HPI Hasso Plattner Institut

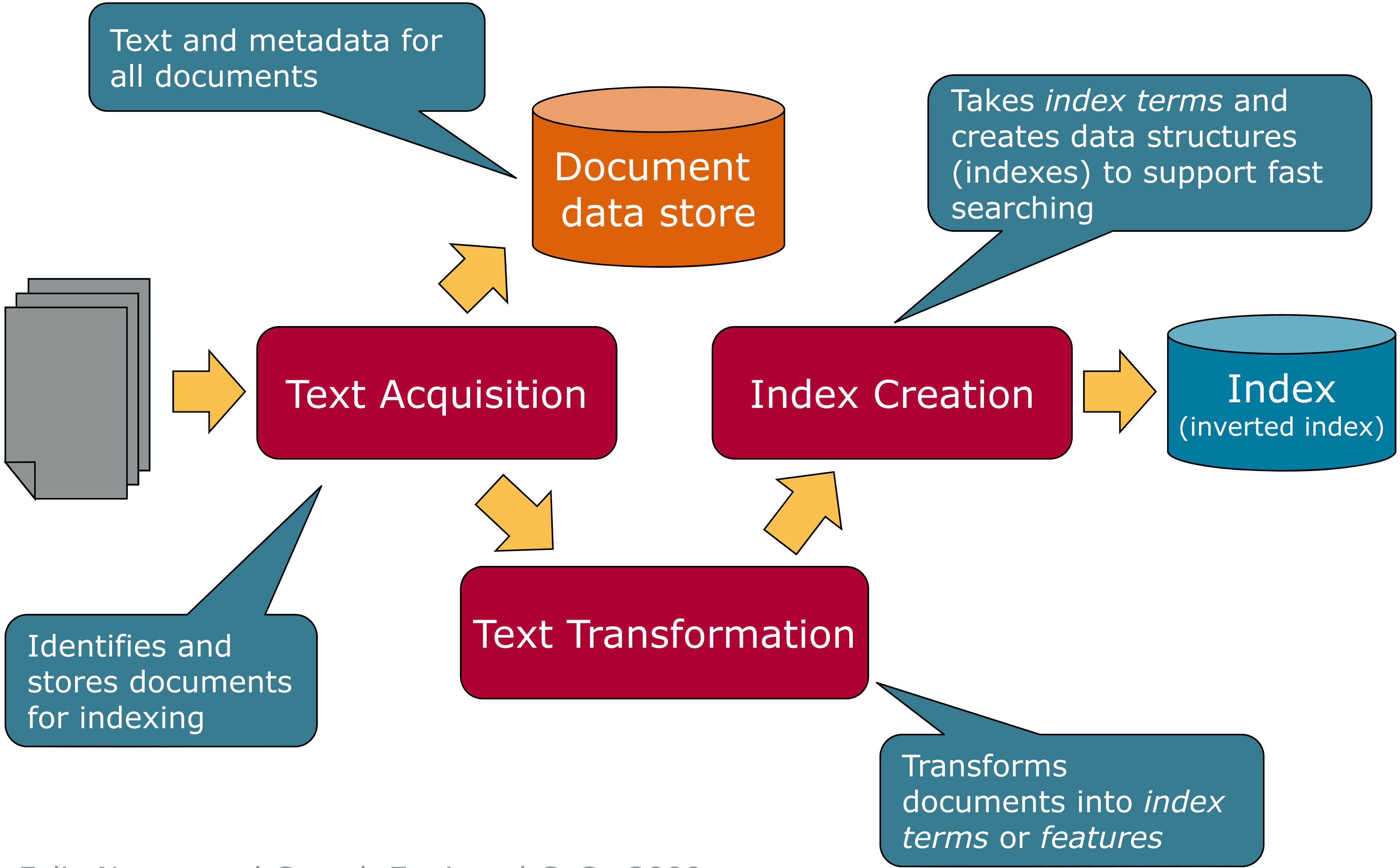
IT Systems Engineering | Universität Potsdam

