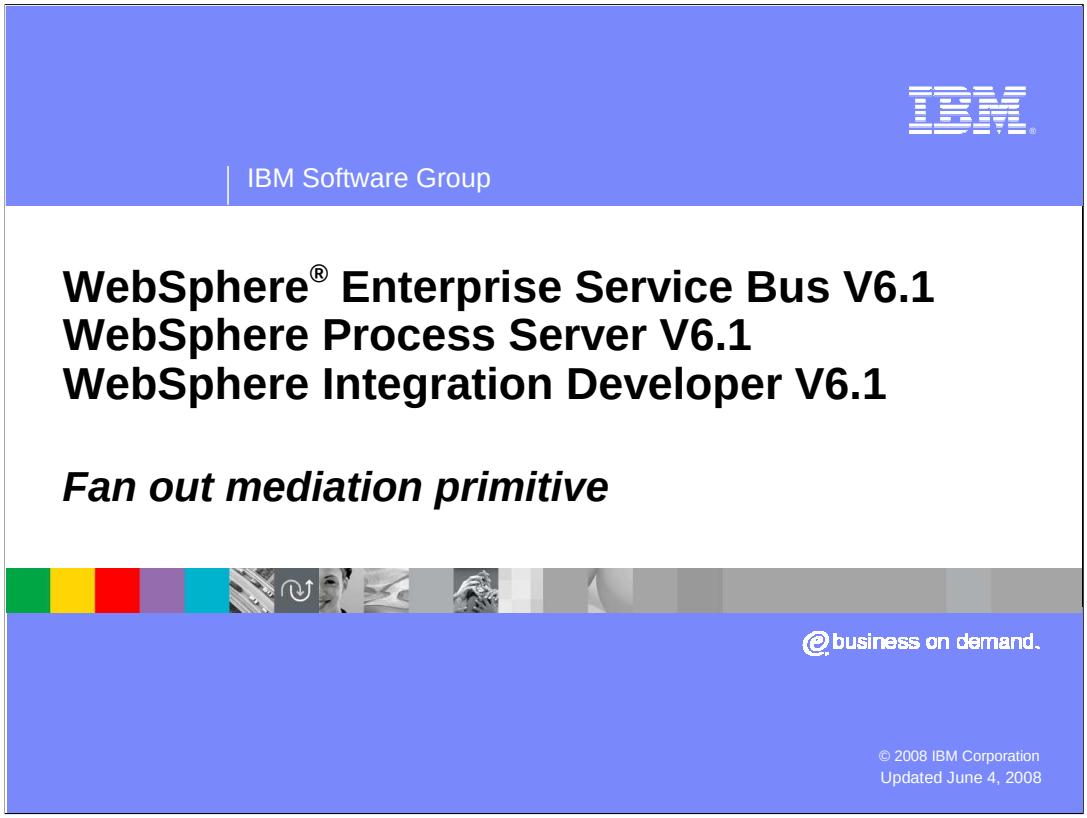


This presentation provides a detailed look at the fan out mediation primitive.

Goals

- Understand the fan out mediation primitive

Fan out

  - Overview of function
  - Use of terminals
  - Definition of properties
  - Error handling
  - Details of usage scenarios

The goal of this presentation is to provide you with a full understanding of the fan out mediation primitive.

The presentation assumes that you are already familiar with the material presented in the **Mediation primitive common details** presentation and the **Common details – Promoted properties** presentation. These two presentations serve as a base for understanding mediation primitives in general.

An overview of the fan out primitive is presented along with information about the primitive's use of terminals and its properties. Some error handling considerations are provided, followed by a series of usage scenarios showing the various ways in which a fan out can be used in a flow.

Overview of function

- The fan out primitive provides either:
  - The front of an aggregation scenario
  - Message broadcast
- For an aggregation scenario
  - There is a fan in primitive which acts as the point of aggregation
  - A fan in must be associated with a specific fan out instance
- Fan out has two modes of operation
  - Iterate mode on
    - Iterates though a repeating element contained within the input message
    - Output terminal fired once for each element instance
    - Output message contains input message plus copy of element instance
  - Iterate mode off
    - Output terminal is fired once for each of multiple flow paths wired to it
    - Output message is identical to the input message

When considering a fan out primitive, there are two basic ways in which it can be used, either participating in an aggregation scenario or used for enabling message broadcast. When used as part of an aggregation scenario, there is a specific fan in primitive instance in the flow that is associated with the fan out. The fan out is the beginning and the fan in is the end of the flow segment that performs the aggregation.

