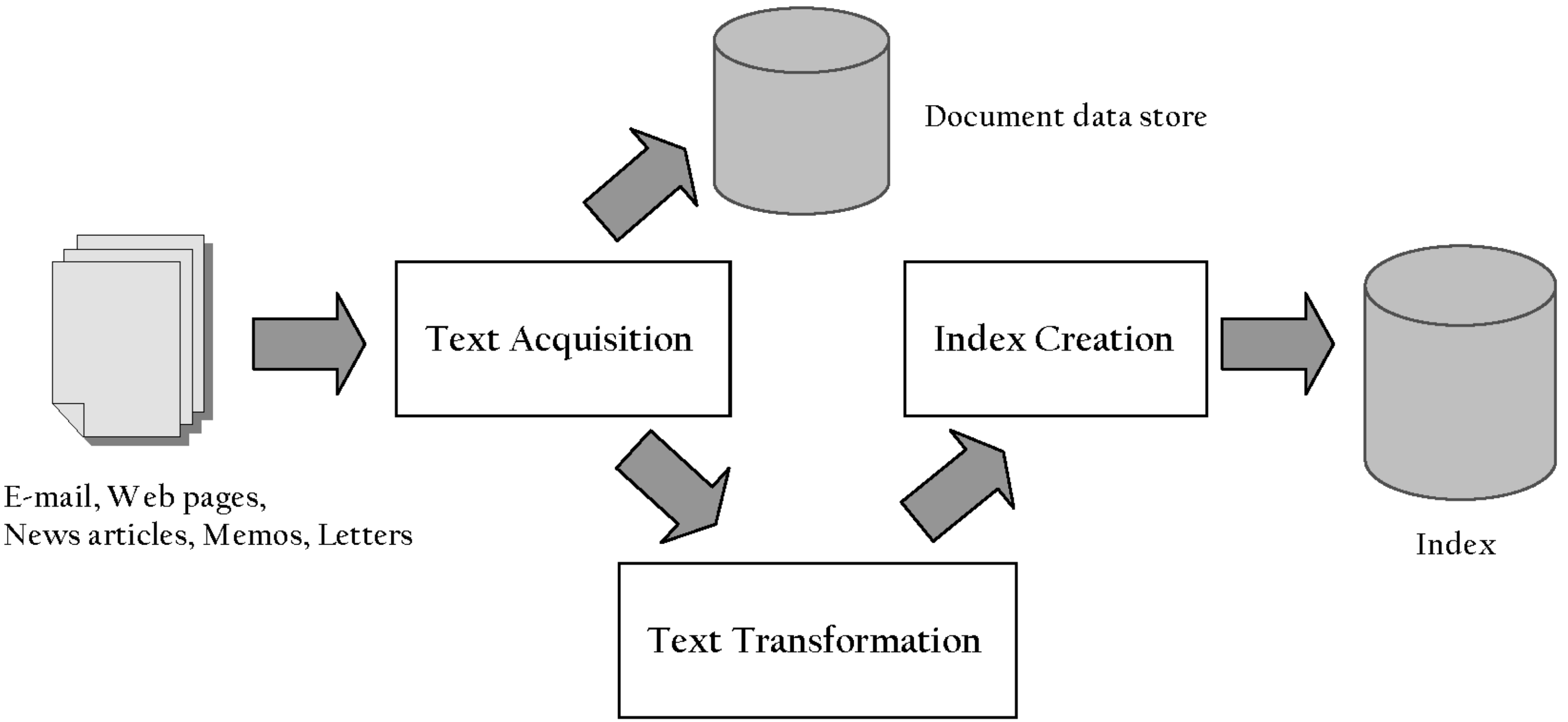
Indexing

- UCSB 290N.
- Mainly based on slides from the text books of Croft/Metzler/Strohman and Manning/Raghavan/Schutze

Table of Content

- Inverted index with positional information
- Compression
- Distributed indexing

