Information Retrieval

BITS Pilani
Pilani Campus

Abhishek
January 2020

innovate achieve lead

BITS Pilani
Pilani Campus

CS F469, Information Retrieval

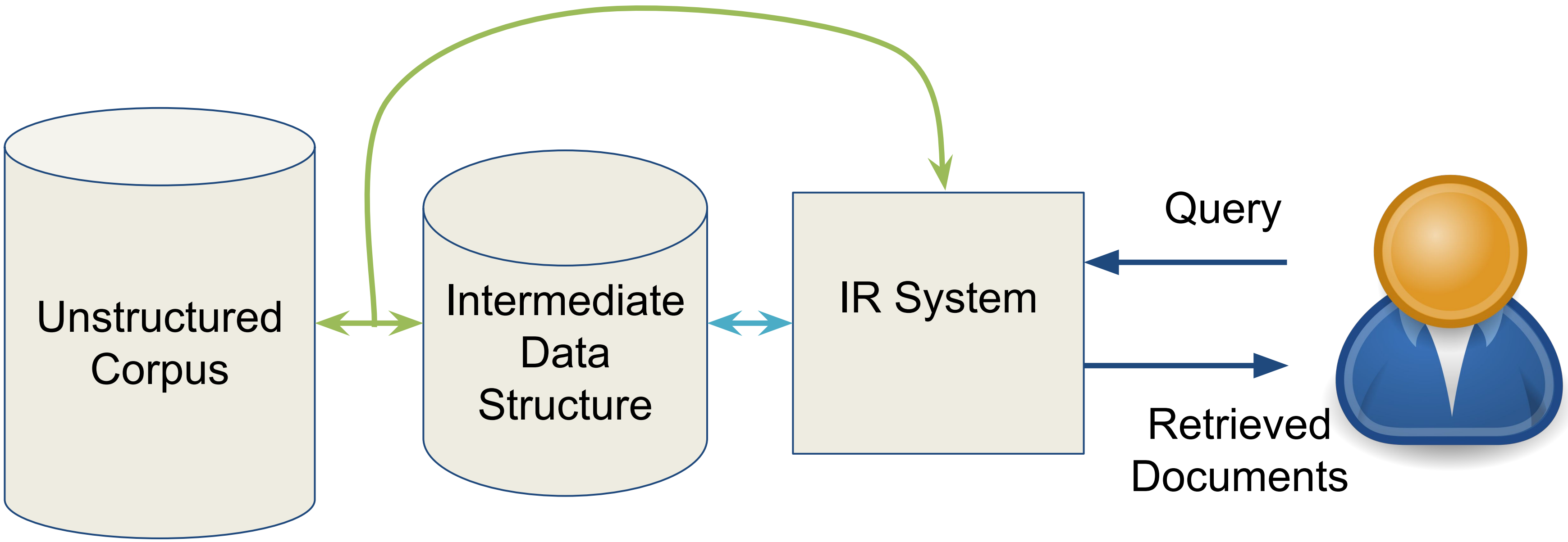
Lecture topics: Scoring, term weighting, and the vector space model

BITS Pilani
Pilani Campus

Most of these slides are based on:

https://web.stanford.edu/class/cs276/
https://www.inf.unibz.it/~ricci/ISR/
https://www.cis.uni-muenchen.de/~hs/teach/14s/ir/

Summary of previous lectures



Document processing, Index construction, index compression, boolean retrieval model, wildcard queries and alternate spellings, document ranking, IR system evaluation, probabilistic models, web crawlers, link analysis.

Recap of Previous Lecture

- Index Construction
  - Blocked sort-based indexing
  - Single pass in-memory indexing
  - Distributed Indexing
  - Dynamic Indexing

This Lecture

- Ranked Retrieval
  - Parametric and zone indexes
  - Term frequency and weighting
  - The vector space model for scoring

Issues with Boolean Retrieval

- Documents either **match** or they **don't**.
- Issue with Boolean Retrieval
  - **Feast or famine:** Boolean queries either returns in **too few (=0)** or **to many (>1000)** documents.
  - Might be good for advanced user or computer programs, however **not good for majority of the users**.
  - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
  - **Most users don't want to wade through 1000s of results**.
    - This is particularly true of web search.

Ranked Retrieval

- **Ranked Retrieval:** Rank the retrieved documents in the order of their **relevance** to the **query**. The retrieved documents are now considered as an **ordered list** instead of a **set**.
- **Free text queries:** Rather than a query language of **operators** and **expressions**, the user's query is just one or more words in a natural language.
- In principle, there are two separate choices here
  - the query language and the retrieval model
  - but in practice, ranked retrieval models have normally been associated with free text queries.

