WebSphere Application Server

WebSphere Liberty Batch: Thoughts on Performance

This document can be found on the web at:
www.ibm.com/support/pages/node/6398358

Version Date: January 6, 2021
See "Document change history" on page 31 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

Many thanks go to Beena Hotchandani for her patience...

Contents

Introduction 5
The Application 5
  The Reader 5
  The Processor 5
  The Writer 6
  Listeners 6
Establishing a Baseline 6
  How Many Items? 7
  Where to Start with Item Count? 7
  Starting Values for Checkpoint Data and Processing Results 7
  What are the Numbers for our Baseline? 7
  Can We Enable the Chunk Listener Monitor? 7
Playing with Checkpoint Intervals 8
  The Raw Results 9
  Looking at the Read Time 10
  Looking at the Writer 11
  Bulk Inserts vs One by One Deletes 13
  Container Overhead 16
  Conclusions 19
  Sidebar: Creating the graphs 20
Playing with Checkpoint Info Sizes 21
Playing with Write Object Size 22
  Changing Our Application 22
  The Results 24
Experimenting with Partitions 27
Enabling Batch Events 29
A Few More Thoughts 29
Conclusions 30
Document change history 31

Tables and Figures

Figure 1 - Step Elapsed Time for Various Item Count Values 10
Figure 2 - Writer Time for Various Item Count Values 12
Figure 3 - Insert Time for Various Item Count Values 14
Figure 4 - Delete Time for Various Item Count Values 15
Figure 5 - Inserts and Deletes as Components of Total Step Time 16
Figure 6 - Container Time for Various Item Count Values 17
Figure 7 - Number of Checkpoints vs Item Count 18
Figure 8 - Components of Elapsed Time for Various Item Count Values 19
Figure 9 - Container Time for Various Checkpoint Data Sizes 22

Table 1 - Step Elapsed Time for Different Item Count Values 9
Table 2 - Writer Time for Various Item Count Values 11
Table 3 - Insert Time for Various Item Count Values 13
Table 4 - Delete Time for Various Item Count Values 14
Table 5 - Container Time for Various Item Count Values 16
Table 6 - Container Time for Variously Sized Checkpoint Data 21

Introduction

Unlike OLTP, there don't seem to be any standard benchmarks for batch applications. There are batch applications that get used to measure the performance of other things (like z/OS pervasive encryption), but no standard batch metrics. But that's ok because what I really wanted to write about was how design decisions you might make when creating a JSR-352 Liberty Batch application affect the performance of the job. In this paper we'll start by looking at the batch application I used for measurements. Then we'll start changing some parameters to affect how it behaves and crawl around in the resulting performance differences.

The Application

The goal was to create a single step batch job that behaved like a real application but wasn't actually. We wanted it to be extremely consistent in its behavior so that we could accurately determine the effect of changing different aspects of how it ran.

To that end, we decided on a job with a chunk step which read data from a flat file, did no processing (more on that in a bit), and then interacted with a database in the writer.

The Reader

Most normal applications that read data from a file will read until the file ends. In our case we wanted to be able to vary how much data was read (how many records were processed) without having to fuss around with changing the contents of the input file.

Therefore, the reader takes an input parameter called `NumRecords` that controls how many records it will read. The parameter `FileName` points us to the file to read from. The reader will read from the file until it hits the required number of records or the end of the file. If the end of the file comes first, the reader closes and re-opens the file to start over. Thus, we didn't need different files in different sizes to control the number of records. We did need a file big enough that we wouldn't be opening and closing it all the time.

Finally, the reader has an input parameter called `ReaderCheckpointDataSize` which controls the amount of data provided on the checkpointInfo method. This data has to be serialized and written into the Job Repository. Our application won't restart in any of our runs but handling checkpoint data that is excessively large might slow things down.

The Processor

My original thought was to have a processor that just did some fixed amount of processing to emulate a real application doing whatever it does with the data we read. However, as I started doing some runs and analyzing the data, I realized that I could just subtract out the processing time because it wasn't interesting – it was the same for every run.

That led me to the conclusion that our measurements should use a job that didn't have a processor element in the chunk. Obviously many normal applications would, but that's a factor in the performance of the application itself and not an interesting part of how the JSR-352 implementation in Liberty manages the step.

Which brings us to the first thing to consider if you're looking at this paper to help improve the performance of your own applications: consider the cost of processing separately. Unless it somehow has contention issues (maybe running as a partitioned step?) you should be able to measure and analyze your processor on its own.

The Writer

For the writer we wanted to access a database to emulate doing some database work using the outcome of the processing. Of course, in our case there is no processor and so it isn't producing any results. But the writer knows the number of records processed in each chunk (because we pass it in as a parameter), so it knows how many things it would be expected to write.

To keep things simple, the writer then just fabricates enough things to write and inserts them into the database. The objects being written are the same as we use in the BonusPayout sample Java Batch application.

However, as the job runs, we can't just keep re-inserting the same rows over and over again because primary keys will collide, and the inserts will fail. We could generate unique keys, but as the job runs data will accumulate in the table and have to be cleared out between runs.

To avoid that problem the writer alternates between writing new rows and deleting the rows it added last time. Essentially the job inserts rows in one chunk, deletes those rows in the next chunk, then adds them back into the table in the third chunk, and so forth on and on until the job ends.

The inserts are done as a bulk (or batch) insert, while the deletes are done row-by-row using the primary key values we know we inserted in the last chunk.

Doing alternating insert/deletes gets us more than just not having to clean up the database after each run (which I was definitely going to forget to do). It allows us to compare bulk record processing vs one-by one processing.

The writer also has a variable called `WriterCheckpointDataSize` to control the amount of checkpoint data it provides.

And finally, the variable `AccountNumberBase` controls the starting index for the primary key values inserted by the writer. Being able to set different values for this will let us avoid collisions if we use this writer in a partitioned step.

Listeners

The WP102780 whitepaper (now IBM support document 6355729) describes a sample Java program that can be included in a Java Batch job to gather information about elapsed time and CPU usage for the reader, processor, and writer. This program implements the reader, processor, writer, chunk, and step listeners and produces comma-separated-value (CSV) files.

Having this extra information will let us look inside the execution of the job to see where we are spending our time (and CPU). However, before we can use it, we need to convince ourselves that enabling this monitoring won't have a serious adverse effect on the elapsed or CPU time for the job.

Establishing a Baseline

Before we could start fiddling around with the knobs we were interested in, we needed a baseline configuration to compare against. We wanted to have the job run long enough that differences would be visible. If a job that runs for 30 seconds now runs for 31, is that interesting? If a job that runs for 30 minutes now runs for 31, is that interesting?

How Many Items?

After some experimentation we settled on a job that processed 10 million records. Using fewer record showed some differences in different configurations, but the more records we processed the more exaggerated the effects were. However, enabling the Chunk Listener produces one line of output for each record processed and the .csv files produced were getting kind of large. Ten million records produced a file around 170Mb which was about as big as we wanted to deal with.

Where to Start with Item Count?

The Item Count will control how frequently we take a checkpoint. It will affect the number of objects that accumulate in memory to be passed to the Writer at the end of the chunk. That, in turn, will affect how many objects are being written (or deleted) by the writer. As the Item Count is increased, our job will take fewer, larger checkpoints. As the count decreases the job will take more, smaller checkpoints.

We want to start with some "middle-ish" value that we can range both up and down to see what the consequences are. We decided to start with a chunk size of 1000 with an intent to lower that down until we are checkpointing every 25 items (cutting it roughly in half each step) and raise it up by doubling to 2,000 then 4,000. Above that we increased by 1,000 for each test until something interesting happened.

Starting Values for Checkpoint Data and Processing Results

We're going to perform experiments with different checkpoint data sizes. To start with we'll perform runs with both the reader and writer returning 1024 bytes of checkpoint data. This feels like it might be a bit larger than is usually necessary but could easily happen if the checkpoint data object ends up with some other odds and ends included. We'll experiment with reducing that down in size to understand if pruning out excess information in checkpoint data matters. We'll also ramp up to larger values to see what (if anything) dumping lots of information into your checkpoint data hurts.

Our normal runs don't include a processor, but I did want to see what happened if the processor produced a large result. So I created a simple processor that returns an object of some fixed size, starting with 1024 bytes, and then varied that size as described later. Remember that our normal runs don't even include an ItemProcessor.

What are the Numbers for our Baseline?

Our baseline run with an item size of 1000 took 10,000 checkpoints. The elapsed time for the job was 4:28:314 (four minutes, 28 seconds, 314 milliseconds). CPU usage as measured by SMF was 56912 milliseconds.

For elapsed time we had both the time reported by SMF and the start/end timestamps in the messages around the step execution in the joblog. It was interesting to note that these values are generally within just a few milliseconds of each other.

SMF times for Liberty Batch jobs are reported in the 120-12 record.

Can We Enable the Chunk Listener Monitor?

The Chunk Listener described earlier will give us information about the elapsed and CPU time spent inside the reader, processor, and writer. But before we can enable it, we need to determine how much enabling it costs and whether it skews the results too much.

We performed a run with the listener enabled but otherwise configured exactly as for our baseline run. Enabling the listener added another 4.469 seconds to the run time for the step. It used an additional 3.935 seconds of CPU.

By comparison to the total time for the job (268 seconds) this is noticeable, but not excessive. As we experiment with different configurations, we'll see that this difference is pretty minor, and we can safely ignore it.

