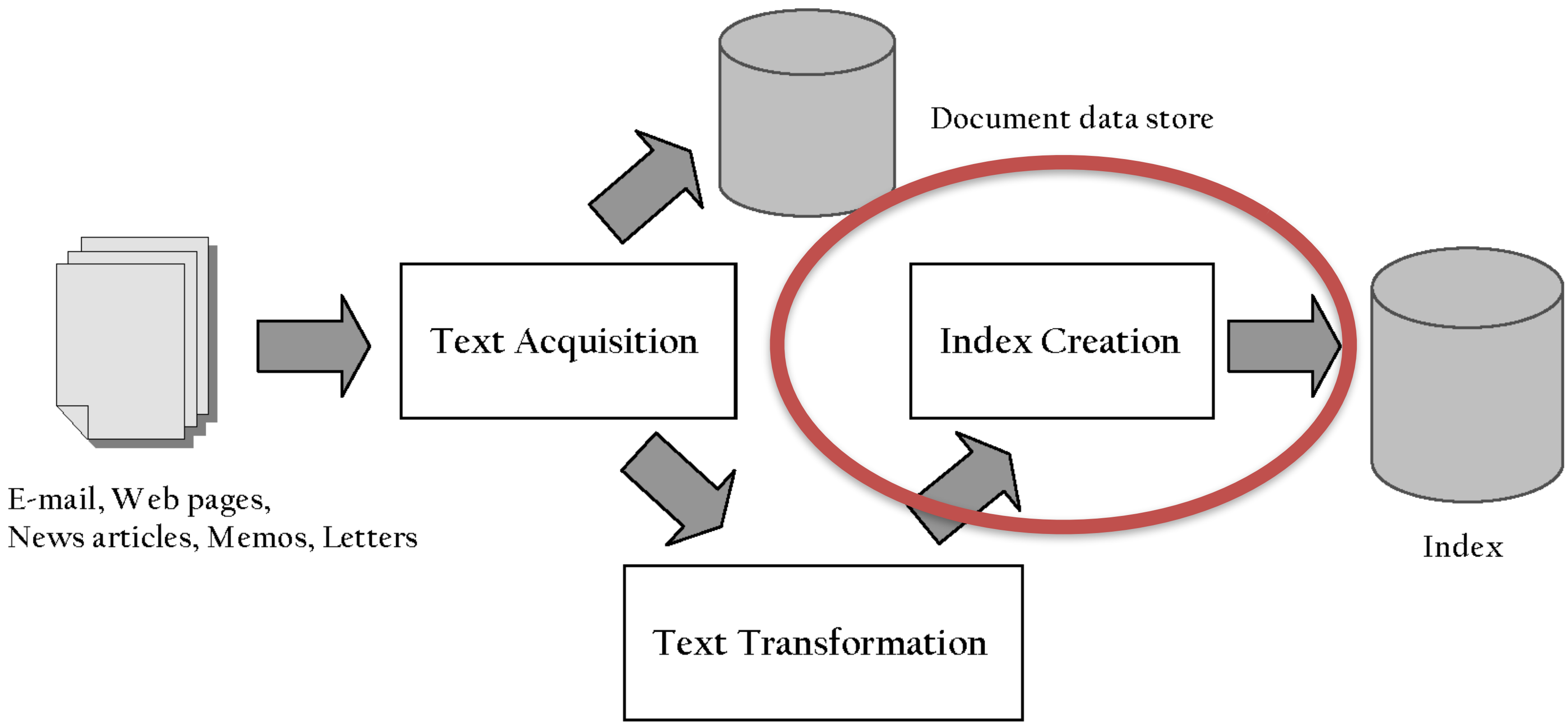
CS6200
Information Retrieval

David Smith
College of Computer and Information Science
Northeastern University

Indexing Process



Indexes

Storing document information for faster queries

Indexes

- *Indexes* are data structures designed to make search faster
  – The main goal is to store whatever we need in order to minimize processing at query time
- Text search has unique requirements, which leads to unique data structures
- Most common data structure is *inverted index*
  – A *forward index* stores the terms for each document
    - As seen in the back of a book
  – An *inverted index* stores the documents for each term
    - Similar to a *concordance*

