Ingénierie de la performance au sein des mégadonnées

Daniel Lemire et collaborateurs

https://lemire.me

> “One Size Fits All”: An Idea Whose Time Has Come and Gone (Stonebraker, 2005)

Redécouvrir Unix

Plusieurs composantes spécialisées et réutilisables:

- Calcite : SQL + optimisation
- Hadoop
- etc.

"Make your own database from parts"

Ensembles

Un concept fondamental (ensemble de documents, d'enregistrements)

$\longrightarrow$ Pour la performance, on utilise des ensemble d'entiers (identifiants).

$\longrightarrow$ Souvent des entiers à 32 bits suffise (identifiants locaux).

- tests : $x \in S$?
- intersections : $S_2 \cap S_1$
- unions : $S_2 \cup S_1$
- differences : $S_2 \setminus S_1$
- Jaccard/Tanimoto : $|S_1 \cap S_1| / |S_1 \cup S_2|$
- iteration:

```
for x in S do
    print(x)
```

Mise en oeuvre

- tableaux triés (`std::vector<uint32_t>`)
- tables de hachage (`java.util.HashSet<Integer>` , `std::unordered_set<uint32_t>`)
- `...`
- bitmap (`java.util.BitSet`)
- ❤️ ❤️ ❤️ bitmap compressé ❤️ ❤️ ❤️

Tableaux

```
while (low <= high) {
    int mI =
    (low + high) >>> 1;
    int m = array.get(mI);
    if (m < key) {
      low = mI + 1;
    } else if (m > key) {
      high = mI - 1;
    } else {
      return mI;
    }
}
return -(low + 1);
```

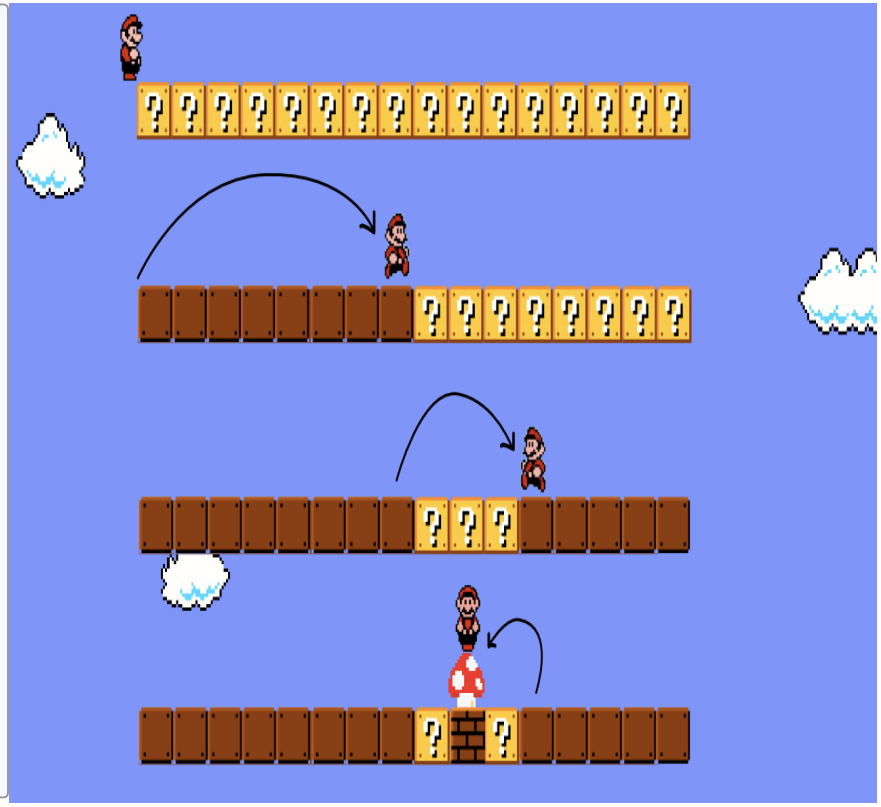


table de hachage

- valeur x stockée à l'index $h(x)$
- accès aléatoire à une valeur rapide
- accès ordonné pénible

accès ordonné pénible

- $[15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, \mathbf{1}, 14, 10, 7]$
- $[15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, \mathbf{2}, 1, 14, 10, 7]$
- $[15, \mathbf{3}, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]$
- $[15, 3, 0, 6, 11, \mathbf{4}, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]$
- $[15, 3, 0, 6, 11, 4, \mathbf{5}, 9, 12, 13, 8, 2, 1, 14, 10, 7]$
- $[15, 3, 0, \mathbf{6}, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]$