Feast or famine: not a problem in ranked retrieval

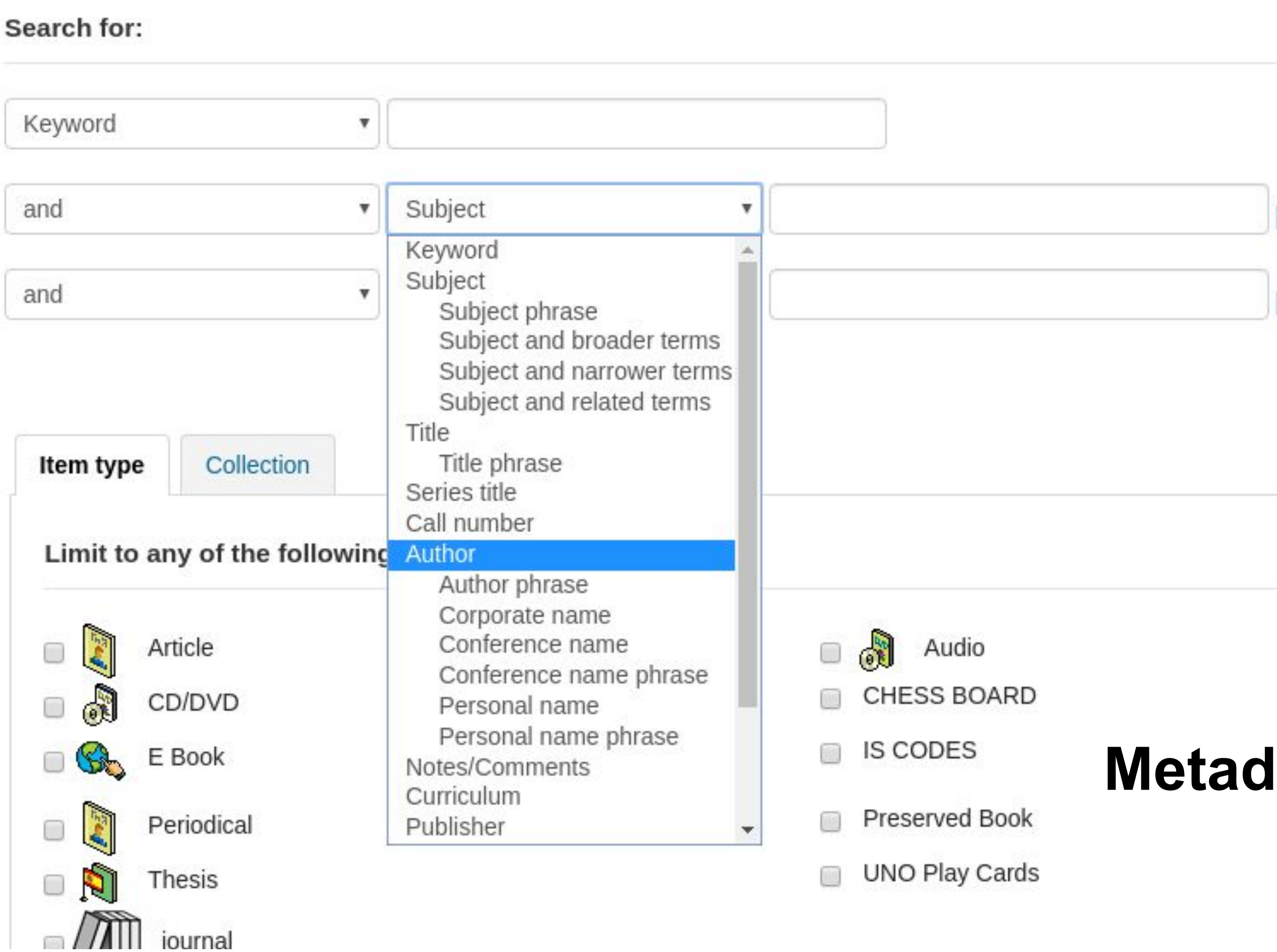
- When a system produces a ranked result set, **large result sets are not an issue**.
  - Indeed, the size of the result set is not an issue.
  - We just show the top k (≈ 10) results.
  - We don’t overwhelm the user.

Premise: the ranking algorithm works

Parametric and Zone Indexes

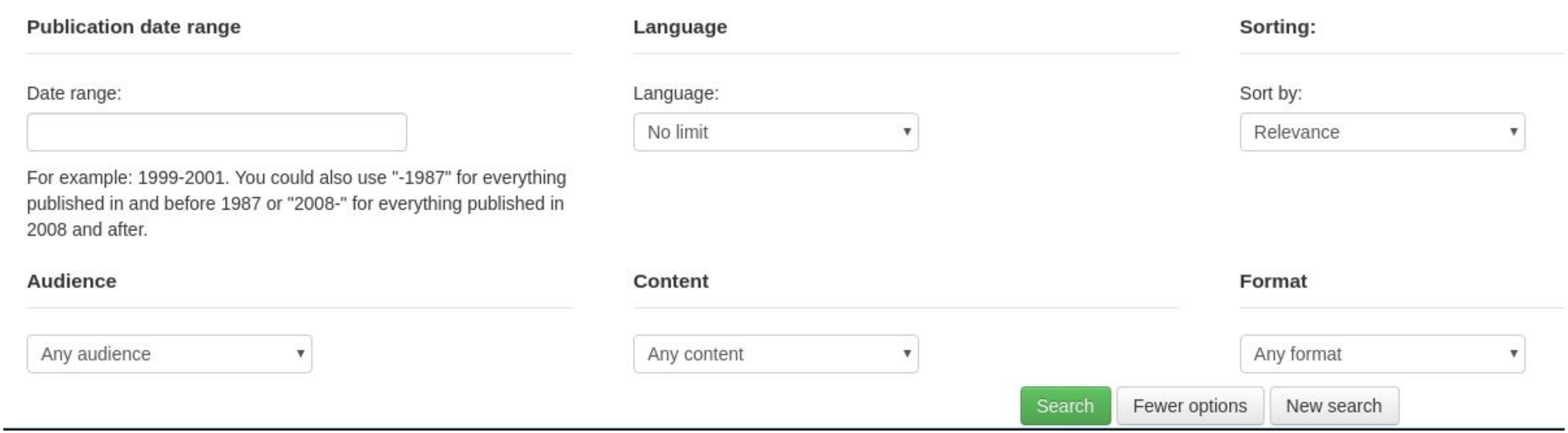
A special case: Boolean retrieval with document ranking

