Reading and Writing Large XML Documents in COBOL

This whitepaper contain three main topics:

1. Challenges of Working with Large XML Documents in COBOL (jump to topic)
2. Reading or Writing a Large XML Document in Streams (jump to topic)
3. Implementing XML Streaming in COBOL using XML Thunder (jump to topic)

Overview

XML documents can be very large. Those adhering to the schemas defined by organizations such as SWIFT and ACORD can contain thousands of different nodes. Others, such as those created for batch processing can contain thousands of occurrences of a given repeating structure (e.g. a list of orders for the month). It is not uncommon for an XML document to exceed 25 megabytes in size.

This white paper identifies some of the challenges COBOL applications face when working with large XML documents and discusses how these challenges can be addressed using a technique called “streaming”. The paper goes on to show how XML Thunder from Canam Software Labs, implements streaming to support reading and writing large XML documents in COBOL.

Challenges of Working with Large XML Documents in COBOL

Working with large XML documents in your application programs can present several challenges. This is especially true for programs written with procedural languages that require pre-defined, static variables – such as COBOL. These challenges include:

- Poor response times
- Inefficient use of system resources
- Compiler limitations
- How to handle unbounded structures

Let’s take a look at each of these challenges in more detail.

Poor Response Times

The elapsed time from the start to finish for parsing or creating extremely large XML documents can be significant. In many cases, the time required may prohibit it from taking place within an online application where a user is waiting for a response.

Inefficient use of System Resources

An XML document is simply a string of text. Working with a document all at once in a program will require variables large enough to hold the document. If the program will be parsing or creating the document, then variables will also be required for the individual contents. Consider the program declaration area below where a variable of PIC X(30000000) has been declared to hold an entire XML document and a repeating structure occurring 1,000,000 times has been declared to hold the Accounts that can be extracted from the document.

```
01 MY-XML-DOCUMENT          PIC X(30000000).
01 MY-XML-DOCUMENT-CONTENTS.
   03 BANK.
      05 BANK-ID          PIC X(9).
      05 ACCOUNTS OCCURS 1000000.
         07 ACCOUNT-ID     PIC X(9).
```

Figure 1: Sample program data structures for XML document and contents
As you can see, declaring variables of this size will require a large amount of program memory at runtime. Even more troubling is that only a portion of this memory may ever be required. The variables have been declared to work with the most extreme case anticipated—a 30,000,000-byte XML document. It may be the case that almost all documents are less than 500,000 bytes, meaning that in most executions much more system memory is being allocated to the program than is required. This can be of special concern for online applications supporting hundreds of users and running in environments such as CICS where resource management is critical.

**Compiler Limitations**

In addition to an inefficient use of program memory, declaring variables to hold the largest XML document anticipated is sometimes not even possible. In some cases, the variable declarations required will exceed compiler size limits and the program will not compile.

**Handling Unbounded Structures**

The main cause of large XML documents is usually an "unbounded" repeating structure. In XML terms, "unbounded" means that the structure can repeat an unlimited number of times (i.e. there is no upper bounds). For practical purposes, an upper bound can be established to support the most extreme case expected. However, using this approach means that the program is arbitrarily limited to processing a certain number of occurrences of a structure. If that limit is ever exceeded, the program will not be able to process the additional occurrences.

**The Solution – “Streaming” XML**

The solution for working with a very large XML document in your COBOL program is to “stream” the document and process it a portion at a time. When done properly, streaming enables COBOL developers to work with XML documents of unlimited size without hitting compiler or resource limitations and with efficient execution.