The fan out primitive has two modes of operation, the first being the iterate mode. In this mode, the fan out iterates through a repeating element that is contained in the input message. The output terminal of the fan out is fired once for each element. When the output terminal is fired, the SMO contains the original message as it arrived at the fan out, plus a copy of the element instance that this iteration is for that is contained in the context.

When iterate mode is off, the output terminal is fired once. In this mode, the flow is constructed with multiple flow paths following the fan out. So in actuality, each flow path wired to the output terminal is driven. In this case, the SMO passed to each path is unchanged from the fan out's inbound SMO.

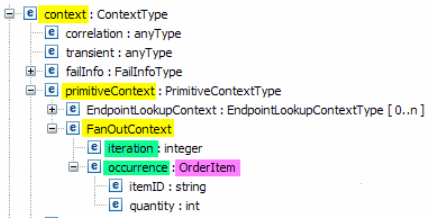
Overview of function

- There are four basic scenarios possible
  - Aggregation with iterate mode on
  - Aggregation with iterate mode off
  - Broadcast with iterate mode on
  - Broadcast with iterate mode off
- Aggregation scenarios and fan in completion criteria
  - A fan in is configured with completion criteria
  - Completion criteria affects overall flow path
    - Between the fan out and fan in
    - Flow following the fan in
  - Configuration of fan out and fan in completion criteria must be complementary

Consider that a fan out can be used in an aggregation or broadcast scenario and that it also has two modes of operation, iterate mode on or iterate mode off. The result is that there are four overall basic usage scenarios in which a fan out can participate. The first is an aggregation using iterate mode to loop through an array of elements, performing the same processing for each element. When all the elements have been processed, the associated fan in completes and the results of the aggregation are constructed in the SMO by the flow following the fan in. The next is also an aggregation, but in this case with iterate mode off. In this case, there is multiple flow paths between the fan out and fan in, with each flow path running once. When all the flow paths have completed, the fan in completes and the results of the aggregation are constructed in the SMO by the flow following the fan in. The third is a broadcast with iterate mode on. This allows each element of an array in the incoming message to have the same processing performed. However, there is no fan in and the results of processing each element are not aggregated together. Finally, there is broadcast with iterate mode off. In this case, the fan out serves as the head of multiple flow paths, each of which is passed the same message, and the results of processing are not aggregated together.

In aggregation scenarios, the fan in associated with the fan out is configured with completion criteria. The completion criteria will affect the overall flow, controlling the flow between the fan out and fan in and determining when the flow following the fan in should be driven. Because of this, it is important that the configuration of the fan out, the construction of the flow between the fan out and fan in and the completion criteria of the fan in complement each other. More details on fan in completion criteria are provided in the fan in primitive presentation.

FanOutContext



- Iterative scenarios and the FanOutContext
  - Problem
    - Each time the fan out output terminal is fired the entire message body is included in the SMO
    - How do the primitives in the flow know which iteration this is and which element to process?
  - Solution
    - FanOutContext in the SMO
    - Contains the array index and array element for this iteration
  - FanOutContext usage
    - Located in the SMO at context/primitiveContext/FanOutContext
    - iteration – zero based integer index identifying the element being processed
    - occurrence – a copy of the array element being processed

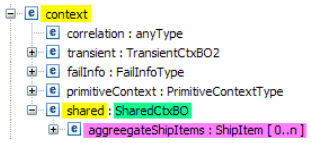
The FanOutContext is a key element that is used during iterative processing scenarios. This slide examines why it is needed, what it provides and how it is used.

When processing in iterate mode, the fan out fires the output terminal once for each element of an array. The SMO that gets propagated contains the entire array as part of the SMO body. The problem is that the primitives downstream from the fan out need to have a way of knowing which of the repeating elements should be processed during this iteration.