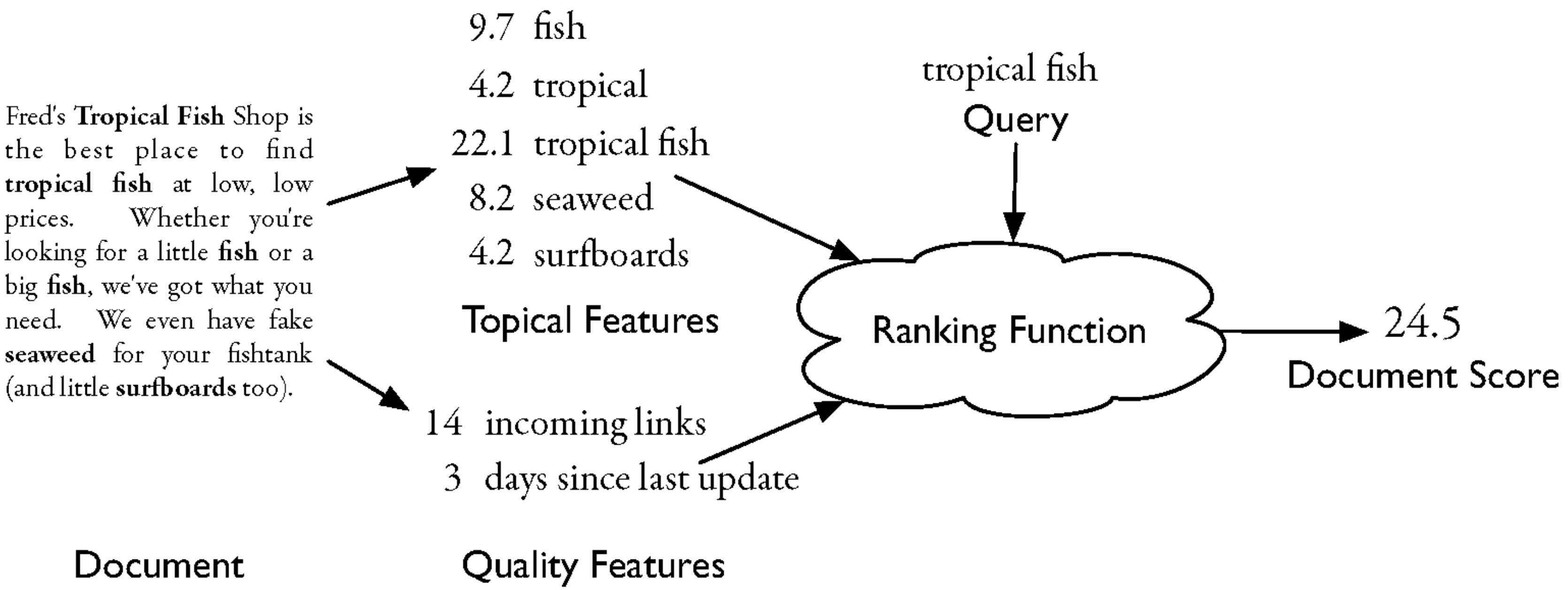
Indexing Process



Indexes

- ***Indexes* are data structures designed to make search faster**
- **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*
- **What is a reasonable abstract model for ranking?**
  - enables discussion of indexes without details of retrieval model

