Fast Decompression Lucene Codec

Ivan Mamontov imamontov@griddynamics.com

Jun 1st, 2015

About me

- Software engineer at Grid Dynamics
- I am interested in low-level system programming

Table of Contents

Compression in Lucene

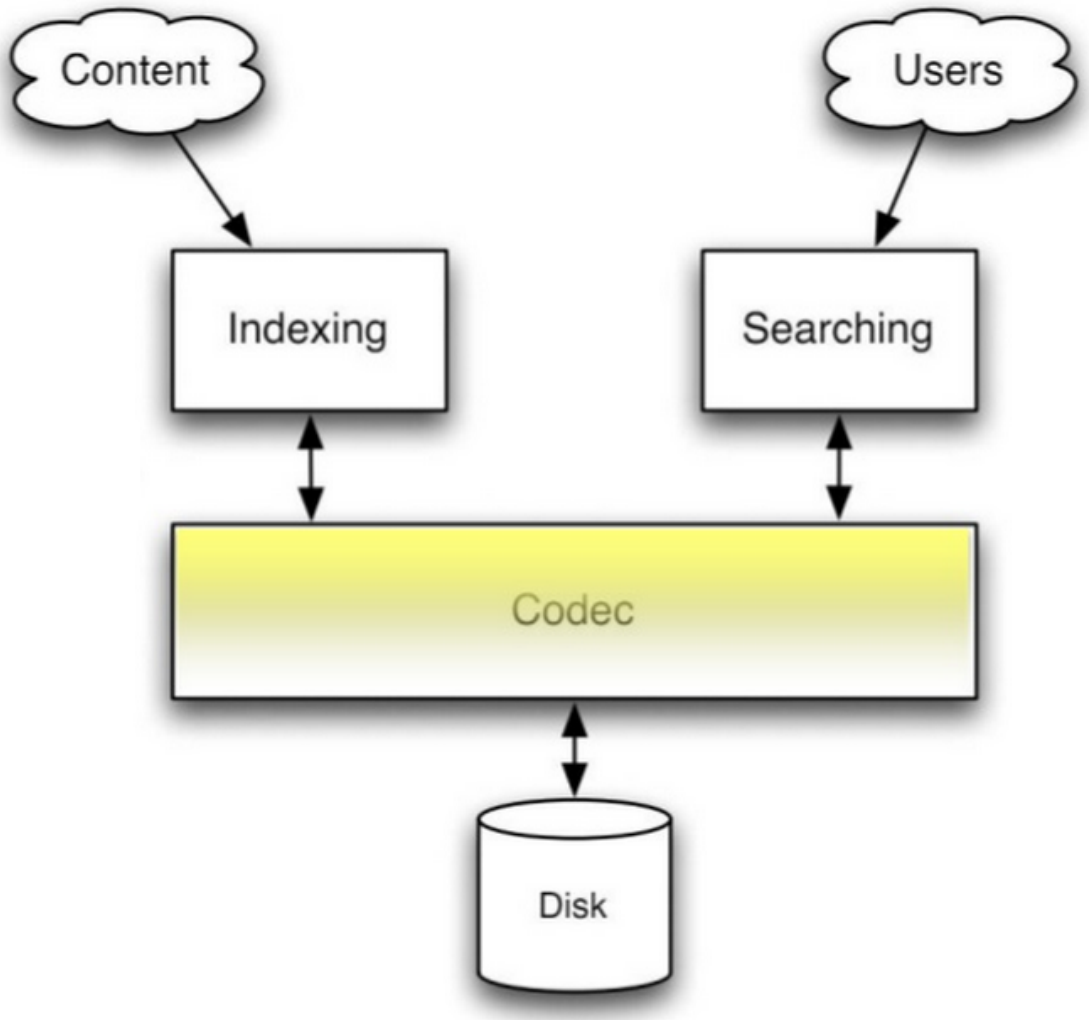
Requirements of a search index

- compress index as possible
  - minimize I/O
  - minimize index size
  - FS/Memory/CPU cache friendly
- avoid disc seeks
  - disc seek is $\approx$ 10ms