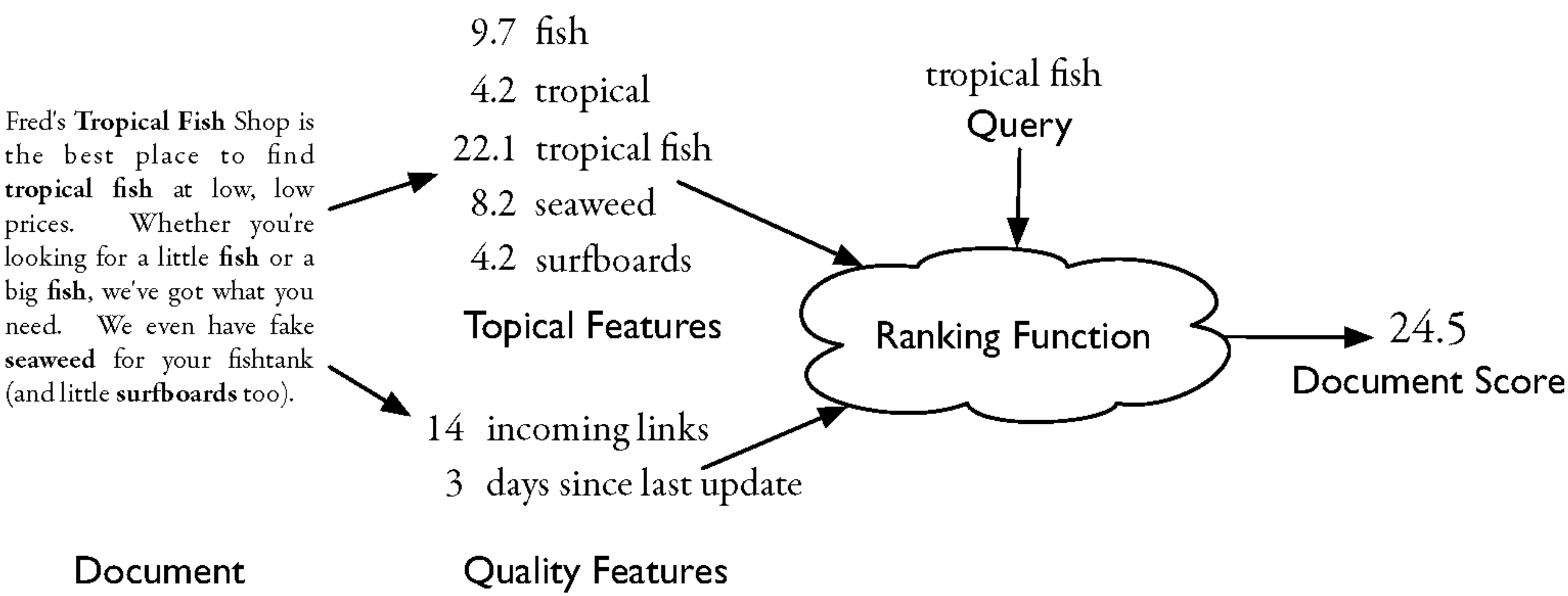
A Shakespeare Concordance

IRISHMAN—an Irishman with.... *Merry Wives*, ii. 2
altogether directed by an Irishman .. *Henry V.* iii. 2
IRISHMEN-against the Irishmen? .2 *Henry VI.* iii. 1
IRK—and yet it irks me............ *As you Like it*, ii. 1
it irks his heart, he cannot ....1 *Henry VI.* i. 4
it irks my very soul ....3 *Henry VI.* ii. 2
IRKSOME—was irksome to me .. *As you Like it*, iii. 5
is an irksome brawling scold .. *Taming of Shrew*, i. 2
irksome is this music to my heart!..2 *Henry VI.* ii. 1
IRON—to wear iron about you.... *Twelfth Night*, iii. 4
my young soldier, put up your iron.. — iv. 1
before barred up with ribs of iron! .. *Much Ado*, iv. 1
runs not this speech like iron through — v. 1
but yet you draw not iron...... *Mid. N.'s Dream*, ii. 2
the iron tongue of midnight hath .... — v. 1
iron may hold with her ...... *Taming of Shrew*, ii. 1
fetch me an iron crow........ *Comedy of Errors*, iii. 1
their iron indignation 'gainst your.. *King John*, ii. 1
with his iron tongue and brazen mouth — iii. 3
heat me these irons hot .... — iv. 1
must you with hot irons burn (*rep.*).. — iv. 1
none, but in this iron age .... — iv. 1
stubborn hard than hammered iron? — iv. 1

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?
  - This will allow us to discuss indexes without deciding the details of the retrieval model