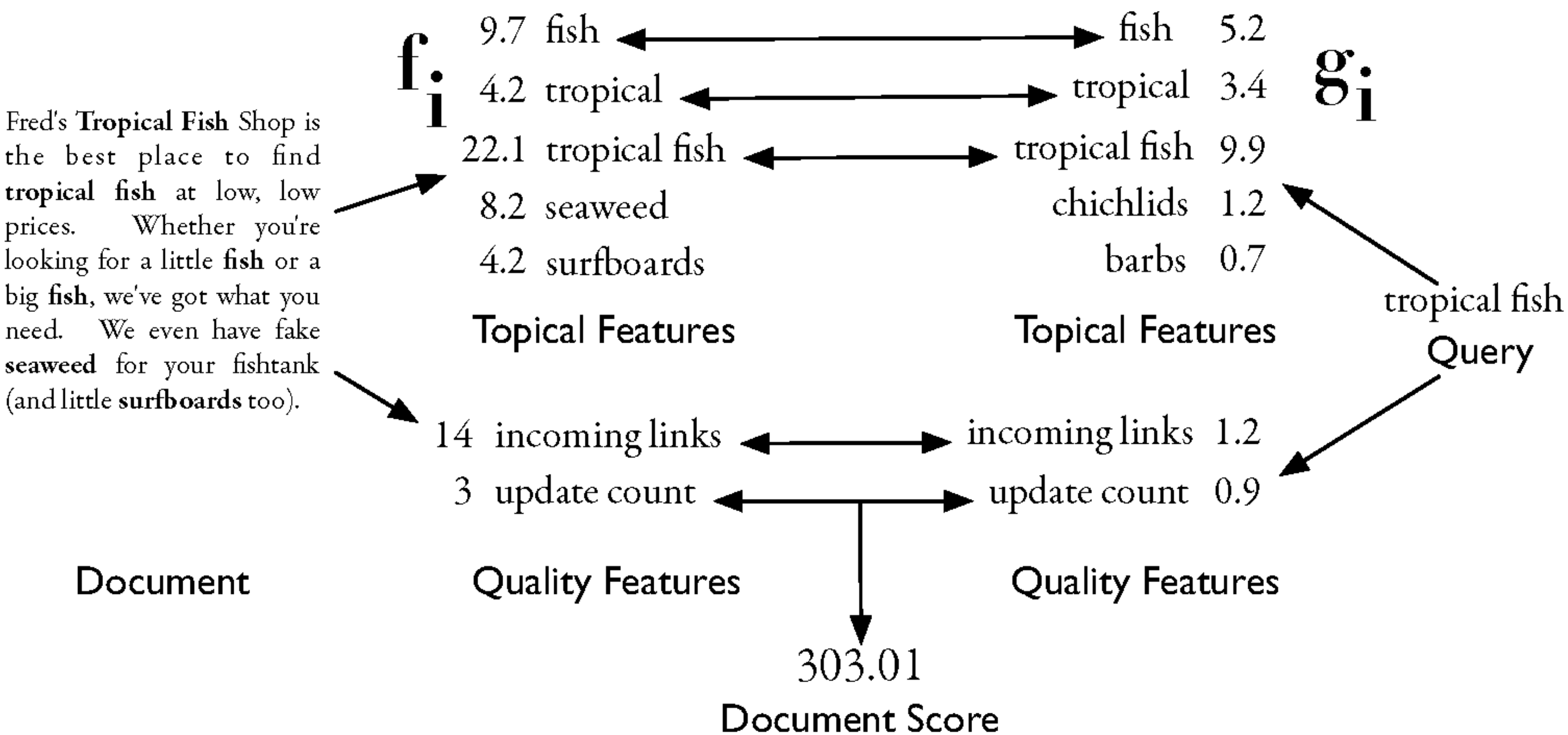
Simple Model of Ranking



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

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

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

Positional indexes

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

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

  doc1: position1, position2 … ;

  doc2: position1, position2 … ;

  etc.>

Positional index example

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

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- **this expands postings storage *substantially***

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 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.,

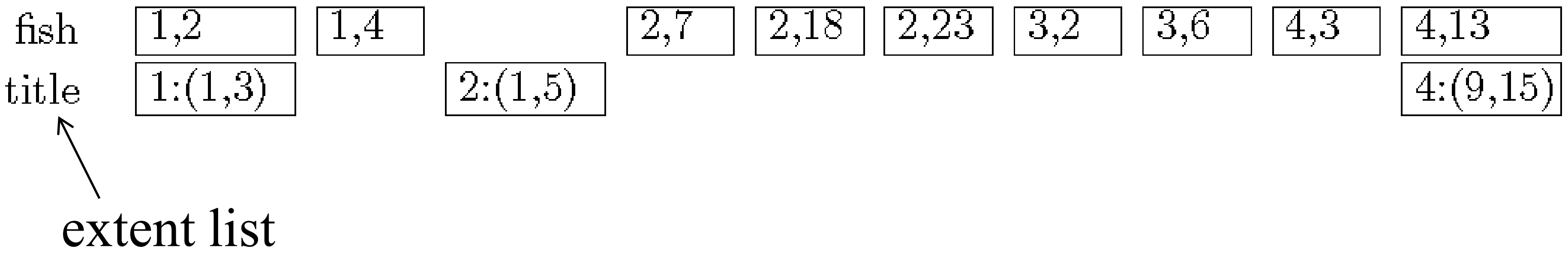
tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

