field1000>
  </tns:body>
  <tns:trailer>
    <tns:type>GOLD</tns:type>
  </tns:trailer>
</tns:Inventory>
```

| Name | Value |
|---|---|
| Inventory | |
| header | |
| version | V100 |
| body | <tns:field1/>123</tns:field1><tns:field2/>345</tns:field2>.....<tns:field1000>1121</tns:field1000> |
| trailer | |
| type | GOLD |

*Figure 10.*

When you design a message structure, if you have the opportunity to group elements based on the parsing needs, then this method greatly improves performance. In the previous example, if you move the <*type*> field into the header, there would be no need for opaque parsing: The on-demand parser would not need to go past the header in this example.

## Avoiding unnecessary parsing

One effective technique to reduce the cost of parsing, is not to parse.

The strategy is to avoid having to parse some parts of the message as shown in Figure 11.

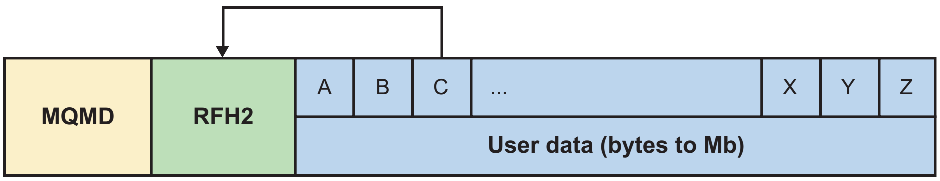


*Figure 11.*

For example:

You have a message routing message flow that needs to look at a field to make a routing decision. If that field is in the body of the message, then the body of the incoming message must be parsed to get access to it. The processing cost varies depending on which field is needed:

- If it is field A, then it is right at the beginning of the body and would be found quickly.

- If it is field Z, then the cost might be different, especially if the message is several megabytes in size.

Here is a technique to reduce this cost:

Use the application that created this message to copy the field that is needed for routing into a header within the message. For an WebSphere MQ message, this field might an MQRFH2 header, and a JMS property for a JMS message for example. If you use this technique, it is no longer necessary to parse the message body, potentially saving a large amount of processing effort. The MQRFH2 or JMS Properties folder still needs to be parsed, but with a smaller amount of data. The parsers in this case are also more efficient than the general parser for a message body because the structure of the header is known. Copy key data structures to MQMD, MQRFH2, or JMS Properties to prevent parsing the user data.

## Identifying problem message flows

When you need to find out whether message flows are creating an excessive number of message tree fields, then parser resource statics can provide the answer.

The statistics show the number of *Fields* in memory, and the approximate amount of user data-related memory that is used for the named message flow parser type. These statistics are grouped by message flow and name, and you can quickly identify when millions of fields are in memory, or gigabytes of data storage is being used.

### Resource statistics

*Resource statistics* are collected by a broker to record performance and operating details of resources that are used by integration servers.

As a system administrator, you can use the resource statistics to ensure that your systems are using the available resources in the most efficient manner. By monitoring systems and analyzing statistical trends, you can keep system resource usage within boundaries that you consider acceptable, and help to prevent situations where system resources are overstretched and might become unavailable. Analysis of the data that is returned might require specialist skills and knowledge of each resource type.

If you detect that system resources are under pressure, you can examine the statistics that are collected by the broker to assess whether the cause of the concern is the use of those resources by processes in IBM Integration Bus.

You must activate statistics collection because collection is not active by default. If you activate statistics, you might experience a minor degradation in operating performance of the broker or brokers for which you are collecting data. You can collect data on one or more integration servers, and all integration servers on a broker, so that you can limit the statistics gathering activity if appropriate.

Resource statistics complement the accounting and statistics data that you can collect on message flows, which are also available in the IBM Integration Explorer.

To start, stop, or check the status of resource statistics collection, use one or more of the following options:

- The IBM Integration Explorer: See the *Starting resource statistics collection in the IBM Integration Explorer* section in the main product documentation.
- See the *mqsichangeresourcestats command* section in the main product documentation.
- See the *mqsireportresourcestats command* section in the main product documentation.
- A CMP application: See the *Working with resource statistics in a CMP application* section in the main product documentation.

To view the output that is generated by statistics collection, use one or more of the following options:

- The IBM Integration Explorer. Numeric data and graphs are displayed for each integration server with activated statistics collection.
- An application that subscribes to a publication message, that is published by the broker every 20 seconds. The message contains the data that is collected for each integration server with activated statistics collection. The published message is available in XML format and in JSON format.

  The topic for each message has the following structure:
  - For XML format:

    `$SYS/Broker/`*`broker_name`*`/ResourceStatistics/`*`integration_server_name`*
  - For JSON format:

    `$SYS/Broker/`*`broker_name`*`/Statistics/JSON/Resource/`*`integration_server_name`*

  You can set up subscriptions for a specific integration server on a specific broker. For example:
  - For XML format:

    `$SYS/Broker/IB9NODE/ResourceStatistics/default`
  - For JSON format:

    `$SYS/Broker/IB9NODE/Statistics/JSON/Resource/default`

  You can also use wildcards in the subscriptions to broaden the scope of what is returned. For example, to subscribe to reports for all integration servers on all brokers, use the following values:
  - For XML format:

```
$SYS/Broker/+/ResourceStatistics/#
```

– For JSON format:

```
$SYS/Broker/+/Statistics/JSON/Resource/#
```

## Parser resource statistics

You can view these statistics in the IBM Integration Explorer, or you can write a program that subscribes to a publication (single XML message) that returns this data.

All message flows in an integration server create parsers to parse and write input and output messages. Use the Parsers statistics to see how much resource is being used by the message trees and bit streams that these parsers own.

A statistics summary is returned, followed by an entry for accumulation by parser type for each message flow. The rows are named in the style <*Message Flow*>.<*Parser*>. A row is shown for every parser type that is used by that message flow. Extra instances are included in the accumulated statistics for each message flow.

The following table describes the statistics that are returned for each message flow parser since the integration server was last restarted.

*Table 3.*

| **Measurements** | **Description** |
|---|---|
| Threads | The number of message flow threads that contributed to the statistics for a message flows parser type accumulation. |
| ApproxMemKB | The approximate amount of user data-related memory that is used for the named message flow parser type. An estimate of storage that the fields themselves are occupying in kilobytes. This value does include data that was in a specific message tree such as names, namespaces, and values. As such, this metric is intended to give a base for comparison from one message flow to another. It is not possible to calculate the exact amount of memory that is used by a parser, and so the returned value is the lowest it could possibly be. |
| MaxReadKB | This metric records the largest bitstream that is passed to the parser for parsing. For most transports, the input bitstream is a contiguous buffer of bytes, which is allocated when it was read in. Therefore, if `maxReadKB` is a large number, it might be contributing to a high integration server memory use. |
| MaxWrittenKB | Shows the largest bit stream that is written by the parser type for the named message flow. |

Table 3. (continued)

| Measurements | Description |
|---|---|
| Fields | Shows how many cached fields this parser is associated with. This number never reduces, even if the DELETE FIELD command is used, because deleted fields are reused. These fields are retained by the parser and are used for constructing the message trees. |
| Reads | The number of successful parses that were completed by the named message flow parser type. |
| TotalReads | The `totalReads` metric reports how many times the parser was assigned a bitstream to parse. These assignments might be as a result of a node reading data from a transport and assigning it to the parser, or it might be from a CREATE with PARSE clause in ESQL for example. |
| FailedReads | The `failedReads` metric reports the amount of `totalReads` that encountered a parsing exception. The `successfulReads` and `failedReads` add up to the `totalReads` for that parser. Examining the `failedReads` value might give an indication that a message flow is driving error handling paths, which might use more memory than expected. |
| Writes | The number of successful writes that were completed by the named message flow parser type. |
| FailedWrites | The `failedReads` metric reports the amount of `totalReads` that encountered a parsing exception. The `successfulReads` and `failedReads` add up to the `totalReads` for that parser. Examining the `failedReads` value might give an indication that a message flow is driving error handling paths, which might use more memory than expected. |

## Examples of reporting parser usage

The parser resource statistics do not show the number of parsers that were created. Only the following runtime command returns detailed information on each parser that was created on each thread:

```
mqsireportproperties broker -e integrationserver -o ComIbmParserManager -r
```

where *broker* is the name of your broker and *integrationserver* is the name of your integration server.

This command shows the number of parsers that are owned by each thread, and how many fields each one has. You can search the text output from this command for the keywords `totalFieldsUsed` or `approxMemKB` to see whether there are any excessively large numbers.

Capture the output of this command to a file and search for *totalParsers*. The results show the total number of parsers that were created on each thread. If this result is an excessive number, then the flow needs to be analyzed to see where these parsers are being created.

Although the number of parser instances is not recorded in the resource statistics, when an excessive amount of parsers are created the `totalFields` count is larger than expected. Even if a parser instance is not parsed to completion, it still creates a root element. This root element contributes to the `totalFields` count for a flow ParserName entry. So if a message flow accidentally created 100,000 parsers for example, then there would be at least 100,000 root elements that contribute to the total fields count.

The following text shows an example output from this **`mqsireportproperties`** command:

```
ComlbmParserManager
 uuid='ComlbmParserManager'
 userTraceLevel='none.
 traceLevel='none.
 userTraceFilter='none.
 traceFilter='none.
 vrmfintroducedAt=.7.0.0.2.
 resourceStatsReportingOn='inactive.
 resourceStatsMeasurements='<ResourceStatsSwitches ResourceType="Parsers" version='2' vrmfIntroducedAt='7.0.0.2'>
  <MeasurementV2 name="Threads" collect="on" /><MeasurementV2 name="ApproxMemKB" collect="on" />
  <MeasurementV2 name="MaxReadKB" collect="on" /><MeasurementV2 name="MaxWrittenKB" collect="on" />
  <MeasurementV2 name="Fields" collect="on" /><MeasurementV2 name="Reads" collect="on" />
  <MeasurementV2 name="FailedReads" collect="on" /><MeasurementV2 name="Writes" collect="on" />
  <MeasurementV2 name="FailedWrites" collect="on" />
  </ResourceStatsSwitches>'
 activityLogSupported='no'
Parser-Statistics
 Threads-Parsers
  Thread
   threadld='8160'
   threadName='Thread-8160'
   totalParsers='0'
   Parsers
  Thread
   threadld='1544'
   threadName='Thread-1L544'
   totalParsers='0'
   Parsers
  Thread
   threadld='11496'
   threadName='Thread-11496'
   totalParsers='7'
   Parsers
  Parser
   name='XMLNSC
   address='Oxla6fd0c0'
   type='xminsc'
   isShared='FALSE'
   creationTime='2015-10-13 21:31:35.177812'
   lastUsedTime=2015-10-13 21:31:35.177867'
   totalTimesUsed='1'
   approxMemKB='7.98'
   fields='44'
   totalFieldsUsed='44'
  Parser
   name='DFDL'
   address='Oxla180190'
   type='dfdl'
   isShared='FALSE'
   creationTime='2015-10-13 21:31:35.177921'
   lastUsedTime='2015-10-13 21:31:35.177939'
   totalTimesUsed='1'
   approxMemKB='7.98'
   fields='14'
   totalFieldsUsed='14'
   Parsing
    totalReads='1'
    maxReadKB='0.02'
    totalReadKB='0.02'
    successfulRead='l'
    fullReads='1'
    approxMemKB='7.B8'
    totalFieldsUsed='1'
    Parsing
     totalReads='1'
     maxReadKB='0.02'
     totalReadKB='0.02'
     successfulReads='1'
    Parser
     name='DFDL'
     address='0x1Bb37770'
     type='dfdl'
     isShared='FALSE'
     creationTime='2015-10-13 21:31:35.386184'
     lastUsedTime='2015-10-13 21:31:35.386188'
     totalTimesUsed='1'
     approxMemKB='7.98'
     fields='1'
     totalFieldsUsed='1'
     Parsing
      totalReads='1'
      maxReadKB='0.02'
      totalReadKB='0.02'
      successfulRead,l'

