Document ranking

Text-based Ranking

(1° generation)

Doc is a binary vector

- Binary vector X,Y in $\{0,1\}^D$

- Score: *overlap measure*

$$|X \cap Y|$$

What's wrong ?

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Normalization

- Dice coefficient (wrt avg #terms):

$$2|X \cap Y| / (|X| + |Y|)$$

NO, triangular

- Jaccard coefficient (wrt possible terms):

$$|X \cap Y| / |X \cup Y|$$

OK, triangular

What’s wrong in binary vect?

Overlap matching doesn’t consider:

- Term frequency in a document
  - Talks more of t ? Then t should be weighted more.

- Term scarcity in collection
  - ***of*** commoner than ***baby bed***

- Length of documents
  - score should be normalized

A famous "weight": tf-idf

$$w_{t,d} = tf_{t,d} \times \log(n / n_t)$$

$tf_{t,d} =$ Number of occurrences of term *t* in doc *d*

$$idf_t = \log\left(\frac{n}{n_t}\right)$$

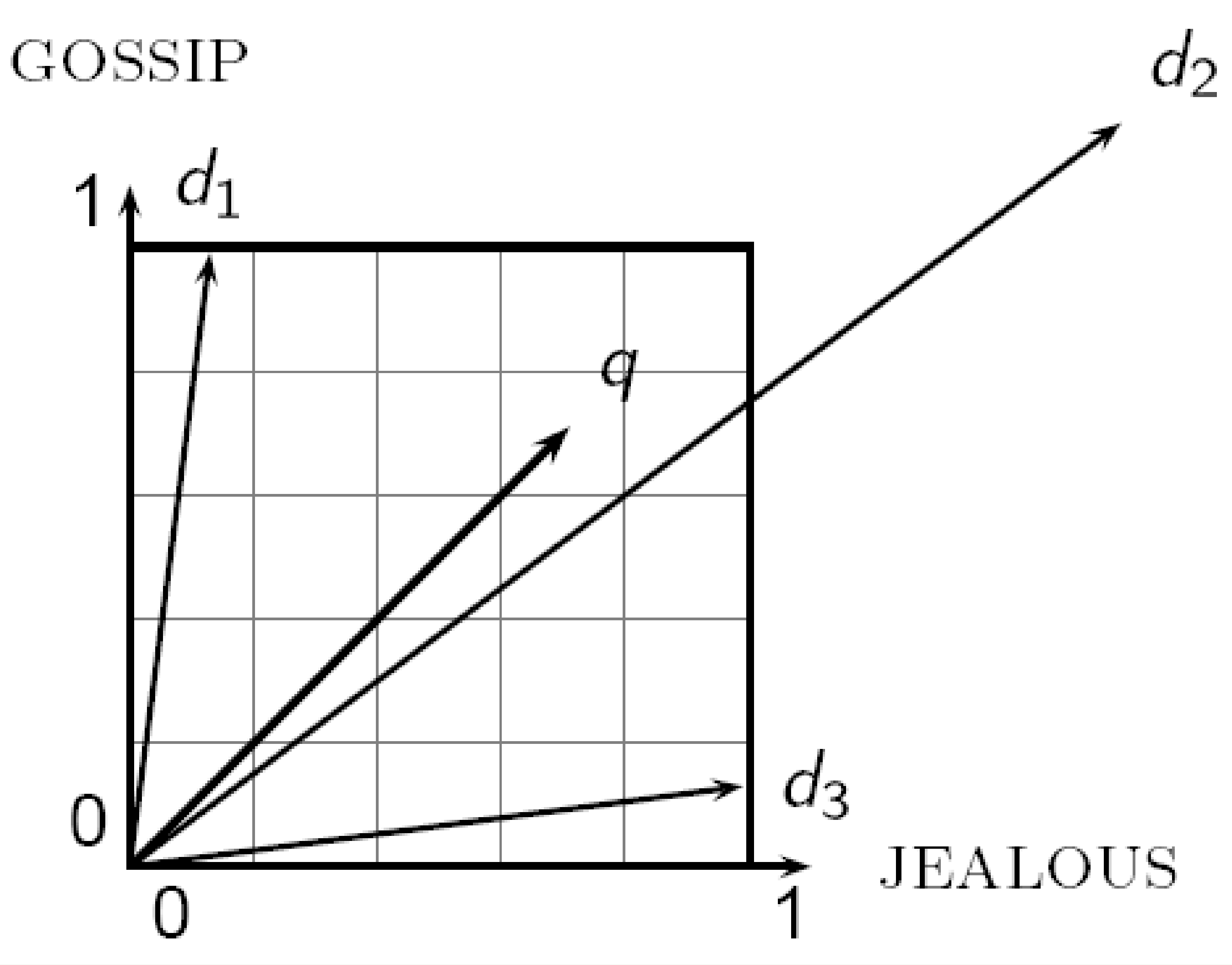
where n_t = #docs containing term *t*

n = #docs in the indexed collection

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13,1	11,4	0,0	0,0	0,0	0,0
Brutus	3,0	8,3	0,0	1,0	0,0	0,0
Caesar	2,3	2,3	0,0	0,5	0,3	0,3
Calpurnia	0,0	11,2				
Cleopatra	17,7	0,0				
mercy	0,5	0,0				
worser	1,2	0,0	0,6	0,6	0,6	0,0

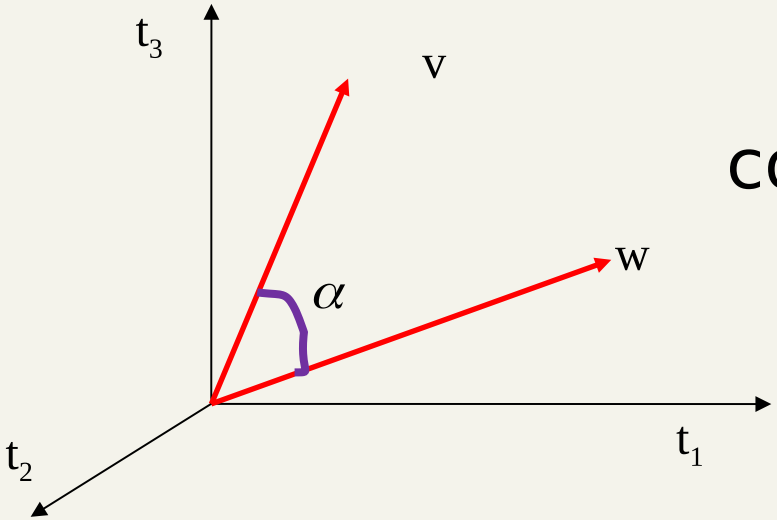
Vector Space model

Why distance is a bad idea



Easy to Spam

An example



$$\cos(\alpha) = v \bullet w \,/\, ||v|| * ||w||$$

Computational Problem

#pages .it ≈ a few billions

terms ≈ some mln

#ops ≈ 10^{15}

1 op/ns ≈ 10^{15} ns ≈ 1 week

!!!!

document	v	w
term 1	2	4
term 2	0	0
term 3	3	1

$$\cos(\alpha) = 2*4 + 0*0 + 3*1 \,/\, \mathrm{sqrt}\{ 2^2 + 3^2 \} * \mathrm{sqrt}\{ 4^2 + 1^2 \} \cong 0{,}75 \rightarrow 40°$$

cosine(query,document)

Dot product

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|D|} q_i d_i}{\sqrt{\sum_{i=1}^{|D|} q_i^2}\sqrt{\sum_{i=1}^{|D|} d_i^2}}$$

q_i is the tf-idf weight of term *i* in the query: $w_{i,q} \rightarrow w_{t,q}$

d_i is the tf-idf weight of term *i* in the document: $w_{i,d} \rightarrow w_{t,d}$

cos(*q*,*d*) is the cosine similarity of *q* and *d* … or, equivalently, the cosine of the angle between *q* and *d*.