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 to mark special areas in a document*

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

- **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

Issue with data size: Example

- **Number of docs = n = 40M**
- **Number of terms = m = 1M**
- **Use Zipf to estimate number of postings entries:**
  - $n + n/2 + n/3 + \ldots. + n/m \sim n \ln m$ = 560M entries
  - 16-byte (4+8+4) records *(term, doc, freq).*
- **9GB**
- **No positional info yet**

Check for yourself

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
- **Consider a term with frequency 0.1%**

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Compression

- **Inverted lists are very large**
  - 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**

Rules of thumb

- **Positional index size factor of 2-4 over non-positional index**
- **Positional index size 35-50% of volume of original text**
- **Caveat: all of this holds for “English-like” languages**

Compression

- ***Basic idea*: Common data elements use short codes while uncommon data elements use longer codes**
  - Example: coding numbers
    - number sequence: 0, 1, 0, 2,0,3,0
    - 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.,

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

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

$$109, 3766, 453, 1867, 992, ...$$

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**

Elias-γ Code

- **To encode a number *k*, compute**
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$
  - k_d is number of binary digits, encoded in unary

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

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**
  - 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

Elias-δ Code

- **Split 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

```
#
# Generating Elias-gamma and Elias-delta codes in Python
#

import math

def unary_encode(n):
 return "1" * n + "0"

def binary_encode(n, width):
    r = ""
    for i in range(0,width):
     if ((1<<i) & n) > 0:
     r = "1" + r
     else:
     r = "0" + r
    return r

def gamma_encode(n):
    logn = int(math.log(n,2))
    return unary_encode( logn ) + " " + binary_encode(n, logn)

def delta_encode(n):
 logn = int(math.log(n,2))
if n == 1:
 return "0"
 else:
 loglog = int(math.log(logn+1,2))
 residual = logn+1 - int(math.pow(2, loglog))
        return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

if __name__ == "__main__":
    for n in [1,2,3, 6, 15,16,255,1023]:
        logn = int(math.log(n,2))
        loglogn = int(math.log(logn+1,2))
        print n, "d_r", logn
        print n, "d_dd", loglogn
        print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
        print n, "delta", delta_encode(n)
        #print n, "gamma", gamma_encode(n)
        #print n, "binary", binary_encode(n)
```

