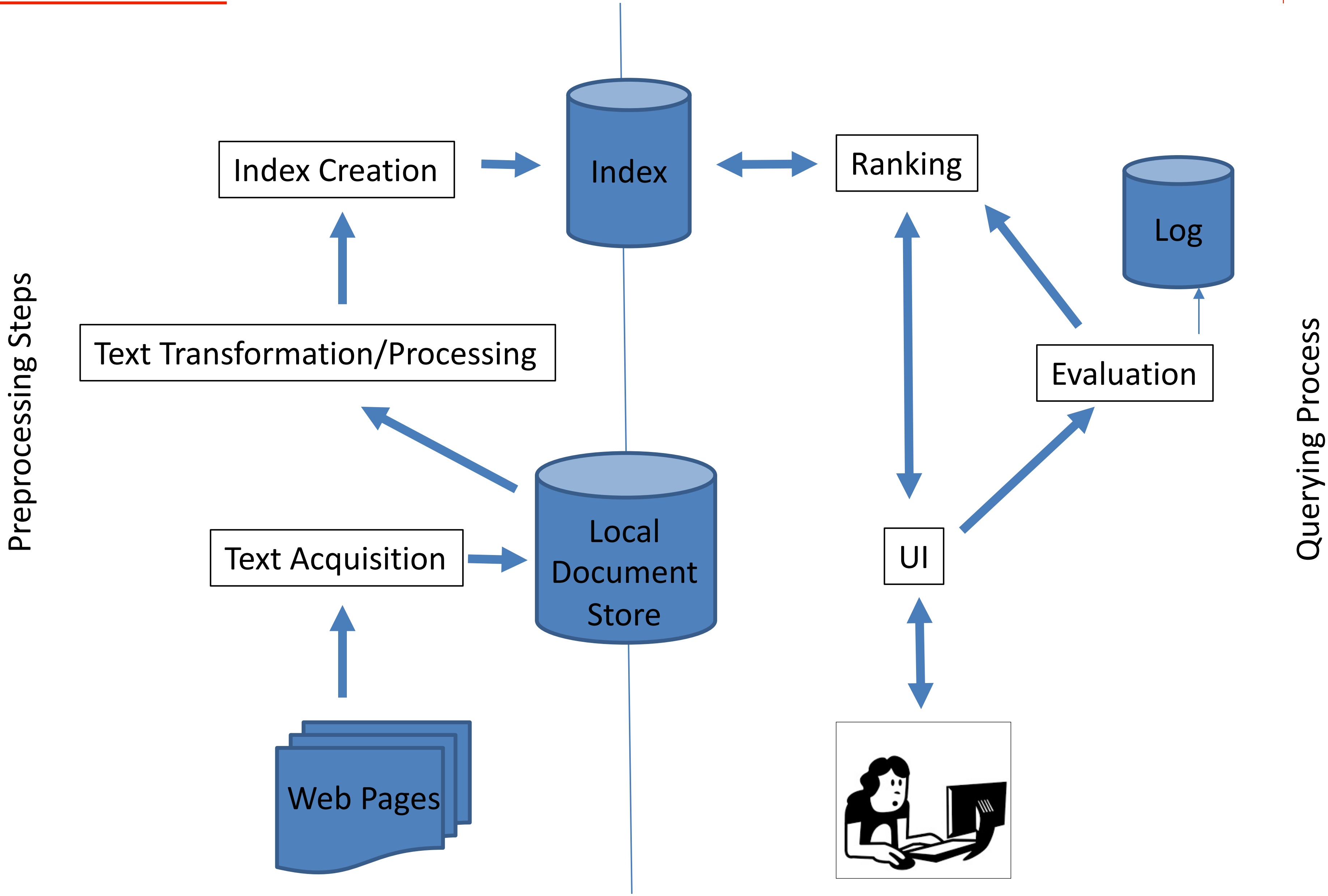
Informatics 225
Computer Science 221

Information Retrieval

Lecture 16

These course materials borrow, with permission, from those of Prof. Cristina Videira Lopes, Addison Wesley 2008, Chris Manning, Pandu Nayak, Hinrich Schütze, Heike Adel, Sascha Rothe, Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie. Powerpoint theme by Prof. André van der Hoek.

