Results for BYTE_STREAM_SPLIT in Apache Parquet based on implementation in Apache Arrow

Martin Radev

November 29, 2019

1 Testing BYTE_STREAM_SPLIT in Apache Arrow

The comparison is based on the implementation of the proposed *BYTE_STREAM_SPLIT* encoding in the Apache Arrow module for reading and writing Parquet files. The test system is Intel I7 3930K (Sandy Bridge) and 4x Kingston DDR3-1600 4GB. Compiler is GCC 7.4.0 and the compiler options are *-O3 -msse4.2*.

The following combinations are examined:

- (UNCOMPRESSED, RLE_DICTIONARY)
- (ZSTD, PLAIN)
- (ZSTD, BYTE_STREAM_SPLIT)
- (ZSTD, BYTE_STREAM_SPLIT (simd))
  This is an optimized implementation using SIMD extensions up to SSE4.2.

In all tables higher is better and the best option is highlighted in green.
Test cases ending with *.dp* represent double-precision data and test cases ending with *.sp* represent single-precision data. The data was collected from various sources [1–4]. For all tests both the input and output are main memory and no IO operations are performed.

Test case	PLAIN, UNCOMPRESSED	DICTIONARY, UNCOMPRESSED	PLAIN, ZSTD	DICTIONARY, ZSTD	BYTE_STREAM_SPLIT, ZSTD
631-tst.dp	1.00	1.00	1.05	1.05	1.11
density.dp	1.00	1.16	2.05	2.07	2.28
diffusivity.dp	1.00	1.00	1.04	1.04	1.13
msg_bt.dp	1.00	1.00	1.11	1.11	1.37
msg_bt.sp	1.00	1.00	1.14	1.13	1.55
msg_lu.dp	1.00	1.00	1.05	1.05	1.28
msg_lu.sp	1.00	0.99	1.07	1.06	1.43
msg_sp.dp	1.00	1.00	1.06	1.06	1.31
msg_sp.sp	1.00	1.00	1.11	1.11	1.60
msg_sppm.dp	1.00	1.27	6.98	6.99	8.04
msg_sppm.sp	1.00	1.38	9.87	9.48	10.92
msg_sweep3d.dp	1.00	1.00	1.68	1.61	2.99
msg_sweep3d.sp	1.00	1.00	4.68	4.49	7.39
num_brain.dp	1.00	1.00	1.06	1.06	1.26
num_brain.sp	1.00	0.99	1.13	1.12	1.40
num_comet.dp	1.00	1.04	1.16	1.16	1.38
num_comet.sp	1.00	1.04	1.15	1.13	1.35
num_control.dp	1.00	1.00	1.06	1.06	1.14
num_control.sp	1.00	0.99	1.08	1.07	1.21
num_plasma.dp	1.00	4.50	39.07	210.82	32.32
num_plasma.sp	1.00	2.27	51.88	128.36	63.07
obs_error.dp	1.00	1.00	1.52	1.51	1.33
obs_error.sp	1.00	0.99	1.30	1.27	1.51
obs_info.dp	1.00	1.00	1.18	1.17	1.46
obs_info.sp	1.00	1.00	1.20	1.13	1.67
obs_spitzer.dp	1.00	1.00	1.18	1.18	1.34
obs_spitzer.sp	1.00	1.00	1.15	1.15	1.32
obs_temp.dp	1.00	0.99	1.04	1.03	1.13
obs_temp.sp	1.00	0.97	1.08	1.04	1.20
pressure.dp	1.00	1.00	1.06	1.05	1.24
sample_r_B_0.5_26	1.00	1.00	1.13	1.13	1.42
sample_r_B_1.0_26	1.00	1.00	1.18	1.18	1.89
sample_r_B_1.5_26	1.00	1.00	1.19	1.18	2.35
velocityx.dp	1.00	1.00	1.05	1.05	1.22
velocityy.dp	1.00	1.00	1.05	1.05	1.23
velocityz.dp	1.00	1.00	1.05	1.05	1.23
viscosity.dp	1.00	1.00	1.04	1.04	1.13

Table 1: Compression ratio.

For the tables on write and read throughput we only highlight the best combination among all combinations which utilize ZSTD since using the PLAIN encoding would almost always be faster without sufficient data size reduction.

