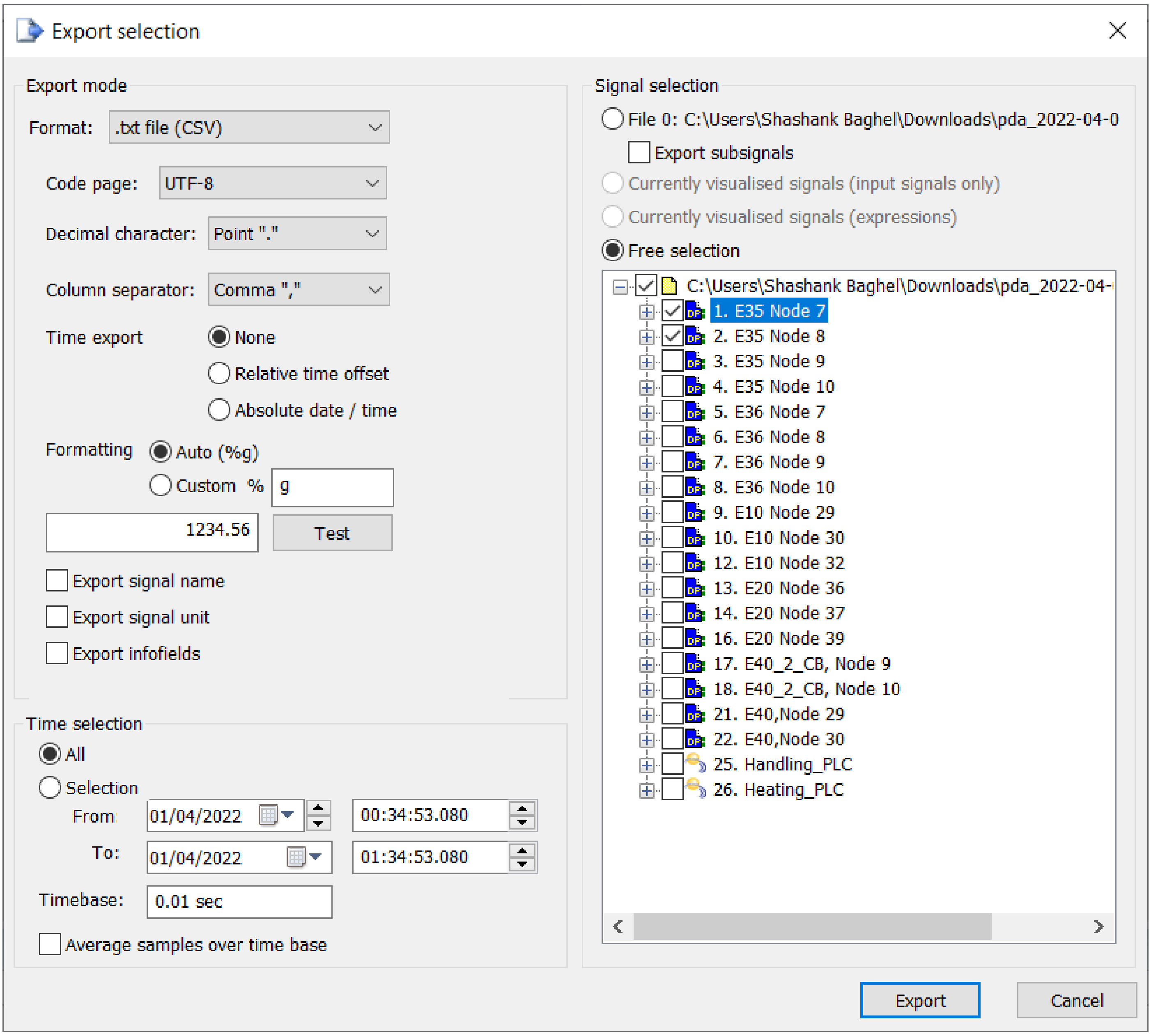
Benchmarking and Performance of our CSV/TSV Parquet tool

Test #1

I have done some benchmarking on the performance of our CSV/TSV to parquet conversion tool, like the time for conversion, and compression ratio using various compression formats also with various types of data encoding.

In this, I have used the file `*pda_2022-04-01_00.34.53.dat*` from our gdrive archive. Using the ibaAnalyser I exported the raw data to CSV using the option shown below:



The format I used is UTF-8 instead of the default ANSI **(and I recommend using UTF-8 over the default ANSI),** using the decimal character Point (.) not of the system and a comma for the column separator. Exported signals are **E35 Node 7 and E35 Node 8.**

And exporting it to a defined location, and the size of the exported file in CSV /TXT form is about 271 MB (284,446,720 bytes).