Architecture



Indexes

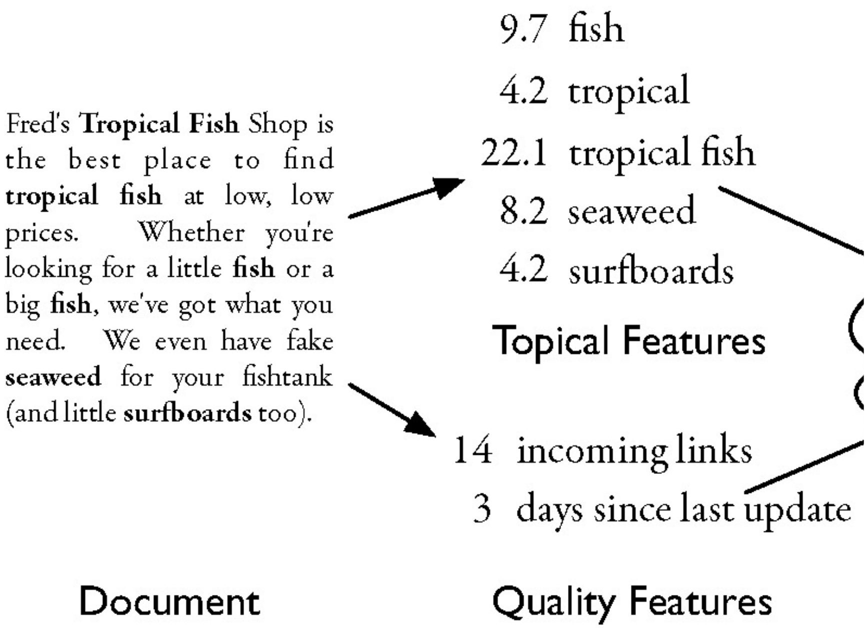
Information Retrieval

Abstract Model of Ranking

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).

