WebSphere Application Server

WebSphere Liberty Batch: Creating a Batch Pipeline with Split/Flow and Messaging

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102784** under the category of "White Papers"

Version Date: February 26, 2019
See "Document change history" on page 17 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com
Don Bagwell
IBM Advanced Technical Sales
301-240-3016
dbagwell@us.ibm.com

Many thanks go to Scott Kurz

Contents

Introduction

Batch jobs often have multiple steps. Those steps are sometimes dependent on each other. A later step may need the results of a prior step in order to do its processing. How do the results of one step make it to the second step?

There are a lot of possibilities. You can write to a file, insert rows into a database, or even just pass them in-memory if the programming model allows it. One consideration is whether you actually need the results of the first step or if it is just a temporary thing only needed by the second step and is then discarded.

Another consideration is how failures and a restart of the job should be handled. Will the job just start again at the beginning or do you want to want to pick up more or less where the failure occurred (at a checkpoint)? If you want to restart at a checkpoint, then the intermediate results will need to be hardened somewhere transactional.

To take this one step further, perhaps the second step doesn't really need the first step to completely process all of its input data but could handle it as it happens. That is, if the first step processes a record and produces a result, the second step could do its processing on that result as it happens rather than wait for the first step to process all the records and produce a complete set of results for the second step to process.

If that's the case, then wouldn't it be nice if there was a way to run the first step and the second step concurrently and have the results of the first step feed directly into the second step, cutting down on the total elapsed time for the complete job, while maintaining transactional integrity so a failure in either step stops the job in a way that it can be restarted at a checkpoint?

This isn't a new concept. Various implementations, such as z/OS BatchPipes, have been around for years. This paper explores an implementation using the JSR-352 Java Batch programming model. We'll start by reviewing the basic concepts involved, then go through a sample implementation and add some scaffolding to create a sample that can actually run (although it doesn't really do anything).

Attached to this Techdoc entry you'll find all the code referred to in the paper, along with some sample Liberty server configurations used to run the job on z/OS with IBM MQ as the messaging engine.

Chunk Step Basics

The approach we'll take to implementing our "pipeline" uses the JSR-352 programming model, we'll start out by reviewing some important aspects of the specification, starting with the Chunk Step model.

A Chunk Step is one of two step types defined in the specification and is intended for iterative processing over a set of input 'records'. A Chunk Step application is broken into three separate interfaces: `ItemReader`, `ItemProcessor`, and `ItemWriter`.

The application's implementations of these interfaces are called in a loop allowing data to be read, processed, and results written. This loop occurs within a transaction created and managed by the batch container. The transaction is committed at intervals which can be

Version Date: Tuesday, February 26, 2019

defined by a count of items read, by elapsed time, or programmatically by the application at runtime. Committing the transaction commits any transactional processing done by the application (i.e. records inserted into a database by the `ItemWriter`) as well as checkpoint data provided by the `ItemReader` and `ItemWriter` into the batch container managed Job Repository tables.

Committing the checkpoint data along with application updates creates a restart point from which the batch job can pick up in the event of a failure and restart of the job. Thus, the `ItemReader` might provide checkpoint data such as the record number last successfully read and on a restart will position itself to start reading at the next record after that.

The `ItemReader` and `ItemProcessor` (which is actually optional) are called in a loop until the checkpoint interval is reached. At that time the `ItemWriter` is called with a list of the results of the processing. This allows the `ItemWriter` to potentially optimize its processing by doing a bulk-insert of rows or something similar depending on the target it is "writing" to.

As we'll see, our pipeline job will have two Chunk Steps with the `ItemWriter` of one step feeding the `ItemReader` of another.

Understanding Split/Flow processing

The other concept from JSR-352 we need to understand is a split/flow. To have a pipeline where two steps are processing concurrently, we need a way to actually have two steps running at one time. The split/flow is the way to do this in JSR-352.

We begin with a flow. A flow is a set of one or more steps, grouped together by wrapping `<flow>` tags around them. A flow has its own identifier and can be the target of flow control within the job. You can define groups of steps as a flow anywhere you like, but flow control from within a flow cannot branch out of the flow.

