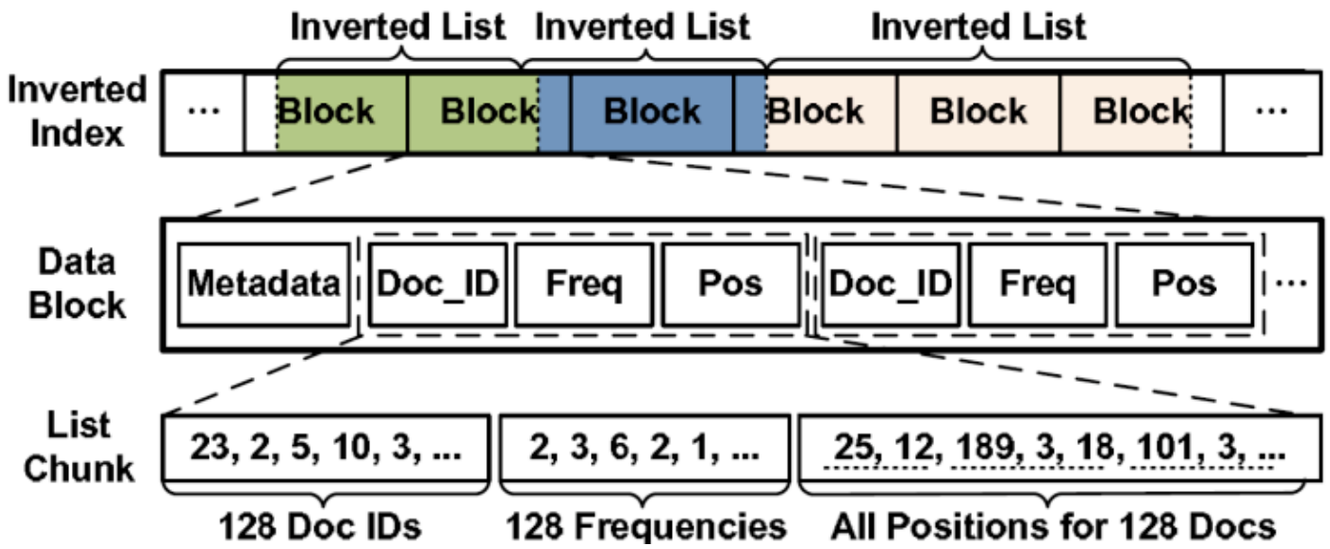
Indexing

Compression and QP

Posting list Layout

Data blocks, say of size 64KB, as basic unit for list caching



List chunks, say of 128 postings, as basic unit of decompression

- Posting list storage is implemented as
  - list of document identifiers (did)
  - list of scores, positions
- Many chunks are skipped over, but very few blocks are
- Also, may prefetch the next, say 2MB of index data from disk

Posting list Compression

- In a collection of 2^{12} documents
- Minimum number of bits required ?
- What is the uncompressed size for this list ?
- Variable byte encoding:
  - **7-bit encoding:** Use 7 bits for every byte and a continuation bit

1220 = 00000000 00000100 11000100

= 00001001 **1**1000100

Doc. id list

2
3
8
15
16
20
1220
1221
1229
1235
1237
1000001
1000003

d-gap Encoding

- Store the first doc. id and gaps or relative differences — d-gaps
- Lists are read sequentially
- compress them using variable byte encoding
- compression and decompression is fast

Doc. id list	Doc. id list (gaps)
2	2
3	1 → 3-2
8	5 → 8-3
15	7
16	1
20	4
1220	1200
1221	1
1229	8
1235	5
1237	2
1000001	998764
1000003	2

13 x 3 bytes

d-gap Encoding

- Store the first doc. id and gaps or relative differences — d-gaps
- Lists are read sequentially
- compress them using variable byte encoding
- compression and decompression is fast

Doc. id list	Doc. id list (gaps)	
2	2	
3	1	→ 3-2
8	5	→ 8-3
15	7	
16	1	
20	4	
1220	1200	
1221	1	
1229	8	
1235	5	
1237	2	
1000001	998764	
1000003	2	
13 x 3 bytes	(11 x 1 + 2 + 3) bytes	

Rice Encoding

For a given **b**, each number x in the list we do the following:

Doc. id list (gaps)	x-1 / b in unary	x-1 mod b in binary
34	33 / **64**	33 % **64**
144	0	10001
113	rice code	
162		

$2^{\lfloor log(g) \rfloor}$

- **b** is nearest (smaller) power of two of the average of the numbers [g = (34+144+113+162) / 4 = 113.33]
- There are no zeros to encode
- simple to implement (bitwise operations) and better compression than var-byte, but slightly slower

Golomb Encoding

For a given **b**, each number x in the list we do the following:

Doc. id list (gaps)	x-1 / b in unary	x-1 mod b in binary
34	33 / **78**	33 % **78**
144	0	10001
113	golomb code	
162		

- **b** is 0.69 of the average of the numbers [g = (34+144+113+162) / 4 = 113.33]*0.69 ~ 78
- need fixed encoding of number 0 to 77 using 6 or 7 bits
- optimal for random gaps (dart board, random page ordering)

Gamma Encoding

Each number x in the list, **b** = $2^{\lfloor log(x) \rfloor}$

nearest smallest power of 2

log(b) in unary

x-1 mod (b) in binary (b bits)

Doc. id list (gaps)

34 → log(33) = 5 → 33 % **32**

144

113

162

111110 00001

gamma code

- Good compression for small values, e.g., frequencies, bad for large numbers, and fairly slow
- **Delta coding:** Gamma code; then gamma the unary part

Patched Frame of Reference (PFoR) Compression

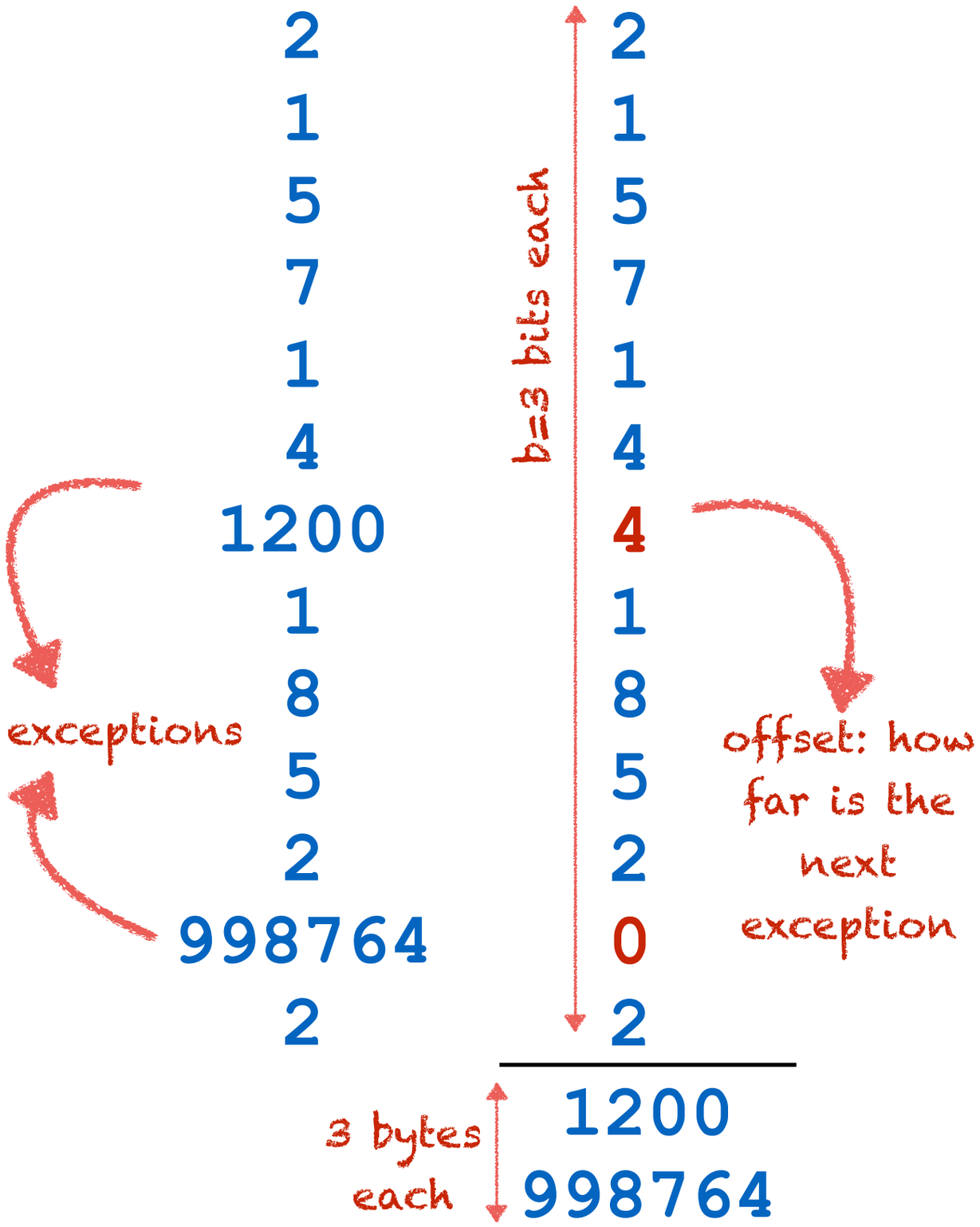
- Variable sizes for each number is bad for decompression due to branching
- Constant size for each number is bad for compression since valuable space is wasted

90% of numbers are small and 10% are large

- Trade-off — Use fixed size for most small integers and treat others as exceptions
- 11 /13 can be encoded in 3 bits and 3 bytes for remaining 2

```
2
1
5
7
1
4
1200
1
8
5
2
998764
2
```

PforDelta Compression



OPT-PFD :

