OnPair: A Viable String Encoding for Apache Parquet

Author: Prateek Gaur

Suggested by : Julien Le Dem, Russell Spitzer, Andrew Lamb

Other file formats that support it : Vortex, anyone else?

Date : Jul 22, 2026

Paper : https://arxiv.org/abs/2508.02280

Motivation. This report makes the case for OnPair — the short-string codec of Gargiulo & Venturini (OnPair: Short Strings Compression for Fast Random Access, arXiv:2508.02280) — as a string encoding for Apache Parquet string columns. We measure it head-to-head against **FSST** (the string encoding currently proposed for Parquet), and against the general-purpose block compressors **zstd -1** and **lz4**, on the three axes that matter for a columnar format: compression ratio, decompression speed (with per-row random access), and compression speed. The takeaway: OnPair's ratio win is real but confined to high-cardinality text, where it beats the best native page by +8–49% on 14 of 30 columns; on low-cardinality columns stock RLE_DICTIONARY already matches it, and on monotonic patterned columns DELTA_BYTE_ARRAY+zstd beats it by orders of magnitude. The decode win is the broad one: OnPair decodes a median 4.7–5.9x faster than any zstd page and 2.4x faster than an lz4 page, while keeping per-row random access.

TL;DR

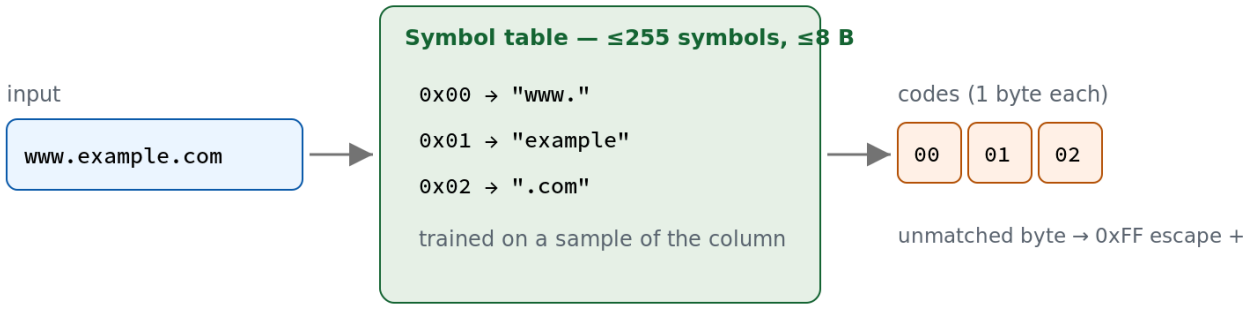
- OnPair wins ratio on high-cardinality text — comments, product names, real book reviews/titles/headlines/tweets, real ClickBench URL and search-phrase columns — beating the best Parquet page by a median +33% (range +8% to +49%) on 14 of 30 columns. That is the bulk of real string data.
- OnPair's broadest win is decode, not ratio. Median 5.27x faster than PLAIN+zstd (4.7–5.9x against every zstd page, losing on 0 of 30) and 2.43x faster than PLAIN+lz4, with per-row random access that no generic codec offers.
- **zstd(1) still wins on patterned-sequential IDs and near-random data** — OnPair is the wrong tool there. Per-shape rule in Findings.

How the codecs work

All four string-specific approaches replace bytes with small fixed-width codes and keep per-row random access (any row decodes on its own); they differ in what a code stands for and how large the vocabulary is. zstd/lz4 are whole-block LZ compressors — great at long repeats, but a block decompresses as a unit.

FSST — FAST STATIC SYMBOL TABLE

Builds, from a sample, a table of up to 255 symbols each 1–8 bytes. Each run of bytes becomes the 1-byte code of the longest matching symbol; a byte in no symbol is emitted as an escape + literal. Fast to build, but short symbols and a tiny code space leave compression ratio on the table.