Just to be sure, we also ran with an item count of 75 which caused us to take 133,333 checkpoints (instead of just 10,000) and compared the elapsed time for runs with and without the listener configured.

The listener writes one record to an 'iteration' file per record processed which will be the same each run (because both runs process 10 million records). The listener also writes one record to a 'chunk' file per checkpoint taken. This had 13.3 times more records for the run with more checkpoints. Did the overhead of the monitor increase significantly?

In this run the listener-enabled job took an extra 7.296 seconds above the baseline or about 3 seconds more than enabling the listener with an item count of 1000. When we look at the increase in run times for lower item count experiments, we'll see this difference is pretty negligible.

Playing with Checkpoint Intervals

One question that comes up a lot is, "How long should your checkpoint intervals be?" On the one hand, checkpointing frequently means less work to potentially re-do if a fatal error occurs running the step. On the other hand, we assume that there is some overhead to taking a checkpoint and so taking a lot of them will make the job run longer. How much longer? And, even knowing that, where is the optimal tradeoff point? Would you let a 10-hour job run five minutes longer if it meant restarting on a failure was easier? What if the 10-hour job ran 2 hours longer? It probably depends on how much easier, of course.

Our goal in this section is to try to get some sort of feel for the answer, or at least give you some idea how you can find out for yourself. That's because what we find for our sample job is probably going to be different than for your own applications.

We'll start by looking at the overall time for a chunk step as we vary the item count values from 25 to 8000. Using data from the chunk listener we'll break out elapsed time for reading records, inserting and deleting database records, and container overhead. We'll see that at low item count values container overhead from handling all the checkpoint processing dominates the time spent in the step, but with larger checkpoints the record writing (and in particular one-by-one deletes of database records) dominates. We'll also discover that gains from increasing the checkpoint size diminish as item count values increase beyond a few thousand.

Let's take a look…

The Raw Results

We started out running our job as described earlier. We used a 1000-item checkpoint interval and processed 10,000,000 records. Our chunk listener was enabled to give us more data about the time spent within the parts of the chunk processing. Then we varied the item count up and down to see how it affected the elapsed time for the step. Here are the results:

Table 1 - Step Elapsed Time for Different Item Count Values

Item Count	Elapsed Time (m:s.ms)	Elapsed Time (ms)
25	23:58.624	1438624
75	10:29.566	629566
125	7:54.214	474214
250	6:13.130	373130
500	5:03.840	303840
1000	4:32.783	272783
2000	4:16.833	256833
4000	4:04.517	244517
5000	4:07.129	247129
6000	4:04.723	244723
7000	4:03.701	243701
8000	18:37.523	1117523

Let's have a look at that in graphical form:

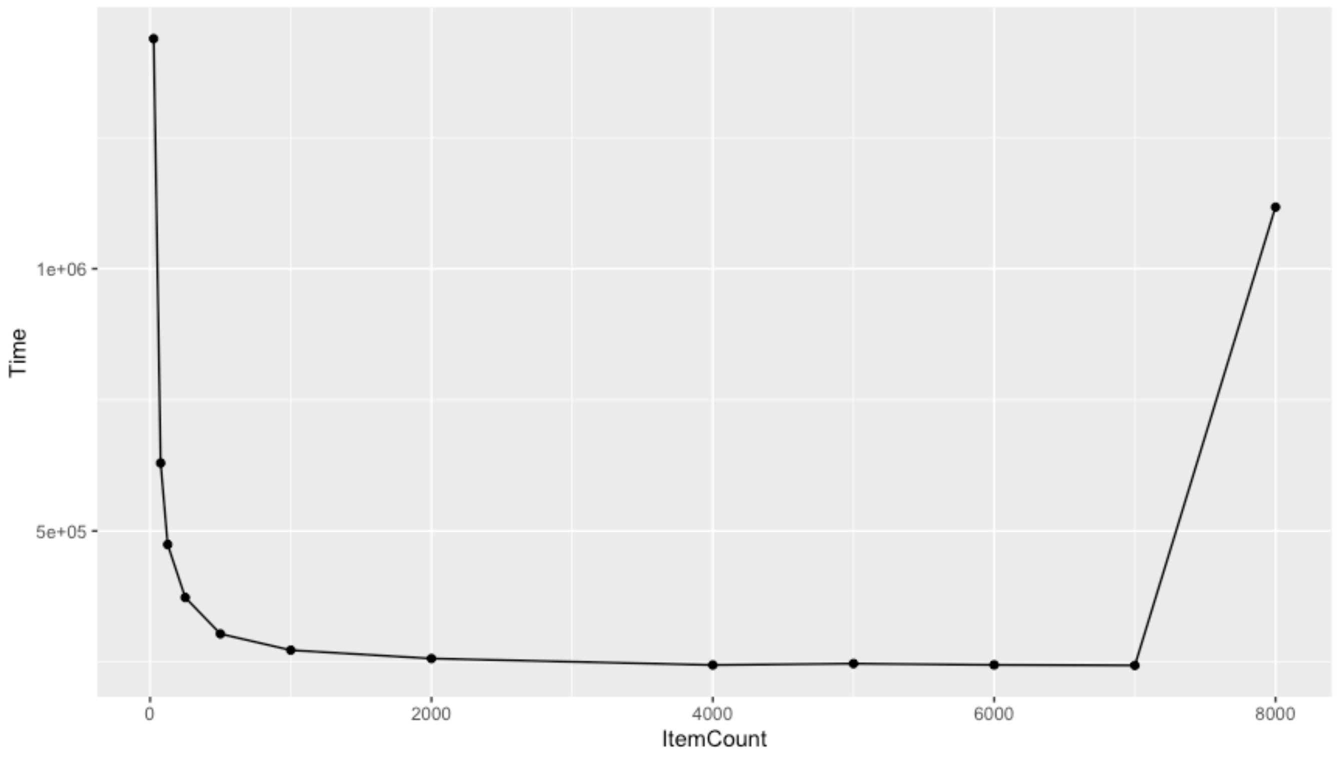


Figure 1 - Step Elapsed Time for Various Item Count Values

Our first, high level observation would be that decreasing the item count (which significantly increases the number of checkpoints taken) has a detrimental effect on the run time of the job as you get into small item-count values. Additionally, as item count increases and the checkpoints grow larger and fewer the improvements grow less and less.

And something rather interesting happened at an item count of 8000. That's an anomaly caused by an issue with DB2 and the number of updates in a single commit scope. We'll see the details later, but you can really just ignore it.

Looking at the Read Time

The Chunk Listener reports the elapsed time spent doing each of our 10 million reads. However, the granularity of that time is just one millisecond and z/OS is pretty good at doing I/O. Almost all of our reads reported taking zero milliseconds, with only around .019% reporting one millisecond. I was initially concerned that we were missing a lot of time here, but it can't be that much. Suppose each of those zero read times were actually 0.1ms. If that was true, then read time for the whole step would be 10,000,000 * 0.1ms which is 1000 seconds which is over 16 minutes. Elapsed time for the entire step only got above that twice and both times we'll see that write processing is most of it.

I concluded that read time is pretty negligible and largely ignored it.

Looking at the Writer

The Chunk Listener also reports time spent in each call to the Writer. For each run we took the write times for each chunk and added them up to find the total time spent writing data. Here are the results:

Table 2 - Writer Time for Various Item Count Values

Item Count	Total Writer (ms)
25	228595
75	202233
125	199447
250	219000
500	224786
1000	229436
2000	230579
4000	228580
5000	232573
6000	231554
7000	231179
8000	1104194

And in graphical form including the total time for the step so we can see how most of the time consumed in the step is by the Writer (except at low item count values). We also see the Writer is responsible for that spike at 8000.

