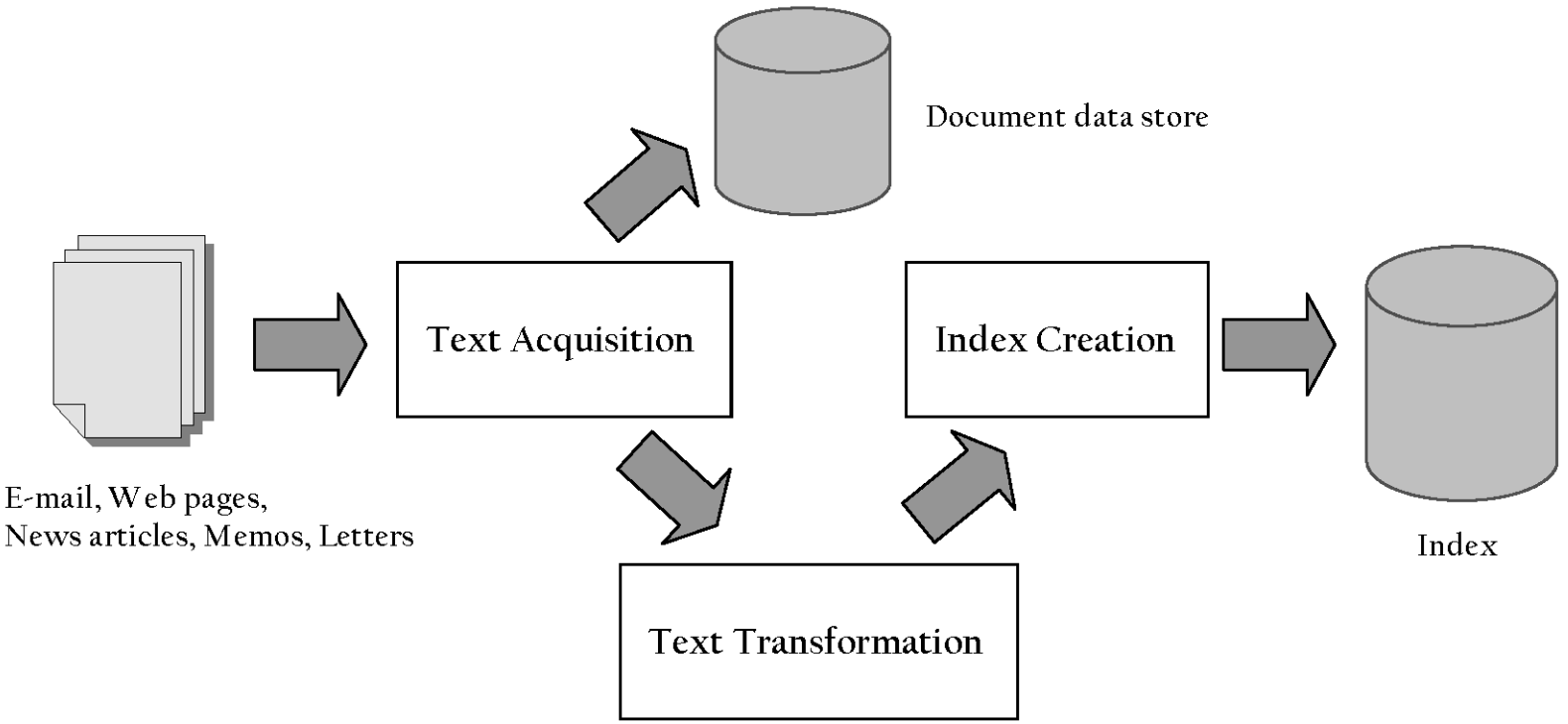
Inverted Indexing for Text Documents

UCSB 293S, Tao Yang 2020

Some of slides from the text books of Croft/Metzler/Strohman and Manning/Raghavan/Schutze

Index Process and Table of Content



- **Inverted index with positional information**
- **Compression**
- **Advanced index for fast query processing**

Indexes

- ***Indexes* are data structures to make search faster**
- **Most common data structure is *inverted index***
  - "inverted" because documents are associated with words, rather than words with documents
- **Inverted index: Each index term is associated with an *inverted list***
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting or postings*
  - Lists are usually *document-ordered* (sorted by document number)

Simple inverted index for example “Collection”

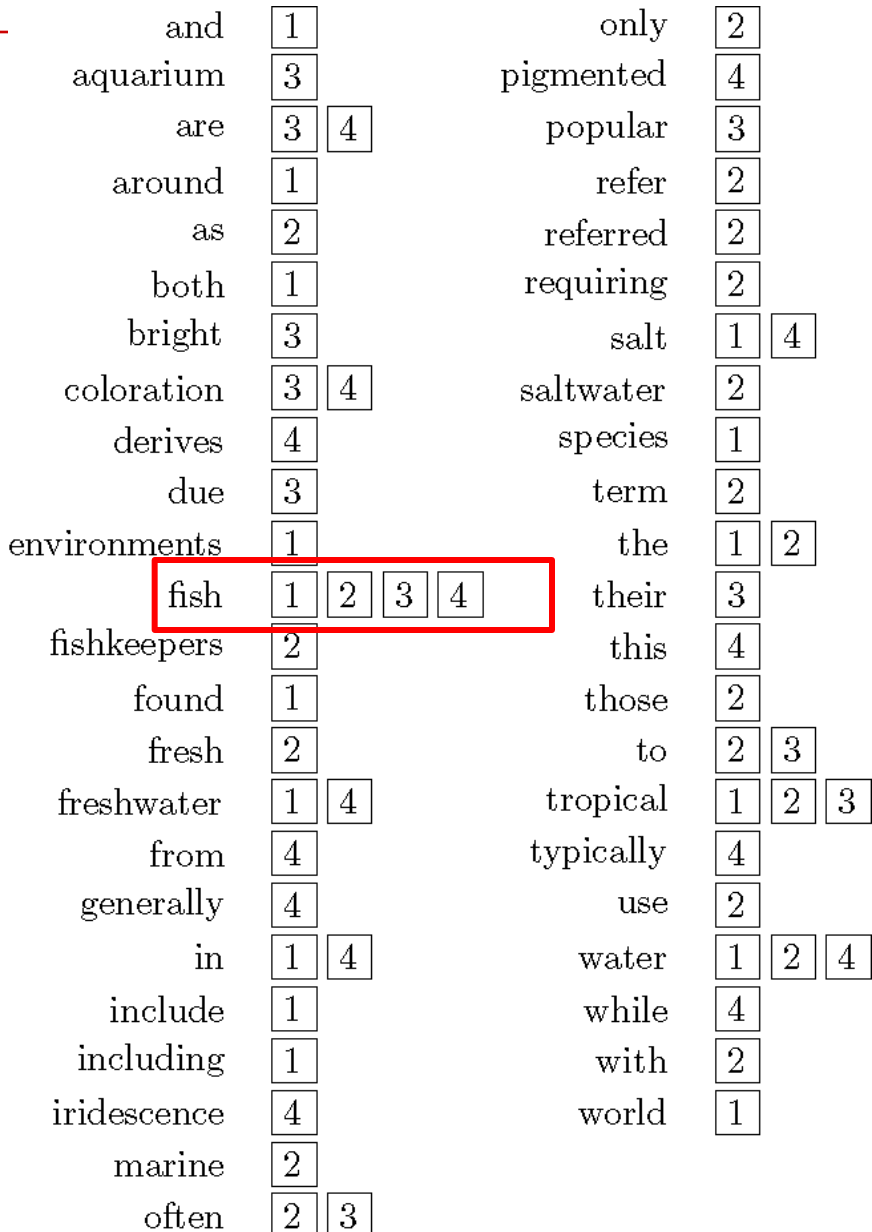
4 documents:

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.



What other information can be added in index to help ranking?

