Index compression and efficient query processing

COMP90042 LECTURE 17, THE UNIVERSITY OF MELBOURNE

by

Matthias Petri

Wed 2/5/2018

Index compression

Inverted Index - Recap

term t	f_t	Postings list for t
and	6	$\langle 1, 6, 7, 8, 9, 12 \rangle$, $\langle 1, 2, 1, 3, 1, 2 \rangle$
big	3	$\langle 2, 5, 42 \rangle$, $\langle 1, 1, 1 \rangle$
old	1	$\langle 32 \rangle$, $\langle 4 \rangle$
in	7	$\langle 2, 3, 5, 6, 8, 14, 25 \rangle$, $\langle 1, 1, 4, 1, 5, 3, 1 \rangle$
the	52	$\langle 1, 2, 3, 4, 5, 7, 8, 9, \ldots \rangle$, $\langle 10, 21, 10, 42, 12, 14, 12, 4, \ldots \rangle$
night	4	$\langle 1, 12, 13, 14 \rangle$, $\langle 2, 2, 1, 3 \rangle$
house	5	$\langle 6, 21, 32, 33, 43 \rangle$, $\langle 2, 3, 4, 2, 1 \rangle$
sleep	3	$\langle 1, 51, 53 \rangle$, $\langle 1, 2, 3 \rangle$
where	4	$\langle 1, 3, 4, 6 \rangle$, $\langle 1, 1, 2, 1 \rangle$

Index Compression - Motivation

Inverted Index size for 420GB of web data (tiny)

Documents	25 Million
Terms	35 Million
Postings	6 Billion
Uncompressed Storage Cost	≈ 32 GB

- Inverted Index mostly stored in RAM (query performance)
- Companies run 1000s of machines to answer search queries
- Space reduction can lead to substantial cost reductions
- Saving 5% means shutting down 50/1000 machines!

Index compression

Benefits of index compression:

- Reduce storage requirements
- Keep larger parts of the index in memory
- Faster query processing

Example

A state-of-the-art inverted index of 25 million websites (420GB) requires only 5GB (1.2%) and can answer queries in ≈ 10 milliseconds.

32GB $\rightarrow$ 5GB corresponds to a 700% space reduction!

Compression Principles

- Compressibility is bounded by the information content of a data set
- Information content of a text T is a characterized by its *Entropy* H:

$$H(T) = -\sum_{s \in \Sigma} \frac{f_s}{n} \log_2 \frac{f_s}{n}$$

  where f_s is the frequency of symbol s in T and n is the length of T.
- For example, $H(\mathit{abracadabra}) = 2.040373$ bits with $n = 11, f_a = 5, f_b = 2, f_c = 1, f_d = 1, f_r = 2$.
- Intuition: Spend less bits on items that occur often.

Posting list Compression

- Minimize storage costs
- Fast sequential access
- Support GEQ (x) operation: Return the smallest item in the list that is greater or equal to x

Posting list Compression - Concepts

- Postings list corresponds to an increasing sequence of integers
- Each integer can be in $[1, N]$ requiring $\log_2(N)$ bits
- Idea: Gaps between two adjacent integers can be much smaller

the	ids:	25	26	29	...	12345	12347
	gaps:	25	1	3	...	1	2
house	ids:	5123	5234	5454	5591	...	
	gaps:	5123	1	220	137	...	
aeronaut	ids:	251235	251239	251239			
	gaps:	251235	4	34			

Variable Byte Compression

Idea

Use variable number of bytes to represent integers. Each byte contains 7 bits "payload" and one continuation bit.

Examples

Number	Encoding	
824	00000110	10111000
5	10000101	

Storage Cost

Number Range	Number of Bytes
$0 - 127$	1
$128 - 16383$	2
$16384 - 2097151$	3

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes?

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$

Variable Byte Compression - Example

Compress number 512312 or $11111010001001110 00$ in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)
  3. Extract the lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)
  3. Extract the lowest 7 bits.
     $4002 \mod 128 = 34 = 0100010$

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)
  3. Extract the lowest 7 bits.
     $4002 \mod 128 = 34 = 0100010$
  4. Discard lowest 7 bits.
     $4002 \div 128 = 31$ (or $4002 >> 7$)

Variable Byte Compression - Example

Compress number 512312 or $11111010001001110 00$ in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)
  3. Extract the lowest 7 bits.
     $4002 \mod 128 = 34 = 0100010$
  4. Discard lowest 7 bits.
     $4002 \div 128 = 31$ (or $4002 >> 7$)
  5. Number smaller than 128. Write in lowest 7 bits and set top bit to 1. $31 = 11111$ So we write 10011111 which is $31 + 128 = 159$

Variable Byte Compression - Example

Compress number 512312 or $11111010001001110 00$ in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.
     $512312 \div 128 = 4002$ (or $512312 >> 7$)
  3. Extract the lowest 7 bits.
     $4002 \mod 128 = 34 = 0100010$
  4. Discard lowest 7 bits.
     $4002 \div 128 = 31$ (or $4002 >> 7$)
  5. Number smaller than 128. Write in lowest 7 bits and set top bit to 1. $31 = 11111$ So we write 10011111 which is $31 + 128 = 159$

Variable Byte - Algorithm

Encoding

```
1: function ENCODE(x)
2:     while x >= 128 do
3:         WRITE(x mod 128)
4:         x = x ÷ 128
5:     end while
6:     WRITE(x + 128)
7: end function
```

Decoding

```
1: function DECODE(bytes)
2:     x = 0
3:     y =READBYTE(bytes)
4:     while y < 128 do
5:         x = 128 × x + y
6:         y =READBYTE(bytes)
7:     end while
8:     x = 128 × x + (y − 128)
9:     return x
10: end function
```

OptPForDelta Compression

Idea

Group k gaps and encode using fixed number of bits. Encode numbers $> 2^b$ separately as an *exception. Pick b "optimally" for each block so there are* $\approx 10\%$ *exceptions.*

Example $k = 8$

`[1 4 7 2 4 5 123 6] [3 4 755 15 12 1 8 4]`

b=3 b=4

Encode `[1 4 7 2 4 5 123 6]` as:

header content exceptions

16 bit 21 bit 8 bit

Decompression Speeds/Space Usage

Algorithm	Space [bits/int]	Speed [Million Integers/sec]
Uncompressed	32	≈ 5400
Variable Byte	8.7	≈ 680
OptPForDelta	4.7	≈ 710
Simple-8b	4.8	≈ 780
SIMD-BP128	11	≈ 2300
$\vdots$	$\vdots$	

Citation: Daniel Lemire, Leonid Boytsov: Decoding billions of integers per second through vectorization. Softw., Pract. Exper. 45(1): 1-29 (2015)

Compression Optimizations

- Commonly lists are split into blocks of 128 integers
- Choose optimal compression for each block
- Often, long lists (“the”) are represented more efficiently using bitvectors of size *N*
- State-of-the-art implementations use SIMD Instructions and bit-parallelism to increase decoding speed

List Compression - Fast Searching

- Compress postings list in blocks of 128 integers at a time
- For each block store an uncompressed sample value representing the largest (or smallest) value in the block
- Use sample values to efficiently seek to any position in the postings list without decompressing everything

Operation GEQ (x) (Greater or Equal *x*):

- Binary search over uncompressed sample values to find destination block
- Decompress destination block to determine final offset in postings list

Postings Compression - Summary

- Compress increasing integer sequences
- Support iterating and searching the compressed sequence
- Store gaps between adjacent numbers
- Different compression schemes provide different time-space tradeoffs

Efficient Query Processing

Query Processing - Motivation

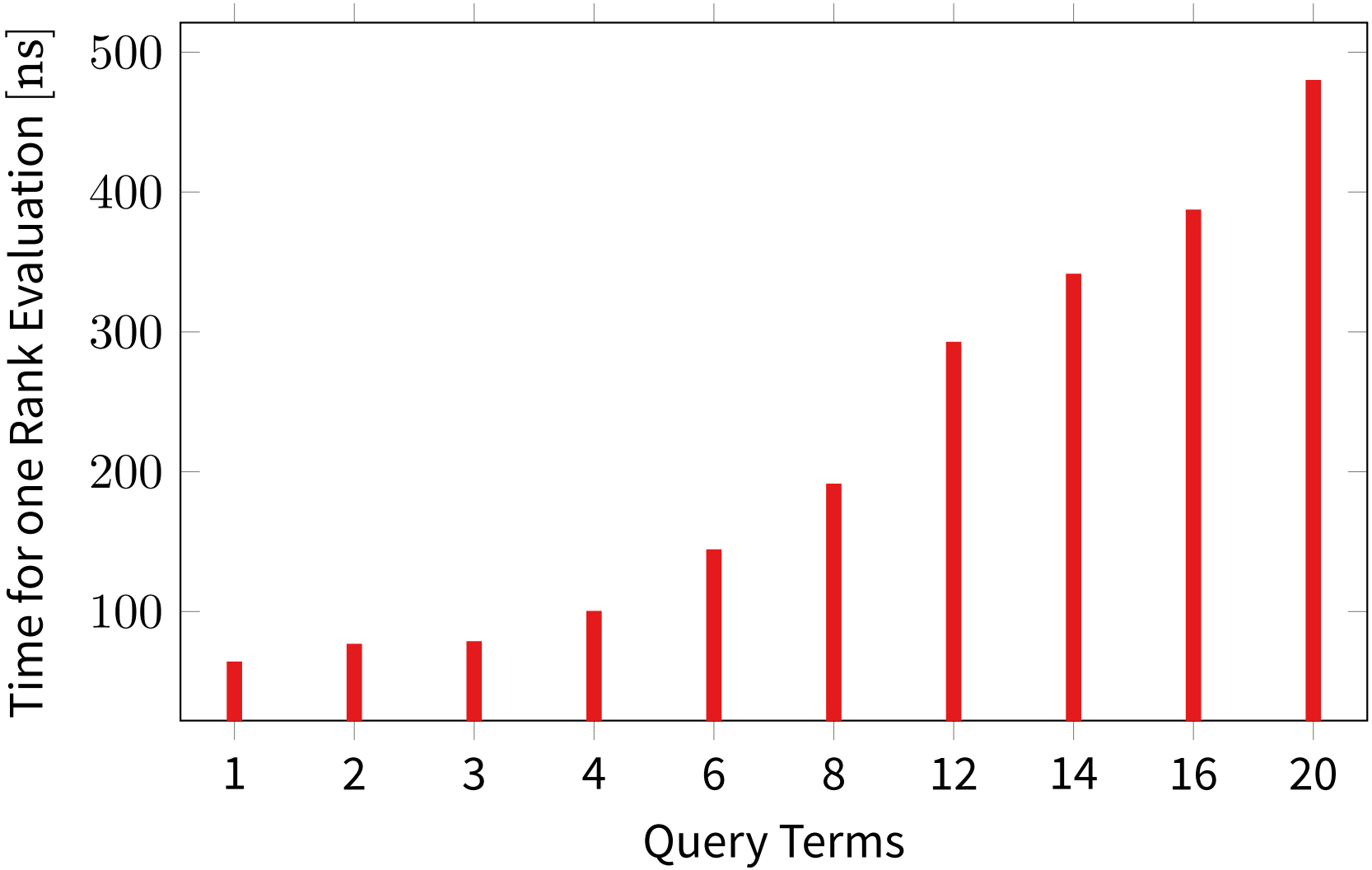
BM25 computation for one document:

$$S_{Q,d}^{\text{BM25}} = \sum_{q \in Q} \frac{(k_1 + 1) f_{d,q}}{k_1 \left(1 - b + b \frac{n_d}{n_{\text{avg}}}\right) + f_{d,q}} \cdot f_{Q,q} \cdot \ln\left(\frac{N - F_{\mathcal{D},q} + 0.5}{F_{\mathcal{D},q} + 0.5}\right)$$

Symbol	Description
Q	Multi set of query terms
q	Query term in Q
d	Evaluated document
$f_{d,q}$	Frequency of q in d (Term frequency)
N	Number of documents in the collection
$F_{\mathcal{D},q}$	Number of documents containing q (Document frequency)
$f_{Q,q}$	Number of times q occurs in Q
n_d	Length of document d
n_{avg}	Average document length in the collection
k_1, b	Constants

Query Processing - Motivation II

BM25 computation speed:



Query Processing - Motivation III

- Evaluating BM25 for one document takes 100 nanoseconds (Fast!).
- Assume query matches 25 million documents (term “the” contained in almost all documents).
- 25 million $\times 100$ nanoseconds ≈ 2.5 seconds.
- Is waiting 2.5 seconds for a search query acceptable?