The solution to this is provided by the fan out context. It is initialized by the fan out primitive to contain an array index and a copy of the array element found at that index.

When building your aggregation flow, you define your primitives to access the array element from the fan out context rather than the message body. It is located in the SMO at context/primitiveContext/FanOutContext and contains two fields. The first field is called iteration and is an integer value defining the current iteration. The value in this field is zero based, so it has a value of zero for the first iteration, a value of one for the second iteration, and so on. The next field is called occurrence and contains a copy of the element at that index. The occurrence field is strongly typed to match the type of the array elements being iterated over. This allows you to make use of the strong typing information when defining your flow between the fan out and fan in.

Shared context



- Aggregation scenarios and the shared context
  - Problem
    - Each time the fan out output terminal is fired a new SMO instance is created
    - Each new SMO instance is a deep copy
    - How are results from each iteration/flow between the fan out and fan in aggregated?
  - Solution
    - Shared context in the SMO
    - Single memory area that is not deep copied with each SMO instance
  - Shared context usage
    - It is defined by a business object (similar to transient and correlation contexts)
      - In iterative aggregations, the business object typically contains an array
    - Flows between fan out and fan in set values to be aggregated into the shared context
    - After the fan in completes, subsequent primitives use the contents of the shared context to build the aggregated message

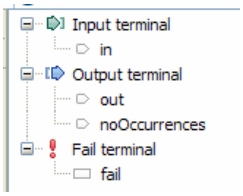
This slide examines the shared context used during aggregation scenarios, examining why it is needed, what it provides and how it is used. The first thing to look at is how the fan out handles the SMO when firing its output terminal. The original message arriving at the fan out is saved by the primitive, and a new deep copy is created and passed through the output terminal to the flow. Whatever changes are made to the SMO during the flow are not seen by the other iterations or flow paths. Each receives a new copy of the message as it arrived at the fan out. This poses a problem in an aggregation scenario where the results of processing each iteration or flow are to be aggregated together.

The solution to this is the shared context which is kept in a shared memory area. Each time the SMO is deep copied, rather than copying the shared context the SMO contains a reference to the shared memory area.

When building your aggregation flow, you define what the shared context will contain using a business object, similar to how you define the transient or correlation contexts. For an iterative aggregation, the business object typically contains an array. Each iteration or flow between the fan out and fan in needs to update the shared context with the data it is contributing to the aggregated result. Once the fan in completion criteria is met, the flow following the fan in can take the contents of the shared context and use it to build the aggregated message in the SMO body.

Terminals

- Terminals:
  - Input terminal
  - Two output terminals
  - Fail terminal



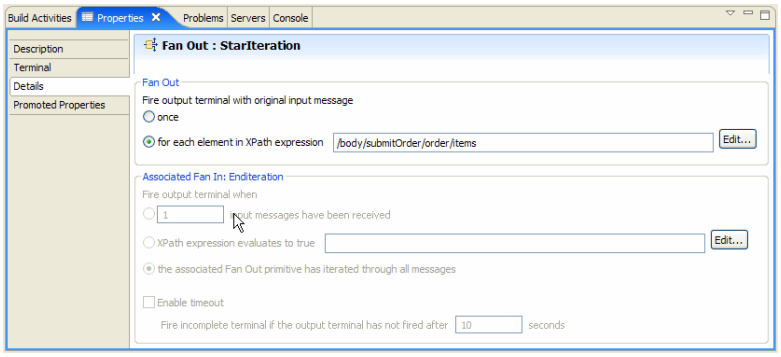
- Output terminals
  - out
    - Iterate mode on – fired once for each repeating element
    - Iterate mode off – fired once for each path wired to terminal
  - noOccurrences
    - Iterate mode on – there are no instances of the repeating element
    - Iterate mode off – exists but will never be fired
- All terminals must be for the same message type
  - Out terminal in iterate mode
    - The terminal message type is implicitly augmented
    - The weakly typed fan out context is cast to the correct type for the iterative element

The fan out primitive has one input terminal, two output terminals and a fail terminal. The first output terminal, named out, is where the message received by the primitive is propagated down the flow. When iterate mode is on, this terminal is fired once for each repeating element in the array being iterated over. When iterate mode is off, this terminal is fired once. However, in this case, it is normal to have multiple flow paths wired from this terminal, and therefore the flow for each flow path is taken. These flow paths are not done in parallel, but rather sequentially, with one starting upon completion of another. The sequential order in which the flow paths are run is not determinate.

