Computing scores in a complete search system

- Previous chapter - theory underlying term weighting in documents for the purposes of scoring, leading up to vector space models and the basic cosine scoring algorithm
- Agenda
  - Heuristics for speeding up scoring and ranking
  - Components needed for a complete search engine
  - Outline of a complete search engine
  - How vector space model for free text queries interacts with common query operators.

Efficient scoring and ranking

- For the purpose of ranking the documents, we are really interested in the relative (rather than absolute) scores of the documents in the collection.
- It suffices to compute the cosine similarity from each document unit vector

$$\vec{v}(d) \text{ to } \vec{V}(q)$$

(in which all non-zero components of the query vector are set to 1), rather than to the unit vector . $\vec{v}(q)$.

- For any two documents *d*1, *d*2

$$\vec{V}(q) \cdot \vec{v}(d_1) > \vec{V}(q) \cdot \vec{v}(d_2) \Leftrightarrow \vec{v}(q) \cdot \vec{v}(d_1) > \vec{v}(q) \cdot \vec{v}(d_2).$$

Efficient Scoring and Ranking

Term	$\vec{v}(d_1)$ SaS	$\vec{v}(d_2)$ PaP	$\vec{v}(d_3)$ WH	$\vec{V}(q)$ QV	$\vec{v}(q)$ QV
affection	0.996	0.993	0.847	0	0
Jealous	0.087	0.120	0.466	1	0.707
gossip	0.017	0	0.254	1	0.707

Length of QV $= \sqrt{0+1^2+1^2} = 1.414$

SaS

$\vec{v}(d_1) \cdot \vec{v}(q) = 0.996 \times 0 + 0.087 \times 0.707 + 0.017 \times 0.707$

$= 0 + 0.061 + 0.012 = 0.073 \rightarrow (3)$

PaP

$\vec{v}(d_2) \cdot \vec{v}(q) = 0.993 \times 0 + 0.120 \times 0.707 + 0 \times 0.707$

$= 0 + 0.085 + 0 = 0.085 \rightarrow (2)$

WH

$\vec{v}(d_3) \cdot \vec{v}(q) = 0.847 \times 0 + 0.466 \times 0.707 + 0.254 \times 0.707$

$= 0 + 0.33 + 0.18 = 0.508 \rightarrow (1)$

SaS

$\vec{v}(d_1) \cdot \vec{V}(q) = 0.087 + 0.017 = 0.104 / 1.414$

$= 0.073 \rightarrow (3)$

PaP

$\vec{v}(d_2) \cdot \vec{V}(q) = 0.120 / 1.414 = 0.085 \rightarrow (2)$

WH $\vec{v}(d_3) \cdot \vec{V}(q) = 0.466 + 0.254 = 0.720 / 1.414 = 0.50$

$\rightarrow (1)$

Efficient scoring and ranking

```
COSINESCORE(q)
1  float Scores[N] = 0
2  Initialize Length[N]
3  for each query term t
4  do calculate w_{t,q} and fetch postings list for t
5     for each pair(d, tf_{t,d}) in postings list
6     do Scores[d] += wf_{t,d} × w_{t,q}
7  Read the array Length[d]
8  for each d
9  do Scores[d] = Scores[d]/Length[d]
10 return Top K components of Scores[]
```

▶ **Figure 6.14** The basic algorithm for computing vector space scores.

```
FASTCOSINESCORE(q)
1  float Scores[N] = 0
2  for each d
3  do Initialize Length[d] to the length of doc d
4  for each query term t
5  do calculate w_{t,q} and fetch postings list for t
6     for each pair(d, tf_{t,d}) in postings list
7     do add wf_{t,d} to Scores[d]
8  Read the array Length[d]
9  for each d
10 do Divide Scores[d] by Length[d]
11 return Top K components of Scores[]
```

▶ **Figure 7.1** A faster algorithm for vector space scores.

Inexact top *K* document retrieval

- We have focused on retrieving precisely the *K* highest-scoring documents for a query.
- Does the exact top-*k* matter?
  - How much are we sure that the 101st ranked document is less important than the 100th ranked?
  - All the scores are simplified models for what information may be associated with the documents
- Suffices to retrieve *k* documents with
  - Many of them from the exact top-*k*
  - The others having score close to the top-*k*

- We now consider schemes by which we produce *K* documents that are *likely* to be among the *K* highest scoring documents for a query
- This will lower the cost of computing the *K* documents we output.

Inexact top K document retrieval

- We now consider a series of ideas designed to eliminate a large number of documents without computing their cosine scores.
- Notice that the similarity score is a proxy of the relevance of a document to a query, so we already have some "approximation".
- The heuristics have the following two-step scheme:
  - Find a set A of documents that are contenders, where $K < |A| \ll N$ (size of the posting list). A does not necessarily contain the K top-scoring documents for the query, but is likely to have many documents with scores near those of the top K.
  - Return the K highest ranked documents in .

