Ranking with Indexes

406.424 Internet Applications

Jonghun Park

jonghun@snu.ac.kr

Dept. of Industrial Eng.

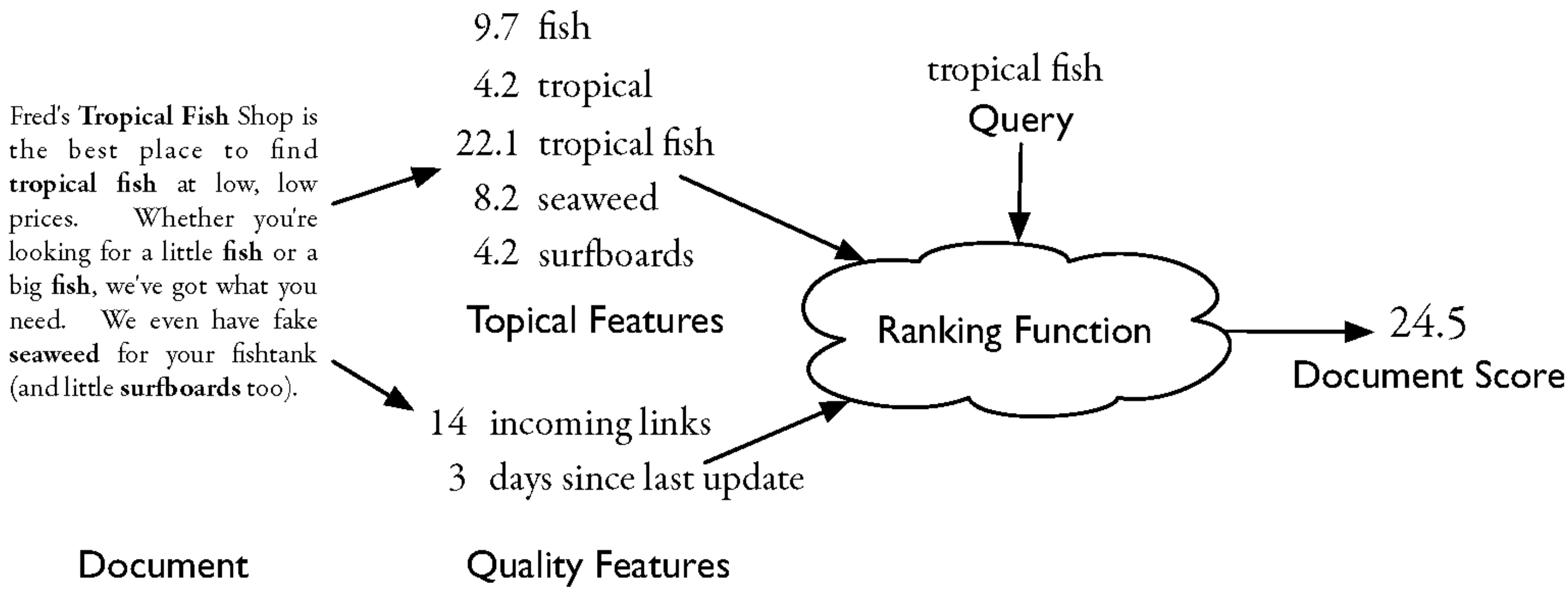
Seoul National University

9/1/2010

indexes and ranking

- 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
- 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?