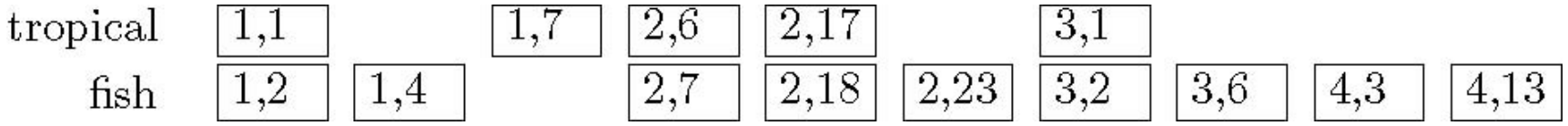
Inverted Index with word counts

- Supports more ranking features

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

Word Positions for Proximity Matches

- **Matching phrases or words within a window explicitly or implicitly.**
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- **Word positions in inverted lists make these types of query features efficient**
  - e.g.,



Fish appears at Positions 2 and 4 of Document 1

Positional indexes

- **Store, for each *term*, entries of the form:**

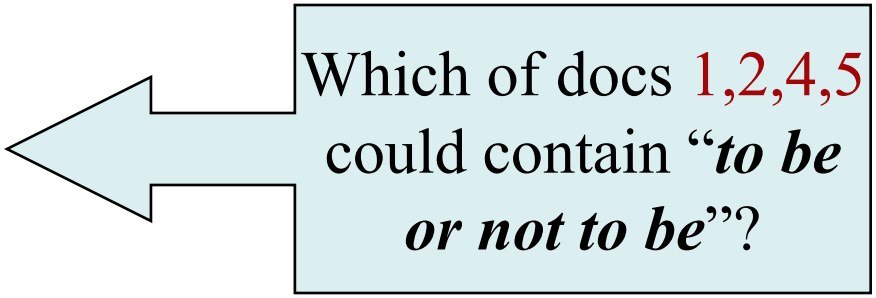
  <number of docs containing ***term***;

  doc1: position1, position2 … ;

  doc2: position1, position2 … ;

  etc.>

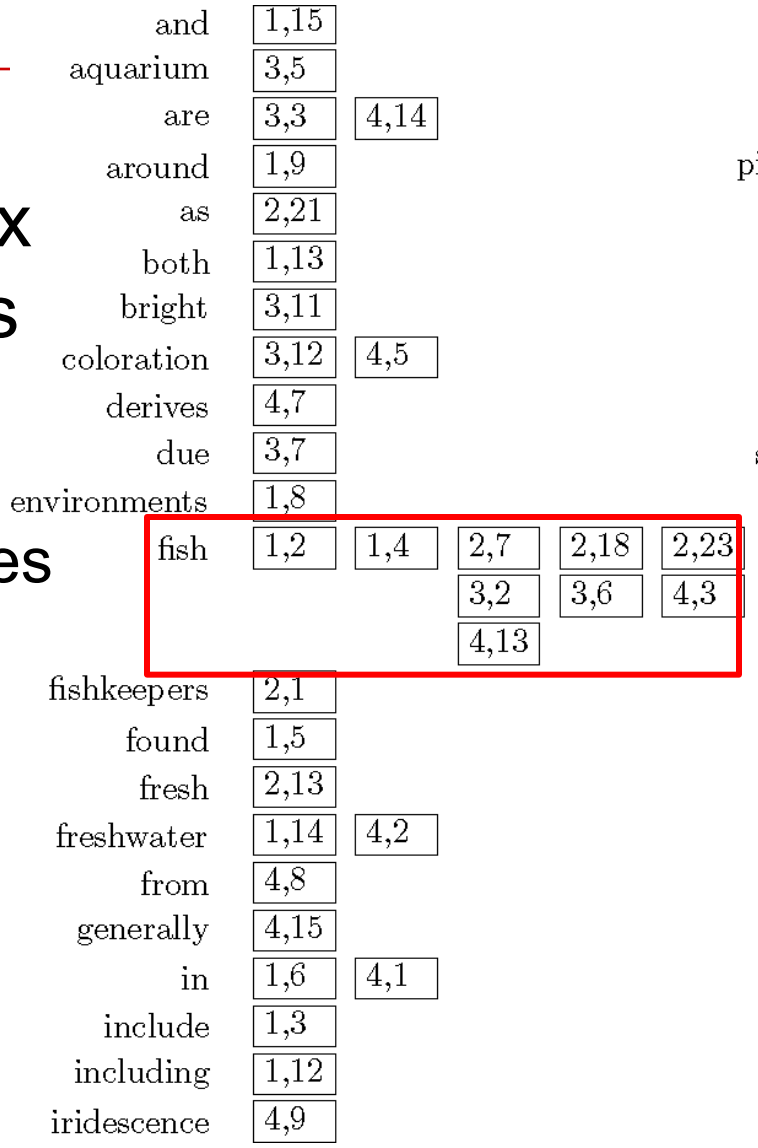
<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>



Expensive storage space for a large collection with long documents

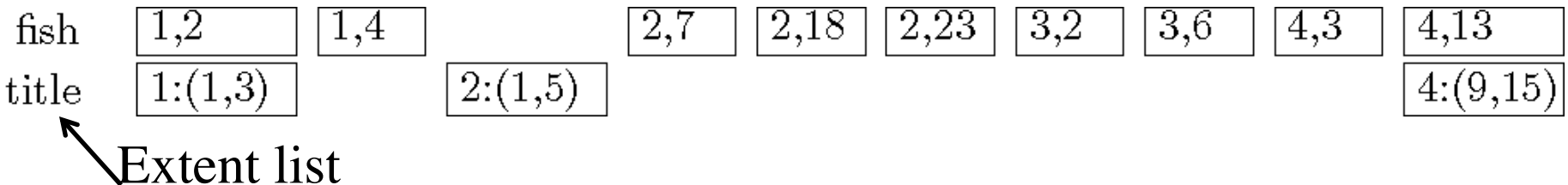
Inverted Index with positions

- supports proximity matches



Inverted Index with Fields

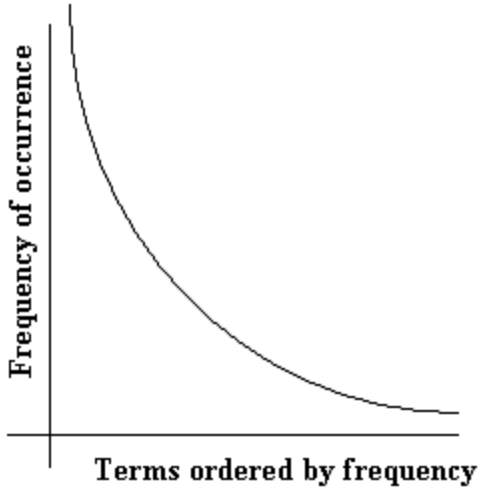
- **Document structure is useful in search**
  - *Text may be divided into multiple fields*
  - *field* restrictions, e.g., date. Field importance, e.g., title
- **Options:**
  - separate inverted lists for each field type
  - add information about fields to postings
    - use *extent lists to mark special areas in a document*
- **An *extent* is a contiguous region of a document**
  - represent extents using word positions
  - e.g. 1:(1,3) → title in document 1 is from 1 to 3



Other Issues

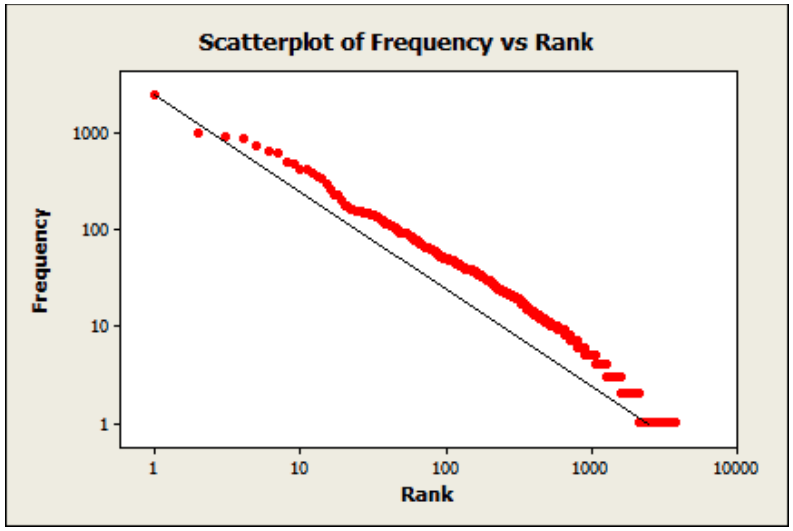
- **Precomputed scores in inverted list**
  - e.g., list for “fish” [(1:3.6), (3:2.2)], where 3.6 is total feature value for document 1
  - improves speed but reduces flexibility
- **Score-ordered lists**
  - query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries
- **How to estimate the storage need for inverted index?**
  - Zipf distribution of word posting lengths

