Search Engines

Information Retrieval in Practice

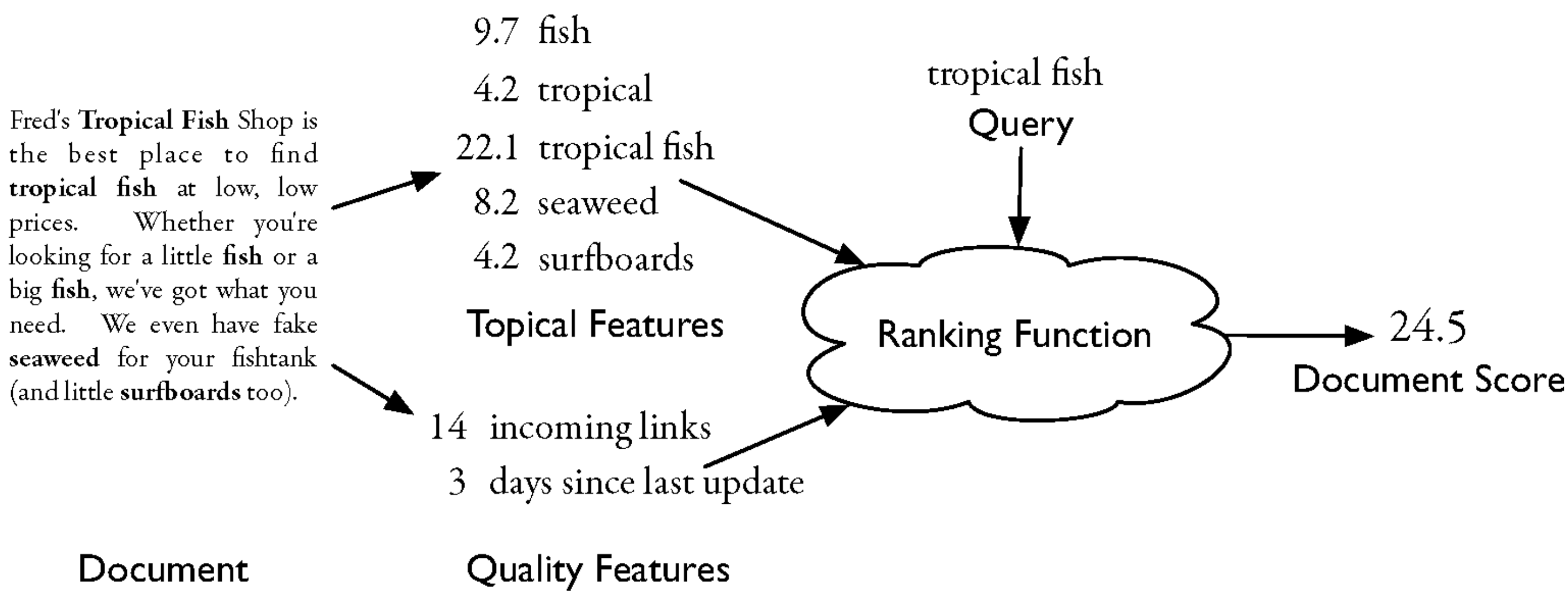
Indexes

- *Indexes* are data structures designed to make search faster
- Text search has unique requirements, which leads to unique data structures
- Most common data structure is *inverted index*
  - general name for a class of structures
  - "inverted" because documents are associated with words, rather than words with documents
    - similar to a *concordance*

Indexes and Ranking

- Indexes are designed to support *search*
  – faster response time, supports updates
- Text search engines use a particular form of search: *ranking*
  – documents are retrieved in sorted order according to a score computing using the document representation, the query, and a *ranking algorithm*
- What is a reasonable abstract model for ranking?
  – enables discussion of indexes without details of retrieval model