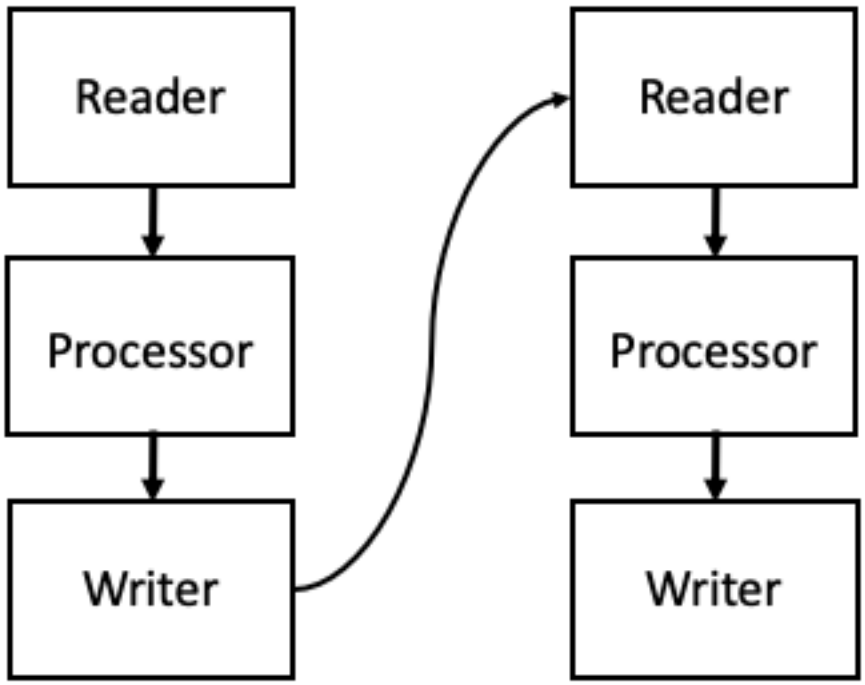
A split is created by wrapping `<split>` tags around a set of two or more flows. When flow control reaches a split, work is spun off to other threads for each flow within the split. Processing on the thread where the split was encountered waits until all the flows complete before continuing.

Presentations of split/flow capabilities typically show a split with two flows because that fits nicely on a chart, but a split can actually contain any number of flows (well, ok there's probably a limit somewhere). A flow can also contain another split with its own flows providing concurrency within already-concurrent processing.

As we'll see, our pipeline job will use a split/flow construct to achieve concurrency with the output of one flow acting as input to another.