Consider the `<BANKS>` XML document (Figure 2. on left) where the `<BANK>` node is repeating and each occurrence of `<BANK>` can contain an unbounded number of `<ACCOUNT>` nodes.

```xml
<?xml version="1.0" ?>
- <BANKS>
  - <BANK BANK-Id="100">
    <BANK-Name>Bank Of Canada</BANK-Name>
    <BANK-Incorporation-Date>2000-07-22</BANK-Incorporation-Date>
  - <ACCOUNT ACCOUNT-Number="12132-E">
    <ACCOUNT-Name>Savings</ACCOUNT-Name>
    <ACCOUNT-Owner>Joe Hendricks</ACCOUNT-Owner>
    <ACCOUNT-Opening-Date>2010-07-22</ACCOUNT-Opening-Date>
    <ACCOUNT-Opening-Time>16:24:37</ACCOUNT-Opening-Time>
    <ACCOUNT-Balance>6748.99</ACCOUNT-Balance>
    <ACCOUNT-Comment>n/a</ACCOUNT-Comment>
  - <ACCOUNT ACCOUNT-Number="8797-A">
    <ACCOUNT-Name>Savings</ACCOUNT-Name>
    <ACCOUNT-Owner>Lawrence Simpson</ACCOUNT-Owner>
    <ACCOUNT-Opening-Date>2010-07-22</ACCOUNT-Opening-Date>
    <ACCOUNT-Opening-Time>16:24:37</ACCOUNT-Opening-Time>
    <ACCOUNT-Balance>69999999999.99</ACCOUNT-Balance>
    <ACCOUNT-Comment>n/a</ACCOUNT-Comment>
  + <ACCOUNT ACCOUNT-Number="49209-A">
    <ACCOUNT-Name>Business</ACCOUNT-Name>
    <ACCOUNT-Owner>ACME Industries</ACCOUNT-Owner>
    <ACCOUNT-Opening-Date>2010-07-22</ACCOUNT-Opening-Date>
    <ACCOUNT-Opening-Time>16:24:37</ACCOUNT-Opening-Time>
    <ACCOUNT-Balance>5259.99</ACCOUNT-Balance>
    <ACCOUNT-Comment>n/a</ACCOUNT-Comment>
  + <ACCOUNT ACCOUNT-Number="78623-3">
    <ACCOUNT ACCOUNT-Number="2321-C">
```
To work with an XML document like this all at once in COBOL, a program data structure similar to the one defined below would be required. Note the "OCCURS 500" clause on BANK-0002 and "OCCURS 1000000000" on ACCOUNT-0007. These are upper bounds set to handle the largest possible XML document expected.

![Program data structures to hold XML content](image)

In addition, a variable large enough to hold the entire XML document will be required.

```
01 CANAN-XML-DATA.
  08 BANK OCCURS 500.
    07 BANK-1N PIC 9(9).
    07 BANK-NAME PIC X(32).
    07 BANK-INCORPORATION-DATE PIC X(10).
    07 ACCOUNT OCCURS 1000000000.
      09 ACCOUNT-NAME PIC X(30).
      09 ACCOUNT-OWNER PIC X(30).
      09 ACCOUNT-OPENING-DATE PIC X(10).
      09 ACCOUNT-OPENING-TIME PIC X(8).
      09 ACCOUNT-BALANCE PIC 9(11)V9(2).
      09 ACCOUNT-COMMENT PIC X(100).
      09 ACCOUNT-ADDRESS:
          11 STREET-ADDRESS PIC X(25).
          11 CITY PIC X(25).
          11 STATE-PROVINCE PIC X(5).
          11 COUNTRY PIC X(25).
```

The above structure will allocate a very large amount of memory when the program runs. There is a possibility that the program may not even compile. Using a streaming approach, these problems are eliminated and runtime efficiency will not be compromised.

The objective behind streaming is to work with only a portion of the document at any one time – thereby allowing you to allocate only the memory required for that portion. There are several approaches for streaming, but one of the better ones is to stream based on repeating structures in a document. This is accomplished by working with a "flattened" structure for your program data fields – one that does not have any "OCCURS" clauses – and processing repeating structures one iteration at a time. To manage a repeating structure while streaming, a counter field should be added to your program data structure. This counter field will be used to keep track of where you are in a repeating structure.

![XML-BUFFER variable to hold entire XML document](image)

Figure 5. shows the modified structure with the OCCURS clause for the BANK and ACCOUNT repeating structures replaced by a new counter field.
Because only a portion of the XML document is worked with at any one time, a smaller variable can be used to hold the XML structure.

Using the flattened data structure and smaller XML buffer reduces the memory requirements of the program. The program can stream the document by processing each Account for a given Bank one occurrence at a time. Once all occurrences of Account have been processed for the first Bank, the next Bank and its Account’s can be processed one-by-one.