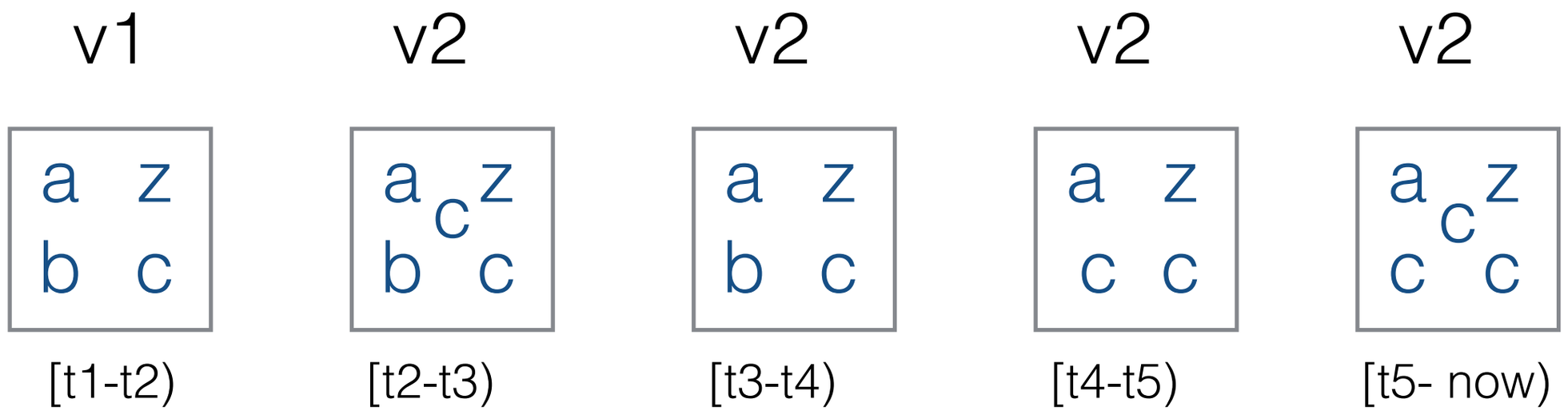
- keep the **last b bits** of exceptions in the original list and
- the **higher-order bits** and the **offsets** of the exceptions are separately encoded using another compression method.
- Instead of choosing a **b** that guarantees less than 10% exceptions in the block, select the b that results in the smallest compressed size for the block

Redundancy in Temporal Collections

- Documents with multiple versions
  - each modification results in a new version
  - Wikipedia, Web Archives
- Most of the edits in versioned collections are minor edits
- Major edits occur in bursts
- Adjacent versions have considerable redundant content
- How do we exploit this for compressing inverted indexes ?

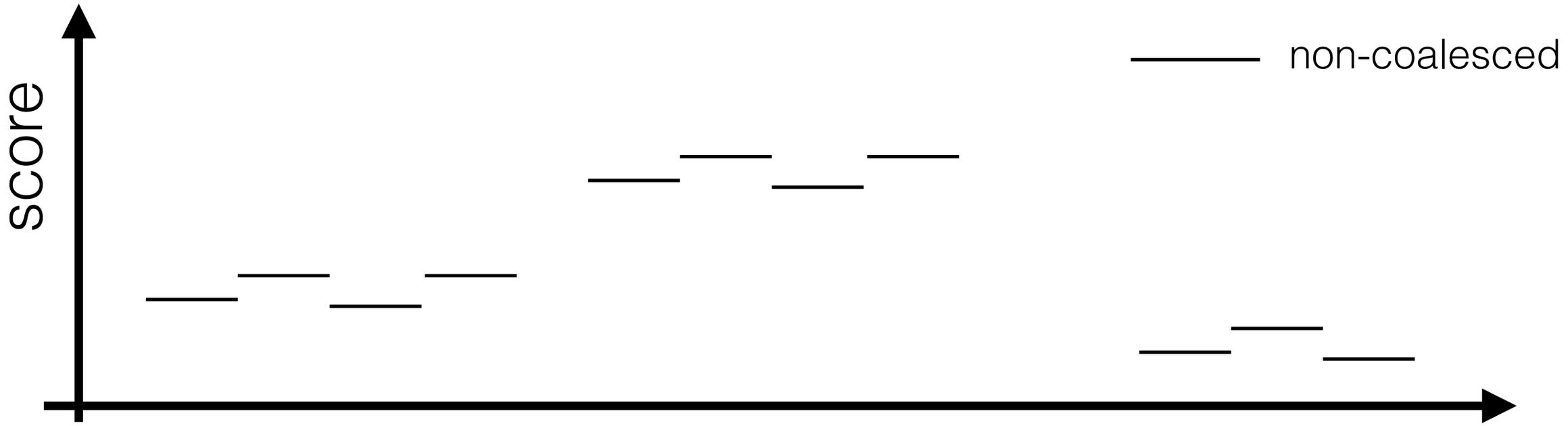
Temporal Coalescing

http://en.wikipedia.org/wiki/Interstellar_(film)



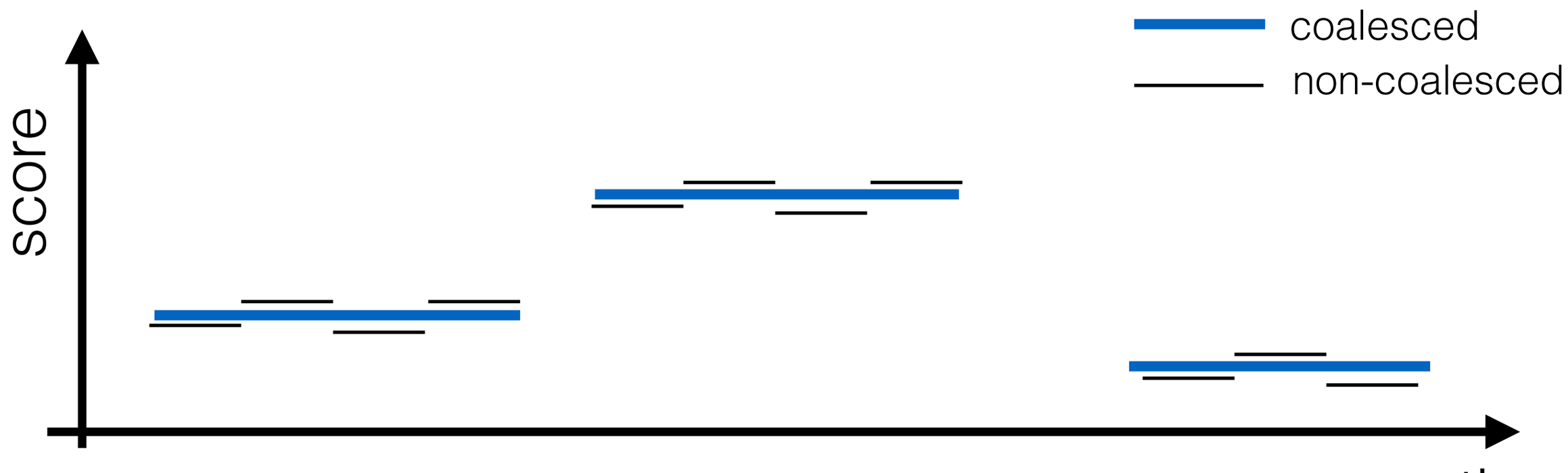
- Recollect time-travel queries: interstellar @ 2013-2014
- Indexing each version results in index blowup
- term frequencies do not change much hence have small impact on the final score
- Coalesce multiple versions of the same doc. into fewer postings

Temporal Coalescing



- Document with multiple versions (hence multiple postings) all having similar scores
- Not much difference in scores, hence no impact on overall scoring
- Coalesce postings for adjacent versions of same doc. with almost identical scores

Temporal Coalescing



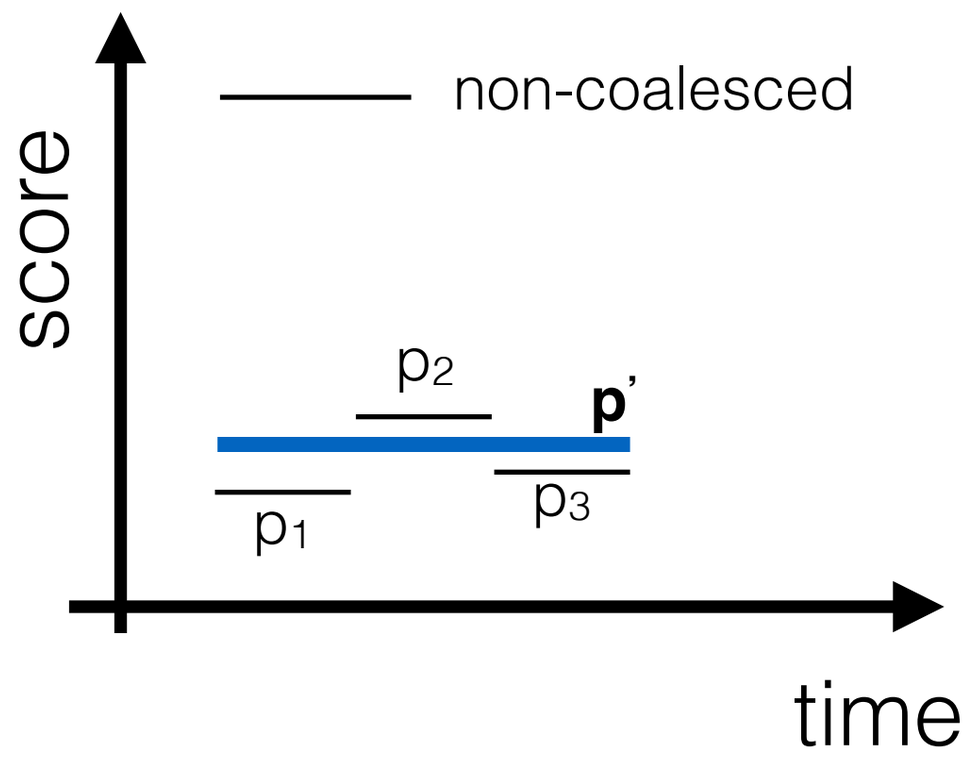
- Document with multiple versions (hence multiple postings) all having similar scores
- Not much difference in scores, hence no impact on overall scoring
- Coalesce postings for adjacent versions of same doc. with almost identical scores

