Internals of IBM Integration Bus

David Coles
IBM Integration Bus Level 3 Technical Lead,
IBM Hursley – dcoles@uk.ibm.com

7th August 2014
16200
Agenda

• Introduction
• Runtime Processes
• Threads
• Memory
• Diagnostic Information
• External Internals
  – Execution Engine / Stack Size
  – Node lifecycle
  – Parsers and the Logical Tree
Introducing IBM Integration Bus

- **IBM’s Strategic Integration Technology**
  - Single engineered product for .NET, Java and fully heterogeneous integration scenarios
  - DataPower continues to evolve as IBM’s integration gateway

- **A Natural Evolution for WebSphere Message Broker users**
  - Significant innovation and evolution of WMB technology base
  - New features for Policy-based WLM, BPM integration, Business rules and .NET

- **Designed to incorporate WebSphere Enterprise Service Bus use cases**
  - Capabilities for WESB are folded in to IBM Integration Bus over time
  - Conversion tools for initial use cases built in to IIB from day one
  - WESB technology remains in market, supported. Migrate to Integration Bus when ready

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Integration Bus Components

Integration Bus Toolkit
Integration Bus Explorer
Command line
Third Party Tools
Web UI
Integration Bus Java API
Integration Bus REST API
Integration Bus Node

// Instantiate an object that describes the connection characteristics to the broker.
new MQBrokerConnectionParameters(brokerHostName, brokerPort, brokerOmq, brokerProxy) = null;

GET /admin/eg/MYEGNAME
From: machine@ibm.com
User-Agent: MyApp/1.0

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Runtime Processes
Processes

- **bipimain**
  - z/OS only. The first process in any IIB address space. APF authorised to set up authorised routines.

- **bipservice**
  - Lightweight and resilient process that starts and monitors the bipbroker process (a.k.a AdminAgent).
  - If the bipbroker process fails, bipservice will restart it.

- **bipbroker**
  - A more substantial process. Contains the deployment manager, for CMP and REST connections, as well as the WebUI server. All commands, toolkit connections and WebUI go through this process.
  - Responsible for starting and monitoring the biphttplistener and DataFlowEngine processes.
  - If either process fails, bipbroker will restart them.

- **biphttplistener**
  - Runs the brokerwide HTTP connector for HTTP and SOAP nodes.

- **DataFlowEngine** (a.k.a Integration Server)
  - Runtime engine for all deployed resources.
Processes by platform

- Linux and Unix systems

- Windows
### Processes grouped by Address Space on z/OS

<table>
<thead>
<tr>
<th>Message Broker Address Spaces</th>
<th>Control Process</th>
<th>Execution group 1</th>
<th>Execution group n</th>
<th>WebSphere MQ</th>
<th>User Process</th>
<th>IMS</th>
<th>OMVS</th>
<th>CICS Region</th>
<th>DB2</th>
<th>RRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>/s nnBRK</td>
<td>Infra-structure main</td>
<td>Infra-structure main</td>
<td>Infra-structure main</td>
<td>WebSphere MQ</td>
<td>User Process</td>
<td>IMS</td>
<td>OMVS</td>
<td>CICS Region</td>
<td>DB2</td>
<td>RRS</td>
</tr>
<tr>
<td></td>
<td>bipservice</td>
<td>DataFlow Engine</td>
<td>DataFlow Engine</td>
<td></td>
<td>USS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bipsbroker</td>
<td>Threads</td>
<td>Threads</td>
<td></td>
<td>wmq/ command</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bihttp listener</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**z/OS**

**LE process**

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Displaying broker at the process level on z/OS

```
SDSF PROCESS DISPLAY MVD1 ALL LINE 1-11 (11)
COMMAND INPUT ===> _ SCROLL ===> CSR
PREFIX=MQ05BRK DEST=(ALL) OWNER=* SORT=ASID/A SYSNAME=*  

<table>
<thead>
<tr>
<th>NP</th>
<th>JOBNAME</th>
<th>JobID</th>
<th>Status</th>
<th>Owner</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQ05BRK</td>
<td>STC52908</td>
<td>FILE SYS KERNEL WAIT</td>
<td>MQ05BRK</td>
<td>BPXBATA8</td>
<td></td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52908</td>
<td>RUNNING</td>
<td>MQ05BRK</td>
<td>biphpplistener</td>
<td>MQ05BRK</td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52908</td>
<td>RUNNING</td>
<td>MQ05BRK</td>
<td>bipservice</td>
<td>MQ05BRK AUTO</td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52908</td>
<td>WAITING FOR CHILD</td>
<td>MQ05BRK</td>
<td>/argoinst/DAVE/usr/lpp/mqsi/</td>
<td></td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52908</td>
<td>RUNNING</td>
<td>MQ05BRK</td>
<td>bpibroker</td>
<td>MQ05BRK</td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52910</td>
<td>RUNNING</td>
<td>MQ05BRK</td>
<td>DataFlowEngine</td>
<td>MQ05BRK 4faaf</td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52910</td>
<td>WAITING FOR CHILD</td>
<td>MQ05BRK</td>
<td>/argoinst/DAVE/usr/lpp/mqsi/</td>
<td></td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52910</td>
<td>FILE SYS KERNEL WAIT</td>
<td>MQ05BRK</td>
<td>BPXBATA8</td>
<td></td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52909</td>
<td>RUNNING</td>
<td>MQ05BRK</td>
<td>DataFlowEngine</td>
<td>MQ05BRK cc32d</td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52909</td>
<td>FILE SYS KERNEL WAIT</td>
<td>MQ05BRK</td>
<td>BPXBATA8</td>
<td></td>
</tr>
<tr>
<td>MQ05BRK</td>
<td>STC52909</td>
<td>WAITING FOR CHILD</td>
<td>MQ05BRK</td>
<td>/argoinst/DAVE/usr/lpp/mqsi/</td>
<td></td>
</tr>
</tbody>
</table>

/u/gormand:
```
psef | grep MQ05
MQ05BRK 84017209 50463369 /argoinst/DAVE/usr/lpp/mqsi/bin/bipimain bipservice MQ05BRK AUTO
MQ05BRK 84017284 84017835 DataFlowEngine MQ05BRK cc32dfb6-3001-0000-0080-a253e63dfe32 DAVE
MQ05BRK 67240352 84017924 biphpplistener MQ05BRK
MQ05BRK 67240502 84017209 bipservice MQ05BRK AUTO
MQ05BRK 84017835 84017546 /argoinst/DAVE/usr/lpp/mqsi/bin/bipimain DataFlowEngine 00071016
DFS 131792 131828 grep MQ05
MQ05BRK 84017924 67240502 bpibroker MQ05BRK

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
IBM Integration Bus supports horizontal scaling. This is achieved by increasing the number of integration servers the service is running in.

