Information Retrieval and Data Mining

Summer Semester 2015
TU Kaiserslautern

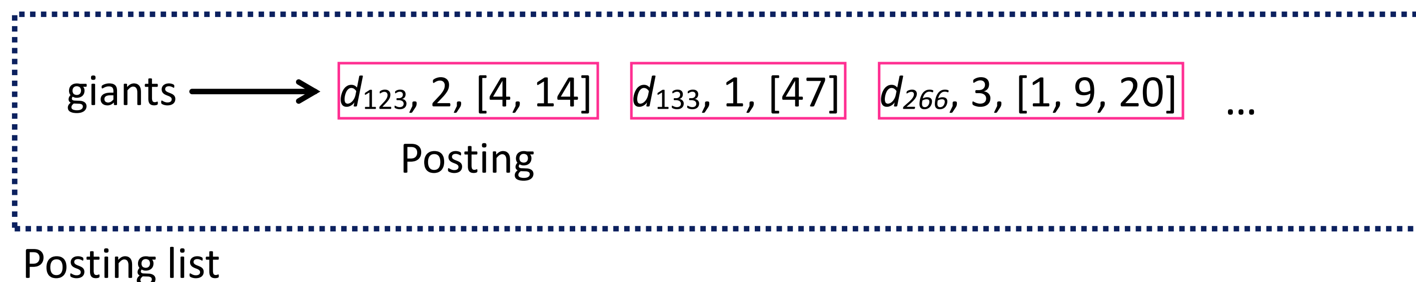
Prof. Dr.-Ing. Sebastian Michel

Databases and Information Systems Group (AG DBIS)

http://dbis.informatik.uni-kl.de/

Recap: Inverted Index

- **Inverted index keeps a posting list for each term**, which usually reside on secondary storage, with **each posting capturing information about term's occurrences in a specific document**
  - **document identifier** (e.g., d_{123}, d_{234}, …)
  - **term frequency** (e.g., tf(house, d_{123}) = 2, tf(house, d_{234}) = 4)
  - **score impacts** (e.g., tf(house, d_{123}) * idf(house) = 3.75)
  - **offsets** (i.e., absolute positions at which the term occurs in the document)



- Posting lists are **usually compressed** for time and space efficiency

3. Variable-Byte Encoding

- 32-bit binary code represents 12,038 using 4 bytes as

00000000	00000000	00101111	00000110

- **Variable-byte encoding** (aka. 7-bit encoding) uses one bit per byte as a **continuation bit** indicating whether the current number expands into the next bytes
- Variable-byte encoding represents 12,038 using only 2 bytes as

01011110	10000110

1 continuation bit

7 data bits

- Byte-aligned, i.e., each number corresponds to sequence of bytes

4. Gamma Encoding

- **Gamma (γ) encoding** represents an integer x as
  - length = $\lfloor \log_2 x \rfloor$ in **unary**
  - offset = $x - 2^{length}$ in **binary**
- results in $(1 + \log_2 x + \log_2 x)$ bits for integer x

- **Not byte-aligned**, i.e., needs to be packed into bytes or words

- **Useful when distribution of numbers is not known ahead of time**
  or **when small numbers (e.g., gaps, tf) are frequent**

