On Optimally Partitioning Variable-Byte Index Data

Giulio Ermanno Pibiri
University of Pisa and ISTI-CNR
Pisa, Italy
giulio.pibiri@di.unipi.it

Rossano Venturini
University of Pisa and ISTI-CNR
Pisa, Italy
rossano.venturini@unipi.it

RMIT
UNIVERSITY

Melbourne, 17/05/2018

Context - Inverted Indexes

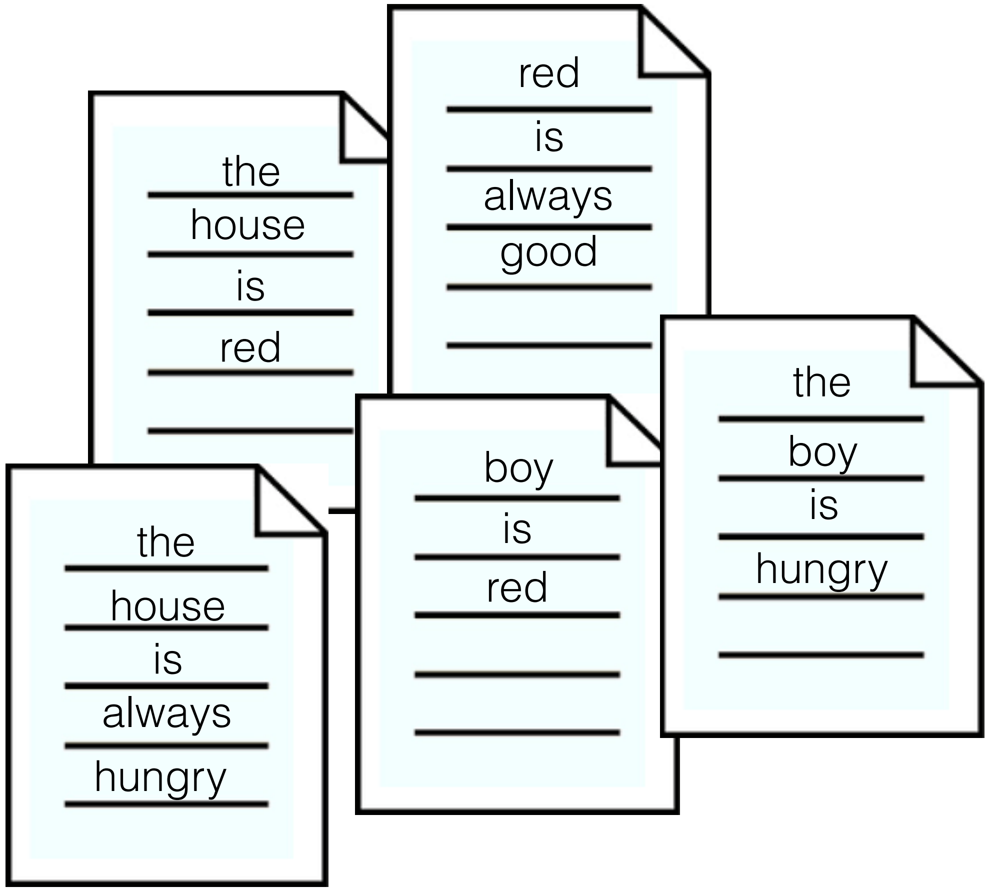
We focus on compression effectiveness and retrieval speed in **inverted indexes**.

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

Context - Inverted Indexes

We focus on compression effectiveness and retrieval speed in **inverted indexes**.

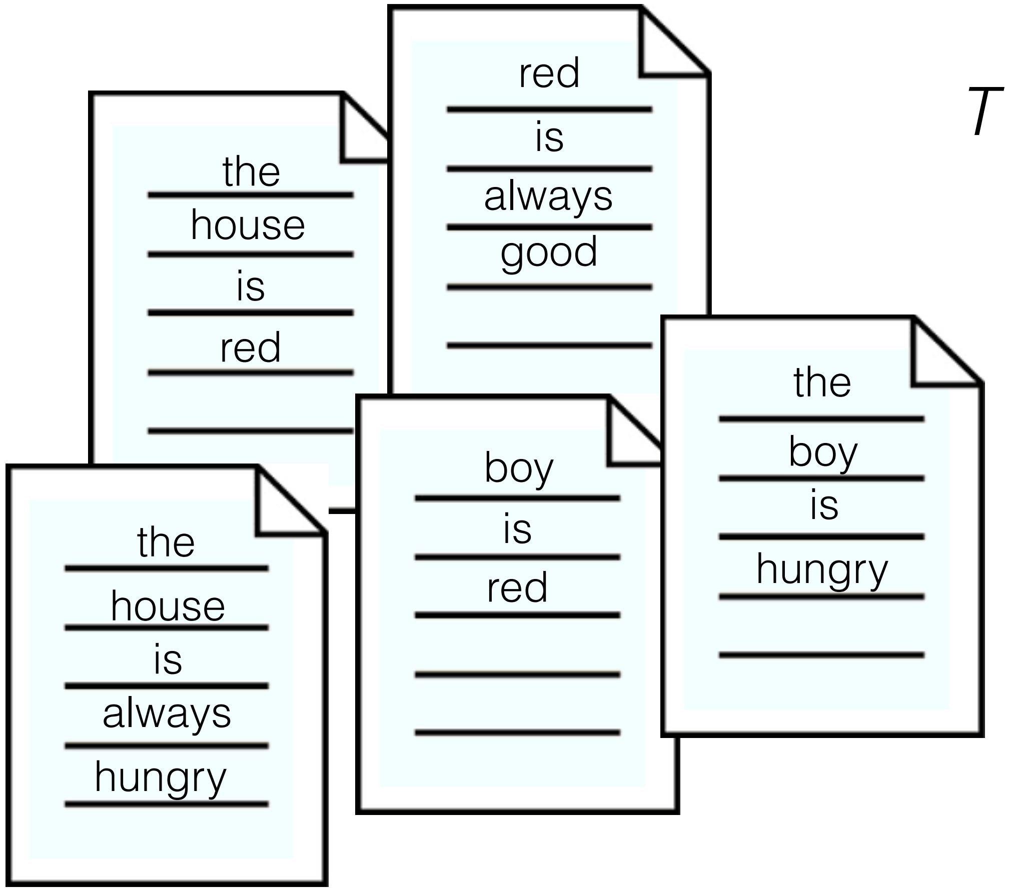
The inverted index is the *de-facto* data structure at the basis of every large-scale retrieval system.



Context - Inverted Indexes

We focus on compression effectiveness and retrieval speed in **inverted indexes**.

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

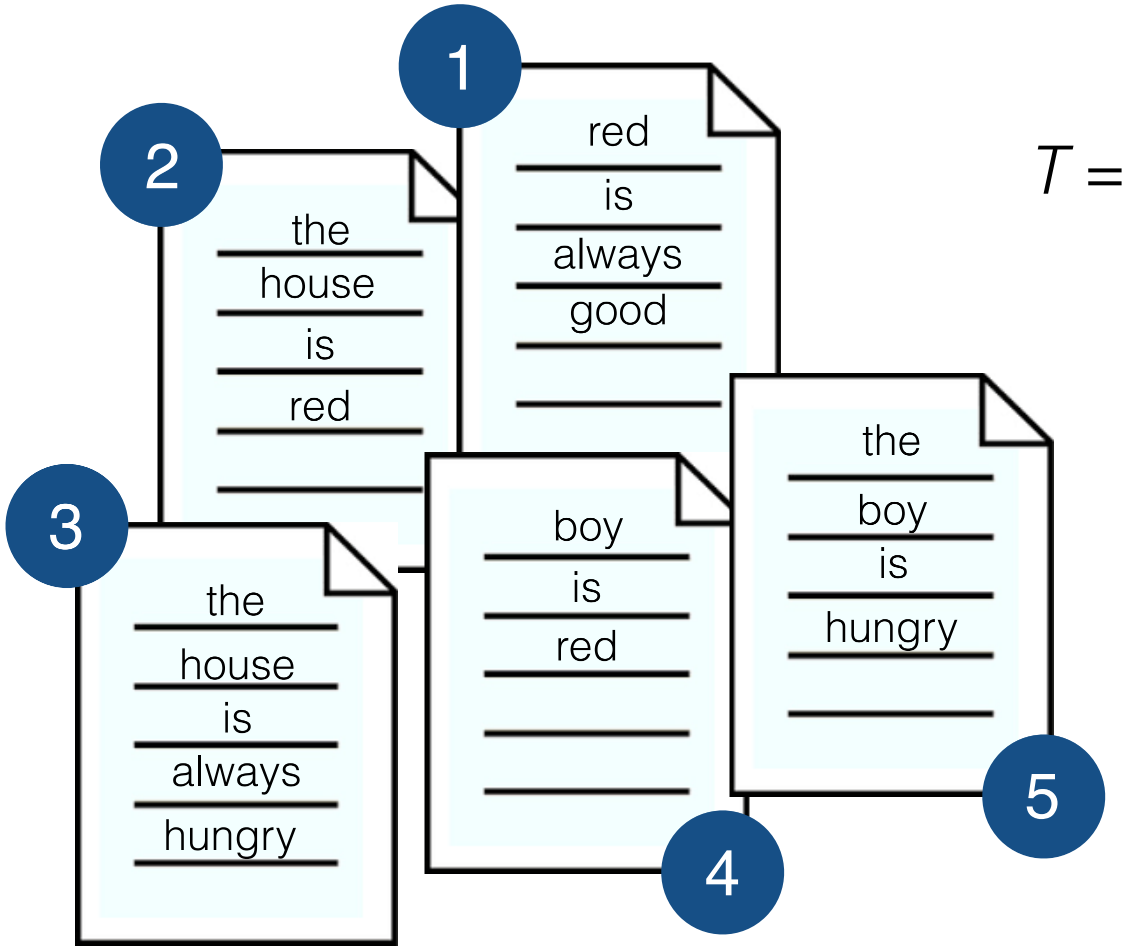


$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

Context - Inverted Indexes

We focus on compression effectiveness and retrieval speed in **inverted indexes**.

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

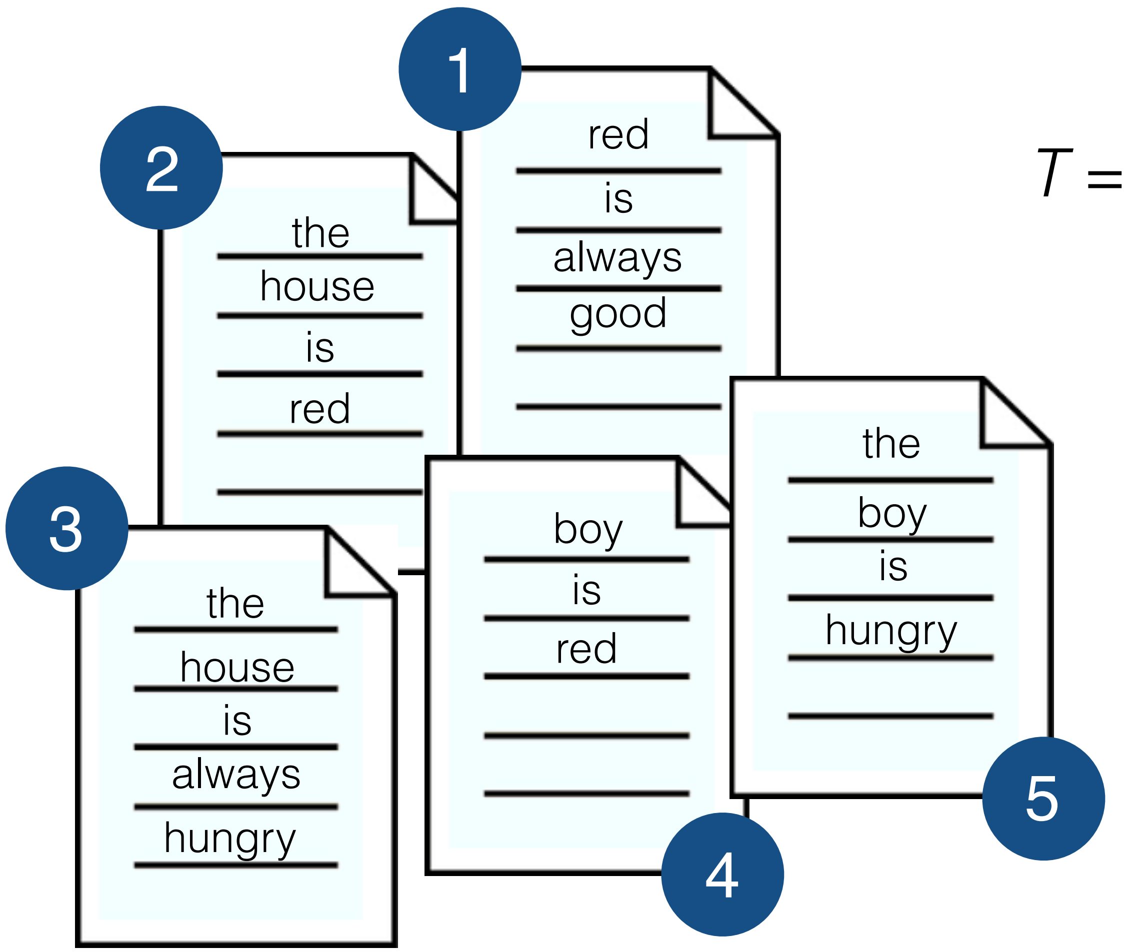


$$\begin{array}{cccccccccccccccc} & & t_1 & & t_2 & & t_3 & & t_4 & & t_5 & & t_6 & t_7 & t_8 & \\ T = & \{ & \text{always}, & & \text{boy}, & & \text{good}, & & \text{house}, & & \text{hungry}, & & \text{is}, & \text{red}, & \text{the} & \} \end{array}$$

Context - Inverted Indexes

We focus on compression effectiveness and retrieval speed in **inverted indexes**.

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



$$\begin{array}{cccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 \end{array}$$

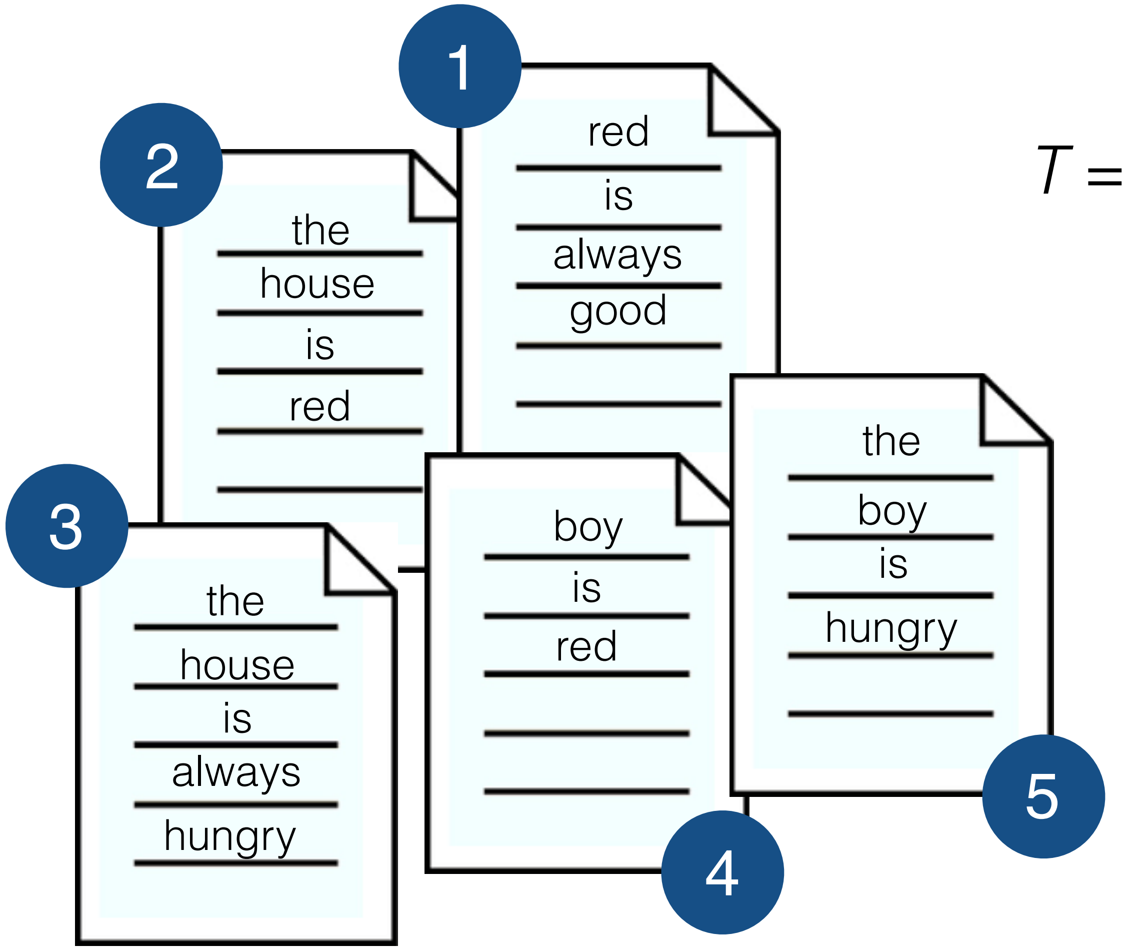
T = {always, boy, good, house, hungry, is, red, the}

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

Context - Inverted Indexes

We focus on compression effectiveness and retrieval speed in **inverted indexes**.

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



$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

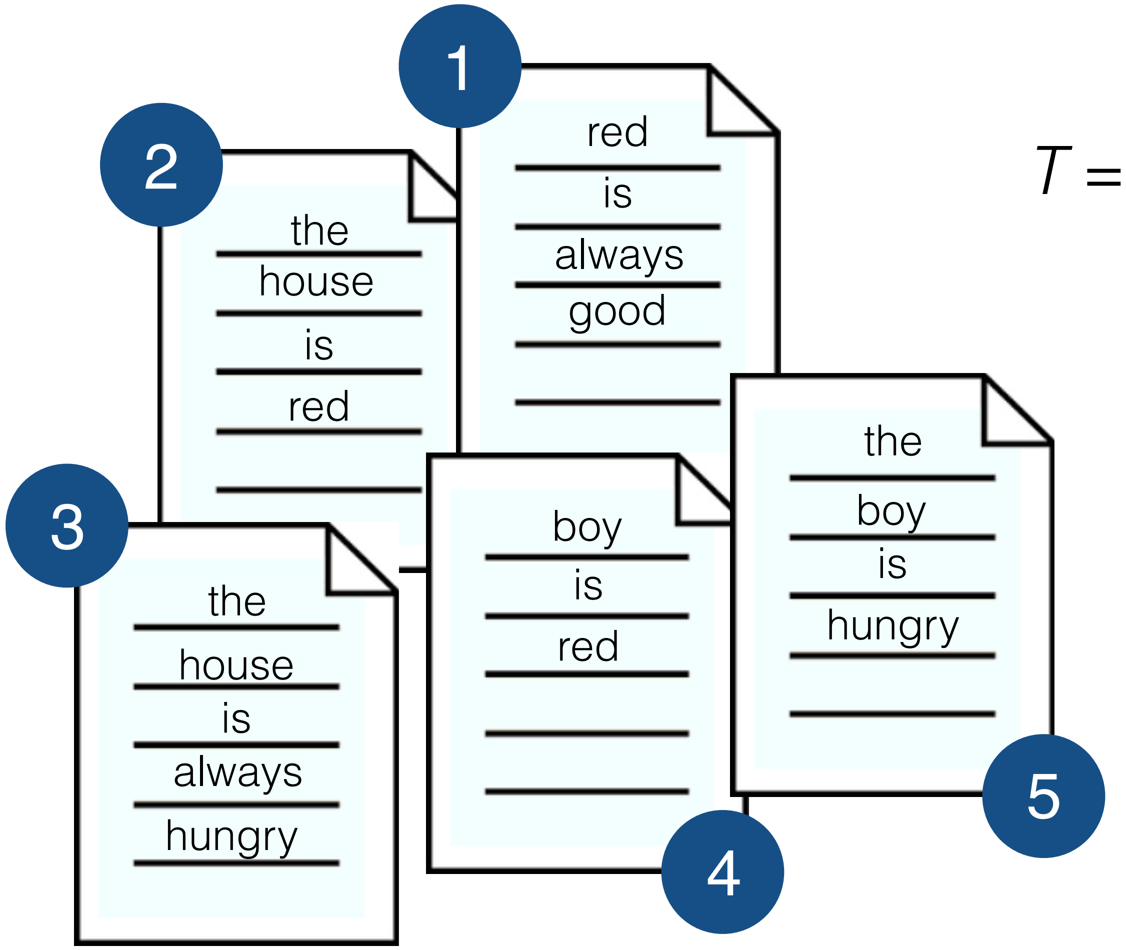
L_{t_1}=[1, 3]
L_{t_2}=[4, 5]
L_{t_3}=[1]
L_{t_4}=[2, 3]
L_{t_5}=[3, 5]
L_{t_6}=[1, 2, 3, 4, 5]
L_{t_7}=[1, 2, 4]
L_{t_8}=[2, 3, 5]

Context - Inverted Indexes

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”.

Context - Inverted Indexes

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”.

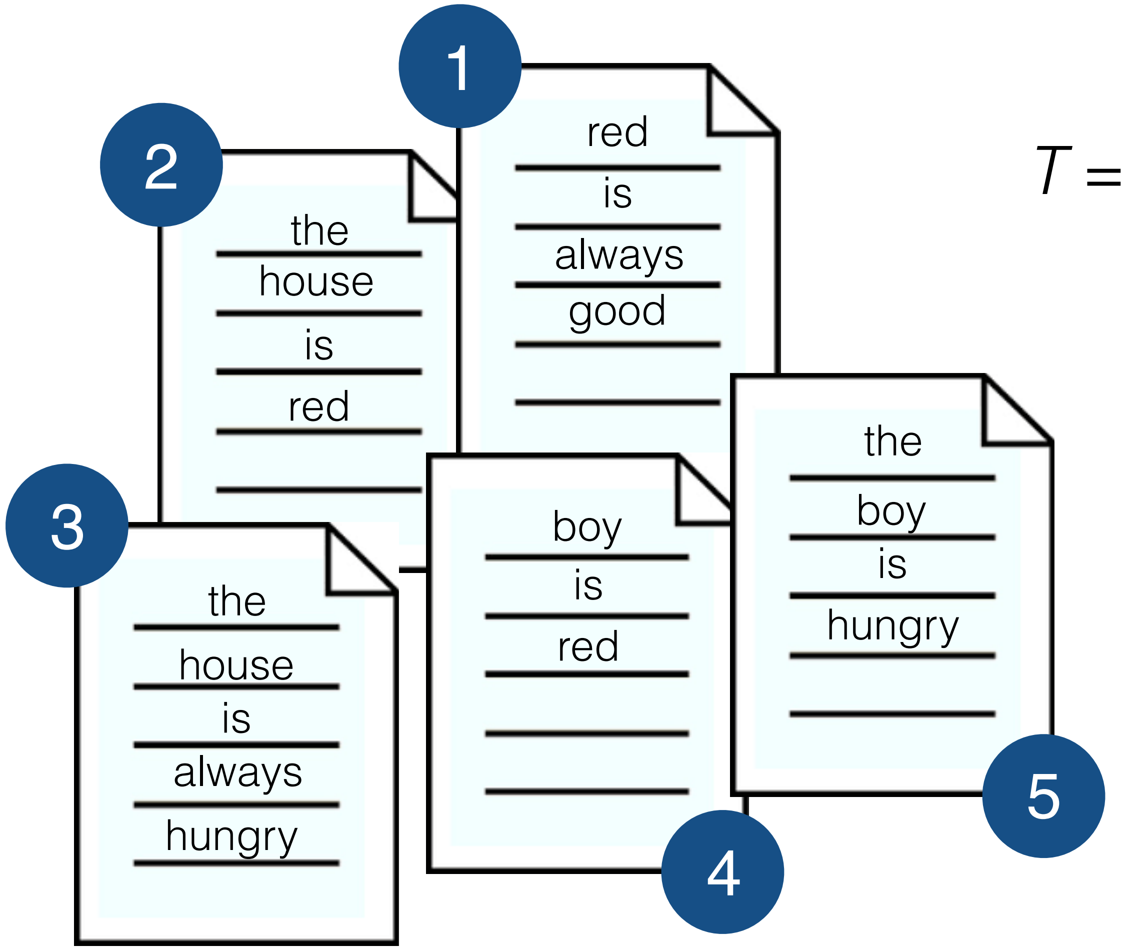


$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

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

Context - Inverted Indexes

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”.



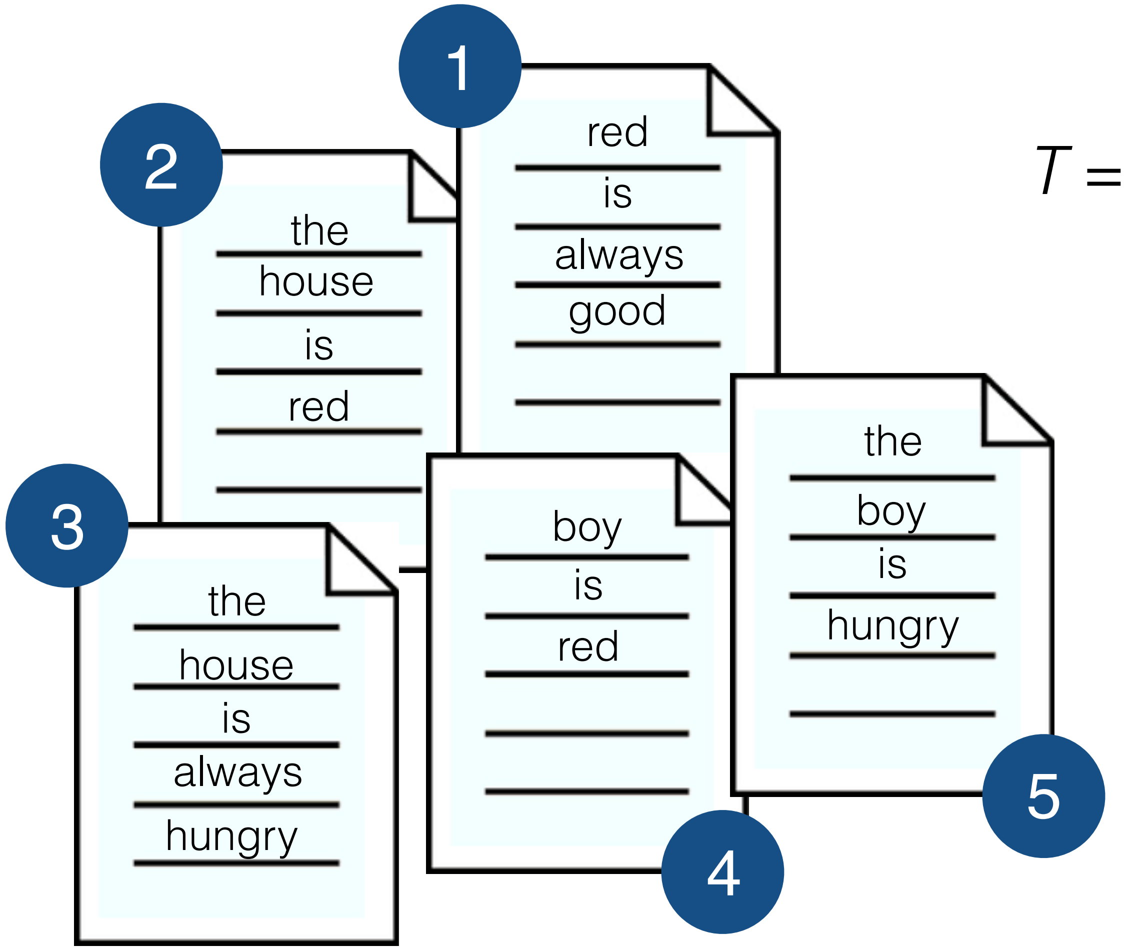
$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

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

$q = \{\text{boy, is, the}\}$

Context - Inverted Indexes

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”.