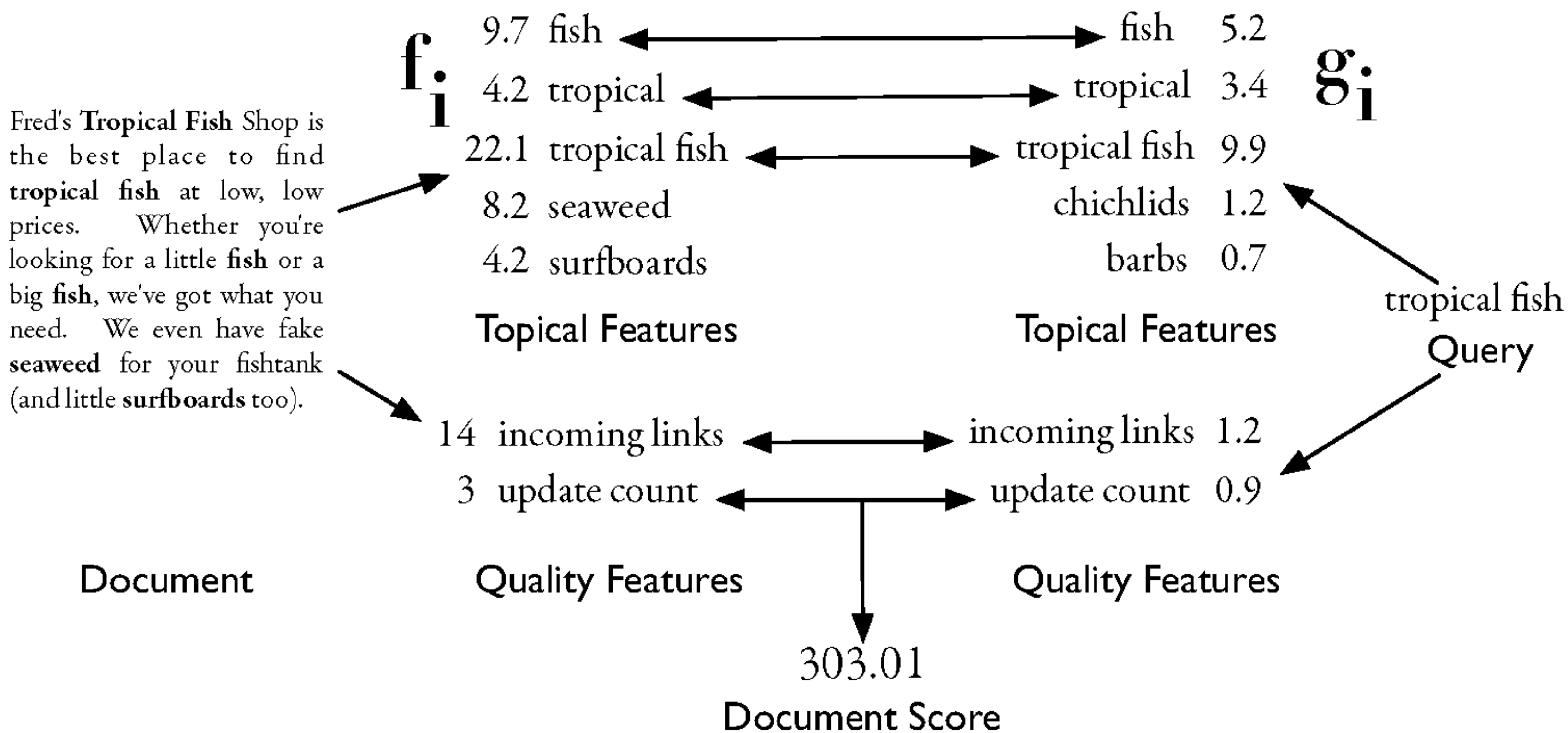
abstract model of ranking



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
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 word 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 word 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.,

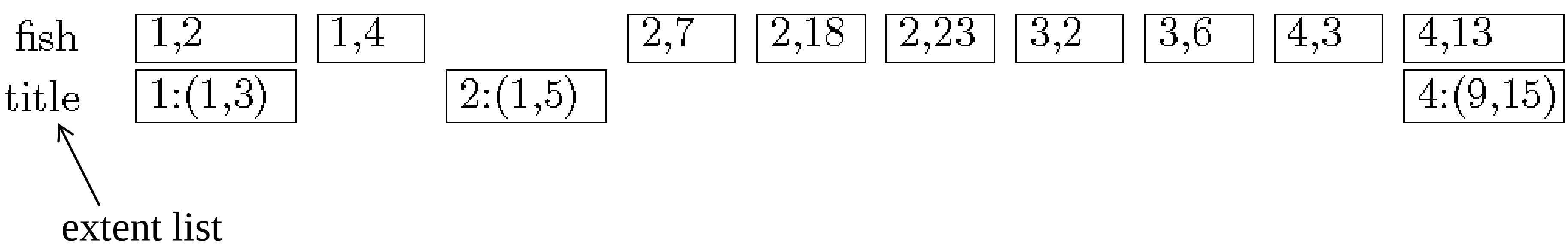
tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

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., (5,9) if title of a book started on the 5th word and ended just before the 9th word