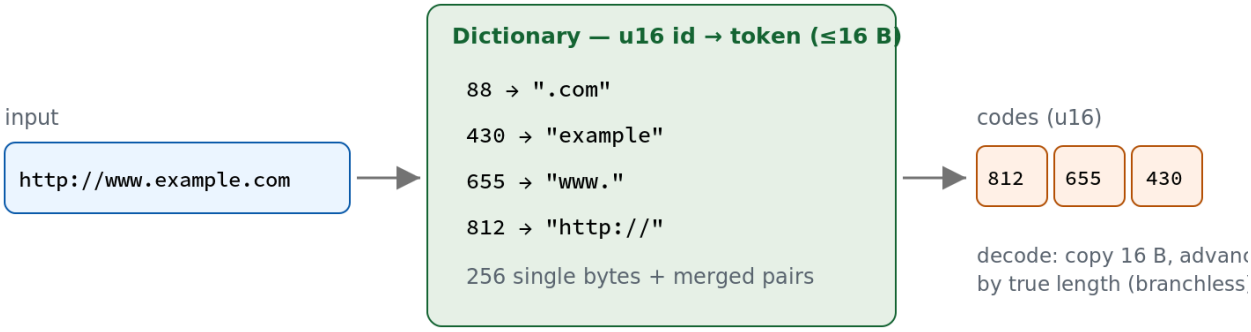


Algorithm 1 FSST-decoding

```
void decode(uint8_t*& in, uint8_t*& out,
            uint64_t sym[255], uint8_t len[255]) {
  uint8_t code = *in++;
  if (code != 255) {
    *((uint64_t*)out) = sym[code];
    out += len[code];
  } else { // escape code
    *out++ = *in++;
  }
}
```

ONPAIR16 — THE PAPER'S 16-BYTE-TOKEN VARIANT

The "16" in OnPair16 is a 16-byte cap on dictionary entries (paper §3.2.2) that enables the fixed-width SIMD decode — not the code width. Keeps a dictionary of 1–16 byte tokens: the 256 single bytes plus frequent adjacent pairs found in one incremental merge pass over a sample. Encoding is a greedy longest-prefix match emitting 16-bit codes (up to 65,536 tokens). Decoding copies a fixed 16 bytes per code and advances by the token's true length — one branchless 128-bit store, so it decodes fast while staying random-access. Longer tokens (16 vs 8 B) and a far larger code space give a better ratio than FSST.



Algorithm 3 Fast unbounded token decompression

```
1: function DECOMPRESS_TOKEN(token_id, data, offsets, buff)
2:     start ← offsets[token_id]
3:     end ← offsets[token_id + 1]
4:     length ← end - start
5:     copy(buff, data + start, 16)                ▷ Exploit SIMD
6:     if length > 16 then
7:         copy(buff + 16, data + start + 16, length - 16)
8:     end if
9:     return length
10: end function
```

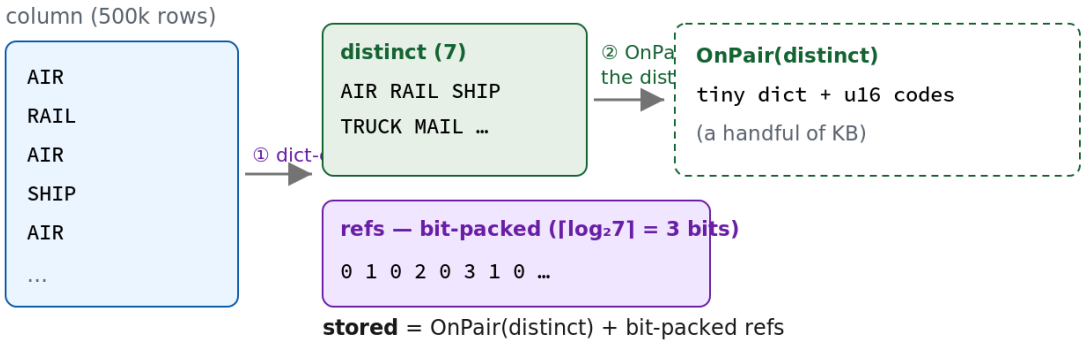
ONPAIR-AUTO — PER-COLUMN CODE WIDTH

The dictionary budget (max tokens = 2^bits) need not be fixed at 16. OnPair-auto picks it per column at encode time to minimize size, then stores the resulting code width (ceil(log2 tokens)) in the column. The decode path is unchanged and random access is preserved. It never hurts ratio versus a fixed budget and usually helps — and often decodes faster, because a smaller dictionary stays in cache. The stored width chosen per column is listed in the Columns-under-test table.

DICT+ONPAIR — DICTIONARY ENCODING, THEN ONPAIR

OnPair's ≤16-byte tokens can't capture repetition of whole values. So first dictionary-encode the column (distinct values + bit-packed integer references — exactly Parquet RLE_DICTIONARY / Arrow DictionaryArray), then OnPair just the distinct values. A cascade: the references collapse whole-value repetition; OnPair shrinks the small distinct set. Decisive for low-cardinality columns.

One codec, two binaries. Both benchmarks measure this layout — the family binary calls it OnPair16-dedup and the cascade binary calls it DICT+OnPair — and they build the identical payload: OnPair over the distinct set, plus bit-packed distinct lengths and a bit-packed reference column. This report uses DICT+OnPair throughout. The two are worth one sentence because they are an accidental control: compressed size agrees on all 30 corpora to within 0.021% (the cascade row also pays its 32-byte page header), while throughput differs by a median 1.06x in the family binary's favour. The plain OnPair16 row, measured by both binaries too, shows the same 1.06x. So the cross-binary throughput offset is the harness, not the codec — which is why a speed measured by one binary must never be read against a speed measured by the other.



