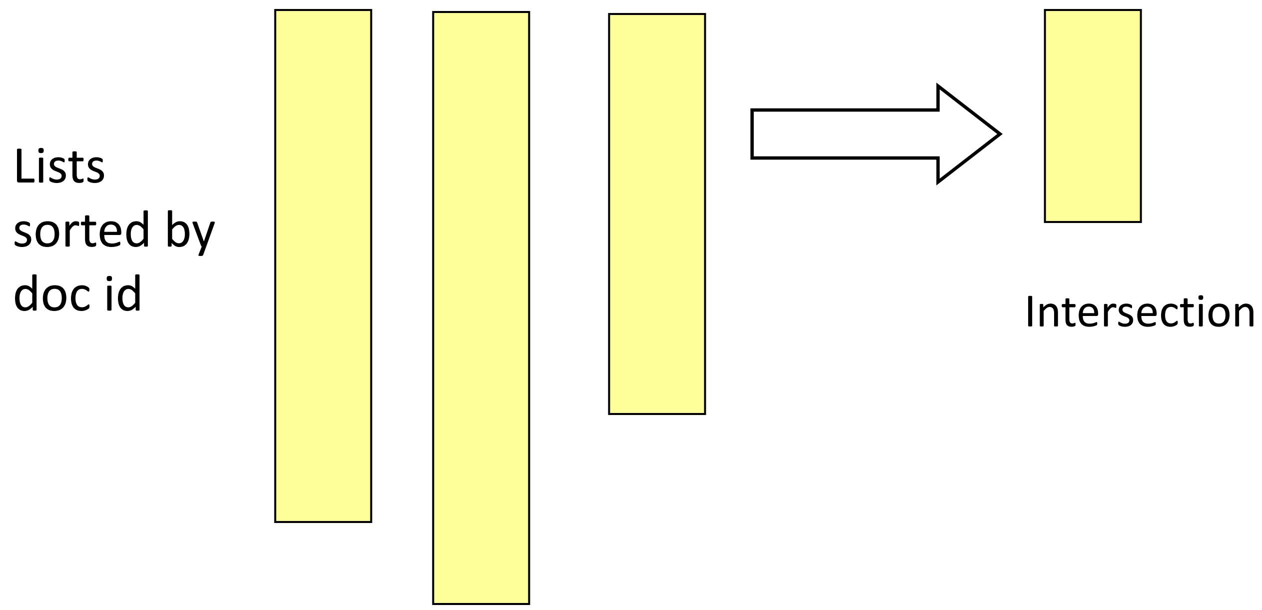
Index Structures and Query Processing

Debapriyo Majumdar
Information Retrieval
Indian Statistical Institute Kolkata

Merge union or intersection

Have to scan the lists fully: $O(m+n)$!

The lists can be VERY large.



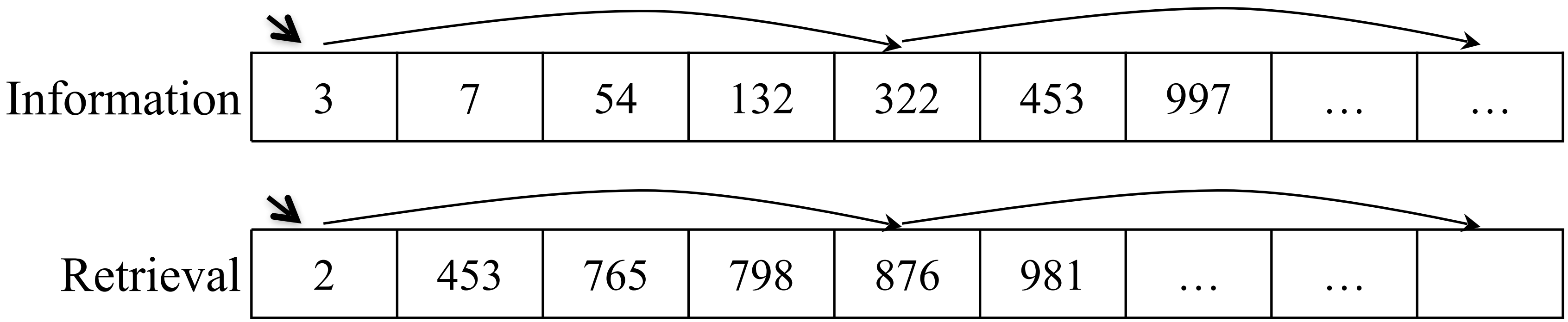
- Most real life queries are AND queries
  - User wants all the terms to be present