Test case	PLAIN, UNCOMPRESSED	DICTIONARY, UNCOMPRESSED	PLAIN, ZSTD	DICTIONARY, ZSTD	BYTE_STREAM_SPLIT, ZSTD	BYTE_STREAM_SPLIT (simd), ZSTD
631-tst.dp	666.45	662.63	242.23	240.27	273.02	367.87
density.dp	979.02	816.06	187.14	205.79	216.52	279.75
diffusivity.dp	1014.98	939.00	308.95	308.21	290.47	407.58
msg_bt.dp	1042.19	968.60	269.43	265.56	216.98	266.65
msg_bt.sp	664.31	587.75	186.11	181.55	183.29	214.09
msg_lu.dp	1066.51	1009.92	312.98	302.60	312.37	459.65
msg_lu.sp	662.73	574.86	249.46	238.09	271.33	335.10
msg_sp.dp	922.78	715.09	259.56	259.30	309.74	449.79
msg_sp.sp	659.17	616.32	248.98	238.40	254.24	306.09
msg_sppm.dp	756.61	730.47	343.89	320.52	340.36	518.74
msg_sppm.sp	669.02	444.33	486.35	343.83	386.70	488.87
msg_sweep3d.dp	1019.32	904.46	244.97	240.50	350.20	536.11
msg_sweep3d.sp	663.60	529.43	467.08	369.72	389.64	506.63
num_brain.dp	1030.36	902.07	291.15	284.70	289.37	416.41
num_brain.sp	645.77	566.73	250.19	236.73	271.02	342.08
num_comet.dp	958.80	889.52	278.79	277.54	222.44	269.95
num_comet.sp	672.38	497.29	230.55	207.92	199.38	233.37
num_control.dp	1011.34	955.87	312.95	308.54	285.15	368.21
num_control.sp	670.37	575.13	256.84	239.54	220.13	270.10
num_plasma.dp	1038.95	349.15	953.69	353.91	489.76	773.15
num_plasma.sp	641.08	175.13	657.18	182.47	449.83	623.11
obs_error.dp	1113.01	837.76	193.72	184.74	221.73	268.88
obs_error.sp	669.59	456.10	153.36	142.15	220.86	265.67
obs_info.dp	976.16	600.87	282.06	239.79	284.32	352.80
obs_info.sp	624.20	193.73	230.97	133.87	239.50	289.50
obs_spitzer.dp	1087.83	1006.39	152.93	151.79	198.66	236.41
obs_spitzer.sp	706.20	567.29	198.28	186.71	192.01	206.72
obs_temp.dp	1117.30	773.15	317.25	278.51	292.93	380.72
obs_temp.sp	673.55	446.60	276.26	220.40	246.69	268.62
pressure.dp	1030.56	950.30	312.24	304.81	257.71	346.14
sample_r_B_0.5_26	699.79	699.99	310.29	303.09	309.61	446.33
sample_r_B_1.0_26	1036.02	946.65	310.62	299.16	343.73	516.48
sample_r_B_1.5_26	1075.57	959.43	303.29	297.11	325.28	477.82
velocityx.dp	986.68	900.63	311.35	304.22	288.92	404.59
velocityy.dp	956.68	921.93	307.17	303.43	292.39	412.21
velocityz.dp	976.80	918.85	299.51	283.30	263.76	402.79
viscosity.dp	967.03	913.38	312.07	301.53	291.87	406.82

Table 2: Write throughput (MiB/s).