In one line each. **FSST** = ≤8-byte symbols → 1-byte codes (fast, modest ratio). **OnPair16** = ≤16-byte tokens → 16-bit codes (better ratio, fastest decode). **OnPair16-dedup** = dictionary-encode first, then OnPair the distinct values (wins low-cardinality). **zstd/lz4** = whole-block LZ (best at long/whole-value repeats; no random access).

Setup.

All ten pages and codecs in the result tables run in one C++ process — FSST is the exact vendored sources from Arnav Balyan's FSST branch, which this work is built on top of; OnPair is a C++ implementation of the paper's algorithm, roundtrip-verified on every corpus. 30 corpora: 18 TPC-H string columns, the OnPair paper's four real-world datasets (Amazon book reviews and titles, ABC news headlines, Sentiment140 tweets), two real ClickBench columns (URL and search phrase) plus a synthetic ClickBench URL column, a synthetic JSON/variant column, two hex-identifier columns (UUIDv4 and SHA-256 digests), and sorted variants of a comment column and the real URL column to test order sensitivity.

Single core (taskset -c 0), warm run, medians. Realistic bit-packed accounting: per-row lengths/offsets delta-bit-packed for FSST/zstd/lz4/OnPair, OnPair codes at their true code width, dedup's reference (index) column bit-packed — comparable column-reconstruction cost across all.

Columns under test

All corpora are 500,000 rows (TPC-H columns generated at the scale factor needed to reach 500k), except the two real ClickBench columns, which are the full 200,005 rows as published. The four real-world corpora are the OnPair paper's own datasets sampled to 500k rows). The result tables refer to these columns by name. The real-world set independently reproduces the paper's OnPair16 ratios (book_reviews 3.31 vs the paper's 3.28).

Column	Dataset	Description	Uncompressed MiB	Distinct values	OnPair-auto bits
o_comment	TPC-H	free-text comment	23.1	497,633	12
l_comment	TPC-H	free-text comment	12.6	452,869	12
c_comment	TPC-H	free-text comment	34.6	499,611	12
p_comment	TPC-H	free-text comment	6.4	277,396	12
s_comment	TPC-H	free-text comment	29.8	499,471	12
p_name	TPC-H	product name (multi-word)	15.6	499,988	13

c_name	TPC-H	patterned name — Customer#000…	8.6	500,000	10
s_name	TPC-H	patterned name — Supplier#000…	8.6	500,000	10
variant_json	Synthetic	synthetic JSON event objects	66.4	500,000	10
urls	Synthetic	synthetic URLs (ClickBench-style)	19.9	500,000	11
o_clerk	TPC-H	patterned ID — Clerk#000…	7.2	1,000	11
c_phone	TPC-H	phone number	7.2	499,997	12
c_address	TPC-H	near-random street address	11.9	500,000	12
s_address	TPC-H	near-random street address	11.9	500,000	12
p_type	TPC-H	type phrase (small vocabulary)	9.8	150	9
c_mktsegment	TPC-H	enum — market segment	4.3	5	9
o_orderpriority	TPC-H	enum — order priority	4.0	5	9
l_shipmode	TPC-H	enum — ship mode	2.0	7	9
p_brand	TPC-H	enum — brand	3.8	25	9
p_container	TPC-H	enum — container	3.6	40	9
book_reviews	Amazon	real book reviews (paper)	227.4	455,611	16
book_titles	Amazon	real book titles (paper)	24.7	482,913	16
news_headlines	ABC News	real news headlines (paper)	18.7	489,004	15
tweets	Sentiment140	real tweets (paper)	35.0	496,474	16

// not sure how but table formatting left this page empty.

Table 1 — Compression ratio

×, higher is better. Green = best in row. The first six columns are pages Parquet writes today, charged as libparquet writes them (no separate length array — see Caveats); the last four are the candidates, each charged a bit-packed per-row length array on top. All ten come from one binary, so every number in this table is comparable with every other.