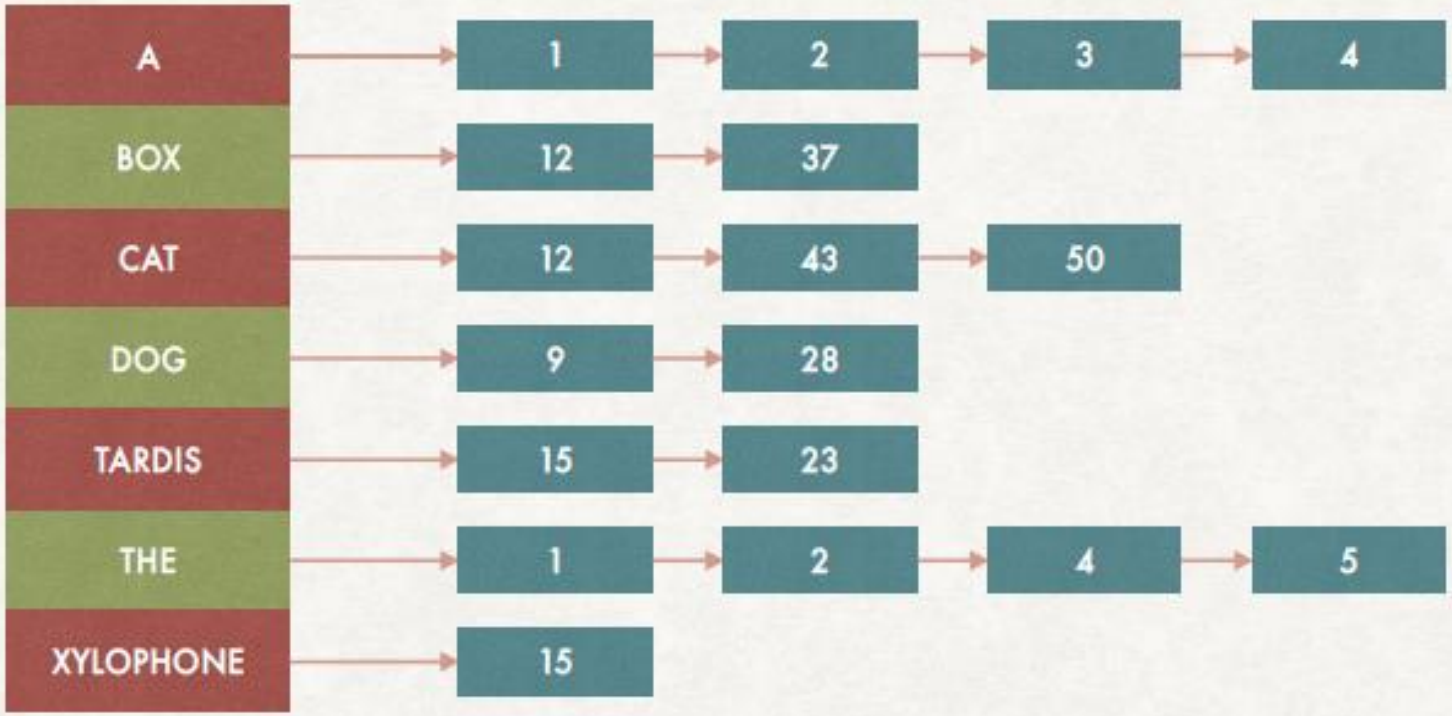
Index Elimination

- For a multi-term query q, some systems consider documents containing at least one of the query terms. We can take this a step further using additional heuristics:
  - We only consider documents containing terms whose idf exceeds a preset threshold.
  - Thus, in the postings traversal, we only traverse the postings for terms with high idf.
  - This has a fairly significant benefit: the postings lists of low-idf terms (stop words) are generally long; with these removed from contention, the set of documents for which we compute cosines is greatly reduced.

- We only consider documents that contain many (and as a special case, all) of the query terms.
- This can be accomplished during the postings traversal; we only compute scores for documents containing all (or many) of the query terms.
- A danger of this scheme is that by requiring all (or even many) query terms to be present in a document before considering it for cosine computation, we may end up with fewer than K candidate documents in the output.