Test case	PLAIN, UNCOMPRESSED	DICTIONARY, UNCOMPRESSED	PLAIN, ZSTD	DICTIONARY, ZSTD	BYTE_STREAM_SPLIT, ZSTD	BYTE_STREAM_SPLIT (simd), ZSTD
631-tst.dp	1654.05	1742.21	402.99	405.07	722.66	947.63
density.dp	1255.46	1266.86	319.56	377.29	467.04	562.74
diffusivity.dp	1192.08	1238.06	401.67	403.32	697.80	928.90
msg_bt.dp	1382.46	1267.48	426.49	422.52	807.28	1021.89
msg_bt.sp	956.92	875.22	345.00	345.76	613.01	658.61
msg_lu.dp	1357.94	1624.57	411.65	404.86	838.03	1175.71
msg_lu.sp	851.59	864.02	327.57	342.96	669.95	788.11
msg_sp.dp	1651.50	1233.01	390.37	397.24	775.42	1056.00
msg_sp.sp	820.33	821.72	325.41	325.78	573.01	634.85
msg_sppm.dp	1310.68	1269.45	548.17	595.19	746.45	989.44
msg_sppm.sp	857.31	890.44	649.85	665.57	770.87	851.43
msg_sweep3d.dp	1627.91	1663.15	399.01	392.23	856.98	1185.70
msg_sweep3d.sp	931.48	947.00	643.64	601.49	808.15	929.43
num_brain.dp	1286.43	1419.75	391.99	396.60	751.55	1080.90
num_brain.sp	834.35	873.00	338.07	336.87	677.45	809.10
num_comet.dp	1388.81	1504.72	432.24	442.78	695.53	818.57
num_comet.sp	914.00	886.60	357.53	359.95	593.86	689.64
num_control.dp	1364.60	1447.99	424.03	415.52	715.34	866.90
num_control.sp	866.16	897.43	339.74	337.50	561.55	669.49
num_plasma.dp	1617.08	1890.75	1438.24	1781.37	1070.26	1556.89
num_plasma.sp	953.99	793.81	859.63	784.97	838.27	1000.24
obs_error.dp	1539.73	1525.38	441.00	440.18	416.47	465.85
obs_error.sp	929.84	939.25	334.89	321.15	377.96	421.86
obs_info.dp	1639.64	1710.24	463.74	460.83	802.17	1021.37
obs_info.sp	831.29	876.82	348.35	358.08	663.85	772.02
obs_spitzer.dp	1658.96	1378.53	391.32	399.79	428.71	472.53
obs_spitzer.sp	899.52	947.95	361.62	363.67	385.85	394.46
obs_temp.dp	1761.93	1715.85	437.33	441.44	693.69	863.61
obs_temp.sp	1028.34	1003.78	382.53	375.26	621.78	657.61
pressure.dp	1187.55	1190.90	401.38	401.75	654.12	847.73
sample_r_B_0.5_26	1314.44	1250.38	403.62	395.99	830.14	1126.17
sample_r_B_1.0_26	1215.07	1211.12	422.33	412.37	811.18	1078.04
sample_r_B_1.5_26	1250.71	1218.96	416.07	413.73	725.30	1007.63
velocityx.dp	1193.14	1139.29	399.33	402.33	766.69	1002.36
velocityy.dp	1197.03	1265.34	397.34	396.70	745.89	1018.97
velocityz.dp	1242.33	1226.58	381.63	373.70	659.84	1007.94
viscosity.dp	1186.70	1217.85	401.59	395.49	721.20	949.79

Table 3: Read throughput (MiB/s).

Next I examine the change of compression ratio, write throughput and read throughput as the entropy (bits/element) varies. The data is generated by sampling a Gaussian distribution and changing the variance to create files with different zero-order entropy.

