Space- and Time-Efficient Data Structures for Massive Datasets

Giulio Ermanno Pibiri  
giulio.pibiri@di.unipi.it

Supervisor  
Rossano Venturini

Department of Computer Science  
University of Pisa

15/11/2018
The increase of information does not scale with technology.
The increase of information does not scale with technology.

“Software is getting slower more rapidly than hardware becomes faster.”
Niklaus Wirth, A Plea for Lean Software
The increase of information does not scale with technology.

“Software is getting slower more rapidly than hardware becomes faster.”

Niklaus Wirth, A Plea for Lean Software
Scenario

Data structures
PERFORMANCE

how quickly a program does its work - **faster** work

Algorithms
EFFICIENCY

how much work is required by a program - **less** work

- **time** ✓
- **space** ✗
Scenario

**Data structures**

PERFORMANCE

*how quickly a program does its work - faster work*

**Algorithms**

EFFICIENCY

*how much work is required by a program - less work*

**Data compression**

*space*  
*time*  
*less*  
*faster*
The dichotomy problem

Small vs. fast?
The dichotomy problem

Small vs. fast?
Choose one.
Small vs. fast? Choose one.

NO
Design **space-efficient** ad-hoc data structures, both from a theoretical *and* practical perspective, that support **fast data extraction**.

Data Compression & Fast Retrieval *together*. 
Achieved results

**Clustered Elias-Fano Indexes**
Giulio Ermanno Pibiri and Rossano Venturini
ACM Transactions on Information Systems (TOIS)

**Dynamic Elias-Fano Representation**
Giulio Ermanno Pibiri and Rossano Venturini
Annual Symposium on Combinatorial Pattern Matching (CPM)

**Variable-Byte Encoding is Now Space-Efficient Too**
Giulio Ermanno Pibiri and Rossano Venturini
arXiv (CoRR), April 2018.
Submitted to IEEE Transactions on Knowledge and Data Engineering (TKDE)

**Fast Dictionary-based Compression for Inverted Indexes**
Giulio Ermanno Pibiri, Matthias Petri and Alistair Moffat
ACM Conference on Web Search and Data Mining (WSDM)

**Efficient Data Structures for Massive N-Gram Datasets**
Giulio Ermanno Pibiri and Rossano Venturini
ACM Conference on Research and Development in Information Retrieval (SIGIR)
Full paper, 10 pages, 2017.

**Handling Massive N-Gram Datasets Efficiently**
Giulio Ermanno Pibiri and Rossano Venturini
### Achieved Results

**Clustered Elias-Fano Indexes**
Giulio Ermanno Pibiri and Rossano Venturini  
*ACM Transactions on Information Systems (TOIS)*  

**Dynamic Elias-Fano Representation**
Giulio Ermanno Pibiri and Rossano Venturini  
*Annual Symposium on Combinatorial Pattern Matching (CPM)*  

**Variable-Byte Encoding is Now Space-Efficient Too**
Giulio Ermanno Pibiri and Rossano Venturini  
arXiv (CoRR), April 2018.  
Submitted to *IEEE Transactions on Knowledge and Data Engineering (TKDE)*  

**Fast Dictionary-based Compression for Inverted Indexes**
Giulio Ermanno Pibiri, Matthias Petri and Alistair Moffat  
*ACM Conference on Web Search and Data Mining (WSDM)*  

**Efficient Data Structures for Massive N-Gram Datasets**
Giulio Ermanno Pibiri and Rossano Venturini  
*ACM Conference on Research and Development in Information Retrieval (SIGIR)*  
Full paper, 10 pages, 2017.