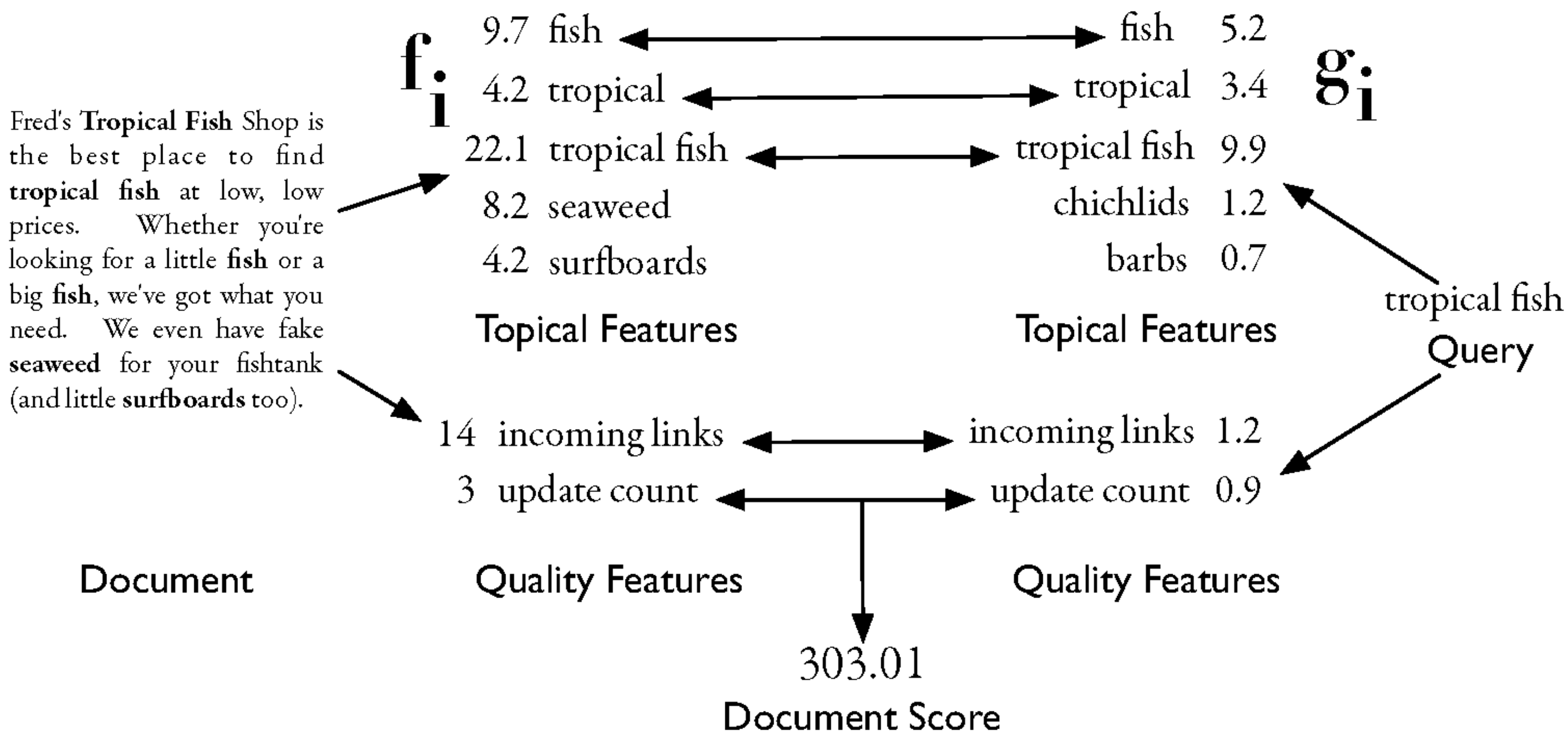
Abstract Model of Ranking



More Concrete Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

f_i is a document feature function
g_i is a query feature function



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*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

Example “Collection”

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.

Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index

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

Inverted Index with counts

- supports better ranking algorithms

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		

Inverted Index with positions

- supports proximity matches

and	1,15								
aquarium	3,5								
are	3,3	4,14							
around	1,9								
as	2,21								
both	1,13								
bright	3,11								
coloration	3,12	4,5							
derives	4,7								
due	3,7								
environments	1,8								
fish	1,2	1,4	2,7	2,18	2,23	3,2	3,6	4,3	4,13
fishkeepers	2,1								
found	1,5								
fresh	2,13								
freshwater	1,14	4,2							
from	4,8								
generally	4,15								
in	1,6	4,1							
include	1,3								
including	1,12								
iridescence	4,9								
marine	2,22								
often	2,2	3,10							
only	2,10								
pigmented	4,16								
popular	3,4								
refer	2,9								
referred	2,19								
requiring	2,12								
salt	1,16	4,11							
saltwater	2,16								
species	1,18								
term	2,5								
the	1,10	2,4							
their	3,9								
this	4,4								
those	2,11								
to	2,8	2,20	3,8						
tropical	1,1	1,7	2,6	2,17	3,1				
typically	4,6								
use	2,3								
water	1,17	2,14	4,12						
while	4,10								
with	2,15								
world	1,11								