Encoding Results

We have different types of encoding available in parquet like plain, rle, bit-packed, delta-binary-packed, delta-length-byte-array, delta-byte-array, rle-dictionary. But seems like the encoding depends on the type of data present in the columns. I need to study more about how this encoding works in detail.

Test config #1

PATH_TO_CSV="/input.csv"

PATH_TO_PQT="/output.parquet"

DICTIONARY=true

Size Difference:

Original Size: 271 MB (284,446,720 bytes)

Parquet Size: 1.51 MB (1,589,248 bytes) **we can see a 99.44% decrease in file size here.**

Program/Binary Benchmark:

Benchmark 1:

Time (mean ± σ): 101.665 s ± 4.105 s [User: 99.941 s, System: 1.553 s]

Range (min … max): 98.046 s … 106.125 s 3 runs

The above benchmark is obtained using **[hyperfine]** which is an open-source command-line benchmarking tool. Consists of the benchmarks for 3 runs of our program for the same config #1. It took 101.665 seconds ± 4.105 s for 3 runs, although this can be improved by doing some using compiler optimisation like native CPU optimisation and LTO.

Test config #2

PATH_TO_CSV="/input.csv"

PATH_TO_PQT="/output.parquet"

DICTIONARY=true

ENCODING="delta-byte-array"

Size Difference:

Original Size: 271 MB (284,446,720 bytes)

Parquet Size: 1.51 MB (1,589,248 bytes) **again we can see a 99.44% decrease in file size here.**

Program/Binary Benchmark:

Benchmark 1:

Time (mean ± σ): 93.982 s ± 3.848 s [User: 92.352 s, System: 1.524 s]

Range (min … max):   89.987 s … 97.664 s   3 runs

Comparison between config #1 and config #2

Benchmark 1:

Time (mean ± σ):   92.199 s ± 1.193 s   [User: 90.660 s, System: 1.475 s]

Range (min … max):   91.078 s … 94.225 s   5 runs

Benchmark 2:

Time (mean ± σ):   92.116 s ± 0.327 s   [User: 90.607 s, System: 1.488 s]

Range (min … max):   91.712 s … 92.603 s   5 runs

Statistics:

```
Command 'csvpqt --dictionary i.csv o.parquet'
  runs:          5
  mean:     92.199 s
  stddev:    1.193 s
  median:   91.941 s
  min:      91.078 s
  max:      94.225 s

  percentiles:
    P_05 .. P_95:    91.204 s .. 93.789 s
    P_25 .. P_75:    91.707 s .. 92.044 s  (IQR = 0.337 s)

Command 'csvpqt --dictionary -e delta-byte-array i.csv o.parquet'
  runs:          5
  mean:     92.116 s
  stddev:    0.327 s
  median:   92.028 s
  min:      91.712 s
  max:      92.603 s

  percentiles:
    P_05 .. P_95:    91.773 s .. 92.527 s
    P_25 .. P_75:    92.018 s .. 92.220 s  (IQR = 0.202 s)
```

Summary: 'csvpqt with dictionary and encoding as delta-byte-array' ran 1.00 ± 0.01 times faster than 'csvpqt with dictionary without encoding'