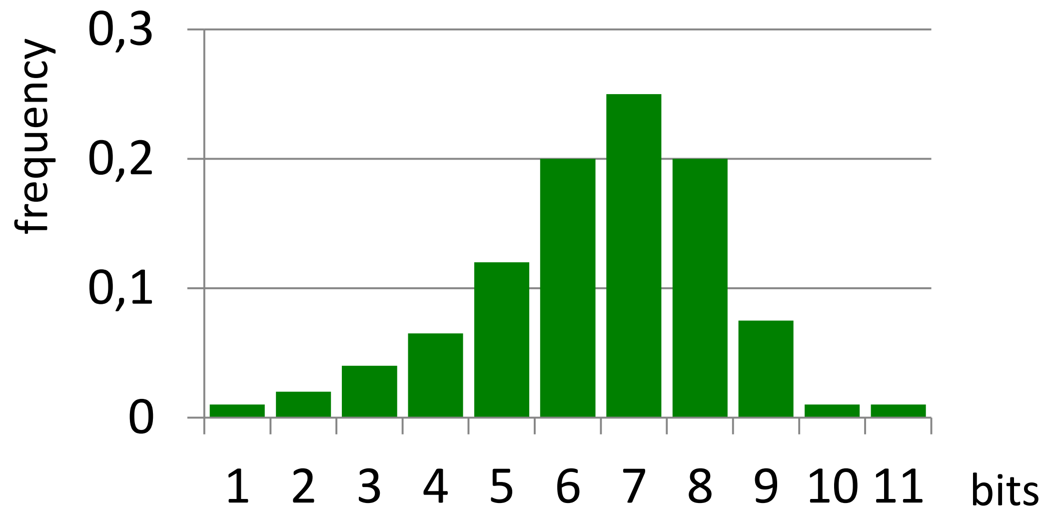
Gamma Encoding (Examples)

x	Gamma Encoding	
$1 = 2^0$	u:0	
$4 = 2^2$	u:110	b:00
$24 = 2^4 + 2^3$	u:11110	b:1000
$131 = 2^7 + 3$	u:11111110	b:0000011

Decoding: Consider 111101000..... Read from left side until first 0, here, 11110, which tells offset=4 bits. The next 4 bits are 1000 which is 8 in decimal. $8 + 2^4 = 24$

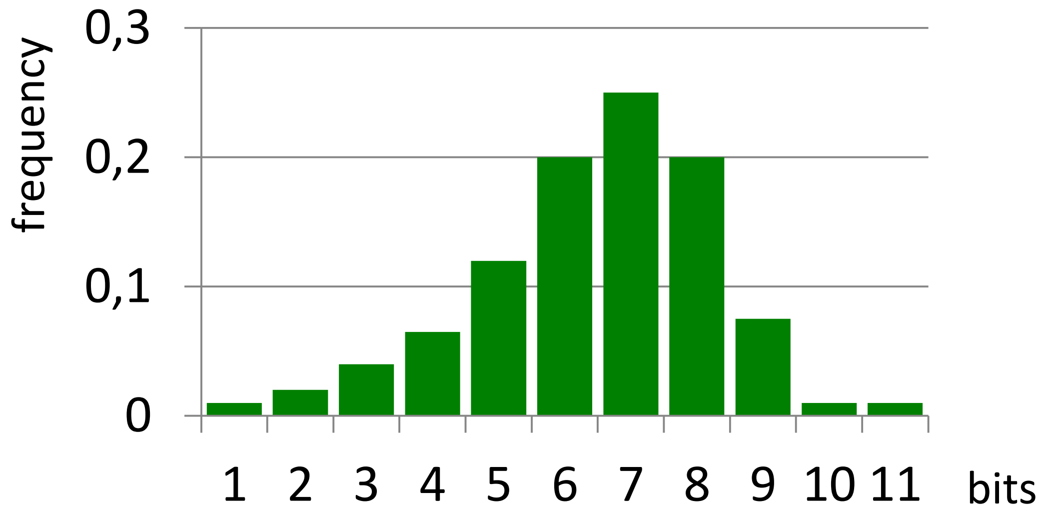
5. Golomb/Rice Encoding

- Assume we know the following frequency (y-axis) distribution of values written by the number of bits they require (x-axis)



- How do we encode? 11 bits for all numbers? (or even max is not known)
- Observation: There are many that require 7 bits only

Golomb/Rice Encoding



- Idea in Golomb encoding: write a number x as **quotient** q = floor(x / M) and **remainder** r = (x mod M)
- Here $M = 2^7$
- We see, here: quotient can be written in less than 3 bits (unary)
- **What if we use a smaller M**, say $M = 2^3$?