Query Processing - Motivation IV

- Google A/B tested the effect of latency on user satisfaction.[1]
  - Users are able to detect changes in latency by only 50 ms.
  - Users searched less when results took longer to be presented.
  - System abandonment was higher when results took longer to be presented.
  - Average revenue per user dropped by $\approx 4\%$ when users had delayed results.
- Every 100 ms boost in latency increases annual revenue for Bing by $\approx 0.6\%$.[2]
  - In 2015, a 1% improvement in revenue per user at Bing was an increase of *tens-of-millions* of dollars per year.

[1] E. Schurman and J. Brutlag. "Performance related changes and their user impact". Velocity, 2009.

[2] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. "Online controlled experiments at large scale". In Proc. KDD, pages 1168–1176, 2013.

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.
- Scaling *up:* Why not add better hardware?

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.
- Scaling *up:* Why not add better hardware?
  - Servers become expensive to purchase.

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.
- Scaling *up:* Why not add better hardware?
  - Servers become expensive to purchase.
  - Cannot scale infinitely: diminishing returns.

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.
- Scaling *up:* Why not add better hardware?
  - Servers become expensive to purchase.
  - Cannot scale infinitely: diminishing returns.
- Many improvements can still be made at the *software* level.

Query Processing – Strategies

- Scaling *out:* Why not just add more servers?
  - Servers are somewhat cheap, but more servers means more maintance, a more complex system, and larger power costs.
- Scaling *up:* Why not add better hardware?
  - Servers become expensive to purchase.
  - Cannot scale infinitely: diminishing returns.
- Many improvements can still be made at the *software* level.
  - **Faster, cheaper algorithms for query processing.**

Efficient Query Processing

IDEA: Top-*k* Retrieval

Retrieve the top *k* items for a given query without having to evaluate all documents.

- Web search engines return only the top-10 results to the user.
- For most queries most users generally do not retrieve more documents.
- No need to score all possible documents to produce a "complete" ranking.

Top-k Retrieval - Concepts

- Avoid scoring documents we know will not appear in the top-k result list.
- For a given similarity metric (for example BM25), prestore some information for each term to avoid scoring.
- Incorporate with block based compression schemes for efficient query processing.
- Utilise the $\mathrm{GEQ}(x)$ operation to avoid decompression of large parts of postings lists.

Inefficient Evaluation of BM25

BM25 computation for one document:

$$S^{\text{BM25}}_{Q,d} = \sum_{q \in Q} \underbrace{\frac{(k_1+1) f_{d,q}}{k_1 \left(1 - b + b \frac{n_d}{n_{\text{avg}}}\right) + f_{d,q}}}_{=w_{d,q}} \cdot \underbrace{f_{Q,q} \cdot \ln\left(\frac{N - F_{\mathcal{D},q} + 0.5}{F_{\mathcal{D},q} + 0.5}\right)}_{=w_{Q,q}}$$

Inefficient Evaluation:

- For each q in Q compute $w_{Q,q}$ in $O(|Q|)$ time.
- For each d in the document collection containing any q in Q evaluate $w_{d,q}$. (Potentially $O(N)$ time!)
- Return the top-k highest scoring documents.

Top-*k* - The WAND Algorithm

Basic Idea:

- Keep track of the top-*k* highest scored documents.
- For each unique term in the collection store the **maximum contribution** it can have to *any* document score in the collection.
- Skip over documents that can not enter the top-*k* highest results.

Citation: Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, Jason Y. Zien: Efficient query evaluation using a two-level retrieval process. CIKM: 426-434 (2003)

WAND - Maximum term contribution

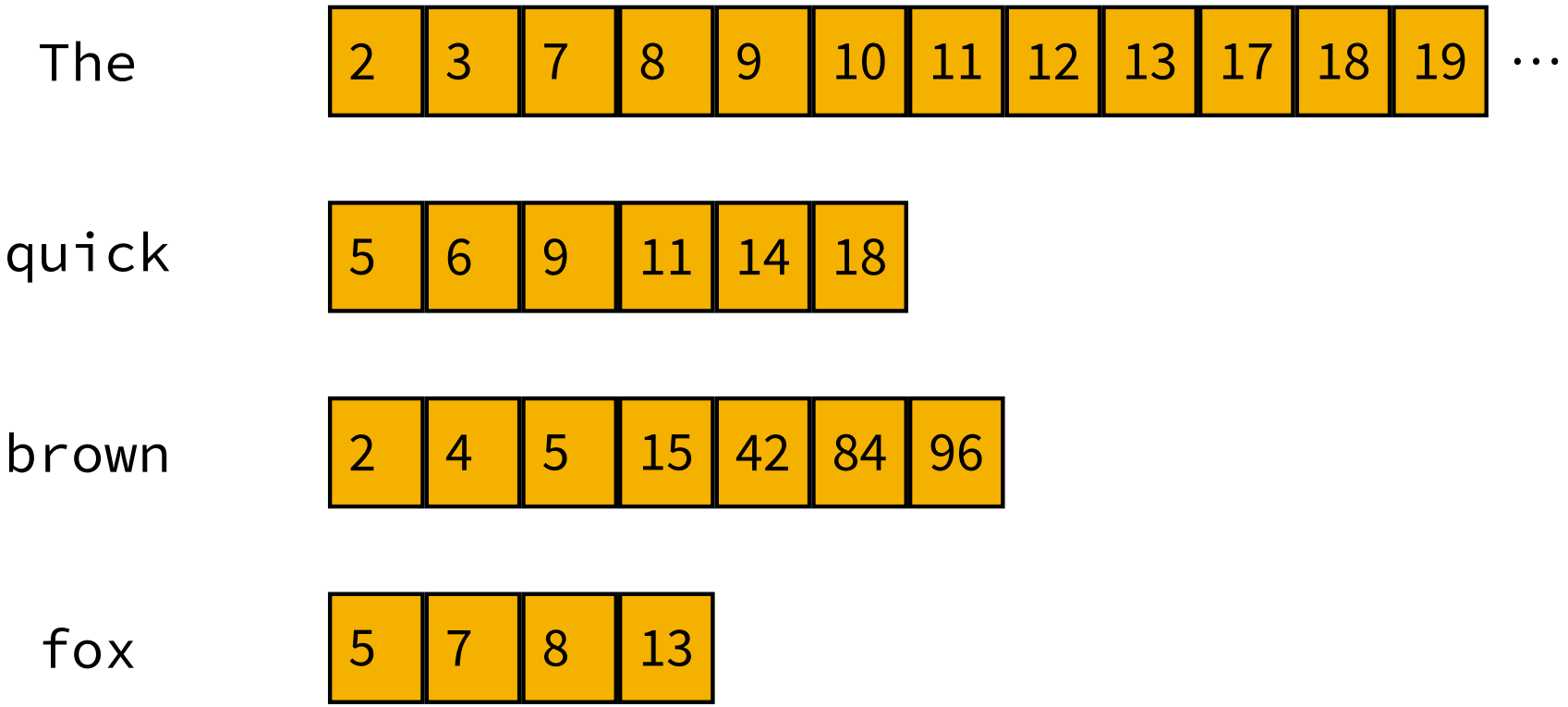
Maximum contribution

The **Maximum contribution** of a term q as the largest score **any** document in the collection can have for the query Q only consisting of q.

- Depends on the similarity measure.
- Can be computed at construction time of the index.
- Only requires storing a single floating point number for each list.
- Can be used to **overestimate** the score of a document in a multi term query.

WAND - Example

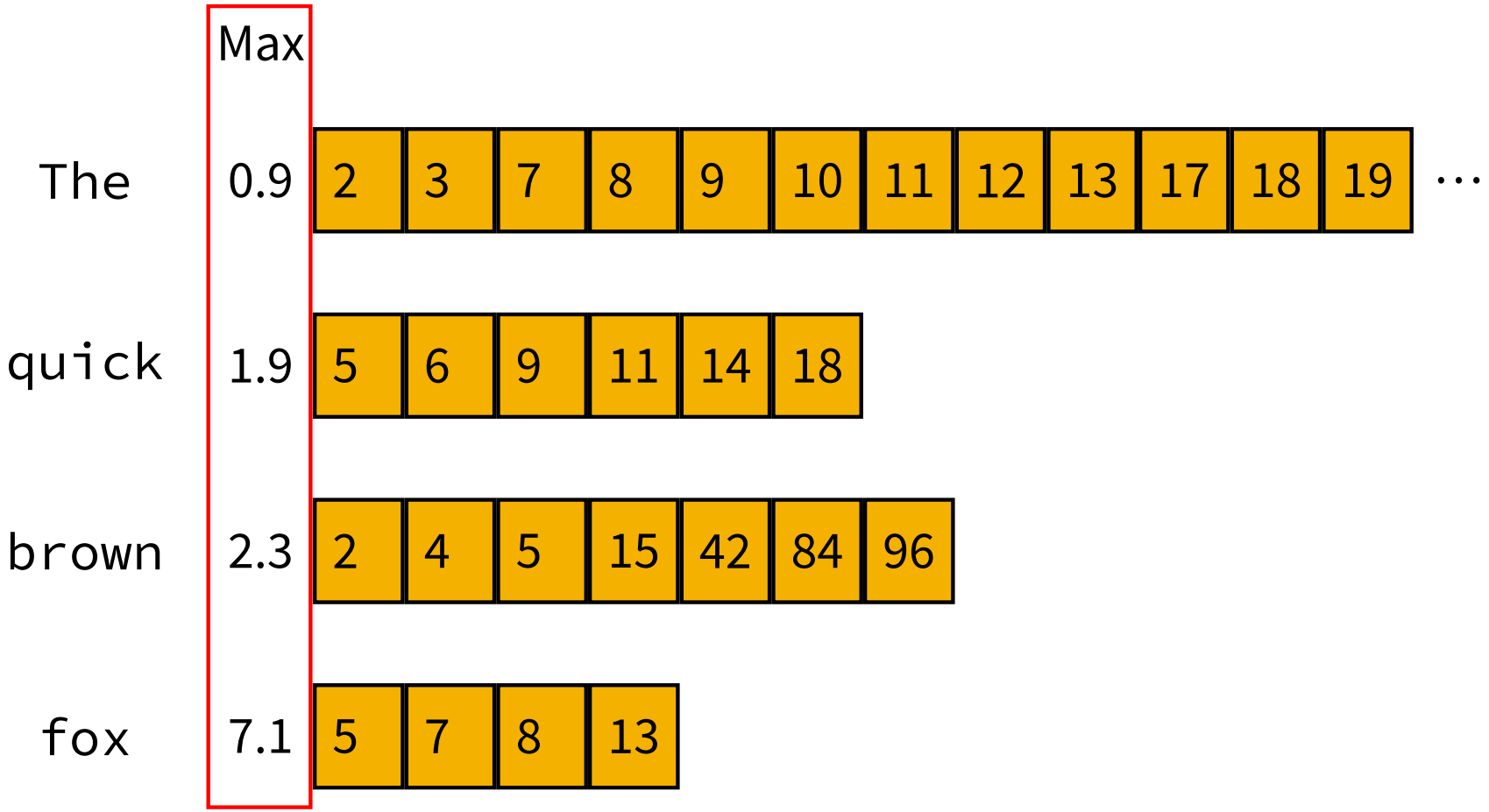
Query Q: `The quick brown fox` with $k = 2$



