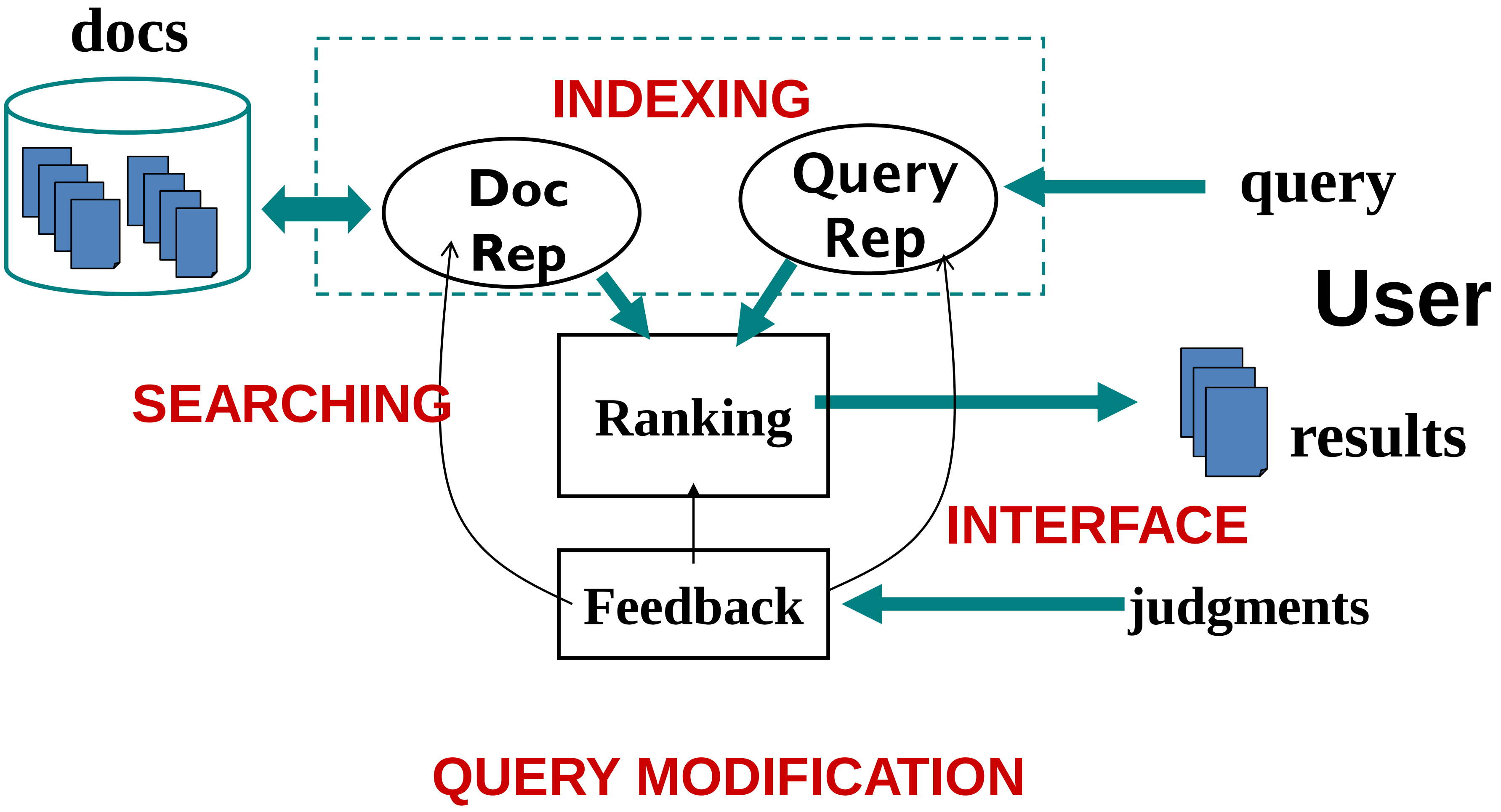
Implementation Issues & IR Systems

Lecture Plan

- How to implement a simple IR system
  - Index construction
  - Scoring
- Open source IR toolkits