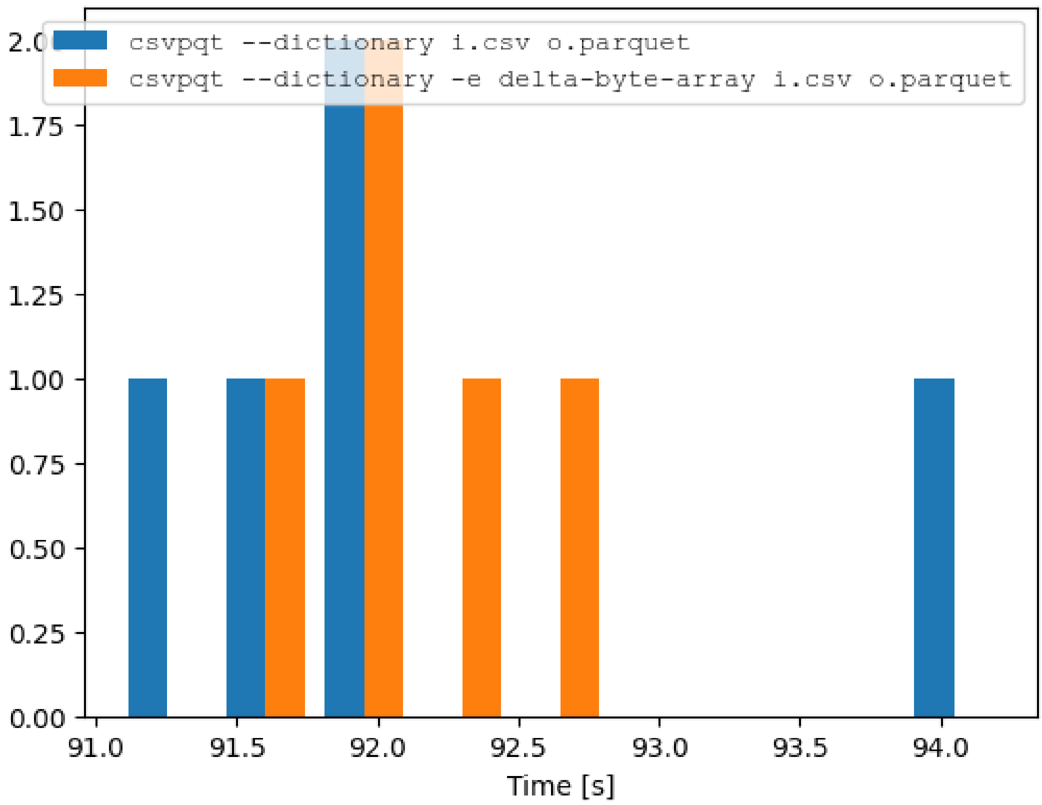


Figure: Comparison of configs using histogram visualisation

Compression Results

Here also we have different types of compression algorithms available for us to use like uncompressed, snappy, gzip, lzo, brotli, lz4, zstd etc.

Compression algorithm like zstd performs well overall whether compression or decompression speed. But instead of using zstd we can use the format according to our need, if we need to regularly access the file then it's best to choose an algorithm which has a higher decompression speed or leave it uncompressed. And if we don't need to access that file regularly then we can go for something which has a higher compression ratio like zstd.

ZSTD: Test config #1

PATH_TO_CSV="/input.csv"

PATH_TO_PQT="/output.parquet"

DICTIONARY=true

COMPRESSION="zstd"

Size Difference:

Original Size: 271 MB (284,446,720 bytes)

Parquet Size: 800 KB (819,200 bytes) **we can see a 99.71% decrease in file size here.**

Program/Binary Benchmark:

Benchmark 1:

Time (mean ± σ): 96.013 s ± 1.833 s [User: 94.280 s, System: 1.614 s]

Range (min … max): 94.699 s … 98.107 s 3 runs

LZ4: Test config #1

PATH_TO_CSV="/input.csv"

PATH_TO_PQT="/output.parquet"

DICTIONARY=true

COMPRESSION="lz4"

Size Difference:

Original Size: 271 MB (284,446,720 bytes)

Parquet Size: 972 KB (995,328 bytes) **we can see a 99.65% decrease in file size here.**

Program/Binary Benchmark:

Benchmark 1:

Time (mean ± σ): 94.642 s ± 1.250 s [User: 93.033 s, System: 1.586 s]

Range (min … max): 93.264 s … 95.703 s 3 runs

Algorithm Benchmarks

Source: Meta

For reference, several fast compression algorithms were tested and compared on a desktop running Ubuntu 20.04 (`Linux 5.11.0-41-generic`), with a Core i7-9700K CPU @ 4.9GHz, using [lzbench], an open-source in-memory benchmark by @inikep compiled with [gcc] 9.3.0, on the [Silesia compression corpus].

Compressor name	Ratio	Compression	Decompress.
zstd 1.5.1 -1	2.887	530 MB/s	1700 MB/s
[zlib] 1.2.11 -1	2.743	95 MB/s	400 MB/s
brotli 1.0.9 -0	2.702	395 MB/s	450 MB/s
zstd 1.5.1 --fast=1	2.437	600 MB/s	2150 MB/s
zstd 1.5.1 --fast=3	2.239	670 MB/s	2250 MB/s
quicklz 1.5.0 -1	2.238	540 MB/s	760 MB/s
zstd 1.5.1 --fast=4	2.148	710 MB/s	2300 MB/s
lzo1x 2.10 -1	2.106	660 MB/s	845 MB/s
[lz4] 1.9.3	2.101	740 MB/s	4500 MB/s
lzf 3.6 -1	2.077	410 MB/s	830 MB/s
snappy 1.1.9	2.073	550 MB/s	1750 MB/s

Note: The Encoding and Compression benchmarks and testing have been done on Ryzen 5 3500U (2.10GHz and Core 4) CPU with 8GB of memory (DDR4 2400MHZ) on a windows 10 OS. Using rustc 1.64 with a release build.