Search Engines
Chapter 5 – Ranking with Indexes

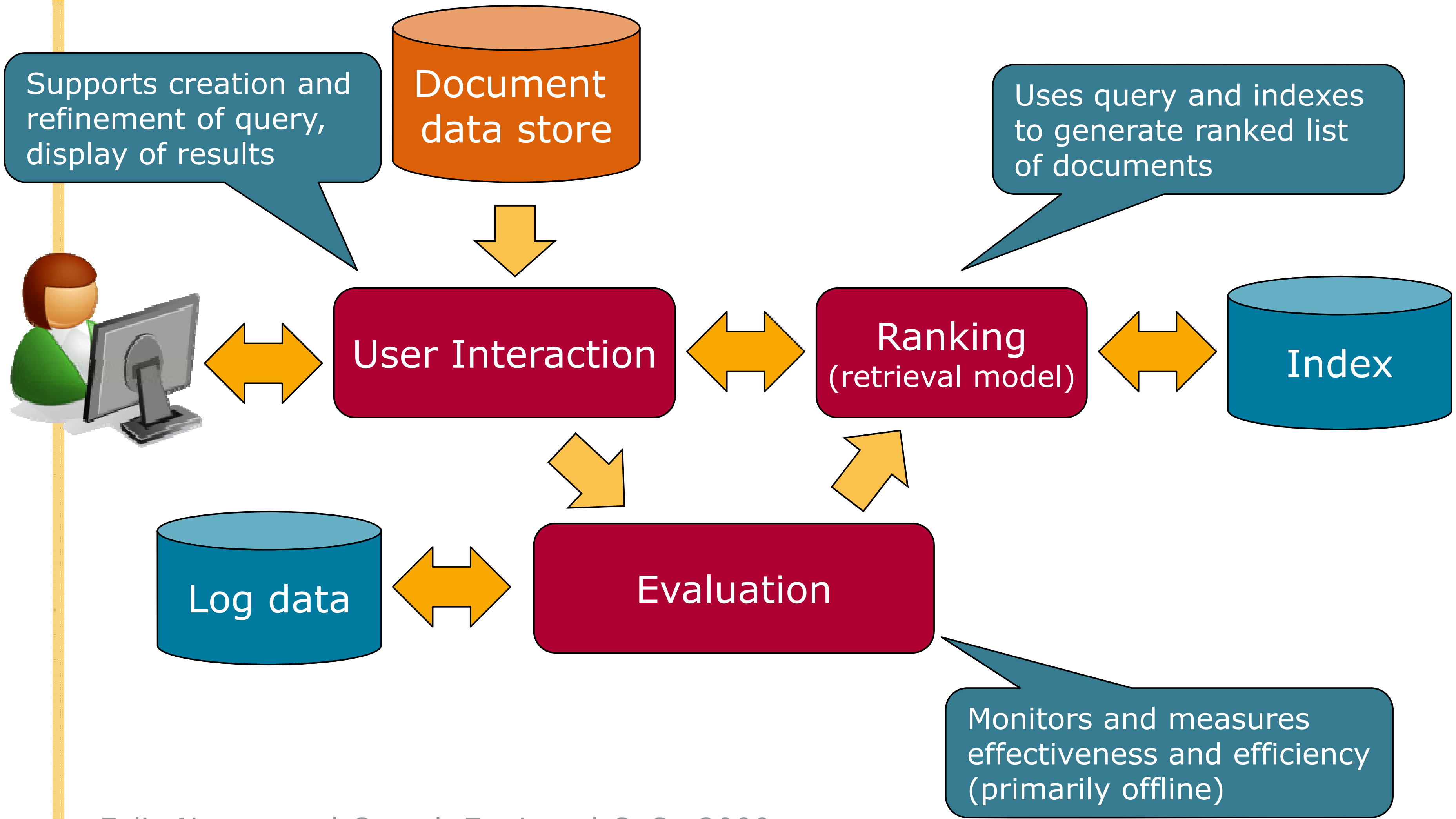
26.5.2009

Felix Naumann

The Indexing Process



The Query Process



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
    - Specialized for different ranking function
  - "Inverted" because documents are associated with words, rather than words with documents
- Components of search engine very dependent
  - Choice of query processing algorithm depends on retrieval model and dictates content of index.

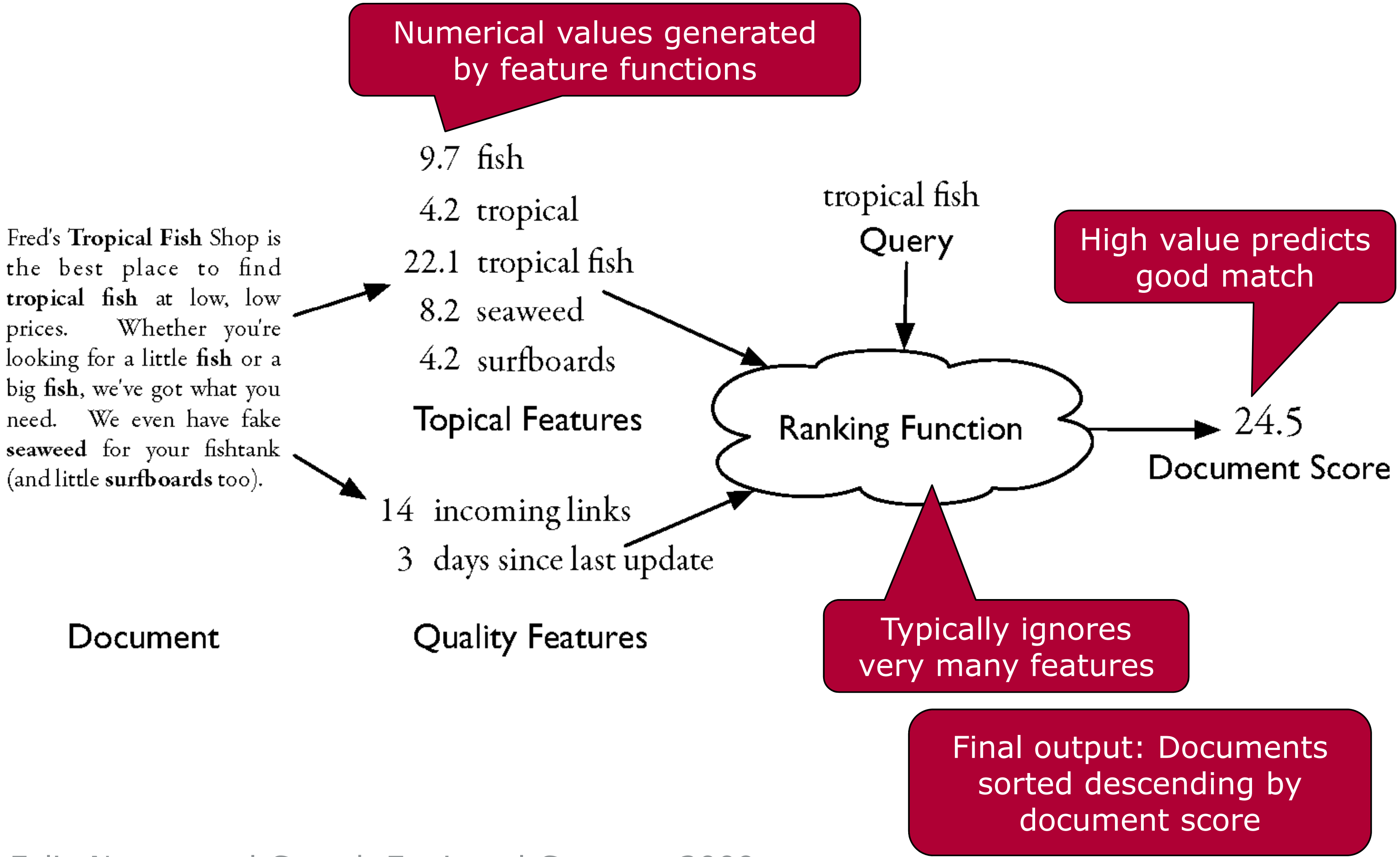
Indexes and Ranking

- 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
    - document representation
    - query
    - *ranking algorithm*
- What is a reasonable abstract model for ranking?
  - Enables discussion of indexes without details of retrieval model (Chapter 7)

Overview

- Abstract model of ranking
- Inverted indexes
- Compression
- Index construction
- Query Processing

Abstract Model of Ranking



More Concrete Model

Hasso Plattner Institut

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

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

Only few; others are zero

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).