Temporal Coalescing

- Problem Statement: Given sequence I of postings for term v in document d, determine minimal-length output sequence that keeps relative approximation error below threshold ε

$$\forall i \ : \ |p_i - p'| \, / \, |p_i| \ \leq \ \epsilon$$

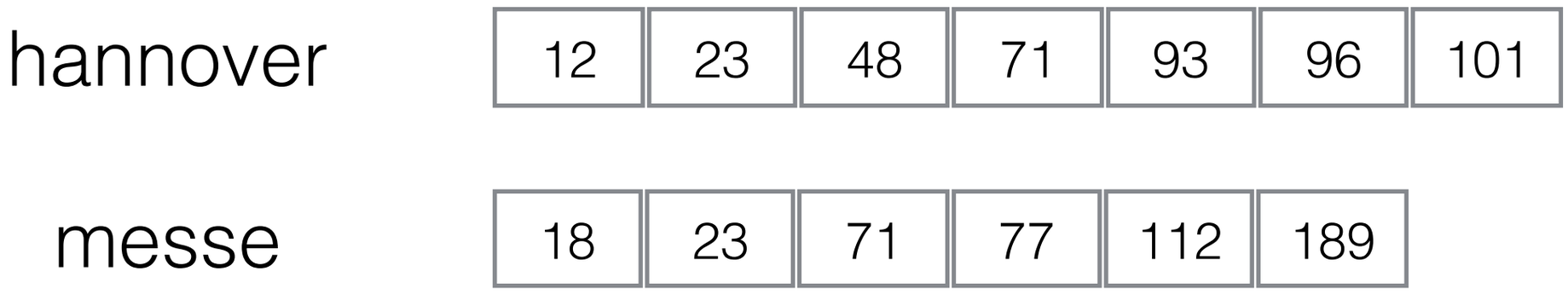
- Approach:
  - Keep a moving average of already seen values
  - Once the average is outside the error bounds of a posting finalize the coalesced posting



Two-Level Index

- Construct a two-level index
- For each posting list for term **t**
  - **first level:** maintains doc. ids if any of its versions mentions the term
  - **second level:** contains a bitmap of the versions which contain **t**
- First level considers a doc. as the union of all terms in its versions
  - Use PforDelta compression as in standard inverted indexes
- Second level is bit-level representation
  - Since terms are bursty, spans consisting bursts can be effectively compressed

Query Processing: Term-At-A-Time

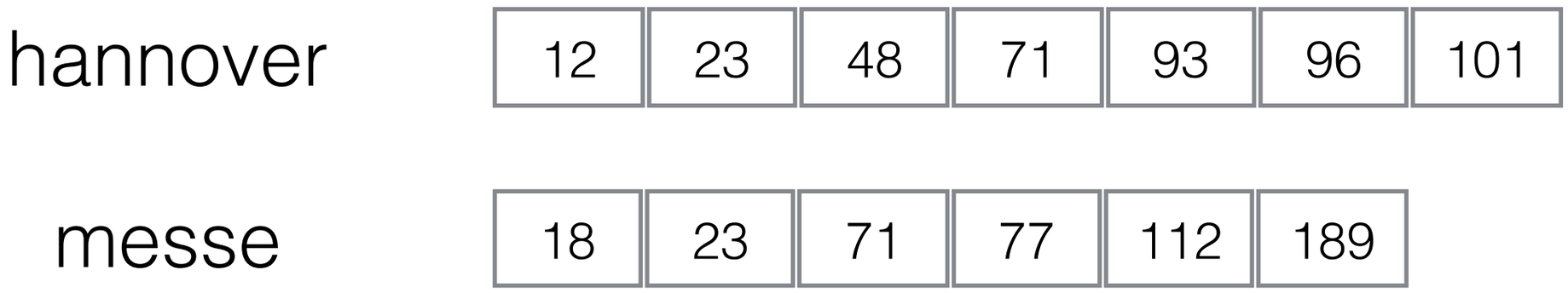


Term-at-a-time

- Process one list at a time
- Maintain accumulators for partial results and update them
- Best for unions

Accumulators in memory

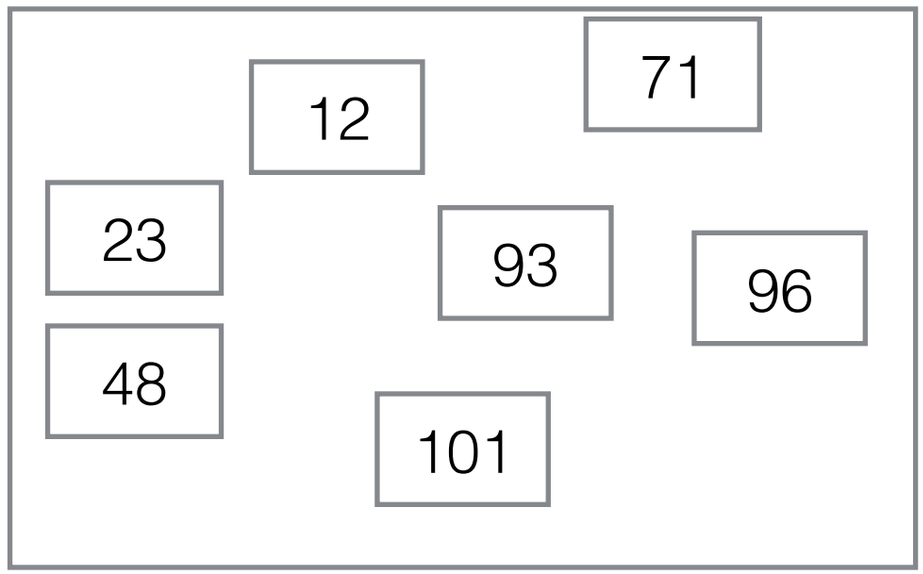
Query Processing: Term-At-A-Time



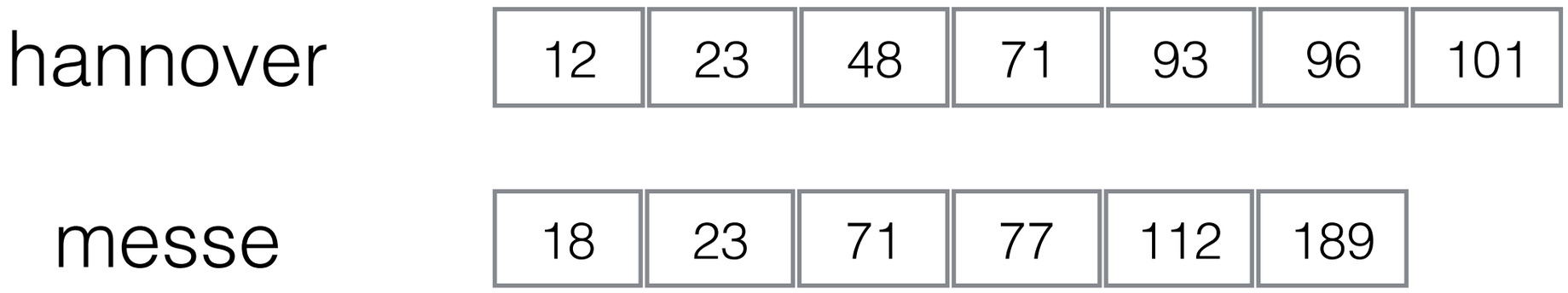
Term-at-a-time

- Process one list at a time
- Maintain accumulators for partial results and update them
- Best for unions

Accumulators in memory



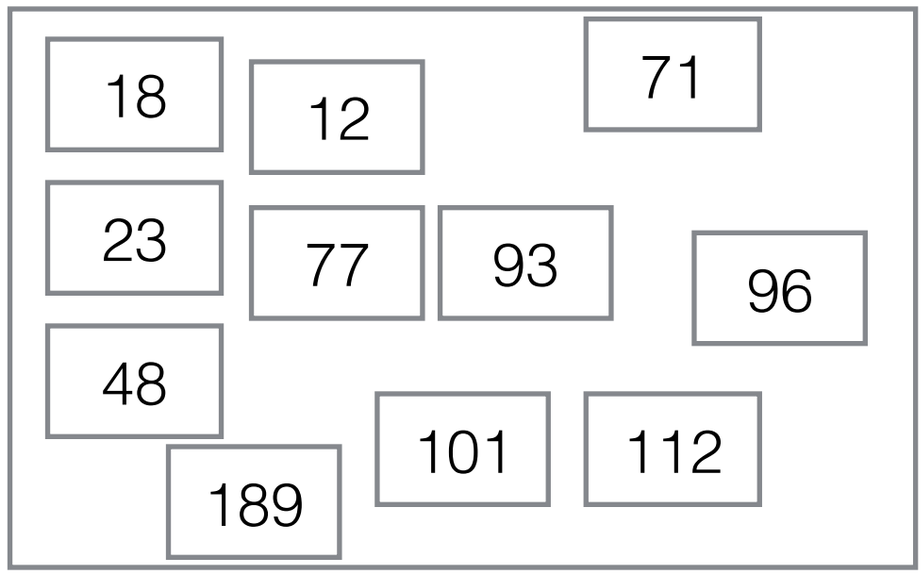
Query Processing: Term-At-A-Time



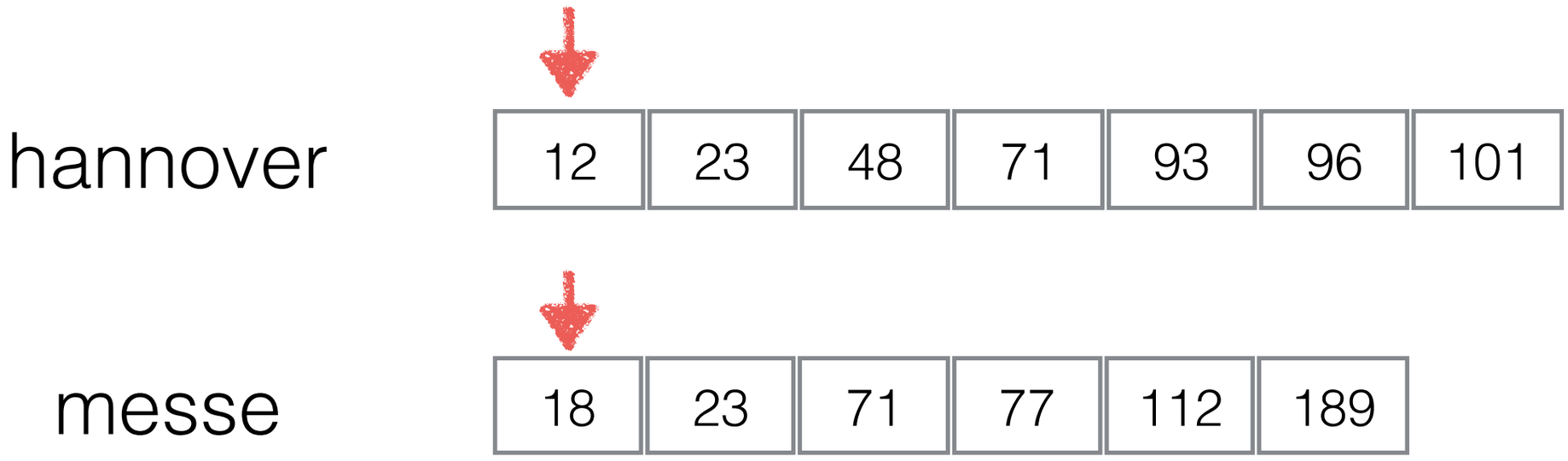
Term-at-a-time

- Process one list at a time
- Maintain accumulators for partial results and update them
- Best for unions