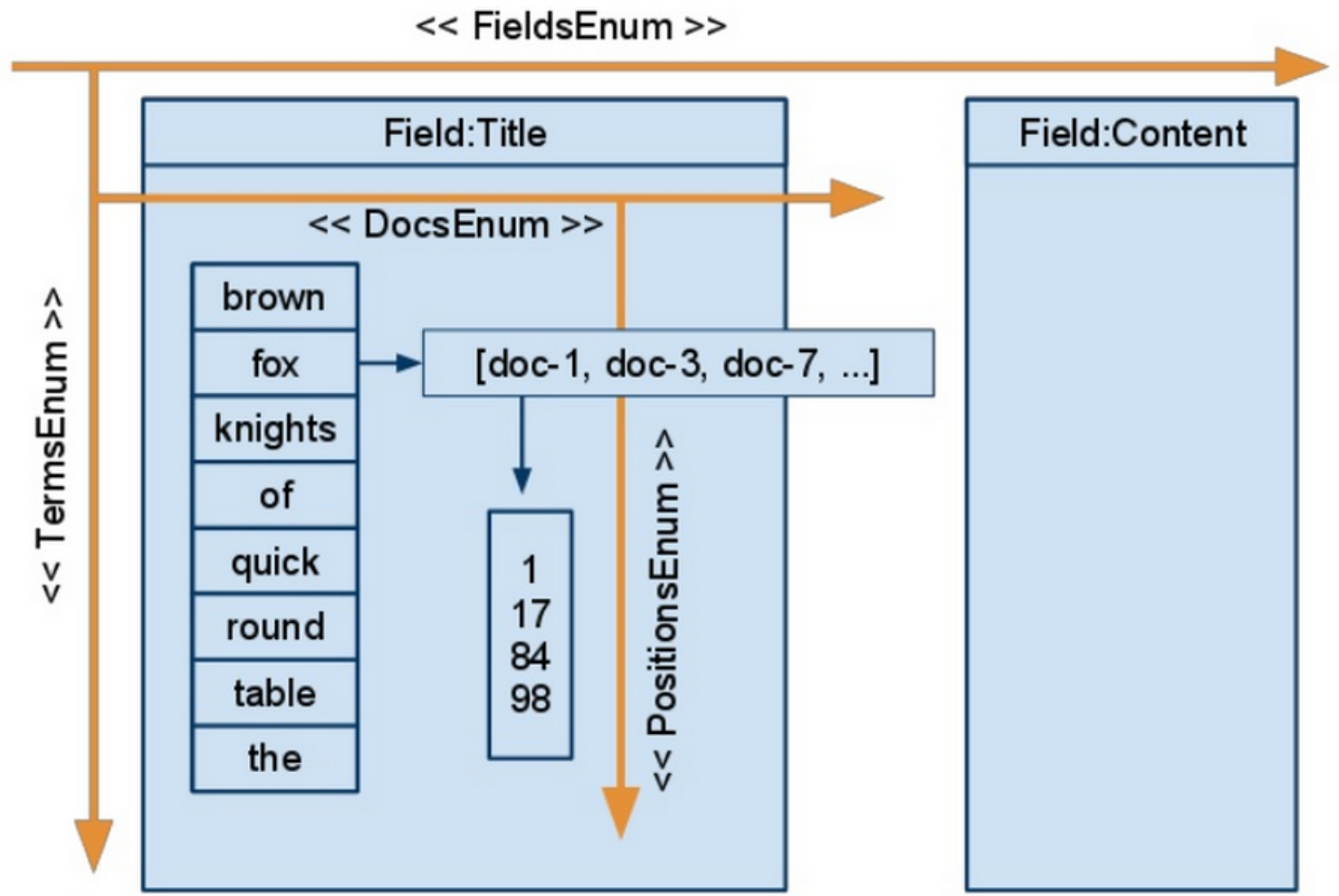
The numbers every engineer should know

- L1 cache reference 0.5 ns
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Main memory reference 100 ns
- Read 1 MB sequentially from memory 250,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns

Codec API



4D Codec API


<< FieldsEnum >>
Field:Title
Field:Content
<< DocsEnum >>
<< TermsEnum >>
brown
fox
knights
of
quick
round
table
the
[doc-1, doc-3, doc-7, ...]
1
17
84
98
<< PositionsEnum >>

Postings lists

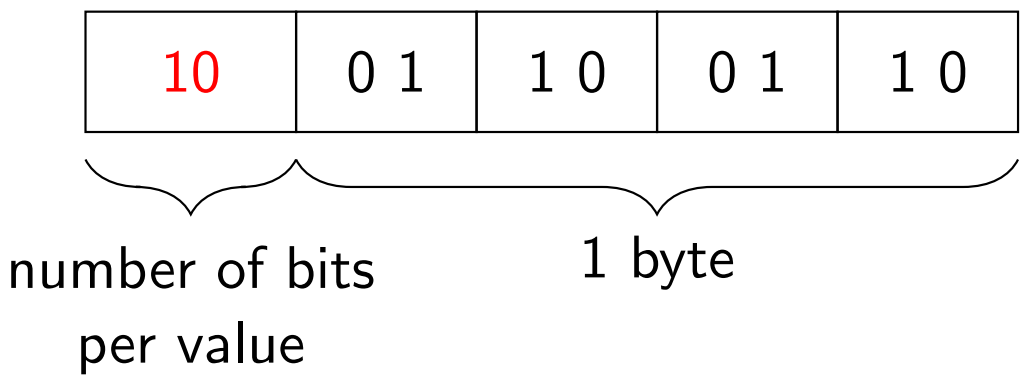
- Encoded using modified FOR delta
  1. uses delta
  2. splits into block of N=128 values
  3. bit packing per block
  4. remaining docs, encode with vint

Example with N=4

1,3,4,6,8,20,22,26,30,158
1,2,1,2,2,12,2,4,4,128
[1,2,1,2] [2,12,2,4] 4,128

What is FOR encoding?

To encode the following 4 numbers 1, 2, 1, 2:



FOR requires 1 byte instead of $4 * 4 = 16$ bytes!

What is FOR encoding?

- pros
  - great compression rate
  - fast decoding speed
  - **can be vectorized**
- cons
  - no random access within the block
  - the cost is determined by the largest delta in a block

Scalar vs. Vectors

Scalar vs. Vectors

```
float a[4], b[4], c[4];
...
for (int i = 0; i < 4; i++) {
 c[i] = a[i] + b[i];
}
```

- JIT $\approx$ 32 machine instructions
- gcc $\approx$ 24 machine scalar instructions
- gcc – 4 machine instructions with SSE2

Scalar vs. Vectors

$$\boxed{A_0} + \boxed{B_0} = \boxed{C_0}$$
$$\boxed{A_1} + \boxed{B_1} = \boxed{C_1}$$
$$\boxed{A_2} + \boxed{B_2} = \boxed{C_2}$$
$$\boxed{A_3} + \boxed{B_3} = \boxed{C_3}$$

$$\begin{array}{|c|}\hline A_0 \\ \hline A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline \end{array} + \begin{array}{|c|}\hline B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline \end{array} = \begin{array}{|c|}\hline C_0 \\ \hline C_1 \\ \hline C_2 \\ \hline C_3 \\ \hline \end{array}$$

Scalar vs. Vectors

- 75% fewer loads
- 75% fewer adds
- 75% fewer stores

Vectorization in HotSpot

- auto-vectorization – vector arithmetic is not supported yet. Only array initialization and array copy.
  - `http://bugs.java.com/view_bug.do?bug_id=6340864`
  - `http://bugs.java.com/view_bug.do?bug_id=7192383`
- explicit vectorization – JVM does not provide interfaces

Workaround

- write kernel code in C/C++
- call via JNI

Workaround

- write kernel code in C/C++
- call via JNI

The cost of the JNI call can be significant.

What makes JNI calls slow?

- Wrap object references to JNI handles.
- Obtain JNIEnv*, jclass/jobject and pass them as parameters.
- Lock an object monitor if the method is synchronized.
- **Call the native function.**
- Check if safepoint is needed.
- Unlock monitor if locked.
- Unwrap object result and reset JNI handles block.
- Handle JNI exceptions.