**Handling Massive N-Gram Datasets Efficiently**
Giulio Ermanno Pibiri and Rossano Venturini  
## Achieved results

**Clustered Elias-Fano Indexes**
Giulio Ermanno Pibiri and Rossano Venturini
ACM Transactions on Information Systems (TOIS)

**Dynamic Elias-Fano Representation**
Giulio Ermanno Pibiri and Rossano Venturini
Annual Symposium on Combinatorial Pattern Matching (CPM)

**Variable-Byte Encoding is Now Space-Efficient Too**
Giulio Ermanno Pibiri and Rossano Venturini
arXiv (CoRR), April 2018.
Submitted to IEEE Transactions on Knowledge and Data Engineering (TKDE)

**Fast Dictionary-based Compression for Inverted Indexes**
Giulio Ermanno Pibiri, Matthias Petri and Alistair Moffat
ACM Conference on Web Search and Data Mining (WSDM)

**Efficient Data Structures for Massive N-Gram Datasets**
Giulio Ermanno Pibiri and Rossano Venturini
ACM Conference on Research and Development in Information Retrieval (SIGIR)
Full paper, 10 pages, 2017.

**Handling Massive N-Gram Datasets Efficiently**
Giulio Ermanno Pibiri and Rossano Venturini
Problem 1

Consider a sorted integer sequence.
Consider a sorted integer sequence.

How to represent it as a bit-vector where each original integer is uniquely-decodable, using as few as possible bits?

How to maintain fast decompression speed?
Consider a sorted integer sequence.

How to represent it as a bit-vector where each original integer is uniquely-decodable, using as few as possible bits?

How to maintain fast decompression speed?

This is a difficult problem that has been studied since the '60.
Applications

Inverted indexes
- Google
- Yahoo!
- bing

Databases
- IBM
- Dropbox
- Oracle

RDF indexing
- ontotext
- Wikipedia

E-Commerce
- eBay
- Amazon

Geo-spatial data
- Google Maps
- Pokémon Go

Graph-compression
- Facebook
- LinkedIn
- Twitter
- Instagram
Applications

Inverted indexes
- Google
- Yahoo!
- Bing

Databases
- IBM
- Dropbox
- Oracle

RDF indexing
- Ontotext
- Wikipedia

E-Commerce
- eBay
- Amazon

Geo-spatial data
- Google Maps
- Pokémon Go

Graph-compression
- Facebook
- LinkedIn
- Instagram
The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.
The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.
Inverted indexes

The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.

{always, boy, good, house, hungry, is, red, the}
Inverted indexes

The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.
The inverted index is the de-facto data structure at the basis of every large-scale retrieval system.
The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.

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

\[
\begin{align*}
Lt_1 &= [1, 3] \\
Lt_2 &= [4, 5] \\
Lt_3 &= [1] \\
Lt_4 &= [2, 3] \\
Lt_5 &= [3, 5] \\
Lt_6 &= [1, 2, 3, 4, 5] \\
Lt_7 &= [1, 2, 4] \\
Lt_8 &= [2, 3, 5]
\end{align*}
\]
Inverted indexes owe their popularity to the *efficient resolution of queries*, such as:
“return 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 all documents in which terms \{t_1, \ldots, t_k\} occur”.

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

Let \( t_1 = \{\text{always, boy, good, house, hungry, is, red, the}\} \)

\[
\begin{align*}
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
Q &= \{\text{boy, is, the}\} \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]
Inverted indexes owe their popularity to the **efficient resolution of queries**, such as:

“return all documents in which terms \( \{t_1, \ldots, t_k\} \) occur”.

Inverted indexes

\[
\begin{align*}
Q &= \{\text{boy, is, the}\} \\
L_{t_1} &= [1, 3] \\
L_{t_2} &= [4, 5] \\
L_{t_3} &= [1] \\
L_{t_4} &= [2, 3] \\
L_{t_5} &= [3, 5] \\
L_{t_6} &= [1, 2, 3, 4, 5] \\
L_{t_7} &= [1, 2, 4] \\
L_{t_8} &= [2, 3, 5]
\end{align*}
\]
Huge research corpora describing different space/time trade-offs.