Accumulators in memory



Query Processing: Document-At-A-Time



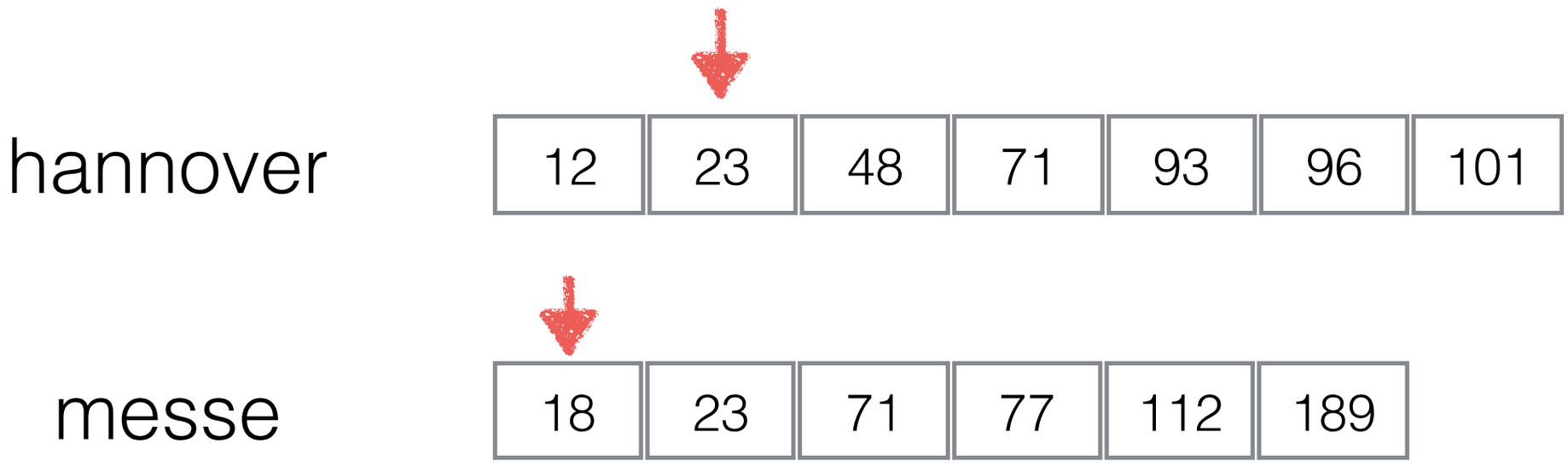
Document-at-a-time

- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

Conjunctive query semantics

- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further

Query Processing: Document-At-A-Time



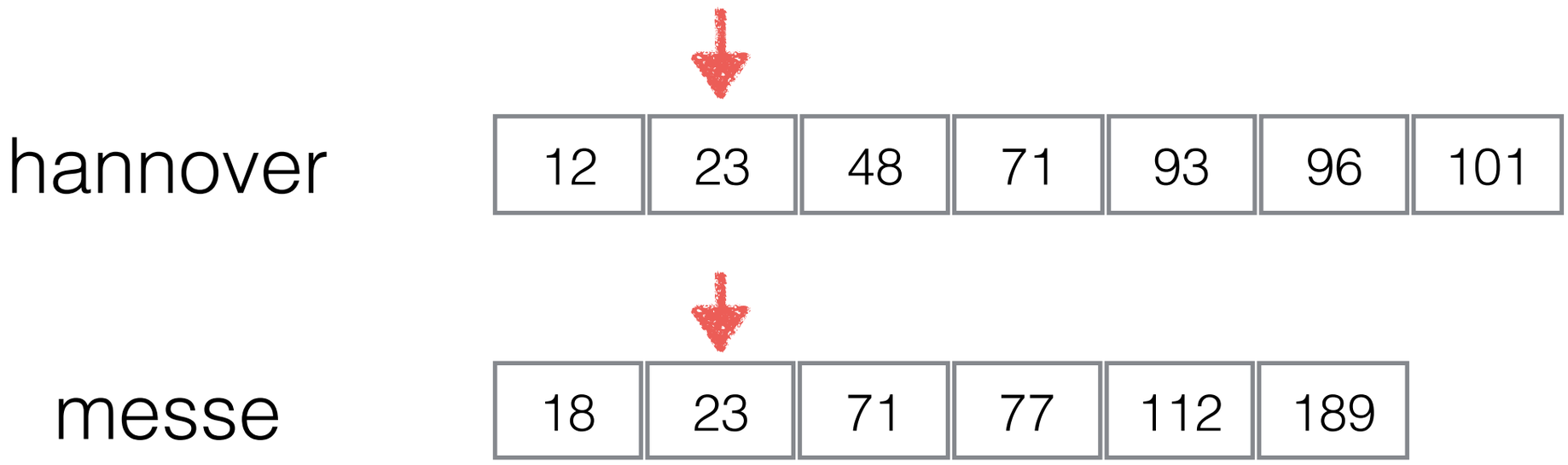
Document-at-a-time

- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

Conjunctive query semantics

- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further

Query Processing: Document-At-A-Time



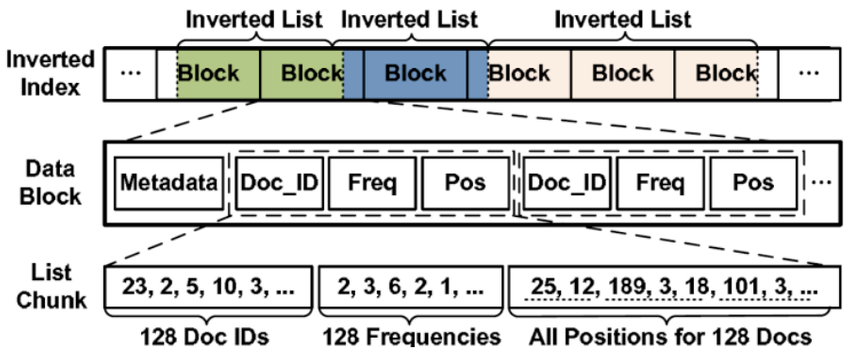
Document-at-a-time

- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

Conjunctive query semantics

- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further

Skip Lists



hannover

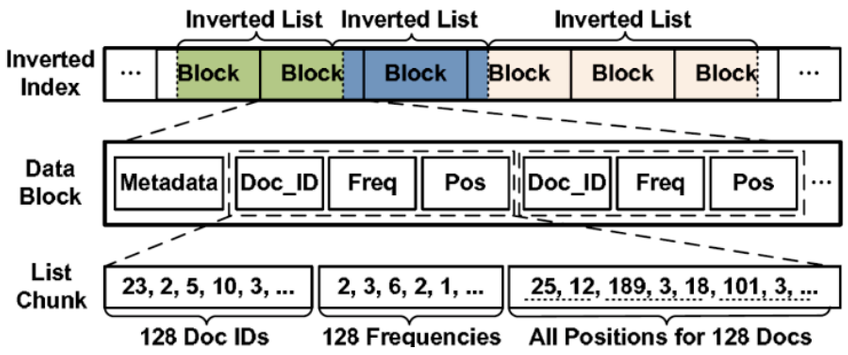
12	23	48	71	93	96	101	…

messe

18	23	71	105	112	189

What if we are allowed to store extra information for each lists ?

Skip Lists



hannover

12	23	48	71	93	96	101	…

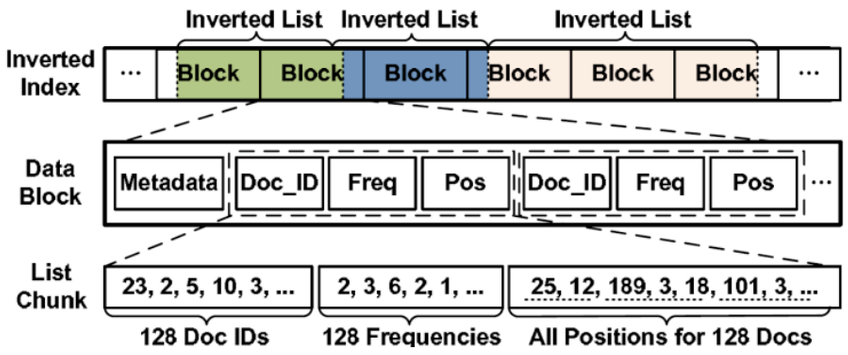
messe

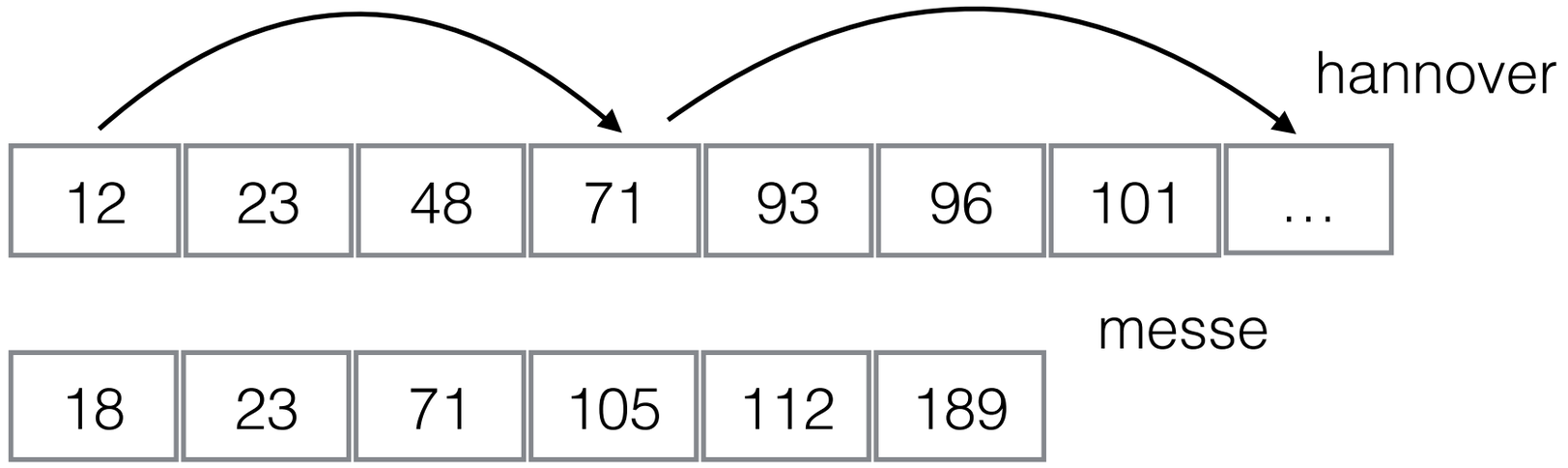
18	23	71	105	112	189

What if we are allowed to store extra information for each lists ?