The second output terminal is named noOccurrences. When operating with iterate mode on, this terminal is fired rather than the out terminal if there are no repeating elements in the incoming SMO. When iterate mode is off, this terminal is present but will never be fired.

All the terminals are for the same message type because the fan out primitive does not change the message body. However, when in iterate mode, the out terminal is augmented so that the weakly typed fan out context is cast to the specific type of the array element that is placed in it.

Properties and promoted properties



- Properties:
  - Fire output terminal:
    - Once
    - For each element within an iteration in the SMO
      - XPath expression used to identify the element to iterate on
      - XPath expression builder dialog enabled to help construct the expression
  - Associated fan in properties displayed read only
- Promoted properties - the fire output terminal property is not promotable

This slide looks at the properties for fan out and its promoted properties.

There is only one property for a fan out primitive, fire output terminal. It can be set to either of two values. Setting it to once indicates the fan out will run with iterate mode off. It can also be set for each element within an iteration in the SMO which indicates iterate mode on. When this is the case, you specify an XPath expression that identifies the location of the repeating element in the SMO.

When being used for an aggregation scenario, the fan out will have an associated fan in. It is important that the property settings of the fan out and fan in complement each other. Therefore, the properties for the fan in are shown on the fan out properties panel. The fan in properties are read only on this panel, but enable you to compare the configurations of the fan out and associated fan in.

The fire output terminal property is not promotable because changing it normally requires changes to the flow logic, and therefore it is not something that should be done at runtime.

Error processing

- MediationBusinessException (fail terminal flow)
  - Element for iterate XPath expression not found in SMO
- Processing when array has no elements
  - noOccurrences terminal wired – that flow is taken
  - noOccurrence terminal not wired
    - With associated fan in – flow continues following the fan in
    - Without associated fan in – flow stops (similar to wiring to a stop primitive)
- Iterate XPath expression identifies a non-array
  - Not considered an error
  - Flow will perform one iteration

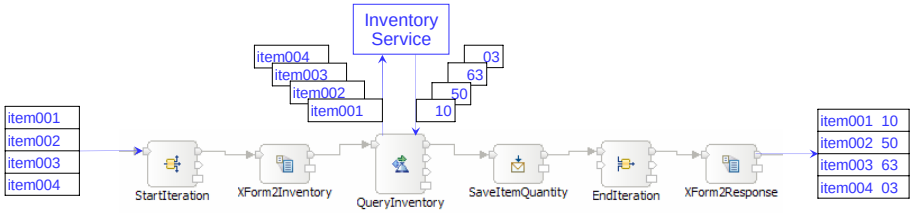
Some of the possible error conditions you might encounter are described on this slide.

A MediationBusinessException is raised if the element identified by the iterate XPath expression is not found in the SMO. If the fail terminal is wired, the fail flow is taken.

Another condition that can occur is the array exists in the SMO but has no elements. When this occurs, if the noOccurrences terminal is wired, the flow from that terminal is taken. If the noOccurrences terminal is not wired, the resulting behavior depends upon whether there is an associated fan in. When there is a fan in, the flow will continue following the fan in, skipping over any flow between the fan out and fan in. If there is no associated fan in, the flow stops, similar to the behavior seen if the noOccurrences terminal was wired to a stop primitive.

It is possible in WebSphere Integration Developer to specify an iterate XPath expression that identifies an element that is not an array. This is not considered an error, and the flow will proceed as if it had been an array with one element.