INDEX ELIMINATION

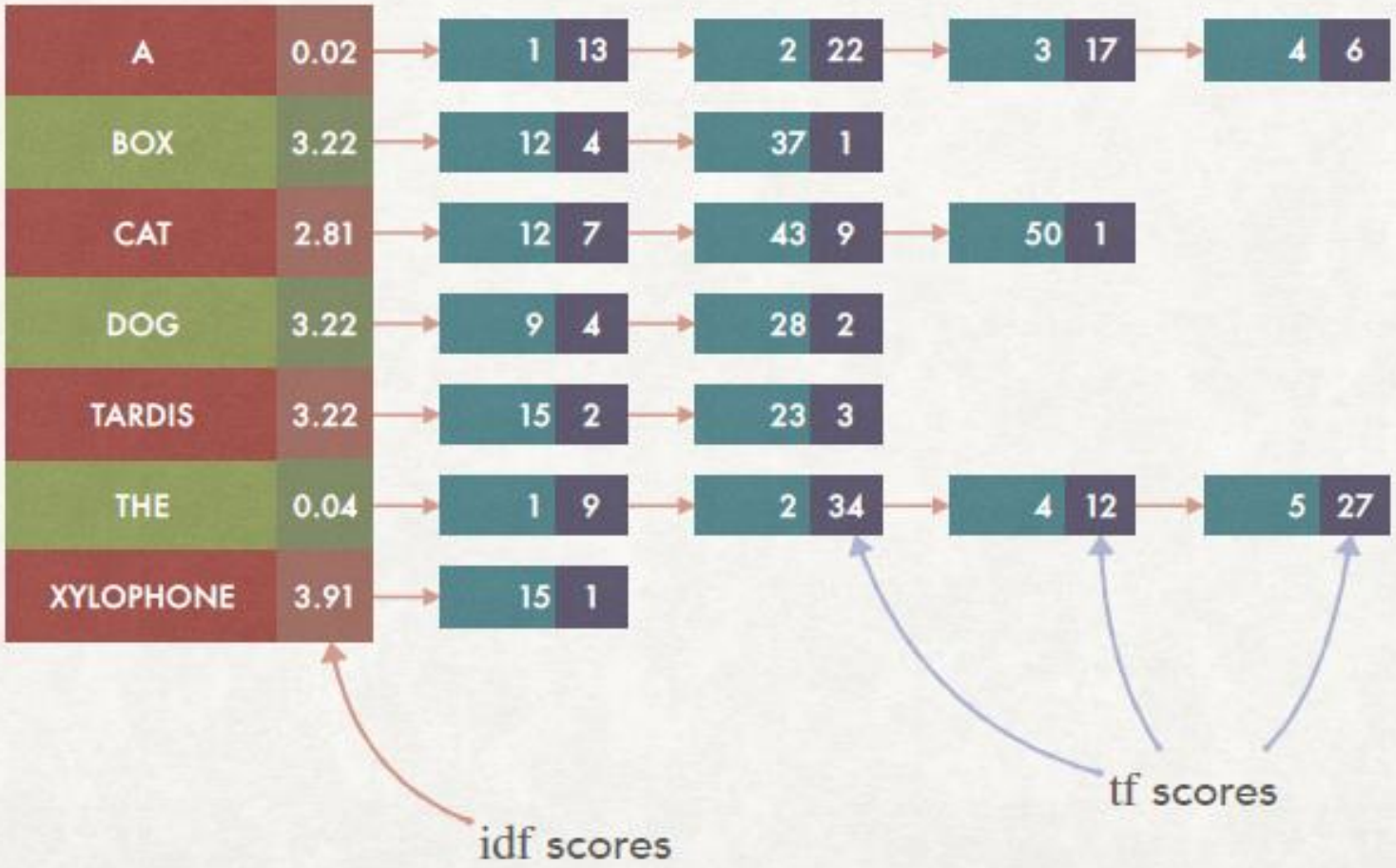
HOW TO IGNORE SOME TERMS



standard inverted index

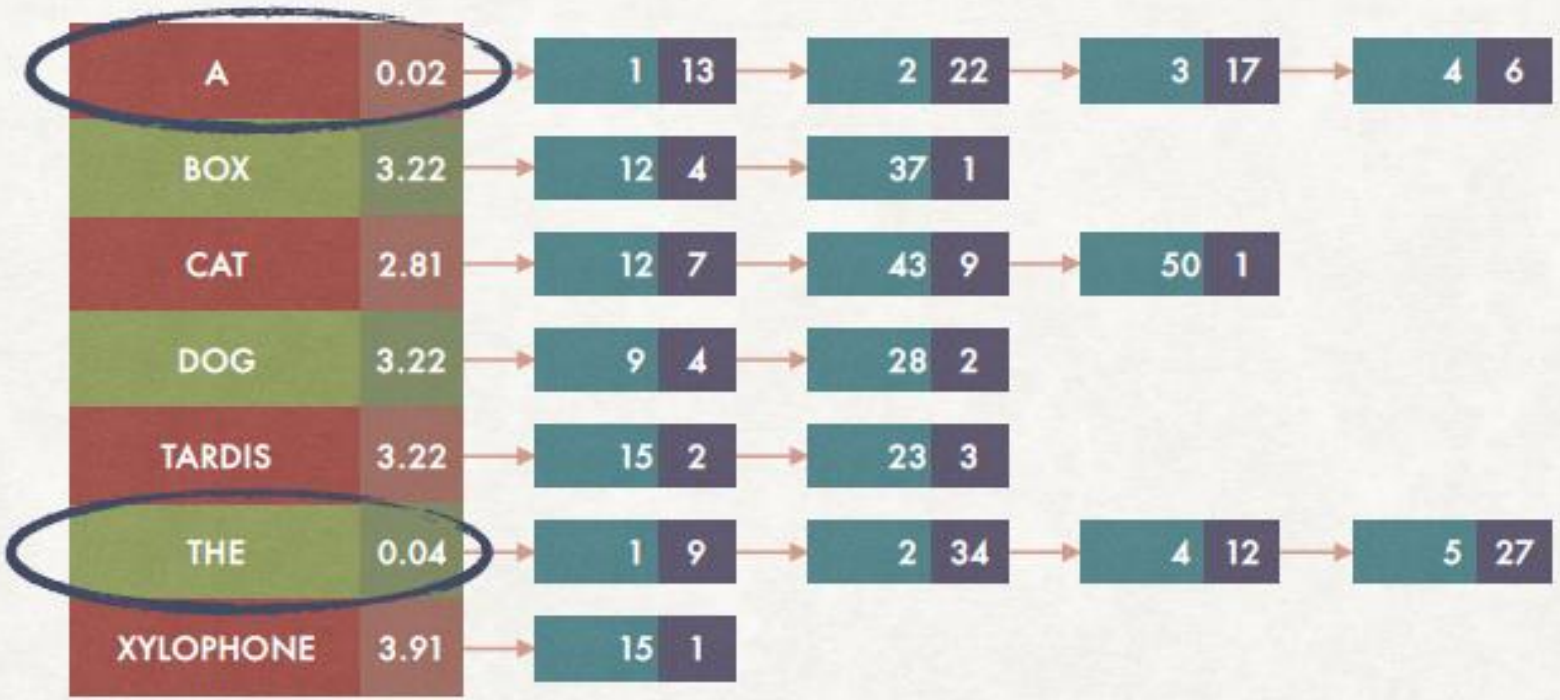
INDEX ELIMINATION

HOW TO IGNORE SOME TERMS



INDEX ELIMINATION

HOW TO IGNORE SOME TERMS



We can remove terms with very low idf score from the search: they are like "stop words" with very long postings list

INDEX ELIMINATION

HOW TO IGNORE SOME TERMS

- By removing terms with low idf value we can only work with relatively shorter lists.
- The cutoff value can be adapted according to the other terms present in the query.
- We can also only consider documents in which most or all the query terms appears…
- …but a problem might be that we do not have at least K documents matching all query terms.

Champion lists

- Precompute for each dictionary term *t*, the *r* docs of highest score in *t*'s posting list
  - Ideally $k < r << n$ (n = size of the posting list)
  - Champion list for *t* (or fancy list or top docs for *t*)
- We compute the union of the champion lists of all terms in the query, obtaining a set of documents.
- We find the K highest ranked documents in .
- Problem: we might have too few documents if K is not known until the query is performed.

