Elias-Fano Encoding

Succinct representation of monotone integer sequences with search operations

Giulio Ermanno Pibiri
giulio.pibiri@di.unipi.it
Computer Science Department
University of Pisa

21/06/2016
Consider a sequence $S[0,n)$ of $n$ positive and monotonically increasing integers, i.e., $S[i-1] \leq S[i]$ for $1 \leq i \leq n-1$, possibly repeated.

How to represent it as a bit vector in which each original integer is self-delimited, using as few as possible bits?
Consider a sequence $S[0,n)$ of $n$ positive and monotonically increasing integers, i.e., $S[i-1] \leq S[i]$ for $1 \leq i \leq n-1$, possibly repeated.

How to represent it as a bit vector in which each original integer is self-delimited, using as few as possible bits?

Huge research corpora describing different space/time trade-offs.

- Elias gamma/delta [Salomon-2007]
- Variable Byte [Salomon-2007]
- Varint-G8IU [Stepanov et al.-2011]
- Simple-9/16 [Anh and Moffat 2005-2010]
- PForDelta (PFD) [Zukowski et al.-2006]
- OptPFD [Yan et al.-2009]
- Binary Interpolative Coding [Moffat and Stuiver-2000]
Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_t, \ldots, L_{t^T}\}$ is the inverted index.
Inverted Indexes

Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_{t_1}, \ldots, L_{t_T}\}$ is the inverted index.
Given a textual collection $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the lexicon $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_t, \ldots, L_T\}$ is the inverted index.

$$T = \{\text{always, boy, good, house, hungry, is, red, the}\}$$
Inverted Indexes

Given a *textual collection* $D$, each document can be seen as a (multi-)set of terms. The set of terms occurring in $D$ is the *lexicon* $T$.

For each term $t$ in $T$ we store in a list $L_t$ the identifiers of the documents in which $t$ appears.

The collection of all inverted lists $\{L_t_1, \ldots, L_t_T\}$ is the inverted index.

$$T = \{\text{always, boy, good, house, hungry, is, red, the}\}$$
Inverted Indexes

Given a *textual collection* \(D\), each document can be seen as a (multi-)set of terms. The set of terms occurring in \(D\) is the *lexicon* \(T\).

For each term \(t\) in \(T\) we store in a list \(L_t\) the identifiers of the documents in which \(t\) appears.

The collection of all inverted lists \(\{L_t : t \in T\}\) is the inverted index.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
L_{t_1} = [1, 3] \\
L_{t_2} = [4, 5] \\
L_{t_3} = [1] \\
L_{t_4} = [2, 3] \\
L_{t_5} = [3, 5] \\
L_{t_6} = [1, 2, 3, 4, 5] \\
L_{t_7} = [1, 2, 4] \\
L_{t_8} = [2, 3, 5]
\]
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]

\[
q = \{\text{boy, is, the}\}
\]
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.

\[ T = \{ \text{always, boy, good, house, hungry, is, red, the} \} \]

\[ q = \{ \text{boy, is, the} \} \]

\[ L_{t_1} = [1, 3] \]
\[ L_{t_2} = [4, 5] \]
\[ L_{t_3} = [1] \]
\[ L_{t_4} = [2, 3] \]
\[ L_{t_5} = [3, 5] \]
\[ L_{t_6} = [1, 2, 3, 4, 5] \]
\[ L_{t_7} = [1, 2, 4] \]
\[ L_{t_8} = [2, 3, 5] \]
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[ T = \{\text{always, boy, good, house, hungry, is, red, the}\} \]

\[ q = \{\text{boy, is, the}\} \]

\[ L_{t_1} = [1, 3] \]
\[ L_{t_2} = [4, 5] \]
\[ L_{t_3} = [1] \]
\[ L_{t_4} = [2, 3] \]
\[ L_{t_5} = [3, 5] \]
\[ L_{t_6} = [1, 2, 3, 4, 5] \]
\[ L_{t_7} = [1, 2, 4] \]
\[ L_{t_8} = [2, 3, 5] \]
Inverted Indexes owe their popularity to the *efficient resolution of queries*, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[
T = \{\text{always, boy, good, house, hungry, is, red, the}\}
\]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]

\[
q = \{\text{boy, is, the}\} \\
q = \{\text{good, hungry}\}
\]
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[ T = \{\text{always, boy, good, house, hungry, is, red, the}\} \]

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]

q = \{\text{boy, is, the}\}  \\
q = \{\text{good, hungry}\}
Inverted Indexes owe their popularity to the efficient resolution of queries, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[ T = \{ \text{always, boy, good, house, hungry, is, red, the} \} \]

\[ L_{t_1} = [1, 3] \]
\[ L_{t_2} = [4, 5] \]
\[ L_{t_3} = [1] \]
\[ L_{t_4} = [2, 3] \]
\[ L_{t_5} = [3, 5] \]
\[ L_{t_6} = [1, 2, 3, 4, 5] \]
\[ L_{t_7} = [1, 2, 4] \]
\[ L_{t_8} = [2, 3, 5] \]

\[ q = \{ \text{boy, is, the} \} \]
\[ q = \{ \text{good, hungry} \} \]
Inverted Indexes owe their popularity to the **efficient resolution of queries**, such as: “return me all documents in which terms \{t_1, \ldots, t_k\} occur”.