- Skip list allow fast intersections acting as a secondary index over posting lists
- Typically skip over fixed number of postings and are square root of the length of the postings list

Skip Lists





What if we are allowed to store extra information for each lists ?

- Skip list allow fast intersections acting as a secondary index over posting lists
- Typically skip over fixed number of postings and are square root of the length of the postings list

Weak AND

- Generalized version of AND operation

$$\sum_{1 \leq i \leq k} x_i w_i \geq \theta,$$

for a query of k words

- Weights and indicator variables specify the degree of AND
- Threshold helps avoid computation of non important documents
- An instance of DAAT scoring

Weak AND or WAND

- Assume a special iterator on the postings of the form "go to the first docID greater than *X*"

Weak AND or WAND

- Assume a special iterator on the postings of the form "go to the first docID greater than *X*"

- Typical state: we have a "cursor" at some docID in the postings of each query term
  - Each cursor moves only to the right, to larger docIDs

Weak AND or WAND

- Assume a special iterator on the postings of the form "go to the first docID greater than *X*"

- Typical state: we have a "cursor" at some docID in the postings of each query term
  - Each cursor moves only to the right, to larger docIDs

- Invariant – all docIDs lower than any cursor have already been *processed*, meaning
  - These docIDs are either pruned away or
  - Their scores have been computed

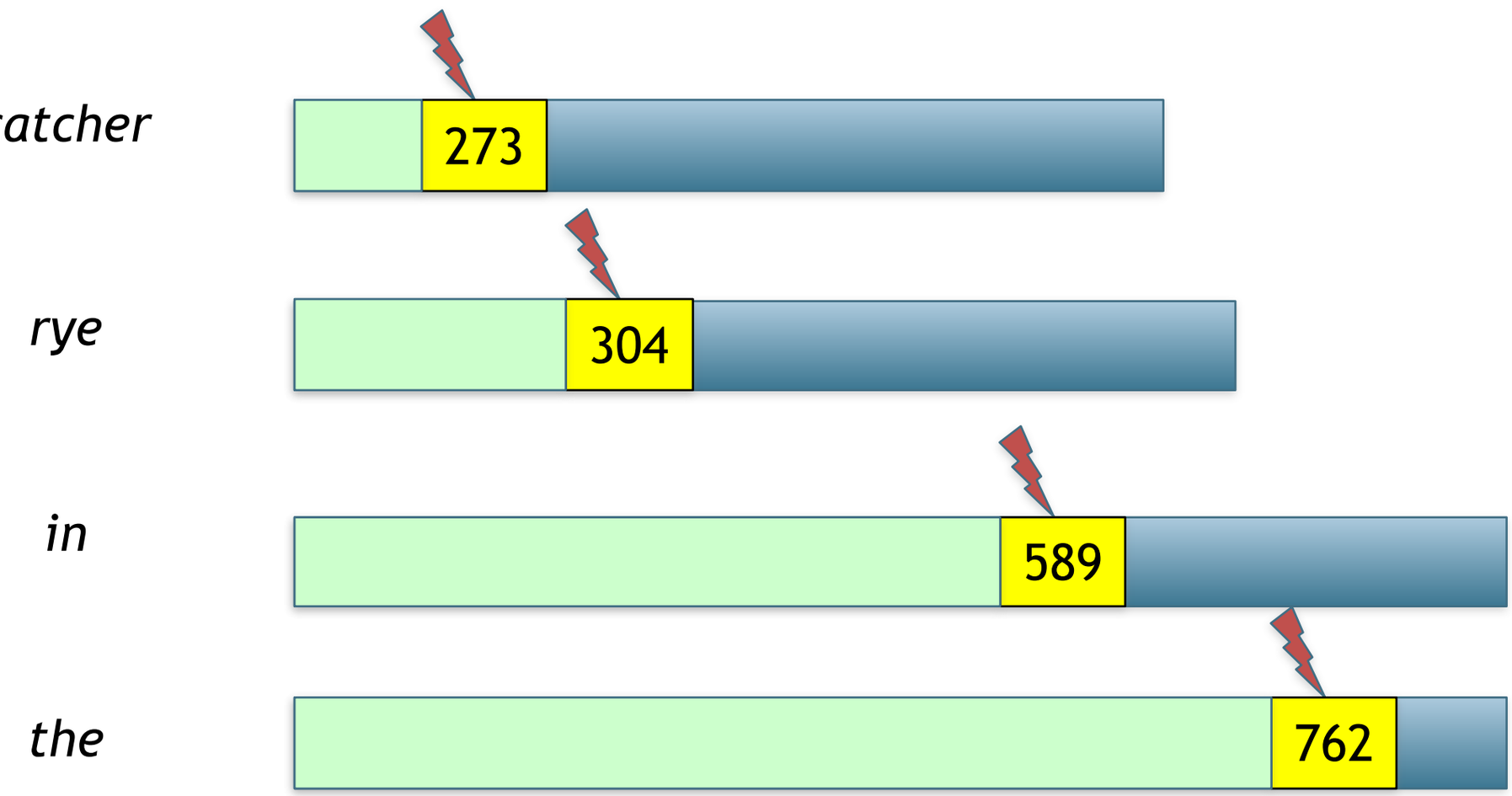
Weak AND

- At all times for each query term *t*, we maintain an *upper bound* UB_t on the score contribution of any doc to the right of the cursor
  - Max (over docs remaining in *t*'s postings) of w_t(doc)
  - As cursor moves right, the UB drops

UB("hannover") = 2.8

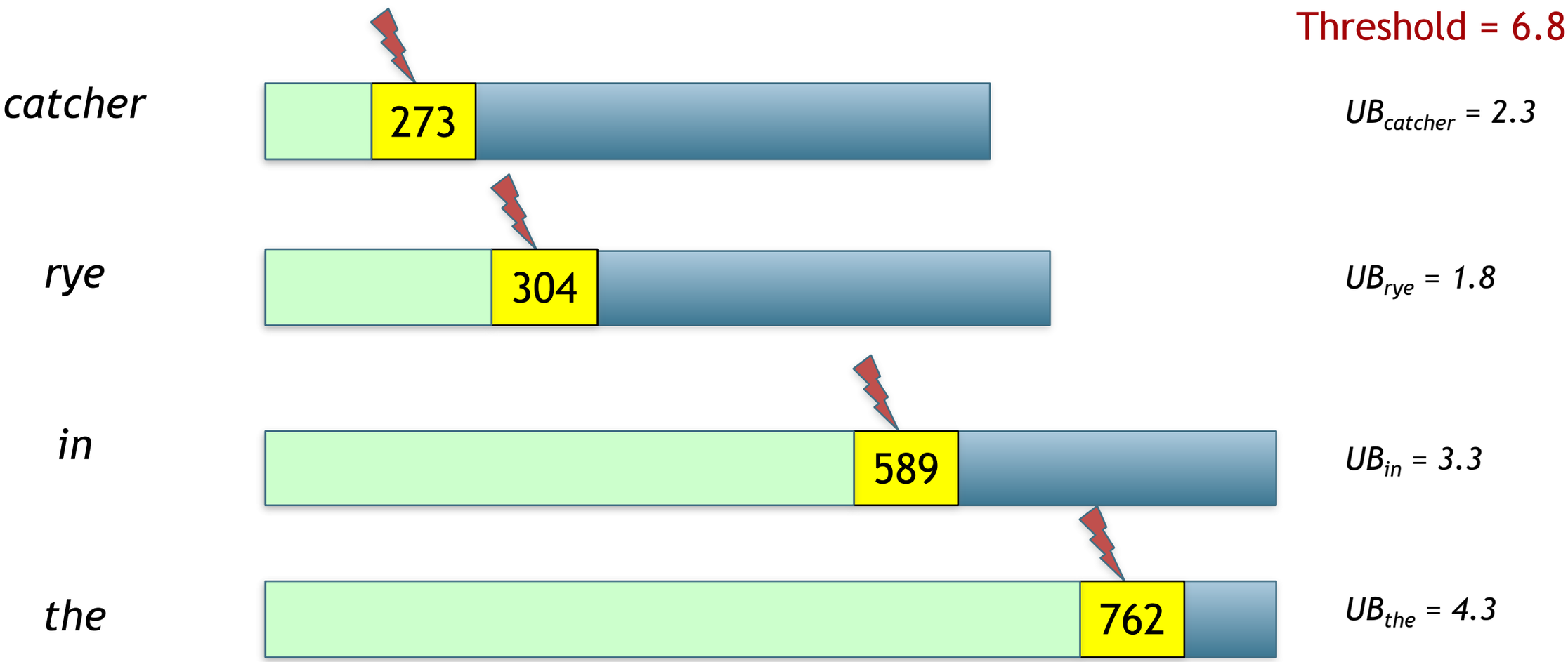
WAND - Pivoting

- Query: *catcher in the rye*
- Let's say the current cursor positions are as below