IR System Architecture



Indexing

- Indexing = Convert documents to data structures that enable fast search

Unstructured data

- Which plays of Shakespeare contain the words Brutus and Caesar, but not Calpurnia?
- One could grep all of Shakespeare's plays for Brutus and Caesar, then strip out lines containing Calpurnia.
- Why is grep not the solution?
  - Slow (for large collections)
  - "not Calpurnia" is non-trivial
  - Other operations (e.g., find the word Romans near countryman) not feasible
  - Ranked retrieval (best documents to return)

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Mac beth	…
Anthony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	$\boxed{1}$	$\boxed{0}$	0	0	0	
…							

Entry is 1 if term occurs. Example: Calpurnia occurs in Julius Caesar. Entry is 0 if term doesn't occur. Example: Calpurnia doesn't occur in The tempest.

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer the query Brutus and Caesar and not Calpurnia:
  - Take the vectors for Brutus, Caesar, and Calpurnia
  - Complement the vector of Calpurnia
  - Do a (bitwise) **AND** on the three vectors
  - 110100 **AND** 110111 **AND** 101111 = 100100

Bigger collections

- Consider $N = 106$ documents, each with about 1000 tokens
- On average 6 bytes per token, including spaces and punctuation ⇒ size of document collection is about 6 GB
- Assume there are $M = 500000$ distinct terms in the collection
- $M = 500{,}000 \times 106 =$ half a trillion 0s and 1s.
- But the matrix has no more than one billion 1s.
  - Matrix is extremely sparse.
- What is a better representations?
  - We only record the 1s.

Indexing

- Inverted index is the dominating indexing method (used by all search engines)
- Other indices (e.g., document index) may be needed for feedback

Inverted Index

- Fast access to all docs containing a given term (along with freq and pos information)
- For each term, we get a list of tuples (docID, freq, pos).
- Given a query, we can fetch the lists for all query terms and work on the involved documents.
  - Boolean query: set operation
  - Natural language query: term weight summing

Inverted Index Example

Doc 1

… news about

Doc 2

… news about organic food campaign

Doc 3

… news of presidential campaign … presidential candidate …

Dictionary (or Lexicon)

Term	# docs	Total freq
news	3	3
campaign	2	2
presidential	1	2
food	1	1
…	…	…

Postings

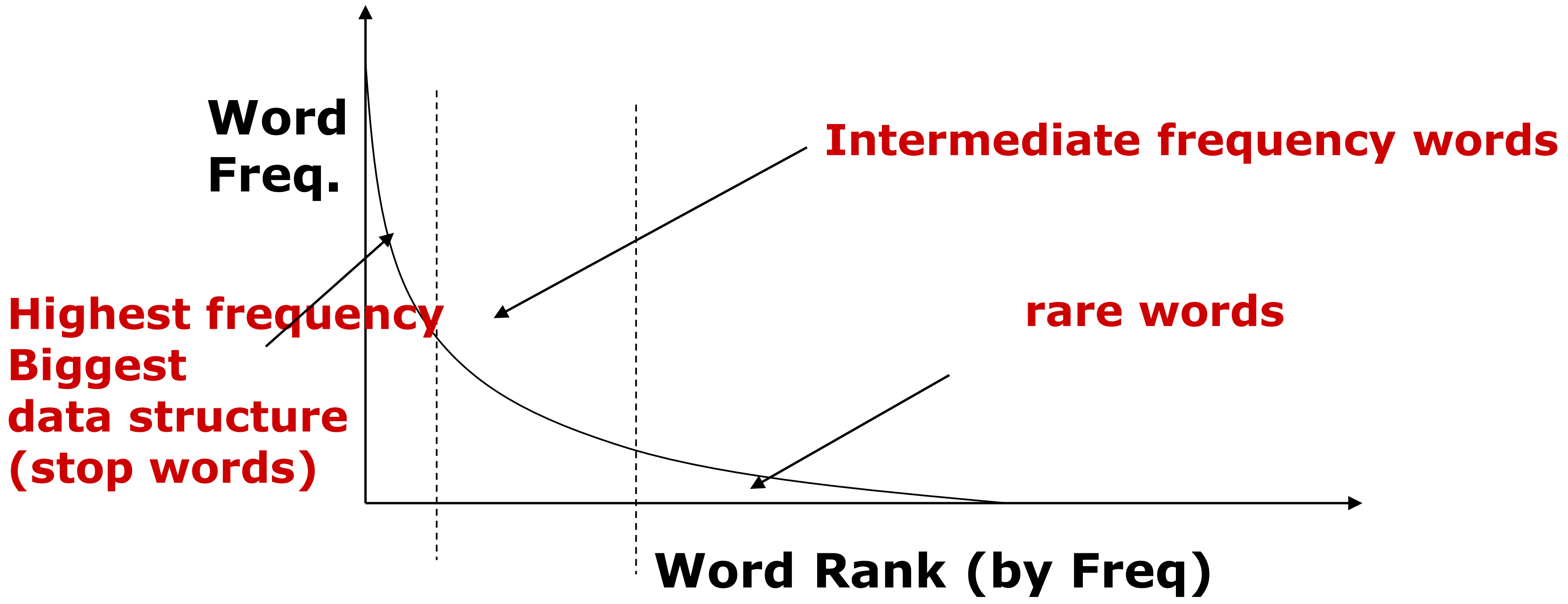
Doc id	Freq	**Position**
1	1	**p1**
2	1	**p2**
3	1	**p3**
2	1	**p4**
3	1	**p5**
2	2	**p6, p7**
2	1	**P8**
…	…	
…	…	