\[ T = \{\text{always, boy, good, house, hungry, is, red, the}\}\]

\[ \begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*} \]

\[ q = \{\text{boy, is, the}\} \]

\[ q = \{\text{good, hungry}\} \]


Elias-Fano solution
Elias-Fano solution

\[ u = 43 \]
## Elias-Fano solution

The Elias-Fano solution for the given set of values is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Elias-Fano Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>000011</td>
</tr>
<tr>
<td>4</td>
<td>000100</td>
</tr>
<tr>
<td>7</td>
<td>001111</td>
</tr>
<tr>
<td>13</td>
<td>011101</td>
</tr>
<tr>
<td>14</td>
<td>011110</td>
</tr>
<tr>
<td>15</td>
<td>011111</td>
</tr>
<tr>
<td>21</td>
<td>010101</td>
</tr>
<tr>
<td>43</td>
<td>101011</td>
</tr>
</tbody>
</table>

**u = 43**
Elias-Fano solution

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\lg n]$</td>
<td>$[\lg(u/n)]$</td>
</tr>
<tr>
<td>0 0 0 0 1 1</td>
<td>3</td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>7</td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td>13</td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td>14</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>15</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>21</td>
</tr>
<tr>
<td>1 0 1 0 1 1</td>
<td>43</td>
</tr>
</tbody>
</table>

$u = 43$
## Elias-Fano solution

| high | low |  |  |  |  |  |
|------|-----|  |  |  |  |  |
| [lg \(n\)] | [lg(\(u/n\))] |  |  |  |  |  |
| 0 0 0 0  \textcolor{green}{0 1 1} | |  |  |  |  |  |
| 0 0 0 1 0 0 | |  |  |  |  |  |
| 0 0 0 1 1 1 | |  |  |  |  |  |
| 0 0 1 1 0 1 | |  |  |  |  |  |
| 0 0 1 1 1 0 | |  |  |  |  |  |
| 0 0 1 1 1 1 | |  |  |  |  |  |
| 0 1 1 1 1 1 | |  |  |  |  |  |
| 0 1 0 1 1 0 1 | |  |  |  |  |  |
| 1 0 1 0 1 1 | |  |  |  |  |  |

\[
u = 43\]
Elias-Fano solution

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>[lg n]</td>
<td>[lg(u/n)]</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

$u = 43$

$L = 011100111101110111101011$
Elias-Fano solution

\[
\begin{array}{c|c|c}
\text{high} & \text{low} & \text{u} \\
[lg(n)] & [lg(u/n)] & \\
0000 & 011 & 3 \\
0000 & 100 & 4 \\
0000 & 111 & 7 \\
0011 & 011 & 13 \\
0011 & 101 & 14 \\
0011 & 110 & 15 \\
0011 & 111 & 21 \\
0101 & 101 & \\
1010 & 111 & \\
\end{array}
\]

\[L = 0111001111011011110101111010111 \]

\[u = 43 \]
### Elias-Fano solution

<table>
<thead>
<tr>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\lg n]$</td>
<td>$[\lg(u/n)]$</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>

$u = \boxed{43}$

$L = \text{011100111101110111101011}$
Elias-Fano solution

\[
\begin{array}{cccc}
\text{high} & \text{low} \\
\lfloor \lg n \rfloor & \lfloor \lg (u/n) \rfloor \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

\[L = \text{0111001111011101111010111101011101011} \]

\[u = 43\]
Elias-Fano solution

\[
\begin{array}{c|c|c}
\text{high} & \text{low} & \text{value} \\
\hline
\lfloor \lg n \rfloor & \lfloor \lg (u/n) \rfloor & \\
0000 & 0111 & 3 \\
0000 & 1000 & 4 \\
0000 & 1111 & 7 \\
0011 & 1011 & 13 \\
0011 & 1100 & 14 \\
0011 & 1111 & 15 \\
0101 & 1010 & 21 \\
0101 & 1011 & 
\end{array}
\]

\[u = 43\]

\[L = 0111001111011101111010111\]
Elias-Fano solution

\[ L = 011100111101110111101011 \]

\[ u = 43 \]

\[ L = 011100111101110111101011 \]
Elias-Fano solution

```
missing
buckets
L = 011100111101110111101011
u = 43
L = 011100111101110111101011
```

