Decoding Billions of Integers Through Vectorization

Daniel Lemire, Leonid Boytsov

Historical Context

Paper @ Software: Practice and Experience 2013

Journal emphasizing on software engineering practices

Leonid Boytsov (1997 Master, 2018 PhD)

Daniel Lemire (1998 PhD)

Daniel Lemire
lemire

Follow | Sponsor

Daniel Lemire is a computer science professor. His research is on software performance in data engineering.

7.4k followers · **70** following

lemire / README.md

Hi there 👋

I am a computer science professor at the Université du Québec. 🇨🇦

- 📰 I write about software performance weekly on my blog.
- 🏆 Among world's top 2% scientists (Stanford University/Elsevier ranking, 2025).
- 🎤 I gave a best voted talk at QCon San Francisco on JSON parsing.
- 📄 I wrote over 90 research papers including over 60 journal articles.
- 📖 I am editor of the Software: Practice and Experience journal (Wiley); it was founded in 1971.
- 📎 On-demand JSON: A better way to parse documents? was the most read article of the last 5 years at SPE (2024).
- 📎 Parsing millions of URLs per second was the most read article of the last 5 years at SPE (2025).
- 📘 My latest book: Mastering Programming: From Testing to Performance in Go

Pinned

simdjson/**simdjson** Public

Parsing gigabytes of JSON per second : used by Facebook/Meta Velox, the Node.js runtime, ClickHouse, WatermelonDB, Apache Doris, Milvus, StarRocks

C++ ☆ 22.6k ⑂ 1.2k

RoaringBitmap/**RoaringBitmap** Public

A better compressed bitset in Java: used by Apache Spark, Netflix Atlas, Apache Pinot, Tablesaw, and many others

Java ☆ 3.8k ⑂ 577

Objective & Overview

Encoding 32-bit integer arrays to **compress storage**…
And decoding **billions** of integers per second

How?

https://github.com/fast-pack/FastPFOR

- Using SIMD
- Better implementations
- Tweaks on previous schemes

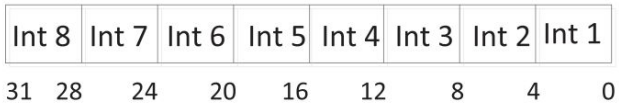
Why?

- To save storage/memory space and loading time
- Often, processing is faster than loading

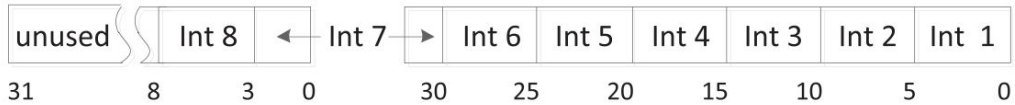
Fast Bit Packing/Unpacking

Given b-bit integers, pack them in bitstring, ideally aligned in 32-bit

They implemented in-house, for all b (C standard didn't offer)



(a) 4-bit integers

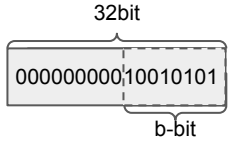


(b) 5-bit integers

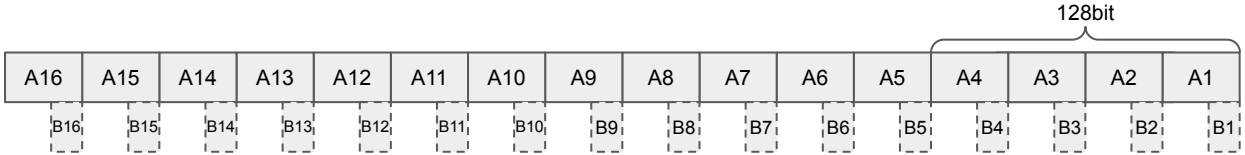
```
void unpack4_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))       & 15;
  *out++ = ((*in) >> 4)  & 15;
  *out++ = ((*in) >> 8)  & 15;
  *out++ = ((*in) >> 12) & 15;
  *out++ = ((*in) >> 16) & 15;
  *out++ = ((*in) >> 20) & 15;
  *out++ = ((*in) >> 24) & 15;
  *out =   ((*in) >> 28);
}
```

```
void unpack5_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))       & 31;
  *out++ = ((*in) >> 5 ) & 31;
  *out++ = ((*in) >> 10) & 31;
  *out++ = ((*in) >> 15) & 31;
  *out++ = ((*in) >> 20) & 31;
  *out++ = ((*in) >> 25) & 31;
  *out   = ((*in) >> 30);
  ++in;
  *out++ |= ((*in) & 7) << 2;
  *out =    ((*in) >> 3) & 31;
}
```