WAND - Example

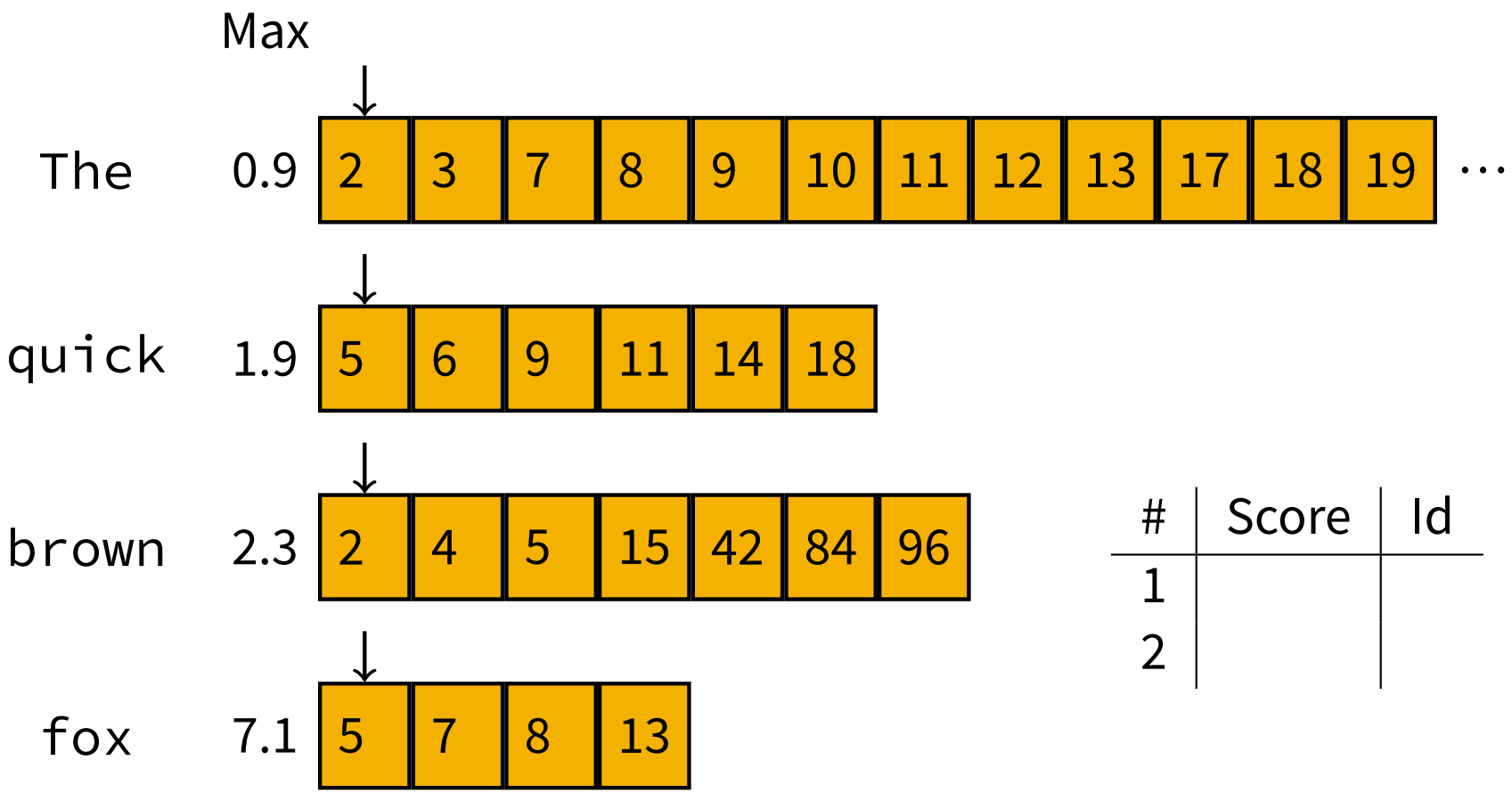
Query Q: `The quick brown fox` with $k = 2$

Maximum Contribution for each query term



WAND - Example

Query Q: `The quick brown fox` with $k = 2$

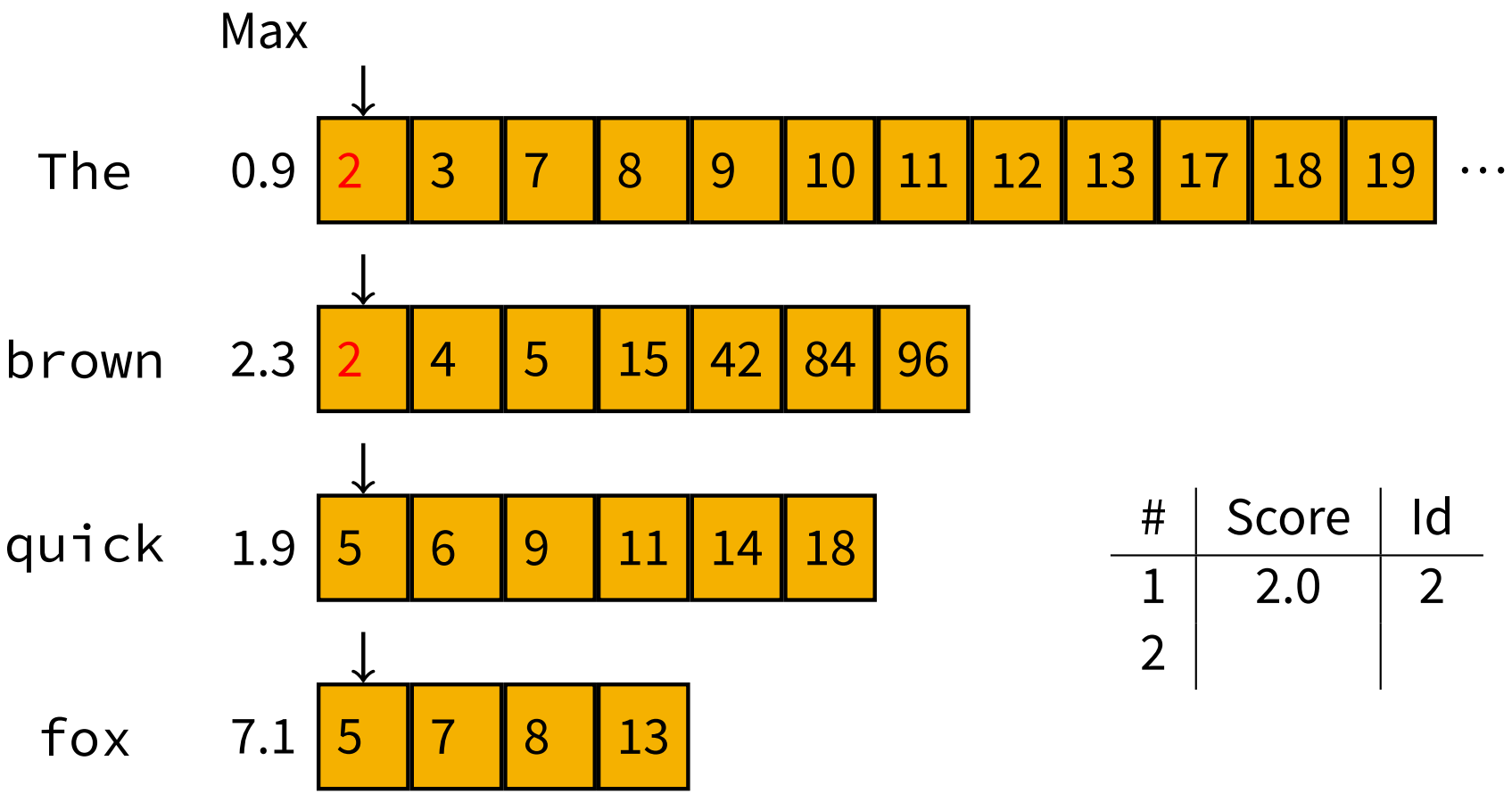


#	Score	Id
1		
2		

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(1) Reorder based on current id and score smallest.

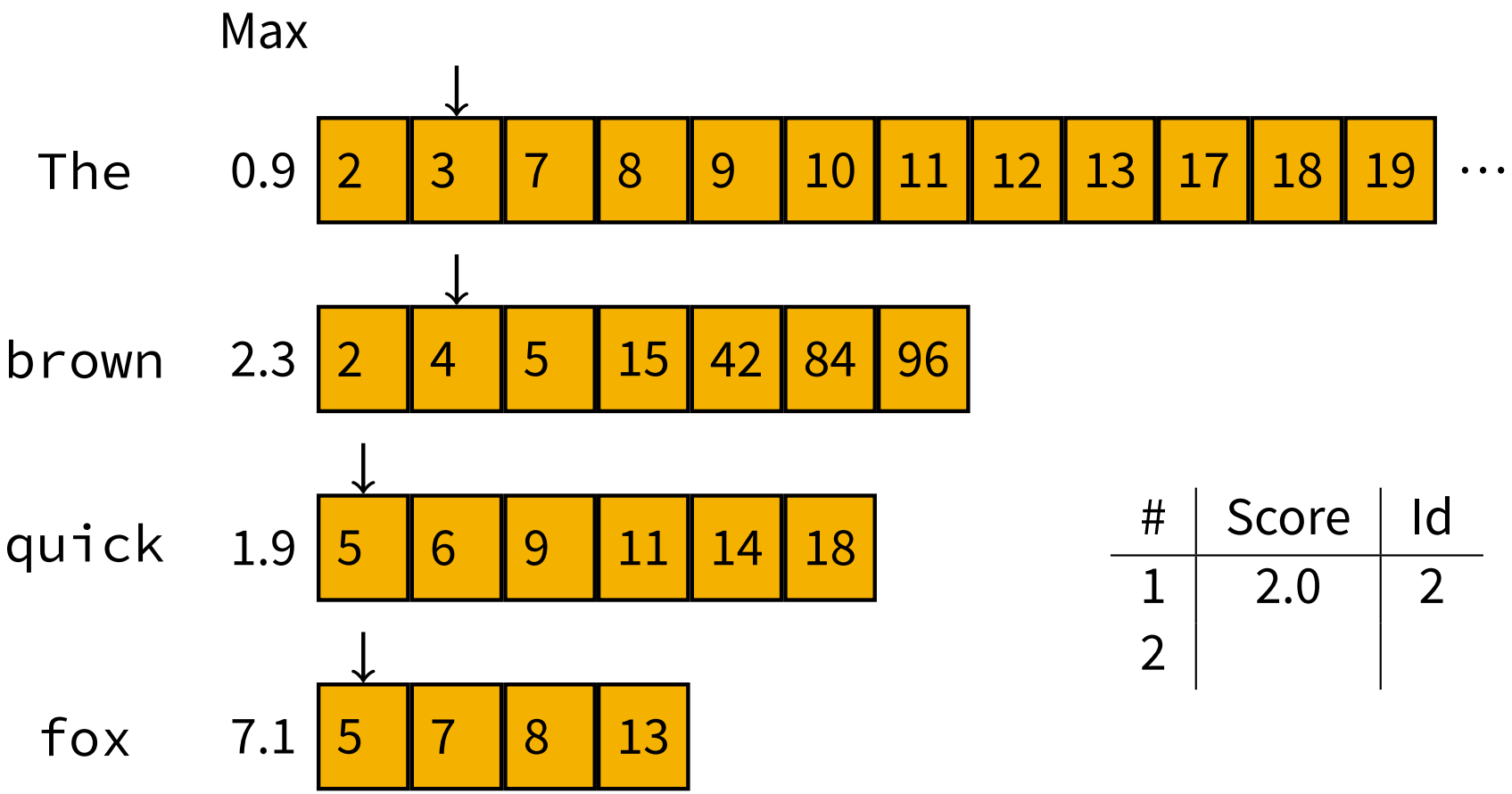


#	Score	Id
1	2.0	2
2		

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(2) Move pointer of scored elements.