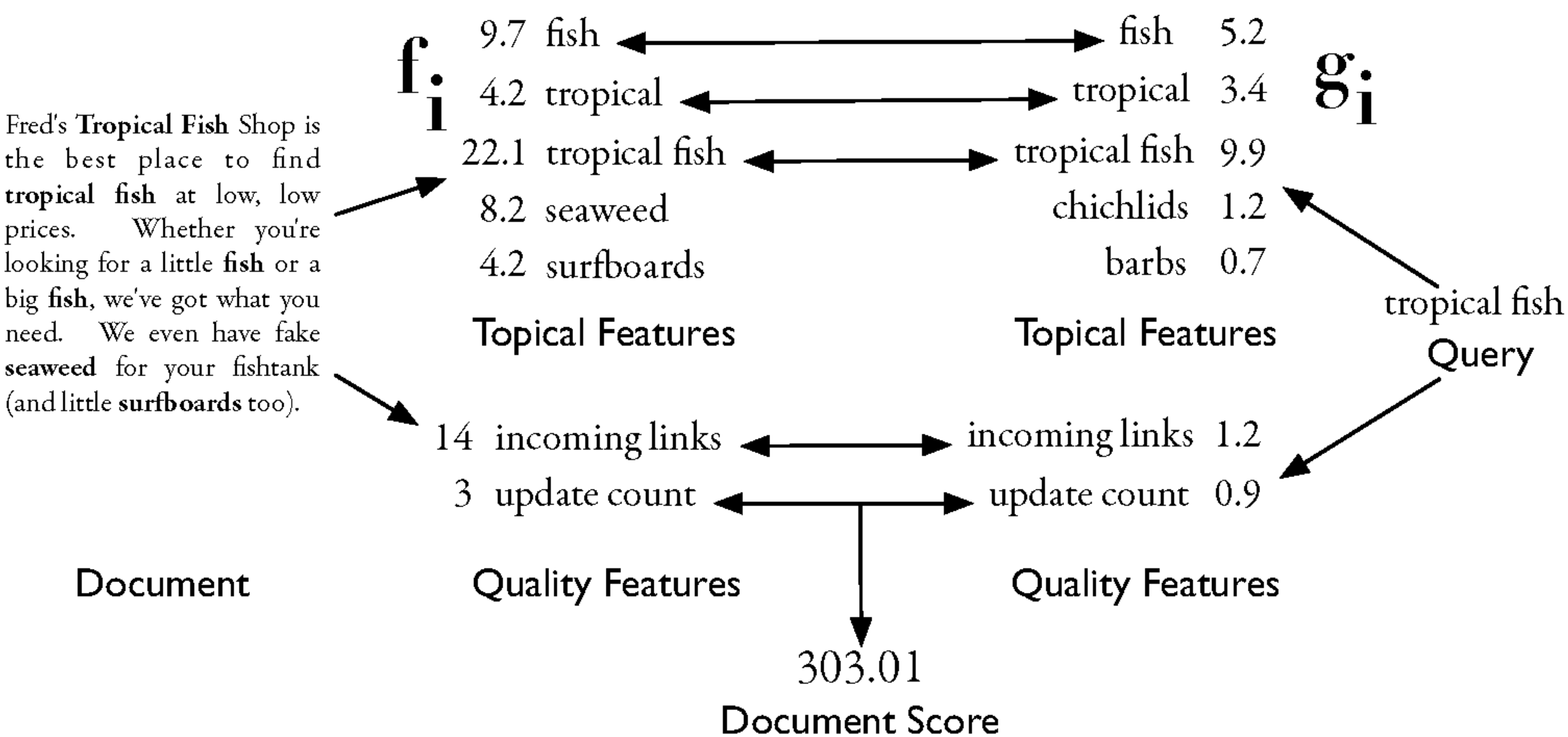
Abstract Model of Ranking



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

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		

Inverted Index with positions

- supports proximity matches

Term	Postings
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.,

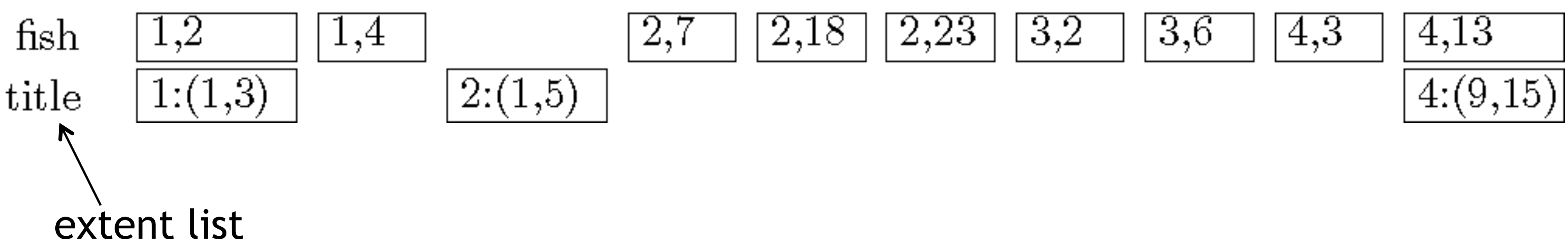
tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

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

- 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

Index Compression

Managing index size efficiently

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:

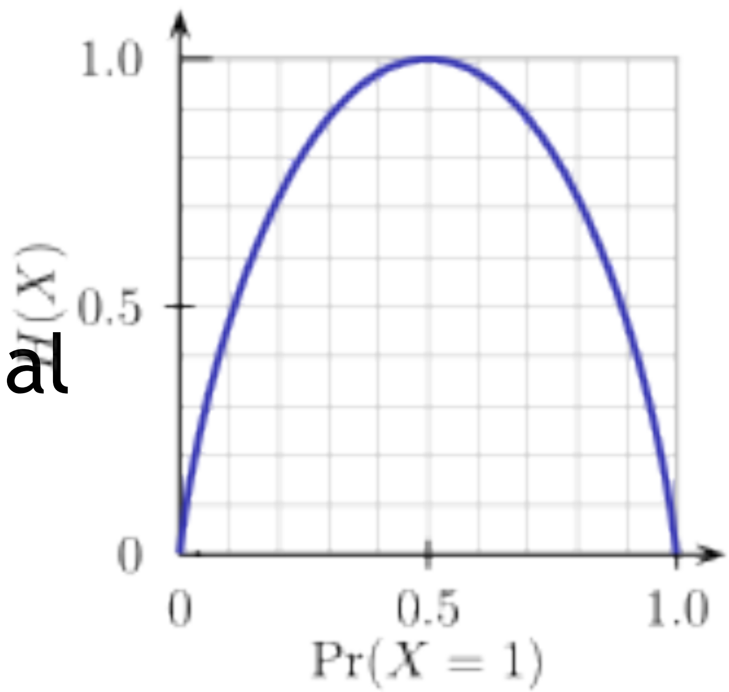
$$0\ 101\ 0\ 111\ 0\ 110\ 0$$

Compression and Entropy

- **Entropy** measures “randomness”
  - Inverse of compressability

$$H(X) \equiv -\sum_{i=1}^{n} p(X = x_i) \log_2 p(X = x_i)$$

  - Log2: measured in *bits*
  - Upper bound: log *n*
  - Example curve for binomial



