FSST string compression

FSST = Fast Static Symbol Table

Peter Boncz (CWI)
Viktor Leis (FSU Jena)
Thomas Neumann (TU Munich)

https://github.com/cwida/fsst

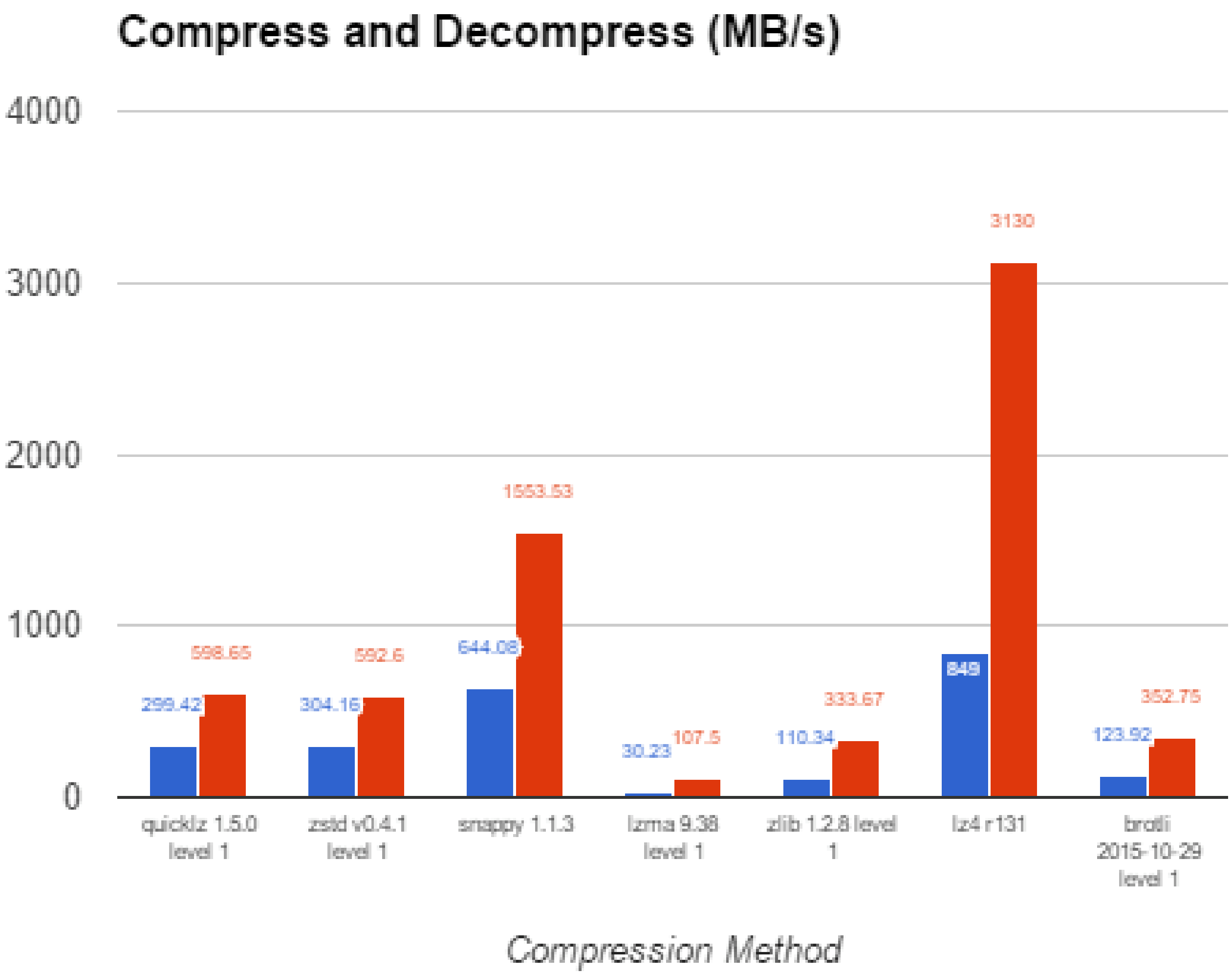
String Compression in a DBMS

- Dictionary Compression
  - Whole string becomes 1 code, points into a dictionary D
  - works well if there are (relatively) few unique strings

- Heavy-weight/general-purpose Compression
  - Lempel-Zif plus possibly entropy coding
  - Zip, gzip, snappy, **LZ4**, zstd, …
  - Block-based decompression