**A streaming approach can be used for both reading and writing XML documents.**
Reading a Large XML Document in Streams

To stream the <BANKS> XML Document for reading:

Step 1: Load the First Portion of the XML Document into Memory

Load the first portion of the XML document into a large text field for parsing. In this example, let’s assume the variable is called XML-BUFFER. The variable does not have to contain the entire document – only enough to support parsing down to the first occurrence of the inner-most repeating structure. In the case of the Banks document, this would be the first bank and its first account. The amount loaded into the field can larger than this, but it cannot be less.

In the example below, the first 100,000 of an XML document is loaded to a COBOL field called XML-BUFFER.

![Figure 7: Fill XML-BUFFER with Portion of XML Document](image)

Step 2: Parse Document from Beginning through to First Occurrence of Inner-most Repeating Structure

Begin parsing the XML document loaded into the XML-BUFFER field:

a. Parse all non-repeating nodes and the first occurrence of the inner-most repeating structure encountered. In the Banks document, this is the first occurrence of Account under the first occurrence of Bank.

b. Set the counter fields (BANK-COUNT) and (ACCOUNT-COUNT) to “1” to indicate that the first occurrence of each repeating structure has been opened.

c. Move the parsed contents to the set of program data fields defined to hold one occurrence of the Bank and Account structures.
Step 3: Process Parsed Content

At this point, suspend parsing the document and perform any processing required for the Bank and Account information that was retrieved. For example, there may be database updates to be performed, or you may be writing the parsed contents to a file.
Step 4: Refill XML-BUFFER Field

Once processing is complete, refill the XML-BUFFER field:

a. Shift the portion of XML-BUFFER that was not parsed to the beginning of the field (removing the portion that was already parsed).

b. Re-fill the XML-BUFFER field by appending the next portion of the XML document to the available area at the end.

![Diagram](image1)

*Figure 9: Shift unprocessed portion to beginning of XML-BUFFER*

![Diagram](image2)

*Figure 10: Refill XML-BUFFER field*
Step 5: Process Remaining Occurrences of Inner-most Repeating Structure

Continue parsing and processing each occurrence of Account - one iteration at a time:

a. Clear the structure used to hold the Account information. This is to ensure that information from one account is not carried over to a subsequent account.

b. Resume parsing and extract the second occurrence of Account from the XML-BUFFER field.

c. Increment ACCOUNT-COUNT to indicate that another occurrence of Account was found under the current Bank.

d. Suspend parsing again and perform any processing required for that account.

e. Refill the XML-BUFFER field, removing the portion processed, shifting the unprocessed portion to the beginning and appending additional portions to the end (if necessary).

Figure 11: Move the second occurrence of Account to COBOL fields
Step 6: Repeat Steps 4 and 5 until all Occurrences of Account Have Been Processed.

Continue extracting each Account from the XML document one-by-one, moving the contents to program fields, processing the contents and then refilling XML-BUFFER.

Figure 12: Parse each occurrence of Account to COBOL fields

Once all accounts for the first bank have been processed, move to the next bank and process its accounts one-by-one.

Figure 13: Begin parsing contents for the next Bank

Step 8: Repeat Steps 4, 5, 6 and 7 until the Entire Document has beenParsed and Processed.
Writing an XML Document in Streams

To write the <BANKS> XML Document using the streaming approach:

Step 1: Load the First Portion of XML Content into Memory

Load content for the XML document from the beginning up to the first occurrence its inner-most repeating structure into the flattened program data structure. In the case of the Banks document, this is the first occurrence <BANK> and its first <ACCOUNT>.

![Figure 14: Populate program fields with content for the first Account of the first Bank]

Step 2: Create XML Document up to First Occurrence of Inner-most Repeating Structure

Create the beginning portion of the XML document using the content from the program data fields. Use a large text field to hold the portion of the XML document created (e.g. XML-BUFFER PIC X(100000)). The text field does not have to be declared large enough to hold the entire XML document – only the portion being created.