Skip lists: intersection

Information	3	7	54	132	322	453	997	…	…
Retrieval	2	453	765	798	876	981	…	…	

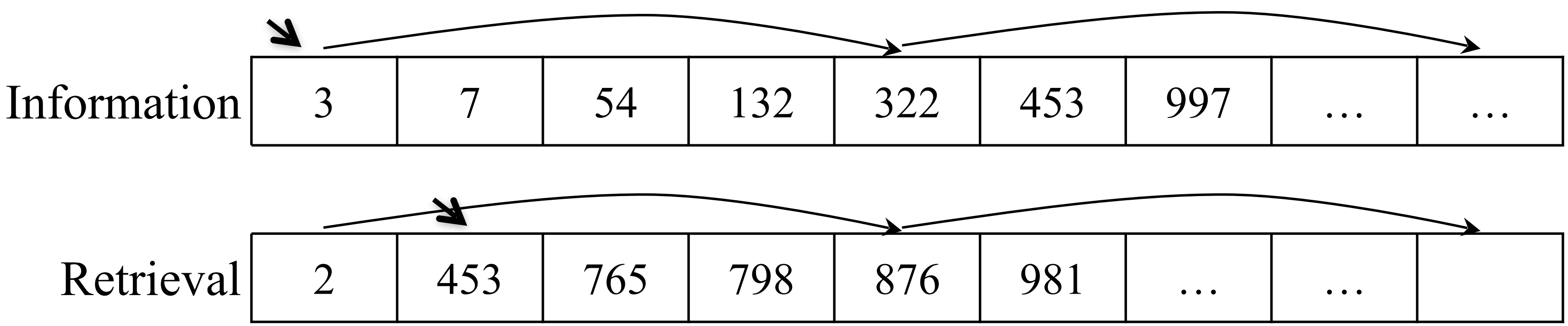
- Linear intersection
  - Start from the beginning
  - Determine the smaller id, move on that list
  - If ids on both lists match, add to result list
  - May have to advance pointer in one list alone and keep comparing
  - Could we skip them?

Skip lists: intersection



- Skip lists
  - Start from the beginning, determine the smaller id (2 < 3)
  - Is the id following the skip pointer on list 2 also smaller than 3?
  - No, so move to 453 on list 2

Skip lists: intersection



- Skip lists
  - Start from the beginning, determine the smaller id (2 < 3)
  - Is the id following the skip pointer on list 2 also smaller than 3?
  - No, so move to 453 on list 2
  - Now 3 < 453, the next id 7 also may be < 453, may be even the next
  - Check: is the next id following the skip pointer (322) < 453? Yes!
  - Skip to 322, skipping some part of the list
  - Continue ☺

Skip lists: discussion

- Built in indexing time
- Where to place the skip pointers?
  - More skip pointers: more comparison overhead
  - Less skip pointers: less skips
  - Empirical tradeoff: for a list of size n, keep $\sqrt{n}$ evenly spaced skip pointers
- Maintaining
  - Building at indexing time is easy
  - Maintaining is difficult if the index is updated frequently
  - In particularly, need to be careful with deletion

Phrase queries

- The simple term → doc ids posting list cannot answer phrase queries such as: "indian statistical institute"
- <u>Bi-word index</u>
  - Keep an index with all *pairs of consecutive words* as keys

indian statistical	3	7	54	132	322	453	997	…	…
statistical institute	2	453	765	798	876	981	…	…	

- Does not exactly correspond to all documents corresponding to the query phrase, there would be some false positives [why?]
- But works fairly well in practice