String Compression in a DBMS

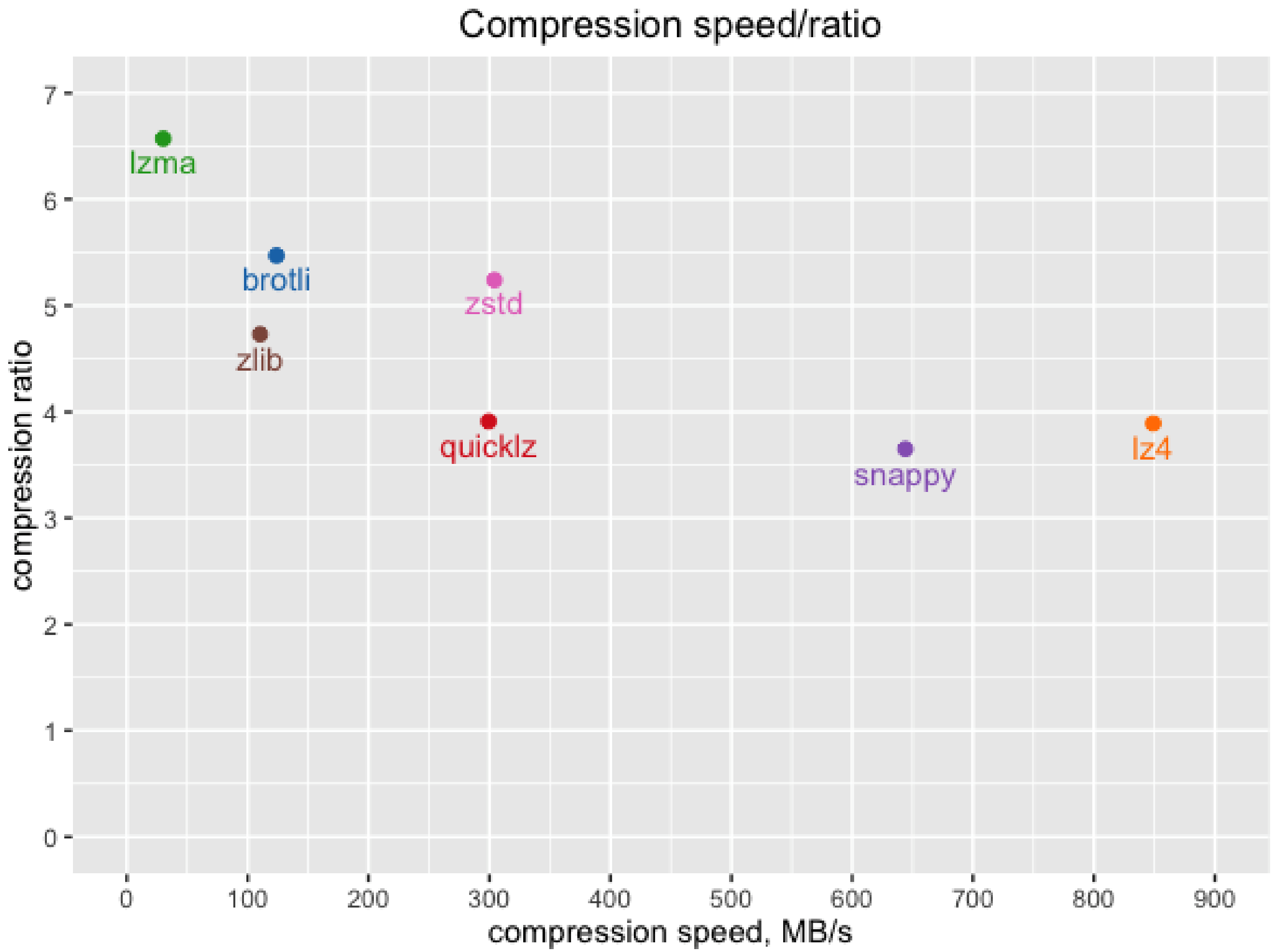
www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/



*"**LZ4** is the best compression library for database applications, if high speed is desired"*

String Compression in a DBMS

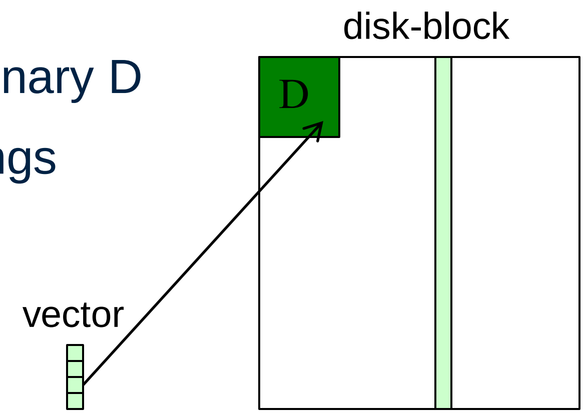
www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/



*"**LZ4** is the best compression library for database applications, if high speed is desired"*

String Compression in a DBMS

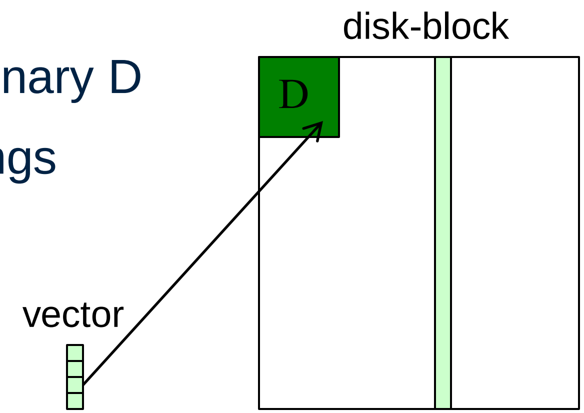
- Dictionary Compression
  - Whole string becomes 1 code, points into a dictionary D
  - works well if there are (relatively) few unique strings



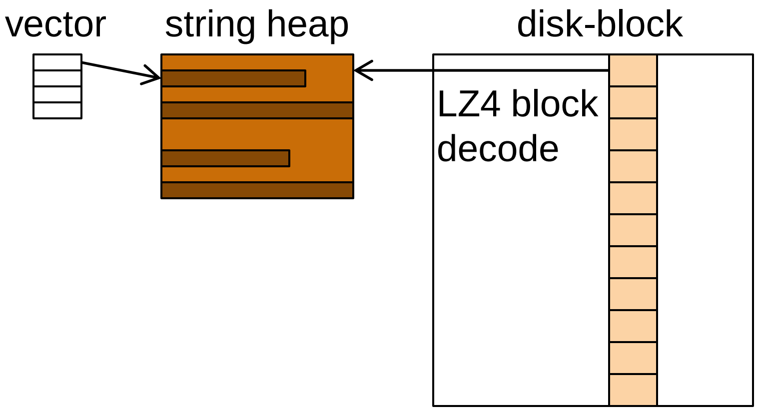
- Heavy-weight/general-purpose Compression
  - Lempel-Zipf plus possibly entropy coding
  - Zip, gzip, snappy, **LZ4**, zstd, …
  - Block-based decompression

String Compression in a DBMS

- Dictionary Compression
  - – Whole string becomes 1 code, points into a dictionary D
  - – works well if there are (relatively) few unique strings

