Introduction to

Information Retrieval

Lectures 4: Skip Pointers, Phrase Queries, Positional Indexing

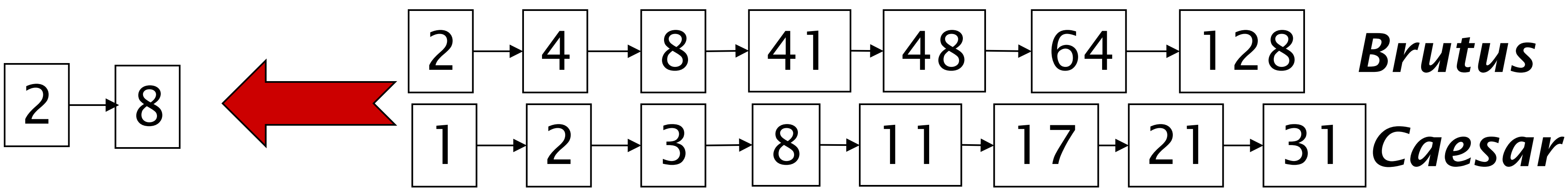
Introduction to

Information Retrieval

Faster postings merges:
Skip pointers/Skip lists

Recall basic merge

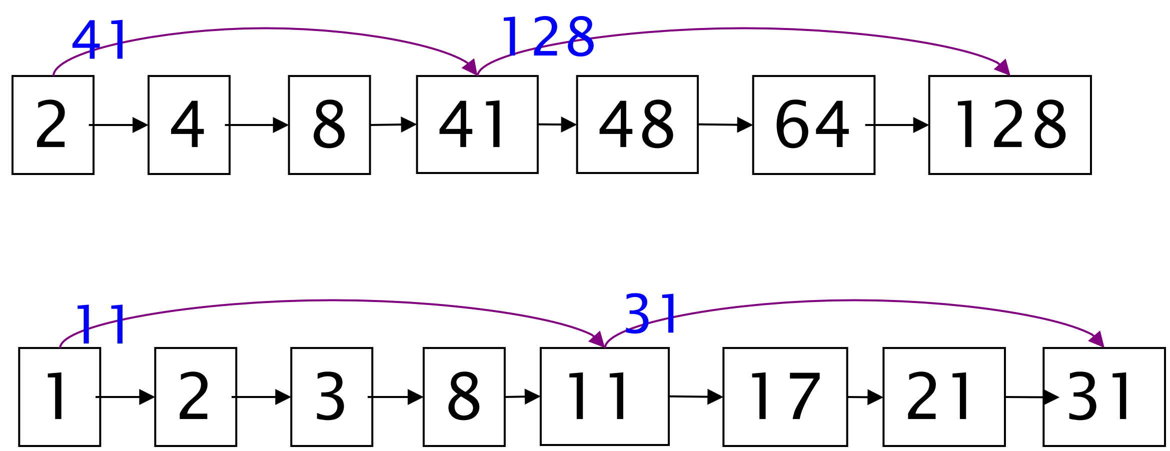
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



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

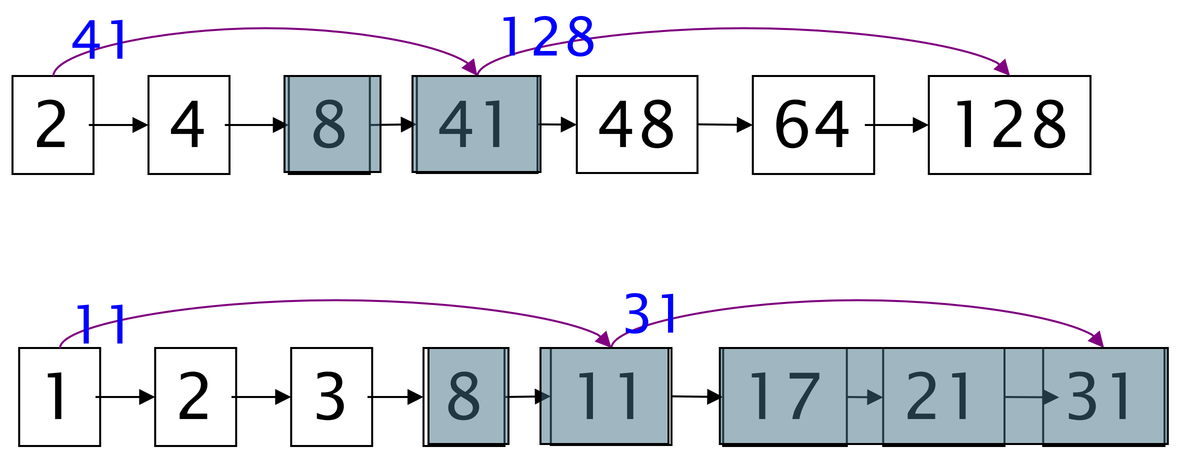
Can we do better?
Yes (if the index isn't changing too fast).