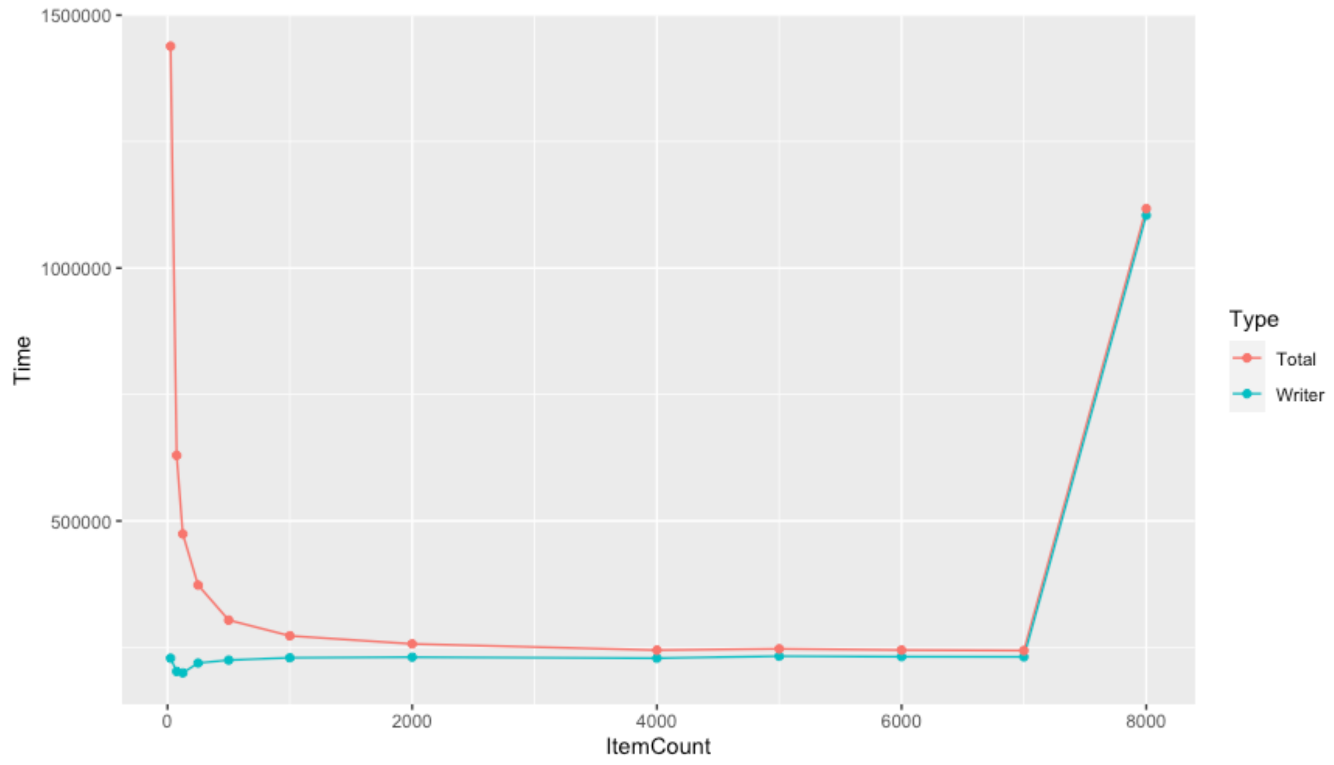


Figure 2 - Writer Time for Various Item Count Values

If read time is negligible and writer time is most of the step time, what happened at low item count values? Before we look at that, we should dig a bit deeper into the Writer times.

Bulk Inserts vs One by One Deletes

Remember that our Writer alternates between inserting and deleting the same rows. The inserts are done with a single bulk insert, while the deletes are done one-by-one using the primary key for the row. Do they contribute differently to the elapsed time for the job? They sure do.

First, we pulled out the data for the inserts. Remember that the item count determines the number of rows being inserted in each chunk.

Table 3 - Insert Time for Various Item Count Values

Item Count	Insert Write Time (ms)
25	43231
75	27568
125	25572
250	23129
500	21107
1000	20256
2000	19562
4000	18758
5000	18687
6000	18278
7000	18116
8000	18242

Looking at it graphically

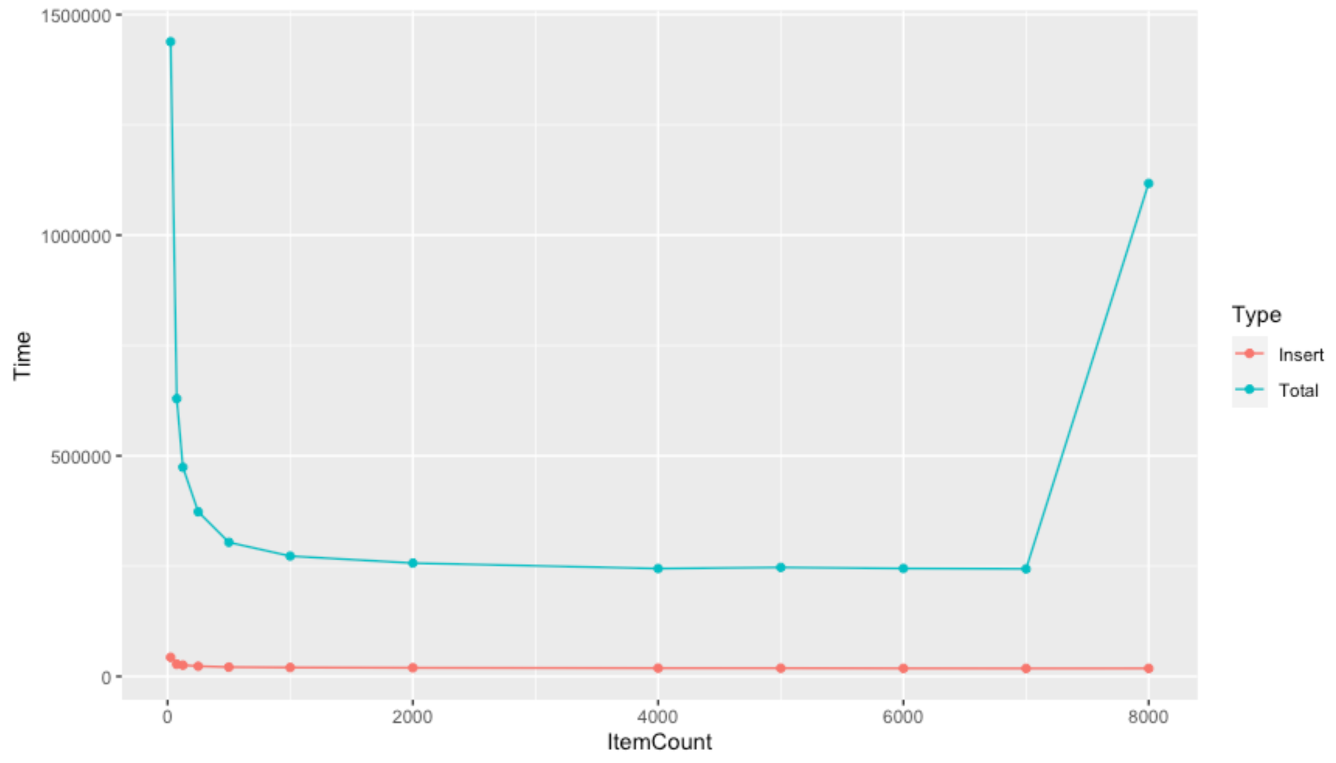


Figure 3 - Insert Time for Various Item Count Values

As you would hope (and perhaps expect) bulk inserts become more efficient as you insert more items. Although the efficiencies taper off as the number increases, it definitely seems as if larger chunk sizes are beneficial if you are doing bulk inserts in the writer for each chunk.

It also appears that the inserts aren't responsible for that spike on the right. Is it the delete processing? Let's see:

Table 4 - Delete Time for Various Item Count Values

Item Count	Delete Write Time (ms)
25	185364
75	174665
125	173875
250	195871
500	203679
1000	209180
2000	211017
4000	209822
5000	213886
6000	213276
7000	213063
8000	1085952

And in graphical form:

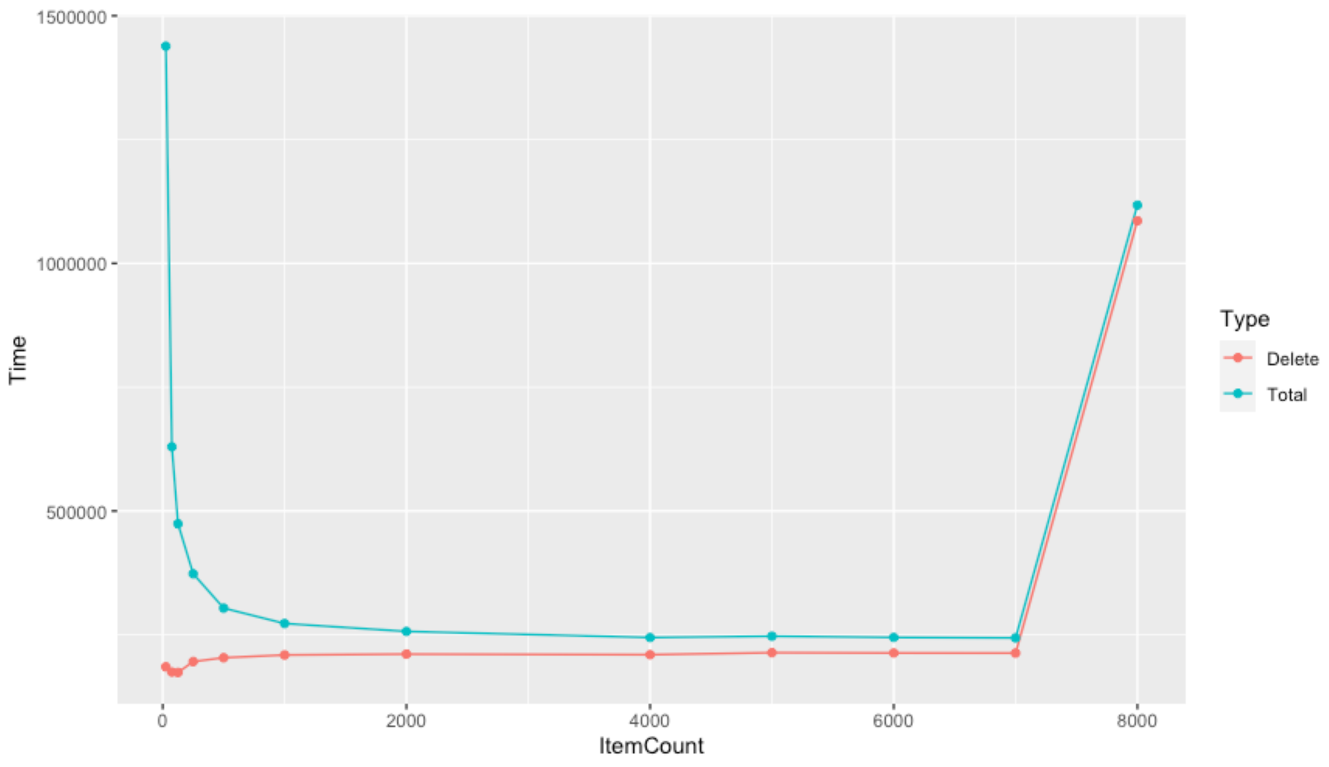


Figure 4 - Delete Time for Various Item Count Values

It appears that delete times make up the bulk of the Writer processing (which in turn makes up the bulk of the step time). However, those times are essentially flat across the different chunk sizes. That makes sense…if you are going to delete 5 million rows one by one it doesn't matter a lot if you do that committing every 500 items or every 5000. It is a bit cheaper at lower chunk sizes suggesting some efficiency in commit processing in DB2 perhaps.