(Robin Hood, sondage linéaire, fonction de mixage MurmurHash3)

Opérations ensemblistes sur les tables de hachage

```
h1 <- hash set
h2 <- hash set
...
for(x in h1) {
   insert x in h2 // échec (mémoire tampon)
}
```

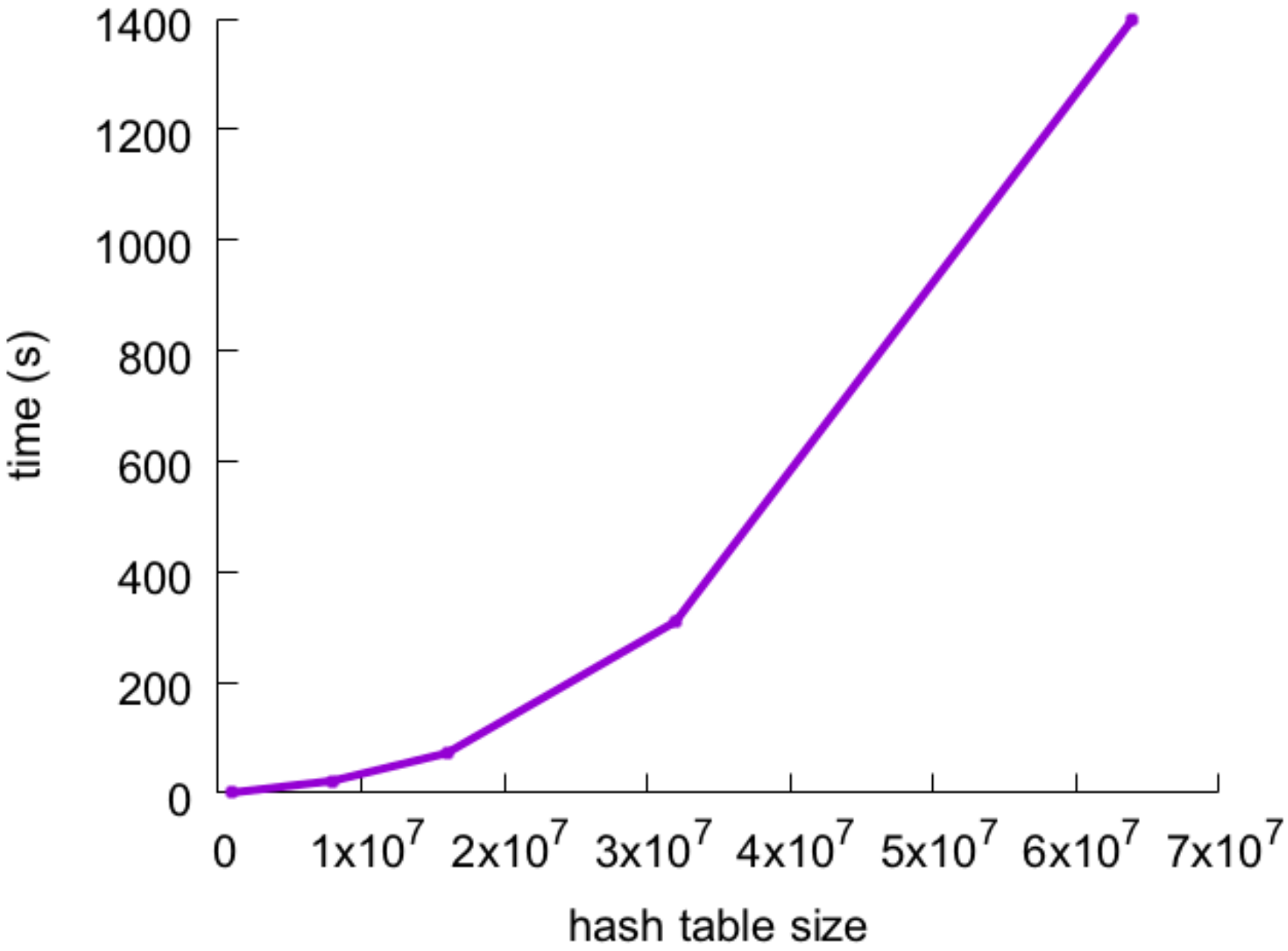
Faire "crasher" Swift

```
var S1 = Set<Int>(1...size)
var S2 = Set<Int>()
for i in S1 {
    S2.insert(i)
}
```