$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

$L_{t_1}=[1, 3]$

$\boxed{L_{t_2}=[4, 5]}$

$L_{t_3}=[1]$

$L_{t_4}=[2, 3]$

$L_{t_5}=[3, 5]$

$\boxed{L_{t_6}=[1, 2, 3, 4, 5]}$

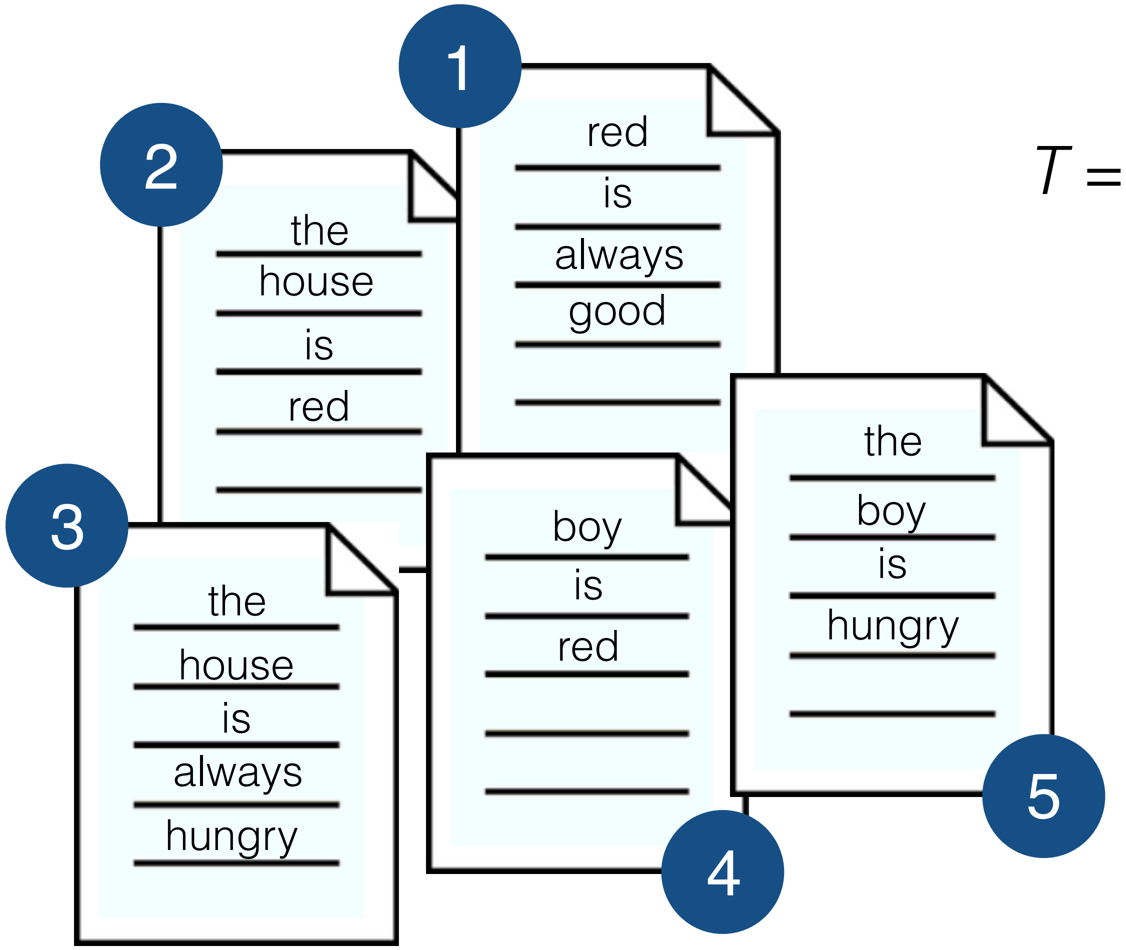
$L_{t_7}=[1, 2, 4]$

$\boxed{L_{t_8}=[2, 3, 5]}$

$q = \{\text{boy}, \text{is}, \text{the}\}$

Context - Inverted Indexes

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”.



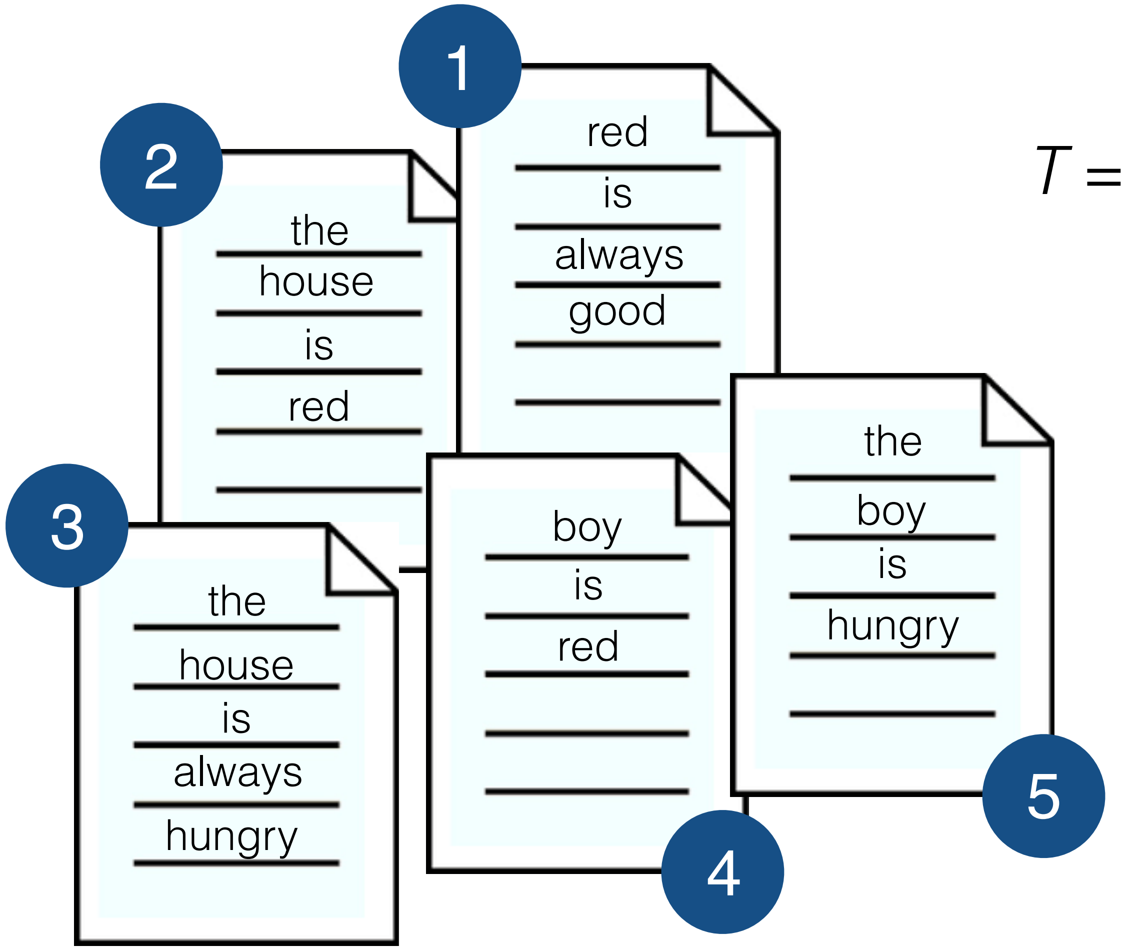
$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

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

$q = \{\text{boy}, \text{is}, \text{the}\}$

Context - Inverted Indexes

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”.



$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

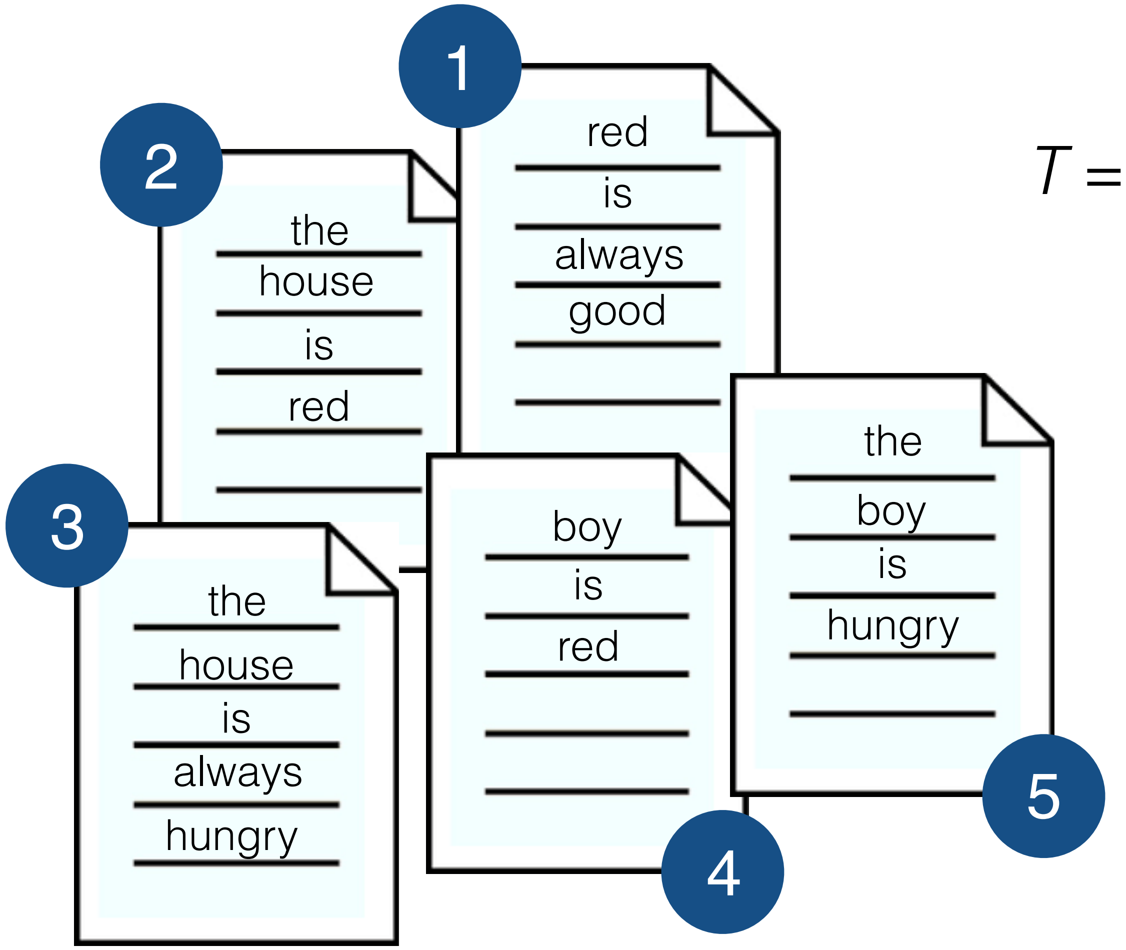
L_{t_1}=[1, 3]
L_{t_2}=[4, 5]
L_{t_3}=[1]
L_{t_4}=[2, 3]
L_{t_5}=[3, 5]
L_{t_6}=[1, 2, 3, 4, 5]
L_{t_7}=[1, 2, 4]
L_{t_8}=[2, 3, 5]

q = {boy, is, the}

q = {good, hungry}

Context - Inverted Indexes

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”.



$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$

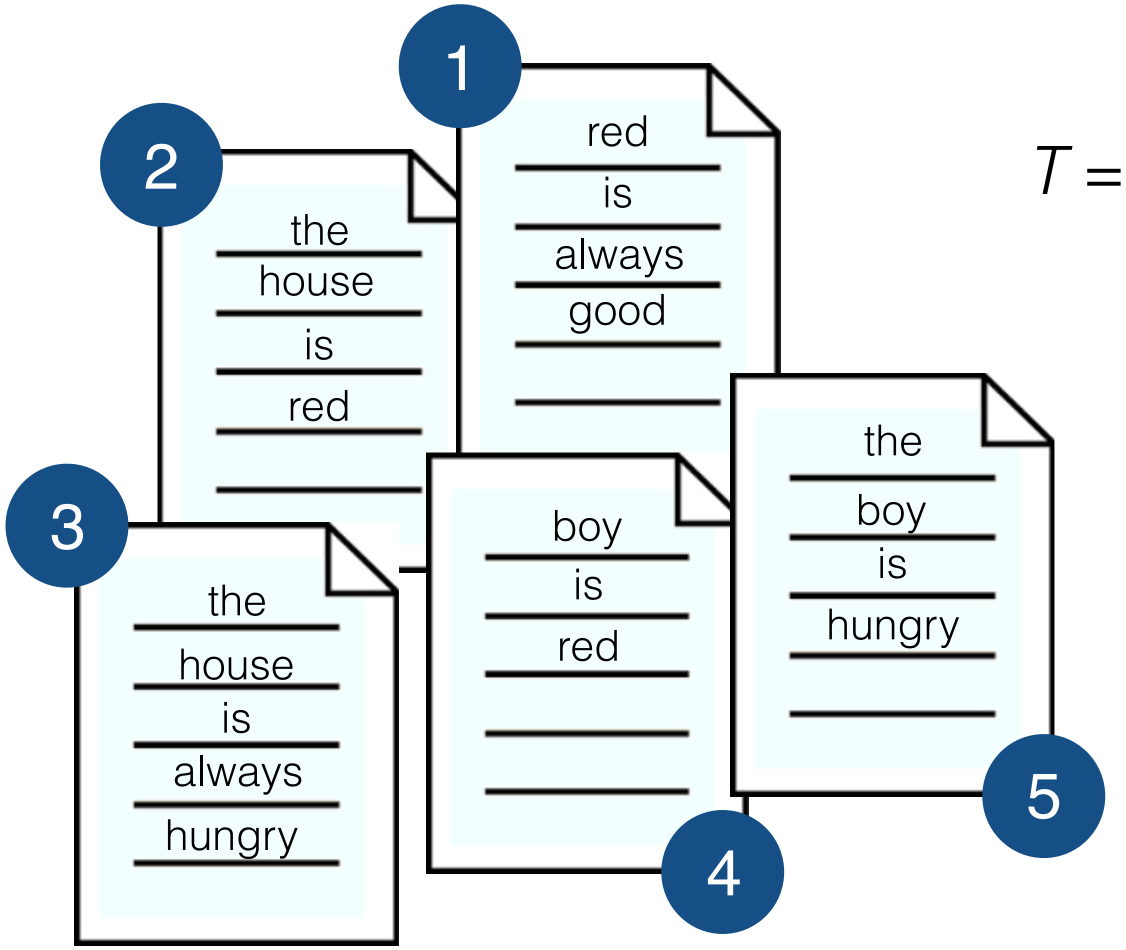
$L_{t_1}=[1, 3]$
$L_{t_2}=[4, 5]$
$\boxed{L_{t_3}=[1]}$
$L_{t_4}=[2, 3]$
$\boxed{L_{t_5}=[3, 5]}$
$L_{t_6}=[1, 2, 3, 4, 5]$
$L_{t_7}=[1, 2, 4]$
$L_{t_8}=[2, 3, 5]$

q = {boy, is, the}

q = {good, hungry}

Context - Inverted Indexes

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”.



$$T = \{\overset{t_1}{\text{always}}, \overset{t_2}{\text{boy}}, \overset{t_3}{\text{good}}, \overset{t_4}{\text{house}}, \overset{t_5}{\text{hungry}}, \overset{t_6}{\text{is}}, \overset{t_7}{\text{red}}, \overset{t_8}{\text{the}}\}$$

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

$q = \{\text{boy, is, the}\}$

$q = \{\text{good, hungry}\}$

Many solutions

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

- Elias gamma/delta
- Variable-Byte
- Binary Interpolative Coding
- Simple-9/16
- PForDelta

- Optimized PForDelta
- Elias-Fano
- Partitioned Elias-Fano
- Clustered Elias-Fano
- Asymmetric Numeral Systems

Many solutions

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

- Elias gamma/delta
- Variable-Byte
- Binary Interpolative Coding
- Simple-9/16
- PForDelta
- Optimized PForDelta
- Elias-Fano
- Partitioned Elias-Fano
- Clustered Elias-Fano
- Asymmetric Numeral Systems

Space

Binary Interpolative Coding

~**3X** smaller

← Spectrum →

Time

Variable-Byte (VByte) Family

~**4.5X** faster

Our research question

Can we improve the space of a VByte-encoded sequence

and

preserve its query processing speed?

Variable-Byte Encoding

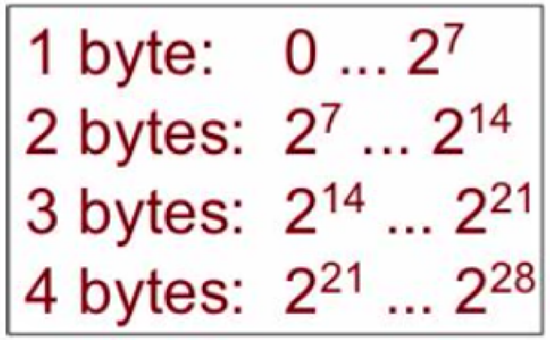
Simple idea: encode each number
using as few bytes as possible.

6 → **1**0000110

127 → **1**1111111

128 → **1**0000001**0**0000000

65790 → **1**0000100**1**0000000**0**1111110



1 byte: 0 ... 2^7
2 bytes: 2^7 ... 2^{14}
3 bytes: 2^{14} ... 2^{21}
4 bytes: 2^{21} ... 2^{28}

Variable-Byte Encoding

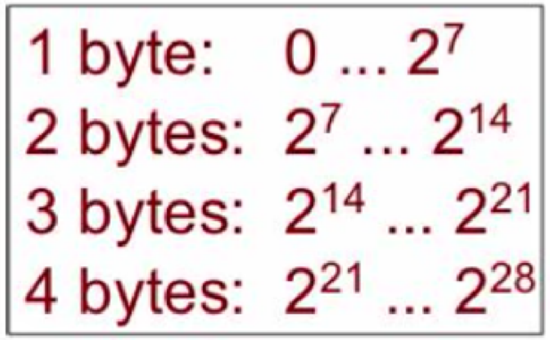
Simple idea: encode each number
using as few bytes as possible.

6 → **1**0000110

127 → **1**1111111

128 → **1**0000001**0**0000000

65790 → **1**0000100**1**0000001**0**1111110

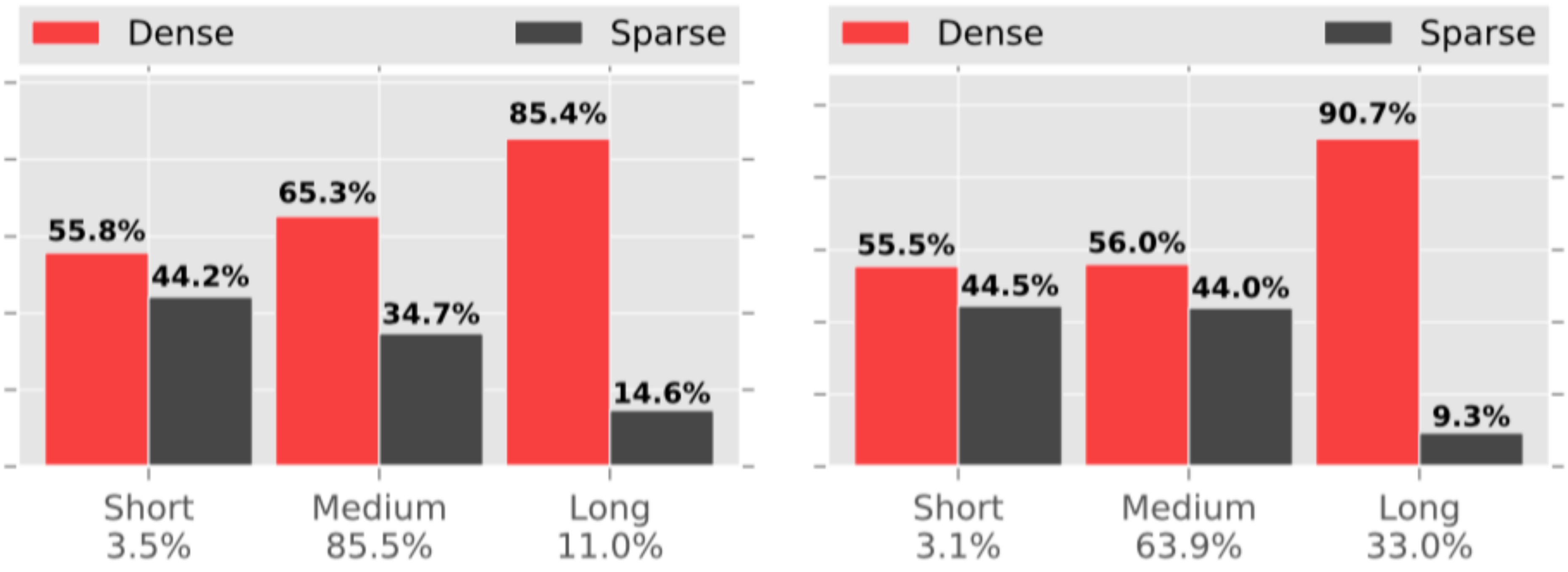


1 byte: $0 \ldots 2^{7}$
2 bytes: $2^{7} \ldots 2^{14}$
3 bytes: $2^{14} \ldots 2^{21}$
4 bytes: $2^{21} \ldots 2^{28}$

Decoding is **fast**:
keep reading bytes until you hit a value smaller than 128.

SIMD (Single Instruction Multiple Data)

So…what's "wrong" with VByte?

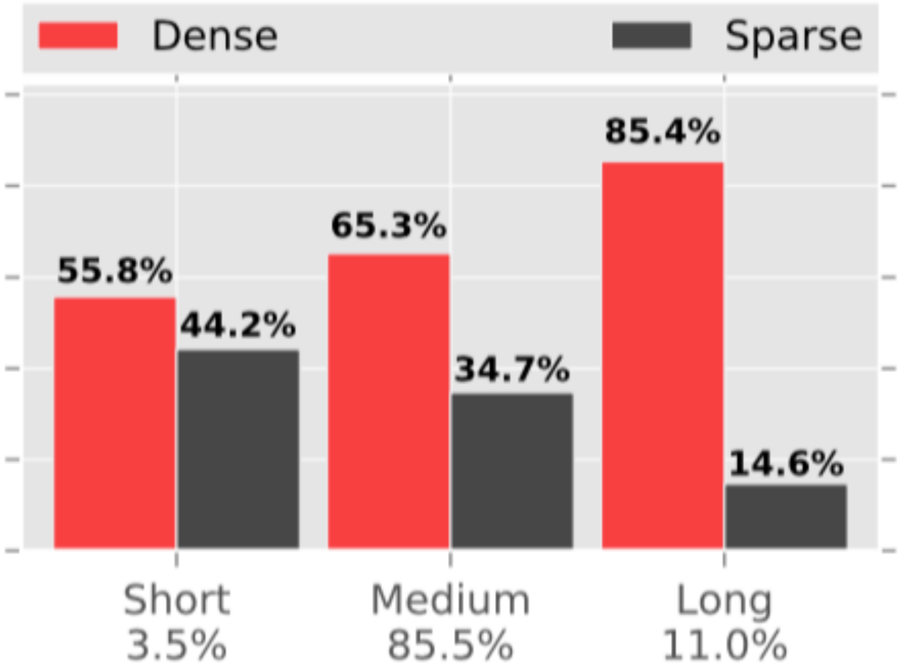


(a) Gov2 (b) ClueWeb09

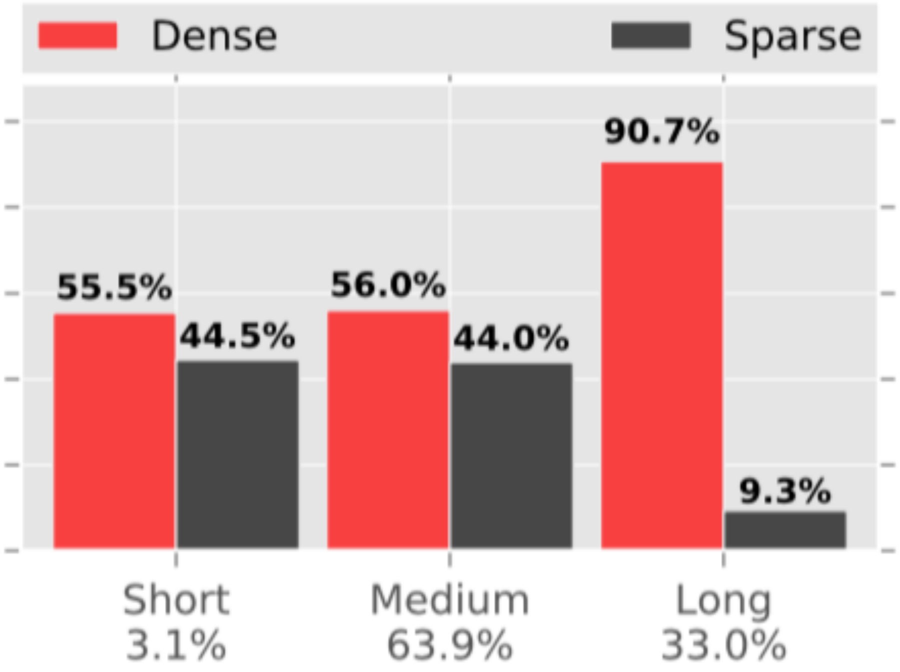
The majority of values are **small** (*very* small indeed).

VByte needs **at least 8 bits** per integer (bpi).

So…what's "wrong" with VByte?



(a) Gov2



(b) ClueWeb09

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

VByte needs **at least 8 bits** per integer (bpi).

Sensibly far away from bit-level effectiveness.
BIC: **3.8** bpi on Gov2
PEF: **4.1** bpi on Gov2

High-level idea

1. Partition each inverted list into variable-length partitions.
2. Encode *dense* partitions with their **characteristic bitvector**.
3. Encode *sparse* partitions with **VByte**.