Reasons to add additional integration servers:
- Increased throughput
- Operational simplicity
- Workload isolation
- Better H/W utilisation
- Higher Availability
Threads
Integration Flow threads (instances)

- Every flow will have at least 1 thread per input node.
  - On z/OS, these threads are TCBs.
- Held within a pool for each input node.
- Increase threads by adding “additional instances”.
- The integration server will start additional instances on demand, as the amount of incoming work increases, up to the limit specified.
- If they are idle then additional instances are stopped.
- Additional instances can be started immediately when the integration server starts.

- Be aware of adding additional instance at the flow level when there are multiple input nodes in the same flow!
Setting additional instances

- In IIB V9, additional instances can be set by creating a workload management policy in the WebUI.

- Additional instances can also be defined on the input node, as a BAR file override and from IIB Explorer.
Vertical Scaling with additional threads

- IBM Integration Bus supports vertical scaling. This is achieved by increasing additional instances (threads) of the service running within the integration server.
- Reasons to add additional instances:
  - Increased throughput
  - Lower memory requirement
  - Better H/W utilisation
  - Policy management

![Diagram showing vertical scaling with additional threads]
Other thread pools

- There are other thread pools for the HTTP connectors.

- HTTP Connector for embedded listener
  - `mqsireportproperties IB9NODE -e default -o HTTPConnector -a`

- HTTP Connector for broker wide listener
  - `mqsireportproperties IB9NODE -b htplistener -o HTTPConnector -a`
HTTP Connector parameters

- Check the IIB Infocenter for a list of all the HTTP Connector parameters, and their default values.

```
HTTPConnector
  uuid='HTTPConnector'
  address=''
  port='7080'
  maxPostSize=''
  acceptCount=''
  compressableMimeTypes=''
  compression=''
  connectionLinger=''
  connectionTimeout=''
  maxHttpHeaderSize=''
  maxKeepAliveRequests=''
  maxThreads=''
  minSpareThreads=''
  noCompressionUserAgents=''
  restrictedUserAgents=''
  socketBuffer=''
  tcpNoDelay=''
  enableLookups='false'
```

- `maxThreads`

Set the value to the maximum number of threads that can be created by the HTTPConnector.

- Value type - integer
- Initial value - 200

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Memory
The Integration Server runtime comprises both C++ and Java. This means there is a multifaceted approach to checking memory.

- The Integration Server is a process (DataFlowEngine). Its size is limited by available memory and configuration on the operating system.
- There is 1 JVM within each Integration Server. The JVM has preconfigured minimum and maximum HEAP settings.
What are the main consumers of memory?

- Normal (low) consumers of memory include:
  - Deployment artefacts
  - Configuration, such as increasing the min JVM HEAP
  - Message flows
  - Additional instance
  - ...

- What to watch for:
  - Size of input messages (1K or 1GB)
  - How messages are parsed (whole file or records)
  - Message Tree copying (transformation nodes)
  - Custom code (ESQL, Java etc)
  - Caching (Shared Variable, GlobalCache etc)
  - JVM settings
How to check process memory usage

- Use operating system tools to view the DataFlowEngine process size
  - **Windows. Process Explorer**
  - **AIX**
    ```bash
    -bash-3.2$ ps -e -o vsz=,rss=,comm= | grep DataFlowEngine
    190456 190496 DataFlowEngine
    ```
  - **Unix/Linux**
    ```bash
    -bash-3.2$ ps -e -o vsz,rss,cmd | grep DataFlowEngine
    1137476 200208 DataFlowEngine CSIM bba82c62-7344-44b3-9568-1790fda4383b default
    ```
  - **z/OS**
    ```bash
    NJ05BRK  LK1EXPERT BROKER STC63051 NJ05BRK  STCFAST COLIM MVD1 IN F4 55,031 0.00 3.62 0.03 205 08CD
    NJ05BRK  NJ05BRK BROKER STC63051 NJ05BRK  STCUSER NJ05 MVD1 IN EC 30,981 0.00 9.00 0.05 203 0900
    NJ05BRK  DAVE BROKER STC63051 NJ05BRK  STCUSER CD05CASE MVD1 IN EC 56,046 0.00 3.62 0.03 240 00F0
    ```
  - vsz = The size in kilobytes of the core image of the process
  - rss = Indicates the real memory (resident set) size of the process (in 1 KB units)
  - Real = 4k pages
How to check JVM memory usage

1. Check defaults:
   - mqsireportproperties IB9NODE -e default -o ComIbmJVMManager –a
     ```
     jvmMinHeapSize='-1'
     jvmMaxHeapSize='-1'
     ```
   - **jvmMinHeapSize**
     - *Initial value: -1, which represents 33554432 bytes (32MB) with the global cache disabled, or 100663296 (96MB) with the global cache enabled*
   - **jvmMaxHeapSize**
     - *Initial value: -1, which represents 268435456 bytes (256 MB)*

2. Check usage with Resource Statistics

![Resource Statistics Table]

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Diagnostic Information
Diagnostic Information in WMB

- Diagnostic Information
  - Resource Statistics
  - Flow Statistics
  - Activity Log
  - Administration Log
  - System Log
  - Trace
  - Stdout/Stderr
Resource Statistics

- **Graphical Performance Monitor**
  - Reports comprehensive usage according of well known resources
    - Message Flows, Nodes, JVM, HTTP, SOAP/HTTP sockets etc
    - Optionally partitioned by Broker, Execution Group and Message Flow