- **[lg n]**
  - 0 0 0 0 1 1
  - 0 0 0 1 0 0
  - 0 0 0 1 1 1
  - 0 0 1 1 0 1
  - 0 0 1 1 1 0
  - 0 0 1 1 1 1
  - 0 1 0 1 0 1
  - 1 0 1 0 1 1

- **[lg(u/n)]**
  - 3 0 0 1 0
  - 3 0 0 1 1
  - 3 0 1 1 1
  - 1 0 1 0 1
  - 1 0 1 0 1
  - 1 0 1 0 1

- **Values for u and L:**
  - u = 43
  - L = 011100111101110111101011
Elias-Fano solution

\[
\begin{array}{c|c|c|c}
& \text{high} & \text{low} \\
\hline
[lg n] & [lg(u/n)] \\
\hline
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[ L = 011100111101110111101011 \]

\[ u = 43 \]

\[ L = 011100111101110111101011 \]
**Elias-Fano solution**

### Table

<table>
<thead>
<tr>
<th></th>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>000</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td>000</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>011</td>
</tr>
</tbody>
</table>

### Missing buckets

- 0 0 1 1
- 0 1 0 0
- 1 0 1 0
- 1 1 0 0
- 0 0 1 1

### L, H, u

- **L** = 011100111101110111101011
- **H** = 1110 1110 10 0 0 10 0 0 0
- **u** = 43

---

**3 1**

**4 2**

**7 3**

**13 4**

**14 5**

**15 6**

**21 7**

**H = 1110 1110 10 0 0 10 0 0 0**
EF(S[0,n)) = ?
EF(S[0,n)) = ?

\[ \lceil \log(u/n) \rceil \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]
\[ EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]
Properties - Space

$$EF(S[0,n)) = n \left\lceil \lg \left( \frac{u}{n} \right) \right\rceil$$

$L = 011100111101110111101011$

$H = 1110 1110 10 0 0 10 0 0$

$n$ ones
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor \]

\[ \left\lfloor \lg(u/n) \right\rfloor \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]

\( n \) ones

We store a 0 whenever we change bucket.
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor \]

\[ \left\lfloor \lg(u/n) \right\rfloor \]

\[ L = 011100111101110111101011 \]
\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]

We store a 0 whenever we change bucket.

\[ n \text{ ones} \]
\[ 2^{\lfloor \lg n \rfloor} \text{ zeros} \]
Properties - Space

\[ \text{EF}(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

\[ \overline{\lg(u/n)} \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]

n ones

\[ 2^{\left\lfloor \lg n \right\rfloor} \] zeros

We store a 0 whenever we change bucket.
Properties - Space

\[ EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]

n ones

\[ 2^{\lceil \lg n \rceil} \] zeros
Properties - Space

\[ \text{EF}(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

\[ [\log(u/n)] \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \ 0 \]

n ones
Properties - Space

\[ EF(S[0,n]) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

\[ L = 011100111101110111101011 \]

\[ H = 1110 \ 1110 \ 10 \ 0 \ 0 \ 10 \ 0 \ 0 \]
\[ EF(S[0,n]) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?
Properties - Space

\[ EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is \( \left\lceil \lg |\mathcal{X}| \right\rceil \) bits.
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( X \) is

\[ \left\lfloor \log |X| \right\rfloor \text{ bits.} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| \text{?} \]
Properties - Space

$$EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits}$$

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set $X$ is

$$\left\lfloor \lg |X| \right\rfloor \text{ bits.}$$

$X$ is the set of all monotone sequence of length $n$ drawn from a universe $u$.

$|X| ?$

000000000000000000
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( X \) is

\[ \left\lfloor \log |X| \right\rfloor \text{ bits.} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X|? \]

00010000000000000000
Properties - Space

\[
\text{EF}(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits}
\]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( X \) is

\[
\left\lceil \log |X| \right\rceil \text{ bits.}
\]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[
|X| ?
\]

0001001000000000000

3 6
Properties - Space

\[ EF(S[0,n]) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is

\[ \left\lfloor \log |\mathcal{X}| \right\rfloor \text{ bits.} \]

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |\mathcal{X}| ? \]

000100100010000000

3 6 10
Properties - Space

EF(S[0,n)) = \( n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \) bits

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is

\[ \left\lfloor \log | \mathcal{X} | \right\rfloor \] bits.

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ | \mathcal{X} | ? \]

0001001000110000000

3 6 1011
Properties - Space

\[ EF(S[0,n)) = n \left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is \( \left\lceil \lg |\mathcal{X}| \right\rceil \) bits.

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[
\begin{array}{cccc}
000 & 1 & 001 & 0001110000001 \\
\hline
3 & 6 & 1011 & 17 \\
\end{array}
\]
Properties - Space

\[ EF(S[0,n)) = n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( X \) is

\[ \left\lceil \log |X| \right\rceil \text{ bits.} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| ? \]

000100100011000001

3 6 1011 17

With possible repetitions!