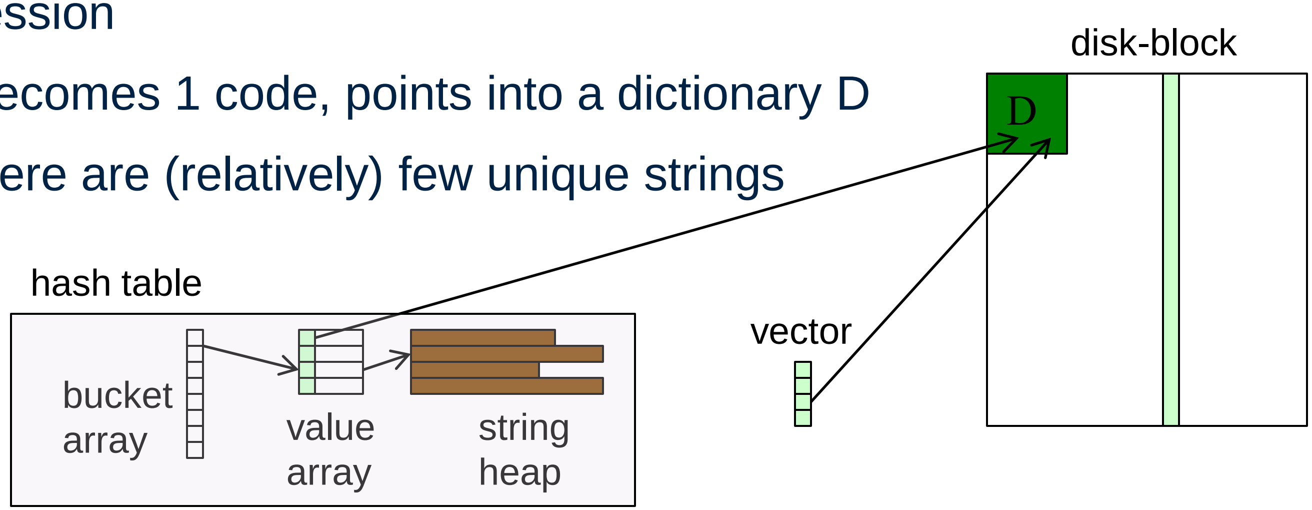


- Heavy-weight/general-purpose Compression
  - – Lempel-Zipf plus possibly entropy coding
  - – Zip, gzip, snappy, **LZ4**, zstd, …
  - – Block-based decompression

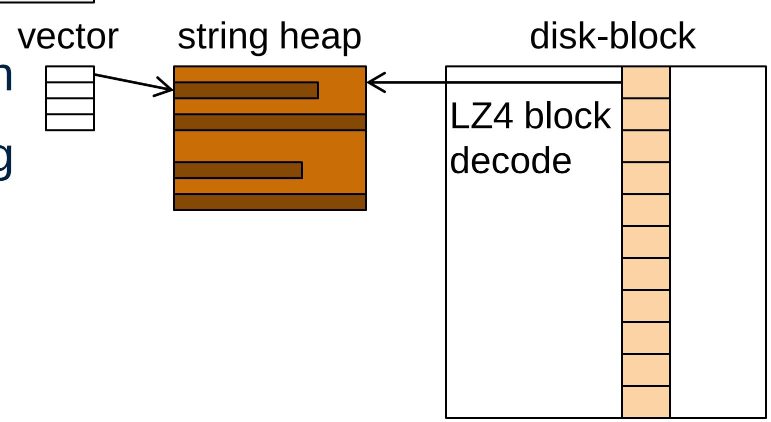


String Compression in a DBMS

- Dictionary Compression
  - Whole string becomes 1 code, points into a dictionary D
  - works well if there are (relatively) few unique strings

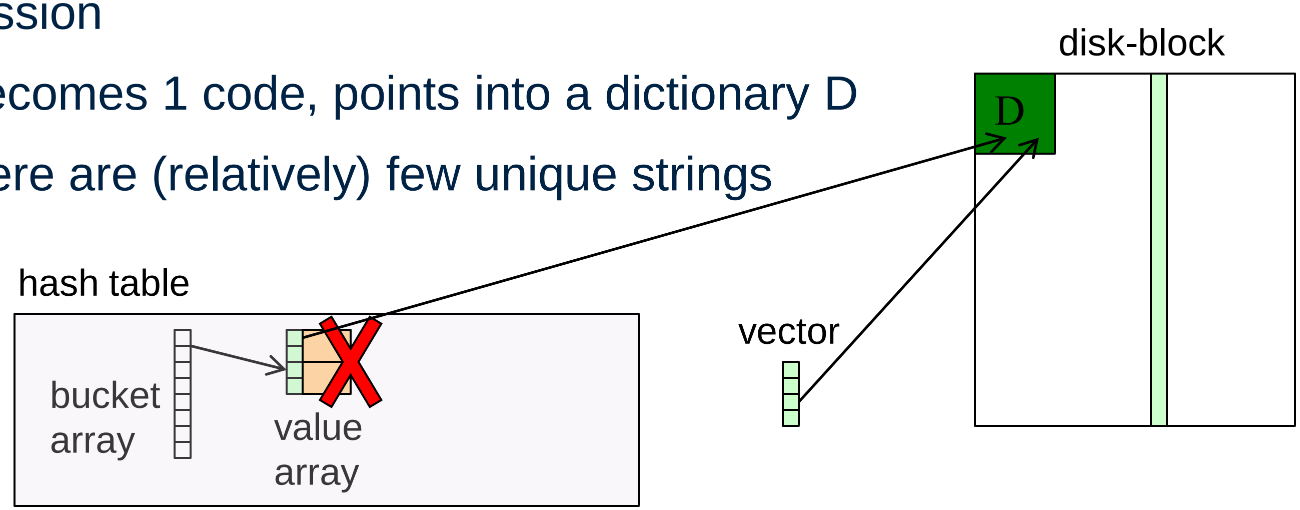


- Heavy-weight/general-purpose Compression
  - Lempel-Zipf plus possibly entropy coding
  - Zip, gzip, snappy, **LZ4**, zstd, …
  - Block-based decompression