Golomb/Rice Encoding

- For **tunable parameter M**, split the number x into
  - **quotient** q = floor(x / M) stored in unary code (using q + 1 bits)
  - **remainder** r = (x mod M) stored in binary code
- If M chosen as 2^n then r needs $\log_2(M)$ bits (**Rice encoding**)
- Otherwise for **b = ceil($\log_2$(M))**
  - If $r < 2^b - M$ then r is stored in binary code using b - 1 bits
  - Otherwise $r + 2^b - M$ is stored in binary code using b bits

- **Not byte-aligned**, i.e., needs to be packed into bytes or words
- **Useful when distribution of numbers is known ahead of time (allows finding "optimal" parameter M)**

Golomb/Rice Encoding (Examples)

Golomb Encoding (M = 10, b = 4)

x	q	bits(q)	r	bits(r)
0	0	u:0	0	b:000
33	3	u:1110	3	b:011
57	5	u:111110	7	b:1101
99	9	u:1111111110	9	b:1111

Decoding: given one large bit string…

$2^b - M = 16-10=6$

11111011011110011111.....

First we read unary 111110 (indicates = 5), then we read b-1=3 bits, which gives 110b (=6)
That means we are in the case of reading 4 bits for r (the „otherwise" case on slide before).
So we read in total 1101b (=13) and obtain as 13-6 = 7 as value for r.
Then we read unary 1110 (indicates 3), then again we read b-1=3 bits to get 011b (=3). Since 3<6, we know we dont have to read one more bit, and obtain r=3.

6. Gap Encoding

- Variable-byte encoding, Gamma encoding, and Golomb/Rice encoding **represent smaller numbers using fewer bytes**

- **Note:** Posting lists contain sequences of increasing integers
  - document identifiers of postings in document-ordered posting list
  - offsets in posting payload if phrase queries need to be supported

- **Gap encoding** (aka. d-gaps) represents sequences of increasing integers as their first element followed by gaps

<7, 12, 20, 25, 33, 78, … > ···············▸ <7, 5, 8, 5, 8, 45, … >

7. Run-Length Encoding

- Run-length encoding (e.g., used in early image formats like PCX) targets **sequences of integers having long runs of the same number** (i.e., **many repetitions** of that number in a row)
- **Run-length encoding represents integer sequences as (number, frequency) pairs**

<7, 7, 7, 8, 8, 1, 1, 1, 1, … > ·······► < (7, 3), (8, 2), (1, 4), …>

Summary of III.2

- **Compression**
  is essential for performance in modern IR systems
- **Ziv-Lempel compression**
  as a dictionary-based encoding scheme that is great for text
- **Variable-byte encoding**
  as a byte-aligned non-parameterized encoding
- **Gamma encoding and Golomb/Rice encoding**
  as bit-aligned non-parameterized/parameterized encodings
- **Gap encoding and Run-length encoding**
  for transforming integer sequences

Additional Literature for III.2

- S. Brin and L. Page: The anatomy of a large-scale hypertextual Web search engine,
  Computer Networks 30:107-117, 1998
- J. Dean: Challenges in Building Large-Scale Information Retrieval Systems, WSDM 2009, http://videolectures.net/wsdm09_dean_cblirs/
- A. Moffat and L. Stuiver: Binary Interpolative Coding for Effective Index Compression, Inf. Retr. 3(1): 25-47 (2000)
- H. Yan, S. Ding, T. Suel: Compressing Term Positions in Web Indexes, SIGIR 2009
- H. Yan, S. Ding, T. Suel: Inverted index compression and query processing with optimized document ordering, WWW 2009
- I. Witten, A. Moffat, and T. Bell: Managing Gigabytes (2nd Edition), Morgan Kaufmann, 1999
- J. Zhang, X. Long, T. Suel: Performance of compressed inverted list caching in search engines, WWW 2008
- M. Zukowski, S. Héman, N. Nes, P. A. Boncz: Super-Scalar RAM-CPU Cache Compression, ICDE 2006

III.3 Query Processing

1. **Term-at-a-Time**
2. **Document-at-a-Time**
3. **WAND**
4. **Quit & Continue**
5. **Buckley’s Algorithm**
6. **Fagin’s Threshold Algorithms**