- Elias Gamma and Delta
- Variable-Byte Family
- Binary Interpolative Coding
- Simple Family
- PForDelta
- QMX
- Elias-Fano
- Partitioned Elias-Fano

‘70

2014
Many solutions

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

- Elias Gamma and Delta
- Variable-Byte Family
- Binary Interpolative Coding
- Simple Family
- PForDelta
- QMX
- Elias-Fano
- Partitioned Elias-Fano

Space

Binary Interpolative Coding

\(~3X\) smaller

Time

Variable-Byte Family

\(~4.5X\) faster
Many solutions

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

- Elias Gamma and Delta
- Variable-Byte Family
- Binary Interpolative Coding
- Simple Family
- PForDelta
- QMX
- Elias-Fano
- Partitioned Elias-Fano

Space

Binary Interpolative Coding

~3X smaller

Time

Variable-Byte Family

~4.5X faster

Spectrum
Key research questions

Space

Binary Interpolative Coding

〜3X smaller

Spectrum

Time

Variable-Byte Family

〜4.5X faster
Is it possible to design an encoding that is as small as BIC and much faster?
Key research questions

Space

Binary Interpolative Coding

~3X smaller

Time

Variable-Byte Family

~4.5X faster

1. Is it possible to design an encoding that is as small as BIC and much faster?

2. Is it possible to design an encoding that is as fast as VByte and much smaller?
Key research questions

1. Is it possible to design an encoding that is as small as BIC and much faster?

2. Is it possible to design an encoding that is as fast as VByte and much smaller?

3. What about both objectives at the same time?!
Idea 1 - Clustered inverted indexes (TOIS ’17)

Every encoder represents each sequence *individually*. No exploitation of redundancy.
Idea 1 - Clustered inverted indexes (TOIS ’17)

Every encoder represents each sequence **individually**. No exploitation of redundancy.
Idea 1 - Clustered inverted indexes (TOIS ’17)

Every encoder represents each sequence *individually*. No exploitation of redundancy.

Encode **clusters** of inverted lists.
Idea 1 - Clustered inverted indexes (TOIS ’17)

Every encoder represents each sequence **individually**.
No exploitation of redundancy.

Encode **clusters** of inverted lists.

**Space**
- Always better than PEF (by up to 11%) and better than BIC (by up to 6.25%)

**Time**
- Much faster than BIC (~103%)
- Slightly slower than PEF (~20%)
The majority of values are small (very small indeed).

VByte needs at least 8 bits per integer, that is sensibly far away from bit-level effectiveness (BIC: 3.54, PEF: 4.1 on Gov2).
The majority of values are small (very small indeed).

VByte needs at least 8 bits per integer, that is sensibly far away from bit-level effectiveness (BIC: 3.54, PEF: 4.1 on Gov2).
The majority of values are small (very small indeed).

VByte needs at least 8 bits per integer, that is sensibly far away from bit-level effectiveness (BIC: 3.54, PEF: 4.1 on Gov2).

Encode dense regions with unary codes, sparse regions with VByte.
Idea 2 - Optimally-partitioned VByte (TKDE ’18)

The majority of values are small (very small indeed).

VByte needs at least 8 bits per integer, that is sensibly far away from bit-level effectiveness (BIC: 3.54, PEF: 4.1 on Gov2).

Encode dense regions with unary codes, sparse regions with VByte.

Optimal partitioning in linear time and constant space.
The majority of values are **small** (*very* small indeed).

VByte needs **at least 8 bits** per integer, that is sensibly far away from bit-level effectiveness (BIC: 3.54, PEF: 4.1 on Gov2).

Encode **dense** regions with unary codes, **sparse** regions with VByte.

Optimal partitioning in linear time and constant space. Compression ratio improves by **2X**.
The majority of values are **small** (very small indeed).

VByte needs **at least 8 bits** per integer, that is sensibly far away from bit-level effectiveness (BIC: **3.54**, PEF: **4.1** on Gov2).

Optimal partitioning in linear time and constant space.

Compression ratio improves by **2X**.

Query processing speed and sequential decoding **not affected**.

Encode **dense** regions with unary codes, **sparse** regions with VByte.
If we consider subsequences of $d$-gaps in inverted lists, these are *repetitive* across the whole inverted index.
If we consider subsequences of $d$-gaps in inverted lists, these are \textbf{repetitive} across the whole inverted index.