Zipf's law on term distribution

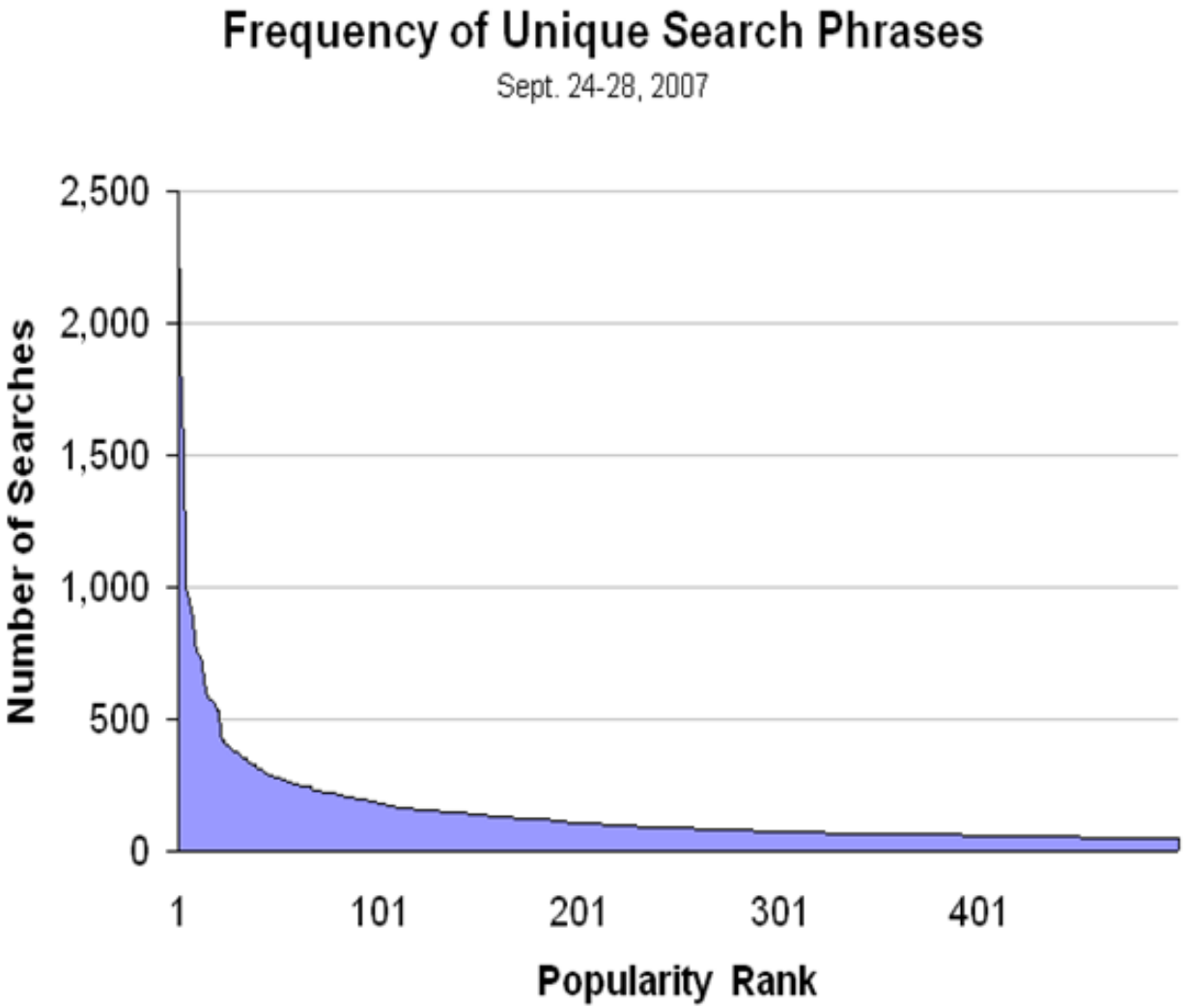


- **Study the relative frequencies of terms.**
  - there are a few very frequent terms and very many rare terms.
- **Zipf's law: The *i*-th most frequent term has frequency proportional to 1/*i* .**
- **cf_i is collection frequency: the number of occurrences of the term t_i**
  - $cf_i \propto 1/i$
  - $cf_i = c/i$ where c is a normalizing constant

$\log(cf_i) + \log(i) = \log(c)$



Zipf distribution for search query traffic



Analyze index size with Zipf distribution

- Number of docs = n = 40M. Number of terms = m = 1M
- The inverted index only stores the document IDs that contain a term and its frequency.
  - No positional information
  - How to estimate the size of inverted index?
  - Assume each postings record:
    - 16-byte (4+8+4) records *(term, doc, freq).*
  - Can you use Zipf to estimate number of postings entries?

Use Zipf to estimate number of postings entries:

Most popular term appears in all n documents.

Second most popular term appears in n/2 documents.

$n + n/2 + n/3 + \ldots. + n/m \approx n \ln m = 560\text{M}$ entries

560M*16B ≈ 9GB

Positional index size

- **Need an entry for each occurrence, not just once per document**
- **Index size depends on average document size**
  - Average web page has <1000 terms
  - SEC filings, PDF files, … easily 100,000 terms

- **Rules of thumb for English languages**
  - Positional index size factor of 2-4 over non-positional index
  - Positional index size 35-50% of volume of original text

Index Compression

- **Motivation: Inverted lists are very large**
  - Much higher if n-grams are indexed
- **Compression of indexes saves disk and/or memory space**
  - *Los less* compression – no information lost
  - Best compression techniques have good *compression ratios* and are easy to decompress
- ***Basic idea*: Common data elements use short codes while uncommon data elements use longer codes**
- **Example:** coding number sequence: 0, 1, 0, 2,0,3,0
  - Possible binary encoding: 00 01 00 10 00 11 00

Store 0 with a single 0: 0 01 0 10 0 11 0

How about this binary bit sequence: 0 1 0 10 0 11 0

Can you convert back to decimal numbers: 0, 1, 0, 2, 0, 3, 0?

Compression with unambiguous encoding

- ***Ambiguous*** **encoding – not clear how to decode when scanning a sequence of bits**
  - 0 1 0 10 0 11 0
  - Can mean 0, 1, 0, 2, 0, 3, 0
  - Or another decoding: 0, 2, 2, 0, 3, 0

- **unambiguous code:**

Number	Code
0	0
1	101
2	110
3	111

  - "0 1 0 1 0" uniquely gives 0, 1, 0

Another takeaway: Small numbers → use a small number of bits

Delta Encoding: encoding differences between consecutive numbers

- **Encode differences between consecutive numbers**
  - Word count data is good candidate for compression with many small numbers and few larger numbers
  - For a sequence of document IDs, delta encoding may also be effective with an ordered list.

- Example: $1, 5, 9, 18, 23, 24, 30, 44, 45, 48$
  - Delta encoding: $1, 4, 4, 9, 5, 1, 6, 14, 1, 3$

- Differences for a high-frequency word are easier to compress, e.g.,

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, ...$$

- Differences for a low-frequency word may be large, and compression is not easy $109, 3766, 453, 1867, 992, ...$

Compression with Bit-Aligned Codes

- Treat compressed data as a sequence of bits and breaks between encoded numbers can occur after any bit position
  - Pro: optimization to a bit level
  - Cons: more time cost
- ***Unary* code**
  - Encode *k* by *k* 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

- **Unary is efficient for small numbers such as 0 and 1, but quickly becomes expensive**
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- **Binary representation is more efficient for large numbers, but it may be ambiguous**

Elias-γ Code: Combine binary/unary representations

- **To encode a number *k*, decompose k into two parts. Compute**
  - k_d is number of binary digits, encoded in unary
  - k_r is the remainder, encoded in binary

- $k_d = \lfloor \log_2 k \rfloor$
- $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Cost Analysis and Elias-δ Code

- **Elias-γ code uses no more bits than unary, many fewer for k > 2**
  - 1023 takes 19 bits instead of 1024 bits using unary
  - In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits
- **To improve coding of large numbers, use Elias-δ code**
  - Apply Elias-γ recursively to the first component
  - Instead of encoding k_d in unary, we encode $k_d + 1$ using Elias-γ
  - Takes approximately $2 \log_2 \log_2 k + \log_2 k$ bits

Example of Elias-δ Code

- **Split the first component k_d into:**
  - $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
  - $k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$
  - encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

Byte-Aligned Codes

- Variable-length bit encodings can be too complex on processors that are more effective in handling bytes
- *v-byte* is a popular byte-aligned code
  - Similar to Unicode UTF-8
  - Shortest v-byte code is 1 byte
  - Numbers are 1 to 4 bytes, with high bit 1 in the last byte, 0 otherwise

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

V-Byte Encoding

k	Binary Code	Hexadecimal
1	1 0000001	81
6	1 0000110	86
127	1 1111111	FF
128	0 0000001 1 0000000	01 80
130	0 0000001 1 0000010	01 82
20000	0 0000001 0 0011100 1 0100000	01 1C A0

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

V-Byte Encoder and Decoder in C++

```
public void encode( int[] input, ByteBuffer output ) {
    for( int i : input ) {
        while( i >= 128 ) {
            output.put( i & 0x7F );
            i >>>= 7;
        }
        output.put( i | 0x80 );
    }
}
```

```
public void decode( byte[] input, IntBuffer output ) {
    for( int i=0; i < input.length; i++ ) {
        int position = 0;
        int result = ((int)input[i] & 0x7F);

        while( (input[i] & 0x80) == 0 ) {
            i += 1;
            position += 1;
            int unsignedByte = ((int)input[i] & 0x7F);
            result |= (unsignedByte << (7*position));
        }

        output.put(result);
    }
}
```

Compression Example with v-bye after delta-encoding

- **Given invert list with positions:**
  - (Doc ID, #occurrence, positions)

$$(1, 2, [1, 7])(2, 3, [6, 17, 197])(3, 1, [1])$$

- **Delta encoding of document numbers and positions:**

$$(1, 2, [1, 6])(1, 3, [6, 11, 180])(1, 1, [1])$$

- **Compress using v-byte:**

```
81 82 81 86 81 82 86 8B 01 B4 81 81 81
```

Word-Aligned Simple-9 Code (Anh/Moffat 2004)

Can we store more numbers in a byte?