Of course, the big question is what happened at an item count of 8000. I believe we ran into some tuning problem with DB2 trying to process 8000 separate deletes in a single transaction. The inserts avoid this. There may be some setting we could have adjusted to push this out farther to the right, but the gains from increasing the item count value were becoming less interesting.

Now let's put the Inserts and Deletes together so we can see how they contribute to the overall elapsed time for the step.

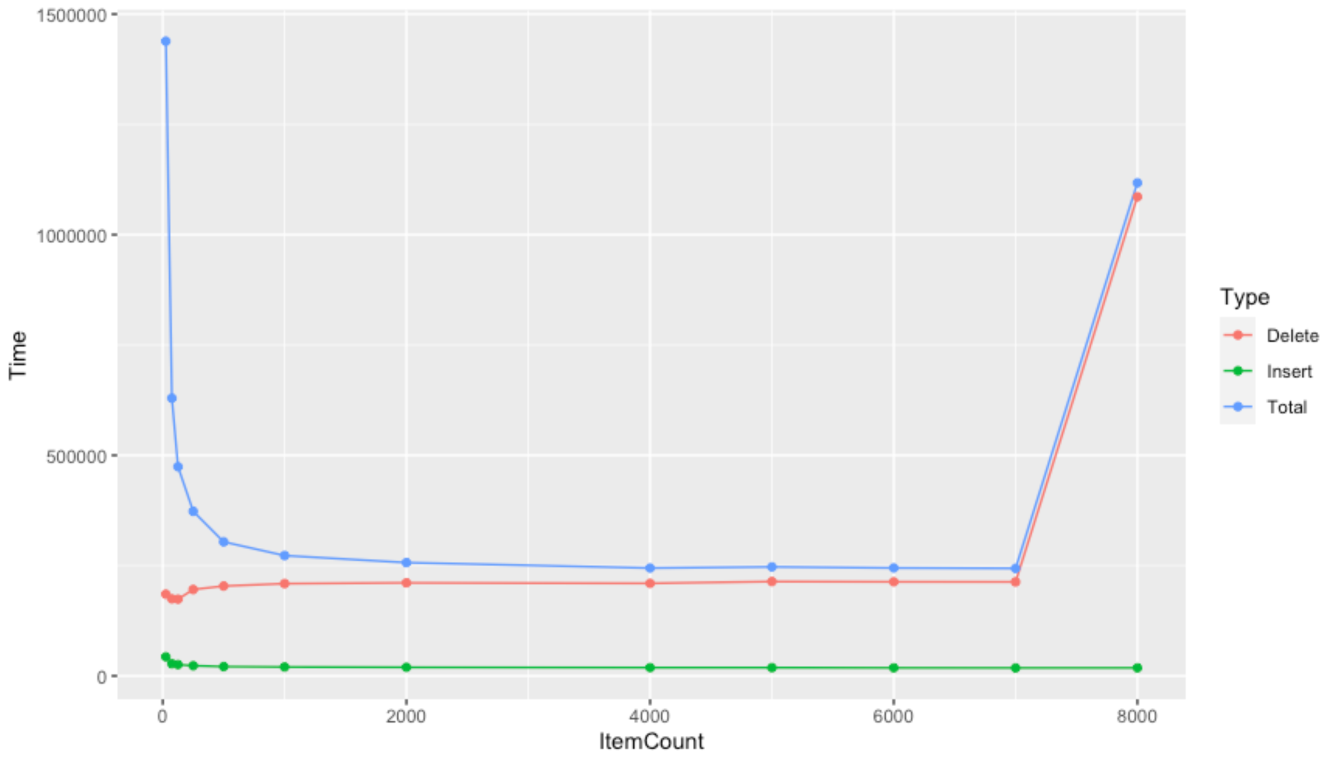


Figure 5 - Inserts and Deletes as Components of Total Step Time

Container Overhead

Which brings us to our final component of the elapsed time for the step: the batch container. At every checkpoint the batch container is going through the work of getting checkpoint data and committing the transaction for the chunk. The more chunks in the step, the more times that will happen. We calculate the container time by subtracting the reader and writer time from the elapsed time for the step. Here are the results:

Table 5 - Container Time for Various Item Count Values

Item Count	Container Time (ms)
25	1207965
75	425263
125	272758
250	152184
500	77179
1000	41500
2000	24413
4000	14068
5000	12747
6000	11410
7000	10742
8000	11408

Looking at it graphically

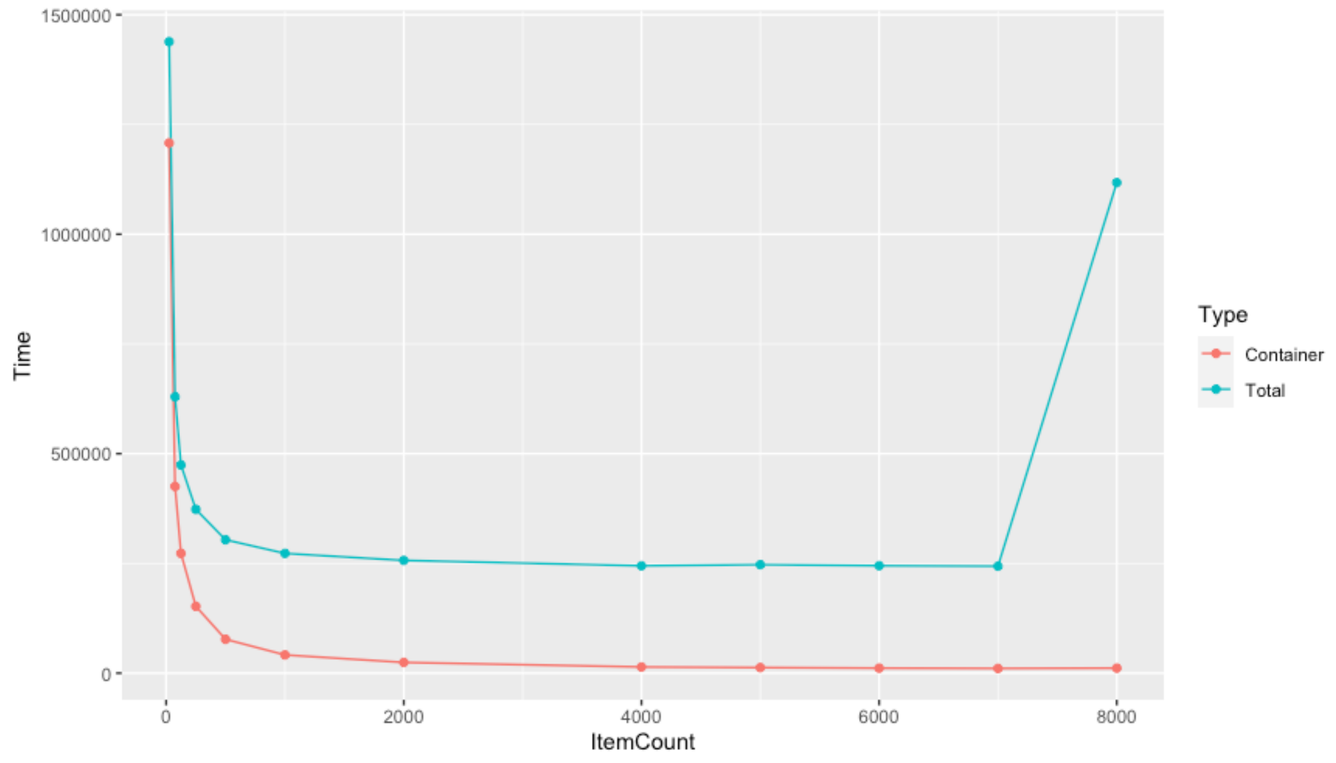


Figure 6 - Container Time for Various Item Count Values

As you would expect, the overhead from the batch container increases as you increase the amount of work it has to do by decreasing the item count (which increases the number of checkpoints taken) to process 10 million records. The cost goes from a huge component of the total step time to a barely noticeable part (around 11 seconds out of four minutes above an item count of 4000).

Just for comparison, here's a graph showing how the number of checkpoints increases as the item count decreases. Looks just like the Container Time curve doesn't it?