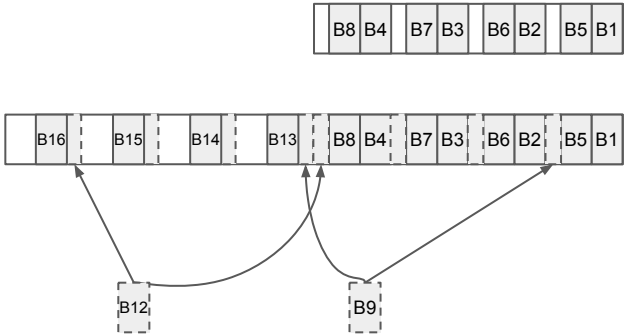
Fast Bit Packing/Unpacking with SIMD



Input
(32-bit aligned)



Output buffer
(b-bit packed)

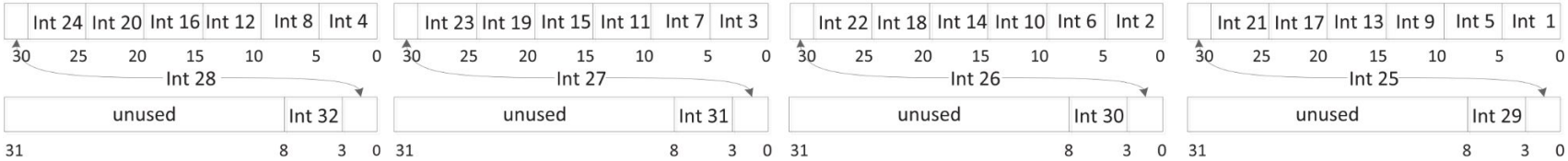


Overall, N * b bits from N 32-bit integers

‘Vertical’ bit packing (SIMD-friendly, no inter-lane shuffle)

Fast Bit Packing/Unpacking with SIMD

…Implemented for all b (1~32)



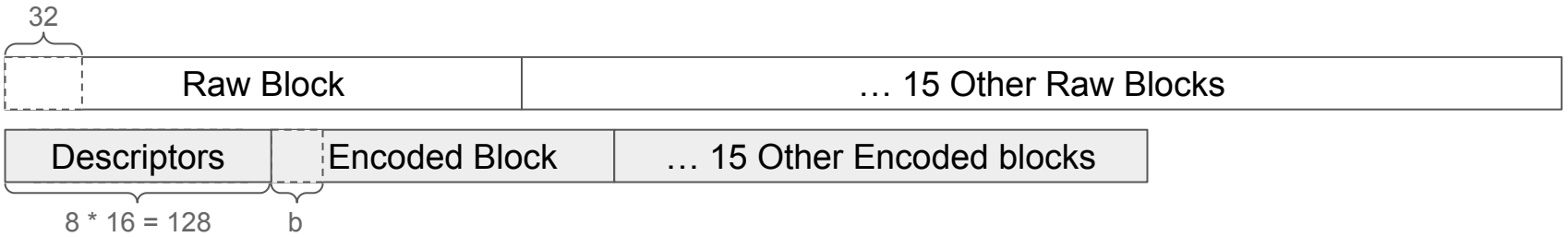

```
void unpack5_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))        & 31;
  *out++ = ((*in) >> 5 ) & 31;
  *out++ = ((*in) >> 10) & 31;
  *out++ = ((*in) >> 15) & 31;
  *out++ = ((*in) >> 20) & 31;
  *out++ = ((*in) >> 25) & 31;
  *out   = ((*in) >> 30);
  ++in;
  *out++ |= ((*in) & 7) << 2;
  *out =    ((*in) >> 3) & 31;
}
```

```
void SIMDunpack5_8(const __m128i* in,  __m128i* out) {
  __m128i i = _mm_load_si128(in);
  _mm_store_si128(out++, _mm_and_si128( i , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,5)  , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,10) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,15) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,20) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,25) , m31));
  __m128i  o = _mm_srli_epi32(i,30);
  i = _mm_load_si128(++in);
  o = _mm_or_si128(o, _mm_slli_epi32(_mm_and_si128(i, m7), 2));
  _mm_store_si128(out++, o);
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,3)  , m31));
}
```