Based on MRS Chapter 7 and RBY Chapter 9

Query Types

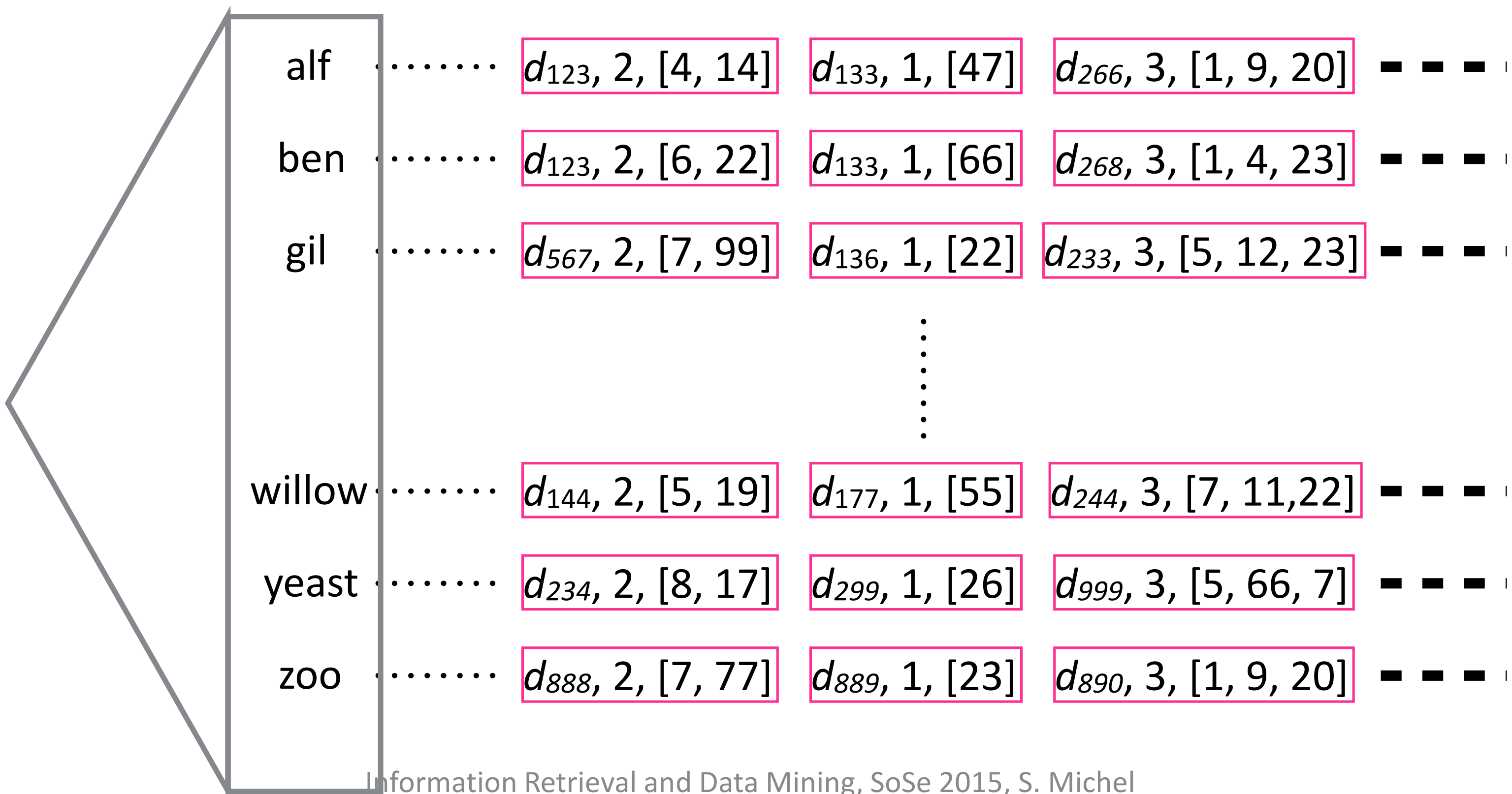
- **Conjunctive**
  (i.e., all query terms are required)
- **Disjunctive**
  (i.e., subset of query terms sufficient)
- **Phrase or proximity**
  (i.e., query terms must occur in right order or close enough)
- **Mixed-mode with negation**
  (e.g., “harry potter” review +movie -book)
- Combined with ranking of result documents according to

$$score(q,d) = \sum_{t \in q} score(t,d)$$

with score(t, d) depending on retrieval model (e.g., $\text{tf.idf}_{t,d}$)

