
CS317

Information Retrieval

Week 02

Muhammad Rafi

Feb 06, 2020

Phrase and Positional
Indexing

Sec. 2.4

Phrase queries

- Want to be able to answer queries such as “**stanford university**” – as a phrase
 - Thus the sentence “*I went to university at Stanford*” is not a match.
 - The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
 - Many more queries are *implicit phrase queries*
 - For this, it no longer suffices to store only `<term : docs>` entries
-

Sec. 2.4.1

Solution 1: Bi-word indexes

- Index every consecutive pair of terms in the text as a phrase
 - For example the text “Friends, Romans, Countrymen” would generate the biwords
 - **friends romans**
 - **romans countrymen**
 - Each of these biwords is now a dictionary term
 - Two-word phrase query-processing is now immediate.
-

Sec. 2.4.1

Longer phrase queries

- Longer phrases are processed as we did with wild-cards:
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

stanford university AND university palo AND palo alto

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!

Sec. 2.4.1

Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Call any string of terms of the form NX*N an extended biword.
 - Each such extended biword is now made a term in the dictionary.
- Example: ***coins are in pocket***

N X X N
- Query processing: parse it into N's and X's
 - Segment query into enhanced biwords
 - Look up in index: ***coin pocket***

Sec. 2.4.1

Issues for biword indexes

- False positives, as noted before
 - Index blowup due to bigger dictionary
 - Infeasible for more than biwords, big even for them
 - Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy
-

Sec. 2.4.2

Solution 2: Positional indexes

- In the postings, store for each ***term*** the position(s) in which tokens of it appear:

<***term***, number of docs containing ***term***;
doc1: position1, position2 ... ;
doc2: position1, position2 ... ;
etc.>

Sec. 2.4.2

Positional index example

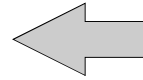
<**be**: 993427;

1: 7, 18, 33, 72, 86, 231;

2: 3, 149;

4: 17, 191, 291, 430, 434;

5: 363, 367, ...>



Which of docs 1,2,4,5
could contain "**to be
or not to be**"?

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Sec. 2.4.2

Processing a phrase query

- Extract inverted index entries for each distinct term: **to**, **be**, **or**, **not**.
- Merge their *doc:position* lists to enumerate all positions with "**to be or not to be**".
 - **to**:
 - 2:1,17,74,222,551; 4:8,16,190,429,433;
7:13,23,191; ...
 - **be**:
 - 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...
- Same general method for proximity searches

Positional Intersect

```

POSITIONALINTERSECT( $p_1, p_2, k$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $I \leftarrow \langle \rangle$ 
5           $pp_1 \leftarrow \text{positions}(p_1)$ 
6           $pp_2 \leftarrow \text{positions}(p_2)$ 
7          while  $pp_1 \neq \text{NIL}$ 
8              do while  $pp_2 \neq \text{NIL}$ 
9                  do if  $|\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k$ 
10                     then  $\text{ADD}(I, \text{pos}(pp_2))$ 
11                     else if  $\text{pos}(pp_2) > \text{pos}(pp_1)$ 
12                         then break
13                      $pp_2 \leftarrow \text{next}(pp_2)$ 
14                 while  $I \neq \langle \rangle$  and  $|I[0] - \text{pos}(pp_1)| > k$ 
15                     do  $\text{DELETE}(I[0])$ 
16                     for each  $ps \in I$ 
17                         do  $\text{ADD}(answer, (\text{docID}(p_1), \text{pos}(pp_1), ps))$ 
18                      $pp_1 \leftarrow \text{next}(pp_1)$ 
19              $p_1 \leftarrow \text{next}(p_1)$ 
20              $p_2 \leftarrow \text{next}(p_2)$ 
21     else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
22         then  $p_1 \leftarrow \text{next}(p_1)$ 
23     else  $p_2 \leftarrow \text{next}(p_2)$ 
24 return  $answer$ 

```

Sec. 2.4.2

Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
 - Again, here, / k means “within k words of”.
- Clearly, positional indexes can be used for such queries; biword indexes cannot.
- Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of k ?
 - This is a little tricky to do correctly and efficiently
 - See Figure 2.12 of IIR
 - There’s likely to be a problem on it!