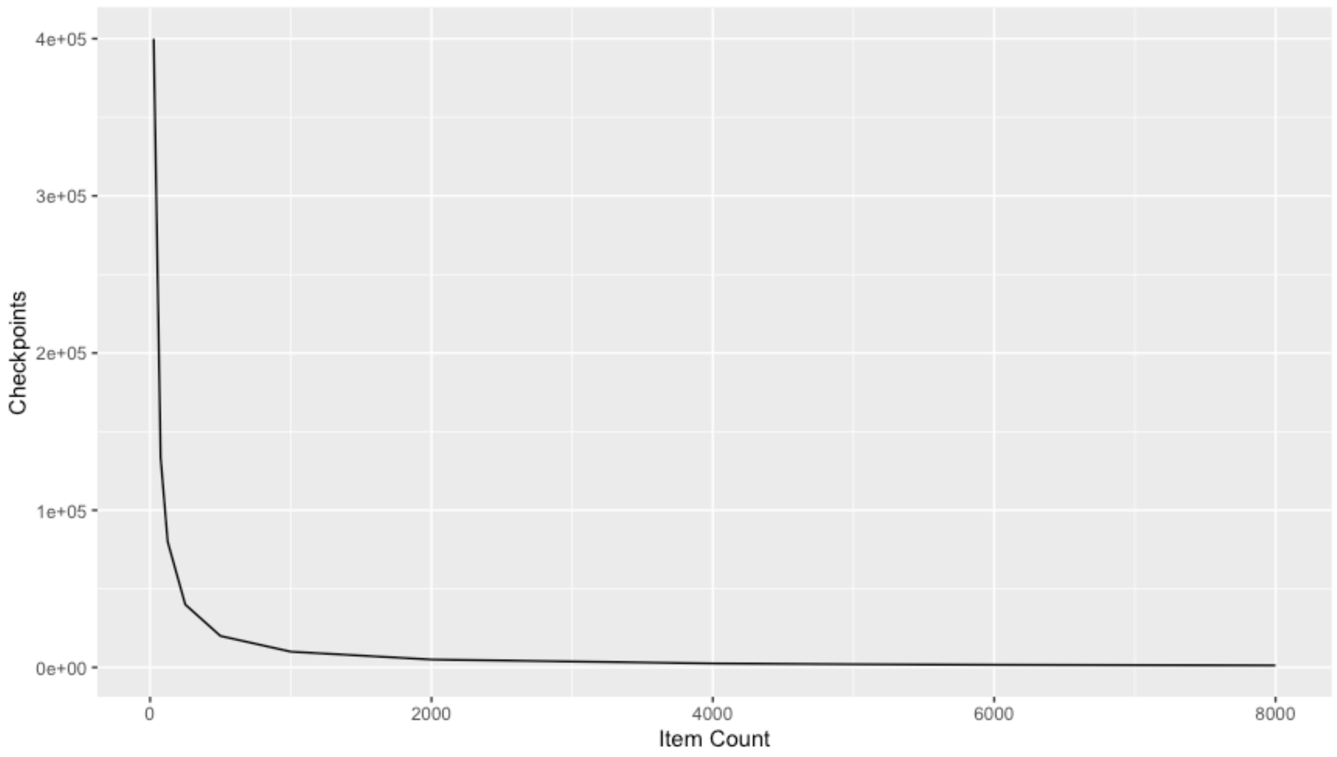


Figure 7 - Number of Checkpoints vs Item Count

Here's one final graph that shows each component (inserts, deletes, container overhead) contributing to the total elapsed time for the step.

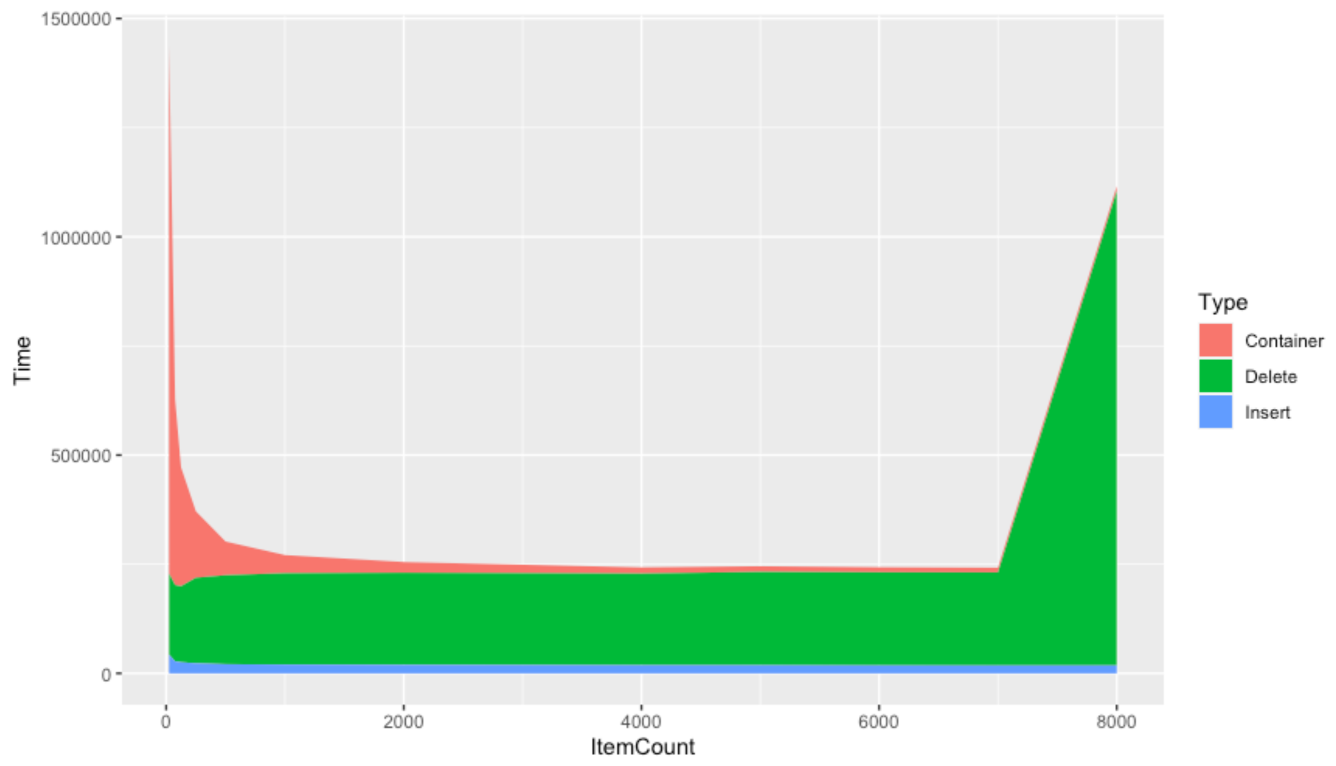


Figure 8 - Components of Elapsed Time for Various Item Count Values

Conclusions

So, where does all this leave us? We've seen that very small chunks create more overhead from the container. Larger chunks reduce that, but above 1000 to 2000 each increase in chunk size produces minor improvements. And as we saw from the spike at 8000, you might run into other limitations or issues as you push the commit size larger and larger. I should also point out that I did no tuning of DB2 at all and there is nothing else running on this system for it to contend with.

We've seen that batch or bulk inserts are more efficient than doing things one row at a time. But it is important to note that the improvements are pretty minor as chunk size increases.

Increasing the chunk size to very large values isn't going to get you much per/item improvement. Depending on your application and environment, cleaning up from a job that failed mid-chunk might involve costs that become unacceptable for large chunks.

Looking at our data from this specific step, an item count value of 1000 or 2000 is probably a reasonable value.

Of course, remember that all this is based on a very specific application and environment. Your results will probably vary. Add in the chunk listener to your own application and experiment with different item count values to draw your own conclusions.

Sidebar: Creating the graphs

I used an open source statistical package called ‘R’ to produce the graphs above. I started with a table with three columns: Time, ItemCount, and Type. The Type field indicated whether the Time value was total time, insert time, delete time, etc. I used the subset command to create new tables with different slices of the table depending on which values I was graphing and then used the ggplot function from the tidyverse package to produce the graphs. I’m really new to R so there’s probably a better/easier way to do this, but if it helps somebody produce similar graphs for your own data, here’s how I did it:

```
Summary <- read_csv("path/to/data/Summary.csv”)
StepTotal<-subset(Summary,Type=="Total”)
ggplot(data=StepTotal,mapping=aes(x=ItemCount,y=Time))+geom_line()+geom_point()
WriterData<-subset(Summary,(Type=="Total"|Type=="Writer”))
ggplot(data=WriterData,mapping=aes(x=ItemCount,y=Time,color=Type))+geom_line()+geom_point()
DeleteData<-subset(Summary,(Type=="Total"|Type=="Delete”))
ggplot(data=DeleteData,mapping=aes(x=ItemCount,y=Time,color=Type))+geom_line()+geom_point()
WriteTotalData<-subset(Summary,(Type=="Total"|Type=="Delete"|Type=="Insert”))
ggplot(data=WriteTotalData,mapping=aes(x=ItemCount,y=Time,color=Type))+geom_line()+geom_point()
ContainerData<-subset(Summary,(Type=="Total"|Type=="Container”))
ggplot(data=ContainerData,mapping=aes(x=ItemCount,y=Time,color=Type))+geom_line()+geom_point()
InsDelContData<-subset(Summary,Type!="Total"&Type!="Writer")
ggplot(data=InsDelContData,mapping=aes(x=ItemCount,y=Time,fill=Type))+geom_area()
```

Playing with Checkpoint Info Sizes

For the next experiment I wanted to explore how changing the size of the serializable checkpoint data returned by the reader or writer impacts the elapsed time of the job. At every checkpoint the batch container gets this information from the reader and writer and updates the relevant row in the Job Repository table as part of the chunk commit scope. A minor change in elapsed time, multiplied by a large number of checkpoints, could theoretically impact the runtime for the job.

In theory the state data required to restart from a checkpoint shouldn't have to be very big. But if you're using a non-transactional resource you might have to do some rollback-type cleanup yourself and that might require more data.