Static quality scores

- We want top-ranking documents to be both *relevant* and *authoritative*
- *Relevance* is being modeled by cosine scores
- *Authority* is typically a query-independent property of a document
- Examples of authority signals
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - (Pagerank)

Modeling authority

- Assign to each document a *query-independent* quality score in [0,1] to each document *d*
  - Denote this by *g(d)*
- Consider a simple total score combining cosine relevance and authority
- Net-score(*q*,*d*) = *g(d)* + cosine(*q*,*d*)
  - Can use some other linear combination
  - Indeed, any function of the two "signals" of user happiness – more later
- Now we seek the top *k* docs by net score

Top *k* by net score – fast methods

- First idea: Order all postings by *g(d)*
- Key: this is a common ordering for all postings
- Thus, can concurrently traverse query terms' postings for
  - Postings intersection
  - Cosine score computation
- Under *g(d)*-ordering, top-scoring docs 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 docs in postings

STATIC QUALITY SCORES

ADDING A PRE-COMPUTABLE SCORE TO DOCUMENTS

- In some cases we might want to add a score to a document that is independent from the query: a **static quality score**, denoted by $g(d) \in [0,1]$.
- Example: good reviews by users might "push" a document higher in the scoring.
- We need to combine $g(d)$ with the scoring given by the query, a simple possibility is a linear combination:
  $\text{score}(q,d) = g(d) + \vec{v}(d) \cdot \vec{v}(q)$.
- We can also sort posting list by $g(d) + \text{idf}_{t,d'}$ to process documents more likely to have high scores first.

IMPACT ORDERING

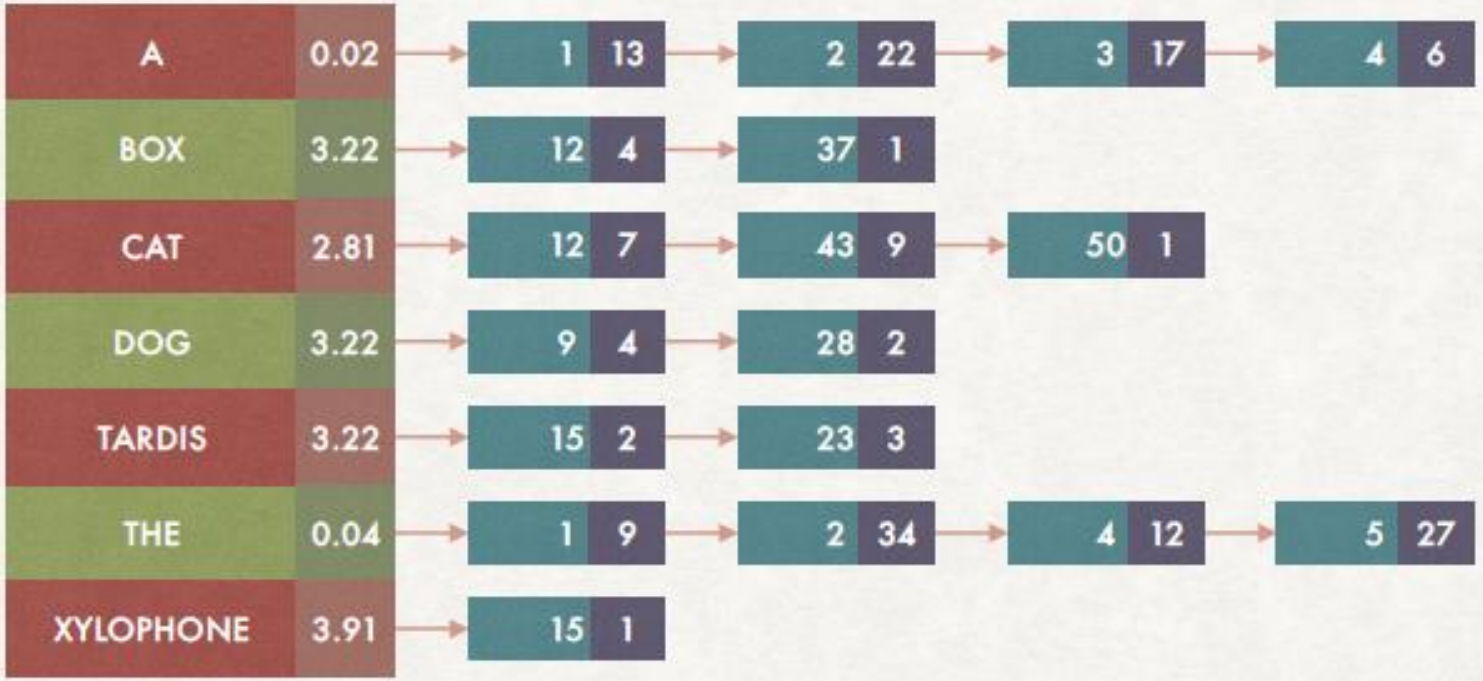
SORTING POSTING LISTS NOT BY DOCID

- Union and intersection for posting lists works efficiently because of the ordering…
- …but everything work as long as they are ordered with some criterium, not necessarily by DocID.
- Idea: Order the documents by decreasing $\mathrm{tf}_{t,d}$. In this way the documents which will obtain the highest scoring will be processed first.
- If the $\mathrm{tf}_{t,d}$ value drops below a threshold, then we can stop.