Quelques chiffres: une demi-heure pour 64M

taille	temps (s)
1M	0.8
8M	22
64M	1400

- Maps and sets can have quadratic-time performance https://lemire.me/blog/2017/01/30/maps-and-sets-can-have-quadratic-time-performance/
- Rust hash iteration+reinsertion https://accidentallyquadratic.tumblr.com/post/153545455987/rust-hash-iteration-reinsertion


1400
1200
1000
800
600
400
200
0
time (s)
0
1x10^7
2x10^7
3x10^7
4x10^7
5x10^7
6x10^7
7x10^7
hash table size

Les bitmaps ou bitsets

Façon efficace de représenter les ensembles d'entiers.

Par ex., 0, 1, 3, 4 devient `0b11011` ou "27".

- $\{0\} \rightarrow$ `0b00001`
- $\{0, 3\} \rightarrow$ `0b01001`
- $\{0, 3, 4\} \rightarrow$ `0b11001`
- $\{0, 1, 3, 4\} \rightarrow$ `0b11011`

Manipuler un bitmap

Processeur 64 bits.

Étant donné `x`, l'index du mot est `x/64` et l'index du bit est `x % 64`.

```
add(x) {
  array[x / 64] |= (1 << (x % 64))
}
```

Est-ce que c'est rapide?

```
index = x / 64          -> un shift
mask = 1 << ( x % 64)   -> un shift
array[ index ] |- mask -> un OR avec la mémoire
```

Un bit par ≈ 1.65 cycles à cause de la superscalarité

Parallélisme des bits

Intersection entre {0, 1, 3} et {1, 3} équivaut à une seule opération AND entre `0b1011` et `0b1010`.

Résultat est `0b1010` ou {1, 3}.

Aucun embranchement!

Les bitmaps bénéficient des mots étendus

- SIMD: Single Intruction Multiple Data
  - SSE (Pentium 4), ARM NEON 128 bits
  - AVX/AVX2 (256 bits)
  - AVX-512 (512 bits)

AVX-512 est maintenant disponible (par ex. chez Dell!) avec les processors ayant une microarchitecture Skylake-X.

Similarité ensembliste avec les bitmaps

- Jaccard/Tanimoto : $|S_1 \cap S_1|/|S_1 \cup S_2|$
- devient $\frac{|B_1 \text{ AND } B_2|}{|B_1 \text{ OR } B_2|}$
- 1.15 cycles par paire de mots (64-bit)
- Wojciech Muła, Nathan Kurz, Daniel Lemire
  Faster Population Counts Using AVX2 Instructions
  Computer Journal 61 (1), 2018
- (adopté par clang)

Les bitsets peuvent être gourmants

{1, 32000, 64000} : 1000 octets pour 3 nombres

On utilise donc la compression!

Qu'est-ce qu'on entend par compression?

- Modèle conventionnel: compresse, stocke, décompresse, traite;
  RAM $\longrightarrow$ CPU $\longrightarrow$ RAM $\longrightarrow$ CPU
- Modèle performant: compresse, stocke, traite *sans décompression*;
  RAM $\longrightarrow$ CPU

Git (GitHub) utilise EWAH

Codage par plage

Exemple: 000000001111111100 est
$00000000 - 11111111 - 00$

On peut coder les longues séquences de 1 ou de 0 de manière concise.

https://github.com/git/git/blob/master/ewah/bitmap.c

> [EWAH in Git] has already saved our users roughly a century of waiting for their fetches to complete (and the equivalent amount of CPU time in our fileservers).
> http://githubengineering.com/counting-objects/

- Après une comparaison exhaustive des techniques de compressions par plage sur les bitmaps, Guzun et al. (ICDE 2014) en arrive à la conclusion...

> EWAH offers the best query time for all distributions.