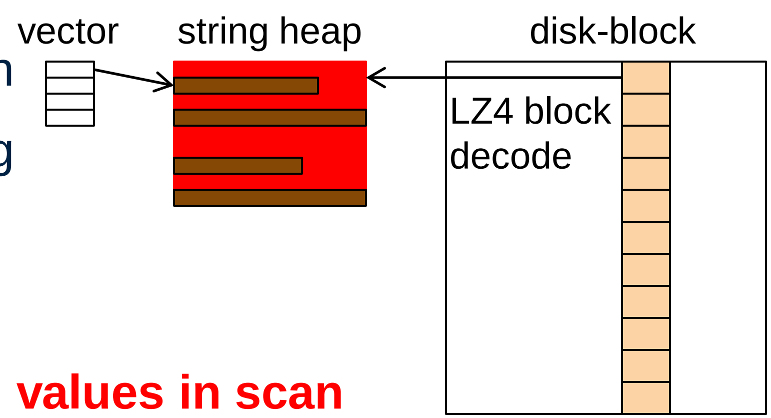


String Compression in a DBMS

- Dictionary Compression
  – Whole string becomes 1 code, points into a dictionary D
  – works well if there are (relatively) few unique strings



- Heavy-weight/general-purpose Compression
  – Lempel-Zipf plus possibly entropy coding
  – Zip, gzip, snappy, **LZ4**, zstd, …
  – Block-based decompression
    - **must decompress (all=) unneeded values in scan**
    - **cannot be leveraged in hash tables, sorting, network shuffles**
    - **FSST targets compression of many small textual strings**



The Idea

- Encode strings as a sequence of bytes, where each byte [0,254] is a
  - **CODE**
- Each code stands for a 1-8 byte
  - **SYMBOL**

corpus (uncompressed)

```
http://in.tum.de
http://cwi.nl
www.uni-jena.de
www.wikipedia.org
http://www.vldb.org
...
```

symbol table

	symbol	length
0	http://	7
1	www.	4
2	uni-jena	8
3	.de	3
4	.org	4
5	a	1
6	in.tum	6
7	cwi.nl	6
8	wikipedi	8
9	vldb	4
...		
255		

corpus (compressed)

```
063
07
123
1854
0194
...
```

Small symbol table(s):
RAM: 2.2KB,
disk/network: ~500B

- Byte 255 is special code marking
  - **EXCEPTION**

  followed by 1 uncompressed byte

Closest existing scheme is ***RePair****, but is >100x slower than FSST (both ways)*

FSST Decoding

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++;
  }
}
```

Fast FFST Decoding

- Fast-skip escapes, handle escapes with Duff's device

```
while (posOut+32 <= size && posIn+4 <= lenIn) {
   unsigned int nextBlock = *((unsigned int*) (strIn+posIn));
   unsigned int escapeMask = (nextBlock&0x80808080u)&((((~nextBlock)&0x7F7F7F7Fu)+0x7F7F7F7Fu)^0x80808080u);
   if (escapeMask == 0) {
      code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
   } else {
      unsigned firstEscapePos=__builtin_ctzl(escapeMask)>>3;
      switch(firstEscapePos) {
      case 3: code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      case 2: code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      case 1: code = strIn[posIn++]; *(unsigned long*) (strOut+posOut) = symbol[code]; posOut += len[code];
      case 0: posIn+=2; strOut[posOut++] = strIn[posIn-1]; /* decompress an escaped byte */
      }
   }
}
```

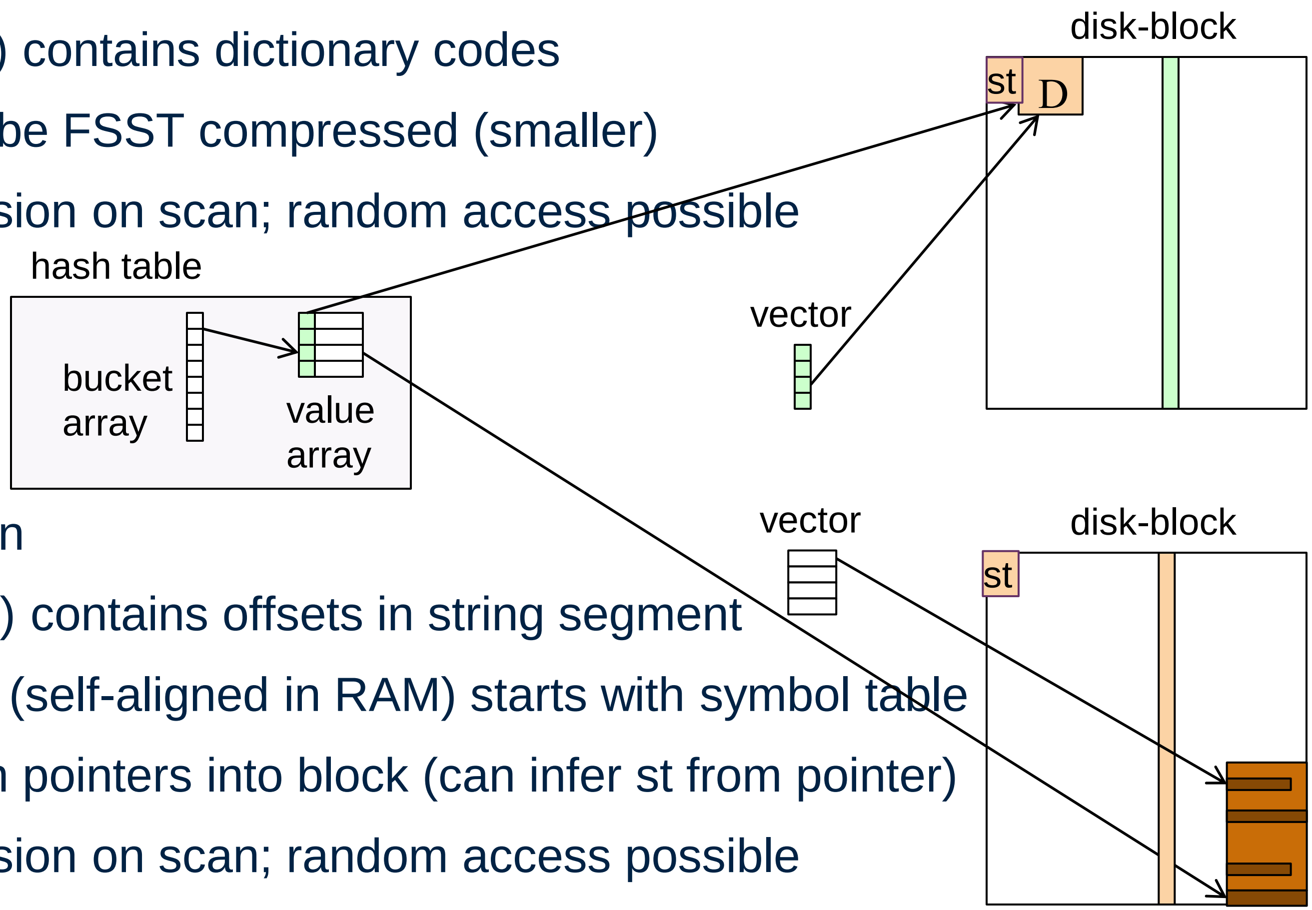
FSST encoding

Algorithm 2 FSST-encoding, given a symbol table.