![Figure 15: Contents of XML-BUFFER field holding first portion of the XML document]

Step 3: Store and clear XML-BUFFER

Write the portion of the XML document created to a permanent data store. The XML-BUFFER field can now be cleared which will allow it to be reused for the next portion of XML.
Step 4: Create Next Portion of XML

a. Populate the program data fields with the next occurrence of the inner-most repeating structure.

b. Create the XML portion for just that occurrence of Account.

c. Append that portion to the final XML document you are building in your permanent data store.

```move
MOVE "221-C-343" TO ACCOUNT-NUMBER
MOVE "Chequing" TO ACCOUNT-NAME
MOVE "James Wilson" TO ACCOUNT-OWNER
MOVE "1998-09-21" TO ACCOUNT-OPENING-DATE
MOVE "09:00:00" TO ACCOUNT-OPENING-TIME
MOVE 36300.50 TO ACCOUNT-BALANCE
MOVE "21 Main Street" TO STREET-ADDRESS
MOVE "Toronto" TO CITY
MOVE "ON" TO STATE-PROVINCE
MOVE "Canada" TO COUNTRY
MOVE "NA" TO ACCOUNT-COMMENT
```

Figure 16: Populate Program Fields with Content for the next occurrence of Account

```xml
<ACCOUNT ACCOUNT-Number="221-C-343"><ACCOUNT-Name>Chequing</ACCOUNT-Name><ACCOUNT-Owner>James Wilson</ACCOUNT-Owner><ACCOUNT-Opening-Date>1998-09-21</ACCOUNT-Opening-Date><ACCOUNT-Opening-Time>09:00:00</ACCOUNT-Opening-Time><ACCOUNT-Balance>36300.50</ACCOUNT-Balance><ACCOUNT-Comment>Na</ACCOUNT-Comment><ACCOUNT-Address><Street-Address>21 Main Street</Street-Address><City>Toronto</City><State-Province>ON</State-Province><Country>Canada</Country></ACCOUNT-Address></ACCOUNT>
```

Figure 17: Contents of XML-BUFFER field holding next iteration of Account

Step 5: Repeat Step 4 until all Occurrences of Inner-most Repeating Structure Have Been Processed

Continue creating the XML for each occurrence of Account one-by-one and appending the XML to the final document in the permanent data store.


Once all Accounts for the first Bank have been processed, move to the next Bank and process its Accounts one-by-one.

Step 7: Repeat Steps 3, 4, 5 and 6 until the Entire XML Document Has Been Created.
Implementing XML Streaming in COBOL using XML Thunder

XML Thunder from Canam Software Labs, Inc. is a leading XML / SOAP mapping and code generation solution for COBOL applications. Using a visual designer, you define the mapping between XML nodes and program data fields. XML Thunder generates a callable COBOL module that you can use within your COBOL applications. The module will read or write an XML document based on your design specifications.

For extremely large documents, XML Thunder supports streaming. This enables XML Readers and Writers generated by XML Thunder to process XML documents of unlimited size and containing structures with an unbounded repeating occurrence.

![XML Thunder toolset](image)

*Figure 18: The XML Thunder toolset*
The following is an overview of XML Thunder and how it supports streaming of large XML documents.

**Step 1 – Create an XML Handler Design**

a) **Launch the New Handler Design Wizard**

To create a new XML handler, launch the XML wizard by either clicking on *XML Handler ➔ New...* from the menu or click on the icon on the tool bar.
b) Select Source For XML and Program Data Field Structures

An XML Thunder Handler Design is based on a mapping between program data fields (called an IDS) and XML nodes. To create a new XML Handler, begin by specifying the source for these two data structures. You can choose from a number of possible IDS and XML combinations, but for this example choose “Derive from XML” for the IDS and “XML Structure -> XSD” for the XML. Prompt out and select the the schema (this example is working with the BAXSD.xsd schema found in the tutorials folder under the XML Thunder installation folder - e.g. c:\program files\canam\XML Thunder\Tutorials\BAXSD.xsd).

![Figure 20: Choose the IDS and XML source](image)

---

c) Select XML Document Root Element

Some schemas can have more than one possible root element (i.e. they define different XML instances). This schema has only one possible root though so we can accept the default.

![Figure 21: Choose the XML document root element](image)
d) **Confirm Selections**

The Summary window lists the selections made. Click “Finish” to confirm the selections and create an XML Handler Design.