Complexité

- Intersection : $O(|S_1| + |S_2|)$ ou $O(\min(|S_1|, |S_2|))$
- Union en place ($S_2 \leftarrow S_1 \cup S_2$): $O(|S_1| + |S_2|)$ ou $O(|S_2|)$

Roaring Bitmaps

http://roaringbitmap.org/

- Apache Lucene, Solr et Elasticsearch, Metamarkets' Druid, Apache Spark, Apache Hive, Apache Tez, Netflix Atlas, LinkedIn Pinot, InfluxDB, Pilosa, Microsoft Visual Studio Team Services (VSTS), Couchbase's Bleve, Intel's Optimized Analytics Package (OAP), Apache Hivemall, eBay's Apache Kylin.
- Mise en oeuvre en Java, C, Go (interopérable)
- Point départ: thèse de S. Chambi (UQAM), co-dirigée avec Robert Godin

Modèle hybride

Décompose l'espace 32 bits en des sous-espaces de 16 bits. Au sein du sous-espace de 16 bits, utilisons la meilleure structure (contenant):

- tableau trié ({1,20,144})
- bitset (0b10000101011)
- plages ([0,10],[15,20])

C'est Roaring!

Travaux similaires: O'Neil's RIDBit + BitMagic

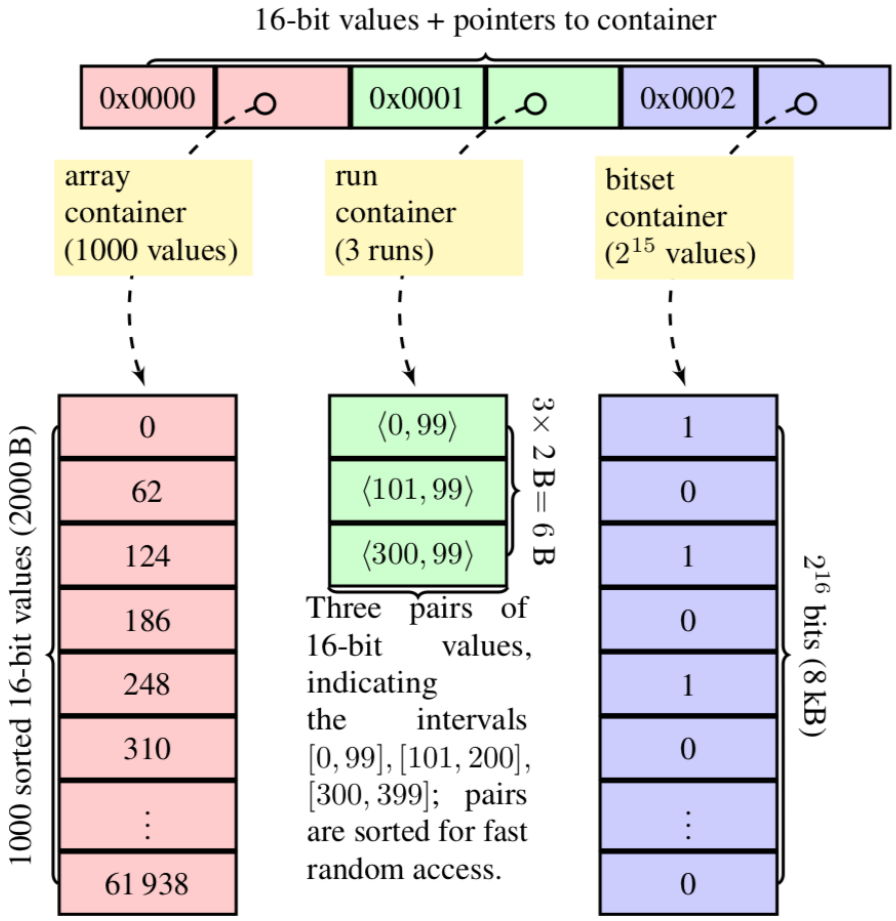


Figure 1. Roaring bitmap containing the first 1000 multiples of 62, all integers in the intervals $[2^{16}, 2^{16} + 100)$, $[2^{16} + 101, 2^{16} + 201)$, $[2^{16} + 300, 2^{16} + 400)$ and all even integers in $[2 \times 2^{16}, 3 \times 2^{16})$.

Voir https://github.com/RoaringBitmap/RoaringFormatSpec

Roaring

- Tous les contenants ont 8 kB ou moins (mémoire cache du processeur)
- On prédit le type de structure à la volée pendant les calculs
- Par ex. quand un tableau devient trop volumineux, on passe au bitset
- L'union de deux grands tableaux est prédite comme étant un bitset…
- Des dizaines d'heuristiques… réseaux de tri, etc.