Inverted Index for Fast Search?

- Single-term query?
- Multi-term Boolean query?
  - Must match term “A” AND term “B”
  - Must match term “A” OR term “B”
- Multi-term keyword query
  - Similar to disjunctive Boolean query (“A” OR “B”)
  - Aggregate term weights
- More efficient than sequentially scanning documents (why?)

Empirical Distribution of Words – Zipf's Law

- $rank \times frequency \quad constant$



Data Structures for Inverted Index

- Dictionary: modest size
  - Needs fast random access
  - Preferred to be in memory
  - Hash table, B-tree, trie, …
- Postings: huge
  - Sequential access is expected
  - Can stay on disk
  - May contain docID, term freq., term pos, etc
  - Compression is desirable

Inverted Index Compression

- In general, leverage skewed distribution of values and use variable-length encoding
- TF compression
  - Small numbers tend to occur far more frequently than large numbers (why?)
  - Fewer bits for small (high frequency) integers at the cost of more bits for large integers
- Doc ID compression
  - "d-gap" (store difference): $d1, d2 - d1, d3 - d2, \ldots$
  - Feasible due to sequential access
- Methods: Binary code, unary code, γ-code, δ-code, …

Integer Compression Methods

- Binary: equal-length coding
- Unary: $x \geq 1$ is coded as x one bits followed by 0, e.g., $3 \Rightarrow 1110;\ 5 \Rightarrow 111110$
- γ-code: $x \Rightarrow$ a pair of length, offset. *Offset* is x in binary with the leading 1 removed. *Length* encodes the length of offset in unary code
- δ-code: same as γ-code, but replace the unary prefix with γ-code.

Integer Compression Methods - Example

$\underline{\boldsymbol{X = 23}}$

- Unary: 11111111111111111111111110
- γ-code: $x \Rightarrow$ a pair of length, offset. *Offset* is x in binary with the leading 1 removed. *Length* encodes the length of offset in unary code
  - 23 in binary: 10111
  - After removing leading 1: 0111 (offset)
  - Length of offset in unary: 11110 (length)
  - 111100111