![XML Handler Wizard: Summary](image)

*Figure 22: Confirm selections*
Step 2 – Review the XML Handler Design

When you click on “Finish”, XML Thunder will use your selections to create an XML Handler Design. In this example, the XML structure and validation rules have been created using the selected schema and any subordinate schemas it imported. The program data fields – also called Interface Data Structure (IDS) – were derived from the XML nodes. The mappings between the XML nodes and program data fields were done automatically.

You can select an XML node in the design to view its properties as well as the program data field it is mapped to.

![XML Thunder designer showing node properties and a mapping between program data fields and XML nodes](image)

*Figure 23: The XML Thunder designer showing node properties and a mapping between program data fields and XML nodes*
Step 3 – Generate the XML Handlers

From the Handler Design, you can generate either an XML Reader or an XML Writer. You can also optionally choose to generate a Test Harness for testing and an IDS (copybook or C Header) file.

![XML Thunder interface showing XML Reader and Writer options](image)

**Figure 24 a) and b): Generate the XML Reader**

You can choose between two XML Handler types for code generation – “document-level” and “node-level”. A document-level handler will work with the entire XML document all at once and process it in one call. A node-level handler will work with only portion of the document at any one time and process it in several calls. Node-level handlers “stream” the document enabling the calling program to work with an XML document of any size. XML Thunder supports streaming for both readers and writers.

** Note: The same XML Thunder Handler Design can be used to generate either a document-level or node-level Handler. It is simply a code generation option to control which handler type is implemented.

**XML Reader Generation Tab**

You can confirm / modify the program Id and source code file names for the XML Reader, the IDS (a copybook for the Handler API) and the Test Harness.
XML Writer Generation Tab

Just as with an XML Reader, you can confirm / modify the program Id and source code file names for the XML Writer, the IDS (a copybook for the Handler API) and the Test Harness.

![XML Writer Generation Tab](image)

**Figure 25: Generate the XML Writer**

If you choose to generate both an XML Reader and an XML Writer - as well as the Test Harnesses and copybook files, then 6 source code files will be created.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAXSDRC.cpy</td>
<td>10/17/2009 12:53 PM</td>
<td>CPY File</td>
</tr>
<tr>
<td>BAXSDWC.cpy</td>
<td>10/17/2009 12:53 PM</td>
<td>CPY File</td>
</tr>
</tbody>
</table>

**Figure 26: The generated source code files**
**XML Reader Source Code**

For an XML Reader, the following code has been generated:

**BAXSDR** – The **XML Reader** module. This is a COBOL sub-program. You call this program passing it the XML document in a variable (called XML-BUFFER). The Reader validates the document, parses the nodes into program data fields and returns these fields back to the calling program in the IDS variables. A node-level XML Reader will be called multiple times for one XML document. Each call will pass in only a portion of the XML document for parsing.

**BAXSDRT** – The **XML Reader Test Harness**. This is a fully functional program used to test the XML Reader. It can read in an XML file, or create one inline using dummy data values. The Test Harness passes the XML document to the XML Reader via the XML-BUFFER variable. It receives back the parsed content in the IDS (i.e. program data fields). A node-level test harness will call the XML Reader multiple times – once for each occurrence of a repeating structure.

Note: The Test Harness is optional. It gives you a great way to quickly test your XML Reader. It also gives you an example of how you would call the XML Reader from your own program.

**BAXSDRC** – The **XML Reader Copybook**. This is the IDS (program data fields) used as the API to the XML Reader. This is optional. Neither the Test Harness, nor the XML Reader use the copybook. It is generated for your convenience – for use in your programs that will call the Reader.

**XML Writer Source Code**

For an XML Writer, the following code has been generated:

**BAXSDW** – The **XML Writer** module. This is a COBOL sub-program. You call this program passing it the XML content via the IDS (i.e. program data fields). The Writer will validate the content and assemble the XML document. The XML document is returned to the calling program via the XML-BUFFER variable. A node-level XML Writer will be called multiple times for one XML document. Each call will pass in the content for only a portion of the XML document.