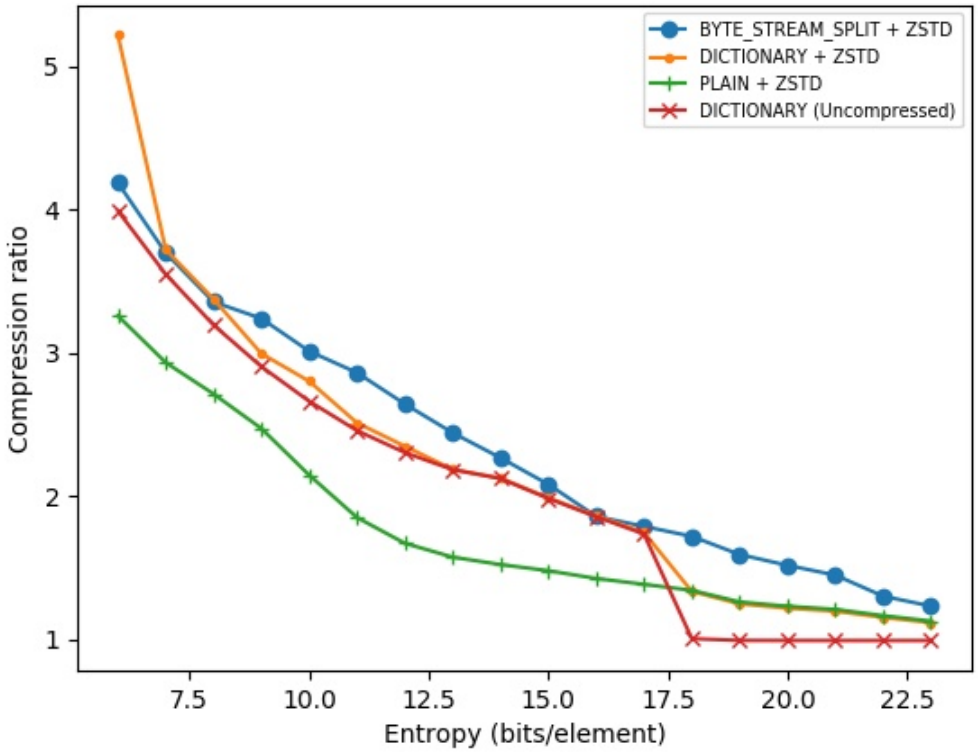


Figure 1: Compression ratio for data with varying entropy for a discrete Gaussian distribution.

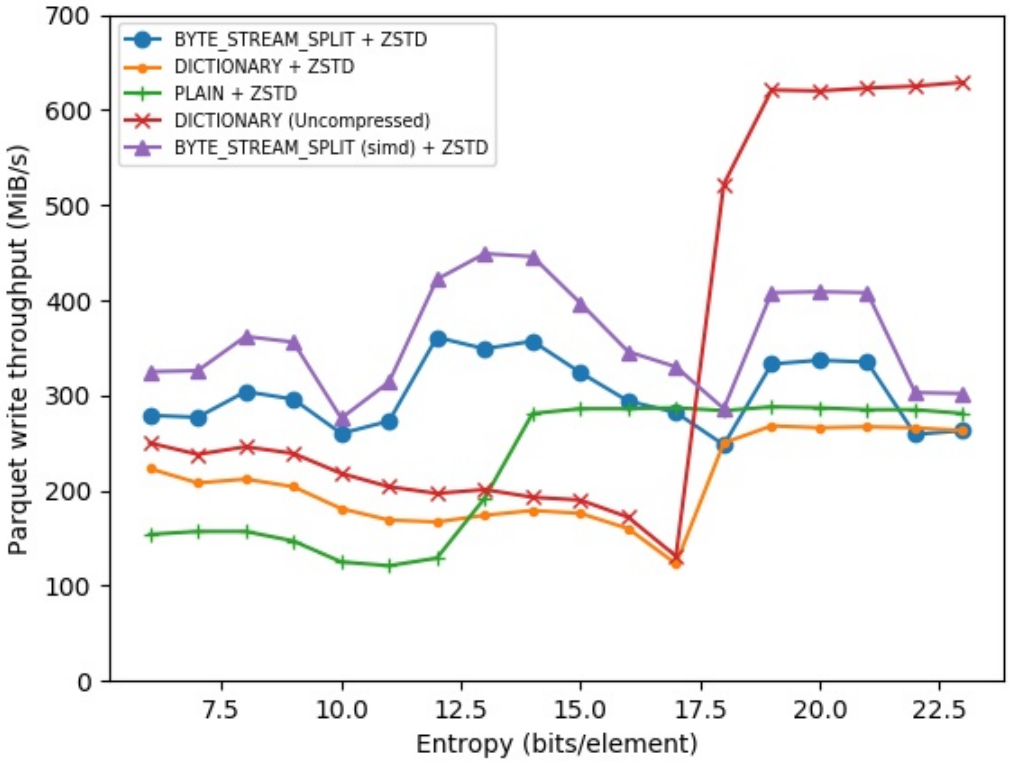


Figure 2: Write throughput for data with varying entropy for a discrete Gaussian distribution.