> Use Roaring for bitmap compression whenever possible. Do not use other bitmap compression methods (Wang et al., SIGMOD 2017)

> kudos for making something that makes my software run 5x faster (Charles Parker, BigML)

Unions de 200 éléments

taille en bits par éléments :

	bitset	tableau	hachage	Roaring
census1881	524	32	195	15.1
weather	15.3	32	195	5.38

cycles par valeur par éléments :

	bitset	tableau	hachage	Roaring
census1881	9.85	542	1010	2.6
weather	0.35	94	237	0.16

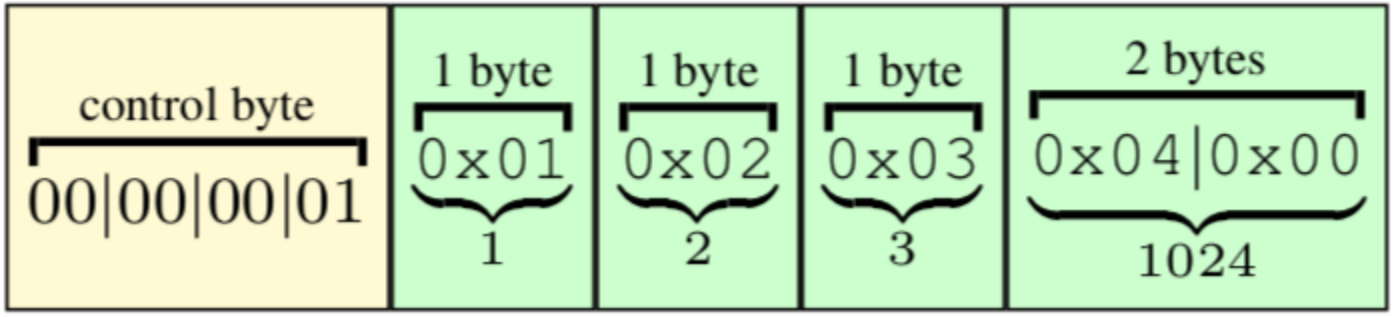
- Roaring Bitmaps: Implementation of an Optimized Software Library, Software: Practice and Experience Volume 48, Issue 4 April 2018 https://arxiv.org/abs/1709.07821

Compression d'entiers

- Technique "standard" : VByte, VarInt, VInt
- Utilisation de 1, 2, 3, 4, ... octets per entiers
- Utilise un bit par octet pour indique longueur des entiers en octets
- Lucene, Protocol Buffers, etc.
- 32 : 00100000
- 128 : 10000000 + 00000001

varint-GB de Google

- VByte: un embranchement par entier
- varint-GB: un embranchememement par bloc de 4 entiers
- chaque bloc de 4 entiers est précédé d'un octet 'descriptif'
- Exige de modifier le format (pas compatible avec VByte)

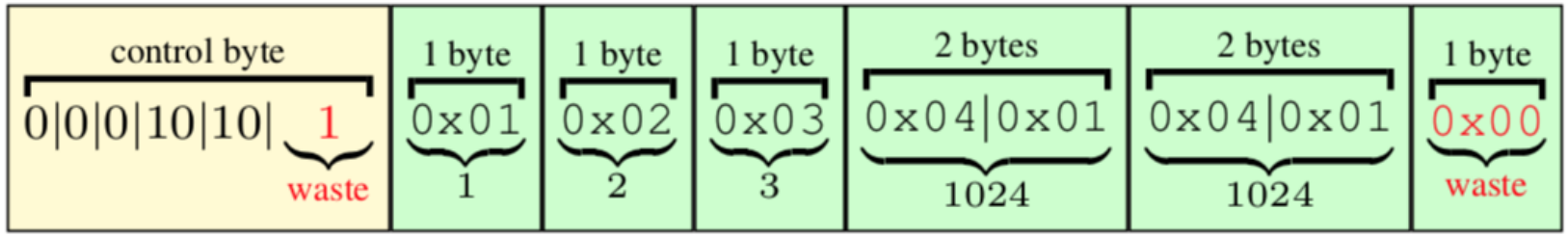


(b) Integer values 1, 2, 3, 1024 using 6 bytes

Accélération par vectorisation

- Stepanov (inventeur du STL en C++) travaillant pour Amazon a proposé varint-G8IU
- Stocke autant d'entiers que possible par blocs de 8 octets
- Utilise la vectorisation (SIMD)
- Exige de modifier le format (pas compatible avec VByte)
- *Protégé par un brevet*

SIMD-Based Decoding of Posting Lists, CIKM 2011
https://stepanovpapers.com/SIMD_Decoding_TR.pdf