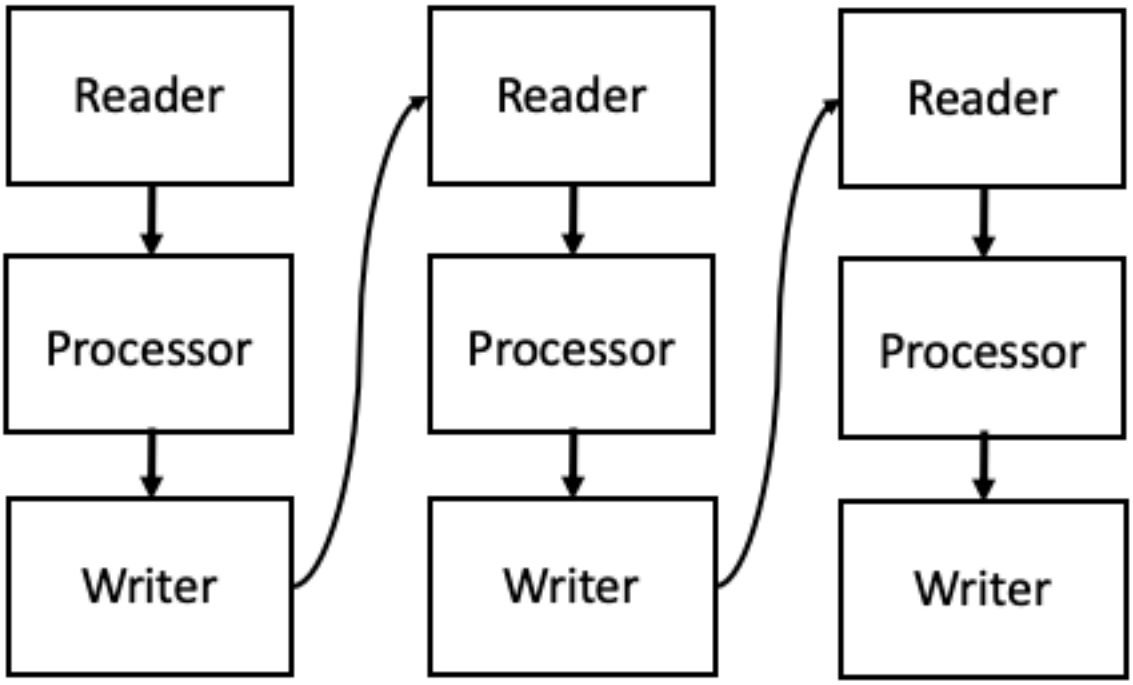
The Big Idea

Our goal is to have multiple steps running concurrently with one step reading from some data source, doing some processing, and feeding its results to another step which uses those results as input to its own processing and writing its results to another target. Here's a picture which might help:

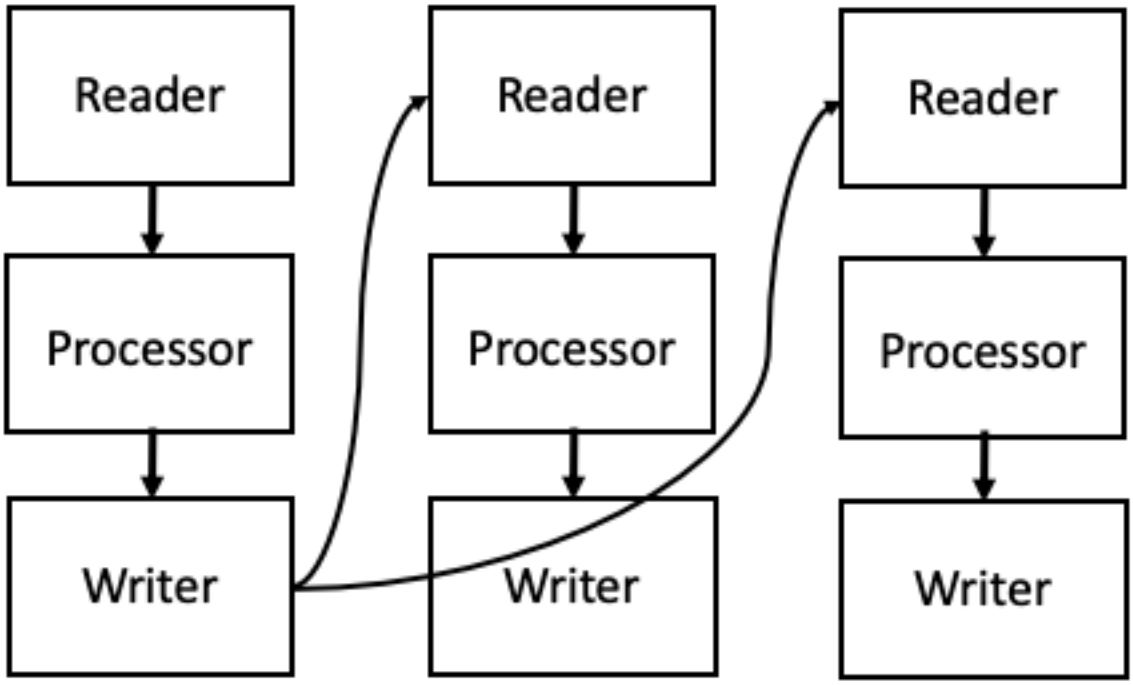


The mechanism we will be using to pass data from the Writer to the Reader is a Transactional Message Queue. Remember that the Writer gets control at each checkpoint with the results of the processing, as part of the chunk transaction. Our Writer will insert one message for each record in the list of processing result objects it receives. When the transaction is committed those messages will become available for the Reader in the second step to process.

If there is a third step that needs the results of the second step, we can easily extend this model to chain things together with all three (or more!) steps processing concurrently as data flows through the job. Here's another picture:



Finally, we might extend the capabilities of our message-producing writer to be able to put the same message on multiple queues. This would allow a model where the results of one step are acting as input to multiple concurrent other steps. An example of this might be an application where the first step reads in data from some source and normalizes it. That normalized data is then fed to multiple other steps which perform different analysis on the data. Something like this:



And, of course, you could mix and match these models creating a complex arrangement of steps producing and consuming data in parallel.

Sounds great. Let's have a look at how you might actually do this.