And, of course, sometimes objects just accumulate information. I've seen HTTP session state objects that were several megabytes.

The question we're looking at here is how much it matters. Of course, there are a lot of other factors at play. Checkpoint data is hardened in the Job Repository which is just a database. The main consideration is how quickly that database row update can take place. In my experiments I was on z/OS accessing a local DB2 instance over a type-4 connection. Switching to a remote database is going to require checkpoint data to flow over a physical network which is going to significantly slow things down.

Note that while the default size for the BLOB that contains the checkpoint data is 2GB, your DBA might have pruned that down (mine did) and you'll get SQL errors trying to save very large checkpoint data (I did). Remember that both the reader and writer checkpoint data are placed together into one column. It is easy enough to alter the table to make it as big as you need, assuming you can convince your DBA you need the space.

For our runs we began with our same baseline, processing 10 million records in 1000 record chunks. I updated the checkpoint data size for both the reader and writer to the same value. Our baseline run uses a 1024 byte object so the total checkpoint data was 2048 bytes. We scaled that up and down by factors of 10. For each run, as before, we determined the reader and writer time from the chunk listener and subtracted that from the elapsed time for the step to determine the time spent in the container (which includes the time spent updating the checkpoint information in the Job Repository).

Our results:

Table 6 - Container Time for Variously Sized Checkpoint Data

Data Size (bytes)	Container Time (ms)
20	41851
2048	42271
20480	45636
204800	66454
2048000	303026

Graphically that's a pretty straight line:

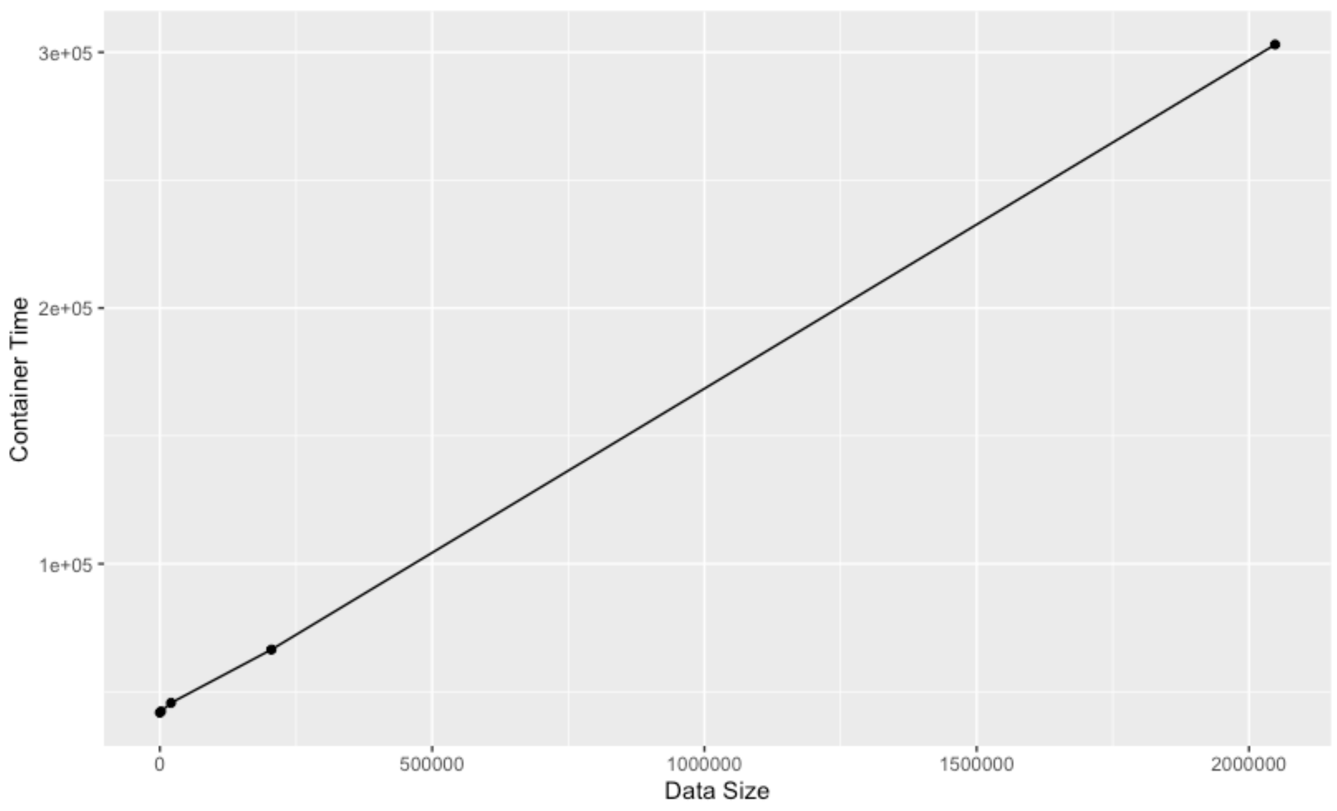


Figure 9 - Container Time for Various Checkpoint Data Sizes

Which means that, as you'd probably expect, the larger your checkpoint data, the longer it takes to commit it at each checkpoint. For my environment making it a little bigger didn't hurt very much. It just added a few seconds to the elapsed time for the job. Making it a lot bigger added about two minutes to the length of the job.

Does that matter? Probably not, but in a different, less co-located, environment it might matter more. As a general rule there's probably no great need to keep your checkpoint data as small as possible. Some extra fluff in the object isn't a big deal. But as it grows and grows eventually it will matter. So pay some attention to checkpoint object size, but don't obsess about it.

Playing with Write Object Size

For this experiment we're going to change the size of the object returned by the processor. These objects are held in memory until they are passed to the writer as part of end-of-chunk checkpoint processing. Basically, this means you can multiply the size of the processing-result object by the number of items processed per checkpoint to estimate how much Java heap you need available for this chunk to run. Remembering, of course, that your job might not be the only one running in the server at the time.

Changing Our Application

All our tests so far haven't had a processor involved, but we'll need one for this experiment. We don't want it to take very long and we don't want it to use much heap beyond what it needs for the returned result.

To achieve this, the constructor for the processor creates a byte array of a specific length and fills it with the alphabet (repeating). We don't want to build the array every time through the processor to save CPU. The `processItem` method then new's up a byte array as long as our initialized string and uses `System.arraycopy` to copy the data and returns the now populated array.

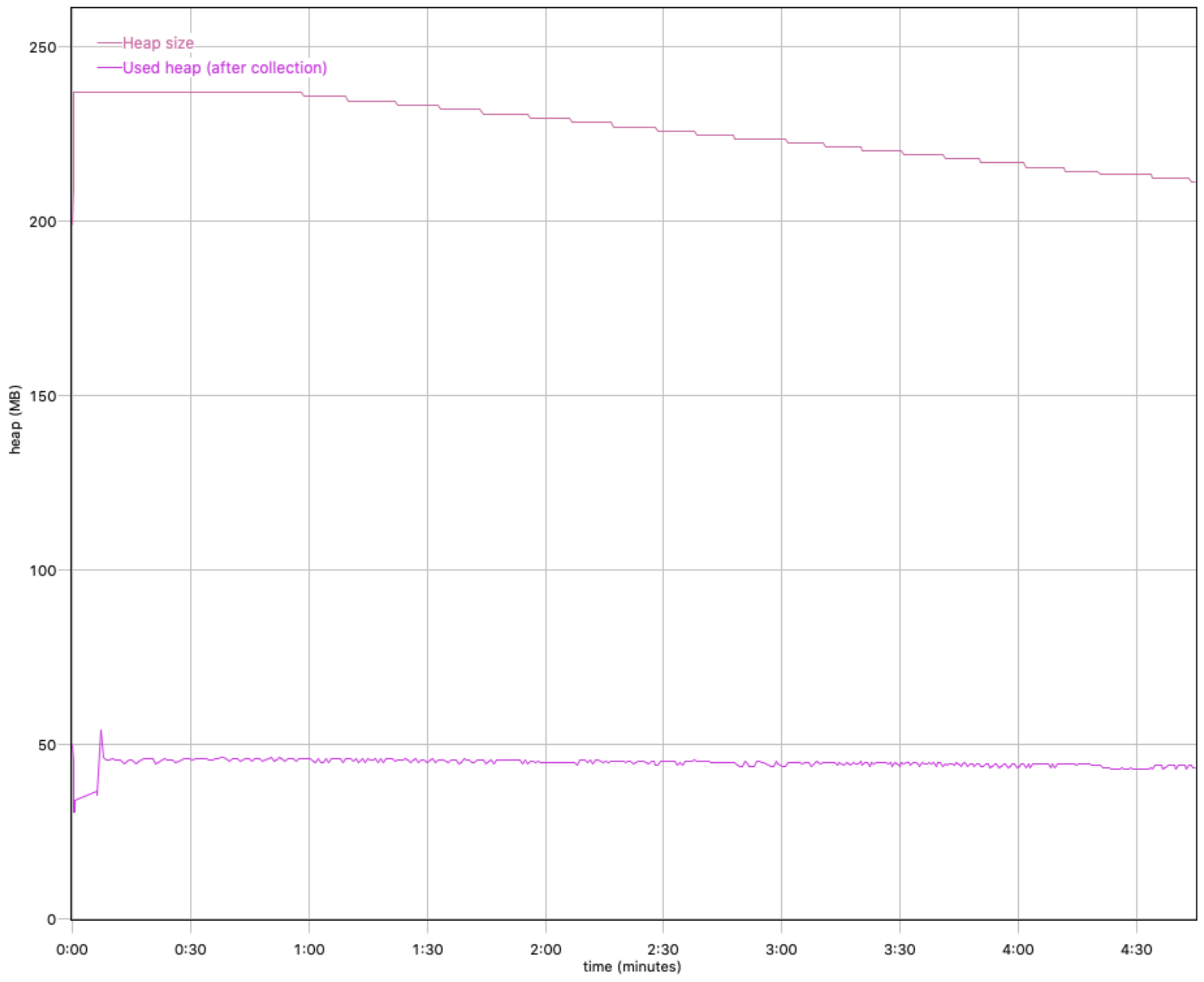
We're going to allocate a new array every time through the read/process loop. Staying with our baseline item count value of 1000, the writer will receive one thousand of these byte arrays. The storage will be released when the writer returns. As before, we'll do this for ten million records.