Try to pack several numbers into one word (32 bits)

- each word has 4 control bits and 28 data bits
- Assume each number requires at most 28 bits.

9 cases of data represented by 28 bits:

- **-** 1 28-bit number - 2 14-bit numbers
- - 3 9-bit numbers (1 bit wasted) - 4 7-bit numbers
- - 5 5-bit numbers (3 bits wasted) - 7 4-bit numbers
- - 9 3-bit numbers (1 bit wasted) - 14 2-bit numbers
- - 28 1-bit numbers

4 Control bits indicate which of these 9 cases is used

Word-Aligned Simple-9 Code (Anh/Moffat 2004)

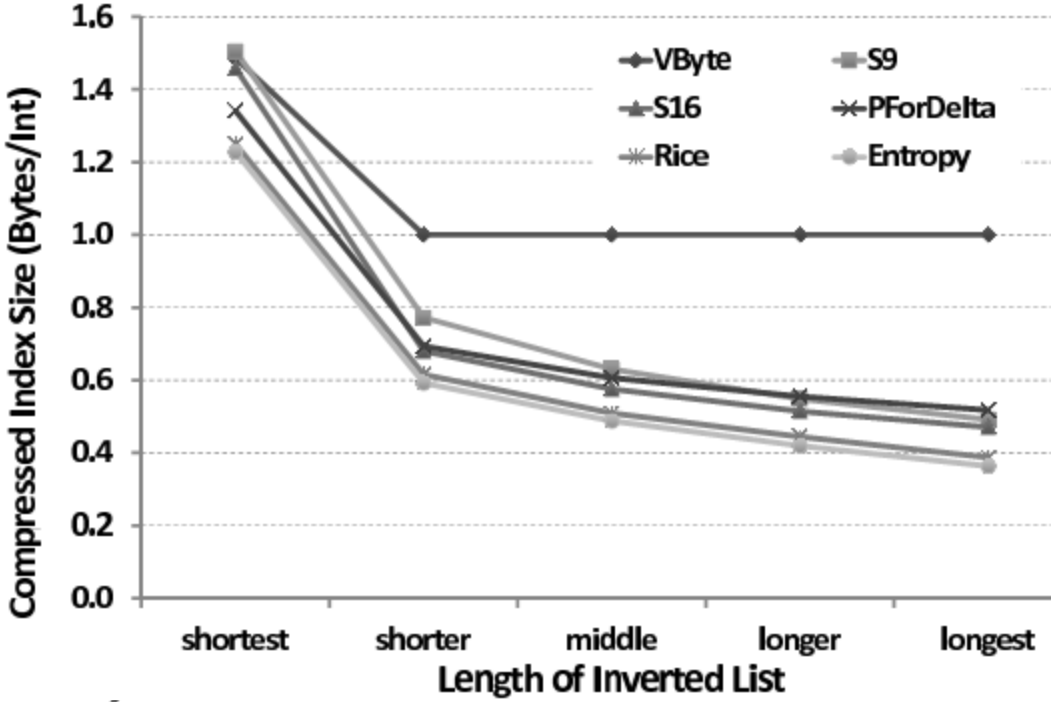
Algorithm:

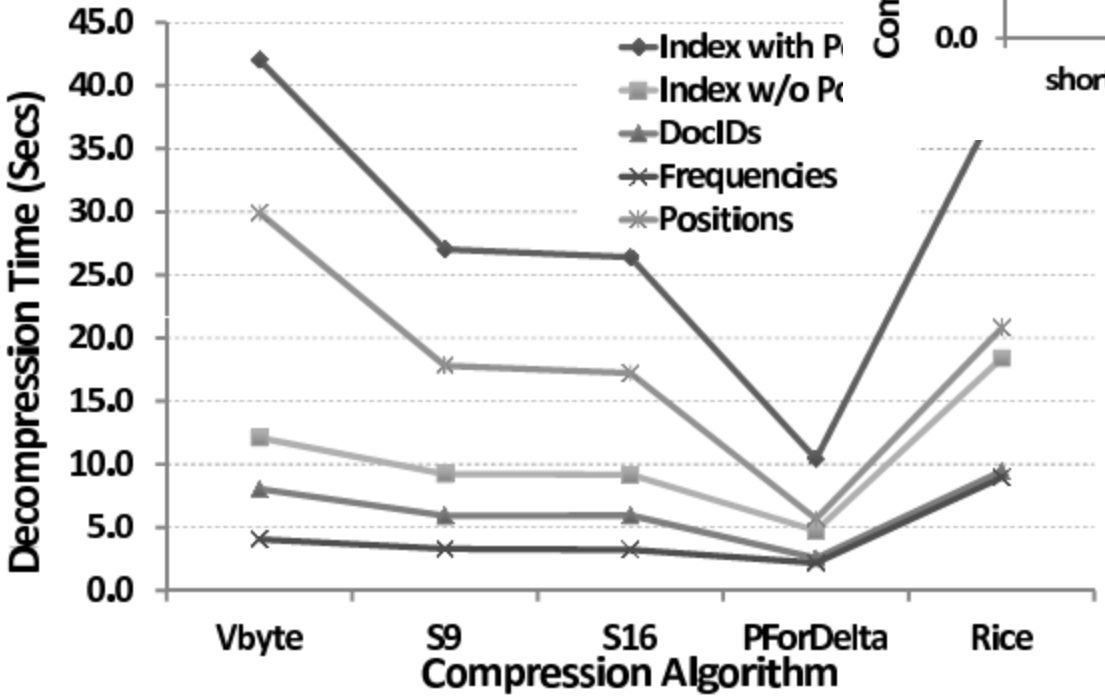
- do the next 28 numbers fit into one bit each?
- if no: do the next 14 numbers fit into 2 bits each?
- if no: do the next 9 numbers fit into 3 bits each?
- …

Fast decoding: only one if-decision for every 32 bits

Decent compression ratio: can use < 1 byte for small numbers

Compression Performance [WWW08, Zhang/Long/Suel]





Advanced Indexing for Fast Query Processing

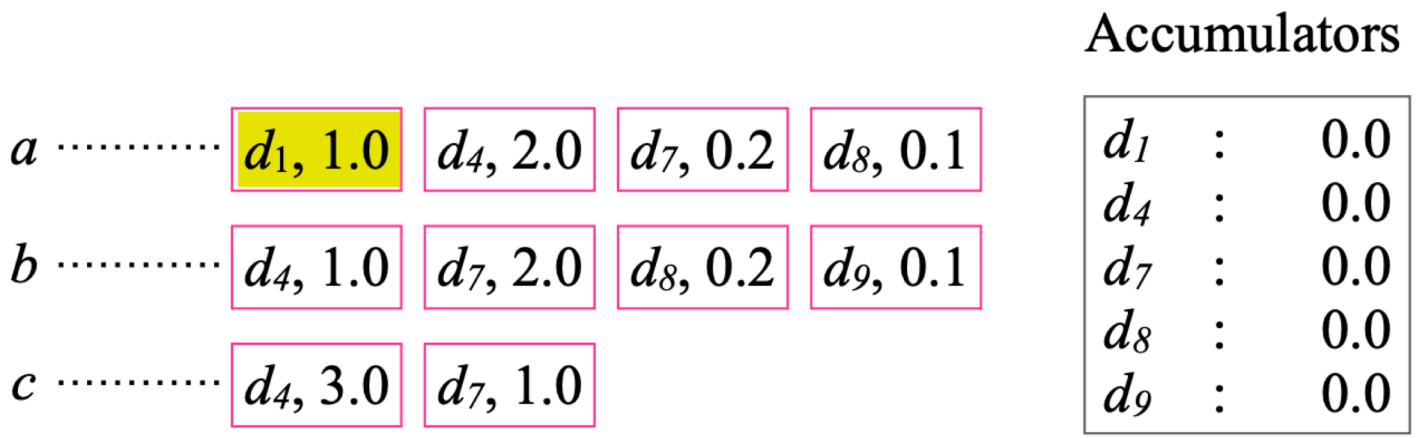
- Index traversal during online query processing
- Skip pointers for conjunctive queries
- Earlier termination for top K disjunctive query processing