Aggregation with iterate mode on



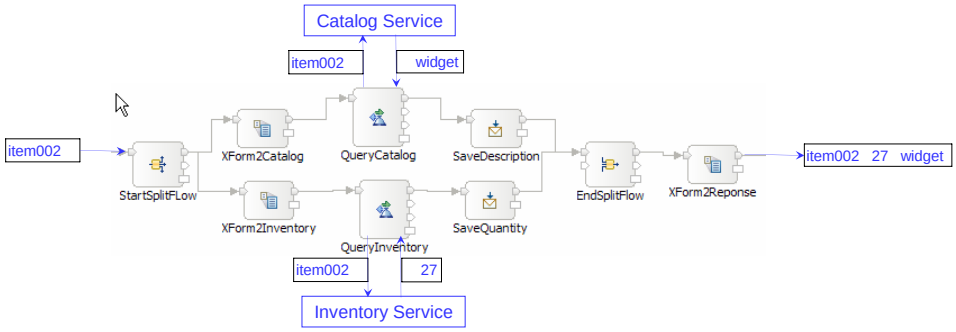
- Determine status of inventory for list of items
  - Request contains array of item IDs
  - Response contains array of item IDs and quantity in stock
  - Flow iterates for each item between fan out and fan in
  - Inventory service called for each individual item
  - Shared context used to save individual results during iteration
  - Contents of shared context use to build the final response message

The next four slides look at the four basic scenarios for using a fan out. On this slide, the aggregation scenario with iterate mode on is examined.

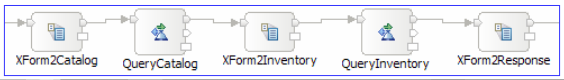
In this scenario, a request is made to find out the inventory status of a list of items. The input contains a list of item IDs and the response is the list of item IDs along with the current quantity of each item that is in stock. There is an inventory service which can be queried to determine the in stock quantity, but this service can only be called for a single item at a time, not for a list of items.

To implement this scenario, there is a fan out and an associated fan in. Looking at the flow above, starting on the left, you can see a list of item IDs being passed into the StartIteration fan out primitive. It iterates through the array, passing the SMO with each element to the XForm2Inventory XSL transformation primitive. This primitive does two things. It sets up the message body so that the call to the inventory service can be made, and it saves the item ID in the shared context. The QueryInventory service invoke primitive calls the inventory service, obtaining the in stock quantity for that one item. The next primitive is the SaveItemQuantity message element setter which takes the item quantity returned and saves it in the shared context. The EndIteration fan in primitive is next, which causes the flow to return to the StartIteration fan out unless all items have already been processed. When that is the case, the flow continues to the XForm2Response XSL transformation which takes the values from the shared context and builds the response message with the list of item IDs and quantities.

Aggregation with iterate mode off



- Determine description and quantity in stock for an item
  - Request contains single item ID
  - Response contains single item ID with quantity and description
  - Two flow paths between fan out and fan in, each calling a service
  - Shared context used to save individual results of each flow path
  - Contents of shared context use to build the final response message
- Consider chained aggregation



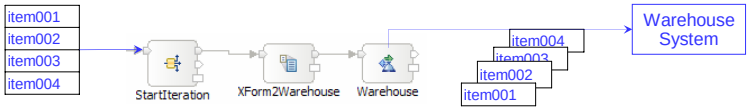
This scenario is for aggregation with iterate mode off.

In this flow, a single item ID is received and the response contains the item ID, an in stock quantity and item description. The in stock quantity is obtained from the inventory service and the description is obtained from a catalog service.

Looking at the flow, you can see the item ID entering the StartSplitFlow fan out primitive, which is configured in once mode. When the out terminal is fired, the flow passes to the XForm2Catalog XSL transformation which saves the item ID in the shared context and sets up the message body for the call to the catalog service. The QueryCatalog service invoke primitive makes the call and receives the item description in response. The SaveDescription message element setter saves the description in the shared context. The flow continues to the EndSplitFlow fan in, which is configured to complete after it receives two messages. Since this is the first message, the flow returns to the StartSplitFlow fan out primitive, and continues to the XForm2Inventory XSL transformation. This primitive sets up the body to call the inventory service. The QueryInventory service invoke primitive calls the inventory service and receives the in stock quantity in response. The SaveQuantity message element setter saves the quantity in the shared context and the flow proceeds again to the EndSplitFlow fan in, which is now complete because this is the second message received. The flow proceeds to the XForm2Response XSL transformation, which builds the response message body from the item ID, quantity and description that is saved in the shared context.

