Search Engines

Information Retrieval in Practice

Annotations by Michael L. Nelson

All slides ©Addison Wesley, 2008
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
Fred's Tropical Fish Shop is the best place to find tropical fish at low, low prices. Whether you're looking for a little fish or a big fish, we've got what you need. We even have fake seaweed for your fishtank (and little surfboards too).

**Abstract Model of Ranking**

**Document**
- 9.7 fish
- 4.2 tropical
- 22.1 tropical fish
- 8.2 seaweed
- 4.2 surfboards

**Topical Features**
- 14 incoming links
- 3 days since last update

**Quality Features**
- tropical fish

**Query**

**Ranking Function**

**Document Score**
- 24.5
**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

**Fred's Tropical Fish Shop** is the best place to find tropical fish at low, low prices. Whether you're looking for a little fish or a big fish, we've got what you need. We even have fake seaweed for your fish tank (and little surfboards too).
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*
<table>
<thead>
<tr>
<th>Term</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
</tr>
<tr>
<td>aquarium</td>
<td>3</td>
</tr>
<tr>
<td>are</td>
<td>3</td>
</tr>
<tr>
<td>around</td>
<td>1</td>
</tr>
<tr>
<td>as</td>
<td>2</td>
</tr>
<tr>
<td>both</td>
<td>1</td>
</tr>
<tr>
<td>bright</td>
<td>3</td>
</tr>
<tr>
<td>coloration</td>
<td>3</td>
</tr>
<tr>
<td>derives</td>
<td>4</td>
</tr>
<tr>
<td>due</td>
<td>3</td>
</tr>
<tr>
<td>environments</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>1</td>
</tr>
<tr>
<td>fishkeepers</td>
<td>2</td>
</tr>
<tr>
<td>found</td>
<td>1</td>
</tr>
<tr>
<td>fresh</td>
<td>2</td>
</tr>
<tr>
<td>freshwater</td>
<td>1</td>
</tr>
<tr>
<td>from</td>
<td>4</td>
</tr>
<tr>
<td>generally</td>
<td>4</td>
</tr>
<tr>
<td>in</td>
<td>1</td>
</tr>
<tr>
<td>include</td>
<td>1</td>
</tr>
<tr>
<td>including</td>
<td>1</td>
</tr>
<tr>
<td>iridescence</td>
<td>4</td>
</tr>
<tr>
<td>marine</td>
<td>2</td>
</tr>
<tr>
<td>often</td>
<td>2</td>
</tr>
<tr>
<td>only</td>
<td>2</td>
</tr>
<tr>
<td>pigmented</td>
<td>4</td>
</tr>
<tr>
<td>popular</td>
<td>1</td>
</tr>
<tr>
<td>refer</td>
<td>2</td>
</tr>
<tr>
<td>referred</td>
<td>2</td>
</tr>
<tr>
<td>requiring</td>
<td>2</td>
</tr>
<tr>
<td>salt</td>
<td>1</td>
</tr>
<tr>
<td>saltwater</td>
<td>4</td>
</tr>
<tr>
<td>species</td>
<td>1</td>
</tr>
<tr>
<td>term</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>their</td>
<td>2</td>
</tr>
<tr>
<td>this</td>
<td>3</td>
</tr>
<tr>
<td>those</td>
<td>2</td>
</tr>
<tr>
<td>to</td>
<td>2</td>
</tr>
<tr>
<td>tropical</td>
<td>3</td>
</tr>
<tr>
<td>typically</td>
<td>3</td>
</tr>
<tr>
<td>use</td>
<td>4</td>
</tr>
<tr>
<td>water</td>
<td>1</td>
</tr>
<tr>
<td>while</td>
<td>4</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>world</td>
<td>1</td>
</tr>
</tbody>
</table>
Inverted Index with counts

- supports better ranking algorithms
Inverted Index with positions

- supports proximity matches
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
```

A better example would include where both "tropical" and "fish" occur, but are not adjacent.
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.,