BIP80711: Successful command completion.
```

# Chapter 5. Message tree

A message tree is a structure that is created, either by one or more parsers when an input message bit stream is received by a message flow, or by the action of a message flow node.

A message is used to describe:

- A set of business data that is exchanged by applications
- A set of elements that are arranged in a predefined structure
- A structured sequence of bytes

IBM Integration Bus routes and manipulates messages after converting them into a *logical tree*. The process of conversion, called parsing, makes obvious the content and structure of a message, and simplifies later operations. After the message has been processed, the parser converts it back into a bit stream.

The *logical tree* structure is the internal (integration node) representation of a message. It is also known as the message assembly. When a message arrives at an integration node, it is received by an input node that you have configured in a message flow. Before the message can be processed by the message flow, the message must be interpreted by one or more parsers that create a logical tree representation from the bit stream of the message data.

The input node creates this message assembly, which consists of four trees:

1. Message tree structure
2. Environment tree structure
3. Local environment tree structure
4. Exception list tree structure

The first of these trees, the *Message tree structure*, is populated with the contents of the input message bit stream. The remaining three trees are initially empty.

The message tree is always present, and is passed from node to node in a single instance of a message flow. The root of a message tree is called Root. The message tree includes all the headers that are present in the message, in addition to the message body. If a supplied parser created a message tree, then the element that represents the **`Properties`** subtree is followed by zero or more headers.

When you design an IBM Integration Bus flow, it is essential to understand the concepts in the Message Tree copying section, and the effective way to navigate the tree. Both of these subjects can have a large impact on the overall performance of a flow. The following sections describe in more detail the considerations that you must take in to account when you develop a flow.

## References & navigating the message tree

Navigation is the process of accessing elements in the message tree. You can access them in the Compute, JavaCompute, .NETCompute, PHPCompute, and Mappingnodes. The cost of accessing elements is not always apparent and is difficult to separate from other processing costs.

The path to elements is not cached from one statement to another. Consider the following ESQL statement:

```
SET Description = InputBody.Level1.Level2.Level3.Description.Line[1];
```

When using the following message:

```
<Level1>
    <Level2>
        <Level3>
            <Description>
                <Line>1</Line>
            </Description>
        </Level3>
    </Level2>
</Level1>
```

The broker runtime accesses the message tree, starting at the correlation Root, and then moves down the tree to Level1, then Level2, then Level3, then Description, and finally it finds the first instance of the Line array. If you have this same statement in the next line of your ESQL module, then the same navigation takes place all over again. There is no cache to the last referenced element in the message tree because the access to the tree is intended to be dynamic to take account of the fact that it might change structure from one statement to another. This behavior can lead to repeated sets of navigation and traversing through the elements in a tree. If it is a large message tree, then the cost of navigation can become significant, leading to poor efficiency of your code.

Therefore, to reduce the memory usage of tree navigation, it is recommended that you use a reference variable (for ESQL) or reference pointers (for Java). Use this technique to save a pointer to a specific place in the message tree, and move it from one field to next as needed.

For example, you can use reference variables to refer to long correlation names such as *InputRoot.XMLNSC.Level1.Level2.Level3.Description*. Declare a reference pointer as shown in the following example:

```
DECLARE refDescPtr REFERENCE TO InputRoot.XMLNSC.Level1.Level2.Level3.Description;
```

Use the correlation name *refDescPtr.Line* to access the element *Line* of the message tree.

Use REFERENCE and MOVE statements to reduce the amount of navigation within the message tree and improve performance. This technique can be useful when you are constructing many SET or CREATE statements: Rather than navigating to the same branch in the tree, you can use a REFERENCE variable to establish a pointer to the branch and then use the MOVE statement to process one field at a time.

You have multiple reference variables that are defined within a flow. In the previous example, if your code needed to access fields at Level2 *InputRoot.XMLNSC.Level1.Level2* and Level3 *InputRoot.XMLNSC.Level1.Level2.Level3* for example, you would define a reference to Level2 and a reference to Level3 as shown in the following example:

```
DECLARE refToLevel2 REFERENCE TO InputRoot.XMLNSC.Level1.Level2;
DECLARE refToLevel3 REFERENCE TO refToLevel2.Level3;
```

The following example shows the ESQL that you can use to reduce the number of times that you navigate when you create new output message tree fields:

```
SET OutputRoot.XMLNSC.Level1.Level2.Level3.Description.Line[1]='1';
DECLARE outRef REFERENCE TO OutputRoot.XMLNSC.Level1.Level2.Level3.Description;
SET outRef.Line2 = '2';
SET outRef.Line3 = '3';
SET outRef.Line4 = '4';
SET outRef.Line5 = '5';
```

When you reference repeating input message tree fields, you can use the following ESQL:

```
DECLARE myChar CHAR;
DECLARE inputRef REFERENCE TO InputRoot.XMLNSC.Level1.Level2.Level3.Descritpion.Line[1];
WHILE LASTMOVE(inputRef) DO
        SET myChar = inputRef;
        MOVE inputRef NEXTSIBLING NAME 'Line';  --Repearing record
END WHILE;
```

In summary, use reference variables or reference pointers to avoid repeated traversing and referencing of the elements that are caused by loops and other ineffective tree navigation.

## Message tree copying

When a message passes through a Compute node, a copy of the message tree can be taken (specified by the **Compute Mode** property) for recovery reasons. If the Compute node changes the message during processing, or if it generates an exception, the message tree can be recovered to a point earlier in the message flow. This process of copying the message tree either in whole or part is called *message tree copying*.

Without this copy, a failure in the message flow downstream might have implications for a different path in the message flow. The tree copy is a copy of a structured object in memory, not a sequence of bytes, and so creating large copies or many copies of these structured objects often requires large amounts of memory. The logical tree is duplicated in memory across the message flow, and therefore to reduce the cost that is associated with message tree copying, you must remember the following points when you design your message flows:

- Minimize the number of message tree copies: It is recommended to minimize the number of Compute nodes in a message flow.
- Look for substitutes instead of using Compute node: For example, a Filter node might be used which does not copy the tree because ESQL references only the data in the message tree without updating it. Or using the **Compute Mode** property to control which components are used by default in the output message.
- Produce a smaller message tree in the first place: A smaller tree costs less to copy. You can produce a smaller message tree in the following ways:
  - Use smaller messages: "Message size versus Message tree" on page 48
  - Use compact parsers, such as XMLNSC and RFH2C: "Selecting a parser" on page 8
  - Use "Opaque parsing" on page 22
  - Use "Partial parsing" on page 21
- Reduce the number of times the whole tree is copied: Reducing the number of Compute nodes (ESQL or Java) helps to reduce the number of times that the whole tree needs to be copied. Avoid designs where you have one Compute node that is followed immediately by another in particular.

In many cases, you might need multiple compute nodes across a message flow. Do not force everything into a single Compute node in these cases. You can optimize the processing in the following ways:

- Copy portions of the tree at the branch level if possible rather than copying individual leaf nodes. This process works only where the structure of the source and destination are the same but it is worth doing if possible. For example, if we use the following input message:

```
<Request>
     <Parent>
         <Address>
             <City>Colorado Springs</City>
             <State>CO</State>
         </Address>
     </Parent>
</Request>
```

and use the following code to copy the **Address** branch to the output:

```
SET OutputRoot.XMLNSC.Target.Address = InputRoot.XMLNSC.Request.Parent.Address;
```

The following message tree is copied:

```
<Target>
    <Address>
        <City>Colorado Springs</City>
        <State>CO</State>
    </Address>
</Target>
```

- Copy data to the Environment tree and work with it in Environment so that the message tree does not have to be copied every time that you run a Compute node. Environment is a scratchpad area that exists for each invocation of a message flow. The contents of Environment are accessible across the whole of the message flow. For example, if we use the following input message:

```
<Request>
     <Parent>
         <Address>
             <City>Colorado Springs</City>
             <State>CO</State>
         </Address>
     </Parent>
</Request>
```

and use the following code to copy the **Address** branch to the Environment:

```
CREATE LASTCHILD OF Environment DOMAIN 'XMLNSC' NAME 'XMLNSC';
SET Environment.XMLNSC = InputRoot.XMLNSC;
```

Then the environment can be used throughout the flow, for example:

```
SET OutputRoot.XMLNSC.Target.Address = Environment.XMLNSC.Request.Parent.Address;
```

These examples result in the following message tree:

```
<Target>
    <Address>
        <City>Colorado Springs</City>
        <State>CO</State>
    </Address>
</Target>
```

- Where possible, set the **Compute Mode** property on the node to exclude the message. The **Compute Mode** property controls which components are used by default in the output message. You can select the property to specify whether the Message, LocalEnvironment, and Exception List components that are either generated in the node or contained in the incoming message are used. For more information, see Compute node in the main product documentation.
- The first Compute node in a message flow can copy InputRoot to Environment. Intermediate nodes then read and update values in the Environment instead of using the InputRoot and OutputRoot correlations.
- In the final Compute node of the message flow OutputRoot must be populated with data from Environment. The MQOutput node then serializes the message as usual. Serialization does not take place from Environment.

While the use of the Environment correlation is good from a performance point of view, be aware that any updates made by a node that then generate an exception remains in place. There is no back-out of changes as there is when a message tree copy was made before the node exception. For more information, see Chapter 7, “ESQL coding practices,” on page 65 and “Environment and Local Environment considerations” on page 66.

## Types of parser copy

When you create a changed output message, the message flow needs to copy the message tree. Nodes such as the Compute node, mapping nodes, and DecisionService node have input and output trees where the output trees are often different from the input trees. In other language nodes such as the JavaCompute node, Java plugin nodes, and .NET nodes, one or many output messages can be created. When you consider memory usage and performance in general, you must take care to determine when the message tree is copied.

When a tree is copied between different domains, the whole of the input tree must be parsed so that it can be copied to the new domain. This behavior is referred to as an *unlike parser copy*. Each individual field in the source message must be individually transferred to the target domain so that the correct structure and field types apply.

When both the source and target messages have the same folder structure at root level, there is the concept of the *like parser copy*, where a message tree is copied to a *like* domain. For example:

```
SET OutputRoot = InputRoot;
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XMLNSC = InputRoot.XMLNSC;
```

In a *like parser copy*, if the source message tree is unchanged since the input node, then a bitstream copy takes place. That is, the message tree is not copied and the new target folder is created using a bitstream reference to the input data. This default capability is useful in routing scenarios where none of the content is updated, or only parts of the headers are updated. Therefore, the potentially larger body can be transferred to the output tree without inflating it. For example:

```
SET OutputRoot = InputRoot;
SET Output.Properties.CorrelId = X'0102030405060708091011121314151617181920212223224';
```

However, if the codepage changes during a routing, then an MQInput node must inflate the output tree to reserialize the new bitstream.

The following example uses an `MQInput->Compute->MQOutput` flow, and the Computenode ESQL is as follows:

```
SET OutputRoot = InputRoot;SET
Output.Properties.CodedCharSetId = 500;
```

The MQInput node parses the output tree that it was passed. But it is important to know that even in this scenario, the original input tree that is received on the input node is not parsed. The high-speed bitstream copy for an unchanged message tree can be useful, and in fact large message handling techniques depend on this process.

In a Compute node, the InputRoot tree is not modifiable. So this technique relies on a high-speed bitstream copy to transfer the tree to the OutputRoot, such that the InputRoot is never inflated.

## Don't change a message tree and copy it

Do not change a message tree, and then copy it unnecessarily in other nodes.

If a message tree is ever changed, then the bitstream copy cannot take place. This situation means that ESQL such as `SET OutputRoot = InputRootRoot` copies the tree. Therefore, a message flow should avoid copying a message tree (especially a large one) after a new output message tree is generated. Consider the simple flow of; `MQInput->Compute1->Compute2->MQOutput` where the ESQL is as follows:

```
Compute1:
 - SET OutputRoot = InputRoot;
 - SET OutputRoot.XMLNSC.TestCase.LastUpdated = CURRENT_TIMESTAMP;
Compute2:
 - SET OutputRoot = InputRoot;
 - INSERT INTO Database.myDB .......
```

Because `Compute2` encounters the changed message tree, `Compute2` forces the whole of the InputRoot trees changed XMLNSC folder to be parsed to transfer it to the OutputRoot version.

The general advice that is given is for the message flow developer to ensure that all the updates are done to the message tree in the same node. However, in a flow where subflows are written by other message flow developers, then this behavior might not always be possible.

Therefore, in this type of scenario it would be better for the second Compute node `Compute2` not to issue the `SET OutputRoot = InputRoot` command, and instead set the **`ComputeMode`** property so that it is not `Message`.These settings mean that the node is not going to propagate a new/changed OutputRoot, and as such propagates the one it received.

The same technique can be used if `Compute2` were updating the LocalEnvironment with routing information. Here the **`ComputeMode`** property must be set to `LocalEnvironment`.

## Always copy at the parser folder level for the bitstream transfer to take place

When a message tree is copied, the high-speed bitstream transfer takes place only at the parser folder level. So `SET OutputRoot.XMLNSC = InputRoot.XMLNSC` considers creating the target XMLNSC folder using a bitstream copy as shown in the following example:

ESQL code such as:

```
SET OutputRoot.XMLNSC.TestCase = InputRoot.XMLNSC.TestCase
```

does not create the target XMLNSC folder with a bitstream. Instead, the message tree is copied in its entirety from `TestCase` downwards. For an XML message, this behavior still contains nearly all of the message tree.

## Parsers must be created when not copying to OutputRoot

Only the OutputRoot correlation name in ESQL has the special property that a parser is created for parser name folders. So in the case of `SET OutputRoot.XMLNSC = InputRoot.XMLNSC`, an XMLNSC parser is created on behalf of the flow in the OutputRoot message tree. Now consider the following ESQL:

```
SET Environment.Variables.XMLNSC = InputRoot.XMLNSC;
```

As the target field is in the Environment tree, an XMLNSC parser is not automatically created in the target, and as such an unlike parser copy takes place. This process means that the bitstream copy cannot take place, and the whole of the source tree is parsed and then copied to the Environment tree. The consequence of this behavior is that any XML-specific field types are lost. For example, if the Environment tree were copied back to the OutputRoot tree then any attributes would now be tags.

To preserve the XMLNSC nature of the target and ensure that a bitstream copy can take place, the ESQL must be as follows to correctly specify that the target folder is XMLNSC:

```
CREATE FIELD Environment.Variables.XMLNSC DOMAIN('XMLNSC');
SET Environment.Variables.XMLNSC = InputRoot.XMLNSC;
```

The same is true of the other language APIs when message trees are copied. The target element must be created specifying an owning parser before the copy takes place. Although ESQL OutputRoot has the special behavior such that the SET statement creates parsers for the direct children of OutputRoot, it is not true if OutputRoot is passed into a PROCEDURE/FUNCTION as a row variable. At this point, the ROW variable representation of OutputRoot no longer has the behavior of OutputRoot. This behavior is why OutputRoot is a globally recognized correlation name throughout all ESQL procedures and functions, and as such does not need to be passed in as a ROW variable.

## Avoid overwriting fields when a tree is copied

When a message tree is copied, any existing children of target message tree field are detached first. Detaching fields is not the same as deleting fields. Detaching fields means that they are still in scope, but are no longer attached to the tree. As such, the detached fields cannot be reused by the parser, and might cause more message tree fields to be created than intended.

In the following example a *RepeatingRecord* structure is copied to an XMLNSC tree so that the XML version of the record can be inserted into a database:

```
DECLARE inputRef REFERENCE TO InputRoot.DFDL.Parent.RepeatingRecord[1];
WHILE LASTMOVE(inputRef) = TRUE DO
  SET OutputRoot.XMLNSC.TestCase = inputRef
  DECLARE recordBytes BLOB ASBITSTREAM(OutputRoot.XMLNSC.TestCase)
  INSERT INTO Database.myDB(Records) VALUES(recordBytes);
  MOVE inputRef NEXTSIBLING NAME 'RepeatingRecord';
END WHILE;
```

If the *RepeatingRecord* structure has 10 fields in it, then each *OutputRoot.XMLNSC.TestCase* parent is created with 10 fields. On each iteration of the loop, these 10 fields are detached by the SET statement. Therefore if there are 100000 *RepeatingRecord* input structures, the XMLNSC parser in the OutputRoot tree creates 1 million fields that it did not need to. If the ESQL is modified to Delete the *TestCase* parent on each iteration of the loop, then the same 10 fields can be reused on each iteration. The following example demonstrates the preferred behavior:

```
DECLARE inputRef REFERENCE TO InputRoot.DFDL.Parent.RepeatingRecord[1];
WHILE LASTMOVE(inputRef) = TRUE  DO
  SET OutputRoot.XMLNSC.TestCase = inputRef
  DECLARE recordBytes BLOB ASBITSTREAM(OutputRoot.XMLNSC.TestCase)
  -- Record is now serialized, delete the Output fields we no longer need;
  DELETE FIELD OutputRoot.XMLNSC.TestCase;
  INSERT INTO Database.myDB(Records) VALUES(recordBytes);
  MOVE inputRef NEXTSIBLING NAME 'RepeatingRecord';
END WHILE;
```

## Forcing a copy of the message tree fields when required

In some message flow implementations, it might be necessary to ensure that the message tree fields are copied instead of the bitstream copy taking place.

When a bitstream copy takes place, the target parser starts from the beginning again. This behavior means that if any message tree fields are parsed in the source message tree, then they are not copied to the target parser. If a node parsed a large portion of the message tree already, then from a performance perspective it would be less expensive to copy the message tree than have the new target parser parse the message again. So a message flow would want to preserve the current message tree, especially in cases where other nodes (possibly in subflows) continue updating the same message tree. You could for example force all the children of a parser folder to be copied by using the following example:

```
SET OutputRoot.XMLNSC.*[] = InputRoot.XMLNSC.*[]
```

This example copies the children of the source XMLNSC folder instead of the XMLNSC folder itself, and as such avoids any bitstream copying. For information about bitstream copying, see "Always copy at the parser folder level for the bitstream transfer to take place" on page 40.

## Avoid copying the message tree even when you alter it

Transformation nodes can produce an entirely different message from the original that was on an input terminal. For this reason, InputRoot and OutputRoot references are provided so that the different message trees can be accessed.

Some transformations need to amend only the existing content, and as such they copy the message tree. For example, the following commands mean that the content of the Input message tree is duplicated in the Output tree and as such more memory is used than required:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XMLNSC.TestCase.myField = 'An updated field';
```

In a Computenode, it is possible to update the input message tree and have it propagated. By default InputRoot is not modifiable, and if any attempt is made to update it, the attempt is rejected with an exception. However, a reference can be taken to the input tree, and the input tree can be modified through this reference. The following example shows that them **ComputeMode** property of a Computenode expects to propagate a new message, and therefore expects a populated OutputRoot:

```
DECLARE inputRef REFERENCE TO InputRoot;
SET inputRef.XMLNSC.TestCase.myField = 'An updated field';
```

For this ESQL, the **ComputeMode** property must be changed so that the original input tree is propagated. Although updating the Input trees directly is efficient, care must be taken that this behavior does not affect upstream node processing in the message flow. If a constant input tree is propagating multiple times to a subflow, then the message flow ensures that the input tree is unchanged. This ESQL example means the input tree that was propagated would get modified, and so all other propagations see the updated value.

## Setting a field to NULL is NOT the same as deleting it

Setting a field to NULL is not the same as deleting a field. The following two ESQL examples do have the same result:

1. `SET OutputRoot.XMLNSC.TestCase = NULL;`
2. `DELETE FIELD OutputRoot.XMLNSC.TestCase;`

The first ESQL example detaches the element from a message tree, but it does not configure it for reuse on the next elements parsed. Therefore, when you attempt to improve the memory usage of message tree usage, setting a field to `NULL` should never be used.

## XML, XMLNS, and XMLNSC are not the same domains

Although XML, XMLNS, and XMLNSC are all XML domains, they are not the same domain when you copy a message tree. Therefore, the efficient bitstream transfer does not take place when the tree is copied between these domains with `SET OutputRoot.XMLNSC = InputRoot.XML;` for example.

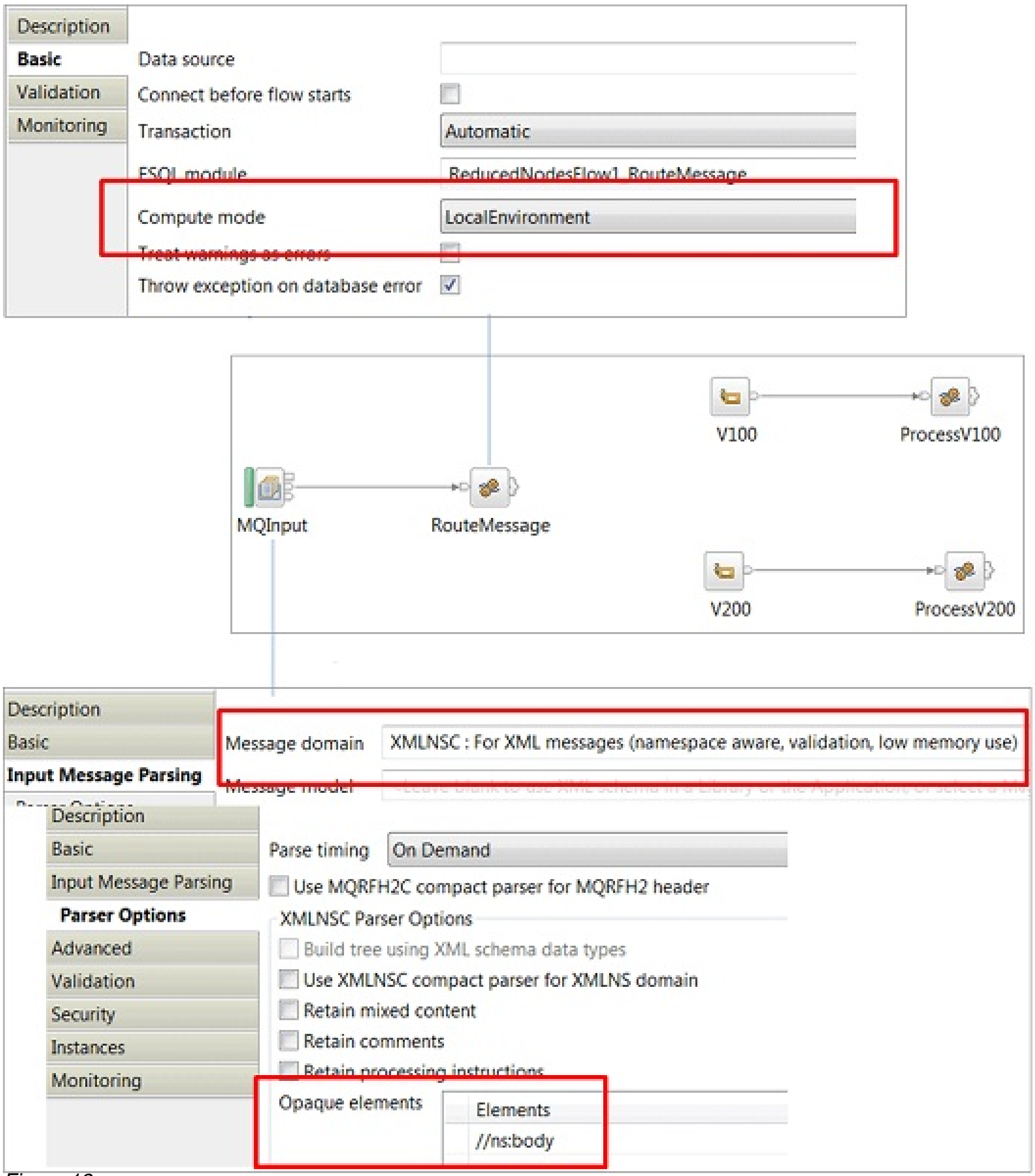
Both XML and XMLNS are deprecated and should be used only for legacy message flows.

