Base Language - Bug #5446

PUT UNFORMATTED performace (stream is flushed on each call)

06/14/2021 09:43 AM - Constantin Asofiei

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD	**version:**	
Description			

History

#1 - 06/14/2021 09:46 AM - Constantin Asofiei

This testcases executes in 4GL almost immediately:

```
def var ch as char.

ch = "12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890".
ch = ch + ch.

def stream rpt.
output stream rpt to a.txt.

def var i as int.
do i = 1 to 100000:
   put stream rpt unformatted ch skip.
end.

output close.
```

In FWD, it takes minutes to complete. Even though there are 100k server-client roundtrips, the most consuming factor is the buffer flush in Stream.putWorker line 5785, which makes the buffer's 4096 bytes useless.

If this line is commented, FWD completes a lot faster (~20 times faster, as each line is 200 bytes and the buffer is 4096 bytes).

The question is: is the buffer flush for a PUT statement required in 4GL?

#4 - 10/01/2021 09:02 AM - Greg Shah

> The question is: is the buffer flush for a PUT statement required in 4GL?

I think we needed to downcall and flush for each statement for **some** use cases in the 4GL. If I recall correctly, it was related to child process streams. Anytime PUT UNFORMATTED is used on a stream created by OUTPUT THROUGH or INPUT-OUTPUT THROUGH, the pipeline won't work unless there is an immediate downcall AND a flush for each downcall.

Hopefully I'm not forgetting any other dependencies here. I think we can do a much better job for the common case where the downcall and flush is not needed. And I do want us to avoid both the downcall and the flush. We will need to implement a deeper automated refactoring of the code, where possible. The idea is to detect when it is safe to emit loops of PUT UNFORMATTED in a way that will perform acceptably.

This is happening for enough customer cases that I think we need to address it now.

#5 - 10/01/2021 10:58 AM - Constantin Asofiei

Something I need to clarify: this flush in FWD is not a problem just for PUT UNFORMATTED (this is just the statement I kept seeing in the customer code).

This is related to any caller of Stream.putWorker, which are PUT, PUT CONTROL, PUT UNFORMATTED and EXPORT. All have the same performance problem, the stream is flushed on each call.

In OE on Windows, the buffer size looks to be around 1029 bytes (it varies if there are page-top header frames) - but this may be dependent on the OS.

My approach at this time would be to let the stream flush on putWorker only for non-file streams.

#6 - 10/01/2021 11:10 AM - Greg Shah

> This is related to any caller of Stream.putWorker, which are PUT, PUT CONTROL, PUT UNFORMATTED and EXPORT. All have the same performance problem, the stream is flushed on each call.

Yes, understood.

> My approach at this time would be to let the stream flush on putWorker only for non-file streams.

Hmm. Good point. For the flushing part of the problem, we can do this at runtime, on the client. This seems reasonable and will provide a large, immediate improvement. We probably should add a isFileResource() abstract method to Stream to let each stream subclass report this.

Please run ChUI regression testing. It is very sensitive to changes in these statements.

In regard to the avoidance of round trips, we could buffer these on the server side for file streams. I guess StreamWrapper may be the right place for that. What do you think?

#7 - 10/01/2021 11:59 AM - Constantin Asofiei

Greg Shah wrote:

> In regard to the avoidance of round trips, we could buffer these on the server side for file streams. I guess StreamWrapper may be the right place for that. What do you think?

Yes, I was thinking of buffering the StreamWrapper.putWorker calls - but it is kind of tricky, because we can't just flush when the buffer is filled. We need to consider that a PUT can be mixed with any other stream output statement - so we need to flush this buffer before any other Stream API (or maybe just output-related APIs?) is executed.

Also, my current approach would be this: take the arguments from the putWorker(FieldEntry[] data, int mode, char delim), use NativeTypeSerializer to serialize them to a stream, and when a flush is needed just sent this stream to the FWD client to execute the putWorker operations. Serializing everything (especially FieldEntry instances) ensures we have the correct data to output (and any errors are reported when the PUT is attempted).

#8 - 10/01/2021 12:06 PM - Greg Shah

> Serializing everything (especially FieldEntry instances) ensures we have the correct data to output (and any errors are reported when the PUT is attempted).

Yes, that is smart.

> We need to consider that a PUT can be mixed with any other stream output statement - so we need to flush this buffer before any other Stream API (or maybe just output-related APIs?) is executed.

Send to client in these cases:

- server-side buffer is full
- output stream is closed (implicitly or explicitly)
- any other downcall

The downcall case can be handled using state synchronization like we do with the other UI features. This way it is always applied in the right order and there is no extra trip to the client. Code that doesn't intermix client calls will be very efficient, gated by the server side buffer size.

#9 - 10/01/2021 01:31 PM - Constantin Asofiei

Greg Shah wrote:

> The downcall case can be handled using state synchronization like we do with the other UI features. This way it is always applied in the right order and there is no extra trip to the client. Code that doesn't intermix client calls will be very efficient, gated by the server side buffer size.

Well, the scenario I'm looking at has exactly this: a single REPEAT loop which reads a file via IMPORT and writes it into another file via PUT. So using state synchronization doesn't work here - best way would be to somehow intercept any other API calls for the same stream, and flush only then... but I don't know if this is possible.

#10 - 10/01/2021 02:26 PM - Constantin Asofiei

The only feasible way I see to make caching the StreamWrapper.putWorker calls work is this:

- encapsulate all usage of StreamWrapper.stream in a getter
- this getter will perform any flushing as needed
- add a flag to StreamWrapper that it will indicate to cache or not the putWorker calls (this flag needs to be set to true only if the stream references a file, and not a process, terminal, etc).

#11 - 10/01/2021 03:40 PM - Greg Shah

> The only feasible way I see to make caching the StreamWrapper.putWorker calls work is this

It seems reasonable.