Storage

$$w_{t,d} = tf_{t,d} \times \log(n / n_t)$$

- For every term t, we have in memory the length n_t of its posting list, so the IDF is implicitly available.

- For every docID *d* in the posting list of term *t*, we store its frequency $tf_{t,d}$ which is tipically small and thus stored with **unary/gamma**.

Computing cosine score

CosineScore(q)

float $Scores[N] = 0$
float $Length[N]$
for each query term t
do calculate $\mathrm{w}_{t,q}$ and fetch postings list for t
    for each pair$(d, \mathrm{tf}_{t,d})$ in postings list
    do $Scores[d] += \mathrm{w}_{t,d} \times \mathrm{w}_{t,q}$
Read the array *Length*
for each d
do $Scores[d] = Scores[d]/Length[d]$
return Top K components of $Scores[]$

We could restrict to docs in the intersection

Vector spaces and other operators

- Vector space OK for bag-of-words queries
  - Clean metaphor for similar-document queries
  - Not a good combination with operators: Boolean, wild-card, positional, proximity
- *First generation* of search engines
  - Invented before “spamming” web search

Top-K documents

Approximate retrieval

Speed-up top-k retrieval

- **Costly** is the computation of the cos()
- Find a set *A* of *contenders*, with $k < |A| << N$
  - Set *A* does not necessarily contain all top-k, but has many docs from among the top-*k*
  - Return the top-k docs in *A*, according to the score
- The same approach is also used for other (non-cosine) scoring functions
- Will look at several schemes following this approach

How to select A’s docs

- Consider docs containing **at least one** query term (obvious… as done before!).

- Take this further:
  1. Only consider docs containing **most** query terms
  2. Only consider **high-idf** query terms
  3. **Champion lists**: top scores
  4. **Fancy hits**: for complex ranking functions
  5. Clustering

Approach #1: Docs containing many query terms

- For multi-term queries, compute scores for docs containing most query terms
  - Say, at least q-1 out of q terms of the query
  - Imposes a “soft AND” on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

Many query terms

Antony	3	4	8	16	32	64	128	
Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	13	21	34
Calpurnia	13	16	32					

Scores only computed for docs 8, 16 and 32.

Approach #2: High-idf query terms only

- High-IDF means short posting lists = rare term

- **Intuition**: *in* and *the* contribute little to the scores and so <u>don’t alter rank-ordering much</u>

- Only accumulate ranking for documents in those posting lists

for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
  for each pair$(d, \mathrm{tf}_{t,d})$ in postings list
  do $Scores[d] += w_{t,d} \times w_{t,q}$

Approach #3: Champion Lists

- Preprocess: Assign to each term, its *m* best documents

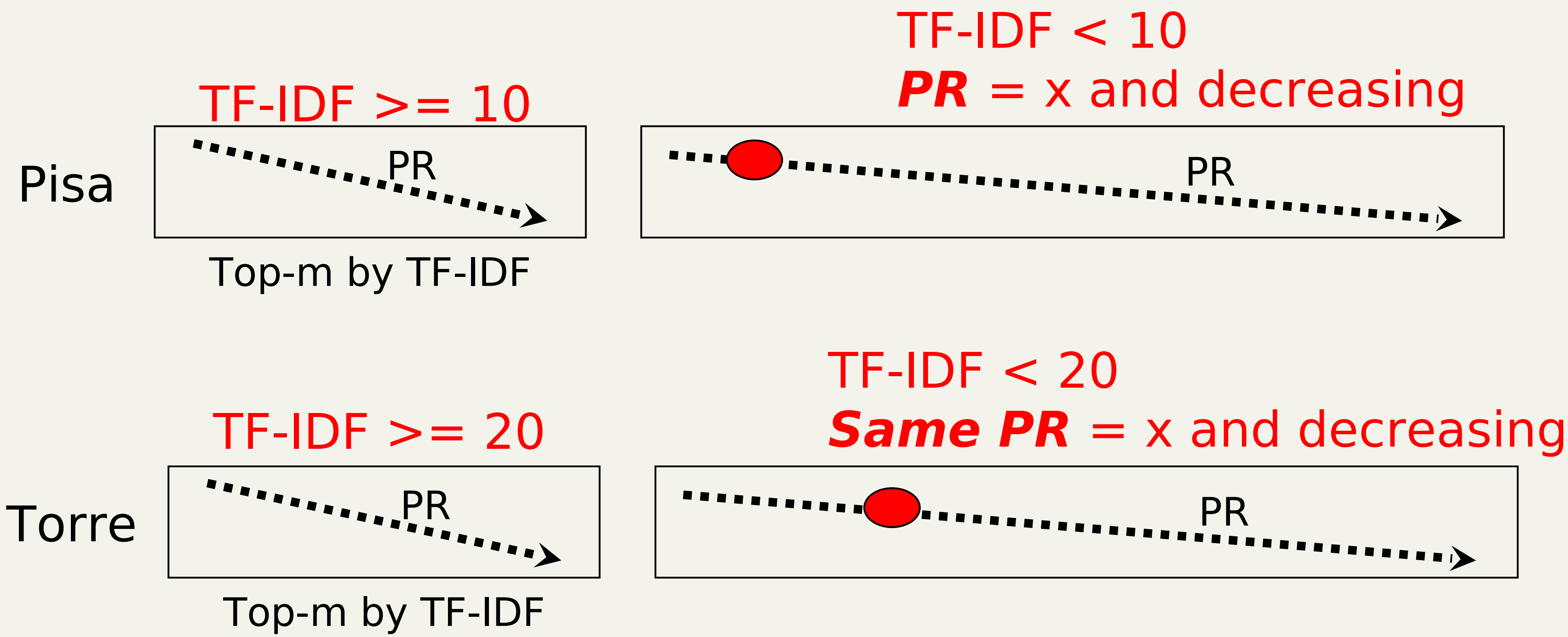
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13.1	11.4	0.0	0.0	0.0	0.0
Brutus	3.0	8.3	0.0	1.0	0.0	0.0
Caesar	2.3	2.3	0.0	0.5	0.3	0.3
Calpurnia	0.0	11.2	0.0	0.0	0.0	0.0
Cleopatra	17.7	0.0	0.0	0.0	0.0	0.0
mercy	0.5	0.0	0.7	0.9	0.9	0.3
worser	1.2	0.0	0.6	0.6	0.6	0.0

- Search:
  - If $|Q| = q$ terms, merge their preferred lists ($\leq mq$ answers).
  - Compute COS between Q and these docs, and choose the top *k*.

Need to pick $m>k$ to work well empirically.

Approach #4: *Fancy-hits* heuristic

- Preprocess:
  - Assign docID by decreasing *PR weight*
  - Sort by docID = order by decring *PR weight*
  - Define FH(t) = *m* docs for t with highest *tf-idf weight*
  - Define IL(t) = the rest
- *Idea: a document that scores high should be in FH or in the front of IL*
- Search for a t-term query:
  - First FH: Compute the score of all docs in their FH, like Champion Lists, and keep the top-*k* docs.
  - Then IL: scan ILs and check the common docs
    - Compute the score and possibly insert them into the top-*k*.
    - Stop when M docs have been checked or the PR score becomes smaller than some threshold.



