Fundamentals of Information Retrieval

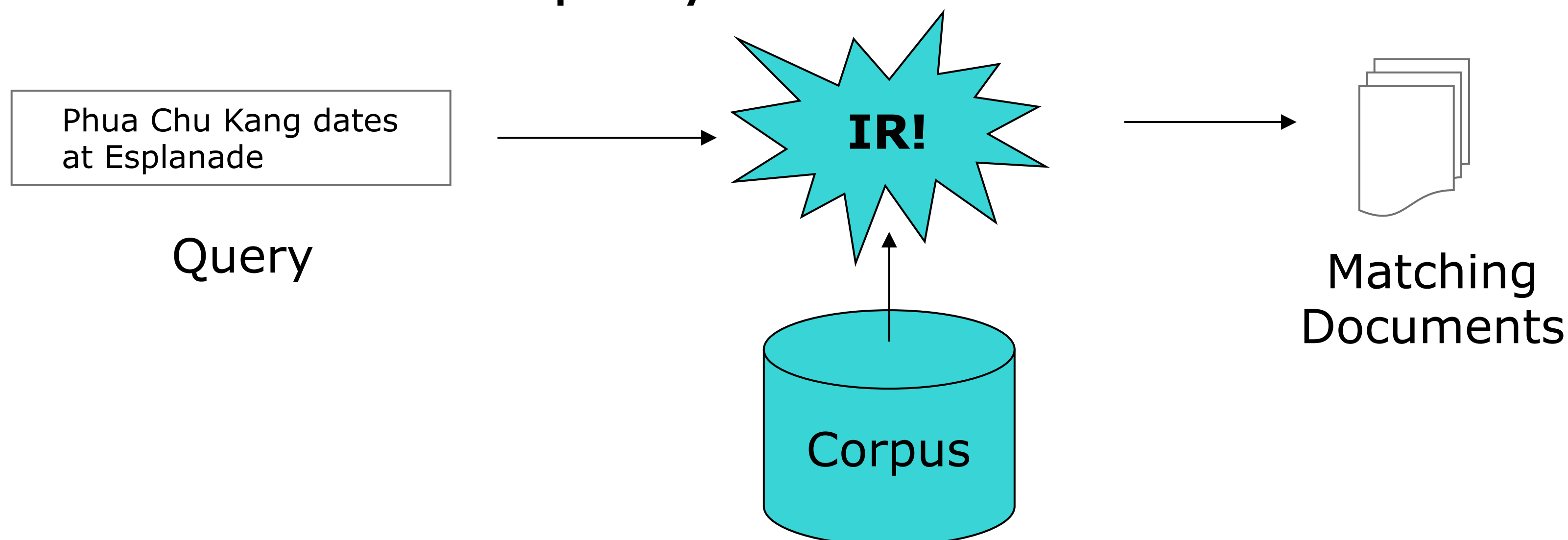
What is information retrieval?

Midterm questions for Digital Libraries | Search

Phua Chu Kang dates at Esplanade | Search

What is information retrieval?

- Part of the information seeking process
- Matches a query with most relevant documents
- View a query as a mini-document



- Table of Contents
- Index
- `grep`

- Procedure:
  - Look up topic
  - Find the page
  - Skim page to find topic

…
Index, 11, 103-151, 443
  Audio, 476
  Comparison of methods 143-145
  Granularity, 105, 112
  N-gram, 170-172
  Of integer sequences, 11
  Of musical themes, 11
  Of this book, 103, 507*ff*
  Within inverted file entry, *see* skipping
Index compression, 114-129, 198-201, 235-237
  Batched, 125,128
  Bernoulli, 119-122, 128, 150, 247, 421
  Context-sensitive, 125-126
  Global, 115-121
  Hyperbolic model, 123-124, 150
  In MG, 421-423
  Interpolative coding, 126-128
  Local, 115, 121-122, 247
  Nonparameterized, 115-119
  Observed frequency, 121, 124-125, 128, 247
  Parameterized, 115
Performance of, 128-129. 421
Skewed Bernoulli, 122-123, 138, 150
Within-document frequencies, 198-201
Index Construction, 223-261 (*see also* inversion)
  bitmaps, 255-256
…

Partial index of *Managing Gigabytes*

- Algorithm
  - (Permute query to fit index)
  - Search index
  - Go to resource
  - (Permute query to fit item)
  - (Search for item)

- Books indices have key words and phrases
- Search engines index (all) words

Why the disparity?