Integer Compression Methods - Example

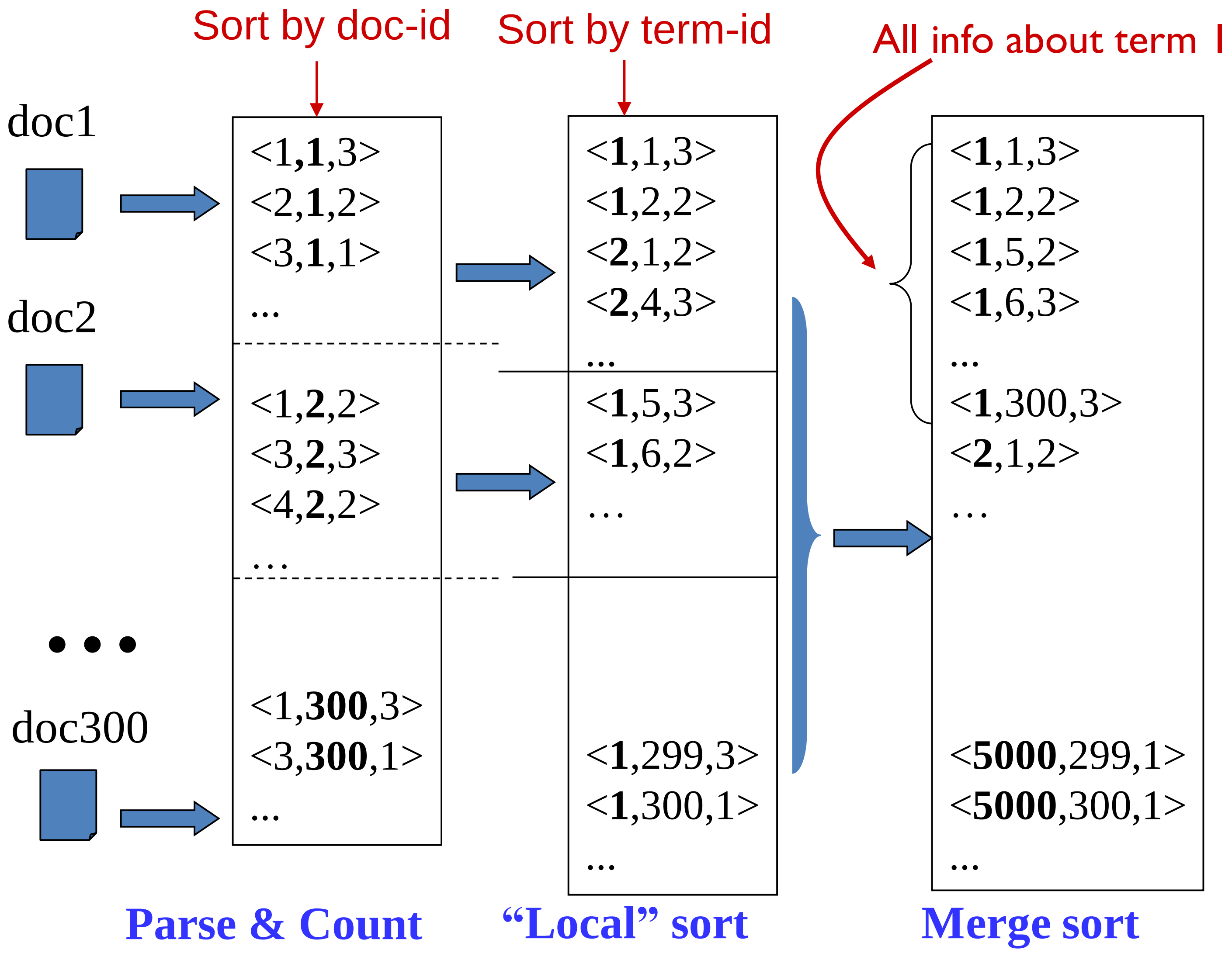
$\underline{\boldsymbol{X = 23}}$

- δ-code: same as γ-code ,but replace the unary prefix with γ-code.
  - Offset: 0111
  - Length in γ-code: 11000
  - 110000111

Constructing Inverted Index

- The main difficulty is to build a huge index with limited memory
- Memory-based methods: not usable for large collections
- Sort-based methods:
  - Step 1: collect local (termID, docID, freq) tuples
  - Step 2: sort local tuples (to make “runs”)
  - Step 3: pair-wise merge runs
  - Step 4: Output inverted file