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:**

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

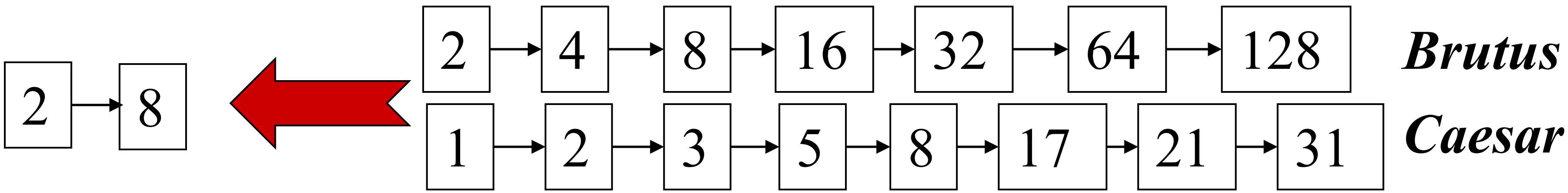
- **Compress using v-byte:**

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

Skip pointers for faster merging of postings

Basic merge

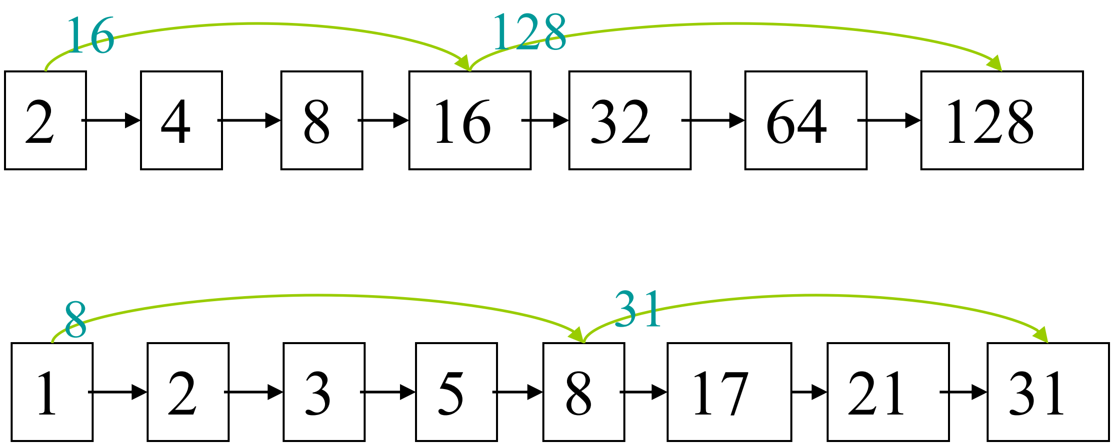
- **Walk through the two postings simultaneously, in time linear in the total number of postings entries**



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

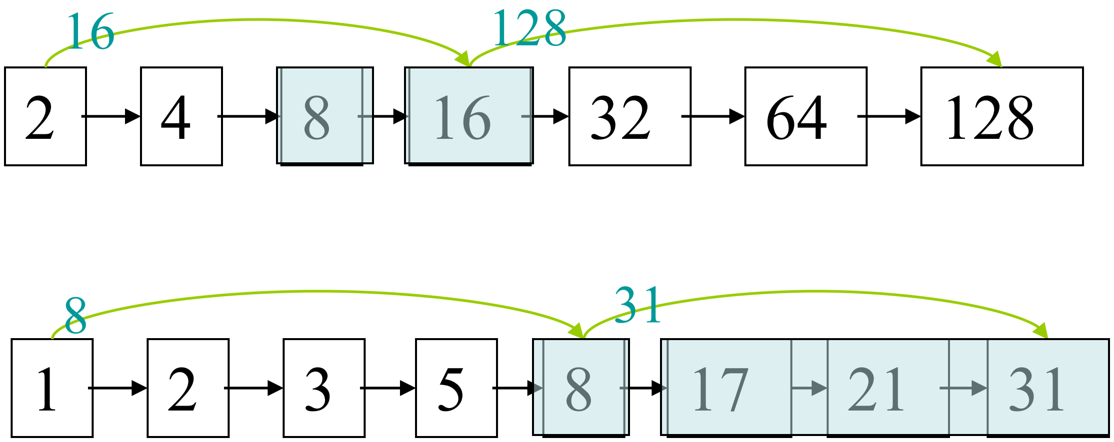
Can we do better?
Yes, if index isn't changing too fast.