SIMD-BP128

Binary Packing with SIMD, using 16 blocks with 128 integers

- Each block has **128** 32-bit integers, each with b
  - All 128 integers less than 2^b; result 128b bits – aligned nicely in 128-bit
- One meta-block has **16** such blocks
  - Along with encoded results, has extra descriptor for saving b's for each block
  - Use 8-bit for each b (1~32), so it fits in 8*16=128 bit word
  - Each meta-block input has 2048 integers, about 64KB (fits in L1 cache)



Patched Schemes: PFOR, PFD

For each block, decide length b (e.g. b=2)

For each integer, save only last b-bits

If cut-off, save their locations and remaining bits (exception) aside

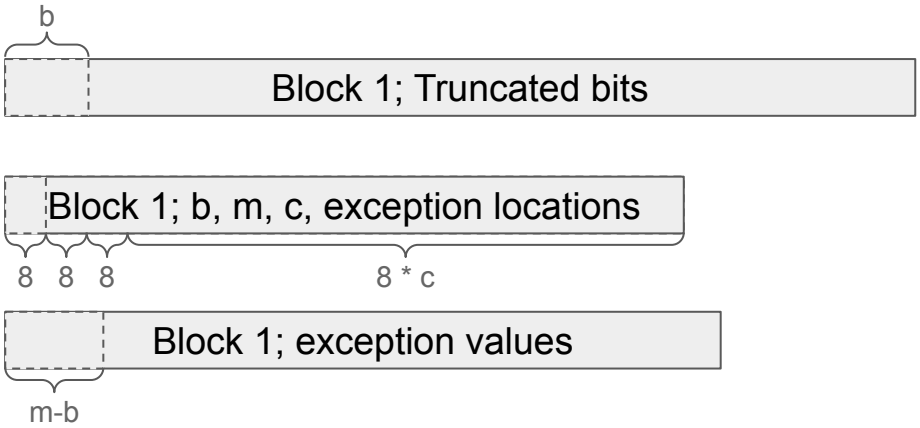
Data to be compressed:	... 10, 10, 1, 10, 100110, 10, 1, 11, 10, 100000, 10, 110100, 10, 11, 11, 1...
Truncated data: (16 × 2 = 32 bits)	... 10, 10, 01, 10, 10, 10, 01, 11, 10, 00, 10, 00, 10, 11, 11, 01 ...
Byte array: (6 × 8 = 48 bits)	... 2, 6, 3, 4, 9, 11 ...
Exception data: (to be compressed)	... 1001, 1000, 1101 ...