What do people really search for?

> What is a **word**?
>
> - Maximal sequence of alphanumeric characters
> - Limited to at most 256 characters and at most 4 numeric characters.
>
> \- MG indexing system

Trading precision for size

Can save up to **32%** without too much loss:

- Stemming
  - Usually just word inflection
  - Information → Inform = Informal, Informed
- Case folding
  - **N.B.**: keep odd variants (e.g., NeXT, LaTeX)
- Stop words
  - Don't index common words, people won't search on them anyways

Pop Quiz: Which of these techniques are more effective?

Indexing output

- Output = $L_w, D_D, I_{W \times D}$
- Inverted File (Index)
  - Postings (*e.g.*, $w_t \rightarrow (d_1, f_{wt,d1}), (d_2, f_{wt,d}), \ldots, (d_n, f_{wt,dn})$
  - Variable length records
- Lexicon:
  - String W_t
  - Document frequency f_t
  - Address within inverted file I_t
  - Sorted, fixed length records

Lexicon	W_f	D_1	D_2	D_3	D_4	D_5	D_6	…	D_m
W_1	2 →		1		1				
W_2	3 →	2			1				
W_3	1 →	1							
W_4	2 →			1			1		
	2 →								
W_5	3 →	1			1				
W_6			1		1	1			
…	→								
W_n									

×

Lexicon

Inverted File (Postings File)

To think about: What type of entries are missing from the search engine index that are present in the book index?

Trading precision for size, redux

Pop Quiz: Which of these techniques are more effective?

Typical:
Lexicon = 30 MB   Inverted File: 400 MB

- Stemming
  - Affects Lexicon   - Small effect – ~1% savings
- Case folding
  - Affects Lexicon   - Small effect – ~1% savings
- Stop words
  - Affects Inverted File   -Big effect! – ~30% savings but will depend on threshold

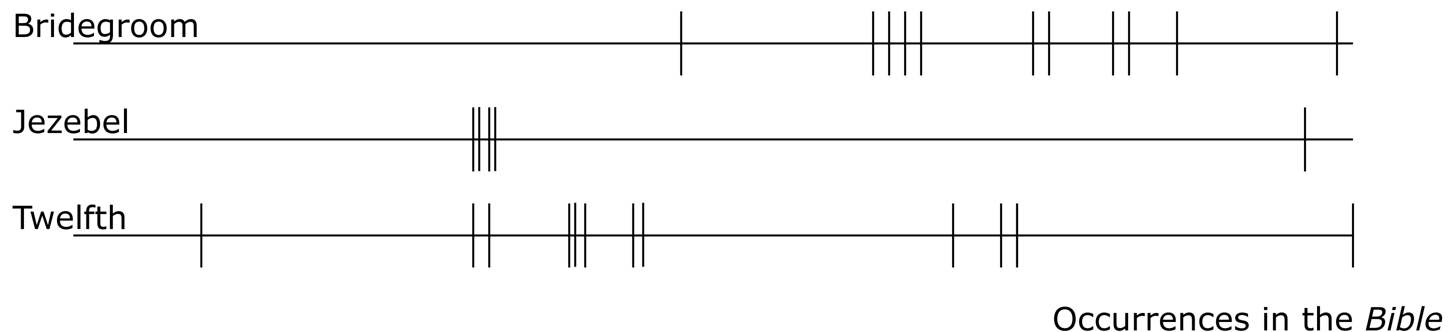
Is fine-grained indexing worthwhile?