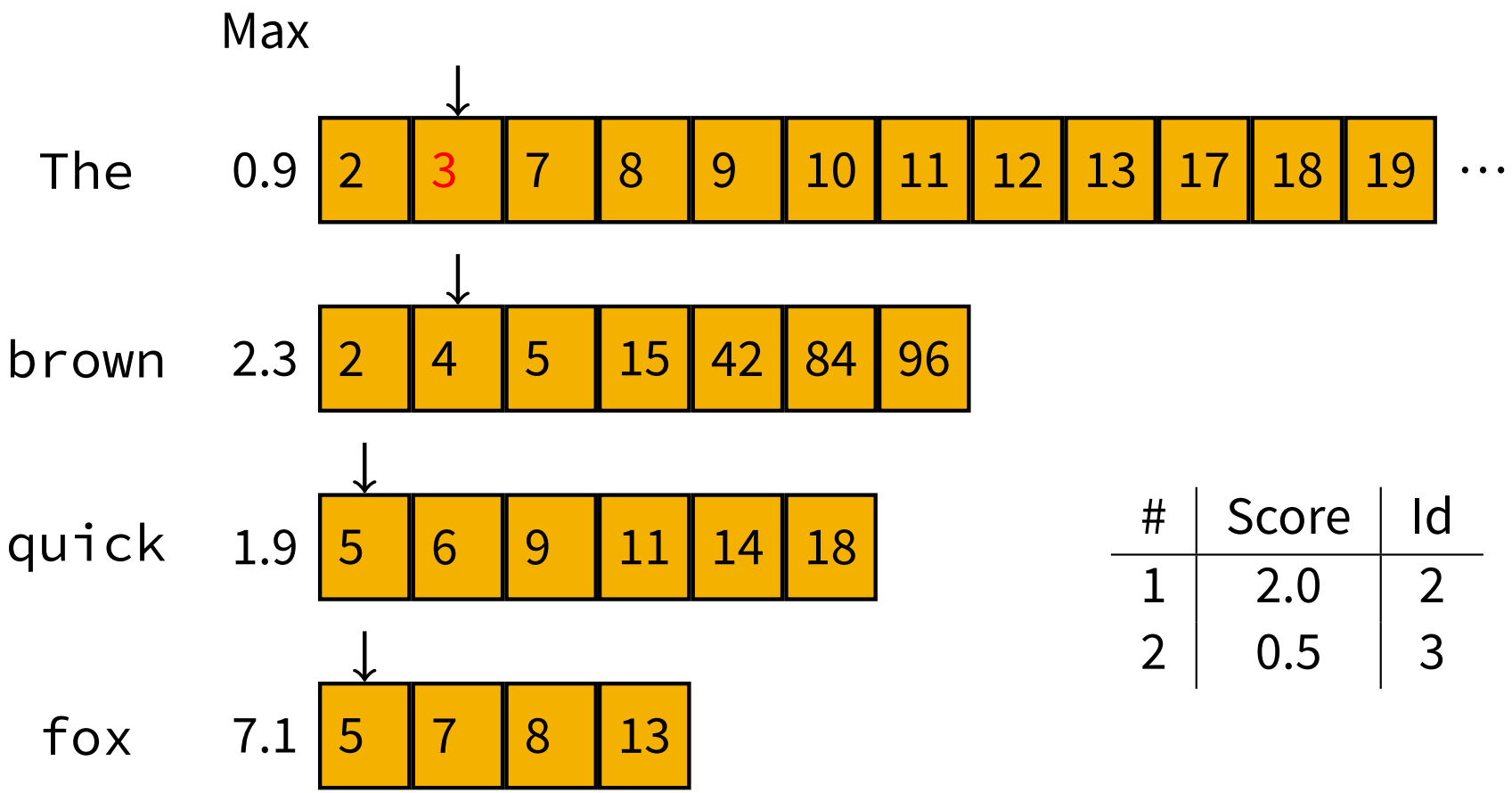


#	Score	Id
1	2.0	2
2		

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(3) Reorder based on current id and score smallest.

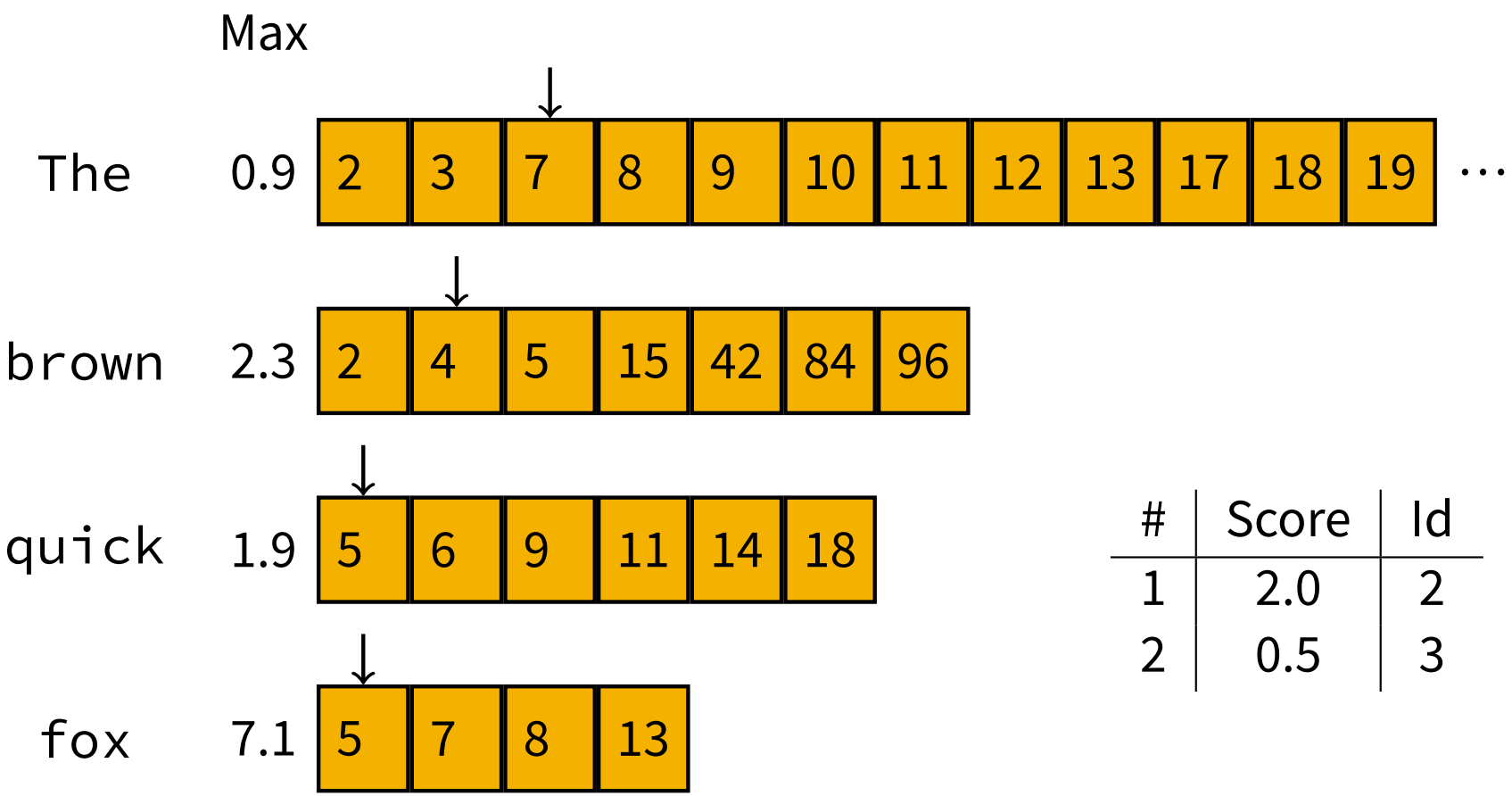


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(4) Move pointer of scored elements