Advanced Indexing for Fast Query Processing

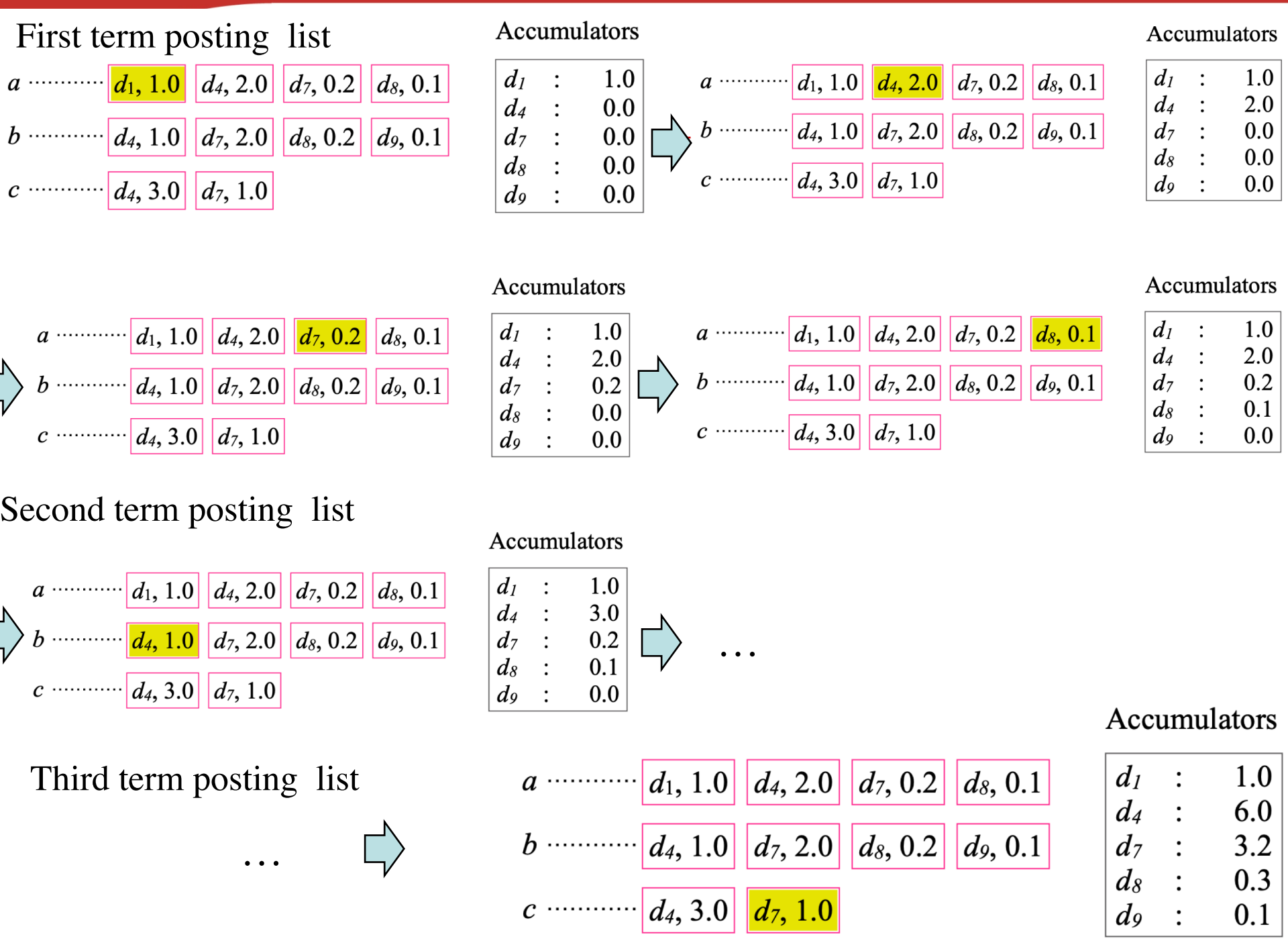
- Search engines commonly separate the ranking process into two or more phases.
  - In the first phase, a very simple and fast ranking function such as BM25 is used to get, say, the top 1000 documents.
    - Query type
      - Conjunctive (all query terms are required)
      - Disjunctive (some of terms are required)
      - Phrase or proximity
    - Significant amount of computation is still spent in the first phase. Index design is critical.
  - Then in the second and further phases, increasingly more complicated ranking functions with more and more features are applied to documents that pass through the earlier phases.

First Phase Fast Ranking

- **Simple rank formula** score (d) = ∑ TermScore(t, d) for all terms t in the query. E.g. TFIDF, BM25
- Data traversal during online query processing
  - Term-at-a-Time (TAAT) query processing
    - reads posting lists for query terms successively
    - maintains an accumulator for each result document with value