*(weak monotonicity)*
Properties - Space

\[ EF(S[0,n]) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is

\[ \left\lfloor \log |\mathcal{X}| \right\rfloor \text{ bits.} \]

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |\mathcal{X}| = \binom{u+n}{n} \]

0001001000110000001

3 6 1011 17

With possible repetitions!

(weak monotonicity)
Properties - Space

$$\text{EF}(S[0,n)) = n \left\lfloor \lg \frac{u}{n} \right\rfloor + 2n \text{ bits}$$

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set $\mathcal{X}$ is

$$\left\lfloor \lg |\mathcal{X}| \right\rfloor \text{ bits.}$$

$\mathcal{X}$ is the set of all monotone sequence of length $n$ drawn from a universe $u$. 

$$|\mathcal{X}| = \binom{u+n}{n}$$

$$\left\lfloor \lg \binom{u+n}{n} \right\rfloor \approx n \lg \frac{u+n}{n}$$

With possible repetitions!

(weak monotonicity)
Properties - Space

\[ \text{EF}(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not?

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( X \) is

\[ \left\lfloor \log |X| \right\rfloor \text{ bits.} \]

\( X \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |X| = \binom{u+n}{n} \]

\[ \left\lfloor \log \left( \binom{u+n}{n} \right) \right\rfloor \approx n \log \frac{u+n}{n} \]

With possible repetitions! (weak monotonicity)
Properties - Space

\[ EF(S[0,n)) = n \left\lfloor \log \frac{u}{n} \right\rfloor + 2n \text{ bits} \]

Is it good or not? (less than half a bit away [Elias-1974])

**Information Theoretic Lower Bound**

The minimum number of bits needed to describe a set \( \mathcal{X} \) is

\[ \left\lfloor \log |\mathcal{X}| \right\rfloor \text{ bits.} \]

\( \mathcal{X} \) is the set of all monotone sequence of length \( n \) drawn from a universe \( u \).

\[ |\mathcal{X}| = \binom{u+n}{n} \]

\[ \left\lfloor \log \binom{u+n}{n} \right\rfloor \approx n \log \frac{u+n}{n} \]

With possible repetitions! (weak monotonicity)
access to each $S[i]$ in $O(1)$ worst-case
Properties - Operations

**access** to each $S[i]$ in $O(1)$ worst-case

$$\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}$$

$$\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}$$

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case
**Properties - Operations**

**access** to each $S[i]$ in $O(1)$ worst-case

**predecessor**($x$) = $\max\{S[i] \mid S[i] < x\}$

**successor**($x$) = $\min\{S[i] \mid S[i] \geq x\}$

queries in $O\left(\log \frac{u}{n}\right)$ worst-case
Properties - Operations

**access** to each $S[i]$ in $O(1)$ worst-case

\[
\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}
\]
\[
\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}
\]

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…
**Properties - Operations**

access to each $S[i]$ in $O(1)$ worst-case

\[
\text{predecessor}(x) = \max\{S[i] \mid S[i] < x\}
\]

\[
\text{successor}(x) = \min\{S[i] \mid S[i] \geq x\}
\]

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…

they need $o(n)$ bits more space in order to support fast \textit{rank/select} primitives on bitvector $H$
access to each $S[i]$ in $O(1)$ worst-case.

\[
\begin{align*}
\text{predecessor}(x) &= \max\{S[i] \mid S[i] < x\} \\
\text{successor}(x) &= \min\{S[i] \mid S[i] \geq x\}
\end{align*}
\]

queries in $O\left(\lg \frac{u}{n}\right)$ worst-case

but…

they need $o(n)$ bits more space in order to support fast rank/select primitives on bitvector $H$.
Definition

Given a bitvector $B$ of $n$ bits:

$\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0,i)$

$\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$
Definition
Given a bitvector $B$ of $n$ bits:

- $\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0,i)$
- $\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$

Examples

$B = 101011010101111010110101$
**Succinct rank/select**

**Definition**
Given a bitvector $B$ of $n$ bits:
- $\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0,i)$
- $\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$

**Examples**
- $B = 101011010101111010110101$
- $\text{rank}_0(5) = 2$
Succinct rank/select

Definition
Given a bitvector $B$ of $n$ bits:

$\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0,i)$

$\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$

Examples

$B = 101011010101111010110101$

$\text{rank}_0(5) = 2$

$\text{rank}_1(7) = 4$
Succinct rank/select

Definition
Given a bitvector $B$ of $n$ bits:

$$\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0, i)$$

$$\text{select}_{0/1}(i) = \text{ position of } i\text{-th } 0/1$$

Examples

$B = 101011010101111010110101$

rank$_0$(5) = 2    select$_0$(5) = 10
rank$_1$(7) = 4
Succinct rank/select

Definition
Given a bitvector $B$ of $n$ bits:
- $\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0, i)$
- $\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$