Document

$\mathbf{f_i}$		$\mathbf{g_i}$	
9.7	fish	fish	5.2
4.2	tropical	tropical	3.4
22.1	tropical fish	tropical fish	9.9
8.2	seaweed	chichlids	1.2
4.2	surfboards	barbs	0.7
Topical Features		Topical Features	
14	incoming links	incoming links	1.2
3	update count	update count	0.9
Quality Features		Quality Features	

tropical fish
Query

303.01
Document Score

http://www.howard.k12.md.us/res/aquariums/chichlids.html

Overview

- Abstract model of ranking
- Inverted indexes
- Compression
- Index construction
- Query Processing

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).
    - Intersect postings
- Analogy: Book index
  - Inverted indexes usually not alphabetized
  - Hash-table instead

Alternative indexing approaches

- Signature files
  - Each document converted to signature (set of bits)
  - Query also converted to set of bits
  - Query processing: Comparison of bit patterns
    - All signatures must be scanned
    - Comparison is noisy (to keep signature small)
  - Generalization for ranked search difficult
- k-d trees
  - Each document encoded as point in high-dimensional space
  - Same with query
  - Data structure helps find documents closest to query
  - But: Not designed for too many dimensions

Example "Collection"

- Four sentences from the Wikipedia entry for *tropical fish*

- *S1: Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.*
- *S2: Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.*
- *S3: Tropical fish are popular aquarium fish, due to their often bright coloration.*
- *S4: In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.*

Simple Inverted Index

- Each box is a posting.
- Does not record term frequency or occurrence
  - Example: S1 and S2 are treated equally for term “tropical”.
- Intersection
  - Query: “freshwater coloration”
  - {1,4}∩{3,4}
  - Sorted lists: $O(\max(m,n))$
    - Can be improved

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

- Before: Binary information
- Now: Term frequencies
- Supports better ranking algorithms
- Query "tropical fish"
  - S1, S2, S3
  - S2 > S1
  - S2 > S3
- Distinguish main topics and secondary topics in documents

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		

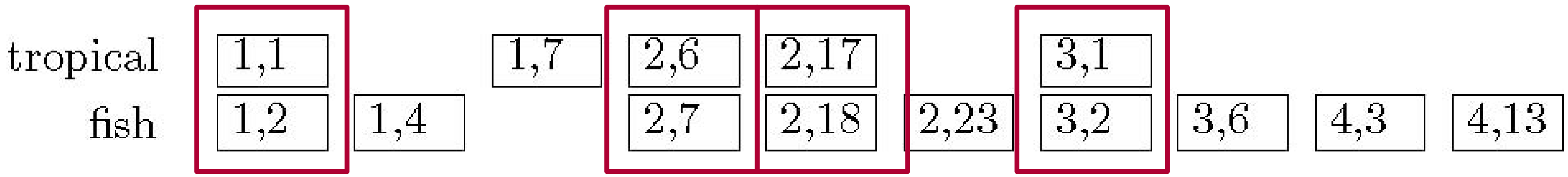
Inverted Index with positions

- Multiple postings per document
  - Each with document number and word position
- Supports proximity matches
- "tropical fish" vs. " 'tropical fish' "

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.



Fields and Extents

- Document structure is useful in search: *document fields*
  - Restrict search to certain fields
    - e.g., date, from:, etc.
  - Some fields more important, even for general search
    - e.g., title, headings
- Options
  - Separate inverted lists for each field type
    - One index for titles, one for headings, one for regular text
    - Problem: General search must read multiple indexes
  - Add information about fields to postings
    - Multiple fields need extensive representation
  - General problem
    - <author>W. Bruce Croft</author>, <author>Donald Metzler</author>, and <author>Trevor Strohman</author>
    - Search for author „Croft Donald"
      - Both are author words; even appear next to each other
- Better: *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
- <author>W. Bruce Croft</author>, <author>Donald Metzler</author>, and <author>Trevor Strohman</author>
  - (1,4)(4,6)(7,9)

- Query: "fish" in title

fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13
title	1:(1,3)		2:(1,5)							4:(9,15)

extent list

Title of document 2 does not contain „fish"

Document 3 has no title

Titel of document 4 starts late and contains „fish"

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
  - Moves complexity from query processing (online) to indexing (offline)
  - Improves speed but reduces flexibility
    - Scoring mechanism cannot be changed
    - Phrase information is lost here
      - But different data structures are possible
- Score-ordered lists (not document-ordered)
  - Only for indexes with precomputed scores
  - 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

Overview

- Abstract model of ranking
- Inverted indexes
- Compression
- Index construction
- Query Processing

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
  - Allows data to move up the memory hierarchy
  - Resuces seek time on disk
- Disadvantage: Decompression time
- Here: *Lossless* compression – no information lost
  - Lossy compression for images, audio, video with very high compression ratios

Compression savings

- Processor can process p inverted list postings per second
- Memory system can supply processor with m postings per second
- Number of postings processed each second: $\min(m, p)$.
  - If $p > m$, the processor will spend some of its time waiting for postings to arrive from memory.
  - If $m > p$, the memory system will sometimes be idle.
- Compression ratio r, decompression factor d
  - Memory supplies rm postings per second
  - Processor processes dp postings per second
  - Number of postings processed each second: $\min(rm, dp)$.
- No compression: $r = d = 1$
- Reasonable: $r > 1$ and $d < 1$
  - Compression useful only if $p > m$
  - Ideal: $rm = dp$

Compression

- *Basic idea*: Common data elements use short codes while uncommon data elements use longer codes
- Inverted lists are lists of numbers
  - Example: coding numbers
    - Number sequence: 0, 1, 0, 3, 0, 2, 0
    - Possible encoding (2 bits): 00 01 00 10 00 11 00
    - Encode 0 using a single 0: 0 01 0 10 0 11 0
    - Only 10 bits, but looks like: 0 01 01 0 0 11 0
    - which encodes: 0, 1, 1, 0, 0, 2, 0
      - Ooops
    - Better: Unambiguous code
      - 0 101 0 111 0 110 0
      - 2-bit encoding was also unambiguous