Column	PLAIN+zstd	PLAIN+lz4	DLBA+zstd	DBA+zstd	DICT+zstd	DBA+lz4	FSST	OnPair16	OnPair-auto	DICT+OnPair
Free text										
o_comment	3.37	2.27	3.75	3.71	2.90	2.44	3.02	4.81	**5.31**	3.90
l_comment	2.83	1.94	3.27	3.21	2.38	2.18	2.65	3.96	**4.41**	3.05
c_comment	3.70	2.43	4.02	3.99	3.31	2.57	3.08	5.32	**5.90**	4.53
p_comment	2.22	1.51	2.61	2.55	2.18	1.77	2.28	2.98	**3.32**	2.58
s_comment	3.58	2.37	3.92	3.89	3.15	2.52	3.15	5.15	**5.70**	4.30
Names / multi-word										
p_name	3.16	2.00	3.30	3.23	2.59	2.10	2.86	4.61	**4.91**	3.46
c_name	47.54	3.47	88.07	**2840.91**	7.31	461.30	3.20	4.16	4.30	2.69
s_name	47.54	3.47	88.08	**2840.91**	7.31	461.30	3.20	4.16	4.30	2.69
Semi-structured										
variant_json	4.38	2.65	**4.39**	4.30	4.09	2.67	2.79	4.08	4.25	3.82
urls	4.47	2.87	4.50	4.97	3.60	3.22	3.48	4.04	**5.37**	3.29
Patterned / high-cardinality										
o_clerk	6.94	3.80	7.03	9.65	**11.93**	4.77	3.33	7.84	7.84	11.91
c_phone	2.06	1.15	**2.35**	2.17	1.56	1.16	2.05	1.79	2.03	1.38
c_address	1.13	0.86	**1.28**	**1.28**	1.03	0.96	1.01	0.96	1.22	0.88
s_address	1.13	0.86	**1.28**	**1.28**	1.03	0.96	1.01	0.95	1.22	0.88
Low-cardinality										
p_type	5.21	2.93	5.70	4.75	**22.53**	2.59	5.21	7.21	7.21	20.52
c_mktsegment	7.35	3.52	7.05	6.02	**26.81**	3.14	3.91	5.53	5.53	23.91
o_orderpriority	7.02	3.14	5.63	5.25	**25.01**	2.88	4.94	5.17	5.17	22.33
l_shipmode	3.50	1.83	3.59	3.23	11.36	1.92	3.12	2.85	2.85	**11.39**
p_brand	9.86	2.74	9.85	7.40	12.75	4.93	3.20	4.92	4.92	**12.77**
p_container	3.42	1.98	3.79	3.12	**10.49**	2.05	3.03	4.66	4.66	10.08
Real-world text (paper datasets)										
book_reviews	2.47	1.66	2.51	2.51	2.44	1.67	1.90	3.31	3.31	**3.34**
book_titles	1.77	1.23	1.89	1.88	1.67	1.29	1.63	**2.54**	**2.54**	2.31
news_headlines	2.09	1.33	2.30	2.30	1.86	1.45	1.84	2.97	**3.02**	2.50
tweets	1.88	1.27	1.99	1.99	1.77	1.33	1.71	**2.57**	**2.57**	2.38
Real web text (ClickBench columns)										
real_url	2.78	1.95	2.86	2.88	2.62	2.03	1.55	3.22	**3.25**	3.01
real_searchphrase	2.54	1.70	2.70	2.62	2.34	1.75	2.12	3.53	**3.62**	3.12
Identifiers (near-random)										
ids_uuid_v4	1.83	0.95	**1.95**	1.93	1.64	1.00	1.74	1.59	1.67	1.44
ids_sha256_hex	1.90	0.96	**2.00**	1.99	1.78	0.99	1.87	1.76	1.76	1.65
Sorted / clustered										
sorted_real_url	3.54	2.52	3.67	**3.90**	3.30	2.76	1.56	3.22	3.24	3.01
sorted_c_comment	4.20	2.75	4.46	4.97	3.69	3.29	3.23	5.06	**5.96**	4.34

Table 2 — Decompression speed

MiB/s, higher is better.

Column	PLAIN+zstd	PLAIN+lz4	DLBA+zstd	DBA+zstd	DICT+zstd	DBA+lz4	FSST	OnPair16	OnPair-auto	DICT+OnPair
Free text										
o_comment	899	1720	982	875	740	1539	4071	5090	**5956**	2483
l_comment	758	1330	820	718	611	1192	3662	4513	**5213**	1494
c_comment	934	1790	1014	927	634	1697	3993	5367	**6414**	2734
p_comment	542	866	605	509	528	759	3297	3764	**4050**	1430
s_comment	884	1643	994	905	768	1646	4160	5344	**6332**	2719
Names / multi-word										
p_name	812	1380	863	764	628	1215	4016	5536	**6024**	2143
c_name	1090	1916	1337	1430	780	1446	4677	**6067**	4417	2735
s_name	1086	1867	1326	1441	780	1440	4576	**6099**	4403	2744
Semi-structured										
variant_json	1030	2109	1088	768	700	1176	3638	**4380**	3637	2338
urls	1059	2166	1111	1024	798	1720	4642	**5603**	5426	2992
Patterned / high-cardinality										
o_clerk	884	1414	997	904	3952	1035	4870	11341	**11428**	10784
c_phone	548	1300	660	458	418	792	**3024**	1967	2972	1515
c_address	393	**2405**	413	385	336	1952	1473	1313	1711	918
s_address	392	**2330**	413	385	338	1974	1474	1319	1729	927
Low-cardinality										
p_type	856	1442	1001	731	4430	1064	**8340**	7445	7328	7296
c_mktsegment	625	976	653	506	2160	697	6407	6693	6606	**6740**
o_orderpriority	522	684	571	454	1305	545	**9687**	6286	6258	6094
l_shipmode	300	430	365	244	786	305	**5909**	4456	4465	3091
p_brand	557	951	574	482	2214	553	5619	8095	**8446**	6092
p_container	404	624	474	335	1019	439	5109	5580	**5586**	5526
Real-world text (paper datasets)										
book_reviews	868	2151	933	836	653	1911	2402	**3009**	3007	2418
book_titles	650	1744	718	666	565	1593	2177	**2460**	2455	1449
news_headlines	729	1786	798	665	627	1299	2612	2957	**3010**	1842
tweets	684	1782	744	690	511	1667	2296	2338	**2339**	1579
Real web text (ClickBench columns)										
real_url	884	1932	977	910	741	1911	2048	3277	**3300**	2119
real_searchphrase	805	1627	871	742	665	1334	2845	3417	**3545**	2189
Identifiers (near-random)										
ids_uuid_v4	630	2148	754	679	520	2039	**2479**	1546	1603	1196
ids_sha256_hex	755	**3581**	839	779	680	3216	2633	2337	2334	2020
Sorted / clustered										
sorted_real_url	1028	2116	1140	1160	836	2315	2078	**3607**	3421	2262
sorted_c_comment	978	1826	1053	753	659	1061	4175	5852	**6591**	2891