## Avoiding tree copying

Sometimes you might need to use a Compute node, but do not need to change the tree. In these situations the Compute node should be configured to use only LocalEnvironment, which avoids a tree copy for the payload.

**Tip:** The payload can still be accessed in the Compute node, but it cannot be modified.

In the following example, a Compute node is used to inspect a version field in the header of a payload, and then propagates to a label with the content of the version:



Data source

Connect before flow starts ☐

Transaction: Automatic

ESQL module: ReducedNodesFlow1_RouteMessage

Compute mode: LocalEnvironment

Treat warnings as errors ☐

Throw exception on database error ☑

Message domain: XMLNSC : For XML messages (namespace aware, validation, low memory use)

Message model: <Leave blank to use XML schema in a Library or the Application, or select a Me

Parse timing: On Demand

☐ Use MQRFH2C compact parser for MQRFH2 header

XMLNSC Parser Options

☐ Build tree using XML schema data types

☐ Use XMLNSC compact parser for XMLNS domain

☐ Retain mixed content

☐ Retain comments

☐ Retain processing instructions

Opaque elements

| Elements |
|---|
| //ns:body |

*Figure 12.*

To further improve performance, the Input node parses only the header portion of the payload, the body is configured for opaque parsing. A full parse of the body is then performed in the subsequent nodes as needed.

The Compute node contains the following code:

```
DECLARE rVersion REFERENCE TO InputRoot.XMLNSC.ns:Inventory.ns:header.ns:version;
PROPAGATE TO LABEL(rVersion);
RETURN FALSE;
```

## Message tree copying summary

This section contained a large amount of information relating to message tree copying, therefore this summary might be useful:

*Table 4.*

| | |
|---|---|
| Don't change a message tree and copy it | If a message tree is ever changed, then the bitstream copy cannot take place, and extra parsing is needed. |
| Always copy at the parser folder level for the bitstream transfer to take place | When a message tree is copied, the high-speed bitstream transfer takes place only at the parser folder level. |
| Parsers must be created when not copying to OutputRoot | Only the OutputRoot correlation name in ESQL has the special property that a parser is created for parser name folders. Not copying to OutputRoot means that the bitstream copy cannot take place, and the whole of the source tree is parsed and then copied to the Environment tree. |
| Avoid overwriting fields when a tree is copied | When a message tree is copied, any existing children of target message tree field are detached first. Detaching fields is not the same as deleting fields, so they are still in scope, but are no longer attached to the tree. |
| Setting a field to NULL is NOT the same as deleting it | When you attempt to improve the memory usage of message tree usage, setting a field to `NULL` should never be used. |
| XML, XMLNS, and XMLNSC are not the same domains | The efficient bitstream transfer does not take place when the tree is copied between these domains. Both XML and XMLNS are deprecated. |

- Don't change a message tree and copy it:
- Always copy at the parser folder level for the bitstream transfer to take place:
- Parsers must be created when not copying to OutputRoot
- Avoid overwriting fields when a tree is copied
- Forcing a copy of the message tree fields when required
- Setting a field to NULL is NOT the same as deleting it
- XML, XMLNS, and XMLNSC are not the same domains

- Avoid tree copying

## Reducing the size of the message tree

When a message flow processes messages, the input bitstream is split up by a parser and allocated to fields in the message trees. The resultant message tree uses far more memory than the bitstream itself. Therefore, when processing large messages, the memory cost is much greater.

As the message tree is stored in UCS-2 (2-byte Unicode), when the input message data is distributed into its separate fields, any text fields double in size for their memory representation. The message tree fields must then have a name, a field type, and other underlying information that is used internally by IBM Integration Bus. When the memory usage to maintain the message tree structure are also added, then it is easy to see that a fully parsed message tree is far larger than the bit-stream itself. This behavior is demonstrated in the following example of a simple fixed-length message model:

```
myParent: minOccurs=1, maxOccurs=unbounded
 - myElement1: STRING: :Length=1
 - myElement2: STRING: :Length=1
 - myElement3: STRING: :Length=1
 - myElement4: STRING: :Length=1
 - myElement5: STRING: :Length=1
```

In a single-byte codepage, an example input message is as follows:

```
ABCDE
```

These 5 bytes produce a body folder message tree that looks like the following example:

```
DFDL
 - myParent
   - myElement1 = A
   - myElement2 = B
   - myElement3 = C
   - myElement4 = D
   - myElement5 = E
```

This body message tree with 7 elements uses 62 characters just to represent the names of these 7 elements. As a minimum, the memory usage is 124 bytes in UCS-2. The 5 bytes of bitstream data is stored in the tree, and doubled to 10 bytes because it is stored as UCS-2 data. So already it is approximately 134 bytes of message tree for just the 5 bytes of data.

The value of each field is stored as a syntax element. Syntax elements do not allocate the exact amount of memory for each field because this behavior is inefficient when syntax elements are reused. Syntax element storage string data has a reserve of 28 bytes, meaning that the name and value pairs are taking up 14 multiples of 28 bytes giving 392 bytes. Each tree field generally has a minimum memory usage of at least 100 bytes (and can be more dependent on domain): This result is another 700 bytes for the 7 fields, and as such the 5 bytes that are parsed now take up 1092 bytes. This tree structure memory usage is unavoidable, and because these resources are reused for each message, they are not directly accessible by a message flow.

However, if the following example used the unbounded nature of the repeating parent structure, and has 1000 repetitions of the 5-byte input message:

```
ABCDEABCDEABCDE .........ABCDE.......
```

The input bitstream data alone would be 5000 bytes. A fully parsed message tree for this input bitstream contains 6001 fields (That is, 6 for each of the 1000 repeating record and 1 for the DFDL root element). Assuming again that there are 100 bytes for each repeating 6 fields, and then 6 * 2 lots of 28 bytes for simple name and value pairs, then that is 936 bytes for each repeating record. So for a 5000-byte input messages the 1000 repeating records would occupy 936000 bytes, which is getting close to 1 MB. These numbers are just to demonstrate the weight of the tree structure in memory, and their exact range and maximum values change between release. However. it is easy to see that for more complex models and for more repetitions, the memory usage becomes much larger.

So how do I reduce the size of the memory tree?

Some simple ways to reduce the size of the memory tree are:

1. Build a smaller message tree where possible. Use compact parsers such as XMLNSC, DFDL, and RFH2C, and use opaque parsing.
2. You can improve performance by reducing the number of times that the message tree is copied, by using the following techniques:
   - Reduce the number of Compute nodes and JavaCompute nodes in a message flow.
   - If possible, set the **Compute Mode** property on the node to not include the message.
   - Copy at an appropriate level in the message tree; for example, copy once rather than for multiple branch nodes.
   - Copy data to the environment.

## Message size versus Message tree

When the size of messages that are being processed is being discussed, then the terms *small* and *large* are often used. It is difficult to quantify these terms in a generic way even to explain where the crossover point is from small to large.

When messages are a couple of kilobytes in a size, then they are classed as small messages. When messages are in the tens of megabytes range, then some users would class them as large. However, these attempts at a classification are just considering the size of the bitstream data that is being processed. Nearly all transports in IBM Integration Bus read the incoming message data into a contiguous memory buffer, and as such a message flow allocates the required storage. But the message tree is larger than just the buffer. For example, if there is a repeating element to the message flow, then the size of the input data can grow over time until it is much larger than was initially planned for.

Some of the file transports such as File and TCP/IP are able to stream data in chunks. For such transports, it is not necessary to read the whole of the input data into memory at the same time. The amount of contiguous storage that is required to read a "chunk" depends on the options that are selected on the input node.

# Chapter 6. Handling large input messages

Although a message flow developer cannot usually change the size of the input data or the model that leads to large memory usage, they can influence how much of a message is actively storied in memory at the same time.

In previous examples, fully parsed message trees were discussed. A standard full parse of a message tree parses a bitstream from start to finish, and fully inflates the message tree in memory. When this process produces a large message tree, large message handling techniques can be implemented to reduce memory usage.

The large message handling techniques presented here work on the principle that any large message tree is large because it has many repeating records. As such, the logic that is implemented by a message flow deals with each record one at a time and undertakes the necessary processing on that record. Because only one record is ever viewed at a time, only that one record needs to be held in memory. This process is achieved by the following sequence:

1. Using partial parsing to parse the bitstream in a message tree.
2. Parse the first repeating record with a reference variable (see "References & navigating the message tree" on page 34 for more information).
3. Move to the next repeating record and delete the parent field of the previous record.

If an input message was large because it contained an embedded video or audio file for example, then this message is a single field, not many fields. Therefore, you cannot use these methods to reduce memory usage. As the whole structure must be populated in the message tree, there is typically less capacity to optimize processing. However, you can still take advantage of the Configuration Recommendations to Reduce Memory Usage Best Practice PDF document, or Configuration Recommendations to Reduce Memory Usage in the main product documentation.

By deleting the parent field of the record after it was processed, the elements that were being used for that record are made available to the parser to use again. This behavior means that when the next repeating record is fully parsed, it reuses the same underlying message tree field objects instead of creating new ones. Using this technique means that only the memory for one record is ever used, irrespective of how many repeating records are in the input message.

Large message handling depends on being able to delete message tree fields, and the following methods are currently supported:

1. For ESQL, use the DELETE FIELD statement.
2. For .NET, use the NbElement delete() method.
3. For Java, use the MbElement delete() method.
4. For C, use the cniDelete() method.

This technique is also dependent on the message tree never being fully parsed by any other aspects of IBM Integration Bus. If at any point all of the repeating records are in memory at the same time, then deleting instances of these records does not reduce memory usage because the largest footprint for message tree was already allocated. The following are examples of IBM Integration Bus function might lead to fully parsing a message tree, and therefore would negate the benefits of large message handling:

- Complete or Immediate parsing on any input node.
- A trace node with ${Root} or ${Body} in it.
- The flow debugger being attached as the content of message trees and is sent to the debugger.
- Serializing a large message tree with a different codepage/encoding or validation options than it was created with.
- Copying the message tree to a different domain such as with SET OutputRoot.XMLNSC = InputRoot.DFDL. This behavior forces both the input and output trees in their entirety to be in memory at the same time

In summary, large message handling techniques can reduce how many message tree fields are in memory at the same time. Deleting fields does not apply only to repeating fields, it can also apply to any message tree field. Because the DELETE FIELD command has an overhead when the field is removed from the tree hierarchy, a message flow developer might not want to do this for every field that is ever referenced.