It is possible that a chained aggregation flow, without using a fan out and fan in, might be a more practical approach to this scenario. The flow at the bottom of the slide illustrates this alternate approach. If it was possible to achieve parallel processing using the fan out fan in approach, it might have an advantage over the chained aggregation. However, within the context of a fan out and fan in, a service invoke primitive cannot be used with the invoke asynchronous with callback invocation style, and therefore parallelism cannot be achieved.

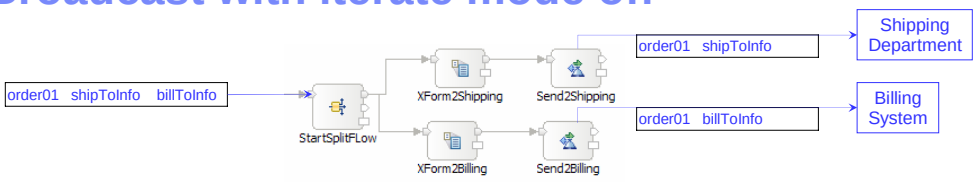
Broadcast with iterate mode on



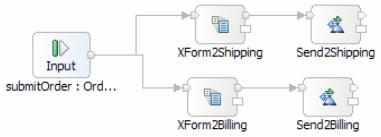
- Send tickets to warehouse to pull individual items
  - Request contains array of item IDs
  - Flow iterates for each item
  - One way message sent to warehouse system for each

This scenario looks at broadcast with iterate mode on. The incoming request contains a list of items that need to be pulled from the warehouse and the output is messages to the warehouse, one for each item that is to be pulled. You can see the list of items on the left as they enter the StartIteration fan out primitive, which is configured in iterate mode. From the fan out the flow proceeds to the XForm2Warehouse XSL transformation which sets up the call to the warehouse system. The Warehouse service invoke primitive makes a one way call to the warehouse system, containing the item ID for a single item. There is nothing wired to the Warehouse service invoke so the flow for that item completes at this point, and the flow returns to the StartIteration fan out. This continues until all items have been processed, at which point the flow completes.

Broadcast with iterate mode off



- Send order information to shipping and billing
  - Request contains order with shipping and billing information
  - Flow split into two paths
  - One way messages sent to shipping and billing systems
- Consider same flow without a fan out primitive
  - Fan out provides no additional function versus splitting flow from input node



This is the last scenario, performing a broadcast with iterate mode off. In this scenario, there is an incoming order that has both billing information and shipping information. It needs to send a request to the shipping department to ship the order and another request to the billing system to bill the order. On the left you can see the order coming in to the StartSplitFlow fan out with the shipping and billing information. The flow proceeds to the XForm2Shipping XSL transformation which sets up the call to the shipping department. The Send2Shipping service invoke primitive passes the order number and shipping information using a one way operation, which is the end of this flow path. Control is returned to the StartSplitFlow fan out, which passes the SMO to the XForm2Billing XSL transformation which sets up the call to billing. The Send2Billing service invoke uses a one way call to send the order number and billing information to the billing system. The flow is then complete.

Turning you attention to the bottom of the slide, there is another version of this flow which does not contain a fan out. The split flow is done directly off of the input node. For this scenario, the fan out provides no additional function over doing a split flow from the out terminal of any other primitive or node.

Summary

- Examined the fan out mediation primitive


Fan out

  - Overview of function
  - Use of terminals
  - Definition of properties
  - Error handling
  - Details of usage scenarios

In summary, this presentation provided details regarding the fan out mediation primitive. It presented an overview of fan out along with information about the primitive's use of terminals and its properties. Some error handling considerations were provided, followed by a series of usage scenarios showing the various ways in which a fan out can be used in a flow.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBIV61_FanOutPrimitive.ppt

This module is also available in PDF format at: ../WBIV61_FanOutPrimitive.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM WebSphere

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

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

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.