(b) Integer values 1, 2, 3, 1024, 1024 compressed using 9 bytes

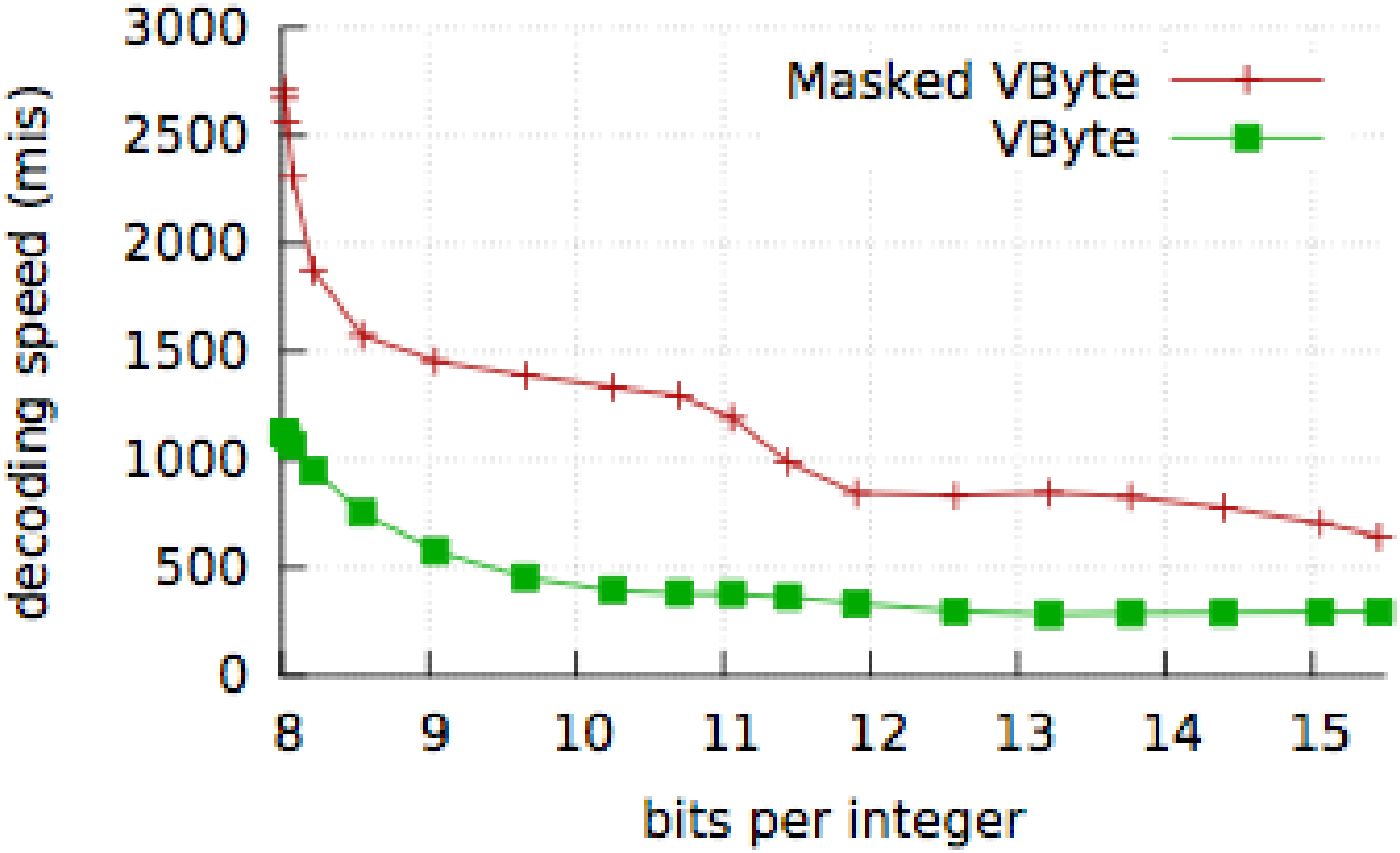
Observations de Stepanov et al. (partie 1)

- Incapable de vectoriser VByte (original)

Accélérer VByte (sans changer le format)

- C'est possible nonobstant l'échec de Stepanov et al.
- Décodeur avec SIMD: Masked VByte
- Travaux réalisés avec Indeed.com (en production)

Jeff Plaisance, Nathan Kurz, Daniel Lemire, Vectorized VByte Decoding, International Symposium on Web Algorithms 2015, 2015.