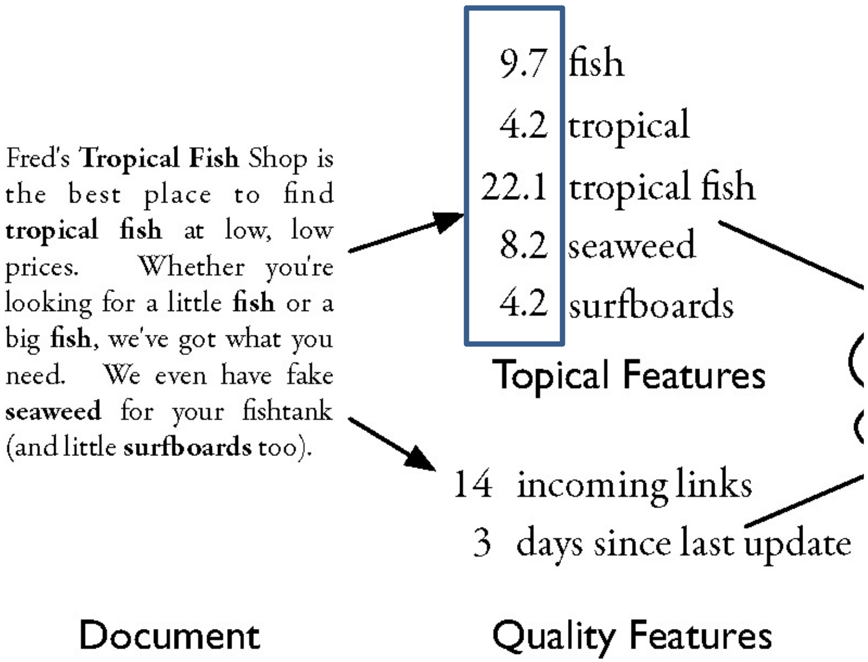
Document

Abstract Model of Ranking

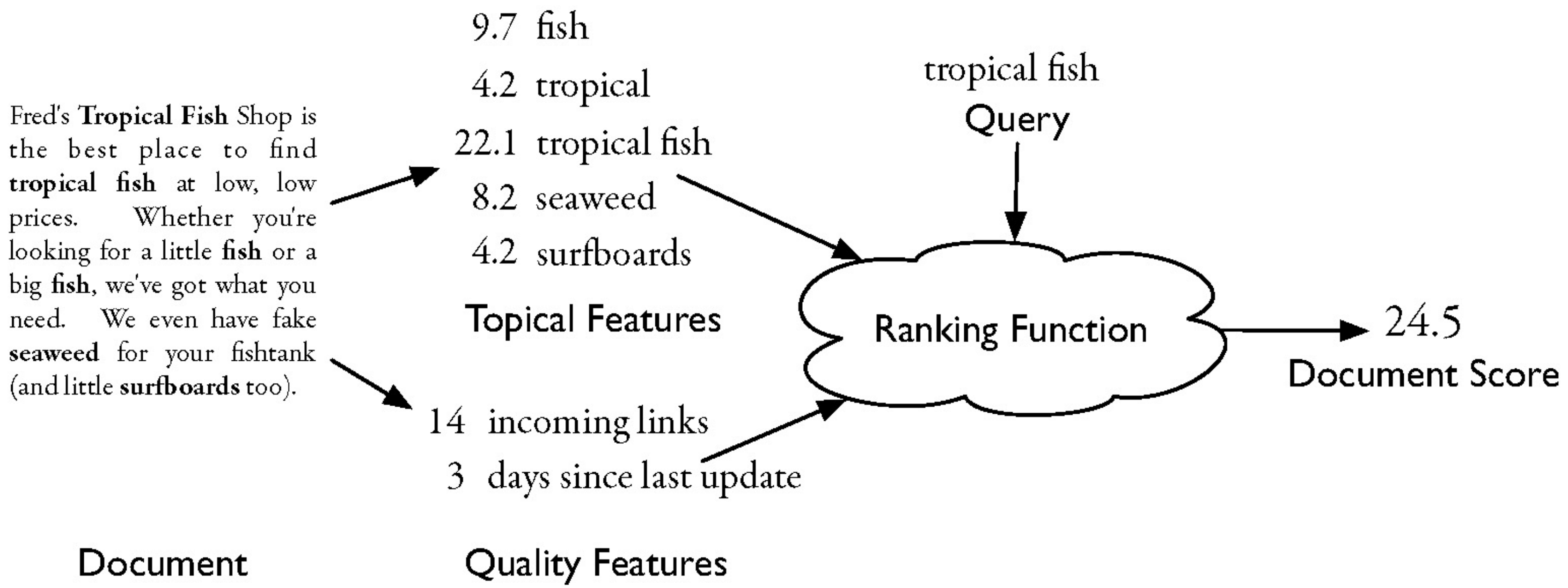


Abstract Model of Ranking

Feature function : f(text) -> R



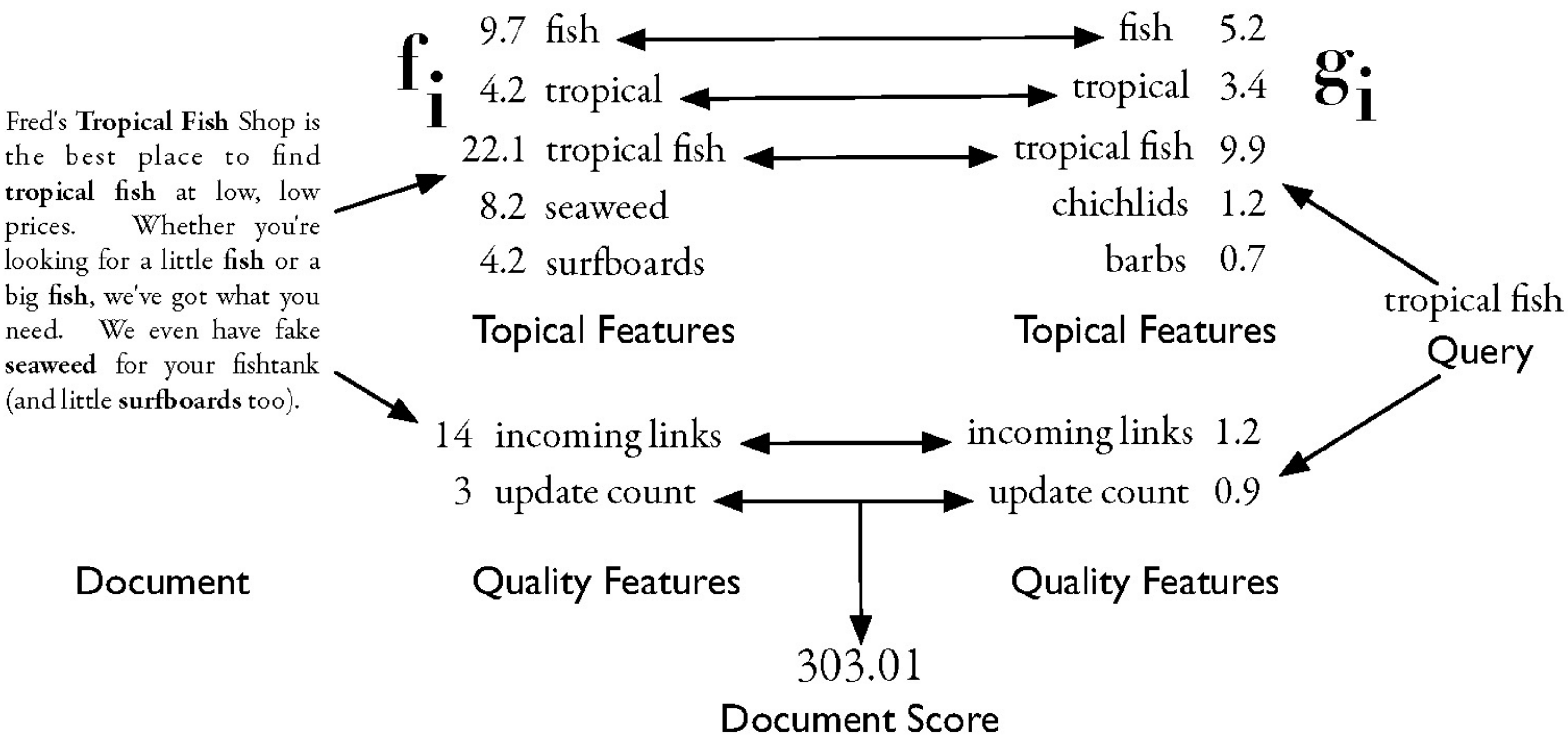
Abstract Model of Ranking



A 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
i runs over the features



Simple Inverted Index

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.

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

Inverted Index with counts

- supports better ranking algorithms

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.

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

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.

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

Term	Postings
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”

Proximity Matches

- Matching phrases or words within a window
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- An inverted index with posting lists that store positions makes these types of query features efficient
  - e.g., postings: [docID, word pos]

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.
  - Words in some fields are more important
    - e.g., title!

Fields and Extents

- Document structure is useful in search
  - *field* restrictions
    - e.g., date, from:, etc.
  - Words in some fields are more important
    - e.g., title!

- Options to capture this in the index:
  - separate inverted indexes 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.,

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

Other possibilities

- Store 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

Other possibilities

- Store 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
- Create score-ordered lists
  - *the 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 Construction

Information Retrieval

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 ← List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

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 ← List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

Index Construction

- Simple in-memory indexer

procedure BUILDINDEX(D) ▷ D is a set of text documents
    $I \leftarrow$ HashTable() ▷ Inverted list storage
    $n \leftarrow 0$ ▷ Document numbering
    for all documents $d \in D$ **do**
        $n \leftarrow n + 1$
        $T \leftarrow$ Parse(d) ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens $t \in T$ **do**
            if $I_t \notin I$ **then**
                $I_t \leftarrow$ List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure

Index Construction: Doc Id

- If doc ids are URLs:
  - https://www.ics.uci.edu/~aburtsev/238P/lectures/lecture03-calling-conventions
    - That’s 77 characters…
    - 77 bytes in C++
    - 126 bytes in Python3
    - 352 bytes in Java

- Remember, they are used in postings lists
  - Depending on the structure of the posting, they can even appear multiple times in the same posting list
  - Strings: Very wasteful to store and to compare

Index Construction: Doc Id

- Map URLs to integers as you process docs:
  - 0 → https://www.ics.uci.edu/~aburtsev/238P/lectures/lecture03-calling-conventions
  - 1 → https://www.ics.uci.edu/~goodrich/teach/ics247/notes
  - 2 → https://www.ics.uci.edu/~thornton/ics184/MidtermSolutions.html
  - Etc.

- Size of integers:
  - 4 bytes in C, C++ and Java
  - 28 bytes in Python
  - Almost always the same size independent of the string
    - Python converts integers to longs when they overflow; the maximum depends on the interpreter's word size: can be 2^(31)-1 or 2^(63)-1.

Index Construction: Doc Id

- Estimate postings size for Python3:
  - 10,000 index terms
  - Avg. 20 postings per term
  - Avg. URL: 51 characters (= 100 bytes)

Index Construction: Doc Id

- Estimate postings size for Python3:
  - 10,000 index terms
  - Avg. 20 postings per term
  - Avg. URL: 51 characters (~ 100 bytes)

- Doc ids as URLs:
  - 10,000 x ~20 x ~100 ~ 20,000,000 ~ 20M

Index Construction: Doc Id

- Estimate postings size for Python3:
  - 10,000 index terms
  - Avg. 20 postings per term
  - Avg. URL: 51 characters (~ 100 bytes)

- Doc ids as URLs:
  - 10,000 x ~20 x ~100 ~ 20,000,000 ~ 20M

- Doc ids as integers:
  - 10,000 x ~20 x ~28 ~ 5,600,000 ~ 5.6M
  - Reduced to 28% of string version
  - Can fit ~4x more postings in memory
  - It is significantly faster to sort

Index Construction: Doc Id

- Map URLs to integers as you process docs:
  - 0 → https://www.ics.uci.edu/~aburtsev/238P/lectures/lecture03-calling-conventions
  - 1 → https://www.ics.uci.edu/~goodrich/teach/ics247/notes
  - 2 → https://www.ics.uci.edu/~thornton/ics184/MidtermSolutions.html
  - Etc.

- Must keep that mapping stored somewhere
  - You will need it for showing the search results
  - Typically not part of inverted index itself, but auxiliary bookkeeping structure (serialized to a file)
    - *More on auxiliary structures later*

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 ← List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

Index Construction: Postings

- Contain context of term occurrence in a document
  - Doc id
  - Frequency count or TF-IDF
  - Fields
  - Positions
  - ...

Required for Project 3

Index Construction: Postings

- Contain context of term occurrence in a document
  - Doc id
  - Frequency count or TF-IDF
  - Fields
  - Positions
  - …

Required for Project 3

```
class Posting:
  def __init__(self, docid, tfidf, fields):
    self.docid = docid
    self.tfidf = tfidf  # use freq counts for now
    self.fields = fields
```

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 ← List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

Index Construction

- Simple in-memory indexer

$$
\begin{array}{ll}
\textbf{procedure } \textsc{BuildIndex}(D) & \triangleright\ D \text{ is a set of text documents} \\
\quad I \leftarrow \text{HashTable}() & \triangleright \text{ Inverted list storage} \\
\quad n \leftarrow 0 & \triangleright \text{ Document numbering} \\
\quad \textbf{for all } \text{documents } d \in D \textbf{ do} & \\
\qquad n \leftarrow n + 1 & \\
\qquad T \leftarrow \text{Parse}(d) & \triangleright \text{ Parse document into tokens} \\
\qquad \text{Remove duplicates from } T & \\
\qquad \textbf{for all } \text{tokens } t \in T \textbf{ do} & \\
\qquad\quad \textbf{if } I_t \notin I \textbf{ then} & \\
\qquad\qquad I_t \leftarrow \text{List<Posting>()} & \\
\qquad\quad \textbf{end if} & \\
\qquad\quad I_t\text{.append(Posting(n))} & \\
\qquad \textbf{end for} & \\
\quad \textbf{end for} & \\
\quad \textbf{return } I & \\
\textbf{end procedure} &
\end{array}
$$

What happens if you run out of memory?