Put the \textbf{top-$k$ frequent patterns} in a dictionary of size $k$. Then encode inverted lists as sequences of log $k$-bit codewords.
If we consider subsequences of $d$-gaps in inverted lists, these are **repetitive** across the whole inverted index.

Put the **top-k frequent patterns** in a dictionary of size $k$. Then encode inverted lists as sequences of log $k$-bit codewords.

Close to the most space-efficient representation (~$7\%$ away from BIC).
If we consider subsequences of $d$-gaps in inverted lists, these are **repetitive** across the whole inverted index.

Put the **top-k frequent patterns** in a dictionary of size $k$. Then encode inverted lists as sequences of log $k$-bit codewords.

Close to the most space-efficient representation (~7% away from BIC).

Almost **as fast as** the fastest SIMD-ized decoders.
The bigger picture

![Graph showing performance metrics for different storage formats across varying space and time dimensions.]

- Time [ns/int] vs. Space [GB]
- Time [ms/query] vs. Space [GB]

Legend:
- BIC
- CPEF
- PEF
- Opt-PFOR
- Simple16
- QMX
- VByte
- Varint-GB
- Varint-G8IU
- Masked-VByte
- Stream-VByte
- Opt-VByte
- DINT Time-Opt
- DINT Space-Opt
The bigger picture
Problem 2

Integer data structures

- van Emde Boas Trees
- X/Y-Fast Tries
- Fusion Trees
- Exponential Search Trees
- ...

Elias-Fano encoding

- $\text{EF}(S(n,u)) = n \log(u/n) + 2n$ bits to encode a sorted integer sequence $S$
- $O(1)$ Access
- $O(1 + \log(u/n))$ Predecessor

+ time
- space
+ dynamic

+ time
+ space
- static
Problem 2

**Integer data structures**
- van Emde Boas Trees
- X/Y-Fast Tries
- Fusion Trees
- Exponential Search Trees
- ...

**Elias-Fano encoding**
- \( \text{EF}(S(n,u)) = n \log(u/n) + 2n \) bits to encode a sorted integer sequence \( S \)
- \( O(1) \) Access
- \( O(1 + \log(u/n)) \) Predecessor

+ time
- space
+ dynamic
+ time
- static

Can we grab the best from both?
Dynamic inverted indexes

Classic solution: use two indexes. One is big and **cold**; the other is small and **hot**. **Merge** them periodically.

**Append-only** inverted indexes.
For $u = n^\gamma$, $\gamma = \Theta(1)$:

- $\text{EF}(S(n,u)) + o(n)$ bits
- $O(1)$ Access
- $O(\min\{1+\log(u/n), \log \log n\})$ Predecessor

- $\text{EF}(S(n,u)) + o(n)$ bits
- $O(1)$ Access
- $O(1)$ Append (amortized)
- $O(\min\{1+\log(u/n), \log \log n\})$ Predecessor

- $\text{EF}(S(n,u)) + o(n)$ bits
- $O(\log n / \log \log n)$ Access
- $O(\log n / \log \log n)$ Insert/Delete (amortized)
- $O(\min\{1+\log(u/n), \log \log n\})$ Predecessor
### Integer dictionaries in succinct space (CPM ’17)

For $u = n^\gamma$, $\gamma = \Theta(1)$:

<table>
<thead>
<tr>
<th>Result 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>• $\text{EF}(S(n,u)) + o(n)$ bits</td>
</tr>
<tr>
<td>• $O(1)$ Access</td>
</tr>
<tr>
<td>• $O(\min{1+\log(u/n), \log\log n})$ Predecessor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• $\text{EF}(S(n,u)) + o(n)$ bits</td>
</tr>
<tr>
<td>• $O(1)$ Access</td>
</tr>
<tr>
<td>• $O(1)$ Append (amortized)</td>
</tr>
<tr>
<td>• $O(\min{1+\log(u/n), \log\log n})$ Predecessor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>• $\text{EF}(S(n,u)) + o(n)$ bits</td>
</tr>
<tr>
<td>• $O(\log n / \log\log n)$ Access</td>
</tr>
<tr>
<td>• $O(\log n / \log\log n)$ Insert/Delete (amortized)</td>
</tr>
<tr>
<td>• $O(\min{1+\log(u/n), \log\log n})$ Predecessor</td>
</tr>
</tbody>
</table>

Optimal time bounds for all operations using a sublunar redundancy.
Problem 3

Consider a large text.
Problem 3

Consider a large text.