BITS Library: Advanced search



Metadata

BITS Library: Advanced search



Metadata

Parametric and Zone indexes

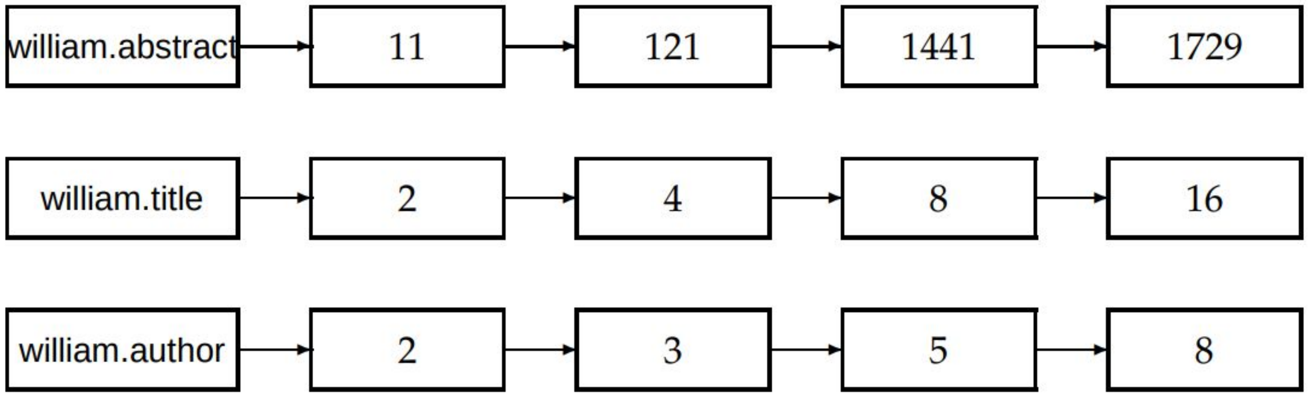
- A kind of boolean retrieval with multiple indexes and document ranking.
- **Parametric index:** An index (parametric index) **for each field** in the **metadata**. Example: date of creation, published year, language, genre.
- **Zone index:** Similar to field, expect the content of a zone can be arbitrary free text. Example: Document titles, abstract.

Parametric and Zone indexes: Query examples

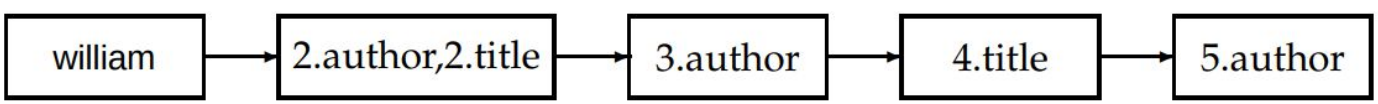
- Document **authored** by **William Shakespeare** in **1601**, containing the **phrase** **alas poor Yorick**.

- Find documents with **merchant** in the **title**, and **william** in the **author** list, and phrase **gentle rain** in the **body**.

Zone Indexes Implementations



Zones are encoded as an extension of dictionary entries.



Zone index in which zone is encoded in the posting rather than the dictionary.

Weighted zone scoring or Ranked Boolean Retrieval

- Query: Normal boolean query
- Assign a (query, document) pair a score, by computing a linear combination of zone score.
- For example:
    - There are three zones: **title**, **abstract**, **body**
    - The weights for these zones are: 0.45, 0.3, 0.25
    - **All the zones have the query term.**
        - Score will be: 0.45 * 1 + 0.3 * 1 + 0.25 * 1 = 1
    - **Only abstract and body has the query term:**
        - Score will be 0.45 * 0 + 0.3 * 1 + 0.25 * 1 =   0.55
- The weights of zones should sum up to 1.

Parametric and Zone Indexes: Summary

- A special case of ranked retrieval.
- Document metadata is added for advanced retrieval.
- Zone based weightage of terms.

Ranked Retrieval in general

Ranked Retrieval in general

- **Free text queries:** Rather than a query language of **operators** and **expressions**, the user's query is just one or more words in a natural language.
- **Query - Document Matching:** We need a way of assigning a score to a query-document pair.

Scoring as the basis of ranked retrieval

- We wish to return in **order** the documents **most likely to be useful** to the searcher.
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in [0, 1] – to each document.
- This score measures how well document and query "match".
- Sort the documents based on the score.

Jaccard coefficient

- A commonly used measure of overlap of two sets A and B.
- Jaccard(A,B) = $|A \cap B| / |A \cup B|$
- Jaccard(A,A) = 1
- Jaccard(A,B) = 0 if $A \cap B = 0$
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1.
- We saw that in the context of k-gram overlap between two words.

Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- **Query:** ides of march
- **Document 1:** caesar died in march
- **Document 2:** the long march
- jaccard(Query, Document1) =
- jaccard(Query, Document2) =

Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- **Query:** ides of march
- **Document 1:** caesar died in march
- **Document 2:** the long march
- jaccard(Query, Document1) = **1/6**
- jaccard(Query, Document2) = **1/5**

Issues with Jaccard for scoring

1. Match score decreases as document length grows.
   - We need a more sophisticated way of **normalizing for length**.
2. It doesn't consider term frequency (how many times a term occurs in a document)
   - For jaccard coefficient documents are **set** of words not **bag** of words.
3. Rare terms in a collection are more informative than frequent terms - Jaccard doesn't consider this information.

innovate achieve lead

Term Frequency

Recap (Lecture 2): Term Document Incidence Matrix

	Document 1	Document 2	Document 3	Document 4	...	Document N
Word 1	1	0	1	0		1
Word 2	0	1	0	0		0
Word 3	0	0	0	0		0
Word 4	0	1	1	0		0
Word 5	1	0	1	1		1
Word 6	1	1	0	1		1
:						
Word M	0	0	1	0		1

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$.

Term-Document Count Matrix

	Document 1	Document 2	Document 3	Document 4	...	Document N
Word 1	**25**	**5**	**30**	**0**		**43**
Word 2	**0**	**1**	**12**	**0**		**1**
Word 3	**12**	**0**	**21**	**0**		**0**
Word 4	**1**	**12**	**13**	**0**		**0**
Word 5	**13**	**45**	**4**	**1**		**0**
Word 6	**0**	**6**	**7**	**2**		**1**
:						
Word M	**0**	**12**	**1**	**23**		**3**

Each document is represented by a ~~binary~~ count vector $\in \mathbb{N}^{|V|}$.

Bag of words model

- Vector representation **doesn't consider** the **ordering** of words in a document.
- "John is quicker than Mary" and "Mary is quicker than John" have the **same vectors**.
- This is called the **bag of words** model.
- In a sense, this is a step back: the positional index was able to distinguish these two documents.
- We will look at "recovering" positional information later in this course.
- For now: bag of words model.