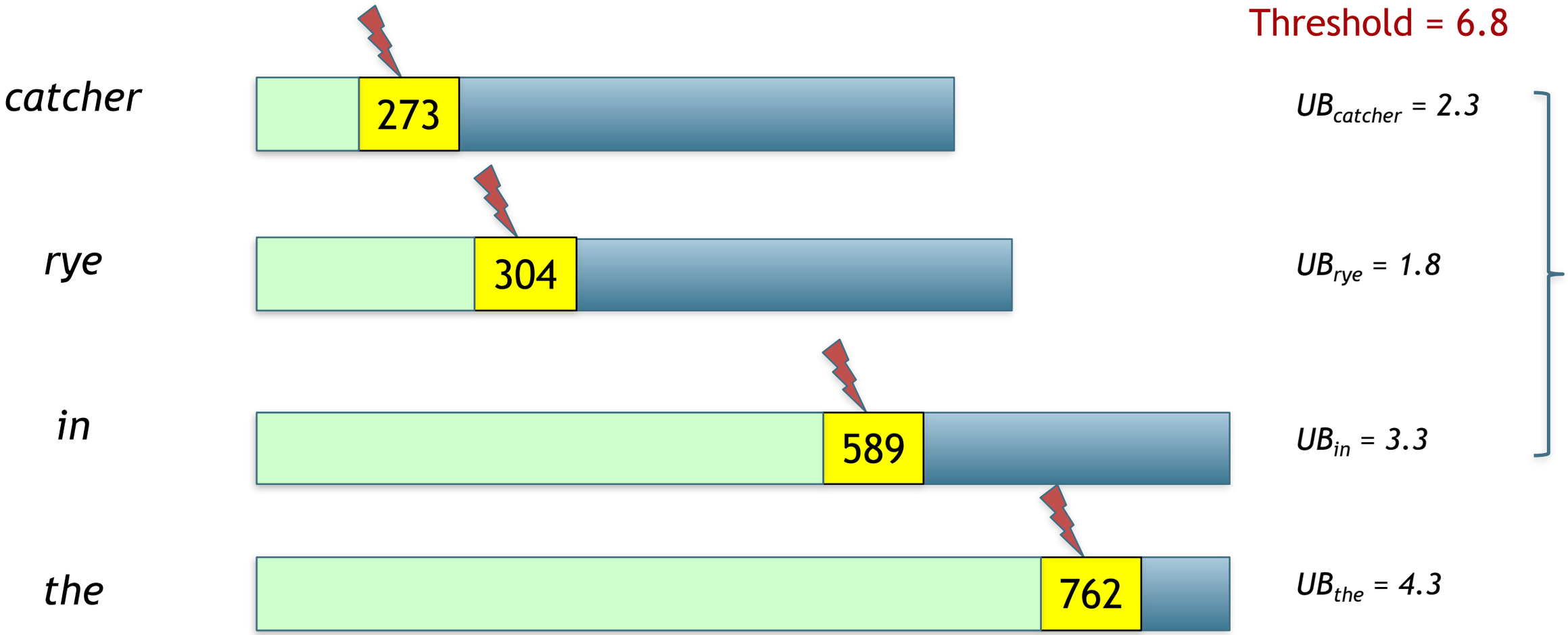
WAND - Pivoting

- Query: *catcher in the rye*
- Let's say the current cursor positions are as below



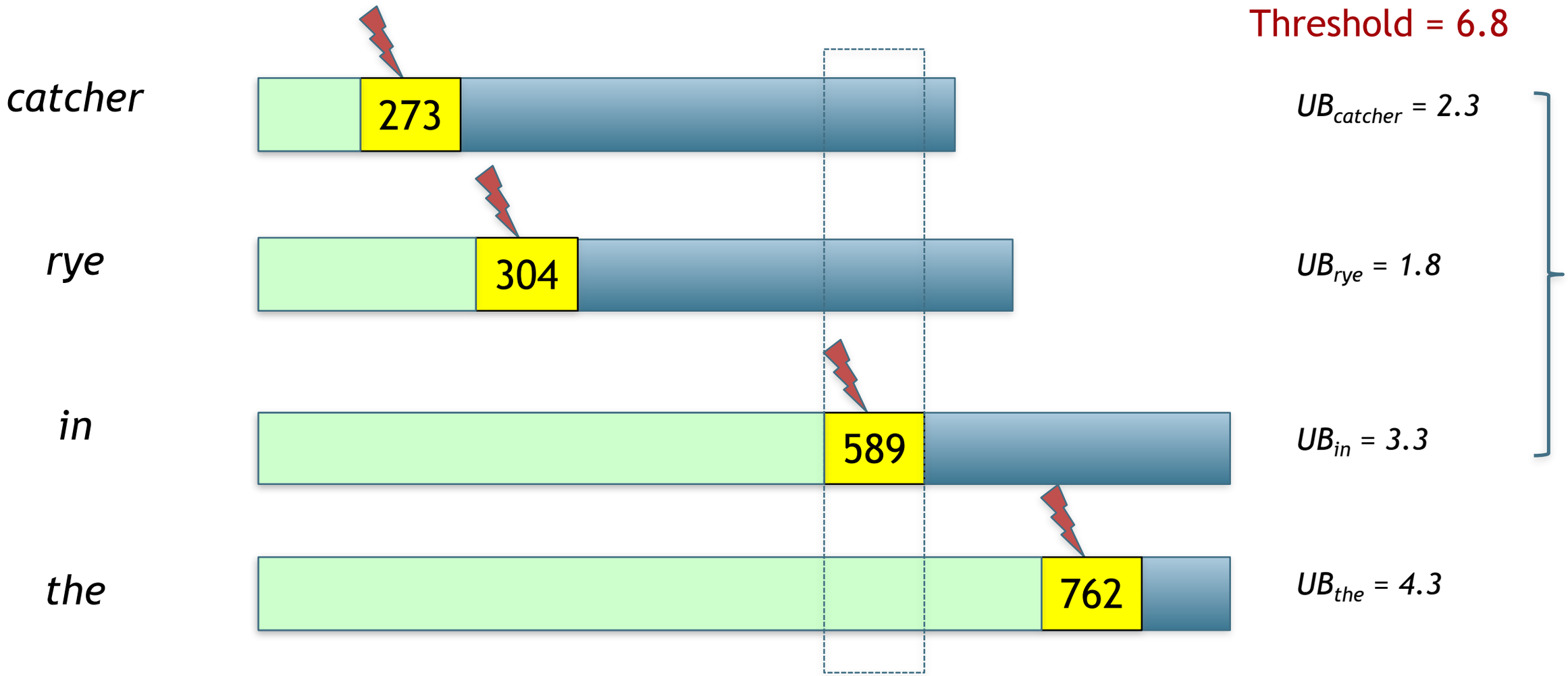
WAND - Pivoting

- Query: *catcher in the rye*
- Let's say the current cursor positions are as below



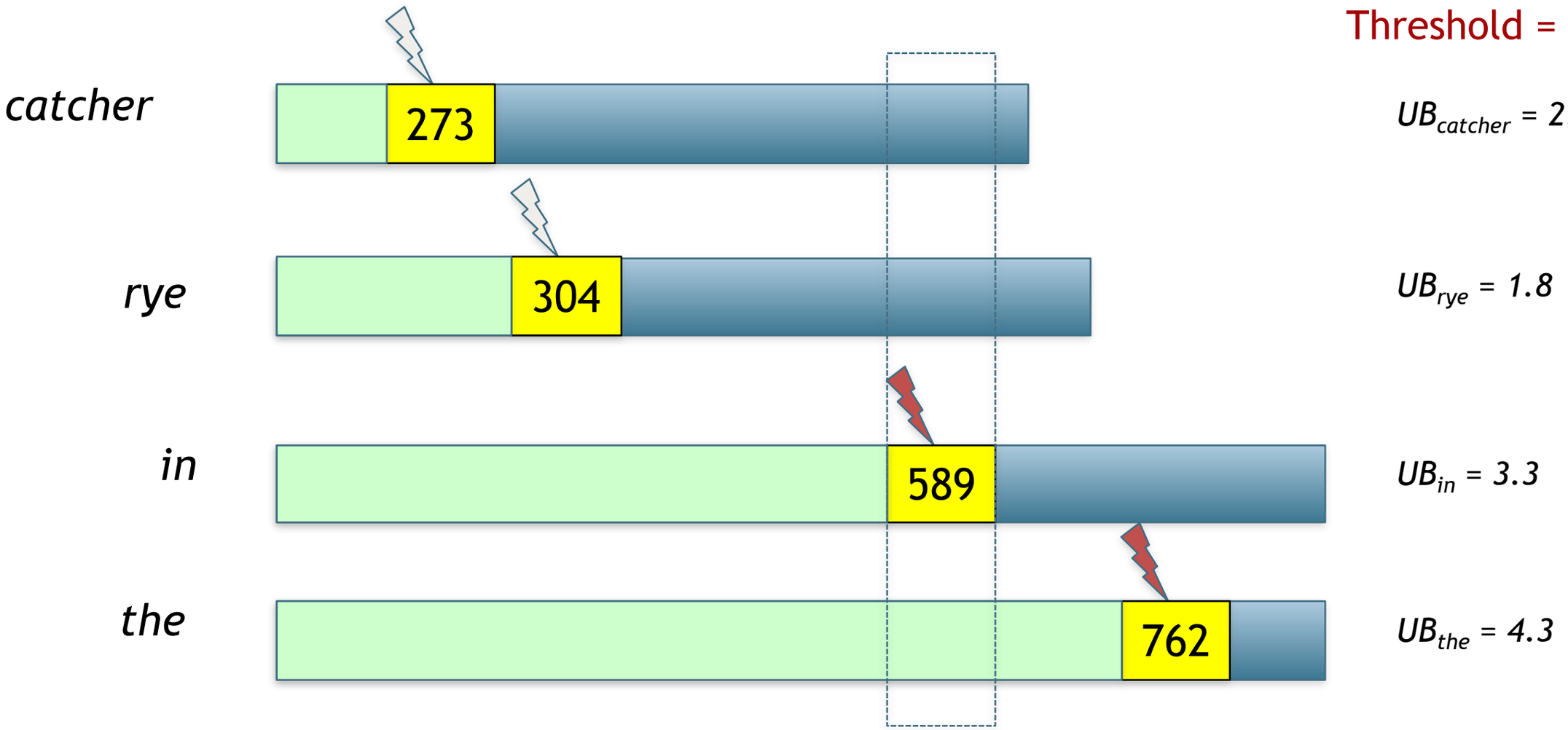
WAND - Pivoting

- Query: *catcher in the rye*
- Let's say the current cursor positions are as below