**BAXSDWT** – The **XML Writer Test Harness**. This is a fully functional program used to test the XML Writer. It populates the IDS (i.e. the program data fields) using dummy data values (x’s for character and 9’s for numeric values). The Test Harness passes the XML content via the program data fields to the XML Writer. It receives back the XML document via the XML-BUFFER variable. A node-level test harness will call the XML Writer multiple times – once for each occurrence of a repeating structure.

Note: The Test Harness is optional. It gives you a great way to quickly test your XML Writer. It also gives you an example of how you would call the XML Writer from your own program.

**BAXSDWC** – The **XML Writer Copybook**. This is the IDS (program data fields) used as the API to the XML Writer. This is optional. Neither the Test Harness, nor the XML Writer use the copybook. It is generated for your convenience – for use in your programs that will call the Writer.
**Step 4: Running a Node-level Handler**

To call a node-level XML Reader, the following steps are taken:

1. Load XML-BUFFER with a portion of the XML document.
2. Call the XML Reader to parse down to – and including - the inner-most nested repeating structure.
3. Perform processing with parsed contents
4. Clear program data fields
5. Refill XML-BUFFER
6. Call XML Reader to continue parsing. Repeating structures are parsed one iteration at a time. If a Bank has 100 Accounts, the XML Reader will be called 100 times.

The following code example shows the repeated calls to the XML Thunder Reader (the actual call is inside the INVOKE-XML-HANDLER paragraph).

```
On the first call, the XML Reader will parse all data down to *(and including) the first occurrence of the inner-most repeating structure.
PERFORM INVOKE-XML-HANDLER THRU
 INVOKE-XML-HANDLER-EXIT
* Repeating structures can be controlled by means of the associated COUNTER variable.
* A COUNTER value greater than zero means that the XML Reader is still reading the repeating structure associated with COUNTER.
* A COUNTER value of zero means that the XML Reader did not parse that repeating structure in the last call.

PERFORM
 UNTIL BANK-COUNT < 1
 * XML Reader found an occurrence of <BANK>

PERFORM
 UNTIL ACCOUNT-COUNT < 1
 * XML Reader found an occurrence of <ACCOUNT>
  * Perform any process required for Bank and current occurrence of Account.
  * Clear the Account program data fields and call the XML Reader to continue parsing.
  PERFORM INVOKE-XML-HANDLER THRU
       INVOKE-XML-HANDLER-EXIT

END-PERFORM
  * Clear the Bank and Account program data fields and call the XML Reader to continue parsing.
  PERFORM INVOKE-XML-HANDLER THRU
       INVOKE-XML-HANDLER-EXIT

END-PERFORM
```

*Figure 27: Repeated calls to the XML Reader*
The call to the XML Reader looks as follows. CANAM-XML-BUFFER is passed to the XML Reader and it contains the XML document for parsing. CANAM-XML-DATA contains program data structures to hold the parsed contents returned by the handler and CANAM-XML-STATUS contains the return code from the call.

```assembly
INVOKE-XML-HANDLER.
*The INVOKE-XML-HANDLER paragraph will prepare the XML-BUFFER field and then call the XML Thunder XML Reader program. This paragraph will be called once for each occurrence of a repeating structure found in the XML document.

*Before calling the XML Reader, add code to do the following:
*1. Fill the XML-BUFFER variable for parsing.
*   For the first call, fill XML-BUFFER with as much of the XML document as it will hold.
*   For each subsequent call, append additional XML to the end of XML-BUFFER. An XML-BPTR-BUFFER field is provided to identify the first free position of the XML-BUFFER.
*   For example:
*   MOVE XML-DOC-REC(CURR-INDEX, NEXT-PORTION-INDEX)
*   TO XML-BUFFER(XML-BPTR-BUFFER,XML-BUFFER-MAX)

CALL "BANKR" USING
   CANAM-XML-DATA
   CANAM-XML-BUFFER
   CANAM-XML-STATUS
END-CALL.
   IF XML-RETURN-CODE-OK
      OR XML-RETURN-CODE-MD
      SET TH-XML-PROCESSING-NORMAL TO TRUE
   ELSE
      SET TH-XML-PROCESSING-ERROR TO TRUE
   END-IF
*The XML Reader (BANKR) does the following:
*1. Parses down to the next occurrence of the inner-most nested repeating structure.
*2. Removes the portion of XML-BUFFER that was parsed.
*3. Shifts any unparsed portion to the beginning of XML-BUFFER.
*
INVOKE-XML-HANDLER-EXIT.
   EXIT.
```

Figure 28: Calling the XML Reader

To call a node-level XML Writer, the following steps are taken:

1. Load the program data fields with content for XML content. For repeating structures, load the first iteration.

2. Call the XML Writer to create XML down to – and including – the first occurrence of the inner-most nested repeating structure.