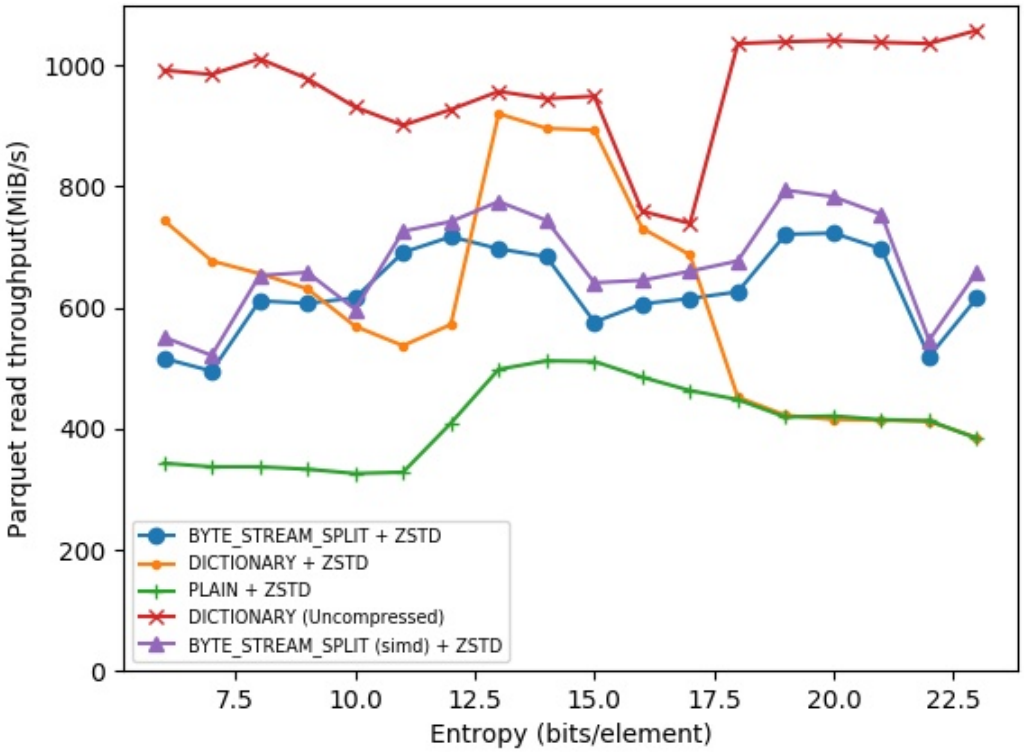


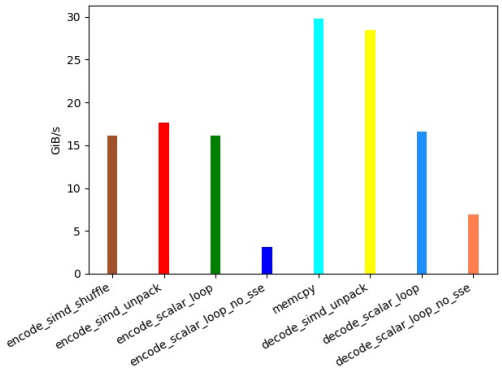
Figure 3: Read throughput for data with varying entropy for a discrete Gaussian distribution.

The sudden change of ratio, write- and read-throughput for the dictionary encoding comes from its sudden decission to use the plain encoding if data doesn't compress very well.

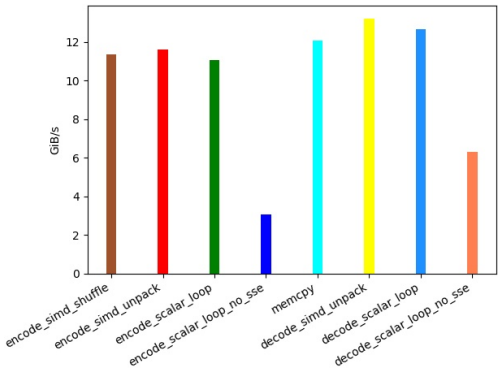
2 The BYTE_STREAM_SPLIT implementation

GCC 7.2.0 did not auto-vectorize the encoder and decoder loop for the Arrow release build, so it was necessary to write a custom implementation using SSE extensions up to SSE4.2. I investigated two approaches: one based on *shuffle instructions* and one based on *packing and swizzle instructions*. The second approach turned out to be faster and more general. This made it easy to implement an encoder and decoder for both single-precision and double-precision data. The implementation of all is available here. Note that the comparison of different possible implementation are done in a standalone program not part of Apache Arrow.