Term Frequency

- The term frequency $\mathbf{tf}_{\mathbf{t,d}}$ of term **t** in document **d** is defined as the number of times that **t** occurs in **d**.
  - Note: Frequency means count in IR.
- We want to use term frequency when computing query-document match scores - but how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term **is more relevant** than a document with 1 occurrence of the term.
  - But **not 10 times** more relevant
- Relevance does not increase proportionally with term frequency.

Instead of raw frequency: Log frequency weighting

- The log frequency weight of term **t** in **d** is:

$$w_{t,d} = \begin{cases} 1 + \log10(\mathrm{tf}_{t,d}) & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms **t** in both **q** and **d**:
- **score(q, d)** $= \sum_{t \in q \cap d} (1 + \log10(\mathrm{tf}_{t,d}))$
- The score is 0 if none of the query terms is present in the document.

Term Frequency: Issues

- All terms within a query are **considered equally**.
- For example for the query: **high arachnocentric**

Document Frequency

- **Rare terms** – in the whole collection - are **more informative** than frequent terms.
    - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., arachnocentric).
- A document containing this term is very likely to be relevant to the (information need originating the) query arachnocentric.
- We want a high weight for rare terms like arachnocentric.

Document Frequency, cont’d

- Consider a **query** term that is frequent in the collection (e.g., **high**, **increase**, **line**).
- A document containing such a term is more likely to be relevant than a document that doesn’t.
- **But** consider a query containing two terms – e.g.: **high arachnocentric**.
- For a **frequent** term in a document, s.a., high, we want a **positive weight but lower than for terms that are rare** in the collection, s.a., arachnocentric.
- We will use **document frequency (df)** to capture this.

Inverse Document Frequency (idf)

- **df_t** is the document frequency, the number of documents that **t** occurs in.
  - **df_t** is an inverse measure of the informativeness of term **t** (the smaller the better)
  - **df_t <= N**
- We define the **idf** (inverse document frequency) of **t** by:
  - **$idf_t = \log10(N/df_t)$**
  - We use **$\log10(N/df_t)$** instead of **N/df_t** to "dampen" the effect of **idf**.
  - **Note: idf** is independent of the document.

idf example, suppose N = 1 million

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

$$\mathbf{idf_t = \log10(\,N/df_t\,)}$$

There is one **idf** value for each term **t** in a collection.

Collection vs. Document frequency

- The **collection frequency** of **t** is the number of occurrences of **t** in the collection, counting multiple occurrences in the same document

Collection vs. Document frequency

- The **collection frequency** of **t** is the number of occurrences of **t** in the collection, counting multiple occurrences in the same document
- For example:

Word	Collection Frequency	Document Frequency
insurance	10440	3997
try	10422	8760

- Why these numbers?
- Which word is a better search term (and should get a higher weight)?
- This example suggests that **df** (and **idf**) is better for weighting than **cf** (and "**icf**").

Effect of idf on ranking

- **idf** affects the ranking of documents for queries with **at least two terms**.
- For example, in the query "**arachnocentric line**", **idf** weighting **increases** the relative weight of **arachnocentric** and **decreases** the relative weight of **line**.

tf-idf weighting

- The **tf-idf** weight of a term is the product of its **tf** weight and its **idf** weight:
  - $\mathbf{w_{t,d} = (1 + \log10(tf_{t,d})) * \log(N/df_t)}$
- Best known weighting scheme in information retrieval.
- Note: the "-" in tf-idf is a hyphen, not a minus sign!
- Alternative names: tf.idf, tf x idf
- **Increases with the number of occurrences** within a document.
- **Increases with the rarity** of the term in the collection

Score for a document given a query

- **score(q, d) = $\sum_{t \in q \cap d}$ $tf.idf_{t,d}$**

- There are several variants …

The Vector Space Model

The Vector Space Model: Worked Example

Worked Example: Corpus

Document 1: mary had a little lamb little lamb little lamb mary had a little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its fleece was white as snow mary had a little lamb

Worked Example: Axis

- The dimensionality of the vector space will be the number of unique terms in the vocabulary.
- In the example corpus the unique terms are **16**.
- Thus, every document will be a 16 dimension vector.

	Doc 1	Doc 2	Doc 3
a			
as			
fleece			
had			
its			
know			
lamb			
little			
loves			
mary			
snow			
the			
was			
white			
why			
you			

Document 1: mary had a little lamb little lamb little lamb mary had a little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its fleece was white as snow mary had a little lamb

Using Term-Frequency only

	Doc 1	Doc 2	Doc 3
a	2	0	1
as	0	0	1
fleece	0	0	1
had	2	0	1
its			
know			
lamb			
little			
loves			
mary			
snow			
the			
was			
white			
why			
you			

Document 1: mary **had a** little lamb little lamb little lamb mary **had a** little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its **fleece** was white **as** snow mary **had** **a** little lamb

TF with IDF

	Doc 1	Doc 2	Doc 3
a	2 * log(3/2)	0	1 * log(3/2)
as	0	0	1 * log(3/1)
fleece	0	0	1 * log(3/1)
had	2 * log(3/2)	0	1 * log(3/2)
its			
know			
lamb			
little			
loves			
mary			
snow			
the			
was			
white			
why			
you			

Document 1: mary **had a** little lamb little lamb little lamb mary **had a** little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its **fleece** was white **as** snow mary **had a** little lamb

TF with IDF

	Doc 1	Doc 2	Doc 3
a	0.35	0	0.18
as	0	0	0.48
fleece	0	0	0.48
had	0.35	0	0.18
its			
know			
lamb			
little			
loves			
mary			
snow			
the			
was			
white			
why			
you			

Document 1: mary **had a** little lamb little lamb little lamb mary **had a** little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its **fleece** was white **as** snow mary **had a** little lamb

Query as a Vector

	Query 1	Query 2
a		
as		
fleece		
had		
its		
know		
lamb		
little		
loves		
mary		
snow		
the		
was		
white		
why		
you		