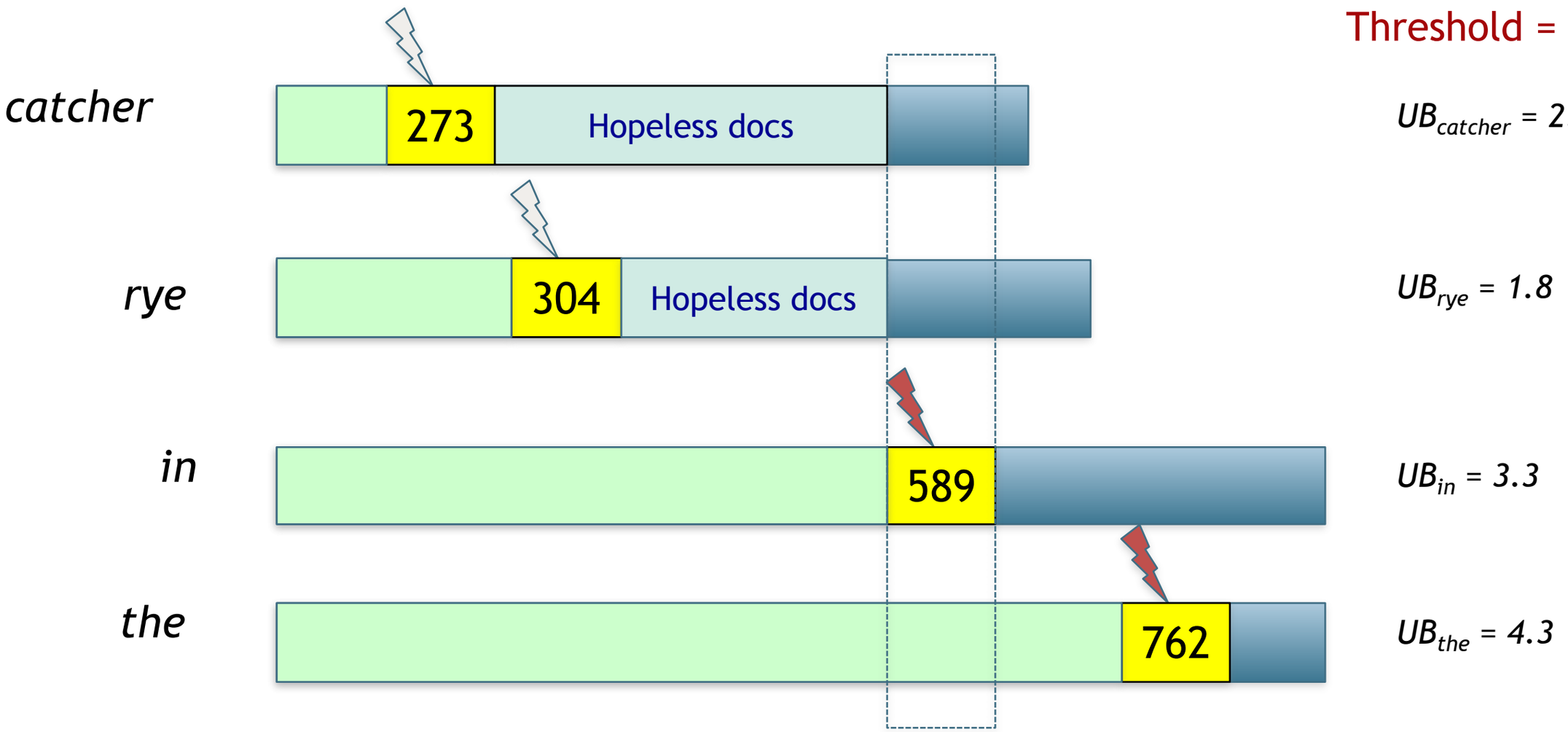
mWAND Algorithm

- Terms sorted in order of cursor positions
- Move cursor to 589 or right



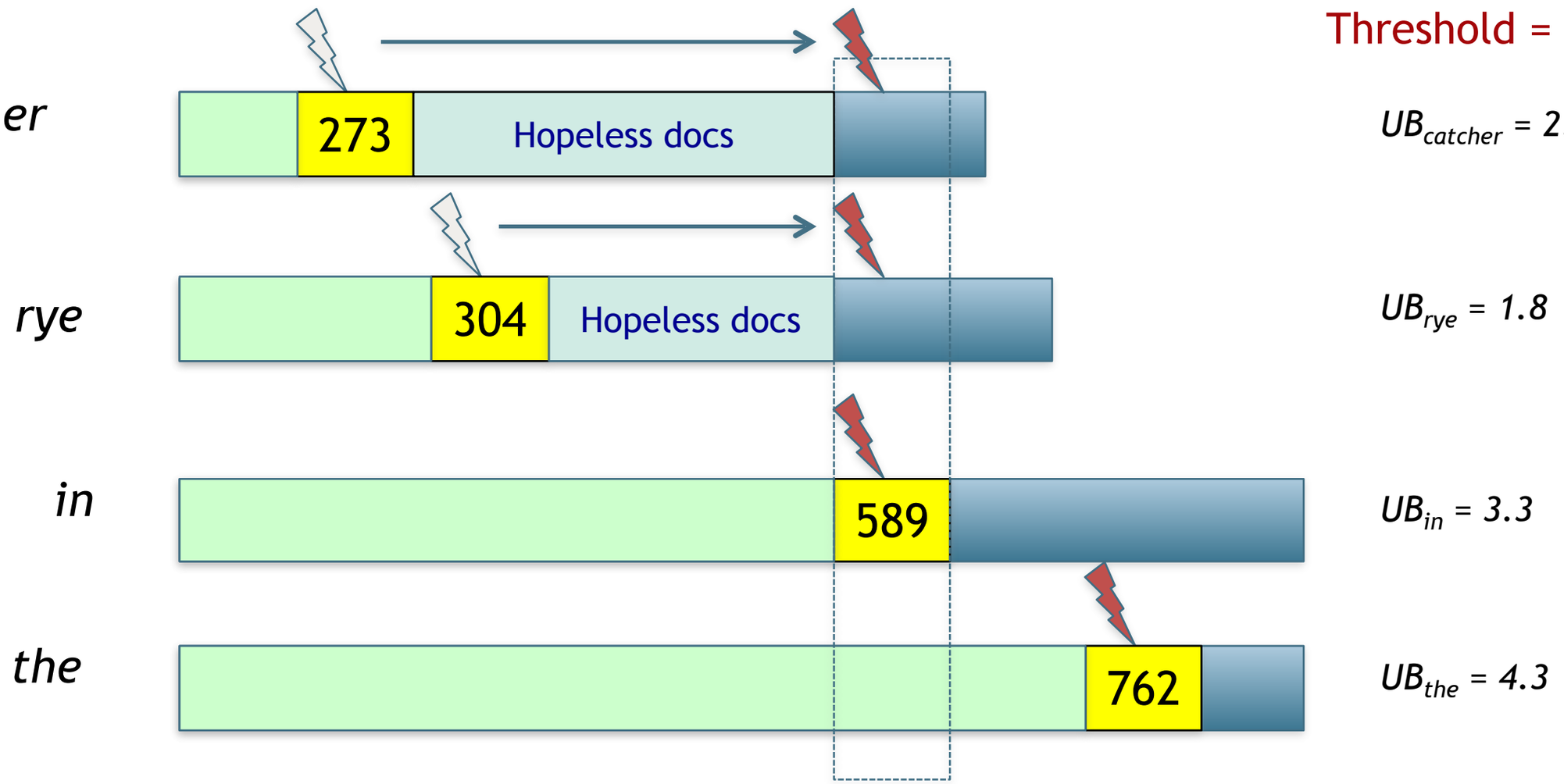
mWAND Algorithm

- Terms sorted in order of cursor positions
- Move cursor to 589 or right



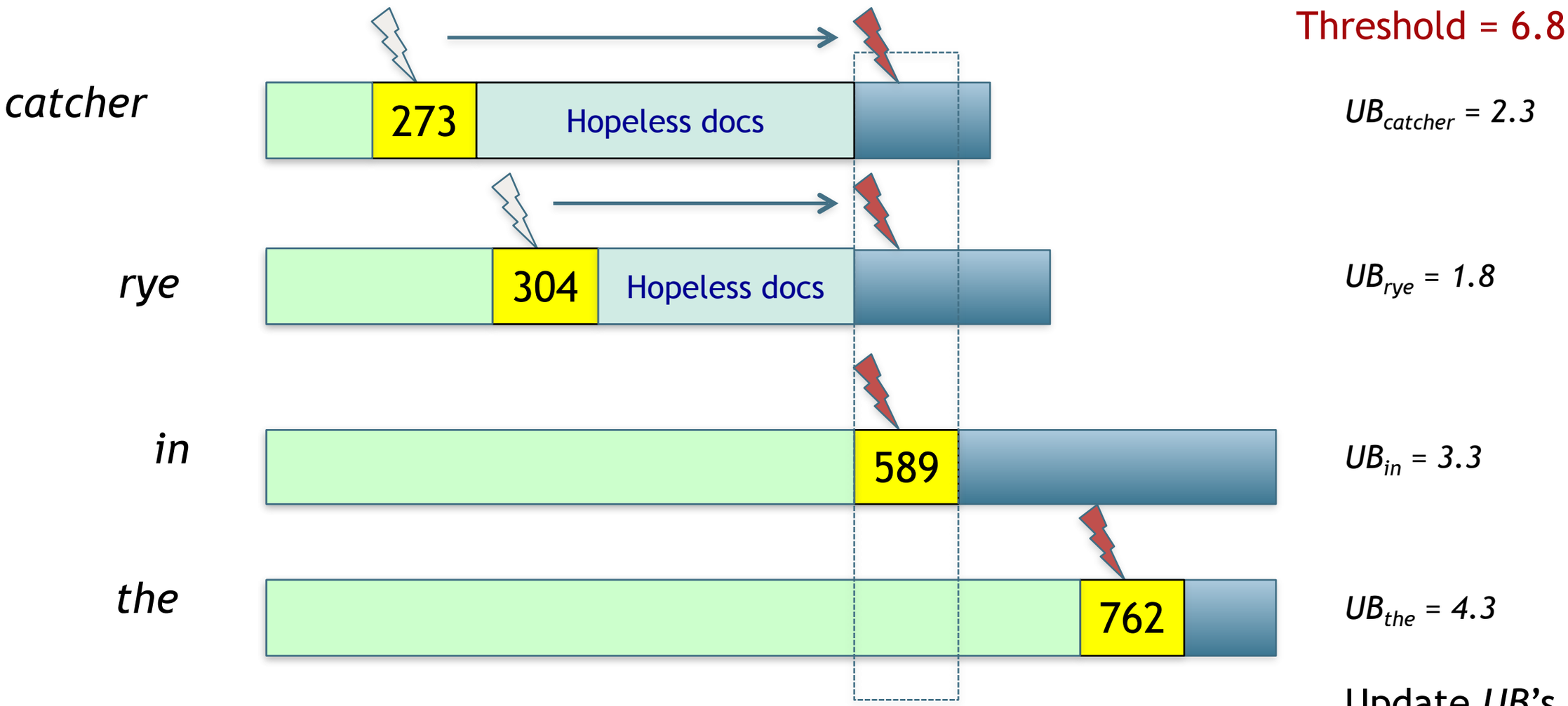
mWAND Algorithm

- Terms sorted in order of cursor positions
- Move cursor to 589 or right



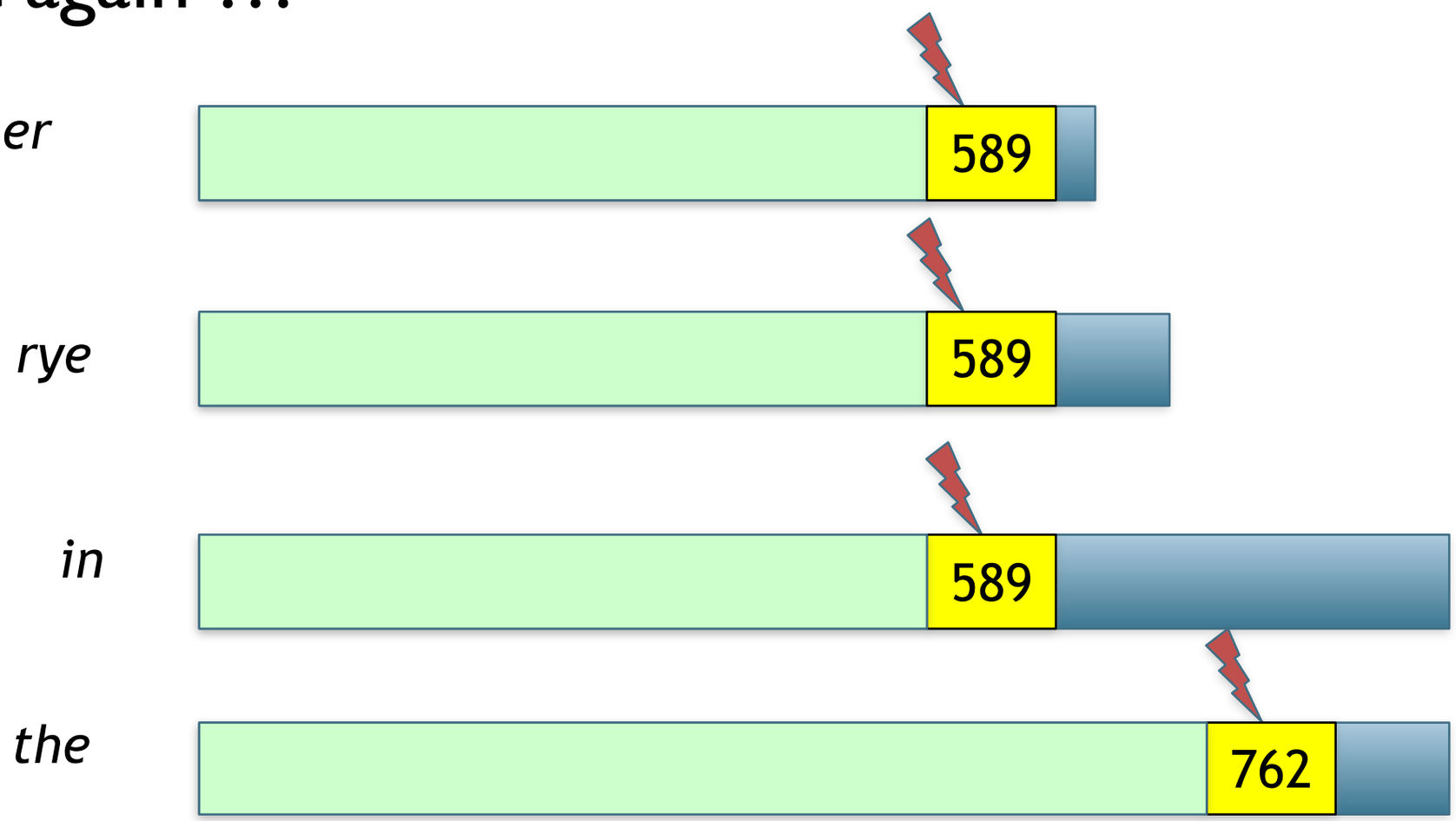
mWAND Algorithm

- Terms sorted in order of cursor positions
- Move cursor to 589 or right



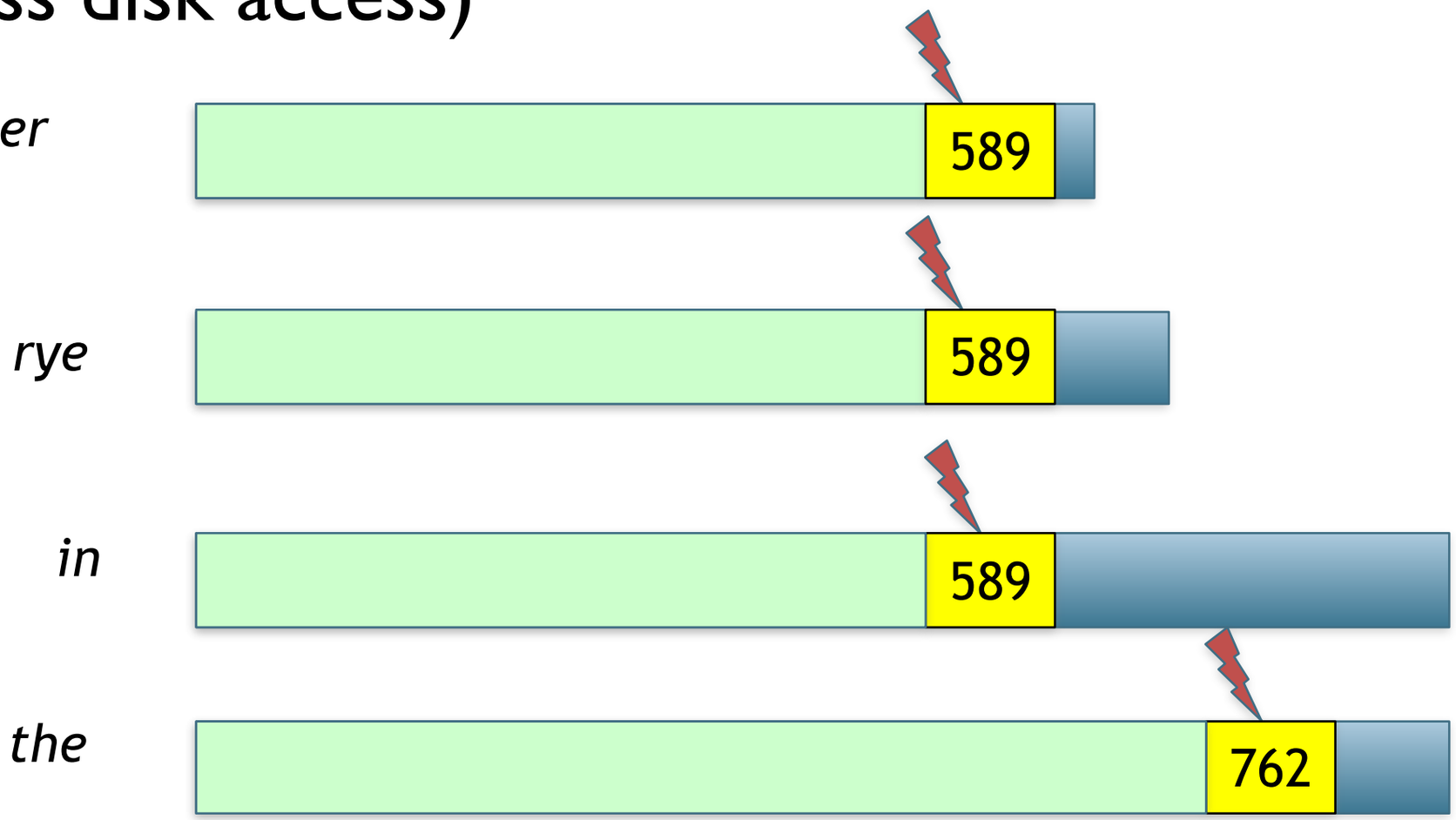
mWAND Algorithm

- If 589 is present in enough postings, compute its full cosine score – else some cursors to right of 589
- Pivot again …



WAND Algorithm

- Disk access typically more expensive
- Can you trade-off sorting and pivoting ?
  - Sorting cost (more CPU intensive) vs Pivoting cost (less disk access)



References and Further Readings

http://www.ir.uwaterloo.ca/book/

Broder et al. Efficient Query Evaluation using a Two-Level Retrieval Process.
http://www.cis.upenn.edu/~jstoy/cis650/papers/WAND.pdf

Fontoura, Marcus, et al. "Evaluation strategies for top-k queries over memory-resident inverted indexes." *Proceedings of the VLDB Endowment* 4.12 (2011): 1213-1224.

http://fontoura.org/papers/vldb2011.pdf