3. Store the XML message portion to a permanent data store.

4. Clear the XML-BUFFER
5. Refill program data fields with content for the next iteration of content.

6. Call XML Writer to continue parsing. Repeating structures are created one iteration at a time. If a Bank has 100 Accounts, the XML Writer will be called 100 times.

The following code example shows the repeated calls to the XML Thunder Writer (the actual call is inside the CALL-TO-HANDLER paragraph).

```
* On the first call, the XML Writer will output all data down to
* the first occurrence of the inner-most nested repeating
* structure.
* The BANK-COUNT field is used to control the Bank repeating
* structure. Set the count to 1 when writing out the last
* occurrence. Set the count to any value > 1 for all occurrences
* before that.

   MOVE 2 TO BANK-COUNT
   PERFORM
     UNTIL BANK-COUNT < 1
     Move content for the XML document to the program data
     fields. For example:
     MOVE 999999999 TO BANK-ID
   END-PERFORM

   For each occurrence of Bank, process all occurrences
   of Account.
   The ACCOUNT-COUNT field is used to control the Account
   repeating structure. Set the count to 1 when writing out
   the last occurrence. Set the count to any value > 1
   for all occurrences before that.

   MOVE 2 TO ACCOUNT-COUNT
   PERFORM
     UNTIL ACCOUNT-COUNT < 1
     Move content for the XML document to the program data
     fields. For example:
     MOVE "XXXX" TO ACCOUNT-NUMBER
     PERFORM CALL-TO-XML-HANDLER
     THRU CALL-TO-XML-HANDLER-EXIT

     END-PERFORM
   END-PERFORM

   BUSINESS-LOGIC-BEFORE-EXIT.

   Exit.
```

*Figure 29: Repeated calls to the XML Writer*
The call to the XML Writer looks as follows. CANAM-XML-DATA is passed to the XML Writer and it consists of program data structures holding the content for the XML document to be created. CANAM-XML-BUFFER contains the portion of the XML document returned by the handler and CANAM-XML-STATUS holds the return code from the call.

```
CALL-TO-XML-HANDLER.
* Call the XML Writer to create the next portion of XML
* The counters assigned to each repeating structure
* Set the counter to 1 to inform the Writer when it is
* on the last occurrence.
* The XML Writer will decrement the counter each time
* an occurrence is output.
  CALL "BANKW" USING
  CANAM-XML-DATA
  CANAM-XML-BUFFER
  CANAM-XML-STATUS
  END-CALL.
  IF XML-RETURN-CODE-OK
    OR XML-RETURN-CODE-MD THEN
      SET TH-XML-PROCESSING-NORMAL TO TRUE
      After each call to the XML Writer, output the portion
      of XML that was created. The XML-BPTR-BUFFER field
      specifies how much of the buffer was populated on the
      last call.
      For example:
      WRITE XML-BUFFER(1:XML-BPTR-BUFFER) TO
      OUTPUT TARGET
    ELSE
      SET TH-XML-PROCESSING-ERROR TO TRUE
  END-IF
.
CALL-TO-XML-HANDLER-EXIT.
  Exit.
```

Figure 30: Calling the XML Writer

Summary

Working with extremely large documents in COBOL presents several challenges in terms of compiler limitations and run-time performance. Using an approach called “streaming”, these challenges can be overcome enabling COBOL applications to process XML documents of unlimited size.

XML Thunder implements streaming using a feature called “node-level” processing. Using a visual designer, a developer can define a mapping between program data fields and XML nodes and then generate a callable COBOL module to stream an XML document for either reading or writing.

For more information on XML Thunder and how it implements streaming please contact Canam Software Labs at sales@canamsoftware.com and/or http://www.xmlthunder.com.