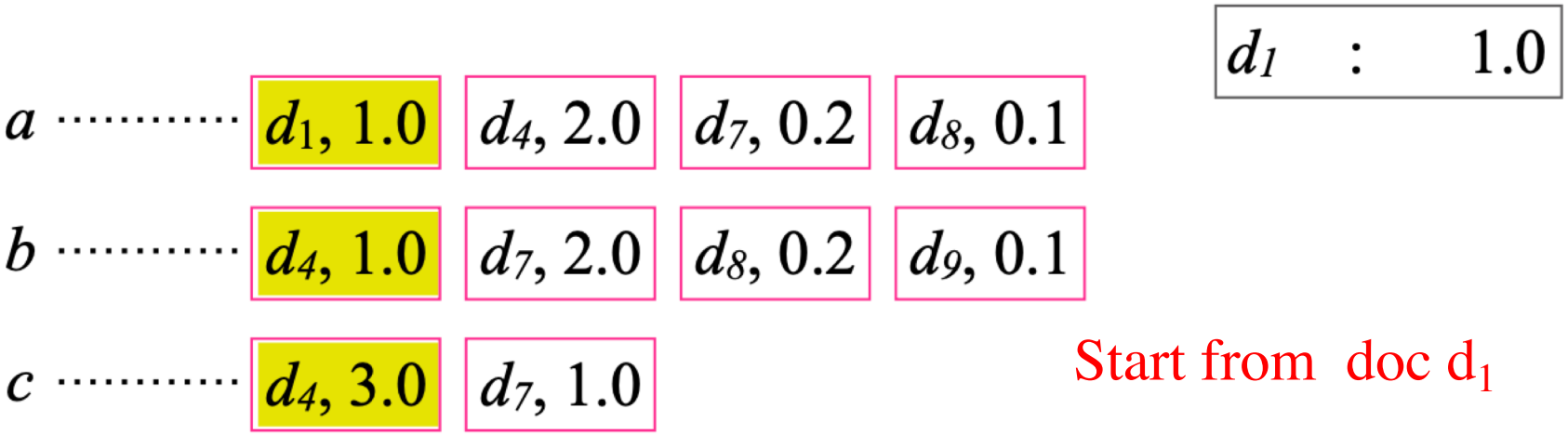


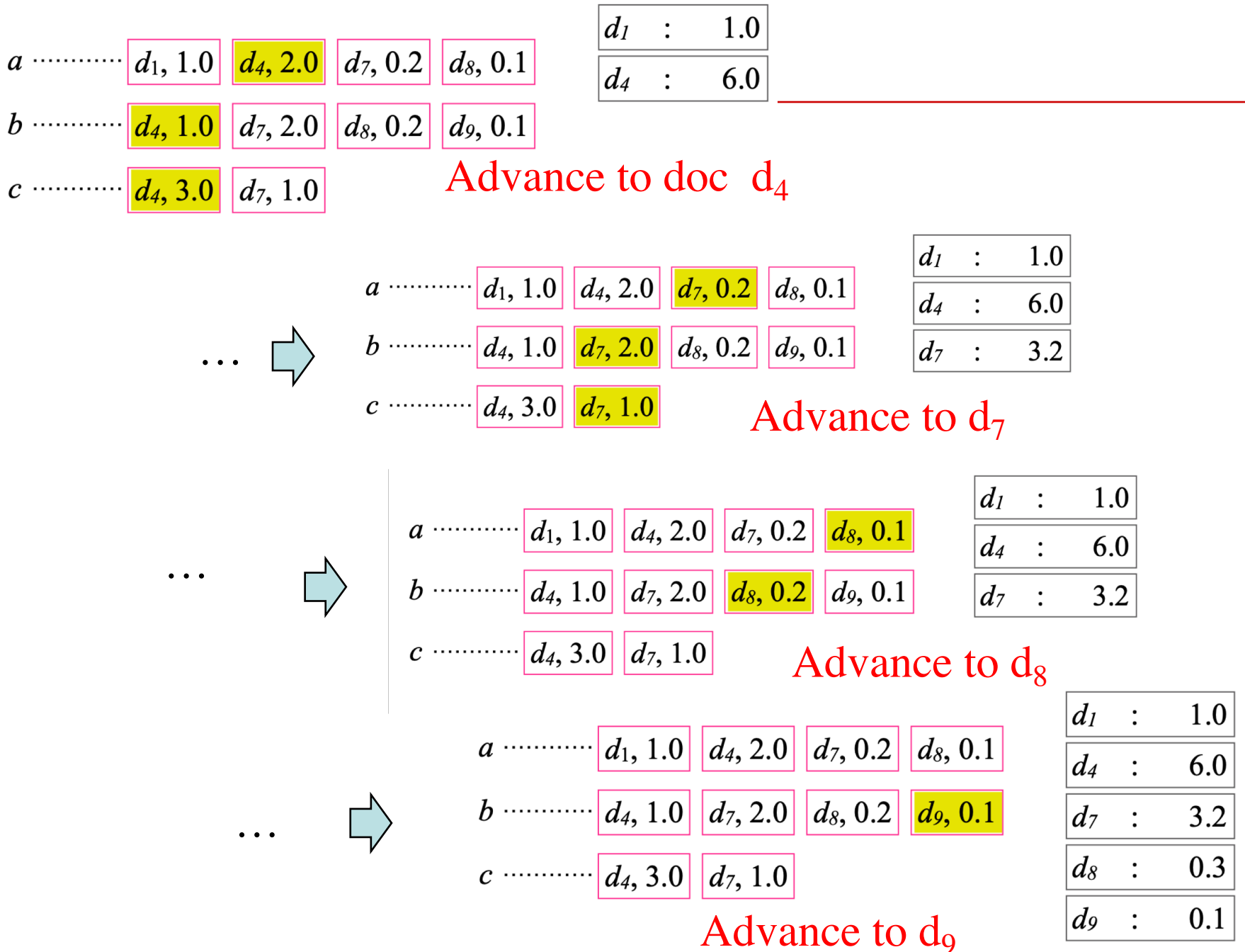
  - Document-at-a-time (DAAT) approach


First term posting list
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 0.0
d7 : 0.0
d8 : 0.0
d9 : 0.0
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 2.0
d7 : 0.0
d8 : 0.0
d9 : 0.0
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 2.0
d7 : 0.2
d8 : 0.0
d9 : 0.0
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 2.0
d7 : 0.2
d8 : 0.1
d9 : 0.0
Second term posting list
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 3.0
d7 : 0.2
d8 : 0.1
d9 : 0.0
...
Third term posting list
...
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
Accumulators
d1 : 1.0
d4 : 6.0
d7 : 3.2
d8 : 0.3
d9 : 0.1

Document-at-a-time (DAAT)

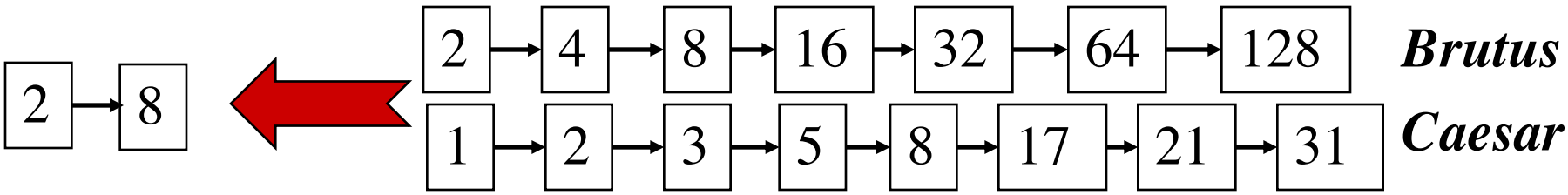
- Assumes document-ordered posting lists
- Reads posting lists for query terms concurrently
- Computes score when same document is seen in one or more posting lists
- Always advances posting list with **lowest current document identifier**




a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
d1 : 1.0
d4 : 6.0
Advance to doc d4
...
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
d1 : 1.0
d4 : 6.0
d7 : 3.2
Advance to d7
...
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
d1 : 1.0
d4 : 6.0
d7 : 3.2
Advance to d8
...
a ····· d1, 1.0 d4, 2.0 d7, 0.2 d8, 0.1
b ····· d4, 1.0 d7, 2.0 d8, 0.2 d9, 0.1
c ····· d4, 3.0 d7, 1.0
d1 : 1.0
d4 : 6.0
d7 : 3.2
d8 : 0.3
d9 : 0.1
Advance to d9

Intersection of posting lists for conjunctive queries

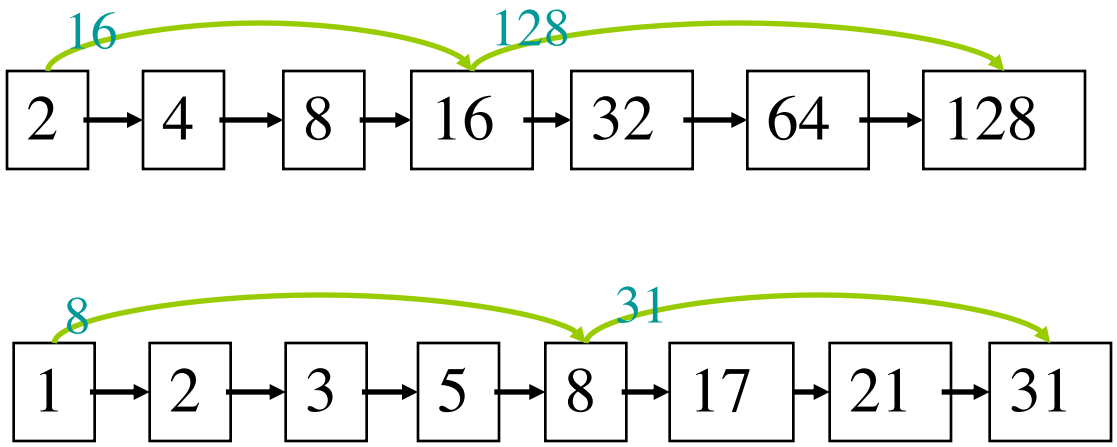
- All query terms are required for a matching document
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are m and n, the merge takes $O(m+n)$ operations.

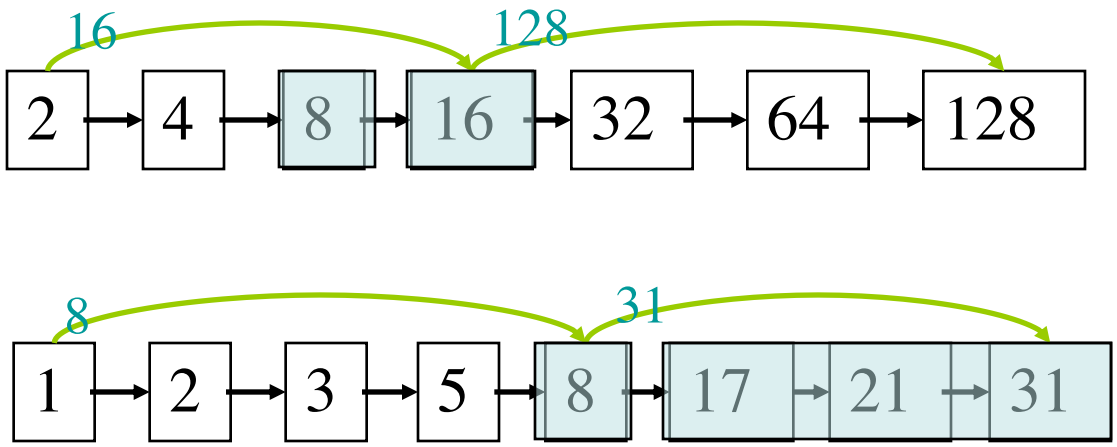
Can we do better?
Yes, if index isn't changing too fast.

Augment postings with skip pointers (at indexing time)



- **Why?**
- **To skip postings that will not be part of the search results.**

Query processing with skip pointers



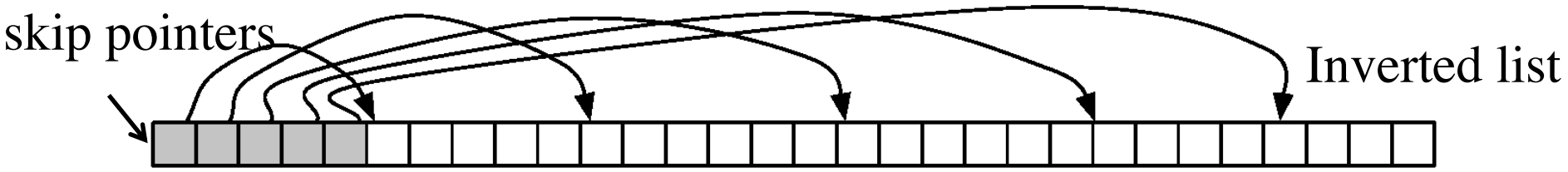
Suppose we've stepped through the lists until we process **8** on each list.