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 (e.g., TF*IDF) for "fish" in 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
  - 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
- basic idea: **common data elements use short codes while uncommon data elements use longer codes**

compression example

- ambiguous encoding
  - given 0, 1, 0, 2, 0, 3, 0
  - a possible encoding: 00 01 00 10 00 11 00
  - another encoding by encoding 0 using a single 0
    - 0 01 0 10 0 11 0: only 10 bits but ambiguous (since the spaces are not stored)
    - it can also be interpreted as 0 01 01 0 0 11 0
  - an unambiguous encoding

Number	Code
0	0
1	101
2	110
3	111

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

- given inverted list (containing doc numbers)

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

- combines the strengths of unary and binary codes
- to encode a number *k*, compute
  - $k_d = \Upsilon\log_2 k - 1/$
  - $k_r = k - 2^{\lfloor \log 2k \rfloor} k_d$
- k_d
  - the **number of binary digits needed to write *k*** in binary form minus 1
  - encoded in unary
  - tells us how many bits to expect
- k_r
  - the remaining binary digits after removing the leftmost binary digit (which is 1) of *k*
- e.g., k = 3
  - $k_d = 1$, $k_r = 1$

Elias-γ code examples

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

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
  - uses short codes for small numbers and longer codes for longer numbers
- shortest v-byte code is 1 byte
- numbers are 1 to 4 bytes
  - low **seven bits** of each byte contain numeric data in binary
  - **high bit is 1 in the last byte**

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

compression example

- assume (document, count, [positions])
- consider inverted lists with positions:

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

- delta encode document numbers and positions:
  - can make the number smaller

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

- compress using v-byte (without the brackets):

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

skipping

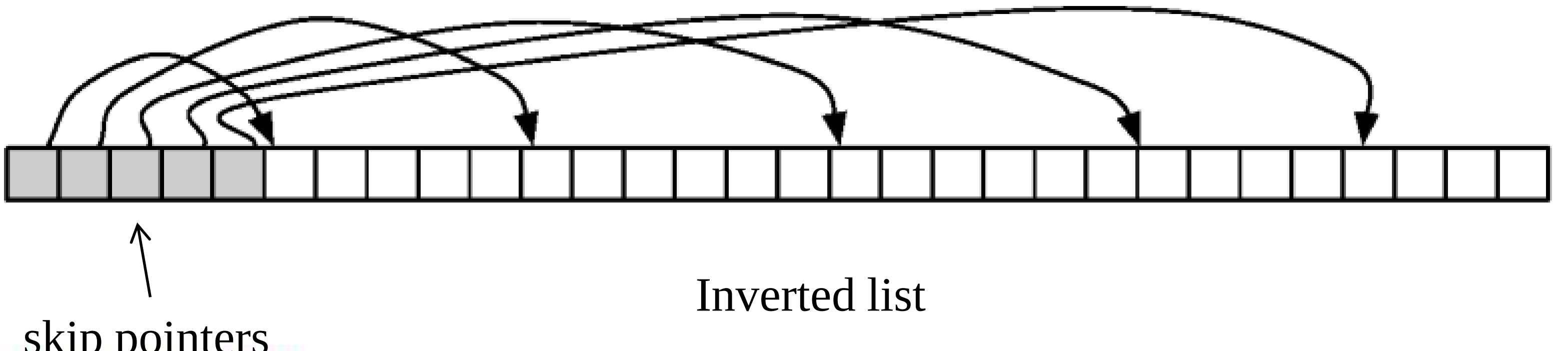
- search involves comparison of inverted lists of **different lengths**
  - can be very inefficient
  - need to **avoid reading all the information** in the inverted list
  - “skipping” ahead to check document numbers is much better
- **skip pointers** are additional data structure to support skipping

need for skipping

- query: “galago AND animal”
  - 300M docs containing animal, and 1M for galago
  - inverted lists for “galago” and “animal” are in doc order
- a very simple algorithm
  - d_g: first doc number in the inverted list for “galago”
  - d_a: first doc number in the inverted list for “animal”
  - while there are still docs in the lists for “galago” and “animal”
    - if $d_g < d_a$, set d_g to the next doc number in the “galago” list
    - if $d_g > d_a$, set d_a to the next doc number in the “animal” list
    - if $d_g = d_a$, the document d_a contains both “galago” and “animal”. move both d_g and d_a to the next doc in the inverted lists respectively
  - very expensive

skip pointer

- better approach
  - every time we find that $d_g > d_a$, we skip ahead k docs in the "animal" list to a new doc s_a
  - if $s_a < d_g$, we skip ahead by another k docs
  - we do this until $s_a >= d_g$
- 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 that the posting **immediately before** it is for document d



skip pointer

- example
  - inverted list with document numbers, uncompressed

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

  - d-gaps

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

  - add some skip pointers
    - e.g., (17, 3): doc number 17 is immediately before position 3