Compression and Entropy

- Entropy bounds compression rate
  - Theorem: $H(X) \leq E[\ |\text{encoded}(X)|\ ]$
  - Recall: $H(X) \leq \log(n)$
  - n is the size of the domain of X
- Standard binary encoding of integers optimizes for the worst case where choice of numbers is completely unpredictable
- It turns out, we can do better. At best:
  - $H(X) \leq E[\ |\text{encoded}(X)|\ ] < H(X) + 1$
  - Bound achieved by *Huffman codes*

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*)
  - makes the posting list more compressible

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

- More efficient when smaller numbers are more common
- Can handle very large integers
- 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 inverted list with counts & positions — (doc, count, 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:

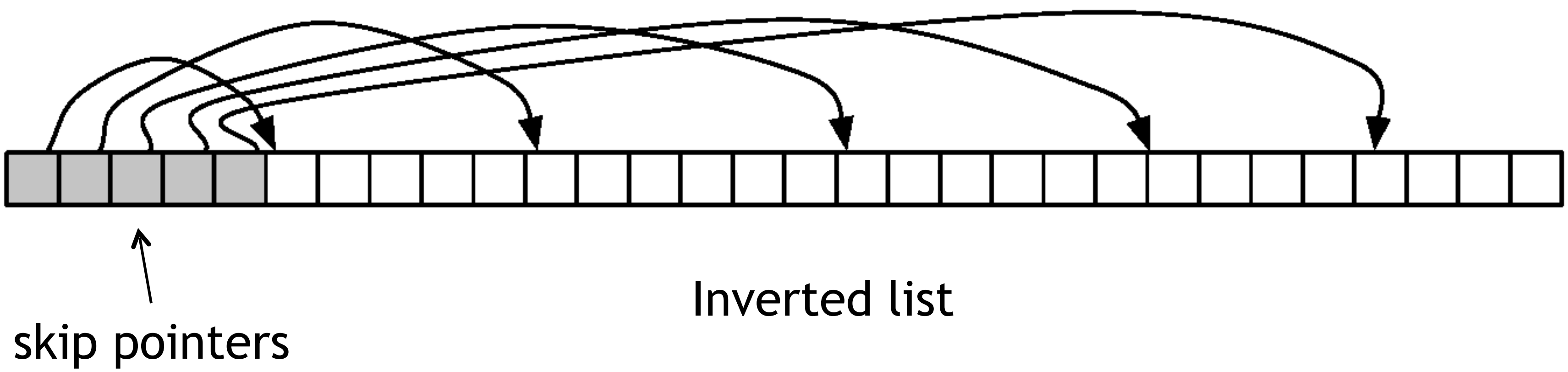
```
81 82 81 86 81 82 86 8B 01 B4 81 81 81
```

Skipping

- Search involves comparison of inverted lists of different lengths
  - Finding a particular doc is very inefficient
  - “Skipping” ahead to check document numbers is much better
  - Compression makes this difficult
    - Variable size, only d-gaps stored
- Skip pointers are additional data structure to support skipping

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 of doc numbers

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

  - D-gaps

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

  - Skip pointers

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

Auxiliary Structures

- Inverted lists often 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
- For very large indexes, distributed filesystems are used instead.