Examples
$B = 101011010101111010110101$
- $\text{rank}_0(5) = 2$, $\text{select}_0(5) = 10$
- $\text{rank}_1(7) = 4$, $\text{select}_1(7) = 11$
**Succinct rank/select**

**Definition**
Given a bitvector $B$ of $n$ bits:

$\text{rank}_{0/1}(i) = \# \text{ of } 0/1 \text{ in } [0, i)$

$\text{select}_{0/1}(i) = \text{position of } i\text{-th } 0/1$

**Examples**

$B = 1010110101011110101101$

$\text{rank}_0(5) = 2 \quad \text{select}_0(5) = 10$

$\text{rank}_1(7) = 4 \quad \text{select}_1(7) = 11$

**Relations**

$\text{rank}_{1/0}(\text{select}_{0/1}(i)) = \text{select}_{0/1}(i) - i$

$\text{rank}_{0/1}(\text{select}_{0/1}(i)) = i - 1$

$\text{rank}_{0/1}(i) + \text{rank}_{1/0}(i) = i$
Succinct rank/select

O(1)-solutions with o(n) bits

**rank**
(multi)-layered index + precomputed table

**select**
three-level directory tree

[Clark-1996]

[Jacobson-1989]
Succinct rank/select

O(1)-solutions with o(n) bits

rank (multi)-layered index + precomputed table [Jacobson-1989]

select three-level directory tree [Clark-1996]
Succinct rank/select

O(1)-solutions with o(n) bits

**rank**
(multi)-layered index + precomputed table  [Jacobson-1989]

- $2^{30}$ bits $\rightarrow \sim 67\%$ more bits!

**select**
three-level directory tree  [Clark-1996]

- $2^{30}$ bits $\rightarrow \sim 60\%$ more bits!
**Succinct rank/select**

O(1)-solutions with o(n) bits

<table>
<thead>
<tr>
<th>rank</th>
<th>(multi)-layered index + precomputed table</th>
<th>[Jacobson-1989]</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>three-level directory tree</td>
<td>[Clark-1996]</td>
</tr>
</tbody>
</table>

Nowadays *practical* solutions are based on [Vigna-2008, Zhou et al.-2013]:

- broadword programming
- interleaving
- Intel hardware *popcnt* instruction:
  - `Long().bitCount(x)` in Java
  - `__builtin_popcountl(x)` in C/C++
Succinct rank/select

O(1)-solutions with o(n) bits

**rank**  (multi)-layered index + precomputed table  
[Jacobson-1989]

2^{30} bits  →  ~67% more bits!

**select**  three-level directory tree  
[Clark-1996]

2^{30} bits  →  ~60% more bits!

Nowadays *practical* solutions are based on [Vigna-2008, Zhou *et al.*-2013]:

- broadword programming
- interleaving
- Intel hardware `popcnt` instruction:
  - `Long().bitCount(x)` in Java
  - `__builtin_popcountl(x)` in C/C++
S = [3, 4, 7, 13, 14, 15, 21, 43]
$S = [3, 4, 7, 13, 14, 15, 21, 43]$

`access(4) = S[4] = ?`
\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{access}(4) = S[4] = ? \]

\[ H = 1110111010001000 \]

\[ L = 011100111101110111101011 \]

\[ k = \lceil \log(u/n) \rceil \]
$S = [3, 4, 7, 13, 14, 15, 21, 43]$

$\text{access}(4) = S[4] = ?$

Recall: we store a 0 whenever we change bucket.

$H = 1110111010001000$

$L = 011100111101110111101011$

$k = \lceil \log(u/n) \rceil$
access example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{access}(4) = S[4] = ? \]

\[ H = 1110111010001000 \]
\[ L = 011100111101110111101011 \]
\[ k = \lceil \log(u/n) \rceil \]

Recall: we store a 0 whenever we change bucket.
access example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{access}(4) = S[4] = ? \]

Recall: we store a 0 whenever we change bucket.

\[ H = 1110111010001000 \]

\[ L = 011100111101110111101011 \]

\[ k = \lceil \log(u/n) \rceil \]

\[ \text{access}(i) = \text{select}_1(i) \]
access example

$S = [3, 4, 7, 13, 14, 15, 21, 43]$

access(4) = $S[4] = ?$

Recall: we store a 0 whenever we change bucket.

$H = 1110111010001000$

$L = 011100111101110111101011$

$k = \lceil \lg(u/n) \rceil$

access(i) = rank_0(\text{select}_1(i))$
access example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[
\begin{array}{cccccccc}
\text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} \\
\end{array}
\]

\[ \text{access}(4) = S[4] = 001000 \]

\[ H = 1110111010001000 \]
\[ L = 011100111101110111101011 \]
\[ k = \lceil \log(u/n) \rceil \]

\[ \text{access}(i) = \text{rank}_0(\text{select}_1(i)) \]

Recall: we store a 0 whenever we change bucket.
S = [3, 4, 7, 13, 14, 15, 21, 43]

access(4) = S[4] = 001000

Recall: we store a 0 whenever we change bucket.