Query 1: little lamb

Query 2: mary snow

Query as a Vector: Unweighted

	Query 1	Query 2
a	0	0
as	0	0
fleece	0	0
had	0	0
its	0	0
know	0	0
lamb	**1**	0
little	**1**	0
loves	0	0
mary	0	**1**
snow	0	**1**
the	0	0
was	0	0
white	0	0
why	0	0
you	0	0

Query 1: little lamb

Query 2: mary snow

Query as a Vector: Weighted, idf weights

	Query 1	Query 2
a	0	0
as	0	0
fleece	0	0
had	0	0
its	0	0
know	0	0
lamb	**0**	0
little	**0.18**	0
loves	0	0
mary	0	**0**
snow	0	**0.48**
the	0	0
was	0	0
white	0	0
why	0	0
you	0	0

Query 1: little lamb

Query 2: mary snow

Matrix will be sparse: Build Inverted Index

Matrix will be sparse: Build Inverted Index

a $\rightarrow$

Doc 1: 0.35	Doc 3: 0.18

Document 1: mary had a little lamb little lamb little lamb mary had a little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its fleece was white as snow mary had a little lamb

Inverted Index: Do not store floats

Document 1: mary had a little lamb little lamb little lamb mary had a little lamb

Document 2: why mary loves the lamb you know lamb you know lamb

Document 3: its fleece was white as snow mary had a little lamb

- Floating point numbers can take 4 bytes.
- They are difficult to compress.

Summary: Documents as vectors

- Each document is now represented as a real-valued vector of **tf-idf** weights $\in R^{|V|}$.
- So we have a |V|-dimensional real-valued vector space.
- **Terms** are **axes** of the space.
- **Documents** are **points** or **vectors** in this space.
- Very high-dimensional: tens of millions of dimensions when you apply this to web search engines.
- Each vector is very sparse - most entries are zero.

Summary: Queries as vectors

- **Key idea 1:** Do the same for queries: represent them as vectors in the space.
- **Key idea 2:** Rank documents according to their proximity to the query in this space.
- proximity = similarity of vectors
- proximity ≈ inverse of distance
- **Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.**
- Instead: rank more relevant documents higher than less relevant documents.

Similarity Between Documents/Queries

- Difference between two vectors?
  - No, two documents might have **similar content** but they have a **significant vector difference**.
  - Consider one document is a double copy of another document.
  - The **relative frequencies of the terms in both the documents are same**, however, the **absolute frequencies are different**.
- To compensate for the effect of document length, the similarity measure used is **cosine similarity**.

Cosine Similarity

cosine(query,document)

Dot product

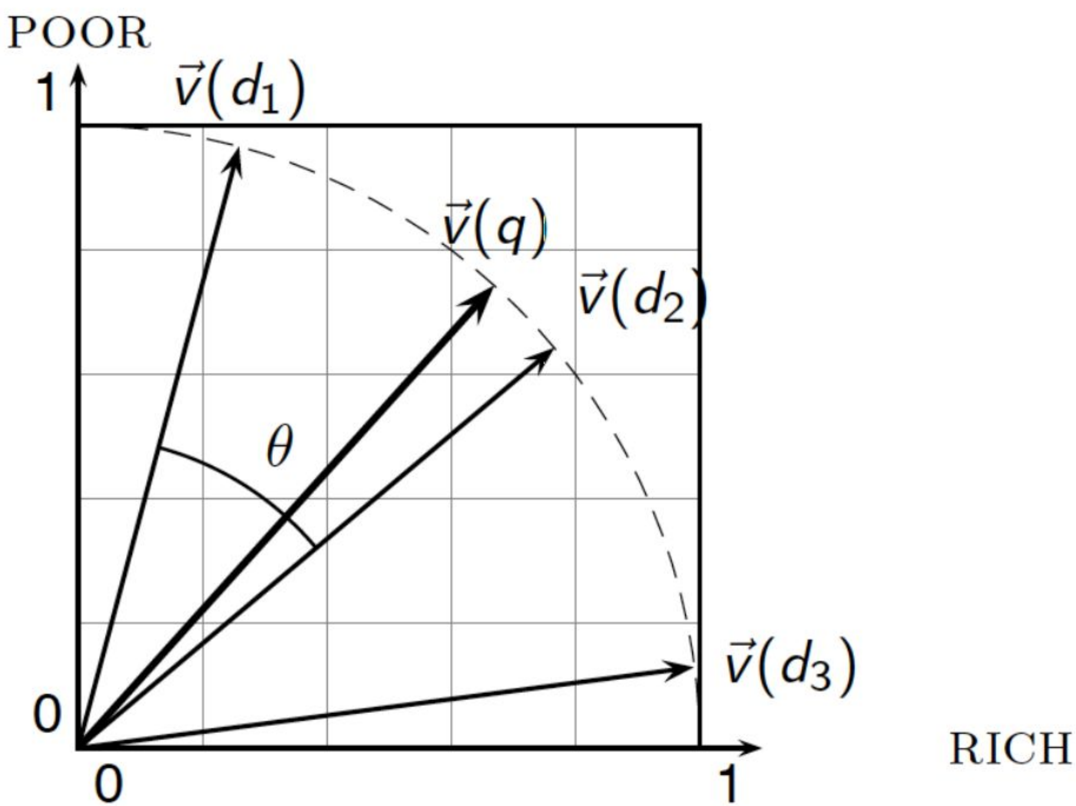
Unit vectors

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

q_i is the weight of term *i* in the query
d_i is the weight of term *i* in the document

$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

Cosine Similarity Illustrated



Computing Vector Scores

$\textsc{CosineScore}(q)$

$\textit{float Scores}[N] = 0$
$\textit{float Length}[N]$
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 $\textit{Scores}[d] += \mathrm{w}_{t,d} \times \mathrm{w}_{t,q}$
Read the array $\textit{Length}$
for each d
do $\textit{Scores}[d] = \textit{Scores}[d]/\textit{Length}[d]$
return Top K components of $\textit{Scores}[]$