- **Problem**: still have to scan document to find the term.

Image	(D1, 2), (D4, 1)	→	Image	(D1, 2; 10, 205), (D4, 1, 3993)
Implicit	(D2, 1), (D3, 1) …		Implicit	(D2, 1; 242), (D3, 1; 233) …
Index	(D5, 3), (D2, 1) …		Index	(D5, 3; 20, 42, 3920), (D2, 1 …
Inverse	(D2, 2)		Inverse	(D2, 2; 599, 847)
Internet	(D1, 2), (D3, 2) …		Internet	(D1, 2; 12, 43), (D3, 2; 302, …

- Cons:
  - Need access methods to take advantage
  - Extra storage space overhead (variable sized)
- Alternative methods:
  - Hierarchical encoding (doc #, para #, sent #, word #) to shrink offset size
  - Split long documents into *n* shorter ones.

Inverted file compression



- Clue: Encode *gap length* instead of offset
- Use small number of bits to encode more common gap lengths
  - (e.g., Huffman encoding)
- Better: Use a distribution of expected gap length (e.g., Bernoulli process)
  - If p = prob that any word x appears in doc y, then
  - Then $p_{\text{gap size } z} = (1-p)^z p$. This constructs a geometric distribution.
- Works for intra and inter-document index compression
  - Why does it hold for documents as well as words?

Building the index – Memory based inversion

Initialize empty dictionary S

// Phase I – collection of term appearances in memory

For each document D_d in collection, $1 \leq d \leq N$

    Read D_d, parsing it into index terms

    For each index term t in D_d

        Calculate $f_{d,t}$

        Search in S for t, if not present, insert it

        Append node $(d, f_{d,t})$ to list for term t

// Phase II – dump inverted file

For each term $1 \leq t \leq n$

    Start a new inverted file entry

        Append each appropriate $(d, f_{d,t})$ in list to entry

    Append to inverted file

Sort-based inversion

- **Idea**: try to make random access of disk (memory) sequential

// Phase I – collection of term appearances on disk
For each document D_d in collection, $1 \le d \le N$
    Read D_d, parsing it into index terms
    For each index term t in D_d
        Calculate fd,t
        Dump to file a tuple (t,d,$f_{d,t}$)

// Phase II – sort tuples
Sort all the tuples (t,d,f) using External Mergesort

// Phase III – write output file
Read the tuples in sorted order and create inverted file