Number	Code
0	0
1	101
2	110
3	111

Delta Encoding

- Entropy measures predictability of input
- 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
  - Larger documents occur more often in index
  - Not large effect
- Idea: Differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*: Encode 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 (*d-gaps*)
  - 1, 4, 4, 9, 5, 1, 6, 14, 1, 3
  - Advantage: Ordered list of (large) numbers turns into list of small numbers
- Differences for a high-frequency word are easier to compress:
  - 1, 1, 2, 1, 5, 1, 4, 1, 1, 3, ...
- Differences for a low-frequency word are large:
  - 109, 3766, 453, 1867, 992, ...
  - Bad: Large numbers
  - Nice: List is short

Bit-Aligned Codes

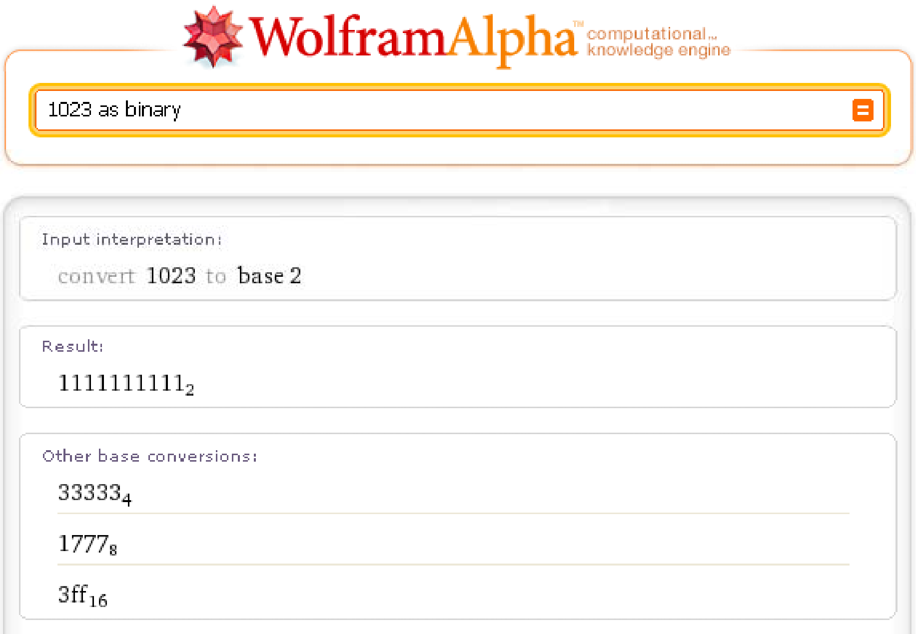
- Breaks between encoded numbers can occur after any bit position
  - Byte-aligned are more favorable to certain operating sytems
- Goal: Small numbers receive small code values
- *Unary* code
  - Encode *k* by *k* 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

- Others: Elias-γ and Elias-δ

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
  - Not useful to encode small numbers



Elias-γ Code

- To encode a number k, compute

$$k_d = \lfloor \log_2 k \rfloor \qquad k_r = k - 2^{\lfloor \log_2 k \rfloor}$$

  - k_d is number of binary digits
  - k_r is k after removing the leftmost 1 of its binary encoding
- Idea: Encode k_d as unary and k_r as binary (in k_d binary digits)
  - Unary part tells us how many binary digits to expect

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
  - $\lfloor\log_2 k\rfloor+1$ for unary part
  - $\lfloor\log_2 k\rfloor$ for binary part
- 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: $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$ $\quad k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$
  - 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

- Sacrifices efficiency for low numbers for smaller encodings of large numbers
  - Numbers larger than 16 require same space as Elias-γ
  - Number larger than 32 require less space

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
- Short codes for small numbers
  - Shortest v-byte code is 1 byte
    - 8 times longer than Elias-γ for number 1
- Numbers are 1 to 4 bytes, with high bit 1 in the last byte, 0 otherwise
- Byte-aligned codes compress and decompress faster

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

High bit of last byte

Compression Example

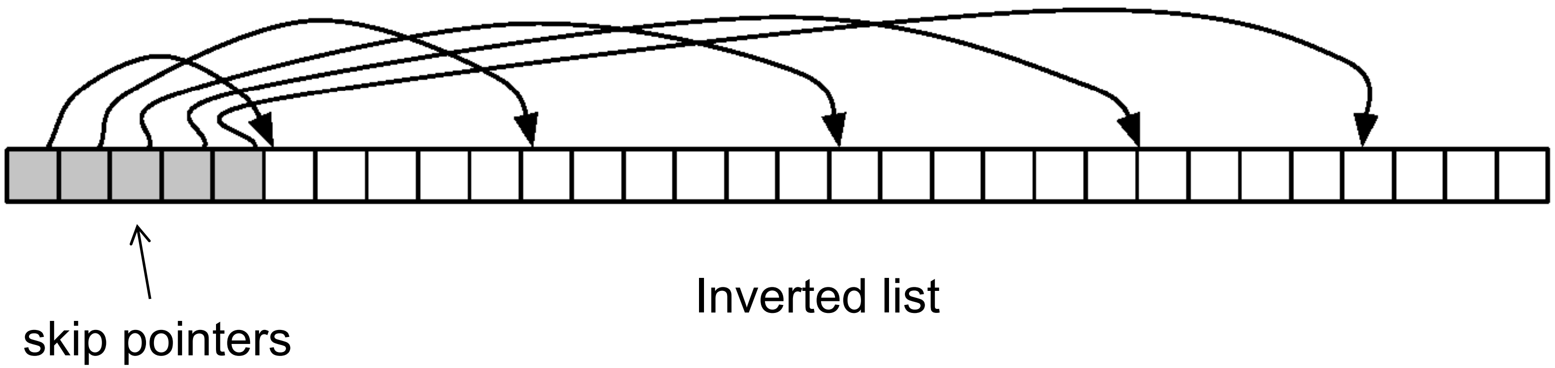
- Original inverted list with positions (docID, position)
  - (1001,1) (1001,7) (1002,6) (1002,17) (1002,197) (1003,1)
- Group positions for each document (docID, count, [positions]):
  - (1001,2,[1,7]) (1002,3,[6,17,197]) (1003,1,[1])
  - Count makes list decipherable even without brackets
    - 1001,2,1,7,1002,3,6,17,197,1003,1,1
- Delta encode document numbers and positions to make numbers even smaller:
  - (1,2,[1,6]) (1,3,[6,11,180]) (1,1,[1])
  - Count cannot be delta-encoded.
- Compress 1,2,1,6,1,3,6,11,180,1,1,1 using v-byte:
  - 81 82 81 86 81 82 86 8B <u>01 B4</u> 81 81 81
  - 13 Bytes for entire list