```
void encode(uint8_t*& in, uint8_t*& out, SymbolTable& st) {
  uint16_t pos = st.findLongestSymbol(in);
  if (pos <= 255) { // no (real) symbol found
     *(out++) = 255;
     *(out++) = *(in++);
  } else {
     *(out++) = (uint8_t) pos;
     in += st.symbols[pos].len; // symbol length in bytes
  }
}
```

FSST Compression in a DBMS

- Dictionary Compression
  - column (green) contains dictionary codes
  - dictionary can be FSST compressed (smaller)
  - no decompression on scan; random access possible



- FSST Compression
  - column (brown) contains offsets in string segment
  - eg 64KB block (self-aligned in RAM) starts with symbol table
  - vectors contain pointers into block (can infer st from pointer)
  - no decompression on scan; random access possible

Challenge: Finding a Good Symbol Table

- Why is this hard? **Dependency Problem!**
- First attempt:
  - Put the corpus in a suffix array
  - Identify the 255 common substrings with most gain (=length*frequency)
  - **Problem 1**:
    - Valuable symbols will be **overlapping** (they are not as valuable as they seem)
    - We tried compensating for overlap ➔ did not work
  - **Problem 2**: (**greedy** encoding)
    - The encoding will not arrive at the start of the valuable symbol, because the previous encoded symbol ate away the first byte(s)
    - We tried dynamic programming encoding (slow!!) ➔ no improvements

FSST bottom-up symbol table construction

- Evolutionary-style algorithm
- Starts with empty symbol table, uses 5 iterations:
  - We encode (a sample of) the plaintext with the current symbol table
  - We count the occurrence of each symbol
  - We count the occurrence of each two subsequent symbols
  - We also count single byte(-extension) frequencies, even if these are not symbols
  - Two subsequent symbols (or byte-extensions) generate a new concatenated symbol
  - We compute the gain (length*freq) of all bytes, old symbols and concatenated symbols and insert the 255 best in the new symbol table

Algorithm Example (maxlen=3, |st|=5)

Uncompressed

t	u	m	c	w	i	t	u	m	v	l	d	b

len = 13

empty symbol table

Algorithm Example (maxlen=3, |st|=5)

Uncompressed

t	u	m	c	w	i	t	u	m	v	l	d	b

len = 13

empty symbol table

Iteration 1

$	t	$	u	$	m	$	c	$	w	$	•••	$	d	$	b

len = 26

Symbol table

symbol	um	tu	wi	cw	mc	
len	2	2	2	2	2	
count	2	2	1	1	1	
gain	4	4	2	2	2	(len * cnt)

CWI Database Architectures

Algorithm Example (maxlen=3, |st|=5)

Iteration 1

$	t	$	u	$	m	$	c	$	w	$	•••	$	d	$	b

len = 26

Symbol table

symbol	um	tu	wi	cw	mc
len	2	2	2	2	2
count	2	2	1	1	1
gain	4	4	2	2	2

(len * cnt)

Iteration 2

tu	mc	wi	tu	$	m	$	v	$	l	$	d	$	b

len = 14

Symbol table

symbol	tum	tu	wit	mcw	vl
len	3	2	3	3	2
count	2	2	1	1	1
gain	6	4	3	3	2

Algorithm Example (maxlen=3, |st|=5)

Iteration 2

tu	mc	wi	tu	$	m	$	v	$	l	$	d	$	b

len = 14

Symbol table

symbol	tum	tu	wit	mcw	vl
len	3	2	3	3	2
count	2	2	1	1	1
gain	6	4	3	3	2

Iteration 3

tum	$	c	wit	$	u	$	m	vl	$	d	$	b

len = 13

Symbol table

symbol	mvl	cwi	vld	tum	wit
len	3	3	3	3	3
count	1	1	1	1	1
gain	3	3	3	3	3

Algorithm Example (maxlen=3, |st|=5)

Iteration 3

tum	$	c	wit	$	u	$	m	vl	$	d	$	b

len = 13

Symbol table

symbol	mvl	cwi	vld	tum	wit
len	3	3	3	3	3
count	1	1	1	1	1
gain	3	3	3	3	3

Iteration 4

tum	cwi	tum	vld	$	b

len = 6

Symbol table

symbol	tum	cwi	vld	b
len	3	3	3	1
count	2	1	1	1
gain	6	3	3	1

Algorithm Example (maxlen=3, |st|=5)

Iteration 4

tum	cwi	tum	vld	$	b	len = 6

Symbol table

symbol	tum	cwi	vld	b
len	3	3	3	1
count	2	1	1	1
gain	6	3	3	1

Compressed

tum	cwi	tum	vld	b	len = 5

Uncompressed

t	u	m	c	w	i	t	u	m	v	l	d	b	len = 13

Making FSST encoding fast

- `findLongestSymbol()`
  - Finds the next symbol
  - How? Range-scan in sorted list, indexed by first byte
  - A for-loop

- Goal: encoding without **for-loop** and without **if-then-else**

- Idea: **Lossy Perfect Hash Table**
  - Perfect: no collisions. How? Throw away colliding symbol with least gain
  - Lossy, therefore. But: we keep filling it with candidates until full anyway

  - Hash table key is next 3 bytes
  - Use a `shortCodes[65536]` direct lookup array for the next two bytes
  - Append a **terminator** single-byte symbol to plaintext (typically byte 0)

Database Architectures

Making FSST encoding fast

- Idea: **Lossy Perfect Hash Table**
  - Perfect: no collisions. How?
    - Throw away colliding symbol with least gain
  - Lossy, therefore.
    - But: we keep filling it with candidates until full anyway
  - Hash table key is next 3 bytes
  - Use a **`shortCodes[65536]`** direct lookup array for the next two bytes
  - Append a **terminator** single-byte symbol to plaintext (typically byte 0)