Table 3 — Compression speed

MiB/s, higher is better. Best OnPair = the better of OnPair-auto / OnPair16-dedup per column (by ratio); † = dedup chosen. Green = best of FSST / zstd(1) / lz4 / Best OnPair.

Column	PLAIN+zstd	PLAIN+lz4	DLBA+zstd	DBA+zstd	DICT+zstd	DBA+lz4	FSST	OnPair16	OnPair-auto	DICT+OnPair
Free text										
o_comment	266	385	321	293	133	**438**	271	131	142	103
l_comment	253	**364**	281	250	99	351	275	111	117	84
c_comment	228	**309**	242	212	136	278	288	146	162	117
p_comment	200	**264**	233	198	85	254	209	91	89	73
s_comment	279	371	332	278	143	**404**	285	141	156	103
Names / multi-word										
p_name	237	361	278	250	110	**399**	244	134	137	102
c_name	529	751	550	796	94	**800**	433	142	184	71
s_name	535	752	550	**799**	92	**799**	433	143	182	68
Semi-structured										
variant_json	316	**405**	316	271	191	330	384	83	215	75
urls	347	481	389	397	148	**530**	348	107	152	92
Patterned / high-cardinality										
o_clerk	454	724	434	541	**926**	545	390	125	125	568
c_phone	227	**320**	308	252	66	296	312	42	68	31
c_address	569	**1014**	560	451	116	967	297	29	53	27
s_address	573	**1018**	563	454	116	971	290	30	54	28
Low-cardinality										
p_type	364	589	391	314	**931**	425	568	224	222	916
c_mktsegment	339	**554**	320	239	456	304	317	250	247	482
o_orderpriority	319	478	300	222	381	259	376	218	215	**525**
l_shipmode	185	**292**	205	131	190	157	186	118	118	218
p_brand	299	488	271	278	521	314	425	239	238	**848**
p_container	229	376	253	185	376	233	270	177	177	**381**
Real-world text (paper datasets)										
book_reviews	216	**274**	226	175	190	211	229	38	38	38
book_titles	252	366	287	266	136	**385**	216	30	30	29
news_headlines	243	333	281	262	118	**357**	223	40	42	36
tweets	211	**278**	217	192	130	243	221	29	29	28
Real web text (ClickBench columns)										
real_url	334	**486**	339	280	210	364	266	48	53	45
real_searchphrase	194	230	251	220	134	**263**	209	53	59	50
Identifiers (near-random)										
ids_uuid_v4	277	452	573	488	125	**736**	294	20	27	20
ids_sha256_hex	221	337	680	489	165	**782**	525	42	43	40
Sorted / clustered										
sorted_real_url	401	579	410	447	240	**597**	290	51	55	49
sorted_c_comment	241	332	257	306	141	**414**	304	104	132	88

Combining with a generic codec (zstd/lz4)

Cascade	Median ratio gain	Corpora improved
`OnPair-auto + zstd(1)`	**+2.3%**	on the 24 non-enum columns
`OnPair-auto (byte-aligned) + zstd(1)`	**−0.4%**	negative on 14 of 24
`FSST + zstd(1)`	**+25.8%**	30 of 30