JMSObjectWriter – In Detail

Note: I'm going to assume you have the source code for these parts handy and can easily look through it while reading this. Which means I'm not going to try to paste code fragments into this paper. The purpose of the paper is to explain the concepts and issues to be considered in creating an implementation of this sort of pipeline processing and not dwell on the specific Java syntax.

An `ItemWriter` needs to implement four methods: `open`, `close`, `writeItems`, and `checkpointInfo`. In the `open` method the writer normally establishes where it is going to write results. It might open a connection to a database or open a file or something like that. In our case we're going to need to connect to a queue to put messages on when `writeitems` is invoked. However, things are a bit tricky when transactions are considered.

The `open` (and `close`) methods are called inside a separate transaction. If committing that transaction closes the connection (as appears to be the case with a JMS connection to MQ) then a connection established in the open method won't be there when we are invoked in `writeItems`.

For this reason, the implementation of open in `JMSObjectWriter` parses through the injected JSON string to find the connection factory and queue name of all the target queues. The parsing supports an array of targets enabling the writer to write output to multiple queues as described above (second picture).

The targets are encapsulated in `JMSTarget` objects (discussed later) and kept in a `LinkedList`. This list is set into the transient user data for the step which makes it accessible to other parts of this step (specifically the `ChunkListener` also discussed later). Transient user data is part of the `StepContext` which is accessible via injection to the application. Each step has its own context, so our various concurrent steps each have their own transient user data, so they don't collide. Transient user data is, obviously, transient and thus won't be preserved in a failure/restart case and will have to be recreated.

Once open has done all this, when we are called to `writeItems` we iterate over the targets, establishing connections to each queue in turn, and put a message on each queue for each object we need to write.

Another tricky part comes when the step is done. The reader will indicate it is out of data to process by returning a null. We need to have a way to tell the concurrent step that is reading the messages we are writing that there will be no more messages. To do that we will define a special message that indicates we are 'done' and the reader will need to recognize and handle that. The `ItemWriter` knows the step is done when it receives control in the close method. Thus, our implementation of close iterates over our target queues and puts a special 'done' message on each queue.

Finally, the `checkpointInfo` method returns nothing. The writer has no checkpoint information to keep track of. We are relying on the transactional nature of our messaging engine to keep any messages that were committed and there is nothing further the writer needs to remember.

The sample also does a lot of logging to note its progress which you wouldn't want in a real application, but it makes it easier to see what is happening when you run the sample.

Also note that at no point does the `JMSObjectWriter` know anything about the objects it is writing. There is a requirement that it be `Serializable` in order to be placed in the message. The logging assumes that `toString` on the object will produce something useful, but that's not important if you remove the logging.

JMSObjectReader – In Detail

At the other end of our message pipe is an implementation of `ItemReader` which is consuming the messages put there by the `JMSObjectWriter`. The reader receives the connection factory and queue name to use to access these messages through an injected JSON string. This is the same as the JSON string used to configure targets for the writer, although the reader only needs one instead of the array of targets the writer supports.

As in the writer, the `open` method parses the injected JSON string and looks up the specified connection factory and queue. Also, as in the writer, the reader has transactional issues with the connection/session to the queue and thus doesn't actually establish those during `open` processing since the commit of the transaction around `open` apparently closes them.

Most of the interesting code in our reader is in the `readItem` implementation. It needs to establish a connection to the queue to receive messages from it, but it doesn't need to do that every time. The connection will be closed when the chunk transaction commits at each checkpoint. Therefore, we only need to re-establish the connection at the start of each new chunk.

For this reason, a flag (`sessionCreated`) is defined. The flag is initially `false` which causes the first call to `readItem` to establish the connection. The flag is set to `true` after the connection is established. This allows us to skip creating the connection on subsequent reads until a checkpoint when the `checkpointInfo` method gets control. We set the flag to `false` in that method which will force us to create a new connection when `readItem` gets control for the first item in the next chunk.

After that, `readItem` uses the JMS APIs to wait for a message. This introduces another special consideration. The JMS `receive` method allows the specification of a timeout after which it will give up and return without a message. It is probably a good idea to specify some value for this timeout, but what value to specify? Conceptually our producer and consumer steps are running concurrently, but in practice there is no guarantee that the producer step will get running before (or at the same time) as the consumer. The consumer step could get assigned to a thread before one is available for the producing step. And remember that the producer chunk step must reach a checkpoint before the `ItemWriter` gets control in `writeItems`. Even if both steps start at the same time, some time will pass before the producer step reaches a checkpoint and writes messages. The

sample sets a value of 10 seconds (10,000ms) which worked well enough in a test environment. This would need to be adjusted in a real application in a production environment. The sample also doesn't handle a timeout (no message) very well.