→ If score is sum PR and TF-IDF values, then

→ Any next match, has PR < ***x*** and TF-IDF < **30**

→ So that if ***x*** + **30** < minimum in the Heap, then stop scan

Modeling authority

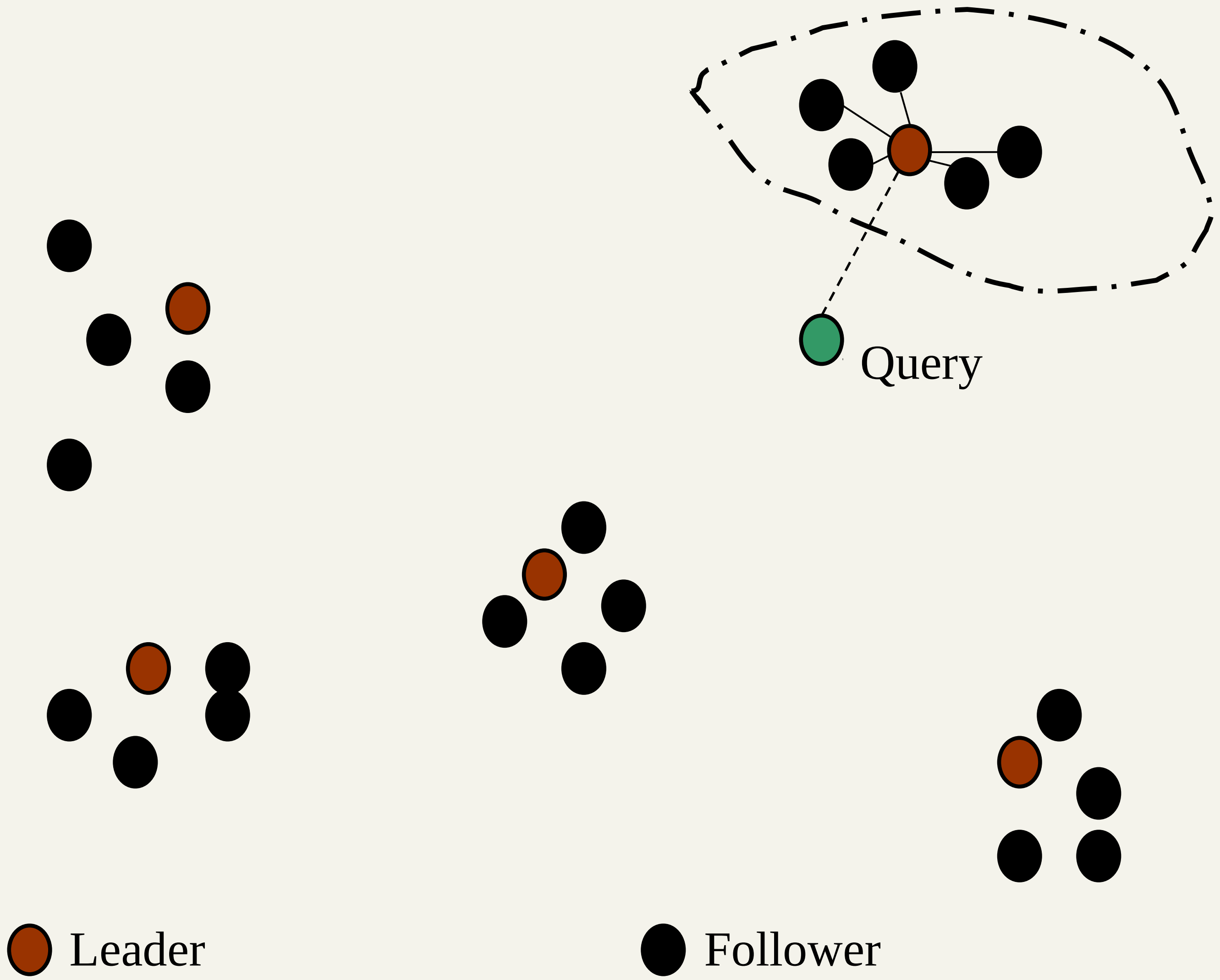
- Assign to each document a *query-independent* quality score in [0,1] to each document *d*
  - Denote this by *g(d)*

- Thus, a quantity like the number of citations (?) is scaled into [0,1]

Champion lists in *g(d)*-ordering

- Can combine champion lists with *g(d)*-ordering
- Or, maintain for each term a champion list of the $r>k$ docs with highest $g(d)$ + tf-idf$_{td}$
- g(d) may be the PageRank
- Seek top-*k* results from only the docs in these champion lists

Approach #5: Clustering



Cluster pruning: preprocessing

- Pick $\sqrt{N}$ *docs* at random: call these *leaders*
- For every other doc, pre-compute nearest leader
  - Docs attached to a leader: its *followers;*
  - Likely: each leader has ~ $\sqrt{N}$ followers.

Cluster pruning: query processing

- Process a query as follows:
  - Given query *Q*, find its nearest *leader L.*
  - Seek *K* nearest docs from among *L*'s followers.

Why use random sampling

- Fast
- Leaders reflect data distribution

General variants

- Have each follower attached still to *the* nearest leader.
- But given now the query, find b=4 (say) nearest leaders and their followers. For them compute the scores and then take the top-k ones
- Can recur on leader/follower construction.

Exact Top-K documents

Exact retrieval

Goal

- Given a query Q, find the **exact** top *K* docs for Q, using some ranking function *r*

- Simplest Strategy:
  1) Find all documents in the intersection
  2) Compute score *r(d)* for all these documents *d*
  3) Sort results by score
  4) Return top *K* results

Background

- Score computation is a large fraction of the CPU work on a query
  - Generally, we have a tight budget on latency (say, 100ms)
  - *We can’t exhaustively score every document!*
- Goal is to cut CPU usage for scoring, without compromising on the quality of results
- Basic idea: avoid scoring docs that won’t make it into the top *K*

The WAND technique

- It is a **pruning method** which uses a **max heap** over the **real** document scores
- There is a proof that the docIDs in the heap at the end of the process **are the exact top-K**
- Basic idea reminiscent of **branch and bound**
  - We maintain a running **threshold** score = the *K*-th highest score computed so far
  - We prune away all docs whose scores are guaranteed to be below the threshold
  - We compute exact scores for only the un-pruned docs

Index structure for WAND

- Postings ordered by docID

- Assume a **special iterator** on the postings that can “go to the first docID > X”
  - *using skip pointers*
  - *Using the Elias-Fano’s compressed lists*

