FSST string compression

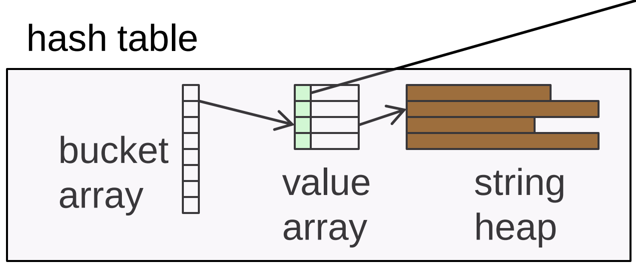
FSST = Fast Static Symbol Table

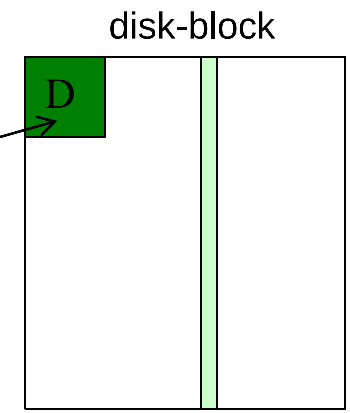
Peter Boncz (CWI)
Viktor Leis (FSU Jena)
Thomas Neumann (TU Munich)

Open-source code (Linux, MacOS, Windows)
+replication package:
https://github.com/cwida/fsst

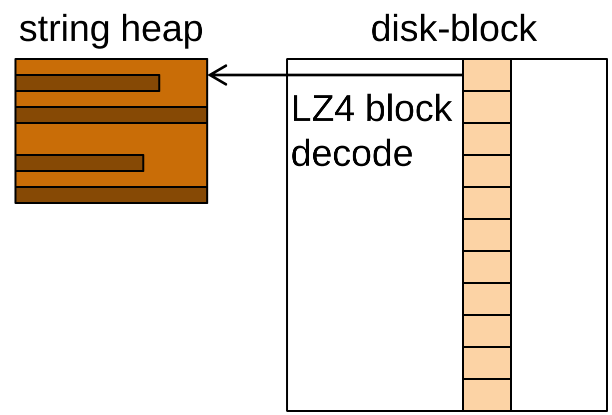
String Compression in a DBMS

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



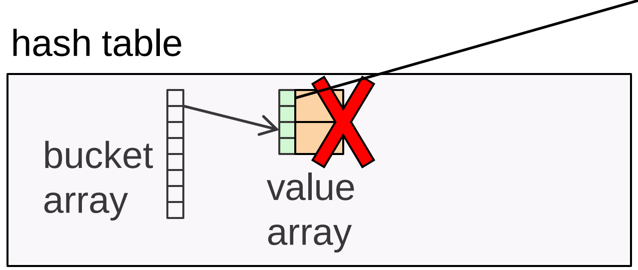


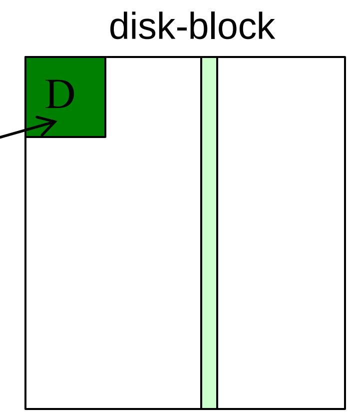
- Heavy-weight/general-purpose Compression
  - Lempel-Zipf plus possibly entropy coding
  - Zip, gzip, snappy, **LZ4**, zstd, …
  - Block-based decompression



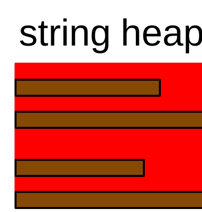
String Compression in a DBMS

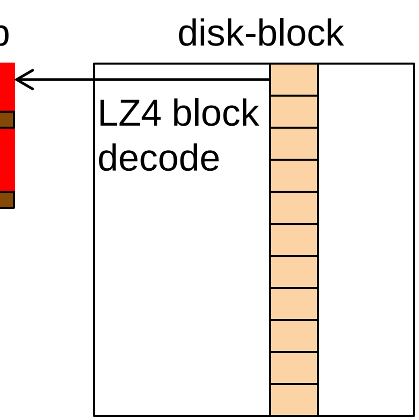
- Dictionary Compression
  - – Whole string becomes 1 code, points into a dictionary D
  - – works well if there are (relatively) few unique strings