Extracting the top K items can be done with a priority queue (e.g., a heap)

tf-idf weighting has many variants

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - \mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/\mathit{CharLength}^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(\mathrm{tf}_{t,d})}{1 + \log(\mathrm{ave}_{t \in d}(\mathrm{tf}_{t,d}))}$				

Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents.
- SMART Notation: denotes the combination in use in an engine, with the notation ddd.qqq, using the acronyms from the previous table.
- A very standard weighting scheme is: **lnc.ltc**
- **Document:** logarithmic tf (**l** as first character), **n**o idf and **c**osine normalization.
- **Query:** logarithmic tf (**l** in leftmost column), idf (**t** in second column), **c**osine normalization.

tf-idf example: lnc.ltc

Document: *car insurance auto insurance*
Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	n'lize	tf-raw	tf-wt	wt	n'lize	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Doc length $=\sqrt{1^2+0^2+1^2+1.3^2} \approx 1.92$

Score = 0+0+0.27+0.53 = 0.8

Summary – vector space ranking

- Represent the query as a weighted tf-idf vector.
- Represent each document as a weighted tf-idf vector.
- Compute the cosine similarity score for the query vector and each document vector.
- Rank documents with respect to the query by score.
- Return the top K (e.g., K = 10) to the user

Issues with the simple score computation algorithm

High-Latency

- Score computation is a large (10s of %) fraction of the CPU work on a query.
- Generally, we have a tight budget on latency (say, 250ms).
- CPU provisioning doesn’t permit exhaustively scoring every on every query.
- We’ll look at ways of cutting CPU usage for scoring, without compromising the quality of results (much)
- Basic idea: avoid scoring docs that won’t make it into the top K.

Safe vs non-safe ranking

- **Safe ranking:** It is guaranteed that the top K documents returned by an algorithm are the K absolute highest scoring documents.
- **Non-safe:** The top K documents returned are closer to the K absolute highest scoring documents.

- Is it ok to be non-safe?

Ranking function is only a proxy

- User has a task and a query formulation.
- Ranking function matches docs to query.
- Thus the ranking function is anyway a **proxy for user happiness**.
- **If we get a list of K docs close to the top K by the ranking function measure, should be ok.**

Generic Approach

- We somehow (in a cheap way) reduce the document set from N to A, where:
  - **K < |A| << |N|**
- Think of **A** as pruning non-contenders.
- The same approach is also used for other (non-cosine) scoring functions.
- Will look at several schemes following this approach.

Index elimination

Index elimination

- The basic cosine similarity score computation algorithms consider documents with at least one query term mentioned.

Index elimination

- The basic cosine similarity score computation algorithms consider documents with at least one query term mentioned.
- Take this further:
  - Only consider high-idf query terms.
  - Only consider docs containing many query terms.

High idf query terms only

High idf query terms only

- For a query such as **catcher in the rye**
- Only accumulate scores from **catcher** and **rye**.

High idf query terms only

- For a query such as **catcher in the rye**
- Only accumulate scores from **catcher** and **rye**.
- Intuition: **in** and **the** contribute little to the scores and so don’t alter rank-ordering much.
- Benefit:
    - Postings of low-idf terms have many docs → these (many) docs get eliminated from set **A** of contenders

Docs containing many query terms

Docs containing many query terms

- Any document with at least one query term is a candidate for the top K output list.
- For multi-term queries, only compute scores for docs containing several of the query terms
- Say, at least 3 out of 4
- Easy to implement in postings traversal

Champion List

Champion List

- For every term (**t**), store a list of **r** documents that have the highest score for term **t**.
  - The score can be tf score.
  - **r** is fixed at the index creation time, thus it's possible that r < K.
- The set of **r** documents are called the <u>champion list</u> for term **t**.

Champion List

- For every term (**t**), store a list of **r** documents that have the highest score for term **t**.
    - The score can be tf score.
    - **r** is fixed at the index creation time, thus it's possible that r < K.
- The set of **r** documents are called the champion list for term **t**.
- Now, for a query, create a set of documents **A** from the champion list of all the terms in the query.
- Compute cosine similarity with these documents.

Static quality scores

Static quality scores

- Each document has a score assigned, **g(d)**, which is independent of the query.

Static quality scores

- Each document has a score assigned, **g(d)**, which is independent of the query.
- Consider this score as a **authoritative** score.
- Example of authority signals:
    - Wikipedia among websites.
    - Articles in certain newspapers.
    - A paper with many citations.
    - Many bitlys, likes, or bookmarks.
    - Pagerank

Quantitative

Static quality scores, cont'd

- The net score of the document can be the **g(d)** + query dependent score.
- NetScore(q,d) = g(d) + cosine(q,d)
  - Other than linear combination, any other combination of the above can be used.
- Now find top K documents based on net score.

