Context:

The existing system sends data from a **single** source over REST. It runs on client machine that has memory constraints wherein the process cannot consume more than 512MB.

The volume of data sent for a SINGLE property is of the order of 26MB (1350 records/day \* 100 days \* 200 bytes/record).

Goals:

1) SCALABILITY 🡪 **multiple** sources should be able to send data without violating the above memory constraints.

2) FAIL SAFE 🡪 Ensure data transfer can be resumed from point of failure to avoid sending the whole data again. This implies that a bi-directional communication channel be established.

Proposed Solution:

In order to find a scalable solution we explored WebSockets . The idea was to spawn a WebSocket per source when data-transfer is initiated, perform transfer and close socket after it is complete.

Spike Tools:

* Language – Groovy , Java, Gradle
* OS : Windows 7 -64bit
* DB : MSSQL 2008 , Oracle 10
* Profiling tools : YourKit, JConsole
* Versioning Control : GIT
* Libraries:
  + MSSQLdriver – sqljdbc4-1.jar
  + JTDSdriver - jtds-1.2.7
  + ODBCdriver - ojdbc14-10.2.0.1.0.jar
  + JavaWebsocketAPI - Java-WebSocket-1.3.0.jar

Approach:

Before we started building, we decided to spike this out and get the facts and figures right to ascertain the viability of such a solution.

1. We started with a DataCreator that would help us generate production dataset sizes.

2. We then wrote a SourceStream which works as WebSocketClient and a SourceSink - WebSocketServer

3. Finally we needed a DataFetcher that would fetch data and feed to SourceStream to be delivered to SourceSink.

4. Once this basic structure was in place, we added configurability which allowed us to configure multiple Source Streams.

Data Points included:

1) Memory footprint

2) Time taken for entire data-transfer

Data Analysis – Stage 1

|  |  |  |  |
| --- | --- | --- | --- |
| Data Size | Time (secs) | Memory Consumed(Mbs) | No of Sources |
| 2000K | 73.63 | 100 | 1 |
| 2000K | 97.809 | 189 | 2 |
| 2000K | 110 | 283.33 | 3 |
| 2000K | 130 | 420 | 4 |
| 2000K | OOM | OOM | 5 |

Data showed us that memory consumption was increasing with number of SourceStreams (which is expected), but would not support more than 4 concurrent SourceStreams. 512MB heap was not sufficient for data transfer with volume of 2000K records per source with 5 sources, and we faced an OOM error.

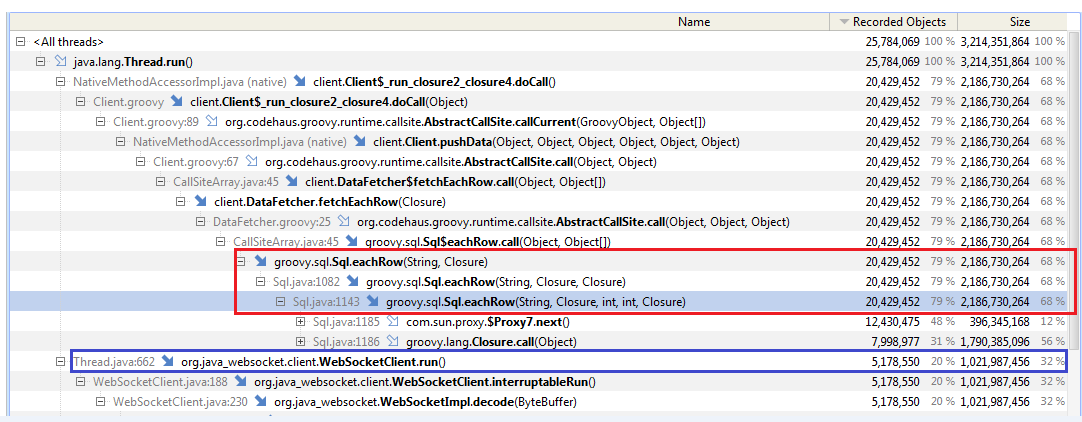
Browsing through various blogs and documentation gave us hints that by default MS-SQL loads the entire resultset in memory and using cursors would reduce memory footprint.