```
title   1:(1,3)        2:(1,5)  4:9,15
extent list
```
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

note: proximity information is lost in a score-ordered list
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:
    
    • 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:
  • which gives:
    \[
    0 \ 101 \ 0 \ 111 \ 0 \ 110 \ 0 \quad \text{13 bits}
    \]
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

  d-gaps

• 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

<table>
<thead>
<tr>
<th>Number</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
</tr>
<tr>
<td>5</td>
<td>111110</td>
</tr>
</tbody>
</table>
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

<# of binary digits needed (in unary)> 0 <binary w/ leftmost 1 truncated>

<table>
<thead>
<tr>
<th>Number $(k)$</th>
<th>$k_d$</th>
<th>$k_r$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>10 0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>10 1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>110 10</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>1110 111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>11110 0000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>11111110 1111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>1111111110 11111111</td>
</tr>
</tbody>
</table>
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 $2^\lceil \log_2 k \rceil + 1$ 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:
  - $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
  - $k_{dr} = (k_d + 1) - 2^{\lceil \log_2(k_d+1) \rceil}$

- Encode $k_{dd}$ in unary, $k_{dr}$ in binary, and $k_r$ in binary

<table>
<thead>
<tr>
<th>Number ($k$)</th>
<th>$k_d$</th>
<th>$k_r$</th>
<th>$k_{dd}$</th>
<th>$k_{dr}$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10 0 1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>10 1 10</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>11 0 0 1 11</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>11 0 0 1 0 0000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>3</td>
<td>0</td>
<td>111 0 000 1 111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>3</td>
<td>2</td>
<td>111 0 010 1 111111111</td>
</tr>
</tbody>
</table>
# 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
• \textit{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

<table>
<thead>
<tr>
<th>$k$</th>
<th>Number of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k &lt; 2^7$</td>
<td>1</td>
</tr>
<tr>
<td>$2^7 \leq k &lt; 2^{14}$</td>
<td>2</td>
</tr>
<tr>
<td>$2^{14} \leq k &lt; 2^{21}$</td>
<td>3</td>
</tr>
<tr>
<td>$2^{21} \leq k &lt; 2^{28}$</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k$</th>
<th>Binary Code</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0000001</td>
<td>81</td>
</tr>
<tr>
<td>6</td>
<td>1 0000110</td>
<td>86</td>
</tr>
<tr>
<td>127</td>
<td>1 1111111</td>
<td>FF</td>
</tr>
<tr>
<td>128</td>
<td>0 00000001 1 0000000</td>
<td>01 80</td>
</tr>
<tr>
<td>130</td>
<td>0 00000001 1 00000010</td>
<td>01 82</td>
</tr>
<tr>
<td>20000</td>
<td>0 00000001 0 0011100 1 0100000</td>
<td>01 1C A0</td>
</tr>
</tbody>
</table>

"1" in high-bit = "here comes the last byte"
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:
  
  tropical = (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

  total space = 13 bytes
  ASCII = 36 bytes
Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be 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
  
  0 1 2 ...
  5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119
  
  — D-gaps
  
  17 34 45 52 ...
  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)

(34, 6) = pos 6 d-gap (0 indexed, so 7th d-gap) = 2; 2+34 = doc 36

looking for doc 80: 52 < 80 < 89, so start at d-gap 12:
52+5 = 57 (miss)
57+23 = 80 (hit)

looking for doc 85: 52 < 80 < 89, so start at d-gap 12:
52+5 = 57 (miss)
52+23 = 80 (miss)
80+9 = 89 (miss); 89 > 85 = doc 85 isn't here
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
Index Construction

- Simple in-memory indexer

```plaintext
procedure BuildIndex(D)

I ← HashTable()

n ← 0

for all documents d ∈ D do

    n ← n + 1
    T ← Parse(d)
    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

greater D is a set of text documents

- Inverted list storage

- Document numbering

- Parse document into tokens
```
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 will make sure Index B doc numbers don't collide with those from Index A.
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

Input

Map

Shuffle

Reduce

Output
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
  
  cf. URI redaction in the IA's Wayback Machine
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
Q = "salt water tropical"
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()`
- `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.
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 $d \in I$ do
    $s_d$ ← 0
    for all inverted lists $l_i$ in $L$ do
      if $l_i$.getCurrentDocument() = $d$ then
        $s_d$ ← $s_d + g_i(Q)f_i(l_i)$  \> 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**

Q = "salt water tropical"

<table>
<thead>
<tr>
<th></th>
<th>1:1</th>
<th>4:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>salt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>partial scores</td>
<td>1:1</td>
<td>4:1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1:1</th>
<th>4:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>old partial scores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>water</td>
<td>1:1</td>
<td>2:1</td>
</tr>
<tr>
<td>new partial scores</td>
<td>1:2</td>
<td>2:1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1:2</th>
<th>2:1</th>
<th>4:2</th>
</tr>
</thead>
<tbody>
<tr>
<td>old partial scores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tropical</td>
<td>1:2</td>
<td>2:2</td>
<td>3:1</td>
</tr>
<tr>
<td>final scores</td>
<td>1:4</td>
<td>2:3</td>
<td>3:1</td>
</tr>
</tbody>
</table>

advantage = efficient disk access to inverted lists
disadvantage = keeping track of partial scores
**Term-At-A-Time**