Positional index

- Together with the term → document ids posting list, store the positions where the term occurs in each document
- Each entry in the posting list:
  `Doc_ID :: Score :: Positions`
- Intersection as usual on posting lists
- In addition to matching docIds, also match the positions

diwali:	d_3::0.3::<1>		
indian:	d_2::0.2::<4,8>	d_3::0.4::<7>	d_7::0.9::<1>
institute:	d_1::0.1::<2,9>	d_2::0.8::<10>	
population:	d_7::0.5::<2>		
autumn:	d_4::0.4::<3>		
statistical:	d_1::0.3::<1>	d_2::0.6::<9>	

Parametric and zone indexes

- Thus far, a document has been a sequence of terms
- In fact documents have multiple parts, some with special semantics:
  - Author
  - Title
  - Date of publication
  - Language
  - Format
  - etc.
- These constitute the *metadata* about a document

Fields

- We sometimes wish to search by these metadata
  - E.g., find docs authored by William Shakespeare in the year 1601, containing *alas poor Yorick*
- Year = 1601 is an example of a field
- Also, author last name = shakespeare, etc.
- Field or parametric index: postings for each field value
  - Sometimes build range trees (e.g., for dates)
- Field query typically treated as conjunction
  - (doc *must* be authored by shakespeare)

Zone

- A zone is a region of the doc that can contain an arbitrary amount of text, e.g.,
  - Title
  - Abstract
  - References …
- Build inverted indexes on zones as well to permit querying
- E.g., "find docs with *merchant* in the title zone and matching the query *gentle rain*"

Example: Zone Index

William.abstract → 11 → 121 → 1441 → 1729

William.author → 2 → 4 → 8 → 16

William.title → 2 → 3 → 5 → 8

Zone index and posting lists

William → 2.author, 2.title → 3.author → 4.title → 5.author

Basics of Ranking

- Single term query
  - The score (may be TF-iDF) for the document corresponding to the term = the score for the document corresponding to the query
- Queries with more than one term: how do we combine scores across multiple posting lists?

Indian	D1 :: 0.5	D2 :: 0.2	D3: 0.7	D4 :: 0.2	D5 :: 0.1	D6 :: 0.2
Statistical	D3 :: 0.7	D5 :: 0.3				
Institute	D2 :: 0.2	D4 :: 0.3	D5 :: 0.8	D6 :: 0.2		

- Simple assumption: sum the scores (analogous to dot product)
- Usual merge union / intersection, add the scores along the way

Query processing

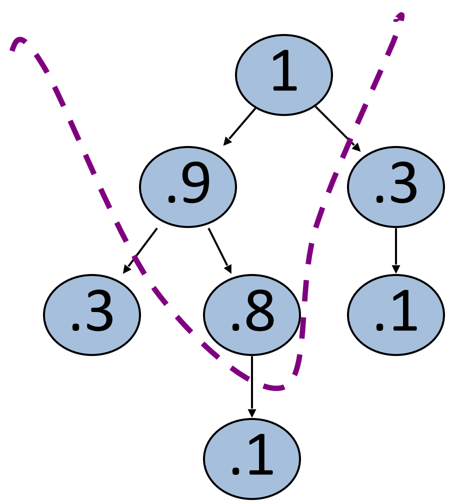
- How to compute cosine similarity and find top-*k*?
- Similar to merge union or intersection using inverted index
  - The aggregation function changes to compute cosine
  - We are interested only in ranking, no need of normalizing with query length
  - If the query words are unweighted, this becomes very similar to merge union
  - Exercise: work out the details
- Need to find top-*k* results after computing the union list with (cosine) scores

Partial sort using heap for selecting top k

Usual sort takes $O(n \log n)$ operations (may be n = 1 million)

<u>Partial sort</u>

- Binary tree in which each node's value > the values of children
- Takes $O(n)$ operations to construct, then read each of top k in $O(\log n)$ steps.
- For n =1 million, k = 100, this is about 10% of the cost of sorting



Length normalization and query – document similarity

Term – document scores

TF.iDF or similar scoring. May already apply some normalization to eliminate bias on long documents

Document length normalization

Ranking by cosine similarity is equivalent to further normalizing the term – document scores by document length. Query length normalization is redundant.