IMPACT ORDERING

SORTING POSTING LISTS NOT BY DOCID

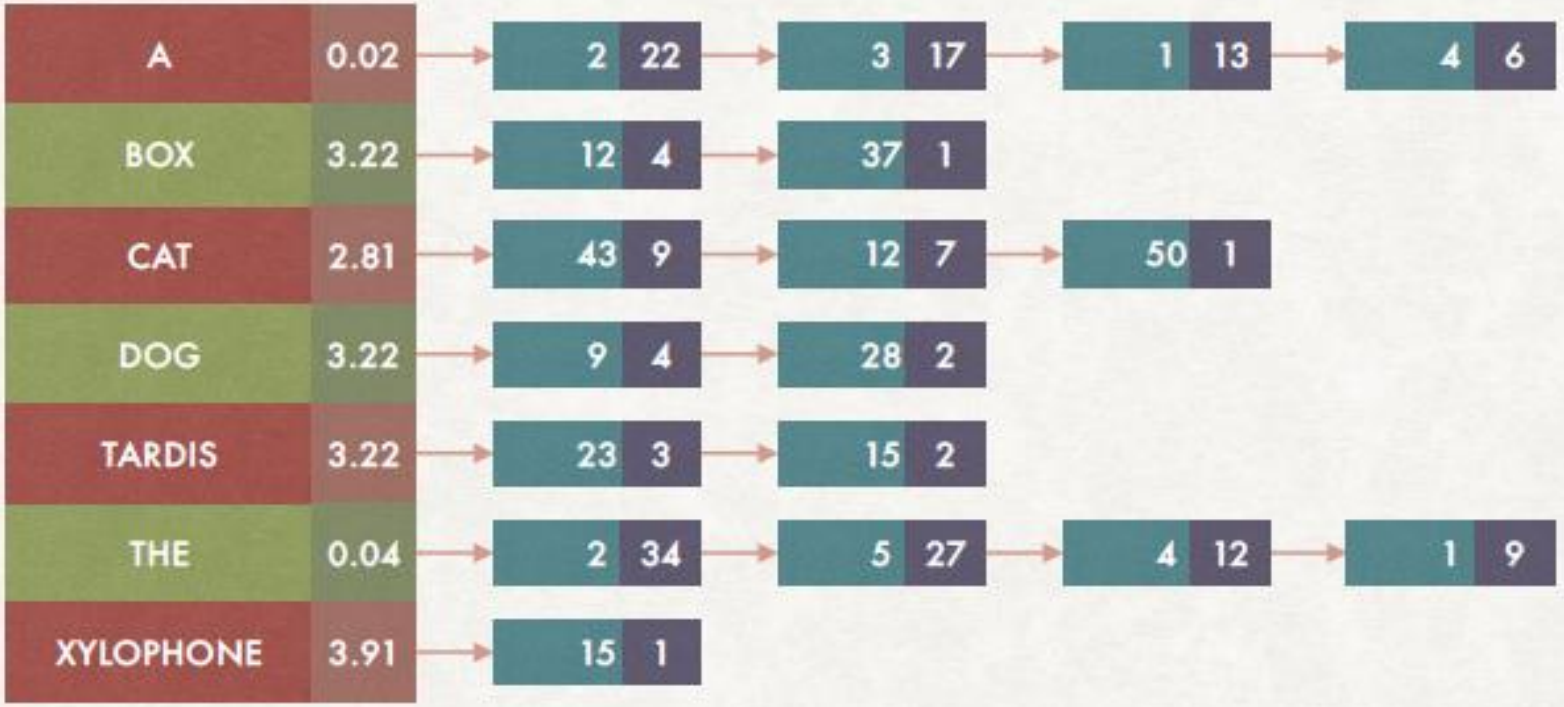
From this…



IMPACT ORDERING

SORTING POSTING LISTS NOT BY DOCID

...to this



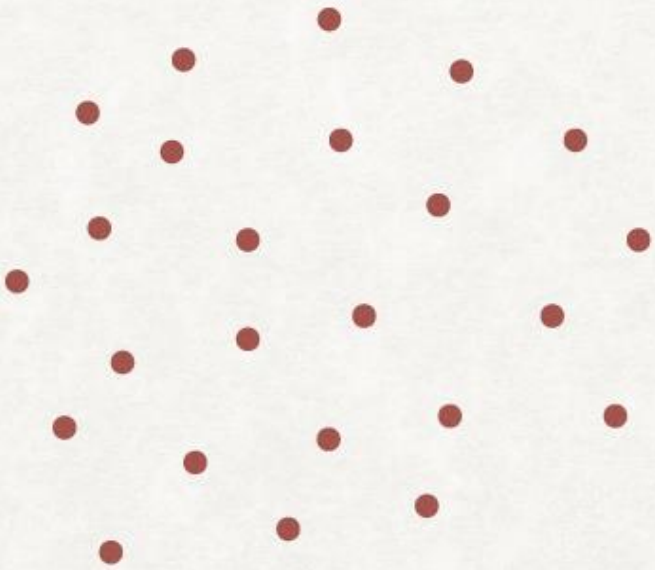
CLUSTER PRUNING

SEARCHING ONLY INSIDE A CLUSTER

- With N document, $M = \sqrt{N}$ are randomly selected as *leaders*. Each leader identifies a cluster of documents.
- For each of the remaining documents, we find the most similar among the M documents selected and we add it to the corresponding cluster.
- For a query q we find the document among the M leaders that is most similar to it.
- The K highest ranked documents are selected among the ones in the cluster of the selected leader.