- **Reporting Mechanisms**
  - Graphically reported through IIB Explorer
    - Sort, filter and chart performance characteristics
    - View CPU, IO and other metrics
    - Log data to file in CSV/Excel readable format for post processing
  - User Configurable Reporting Interval
    - XML report messages consumed by any end user application

- **Examples of Available Resource Report Metrics**
  - **JVM**: Memory used, thread count, heap statistics…
  - **Sockets**: Socket host/port open; bytes sent, bytes received
Flow Statistics

- Using the WebUI in Integration Bus v9:
  - Control statistics at all levels
  - Easily view and compare flows, helping to understand which are processing the most messages or have the highest elapsed time
  - Easily view and compare nodes, helping to understand which have the highest CPU or elapsed times.
  - View all statistics metrics available for each flow
  - View historical flow data
Integration Bus Explorer & Activity Log

- View activity as it happens using explorer
- Filter by resource managers

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Administration Queue / Log

- The tools include a lot of information that is useful to the administrator, for example:
  - **Administration queue**: What operational changes are currently pending
  - **Administration log**: What changes have been recently applied to the broker’s configuration, and by whom?
Types of trace in Integration Bus

Trace is available for separate components which can be formatted to a file using:

- mqsichangetrace to enable trace
- mqsireadlog (BIPRELG) to read trace
- mqsiformatlog (BIPFMLG) to format

Types of trace available:

- User Trace – for you.
- Service Trace – for IBM Support
- Command Trace – for IBM Support
- CVP (Component Verification) Trace – for all

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
User Trace Example

UserTrace tells you exactly what is happening as a message passes through an integration data flow. See which nodes are being called, which parsers are being used, and what ESQL is executed.