High-level idea

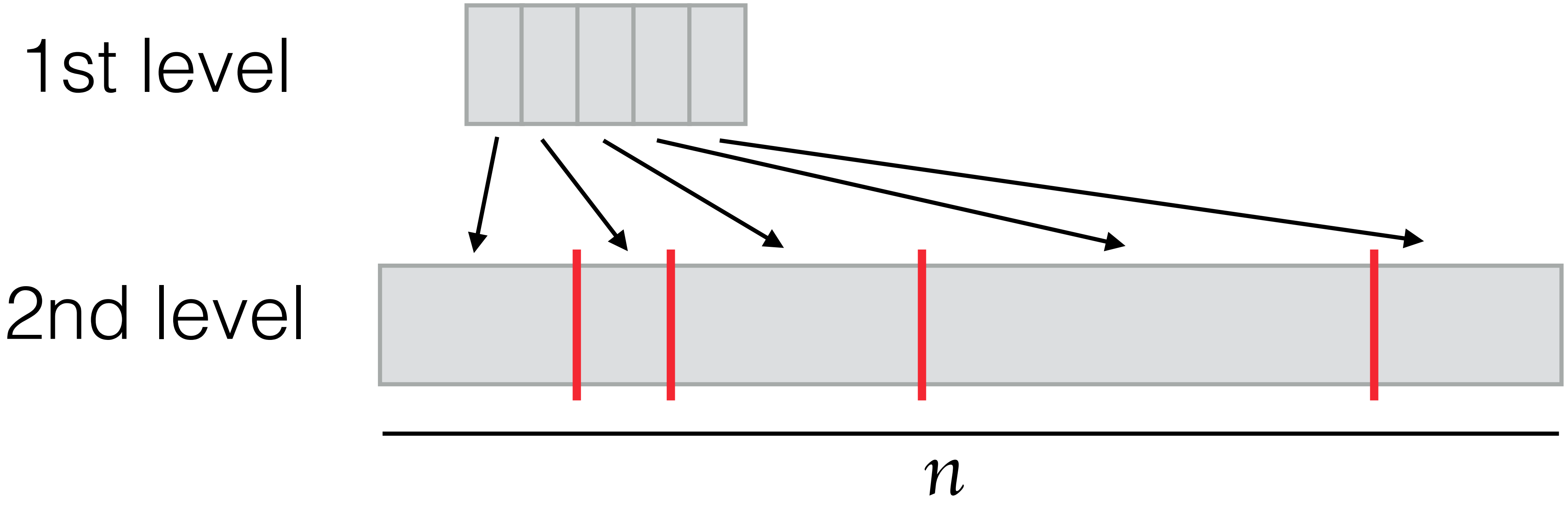
1. Partition each inverted list into variable-length partitions.
2. Encode *dense* partitions with their **characteristic bitvector**.
3. Encode *sparse* partitions with **VByte**.

[13, 15, 16, 17, 20, 21, 23, 24]

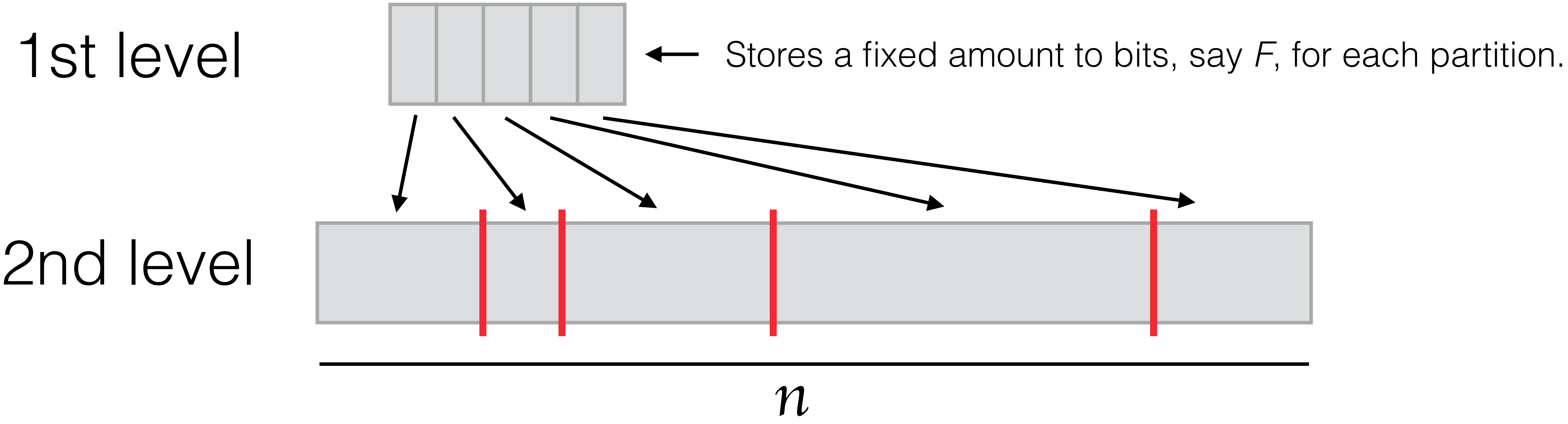
1	0	**1**	**1**	**1**	0	0	**1**	**1**	0	**1**	**1**
13	14	15	16	17	18	19	20	21	22	23	24

24 - 13 - 1 = 12 bits VS 64 bits (**5.33X**)

Computing an optimal partition

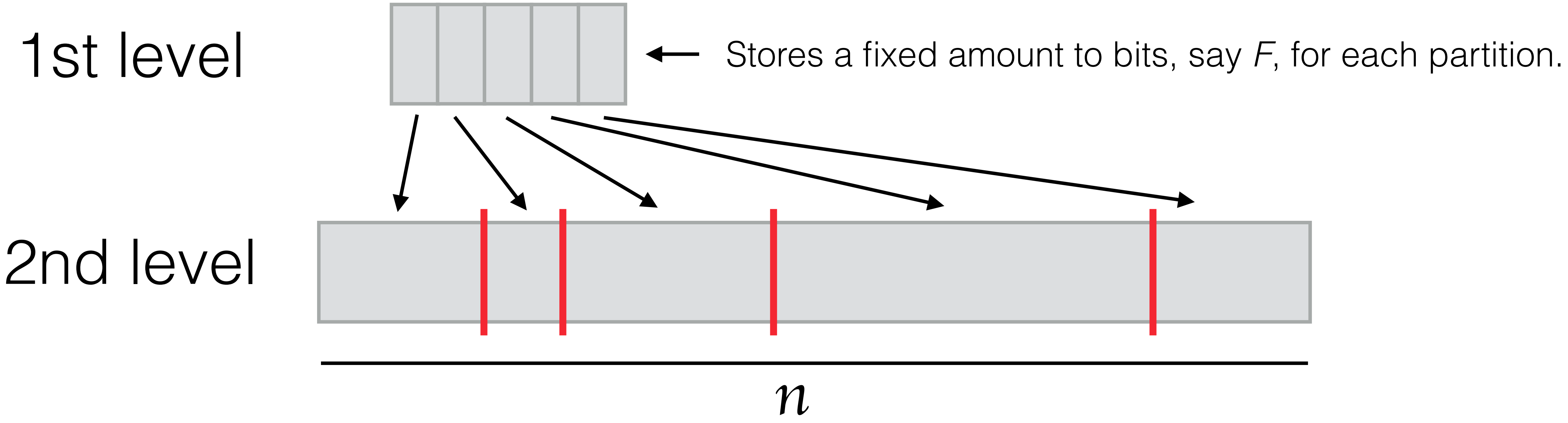


Computing an optimal partition



Stores a fixed amount to bits, say *F*, for each partition.

Computing an optimal partition

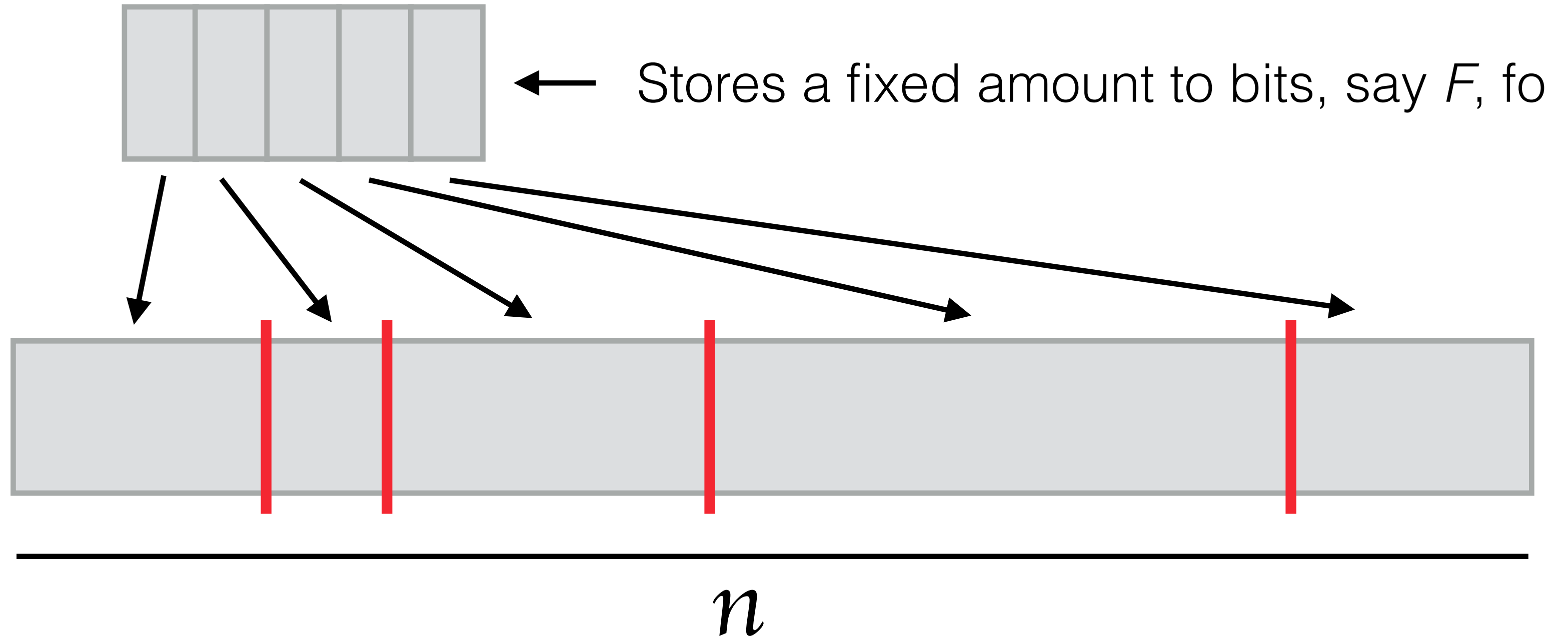


Dynamic Programming (DP)

guarantee

Optimal

Computing an optimal partition



1st level

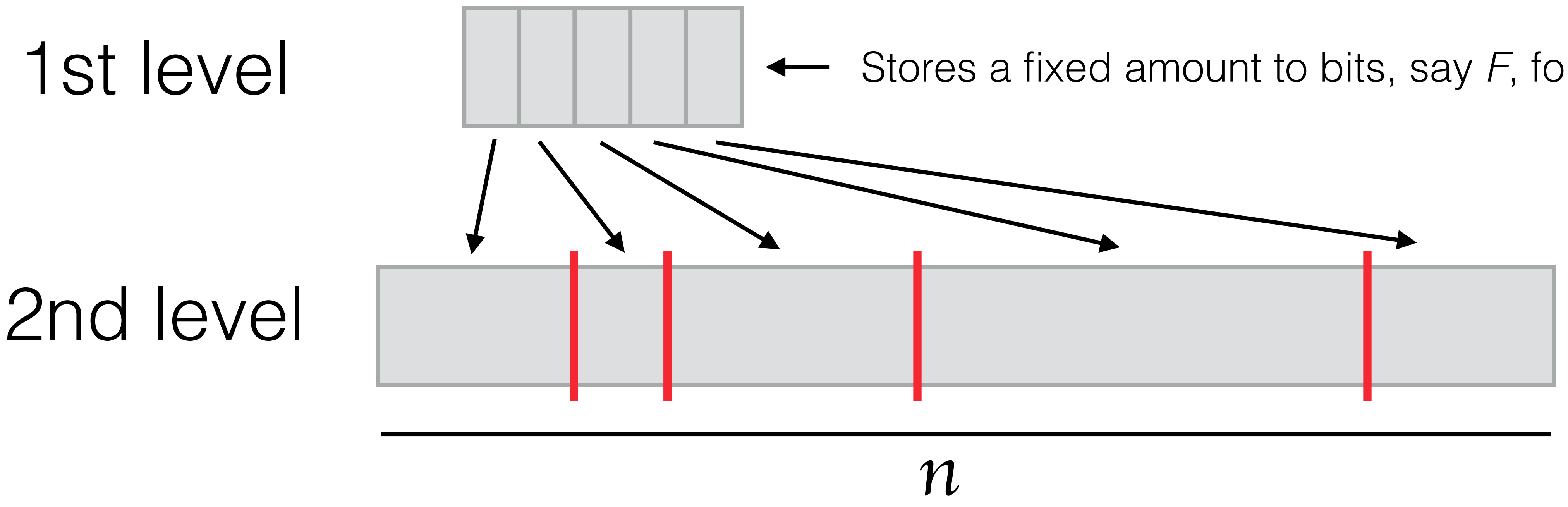
← Stores a fixed amount to bits, say *F*, for each partition.

2nd level

n

	guarantee	time	space	
Dynamic Programming (DP)	Optimal	$\Theta(n^2)$	$O(n)$	😑

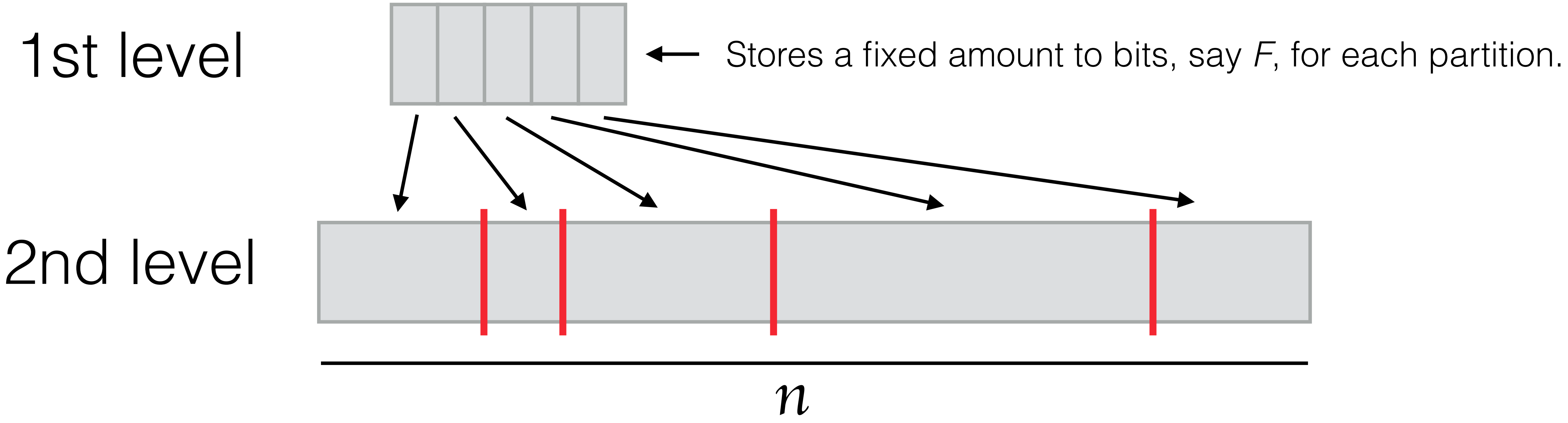
Computing an optimal partition



← Stores a fixed amount to bits, say *F*, for each partition.

	guarantee	time	space	
Dynamic Programming (DP)	Optimal	$\Theta(n^2)$	$O(n)$	😑
DP Approximation	ϵ-Optimal	$O(n \log_{1+\epsilon} 1/\epsilon)$	$O(n)$	😀

Computing an optimal partition



1st level

← Stores a fixed amount to bits, say *F*, for each partition.

2nd level

n

	guarantee	time	space	
Dynamic Programming (DP)	Optimal	$\Theta(n^2)$	$O(n)$	😑
DP Approximation	ϵ-Optimal	$O(n \log_{1+\epsilon} 1/\epsilon)$	$O(n)$	😀
Our solution	Optimal	$\Theta(n)$	$O(1)$	😍

Computing an optimal partition

Why is it so difficult?

Computing an optimal partition

Why is it so difficult?

[8, 9, 10, 11, 12, 36, 37, 38, 39, 40]

Computing an optimal partition

Why is it so difficult?

[8, 9, 10, 11, 12, 36, 37, 38, 39, 40]

splitting	**Elias-Fano**	**VByte**
[0,10)	4 bpi	
[0,5)[5,10)	[4][5] bpi	
[0,6)[6,10)	[5][2] bpi	

Computing an optimal partition

Why is it so difficult?

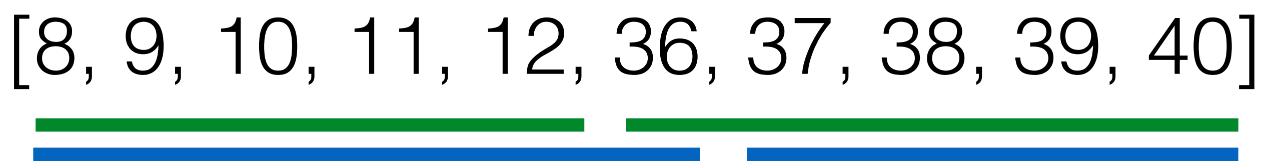
[8, 9, 10, 11, 12, 36, 37, 38, 39, 40]

splitting	**Elias-Fano**	**VByte**
[0,10)	4 bpi	
[0,5)[5,10)	[4][5] bpi	
[0,6)[6,10)	[5][2] bpi	

Different costs to represent *the same* integers! —> DP

Computing an optimal partition

Why is it so difficult?



[8, 9, 10, 11, 12, 36, 37, 38, 39, 40]

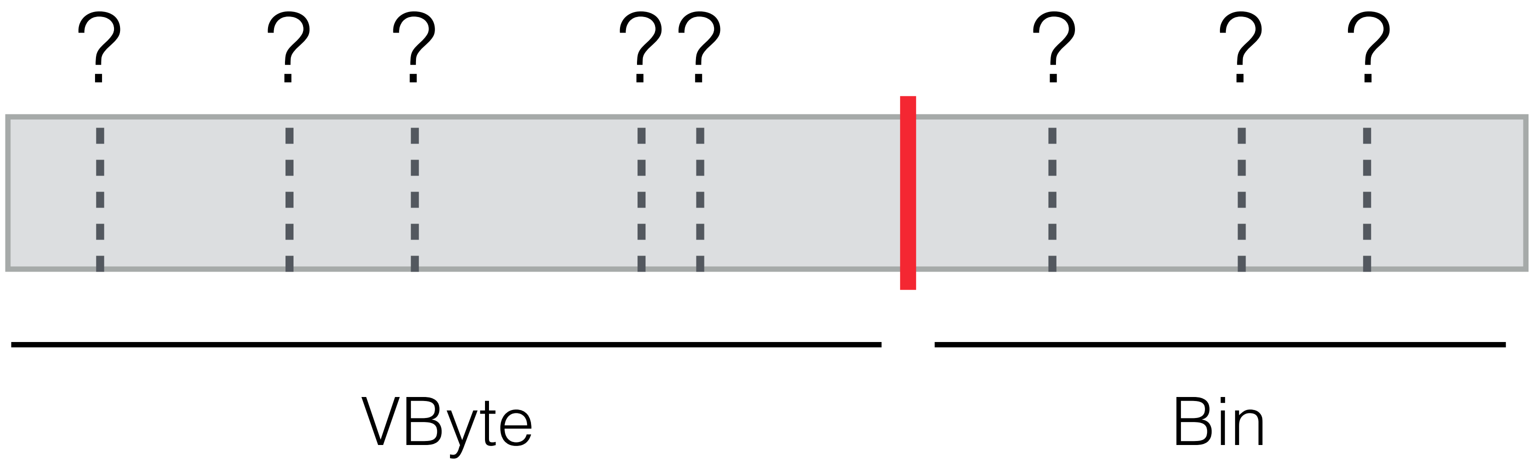
splitting	**Elias-Fano**	**VByte**
[0,10)	4 bpi	8 bpi
[0,5)[5,10)	[4][5] bpi	[8][8] bpi
[0,6)[6,10)	[5][2] bpi	[8][8] bpi

Different costs to represent *the same* integers! —> DP

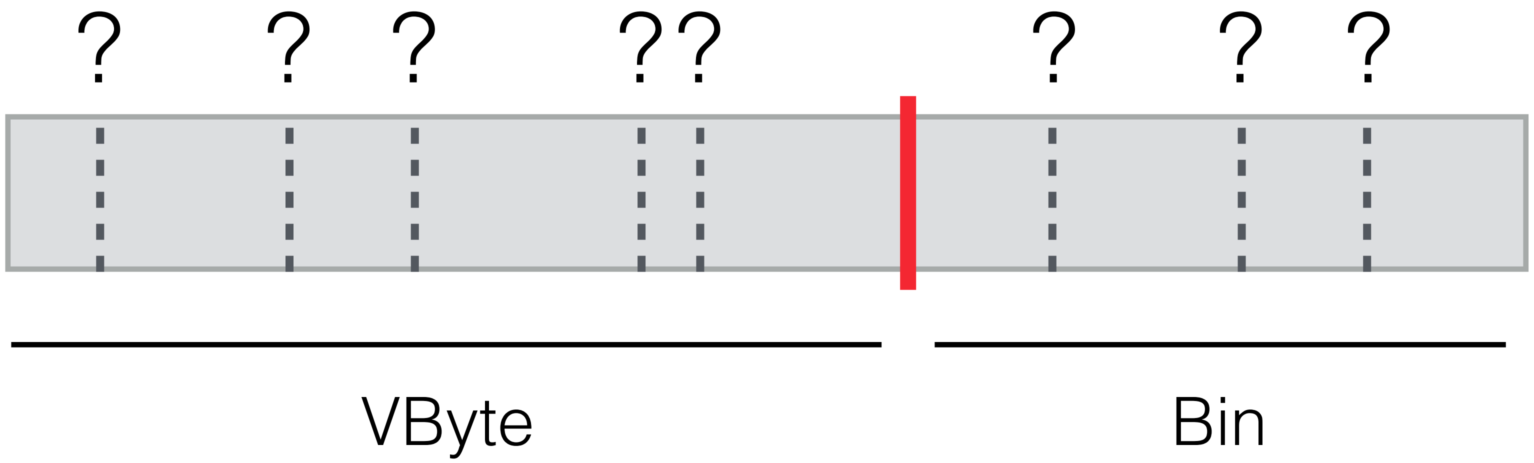
Costs do NOT change if we consider different splittings.

Our solution - The intuition

Our solution - The intuition



Our solution - The intuition

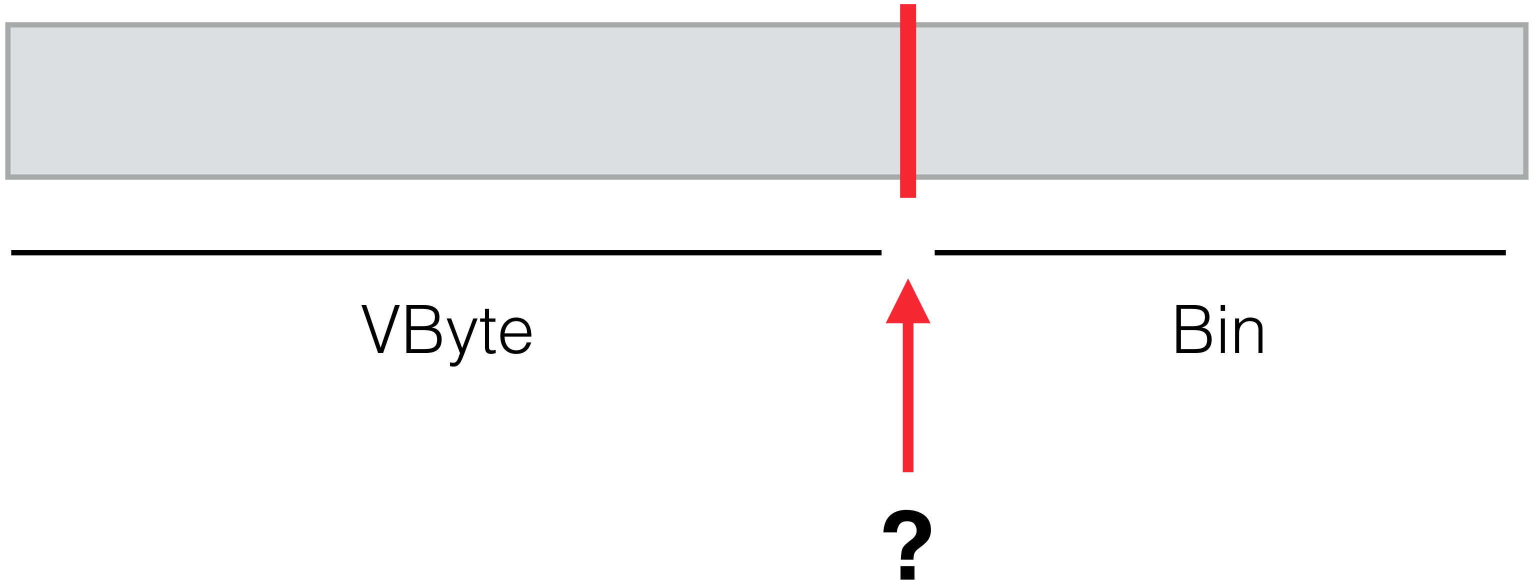


If VByte is winning over Bin, we do NOT have to try any split.

Our solution - The intuition

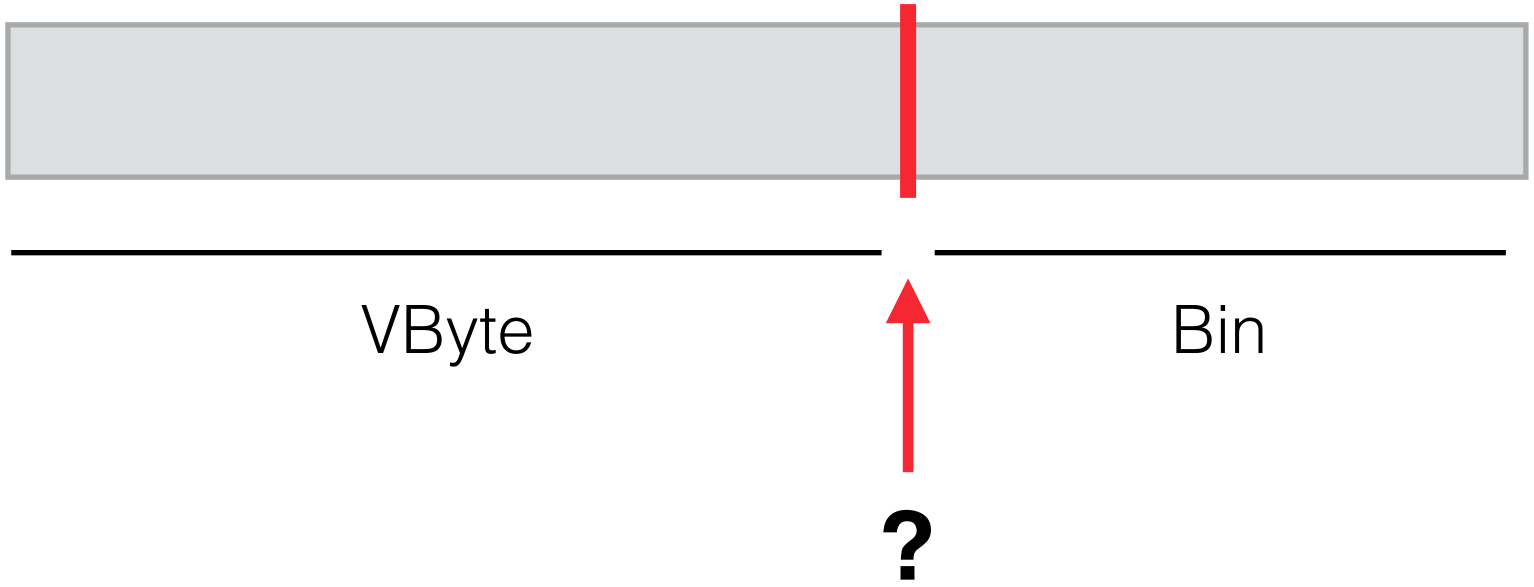
If VByte is winning over Bin, we do NOT have to try any split.