Index Construction

Algorithms for indexing

Index Construction

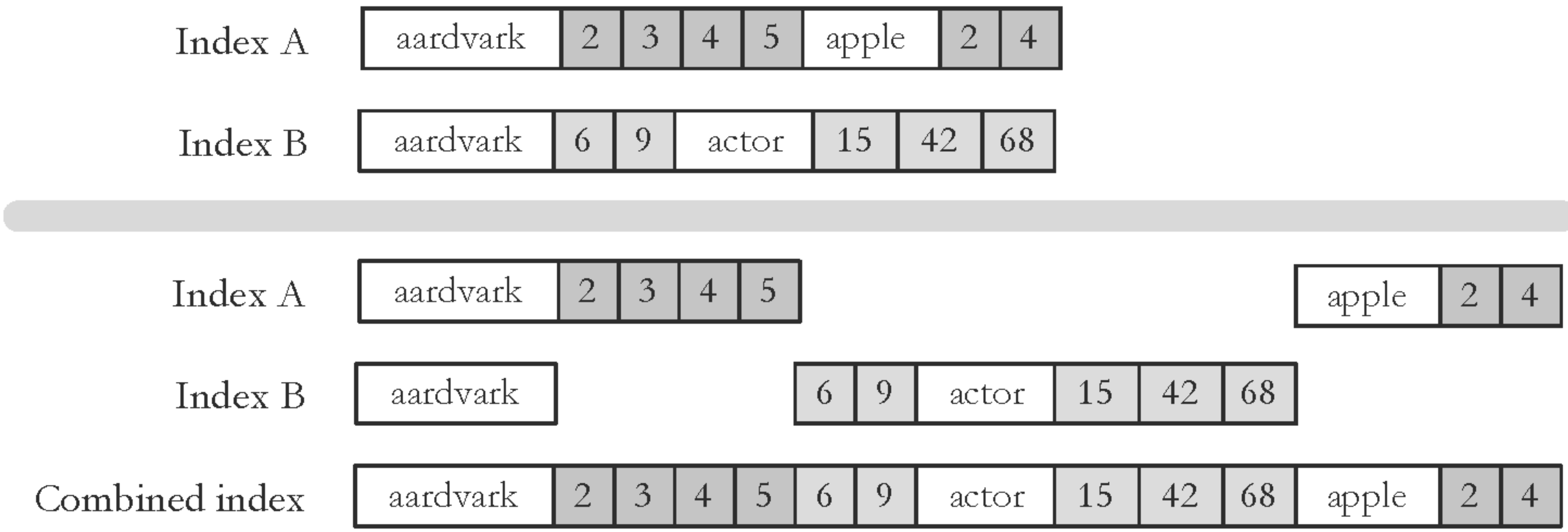
- Simple in-memory indexer

```
procedure BuildIndex(D)                                  ▷ D is a set of text documents
    I ← HashTable()                                             ▷ Inverted list storage
    n ← 0                                                       ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                                      ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

Merging

- Merging addresses limited memory problem
  - Build the inverted list structure until memory runs out
  - Then write the partial index to disk, start making a new one
  - At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists must be designed so they can be merged in small pieces
  - e.g., storing in alphabetical order

Merging


Index A
aardvark 2 3 4 5 apple 2 4
Index B
aardvark 6 9 actor 15 42 68
Index A
aardvark 2 3 4 5
apple 2 4
Index B
aardvark
6 9 actor 15 42 68
Combined index
aardvark 2 3 4 5 6 9 actor 15 42 68 apple 2 4

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

Example

- Given a large text file that contains data about credit card transactions
  - Each line of the file contains a credit card number and an amount of money
  - Determine the number of unique credit card numbers
- Could use hash table – memory problems
  - counting is simple with sorted file
- Similar with distributed approach
  - sorting and placement are crucial

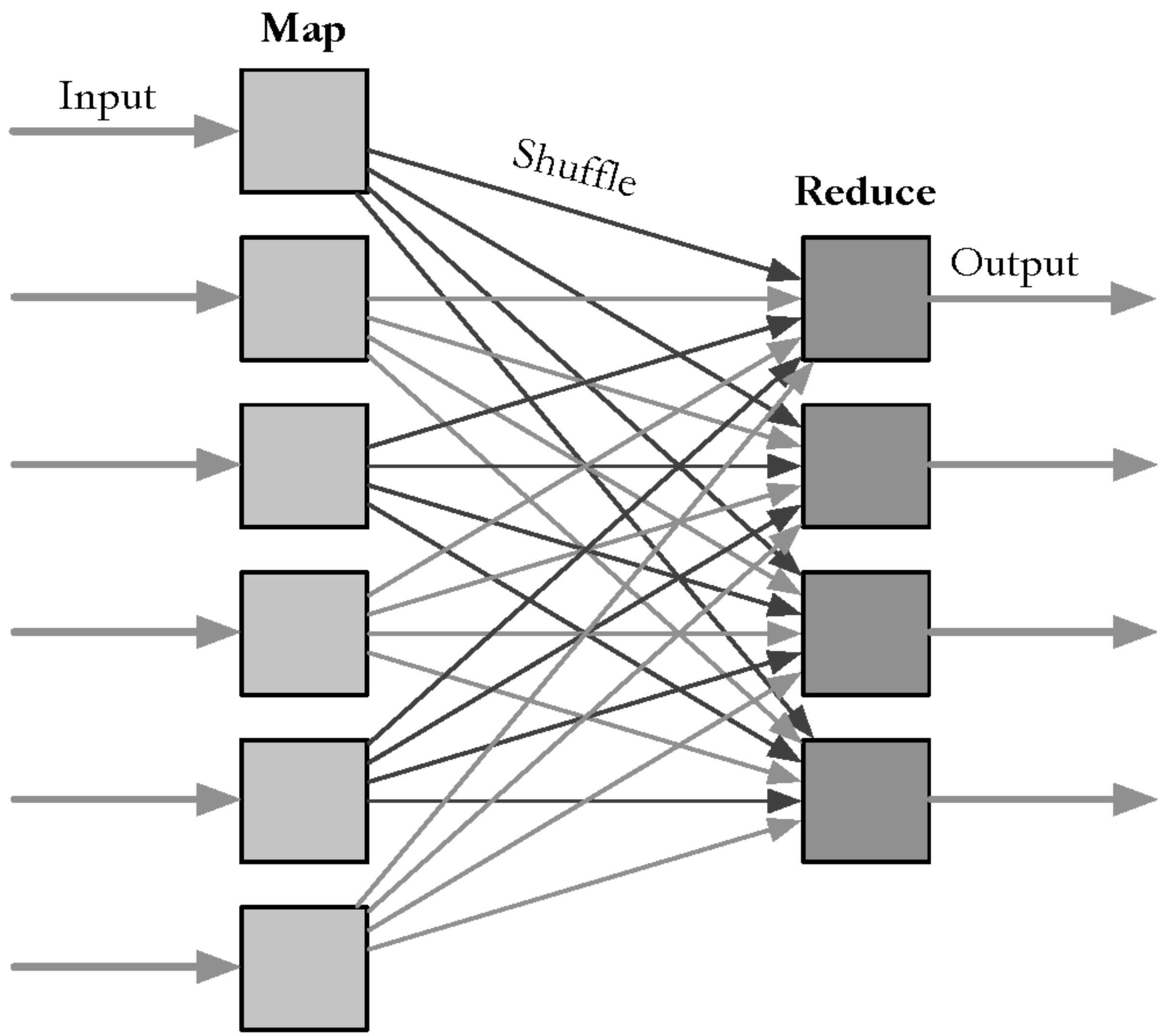
MapReduce

- 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

- 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

MapReduce



Example

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure

procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
```