CLUSTER PRUNING

AN EXAMPLE



Documents represented as points in space

CLUSTER PRUNING

AN EXAMPLE

Documents represented as points in space

Selection of the leaders

CLUSTER PRUNING

AN EXAMPLE

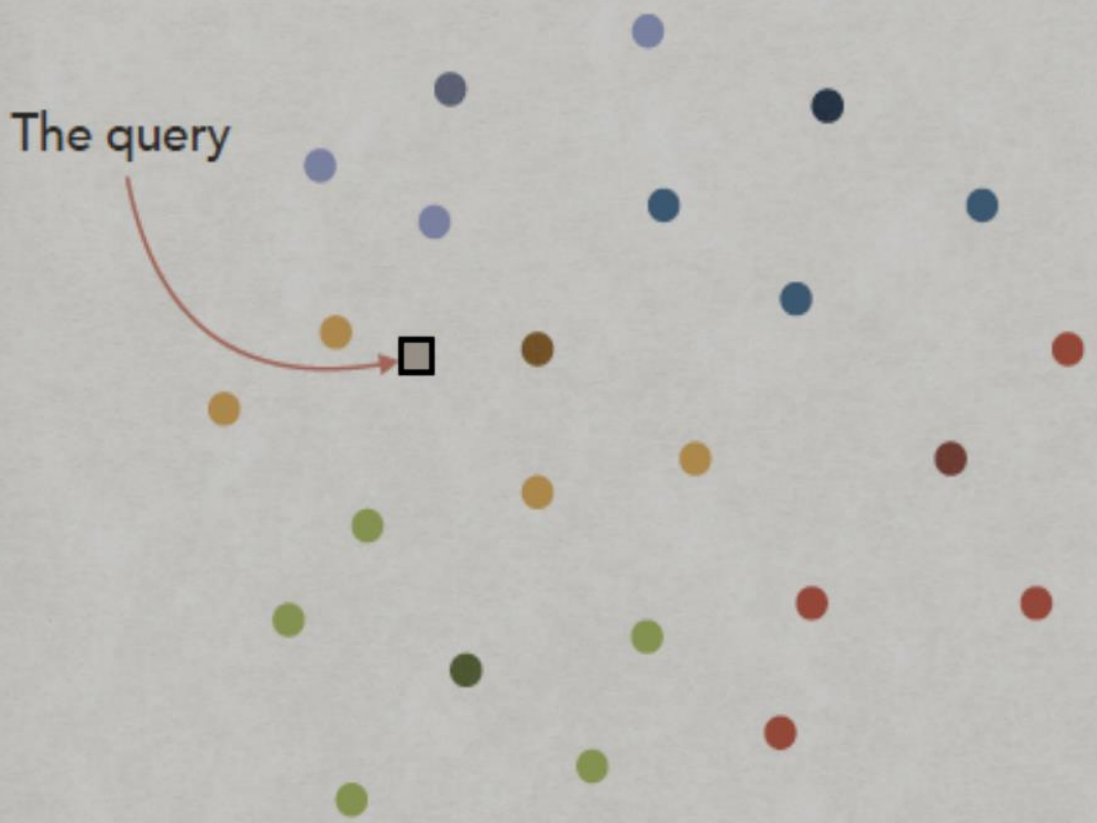
Documents represented as points in space