- 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
...
```

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

  followed by 1 uncompressed byte

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

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

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. For bootstrap and robustness.
  - 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

Making FSST encoding fast

- `findLongestSymbol()`
  - Finds the next symbol
  - Naïve: range-scan in sorted list, indexed by first byte

- 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 lookup key is next 3 bytes
  - Use a **`shortCodes[65536]`** direct lookup array for the next two bytes
  - Choose between these two lookups with a conditional move

    `hashHit?hashCode:shortCode`

AVX512 Implementation

- Idea: compress 8 strings in parallel (8 lanes of 64-bits)
  - *3 = 24 in parallel (unrolled loop)
  - **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)

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

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×

FSST vs LZ4

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

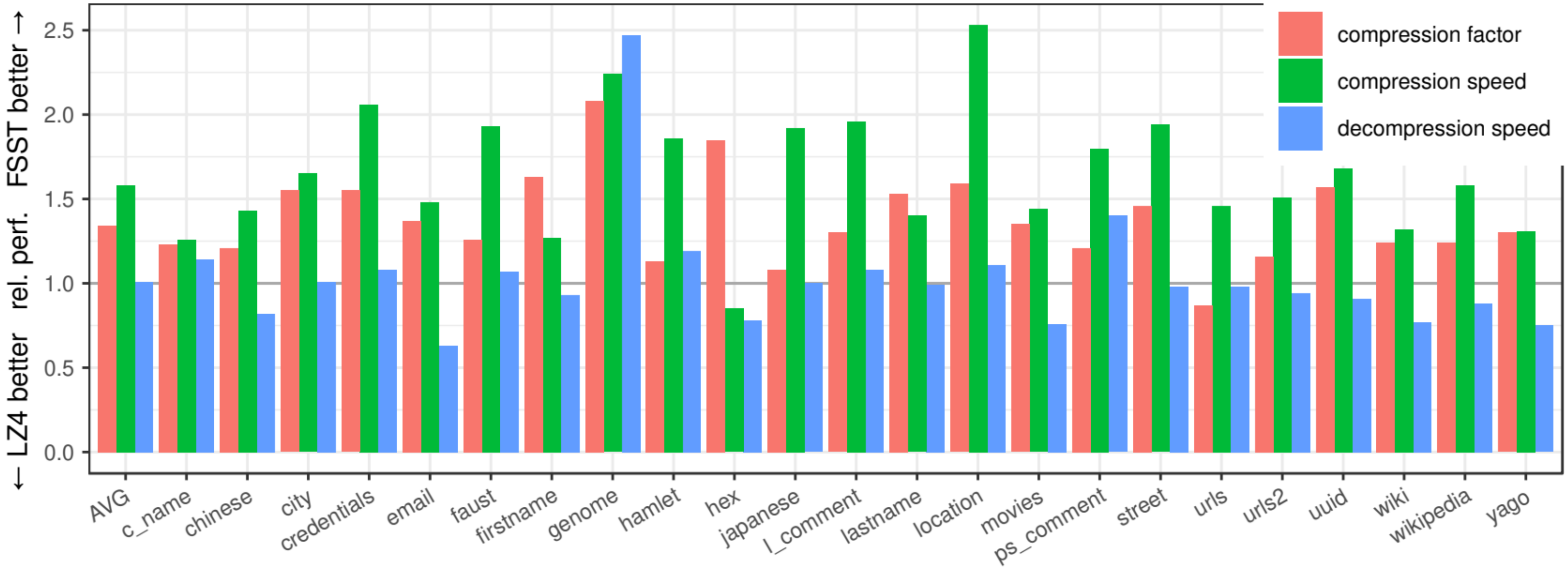


Figure 3: Relative performance of FSST versus LZ4 in terms of compression factor, compression speed, and decompression speed. Each data set is treated as a 8MB file.

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)

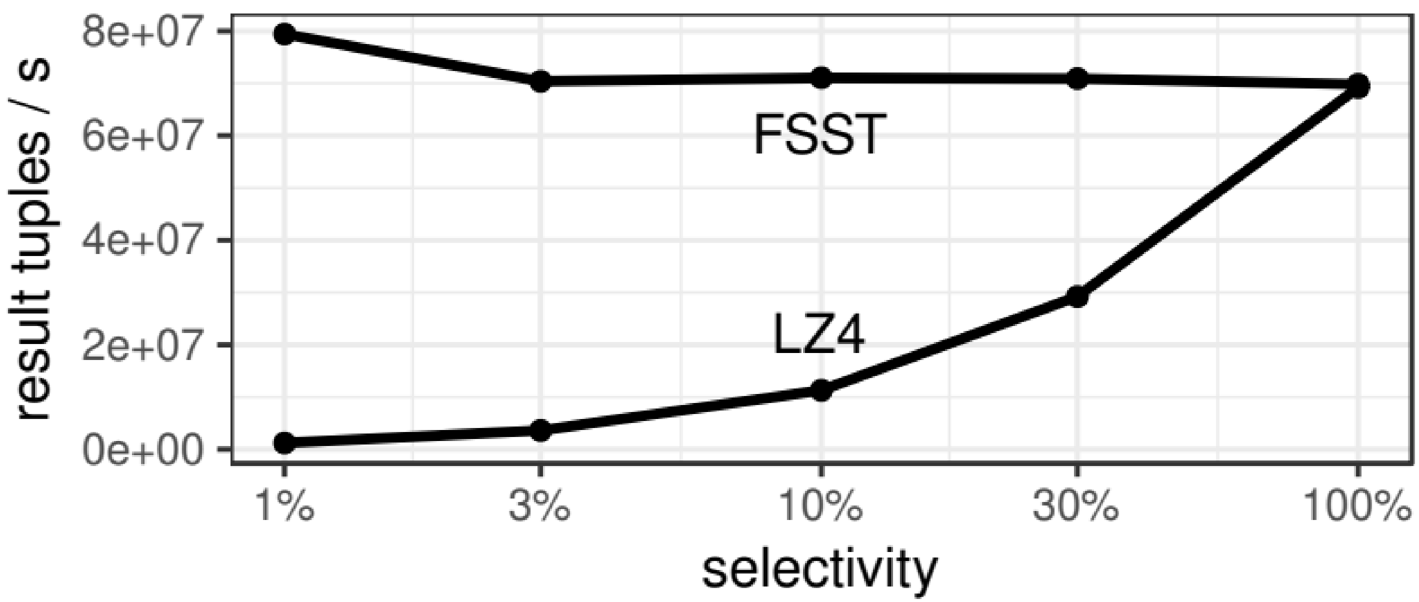


Figure 5: Selective queries are fast in FSST due to random access to individual values.

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-strings**). Has 0-terminated mode.
  - Easy integration in existing (database) systems

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