- **Don't cascade over OnPair.** A +2.3% median gain does not pay for losing per-row random access, which is the entire reason to choose OnPair over a compressed page. The mechanism is that OnPair's output is a bit-packed code stream at the dictionary's true width — close to incompressible by construction, because bit-packing has already removed the byte-level regularity an LZ matcher looks for.
- **But "don't cascade" is not a general rule: over FSST it pays.** FSST emits byte-aligned 1-byte codes, which leaves cross-row redundancy on the table for zstd to find; it gains on 30 of 30 columns and sometimes overtakes OnPair-alone. Any claim in this space has to be made per codec.
- **Byte-aligning OnPair to feed the cascade does not fix it.** Storing u16 codes and u32 dictionary offsets instead of bit-packing them — the strongest form of the question, since bit-packing could be argued to handicap the cascade — measures a median −0.4%, negative on 14 of the 23 high-cardinality columns. The bytes given up to alignment exceed what zstd then recovers.
- **The decisive argument is not about cascades at all.** Where the OnPair cascade does pay (low-cardinality columns, median +43.1%), RLE_DICTIONARY beats OnPair-alone by a median +240% anyway. So the cascade question is moot on exactly the columns where cascading would help: those columns should be dictionary-encoded, which is what Parquet already does.

Findings

1. **OnPair gives the best ratio on high-cardinality text**: TPC-H comments and product names (the real-world corpora are findings 6) — beats FSST +31–73% as OnPair16 and +46–92% as OnPair-auto, and the best Parquet page +27..49% (c_comment: OnPair-auto 5.90x, OnPair16 5.32x, vs DLBA+zstd 4.02x, FSST 3.08x). Many-distinct values sharing ≤16-byte substrings are OnPair's target; its edge widened once codes are bit-packed at their true width. This holds on 14 of 30 columns, median +33%.
2. **OnPair decodes 4.7–5.9x faster than any zstd page and 2.43x faster than an lz4 page**, via a branchless fixed-16-byte gather per code — losing to the zstd pages on 0 of 30 columns and to lz4 on 4. These decode figures are bulk whole-column and unpack the bit-packed code stream, so they are consistent with the ratios; per-row random-access decode pays per-row unpacking and is slower. After decode-path tuning (narrowed gather-copy, width-templated loop, fused offset load) OnPair also decodes faster than FSST on high-cardinality text (o_comment/c_comment +46–61%, real reviews +25%); FSST leads on enums and short/patterned columns. This is the report's broadest result — unlike the ratio win, it is not confined to one column shape.
3. **On low-cardinality columns, stock RLE_DICTIONARY is already the answer — this is not an OnPair win**. OnPair16-dedup reaches 10–24x (c_mktsegment 23.9x, o_orderpriority 22.3x, p_type 20.5x), but plain RLE_DICTIONARY reaches 10.06–23.86x on the same columns (l_shipmode 11.36 vs 11.39, p_brand 12.75 vs 12.77) and DICT+zstd beats dedup by up to 12.1%. An earlier draft of this report claimed the 10–24x as an OnPair result against a zstd-in-isolation baseline of 5.7–6.2x; that comparison was against the wrong baseline. Dict-then-OnPair is still a way to keep random access here, but ship RLE_DICTIONARY.
4. **On patterned, near-sequential values Parquet's own DELTA_BYTE_ARRAY wins by orders of magnitude**. On c_name/s_name (Customer#000000001…), DBA alone reaches 10.05x and DBA+zstd reaches 2840.91x, against OnPair16 4.16x and OnPair-auto 4.30x. DBA stores a shared-prefix length plus a suffix, so an incrementing pattern costs a couple of bytes per row. Near-random addresses compress to ~1.2x for everyone, and OnPair loses there too (−4.6%).
5. **Encoding is OnPair's cost**: OnPair-auto encodes a median 2.6–2.8x slower than a native page plus zstd (2.0x slower than DICT+zstd), and a median 2.4x slower than FSST (train + greedy tokenize; this port uses open-addressed hash tables, not the paper's static perfect-hash — an implementation gap, not algorithmic). The encode columns do not include the 9–16-bit budget sweep: like every codec's parameter choice it runs in per-corpus setup, outside the timed region, so OnPair-auto and OnPair16 encode at close to the same speed (median 1.07x, and where they diverge it is auto's smaller dictionary training faster, not the sweep). A writer that had to pick the width online would pay up to eight trainings, so a cheaper picker with a verify-against-ceiling fail-safe is the obvious optimization. For write-once/read-many analytical data the trade (slower encode for best ratio + fastest decode) is usually right.

6. **The ratio win holds on real-world text — not a TPC-H artifact**: on the OnPair paper's own datasets (Amazon book reviews/titles, ABC news headlines, Sentiment140 tweets; 500k rows each) OnPair gives the best ratio, +50–74% over FSST and +28.8–34.7% over the best Parquet page (reviews 3.31x vs FSST 1.90x, DLBA+zstd 2.51x), and it decodes 3.1–3.8x faster than that page. Absolute ratios are lower than TPC-H's synthetic comments (real text is less redundant), but the margins hold and reproduce the paper (reviews 3.31 vs 3.28). FSST+/OnPair+ add nothing on this all-distinct text, so plain OnPair16/OnPair-auto is the pick.