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



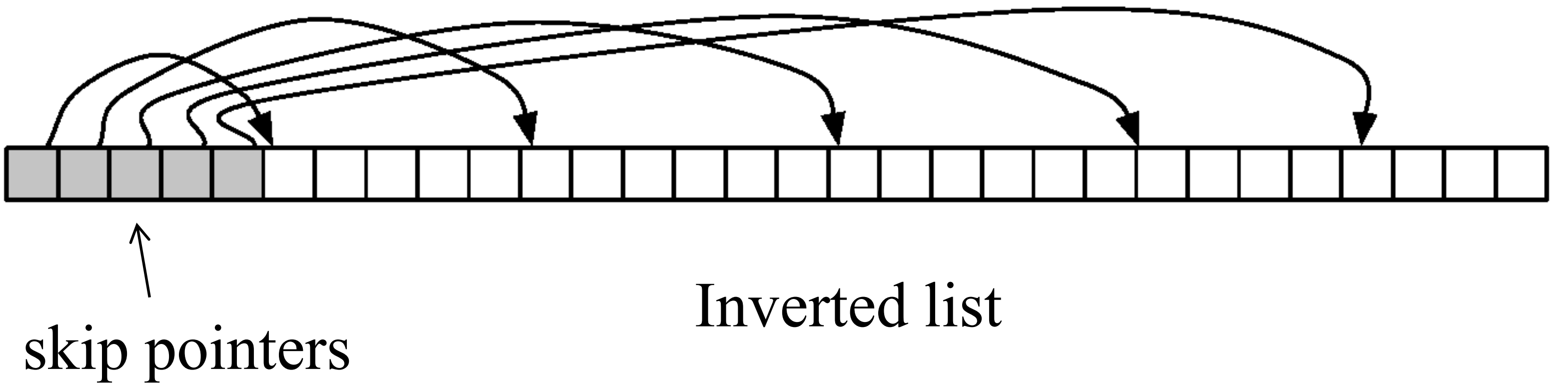
Suppose we've stepped through the lists until we process **8** on each list.

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*



Skip Pointers

- **Example**
  - Inverted list

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

  - D-gaps

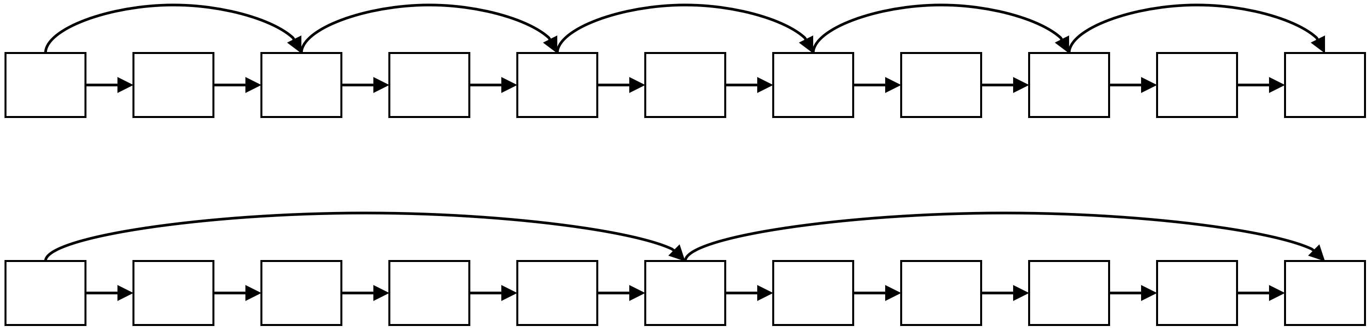
$$5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15$$

  - Skip pointers

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

Where do we place skips?

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



Placing skips

- Simple heuristic: for postings of length *L*, use $\sqrt{L}$ evenly-spaced skip pointers.
- This ignores the distribution of query terms.
- Easy if the index is relatively static; harder if *L* keeps changing because of updates.

Auxiliary Structures

- **Inverted lists usually stored together in a single file for efficiency**
  - *Inverted file*
- ***Vocabulary* or *lexicon***
  - Contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - Either hash table in memory or B-tree for larger vocabularies
- **Term statistics stored at start of inverted lists**
- **Collection statistics stored in separate file**