H = 1110111010001000
L = 011100111101110111101011
k = \left\lceil \log(u/n) \right\rceil

access(i) = \text{rank}_0(\text{select}_1(i)) = \text{select}_1(i) - i
access example

S = [3, 4, 7, 13, 14, 15, 21, 43]

access(4) = S[4] = 001000

Recall: we store a 0 whenever we change bucket.

H = 1110111010001000
L = 011100111101110111101011
k = \lceil \log(u/n) \rceil

access(i) = select_1(i) - i
access example

$S = [3, 4, 7, 13, 14, 15, 21, 43]$

$access(4) = S[4] = 001\ 101$

Recall: we store a 0 whenever we change bucket.

$H = \text{1110111010001000}$

$L = \text{011100111101110111101011}$

$k = \lceil \log(u/n) \rceil$

$access(i) = select_1(i) - i \ll k \mid L[(i-1)k, ik]$
S = [3, 4, 7, 13, 14, 15, 21, 43]

access(4) = S[4] = 001101

Recall: we store a 0 whenever we change bucket.

H = 1110111010001000
L = \text{011100111101110111101011}

k = \lceil \lg(u/n) \rceil

access(i) = \select_1(i) - i \ll k \mid L[(i-1)k, ik)
access example

\[
S = [3, 4, 7, 13, 14, 15, 21, 43]
\]

Access examples:

access(4) = S[4] = 001101
access(7) = S[7] = ?

\[
H = 1110111010001000
\]

\[
L = 011100111101110111101011
\]

\[
k = \left\lceil \lg(u/n) \right\rceil
\]

\[
\text{access}(i) = \text{select}_{1}(i) - i \ll k | L[(i-1)k,ik)
\]
access example

S = [3, 4, 7, 13, 14, 15, 21, 43]

access(4) = S[4] = 001101
access(7) = S[7] = ?

Recall: we store a 0 whenever we change bucket.

H = 1110111010001000
L = 011100111101110111101011
k = \lceil \log(u/n) \rceil

access(i) = \text{select}_1(i) - i \ll k \mid L[(i-1)k, ik)
**access example**

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

access(4) = \( S[4] = 001101 \)

access(7) = \( S[7] = 010000 \)

\[ H = 1110111010001000 \]

\[ L = 011100111101110111101011 \]

\[ k = \lceil \lg(u/n) \rceil \]

access(i) = select1(i) - i \( \ll k \mid L[(i-1)k, ik) \)
access example

S = [3, 4, 7, 13, 14, 15, 21, 43]

access(4) = S[4] = 001101
access(7) = S[7] = 010101

Recall: we store a 0 whenever we change bucket.

H = 1110111010001000
L = 011100111101110111101011
k = [\lg(u/n)]

access(i) = select_1(i) - i \ll k \mid L[(i-1)k,ik)
$S = [3, 4, 7, 13, 14, 15, 21, 43]$

$\text{access}(4) = S[4] = 001\,101$

$\text{access}(7) = S[7] = 010\,101$

$H = 1110111010001000$

$L = 011100111101110111101011$

$k = \lceil \lg(u/n) \rceil$

$\text{access}(i) = \text{select}_1(i) - i \ll k \mid L[(i-1)k, ik)$
$S = [3, 4, 7, 13, 14, 15, 21, 43]$  

$H = 1110111010001000$  
$L = 011100111101110111101011$
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{successor}(12) = ? \]

\[ H = 1110111010001000 \]
\[ L = 011100111101110111101011 \]
S = [3, 4, 7, 13, 14, 15, 21, 43]

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

successor(12) = ?
001100

H = 1110111010001000
L = 011100111101110111101011
S = [3, 4, 7, 13, 14, 15, 21, 43]

successor(12) = ?

h_{12} = 001100

H = 11101111010001000
L = 011100111101110111101011
successor example

S = [3, 4, 7, 13, 14, 15, 21, 43]

successor(12) = ?

$h_{12} = \boxed{001100}$

$p_1 = \text{select}_0(h_x) - h_x$
$p_2 = \text{select}_0(h_x+1) - h_x - 1$

H = 1110111010001000
L = 011100111101101111010111101011
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[
\text{successor}(12) = ? \\
\text{h}_{12} = 001100
\]

\[
\begin{align*}
H &= 1110111010001000 \\
L &= 011100111101110111010111101011
\end{align*}
\]

\[
p_1 = \text{select}_0(h_x) - h_x \\
p_2 = \text{select}_0(h_x + 1) - h_x - 1
\]
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{successor}(12) = ? \]

\[ h_{12} = \boxed{001100} \]

\[ H = 1110111010001000 \]

\[ L = 011100111101110111101011 \]

