**WebSphere Application Server**

# WebSphere Liberty Batch – Monitoring a Chunk Step

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102780** under the category of "White Papers"

*Version Date*: December 4, 2018
See "Document change history" on page 9 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, Don Bagwell, and the various customers who commented/tried this out..

# Contents

# Introduction

So you've got a Java Batch job and it is taking longer to run than you think it should. What to do? Well, first have a look at the job log and find the start and end messages for each step and look at the timestamps. The whole job might be slow, but there's probably one step that catches your eye as being the source of the long-running job problem.

If that step is a batchlet, well, there's not much you can do except open the application up and have a look inside to see what is going on in there. You might have to insert some debug/print statements or something else depending on what is going on. But, if the step is a chunk step, there is a way to get more information without having to touch the application itself.

A chunk step supports a whole set of Listener interfaces that can get control at various points in the processing of a chunk step. Remember that a chunk step basically does read-item/process-item in a loop until it reaches a checkpoint and then writes the results, committing those results and any checkpoint data. Implementing the listeners gives you the ability to "watch" the reads, processing, and writes take place and observe how long each one takes.

This paper describes a sample of a Java class that implements the Reader, Processor, Writer, Chunk, and Step listeners. The class collects time (and maybe CPU) data along the way and reports it in a comma-separated-value format so you can do analysis with your favorite spreadsheet application.

I'm not going to go into the details of the code here. There are plenty of comments (or at least I think so : - ) in the code itself for that. This paper is intended to be more of a User's Guide.

The sample itself is located here:

https://github.com/WASdev/sample.batch.misc

# Configuration

JSR-352 supports a lot of different Listener interfaces, but in the JSL where you specify a listener to be invoked, you just tell it the package and class of the listener. The batch container figures out at runtime which Listener interfaces the class implements and drives the right methods at the right time. That's pretty nice because if you have a listener that implements several interfaces you don't have to list it in the JSL for each one. That's really nice for our sample because it implements five different Listener interfaces. And because a listener can be implemented more than once, adding this listener won't interfere with any application implementations of the same listeners.

To start out then, all we need to do is find the JSL for the chunk step we want to gather data about and add this block inside the `<step>` block.

```
<listeners>
   <listener ref=”com.ibm.websphere.samples.batch.artifacts.ChunkTimeListener”>
</listeners>
```

That will do it, but the listener supports a few parameters you should specify. The parameters are:

| Parameter | Comments |
|---|---|
| outputDir | This is the absolute path where you want the two output files to be produced. If not specified, no files are produced. |
| writeAt | Either “ChunkEnd” or “StepEnd” depending when you want data written |
| logToJoblog | Set to true or false to indicate if information should be written to the joblog, possibly in addition to the files produced in the outputDir |

The intended approach was that you would specify an outputDir and the results are produced in ChunkData and IterationData comma separated value files. The file names will contain prefixes include the job name, execution id, and step name to avoid collisions with other jobs or previous runs of this job. A review of the contents of the files are in a later section of this paper.

You can choose to have data written at the end of each chunk or at the end of the step. Waiting until the end of the step avoids the overhead of writing the data as the step progresses, but could produce a lot of in-memory data that could impact heap usage in a bad way. ChunkEnd is probably preferable.

When developing the listener, we first had the output going to the job log in a human-readable format. After adding the .csv file support, we decided to keep the joblog output as an option in case it was easier to understand the data if it was mixed in with other messages logged by the application itself.

Finally, it might be helpful to externalize these settings as Job Parameters that can be set by the submitter of the job. A sample is shown below.

```
<listeners>
   <listener ref="com.ibm.websphere.samples.batch.artifacts.ChunkTimeListener">
          <properties >
          <property name="outputDir" value="#{jobParameters['outputDir']}"/>
          <property name="writeAt" value="#{jobParameters['writeAt']}?:ChunkEnd;"/>
          <property name="logToJoblog" value="#{jobParameters['logToJoblog']}?:false;"/>
          </properties>
   </listener>
</listeners>
```