Sort-based Inversion



Term Lexicon:

the 1
campaign 2
news 3
a 4
…

DocID Lexicon:

doc1 1
doc2 2
doc3 3
…

Searching

- Given a query, score documents efficiently
- Boolean query
  - Fetch the inverted list for all query terms
  - Perform set operations to get the subset of docs that satisfy the Boolean condition
  - E.g., $Q_1 = \text{“}info\text{”}\ AND\ \text{“}security\text{”}$, $Q_2 = \text{“}info\text{”}\ OR\ \text{“}security\text{”}$
    - info: d_1, d_2, d_3, d_4
    - security: d_2, d_4, d_6
    - Results: $\{d_2, d_4\}$ (Q_1) $\{d_1, d_2, d_3, d_4, d_6\}$ (Q_2)

How to Score Documents Quickly?

General form of scoring functions

Final score adjustment

$$f(q,d) = f_a\big(h(\underline{g(t_1,d,q)},\ldots,\underline{g(t_k,d,q)}),f_d(d),f_q(q)\big)$$

Weight **aggregation**

Weight of a **matched** query term in d

A General Algorithm for Ranking Documents

$$\boldsymbol{f}(\boldsymbol{q},\boldsymbol{d}) = \boldsymbol{f_a}\left(\boldsymbol{h}\left(\boldsymbol{g}(\boldsymbol{t_1},\boldsymbol{d},\boldsymbol{q}),\ldots,\boldsymbol{g}(\boldsymbol{t_k},\boldsymbol{d},\boldsymbol{q})\right),\boldsymbol{f_d}(\boldsymbol{d}),\boldsymbol{f_q}(\boldsymbol{q})\right)$$

- $\boldsymbol{f_d}(\boldsymbol{d})$ and $\boldsymbol{f_q}(\boldsymbol{q})$ are pre-computed
- Maintain a score accumulator for each $\boldsymbol{d}$ to compute $\boldsymbol{h}$
- For each query term t_i
  - Fetch the inverted list $\{(d_1, f_1), \ldots, (d_n, f_n)\}$
  - For each entry (d_j, f_j), compute $\boldsymbol{g}(\boldsymbol{t_i},\boldsymbol{d_j},\boldsymbol{q})$, and update score accumulator for d_j to incrementally compute h
- Adjust the score to compute $\boldsymbol{f_a}$, and sort

Ranking Documents: Example

$f(d,q) = g(t_1,d,q) + \cdots + g(t_k,d,q)$ where $g(t_i,d,q) = c(t_i,d)$
Query = "*info security*"

Info: $(d_1,3),(d_2,4),(d_3,1),(d_4,5)$
Security: $(d_2,3),(d_4,1),(d_5,3)$

	Accumulators:	$\boldsymbol{d_1}$	$\boldsymbol{d_2}$	$\boldsymbol{d_3}$	$\boldsymbol{d_4}$	$\boldsymbol{d_5}$
		0	0	0	0	0
info	$(d_1,3) \Rightarrow$	**3**	0	0	0	0
	$(d_2,4) \Rightarrow$	3	**4**	0	0	0
	$(d_3,1) \Rightarrow$	3	4	**1**	0	0
	$(d_4,5) \Rightarrow$	3	4	1	**5**	0
security	$(d_2,3) \Rightarrow$	3	**7**	1	5	0
	$(d_4,1) \Rightarrow$	3	7	1	**6**	0
	$(d_5,3) \Rightarrow$	3	7	1	6	**3**

Further Improving Efficiency

- Caching (e.g., query results, list of inverted index)
- Keep only the most promising accumulators
- Sort the inverted list in decreasing order of weights and fetch only N entries with the highest weights
- Scaling up to the Web-scale (more about this later)

Some Text Retrieval Toolkits

- Smart (Cornell) (no longer popular)
- Lucene (http://lucene.apache.org/)
- Lemur/Indri (http://lemurproject.org/)
- Galago
- Terrier (http://terrier.org/)
- MeTA (http://meta-toolkit.github.io/meta/)

Questions?