Skipping

- Search involves comparison of inverted lists of different lengths (intersection)
- Can be very inefficient (for 2-word queries)
  - Like merge join algorithm (two cursors)
  - Reads almost entire lists of both keywords
    - Many millions
- Example: "*animal jaguar*"
  - *animal*: 300 million pages; *jaguar* 1 million pages
  - 99% of the time spent processing the 299 million pages that contain *animal* but not *jaguar*.
- If $d_a < d_j$: Repeatedly skip ahead *k* documents for *animal* until $d_a \geq d_j$
  - Then search linearly
- Determine k using sample queries (100 byte is typical)

Skip Pointers

- Compression makes skipping difficult
  - Variable size, only d-gaps stored
- Skip pointers are additional data structure to support skipping
- 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)
- Decode using skip pointer (34,6)
  - Move to position 6 in d-gaps list (number 2)
  - Add 34 to 2 = document number 36
- Find document number 80
  - Move along skip pointers until (89,15), because 52 > 80 > 89
  - Start decoding at position 12:
    - 52 + 5 = 57
    - 57 + 23 = 80
- Exercise: Find document 85

Auxiliary Structures

- Inverted lists usually stored together in a single file for efficiency.
  - *Inverted file*
  - Single file per index term is space inefficient.
- *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
- Separate system to convert document IDs to URLs, titles, snippets, etc.
  - E.g. BigTable

Overview

- Abstract model of ranking
- Inverted indexes
- Compression
- Index construction
- Query Processing

Index Construction

- Simple in-memory indexer for simple inverted list
  - No positional information, no count information