FastPFOR, SIMD-FastPFOR

Each block has 128 integers, and each page has 2^16 integers (=2^9 blocks)

For each block, get truncated bits (b), max bits (m), and number of exceptions (c)

Consider all b (to get c), and choose one minimizing the size (b*128 + c(m+8-b))

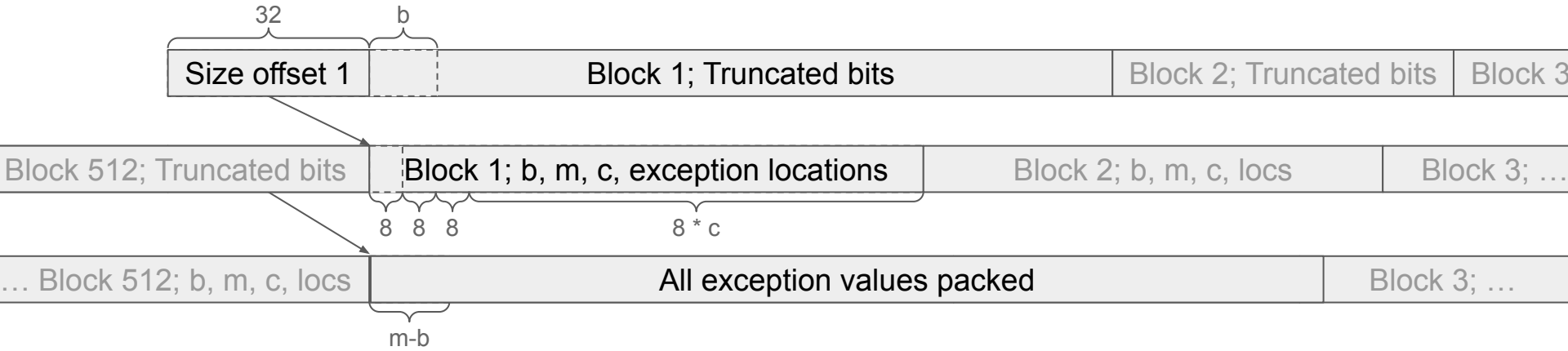


FastPFOR, SIMD-FastPFOR

For one page, each section is concatenated

Add size offsets to reach next section

Compress exception values with bit packing



For SIMD-FastPFOR, SIMD is used for bit packing for truncated and exception values

Differential Coding with SIMD

Save the difference between neighboring values, instead of raw values

- Why? In many data, closely located values have close values

For SIMD, use A(i) - A(i-4), instead of A(i) - A(i-1)

- Real world data had 4x larger differences, adding ~2 bits

In-place encoding/decoding

- Decoding needs two passes, chucked with 2^16 integers
- Two passes with cache still works better than using memory

Implementation Details

(Not using multiple threads)

Deciding length (b in BP, m in PFOR)

    Logarithm with SIMD OR and single bsr (Bit Scan Reverse)

SIMD requires 128 bit alignment, sometimes leads to less compression

Paper offers non-SIMD portable implementations, still in billion throughput

Results

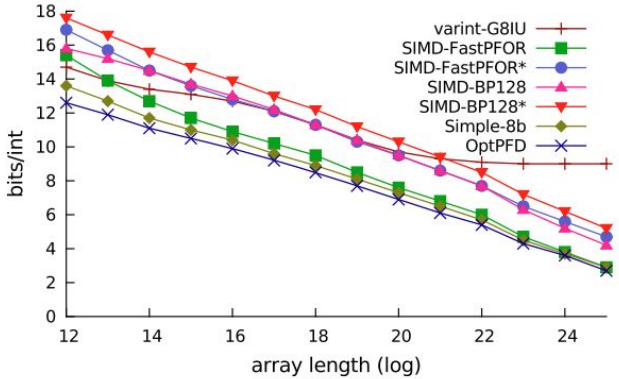
RAM bandwidth:

20GB/s ~= 5300mis

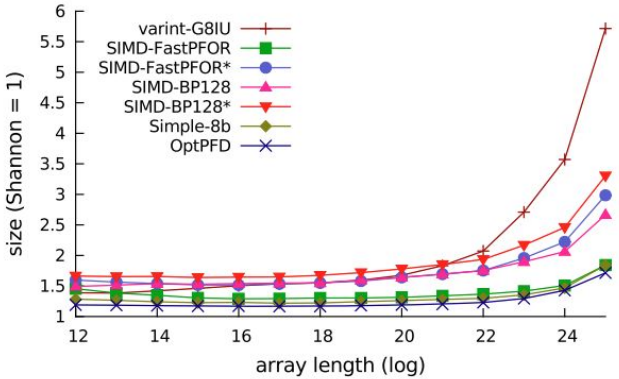
Used SSE3 instructions

Real-world text files

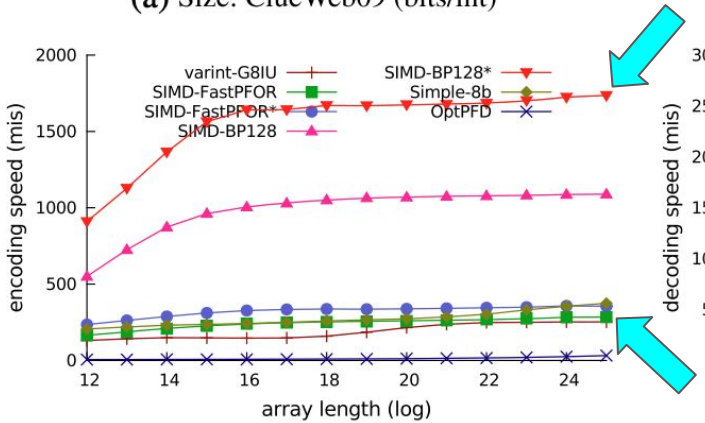
* are with diff encoding



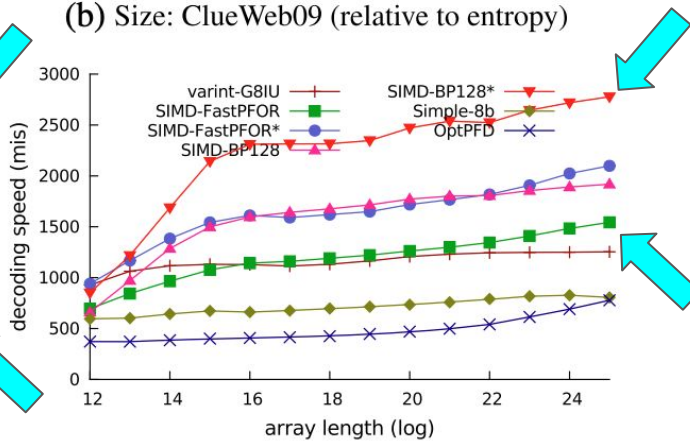
(a) Size: ClueWeb09 (bits/int)



(b) Size: ClueWeb09 (relative to entropy)