- The **“iterator”** moves only to the right, to larger docIDs

Score Functions

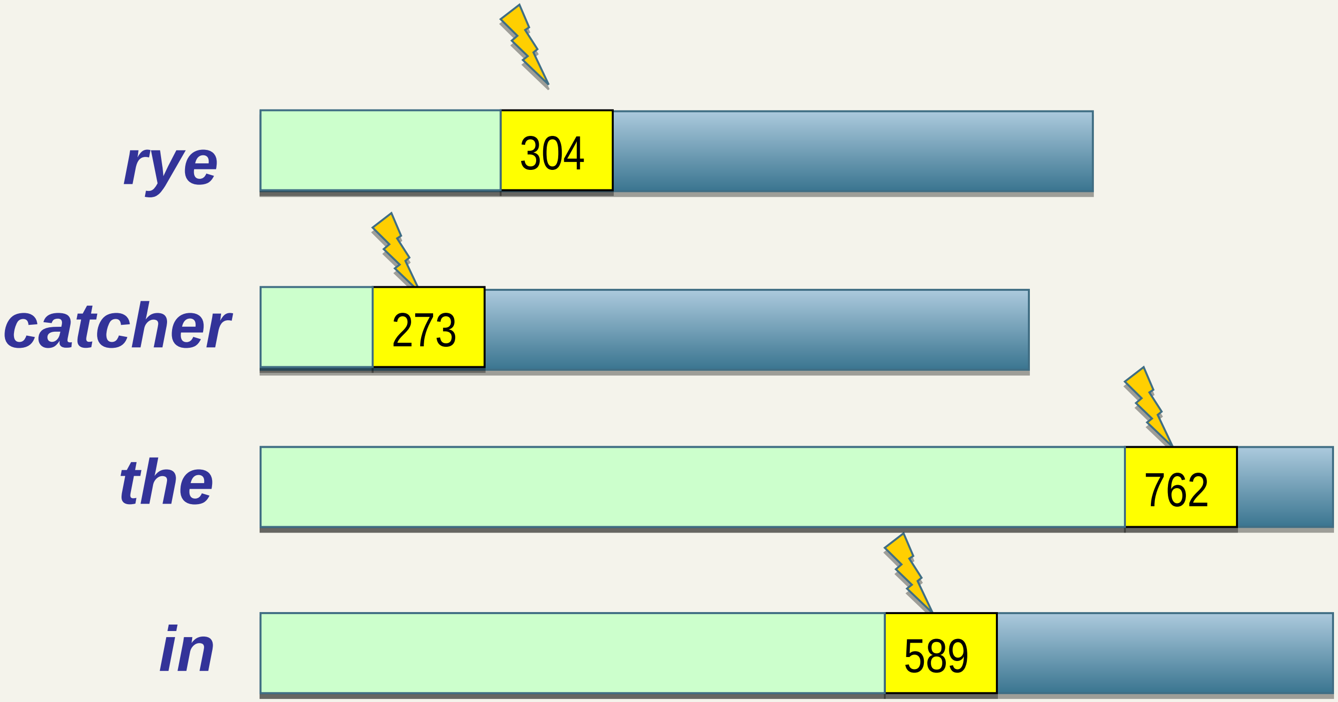
- We assume that:
  - $r(t,d)$ = *score of t in d*
  - The score of the document d is the sum of the scores of query terms: $r(d) = r(t_1,d) + \ldots + r(t_n,d)$
- Also, for each query term *t*, there is some upper-bound *UB(t)* such that, for all *d*,
  - $r(t,d) \leq UB(t)$
  - These values are pre-computed and stored

Threshold

- We keep inductively a threshold θ such that for **every document *d* within the top-*K***, it holds that $r(d) \geq \theta$
  - θ can be initialized to 0
  - It is raised whenever the “worst” of the currently found top-*K* has a score above the threshold

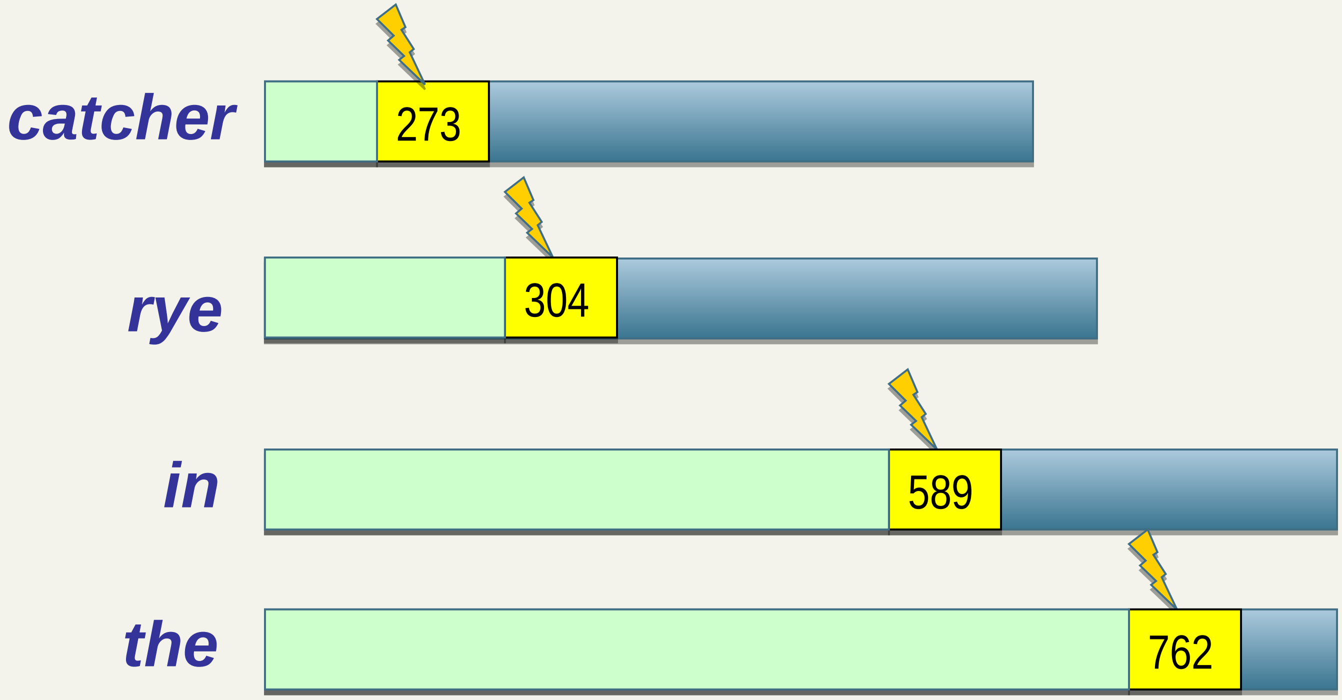
The Algorithm

- Example Query: *catcher in the rye*
- Consider a generic step in which each iterator is in some position of its posting list