Our solution - The intuition



If VByte is winning over Bin, we do NOT have to try any split.

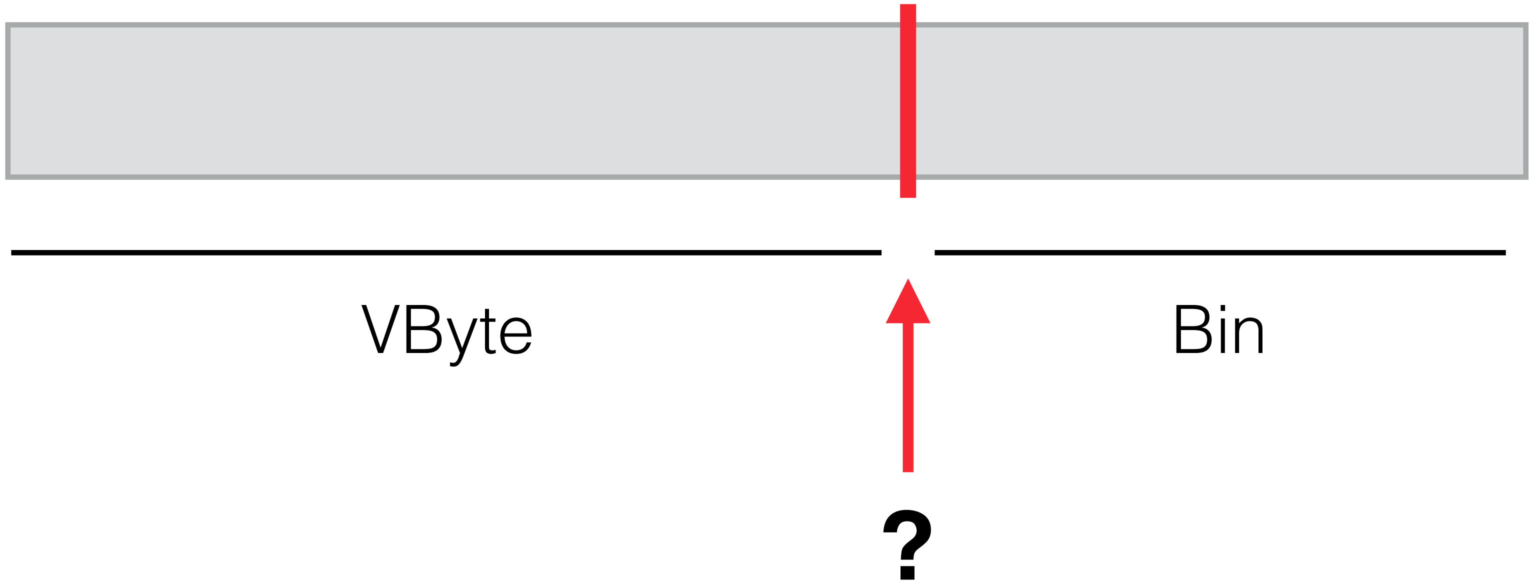
Our solution - The intuition



If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

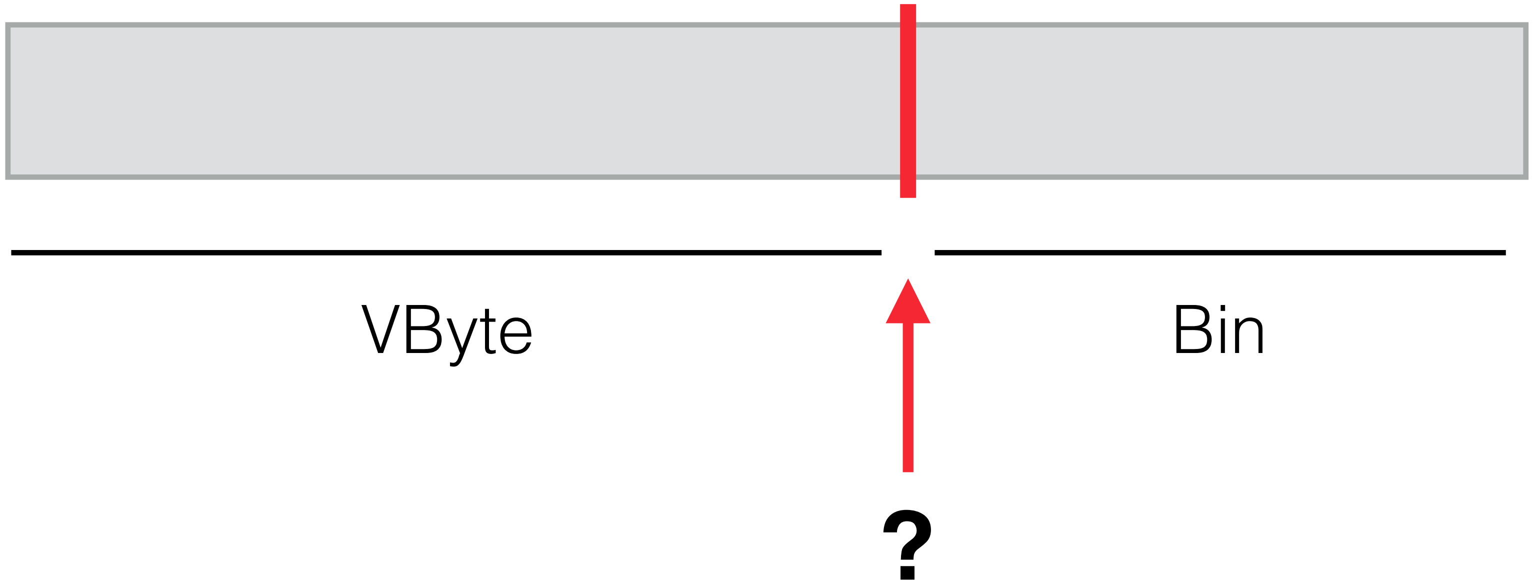
Our solution - The intuition



If VByte is winning over Bin, we do NOT have to try any split.

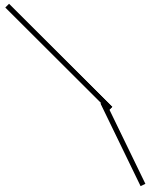
gain = VByte - Bin

Our solution - The intuition

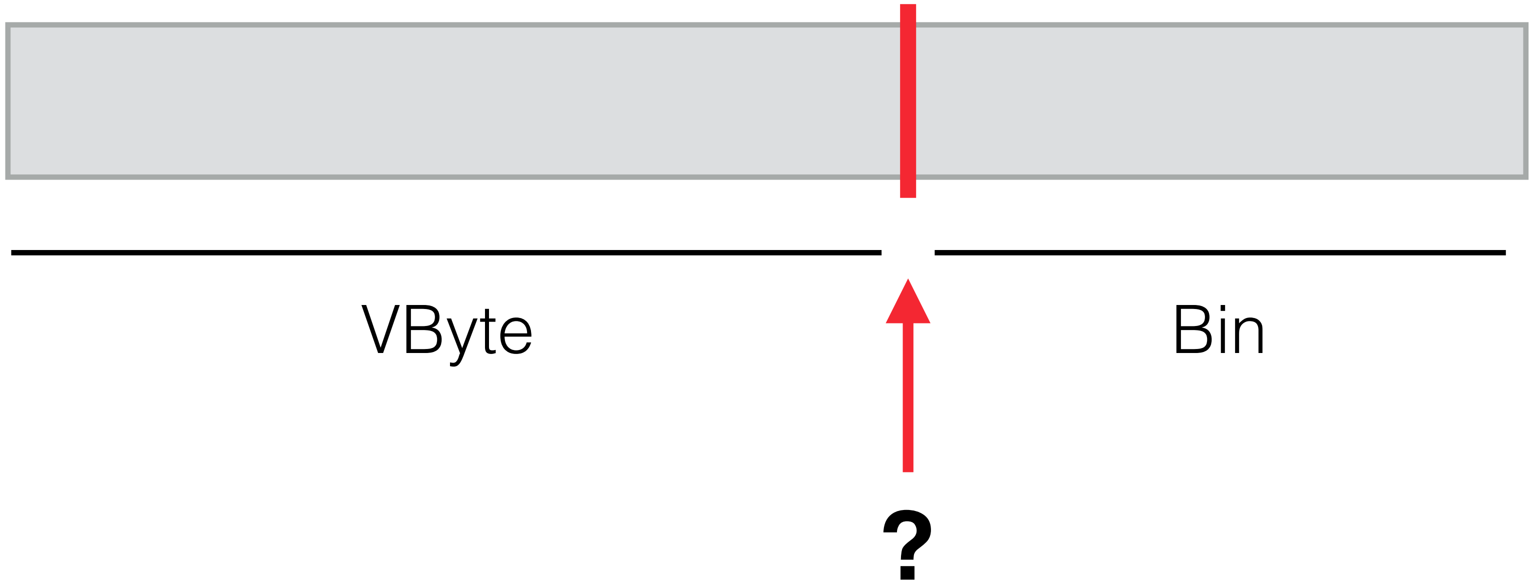


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

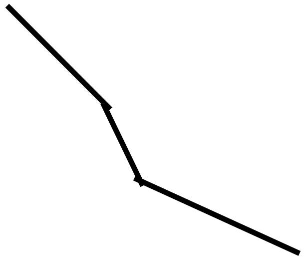


Our solution - The intuition

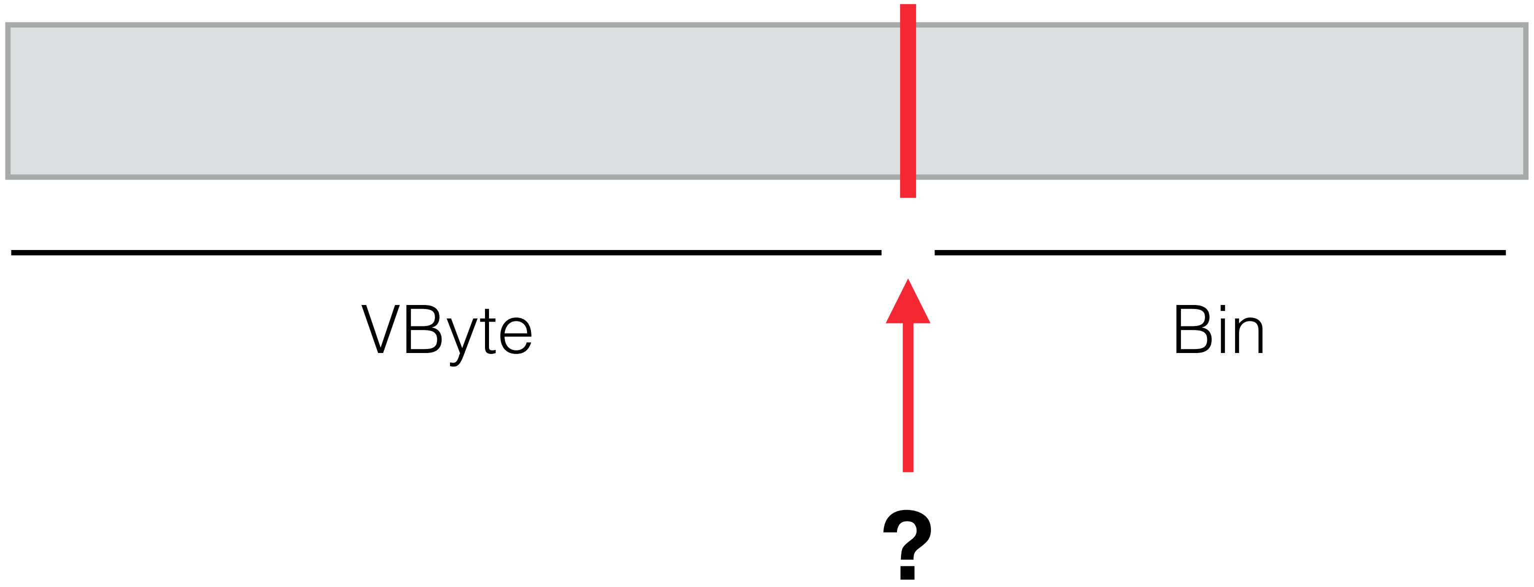


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

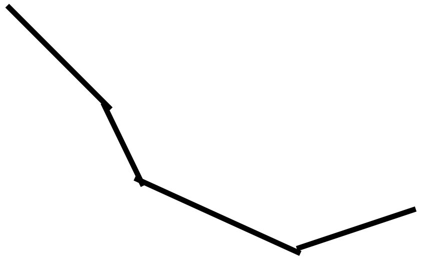


Our solution - The intuition

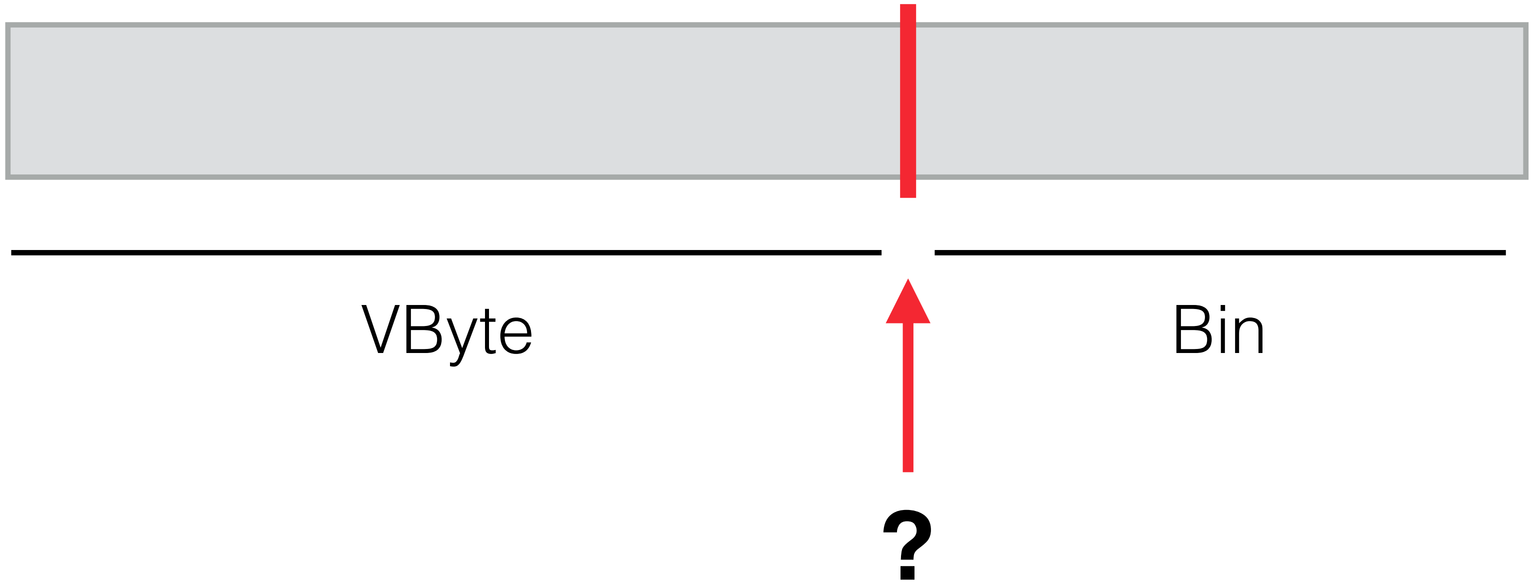


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

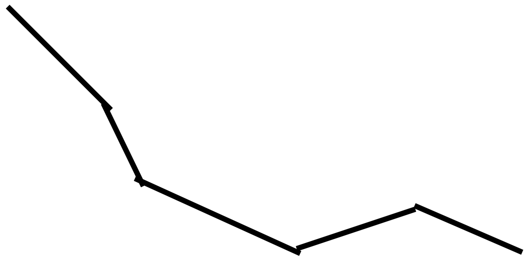


Our solution - The intuition

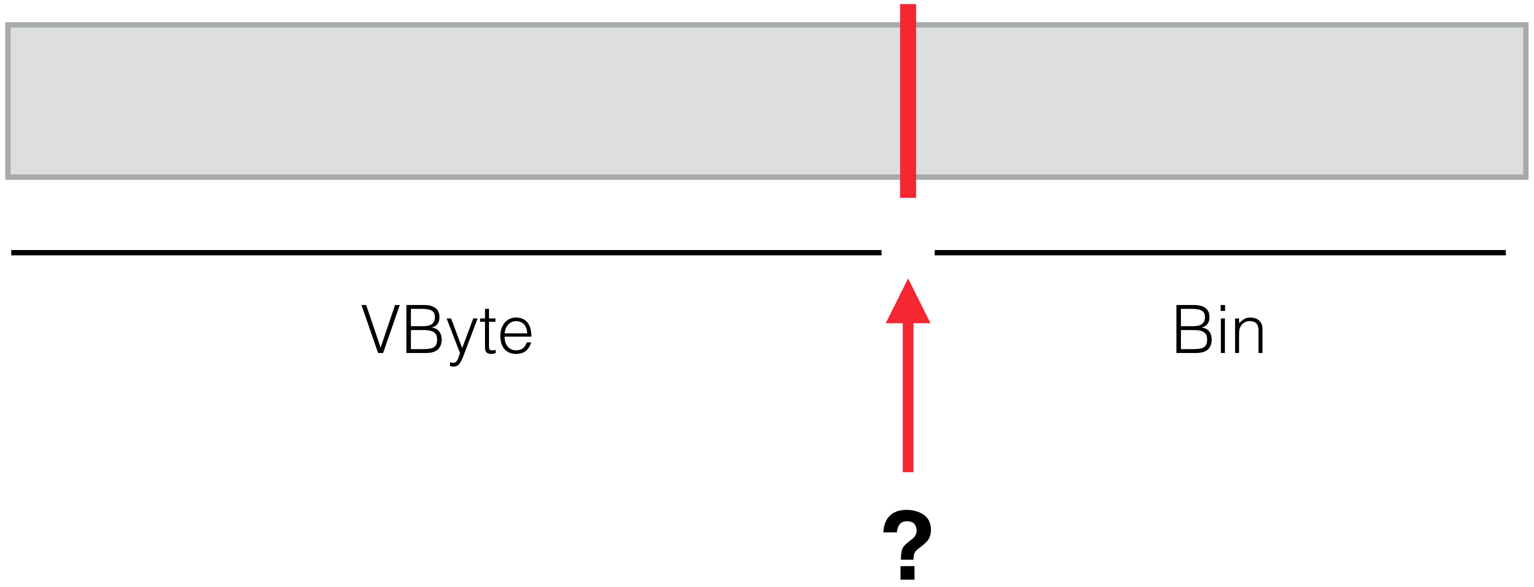


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

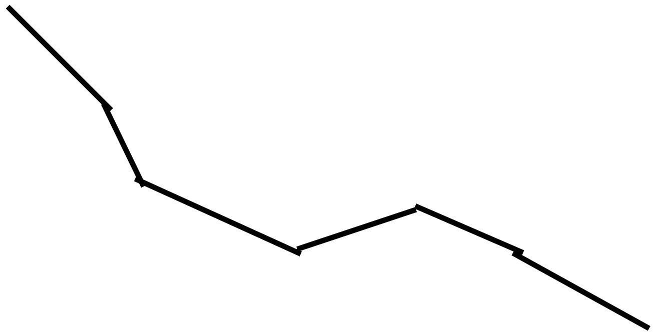


Our solution - The intuition

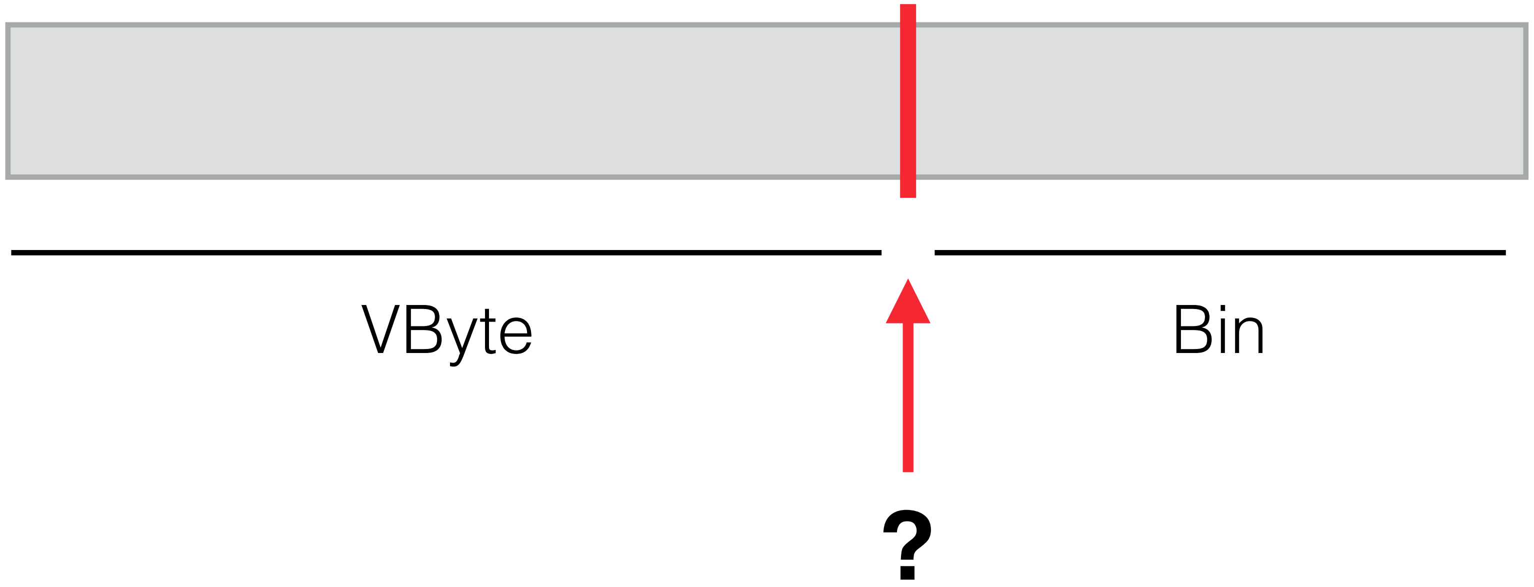


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

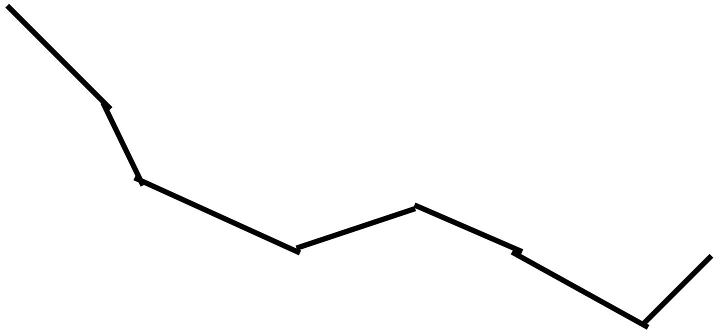


Our solution - The intuition

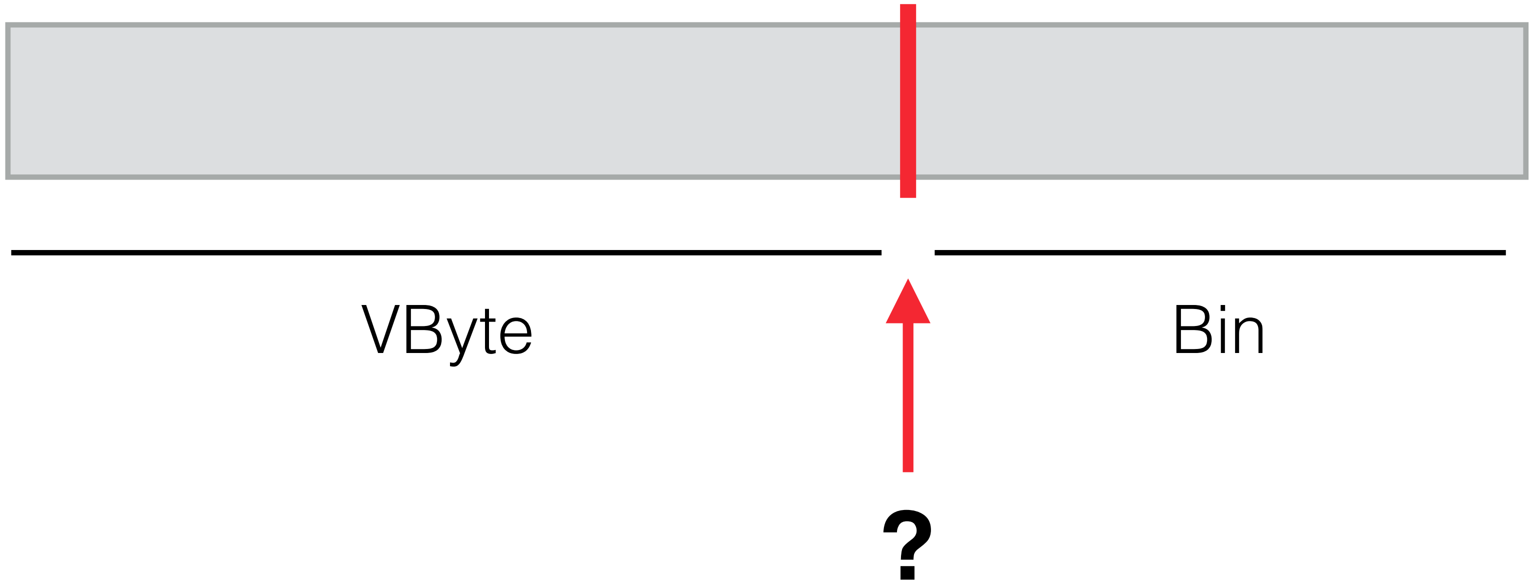


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin



Our solution - The intuition

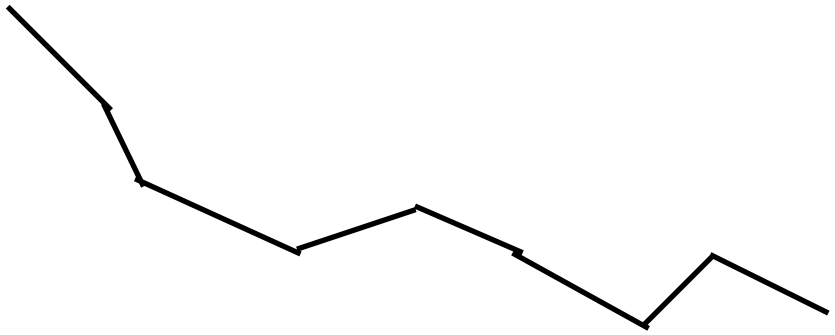

VByte

Bin

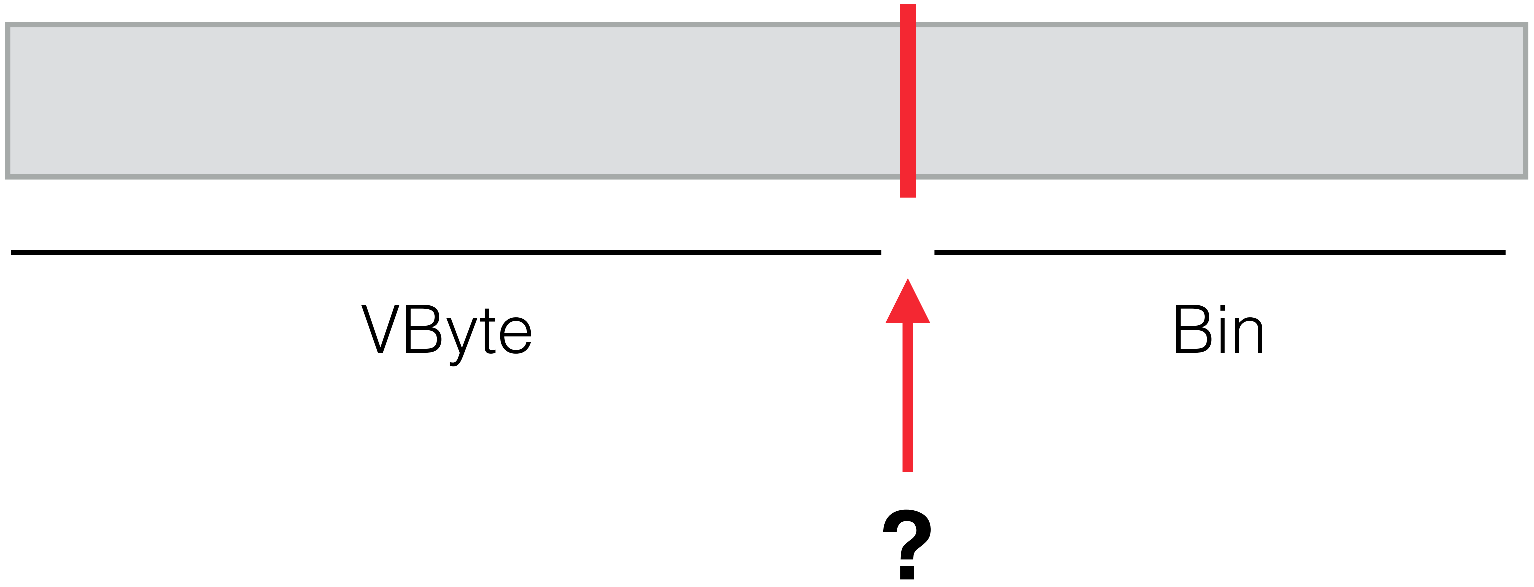
?