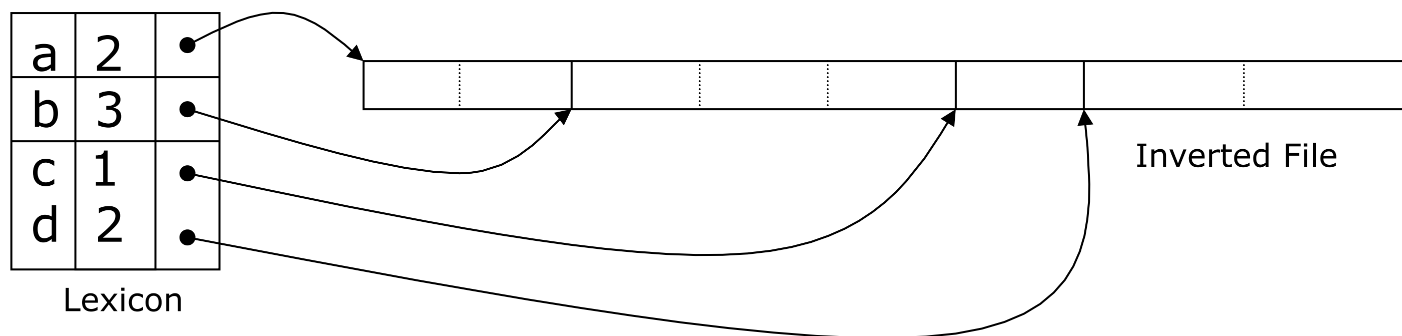
Sort based inversion: example

Initial dump from corpus	Sorted Runs	Merged Runs (fully sorted)
<a,1,2>	<a,1,1>	<a,1,1>
<b,1,2>	<a,2,2>	<a,2,2>
<c,1,1>	<b,1,2>	<b,1,2>
<a,2,2>	<c,1,1>	<b,2,1>
<d,2,1>	<b,2,1>	<b,3,1>
<b,2,1>	<b,3,1>	<c,1,1>
<b,3,1>	<d,2,1>	<d,2,1>
<d,3,1>	<d,3,1>	<d,3,1>

- What’s the performance of this algorithm?
- Saves memory but very disk intensive!

Using a first pass for the lexicon

- Gets us $f_{d,t}$ and N
  - **Savings**: For any t, we know $f_{d,t}$, so can use an array vs. LL (shrinks record by 40%!)



Lexicon-based inversion

- Partition inversion as |I|/|M| = k smaller problems
  - build 1/k of inverted index on each pass
  - (e.g., a-b, b-c, …, y-z)
  - Tuned to fit amount of main memory in machine
  - Just remember *boundary words*
- Can pair with disk strategy
  - Create k temporary files and write tuples $(t,d,f_{d,t})$ for each partition on first pass
  - Each second pass builds index from temporary file

Inversion – Summary of Techniques

- How do these techniques stack up?
- Assume a 5 GB corpus and 40 MB main memory machine

Technique	Memory (MB)	Disk (GB)	Time (Hours)
*Linked lists (memory)	4000	0	6
Linked lists (disk)	30	4	1100
Sort-based	40	8	20
Lexicon-based	40	0	79
Lexicon w/ disk	40	4	12

Source – Managing Gigabytes

Query Matching

- Now that we have an index, how do we answer queries?

Assuming a simple word matching engine:

For each query term t
  Stem t
  Search lexicon
  Record f_t and its inverted entry address, I_t
Select a query term t
Set list of candidates, $C = I_t$
For each remaining term t
  Read its I_t
  For each d in C, if d not in I_t set $C = C - \{d\}$

Conjunctive (AND) processing

- X and Y and Z – high precision
- X or Y or Z – high recall
- Which algorithm is the above?

Boolean Model

- Query processing strategy:
  - Join less frequent terms first
  - Even in ORs, as merging takes longer than lookup
- Problems with Boolean model:
  - Retrieves too many or too few documents
  - Longer documents are tend to match more often because they have a larger vocabulary
  - Need ranked retrieval to help out

- Boolean assigns same importance to all terms in a query

Phua Chu Kang dates at Esplanade | Search

  - "Esplanade" has same weight as "date"

- One way:
  - Assign weights to the words, make more important words worth more
  - Process results in *q* and *d* vectors: (word, weight), (word, weight) … (word, weight)

Term Frequency

Xxxxxxxxxxxxxx Mee Swa xxxxxxxxxxx xxxxxxxx xxxxxxxxxxx Prata xxxxxxx xxxxxxxxxx xxxxxxxx Chili Crab. Xxxxxxxxxx xxxxxxxxxx Chili Crab xxxxxxxx. Xxxxxxxxxx xxxxxxxx Laksa. Xxxxxxxxx xxxxxxx Chili Crab.

$\frac{f_{d}}{\max_i(f_{d,i})}$

(Relative) term frequency can indicate importance.

- $R_{d,f} = f_{d,t}$
- $R_{d,t} = 1 + \ln f_{d,t}$
- $R_{d,t} = (K + (1-K) \frac{f_{d,t}}{\max_i(f_{d,i})})$

Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, "memex" will do.

Inverse Document Frequency

Consider a future **device** for **individual** use, which is a sort of **mechanized private** file and **library**. It needs a name, and, to coin one at **random**, "**memex**" will do.