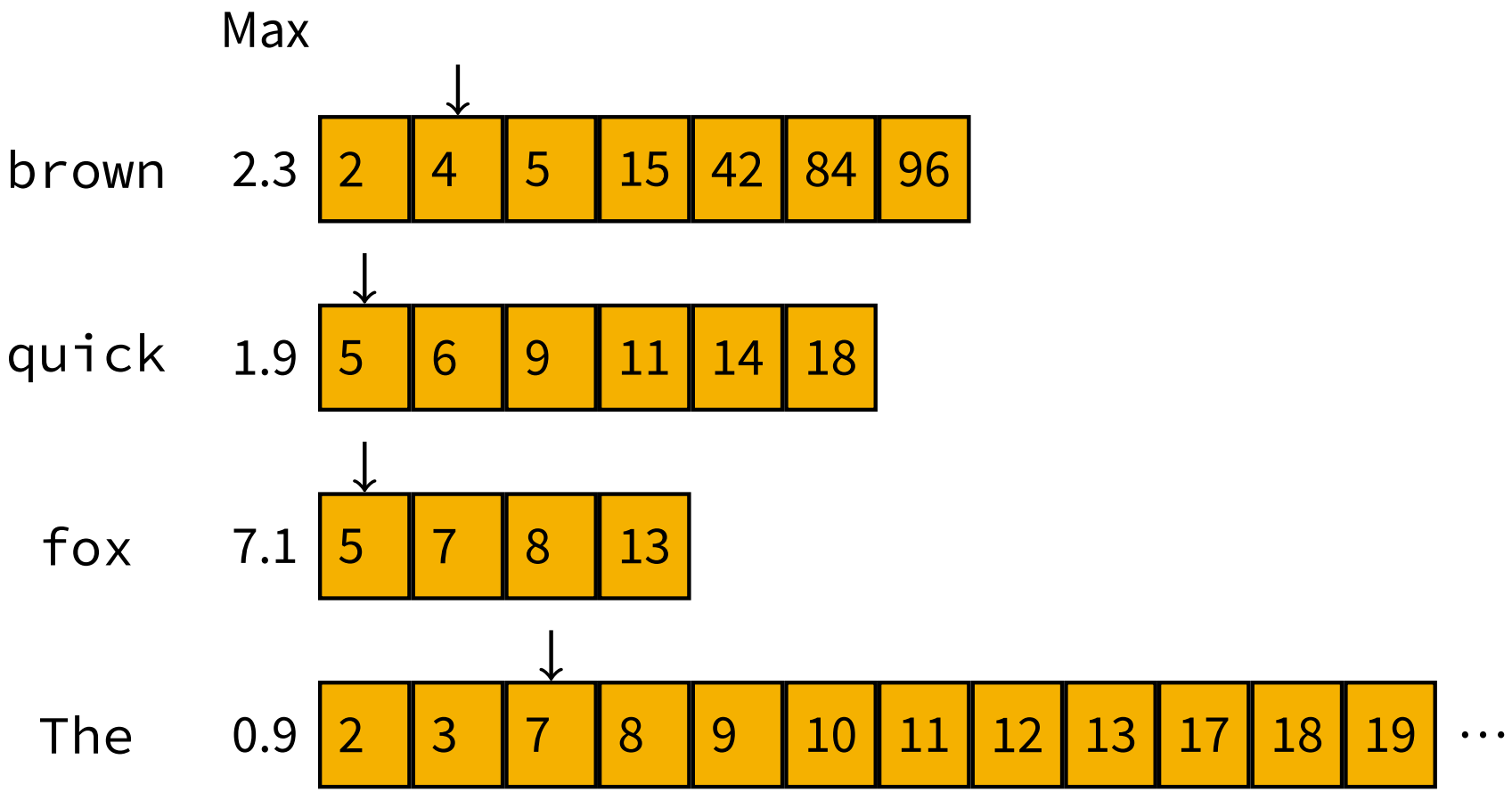


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

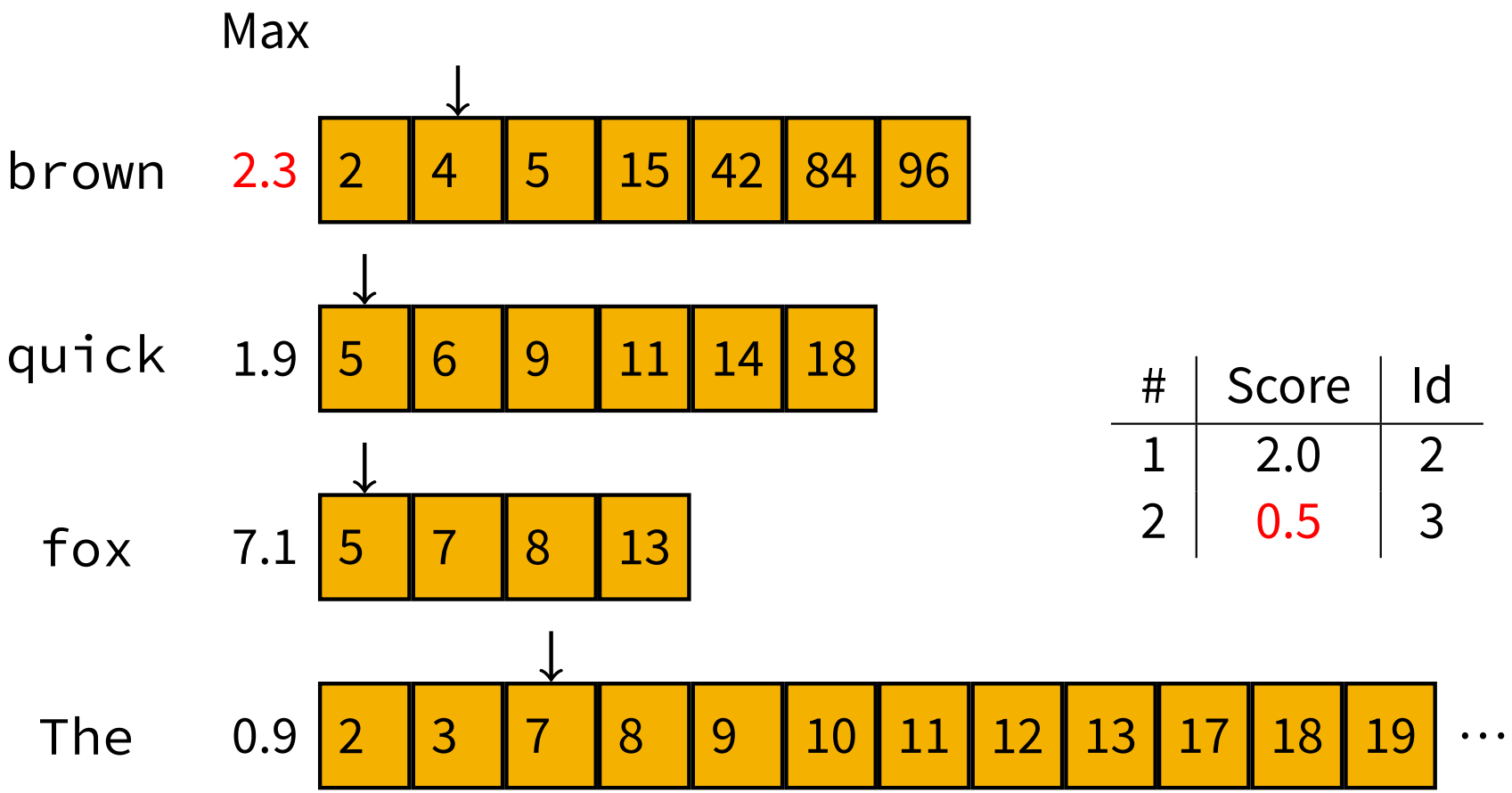
(5) Reorder based on current id and score smallest.



WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(6) Decide if we need to score smallest id.

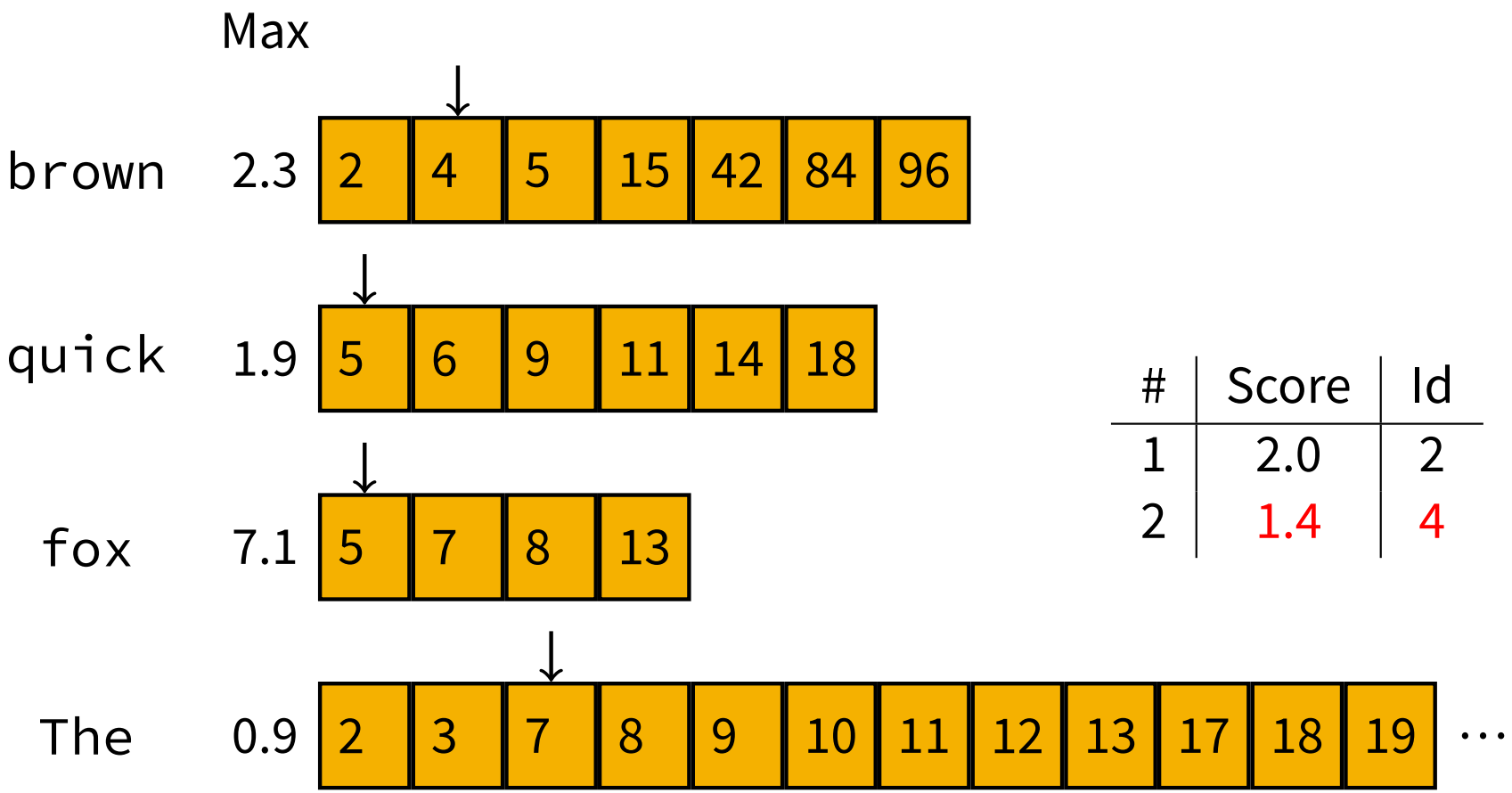


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(7) Replace 3 with 4 on the heap.

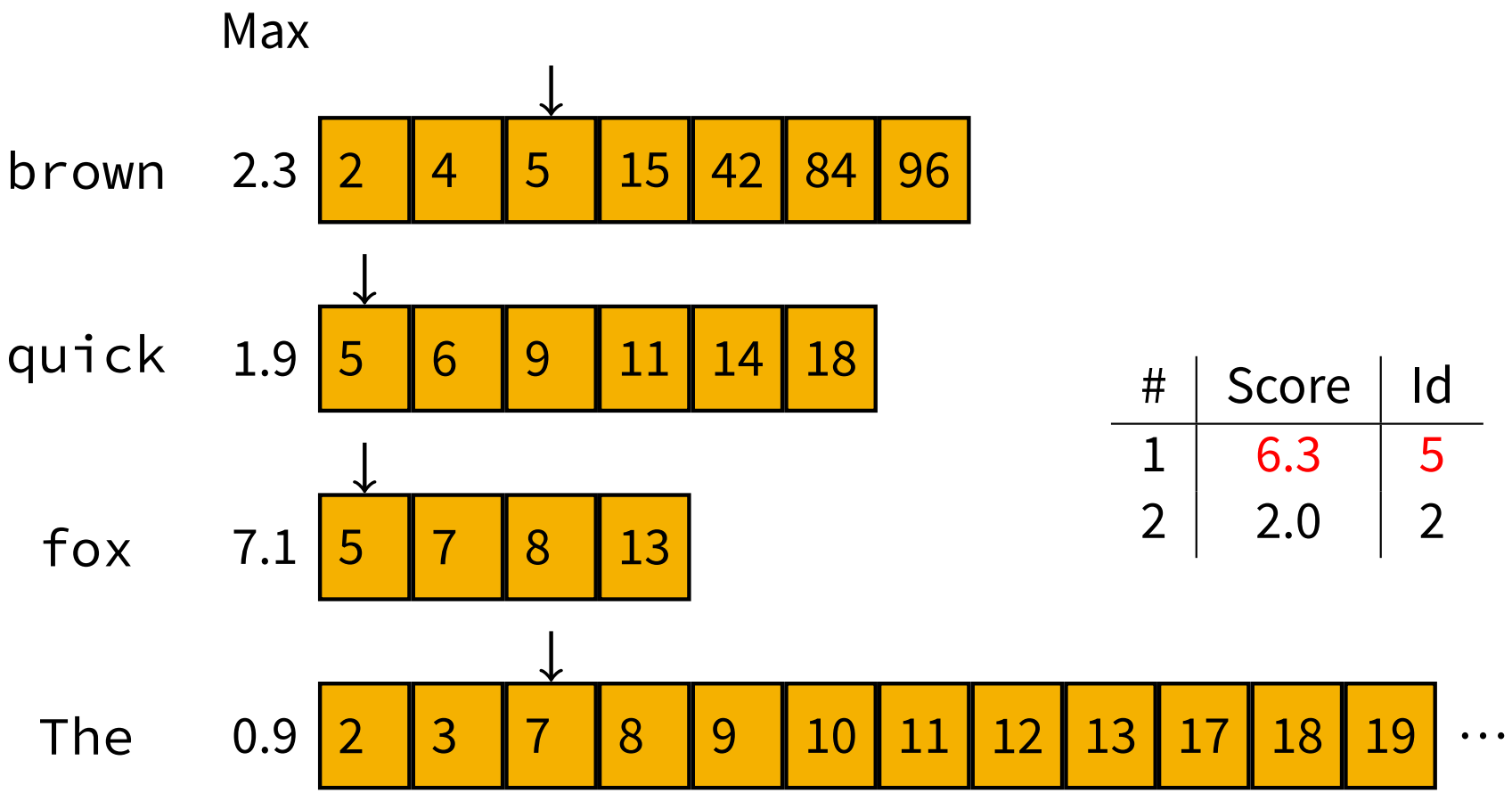


#	Score	Id
1	2.0	2
2	1.4	4

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(8) Sort by current id. Evaluate 5. Add to Heap.