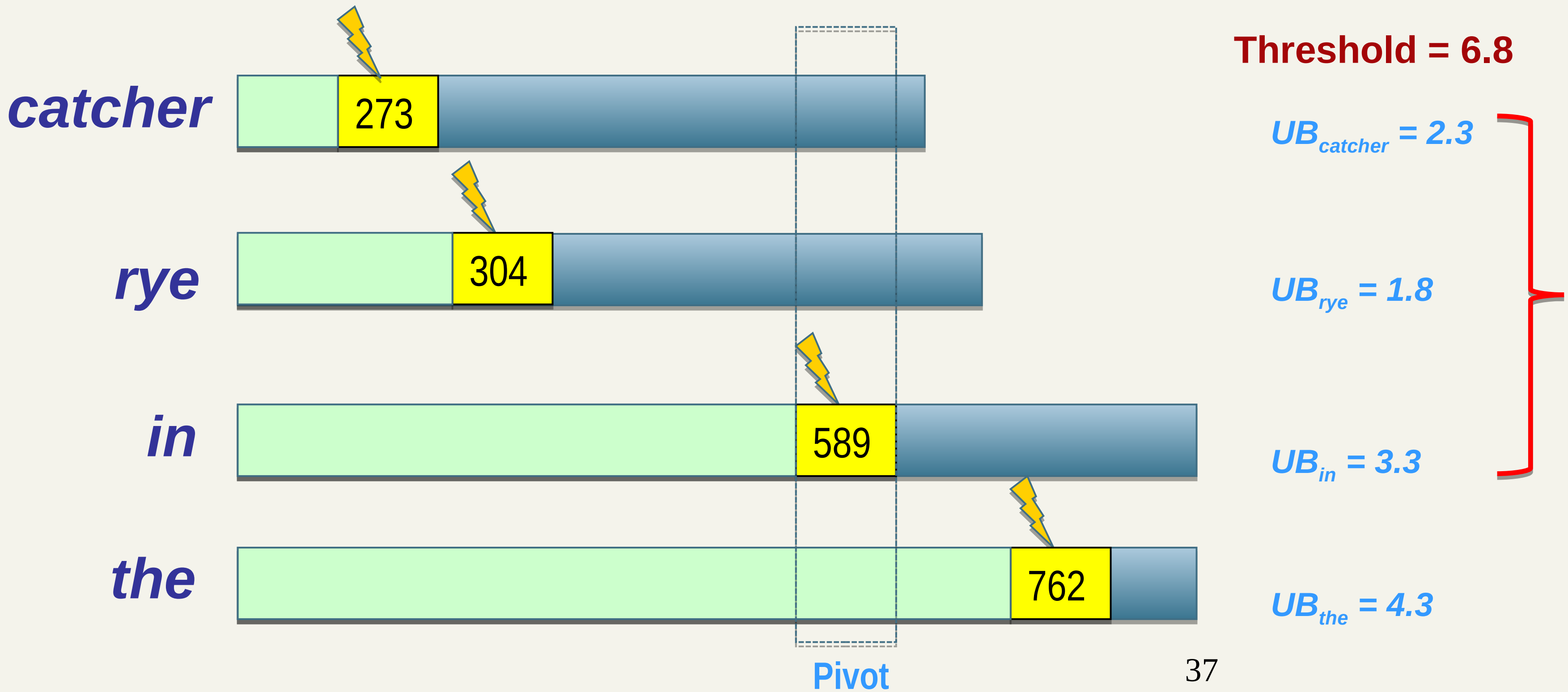
Sort Pointer

- Sort the pointers to the inverted lists by increasing document id

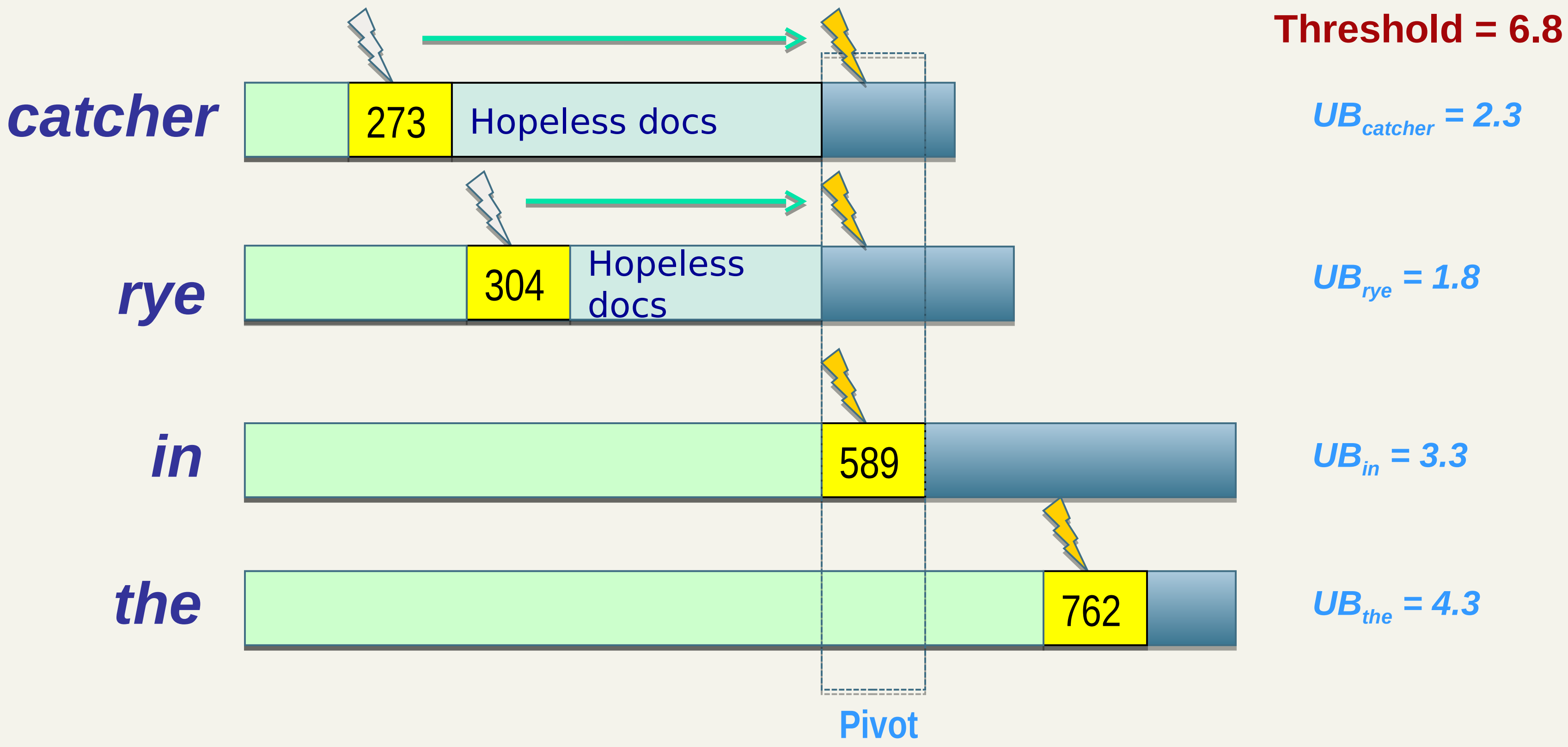


Find Pivot

- Find the "pivot": The first pointer in this order for which the sum of upper-bounds of the terms is at least equal to the threshold