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
   --CALL CopyEntireMessage();
   SET OutputRoot.XMLNSC.ID = InputRoot.XMLNSC.Customer.ID;
   RETURN TRUE;
END;
```

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
The Integration Bus runtime writes important operational messages to the system log:

- On z/OS, each JOBLOG includes all messages written by processes within the address space. The SYSLOG includes messages from all IIB JOBLOGs.
- On Windows, these messages are written to the event log.
- On Unix and Linux, these messages are written to the syslog.

STDOUT/STDERR may also be written to:

- On z/OS, each JOBLOG includes any STDOUT/STDERR written by processes within the address space.
- On Windows, the `console.txt` file in `%MQSI_REGISTRY%\components\<node>\<EG UUID>`
- On Unix/Linux, the `stdout` and `stderr` files in `$MQSI_REGISTRY/components/<node>/<EG UUID>`
External Internals
Stack based execution engine

- Useful to understand how flows execute when designing them

Start ‘MQ Input’
  Start ‘Compute’
    Start ‘Mapping’
      Start ‘MQ Output’
      Finish ‘MQ Output’
    Finish ‘Mapping’
  Start ‘Java Compute’
  Finish ‘Java Compute’
Finish ‘Compute’
Finish ‘MQ Input’

- Looping node connections can lead to large stack requirements
- With looping and large flows may need to increase the threadstack size
  - MQSI_THREAD_STACK_SIZE=<sizeInBytes> (Unix/Windows) (default is 1Mb)
- On z/OS the thread stack size is dynamic
  - Default size is 1Mb with 1Mb extents
  - Using the extents can impact performance if you regularly use them
  - Increase the default or extent size
    - _CEE_RUNOPTS=THREADSTACK64(ON,4M,1M)
    - Use RPTSTG(ON) option to get a report of stack sizes which were used
The z/OS broker has a global transaction model exactly as you’d expect. It is possible for nodes to elect to commit outside this transaction. RRS is used for context management & commitment control between the flows resource managers, but only when required.
Notes: Transaction Model

- **Transactional message flows are important**
  - A message flow which transforms and routes data often has a need to be transactional. That is, the message flow must complete either *all or none* of its processing. Remember, from an end-to-end application perspective, the message flow is *part* of the application.

- **Transactional data flows and data nodes.**
  - A message flow can be identified as transactional using the Coordinated Transaction checkbox on a broker assigned message flow. The intention behind this attribute is that all node operations within the message flow can be coordinated under the same, global, transaction. On z/OS, this option is always used for message flows, whether selected or not.
  - A node performs its operations within the envelope of this message flow global transaction, and can elect to be within the global transaction or not. A Transaction Mode checkbox enables this for WMQ and database nodes. Note the visibility (ACID) implications!

- **Resource Recovery Services (RRS) is *NOT* always the transaction coordinator.**
  - As message flows run in essentially a batch type address spaces, RRS is the global transaction coordinator, if required.
  - Execution groups are linked with an MQ RRS stub, so WMQ registers an interest with RRS for commitment control.
  - Specifying the keywords CONNECTTYPE=2, AUTOCOMMIT=0, MULTICONTEXT=0, and MVSATTACHTYPE=RRSAF in the initialization file BIPDSNAO enables global transaction processing.

- **RRS Context**
  - RRS Context is a concept that enables a program to have different roles. It's like one person having many ways of behaving which don't interact with each other. It means that applications can simultaneously be different things to different systems.
  - Broker flows have two contexts. A *native* context is used whenever it wants to perform the role of including node operations under the global transaction. A *private* one has the effect of excluding database node operations from a global transaction.
  - Plug-in nodes are always within the global transaction. A message flow is always in *native* context for these nodes.

- **WebSphere MQ**
  - *Transaction Mode* within a message queuing node governs whether MQPUT and MQGET operations are explicitly performed either inside or outside syncpoint on a per call basis. These nodes therefore always use the native, and never the private, RRS context.

- **Database**
  - For database nodes, *Transaction Mode* determines under which RRS context the transaction will be performed. If the node is within the global transaction, then the native context is used. For a Transaction Mode of *commit*, the private context, so that DB2 and RRS see the operation from a logically different party. These nodes commit (using SQLTransact) their operations as the node is exited.

- **Commitment Control**
  - The global transaction is begun implicitly when a resource manager communicates with RRS. The overall message transaction is committed (or backed out!) control returns to the input node. At COMMIT time, WMQ will pass control to RRS only if required.
  - RRS will call all registered resource managers (WMQ, DB2) in a two phase commit protocol to ensure a global transaction. Recall that nodes which elected for a Transaction Mode of commit had resources updated (and externally visible!) close to their point of issuing. If RRS is not required WMQ will perform the commitment control and delete any RRS interests.

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Node Lifecycle (Mid-flow)

- All mid-flow nodes follow this life cycle
  - IBM written and plugin/compute nodes
- Constructor
  - Called when the node is created as the flow is initialized
- Properties are then set on the node
- onInitialize()
  - Validate configured properties
  - Called either during deployment or on broker startup.
  - If it throws an exception, deployment or startup is rolled back/stopped
    - The broker does not try to start the flow again until the broker is restarted
  - Complete tasks that will always work or always fail
    - If you need to initialize an external connection that might need to be retried, consider doing so on the first message through the flow so that the flow can retry the connection as necessary
- Evaluate
  - Called when execution of a message is required
  - Perform required node processing
  - Exceptions which are thrown and thrown back down the flow to be handled
- OnDelete
  - Called before a node is deleted
  - Use if you want the node to perform cleanup operations, for example closing sockets,
  - Should not throw exceptions
Node Lifecycle (Input)

- All non-connector input nodes follow this basic lifecycle

- Constructor
  - Called when the node is created as the flow is initialized
- Properties are then set on the node
- onInitialize()

- run
  - Called by broker when we want the node to try and read data
  - Node reads data and then propagates the resulting message
  - Additional instances
    - If you want your input node to support additional instances then before propagating the message the node needs to call dispatchThread to try and dispatch another thread to read more data
    - Returns success/failure/timeout depending on result

- onDelete
Parsers

Input Message Bit-stream

Fred Smith, Graphics Card...

Parser converts bit-stream to logical structure

Model

Parser converts logical structure to bit-stream

<order><name>Mr. Smith</name></order>

Output Message Bit-stream

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Message Modeling

John, Smith, Graphics Card, 32, 200, 07/11/09

Here is an example of how a physical data structure could be mapped to a logical tree.

- Notice how multiple physical formats can correspond to the same logical tree. The first physical format is an XML structure that shows our Order message. The second is a comma separated value (CSV) structure of the same. The third comprises a set of fixed length fields in a custom wire format.
- By manipulating the logical tree inside the Message Broker rather than the physical bit-stream, the nodes can be completely unaware of the physical format of the data being manipulated. It also makes it easy to introduce new message formats into the broker.

Applications have and require diverse data formats.

- We all know that XML is the data format that's going to solve every data processing problem that exists! We also know that "XML++", the follow-on compatible meta format that someone in a research laboratory is working on will solve all the problems we don't even know we have today! The fact is that, without wanting to appear cynical, every generation goes through this process. Surely it was the same when COBOL superseded assembler.
- The fact is, that for historic, technical, whimsical, political, geographical, industrial and a whole host of other reasons you probably never even thought of, a hugely diverse range of data formats exist and are used successfully by a myriad of applications every second of every day. It's something that we have to live with and embrace because it isn't going to get any better any time soon.
- The advantage WebSphere Message Broker brings by modelling all these messages is that we can rise above the message format detail; so that whether it’s a tag delimited SWIFT or EDIFACT message, a custom record format closely mapping a C or COBOL data structure, or good old XML, we can talk about messages in a consistent, format independent way. Message Broker can manage this diversity.

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
• The Logical Message Model.

- Reconsider messages and their structure. When we architect messages (no matter what the underlying transport technology), we concern ourselves firstly with the logical structure. For example, a funds transfer message might contain an amount in a particular currency, a transaction date and the relevant account details of the parties involved. These are the important business elements of the message; when discussing the message, we refer to these elements.

- However, when we come to realize the message, we have to choose a specific data format. This may be driven by many factors, but we have to choose one. You may be aware of the advantages of various message formats or have your own personal favourite, or may fancy inventing a new one, but the fact remains that you have to choose a physical *wire format*. So for our transfer message, we might decide to use XML, with its elements, attributes and PCDATA (and a DTD, if we're being really exact), or we might map more closely to a C data structure modelling our message with ints, shorts, chars etc. and worry about *their* various representations(!)

- The Logical message model provided by IBM Integration Bus allows one to describe a message in terms of a tree of elements, each of which has a (possibly user defined) type. At the message tree leaf nodes, the elements have simple types such as strings, integers, decimals, booleans etc. Moreover, elements can have various constraints and qualifiers applied to them that more fully describe them; e.g. elements might be optional, appear in a certain order or only contain certain values.
Message Tree Structure

Root

Properties
- CodedCharSetId
- CreationTime
- Encoding
- Transactional

Headers

Body

Order
- Name
- Item
- Qty
- Price
- Date
- First
- Last
Parser Domains

- The message domain identifies the parser that is used to parse and write instances of the message.
  - Eg: BLOB, XMLNSC, DFDL
- The remaining parts of the message template, message model, message (type), and physical format, are optional, and are used by model-driven parsers such as the DFDL parser.
- 4 Main parser types
  - Root
  - Properties
  - Header
  - Body
- Neither the Root or Properties parsers claim any of the incoming bitstream
The logical tree

- Tree is made up of `SyntaxElement` objects which are logically linked
  - MbElement / NbElement / CciElement
- Each element contains the Name, Namespace and Value which describe that element
- Each element knows what type of element it is (Folder, Name, Value, NameValue, etc)
- Each element knows its family relationship
- Each element knows its parse state (leftComplete/rightComplete)
  - This supports partial/onDemand parsing