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

- e.g., find the doc number 80 in the list
  - scan the list of skip pointers until we find (52, 12) and (89, 15)
  - start decoding at position 12 in the d-gaps list
  - we find 52 + 5 = 57 and 57 + 23 = 80

auxiliary structures

- inverted lists usually stored together in a single file for efficiency
  - inverted file
- additional directory structure: 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

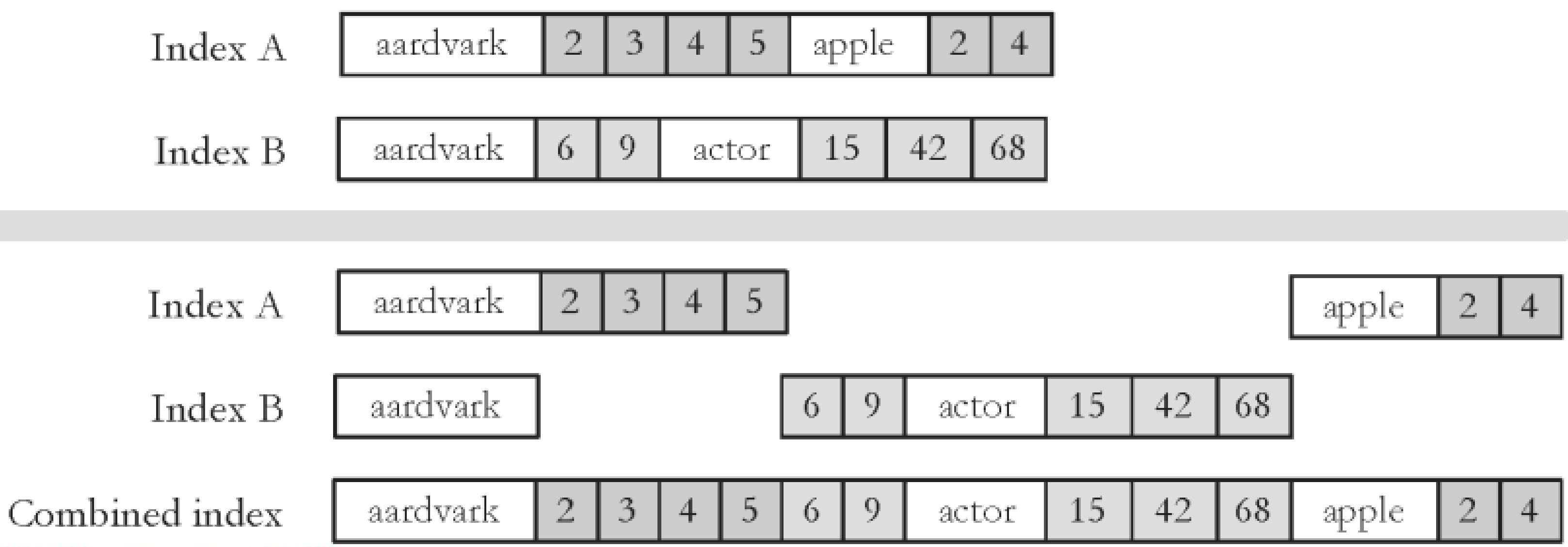
index construction

- simple, sequential in-memory indexer
  - I_t: new inverted list