Inverted Index

- **Document-ordered** or **score-ordered posting lists**
- **Posting lists** with skip pointers allow for faster traversal



Overview Query Processing Methods

- **Holistic query processing methods** determine whole query result
  - Term-at-a-Time
  - Document-at-a-Time

- **Top-k query processing methods** determine top-k query result
  - WAND
  - Quit & Continue
  - Fagin's Threshold Algorithms

- **Opportunities for optimization** over naïve merge & sort baseline
  - skipping in document-ordered posting lists
  - early termination of query processing for score-ordered posting lists

1. Term-at-a-Time Query Processing

- **Term-at-a-Time (TaaT) query processing**
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ successively
  - **maintains an accumulator for each result document with value**

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$

after the first j posting lists have been read

a ········ d_1, 1.0 | d_4, 2.0 | d_7, 0.2 | d_8, 0.1

b ········ d_4, 1.0 | d_7, 2.0 | d_8, 0.2 | d_9, 0.1

c ········ d_4, 3.0 | d_7, 1.0

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.2
d_8	:	0.1
d_9	:	0.0

  - required memory depends on the number of accumulators maintained
  - top-k results can be determined by sorting accumulators at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time (TaaT) query processing**
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ successively
  - **maintains an accumulator for each result document with value**

    $acc(d) = \sum_{i \leq j} score(t_i, d)$ after the first j posting lists have been read

a ········ d_1, 1.0 | d_4, 2.0 | d_7, 0.2 | d_8, 0.1

b ········ d_4, 1.0 | d_7, 2.0 | d_8, 0.2 | d_9, 0.1

c ········ d_4, 3.0 | d_7, 1.0

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.1

  - required memory depends on the number of accumulators maintained
  - top-k results can be determined by sorting accumulators at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time (TaaT) query processing**
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ successively
  - **maintains an accumulator for each result document with value**

    $acc(d) = \sum_{i \le j} score(t_i, d)$ after the first j posting lists have been read

a ········ d_1, 1.0 | d_4, 2.0 | d_7, 0.2 | d_8, 0.1

b ········ d_4, 1.0 | d_7, 2.0 | d_8, 0.2 | d_9, 0.1

c ········ d_4, 3.0 | d_7, 1.0

Accumulators

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - required memory depends on the number of accumulators maintained
  - top-k results can be determined by sorting accumulators at the end

Term-at-a-Time Query Processing

- **Optimizations for conjunctive queries**
  - **process query terms in ascending order of their document frequency** (i.e., the length of the list) to keep the number of accumulators and thus required memory low
  - **for document-ordered posting lists, keep accumulators sorted**
    to make use of skip pointers when read posting lists

2. Document-at-a-Time Query Processing

- Document-at-a-Time (DaaT) query processing
  - assumes document-ordered posting lists
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ concurrently
  - computes score when same document is seen in one or more posting lists

a ········ d_1, 1.0 | d_4, 2.0 | d_7, 0.2 | d_8, 0.1

b ········ d_4, 1.0 | d_7, 2.0 | d_8, 0.2 | d_9, 0.1

c ········ d_4, 3.0 | d_7, 1.0

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - always advances posting list with lowest current document identifier
  - required main memory depends on the number of results to be reported
  - top-k results can be determined by keeping results in priority queue

Document-at-a-Time Query Processing

- **Optimization for conjunctive queries using skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier larger or equal to $\max_i cdid(i)$

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0
d_7	:	3.2

where cdid(i) is the current document identifier in the i-th posting list

3. WAND

- **Weak AND (WAND) query processing**
  - **assumes document-ordered posting lists** with
  - **known maximum score maxscore(i) of any posting in the i-th posting list**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ concurrently
  - computes score when same document is seen in one or more posting lists
  - always **advances posting list with lowest current document identifier** **up to pivot** document identifier computed from current top-k result