```
SyntaxElement:
   Name       (String)
   Namespace  (String)
   Type       (Integer)
   Value      (Value Type)
   leftSibling (Pointer to SyntaxElement)
   firstChild (Pointer to SyntaxElement)
   parent     (Pointer to SyntaxElement)
   lastChild  (Pointer to SyntaxElement)
   rightSibling (Pointer to SyntaxElement)
```
The logical tree - linkage

- The logical tree is made up of `ImbSyntaxElement` objects which are logically linked
The logical tree - navigation

- Navigation of the tree is done using a set of similar methods across all languages.
( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
 (0x03000000:NameValue):MessageSet = " (CHARACTER)
...
 (0x03000000:NameValue):IdentityMappedIssuedBy = " (CHARACTER)
)
 (0x01000000:Name ):MQMD = ( ['MQHMD' : 0xed2bcb0]
 (0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
...
 (0x03000000:NameValue):OriginalLength = -1 (INTEGER)
)
 (0x01000000:Folder):XMLNSC = ( ['xmlnsc' : 0x1d24b30]
 (0x01000000:Folder):order = (
...
)
)
)
Trace node output – Root element/Parser

[MQROOT' : 0xed8efe0]
(0x01000000:Name ) : Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
(0x03000000:NameValue):MessageSet = " (CHARACTER)
...
(0x03000000:NameValue):IdentityMappedIssuedBy = " (CHARACTER)
)
(0x01000000:Name ) : MQMD = ( ['MQHMD' : 0xed2bcb0]
(0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
...
(0x03000000:NameValue):OriginalLength = -1 (INTEGER)
)
(0x01000000:Folder):XMLNSC = ( ['xmlnsc' : 0x1d24b30]
(0x01000000:Folder):order = ( ...

)...
Trace node output - Parsers

( ["MQROOT" : 0xed8efe0]
 (0x01000000:Name ) : Properties = ( ["MQPROPERTYPARSER" : 0xed2b2c0]
 (0x03000000:NameValue) : MessageSet = " (CHARACTER)
 ... 
 (0x03000000:NameValue) : IdentityMappedIssuedBy = " (CHARACTER)
 )
 (0x01000000:Name ) : MQMD = ( ["MQHMD" : 0xed2bcb0]
 (0x03000000:NameValue) : SourceQueue = 'VFE.IN1' (CHARACTER)
 ... 
 (0x03000000:NameValue) : OriginalLength = -1 (INTEGER)
 )
 (0x01000000:Folder) : XMLNSC = ( ["xmlncs" : 0x1d24b30]
 (0x01000000:Folder) : order = ( 
 ... 
 )
 )
)
Trace node output – Element Type

( ['MQROOT' : 0xed8efe0]
  (0x01000000:Name  ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
    (0x03000000:NameValue):MessageSet = " (CHARACTER)
    ...
    (0x03000000:NameValue):IdentityMappedIssuedBy = " (CHARACTER)
  )
  (0x01000000:Name  ):MQMD = ( ['MQHMD' : 0xed2bcb0]
    (0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
    ...
    (0x03000000:NameValue):OriginalLength = -1 (INTEGER)
  )
  (0x01000000:Folder):XMLNSC = ( ['xmlnsc' : 0x1d24b30]
    (0x01000000:Folder):order = ( ...
    )
  )
)
Trace node output – Element Name

( ['MQROOT' : 0xed8efe0]
 (0x01000000:Name ):Properties = ( ['MQPROPERTYPARSER' : 0xed2b2c0]
 (0x03000000:NameValue):MessageSet = " (CHARACTER)
 ... 
 (0x03000000:NameValue):IdentityMappedIssuedBy = " (CHARACTER)
 )
 (0x01000000:Name ):MQMD = ( ['MQHMD' : 0xed2bcb0]
 (0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
 ... 
 (0x03000000:NameValue):OriginalLength = -1 (INTEGER)
 )
 (0x01000000:Folder):XMLNSC = ( ['xmlnsc' : 0x1d24b30]
 (0x01000000:Folder):order = ( 
 ... 
 )
 )
 )
Trace node output – Element Value

( 'MQROOT' : 0xed8efe0
  (0x01000000:Name ):Properties = ( 'MQPROPERTYPARSER' : 0xed2b2c0]
    (0x03000000:NameValue):MessageSet = "" (CHARACTER)
    ...
    (0x03000000:NameValue):IdentityMappedIssuedBy = "" (CHARACTER)
  )
  (0x01000000:Name ):MQMD = ( 'MQHMD' : 0xed2bcdb0]
    (0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
    ...
    (0x03000000:NameValue):OriginalLength = -1 (INTEGER)
  )
  (0x01000000:Folder):XMLNSC = ( 'xmlnssc' : 0x1d24b30]
    (0x01000000:Folder):order = ( ...
    )
  )
)
Trace node output – Element Value Type

( [MQROOT' : 0xed8efe0]
  (0x01000000:Name ):Properties = ( [MQPROPERTYPARSER' : 0xed2b2c0]
    (0x03000000:NameValue):MessageSet = " (CHARACTER)
      ...
    (0x03000000:NameValue):IdentityMappedIssuedBy = " (CHARACTER)
  )
  (0x01000000:Name ):MQMD = ( [MQHMD' : 0xed2bcb0]
    (0x03000000:NameValue):SourceQueue = 'VFE.IN1' (CHARACTER)
      ...
    (0x03000000:NameValue):OriginalLength = -1 (INTEGER)
  )
  (0x01000000:Folder):XMLNSC = ( [xmlnsC' : 0x1d24b30]
    (0x01000000:Folder):order = ( ...
      )
    )
  )
)
Building the logical tree – aka parsing!

- For XML it is easy to visually see how we get from the input message to the logical model
- But how?
- The parser fires events back to a handler and the handler then creates the tree
  - Java SAX like parsing
  - DFDL parser follows same model
  - This model allows of OnDemand/partial parsing
    - Reduced memory as you do not always require all of the logical tree in memory

```xml
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```
Building the logical tree – aka parsing!

Events:
- startDocument
  - startElement (Order)

<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

• startDocument
  – startElement (Order)
    • startElement (Name)

```
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```
Building the logical tree – aka parsing!

Events:
- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)

```xml
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```
Building the logical tree – aka parsing!