How to represent all its substrings of size $1 \leq k \leq N$ words for fixed $N$ (e.g., $N = 5$), using as few as possible bits?

Fast Access to individual N-grams?

How to estimate the probability of occurrence of the patterns under a given probability model?
Consider a large text.

How to represent all its substrings of size $1 \leq k \leq N$ words for fixed $N$ (e.g., $N = 5$), using as few as possible bits?

How to estimate the probability of occurrence of the patterns under a given probability model?

Fast Access to individual N-grams?

This problem is central to applications in IR, ML, NLP, WSE.
Next word prediction.
Applications

Next word prediction.

space and time-efficient

context

?
Applications

Next word prediction.

space and time-efficient
context

?  

algorithms
foo
bar
baz

data structures

frequency count
1214
2
3647
3
1
Applications

Next word prediction.

\[ P(\text{“data structures”} \mid \text{“space and time-efficient”}) \approx \frac{f(\text{“space and time-efficient data structures”})}{f(\text{“space and time-efficient”})} \]

frequency count

<table>
<thead>
<tr>
<th></th>
<th>1214</th>
<th>2</th>
<th>3647</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithms</td>
<td>foo</td>
<td>data structures</td>
<td>bar</td>
<td>baz</td>
<td></td>
</tr>
</tbody>
</table>

space and time-efficient

context
What can I help you with?
Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of one trillion words from public Web pages.
Applications

Google Research Blog

The latest news from Research at Google

All Our N-gram are Belong to You
Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of one trillion words from public Web pages.
Google Books

~6% of the books ever published

<table>
<thead>
<tr>
<th>n</th>
<th>number of n-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24,359,473</td>
</tr>
<tr>
<td>2</td>
<td>667,284,771</td>
</tr>
<tr>
<td>3</td>
<td>7,397,041,901</td>
</tr>
<tr>
<td>4</td>
<td>1,644,807,896</td>
</tr>
<tr>
<td>5</td>
<td>1,415,355,596</td>
</tr>
</tbody>
</table>

More than 11 billion $n$-grams.
The number of words following a given context is small.
The number of words following a given context is small.

$k = 1$

Map a word ID to the position it takes within its sibling IDs (the IDs following a context of fixed length $k$).
The number of words following a given context is small.

Map a word ID to the position it takes within its sibling IDs (the IDs following a context of fixed length $k$).
The number of words following a given context is **small**.

Map a word ID to the position it takes within its *siblings* IDs (the IDs following a context of fixed length $k$).

$k = 1$
Idea 1 - Context-based remapped tries (SIGIR ’17)

The number of words following a given context is small.

Map a word ID to the position it takes within its sibling IDs (the IDs following a context of fixed length $k$).
The number of words following a given context is **small**.

Map a word ID to the **position** it takes within its *sibling* IDs (the IDs following a context of fixed length $k$).

The (Elias-Fano) context-based remapped trie is **as fast as** the fastest competitor, but up to **65% smaller**.
The number of words following a given context is **small**.

$k = 1$

Map a word ID to the **position** it takes within its *sibling* IDs (the IDs following a context of fixed length $k$).

The (Elias-Fano) context-based remapped trie is **even smaller** than the most space-efficient competitors, that are lossy and with false-positives allowed, and up to **5X faster**.