**Tip:** Large message handling relies on the Input trees not being parsed at all, and being copied to a modifiable output tree for record handling. Care must be taken that this InputRoot tree is never fully parsed, upstream as well as downstream. You must avoid situations where the InputRoot is handled correctly from the Compute node onwards, but then when processing returns to the nodes before the Compute node, the original input tree gets accidentally parsed.

## Large messages: Tricky trailer records

Message data is typically large when it contains many repeating records, and these repeating records might be enclosed by *Header* and *Trailer* records. The header record gives origin or batch information on the batch of records that is being received and the trailer record gives summary information about the batch as whole, such as counts on items or total prices.

In a routing application, a message flow might need to examine the header and trailer records before routing. In augmentation scenarios, a message flow might need to update only the trailer record. To get to the trailer record in each case, all repeating record instances must be parsed. For these types of scenario, different approaches must be considered when you use the XMLNSC or DFDL domains. The aim of the different approaches is to reduce the number of fields that are handled in the message tree.

XMLNSC supports opaque elements. This capability is covered in *Opaque parsing* in the *Parsing strategies* section. That section describes the opaque parsing functionality, which enables XPaths to be entered for elements that are not intended to be "inflated" in the message tree. That is, instead of parsing all the fields for the named record, a single opaque field is added to the message tree. This opaque field contains the portion of the bitstream that would be parsed for this record. If the XPath identifies all the repeating records between a header and trailer record, then no individual fields would be created for each record.

For model-based domains such as DFDL and XMLNSC, the model determines the structure of the message tree that is parsed. When you parse a large message with a repeating record that does not need to be inflated, memory can be saved by constructing an alternative model to represent the sparse message. This alternative model has just a single field for a record instead of all of its detailed fields. The single field is defined to consume the same bitstream content as the detailed version of the record.

The following example is a simple DFDL fixed-length model:

```
myHeader
 - BatchNumber: String: Length=16
myRecord: minOccurs=1, maxOccurs=unbounded
 - Field1 : String: Length=1
 - Field2 : String: Length=1
 - Field3 : String: Length=1
 - Field4 : String: Length=1
 - Field5 : String: Length=1
myTrailer
 - TotalRecord: Int: Length=8
```

In this example, the *myRecord* parent field could repeat millions of times. As such, even with large message handling techniques, the 5 fields of every repeating record must be parsed and then deleted. The following model can be used instead, to reduce memory usage:

```
myHeader
 - BatchNumber: String: Length=16
myRecord: hexBinary: Length=5, minOccurs=1, maxOccurs=unbounded
myTrailer
 - TotalRecord: Int: Length=8
```

In this example, each single *myRecord* is parsed and deleted, but the parser does not have to parse and inflate the complexity of each record. This technique does not only save memory, but also improves performance of a large parse, especially when each repeating record has a complex or large structure itself. However, if that data is needed, then there is no saving in performance.

## Large output messages

So far the recommendations have covered the input message trees and how these trees can be handled and copied efficiently. However, some message flows might need to generate a large output message, especially if a large input message is being augmented and routed.

In the same way that large input trees need to be handled correctly, large output trees can be optimized to reduce memory usage. The following example is a large output tree that requires a large amount of memory:

```
Root
 - Properties
   -
 - MQMD
   - ..
 - XMLNSC
   - TestCase
     - Record
       - Field01 = A
       - Field02 = B
       - Field03 = C
       - Field04 = D
       - Field05 = E
     - Record
       - ...
     - Record
       - ...
```

In this example tree, *Record* repeats 100 000 times. The parent field that is combined with its 5 child fields, means that there are 600 000 output fields for the repeating records. The resulting large output message tree could cause memory issues in the **DataFlowEngine** process.

While most domains can parse large input messages using large message handling techniques, only a few domains are able to store large message trees. The domain needs to be able to serialize using the **FolderBitStream** instruction (that is, serialize a portion at a time).Then, the domains serialization needs to support elements of type **BitStream**.

The following example demonstrates how the ESQL can be changed to build a large message tree. The following ESQL creates over 600 000 fields for the repeating records:

```
CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC') NAME 'XMLNSC';
DECLARE outRef REFERENCE TO OuptutRoot.XMLNSC;
CREATE LASTCHILD OF outRef AS outRef NAME 'TestCase';
DECLARE currentRecord REFERENCE TO outRef
DECLARE recordTotal INT 100000;
DECLARE recordCount INT 0;
WHILE recordCount < recordTotal DO
  CREATE LASTCHILD OF outRef AS currentRecord NAME 'Record';
  SET currentRecord.Field01 = 'A';
  SET currentRecord.Field02 = 'B';
  SET currentRecord.Field03 = 'C';
  SET currentRecord.Field04 = 'D';
  SET currentRecord.Field05 = 'E';
  SET recordCount = recordCount + 1;
END WHILE;
```

When augmenting a large message then the large message handling techniques for parsing and writing large messages may be combined. The next record would be parsed using *partial parsing*. This next record could be updated and then ASBITSTREAM called on it in `FolderBitStream` mode, and inserted into the tree as a **BitStream** field. The current record is then deleted, as shown in the following example:

```
CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC') NAME 'XMLNSC';
DECLARE outRef REFERENCE TO OuptutRoot.XMLNSC;
DECLARE currentRecord REFERENCE TO outRef
CREATE LASTCHILD OF currentRecord AS currentRecord NAME 'Record';
CREATE LASTCHILD OF outRef AS outRef NAME 'TestCase';
DECLARE folderBytes BLOB;
DECLARE recordTotal INT 100000;
DECLARE recordCount INT 0;
WHILE recordCount < recordTotal DO
  SET currentRecord.Field01 = 'A';
  SET currentRecord.Field02 = 'B';
  SET currentRecord.Field03 = 'C';
  SET currentRecord.Field04 = 'D';
  SET currentRecord.Field05 = 'E';
  SET folderBytes = ASBITSTREAM(currentRecord OPTIONS FolderBitStream);
  CREATE LASTCHILD OF outRef TYPE XMLNSC.BitStream NAME 'Record' VALUE folderBytes;
  SET recordCount = recordCount + 1;
  DELETE FIELD currentRecord; -- Free up the temporary record that was used in serialization.
END WHILE;
```

Although a message flow that handles real business data is likely to be more complex than this example, the technique is the same. Use ASBITSTREAM in `FolderBitStream` mode on the next record, insert it into the tree as **BitStream** field and then delete the current record as shown in the following example:

```
--Parse in the Environment message tree to avoid clashing with the OutputRoot.XMLNSC folder.
CREATE FIELD Environment.Variables.XMLNSC DOMAIN('XMLNSC');
SET Environment.Variables.XMLNSC = InputRoot.XMLNSC; -- Assumes we are copying an unparsed
-- XMLNSC folder by its bitstream
DECLARE inputRef REFERENCE TO Environment.Variables.XMLNSC.RepeatingRecord;

--Create the output folders
CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC') NAME 'XMLNSC';
DECLARE outRef REFERENCE TO OuptutRoot.XMLNSC;
CREATE LASTCHILD OF outRef AS outRef NAME 'TestCase';

WHILE LASTMOVE(inputRef) = TRUE DO
  SET inputRef.Field02 = 'Z';
  DECLARE folderBytes BLOB ASBITSTREAM(inputRef OPTIONS FolderBitStream);
  CREATE LASTCHILD OF outRef TYPE XMLNSC.BitStream NAME 'RepeatingRecord' VALUE folderBytes;
  DECLARE previousRecord REFERENCE TO inputRef;
  MOVE inputRef NEXTSIBLING 'RepeatingRecord'; -Move to the next repeating record
  DELETE FIELD previousRecord;  -- delete the record we have dealt with
END WHILE;
DELETE Environmet.Variables.XMLNSC; -- Delete the XMLNSC folder that was being used.
```

Large data is often associated with the File transport. The FileOutput node can append data to a file during the flow processing. Therefore, you can write message flows where records are written to a file individually, and as such large output message trees are not required. The **FolderBitStream** and **BitStream** elements provide a method of constructing a large message tree where multiple records are represented by single records. If a domain does not support these capabilities, but is model-based, then it is possible to write an alternative model where records are represented as single binary elements. If a global element represents the record content, it is possible to avoid the use of the **FolderBitStream** and **BitStream** elements.

## Binary large object (BLOB) processing

If your message flow solutions uses raw BLOB processing in ESQL to build an output message, then use these solutions to concatenate function to join BLOB portions together in `OutputRoot.BLOB.BLOB` for example. This technique can cause excessive memory use due to fragmentation and a large final result. Consider a message flow that reads in a 1 MB BLOB and assigns it to the BLOB domain. For the purposes of the following demonstration, ESQL uses a WHILE loop that causes the repeated concatenation of the 1 MB BLOB to produce a 57 MB output message:

```
DECLARE c, d CHAR;
SET c = CAST(InputRoot.BLOB.BLOB AS CHAR CCSID InputProperties.CodedCharSetId);
SET d = c;
DECLARE i INT 1;
WHILE (i <= 56) DO
   SET c = c || d;
   SET i = i + 1;
END WHILE;
SET OutputRoot.BLOB.BLOB = CAST(c AS BLOB CCSID InputProperties.CodedCharSetId);
```

In this example, the following sequence of events occurs:

1. A 1 MB input message is assigned to a variable *c* and is then also copied to *d*.

2. The loop then concatenates *c* to *d*, and assigns the result back to *c* on iteration.
3. Variable *c* grows by 1 MB on every iteration.

As this processing generates a 57 MB BLOB, you might expect the message flow to use around 130 MB of storage (The ~60 MB of variables in the Compute node, and then 57 MB in the Output BLOB parser, which is serialized on the MQOutput node.)

However, this is not the case. This ESQL causes a significant growth in the integration server's storage usage due to the nature of the processing. This ESQL encourages what is known as *fragmentation* in the memory heap. This condition means that the memory heap has enough free space on the current heap, but has no contiguous blocks that are large enough to satisfy the current request.

When you deal with BLOB or CHAR Scalar variables in ESQL, these values must be held in contiguous buffers in memory. These buffers are continuous blocks of storage, which are large enough to hold the values. Therefore, when the ESQL statement: `SET c = c || d;` is executed, in memory terms it is not just a case of appending the value of *d* to the current memory location of *c*. The concatenation operator takes two operands and then assigns the result to another variable, and in this case the variable is one of the input parameters.

So logically the concatenation operator could be written: `SET c = concatenate(c,d);`

This example is not valid syntax, but is being used to illustrate that this operator is like any other binary operand function.

The value that was originally contained in *c* cannot be deleted until the operation is complete because *c* is used on input. Furthermore, the result of the operation needs to be contained in temporary storage before it can be assigned to *c*. The scenario has ever increasing values that makes it more likely that the current heap does not have enough contiguous free blocks to contain the larger value.

This limitation is because the blocks that are being freed are smaller than the larger values that are being generated.