Top K by net score – fast methods

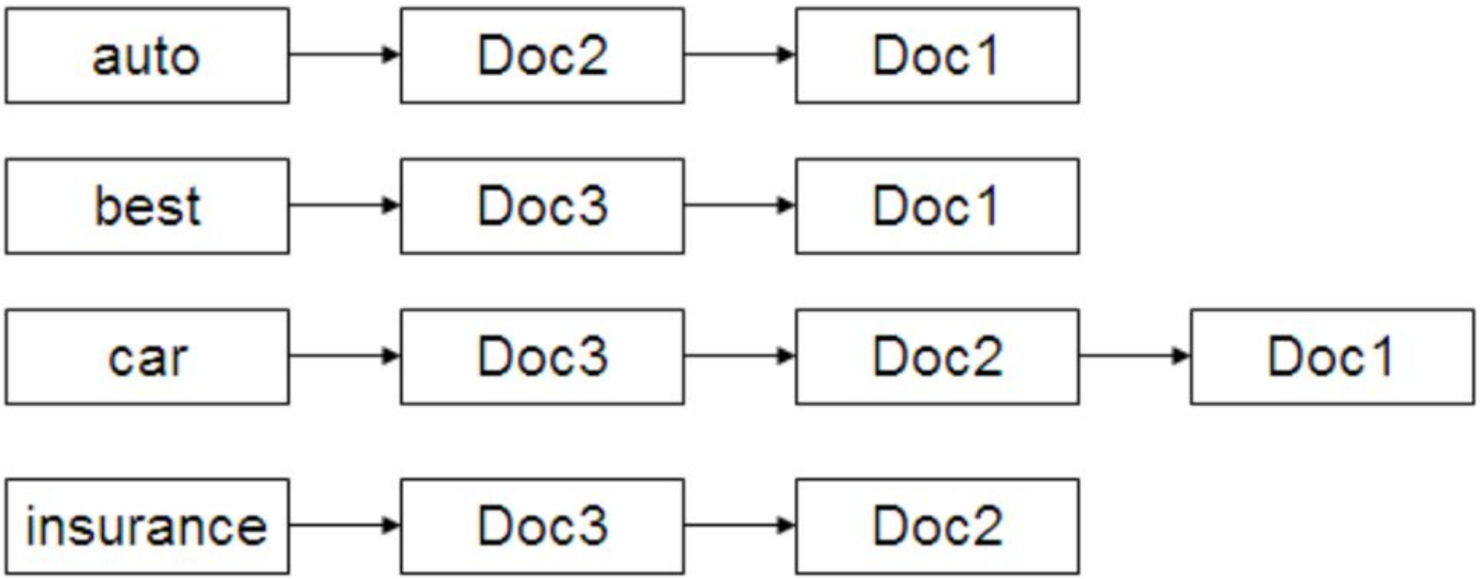
Top K by net score – fast methods

- If **g(d)** is globally computed, we can order the documents in the posting list with **g(d)**.
- Under g(d)-ordering, top-scoring documents likely to appear early in postings traversal.

Top K by net score – fast methods

- If **g(d)** is globally computed, we can order the documents in the posting list with **g(d)**.
- Under g(d)-ordering, top-scoring documents likely to appear early in postings traversal.
- In time-bound applications (say, we have to return whatever search results we can in 50 ms), this allows us to stop postings traversal early
  - Short of computing scores for all documents in postings

Ordering of posting list by g(d) example



► **Figure 7.2** A static quality-ordered index. In this example we assume that Doc1, Doc2 and Doc3 respectively have static quality scores $g(1) = 0.25, g(2) = 0.5, g(3) = 1$.

Champion lists in g(d) ordering

Champion lists in g(d) ordering

- Can combine champion lists with g(d)-ordering
- Maintain for each term a champion list of the **r** docs with highest **g(d)** + **tf**
- Seek top-K results from only the documents in these champion lists.

Impact ordering

Impact ordering

- Order the posting list of a terms by document impact for that term.
- Now: not all postings are in common order!

Impact ordering

- Order the posting list of a terms by document impact for that term.
- Now: not all postings in a common order!
- While computing cosine similarity as per Figure 6.14, few heuristics can be applied:
  - After travelling of **x** elements posting list, end inner for loop.
  - The outer for loop traversal should be in decreasing order of **idf**.
  - The inner for loop can be terminated when there is only very small increase in overall score.

Figure 6.14

```
CosineScore(q)
float Scores[N] = 0
float Length[N]
for each query term t
do calculate w_{t,q} and fetch postings list for t
    for each pair(d, tf_{t,d}) in postings list
    do Scores[d] += w_{t,d} × w_{t,q}
Read the array Length
for each d
do Scores[d] = Scores[d]/Length[d]
return Top K components of Scores[]
```

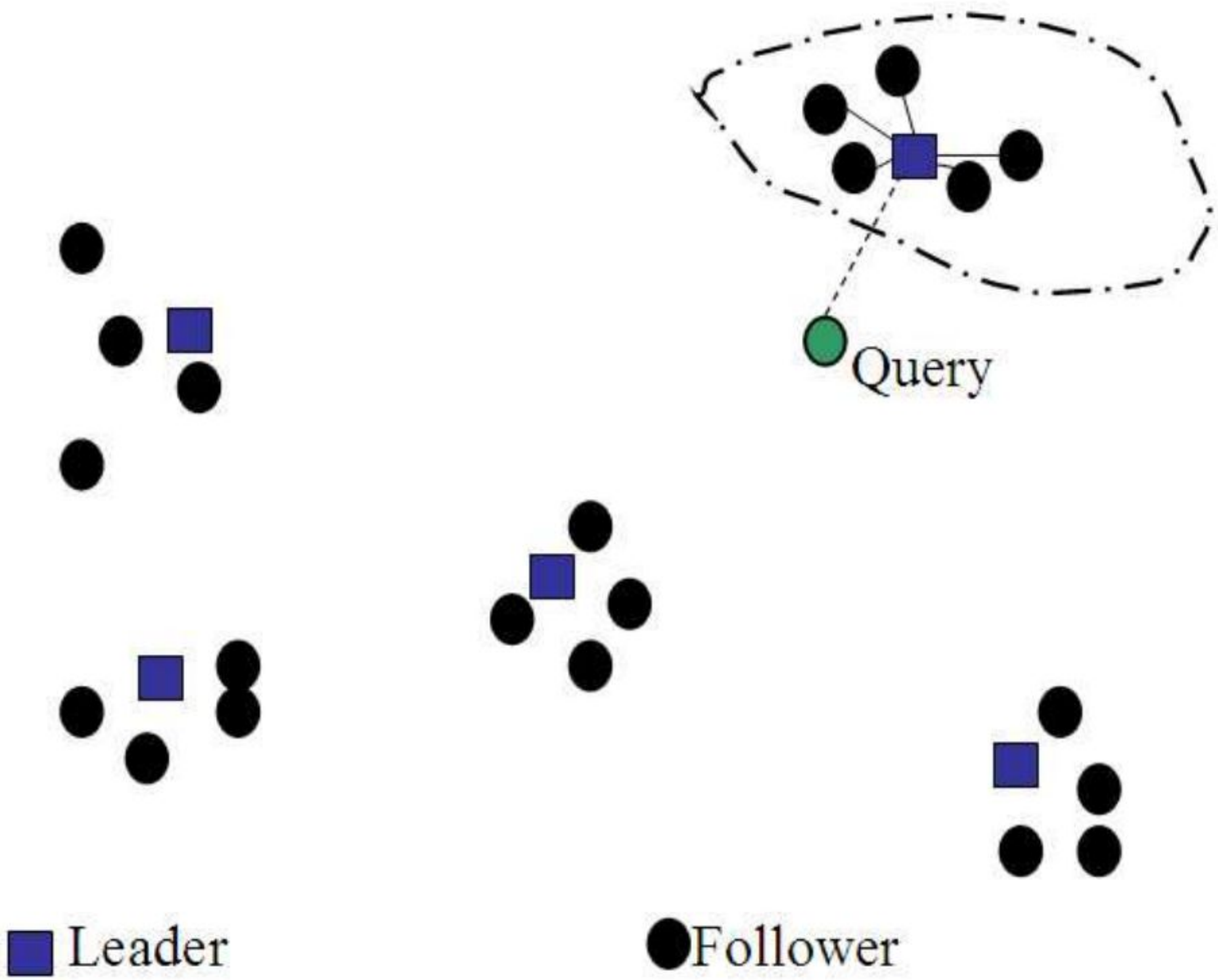
Extracting the top K items can be done with a priority queue (e.g., a heap)