  - result: a hash table of tokens and inverted lists

```
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

- 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



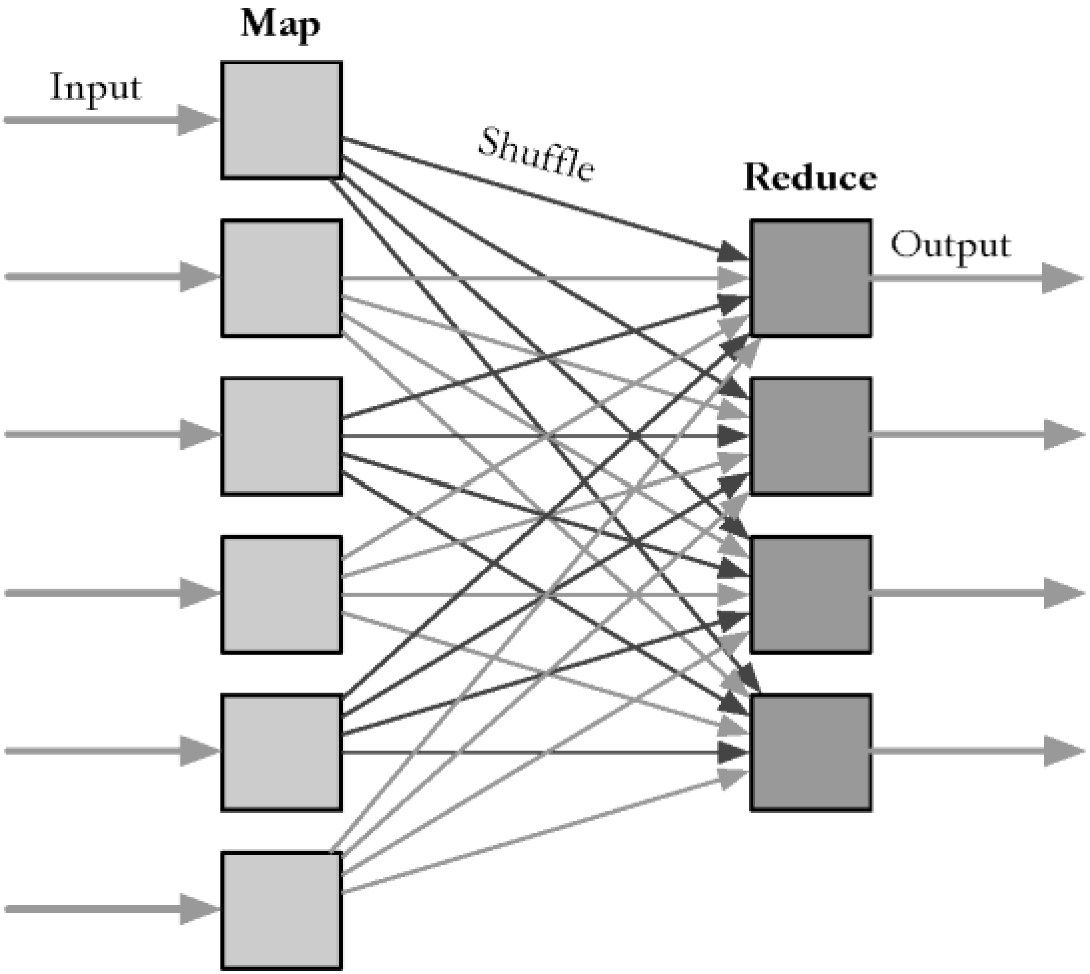
distributed indexing

- 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
    - e.g., (word, document:position)
  - **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

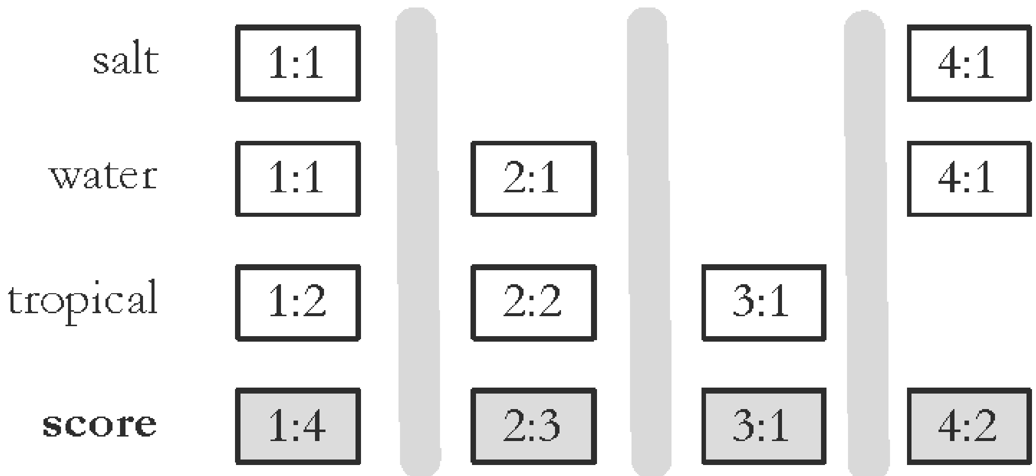
```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, document:position)
            position = position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