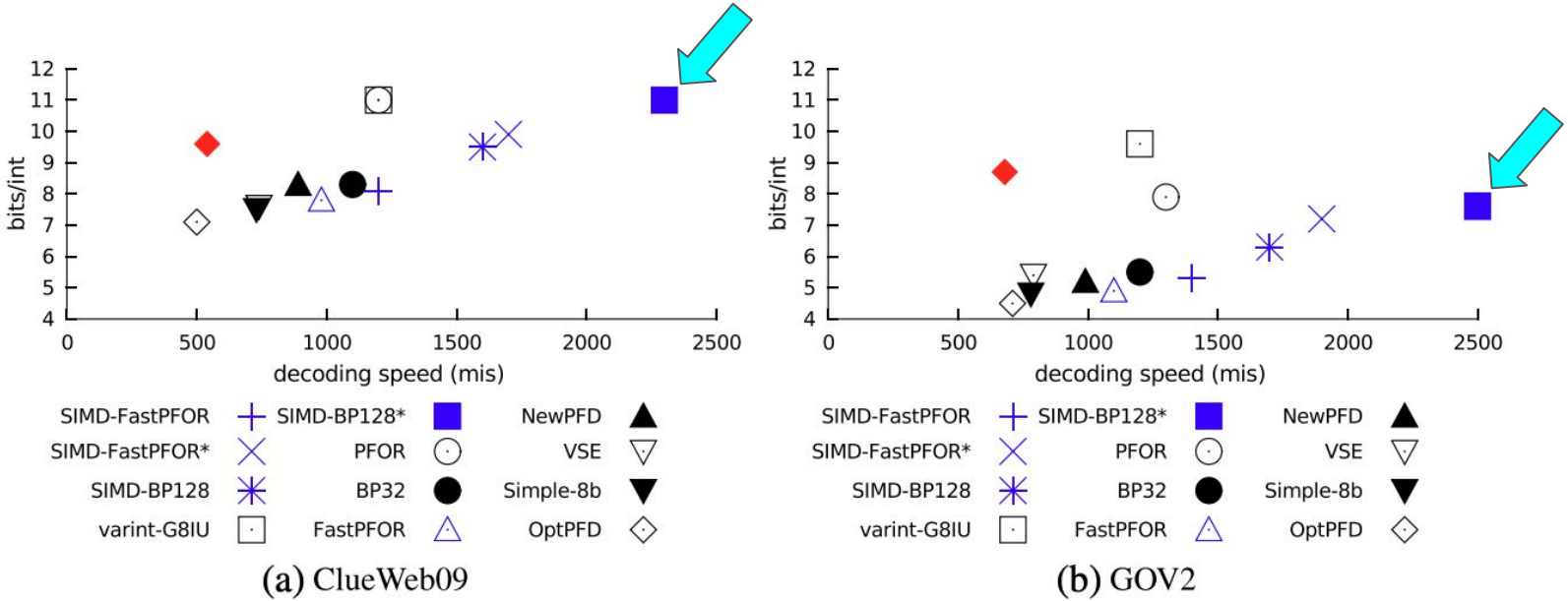


(c) Encoding: ClueWeb09



(d) Decoding: ClueWeb09

Results


bits/int
decoding speed (mis)
SIMD-FastPFOR
SIMD-FastPFOR*
SIMD-BP128
varint-G8IU
SIMD-BP128*
PFOR
BP32
FastPFOR
NewPFD
VSE
Simple-8b
OptPFD
(a) ClueWeb09
(b) GOV2

Results

Table V. Experimental results. Coding and decoding speeds are given in millions of 32-bit integers per second. Averages are weighted based on AOL query logs.

	(a) ClueWeb09			(b) GOV2		
	Coding	Decoding	Bits/int	Coding	Decoding	Bits/int
SIMD-BP128⋆	**1600**	**2300**	11	**1600**	**2500**	7.6
SIMD-FastPFOR⋆	330	1700	9.9	350	1900	7.2
SIMD-BP128	1000	1600	9.5	1000	1700	6.3
varint-G8IU⋆	220	1400	12	240	1500	10
SIMD-FastPFOR	250	1200	8.1	290	1400	5.3
PFOR2008	260	1200	10	250	1300	7.9
PFOR	330	1200	11	310	1300	7.9
varint-G8IU	210	1200	11	230	1300	9.6
BP32	760	1100	8.3	790	1200	5.5
SimplePFOR	240	980	7.7	270	1100	4.8
FastPFOR	240	980	7.8	270	1100	4.9
NewPFD	100	890	8.3	150	1000	5.2
VSEncoding	11	740	7.6	11	810	5.4
Simple-8b	280	730	7.5	340	780	4.8
OptPFD	14	500	**7.1**	23	710	**4.5**
Variable Byte	570	540	9.6	730	680	8.7

Reviewing the paper

Extensive experiments, with very good implementations

How did recent changes on HW affect these algorithm?

- Different SIMD standards, multicore

Is this applicable to other data types? (JSON, floats, strings, etc)

Can we avoid GPU memory transfer bottleneck, by encoding/decoding on the fly?

Random access or binary search performance?