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?

Query processing with skip pointers



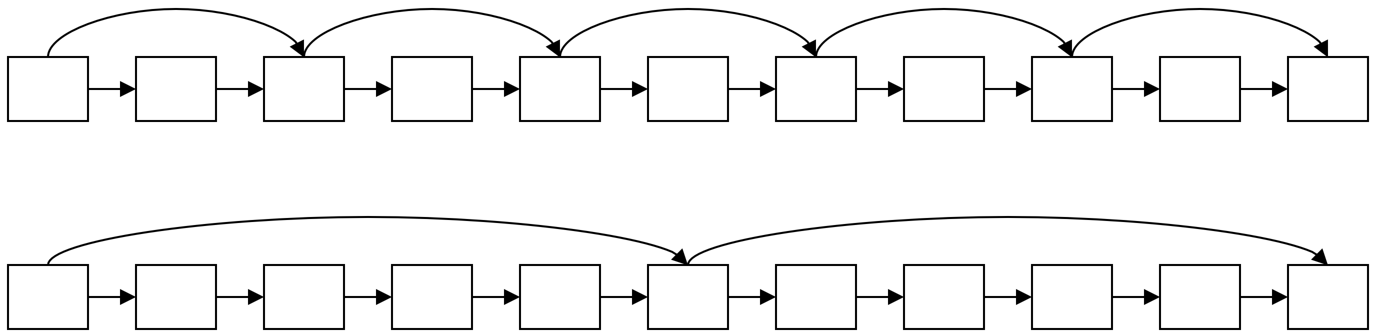
Suppose we've stepped through the lists until we process **8** on each list. We match it and advance.

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.

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



Placing skips

- Simple heuristic: for postings of length *L*, use $\sqrt{L}$ evenly-spaced skip pointers [Moffat and Zobel 1996]
- 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!

Introduction to

Information Retrieval

Handling phrase queries

Phrase queries

- We want to answer a query such as [stanford university] – as a phrase.
- Thus *The inventor Stanford Ovshinsky never went to university* should not be a match.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists for terms.
- Two ways of extending the inverted index:
  - biword index
  - positional index

Biword indexes

- Index every consecutive pair of terms in the text as a phrase.
- For example, *Friends, Romans, Countrymen* would generate two biwords: *"friends romans"* and *"romans countrymen"*
- Each of these biwords is now a vocabulary term.
- Two-word phrases can now easily be answered.

Longer phrase queries

- A long phrase like *"stanford university palo alto"* can be represented as the Boolean query "STANFORD UNIVERSITY" AND "UNIVERSITY PALO" AND "PALO ALTO"
- Does this always guarantee the correct match? -- We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase.
- What about phrases like, "*abolition of slavery*"?

Extended biwords

- Parse each document and perform part-of-speech tagging
- Bucket the terms into (say) nouns (N) and articles/prepositions (X)
- Now deem any string of terms of the form NX*N to be an *extended biword*
- Examples: catcher in the rye

  N X X N

  king of Denmark

  N X N
- Include extended biwords in the term vocabulary
- Queries are processed accordingly

Issues with biword indexes

- Why are biword indexes rarely used?
- False positives, as noted above
- Index blowup due to very large term vocabulary

- *What can be an alternative?*

Positional indexes

- Positional indexes are a more efficient alternative to biword indexes.
- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions

Positional indexes: Example

Query: *"to$_1$ be$_2$ or$_3$ not$_4$ to$_5$ be$_6$"*

TO, 993427:

‹ 1: ‹7, 18, 33, 72, 86, 231›;

2: ‹1, 17, 74, 222, 255›;

4: ‹8, 16, 190, 429, 433›;

5: ‹363, 367›;

7: ‹13, 23, 191›; . . . ›

BE, 178239:

‹ 1: ‹17, 25›;

4: ‹17, 191, 291, 430, 434›;

5: ‹14, 19, 101›; . . . ›

Proximity search

- We just saw how to use a positional index for phrase searches.
- *Can we also use it for proximity search?*
- For example: employment /4 place
- Find all documents that contain EMPLOYMENT and PLACE within 4 words of each other.
  - *Employment agencies that place healthcare workers are seeing growth* is a hit.
  - *Employment agencies that have learned to adapt now place healthcare workers* is not a hit.

Proximity search

- Use the positional index
- Simplest algorithm: look at cross-product of positions of (i) EMPLOYMENT in document and (ii) PLACE in document
- Very inefficient for frequent words, especially stop words
- Note that we want to return the actual matching positions, not just a list of documents.

Combination scheme

- Biword indexes and positional indexes can be profitably combined.
- Many biwords are extremely frequent: Michael Jackson etc
- For these biwords, increased speed compared to positional postings intersection is substantial.
- Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme – *Next Word Index*. Faster than a positional index, at a cost of 26% more space for index.