Finally, our `readItem` processing must recognize that special 'done' message and return a null to inform this Chunk Step that processing is complete.

And, again, there is some logging in here to make the output from the sample more interesting. Most, if not all, of that would be removed in a real application.

Supporting Characters – JMSTarget

Because the `JMSObjectWriter` can put messages to multiple queues the operations and information around each queue are encapsulated in a `JMSTarget` object. The reader only needs one queue and the operations are slightly different, so it seemed easier to leave all the JMS interactions in the reader itself.

There's nothing very interesting in here, just interactions with the JMS APIs. A little less logging and a little better error handling might be nice.

Supporting Characters – ObjectMessageContent

As discussed earlier, we need a special message to indicate this is the final message so the reader/consumer knows when we are done. The sample manages this by defining an object to wrap the actual message, adding a `Boolean` 'done' flag to the content. The actual payload object and this wrapping class must be serializable to be put on a queue.

Obviously, it is important that both the writer/producer and reader/consumer use the same version of this serializable object.

Error Handling - JMSObjectWriterChunkListener

What if something bad happens to the writer/producer step? That step will fail, but the reader/consumer is still waiting for more messages. Depending on the value we picked for a read-timeout and how we handled it, the job might just end nicely that way. But it would be a bit more elegant to try to handle it ourselves.

To do that there is an implementation of the `ChunkListener` to be used by a message-producing thread. A `ChunkListener` gets control before and after each chunk, which we really don't care about here. It also gets control for any error (exception) thrown by the chunk processing.

Our implementation assumes any exception that makes it to the `onError` method of the `ChunkListener` is fatal for the step and we need to tidy up.

To do that, we retrieve the list of `JMSTarget` objects from the `StepContext's` transient user data. Then, much like in `close` processing for the `JMSObjectWriter`, we iterate over all the targets and send a 'done' message to let the reader/consumers know that there won't be any more messages.

Version Date: Tuesday, February 26, 2019

For the sample we don't really care that the job is ending badly, but you could easily add a separate flag in the `ObjectMessageContent` object to let the reader know if it was important to something being done on the reader/consumer side.

Error Handling - JMSObjectReaderChunkListener

Error handling on the reader/consumer side is more interesting. If the reader fails in some way, there is no particular reason we have to inform the writer/producer. If we don't then presumably it will continue merrily on its way putting messages on the queue until it runs out of data to process. In a restart of the job the writer/producer thread will have nothing to do and the reader/consumer will immediately have a lot of messages to process.

But that eliminates all the advantages we are getting out of running all this concurrent. It might be nice to get the job to stop processing now so we can fix whatever is wrong and get it restarted.

To that end, the sample includes a second `ChunkListener` implementation for use by the reader/consumer step.

Unlike the `ChunkListener` for the writer/producer we don't really have a way to communicate back to the other thread (no magic 'done' message to put anywhere). What we really want to do is just stop the job. Fortunately, that's pretty easy to do.

The batch specification allows an application access to the `JobOperator` API which includes a method to stop a running job. That API requires the caller to specify the job instance identifier for the job to be stopped. Fortunately, that's easily accessible using the `JobContext` API.

When a job is stopped, the batch container running a chunk step for that job will notice at the end of a pass through the read/process loop and cease processing for the step. That should prevent any more messages from being placed on the queue (that aren't already there when the reader/consumer thread fails).

Of course, any other steps in other flows running concurrently will also notice and be stopped which shuts the whole job down quite nicely and positions it to be restarted.

A Running Sample – DummyReader and DummyWriter

To create a real running sample of this process our writer/producer step needs an `ItemReader` to provide data to the `JMSObjectWriter` and our reader/consumer step needs an `ItemWriter` to receive the data from the queue retrieved by the `JMSObjectReader`.

Obviously, a real application would do some real things here. We just need a dummy reader and writer to do something.

The `DummyReader` provided with the sample simply generates increasing `Integer` values and provides those objects to the `JMSObjectWriter`. An injected property controls how many such `Integers` are created.