Events:

• startDocument

  • startElement (Order)

  • startElement (Name)

    • startElement (First)

      » elementValue (John)

    • endElement

<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
  - startElement (Name)
    - startElement (First)
      » elementValue(John)
    - endElement
    - startElement (Last)

```
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```
Building the logical tree – aka parsing!

Events:
- startDocument
- startElement (Order)
  - startElement (Name)
    - startElement (First)
      » elementValue(John) (String)
    - endElement
  - startElement (Last)
    » elementValue(Smith) (String)
    - endElement

<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
<Item>Graphics Card</Item>
<Qty>32</Qty>
<Price>200</Price>
<Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

- startDocument
  - startElement (Order)
    - startElement (Name)
      - startElement (First)
        » elementValue(John) (String)
      - endElement
    - startElement (Last)
      » elementValue(Smith) (String)
    - endElement
  - endElement

<Order>
<Name>
  <First>John</First>
  <Last>Smith</Last>
</Name>
<Item>Graphics Card</Item>
<Qty>32</Qty>
<Price>200</Price>
<Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:
• startDocument
  • startElement (Order)
    • startElement (Name)
      • startElement (First)
        » elementValue(John) (String)
      • endElement
    • startElement (Last)
      » elementValue(Smith) (String)
  • endElement
• endElement
• startElement (Item)

<Order>
 <Name>
  <First>John</First>
  <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

• startElement (Name)
  • startElement (First)
    » elementValue(John) (String)
  • endElement
  • startElement (Last)
    » elementValue(Smith) (String)
  • endElement
• endElement
• startElement (Item)
  • elementValue(Graphics Card) (String)
• endElement

<Order>
 <Name>
   <First>John</First>
   <Last>Smith</Last>
 </Name>
 <Item>Graphics Card</Item>
 <Qty>32</Qty>
 <Price>200</Price>
 <Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

... Repeat 3 more times for Qty, Price and Date ...

• startElement (Qty)
  – elementValue(32) (Integer)
• endElement
• startElement (Price)
  – elementValue(Smith) (Decimal)
• endElement
• startElement (Date)
  – elementValue (07/11/09) (Date)
• endElement

<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
Building the logical tree – aka parsing!

Events:

- startElement (Date)
  - elementValue (07/11/09) (Date)
- endElement
- endElement
- endDocument