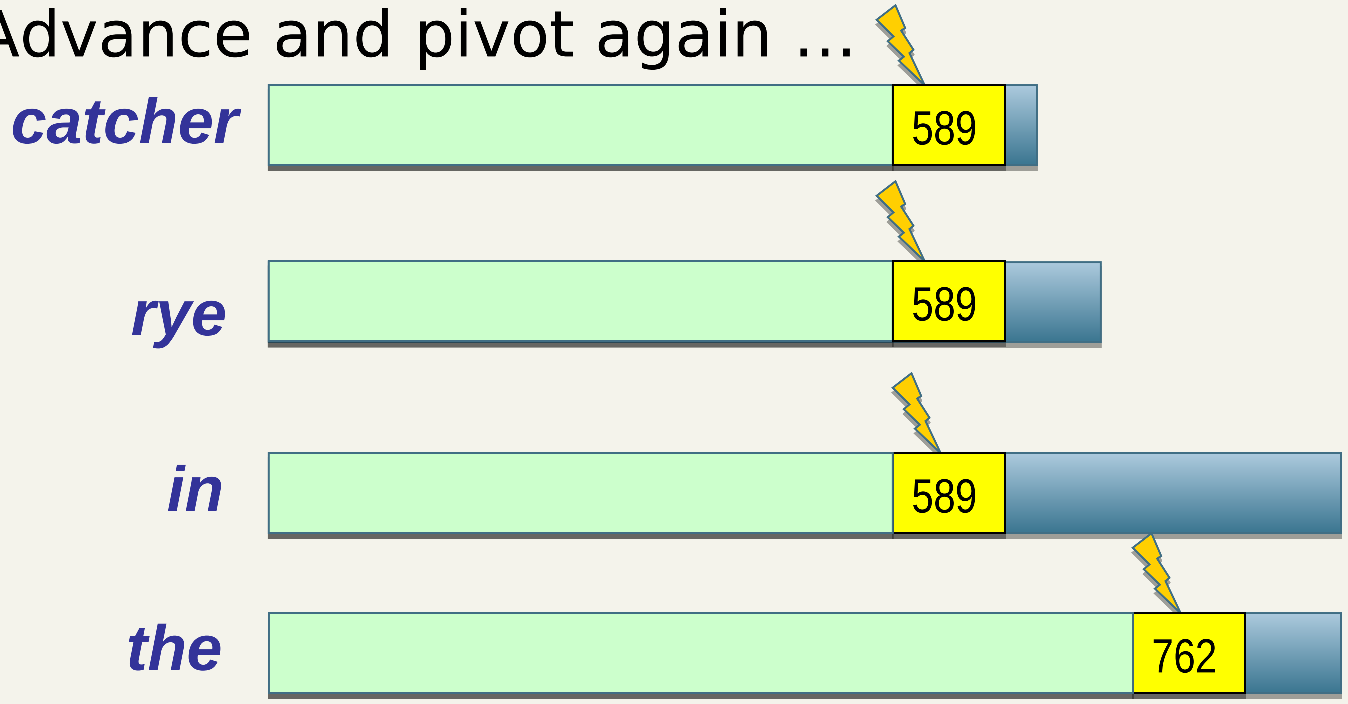


Prune docs that have no hope



Compute pivot's score

- **If 589 is present** in enough postings **(soft AND)**, compute its full score – else move pointers right of 589
  - If 589 is inserted in the current top-K, update threshold!
- Advance and pivot again …

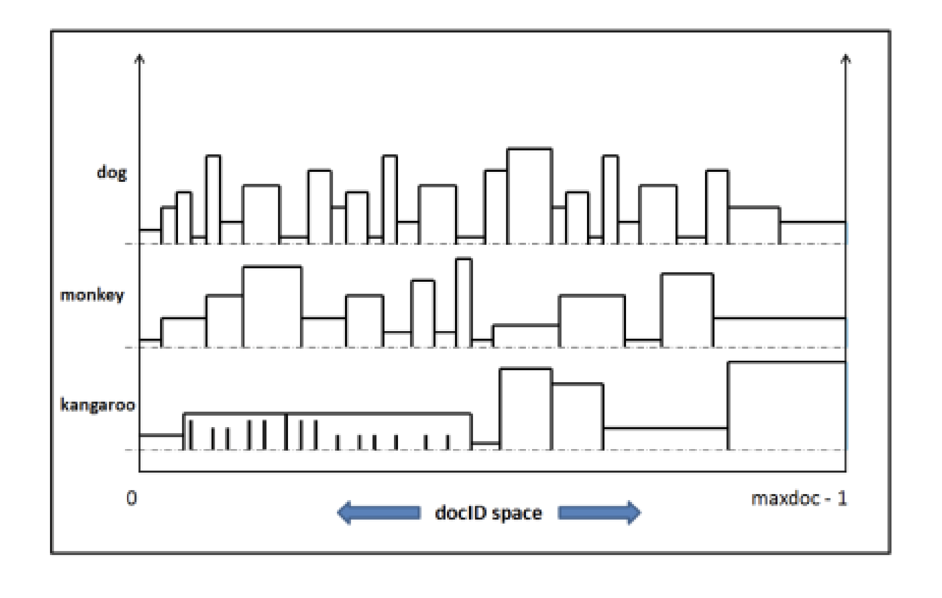


WAND summary

- In tests, WAND leads to a 90+% reduction in score computation
  - Better gains on longer queries
- WAND gives us safe ranking

Blocked WAND

- UB(t) was over the **full list** of t
- To improve this, we add the following:
  - Partition the list into blocks
  - Store for each block b the maximum score UB_b(t) among the docIDs stored into it



The new algorithm: Block-Max WAND

Algorithm *(2-levels check)*

- As in previous WAND:
  - **p** = pivoting docIDs via threshold θ taken from the max-heap, and let **d** be the pivoting docID in list(p)

- Move block-by-block in lists 0..p-1so reach blocks that **may contain d** (their docID-ranges overlap)
  - Sum the **UBs** of those blocks
  - if the sum ≤ θ then skip the block whose right-end is the leftmost one; **repeat from the beginning**
  - Compute score(d), if it is ≤ θ then move iterators to next first docIDs > d; **repeat from the beginning**
  - Insert d in the min-heap and re-evaluate θ

Document RE-ranking

Relevance feedback

Relevance Feedback

- Relevance feedback: user feedback on relevance of docs in initial set of results
  - User issues a (short, simple) query
  - The user marks some results as relevant or non-relevant.
  - The system computes a better representation of the information need based on feedback.
  - Relevance feedback can go through one or more iterations.

Rocchio (SMART)

- Used in practice:

$$\vec{q}_m = \alpha \vec{q}_0 + \beta \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \gamma \frac{1}{|D_{nr}|} \sum_{\vec{d}_j \in D_{nr}} \vec{d}_j$$

- ***D_r* =** set of <u>known</u> relevant doc vectors
- ***D_{nr}* =** set of <u>known</u> irrelevant doc vectors
- ***q_m* =** modified query vector; *q_0* = original query vector; ***α,β,γ***: weights (hand-chosen or set empirically)

- New query moves toward relevant documents and away from irrelevant documents

Relevance Feedback: Problems

- Users are often reluctant to provide explicit feedback

- It’s often harder to understand why a particular document was retrieved after applying relevance feedback

- There is **no clear evidence** that relevance feedback is the “best use” of the user’s time.

Pseudo relevance feedback

- Pseudo-relevance feedback automates the "manual" part of true relevance feedback.
  - Retrieve a list of hits for the user's query
  - Assume that the top k are relevant.
  - Do relevance feedback (e.g., Rocchio)
- Works very well on average
- But can go horribly wrong for some queries.
- Several iterations can cause query drift.

Query Expansion

- In **relevance feedback**, users give additional input (relevant/non-relevant) on documents, which is used to reweight terms in the documents
- In **query expansion**, users give additional input (good/bad search term) on words or phrases

How augment the user query?

- **Manual thesaurus** (costly to generate)
  - E.g. MedLine: physician, syn: doc, doctor, MD
- **Global Analysis** (static; all docs in collection)
  - Automatically derived thesaurus
    - (co-occurrence statistics)
  - Refinements based on query-log mining
    - Common on the web
- **Local Analysis** (dynamic)
  - Analysis of documents in result set

Quality of a search engine

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

Is it good ?