Java Critical Native

JDK-7013347 Critical Native

Critical native looks like JNI method:

- static and not synchronized
- not throw exceptions
- does not use wrappers
- works with primitives

See details in JDK-7013347

Benchmarks

Native FOR

A simple C library for compressing lists of integers `https://github.com/lemire/simdcomp` (thanks to Daniel Lemire, Leonid Boytsov)

- supports SSE2, SSE4.1, AVX
- uses C99 syntax
- uses SIMD intrinsics

Microbenchmark

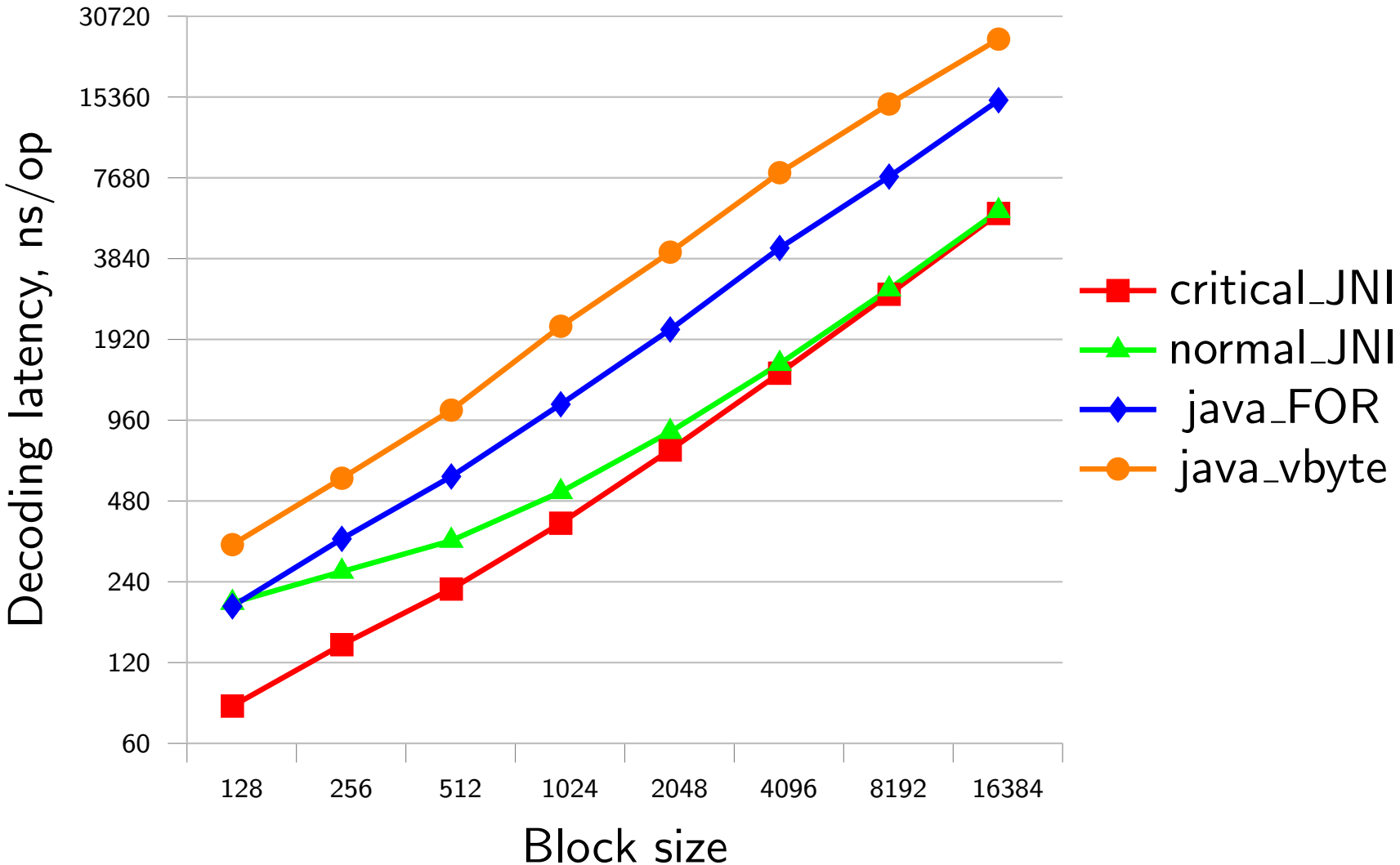
- java code
  - java_vint – classic vint implementation
  - java_FOR – classic FOR implementation
- JNI + native FOR implementation
  - normal_JNI – usual JNI call
  - critical_JNI – critical native call