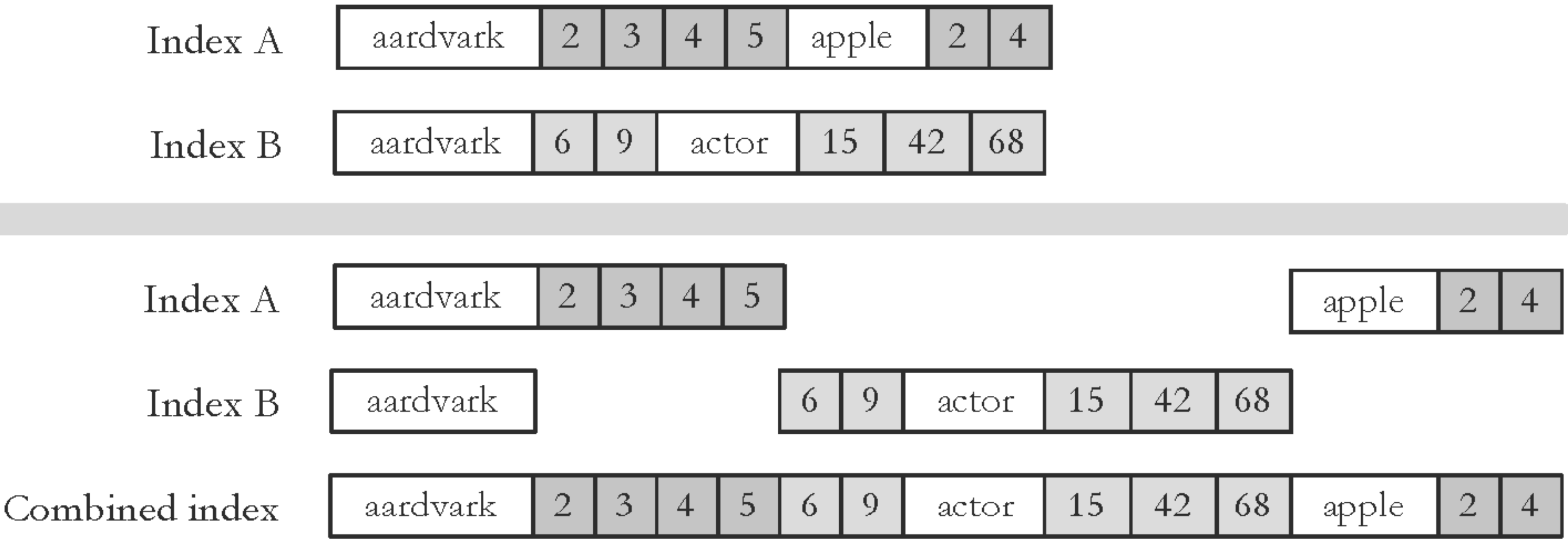
```
procedure BuildIndex(D)                          ▷ D is a set of text documents
    I ← HashTable()                                   ▷ Inverted list storage
    n ← 0                                               ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)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

Two problems
- RAM-based
- Sequential execution

Merging

- Merging addresses limited memory problem
    1. Build the inverted list structure until memory runs out.
    2. Then write the partial index to disk, start making a new one.
    3. 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 easily merged in small pieces
    - By definition, no two partial indexes can be in memory simultaneously.
    - Solution: Store in alphabetical order

Merging



- Can be generalized to merge many partial lists at once
- Documents may have to be renumbered.
- Minimum space requirement:
  two words, one posting, some file pointers
  - In practice: Large chunks in memory

Distributed Indexing

- Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)
  - Fast and increasing growth of Web
  - Not just search engines but also applications that analyze the Web.
- Large numbers of inexpensive servers used rather than larger, more expensive machines
  - Smaller machines are sold more often
  - Large machines do not develop economy of scale
  - Disadvantages
    - Small servers fail more often
    - Among many servers, the likelihood that one fails increases.
    - Difficult to program: Programmers trained for single-threaded applications, not for multi-threaded, multiprocessor, networked applications.
      - Some help: RPC, CORBA, Java RMI, SOAP, Hadoop

Data Placement – Example

- Key problem: Place data efficiently among multiple servers / disks
- 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.
  - Task: Determine the sum of transactions for each unique credit card number.
- Could use hash table – hash the credit card number
  - But: Memory problems
- Same task, but file is sorted by credit card numbers
  - Aggregating is simple with sorted file
- Similar with distributed approach
  - Distribute small (random) batches – but how to combine?
  - Thus: Careful distribution, so that all transactions of one card end up in same batch: Sorting
  - Sorting and placement are crucial

MapReduce

- *MapReduce* is a distributed programming framework/paradigm/tool designed for indexing and analysis tasks
  - Focus on data placement and distribution
- Functional languages
  - *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 for MapReduce not so strict in terms of number of outputs
- Many mapper and reducer tasks on a cluster of machines

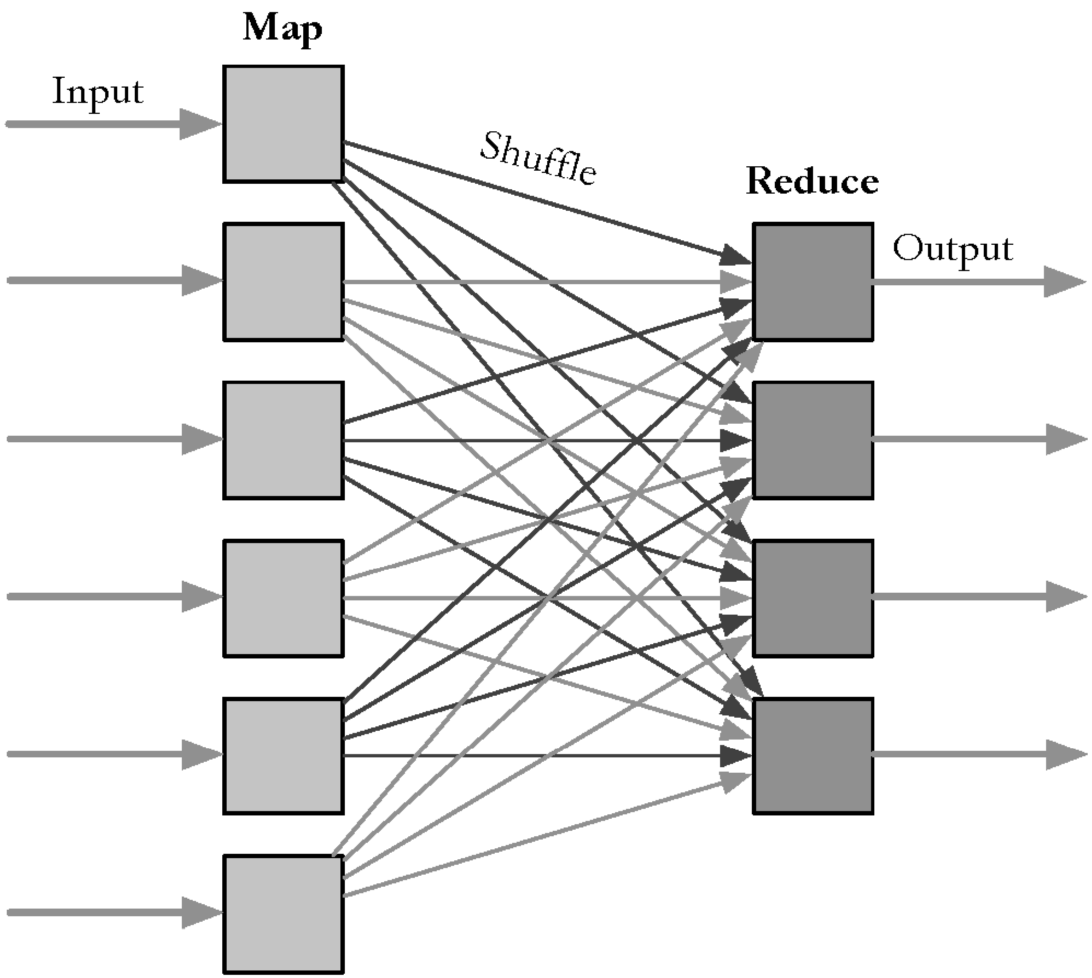
MapReduce algorithms on Hadoop

- http://www.hpi.uni-potsdam.de/naumann/lehre/ss_09/mapreduce_algorithms_on_hadoop.html

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
    - Not implemented by developer
  - *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
  - In case of hardware failure, that set of tasks is performed again (on a different machine)
- Backup processes replicate results of slowest machines

MapReduce



Credit Card 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, document:position)
            position = position + 1
        end for
    end while
end procedure
```

Chapter 4

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

e.g. compression

Updates: Result Merging

- Collections of text grow and change
- *Index merging* is a good strategy for handling updates when they come in large batches
  - Inefficient for small updates: Entire index must be written to disk each time.
- *Result merging* for small updates: Create separate index for new documents, merge *results* from both searches
  - Separate index in memory, thus fast to update and search
- Deletions handled using *delete list*
  - Before showing result, search engine verifies that no result element is on delete list.
- Modifications done by insert and delete
  - Put old version on delete list
  - Add new version to new documents index