If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

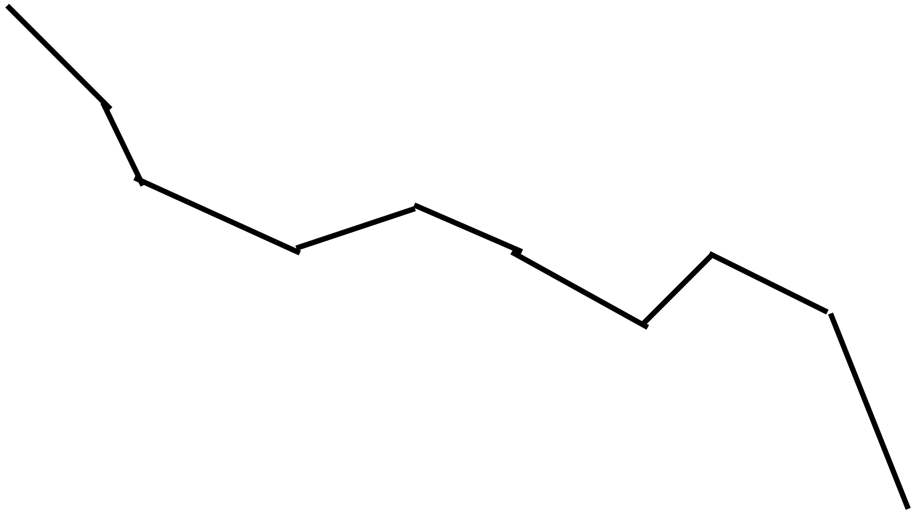


Our solution - The intuition

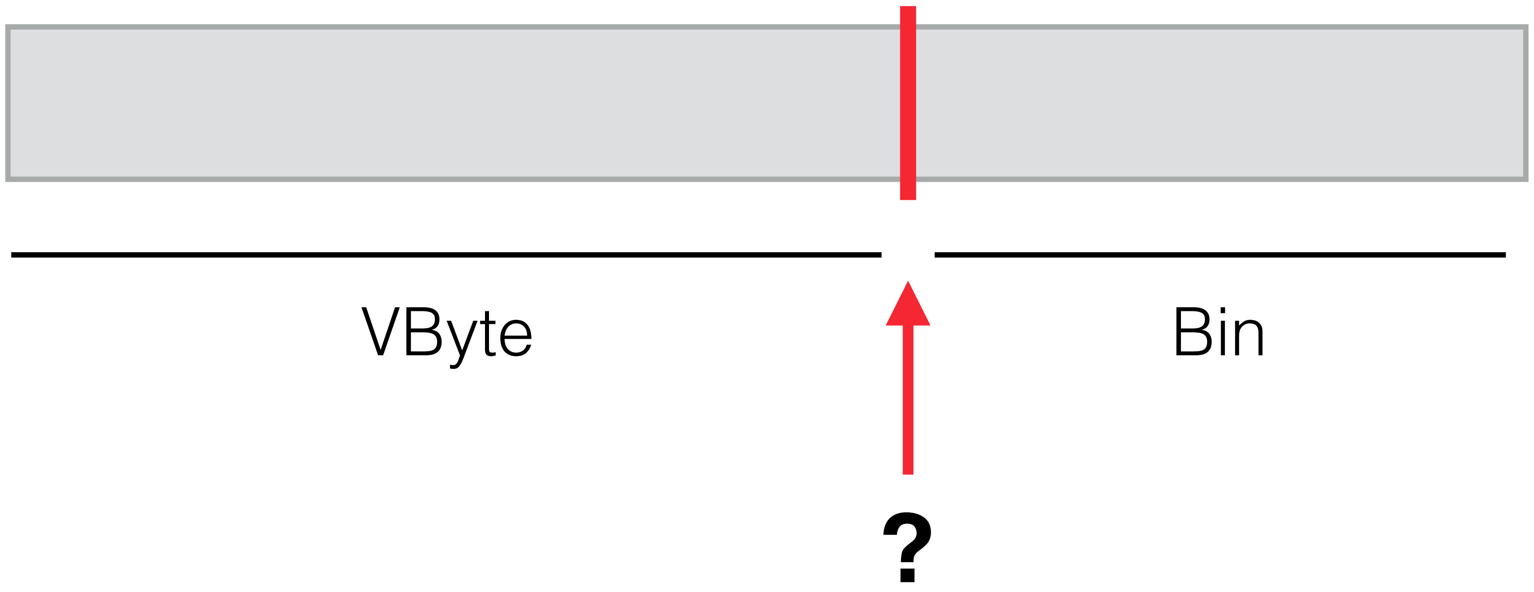


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

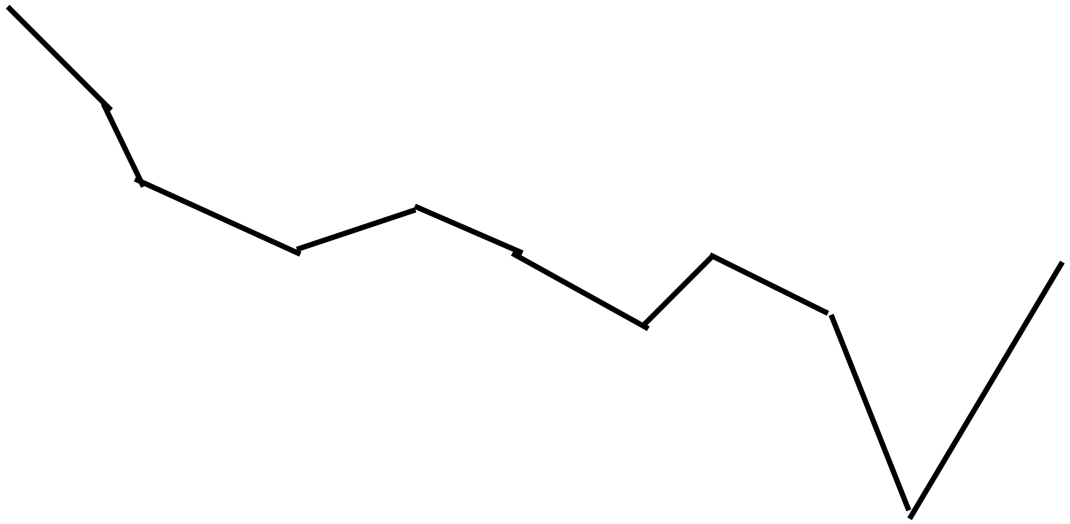


Our solution - The intuition

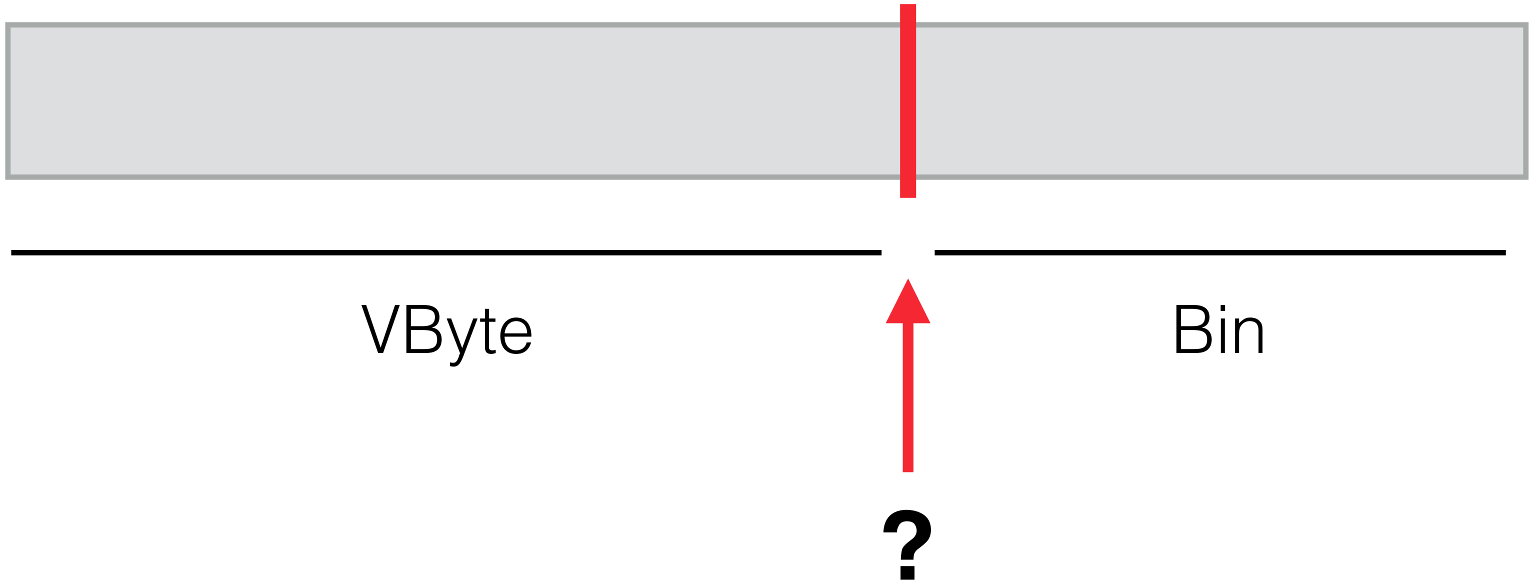


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

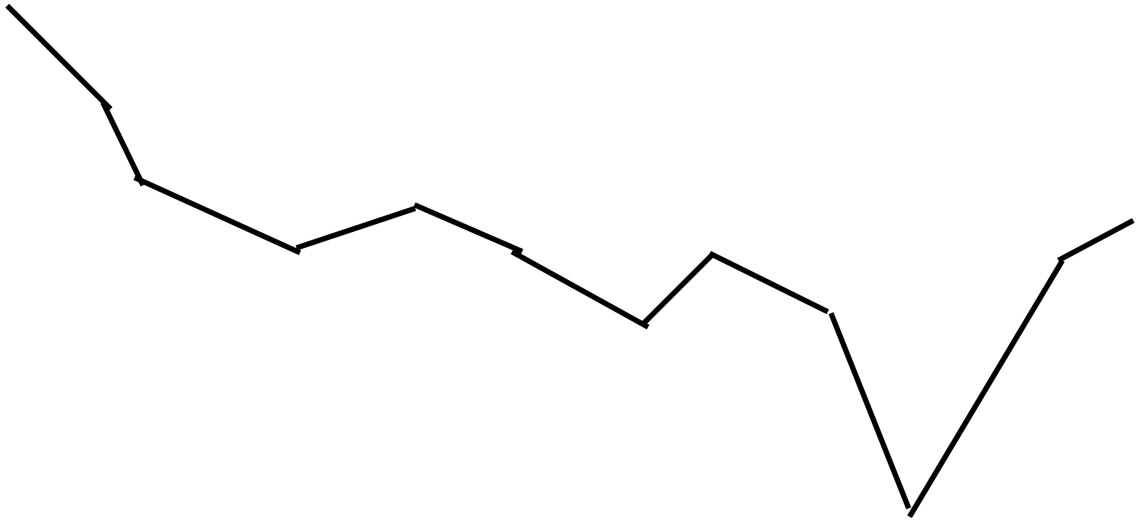


Our solution - The intuition

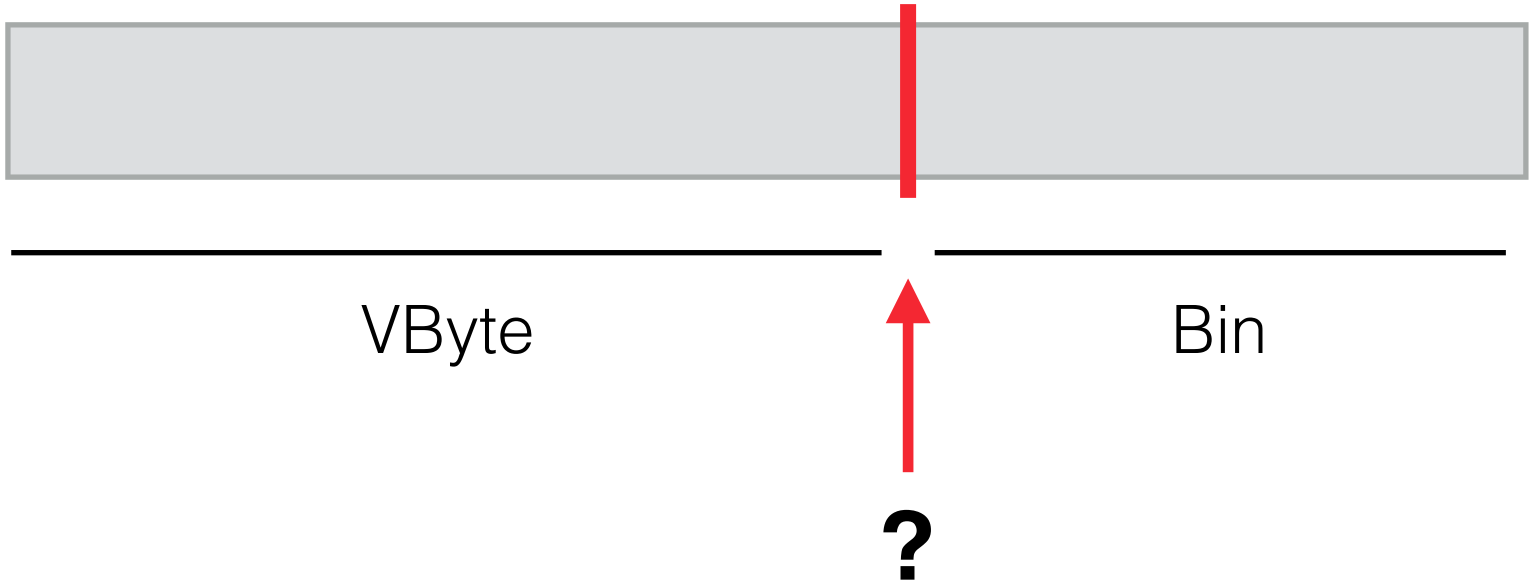


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

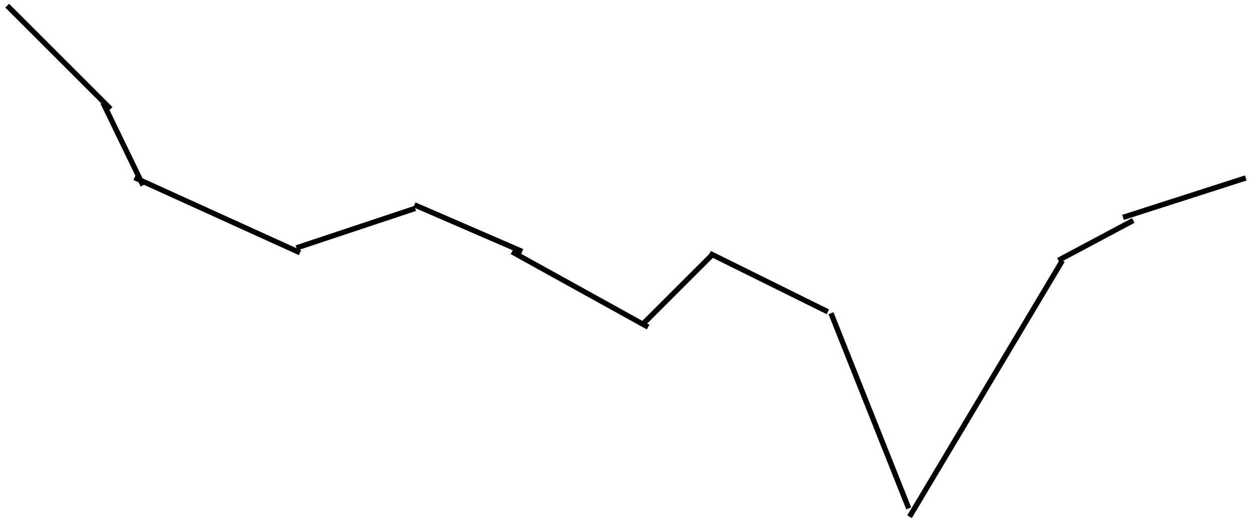


Our solution - The intuition

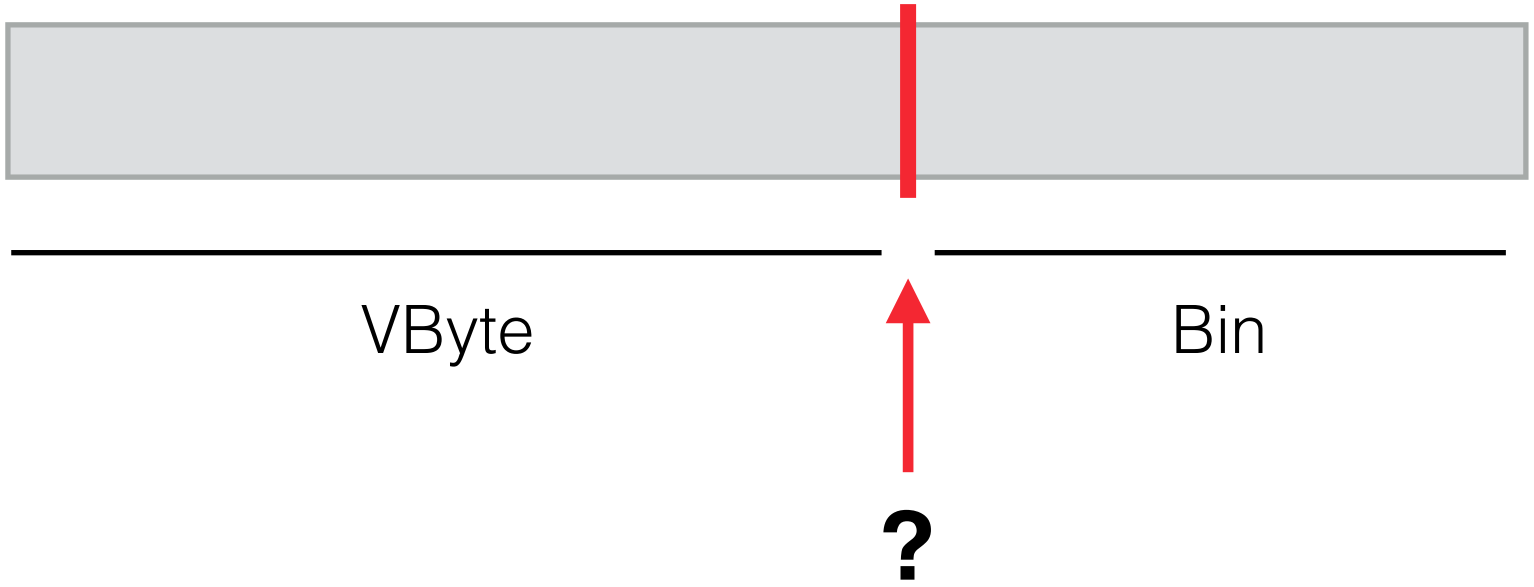


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

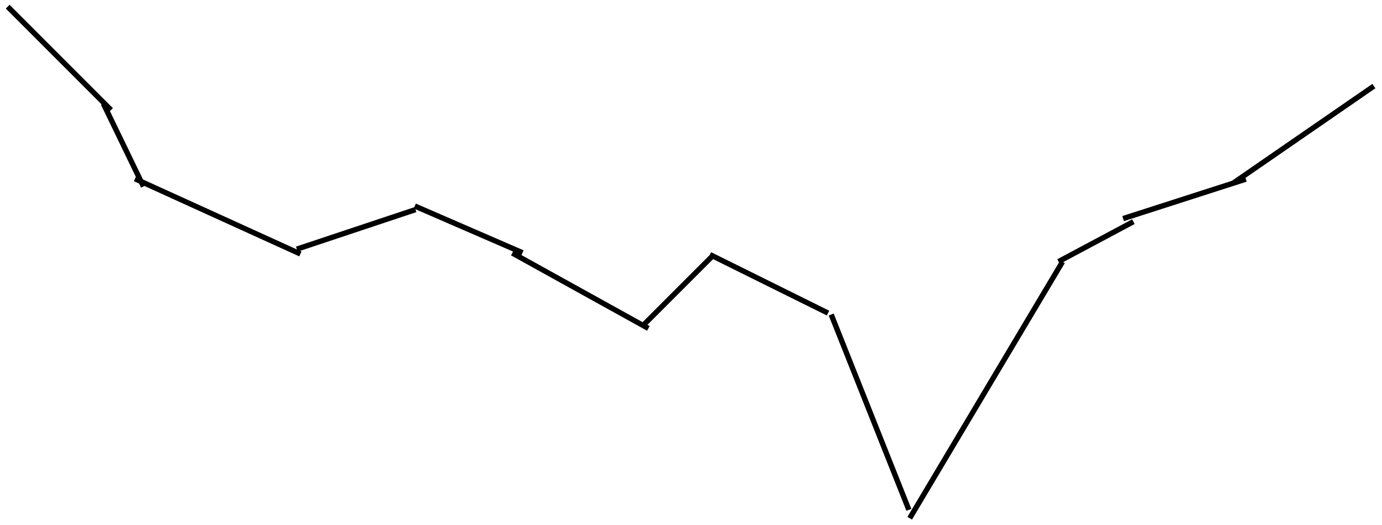


Our solution - The intuition

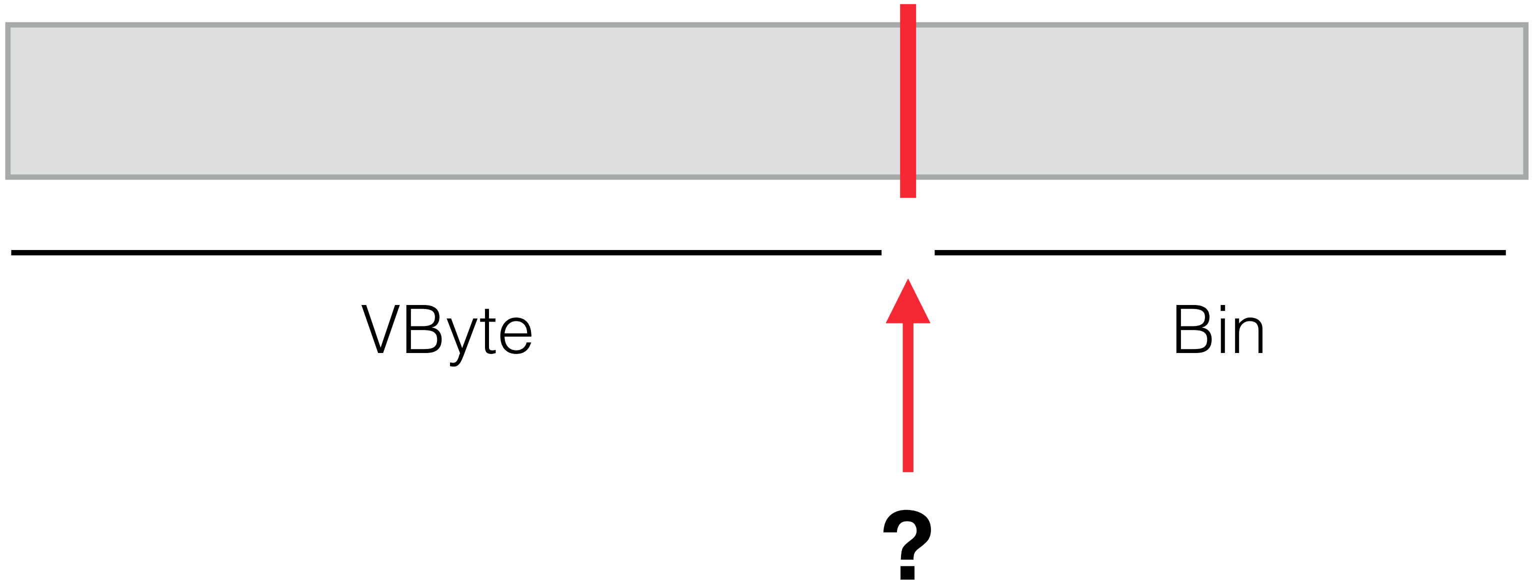


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

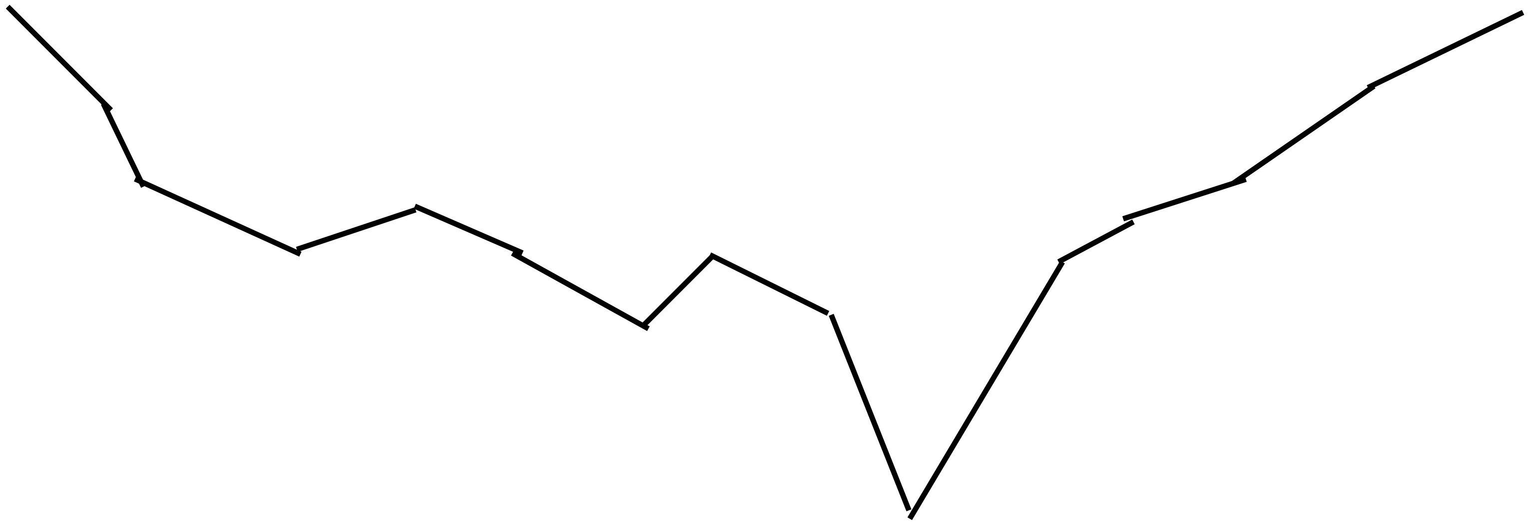


Our solution - The intuition

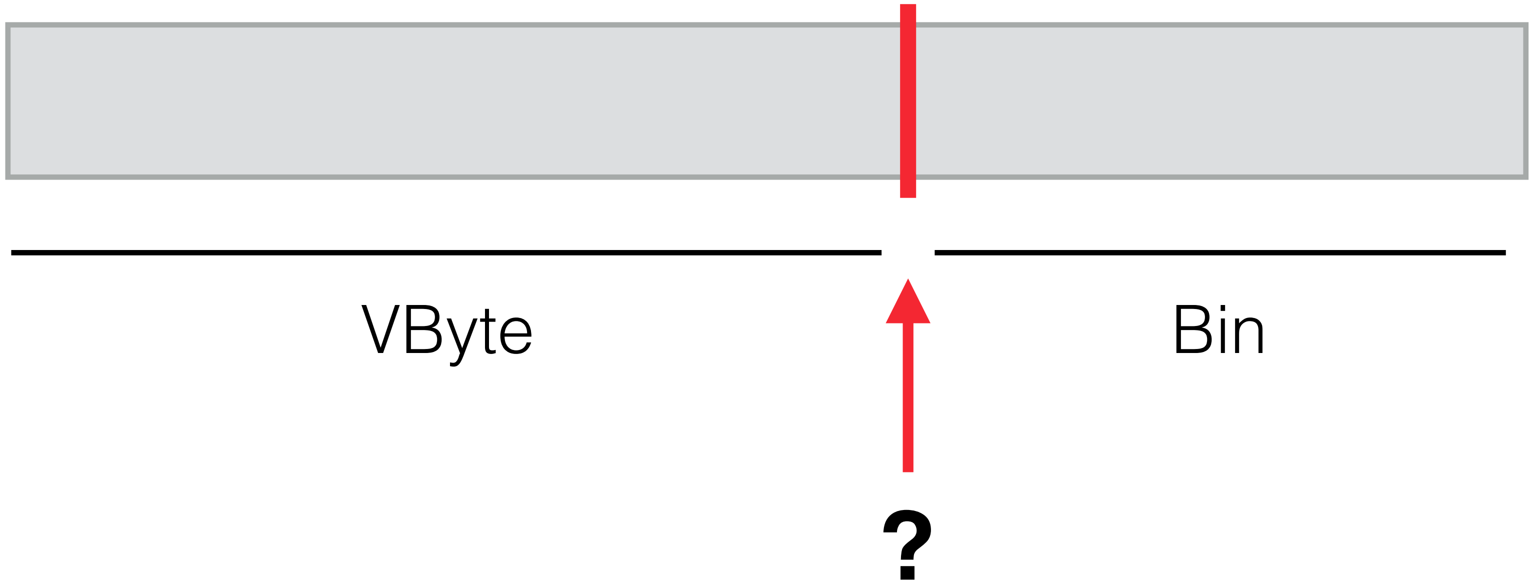


If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

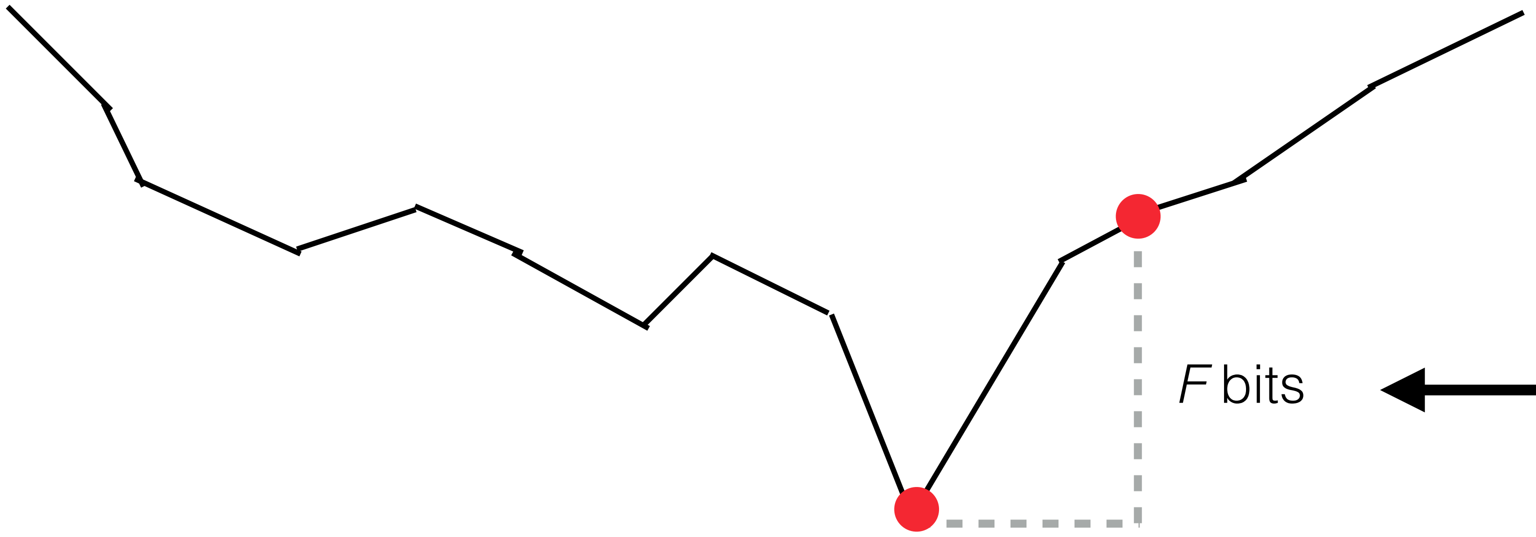


Our solution - The intuition



If VByte is winning over Bin, we do NOT have to try any split.

gain = VByte - Bin

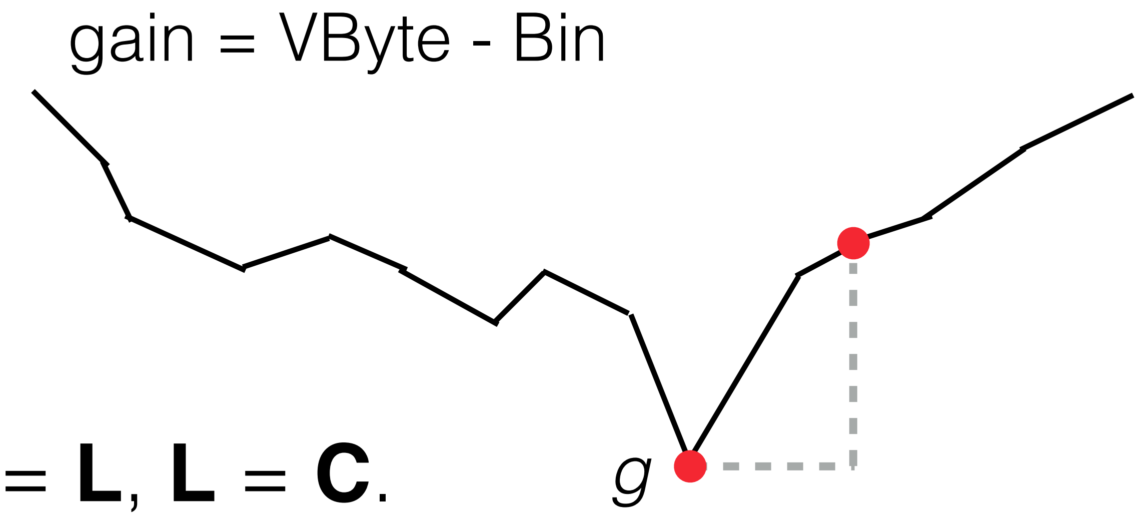


We pay *F* bits for each partition!

Our solution - The algorithm

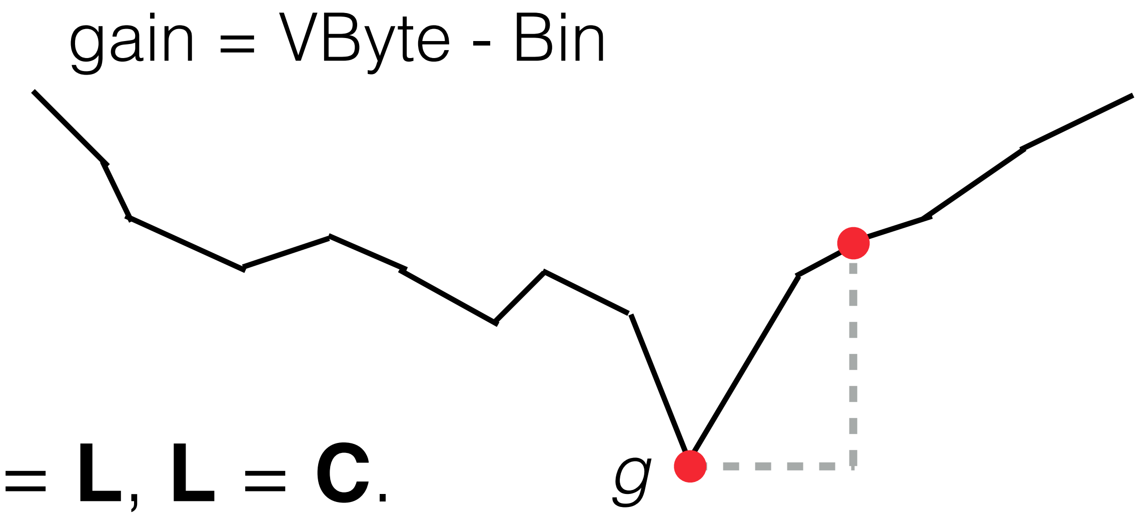
C := current encoder
L := last encoder

1. Encode first partition.
2. If |**C** - **L**| and *g* are > 2*F*, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.



Our solution - The algorithm

C := current encoder
L := last encoder



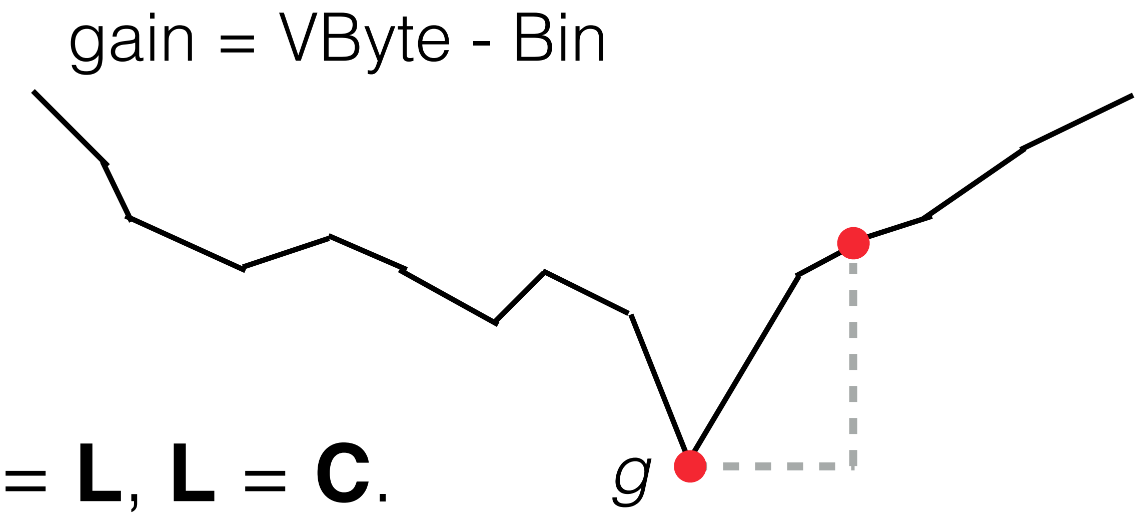
1. Encode first partition.
2. If |**C** - **L**| and *g* are > 2*F*, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.

All details (including proof of optimality), here:
https://arxiv.org/abs/1804.10949

Our solution - The algorithm

C := current encoder
L := last encoder

1. Encode first partition.
2. If |**C** - **L**| and *g* are > 2*F*, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.

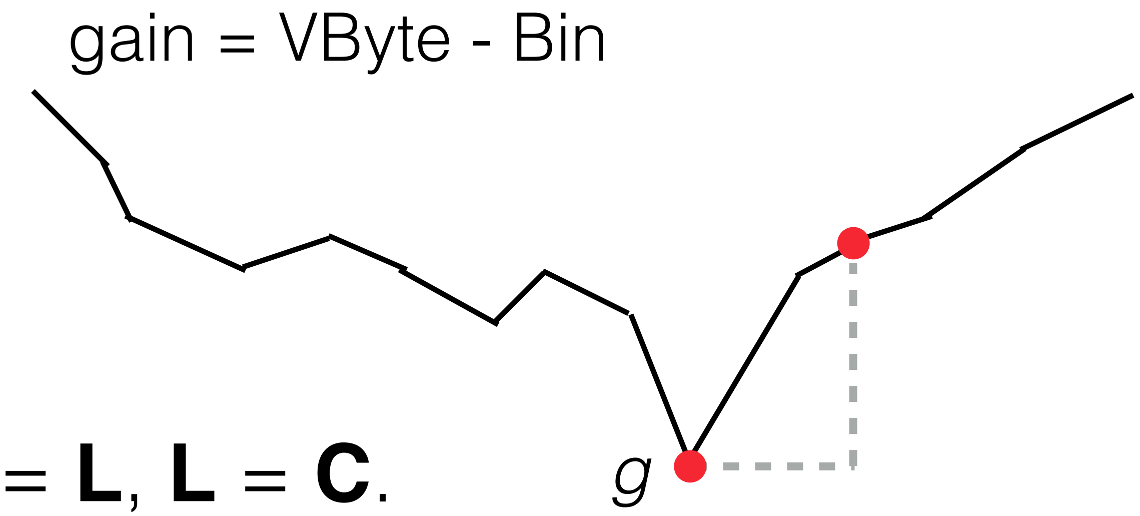


Optimal.

All details (including proof of optimality), here:
https://arxiv.org/abs/1804.10949

Our solution - The algorithm

C := current encoder
L := last encoder



1. Encode first partition.
2. If |**C** - **L**| and g are > $2F$, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.

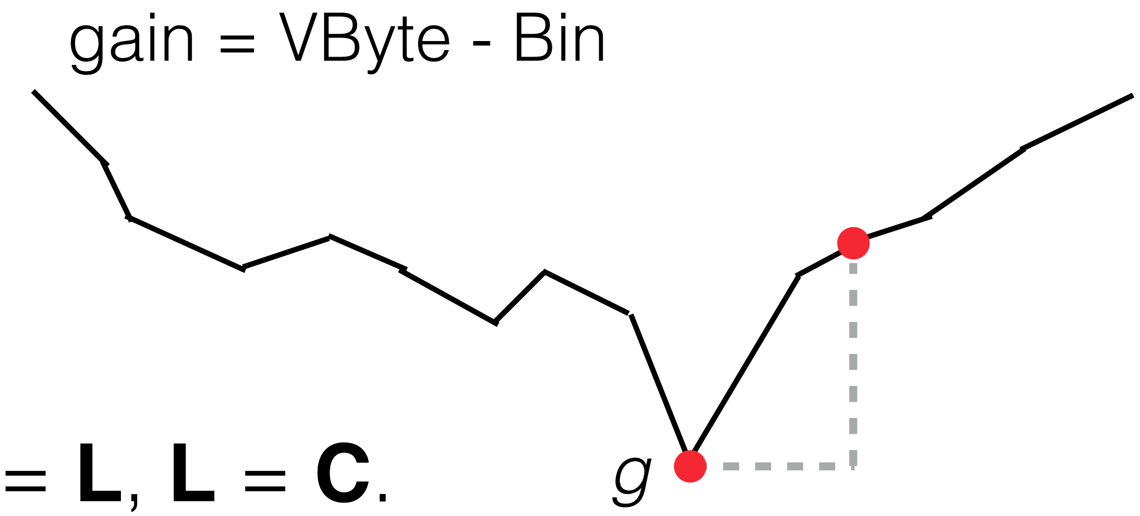
All details (including proof of optimality), here:
https://arxiv.org/abs/1804.10949

Optimal.

Valid for ANY *point-wise* encoder.

Our solution - The algorithm

C := current encoder
L := last encoder



1. Encode first partition.
2. If |**C** - **L**| and *g* are > 2*F*, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.

All details (including proof of optimality), here:
https://arxiv.org/abs/1804.10949

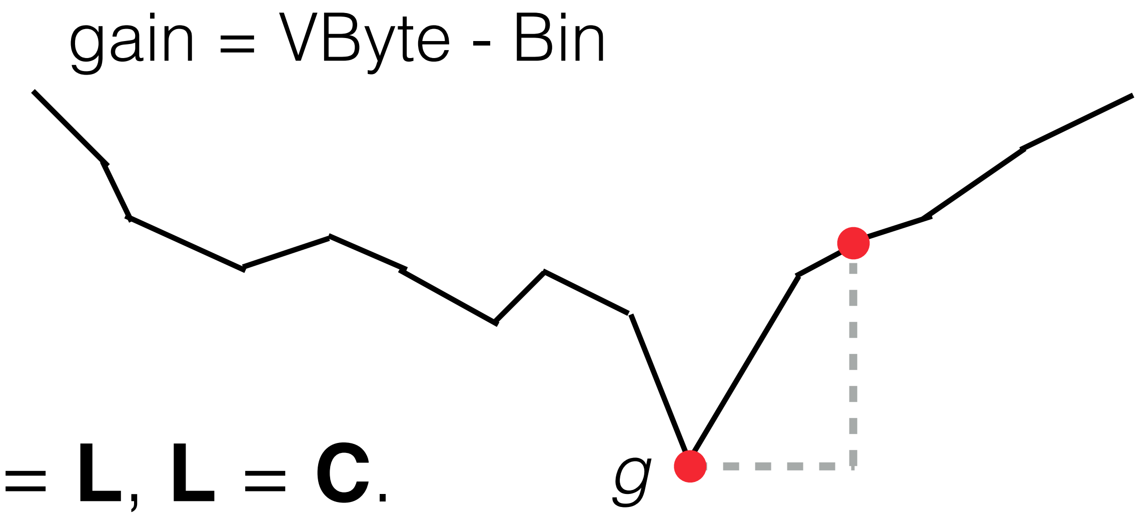
Optimal.

Valid for ANY *point-wise* encoder.

Very low constant factors.

Our solution - The algorithm

C := current encoder
L := last encoder



1. Encode first partition.
2. If |**C** - **L**| and g are > $2F$, then encode the current partition with **C** and set **C** = **L**, **L** = **C**.
3. Repeat step (2) until the end of the sequence.
4. Encode last partition.

All details (including proof of optimality), here: https://arxiv.org/abs/1804.10949

Optimal.

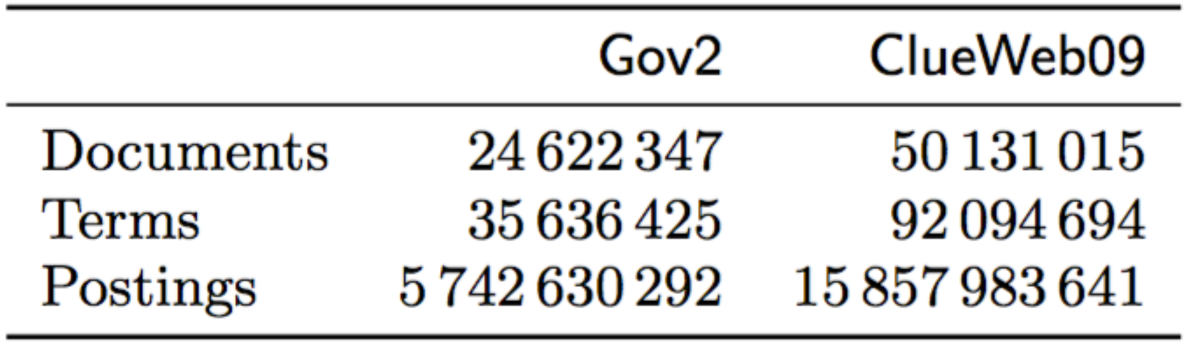
Valid for ANY *point-wise* encoder.

Very low constant factors.

Linear time and constant space.

Experimental Results on Gov2 and ClueWeb09

`C++14` implementation compiled with `gcc` 7.2.0 with highest optimization setting.

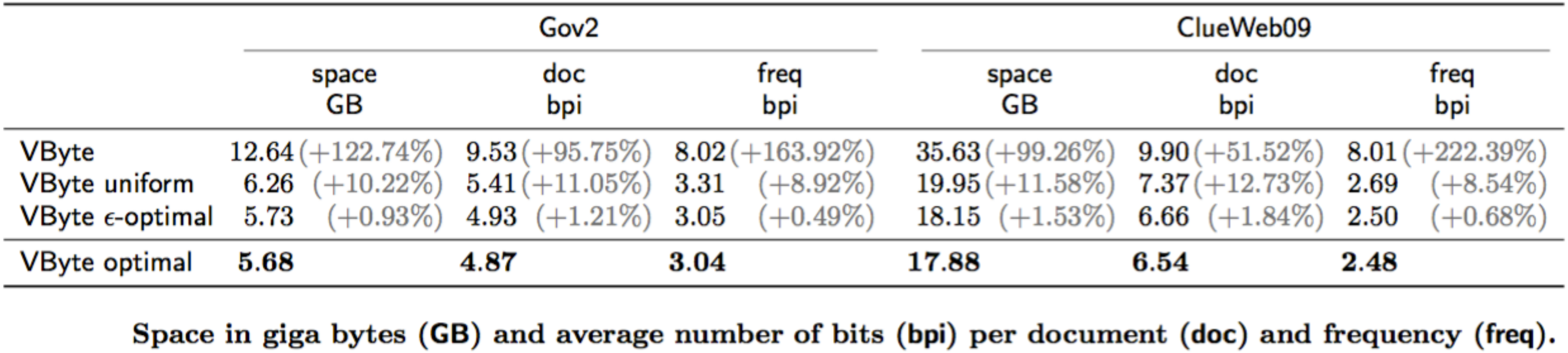


	Gov2	ClueWeb09
Documents	24 622 347	50 131 015
Terms	35 636 425	92 094 694
Postings	5 742 630 292	15 857 983 641

Basic statistics for the tested collections.

Code will be available upon acceptance of the paper.

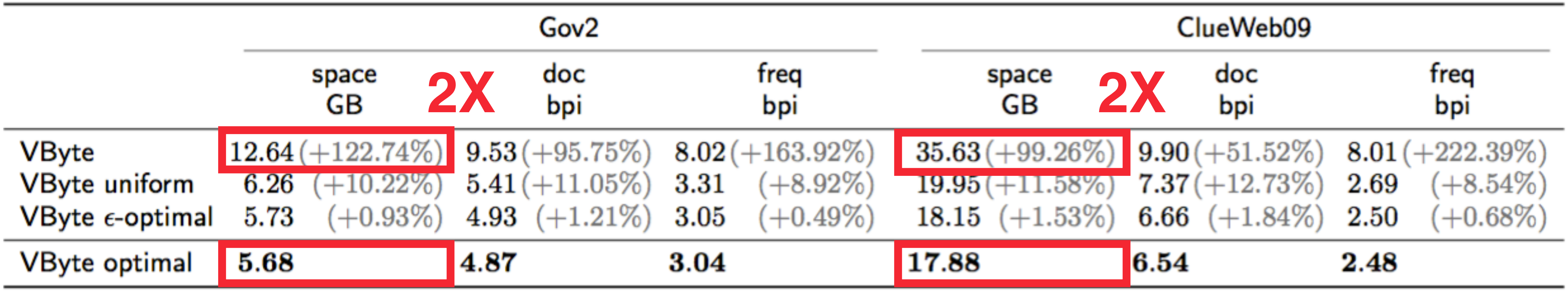
Partitioned VS. Unpartitioned: Space and Indexing Time



	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
VByte	12.64 (+122.74%)	9.53 (+95.75%)	8.02 (+163.92%)	35.63 (+99.26%)	9.90 (+51.52%)	8.01 (+222.39%)
VByte uniform	6.26 (+10.22%)	5.41 (+11.05%)	3.31 (+8.92%)	19.95 (+11.58%)	7.37 (+12.73%)	2.69 (+8.54%)
VByte ϵ-optimal	5.73 (+0.93%)	4.93 (+1.21%)	3.05 (+0.49%)	18.15 (+1.53%)	6.66 (+1.84%)	2.50 (+0.68%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).

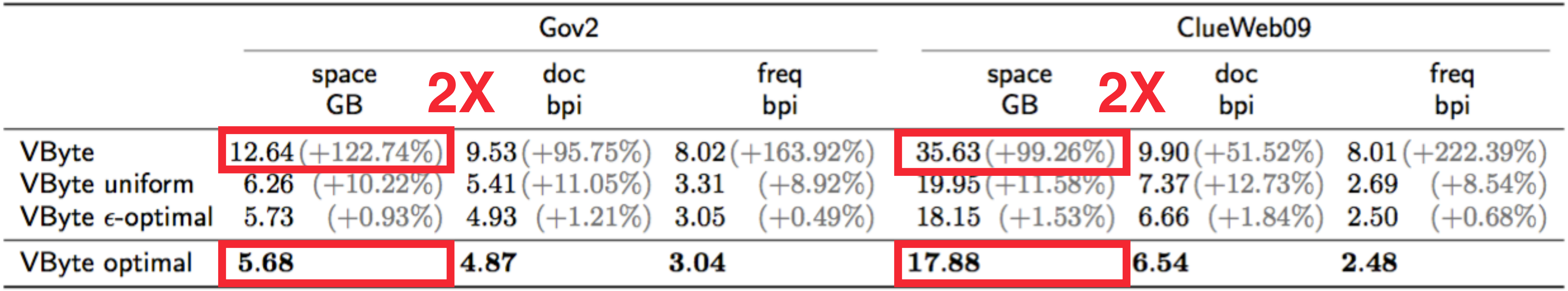
Partitioned VS. Unpartitioned: Space and Indexing Time



	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
VByte	12.64 (+122.74%)	9.53 (+95.75%)	8.02 (+163.92%)	35.63 (+99.26%)	9.90 (+51.52%)	8.01 (+222.39%)
VByte uniform	6.26 (+10.22%)	5.41 (+11.05%)	3.31 (+8.92%)	19.95 (+11.58%)	7.37 (+12.73%)	2.69 (+8.54%)
VByte ϵ-optimal	5.73 (+0.93%)	4.93 (+1.21%)	3.05 (+0.49%)	18.15 (+1.53%)	6.66 (+1.84%)	2.50 (+0.68%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

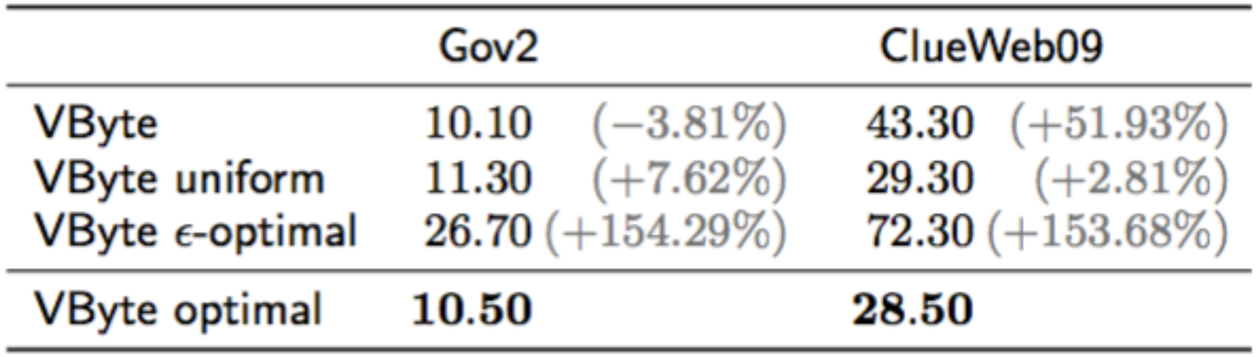
Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).