Recommendation — pick per column shape

Realistic bit-packed accounting for our codecs; native pages charged as libparquet writes them.

Column shape	Use	Why
Free text & multi-word names (comments, product names)	**OnPair16 / OnPair-auto**	Best ratio (+27–49% vs the best native page, +46–92% vs FSST) and 3.2–4.1× the native page's decode
Real web text (URLs, search phrases, titles)	**OnPair-auto**	+13–35% vs the best native page, +56–110% vs FSST; decodes 24–105% faster
Low-cardinality enums (status, type, country)	**RLE_DICTIONARY (+ zstd if ratio matters more than decode)**	Already 10.06–23.86× alone; dict-then-OnPair only ties it. Use dict-then-OnPair only if you need OnPair's random-access decode
Patterned sequential IDs ("Customer#000…")	**DELTA_BYTE_ARRAY (+ zstd)**	DBA alone 10.05× and DBA+zstd 2840.91× vs OnPair 4.30× — not close
Sorted / clustered prefix-heavy columns (URLs, keys)	**DELTA_BYTE_ARRAY+zstd — or OnPair-auto if random access is required**	Sorting lifts the native page +35% (3.90×) past OnPair (3.24×); OnPair is order-invariant
Semi-structured JSON (variant / event payloads)	**DLBA+zstd or OnPair-auto — effectively a tie**	4.39× vs 4.25× (−3.1%); pick on decode, where OnPair is +58%
Short patterned (phone numbers)	**DLBA+zstd**	OnPair loses on ratio (2.03× vs 2.35×) though it decodes +93% faster
Hex identifiers (UUIDs, hashes, digests)	**DLBA+zstd or FSST**	~2× from the 16-char alphabet; OnPair is −12..15% on ratio vs DLBA+zstd and −12..36% on decode vs FSST, encoding at 27–43 MiB/s against 570–680 for DLBA+zstd
Near-random text (street addresses)	**DLBA+zstd, or store raw**	~1.3× ceiling; OnPair −4.6% on ratio and −42% on decode
Ingest-bound / write-heavy	**FSST (random access) or a native page + lz4 (raw speed)**	a native page + lz4 encodes a median 3.8× faster (1.8–35× by column); FSST a median 2.4× (1.3–12×)

Where this implementation differs from the paper

Changes the trained dictionary, therefore the compressed bytes

- **Merge threshold** – paper (sec 3.2.1) fixes it per dataset as max(2, floor(log2 S_MiB)); this port uses an adaptive controller paced to a byte budget.
- **Long-bucket overflow** – the paper caps each bucket at 128 suffixes and drops the extras (sec 3.4.4); this port promotes an over-full bucket to a trie and keeps every suffix. The paper attributes its own OnPair16 ratio loss on URLs (2.68 vs OnPair's 2.80) to that dropping, so keeping them removes the loss and bounds lookup cost instead.
- **Training-sample shuffle** – unspecified by the paper; this port uses a fixed-seed splitmix64 full Fisher-Yates. Shuffle extent is what moves the dictionary, not the RNG choice: partial-shuffling leaves the sampled rows in the slice tail while the trainer iterates from index 0, so on sequentially-ordered data it trains mostly on low-numbered rows. Worth 22.7% of ratio on c_name.
- **Dictionary budget** – not in the paper, which fixes 16 bits (sec 3.6); this port picks per column from 9–16, the OnPair-auto rows above. 16 is optimal only when codes are stored two bytes wide, where a narrower dictionary shrinks the dictionary and leaves the code stream alone. Once codes are packed at the dictionary's true width a narrower budget shortens every code, and the optimum moves: 9 bits for enums, 10 for patterned names, 12–13 for free text, 14 for URLs. Worth up to +18% of stored size.

References

7. F. Gargiulo and R. Venturini. *OnPair: Short Strings Compression for Fast Random Access.* arXiv:2508.02280, 2025. https://arxiv.org/abs/2508.02280
8. P. Boncz, T. Neumann, V. Leis. *FSST: Fast Random Access String Compression.* PVLDB 13(11), 2020 https://www.vldb.org/pvldb/vol13/p2649-boncz.pdf .
9. Y. L. Alexandre. *FSST+: Enhancing String Compression Through Common Prefix Extraction.* MSc thesis, CWI, 2025.

FSST vs OnPair

Similarity : At the encode/decode model.

The encoded form of both is the same: a static table, greedy longest-prefix match to emit fixed-width codes. Both formats dodecode by table lookup and copy. Both give per-row random access. They're interchangeable at the format level.