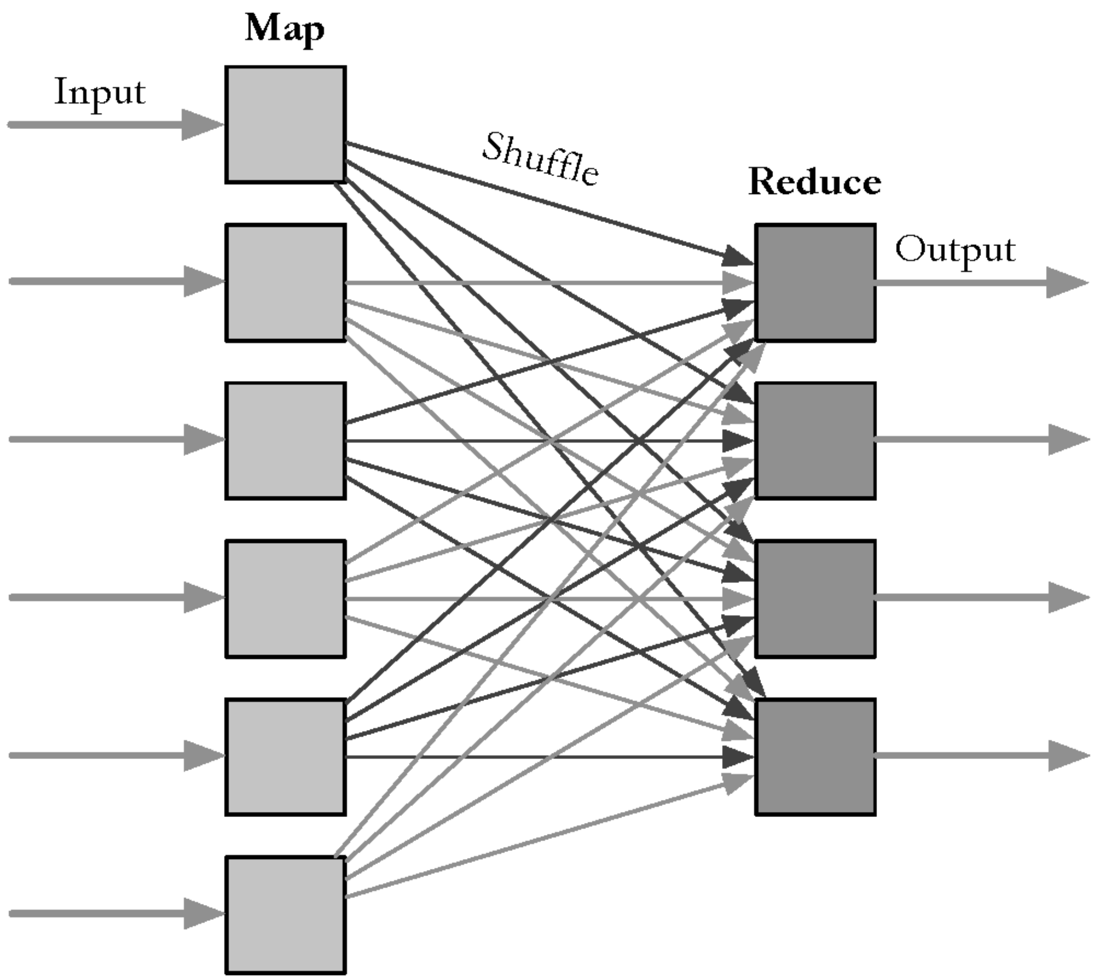
Distributed Indexing

- **Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)**
- **Large numbers of inexpensive servers used rather than larger, more expensive machines**
- ***MapReduce* is a distributed programming tool designed for indexing and analysis tasks**

MapReduce

- **Distributed programming framework that focuses on data placement and distribution**
- ***Mapper***
  - Generally, transforms a list of items into another list of items of the same length
- ***Reducer***
  - Transforms a list of items into a single item
  - Definitions not so strict in terms of number of outputs
- **Many mapper and reducer tasks on a cluster of machines**

MapReduce



MapReduce

- **Basic process**
  - *Map* stage which transforms data records into pairs, each with a key and a value
  - *Shuffle* uses a hash function so that all pairs with the same key end up next to each other and on the same machine
  - *Reduce* stage processes records in batches, where all pairs with the same key are processed at the same time
- ***Idempotence* of Mapper and Reducer provides fault tolerance**
  - multiple operations on same input gives same output

Indexing Example

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, number:position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```