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
```

Result Merging

- Index merging is a good strategy for handling updates when they come in large batches
- For small updates this is very inefficient
  - instead, create separate index for new documents, merge *results* from both searches
  - could be in-memory, fast to update and search
- Deletions handled using *delete list*
  - Modifications done by putting old version on delete list, adding new version to new documents index

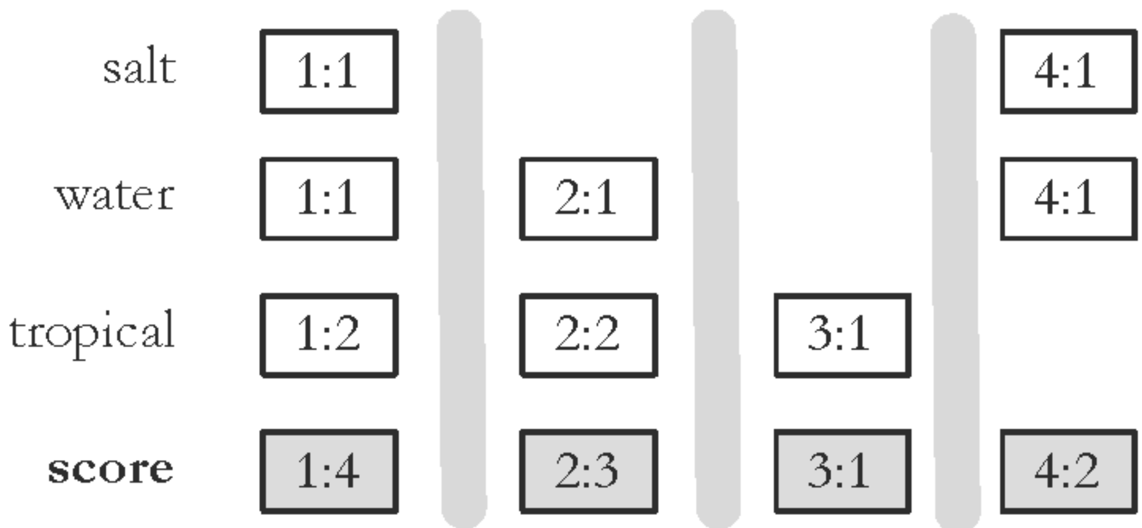
Query Processing

Using the index to search efficiently

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
  - Accumulates scores for documents by processing term lists one at a time
- Both approaches have optimization techniques that significantly reduce time required to generate scores

Document-At-A-Time



Pseudocode Function Descriptions

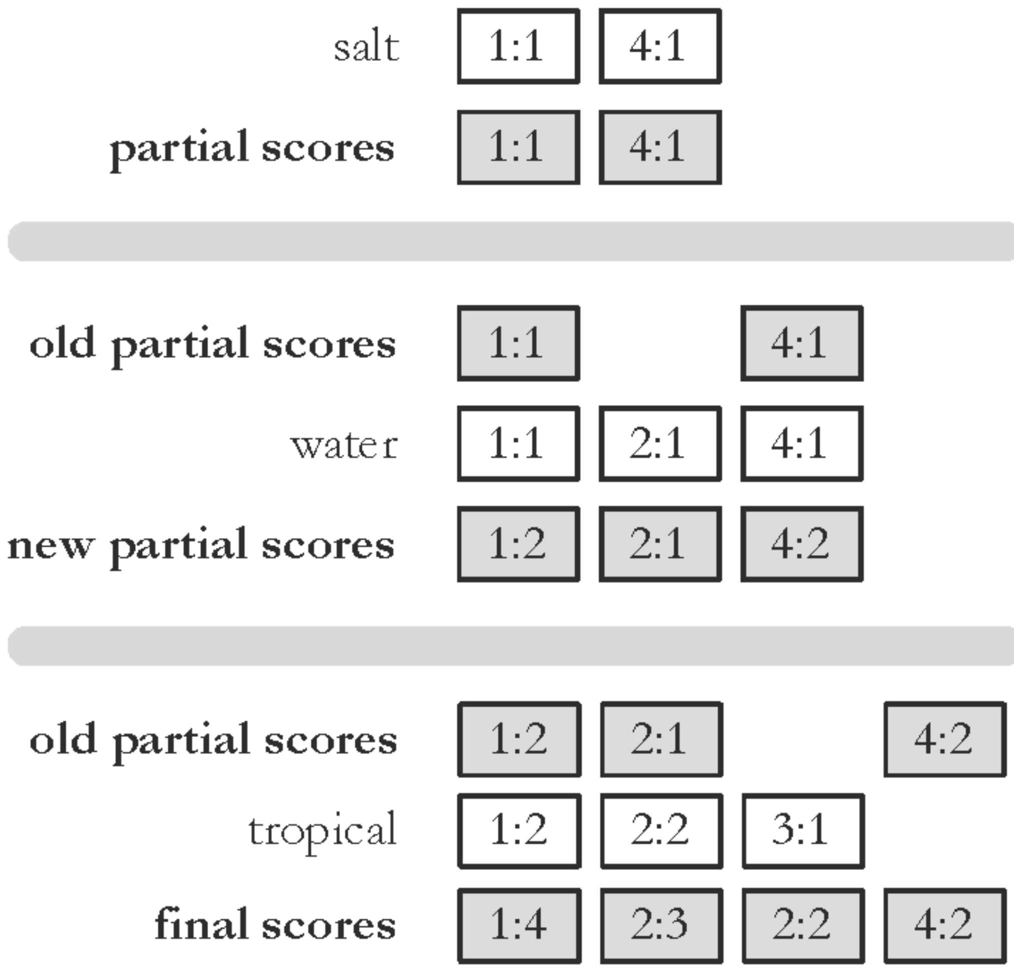
- getCurrentDocument()
  - Returns the document number of the current posting of the inverted list.
- skipForwardToDocument(d)
  - Moves forward in the inverted list until getCurrentDocument() <= d. This function may read to the end of the list.
- movePastDocument(d)
  - Moves forward in the inverted list until getCurrentDocument() < d.
- moveToNextDocument()
  - Moves to the next document in the list. Equivalent to movePastDocument(getCurrentDocument()).
- getNextAccumulator(d)
  - returns the first document number d' >= d that has already has an accumulator.
- removeAccumulatorsBetween(a, b)
  - Removes all accumulators for documents numbers between a and b. A_d will be removed iff $a < d < b$.

Document-At-A-Time

Get best k documents for query Q from index I, with query score function g() and document score function f(). Process one document at a time.

procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
  $L \leftarrow$ Array()
  $R \leftarrow$ PriorityQueue(k)
  for all terms w_i in Q **do**
    $l_i \leftarrow$ InvertedList(w_i, I)
    L.add(l_i)
  end for
  for all documents $d \in I$ **do**
    $s_d \leftarrow 0$
    for all inverted lists l_i in L **do**
      if l_i.getCurrentDocument() $= d$ **then**
        $s_d \leftarrow s_d + g_i(Q) f_i(l_i)$ ▷ Update the document score
      end if
      l_i.movePastDocument(d)
    end for
    R.add(s_d, d)
  end for
  return the top k results from R
end procedure

Term-At-A-Time



Term-At-A-Time

Get best k documents for query Q from index I, with query score function g() and document score function f(). Process one term at a time.

procedure TermAtATimeRetrieval(Q, I, f, g k)
    $A \leftarrow$ HashTable()
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue(k)
    for all terms w_i in Q **do**
        $l_i \leftarrow$ InvertedList(w_i, I)
        L.add(l_i)
    end for
    for all lists $l_i \in L$ **do**
        while l_i is not finished **do**
            $d \leftarrow l_i$.getCurrentDocument()
            $A_d \leftarrow A_d + g_i(Q)f(l_i)$
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A **do**
        $s_d \leftarrow A_d$ ▷ Accumulator contains the document score
        R.add(s_d, d)
    end for
    return the top k results from R
end procedure

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently
- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists
    - better for simple feature functions
  - Calculate scores for fewer documents
    - e.g., conjunctive processing
    - better for complex feature functions

Conjunctive Term-at-a-Time