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
- Bbth approaches have optimization techniques that significantly reduce time required to generate scores

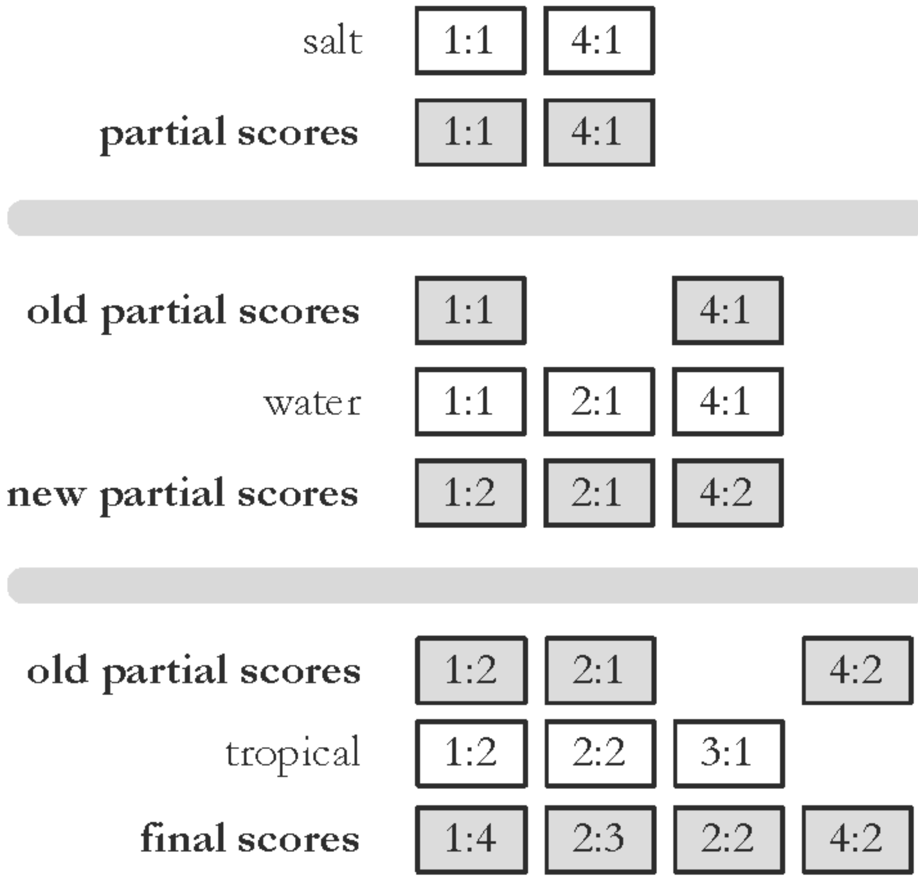
document-at-a-Time



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
    for all documents d ∈ I do
        for all inverted lists l_i in L do
            if l_i points to d then
                s_D ← s_D + g_i(Q)f_i(l_i)          ▷ Update the document score
                l_i.movePastDocument( d )
            end if
        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

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    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
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← 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 ← 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

```
procedure TermAtATimeRetrieval(Q, I, f, g, k)
    A ← LinkedList()
    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)
            else
                d ← l_i.getCurrentDocument()
                d' ← A.getNextAccumulatorAfter(d)
                A.removeAccumulatorsBetween(d_0, d')
                if l_i.getCurrentDocument() = d' then
                    A_d ← A_d + g_i(Q)f(l_i)
                    l_i.moveNextDocument()
                else
                    l_i.skipForwardTo(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 term-at-a-time: works best when one of the query terms is rare

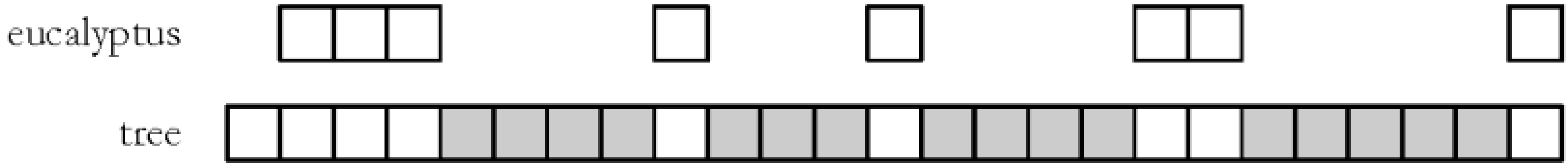
conjunctive document-at-a-time