```
<Order>
  <Name>
    <First>John</First>
    <Last>Smith</Last>
  </Name>
  <Item>Graphics Card</Item>
  <Qty>32</Qty>
  <Price>200</Price>
  <Date>07/11/09</Date>
</Order>
```
Parsers – Key Methods

Input Message Bit-stream

Model

Parser converts logical structure to bit-stream

refreshBitstreamFromElements
toBitstream

Model

Parser converts bit-stream to logical structure

refreshElementsFromBitstream
createElementAsLastChildFromBitstream

logical structure

<order><name>M r . S m i t h</name></order>

Output Message Bit-stream

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Parsers – Key Methods

- Applies to all parsers, IBM written and plugin parsers
- refreshElementsFromBitstream
  - Turns the bitstream into the logical tree
  - Takes various options
    - Encoding and ccsid to use for the parse
    - Message Set, Type & Format to use for the parse
    - Parser options to apply to the parse, such as validation
    - The parser remembers the options it was initiated with
  - Can be driven from createElementAsLastChildFromBitstream calls in transformation

- refreshBitstreamFromElements
  - Turns the logical tree into a bitstream
  - Takes various options that match those on the refreshElementsFromBitstream call
  - Can be driven from toBitstream calls in transformation

- There is an optimization present in the parsers so that if a parser is called with exactly the same options on the refreshBitstreamFromElements call as it initialized with on the refreshElementsFromBitstream call, and the logical tree hasn’t been modified then the input bitstream is returned
  - This helps support simple pass-through and routing scenarios
Parser Internals – reuse example

1. MQInput Node
2. MQInput Node Compute Node
3. MQInput Node Compute Node Mapping Node
4. MQInput Node Compute Node JavaCompute Node
5. MQInput Node Compute Node
6. MQInput Node Compute Node
7. MQInput Node Compute Node
8. MQInput Node Compute Node JavaCompute Node
9. MQInput Node Compute Node
10. MQInput Node

Notes:
- 1: 7 Parsers are available in the flow pool
- 2: The Compute node creates a message using 2 parsers
- 3: The Mapping node creates a message using 2 parsers
- 4: The MQOutput node creates a message using 1 parser
- 5-6: As the stack unwinds parsers are returned for reuse
- 7: The Compute node creates a new message using 1 of the previously used parsers
- 8: Java Compute node creates a new message using 2 previously used parsers
- 9-10: As the stack unwinds parsers are returned for reuse
Parser Internals - reuse

- Each Message Flow Instance has its own pool of parsers
- Nodes in a flow ‘borrow’ parsers from the flow pool
- Message’s created in nodes use the node ‘borrowed’ parsers
- If a Message requires a new parser it asks the Node for one who in turn ask the flow pool to borrow one
- If the flow pool does not have a free parser it creates a new one
- When a node goes off of the stack the parsers are returned to the flow pool for reuse.
- Only in this instance are parsers returned to the pool for reuse mid flow – example next slide
- At the end of processing a message the flow pool is reset meaning all parsers are reset and NOT DELETED
- This means their memory is still in use
- The parsers are only deleted when the flow is stopped or undeployed
Each Message Flow Instance has its own pool of parsers
Nodes in a flow ‘borrow’ parsers from the flow pool
Message’s created in nodes use the node ‘borrowed’ parsers
If a Message requires a new parser it asks the Node for one who in turn ask the flow pool to borrow one
If the flow pool does not have a free parser it creates a new one
When a node goes off of the stack the parsers are returned to the flow pool for reuse.
Only in this instance are parsers returned to the pool for reuse mid flow – example next slide
At the end of processing a message the Master ImbMessageGroup is reset and ALL PARSERS are DELETED and not reset
This means their memory is returned to the OS
Parser Internals - Element Pools

- Each Parser has a pool of elements
- As a parser is reused the elements in the pool are reused
- The elements owned by a parser are kept until the parser is deleted
- When a parser is reused it may not be used for the same purpose as last time
  - i.e., one use maybe on the input message and another time for the output message
- Each reuse may require a different number of elements
- Over time the element pool for each parser of the same domain will grow until it is the maximum size required
  - This is the plateau’ing effect we sometimes describe in PMRs
**Parser resource statistics**

- Use the Parsers statistics to see how much resource is being used by the message trees and bit streams that these parsers own.

<table>
<thead>
<tr>
<th>Measurements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>The number of message flow threads that contributed to the statistics for a message flows parser type accumulation.</td>
</tr>
<tr>
<td>ApproxMemKB</td>
<td>The approximate amount of user data-related memory used for the named message flow parser type. It is not possible to calculate the exact amount of memory used by a parser.</td>
</tr>
<tr>
<td>MaxReadKB</td>
<td>Shows the largest bit stream parsed by the parser type for the named message flow.</td>
</tr>
<tr>
<td>MaxWrittenKB</td>
<td>Shows the largest bit stream written by the parser type for the named message flow.</td>
</tr>
<tr>
<td>Fields</td>
<td>Shows the number of message fields associated with the named message flow parser type. These fields are retained by the parser and are used for constructing the message trees.</td>
</tr>
<tr>
<td>Reads</td>
<td>The number of successful parses that were completed by the named message flow parser type.</td>
</tr>
<tr>
<td>FailedReads</td>
<td>The number of failed parses that occurred for the named message flow parser type.</td>
</tr>
<tr>
<td>Writes</td>
<td>The number of successful writes that were completed by the named message flow parser type.</td>
</tr>
<tr>
<td>FailedWrites</td>
<td>The number of failed writes that occurred for the named message flow parser type.</td>
</tr>
</tbody>
</table>
## Parser Resource Statistics - Interpretation

<table>
<thead>
<tr>
<th>name</th>
<th>Threads</th>
<th>ApproxMemKB</th>
<th>MaxReadKB</th>
<th>MaxWrittenKB</th>
<th>Reads</th>
<th>Failed Reads</th>
<th>Writes</th>
<th>Failed Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>summary</td>
<td>1</td>
<td>119.77</td>
<td>0.53</td>
<td>0.53</td>
<td>159</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>VFE_1_Flow:MQMD</td>
<td>1</td>
<td>15.97</td>
<td>0.36</td>
<td>0.43</td>
<td>62</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VFE_1_Flow:MQROOT</td>
<td>1</td>
<td>55.89</td>
<td>0.53</td>
<td>0.00</td>
<td>72</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VFE_1_Flow:Properties</td>
<td>1</td>
<td>31.94</td>
<td>0.53</td>
<td>0.00</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[Deleted]</td>
<td>0</td>
<td>0.06</td>
<td>3.97</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[Administration]</td>
<td>3</td>
<td>223.56</td>
<td>3.97</td>
<td>0.00</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### XMLNSC

1. `XMLNSC = ( ['xmlnsc' : 0x342ba2f8]` (0x01000000:Folder):order = ( 2. `order = ( 3. `name = ( 4. `first = 'John' (CHARACTER) 5. `last = 'Smith' (CHARACTER) 6. `item = 'Graphics Card' (CHARACTER) 7. `quantity = '32' (CHARACTER) 8. `price = '200' (CHARACTER) 9. `date = '07/11/09' (CHARACTER) ) ) ) ) )

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval
Parser Resource Statistics - Interpretation

- Using our earlier example flow
- ESQL changed from this:
  ```
  SET OutputRoot = InputRoot;
  ```
- To:
  ```
  CALL CopyMessageHeaders();
  SET OutputRoot.XMLNSC.order = InputRoot.XMLNSC.order;
  ```
- Previously a parser-to-parser copy took place and because no modifications have taken place a new set of parsers were initialised with the original bitstream hence a read of 2 as a new parse takes place
- With the new code a tree copy is taking place having navigated both sides to the order element – now only 1 read taking place
- Parser stats can be very powerful when analysing changes to message flows with high parser memory usage
Partial parsing

- Partial parsing is where only part of the message is parsed at a time, and only if required.
- Utilised correctly this can reduce memory usage and increase performance.
- Default parse mode is “on Demand” which means only parse as far as you need to, to satisfy the current request.
- In our example message if we only needed to read the ‘name’ element to route the message and didn’t need to make a modification then as long as we only referenced as far as the name element then we wouldn’t need to parse all of the message.

- With our example flow
  - If we disable the trace nodes then we will see the field count in the parser stats reduce as we haven’t needed to parse all of the message.
  - Thus partial parsing has been utilised.

<order>
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <item>Graphics Card</item>
  <quantity>32</quantity>
  <price>200</price>
  <date>07/11/09</date>
</order>
Parser Resource Statistics - Interpretation

- Use Parser statistics to understand memory costs associated with processing messages.
  - This is a simple XML file, parsed and serialised using **XMLNSC**.
  - The actual size of the file is **118** bytes (matches **0.12KB** reported).
  - Not all fields have been parsed, as the flow has parse on demand.

```xml
<Customer>
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <ID>1234567890123456789</ID>
</Customer>
```