3000
2500
2000
1500
1000
500
0
decoding speed (mis)
Masked VByte
VByte
8
9
10
11
12
13
14
15
bits per integer

Observations de Stepanov et al. (partie 2)

- Possible de vectoriser le varint-GB de Google, mais moins performant que varint-G8IU

Stream VByte

- Reprend l'idée du varint-GB de Google
- Mais au lieu de mélanger les octets descriptifs avec le reste des données…
- On sépare les octets descriptifs du reste

Daniel Lemire, Nathan Kurz, Christoph Rupp
Stream VByte: Faster Byte-Oriented Integer Compression
Information Processing Letters 130, 2018

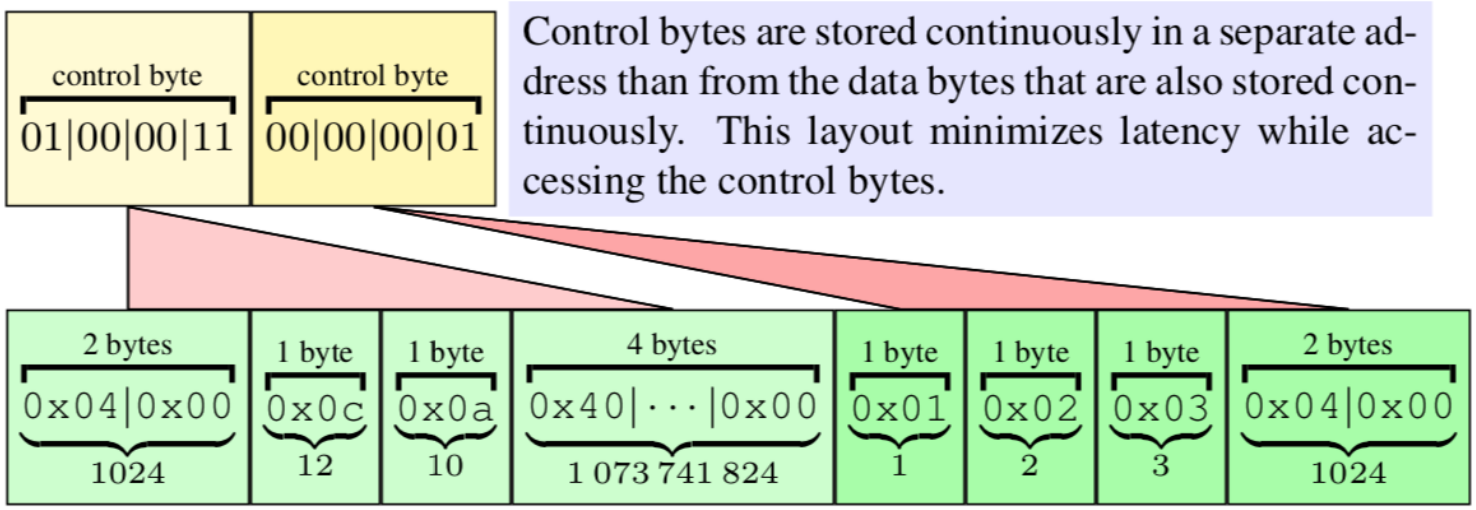
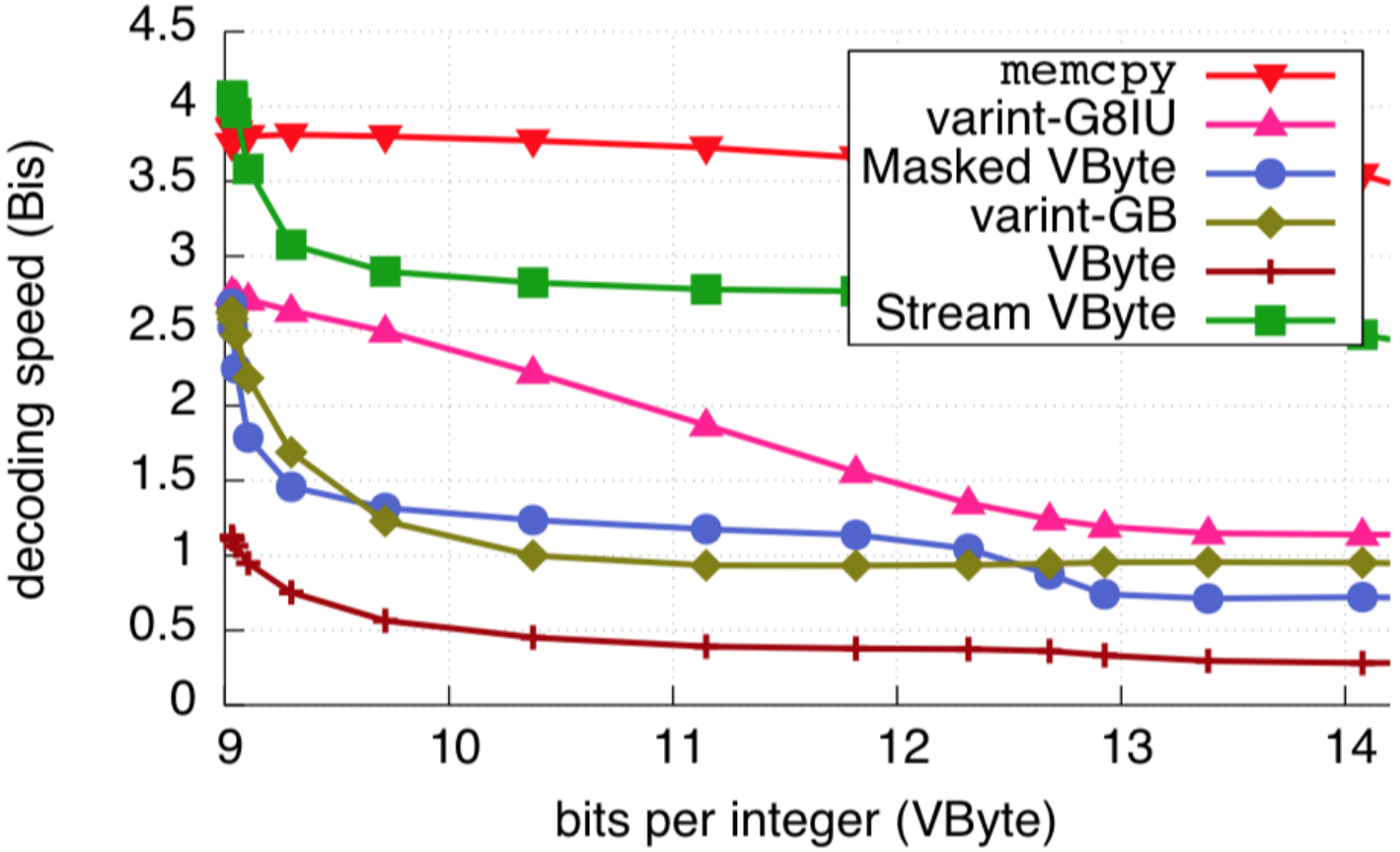