```
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
    while all lists in L are not finished do
        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 points to d then
                s_d ← s_d + g_i(Q)f_i(l_i)          ▷ Update the document score
                l_i.movePastDocument( d )
            else
                break
            end if
        end for
        R.add( s_d, d )
    end while
    return the top k results from R
end procedure
```

threshold methods

- threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small: 10 or 20
- 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
- **MaxScore** method compares the maximum score that remaining documents could have to τ'
  - **safe** optimization in that ranking will be the same without optimization

MaxScore example



- query: eucalyptus tree
- indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- assume k =3, τ' is lowest score after first three docs containing "eucalyptus" and "tree"
- likely that $\tau' > \mu_{tree}$
  - τ' is the score of a document that contains both query terms
- can safely skip over all gray postings

other approaches

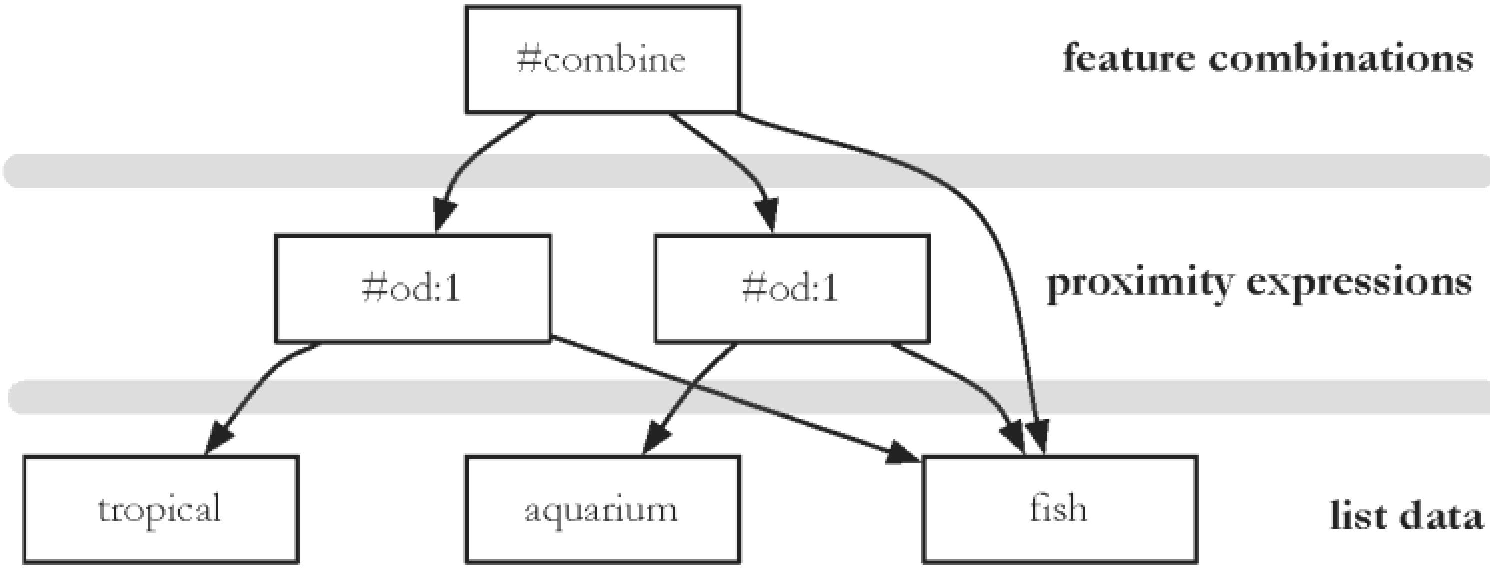
- early termination of query processing
  - simply ignore high-frequency word lists in term-at-a-time
    - similar to using a stopword list
  - ignore documents at end of lists in doc-at-a-time
  - **unsafe** optimization
- list ordering
  - order inverted lists by quality metric (e.g., PageRank) or 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:
    - #od:I: the terms inside it need to appear next to each other in that order

#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
  - e.g., IDF

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
    - e.g., “dog” by the 3rd server, “cat” by the 5th server
  - 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