To assist in experimenting with the sample, the `DummyReader` also supports an injected property that can cause an exception to be thrown when a particular integer value is

generated. This helps explore how error handling is managed. Another property injects a configurable delay in the reader to simulate actual delays reading real data.

The `DummyWriter` receives the `Integer` values that have made their way through the message queue and simply logs the value it receives. An injected property allows you to force an exception to be thrown when a specific integer value is received, again to help simulate error conditions.

A Running Sample – The JSL

The sample JSL, `Pipeline2.xml`, provided with the sample code consists of three flows within the split that makes up the entire job. The first flow (`Flow1`) is the message producer. It uses the `DummyReader` to generate integer values. The number of values defaults to 25 but can be overridden by specifying a job parameter named `totalRecords`.

The writer, implemented by `JMSObjectWriter`, is configured in the JSL to put messages on two queues whose connection factory and queue names are hard coded in the JSL. These could be made into values that can be specified as job parameters.

The second and third flows both use the `JMSObjectReader` to read the messages, each from one of the two queues where messages are placed by the first flow. The object encapsulated in the message is passed to the `DummyWriter` which logs them. Again, the connection factory and queue names used by each flow are hard coded in the JSL. Even if these values aren't made into job parameter values, it might be better to at least set them as job properties at the top of the JSL. This helps avoids problems caused by errors specifying the queues incorrectly.

Finally, note that the checkpoint intervals for the Chunk Steps in the three flows are different. This shows up clearly in the output logging to help distinguish the different threads.

A Running Sample – Server Configuration

To run the sample, I began with the multi-server configuration created for the Liberty Java Batch Workshop described in IBM Techdocs PRS5365 (http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/PRS5365). This includes configuration in the executor server to access an IBM MQ messaging engine.

To make the application work, I updated the configuration for the executor (JSREXEC1) to include two more features: jms-2.0 (for the JMS APIs used by the application) and jsonp-1.0 (to parse the JSON property strings).

I also added a definition for the connection factory. These values work for the lab used for the Batch Workshop. The specifics of your configuration will vary from mine, but it is important that the `jndiName` match the value provided in the JSL.

```
<jmsQueueConnectionFactory id="batchDataConnectionFactory"
   jndiName="jms/batchData/connectionFactory">
   <properties.wmqJms
     hostName="wg31.washington.ibm.com"
     transportType="CLIENT"
```

```
      channel="SYSTEM.DEF.SVRCONN"
      port="1414"
      queueManager="MQS1">
    </properties.wmqJms>
  </jmsQueueConnectionFactory>
```

I also added definitions of the two queues used by the job. Again, the `jndiNames` have to match the values specified in the JSL. The actual queue names and queue manager name should match values in your MQ configuration.

```
<jmsQueue id="batchPipelineQueue1"
  jndiName="jms/batch/pipelineQueue1">
  <properties.wmqJms baseQueueName="JAVA.BATCH.DAVE.QUEUE"
     baseQueueManagerName="MQS1">
  </properties.wmqJms>
</jmsQueue>

<jmsQueue id="batchPipelineQueue2"
  jndiName="jms/batch/pipelineQueue2">
  <properties.wmqJms baseQueueName="JAVA.BATCH.DON.QUEUE"
     baseQueueManagerName="MQS1">
  </properties.wmqJms>
</jmsQueue>
```

A Running Sample – A Look at the Output

To wrap things up, let's take a look at some output from a run of the sample. To start, we'll look at the output from the job itself.

```
CWWKY0009I: Job Pipeline2 started for job instance 4 and job execution 4.
CWWKY0015I: Flow Flow1 started for job instance 4 and job execution 4.
CWWKY0015I: Flow Flow2 started for job instance 4 and job execution 4.
CWWKY0015I: Flow Flow3 started for job instance 4 and job execution 4.
CWWKY0016I: Flow Flow1 ended for job instance 4 and job execution 4.
CWWKY0016I: Flow Flow2 ended for job instance 4 and job execution 4.
CWWKY0016I: Flow Flow3 ended for job instance 4 and job execution 4.
CWWKY0010I: Job Pipeline2 ended with batch status COMPLETED and exit status
COMPLETED for job instance 4 and job execution 4.
```

This makes clear that the main thread of the job doesn't really do anything at all. It launches the three flows and waits for them to complete.