Figure 3: Compressed STREAM VBYTE bytes for 1024, 12, 10, 1 073 741 824, 1, 2, 3, 1024.


decoding speed (Bis)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
9
10
11
12
13
14
bits per integer (VByte)
memcpy
varint-G8IU
Masked VByte
varint-GB
VByte
Stream VByte

```
1 // "databytes" is a byte pointer to compressed data
2 // "control" contains control byte
3 uint8_t C = lengthTable[control]; // C is between 4 and 16
4 __m128i Data = _mm_loadu_si128((__m128i *) databytes);
5 __m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
6 Data = _mm_shuffle_epi8(Data, Shuf);  // final decoded data
7 datasource += C;
```

Figure 4: Core of the STREAM VBYTE decoding procedure in C with Intel intrinsics

Stream VByte est utilisé par...

- Redis (au sein de RediSearch) https://redislabs.com
- upscaledb https://upscaledb.com
- Trinity https://github.com/phaistos-networks/Trinity

> As you can see, Stream VByte is over 30% faster than the second fastest, (...) This is quite an impressive improvement in terms of query execution time, which is almost entirely dominated by postings list access time (i.e integers decoding speed).
> https://medium.com/@markpapadakis/trinity-updates-and-integer-codes-benchmarks-6a4fa2eb3fd1

Pour en savoir plus...

- Blogue (2 fois/semaine) : https://lemire.me/blog/
- GitHub: https://github.com/lemire
- Page personnelle : https://lemire.me/fr/
- CRSNG : *Faster Compressed Indexes On Next-Generation Hardware* (2017-2022)
- Twitter @lemire