Selection of the leaders

Assigning documents to clusters

CLUSTER PRUNING

AN EXAMPLE



Documents represented as points in space

Selection of the leaders

Assigning documents to clusters

A query arrives

CLUSTER PRUNING

AN EXAMPLE

Documents represented as points in space

Selection of the leaders

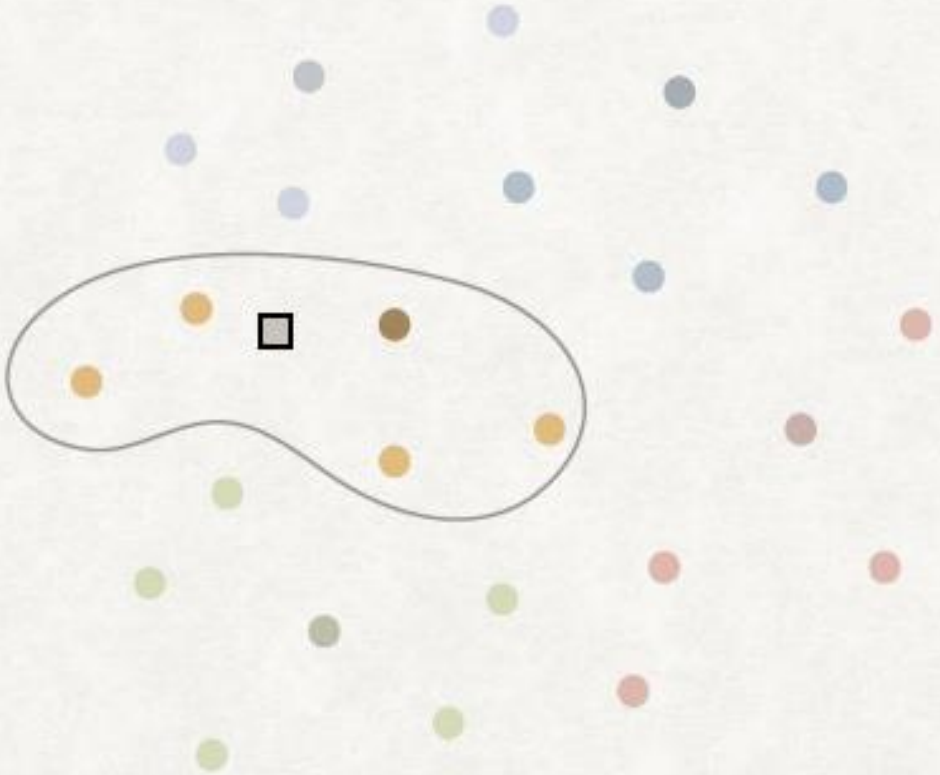
Assigning documents to clusters

A query arrives

The nearest leader is found

CLUSTER PRUNING

AN EXAMPLE



Documents represented as points in space

Selection of the leaders

Assigning documents to clusters

A query arrives

The nearest leader is found

The similarity is computed only in one cluster

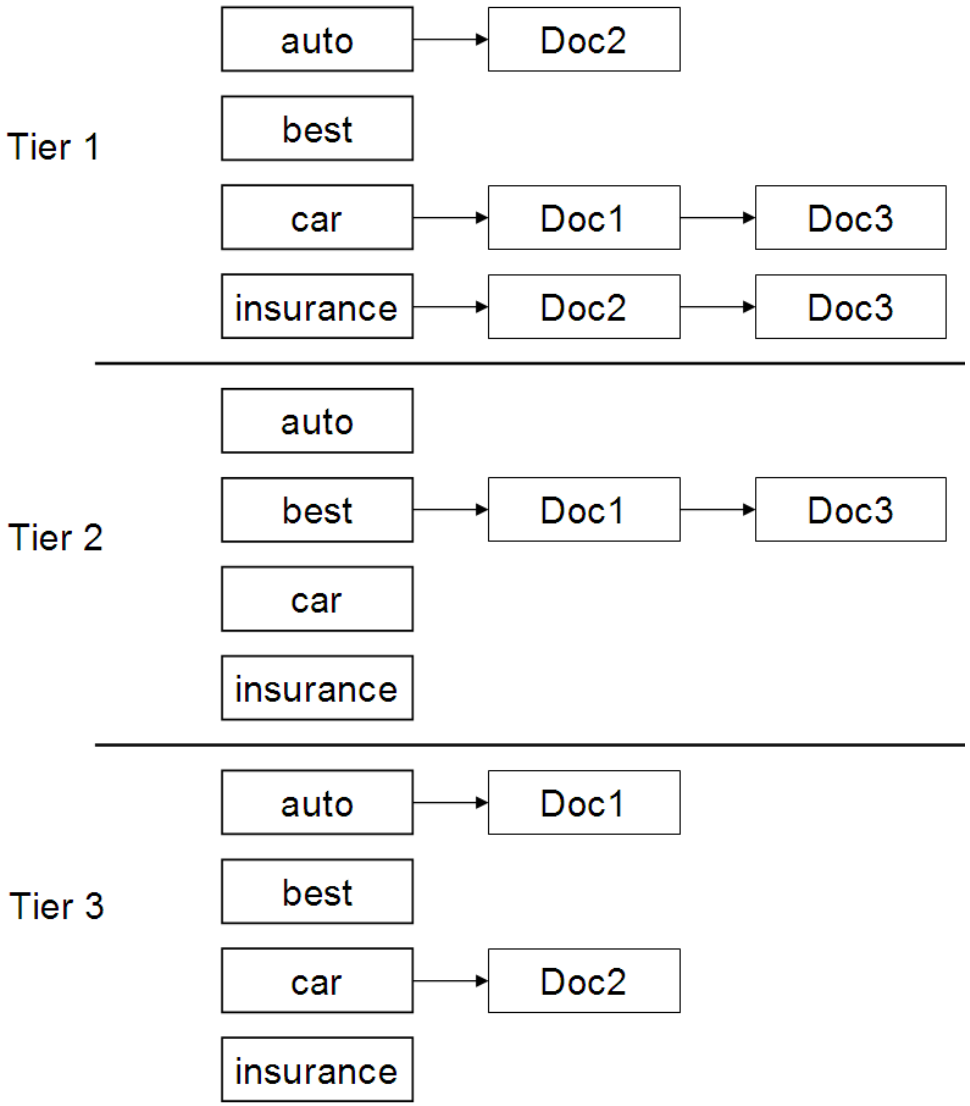
Components of an information retrieval system

- Further ideas for scoring, beyond vector spaces - **Tiered indexes**
- **Query-term proximity**
- **Designing parsing and scoring functions**
- **Putting it all together**

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

Example tiered index



Query-term proximity

- For free text queries on the web , users prefer a document in which most or all of the query terms appear close to each other.
- Consider a query with two or more query terms, $t1, t2, \ldots, tk$.
- Let ω be the width of the smallest window in a document d that contains all the query terms, measured in the number of words in the window.
- For instance, for the text on the document, The quality of mercy is not strained, the smallest window for the query strained mercy would be 4.
- Intuitively, the smaller that ω is, the better that d matches the query.
- We could also consider variants in which only words that are not stop words are considered in computing ω.
- We need to design such a *proximity-weighted* scoring function that depends on ω

Designing parsing and scoring functions

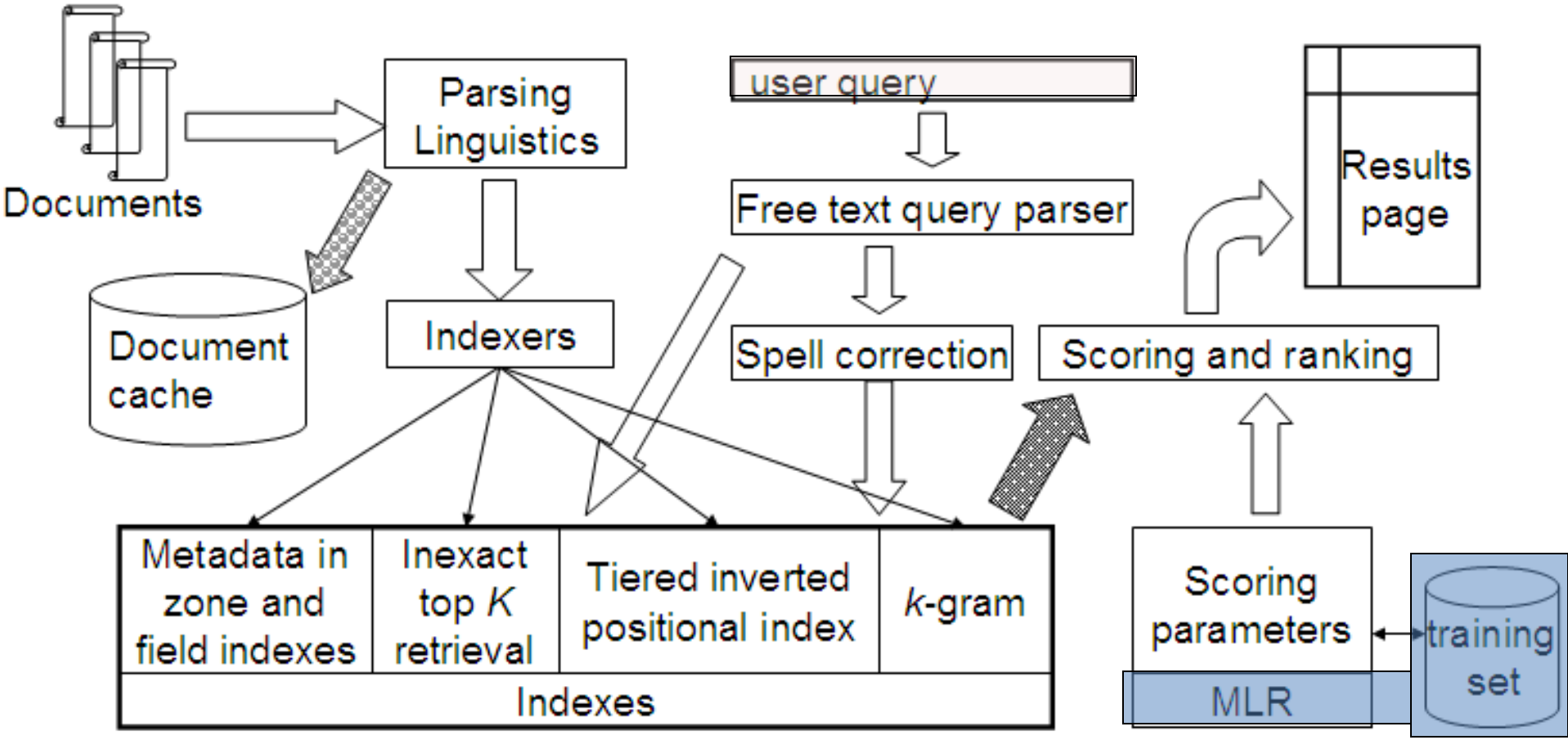
- Given *free text queries* interface, how should a search equipped with indexes for various retrieval operators treat a query such as rising interest rates?
- More generally, given the various factors we have studied that could affect the score of a document, how should we combine these features?
- A *query parser* is used to translate the user-specified keywords into a query with various operators that is executed against the underlying indexes.
- The query parser may issue a stream of queries:
  - Run the user-generated query string as a phrase query. Rank them by vector space scoring using as query the vector consisting of the 3 terms rising interest rates.
  - If fewer than ten documents contain the phrase rising interest rates, run the two 2-term phrase queries rising interest and interest rates; rank these using vector space scoring, as well.
  - If we still have fewer than ten results, run the vector space query consisting of the three individual query terms.