Partitioned VS. Unpartitioned: Space and Indexing Time



	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
VByte	12.64 (+122.74%)	9.53 (+95.75%)	8.02 (+163.92%)	35.63 (+99.26%)	9.90 (+51.52%)	8.01 (+222.39%)
VByte uniform	6.26 (+10.22%)	5.41 (+11.05%)	3.31 (+8.92%)	19.95 (+11.58%)	7.37 (+12.73%)	2.69 (+8.54%)
VByte ϵ-optimal	5.73 (+0.93%)	4.93 (+1.21%)	3.05 (+0.49%)	18.15 (+1.53%)	6.66 (+1.84%)	2.50 (+0.68%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

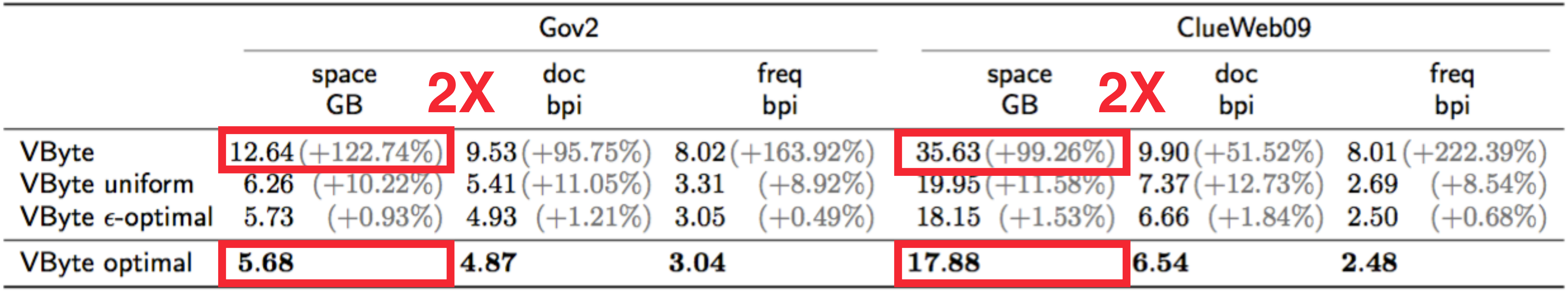
Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



	Gov2	ClueWeb09
VByte	10.10 (−3.81%)	43.30 (+51.93%)
VByte uniform	11.30 (+7.62%)	29.30 (+2.81%)
VByte ϵ-optimal	26.70 (+154.29%)	72.30 (+153.68%)
VByte optimal	**10.50**	**28.50**

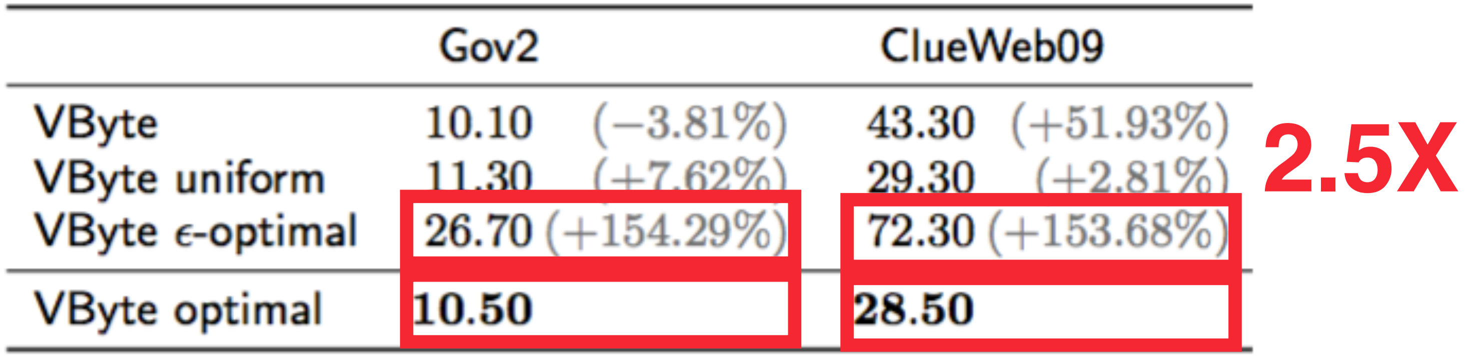
Index building timings in minutes.

Partitioned VS. Unpartitioned: Space and Indexing Time



	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
VByte	12.64 (+122.74%)	9.53 (+95.75%)	8.02 (+163.92%)	35.63 (+99.26%)	9.90 (+51.52%)	8.01 (+222.39%)
VByte uniform	6.26 (+10.22%)	5.41 (+11.05%)	3.31 (+8.92%)	19.95 (+11.58%)	7.37 (+12.73%)	2.69 (+8.54%)
VByte ϵ-optimal	5.73 (+0.93%)	4.93 (+1.21%)	3.05 (+0.49%)	18.15 (+1.53%)	6.66 (+1.84%)	2.50 (+0.68%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

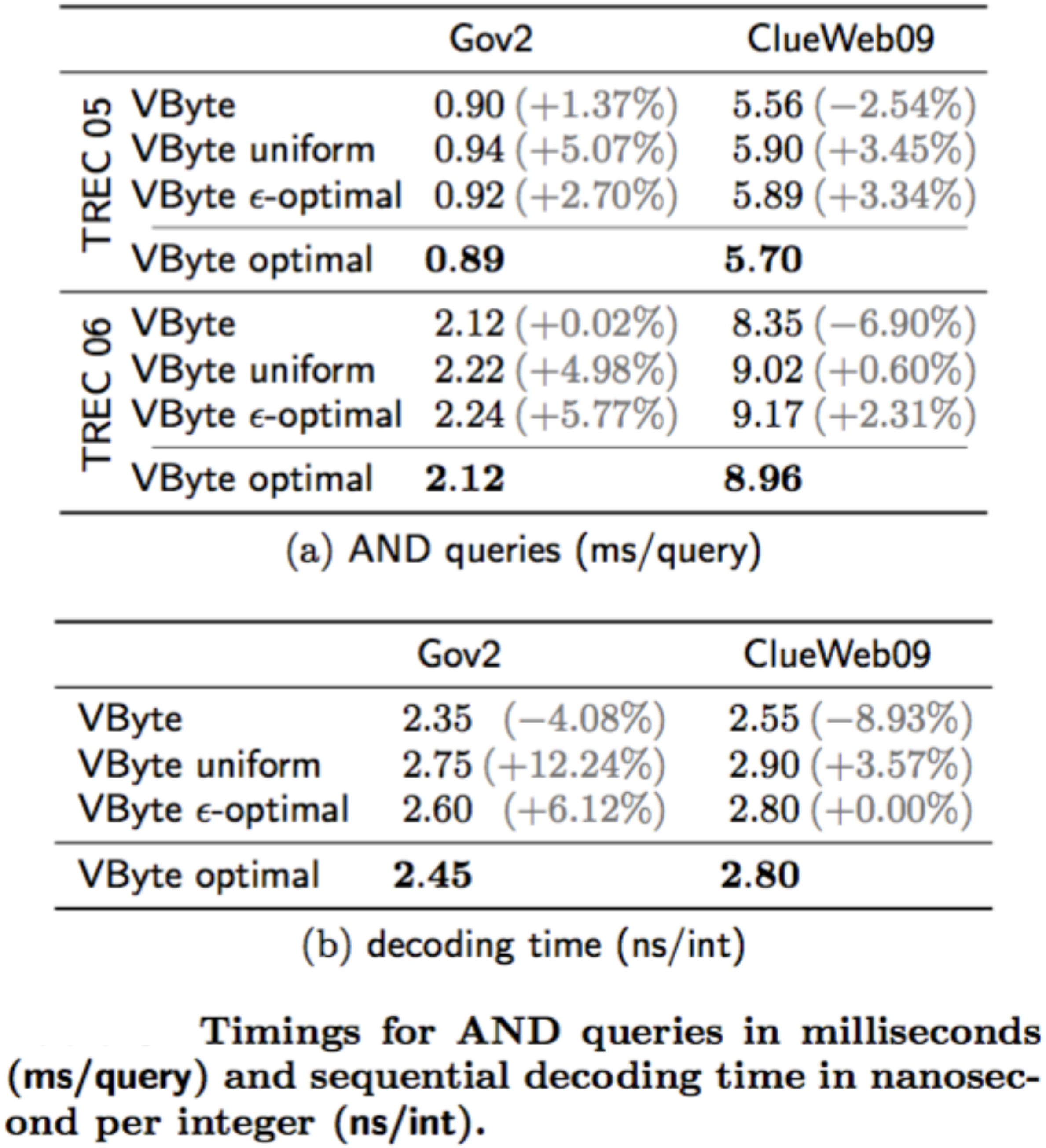
Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



	Gov2	ClueWeb09
VByte	10.10 (−3.81%)	43.30 (+51.93%)
VByte uniform	11.30 (+7.62%)	29.30 (+2.81%)
VByte ϵ-optimal	26.70 (+154.29%)	72.30 (+153.68%)
VByte optimal	**10.50**	**28.50**

Index building timings in minutes.

Partitioned VS. Unpartitioned: AND queries and decoding



		Gov2	ClueWeb09
TREC 05	VByte	0.90 (+1.37%)	5.56 (−2.54%)
	VByte uniform	0.94 (+5.07%)	5.90 (+3.45%)
	VByte ϵ-optimal	0.92 (+2.70%)	5.89 (+3.34%)
	VByte optimal	**0.89**	**5.70**
TREC 06	VByte	2.12 (+0.02%)	8.35 (−6.90%)
	VByte uniform	2.22 (+4.98%)	9.02 (+0.60%)
	VByte ϵ-optimal	2.24 (+5.77%)	9.17 (+2.31%)
	VByte optimal	**2.12**	**8.96**

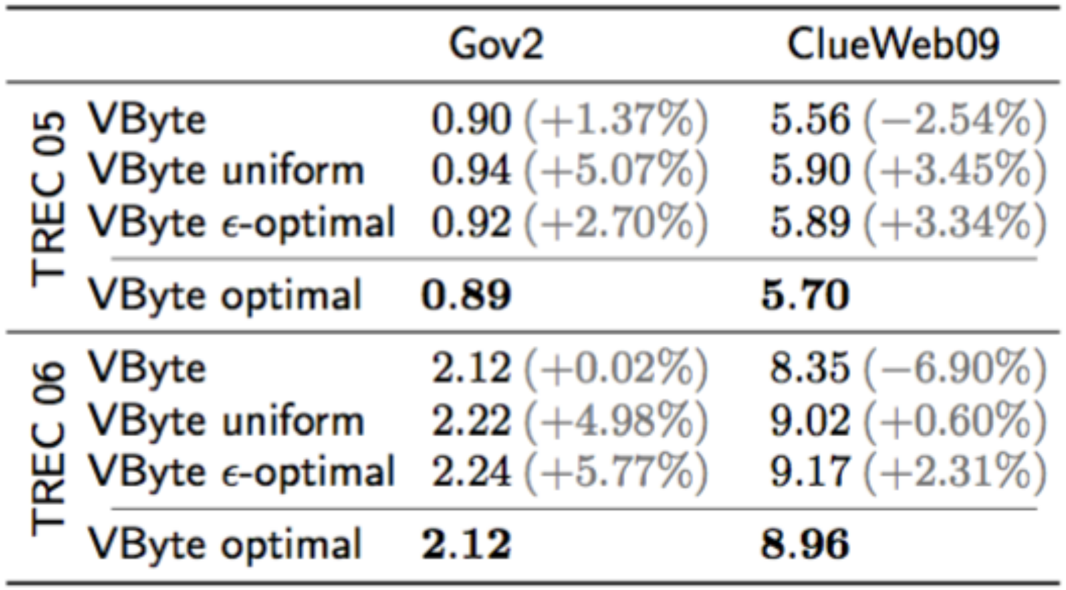
(a) AND queries (ms/query)

	Gov2	ClueWeb09
VByte	2.35 (−4.08%)	2.55 (−8.93%)
VByte uniform	2.75 (+12.24%)	2.90 (+3.57%)
VByte ϵ-optimal	2.60 (+6.12%)	2.80 (+0.00%)
VByte optimal	**2.45**	**2.80**

(b) decoding time (ns/int)

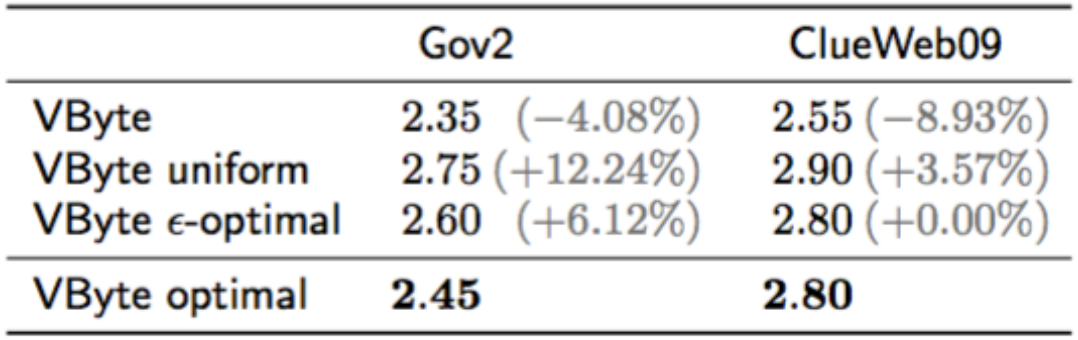
Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

Partitioned VS. Unpartitioned: AND queries and decoding



		Gov2	ClueWeb09
TREC 05	VByte	0.90 (+1.37%)	5.56 (−2.54%)
	VByte uniform	0.94 (+5.07%)	5.90 (+3.45%)
	VByte ϵ-optimal	0.92 (+2.70%)	5.89 (+3.34%)
	VByte optimal	**0.89**	**5.70**
TREC 06	VByte	2.12 (+0.02%)	8.35 (−6.90%)
	VByte uniform	2.22 (+4.98%)	9.02 (+0.60%)
	VByte ϵ-optimal	2.24 (+5.77%)	9.17 (+2.31%)
	VByte optimal	**2.12**	**8.96**

(a) AND queries (ms/query)



	Gov2	ClueWeb09
VByte	2.35 (−4.08%)	2.55 (−8.93%)
VByte uniform	2.75 (+12.24%)	2.90 (+3.57%)
VByte ϵ-optimal	2.60 (+6.12%)	2.80 (+0.00%)
VByte optimal	**2.45**	**2.80**

(b) decoding time (ns/int)

Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

Speed NOT affected by partitioning.

Partitioned VS. Unpartitioned: AND queries and decoding

		Gov2	ClueWeb09
TREC 05	VByte	0.90 (+1.37%)	5.56 (−2.54%)
	VByte uniform	0.94 (+5.07%)	5.90 (+3.45%)
	VByte ϵ-optimal	0.92 (+2.70%)	5.89 (+3.34%)
	VByte optimal	**0.89**	**5.70**
TREC 06	VByte	2.12 (+0.02%)	8.35 (−6.90%)
	VByte uniform	2.22 (+4.98%)	9.02 (+0.60%)
	VByte ϵ-optimal	2.24 (+5.77%)	9.17 (+2.31%)
	VByte optimal	**2.12**	**8.96**

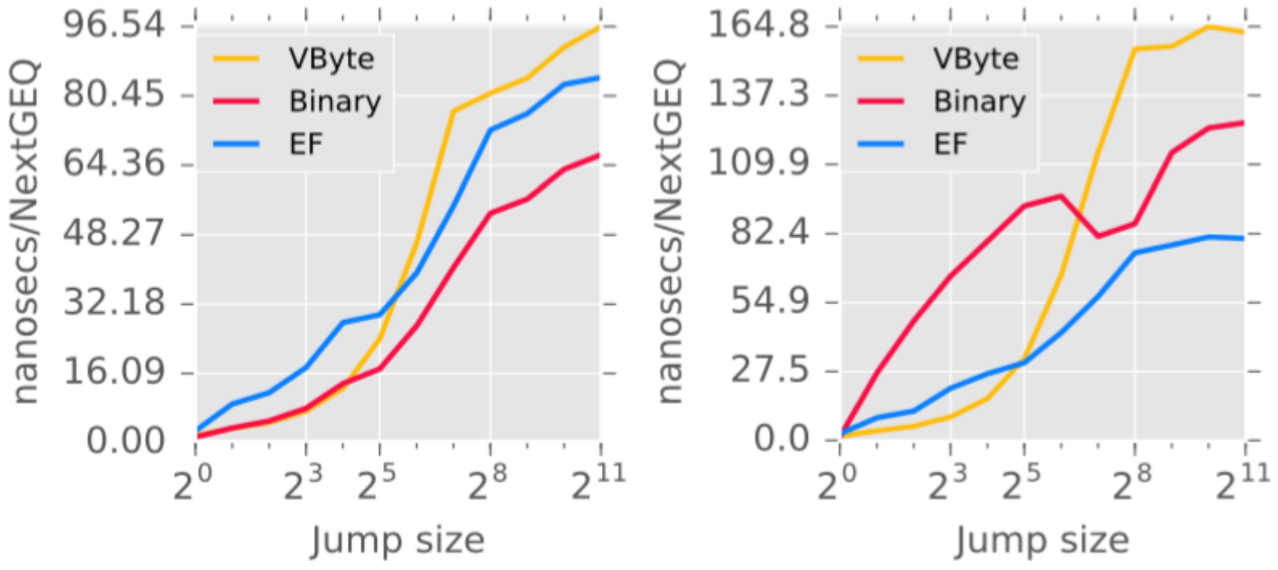
(a) AND queries (ms/query)

	Gov2	ClueWeb09
VByte	2.35 (−4.08%)	2.55 (−8.93%)
VByte uniform	2.75 (+12.24%)	2.90 (+3.57%)
VByte ϵ-optimal	2.60 (+6.12%)	2.80 (+0.00%)
VByte optimal	**2.45**	**2.80**

(b) decoding time (ns/int)

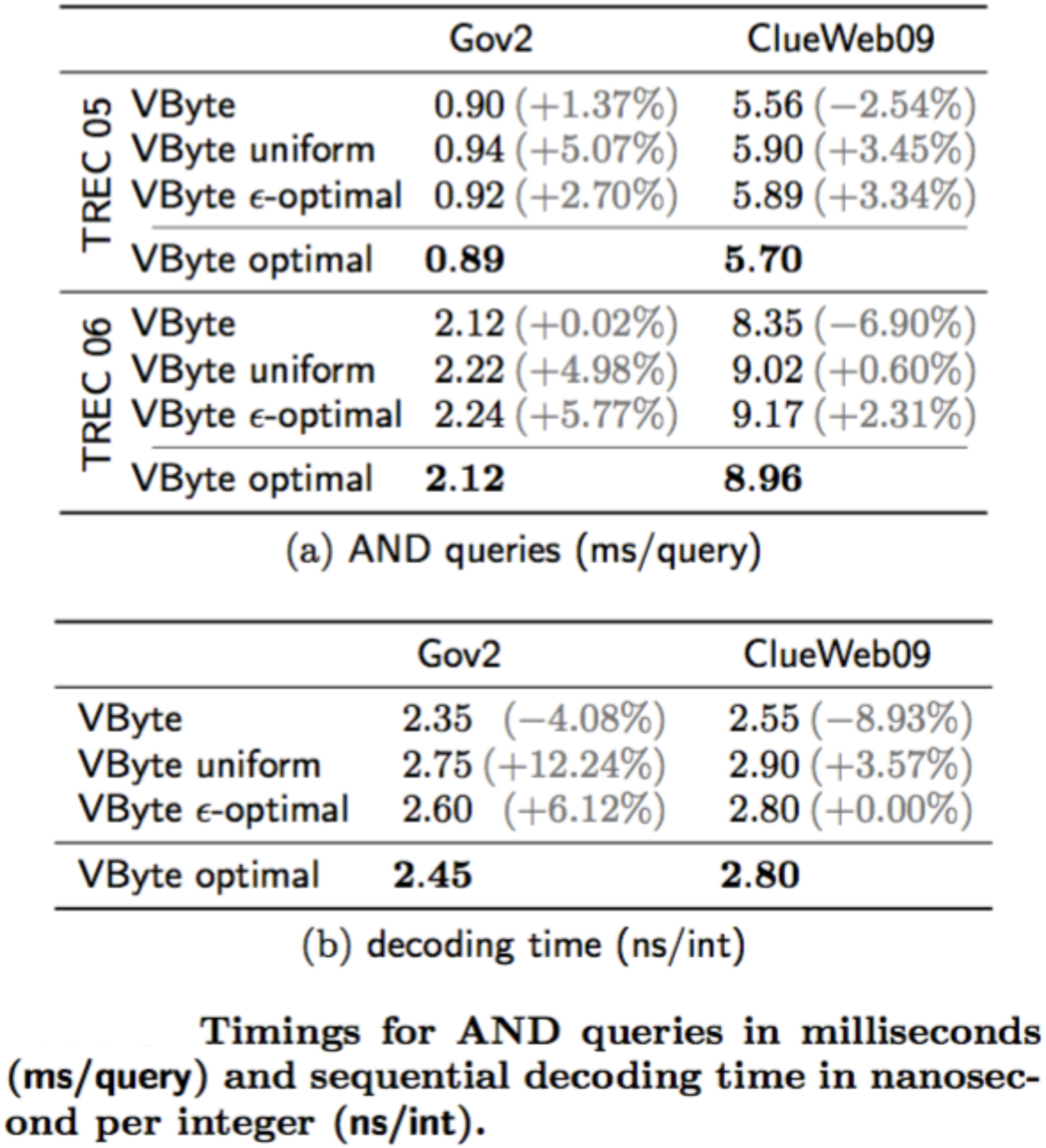
Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

Speed NOT affected by partitioning.



(a) Dense (b) Sparse

Partitioned VS. Unpartitioned: AND queries and decoding



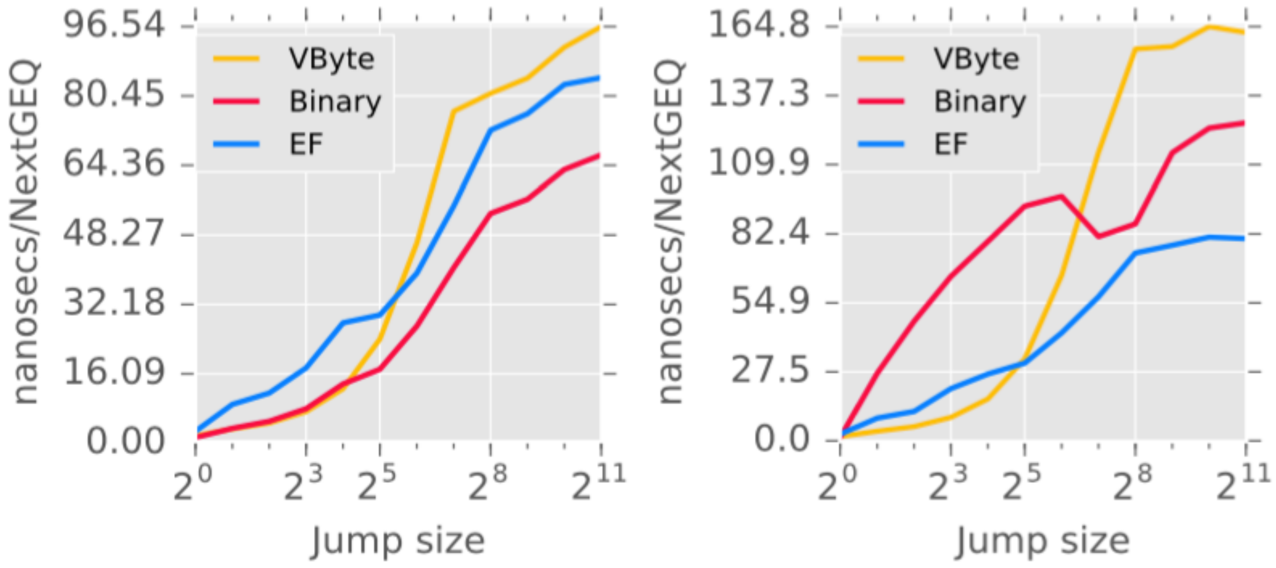
		Gov2	ClueWeb09
TREC 05	VByte	0.90 (+1.37%)	5.56 (−2.54%)
	VByte uniform	0.94 (+5.07%)	5.90 (+3.45%)
	VByte ϵ-optimal	0.92 (+2.70%)	5.89 (+3.34%)
	VByte optimal	**0.89**	**5.70**
TREC 06	VByte	2.12 (+0.02%)	8.35 (−6.90%)
	VByte uniform	2.22 (+4.98%)	9.02 (+0.60%)
	VByte ϵ-optimal	2.24 (+5.77%)	9.17 (+2.31%)
	VByte optimal	**2.12**	**8.96**

(a) AND queries (ms/query)

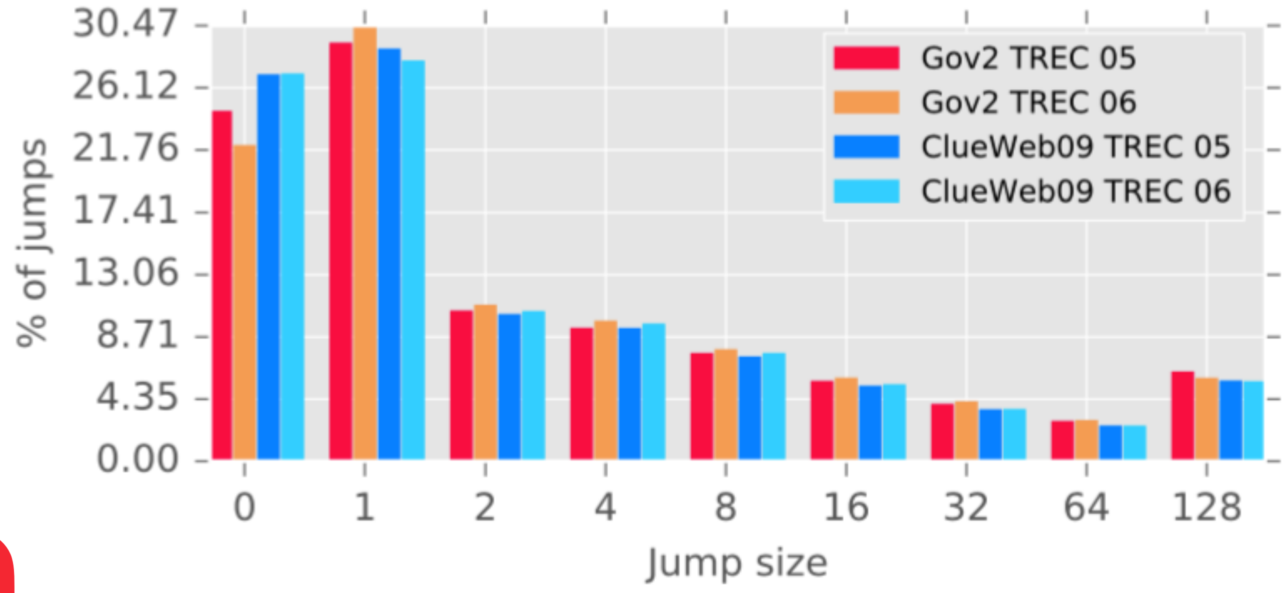
	Gov2	ClueWeb09
VByte	2.35 (−4.08%)	2.55 (−8.93%)
VByte uniform	2.75 (+12.24%)	2.90 (+3.57%)
VByte ϵ-optimal	2.60 (+6.12%)	2.80 (+0.00%)
VByte optimal	**2.45**	**2.80**