Similarity measure and aggregation

- Cosine similarity
- Sum the scores with union
- Sum the scores with intersection
- Other possible approaches

Top-k algorithms

- If there are millions of documents in the lists
  - Can the ranking be done without accessing the lists fully?
- Exact top-k algorithms (used more in databases)
  - Family of threshold algorithms (Ronald Fagin et al)
  - Threshold algorithm (TA)
  - No random access algorithm (NRA) [we will discuss, as an example]
  - Combined algorithm (CA)
  - Other follow up works

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm:

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm: round 1

0.6 + 0.6 + 0.9 = 2.1

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

current score

best-score

Candidates

Candidates
doc 83 [0.9, **2.1**]
doc 17 [0.6, **2.1**]
doc 25 [0.6, **2.1**]

min top-2 score: 0.6

maximum score for unseen docs: **2.1**

min-top-2 < best-score of candidates

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm: round 2

0.5 + 0.6 + 0.7 = 1.8

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

Candidates

doc 17 [1.3, **1.8**]
doc 83 [0.9, **2.0**]
doc 25 [0.6, **1.9**]
doc 38 [0.6, **1.8**]
doc 78 [0.5, **1.8**]

min top-2 score: 0.9

maximum score for unseen docs: **1.8**

min-top-2 < best-score of candidates

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm: round 3

0.4 + 0.6 + 0.3 = 1.3

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

Candidates

doc 83 [1.3, **1.9**]
doc 17 [1.3, **1.7**]
doc 25 [0.6, **1.5**]
doc 78 [0.5, **1.4**]

min top-2 score: 1.3

maximum score for unseen docs: 1.3

min-top-2 < best-score of candidates

no more new docs can get into top-2

but, extra candidates left in queue

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm: round 4

0.3 + 0.6 + 0.2 = 1.1

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

Candidates

doc 17 **1.6**
doc 83 [1.3, **1.9**]
doc 25 [0.6, **1.4**]

min top-2 score: 1.3

maximum score for unseen docs: 1.1

min-top-2 < best-score of candidates

no more new docs can get into top-2

but, extra candidates left in queue

NRA (No Random Access) Algorithm

Fagin's NRA Algorithm: round 5

0.2 + 0.5 + 0.1 = 0.8

List 1	List 2	List 3
doc 25 0.6	doc 17 0.6	doc 83 0.9
doc 78 0.5	doc 38 0.6	doc 17 0.7
doc 83 0.4	doc 14 0.6	doc 61 0.3
doc 17 0.3	doc 5 0.6	doc 81 0.2
doc 21 0.2	doc 83 0.5	doc 65 0.1
doc 91 0.1	doc 21 0.3	doc 10 0.1
	doc 44 0.1	

lists sorted by score

read one doc from every list

Candidates

doc 83 **1.8**
doc 17 **1.6**

min top-2 score: **1.6**

maximum score for unseen docs: 0.8

no extra candidate in queue

Done!

More approaches:

- Periodically also perform random accesses on documents to reduce uncertainty (CA)
- Sophisticated scheduling on lists
- Crude approximation: NRA may take a lot of time to stop. Just stop after a while with approximate top-k – who cares if the results are perfect according to the scores?

Inexact top-k retrieval

- 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*

Champion lists

- Precompute for each dictionary term *t*, the *r* docs of highest score in *t*'s posting list
  - Ideally $k < r \ll n$ (n = size of the posting list)
  - Champion list for *t* (or fancy list or top docs for *t*)
- Note: *r* has to be chosen at index build time
  - Thus, it's possible that $r < k$
- At query time, only compute scores for docs in the champion list of some query term
  - Pick the *K* top-scoring docs from amongst these

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$(q,d) = g(d) + \text{cosine}(q,d)$
  - 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 (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

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-idf_{td}
- Seek top-k results from only the docs in these champion lists

High and low lists

- For each term, we maintain two postings lists called *high* and *low*
  - Think of *high* as the champion list
- When traversing postings on a query, only traverse *high* lists first
  - If we get more than k docs, select the top k and stop
  - Else proceed to get docs from the *low* lists
- Can be used even for simple cosine scores, without global quality $g(d)$ scores
- A means for segmenting index into two <u>tiers</u>