Environment

- i5-4300M CPU @ 2.60GHz (Haswell)
- fedora 21 (kernel 3.17.4)
- JRE 1.8.0_40
- gcc 4.9.2

Decodes blocks with fixed size
Every block contains random elements with fixed density

Microbenchmark



SIMD codec

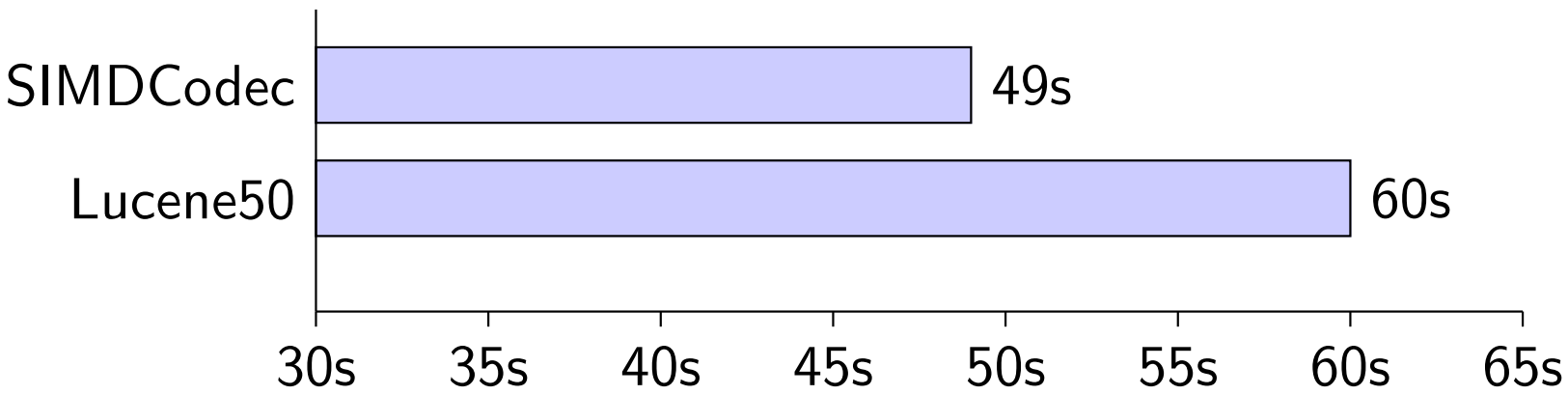
- based on Lucene50 codec
- uses `https://github.com/lemire/simdcomp` as native FOR implementation
- still in progress so it does not support
  - freqs
  - positions
  - offsets
  - payloads

Source code available at `http://git.io/vkY1o`

Lucene benchmark

- indexes all of Wikipedia's English XML export
  - only documents are indexed: term frequencies and positions are omitted
  - one large segment is used(about 1GB)
- measures how long it takes to search top 10K frequent terms
- environment
  - i5-4300M CPU @ 2.60GHz (Haswell)
  - fedora 21 (kernel 3.17.4)
  - JRE 1.8.0_40
  - gcc 4.9.2
- ant run-task -Dtask.alg=conf/searchOnlyWiki.alg -Dtask.mem=8G

Benchmark results



Future work

- Fast compression and intersection of lists of sorted integers `https://github.com/lemire/SIMDCompressionAndIntersection`
- Fast decoder for VByte-compressed integers `https://github.com/lemire/MaskedVByte`
- Native roaring codec
- Native facet component
- Native docvalues decoder

Thank you!