Cluster Pruning

Cluster Pruning

- Cluster the documents and assign a leader to each cluster.
- For a query, get the best matching leader.
- Compute scores with all the followers of that leader.

Cluster Pruning

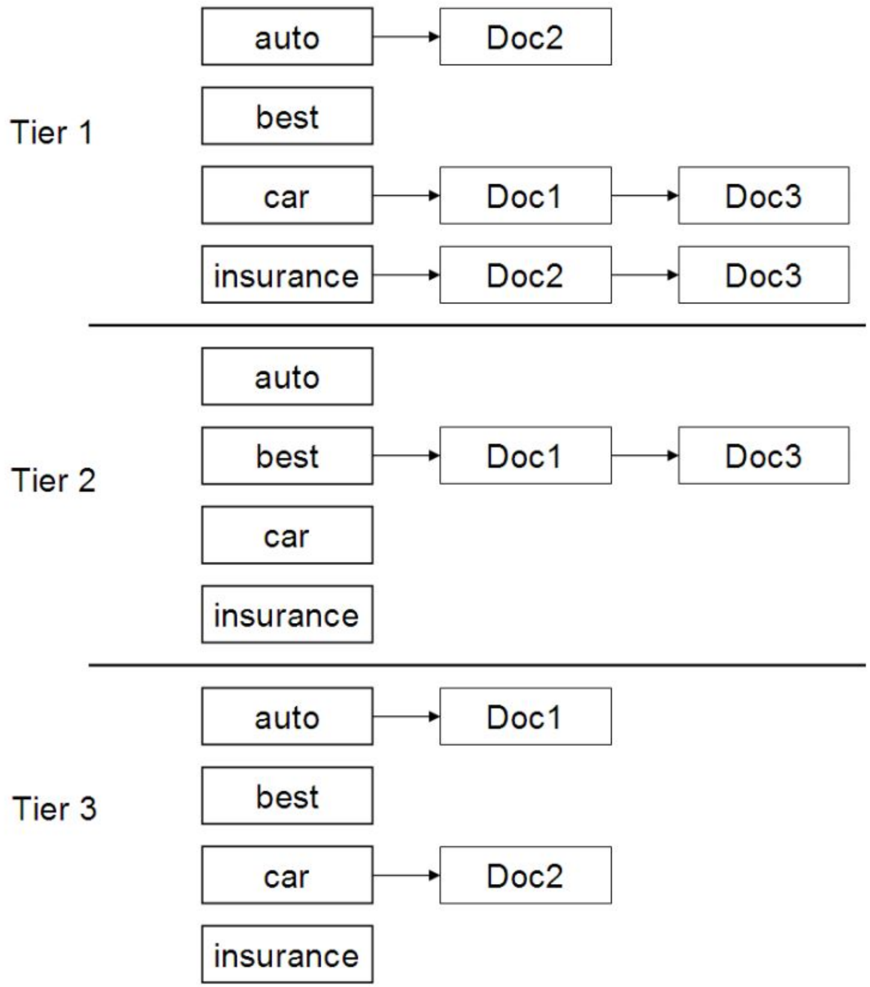


Tiered Indexes

Tiered Indexes

- Break postings up into a hierarchy of lists.
  - Most important
  - …
  - Least important
- Can be done by g(d) or another measure.
- Inverted index thus broken up into tiers of decreasing importance.
- At query time use top tier unless it fails to yield K docs.
  - If so drop to lower tiers

Tiered Indexes: Example



Finishing touches for a complete scoring system

Query term proximity

- **Free text queries:** just a set of terms typed into the query box – common on the web.
- Users prefer docs in which query terms occur within close proximity of each other.

Query term proximity

- **Free text queries:** just a set of terms typed into the query box – common on the web.
- Users prefer docs in which query terms occur within close proximity of each other.
- Let **w** be the smallest window in a doc containing all query terms, e.g.,
- For the query **strained mercy** the smallest window in the doc *The quality of mercy is not strained* is **4** (words)
- Would like scoring function to take this into account
  - how?

Query Parser

Query Parser

- Free text query from user may in fact spawn one or more queries to the indexes, e.g., query **rising interest rates**
  - Run the query as a phrase query.
  - If <K docs contain the phrase rising interest rates, run the two phrase queries **rising interest** and **interest rates**
  - If we still have <K docs, run the vector space query **rising interest rates**.
  - Rank matching docs by vector space scoring.
- This sequence is issued by a **query parser**

Aggregate scores

- We have seen that score functions can combine cosine, static quality, proximity, etc.
- How do we know the best combination?
- Some applications – expert-tuned
- Increasingly common: machine-learned

A Complete Search System