- Each of the above steps (if invoked) may yield a list of scored documents, for each of which we compute a score. This score must combine contributions from vector space scoring, static quality, proximity weighting and potentially other factors – this demands an aggregate scoring function.

Solutions

- Application builders make use of a toolkit of available scoring operators, along with a query parsing layer, with which they manually configure the scoring function as well as the query parser.
- Web search on the other hand is faced with a constantly changing document collection with new characteristics being introduced all the time. It is also a setting in which the number of scoring factors can run into the hundreds, making hand-tuned scoring a difficult exercise. To address this, it is becoming increasingly common to use machine-learned scoring, extending the ideas discussed.

Putting it all together



Vector space scoring and query operator interaction

- We discuss how the vector space scoring model relates to the query operators.
- The relationship should be viewed at two levels:
  - in terms of the expressiveness of queries that a sophisticated user may pose, and
  - in terms of the index that supports the evaluation of the various retrieval methods.
- In building a search engine,
  - we may opt to support multiple query operators for an end user
  - In doing so we need to understand what components of the index can be shared for executing various query operators,
  - as well as how to handle user queries that mix various query operators.

- Vector space scoring supports so-called *free text* retrieval, in which a query is specified as a set of words without any query operators connecting them.
  - It allows documents matching the query to be scored and thus ranked, unlike the Boolean, wildcard and phrase queries studied earlier.
  - Classically, the interpretation of such free text queries was that at least one of the query terms be present in any retrieved document.
  - However more recently, web search engines such as Google have popularized the notion that a set of terms typed into their query boxes retrieves documents containing all or most query terms.

Boolean Retrieval

- Clearly a vector space index can be used to answer Boolean queries, as long as the weight of a term *t* in the document vector for *d* is non-zero whenever *t* occurs in *d*. The reverse is not true, since a Boolean index does not by default maintain term weight information.
- There is no easy way of combining vector space and Boolean queries from a user's standpoint:
  - vector space queries are fundamentally a form of *evidence accumulation*, where the presence of more query terms in a document adds to the score of a document.
  - Boolean retrieval on the other hand, requires a user to specify a formula for *selecting* documents through the presence (or absence) of specific combinations of keywords, without inducing any relative ordering among them.
- Mathematically, it is in fact possible to invoke so-called, p-norms, to combine Boolean and vector space queries, but we know of no system that makes use of this fact.
  - a **norm** is a function from a real or complex vector space to the non-negative real numbers that behaves in certain ways like the distance

Wildcard queries

- Wildcard and vector space queries require different indexes, except at the basic level that both can be implemented using postings and a dictionary (e.g., a dictionary of trigrams for wildcard queries).
- If a search engine allows a user to specify a wildcard operator as part of a free text query (for instance, the query rom* restaurant), we may interpret the wildcard component of the query as spawning multiple terms in the vector space (in this example, rome and roman would be two such terms) all of which are added to the query vector.
- The vector space query is then executed as usual, with matching documents being scored and ranked; thus a document containing both rome and roma is likely to be scored higher than another containing only one of them. The exact score ordering will of course depend on the relative weights of each term in matching documents.

Phrase queries

- The representation of documents as vectors is fundamentally lossy: the relative order of terms in a document is lost in the encoding of a document as a vector.
- Even if we were to try and somehow treat every biword as a term (and thus an axis in the vector space), the weights on different axes not independent:
  - for instance the phrase German shepherd gets encoded in the axis german shepherd, but immediately has a non-zero weight on the axes german and shepherd. Further, notions such as idf would have to be extended to such biwords.
- Thus an index built for vector space retrieval cannot, in general, be used for phrase queries.
- Moreover, there is no way of demanding a vector space score for a phrase query -- we only know the relative weights of each term in a document.
- On the query german shepherd, we could use vector space retrieval to identify documents heavy in these two terms, with no way of prescribing that they occur consecutively.
- Phrase retrieval, on the other hand, tells us of the existence of the phrase german shepherd in a document, without any indication of the relative frequency or weight of this phrase.
- While these two retrieval paradigms (phrase and vector space) consequently have different implementations in terms of indexes and retrieval algorithms, they can in some cases be combined usefully, as in the three-step example of query parsing.