Scaling index construction

- In-memory index construction does not scale
  - Can’t fit entire collection into memory

Scaling index construction

- In-memory index construction does not scale
  - **Can’t fit entire collection into memory**

- How can we construct an index for very large collections?

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 ←  List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

Could this be a file, directly?

Scaling index construction

- In-memory index construction does not scale
  - Can’t fit entire collection into memory

- How can we construct an index for very large collections?

- Taking into account hardware constraints. . .
  - Memory, disk, speed, etc.

Hardware basics

- Some servers used in IR systems typically have several GB of main memory, sometimes hundreds of GB.

Hardware basics

- Some servers used in IR systems typically have several GB of main memory, sometimes hundreds of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- But... Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.
  - Regular machines ➔ Much smaller RAM

Hardware basics

- Large scale storage is based on spinning disks
- Access to data in memory is ***much*** faster than access to data on disk.

Hardware basics

- Access to data in memory is ***much*** faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
  – Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.

Hardware basics

- Access to data in memory is ***much*** faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
  - Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
  - Block sizes: 8KB to 256 KB.

Jeff Dean's (*)
"Latency Numbers Every Programmer Should Know"

- Latency Comparison Numbers (updated 2020)

L1 cache reference	0.5-1.5 ns	
L2 cache reference	5-7 ns	
L3 cache reference	16-25 ns	
Mutex lock/unlock	25 ns	
64MB Main memory reference	50-75 ns	
Send 4KB over 100 Gbps HPC fabric	1,040 ns	
Compress 1K bytes with Zippy	2,000 ns	2 us
Read 1 MB sequentially from memory	3,000 ns	3 us
Send 4KB over 10 Gbps ethernet	10,000 ns	10 us
Read 1 MB sequentially from SSD	49,000 ns	49 us
Read 1 MB sequentially from HDD	825,000 ns	825 us
Disk seek (up to)	2,000,000 ns	2,000 us
Send packet CA->Netherlands->CA	150,000,000 ns	150,000 us

(*) https://ai.google/research/people/jeff/
Original: http://norvig.com/21-days.html#answers
2019 : https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646
2020: https://colin-scott.github.io/personal_website/research/interactive_latency.html

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 ← List<Posting>()
            end if
            I_t.append(Posting(n))
        end for
    end for
    return I
end procedure
```

Could this be a file, directly?

NO.

ON ASSIGNMENT 3

Project 3: Indexer MS1

- Milestone #1: Implement a **simple** Indexer
  - **Start with a small set of files**. Short development cycle!
    - Traversing folders and reading JSON
    - Opening and reading one file at a time
    - Parsing (dealing with broken HTML!)
    - Tokenization & stemming
    - Simple **in-memory** inverted index
    - Simple index serialization to disk
  - Expand gradually!!
  - No need to scale up yet...

Project 3: Boolean search MS2

- Milestone #2: Implement simple Boolean retrieval – AND only
  - E.g.
    - cristina lopes means AND
    - eppstein Wikipedia means AND
    - master of software engineering means AND
  - Required: text interface.
    - **Bonus** points for a web GUI in MS3
  - No speed restrictions yet…

Project 3: Ranked search MS3

- Milestone #3: Implement ranked retrieval
  - Scale up
  - **Completely different approach,** will be covered in future
  - **May** benefit from positional information
  - **Must perform under 300ms,** ideally ~100ms but no penalty for anything < 300ms. **We will see in the future who to achieve this.**
    - **Add a timer to your code to print the time between the arrival of the query and the production of the result by your search engine** (no need to consider the rendering on the part of the user if you create a web GUI).