Difference in the code format: escape vs. guaranteed coverage

OnPair keeps all 256 single bytes in its dictionary, so every possible byte has a code. Codes are therefore uniform width and the decode loop is branch-free — read a code, look up the token, copy it, advance. FSST instead reserves one code as an escape and spends the rest on multi-byte symbols; a byte with no symbol is emitted as escape plus literal, so its output is a mix of one- and two-byte codes and its decoder must test for the escape.

That single choice explains most of what follows. Reserving all 256 codes for single bytes is what buys OnPair a branch-free decode, and it is also what costs OnPair the 8-bit design point: with the whole 8-bit space consumed by singles there is no room left for merged pairs, which is why OnPair's dictionary budget starts at 9 bits. FSST reaches 8 bits by giving up guaranteed coverage and paying a penalty on the bytes it did not select. The two are different designs, not one design at two widths.

Difference in dictionary construction: objective-driven search vs. single-pass agglomeration.

FSST is an iterative local search with an explicit objective. It counts symbol and symbol-pair frequencies over a sample, scores every candidate by frequency times length (with single-byte symbols promoted, to hold down the escape rate), orders candidates in a priority queue by that score, then discards the entire table and refills it from the top of the queue. It repeats this over five rounds on a progressively larger sample, evaluates each resulting table against the objective, and keeps the best table it saw rather than the last one. The final round scores only and creates no new symbols. So: generate, score, evict, repeat — with a stated objective and best-solution tracking.

OnPair is single-pass greedy agglomeration. There is no gain function at selection time, no priority queue, no eviction, no rounds, and no best-of tracking. One streaming pass over a shuffled sample counts adjacent token pairs, and the moment a pair's count crosses a threshold that pair is merged into the dictionary permanently. The newly merged token immediately becomes the left half of the next pair, so tokens grow transitively within the same pass. It is BPE with the global pair-position index removed — dropping exactly the part of BPE that makes training expensive. The consequence is that OnPair trains far faster and never reconsiders: a token that earns its place early keeps it, whether or not a better table existed.

What the two differences imply about where each wins.

The code format sets each codec's break-even token length. OnPair spends two bytes per code at a full budget, so it only profits from tokens of three bytes or more; FSST spends one byte per code, so any two-byte symbol already pays. On data with real redundancy — comments, descriptions, reviews, URLs, search phrases, addresses — OnPair's transitive merging grows long tokens and it wins comfortably, because a long token amortizes a wide code many times over. On near-random data neither codec can grow tokens, both sit close to their floors, and the codec with the cheaper code wins: FSST. Note that

this is about how compressible the data is, not how short the values are; long hex hashes and UUIDs fall on FSST's side for exactly this reason.

Decode splits along a different mechanism. OnPair's loop is bound by the bytes it stores, not by dependency latency, so decode speed tracks how many raw bytes each code expands to. On small-vocabulary short columns FSST can reconstruct an entire value from a single code where OnPair needs several, and it decodes faster there. Unpacking cost is a separate effect, and it turns on byte alignment rather than code size. A code that lands on a byte boundary needs no shift or mask, which is what makes both FSST's 8-bit code and OnPair's 16-bit code cheap to unpack. Holding the dictionary and the code stream fixed and re-packing the same codes at every width, decode is a step rather than a ramp — every width from 9 to 15 bits lands within a few percent of every other, and 16 bits alone is distinctly faster because it is the only byte-aligned width in the range. So a wider code does not decode faster in general; an aligned one does. A 12-bit code buys nothing over a 9-bit one, and the win at 16 is paid for on the ratio axis, since every code then stores more bits than it needs. Note this is about OnPair's packing width, not FSST's code space — a wider-code FSST is a separate question.

Training cost runs the other way and is a direct consequence of the construction difference. FSST's five scored rounds are cheap because the sample is small and the objective is closed-form; OnPair's single pass is cheap per byte but must tokenize the whole column afterwards against the frozen dictionary, and that tokenization is the dominant encode cost.

Selection.

- **FSST** for data with little exploitable redundancy — hashes, UUIDs, near-random identifiers — and for tiny-alphabet very short values, where OnPair cannot build tokens long enough to justify a wider code. Also for decode-bound work on small-vocabulary short columns, where one FSST code covers a whole value.
- OnPair for the bulk of real string bytes — free text, descriptions, URLs, addresses, semi-structured payloads. Against FSST that includes addresses, though a delta-length page plus zstd beats both there. The dictionary budget is a real trade rather than free compression. A per-column budget gives the smaller stored size; a fixed 16-bit budget gives the faster decode. Which to prefer depends on whether the column is written once and scanned often.
- **Neither**, on low-cardinality columns. Most of what looks like an OnPair win there is the dictionary effect, which a plain dictionary encoding already delivers, and on sequential or monotonic columns a delta encoding beats both by orders of magnitude. Symbol-table compression is the right tool for high-cardinality text, and reaching for it on enum-like data is the common mistake.