- How fast does it index
  - Number of documents/hour
  - (Average document size)

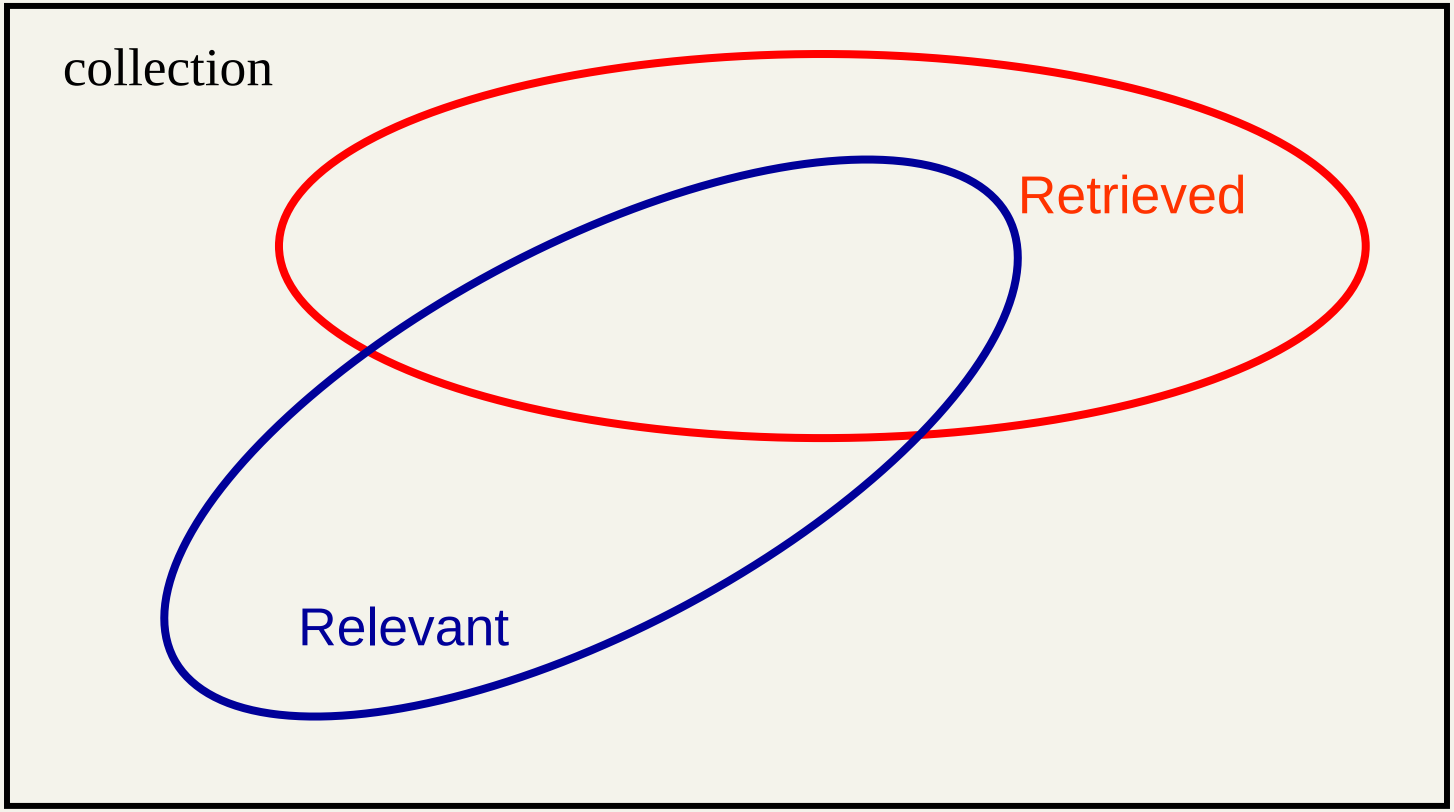
- How fast does it search
  - Latency as a function of index size

- Expressiveness of the query language

Measures for a search engine

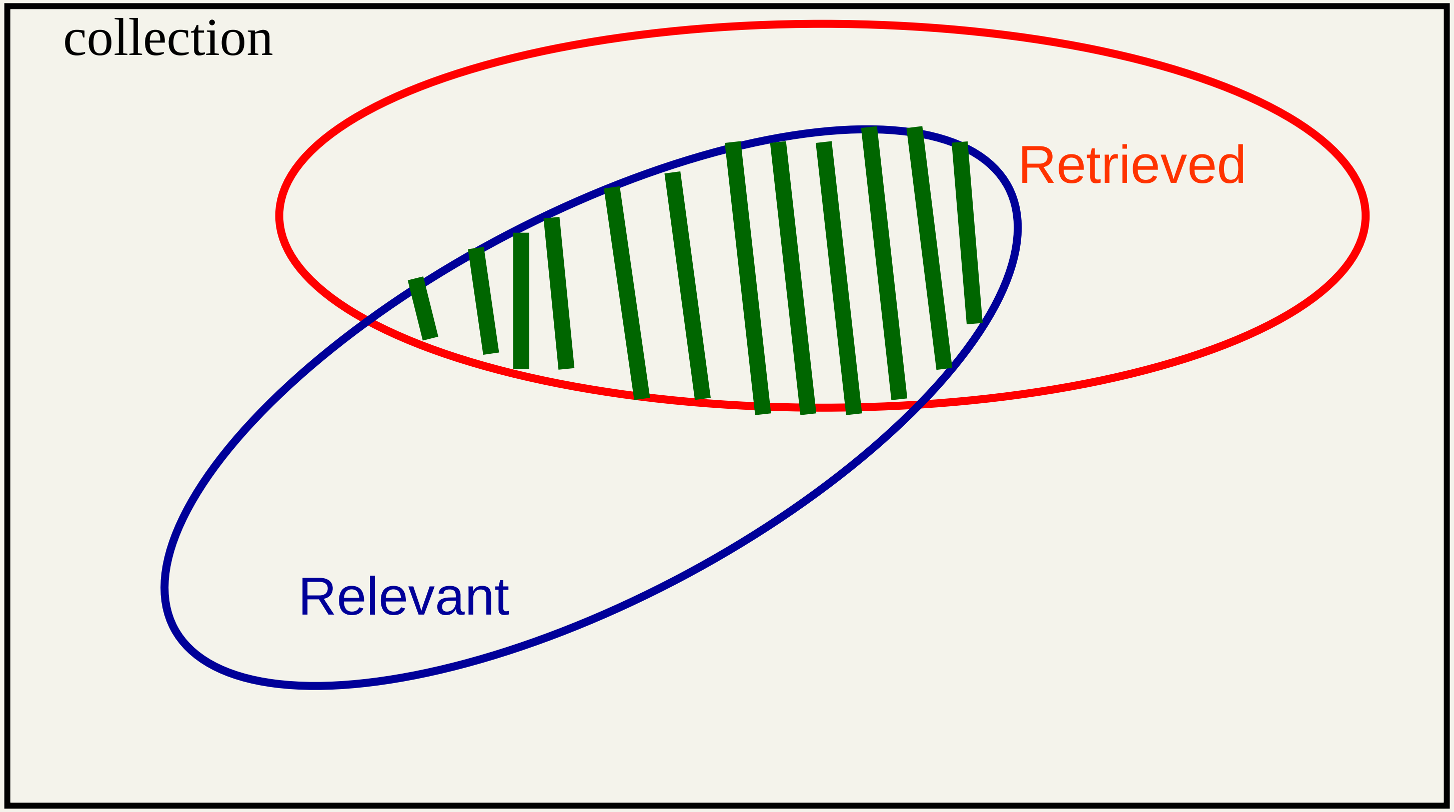
- All of the preceding criteria are *measurable*
- The key measure: *user happiness*
  …useless answers won't make a user happy
- User groups for testing !!