On using the connection parameter - 'selectMethod' = 'cursor', we observed there were intermittent thread locks so the data transfer was never completed. We were quite intrigued by it.

Upon profiling the client we observed that there was initial 45MB taken up upon the JVM start, whether it was 1 SourceStream or 3 SourceStreams for any volumes 10K or 100K or 1000K. Clearly something was amiss here. To our surprise discovered that

1) ResultSet size occupying 68% of JVM space

2) WebSocket 30 %



This profiling forked 2 paths

1. Change driver and re-test
2. Use Java instead of Groovy and re-test

Both these paths produced similar output, so clearly data fetching needed a re-look.

Using fetchSize for Resultset:

Next option was to try 'fetchSize' with range of fetchSizes to see the effect on memory. Results of testing with combination of cursor and fetch size , No Cursor but use fetchSize alone or Use only Cursor, all had similar memory and time requirements. Data showed that even if the fetchSize was increased, there was no observed variance in memory consumption.

Data Analysis – Stage 2 (Cursor and fetchSize)

|  |  |
| --- | --- |
| 100K - CURSOR ON | |
| Fetch Size | Memory consumed(MB) |
| 1 | 62 |
| 50 | 147 |
| 100 | 148 |
| 150 | 132 |
| 500 | 148 |
| 1000 | 150 |

|  |  |  |
| --- | --- | --- |
| 100K - All Cursor Combinations | | |
| Fetch Size | Cursor | After Complete |
| 1 | OFF | 209 |
| OFF | OFF | 151 |
| OFF | ON | 132 |
| OFF | OFF | 160 |

At this point , We stumbled upon the blog:

<http://blogs.msdn.com/b/jdbcteam/archive/2007/05/02/what-is-adaptive-response-buffering-and-why-should-i-use-it.aspx>

<http://technet.microsoft.com/en-us/library/bb879937(v=sql.110).aspx>

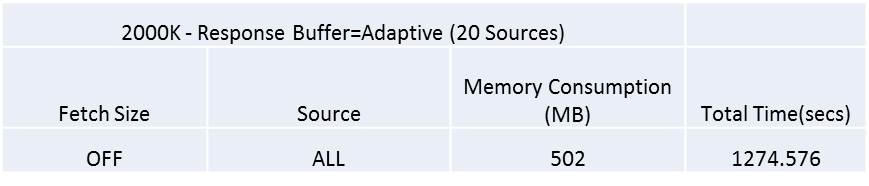
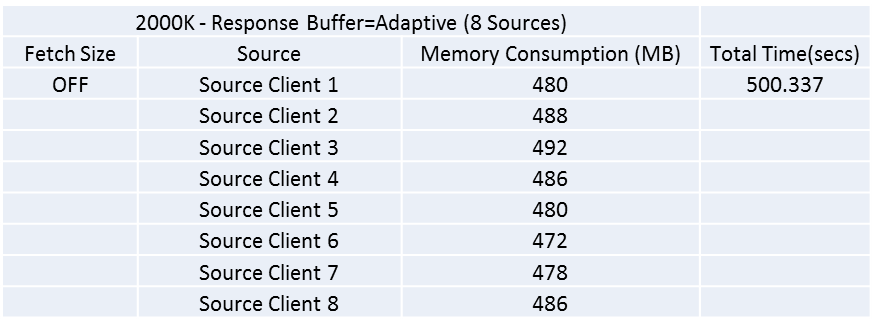
MSDN recommendation suggests DO NOT USE 'selectMethod'='cursor' for read-only large data-sets and a better option is to use 'responseBuffering'='adaptive'.

Using 'responseBuffering'='adaptive' we were able to get rid of both the problems:

1. Out of Memory issue was resolved
2. There were no deadlocks.

With the simplest code, data transfer of 2000K records with 5 sources was failing. Adding adaptive response buffering enabled a transfer of 2000K records per source with 40 sources without any issues.

However, this does not mean that above 40 sources failed in transfer.



Conclusion:

For large read-only result sets, setting response-buffering as adaptive is the best choice.