```
u16 shortCodes[65536]; // code:12,length:4              +-----------+----+
struct Symbol {                                          |  code     | len|
     union { char str[maxLength]; u64 num; } val;       +-----------+----+
     u32 icl;     // icl = u64 ignoredBits:16,code:12,length:4
}                                           +--------------+-----------+----+
Symbol hashTab[4096];                       |  ignoredBits |  code     | len|
                                            +--------------+-----------+----+
```

Lossy Perfect Hash Table

- String encoding without **for-loop** and without **if-then-else**

```
auto hashFind = [&]() {
  out[1] = word; // dirty trick: speculatively write out escaped byte
  uint64_t idx = hash(word & 0xFFFFFF) & (st.hashTabSize-1);
  s = st.hashTab[idx]; // fetch symbol from hash table
  uint64_t clean = (-1LL >> /*ignoredBits*/ (uint8_t) s.icl);
  return (s.icl < 0xFFFFFFFF && s.val.num == (word & clean));
};
auto hashedKernel = [&]() {
  code = hashFind() ? (s.icl>>16) : code; // conditional move
  *out = code; // write out code byte (or 255)
  out += 1+((code>>8)&1); // increase with 1 or 2 (escape = 9th bit)
  cur += (code>>12); // symbol length is in bits [12..15] of code
 }
);
void
encodeHash(uint8_t*& cur, uint8_t*& out, SymbolTable& st){
  Symbol s;
  uint64_t word = *(uint64_t*)cur;
  uint64_t code = st.shortCodes[word & 0xFFFF];
  hashedKernel();
}
```

AVX512 Implementation

- Idea: compress 8 strings in parallel (8 lanes of 64-bits)
  - *3 = 24 in parallel (unrolled loop) – not *4 because of register pressure
  - **job** queue: 511 byte (max) string chunks
    - Add terminator symbol to each chunk
    - Sort jobs on string length (longest first) – load balancing, keep lanes busy
    - 512 jobs of 511B input, 1024B output (768KB buffer)
  - Each iteration:
    - Insert new jobs in (any) free lanes (**expand-load**)
    - **`findLongestSymbol()`** in AVX512
      - Match 1 symbol in input, add 1 code to output strings (in each lane)
      - Involves 3x**gather** (2x **`hashTab`** 1x**`shortCodes`**) + 1x**scatter** (output)
    - Append finished jobs in result job array (**compress-store**)

Evaluation: dbtext corpus

- **machine-readable identifiers** (hex, yago, email, wiki,`uuid, urls2, urls),