\[ p_1 = \text{select}_0(h_x) - h_x \]
\[ p_2 = \text{select}_0(h_x+1) - h_x - 1 \]
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ h_{12} = 001100 \]

\[ \text{successor}(12) = ? \]

\[
\begin{align*}
H &= 1110111010001000 \\
L &= 011100111101110111101011
\end{align*}
\]

\[
\begin{align*}
p_1 &= \text{select}_0(h_x) - h_x \\
p_2 &= \text{select}_0(h_x + 1) - h_x - 1
\end{align*}
\]
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

\[ \text{successor}(12) = ? \]

\[ h_{12} = 001100 \]

\[ H = \text{1110111010001000} \]

\[ L = \text{011100111101110111101011} \]

\[ p_1 = \text{select}_0(h_x) - h_x \]

\[ p_2 = \text{select}_0(h_x+1) - h_x - 1 \]

binary search in \([p_1, p_2)\)
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

successor(12) = 13

\[ h_{12} = 001100 \]

\[ H = \overline{1110111010001000} \]
\[ L = 011100111101110111010111 \]

\[ \text{p}_1 = \text{select}_0(h_x) - h_x \]
\[ \text{p}_2 = \text{select}_0(h_x+1) - h_x - 1 \]

binary search in \([\text{p}_1, \text{p}_2]\)
successor example

\[ S = [3, 4, 7, 13, 14, 15, 21, 43] \]

successor(12) = 13

\[ h_{12} = 001100 \]

\[ H = 1110111010001000 \]

\[ L = 011100111101110111101011 \]

\[ p_1 = \text{select}_0(h_x) - h_x \]

\[ p_2 = \text{select}_0(h_x+1) - h_x - 1 \]

\[ \text{binary search in } [p_1, p_2] \]

Complexity: \( O\left(\lg \frac{u}{n}\right) \)
4 Intel i7-4790K cores (8 threads) clocked at 4Ghz, with 32 GB RAM, running Linux 4.2.0, 64 bits

C++11, compiled with gcc 5.3.0 with the highest optimisation setting

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>access</th>
<th>successor</th>
<th>iterated successor</th>
<th>iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>~2.4x10^6</td>
<td>~1.76x10^9</td>
<td>27.6 ns</td>
<td>0.24 µs</td>
<td>7.61 ns</td>
<td>2.34 ns</td>
</tr>
<tr>
<td>~10.5x10^6</td>
<td>~7.83x10^9</td>
<td>41.4 ns</td>
<td>0.29 µs</td>
<td>7.61 ns</td>
<td>2.36 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>uncompressed sequence bytes</th>
<th>Elias-Fano bytes</th>
<th>compression ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>~2.4x10^6</td>
<td>18,787,288</td>
<td>3,530,704</td>
<td>532%</td>
</tr>
<tr>
<td>~10.5x10^6</td>
<td>83,565,504</td>
<td>15,704,680</td>
<td>532%</td>
</tr>
</tbody>
</table>
Performance

Datasets

<table>
<thead>
<tr>
<th></th>
<th>Gov2</th>
<th>ClueWeb09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents</td>
<td>24,622,347</td>
<td>50,131,015</td>
</tr>
<tr>
<td>Terms</td>
<td>35,636,425</td>
<td>92,094,694</td>
</tr>
<tr>
<td>Postings</td>
<td>5,742,630,292</td>
<td>15,857,983,641</td>
</tr>
</tbody>
</table>

Space

<table>
<thead>
<tr>
<th></th>
<th>Gov2</th>
<th>ClueWeb09</th>
</tr>
</thead>
<tbody>
<tr>
<td>space GB</td>
<td>doc bpi</td>
<td>freq bpi</td>
</tr>
<tr>
<td>EF single</td>
<td>7.66 (+64.7%)</td>
<td>3.14 (+32.4%)</td>
</tr>
<tr>
<td>EF uniform</td>
<td>5.17 (+11.2%)</td>
<td>2.58 (+8.4%)</td>
</tr>
<tr>
<td>EF $\epsilon$-optimal</td>
<td>4.65</td>
<td>2.38</td>
</tr>
<tr>
<td>Interpolative</td>
<td>4.57 (-1.8%)</td>
<td>2.33 (-1.8%)</td>
</tr>
<tr>
<td>OptPFD</td>
<td>5.22 (+12.3%)</td>
<td>2.55 (+7.4%)</td>
</tr>
<tr>
<td>Varint-G8IU</td>
<td>14.06 (+202.2%)</td>
<td>8.98 (+278.3%)</td>
</tr>
</tbody>
</table>