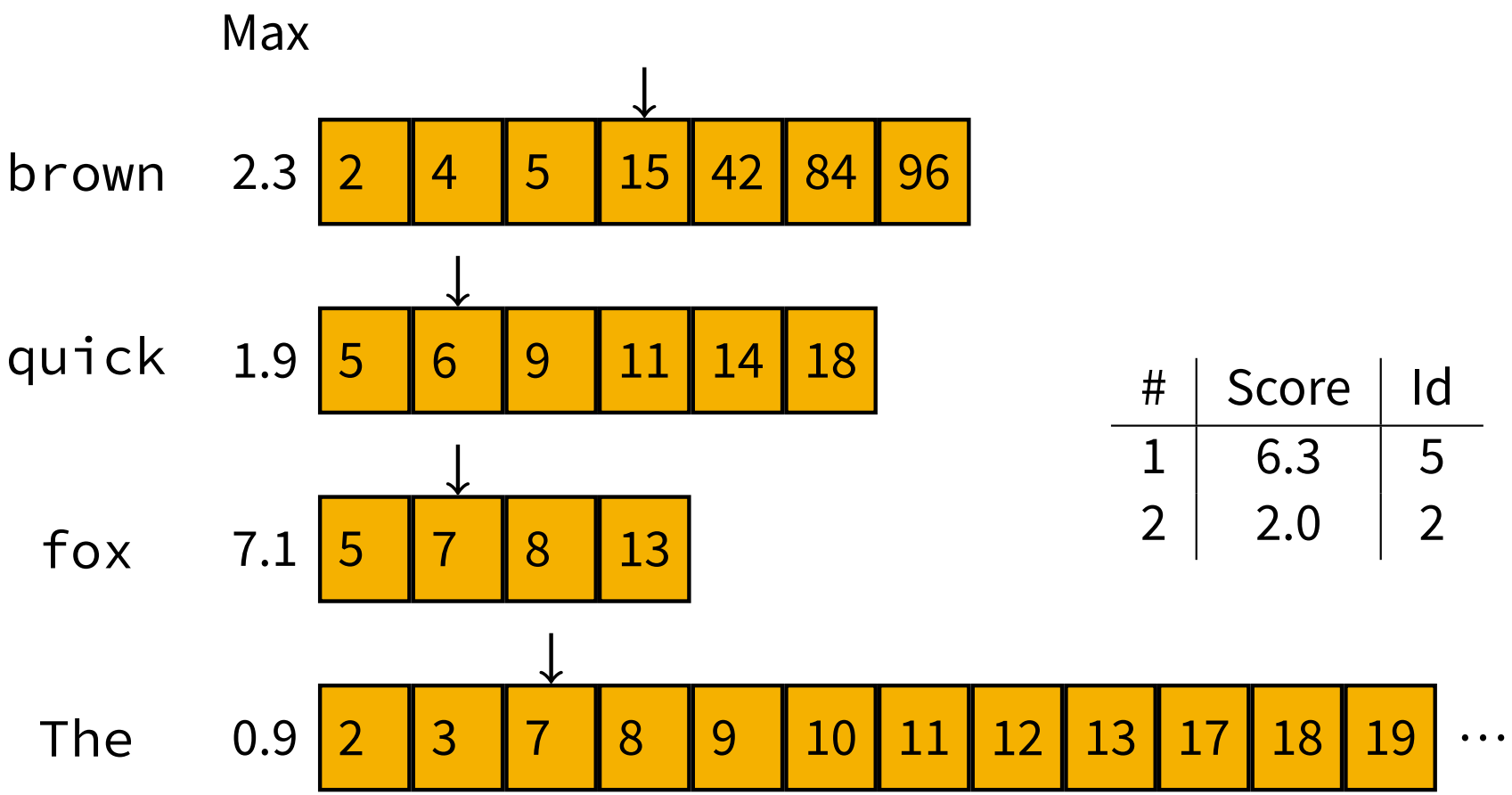


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(9) Move pointers and sort.

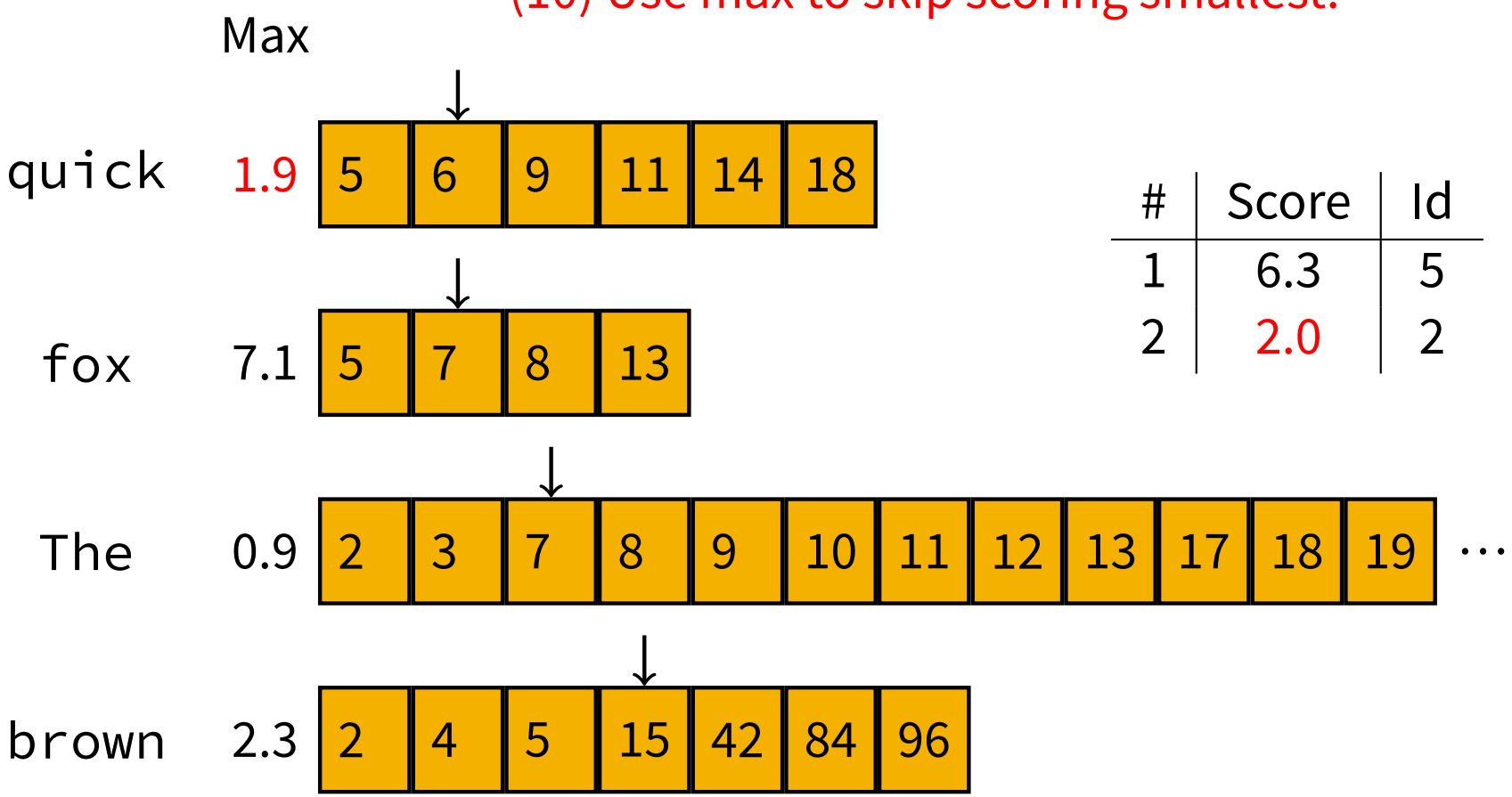


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example

Query Q: `The quick brown fox` with $k = 2$

(10) Use max to skip scoring smallest.

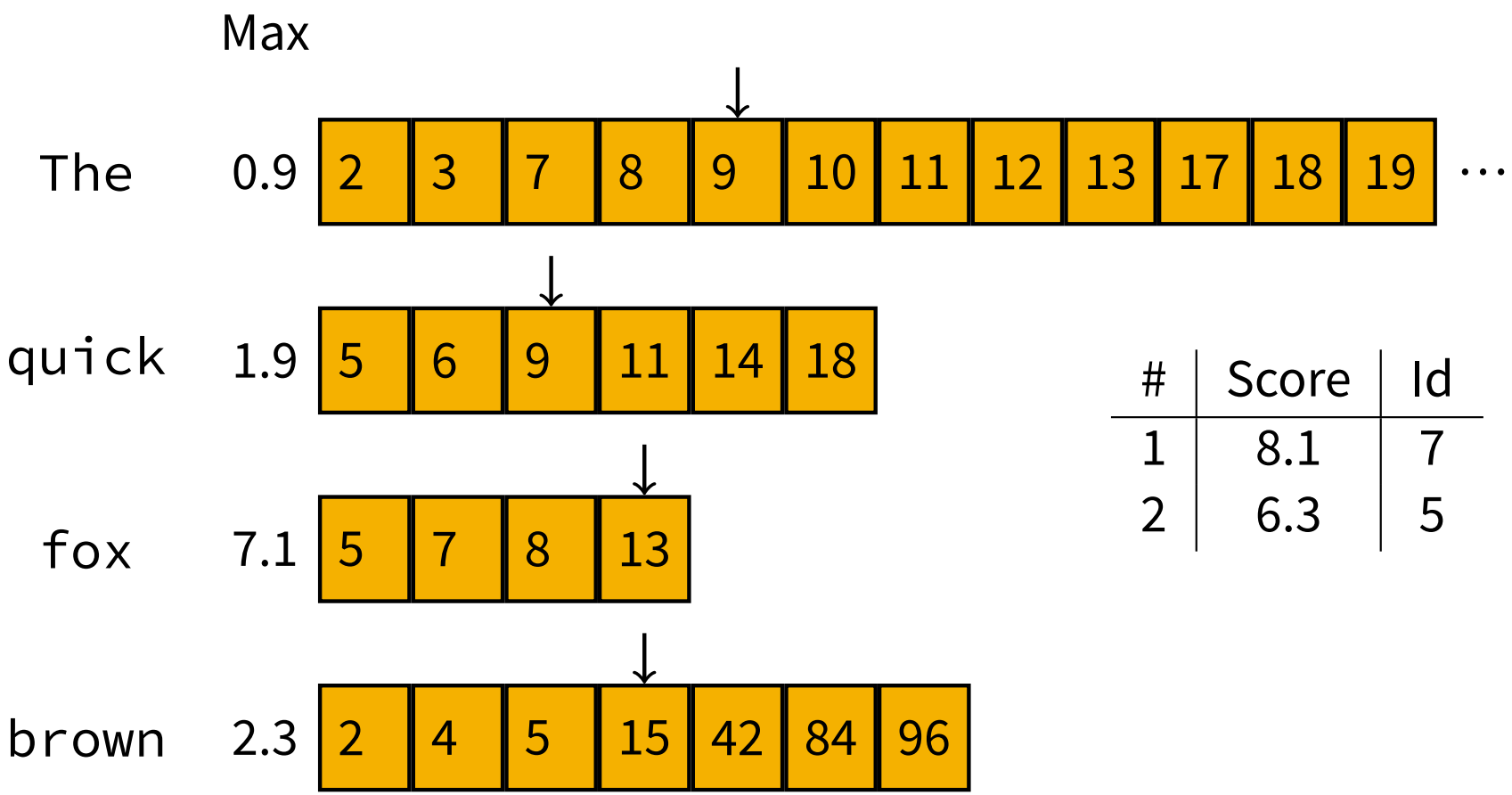


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example - Fast Forward

Query Q: `The quick brown fox` with $k = 2$

Do we have to evaluate document **9**?