WAND

- Computation of pivot document identifier
  - let min_k denote the lowest score in current top-k results
  - sort posting lists in ascending order of cdid(i)
  - pivot is cdid(j) of minimal j such that $min_k < \sum_{i \leq j} maxscore(i)$

a	········	d_2, 0.5	d_7, 0.1	d_8, 0.2	d_9, 0.6	·······	d_{99}, 1.0	·······
b	········	d_2, 0.5	d_9, 0.3	d_{11}, 0.2	d_{13}, 0.1	·······	d_{33}, 1.0	·······
c	········	d_2, 0.5	d_3, 0.4	d_4, 0.2	d_5, 0.1	·······	d_{57}, 1.0	·······

Top-1

d_2	:	1.5

Pivot Computation

d_3, 0.4	1.0
d_7, 0.1	2.0
d_9, 0.3	3.0

$maxscore(i) = 1.0$

d_7 is pivot

WAND

- **Intuition:** No document with an identifier smaller than the pivot
  can have a score large enough to make it into the top-k result

- **Observation:** As the value of min_k can only increase over time, WAND skips more and more postings as time progresses

- **WAND can be made an approximate top-k query processing method** by computing the pivot such that

$$F \times min_k < \sum_{i \leq j} maxscore(i)$$

  with tunable parameter **F controlling fidelity of results**

- Full details: [Broder et al. '03]

4. Quit & Continue

- **Quit & Continue query processing**
  - reads score-ordered posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ successively in descending order of $idf(t_i)$
- **Quit heuristics**
  - ignore posting lists for terms t_i with $idf(t_i)$ below threshold
  - stop scanning posting list for t_i if $tf(t_i, d_j)*idf(t_i)$ drops below threshold
  - stop scanning posting list when the number of accumulators is too high
- **Continue heuristics**
  - upon reaching accumulator limit, continue reading remaining posting lists, update existing accumulators but do not create new accumulators

- Full details: [Moffat and Zobel '96]

5. Buckley's Algorithm

- Buckley's query processing method
  - **reads score-ordered posting lists** concurrently in round-robin manner
  - maintains **partial scores** of documents and keeps track of k-th best score
  - **computes upper bound for any unseen document based on current scores** $ub = \sum_i cscore(i)$ with cscore(i) as the current score in the i-th posting list
  - stops if upper bound ub is less than k-th best partial score

a ········	d_2, 0.5	d_1, 0.4	d_5, 0.3	d_9, 0.2	·······
b ········	d_2, 0.5	d_3, 0.5	d_{61}, 0.4	d_{13}, 0.1	·······
c ········	d_3, 0.4	d_5, 0.3	d_7, 0.2	d_4, 0.1	·······

Top-1

d_2	:	1.0

$ub = 0.9$

[Buckley and Lewitt '85]

6. Fagin's Threshold Algorithms

- **Threshold Algorithm (TA)**
  - original version, often used as synonym for entire family of algorithms
  - requires eager random access to candidate objects
  - worst-case memory consumption: O(k)
- **No-Random-Accesses (NRA)**
  - no random access required, may have to scan large parts of the lists
  - worst-case memory consumption: O(m*n + k)
- **Combined Algorithm (CA)**
  - cost-model for scheduling random accesses to candidate objects
  - algorithmic skeleton very similar to NRA, but typically terminates faster
  - worst-case memory consumption: O(m*n + k)

m=number of lists
n=docs per list

R. Fagin, A. Lotem, and M. Naor: Optimal Aggregation Algorithms for Middleware, Journal of Computer and System Sciences 2003