```
procedure TermAtATimeRetrieval(Q, I, f, g, k)
    A ← Map()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        d_0 ← −1
        while l_i is not finished do
            if i = 0 then
                d ← l_i.getCurrentDocument()
                A_d ← A_d + g_i(Q)f(l_i)
                l_i.moveToNextDocument()
            else
                d ← l_i.getCurrentDocument()
                d′ ← A.getNextAccumulator(d)
                A.removeAccumulatorsBetween(d_0, d′)
                if d = d′ then
                    A_d ← A_d + g_i(Q)f(l_i)
                    l_i.moveToNextDocument()
                else
                    l_i.skipForwardToDocument(d′)
                end if
                d_0 ← d′
            end if
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                    ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Conjunctive Document-at-a-Time

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    d ← −1
    while all lists in L are not finished do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() > d then
                d ← l_i.getCurrentDocument()
            end if
        end for
        for all inverted lists l_i in L do
            l_i.skipForwardToDocument(d)
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q)f_i(l_i)          ▷ Update the document score
                l_i.movePastDocument( d )
            else
                d ← −1
                break
            end if
        end for
        if d > −1 then R.add( s_d, d )
        end if
    end while
    return the top k results from R
end procedure
```

Threshold Methods

- Threshold methods use the number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small
- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
  - score of the *k*th-highest scoring document
  - gives *threshold* τ
  - optimization methods estimate τ' to ignore documents

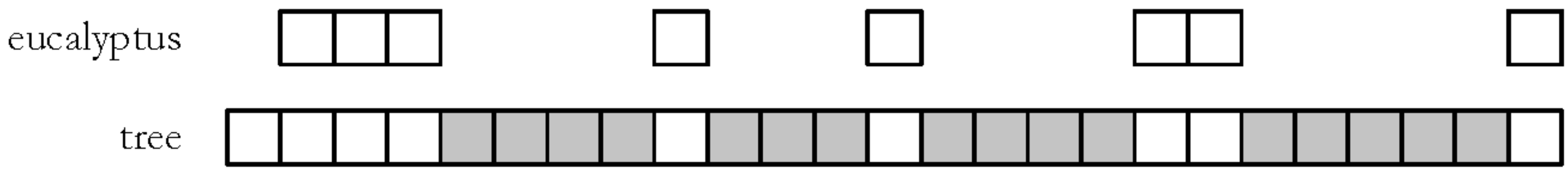
Threshold Methods

- Example: find the top 2 documents
  - Query term weights: [0.7, 0.1, 0.2]
  - Doc term weights are between 0 and 1
  - Ranker uses dot product of query and doc weights
- Doc 1 term weights: [0.3, 0.4, 0.5]
  - Score: 0.3*0.7 + 0.4*0.1 + 0.5*0.2 = 0.35
- Doc 2 term weights: [0.5, 0.1, 0.1]
  - Score: 0.5*0.7 + 0.1*0.1 + 0.1*0.2 = 0.38
- Doc 3 term weights: [0.01, 1, 1]
  - Score: 0.01*0.7 +1*0.1 + 1*0.2 = 0.307
  - We know from the first term that doc 3 can't possibly get a high enough score to beat docs 1 and 2
  - We can discard the document after looking at just one term

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table
- *MaxScore* method compares the maximum score that remaining documents could have to τ'
  - uses the maximum score observed in term posting lists to estimate the best possible document score
  - *safe* optimization in that ranking will be the same without optimization (cf. A* search)

MaxScore Example



- Indexer computes μ_{tree}
  - maximum score any document got for term "tree"
- Assume k =3, τ' is lowest score for entire query after first three docs
- Likely that $\tau' > \mu_{tree}$ because of additional terms
  - τ' is the score of a document that contains both query terms
- Can safely skip over all gray postings, which have scores < μ_{tree}

Other Approaches

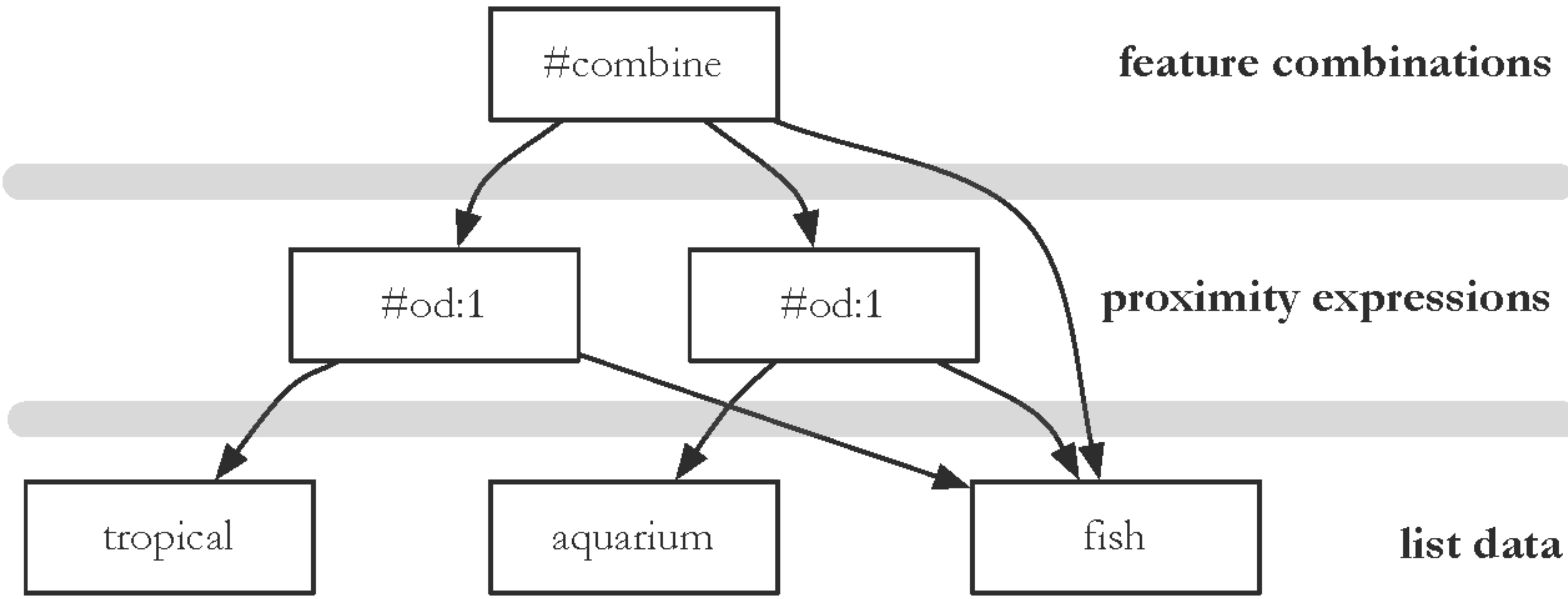
- Early termination of query processing
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization
- List ordering
  - order inverted lists by quality metric (e.g., PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to produce good documents

Structured Queries

- *Query language* can support specification of complex features
  - similar to SQL for database systems
  - *query translator* converts the user's input into the structured query representation
  - Galago query language is the example used here
  - e.g., Galago query:

#combine(#od:1(tropical fish) #od:1(aquarium fish) fish)

Evaluation Tree for Structured Query



Distributed Evaluation

- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*
  - Each index server does some portion of the query processing
  - Director organizes the results and returns them to the user
- Two main approaches
  - Document distribution
    - by far the most popular
  - Term distribution

Distributed Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection
  - director sends a copy of the query to each of the index servers, each of which returns the top-*k* results
  - results are merged into a single ranked list by the director
- Collection statistics should be shared for effective ranking

Distributed Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - One of the index servers is chosen to process the query
    - usually the one holding the longest inverted list
  - Other index servers send information to that server
  - Final results sent to director

Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular
- Caching can significantly improve effectiveness
  - Cache popular query results
  - Cache common inverted lists
- Inverted list caching can help with unique queries
- Cache must be refreshed to prevent stale data