Next, we'll have a look at the output from the first flow which is generating increasing integer values and putting messages on two queues. By default, the step produces 25 values and puts them on two queues, so there is quite a lot of output. We'll just look at the

first chunk's worth of processing here. Remember that the checkpoint interval for the first flow is five values.

```
CWWKY0018I: Step Step1 started for job instance 4 and job execution 4.
JMSTarget opening for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget opening for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMS Writer writing data
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent 1 for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent 2 for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent 3 for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent 4 for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent 5 for jms/batchData/connectionFactory jms/batch/pipelineQueue1
JMSTarget Session jms/batch/pipelineQueue1 closing
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent 1 for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent 2 for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent 3 for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent 4 for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent 5 for jms/batchData/connectionFactory jms/batch/pipelineQueue2
JMSTarget Session jms/batch/pipelineQueue2 closing
```

As you can see, the writer creates JMSTarget objects for both queues when the writer is opened. Then, at the checkpoint, the writer is called to write five objects. It first creates a connection to the first queue (Queue1) and writes all five objects. Then it creates a connection to the second queue (Queue2) and writes the same five objects to that queue.

Now let's move on to the second flow which is reading from the first queue. Again, we'll just show the first chunk's worth of processing. Remember that the checkpoint interval for the second flow is ten values.

```
CWWKY0018I: Step Step2 started for job instance 4 and job execution 4.
JMS Reader Open complete
JMS Reader Session created
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - Reading Message
Handling record 1
Handling record 2
```

```
Handling record 3
Handling record 4
Handling record 5
Handling record 6
Handling record 7
Handling record 8
Handling record 9
Handling record 10
JMS Reader Session closed
JMS Reader at checkpoint
```

And then on to the third flow which is reading messages from the second queue and checkpointing every two values. We'll show a few chunks worth of data since each one is so short.

```
CWWKY0018I: Step Step3 started for job instance 4 and job execution 4.
JMS Reader Open complete
JMS Reader Session created
JMS Reader - Reading Message
JMS Reader - Reading Message
Handling record 1
Handling record 2
JMS Reader Session closed
JMS Reader at checkpoint
JMS Reader Session created
JMS Reader - Reading Message
JMS Reader - Reading Message
Handling record 3
Handling record 4
JMS Reader Session closed
JMS Reader at checkpoint
JMS Reader Session created
JMS Reader - Reading Message
JMS Reader - Reading Message
Handling record 5
Handling record 6
JMS Reader Session closed
JMS Reader at checkpoint
```

Finally, we'll look at the end of processing in the first flow where the DummyReader has returned a null and close processing for the ItemWriter places a 'done' object on the queue.

```
JMSTarget sending to jms/batch/pipelineQueue1
JMS Writer sent end of data for jms/batchData/connectionFactory
jms/batch/pipelineQueue1
JMSTarget Session jms/batch/pipelineQueue1 closing
JMSTarget sending to jms/batch/pipelineQueue2
JMS Writer sent end of data for jms/batchData/connectionFactory
jms/batch/pipelineQueue2
JMSTarget Session jms/batch/pipelineQueue2 closing
JMS Writer Close complete
CWWKY0020I: Step Step1 ended with batch status COMPLETED and exit status
COMPLETED for job instance 4 and job execution 4.
CWWKY0010I: Job Pipeline2 ended with batch status COMPLETED and exit status
COMPLETED for job instance 4 and job execution 4.
```

And here is the output from the third flow catching the 25th message followed by the ‘done’ message.

```
JMS Reader Session created
JMS Reader - Reading Message
JMS Reader - Reading Message
JMS Reader - We're done
Handling record 25
JMS Reader Session closed
JMS Reader at checkpoint
CWWKY0020I: Step Step3 ended with batch status COMPLETED and exit status
COMPLETED for job instance 4 and job execution 4.
CWWKY0010I: Job Pipeline2 ended with batch status COMPLETED and exit status
COMPLETED for job instance 4 and job execution 4.
```

Conclusion

While clearly just intended as a sample, the pipelining technique shown here is a powerful way to reduce elapsed time for job processing by exploiting the concurrency capabilities that are part of the JSR-352 programming model.

Document change history

Check the date in the footer of the document for the version of the document.

February 19, 2019	Initial Version
February 26, 2019	Add document number

End of WP102784