The JVM for our server is configured with a min of 8M and a max of 512M.

The Results

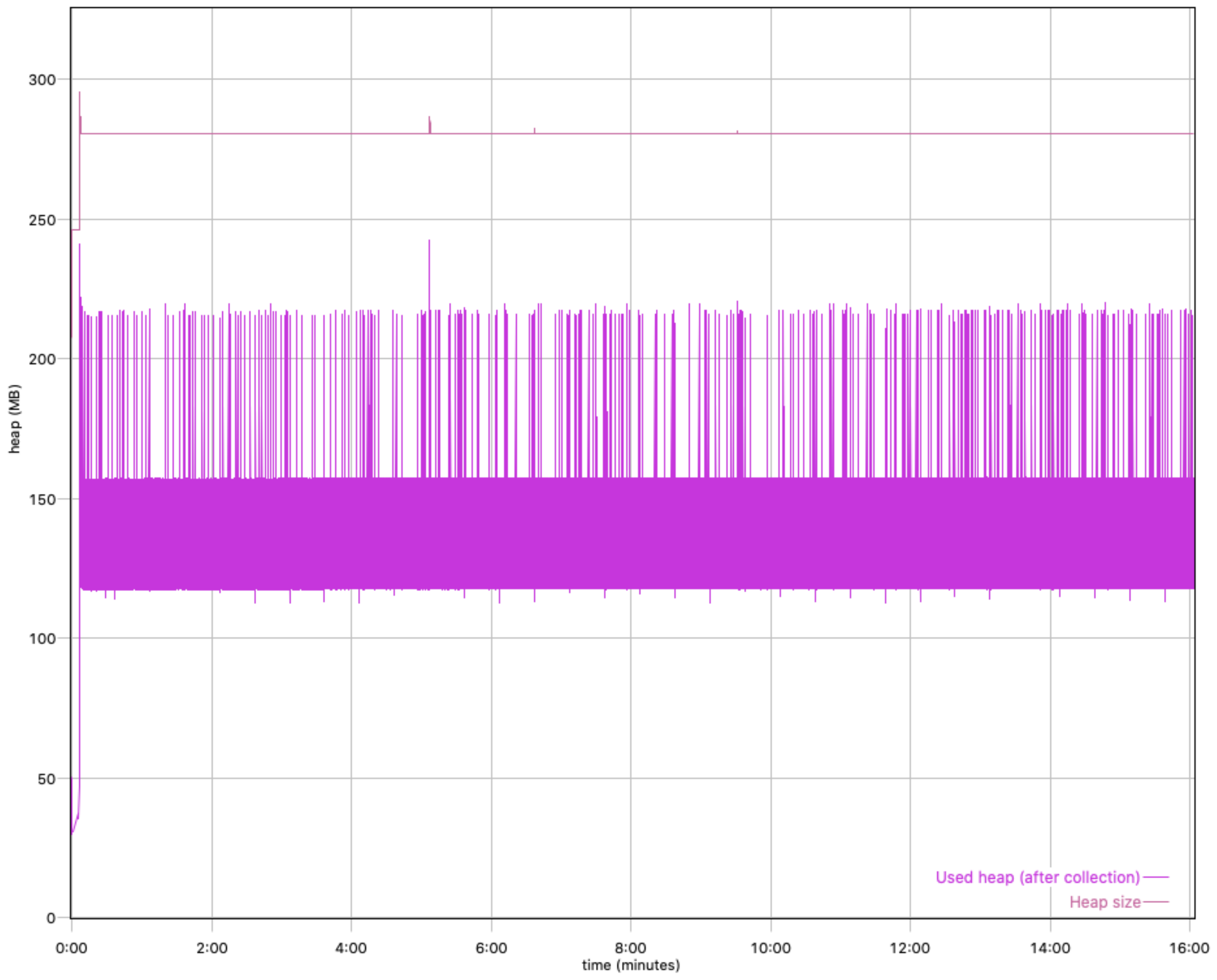
It turned out we could just do two runs to prove the point we wanted to make. If you didn't already know, excessive garbage collection is bad. Let's see how bad.

For the first run the processor returned a 1k byte array. That means we built up 1000k (1MB) of data for each chunk. That's not very much. So our results looked much like they did when we ran without a processor at all. The chunk step took just over 4 minutes and 37 seconds (compared to 4 minutes and 32 seconds with out the processor). I captured the verbose GC output and fed it to the GCMV Eclipse plugin. This provided me with the graph below of heap usage over the run time for the job.



You can see it jiggling around but nothing very exciting happens.

For the next run we increased the size of the returned byte array to 100K which means each chunk loop will build up 100,000K of data (or 100MB). We'll look at the GC picture returned by GCMV first and then talk about what it did to the elapsed time for the step.

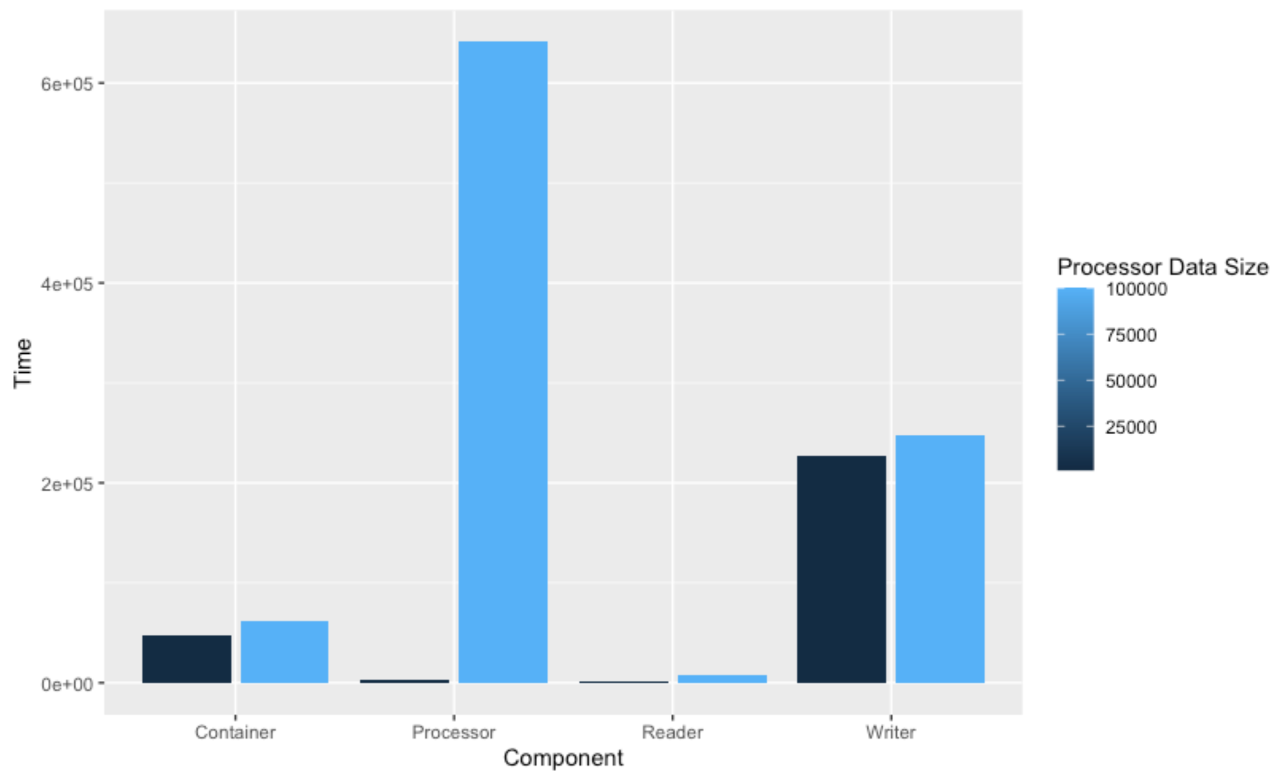


Wow. That stirred things up. The elapsed time for the step increased to almost 16 minutes and the heap thrashing affected every part of the job. In the following table we'll compare the elapsed time for the reader, processor, writer, and container where the only change is the increase in the size of the byte array returned by the processor.

Processor Data Size (bytes)	Reader Time (ms)	Processor Time (ms)	Writer Time (ms)	Container Time (ms)
1000	1704	2443	227055	46651
100000	7281	640866	247327	62242

Looking at this data you can see that most of the 12 extra minutes of elapsed time for the job was spent in the processor copying a 100k byte array. But time spent in the reader, which has been consistent for all our other runs, jumped around by about 3x. The writer and container times were also affected by the excessive amount of garbage collection being performed. Make sure you have enough heap to contain the processor results that will accumulate.