When we get to **16** on the top list, we see that its successor is **32**.

But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings.

Skip Pointers

- **A skip pointer (*d, p*) contains a document number *d* and a byte (or bit) position *p***
  - Means there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*



- **Example for inverted list**

$$5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119$$

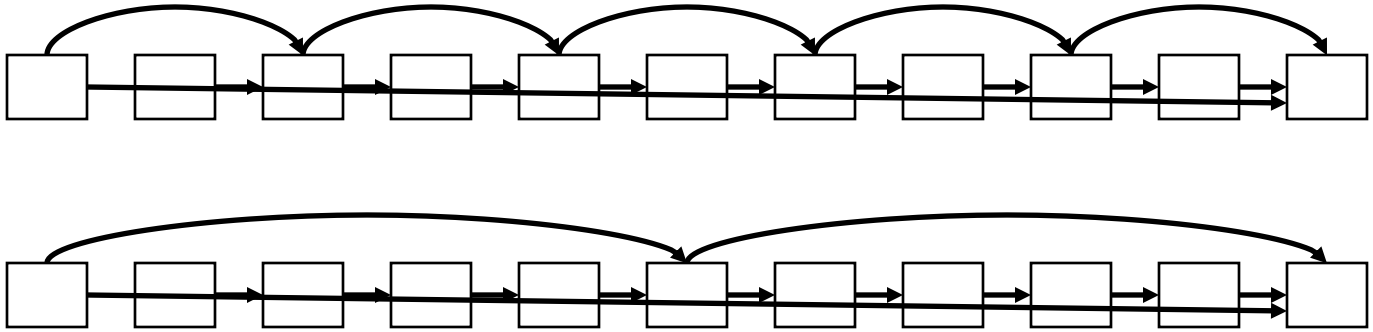
  - D-gaps
  - Skip pointers

$$5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15$$

$$(17, 3), (34, 6), (45, 9), (52, 12), (89, 15), (101, 18)$$

How many skip pointers?

- **Tradeoff:**
  - More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers.
  - Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.



- Simple heuristic: for postings of length *L*, use sqrt(L) evenly-spaced skip pointers.
  - Easy if the index is relatively static; harder if *L* keeps changing because of updates.

Earlier Termination for Fast Query Processing

- **Exhaustive Search vs Earlier Termination**
  - Search algorithm is exhaustive if it fully evaluates all documents that satisfy the Boolean condition
  - Otherwise it is called earlier termination
- **Earlier termination strategies**
  - Stopping early, where each inverted list is arranged from most to least promising posting and traversal is stopped once enough good results are found,
  - Skipping, where inverted lists are sorted by document IDs, and thus promising documents spread out over the lists, but we can skip over uninteresting parts of a list
  - Partial scoring, where candidate documents are only partially or approximately scored.

Types of Index Organization

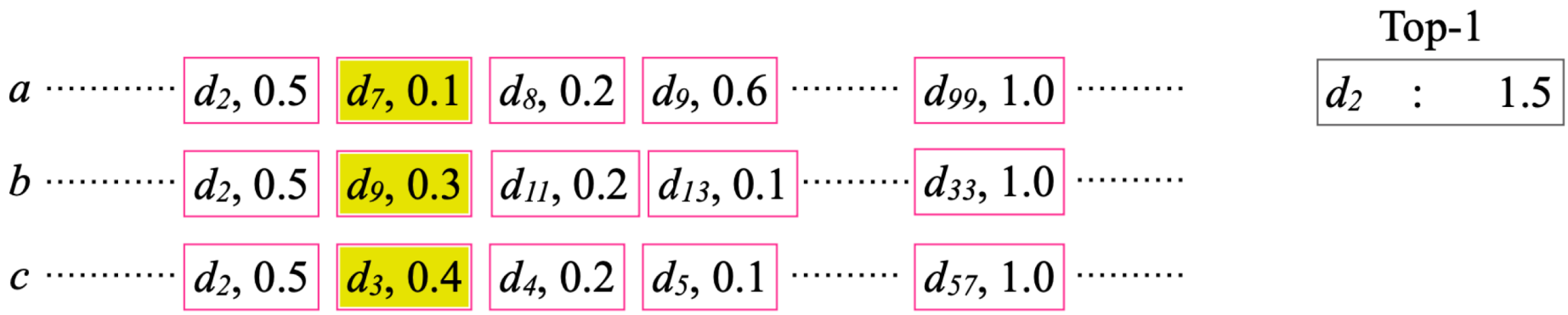
- **Document-Sorted Indexes**: the postings in each inverted list are sorted by document ID.
  - Popular. Good for DAAT traversal, skip pointer optimization, delta encoding based compression.
  - WAND/BMW top-K algorithms for disjunctive queries
- **Impact-Sorted Indexes**: Postings in each list are sorted by their impact, that is, their contribution to the score of a document.
  - Good for stopping early strategy where each inverted list is arranged from most to least promising posting and traversal is stopped once enough good results are found.
  - TAAT traversal is often used
  - Not easy for delta-encoding based compression
- **Impact-Layered Indexes**: partition the postings in each list into a number of layers, such that all postings in layer i have a higher impact than those in layer i + 1, and then sort the postings in each layer by document IDs.

Safe Earlier Termination with WAND for Disjunctive Queries

- **Safe earlier termination allows faster processing while having the same result as exhaustive top-K search.**
  - Only need top K documents with the highest scores
- **Weak AND (WAND) query processing**
  - Simple rank formula score (d) = $\sum$ TermScore(t, d) for all terms t in the query
  - Assume document-ordered posting lists. Each *i-th* posting list maintains *maxscore*(*i*)
  - Follow DAAT and read related posting lists concurrently
    - Compute score when same document is seen in one or more posting lists
  - Skip some documents which are impossible to be in top K
    - Always advances posting list with lowest current document identifier up to pivot document identifier computed from current top-*K* result

How to Skip Low-score Documents in WAND

- Maintain a current document ID pointer at each posting list of term t: cdid(t)
- MaxScore(t) is the maximum term score in term t's posting list.
- Dynamically maintain minScore as minimum score to be in top K



MaxScore(a) =1
MaxScore(b)=1
MaxScore(c)=1

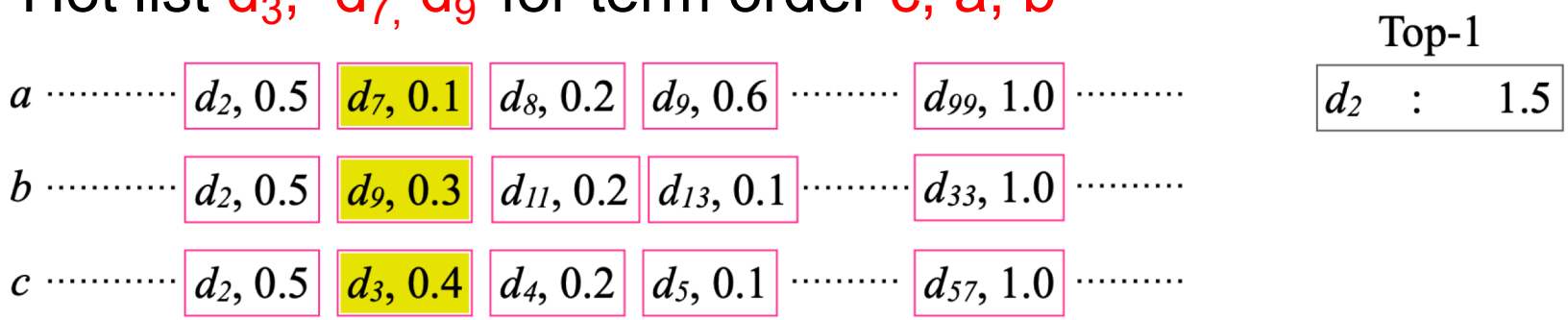
cdid(a) = d_7
cdid(b)=d_9
cdid(c)= d_3

minScore=1.5
Only top 1 result is needed

For current cdid(a), cdid(b), cdid(c) list: d_3, d_7, d_9 is it possible that d_3 appears in the posting list of term "a" or "b"? No. Why?