- Words with higher f_t are less discriminative.
- Use inverse to measure importance:
  - $w_t = 1/f_t$
  - $w_t = \ln(1 + N/f_t)$ ← this one is most common
  - $w_t = \ln(1 + f^m/f_t)$, where f^m is the max observed frequency

Question: What's the ln () here for?

- Many variants, but all capture:
  - Term frequency:
    $R_{d,t}$ as being monotonically increasing
  - Inverse Document Frequency:
    W_t as being monotonically decreasing
- Standard formulation is:

$$w_{d,t} = r_{d,t} \times w_t$$

$$= (1 + \ln(f_{d,t})) \times \ln(1 + N/f_t)$$

- Problem:
  - $r_{d,t}$ grows as document grows, need to normalize; otherwise biased towards long documents

Calculating Similarity

- Euclidean Distance - bad
  - $M(Q,D_d) = sqrt\ (\Sigma\ |w_{q,t} - w_{d,t}|^2)$
  - Dissimilarity Measure; use reciprocal
  - Has problem with long documents, **why**?
- Actually don’t care about vector length, just their direction
  - Want to measure difference in direction

Cosine Similarity

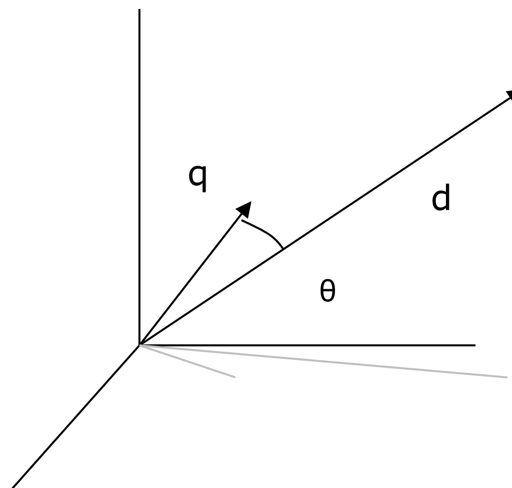
- If X and Y are two *n*-dimensional vectors:

  $X \cdot Y = |X|\ |Y| \cos\theta$

  $\cos\theta = X \cdot Y \,/\, |X|\ |Y|$

  $= 1$ when identical

  $= 0$ when orthogonal



$$\mathrm{Cos}\ (Q,Dd) = Q \cdot D_d \,/\, |Q|\ |D_d|$$
$$= (1/W_q W_d)\ \Sigma\ w_{q,t} \cdot w_{d,t}$$
$$= (1/W_d)\ \Sigma\ w_{q,t} \cdot w_{d,t}$$

Calculating the ranked list

$$\frac{1}{W_d W_q} \sum_{t \in Q \cap D_d} (1 + \ln f_{d,t}) \bullet \ln(1 + \frac{N}{f_t})$$

- To get the ranked list, we use doc. accumulators:

For each query term t, in order of increasing f_t,
    Read its inverted file entry I_t
    Update acc. for each doc in I_t: A_d += ln (1 + $f_{d,t}$) $\times w_t$
For each A_d in A
    A_d /= W_d **// that's basically cos θ, don't use w_q**
Report top r of A

Accumulator Storage

- Holding all possible accumulators is expensive
  - Could need one for each document if query is broad
- In practice, use fixed |A| wrt main memory. What to do when all used?
  - Quit: use ranks as they are
  - Continue processing on |A| documents to get accurate ranks (preferred)

- Want to return documents with largest cos values.

- How? Use a min-heap

Load r A values into the heap H

Process remaining A-r values

  If $A_d > \min\{H\}$ then

    Delete min{H}, add A_d, and sift

// H now contains the top r exact cosine values

To think about

- How do you deal with a dynamic collection?
- How do you support phrasal searching?
- What about wildcard searching?
  - What types of wildcard searching are common?