```plaintext
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 (i.e., Boolean "AND")
    • better for complex feature functions
1: procedure TERMATATIMERETRIEVAL\( (Q, I, f, g, k) \)
2: \( A \leftarrow \text{Map()} \)
3: \( L \leftarrow \text{Array()} \)
4: \( R \leftarrow \text{PriorityQueue}(k) \)
5: \begin{align*}
&\text{for all terms } w_i \text{ in } Q \text{ do} \\
&\quad l_i \leftarrow \text{InvertedList}(w_i, I) \\
&\quad L.\text{add}( l_i ) \\
&\text{end for} \\
&\text{for all lists } l_i \in L \text{ do} \\
&\quad d_0 \leftarrow -1 \\
&\quad \text{while } l_i \text{ is not finished do} \\
&\quad\quad \text{if } i = 0 \text{ then} \\
&\quad\quad\quad d \leftarrow l_i.\text{getCurrentDocument()} \\
&\quad\quad\quad A_d \leftarrow A_d + g_i(Q)f(l_i) \\
&\quad\quad\quad l_i.\text{moveToNextDocument()} \\
&\quad\quad\quad \text{else} \\
&\quad\quad\quad d \leftarrow l_i.\text{getCurrentDocument()} \\
&\quad\quad\quad d' \leftarrow A.\text{getNextAccumulator}(d) \\
&\quad\quad\quad A.\text{removeAccumulatorsBetween}(d_0, d') \\
&\quad\quad\quad \text{if } d = d' \text{ then} \\
&\quad\quad\quad\quad A_d \leftarrow A_d + g_i(Q)f(l_i) \\
&\quad\quad\quad\quad l_i.\text{moveToNextDocument()} \\
&\quad\quad\quad\quad \text{else} \\
&\quad\quad\quad\quad l_i.\text{skipForwardToDocument}(d') \\
&\quad\quad\quad\quad \text{end if} \\
&\quad\quad\quad d_0 \leftarrow d' \\
&\quad\quad\quad \text{end if} \\
&\quad\text{end while} \\
&\text{end for} \\
&\text{for all accumulators } A_d \text{ in } A \text{ do} \\
&\quad s_d \leftarrow A_d \quad \triangleright \text{Accumulator contains the document score} \\
&\quad R.\text{add}( s_d, d ) \\
&\text{end for} \\
&\text{return the top } k \text{ results from } R \\
&\text{end procedure}
1: **procedure** DOCUMENTATATIMERETRIEVAL($Q$, $I$, $f$, $g$, $k$) 
2: \hspace{1em} $L \leftarrow$ Array() 
3: \hspace{1em} $R \leftarrow$ PriorityQueue($k$) 
4: \hspace{1em} for all terms $w_i$ in $Q$ do 
5: \hspace{2em} $l_i \leftarrow$ InvertedList($w_i$, $I$) 
6: \hspace{2em} $L$.add($l_i$) 
7: \hspace{1em} end for 
8: \hspace{1em} $d \leftarrow -1$ 
9: \hspace{1em} while all lists in $L$ are not finished do 
10: \hspace{2em} $s_d \leftarrow 0$ 
11: \hspace{2em} for all inverted lists $l_i$ in $L$ do 
12: \hspace{3em} if $l_i$.getCurrentDocument() > $d$ then 
13: \hspace{4em} $d \leftarrow l_i$.getCurrentDocument() 
14: \hspace{3em} end if 
15: \hspace{2em} end for 
16: \hspace{2em} for all inverted lists $l_i$ in $L$ do 
17: \hspace{3em} $l_i$.skipForwardToDocument($d$) 
18: \hspace{3em} if $l_i$.getCurrentDocument() = $d$ then 
19: \hspace{4em} $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$ \hspace{1em} \text{▷ Update the document score} 
20: \hspace{4em} $l_i$.movePastDocument( $d$ ) 
21: \hspace{3em} else 
22: \hspace{4em} $d \leftarrow -1$ 
23: \hspace{4em} break 
24: \hspace{3em} end if 
25: \hspace{2em} end for 
26: \hspace{2em} if $d > -1$ then $R$.add($s_d$, $d$) 
27: \hspace{2em} end if 
28: \hspace{1em} end while 
29: \hspace{1em} return the top $k$ results from $R$ 
30: **end procedure**
Threshold Methods

• Threshold methods use 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 \(\tau\)
  – optimization methods estimate \(\tau^{'}\) to ignore documents
Threshold Methods

• For document-at-a-time processing, use score of lowest-ranked document so far for $\tau'$
  – 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 $\tau'$
  – safe optimization in that ranking will be the same without optimization
MaxScore Example

- Indexer computes $\mu_{\text{tree}}$
  - maximum score for any document containing just "tree"
- Assume $k = 3$, $\tau'$ is lowest score after first three docs
- Likely that $\tau' > \mu_{\text{tree}}$
  - $\tau'$ is the score of a document that contains both query terms
- Can safely skip over all gray postings

Imagine all boxes are initially white.
We start processing left to right.
After 4th doc, we have 3 "good candidates", so any docs with just "tree" probably won't make the cut.
Gray those docs out and process only the remaining white docs to look for best 3 docs.
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)

#od:1 = proximity of 1
"tropical fish" OR "aquarium fish" or "fish"
(tropical AND fish) OR (aquarium AND fish) OR 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
  IS1's best result might be not as good as IS2's worst result…
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