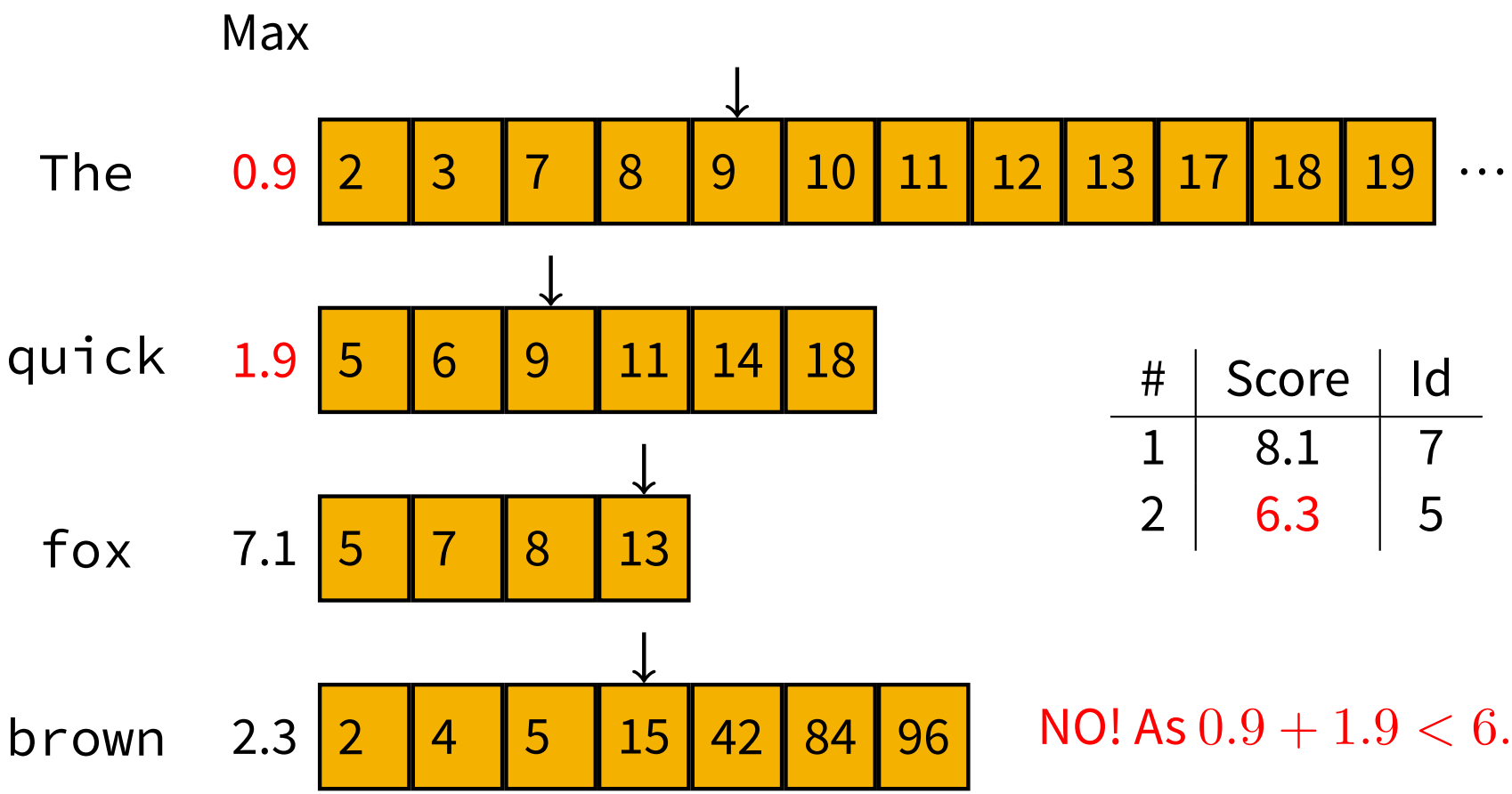


#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Fast Forward

Query Q: `The quick brown fox` with $k = 2$

Do we have to evaluate document **9**?



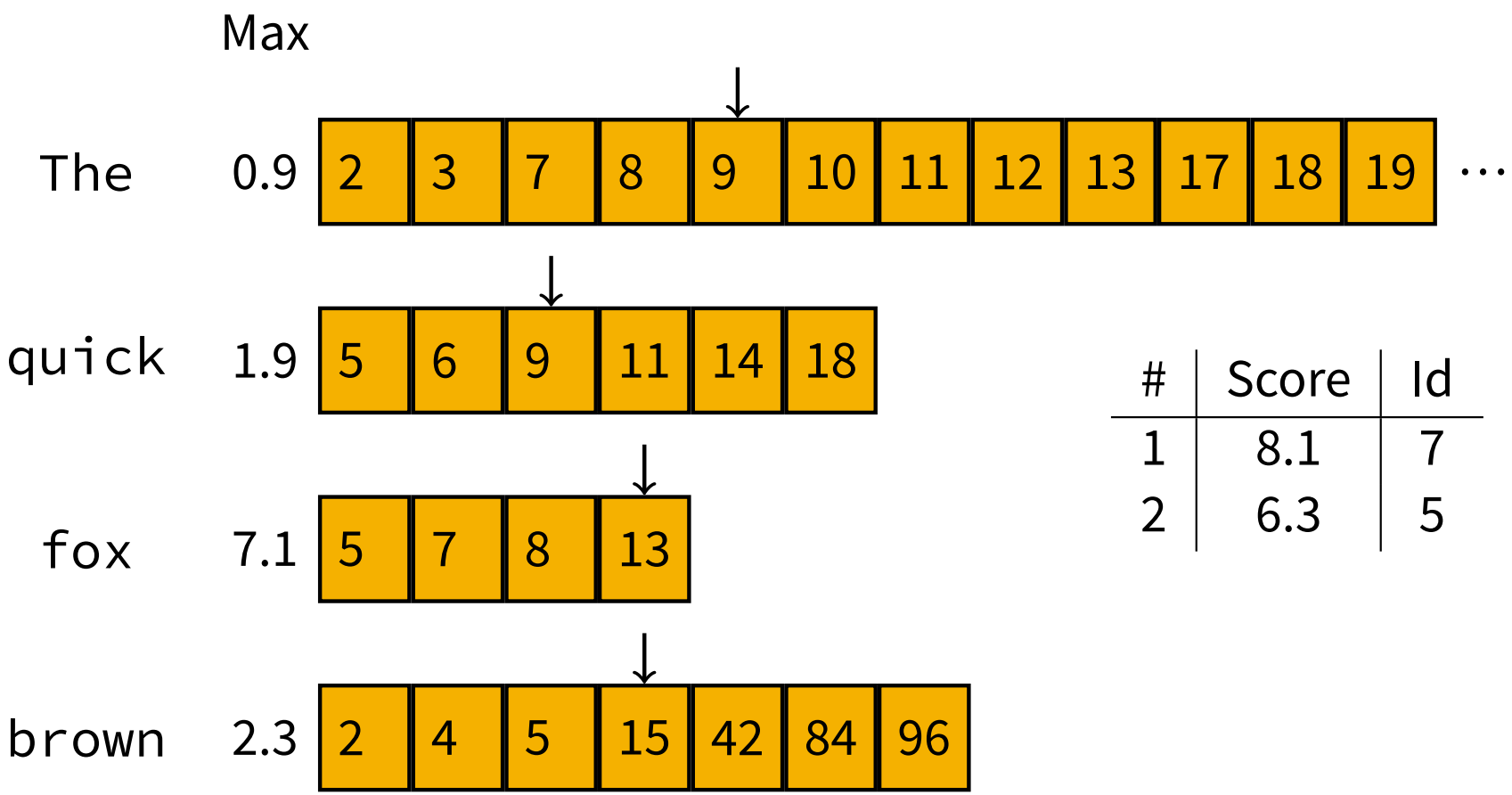
#	Score	Id
1	8.1	7
2	6.3	5

NO! As $0.9 + 1.9 < 6.3$!

WAND - Example - Fast Forward

Query Q: `The quick brown fox` with $k = 2$

What is the next document that has to be evaluated?

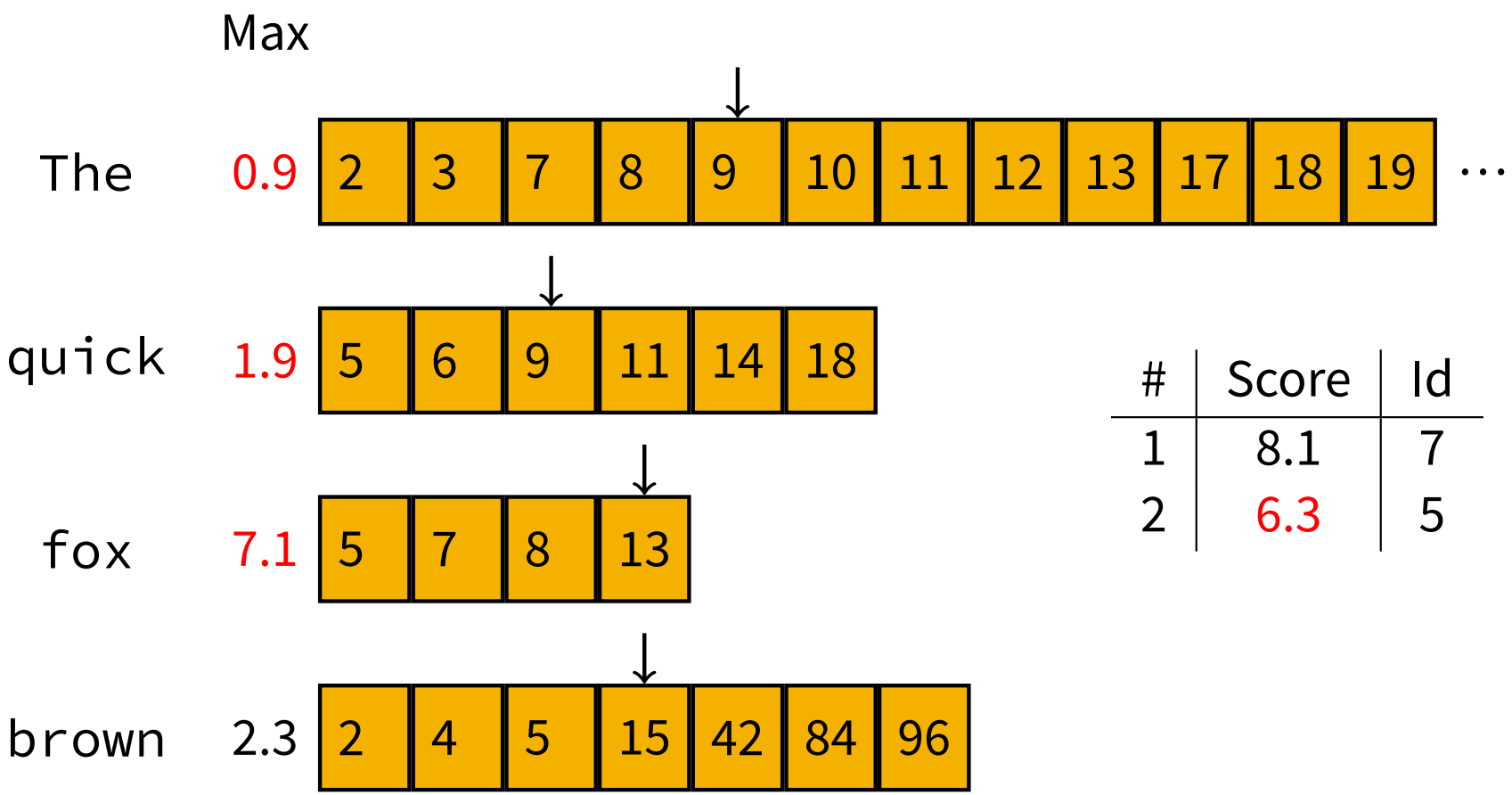


#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Fast Forward

Query Q: `The quick brown fox` with $k = 2$

What is the next document that has to be evaluated?