Impact-ordered postings

- We only want to compute scores for docs *d* for which $wf_{t,d}$, for query term *t,* is high enough
- Sort each postings list by $wf_{t,d}$
- Now: not all postings in a common order!
- How do we compute scores in order to pick top *k?*
  - Two ideas follow

1. Early termination

- When traversing *t*'s postings, stop early after either
  - a fixed number of *r* docs
  - $wf_{t,d}$ drops below some threshold
- Take the union of the resulting sets of docs
  - One from the postings of each query term
- Compute only the scores for docs in this union

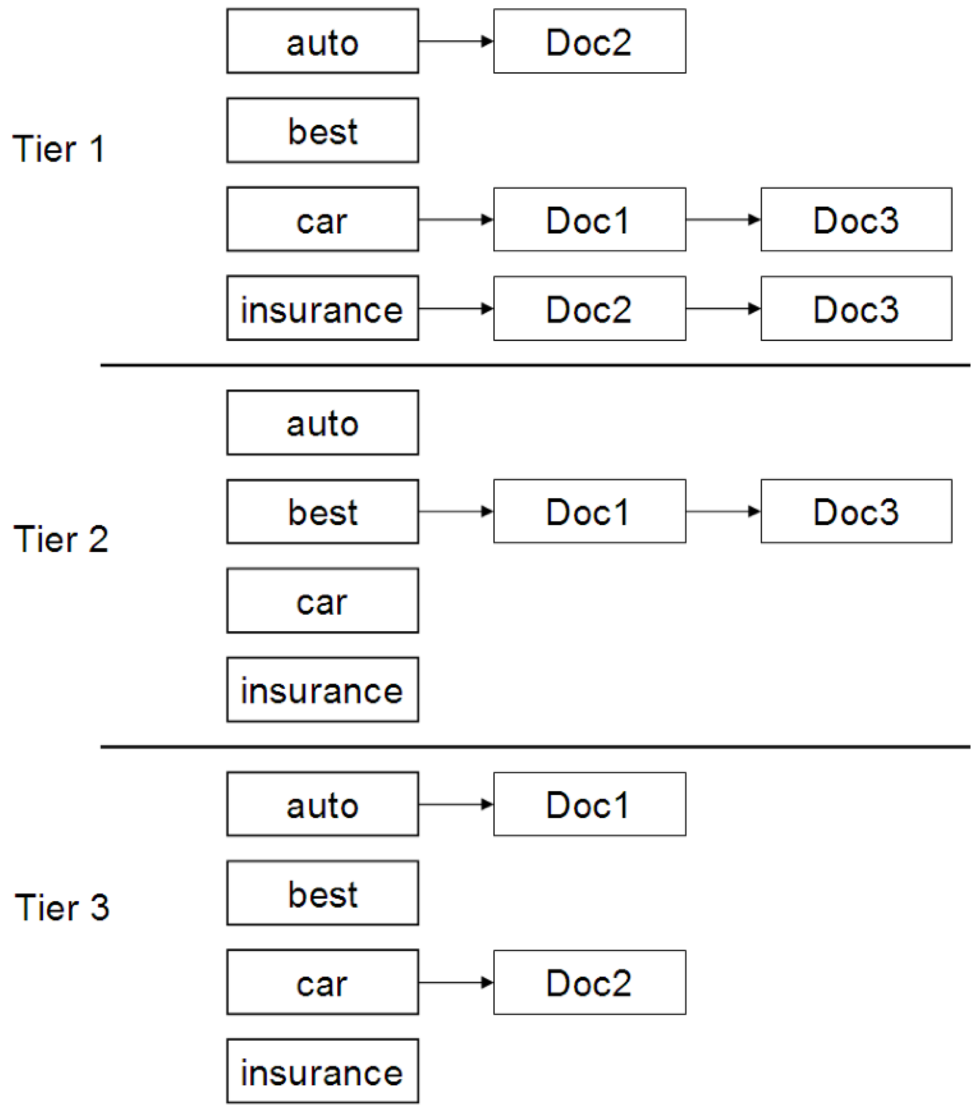
2. iDF-ordered terms

- When considering the postings of query terms
- Look at them in order of decreasing idf
  - High idf terms likely to contribute most to score
- As we update score contribution from each query term
  - Stop if doc scores relatively unchanged
- Can apply to cosine or some other net scores