AND queries (timings are in milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>TREC 05</th>
<th>TREC 06</th>
<th>TREC 05</th>
<th>TREC 06</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF single</td>
<td>2.1 (+10%)</td>
<td>4.7 (+1%)</td>
<td>13.6 (-5%)</td>
<td>15.8 (-9%)</td>
</tr>
<tr>
<td>EF uniform</td>
<td>2.1 (+9%)</td>
<td>5.1 (+10%)</td>
<td>15.5 (+8%)</td>
<td>18.9 (+9%)</td>
</tr>
<tr>
<td>EF $\epsilon$-optimal</td>
<td>1.9</td>
<td>4.6</td>
<td>14.3</td>
<td>17.4</td>
</tr>
<tr>
<td>Interpolative</td>
<td>7.5 (+291%)</td>
<td>20.4 (+343%)</td>
<td>55.7 (+289%)</td>
<td>76.5 (+341%)</td>
</tr>
<tr>
<td>OptPFD</td>
<td>2.2 (+14%)</td>
<td>5.7 (+24%)</td>
<td>16.6 (+16%)</td>
<td>21.9 (+26%)</td>
</tr>
<tr>
<td>Varint-G8IU</td>
<td>1.5 (-20%)</td>
<td>4.0 (-13%)</td>
<td>11.1 (-23%)</td>
<td>14.8 (-15%)</td>
</tr>
</tbody>
</table>

Numbers from [Ottaviano and Venturini-2014].

24 Intel Xeon E5-2697 Ivy Bridge cores (48 threads) clocked at 2.70Ghz, with 64 GB RAM, running Linux 3.12.7, 64 bits

C++11, compiled with gcc 4.9 with the highest optimisation setting.
1. Inverted Indexes


Killer applications

1. Inverted Indexes


2. Social Networks
Killer applications

1. Inverted Indexes


2. Social Networks

*Unicorn: A System for Searching the Social Graph*

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijncu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, Ning Zhang

Facebook, Inc.

ABSTRACT

Unicorn is an online, in-memory social graph-aware indexing system designed to search trillions of edges between tens of billions of users and entities on thousands of commodity servers. Unicorn is based on standard concepts in information retrieval and graph search. It provides a scalable and performant platform for graph search and provides on-demand execution of graph algorithms. The system is designed to enable rapid experimentation with new algorithms and data sources. The system is designed to enable rapid experimentation with new algorithms and data sources.

To the best of our knowledge, no other online graph retrieval system has ever been built with the scale of Unicorn. Unicorn is an excellent example of the evolution of Unicorn’s architecture, as well as the documentation for the major features and components of the system.
Killer applications

1. Inverted Indexes


2. Social Networks

**Unicorn: A System for Searching the Social Graph**

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijnucu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, Ning Zhang

Facebook, Inc.

**ABSTRACT**

Unicorn is an online, in-memory social graph-aware indexing system designed to search trillions of edges between tens of billions of users and entities on thousands of commodity servers. Unicorn is based on standard concepts in information retrieval and distributed systems, with a focus on providing performance guarantees on large graphs.

To the best of our knowledge, no other online graph retrieval system has ever been built with the scale of Unicorn.
Killer applications

1. Inverted Indexes


2. Social Networks

*Unicorn: A System for Searching the Social Graph*

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, Ning Zhang

Facebook, Inc.

**Abstract**

Unicorn is an online, in-memory social graph-aware indexing system designed to search trillions of edges between tens of billions of users and entities on thousands of commodity servers. Unicorn is based on standard concepts in information retrieval but has been designed to answer complex, natural-language queries on billions of nodes and billions of edges in milliseconds.

Open Source

All Unicorn index server and aggregator code is written in C++. Unicorn relies extensively on modules in Facebook's "Folly" Open Source Library [5]. As part of the effort of releasing Graph Search, we have open-sourced a C++ implementation of the Elias-Fano index representation [31] as part of Folly.
## Available Implementations

<table>
<thead>
<tr>
<th>Library</th>
<th>Author(s)</th>
<th>Link</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>folly</td>
<td>Facebook, Inc.</td>
<td><a href="https://github.com/facebook/folly">https://github.com/facebook/folly</a></td>
<td>C++</td>
</tr>
<tr>
<td>ds2i</td>
<td>Giuseppe Ottaviano, Rossano Venturini, Nicola Tonellotto</td>
<td><a href="https://github.com/ot/ds2i">https://github.com/ot/ds2i</a></td>
<td>C++</td>
</tr>
<tr>
<td>Sux</td>
<td>Sebastiano Vigna</td>
<td><a href="http://sux.di.unimi.it">http://sux.di.unimi.it</a></td>
<td>Java/C++</td>
</tr>
</tbody>
</table>
Elias-Fano encodes *monotone integer sequences* in *space close to the information theoretic minimum*, while allowing *powerful search operations*, namely *predecessor/successor* queries and random *access*.

Successfully applied to crucial problems, such as *inverted indexes* and *social graphs* representation.

Several *optimized* software implementations are available.
References


References


Thanks for your attention, time, patience!

Any questions?