## The Chunk Output

The first file produced by the listener is a summary for each chunk processed. The output can be in two forms, depending on whether CPU data is available. If it isn't, those columns are left out. Here's a sample of a few rows of the table without CPU data:

| Chunk | WriteTMillis | ChunkTMillis | Reads | ReadErr | Processes | ProcessErr |
|---|---|---|---|---|---|---|
| 1 | 90 | 705 | 5 | 0 | 5 | 0 |
| 2 | 98 | 800 | 5 | 0 | 5 | 0 |
| 3 | 53 | 756 | 5 | 0 | 5 | 0 |
| 4 | 64 | 774 | 5 | 0 | 5 | 0 |
| 5 | 85 | 748 | 5 | 0 | 5 | 0 |

The table shows data for the first five chunks processed by a sample run of some chunk step. You can see that the 'write' operation for each chunk took between 50 and 98 milliseconds. The overall time for the chunk ranged between 700 and 800 milliseconds. Each chunk read and processed five records with no exceptions from the reader or processor.

Loading this data into a spreadsheet (with a lot more rows from a real application) will allow you to easily average and find min and max values.

It is important to note that use of advanced error handling features like skip, retry, and retry-rollback will affect how this data looks. Non-zero values in the error columns combined with a knowledge of how the job handles errors will be required to understand the data. But presumably if you are tuning performance for an application, you are less worried about the error cases than the normal mainline path. The error counts are mostly here to allow you to know something odd happened in this chunk.

If CPU data is available, the table will contain extra columns. Here's a sample:

| Chunk | WriteTMillis | WriteCPUMicro | ChunkTMillis | ChunkCPUMicro | Reads | ReadErr | Processes | ProcessErr |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 11 | 5 | 1437 | 5 | 0 | 5 | 0 |
| 2 | 2 | 14 | 10 | 1473 | 5 | 0 | 5 | 0 |
| 3 | 0 | 4 | 9 | 1534 | 5 | 0 | 5 | 0 |
| 4 | 1 | 11 | 7 | 1449 | 5 | 0 | 5 | 0 |

Note that write and total chunk CPU times are in microseconds while elapsed times are in milliseconds.

## The Iteration Output

The Iteration report produces information about each pass through the read/process cycle within each chunk. As noted under the Chunk Output, errors and error handling might complicate this picture.

We'll start with sample output that does not include CPU data:

| Chunk | Iter | ReadTMillis | ProcessTMillis |
|---|---|---|---|
| 1 | 1 | 56 | 59 |
| 1 | 2 | 37 | 55 |
| 1 | 3 | 55 | 93 |
| 1 | 4 | 98 | 32 |
| 1 | 5 | 80 | 50 |
| 2 | 1 | 66 | 71 |
| 2 | 2 | 37 | 51 |
| 2 | 3 | 66 | 98 |
| 2 | 4 | 95 | 28 |
| 2 | 5 | 91 | 99 |

The chunk numbers will match the chunk numbers seen in the Chunk Output. If a particular chunk took extra long, you can find the details in the Iteration Output and see where the time was spent.

The iteration numbers count each pass through the read/process loop and you can see the counts of five passes match the read/process counts from the Chunk Output. The last two columns provide the elapsed time, in milliseconds, spent in the reader and processor.

And here is some sample data that includes CPU time:

| Chunk | Iter | ReadTMillis | ReadCPUmicro | ProcessTMillis | ProcessCPUmicro |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 5 | 0 | 4 |
| 1 | 2 | 0 | 3 | 1 | 10 |
| 1 | 3 | 0 | 3 | 0 | 1 |
| 1 | 4 | 0 | 2 | 0 | 1 |
| 1 | 5 | 0 | 3 | 2 | 14 |
| 2 | 1 | 1 | 11 | 0 | 2 |
| 2 | 2 | 2 | 12 | 0 | 2 |
| 2 | 3 | 0 | 3 | 0 | 1 |
| 2 | 4 | 1 | 12 | 2 | 11 |
| 2 | 5 | 0 | 3 | 1 | 10 |

## Conclusion

The Chunk Time Listener is a handy way to add some metrics into a chunk step without having to touch the application itself. The data, perhaps combined with a bit of logging from the application about what was going on in each iteration, might make help identify performance bottlenecks that are dramatically adding to the execution cost of a chunk step.



*Version Date:* Tuesday, December 04, 2018

# Document change history

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

| | |
|---|---|
| *December 3, 2018* | Initial Version |
| *December 4, 2018* | Add document number |

**End of WP102780**