## Why does the memory usage grow?

This section continues from the previous *fragmentation* example, which is repeated here for reference:

```
DECLARE c, d CHAR;
SET c = CAST(InputRoot.BLOB.BLOB AS CHAR CCSID InputProperties.CodedCharSetId);
SET d = c;
DECLARE i INT 1;
```

```
WHILE (i <= 56) DO
   SET c = c || d;
   SET i = i + 1;
END WHILE;
SET OutputRoot.BLOB.BLOB = CAST(c AS BLOB CCSID InputProperties.CodedCharSetId);
```

So, now consider the possible pattern of allocations that could take place in this scenario. The best possible case is where no other threads are running that might make allocations in the freed blocks. This scenario also assumes that during the execution of ESQL for this thread, no small allocations are made in the free blocks. In a real integration server, these allocations would take place even with one flow running, because the broker has administration threads that run periodically. As this scenario considers only the memory increases, the starting size of the integration server is ignored and only the allocations that are made around the while loop are discussed.

For this example, the following assumptions are made:

- The variables `c` and `d` are 1 MB blocks, and so in the example each character is 1 MB.
- The X character represents a used block.
- The - character represents a free block.

1. First, `c` and `d` occupy a total of 2 MB storage:
   ```
   c d
   X X
   ```
2. Then, `c` and `d` are concatenated together, which requires another 2 MB of storage. This storage must be allocated on the heap to store the result of the concatenation, which gives:
   ```
   c d c d
   X X X X
   ```
3. After the result is assigned to $c_1$, the original `c` 1 MB block is freed:
   ```
   - d c d
   - X X X
       |_|
        c1
   ```
4. The heap grows to 4 MB, with 1 MB free. Now, `d` is concatenated to `c` again, and therefore needs 3 MB because $c_1$ is 2 MB. There is not 3 MB free on the heap, and so the heap must expand by 3 MB to give:
   ```
   - d c d c d d
   - X X X X X X
       |_| |___|
        c1   c2
   ```
5. Now the original $c_1$ is freed, which gives a heap of 7 MB, with 3 MB of free blocks:

```
- d - - c d d
- X - - X X X
        |___|
         c2
```

6. A further concatenation of 3 MB and 1 MB now requires 4 MB for the result, and there is not a contiguous 4 MB block on the heap. Therefore, the heap needs to expand to satisfy this request, giving:

```
- d - - c d d c d c d
- X - - X X X X X X X
        |___| |_____|
         c2     c3
```

7. And the original $c_2$ is freed to give a heap of 11 MB, with 6 MB of free blocks:

```
- d - - - - - - c d c d
- X - - - - - - X X X X
                |_____|
                  c3
```

So even in the unrealistic best case possible scenario, this heap keeps expanding on the basis that the variable cannot be freed during the processing of the current iteration. Therefore, the heap must contain both inputs and targets at the same time. If this pattern is projected to 56 iterations to produce a 57 MB output message, then this behavior causes the integration server using 500 - 600 MB of memory which is much larger than the original estimate.

However, this example was the best case scenario. The worst case scenario is that there is no heap reuse, and so every iteration causes an ever increasing growth. When this growth is projected out, the integration server requires over 1.5 GB of storage. Therefore, it is possible that this scenario causes a situation where the operating system refuses to allocate storage to this process which results in an abend.

As demonstrated, this type of costly BLOB processing must be avoided. The BLOB parser supports multiple BLOB children on output. Therefore, when you manually construct an output message in the BLOB domain, the different portions of the BLOB message must be assigned to multiple BLOB children. The following example demonstrates this behavior with changes the previous *fragmentation* ESQL example. In this example, the memory can build up as the message is processed through the loop with concatenation (`c || d`) leading to fragmentation. The last piece of code avoids fragmentation in this example with no concatenation and `CREATE LASTCHILD OF OutputRoot.BLOB NAME 'BLOB' VALUE c :`

```
DECLARE c, d CHAR;
SET c = CAST(InputRoot.BLOB.BLOB AS CHAR CCSID InputProperties.CodedCharSetId);
DECLARE i INT 1;
```

```
WHILE (i <= 56) DO
   CREATE LASTCHILD OF OutputRoot.BLOB NAME 'BLOB' VALUE c;
   SET i = i + 1;
END WHILE;
```

## The Splitter pattern

A common large message scenario which results in large memory requirements is is to take an input message with a batch of records, and then create individual output messages for each record. The processing loops around the repeating records, and then builds a small output tree that is propagated to the rest of the message flow.

This type of scenario can also be referred to as the *read large -> propagate many* scenario. To support this type of processing, the ESQL language has the PROPAGATE statement. This statement allows a Compute node to propagate the current set of Output message trees to the named terminal (or named label). The following example message flow demonstrates this behavior:

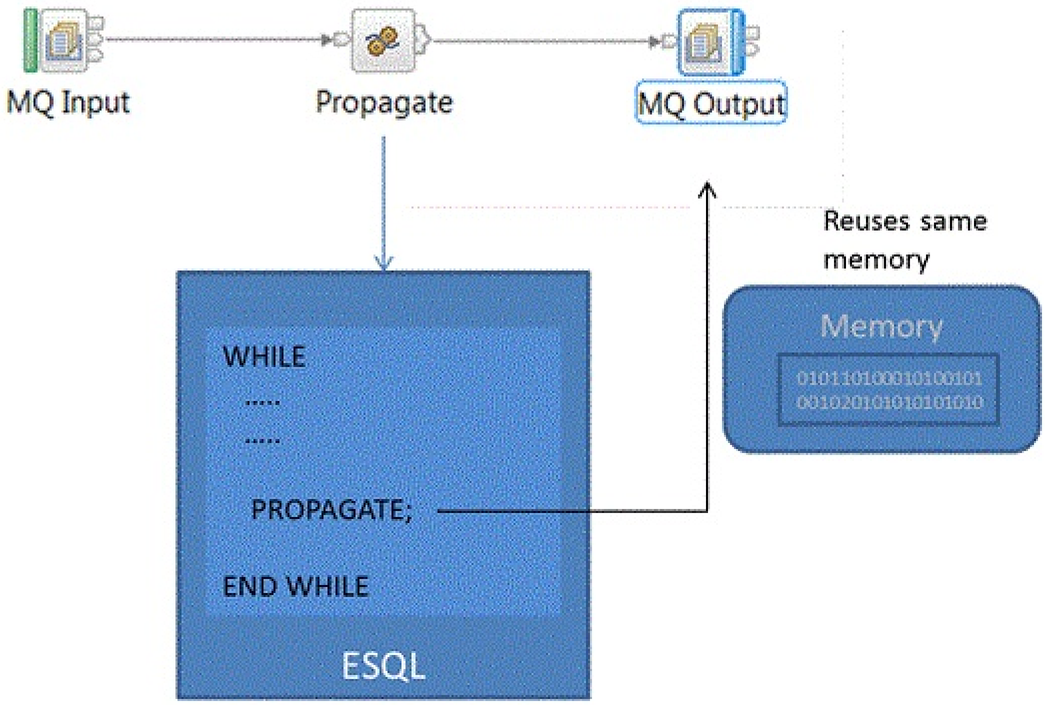


*Figure 13.*

When the down stream nodes are called, the processing returns back to the Compute node. The next line of ESQL is then called after the PROPAGATE statement.

```
DECLARE msgCount INT 0;
WHILE msgCount < 5 DO
  SET OutputRoot.Properties = InputRoot.Properties;
  SET OutputRoot.MQMD = InputRoot.MQMD;
  SET OutputRoot.XMLNSC.TestCase.MsgNumber = msgCount;
  SET msgCount = msgCount + 1;
  PROPAGATE;
END WHILE;
RETURN FALSE;
```

This ESQL propagates five output messages, where each output message contains the message number, and the message headers are populated each time around the loop, as shown in the following example:

```
Message
----- Properties
    *****
----- MQMQ
    *****
----- XMLNSC
        TestCase
            MsgNumber
```

This propagation works because the PROPAGATE statement clears all the Output message trees (`OutputRoot`, `OutputLocalEnvironment`, and `OutputExceptionList`) when the propagation is complete. By clearing all these message trees, any parsers that were created within them can are reset and freed for reuse. Therefore, no matter how many times the PROPAGATE statement is called, the same memory is reused for each output message. If all the output messages are approximately the same size, then such a propagation scenario does not cause any memory growth irrespective of the number of propagations. This process results in the path shown in the following example:

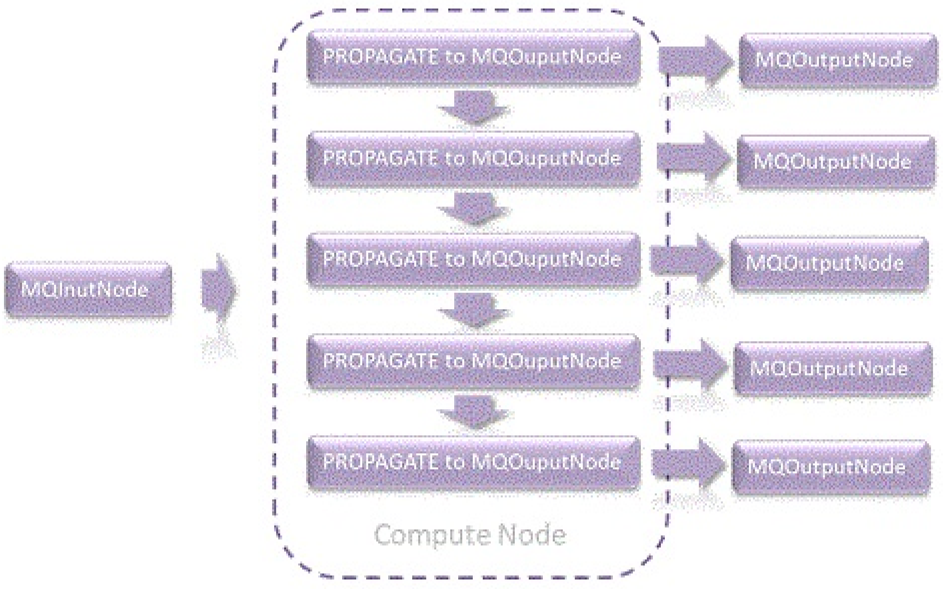


*Figure 14.*

**Tip:** The Environment tree is not cleared by the PROPAGATE call because it is not considered an Output message tree.

What changes were made to the Java API?

The Java API has some similar functionality to ESQL:

1. The clearMessage() method takes a boolean parameter to indicate if the root element is to be deleted as well.
   - If the old clearMessage() method is called, then the default is NOT to delete the root element.
   - By calling clearMessage(true), the API code is giving the broker permission to delete the root element and reset any parsers that are associated with the message.
   - If, after the root element is deleted, elements are still in scope (because they were attached elsewhere) then the owning parser is NOT reset.
   - Any MbElement references to the message trees in the message should not be used after calling clearMessage(true).
2. The MbOuptutTerminal.propagate() method now has a boolean parameter that indicates whether the message objects in the assembly are to be cleared.
   - When `true` is specified, the clearMessage(true) is called on the three message objects in the MbMessageAssembly that was propagated.

- `False` is the default, and gives the same behavior as before where the MbMessageAssembly are not cleared.
- If any of these MbMessage objects are read-only, then clearMessage() is not called on them.
- When MbOutputTerminal.propagate(MbMessageAssembly, true) is called, there is no need to call clearMessage() on the MbMessage objects that were propagated.

Although it might appear that the functionality of #2 supersedes that of #1, it is possible that some implementations build a temporary MbMessage object that is not propagated. By providing a clearMessage(boolean) method, these MbMessage objects can also be cleared completely.

MbOutputTerminal.propagate(MbMessageAssembly, true) has the same behavior of ESQL PROPAGATE, and as such any number of iterations should cause the same amount of parser resources to be used throughout. When you use Java, there might be some JVM heap growth because Java objects are not cleared until a garbage collection cycle.