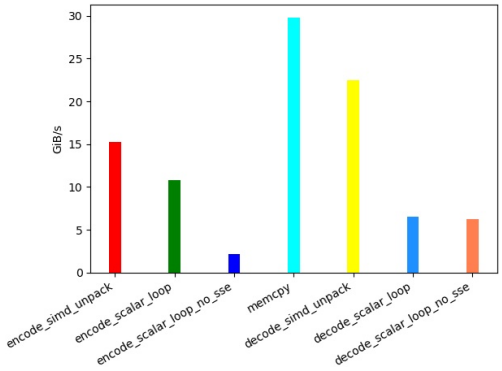
In the comparison I evaluate how the different approaches perform on data which is mostly present in the cache and data which is purposely evicted from the cache. In the first case the data is of size 1MiB which fits into the LLC. Since caches offer lower latency and higher bandwidth than main memory, the number of operations per loaded byte have a significant impact on performance. In the second case the data is of size 32MiB and is purposely evicted using the *clflush* instruction. The number of operation have a significantly lower impact since memory bandwidth is lower and latency is huge.



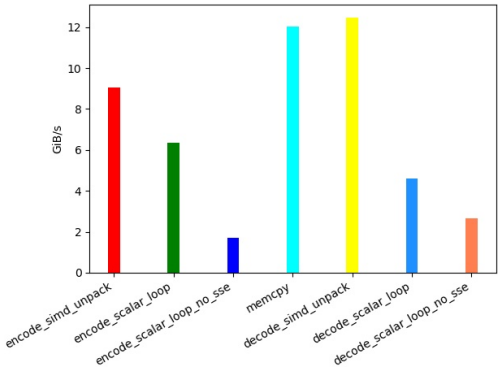
(a) Single-precision encoder/decoder comparison. 1MiB input present in cache.



(b) Single-precision encoder/decoder comparison. 32MiB input evicted from cache.



(c) Double-precision encoder/decoder comparison. 1MiB input present in cache.



(d) Double-precision encoder/decoder comparison. 32MiB input evicted from cache.

Figure 4: Comparison of single and double precision encoder/decoders using a cached 1MiB input and an evicted 32MiB input.

In Subfigure 4a the SIMD-unpack encoder performs the best among all encoders but offers a little more over half the performance of a memcpy. However, the decoder only needs a combination of very few unpack instructions and thus performs just a little slower than memcpy. It could be that the encoder can be implemented more efficiently. In Subfigure 4b all three encoders are perform very close to memcpy. Again the SIMD-unpack encoder performs the best. The SIMD-unpack decoder performs slightly better than memcpy. It could be due to the HW cache-line prefetcher but I cannot be sure in that without disabling it and measuring performance again.

In Subfigure 4c the SIMD-unpack encoder outperforms significantly all other encoders but similarly offers only half the performance of memcpy. We examined the produced machine code for the scalar decoder and interestingly the compiler did vectorize the implementation but very inefficiently. Thus, the regular scalar decoder and the decoder with explicitly disabled vectorization have the same performance. The SIMD-unpack encoder and decoder have to perform few more operation per loaded byte than the single precision implementation. Thus, the implementation has a lower throughput than that for single-precision data. In Subfigure 4d the SIMD-unpack encoder is again the fastest among all encoders but slower than memcpy. From Subfigure **??** we can see that the single-precision encoder is almost as fast as memcpy. The difference between the single and double precision encoders comes from the fact that the double precision encoders performs more operations per loaded byte. The SIMD-unpack decoder is also the fastest among all decoders and slightly faster than memcpy.

References

[1] https://userweb.cs.txstate.edu/ burtscher/research/datasets/FPdouble/

[2] https://userweb.cs.txstate.edu/ burtscher/research/datasets/FPsingle/

[3] Lindstrom, Peter. "Fixed-rate compressed floating-point arrays." IEEE transactions on visualization and computer graphics 20.12 (2014): 2674-2683.

[4] https://sdrbench.github.io/