Overview

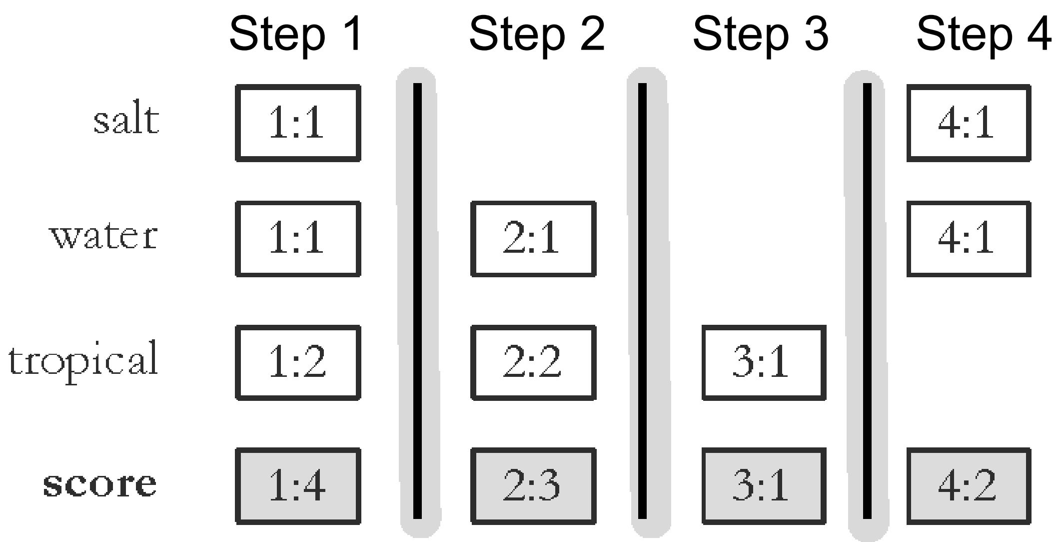
- Abstract model of ranking
- Inverted indexes
- Compression
- Index construction
- Query Processing

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

- Query: *salt water tropical*
- Inverted list with word counts
- Score: Sum of word counts
- One step per document



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
```

Q Query
I Index
f, *g* sets of feature functions
k number of documents to retrieve

$s_D \leftarrow 0$

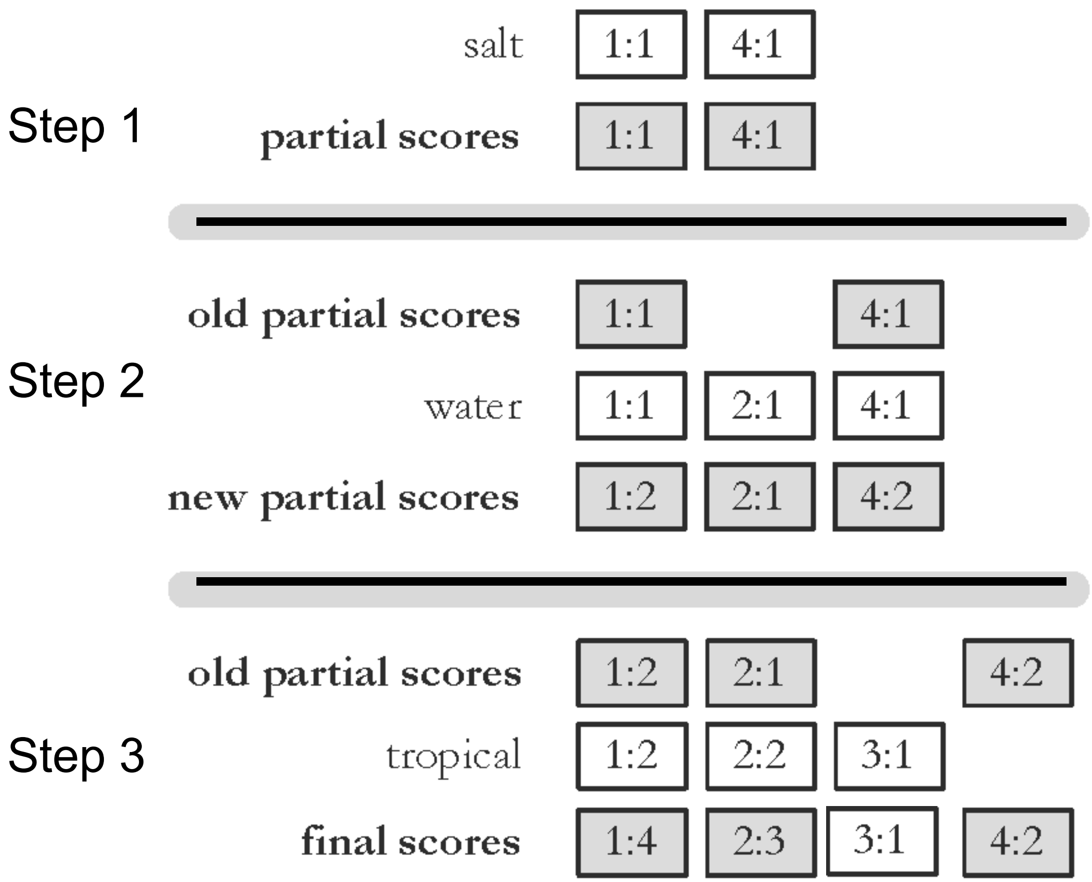
Should be restricted to documents that appear at least in one list

Move cursor (lists are sorted by document number

Should hold only *k* documents

Term-At-A-Time

- Query: *salt water tropical*
- Accumulators accumulate scores for each document
- One step per query term



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
```

New!

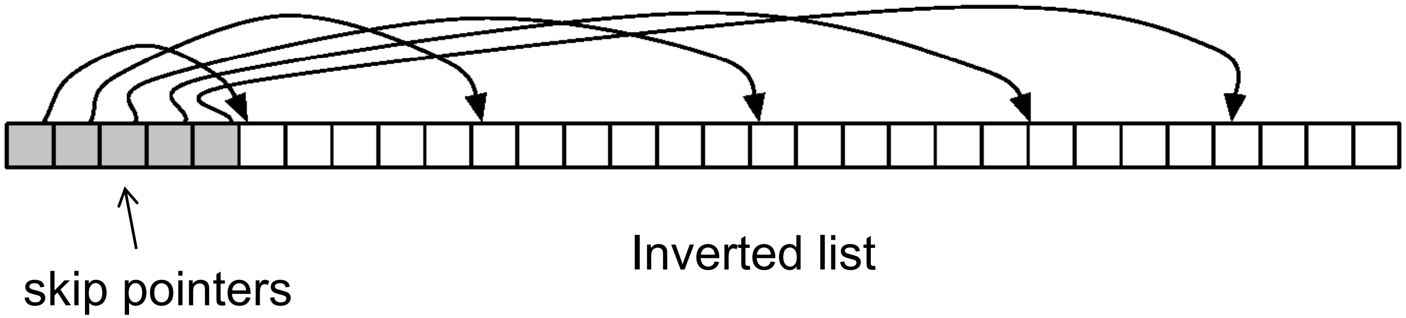
High memory load

Advantage: Less disk seeking (each list is read only once)