.NET and the C API do not yet have this support available, which means that a large message splitter scenario in the .NETCompute node or C Plugin node leads to large memory growth. Both the .NETCompute and C Plugin interface support the deleting of elements using MbElement.delete() and cniDelete(). Therefore, the root element in any message can be accessed and a delete that is issued on it.

Now that deletion of fields can free up parser resources for use, deletion can be used as a manual approach that stops memory growth after each propagation. That is, after propagation the root element of each message can be accessed and deleted before cnoClearMessage() or the NbMessage is deleted.

The .NET and C language APIs support multiple propagations, meaning that the .NETCompute, and C Plugin nodes can attempt splitting a large message. In these languages, a new message object can be created on each iteration of the loop, and then the message (NbMessage or CciMessage) is propagated. This process uses the NbOutputTerminal.propagate() or cniPropagate() methods. These propagate methods do not have the same behavior as ESQL PROPAGATE, such that the message trees and parsers are NOT reset for reuse. The reason for the difference is due to the scope and ownership of the objects that are involved. In a Compute node, the Output trees are owned by the broker code and all references to these trees can be managed, and so the trees are cleared. However, for the .NETCompute and plugin nodes the message flow developer's code owns the message objects. Therefore, the propagate methods cannot clear them because the caller might still be either using them or referencing them.

Each API offers a method of clearing the resources that are used by the method:

- In Java, MbMessage has a clearMessage() method. MbMessage needs to have a clearMessage() call on it so that the associated C++ objects and message bitstream are deleted. For example, if your Java node (plugin/JCN) is creating a new output message, then you would have lines like:

  ```
  MbMessage newMsg = null;
  try
  {
    newMsg = createMessage(inputFileBytes);
    MbMessageAssembly outputAssembly = new MbMessageAssembly(assembly, newMsg);
    MbOutputTerminal outTerm = getOutputTerminal("out");
    if (outTerm != null)
    {
       outTerm.propagate(outputAssembly);
    }
  }
  ```

  In this case, if the createMessage() method is called, a buffer is allocated ready to serialize the message tree. As indicated previously, for every createMessage() issued, an associated clearMessage() needs to be called. This needs to be done after the "`outTerm.propagate`" and regardless of whether an exception is thrown or not. For example:

  ```
  MbMessage newMsg = null;
  try
  {
    newMsg = createMessage(inputFileBytes);
    MbMessageAssembly outputAssembly = new MbMessageAssembly(assembly, newMsg);
    MbOutputTerminal outTerm = getOutputTerminal("out");
    if (outTerm != null)
    {
       outTerm.propagate(outputAssembly);
    }
  }
  finally
  {
     if(newMsg != NULL)
     {
       newMsg.clearMessage();
     }
  }
  ```

- In .NET, the NbMessage can be deleted.
- In C, the cniDeleteMessage() method can be called.

Although these methods erase the message object and its associated buffers, they do not erase the parse trees or reset the parsers for reuse. A parser and its parse tree can have wider scope than a single message object because message tree fields could be detached and attached between them.

Therefore, by default a large message splitter pattern in either of the .NETCompute or plugin nodes causes a large amount of memory to be used if multiple propagations take place within a loop. This behavior is due to new parsers that are created for each new message object that is constructed within the loop.

# Chapter 7. ESQL coding practices

Individual sections of this Good operating practice have ESQL samples and recommendations were appropriate. The following recommendations do not fit neatly into a section.

The following recommendations are intended to ensures that flows are optimized for performances. Before the specific recommendations, there is a list of more general coding recommendations.

## ESQL and flow coding practices

1. Initialize variables within DECLARE statements to reduce the number of statements required. The following example uses multiple statements, which is not recommended:

   ```
   DECLARE iIndex INTEGER;      --Declare only
   SET iIndex = 0;       --Initialize
   ```

   Instead of using multiple statements, use a single statement as shown in the following example so that the memory required to process the statements is reduced:

   ```
   DECLARE iIndex INTEGER 0;   --Declare and initialize in one statement
   ```
2. Use a single DECLARE statement when you define multiple variables of the same type. The following example uses multiple statements, which is not recommended:

   ```
   DECLARE cVar1 CHARACTER;
   DECLARE cVar2 CHARACTER;
   DECLARE cVar3 CHARACTER;
   ```

   Instead of using multiple statements, use a single statement as shown in the following example. This action reduces the number of statements to execute, and reduces the memory that is required to define the variables:

   ```
   DECLARE cVar1, cVar2, cVar3  CHARACTER;       --Declare multiple variables
   ```
3. Declare a REFERENCE to avoid excess navigation of the Message Tree. For more information, see the *Using Reference variables* section.
4. Use CREATE with PARSE clause in preference to a read/write operation. For more information, see *Using CREATE with PARSE in the Environment tree* in the *Environment and Local Environment considerations* section.
5. Make code as concise as possible. Restricting the number of statements reduces the memory that the parsing requires.

6. Combine ESQL into the minimum number of compute nodes possible. Fewer compute nodes usually means less tree copying. For more information, see the *Reduce the number of Computenodes* section.

## Environment and Local Environment considerations

Individual sections of this Good operating practice have some information about the Environment and Local Environment, however the following recommendations do not fit neatly into those sections.

Do not use the Environment tree to handle local variables

For most circumstances, using local variables with the node is the optimal solution. This section covers the use of Local Environment to store the variables when more than a flat structure is required.

All data that is stored in the Environment tree is stored in memory during the whole message flow, and is not freed after it is out of scope. Even if you delete the fields from the Environment tree, the memory is not freed until the message flow finishes processing the message. The broker uses pool allocation and reuses resources of the Environment Tree. Therefore, if you need space to store local data for node work, then use the LocalEnvironment tree and avoid high memory consumption.

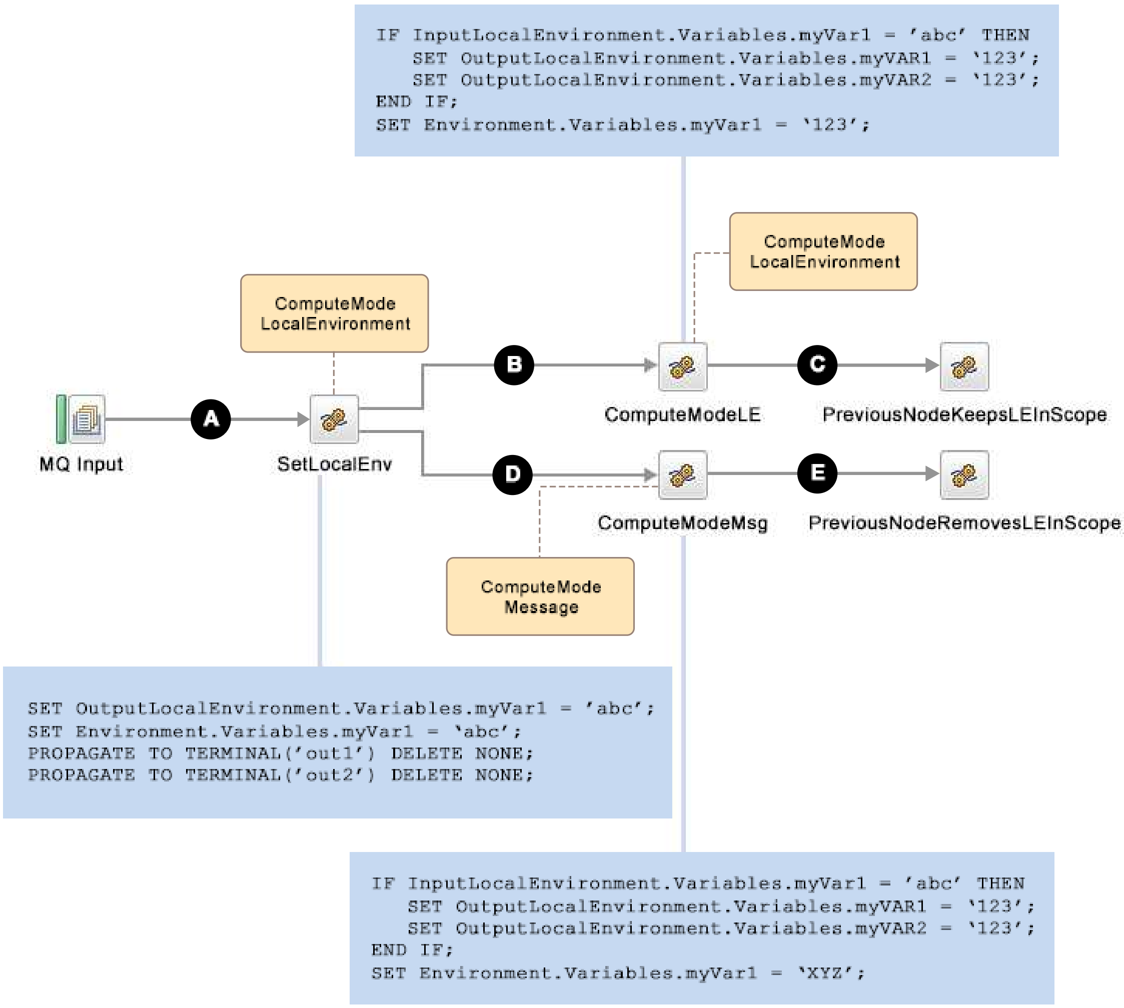
If you create local environment variables, store them in a subtree called *Variables*. This action provides a work area that you can use to pass information between nodes. This subtree is never inspected or modified by any supplied node.

Variables in the local environment can be changed by any subsequent message processing node, and the variables persist until the node that created them goes out of scope.

Generally, use the Environment for scratchpad work where you need to maintain the content beyond the scope of the current node, and use the LocalEnvironment for larger scratchpad areas that are only required during the current node, and you need to release the memory after use.

The scope of Environment and Local Environment

The following flow diagram demonstrates the use of Environment and Local Environment:


ComputeMode LocalEnvironment
ComputeMode LocalEnvironment
MQ Input
A
SetLocalEnv
B
ComputeModeLE
C
PreviousNodeKeepsLEInScope
D
ComputeModeMsg
E
PreviousNodeRemovesLEInScope
ComputeMode Message

```
IF InputLocalEnvironment.Variables.myVar1 = 'abc' THEN
   SET OutputLocalEnvironment.Variables.myVAR1 = '123';
   SET OutputLocalEnvironment.Variables.myVAR2 = '123';
END IF;
SET Environment.Variables.myVar1 = '123';
```

```
SET OutputLocalEnvironment.Variables.myVar1 = 'abc';
SET Environment.Variables.myVar1 = 'abc';
PROPAGATE TO TERMINAL('out1') DELETE NONE;
PROPAGATE TO TERMINAL('out2') DELETE NONE;
```

```
IF InputLocalEnvironment.Variables.myVar1 = 'abc' THEN
   SET OutputLocalEnvironment.Variables.myVAR1 = '123';
   SET OutputLocalEnvironment.Variables.myVAR2 = '123';
END IF;
SET Environment.Variables.myVar1 = 'XYZ';
```

*Figure 15.*

The following table details the contents of the Environment and Local Environment as a message traverses through the flow:

*Table 5.*

| **Stage** | **Environment** | **Local Environment** |
|---|---|---|
| A | Empty | Empty |
| B | myVar1 = 'abc' | myVar1 = 'abc' |
| C | myVar1 = '123' | myVar1 = '123'<br><br>myVar2 = '123' |

*Table 5. (continued)*

| Stage | Environment | Local Environment |
|---|---|---|
| D | myVar1 = ‘123’ | myVar1 = ‘abc’ |
| E | myVar1 = ‘XYZ’ | myVar1 = ‘abc’ |

Environment variables maintain state throughout the flow, and the memory is not released regardless of whether the elements are removed from the environment.