Looking at it graphically:



In case you're wondering, I created that in R by restructuring the table above to have just three columns: Processor Data Size, Component, and Time. Then I used this command to create the table:

```
ggplot(data=Summary)+geom_bar(mapping=aes(x=Component,y=Time,fill=`Proces
sor Data Size`),stat='identity',position='dodge2')
```

The stat value of identity tells it to use the specified 'y' value for the height of the bars, and the position value of dodge2 tells it to place the values for different data sizes side by side instead of stacked.

Experimenting with Partitions

The next item of interest was how use of partitions can reduce elapsed time. In this experiment we used the same reader/writer (no processor) job we started with and continued to process 10,000,000 records with an item-count value of 1000.

What we changed was the level of concurrency. In all our runs so far we used a single thread to process all the records. This time we'll split up processing of the records across multiple threads. We'll begin with just two partitions each processing 5,000,000 records and proceed to 10 partitions processing 1,000,000 records each.

Our partition mapper arranged for each partition to use different primary key value ranges for the database inserts/deletes to avoid contention. In a real application we'd hopefully be inserting (or deleting) different rows, but how the table is locked (row etc) could impact you.

Remember that just because we have 10 partitions doesn't mean all 10 will run concurrently on 10 separate threads. Instead 10 work items will be queued up inside the server and will be picked up when a thread is available to run the work. Liberty will dynamically adjust the number of threads it has to try to handle the work, but we don't pre-configure the server with some number of threads.

After 10 partitions, we'll proceed to 20 and then 100 and finally 1,000 partitions each processing a single chunk of 1,000 records.

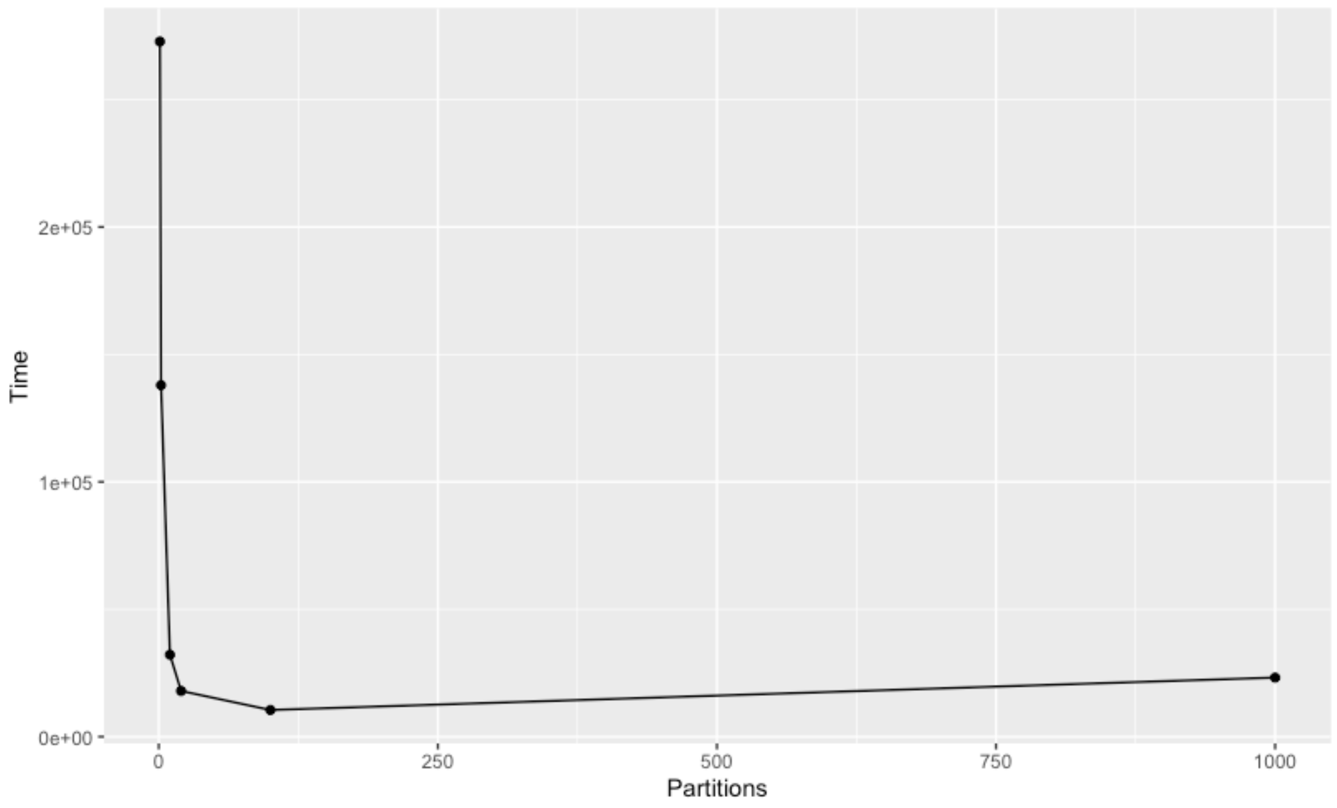
Looking at the data below you can see that the first increment to two partitions nearly cut the elapsed time in half. Moving to 10 threads cut the time to not quite 1/10 of the single threaded version (32 seconds instead of 27 as you might hope). But 20 partitions almost cut the time in half from 10 partitions. But our gains began to max out as going to 100 partitions didn't even cut the time from 20 partitions in half. Why is that?

Most likely it is because the server can't spin up 100 threads very fast and instead just re-used existing threads to process different partitions sequentially. And we can see that when we moved to 1000 partitions the step actually took longer than with just 20).

I should point out that these experiments were run on a dedicated LPAR. Nothing else was running on this OS image, nothing else was touching DB2. The system I was running on was a z14 with 20 GPs. Clearly a real-world test would involve some contention for resources and affect the results.

Nevertheless, it seems quite clear that, when possible, the use of partitions can dramatically reduce elapsed time for a step.

Partitions	Elapsed Time (ms)	Elapsed Time (m:ss:nnn)
1	272783	4:32.783
2	137946	2:17.946
10	32170	0:32.170
20	17960	0:17.960
100	10485	0:10.485
1000	23204	0:23.204


Time
2e+05
1e+05
0e+00
0
250
500
750
1000
Partitions

Enabling Batch Events

The JSR-352 implementation in WebSphere Liberty includes the ability to publish messages into a topic tree at interesting points in the lifecycle of a job. Messages (or events) are published around the start and end of a job, as well as the start and end of each step. Most interesting to us is the event published at each checkpoint. In our baseline job there will be 10,000 checkpoints and thus 10,000 messages published. Does enabling this feature (which allows a monitor to track the status of the job) significantly slow down execution of the job itself? Let's find out.

We can configure the server to publish batch event messages by pointing it to a messaging engine. We can use the WAS messaging server built into Liberty or point to an instance of MQ running on our z/OS system. If we use MQ we can connect over TCP/IP or cross-memory (Client or Bindings mode).

For my experiment I choose to run using the built in Liberty messaging services located in the same server running the batch job. I thought that would probably interfere the most with the running job and be slower (going through TCP/IP) than a Bindings mode connection to MQ.

Remember that we're comparing to our baseline run with no ItemProcessor, handling 10 million records, checkpointing every 1000 items. That step ran in about 4 minutes and 32 seconds. For our test I ran the exact same configuration except that the server was publishing messages as the job progressed. This time the job took 4 minutes and 42 seconds.

So there is clearly a cost, but not a very big one. There is no way to configure the server to only publish messages for selected events (e.g. skipping the checkpoint messages). A very long running job that had hundreds of millions of checkpoints (instead of the 10,000 we had) might see the job run a few minutes longer with events enabled. That seems to be acceptable to me.

A Few More Thoughts

I should point out that all of these runs were done using a Job Repository co-located on the same z/OS image as the Liberty server running the job and I accessed it using a Type-4 JDBC driver. It is possible some of the numbers might improve if I'd used a Type-2 (local) connection. It seems almost certain things would have gotten worse if I'd moved to using a remote database.

Likewise for the batch event measurements, everything was local using a TCP/IP connection. I could have configured the server to use a remote messaging engine would surely have slowed things down a bit.

In addition, I made no attempts to tune the database or messaging engine or adjust the heap size or other JVM options. I just took the environment as I found it.

Finally, all of these experiments were done using a single Liberty server. Configuring a dispatcher/executor model would affect overall job performance, but shouldn't have any effect on the execution of the step itself (unless the partition experiments were performed with the partitions spread across multiple servers).

Conclusions

Looking back across all the experiments and all the data and analysis our conclusions are pretty simple and, truth be told, should probably have been obvious from the start.

1. Checkpoint processing has a cost and if you do a lot of it your job will take longer
2. Larger and larger checkpoint intervals have decreasing benefits and eventually you'll hit some limit
3. Bulk interactions with DB2 are more efficient than one-by-one interactions
4. Pushing excessively large checkpoint data to the Job Repository will slow things down, but you don't have to get obsessive about making it as small as possible
5. Excessive garbage collection will slow down your job, so don't let your processing result object (or checkpoint interval) get so big you run tight on heap every chunk.
6. Partitions are good, but don't go crazy..there's a limit.
7. Turning on batch events doesn't cost very much

And finally, probably most importantly

8. Your results will vary, do your own measurements

Document change history

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

January 6, 2021	Initial Version

End of 6398358