The (Elias-Fano) context-based remapped trie is **as fast as** the fastest competitor, but up to **65% smaller**.
To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.
To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

**Computing the distinct left extensions.**

<table>
<thead>
<tr>
<th>Suffix order</th>
<th>Context order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A A A A B B C C X X X X</td>
<td>C A C B A A B X X X X X X A</td>
</tr>
<tr>
<td>A B B X A C A A C X X X</td>
<td>A A A A B B C C X X X X</td>
</tr>
<tr>
<td>B A C X X A A B A C X X</td>
<td>A B B X A C A A C X X X</td>
</tr>
<tr>
<td>A X A X X A B C B A C X</td>
<td>B A C X X A A B A C X X</td>
</tr>
<tr>
<td>X X A X X B A A C B A C</td>
<td>A X A X X A B C B A C X</td>
</tr>
</tbody>
</table>

U -> S -> C
To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

**Computing the distinct left extensions.**

<table>
<thead>
<tr>
<th>Suffix order</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A A A A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>X</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context order</th>
<th>7</th>
<th>1</th>
<th>8</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>X</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
To compute the modified Kneser-Ney probabilities of the \( n \)-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

**Computing the distinct left extensions.**

**Suffix order**

**Context order**
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

Computing the distinct left extensions.

**Suffix order**

**Context order**
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Computing the distinct left extensions.

Suffix order

Context order
To compute the modified Kneser-Ney probabilities of the $n$-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

### Computing the distinct left extensions.

<table>
<thead>
<tr>
<th>Suffix order</th>
<th>Context order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A A A A B B C C X X X X</td>
<td>C A C B A A A B B C C X X X X</td>
</tr>
<tr>
<td>A B B X A C A A C X X X</td>
<td>A A A A B B C C X X X X</td>
</tr>
<tr>
<td>B A C X X A A B A C X X</td>
<td>B A C X X A A B A C X X</td>
</tr>
<tr>
<td>A X A X A B C B A C X</td>
<td>A X A X A B C B A C X</td>
</tr>
<tr>
<td>X X A X X B A A C B A C</td>
<td>X X A X X A B C B A C</td>
</tr>
</tbody>
</table>

*Using a scan of the block and $O(|V|)$ space.*
To compute the modified Kneser-Ney probabilities of the \( n \)-grams, the fastest algorithm in the literature uses **3 sorting steps** in external memory.

**Computing the distinct left extensions.**

**Suffix order**

**Context order**

Using a scan of the block and \( O(|V|) \) space.
Idea 2 - Fast estimation in external memory (TOIS '18)

To compute the modified Kneser-Ney $n$-grams probabilities, the fastest algorithm in the literature uses 3 sorting steps in external memory.

**Computing the distinct left extensions.**

Using a scan of the block and $O(|V|)$ space.

**Rebuilding the last level of the trie.**
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney $n$-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Computing the distinct left extensions.

Using a scan of the block and $O(|V|)$ space.

Rebuilding the last level of the trie.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>X</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

Rebuilding the last level of the trie.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney probabilities of the n-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Rebuilding the last level of the trie.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Idea 2 - Fast estimation in external memory (TOIS '18)

To compute the modified Kneser-Ney \( n \)-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Rebuilding the last level of the trie.
Idea 2 - Fast estimation in external memory (TOIS '18)

To compute the modified Kneser-Ney \( n \)-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Rebuilding the last level of the trie.

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

A 4
B 2
C 2
X 4
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney \( n \)-grams, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Rebuilding the last level of the trie.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
1 & 2 & 3 & 4 & 5 & 6 \\
\hline
7 & 8 & 9 & 10 & 11 & 12 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
10 & 11 & 12 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
A & 4 & B & 2 & C & 2 & X & 4 \\
\hline
\end{array}
\]
Idea 2 - Fast estimation in external memory (TOIS ’18)

To compute the modified Kneser-Ney $n$-gram probabilities, the fastest algorithm in the literature uses 3 sorting steps in external memory.

Computing the distinct left extensions.

Using a scan of the block and $O(|V|)$ space.

Rebuilding the last level of the trie.

Estimation runs 4.5X faster with billions of strings.
Thanks for your attention, time, patience!

Any questions?