- **human-readable names** (firstname, lastname, city, credentials,street, movies),
- **text** (faust, hamlet, chinese, japanese, wikipedia),
- **domain-specific codes** (genome, location)
- **TPC-H data** (c_name, l_comment, ps_comment)

name	avg len	example string	LZ4 factor	FSST factor
hex	8	DD5AF484	1.14×	2.11×
yago	19	Ralph_A._Brown	1.25×	1.63×
email	22	xnj_14@hotmail.com	1.55×	2.13×
wiki	23	Benzil	1.31×	1.63×
uuid	37	84e22ac0-2da5-11e8-9d15- ...	1.55×	2.44×
urls2	55	http://fr.wikipedia.org/ ...	1.75×	2.05×
urls	63	http://reference.data.go ...	2.77×	2.42×
firstname	7	RUSSEL	1.25×	2.04×
lastname	10	BALONIER	1.28×	1.97×
city	10	ROELAND PARK	1.37×	2.14×
credentials	11	PHD, HSPP	1.48×	2.31×
street	13	PURITAN AVENUE	1.60×	2.35×
movies	21	Return to 'Giant'	1.23×	1.66×
faust	24	Erleuchte mein bedÃijrftig Herz.	1.48×	1.87×
hamlet	30	<LINE>That to Laertes ...	2.13×	2.41×
chinese	87	道人决心消除肉会 ...	1.40×	1.69×
japanese	90	せん。しかし、...	1.84×	2.00×
wikipedia	130	Weniger hÃďufig fressen sie ...	1.45×	1.81×
genome	10	atagtgaag	1.59×	3.32×
location	40	(40.84242764486843, -73 ...	1.58×	2.51×
c_name	19	Customer#000010485	3.08×	3.80×
l_comment	27	nal braids nag carefully expres	2.22×	2.90×
ps_comment	124	c foxes. fluffily ironic ...	2.79×	3.40×

Note: traditional compression datasets (e.g. Silesia) contain >50% binary files. Our new corpus is representative for DB text.

FSST vs LZ4

- Note **first bar** with overall average (AVG)
- FSST has **better compression factor** and **better compression speed** than LZ4
  - **equal decompression speed**

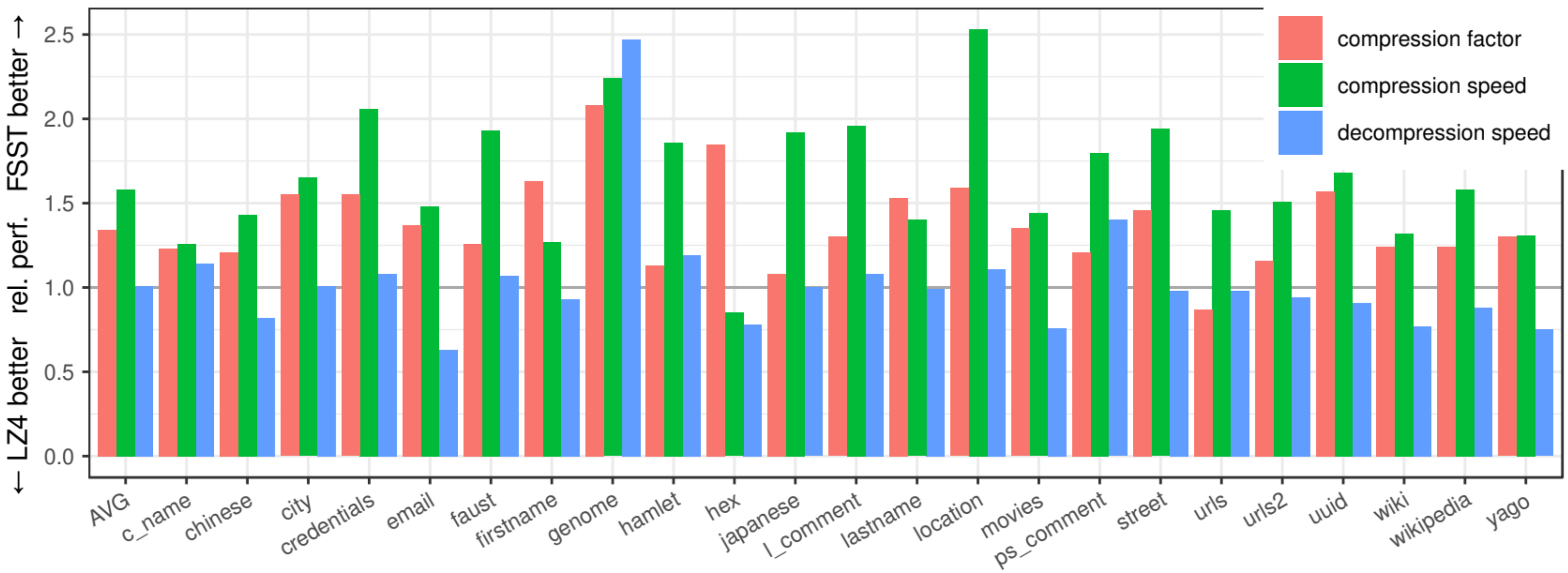


Figure 3: Relative performance of FSST versus LZ4 in terms of compression factor, compression speed, and decompression speed. Each data set is treated as a 8MB file.

LZ4 as a database compressor

- It does not make sense to use LZ4 to compress strings one-by-one (“line”), even when using a pretrained zstd dictionary (“dict”). It is slow and has bad compression factor. General-purpose compression should be block-based.

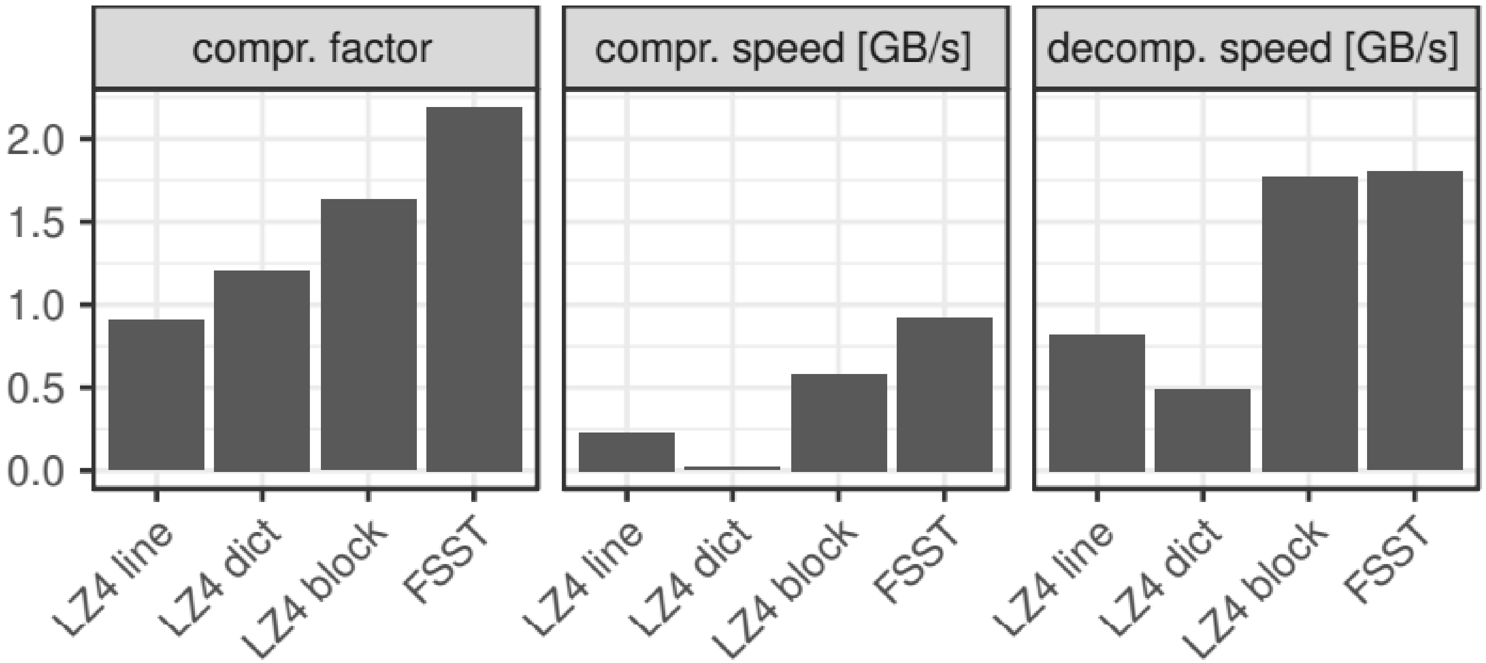


Figure 4: With LZ4 short strings do not compress well, even with a pre-generated dictionary.

Random Access: FSST vs LZ4

- Use case: **Scan** with a pushed down predicate (selection % on X axis)