General scenario



Precision vs. Recall

- Precision: % docs retrieved that are relevant [issue "junk" found]
- Recall: % docs relevant that are retrieved [issue "info" found]



How to compute them

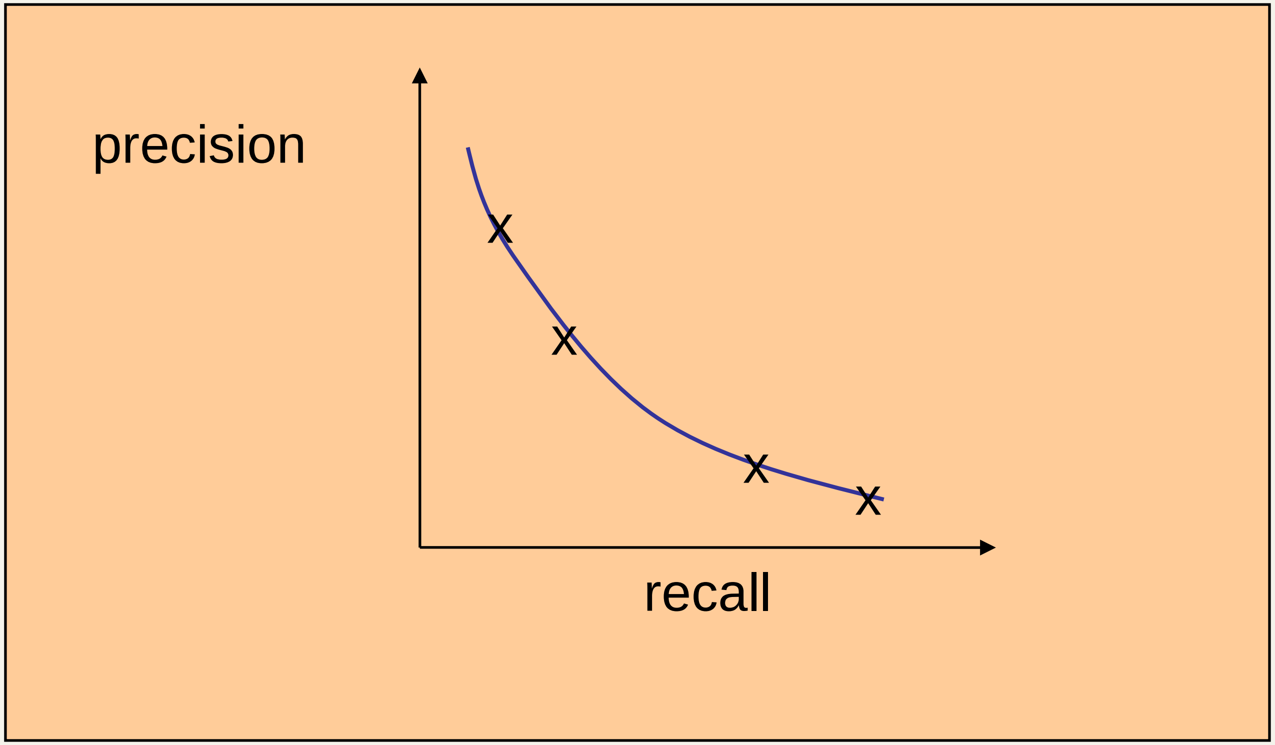
- **Precision**: fraction of retrieved docs that are relevant
- **Recall**: fraction of relevant docs that are retrieved

	Relevant	Not Relevant
Retrieved	tp (true positive)	fp (false positive)
Not Retrieved	fn (false negative)	tn (true negative)

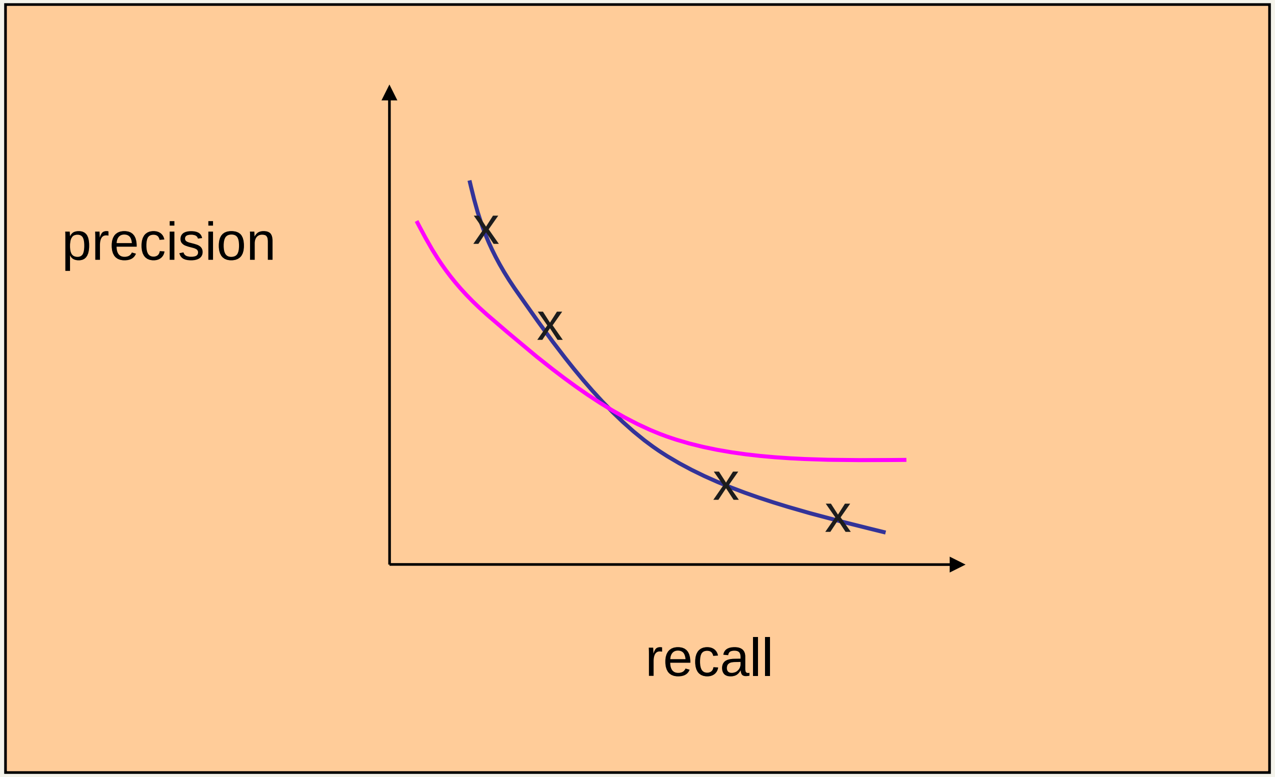
- Precision P = tp/(tp + fp)
- Recall R = tp/(tp + fn)

Precision-Recall curve

- Measure Precision at various levels of Recall



A common picture



F measure

- Combined measure *(weighted harmonic mean)*:

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha)\frac{1}{R}}$$

- People usually use balanced F_1 measure
  - i.e., with α = ½ thus 1/F = ½ (1/P + 1/R)