How to Skip Low-score Documents in WAND

- Sort posting lists in ascending order of *cdid()'s* document IDs and focus on these hot documents
  - Hot list d_3, d_7, d_9 for term order c, a, b



Define the pivot be j^{th} document in the above sorted hot list:

- Any document between 1^{th} and $(j-1)^{th}$ positions of the hot list cannot qualify for top K results due to low score
- j^{th} document satisfies: $\sum_{1 \le i \le j} \text{MaxScore}(t_i) > \text{minScore}$
  - j=1. MaxScore(c)=1 < 1.5 d_3 is not possible to score higher than 1.5
  - j=2. MaxScore(c)+ MaxScore(a) =2 > 1.5
  - d_7 is possible to score higher than 1.5. Thus it is the pivot for current hot list. Advance the current lowest doc pointer to the pivot

WAND Reference and Block-MAX WAND

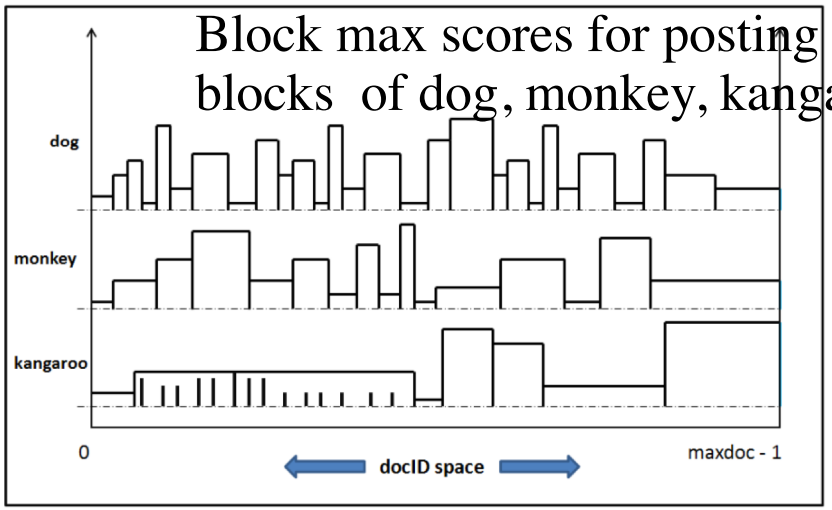
- **WAND:** A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. ACM CIKM, 2003.
- **Block-MAX WAND (BMW)**
  - S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. SIGIR 2011.
  - Motivation for improvement
    - Max impact score in a term posting list can be much larger than the average individual doc score
    - Splits the inverted lists into blocks of, say, 64 or 128 docIDs such that each block can be decompressed separately
  - Create an extra table, which stores for each block
    - The max/min docID, Maximum score for each block
    - Still need to compute the maximum score per term posting list
  - Leverage more accurate per-block max score while skipping blocks of documents quickly

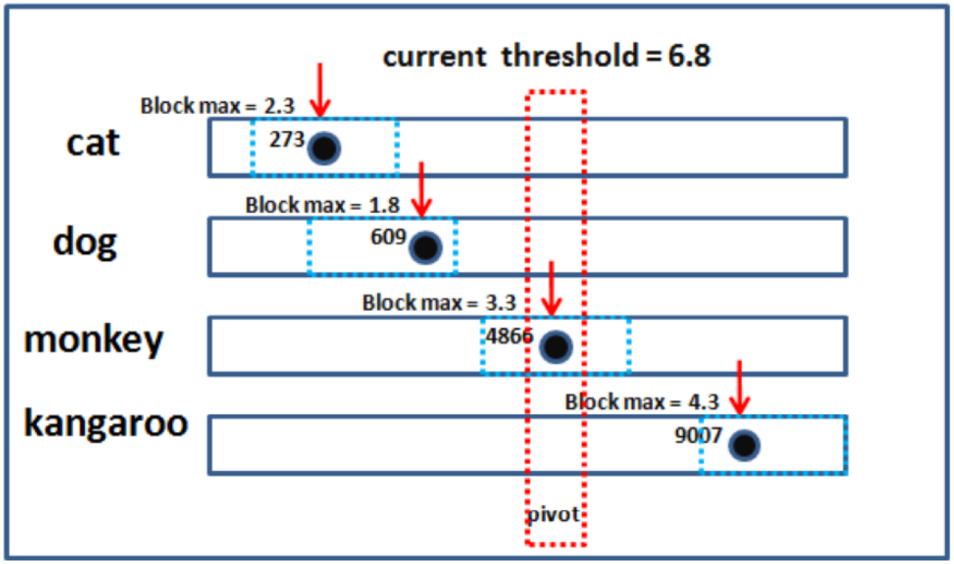
Illustration of Key Ideas in BMW

- Maintain piece-wise upper-bound approximation of the impact scores in the lists.

- Naive use of block max score is incorrect

$$\sum_{1 \le i \le j} \text{BlockMax}(t_i) > \text{minScore}$$





Still use WAND idea to find a candidate pivot.
Once a candidate pivot is found, dynamically locate the block in each term posting list that may own this pivot document

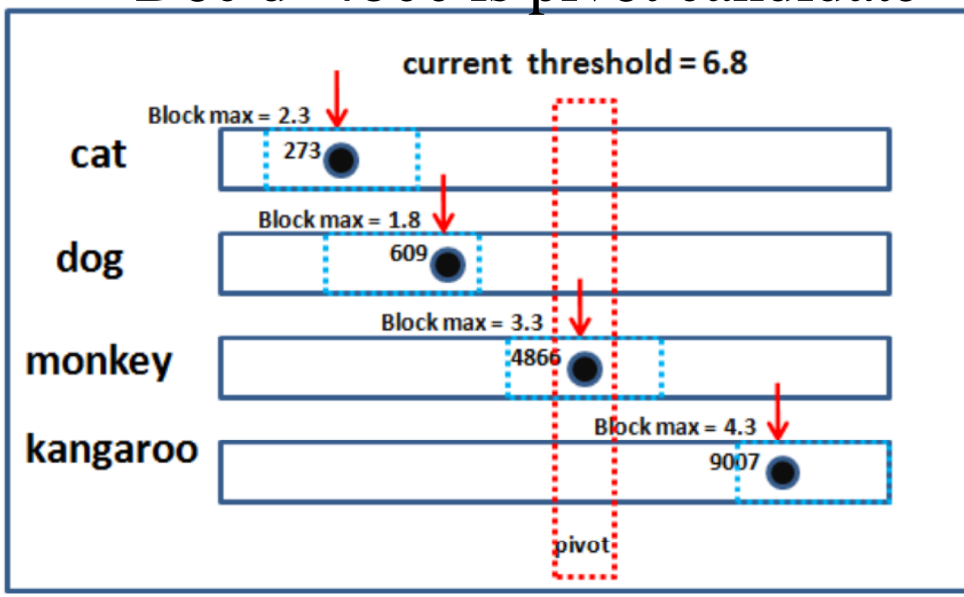
Which blocks contain doc #4866 in cat&dog?

How to skip low-score documents in BMW

- Still use WAND idea to find a pivot candidate

  $\sum_{1 \le i \le j} \text{MaxScore}(t_i) > \text{minScore}$

- Once a candidate pivot d is found, skip any document before d in the focused hot list.



- Dynamically locate the block B_i in each term posting list that may own doc d
  - Use the next block max/min IDs to filter unnecessary blocks that cannot contain d in each term posting list.
- Double check if d is a real candidate by using

  $\sum_{1 \le i \le j} \text{BlockMax}(t_i, B_i) > \text{minScore}$

- If d does not satisfy, find the next minimum doc ID to move forward

A Comparison of Exhaustive Search, WAND, and BMW with DAAT/TAAT

- TREC GOV2 web page collection. TREC 2006 query log
- Query processing time in milliseconds on average and for different query lengths: 2, 3, 4, 5, >5.
- Exhaustive OR, WAND, SC, and BMW are for disjunctive queries, while Exhaustive AND is for conjunctive queries.
- SC is a TAAT-based algorithm (Strohman&Croft. Efficient document retrieval in main memory, SIGIR 2007). Other algorithms follow DAAT.

TREC 2006						
	avg	2	3	4	5	> 5
exhaustive OR	225.7	60	159.2	261.4	376	646.4
WAND	77.6	23.0	42.5	89.9	141.2	251.6
SC	64.3	12.2	36.7	75.6	117.2	226.3
BMW	27.9	4.07	11.52	33.6	54.5	114.2
exhaustive AND	11.4	10.3	10.8	14.0	15.4	15.2

The authors also propose Block-max AND (BMA), similar to BMW, for conjunctive queries and outperforms exhaustive AND.

Summary

- **Inverted index with positional information**
  - Efficient data structure for fast online query processing
  - Zipf distribution for storage estimation
- **Compression**
  - Reduce storage need of a large index
  - Delta encoding
  - Bit aligned methods: Elias-γ, Elias-δ
  - Byte aligned methods: V-Byte, Simple-9
- **Advanced indexing for fast query processing**
  - Skip pointers
  - TAAT or DAAT order for online index traversal
  - WAND, Block MAX WAND for safe earlier termination in top K ranking