Fagin's Threshold Algorithms

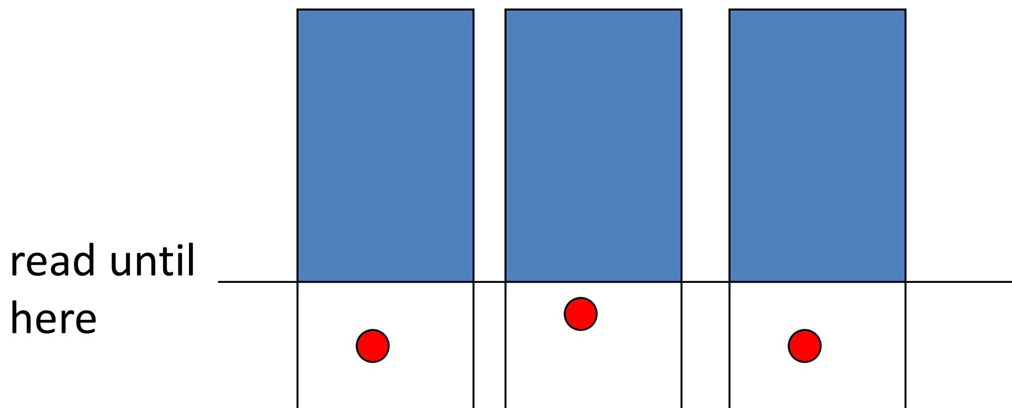
- Assume **score-ordered posting lists** and additional index for score look-ups by document identifier
- **Scan posting lists using inexpensive sequential accesses (SA) in round-robin manner**
- **Perform expensive random accesses (RA) to look up scores for a specific document when beneficial**
- Support **monotone score aggregation function**

$$aggr : \mathbb{R}^m \rightarrow \mathbb{R} \; : \; \forall x_i \geq x'_i \Rightarrow aggr(x_1, \ldots, x_m) \geq aggr(x'_1, \ldots, x'_m)$$

- **Compute aggregate scores incrementally in candidate queue**
- Compute score bounds for candidate results and stop when **threshold test** guarantees correct top-k result

Monotonicity Explained

- Means: if item a is below item b in all considered index lists (that is, its score is smaller), it cannot have a final score higher than the one of b.
- Assume we have read already from the lists and have the following situation



All items seen already in all three lists have aggr. score higher than “red-dot” item

Threshold Algorithm (TA)

- **Sequential accesses (SA) mixed with eager random accesses (RA)**
- Worst-case memory consumption O(k)

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
    look up $score(t_j, d)$ for all $j \neq i$
    $score(d) = aggr\{ score(t_j, d) \mid j = 1 \ldots |q| \}$

if $score(d) >$ min-k **then** // update top-k
    add d to top-k and remove min-score d'
    $min_k = min\{ score(d') \mid d' \in$ top-k $\}$

// update upper bound from current scan line
$ub = aggr\{high(i) \mid i = 1 \ldots |q|\}$
if $ub \leq min_k$ **then**
    exit

Threshold Algorithm (TA)

- Read from index lists in sequential order
- Lookup immediately missing scores (zero if not in list)
- Stop if seen at least k objects with aggregated score higher than aggregated “scan line”

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 1

- Start seq. scanning. See doc3, lookup its score in list 2 and 3. Get:
- doc3: 18+7+12 = 37

Top-1 query

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 2

- Continue with doc1 seen in list 2.
- doc1: 0+9+19 = 28
- doc3: 18+7+12 = 37

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 3

- doc3: 18+7+12 = 37
- doc1: 0+9+19 = 28
- Scan line scores: 18+9+19=46
- We cannot stop now, why?

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 4

- doc3: 18+7+12 = 37
- doc1: 0+9+19 = 28; doc4 = 12+0+15= 27
- Scan line scores: 12+9+19=40
- We cannot stop now, why?

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 5

- doc3: 18+7+12 = 37; doc1: 0+9+19 = 28; doc4: 12+0+15 = 27
- Scan line scores: 12+7+19=38
- Still cannot stop!

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Step 6

- doc3: 18+7+12 = 37; doc1: 0+9+19 = 28; doc4: 12+0+15 = 27
- Scan line scores: 12+7+15 = 34
- We can stop!

document	score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

document	score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

document	score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Restriction of Random Accesses

- Recall the following numbers:
  - Disk seek 10,000,000 ns
  - Read 1 MB sequentially from disk 30,000,000 ns

- Variations of threshold algorithms that consider tradeoff between random and sequential accesses
- Or prohibit random accesses at all

  => No Random Access (NRA) Algorithm