(b) decoding time (ns/int)

Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

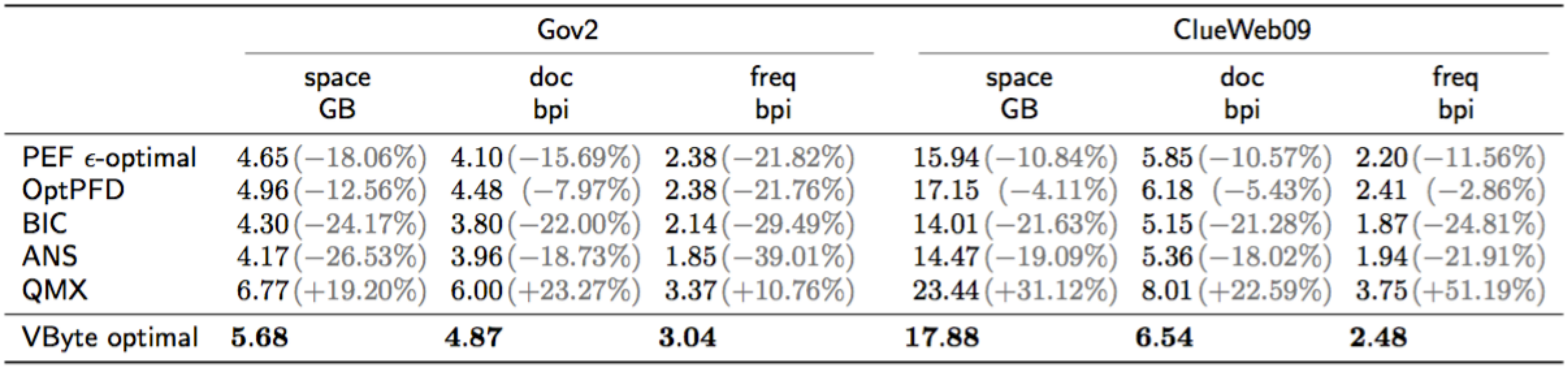


(a) Dense (b) Sparse



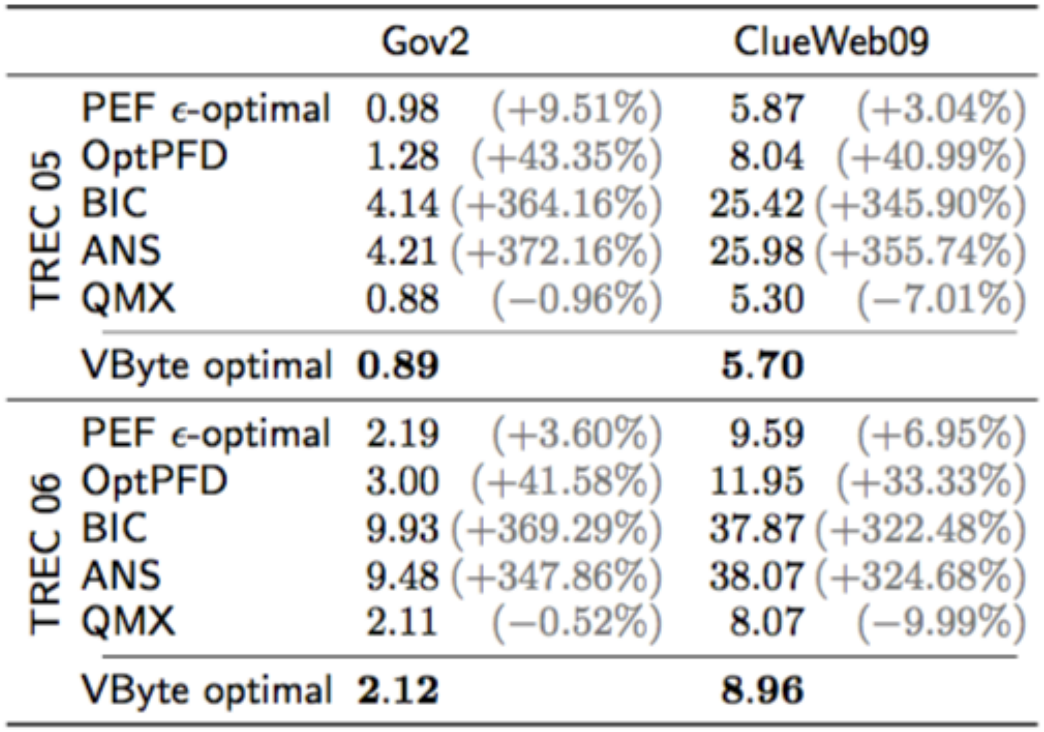
Speed NOT affected by partitioning.

Overall Comparison



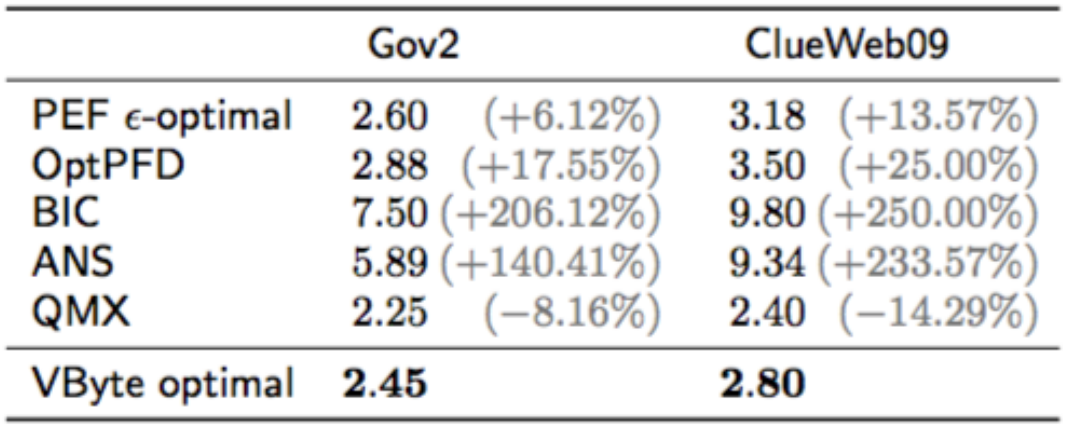
	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
PEF ϵ-optimal	4.65 (−18.06%)	4.10 (−15.69%)	2.38 (−21.82%)	15.94 (−10.84%)	5.85 (−10.57%)	2.20 (−11.56%)
OptPFD	4.96 (−12.56%)	4.48 (−7.97%)	2.38 (−21.76%)	17.15 (−4.11%)	6.18 (−5.43%)	2.41 (−2.86%)
BIC	4.30 (−24.17%)	3.80 (−22.00%)	2.14 (−29.49%)	14.01 (−21.63%)	5.15 (−21.28%)	1.87 (−24.81%)
ANS	4.17 (−26.53%)	3.96 (−18.73%)	1.85 (−39.01%)	14.47 (−19.09%)	5.36 (−18.02%)	1.94 (−21.91%)
QMX	6.77 (+19.20%)	6.00 (+23.27%)	3.37 (+10.76%)	23.44 (+31.12%)	8.01 (+22.59%)	3.75 (+51.19%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



		Gov2	ClueWeb09
TREC 05	PEF ϵ-optimal	0.98 (+9.51%)	5.87 (+3.04%)
	OptPFD	1.28 (+43.35%)	8.04 (+40.99%)
	BIC	4.14 (+364.16%)	25.42 (+345.90%)
	ANS	4.21 (+372.16%)	25.98 (+355.74%)
	QMX	0.88 (−0.96%)	5.30 (−7.01%)
	VByte optimal	**0.89**	**5.70**
TREC 06	PEF ϵ-optimal	2.19 (+3.60%)	9.59 (+6.95%)
	OptPFD	3.00 (+41.58%)	11.95 (+33.33%)
	BIC	9.93 (+369.29%)	37.87 (+322.48%)
	ANS	9.48 (+347.86%)	38.07 (+324.68%)
	QMX	2.11 (−0.52%)	8.07 (−9.99%)
	VByte optimal	**2.12**	**8.96**

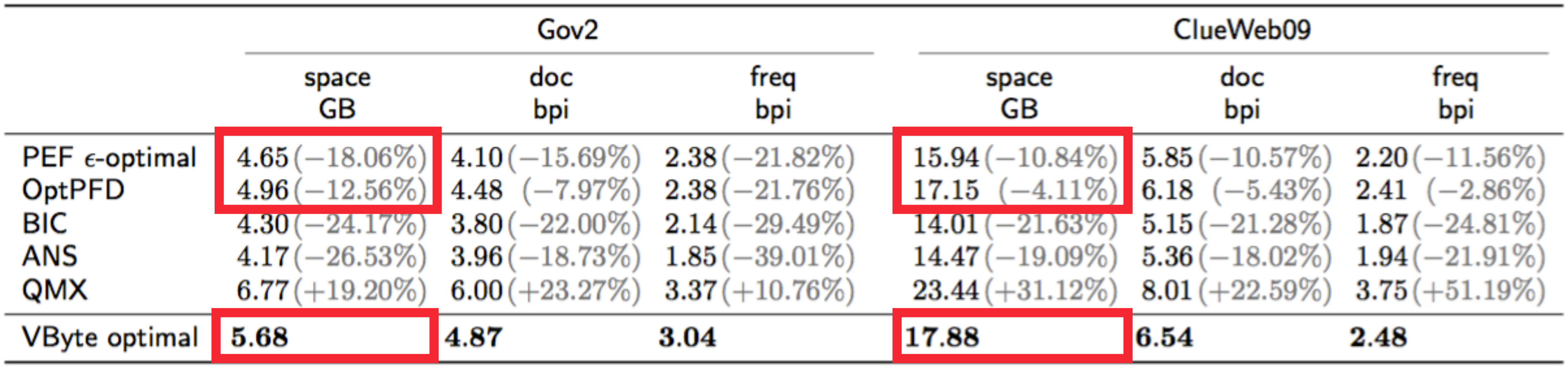
(a) AND queries (ms/query)



	Gov2	ClueWeb09
PEF ϵ-optimal	2.60 (+6.12%)	3.18 (+13.57%)
OptPFD	2.88 (+17.55%)	3.50 (+25.00%)
BIC	7.50 (+206.12%)	9.80 (+250.00%)
ANS	5.89 (+140.41%)	9.34 (+233.57%)
QMX	2.25 (−8.16%)	2.40 (−14.29%)
VByte optimal	**2.45**	**2.80**

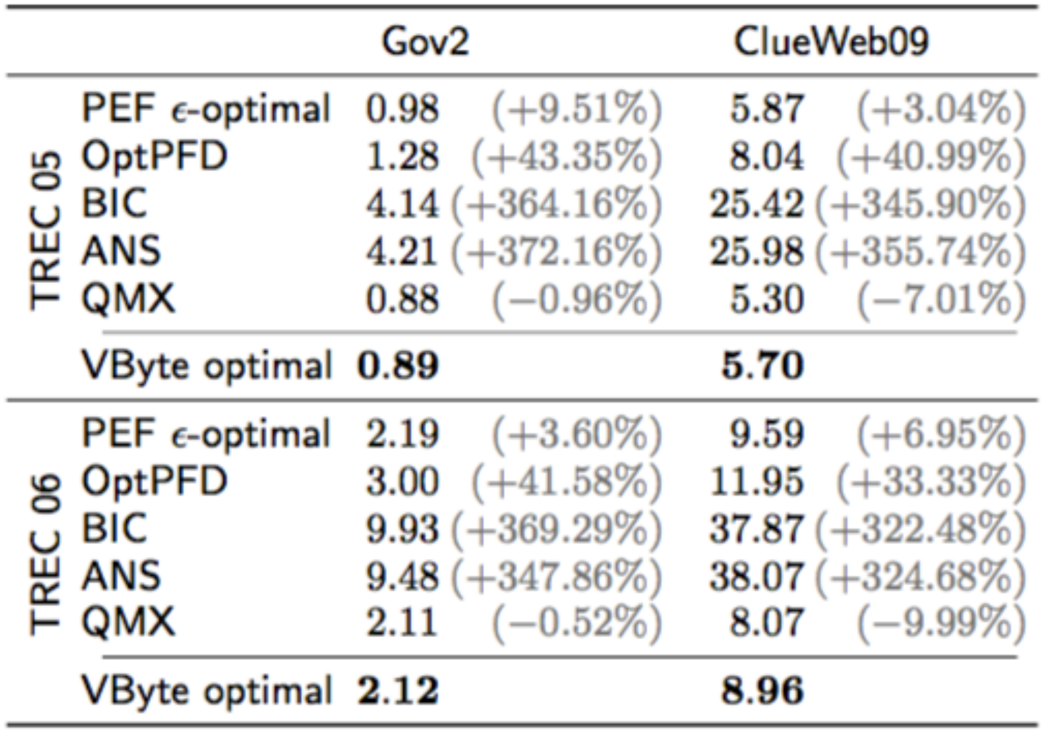
(b) decoding time (ns/int)

Overall Comparison



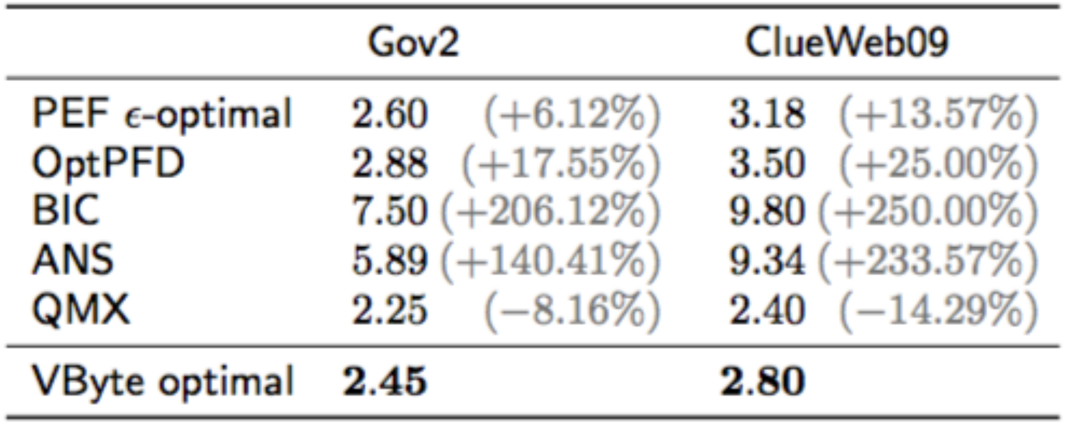
	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
PEF ϵ-optimal	4.65 (−18.06%)	4.10 (−15.69%)	2.38 (−21.82%)	15.94 (−10.84%)	5.85 (−10.57%)	2.20 (−11.56%)
OptPFD	4.96 (−12.56%)	4.48 (−7.97%)	2.38 (−21.76%)	17.15 (−4.11%)	6.18 (−5.43%)	2.41 (−2.86%)
BIC	4.30 (−24.17%)	3.80 (−22.00%)	2.14 (−29.49%)	14.01 (−21.63%)	5.15 (−21.28%)	1.87 (−24.81%)
ANS	4.17 (−26.53%)	3.96 (−18.73%)	1.85 (−39.01%)	14.47 (−19.09%)	5.36 (−18.02%)	1.94 (−21.91%)
QMX	6.77 (+19.20%)	6.00 (+23.27%)	3.37 (+10.76%)	23.44 (+31.12%)	8.01 (+22.59%)	3.75 (+51.19%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



		Gov2	ClueWeb09
TREC 05	PEF ϵ-optimal	0.98 (+9.51%)	5.87 (+3.04%)
	OptPFD	1.28 (+43.35%)	8.04 (+40.99%)
	BIC	4.14 (+364.16%)	25.42 (+345.90%)
	ANS	4.21 (+372.16%)	25.98 (+355.74%)
	QMX	0.88 (−0.96%)	5.30 (−7.01%)
	VByte optimal	**0.89**	**5.70**
TREC 06	PEF ϵ-optimal	2.19 (+3.60%)	9.59 (+6.95%)
	OptPFD	3.00 (+41.58%)	11.95 (+33.33%)
	BIC	9.93 (+369.29%)	37.87 (+322.48%)
	ANS	9.48 (+347.86%)	38.07 (+324.68%)
	QMX	2.11 (−0.52%)	8.07 (−9.99%)
	VByte optimal	**2.12**	**8.96**

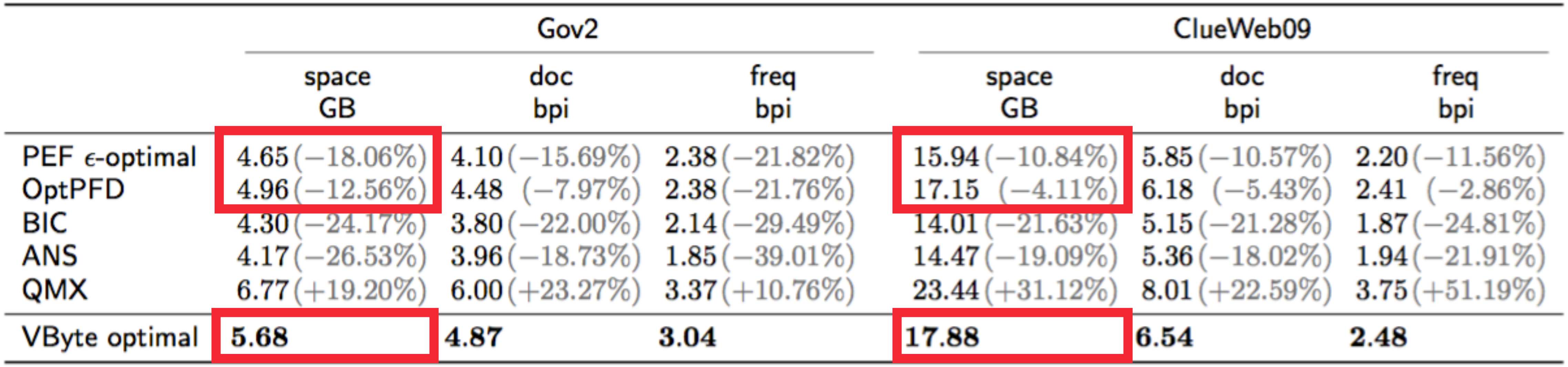
(a) AND queries (ms/query)



	Gov2	ClueWeb09
PEF ϵ-optimal	2.60 (+6.12%)	3.18 (+13.57%)
OptPFD	2.88 (+17.55%)	3.50 (+25.00%)
BIC	7.50 (+206.12%)	9.80 (+250.00%)
ANS	5.89 (+140.41%)	9.34 (+233.57%)
QMX	2.25 (−8.16%)	2.40 (−14.29%)
VByte optimal	**2.45**	**2.80**

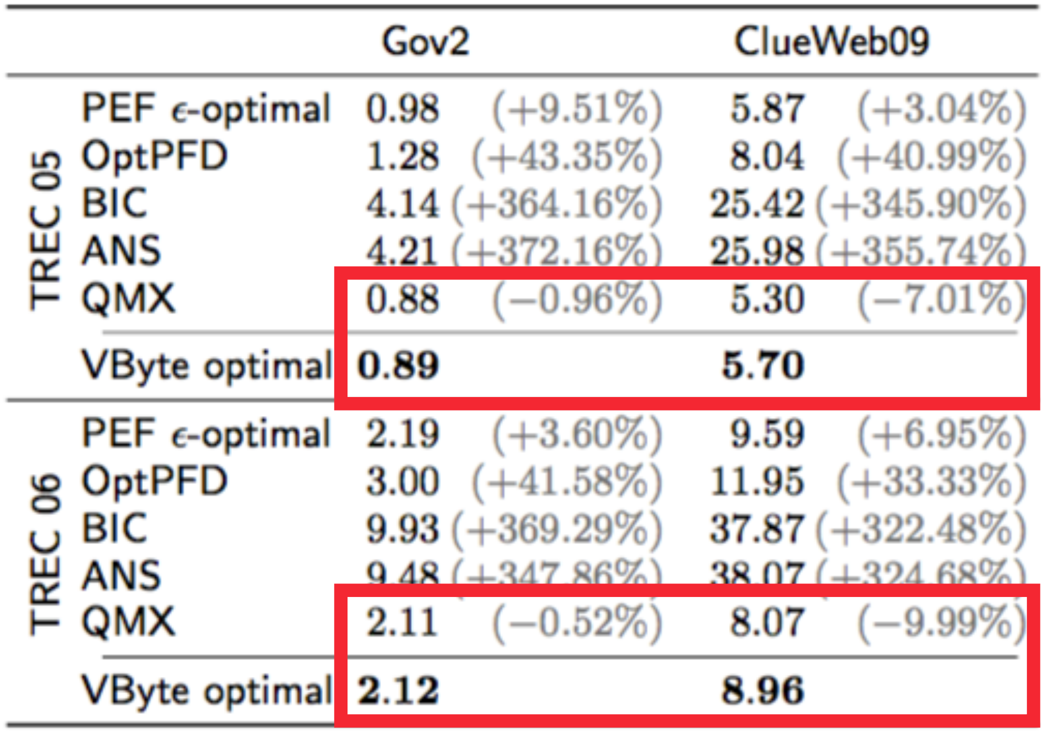
(b) decoding time (ns/int)

Overall Comparison



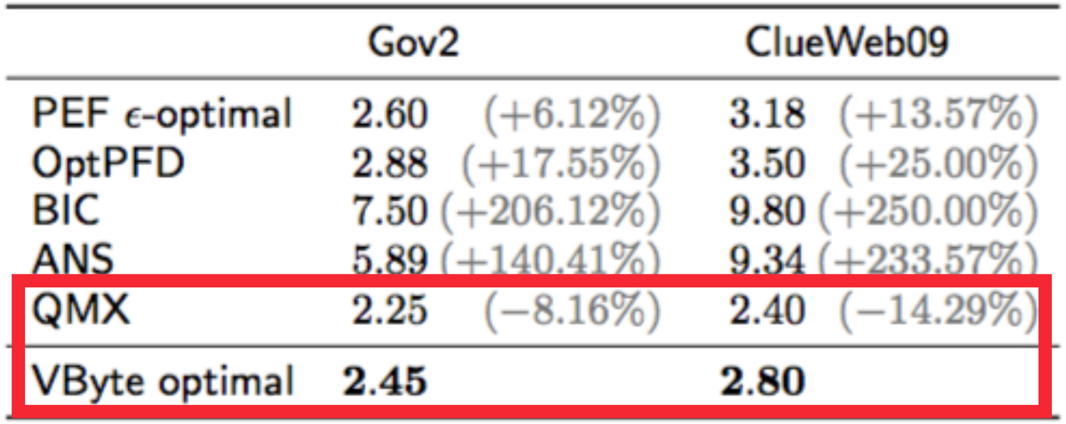
	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
PEF ϵ-optimal	4.65 (−18.06%)	4.10 (−15.69%)	2.38 (−21.82%)	15.94 (−10.84%)	5.85 (−10.57%)	2.20 (−11.56%)
OptPFD	4.96 (−12.56%)	4.48 (−7.97%)	2.38 (−21.76%)	17.15 (−4.11%)	6.18 (−5.43%)	2.41 (−2.86%)
BIC	4.30 (−24.17%)	3.80 (−22.00%)	2.14 (−29.49%)	14.01 (−21.63%)	5.15 (−21.28%)	1.87 (−24.81%)
ANS	4.17 (−26.53%)	3.96 (−18.73%)	1.85 (−39.01%)	14.47 (−19.09%)	5.36 (−18.02%)	1.94 (−21.91%)
QMX	6.77 (+19.20%)	6.00 (+23.27%)	3.37 (+10.76%)	23.44 (+31.12%)	8.01 (+22.59%)	3.75 (+51.19%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



		Gov2	ClueWeb09
TREC 05	PEF ϵ-optimal	0.98 (+9.51%)	5.87 (+3.04%)
	OptPFD	1.28 (+43.35%)	8.04 (+40.99%)
	BIC	4.14 (+364.16%)	25.42 (+345.90%)
	ANS	4.21 (+372.16%)	25.98 (+355.74%)
	QMX	0.88 (−0.96%)	5.30 (−7.01%)
	VByte optimal	**0.89**	**5.70**
TREC 06	PEF ϵ-optimal	2.19 (+3.60%)	9.59 (+6.95%)
	OptPFD	3.00 (+41.58%)	11.95 (+33.33%)
	BIC	9.93 (+369.29%)	37.87 (+322.48%)
	ANS	9.48 (+347.86%)	38.07 (+324.68%)
	QMX	2.11 (−0.52%)	8.07 (−9.99%)
	VByte optimal	**2.12**	**8.96**

(a) AND queries (ms/query)



	Gov2	ClueWeb09
PEF ϵ-optimal	2.60 (+6.12%)	3.18 (+13.57%)
OptPFD	2.88 (+17.55%)	3.50 (+25.00%)
BIC	7.50 (+206.12%)	9.80 (+250.00%)
ANS	5.89 (+140.41%)	9.34 (+233.57%)
QMX	2.25 (−8.16%)	2.40 (−14.29%)
VByte optimal	**2.45**	**2.80**

(b) decoding time (ns/int)

Take-home messages

Just **do not waste space** with VByte:
partition the sequences!

Compression ratio is likely to improve a lot,
without affecting speed.

The partitioning algorithm is **fast**, **optimal**,
and makes indexing even more efficient.

Thanks for your attention, time, patience!

Any questions?

Problem

Scale with the quantity of indexed data.

(Some people would say: “Big Data”.)

Dataset	Uncompressed	Compressed
Gov2	46GB	**4GB (~11X)**
ClueWeb09	128GB	**14GB (~9X)**

Problem

Scale with the quantity of indexed data.
(Some people would say: “Big Data”.)

Dataset	Uncompressed	Compressed
Gov2	46GB	**4GB (~11X)**
ClueWeb09	128GB	**14GB (~9X)**

Can we put everything on disk ?

Memory hierarchy!

Problem

Scale with the quantity of indexed data.

(Some people would say: “Big Data”.)

Dataset	Uncompressed	Compressed
Gov2	46GB	**4GB (~11X)**
ClueWeb09	128GB	**14GB (~9X)**

Can we put everything on disk ?

Memory hierarchy!

See, for example:

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/

Problem

Scale with the quantity of indexed data.
(Some people would say: “Big Data”.)

Dataset	Uncompressed	Compressed
Gov2	46GB	**4GB (~11X)**
ClueWeb09	128GB	**14GB (~9X)**

Can we put everything on disk ?

Memory hierarchy!

See, for example:

https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/