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

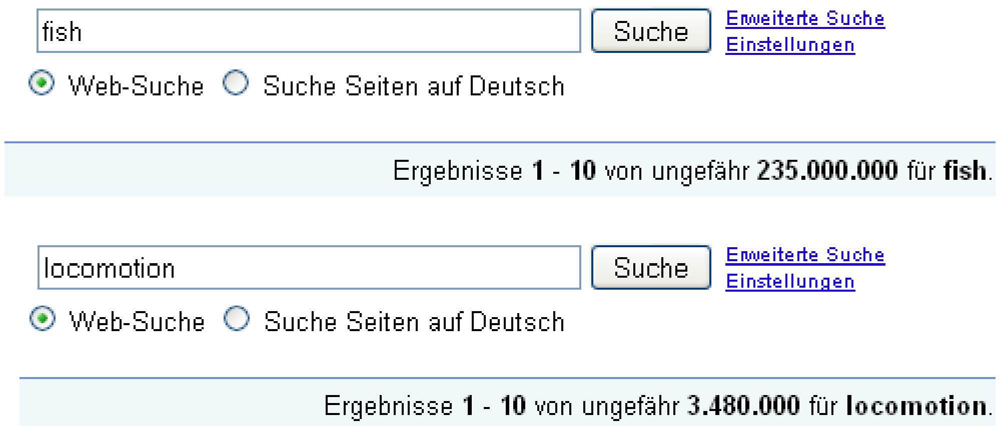
List skipping: Read less data from inverted lists



- n bytes in list, skip pointers after each c bytes, pointer are k long
- Read entire list: $O(n)$
- Jumping through list: $O(kn/c) = O(n)$
  - But: If $c = 100$ and $k = 4$ we read just 2.5% of total data.
- c should not be arbitrarily large: Need to find p postings
  - n/c intervals; posting is halfway into interval: $pc/2$
  - Total: $kn/c + pc/2$
    - Assuming $p << n/c$ (otherwise multiple postings within interval)
  - Find optimal c using previous queries
- In reality c > 100.000 to observe any improvement
  - Disks perform poorly at jumping to arbitrary positions
- And: Skipping reduces decompression load

Conjunctive processing: Calculate scores for fewer documents

- All query terms need to be present in result documents
  - Default for most search engines
  - Not usful for very long queries (plagiarism)
- Optimizes performance and effectiveness
- Especially helpful with query terms of different frequency



- Can be used for term-at-a-time and document-at-a-time

Conjunctive 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
            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.getNextDocumentAfter(d)
                l_i.skipForwardTo(d)
                if l_i.getCurrentDocument() = d then
                    A_d ← A_d + g_i(Q)f(l_i)
                else
                    A.remove(d)
                end if
            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
```

Skip ahead using accumulator table

Runs best if lists are sorted by size

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
```

Get largest document currently pointed to.
Not guaranteed to contain all terms, but good candidate

Try to skip each list to that document.
If fails, use next largest document.

Runs best if lists are sorted by size

Threshold Methods

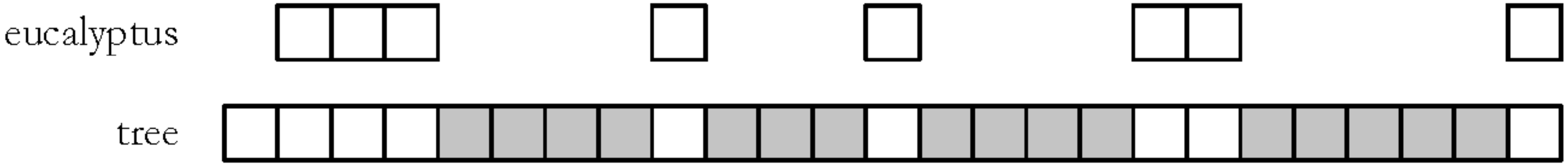
- Threshold methods use limit of top-ranked documents needed (*k*) to optimize query processing
  - For most applications, *k* is small

Ergebnisse **1** - **10** von ungefähr **235.000.000** für **fish**.

- 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
  - But: Yet unknown
- Optimization methods estimate τ' to ignore documents
  - $\tau' \leq \tau$ for safety
  - For document-at-a-time processing, use score of lowest-ranked document in list of top-*k* documents so far for τ'
  - For term-at-a-time, have to use k_{th}-largest score in the accumulator table

Threshold Methods – MaxScore

- *MaxScore* method compares the maximum score that remaining documents could have to τ'.
  - τ' is *lower bound.*
  - *Safe* optimization: Ranking will be same without optimization



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

Early termination of query processing

- Term-at-a-time
  - Ignore high-frequency word lists in
    - Similar to stop word lists
  - Ignore all terms above some constant
    - For queries with very many terms
    - Later terms only change the ranking slightly
- Document-at-a-time
  - Ignore documents at end of lists
  - Works well only if documents are sorted by quality
- In general, early termination is an *unsafe* optimization
  - But: “To be or not to be” is immune to other optimizations, because it has very long index lists.
  - Thus: Early termination is only choice

List ordering

- In general: Document IDs are assigned randomly to web pages
  - Best documents can be at end of lists
  - Assignment is unused degree of freedom
- Order inverted lists by quality metric (e.g., PageRank) or by partial score
  - Metric independent of query
  - Can compute upper bounds more easily
- Order inverted lists by partial score
  - As for one-word queries
  - Works well for term-at-a-time, but read only partial lists until satisfied.
- Makes unsafe (and fast) optimizations more likely to produce good documents

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
    - Much network traffic

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
- Disk seek time for k terms and n index servers
  - Document distribution: $O(kn)$
  - Term distribution: $O(k)$

Caching

- Insight: Query distributions similar to Zipf
  - About ½ of queries 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
  - And not only one-word queries
- Cache must be refreshed to prevent stale data