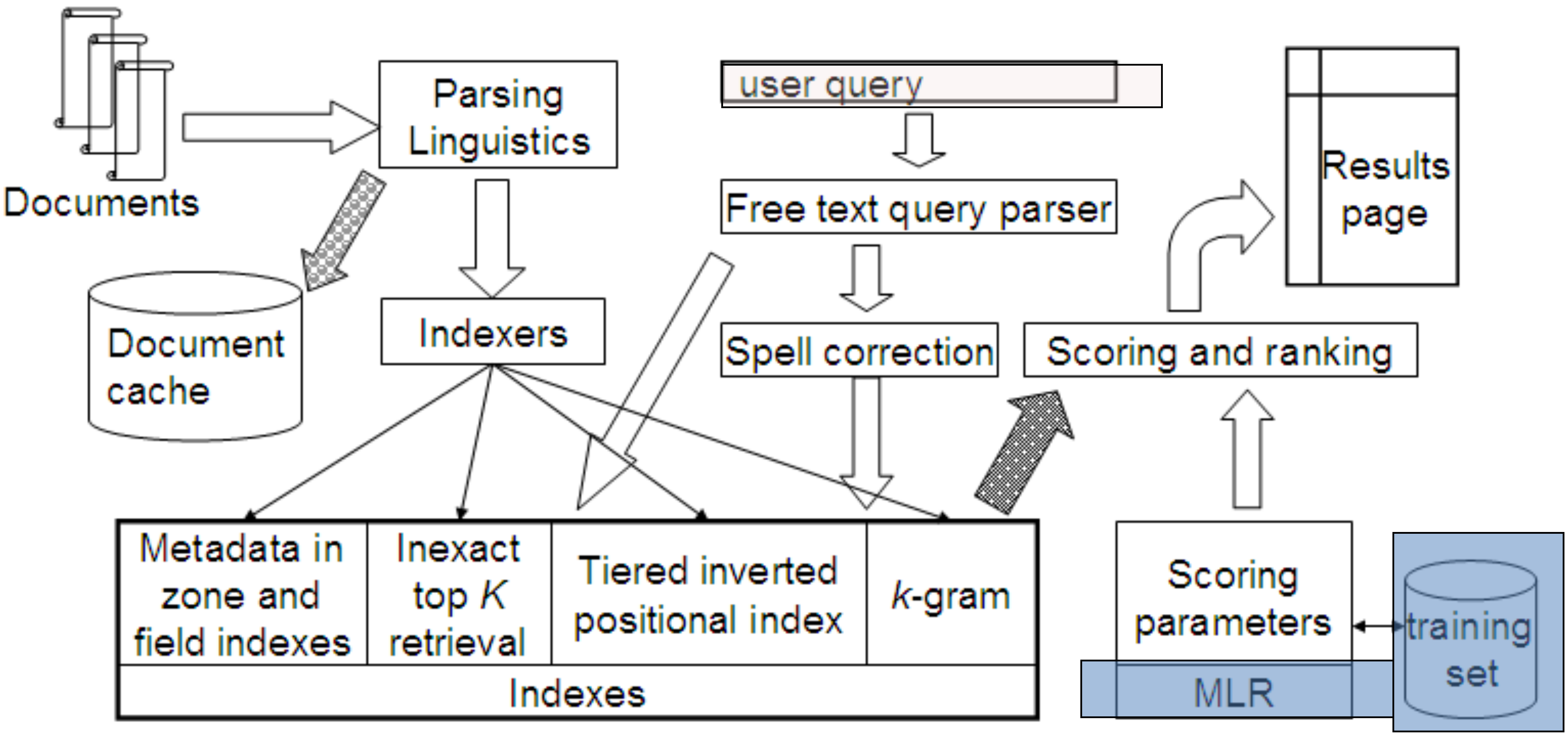
Tiered indexes

- 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



Putting it all together



Sources and Acknowledgements

- IR Book by Manning, Raghavan and Schuetze: http://nlp.stanford.edu/IR-book/
- Acknowledgment: Some slides are adapted from the slides by Prof. Nayak and Prof. Raghavan for their course in Stanford University