Sec. 2.4.2

Positional index size

- You can compress position values/offsets: we'll talk about that in lecture 5
 - Nevertheless, a positional index expands postings storage *substantially*
 - Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries ... whether used explicitly or implicitly in a ranking retrieval system.
-

Sec. 2.4.2

Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
 - Average web page has <1000 terms  Why?
 - SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

Document size	Postings	Positional postings
1 000	1	1
100,000	1	100

Sec. 2.4.2

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
 - Positional index size 35–50% of volume of original text
 - Caveat: all of this holds for “English-like” languages
-

Sec. 2.4.3

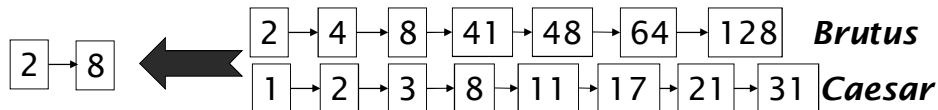
Combination schemes

- These two approaches can be profitably combined
 - For particular phrases (“**Michael Jackson**”, “**Britney Spears**”) it is inefficient to keep on merging positional postings lists
 - Even more so for phrases like “**The Who**”
 - Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
 - A typical web query mixture was executed in $\frac{1}{4}$ of the time of using just a positional index
 - It required 26% more space
-

Sec. 2.3

Recall basic merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries



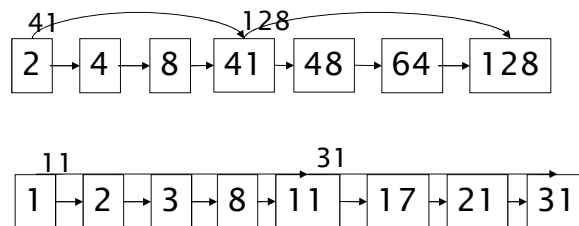
If the list lengths are m and n , the merge takes $O(m+n)$ operations.

Can we do better?

Yes (if the index isn't changing too fast).

Sec. 2.3

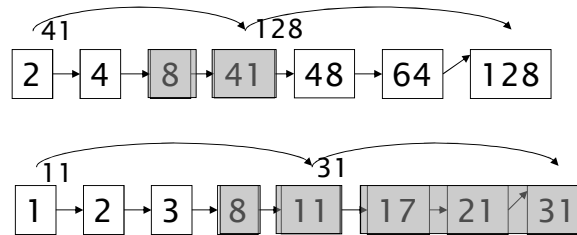
Augment postings with skip pointers (at indexing time)



- Why?
 - To skip postings that will not figure in the search results.
 - How?
 - Where do we place skip pointers?
-

Sec. 2.3

Query processing with skip pointers



Suppose we've stepped through the lists until we process 8 on each list. We match it and advance.

We then have 41 and 11 on the lower. 11 is smaller. But the skip successor of 11 on the lower list is 31, so we can skip ahead past the intervening postings.

Faster Skip Lists

```

INTERSECTWITHSKIPS( $p_1, p_2$ )
1   $answer \leftarrow \{\}$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $docID(p_1) = docID(p_2)$ 
4      then ADD( $answer, docID(p_1)$ )
5       $p_1 \leftarrow next(p_1)$ 
6       $p_2 \leftarrow next(p_2)$ 
7  else if  $docID(p_1) < docID(p_2)$ 
8      then if hasSkip( $p_1$ ) and ( $docID(skip(p_1)) \leq docID(p_2)$ )
9          then while hasSkip( $p_1$ ) and ( $docID(skip(p_1)) \leq docID(p_2)$ )
10             do  $p_1 \leftarrow skip(p_1)$ 
11          else  $p_1 \leftarrow next(p_1)$ 
12      else if hasSkip( $p_2$ ) and ( $docID(skip(p_2)) \leq docID(p_1)$ )
13          then while hasSkip( $p_2$ ) and ( $docID(skip(p_2)) \leq docID(p_1)$ )
14             do  $p_2 \leftarrow skip(p_2)$ 
15          else  $p_2 \leftarrow next(p_2)$ 
16  return  $answer$ 

```

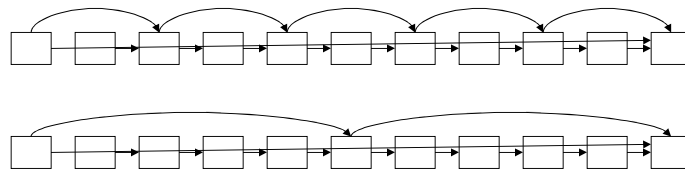
► Figure 2.10 Postings lists intersection with skip pointers.

Sec. 2.3

Where do we place skips?

■ Tradeoff:

- More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers.
- Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.



Sec. 2.3

Placing skips

- Simple heuristic: for postings of length L , use \sqrt{L} evenly-spaced skip pointers [Moffat and Zobel 1996]
- This ignores the distribution of query terms.
- Easy if the index is relatively static; harder if L keeps changing because of updates.
- This definitely used to help; with modern hardware it may not unless you're memory-based [Bahle et al. 2002]
 - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!