Proximity Matches

- Matching phrases or words within a window
  - 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.,

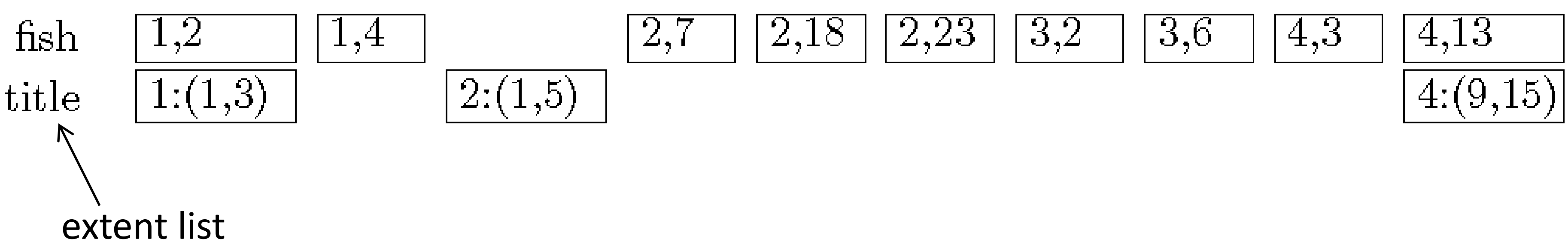
tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

Fields and Extents

- Document structure is useful in search
  - *field* restrictions
    - e.g., date, from:, etc.
  - some fields more important
    - e.g., title
- Options:
  - separate inverted lists for each field type
  - add information about fields to postings
  - use *extent lists*

Extent Lists

- An *extent* is a contiguous region of a document
  - represent extents using word positions
  - inverted list records all extents for a given field type
  - e.g.,



Other Issues

- Precompiled 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

Compression

- Inverted lists are very large
  - e.g., 25-50% of collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good *compression ratios* and are easy to decompress
- *Lossless* compression – no information lost

Compression

- *Basic idea*: Common data elements use short codes while uncommon data elements use longer codes
  - Example: coding numbers
    - number sequence:

$$0, 1, 0, 3, 0, 2, 0$$

    - possible encoding:

$$00\ 01\ 00\ 10\ 00\ 11\ 00$$

    - encode 0 using a single 0:

$$0\ 01\ 0\ 10\ 0\ 11\ 0$$

    - only 10 bits, but...

Compression Example

- *Ambiguous* encoding – not clear how to decode
  - another decoding:

$$0\ 01\ 01\ 0\ 0\ 11\ 0$$

  - which represents:

$$0, 1, 1, 0, 0, 3, 0$$

  - use unambiguous code:

Number	Code
0	0
1	101
2	110
3	111

  - which gives:

$$0\ 101\ 0\ 111\ 0\ 110\ 0$$

Delta Encoding

- Word count data is good candidate for compression
  - many small numbers and few larger numbers
  - encode small numbers with small codes
- Document numbers are less predictable
  - but differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*:
  - encoding differences between document numbers (*d-gaps*)

Delta Encoding

- Inverted list (without counts)

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

- Differences between adjacent numbers

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

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

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$$

- Differences for a low-frequency word are large, e.g.,

$$109, 3766, 453, 1867, 992, \ldots$$

Bit-Aligned Codes

- Breaks between encoded numbers can occur after any bit position
- *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 and Binary Codes

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

Byte-Aligned Codes

- Variable-length bit encodings can be a problem on processors that process 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

V-Byte Encoding

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

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

V-Byte Encoder

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

V-Byte Decoder

```
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

- Consider invert list with positions:

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

- Delta encode 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
```