  - LZ4 must decompress all strings, FSST only the selected tuples
  - FSST might even choose not to decompress strings (would even be faster)

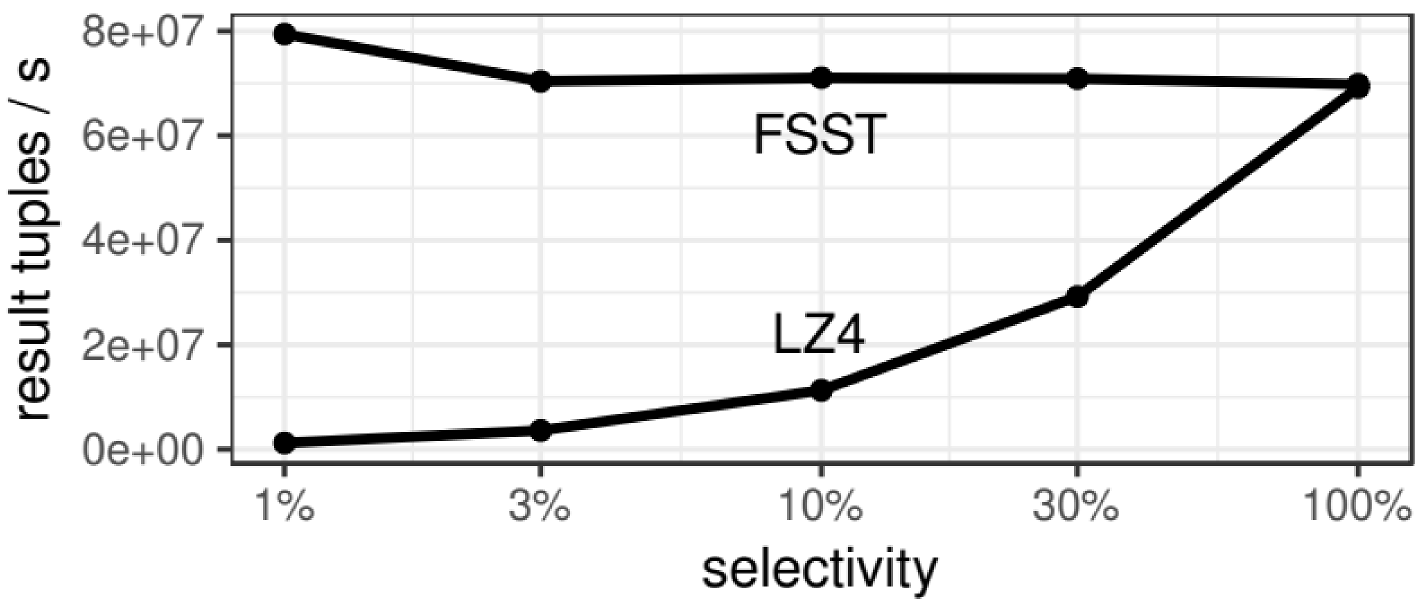


Figure 5: Selective queries are fast in FSST due to random access to individual values.

SIMD vs Scalar

- Encoding performance
  - 400MB/s scalar
  - 900MB/s AVX512
- Note:
  - Without AVX512, LZ4 compression is faster than FSST
  - But FSST speed still 70% of LZ4

$simd_1$	$simd_2$	$simd_3$	$simd_4$	$adptv$	
8.01	5.36	**4.98**	5.16	**11.26**	firstname
7.97	5.08	**4.17**	4.42	**5.09**	hex
8.70	5.51	**5.07**	4.69	**10.82**	city
5.29	3.57	**2.85**	3.26	**9.00**	genome
8.05	5.12	**4.18**	4.44	**10.35**	lastname
7.82	5.03	**4.68**	4.61	**13.41**	credentials
7.35	5.02	**4.36**	4.52	**11.46**	street
3.87	2.76	**2.23**	2.42	**5.37**	c_name
9.36	5.91	**5.03**	5.31	**11.50**	yago
9.10	5.74	**4.82**	5.14	**11.36**	movies
7.19	4.60	**4.03**	4.25	**10.13**	email
9.37	5.92	**4.93**	5.30	**11.88**	wiki
8.55	5.51	**4.98**	5.18	**13.05**	faust
5.59	3.63	**3.16**	3.42	**9.78**	l_comment
7.35	4.56	**4.13**	4.23	**10.50**	hamlet
6.36	4.08	**3.40**	3.59	**8.06**	uuid
5.64	3.70	**3.05**	3.14	**5.73**	location
7.50	4.72	**4.03**	4.28	**10.12**	urls2
6.32	4.04	**3.51**	3.74	**8.55**	urls
9.19	5.78	**5.04**	5.15	**13.74**	chinese
8.22	5.27	**4.69**	5.06	**14.47**	japanese
4.70	3.11	**2.77**	3.07	**9.14**	ps_comment
8.21	5.27	**4.46**	4.87	**12.65**	wikipedia
7.38	4.75	**4.11**	4.32	**10.32**	

Conclusion

- Databases are full of strings (see Public BI benchmark, DBtest “get real” paper)
  - String processing is a big bottleneck (CPU, RAM, network, disk)
  - **String compression** is therefore a good idea (less RAM, network, disk)
  - Operating on compressed strings is **very beneficial**
- FSST provides:
  - **random access to compressed strings!**
  - comparable/better (de)compression **speed** and **ratio** than the fastest general purpose compression schemes (LZ4)
- Useful opportunities of FSST:
  - Compressed execution, comparisons on compressed data
  - Late decompression (strings-stay-string). Has 0-terminated mode.
  - Easy integration in existing (database) systems

- **MIT licensed, code, paper + replication package** github.com/cwida/fsst