Local Environment variables maintain state only while the flow progresses (along a single propagated path). The Local Environment variables are available to all nodes on the same propagated path, but changes are only passed out of a Computenode if the **ComputeMode** property includes Local Environment. If the **ComputeMode** is set to `Message`, the LocalEnvironment is passed through the node as-is. The Local Environment can still be used within a Computenode (read or written to) when the **ComputeMode** does not include Local Environment but the scope of these additions and modifications survives this node.

Environment with Domain

Copying a message tree (or a portion of it) into the Environment can be useful for many scenarios, but it is essential that the correct domain is created prior to copying the structure. Forgetting to assign the correct domain to the Environment tree before copying the fragment of the input message causes inflation of the tree to occur before the copy happens. For more information, see the *Types of parser copy* section.

Sample with PARSE included:

```
CREATE LASTCHILD OF Environment DOMAIN 'XMLNSC' PARSE(InputRoot.BLOB.BLOB);
```

Sample without PARSE included:

```
CREATE LASTCHILD OF Environment DOMAIN 'XMLNSC';
```

When the source and target trees have the same domain, it is just a bitstream copy.

Using CREATE with PARSE in the Environment tree

When the subject of a parser is discussed in terms of the broker, then it is seen as a single entity such as the XMLNSC parser. However, this situation is not the case, and many instances of a parser can be created based on the logic implemented in a message flow. A parser is responsible for creating a message tree from a bitstream and vice versa.

Since the introduction of the CREATE with PARSE ESQL statement, the term *message tree* could apply to a portion of a message tree. In the following example, an XMLNSC parser is created in the CREATE statement, and another is created in OutputRoot message:

```
CREATE FIELD Environment.Variables;
DECLARE envRef REFERENCE TO Environment.Variables;
DECLARE count INT 0;
WHILE count < 50000 DO
    SET count = count + 1;
    CREATE LASTCHILD OF envRef DOMAIN('XMLNSC')
    PARSE(InputRoot.DFDL.record[count], .....);
    CALL CopyMessageHeaders();
    SET OutputRoot.XMLNSC = Environment.Variables.XMLNSC[count];
    PROPAGATE;
END WHILE;
```

This example assumes that the input DFDL tree has 5000 repeating records that contain XML blobs. The XMLNSC parser created in the Environment tree is owned by the Environment tree's message group. Hence all parsers created in the Environment tree are owned by the Input message group, and are not cleared and reused until processing returns to the input node. However the XMLNSC parser created in the OutputRoot message tree, is owned by a sub message group. This parser is cleared, reset, and all parsers are freed for reuse at the end of the PROPAGATE. For more information, see the *The Splitter pattern* section.

This ESQL leads to 5001 XML parsers being created. Once created, they exist for the life of the integration server, and are reused throughout. Even modifying the ESQL to the following example makes no difference:

```
CREATE FIELD Environment.Variables;
DECLARE envRef REFERENCE TO Environment.Variables;
DECLARE count INT 0;
WHILE count < 50000 DO
    SET count = count + 1;
    CREATE LASTCHILD OF envRef DOMAIN('XMLNSC')
    PARSE(InputRoot.DFDL.record[count], .....);
    CALL CopyMessageHeaders();
    SET OutputRoot.XMLNSC = Environment.Variables.XMLNSC[count];
    DELETE FIELD  Environment.Variables.XML[count];
    PROPAGATE;
END WHILE;
```

The DELETE FIELD statement just deletes all the message tree fields for reuse, but the parsers syntax element pool owns them and gets them ready for reuse. Therefore, the parser must still exist to do this. A parser can be reset and reused when the message tree it is associated with goes out of scope. So when the CREATE with PARSE is used in the OutputLocalEnvironment, this message tree goes out of scope when:

- Either when returning from a PROPAGTAE statement, or

- When the Computenode completes and the processing returns to a previous point in the message flow.

Any parsers that are associated with the Environment tree are not be freed and reused until the message flow completes for that message. However if many parsers are created in the Environment tree for one message, then this behaviour might lead to storage exhaustion before the flow completes.

If such conditions are found to be present in a message flow, then the use of OutputLocalEnvironment should be considered along with the PROPAGATE statement.

**Tip:** Use Environment when the parsed results are required throughout all paths in the flow or need to survive a rollback. But keep in mind the performance implications with loops as described above. Otherwise consider the use of a Local Environment variable where scope can be managed to within a Propagate path.

## Using reference variables

Reference variables have two benefits:

1. They make code more readable.
2. They reduce the overhead of repeatedly traversing the message tree each time a location in the tree structure is required to be accesses. This behavior applies to any message tree such as InputRoot or OutputRoot.

The following example demonstrates Reference declarations:

```
DECLARE rInputAddress REFERENCE TO InputRoot.DFDL.Invoice.Customer.Address;
DECLARE rOutputCustomer REFERENCE TO OutputRoot.DFDL.Invoice.Customer;
```

References can be used within references. So when you need to access multiple levels of a tree structure with code, create a reference for each level. The following example demonstrates multiple reference levels:

```
DECLARE rInputInvoice REFERENCE TO InputRoot.DFDL.Invoice.Customer.Address;
DECLARE rInputCustomer REFERENCE TO rInputInvoice.Customer.Address;
DECLARE rInputAddress REFERENCE TO rInputCustomer.Address;
```

References only maintain the scope of the code block that they are defined in. For example, if you defined a reference within an IF statement, then the Reference is only accessible within that IF block. Therefore, if you need to use the same reference in multiple code blocks, you must define the reference outside of any logical statements. References are often defined at the top of each Function or Procedure to add to the readability of the code.

References can be either static (defined to a fixed location) or dynamic. The definition for both is identical but the pointer within the reference can be moved.

In the following code sample, a reference is created to *Product* which can occur one or more times. The LASTMOVE instruction checks if the reference that is defined has found the tree location (Product of parent Invoice). The WHILE loop iterates through each *Product* found by moving the reference *rProduct* through each sibling element of name *Product*:

```
DECLARE rProduct REFERENCE TO InputRoot.DFDL.Invoice.Product;
WHILE LASTMOVE(rProduct) DO
   ...some logic
   MOVE rProduct NEXTSIBLING NAME 'Product';
END WHILE;
```

## Reduce statements with SELECT

The number of statements used in an ESQL procedure or function impacts the performance of the flow. Where possible, consider using a SELECT statement when a single ESQL statement can be used instead of multiple SET statements.

The following example uses SET statements, and is not recommended:

```
SET rXMLRoot.ota:parentFolder.ota:childFolder. = ota:field1 = rLocInfo.fielda;
SET rXMLRoot.ota:parentFolder.ota:childFolder. = ota:field2 = rLocInfo.fieldb;
SET rXMLRoot.ota:parentFolder.ota:childFolder. = ota:field3 = rLocInfo.fieldc;
```

However, the following example, although obviously very simple, has performance benefits that become substantial when you have more fields to map:

```
SET rXMLRoot.ota:parentFolder.ota:childFolder =
  (SELECT
    rLocInfo.fielda AS ota:field1,
    rLocInfo.fieldb AS ota:field2,
    rLocInfo.fieldc AS ota:field3
   FROM rRQD.LOCATION_INFO AS rLocInfo);
```

SELECT statements can also be embedded within another SELECT statement to allow nesting of the output structure as shown in the following example:

```
SET rSoapEnv =
  (SELECT
    nsSoapenv  AS  (XMLNSC.NamespaceDecl)xmlns:"soap,
    nsXsi  AS  (XMLNSC.NamespaceDecl)xmlns:"xsi",
    nsXsd  AS  (XMLNSC.NamespaceDecl)xmlns:"xsd",
    ''  AS  nsSoapenv:Header,
    (SELECT
        nsRnt  AS  (XMLNSC.NamespaceDecl)xmlns:"rnt",
        fs.up  AS  nsSch:pickup,
        fs.off  AS  nsSch:dropoff,
        cDate  AS  nsSch:"date"
    FROM rRent AS fs)  AS  nsSoapenv:Body.nsSch:Rental
  FROM rRental);
```

To include logic within the SELECT statement, you can add a CASE statement as shown in the following example:

```
SET rData.Original =
  (SELECT
    'XYZ'  AS  SystemID,
    (CASE
        WHEN rDetails.Error = 'A' THEN '001'
        WHEN rDetails.Error = 'B' THEN '002'
        WHEN rDetails.Error = 'Z' THEN '027'
        ELSE 'N/A'
    END)  AS  ErrorCode
  FROM rEnv.Details AS rDetails);
```

## Database interaction

Individual sections of this Good operating practice have ESQL samples and recommendations were appropriate. The following recommendations do not fit neatly into a section.

### The WHERE clause

The WHERE clause expression can use any broker operator or function in any combination. It can refer to table columns, message fields, and any declared variables or constants. However, the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. To be evaluated by the database, it must use only those functions and operators that are supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants within scope. If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each subexpression separately. It then attempts to give the database those subexpressions that it can evaluate, leaving the broker to evaluate the rest. This information is important for or two reasons:

1. Trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some database functions exhibit subtle differences of behavior from the behavior of the broker.

## Host variables

It is essential to use host variables so that dynamic SQL statements can be reused; host variables map a column value to a variable. An SQL PREPARE statement is expensive in terms of memory, so reuse where possible.

The following statement can be used only when *Price* is *100* and *Company* is *IBM*. Therefore, when *Price* or *Company* change, another statement is needed, with another PREPARE statement:

```
PASSTHRU('UPDATE SHAREPRICES AS SP SET Price = 100 WHERE SP.COMPANY = 'IBM'');
```

Recoding the sample allows *Price* and *Company* to change, but still uses same statement as follows:

```
PASSTHRU('UPDATE SHAREPRICES AS SP SET Price = ? WHERE SP.COMPANY = ?', rMessage.Price, rMessage.Company);
```

To see the level of dynamic statement cache activity with DB2, use the following commands:

```
db2 connect to <database name>
db2 get snapshot for database on <database name>
```

To see the contents of the dynamic statement cache, use the following commands:

```
db2 connect to <database name>
db2 get snapshot for dynamic SQL on <database name>
```

## Conditional logic

Avoid nested IF statements. The following example uses nested IF statements, and the structure is not recommended:

```
IF mylogic = myvalue1 THEN
    -- Do something
ELSE
    IF mylogic = myvalue1 THEN
        -- Do something else
    ELSE
        IF mylogic = myvalue1 THEN
            -- Do something else
        ELSE
            -- Do something else
        END IF;
    END IF;
END IF;
```

Use ELSEIF or better still, use CASE with WHEN clauses to provide a quicker drop-out from the conditional logic. The following example uses ELSEIF conditional logic:

```
IF mylogic = myvalue1 THEN
    -- Do something
ELSEIF mylogic = myvalue1 THEN
    -- Do something else
```

```
ELSEIF mylogic = myvalue1 THEN
    -- Do something else
ELSE
    -- Do something else
END IF;
```

The following example uses a CASE statement:

```
CASE mylogic
WHEN myvalue1 THEN
    -- Do something
WHEN myvalue1 THEN
    -- Do something else
WHEN myvalue1 THEN
    -- Do something else
ELSE
    -- Do something else
END CASE;
```

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

## Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

**Important:** Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

## Trademarks

# Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

**To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.**

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Email: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in USA