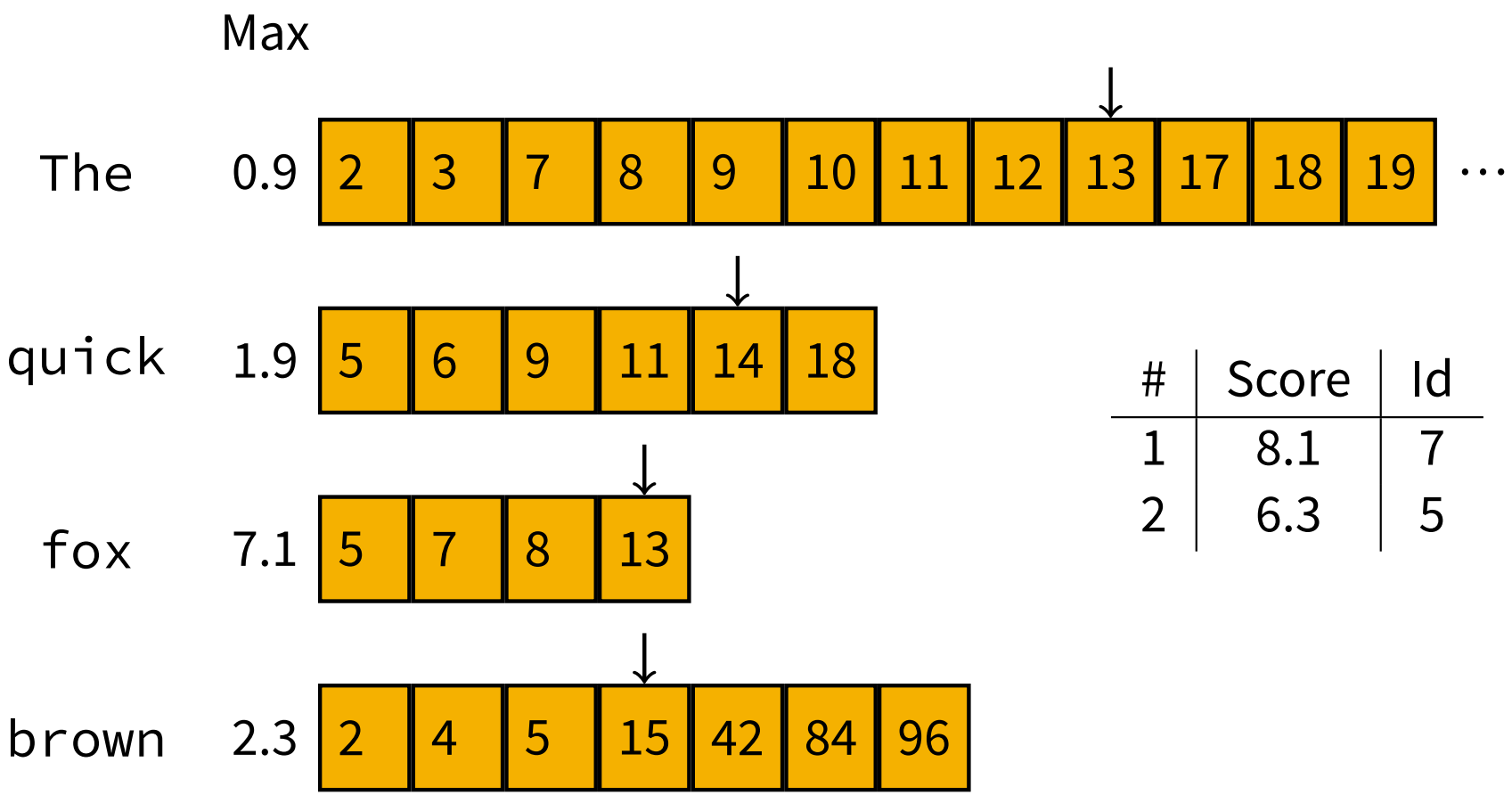
#	Score	Id
1	8.1	7
2	6.3	5

13, as $0.9 + 1.9 + 7.1 > 6.3$!

WAND - Example - Fast Forward

Query Q: `The quick brown fox` with $k = 2$

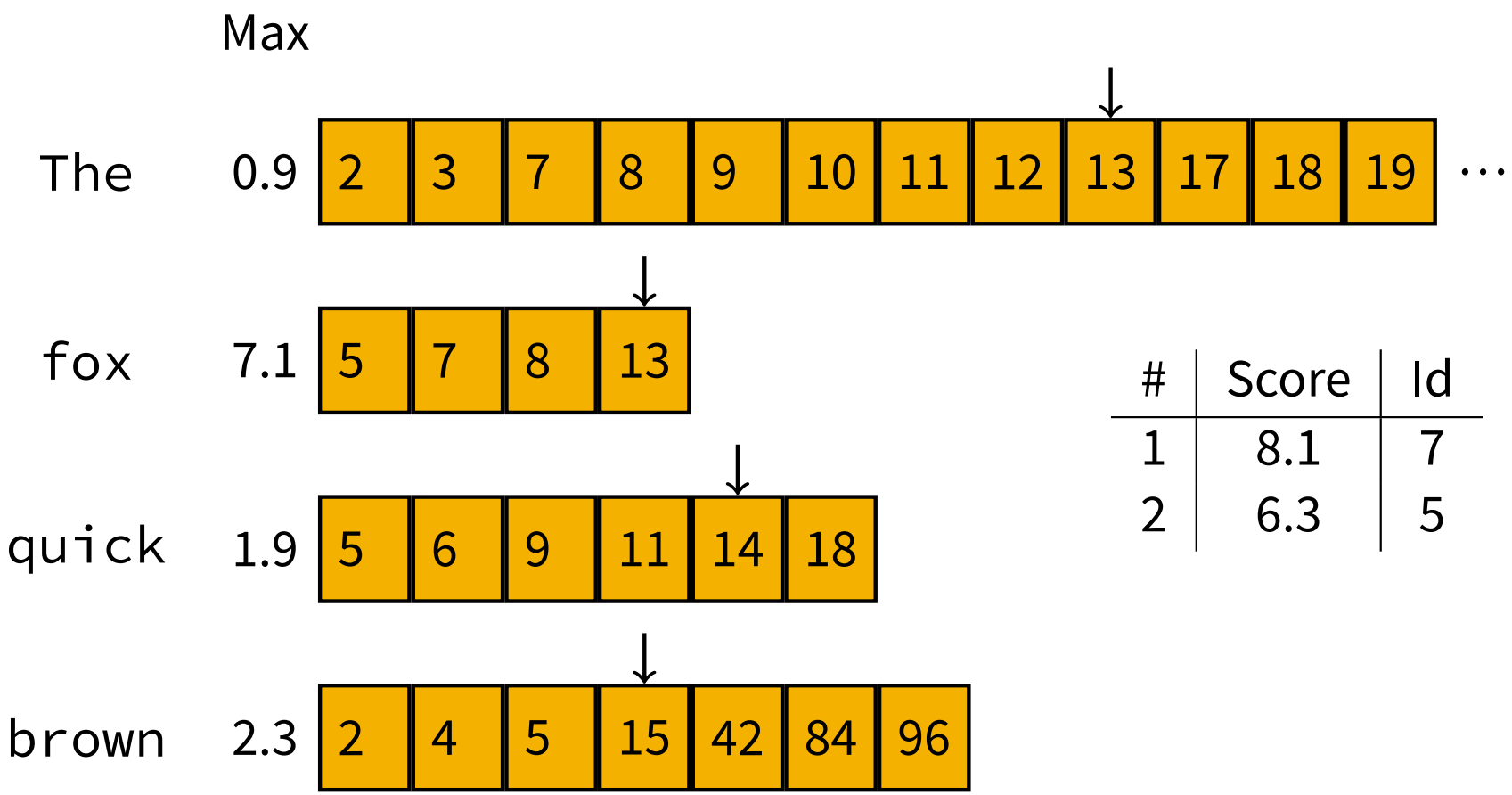
Fast forward smaller ids to 13 (GEQ) and sort.



#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Evaluate 13?

Query Q: `The quick brown fox` with $k = 2$



#	Score	Id
1	8.1	7
2	6.3	5

WAND - Algorithm

Given Q, k and the postings lists $L[0 \ldots |Q| - 1]$ with:

$L[i].max \hat{=}$ the **maximum contribution** of the list
$L[i].cur \hat{=}$ the current element of the list

1: **function** WAND(Q,k,$L[0 \ldots |Q| - 1]$)
2: $TopDocs = \varnothing$ ▷ Min Heap of size k
3: Threshold $\Theta = 0$ ▷ Smallest score in *TopDocs*
4: **while** Not all lists are processed **do**
5: Sort L based on $L[i].cur$
6: Select pivot list p such that $\sum_0^{p-1} L[i].max >= \Theta$
7: Forward all lists $L[0 \ldots |p| - 1]$ to $d_p = L[p].cur$
8: Compute S^{BM25}_{Q,d_p} and insert into *TopDocs* if score $> \Theta$
9: $\Theta = \min(TopDocs)$ or 0 if $|TopDocs| < k$.
10: **end while**
11: Return *TopDocs*
12: **end function**

WAND - Discussion

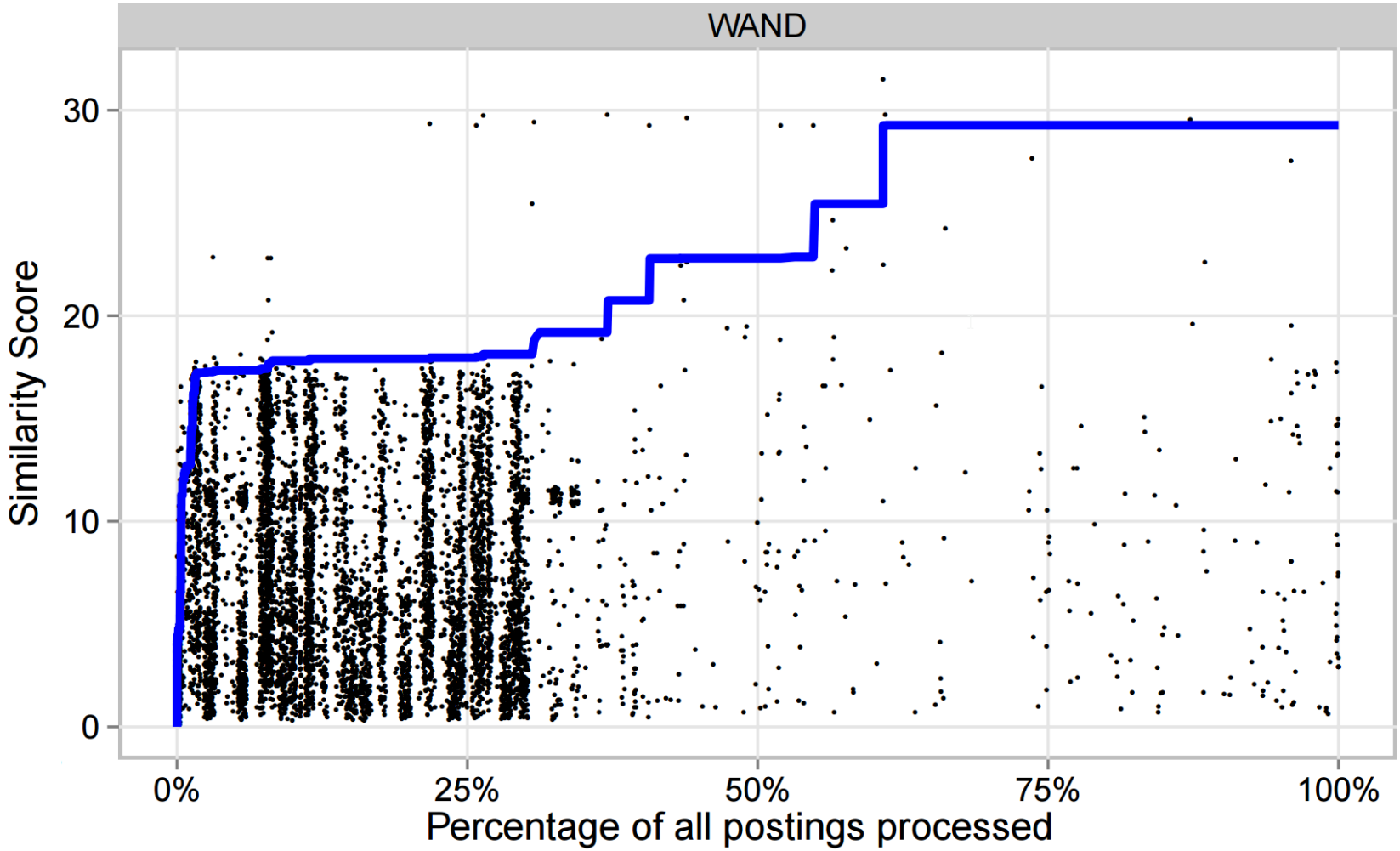
- Use max contribution of query term to overestimate score of a document.
- Do not score document if it can not enter the top-*k* heap.
- Utilize GEQ function of compressed representation to skip over large parts of the postings lists.
- Similarity metric fixed at index construction time.
- Works very well in practice.

WAND - Performance

Method	$k = 10$		$k = 1{,}000$	
	Avg.	Med.	Avg.	Med.
Exhaustive	100.0	100.0	100.0	100.0
BM25	3.5	1.0	28.0	11.7

Fraction of pointers processed as a percentage of the total number of pointers associated with each query, GOV2, using TREC topics 701–850. Across the set of queries, the average number of postings per query for exhaustive processing is 1,460,562, and the median number of postings is 1,080,008. The percentages shown in the table are relative to these two numbers.

WAND - Performance II



Evaluation of one query ""north korean counterfeiting" for $k = 10$.

Further Reading

Reading:

- Manning, Christopher D; Raghavan, Prabhakar; Schütze, Hinrich; Introduction to information retrieval, Cambridge University Press 2008. (Chapter 5)

Additional References:

- Daniel Lemire, Leonid Boytsov: Decoding billions of integers per second through vectorization. Softw., Pract. Exper. 45(1): 1-29 (2015)
- Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, Jason Y. Zien: Efficient query evaluation using a two-level retrieval process. CIKM: 426-434 (2003)