```
CREATE FUNCTION Main() RETURNS BOOLEAN BEGIN
  CALL CopyEntireMessage();
  RETURN TRUE;
END;
```

<table>
<thead>
<tr>
<th>DotNet App Domains</th>
<th>CICS</th>
<th>DotNet GC</th>
<th>CORBA</th>
<th>ConnectDirect</th>
<th>DecisionServices</th>
<th>FTEAgent</th>
<th>FTP</th>
<th>File</th>
<th>GlobalCache</th>
<th>JDBCConnectionPools</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionGroup</td>
<td>name</td>
<td>Threads</td>
<td>ApproxMemKB</td>
<td>MaxReadKB</td>
<td>MaxWrittenKB</td>
<td>Files</td>
<td>Reads</td>
<td>FailedReads</td>
<td>Writes</td>
<td>FailedWrites</td>
</tr>
<tr>
<td>default</td>
<td>summary</td>
<td>1</td>
<td>111.78</td>
<td>0.47</td>
<td>0.47</td>
<td>17</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>ParserStats.MQMD</td>
<td>1</td>
<td>15.97</td>
<td>0.36</td>
<td>0.43</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>ParserStats.MQROOT</td>
<td>1</td>
<td>55.89</td>
<td>0.47</td>
<td>0.00</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>ParserStats.Properties</td>
<td>1</td>
<td>23.95</td>
<td>0.00</td>
<td>0.00</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>ParserStats.XMLNSC</td>
<td>1</td>
<td>15.97</td>
<td><strong>0.12</strong></td>
<td><strong>0.47</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>[Deleted]</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>default</td>
<td>[Administration]</td>
<td>2</td>
<td>63.88</td>
<td>0.52</td>
<td>0.00</td>
<td>73</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Complete your session evaluations online at [www.SHARE.org/Pittsburgh-Eval](http://www.SHARE.org/Pittsburgh-Eval)
More Parser Resource Statistics - Interpretation

- The same input message as before, but different ESQL. This time the ESQL statement refers to the last element (Customer.ID).
- The parser statistics show how many more fields are read.

```sql
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    --CALL CopyEnt1;
    SET OutputRoot.XMLNSC.ID = InputRoot.XMLNSC.Customer.ID;
    RETURN TRUE;
END;
```

---

<table>
<thead>
<tr>
<th>DotNet App Domains</th>
<th>CICS</th>
<th>DotNet GC</th>
<th>CORBA</th>
<th>ConnectDirect</th>
<th>DecisionServices</th>
<th>FTEAgent</th>
<th>FTP</th>
<th>File</th>
<th>GlobalCache</th>
<th>JDBCConnectionPools</th>
<th>JMS</th>
<th>JVM</th>
<th>ODBC</th>
<th>Parsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Threads</td>
<td>ApproxMemKB</td>
<td>MaxReadKB</td>
<td>MaxWrittenKB</td>
<td>Fields</td>
<td>Reads</td>
<td>FailedReads</td>
<td>Writes</td>
<td>FailedWrites</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>summary</td>
<td>1</td>
<td>95.81</td>
<td>0.47</td>
<td>0.03</td>
<td>24</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParserStats.MQMD</td>
<td>1</td>
<td>7.98</td>
<td>0.36</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParserStats.MQROOT</td>
<td>1</td>
<td>55.89</td>
<td>0.47</td>
<td>0.00</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParserStats.Properties</td>
<td>1</td>
<td>15.07</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParserStats.XMLNSC</td>
<td>1</td>
<td>15.07</td>
<td>0.12</td>
<td>0.03</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Deleted] P4</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'Administration'</td>
<td>2</td>
<td>71.85</td>
<td>4.88</td>
<td>0.00</td>
<td>144</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
and finally ... Typed trees

- When parsing is performed using a schema or message set the element types can be set to be appropriate types as defined in the model
  - MRM / DFDL - automatically typed based on the model
  - XMLNSC – requires content + validation enabled and “Build tree using XMLNSC schema data types” selected

Before:

```
(0x01000000:Folder):XMLNSC = ( [xmlnsc : 0x1cd2360]
(0x01000000:Folder):order = (  
(0x01000000:Folder ):name   = (  
(0x03000000:PCDataField):first = 'John' (CHARACTER)
(0x03000000:PCDataField):last  = 'Smith' (CHARACTER)
  )
(0x03000000:PCDataField):item     = 'Graphics Card' (CHARACTER)
(0x03000000:PCDataField):quantity = '32' (CHARACTER)
(0x03000000:PCDataField):price    = '200' (CHARACTER)
(0x03000000:PCDataField):date     = '07/11/09' (CHARACTER)
  )
)
```

After:

```
(0x01000000:Folder):XMLNSC = ( [xmlnsc : 0x3441b838]
(0x01000000:Folder):order = (  
(0x01000000:Folder ):name   = (  
(0x03000000:PCDataField):first = 'John' (CHARACTER)
(0x03000000:PCDataField):last  = 'Smith' (CHARACTER)
  )
(0x03000000:PCDataField):item     = 'Graphics Card' (CHARACTER)
(0x03000000:PCDataField):quantity = 32 (DECIMAL)
(0x03000000:PCDataField):price    = 200 (DECIMAL)
(0x03000000:PCDataField):date     = DATE '2007-11-09' (DATE)
  )
)```
### Introduction to MQ

MQ Clustering - The Basics, Advances and What’s New in v8

MQ for z/OS v8 new features deep dive

First Steps with IBM Integration Bus: Application Integration for a new world

---

<table>
<thead>
<tr>
<th>Time</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td></td>
<td>Application programming with MQ verbs</td>
<td>The Dark Side of Monitoring MQ - SMF 115 and 116 Record Reading and Interpretation</td>
<td>CICS and MQ - Workloads Unbalanced!</td>
<td></td>
</tr>
<tr>
<td>10:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:15</td>
<td>Introduction to MQ</td>
<td>What's New in IBM Integration Bus &amp; WebSphere Message Broker</td>
<td>MQ – Take Your Pick Lab</td>
<td>Using IBM WebSphere Application Server and IBM WebSphere MQ Together</td>
<td></td>
</tr>
<tr>
<td>12:15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01:30</td>
<td></td>
<td>All about the new MQ v8</td>
<td>MQ Security: New v8 features deep dive</td>
<td>New MQ Chinit monitoring via SMF</td>
<td></td>
</tr>
<tr>
<td>03:00</td>
<td>MQ Beyond the Basics</td>
<td>MQ &amp; DB2 – MQ Verbs in DB2 &amp; InfoSphere Data Replication (Q Replication) Performance</td>
<td>What's wrong with MQ?</td>
<td>IIIB - Internals of IBM Integration Bus</td>
<td></td>
</tr>
<tr>
<td>04:15</td>
<td>First Steps with IBM Integration Bus: Application Integration for a new world</td>
<td>MQ for z/OS v8 new features deep dive</td>
<td>MQ Clustering - The Basics, Advances and What’s New in v8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions?

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval