Positional and distributed indexes

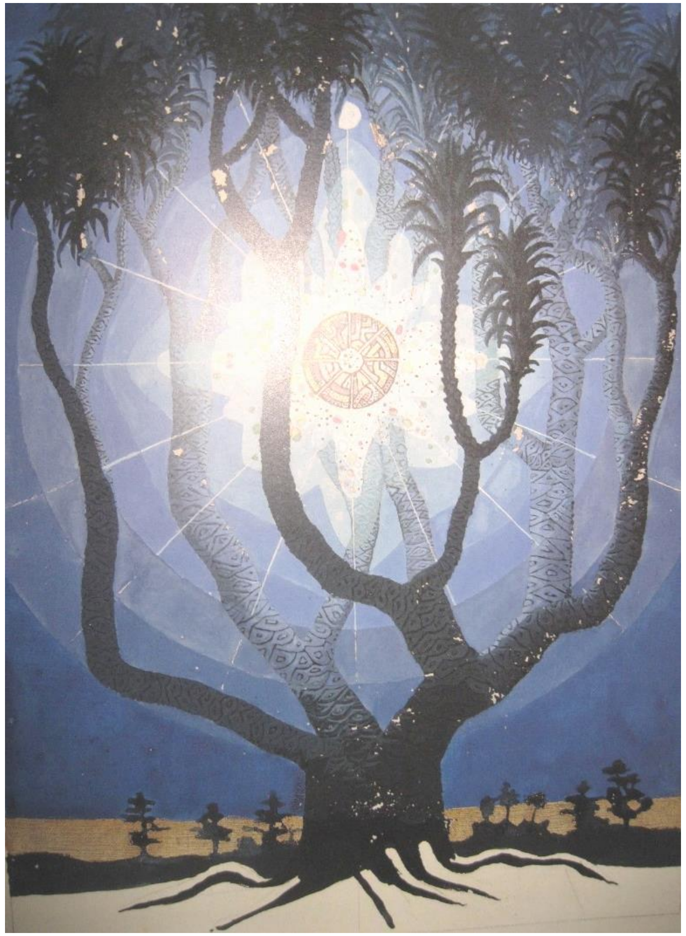
Phrase and proximity querying, large-scale indexing

Acknowledgments: **Standford NLP group**

Rui Henriques rmch@tecnico.ulisboa.pt

PRI 2023/24

Outline



- **Phrase and proximity queries**
  - **Biword indexes**
    - extension to long phrase queries
  - **Positional indexes**
    - processing phrase and proximity queries
    - positional index size
  - **Integrative scheme**
- **Large-scale libraries**
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - index compression

Phrase and proximity queries

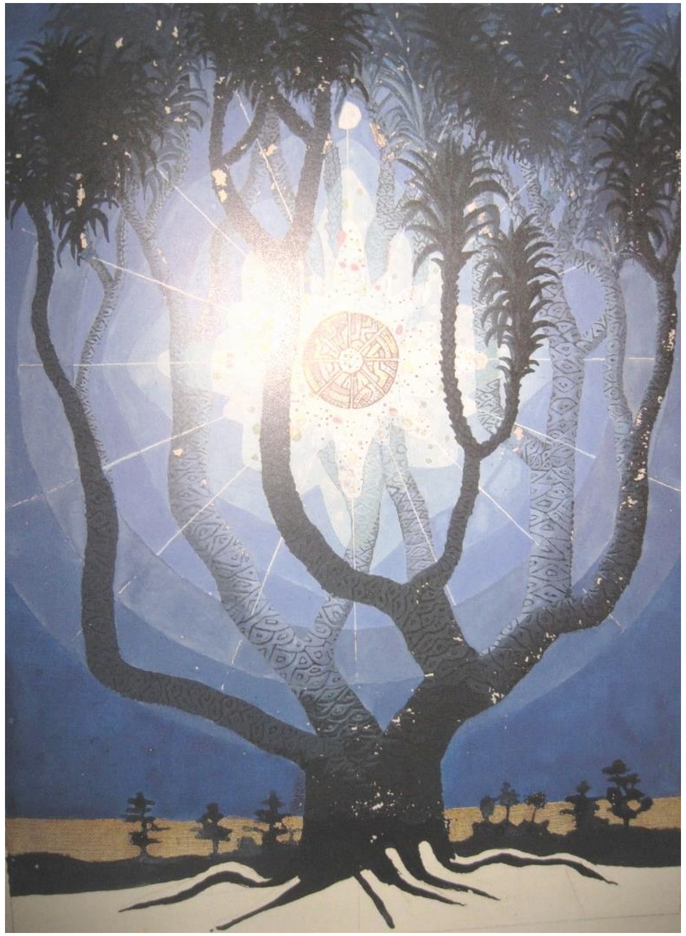
- We want to be able to answer queries such as "*stanford university*" as a phrase
  - thus that "*The inventor Stanford Ovshinsky never went to university*" is **not** a match
- Concept of **phrase queries** easily understood by users
  - one of the few query commands that globally works!
  - about 10% of web queries are phrase queries
- In fact: many more queries are *implicitly* phrase queries

- **Proximity queries** also very common in many domains (recall WestLaw)
  - *example*: 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

- **Problem**: we cannot do this with our earlier inverted index

  < term : docs > entries

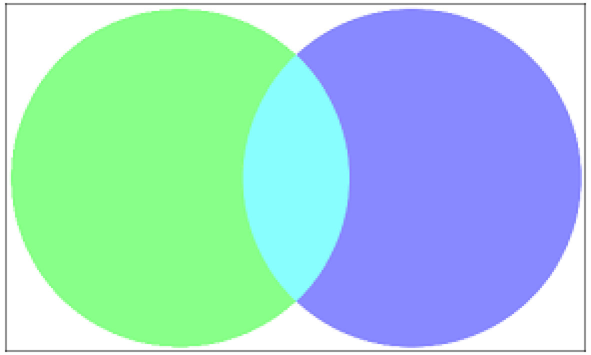
Outline



- Phrase and proximity queries
  - **Biword indexes**
    - **extension to long phrase queries**
  - Positional indexes
    - processing phrase and proximity queries
    - positional index size
  - Integrative scheme
- Large-scale libraries
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - index compression

Biword indexes: a first attempt

- Index every consecutive pair of terms in the text as a phrase
- Example: "*Friends, Romans, Countrymen*" will generate two biwords
  - *friends romans*
  - *romans countrymen*
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate



- Problems?
  - false positives
  - bigger dictionary
    - infeasible for more than biwords (can be hard even for them)

Longer phrase queries

- Longer phrases can be processed by breaking them down

stanford university palo alto

broken into the Boolean query on biwords:

stanford university *AND* ***university palo*** *AND* ***palo alto***

can have false positives!

Without the raw document text…

– we cannot verify that the docs matching the above Boolean query do contain the long phrase

Extended biwords

- Parse each document and perform *part-of-speech tagging*
- Bucket the terms into: *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*

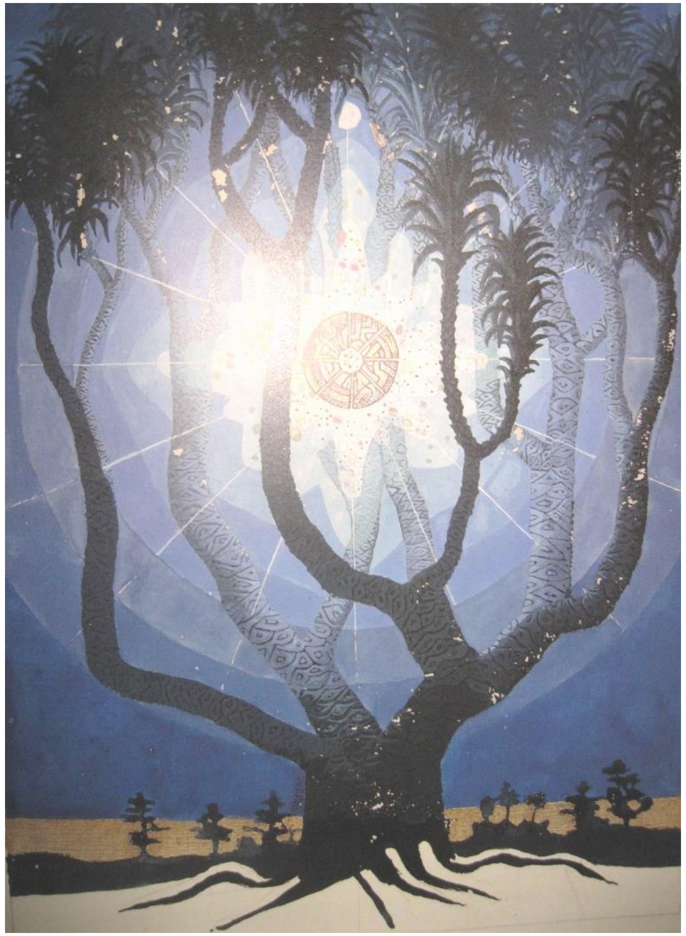
  N X X N

  king of Denmark

  N X N

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

Outline



- Phrase and proximity queries
  - Biword indexes
    - extension to long phrase queries
  - **Positional indexes**
    - **processing phrase and proximity queries**
    - **positional index size**
  - Integrative scheme
- Large-scale libraries
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - index compression

Alternative solution: positional indexes

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

<***term****,* number of docs containing ***term***;

doc1: position1, position2 … ;

doc2: position1, position2 … ;

etc.>

– positional indexes are a **more efficient** alternative to biword indexes
  - – postings lists in a *non-positional 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"*
- **Index**:

  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›; . . . ›

Document 4 is a match!

- 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"*

Proximity queries

- We previously saw that proximity queries are essential in many domains (e.g. WestLaw)
  - LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
    – Again, here, /*k* means “within *k* words of”.

- Ah! 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 (Figure 2.12 of *IIR*)

Proximity search

- Use the positional index
- Example: all documents that contain EMPLOYMENT and PLACE within 4 words of each other

- *Simplest algorithm*:
  - look at cross-product of positions of (i) EMPLOYMENT in document and (ii) PLACE in document

- ***Problem***: very inefficient for frequent words, especially stop words
- We may as well want to return the actual matching positions, not just a list of documents

- ***Solution***: positional intersect

```
POSITIONALINTERSECT(p1, p2, k)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then l ← ⟨ ⟩
           pp1 ← positions(p1)
           pp2 ← positions(p2)
           while pp1 ≠ NIL
           do while pp2 ≠ NIL
                 do if |pos(pp1) − pos(pp2)| ≤ k
                       then ADD(l, pos(pp2))
                       else if pos(pp2) > pos(pp1)
                               then break
                    pp2 ← next(pp2)
                 while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
                 do DELETE(l[0])
                 for each ps ∈ l
                 do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
                 pp1 ← next(pp1)
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

Positional index size

- A positional index expands postings storage *substantially*
  - even though indices can be compressed

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

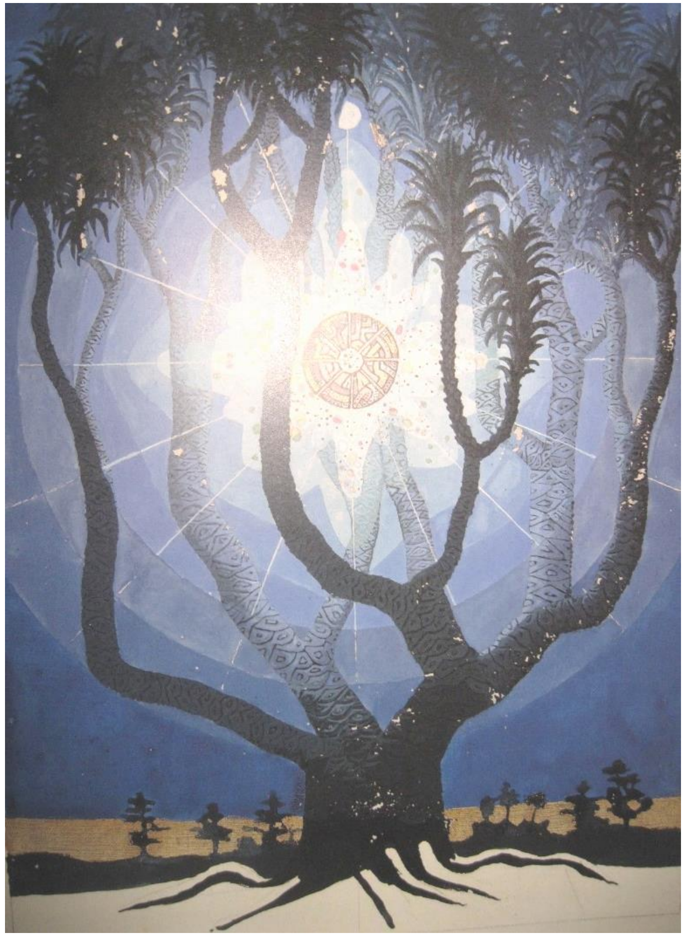
Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Still…

- 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*: this holds for "English-like" languages

- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries …
  - not only for simple retrieval but very important for **ranking** retrieval system
    - score higher keywords within query that appear nearby each other in documents

Outline



- Phrase and proximity queries
  - Biword indexes
    - extension to long phrase queries
  - Positional indexes
    - processing phrase and proximity queries
    - positional index size
  - **Integrative scheme**
- Large-scale libraries
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - index compression

Integrative schemes

- Biword indexes and positional indexes can be profitably combined
  - many biwords are extremely frequent: Michael Jackson, Britney Spears etc
  - for these particular phrases it is inefficient to keep on merging positional postings lists
    - even more so for phrases like *"The Who"*

- ***Solution***: further include frequent biwords as vocabulary terms in the index
  - do all other phrases by positional intersection

- Williams et al. (2004) evaluate such mixed indexing scheme
  - a typical web query mixture was executed in ¼ of the time of using just a positional index
  - it required 26% more space than having a positional index alone

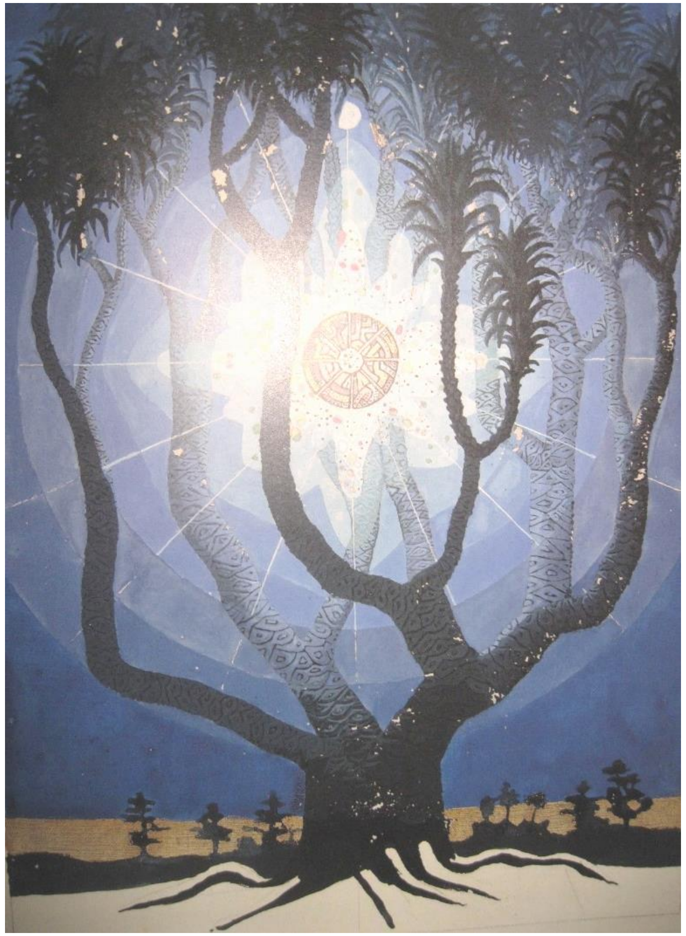
Remarks

Positional queries on Google

- Google uses a **ranking schema** considers that values documents where there is **proximity between terms** in the query
  - what could be an adequate scoring index?
- can you demonstrate on Google that phrase queries are more expensive than Boolean queries?
  - list two reasons for why that is so

- ***Take aways***
  - positional indexes
    - come with a price on index space, yet pervasive nowadays
    - support efficient proximity searches based on positional intersection
  - biword terms further considered to boost phrase queries

Outline



- Phrase and proximity queries
  - Biword indexes
    - extension to long phrase queries
  - Positional indexes
    - processing phrase and proximity queries
    - positional index size
  - Integrative scheme
- **Large-scale libraries**
  - **disk-aware and distributed indexing**
  - boosting querying: skip pointers
  - index compression

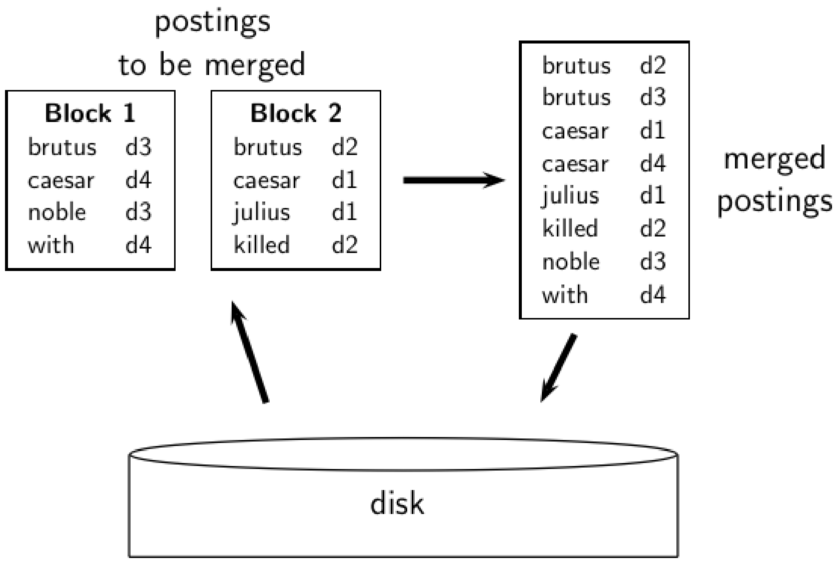
Scaling index construction

- In-memory index construction?
  - BBC News can be indexed in memory… yet most libraries are massive, impossible to index in memory
    - *New York Times* provides an index of >150 years of newswire
    - the Web

- **Implications?**
  - design IR systems (indexing and querying mechanism) depend on the characteristics of hardware
  - **hardware constraints**
    - access to data in **memory** is *much* faster than access to data on disk/SSD
    - **disk seeks**: no data is transferred from disk while the disk head is being positioned
    - **disk I/O** is **block-based**: reading/writing entire block (8KB to 256KB) takes same time as smaller chunks
  - solution? **disk-aware indexing and querying**

Indexing on disk? *External sorting*

- Same indexing algorithm but using disk instead of memory?
- No. Use instead *blocked indexing*. Idea:
  - assuming each posting has approximately 20 bytes (4+4+4+2*4: termID, docID, doc frequency, and some positions)
  - one disk block can have up to 10,000,000 of such postings
  - for each block: accumulate term-related postings and write them to disk ensuring long sorted order
  - periodic disk reindexing necessary for dynamic collections for an effective block-aware querying



- What about massive collections that may not fit on disk?

Distributed indexing

- For **web-scale indexing** (*don't try this at home*!)
  - distributed computing cluster
  - individual machines are *fault-prone* (can unpredictably slow down or fail)
    - exercise: if in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime. What is the uptime of the system? [63%] How many servers fail on average per minute for an installation of 1 million servers?

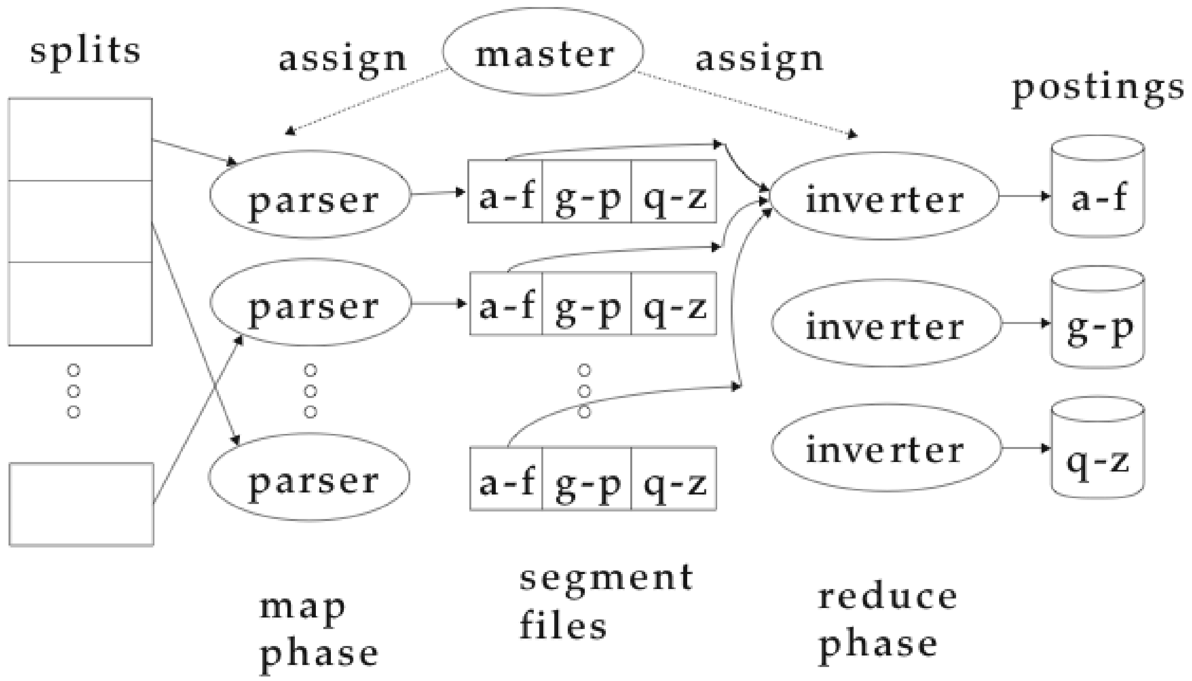
- **Web search data centers** (Google, Bing, Baidu):
  - data center machines distributed around the world
  - *estimate*: Google **>1M** servers, **>7M** processors/cores
  - how do we exploit a pool of machines?

Distributed indexing

- Maintain **master** machines directing the indexing job (considered *safe*)
- Break up indexing into sets of (parallel) tasks that are assigned to an idle machine from a pool
  - we will use two sets of **parallel tasks** (parsers and inverters)

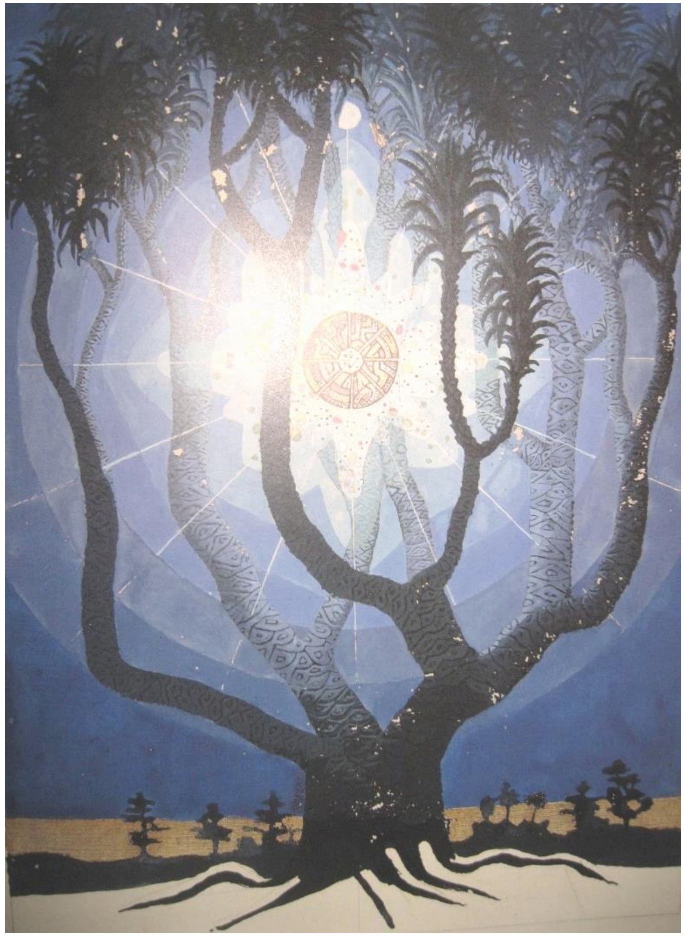
    … to break the fetched documents into *splits* (corresponding to disk blocks)
    - **parsers**
      - reads a document at a time and emits (term, doc) pairs
      - writes pairs into *j* partitions
    - **inverters**
      - collects all (term,doc) pairs (= postings) for one term-partition
      - sorts and writes to postings lists
- The described index construction algorithm is implemented as an instance of ***MapReduce***
  - *MapReduce* is robust and conceptually simple framework for distributed computing

Distributed indexing



- what information is contained in the task description that the master gives to a parser?
- what information does the parser report back to the master upon completion of the task?
- what information is contained in the task description that the master gives to an inverter?
- what information does the inverter report back to the master upon completion of the task?

Outline



- Phrase and proximity queries
  - Biword indexes
    - extension to long phrase queries
  - Positional indexes
    - processing phrase and proximity queries
    - positional index size
  - Integrative scheme
- **Large-scale libraries**
  - disk-aware and distributed indexing
  - **boosting querying: skip pointers**
  - index compression

Can we boost queries?

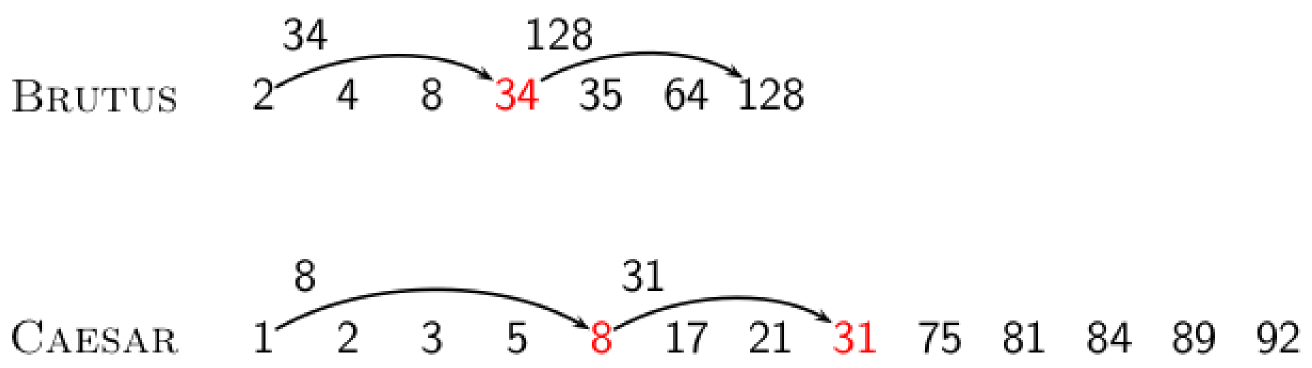
Let us consider **Boolean querying**

– as covered, advanced search engines, such as Google, rely on tolerant Boolean queries

- Querying in linear time of the size of postings lists
  - large collections ⇒ large posting lists 😐
  - some postings lists contain several million entries
- Can we improve **querying time**?

- Yes!
  - **skip pointers** allow us to skip postings that will not figure in the search results
    - where do we put skip pointers?
    - how do we make sure intersection results are correct?

Query processing with skip pointers



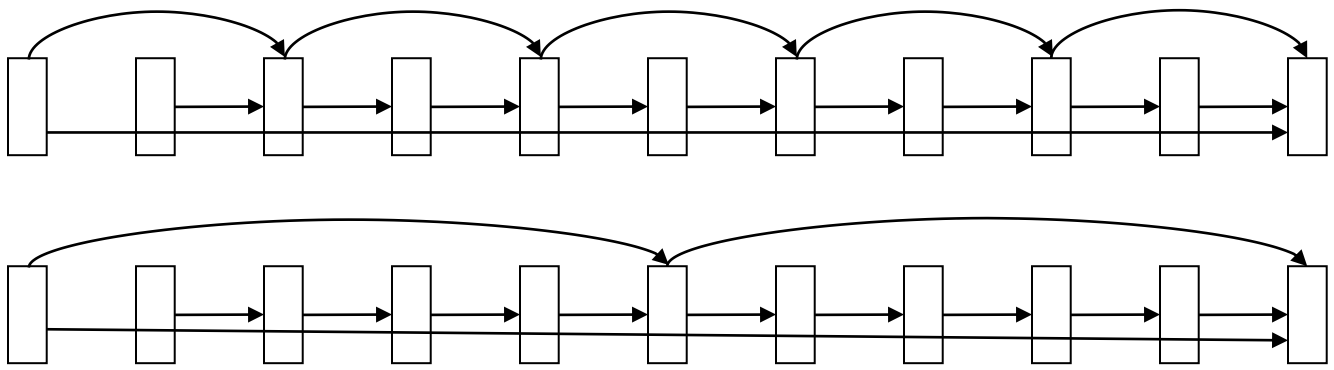
Suppose we have stepped through the lists until we process 8 on each list.
We match it and advance. We then have 34 on the upper. But the skip successor of 8 on the lower list is 31, so we can skip ahead past the intervening postings.

Where do we place skips?

Tradeoff

– More skips → shorter skip spans ⇒ more likely to skip, but many comparisons to skip pointers

– Fewer skips → few pointer comparison, but long skip spans ⇒ few successful skips



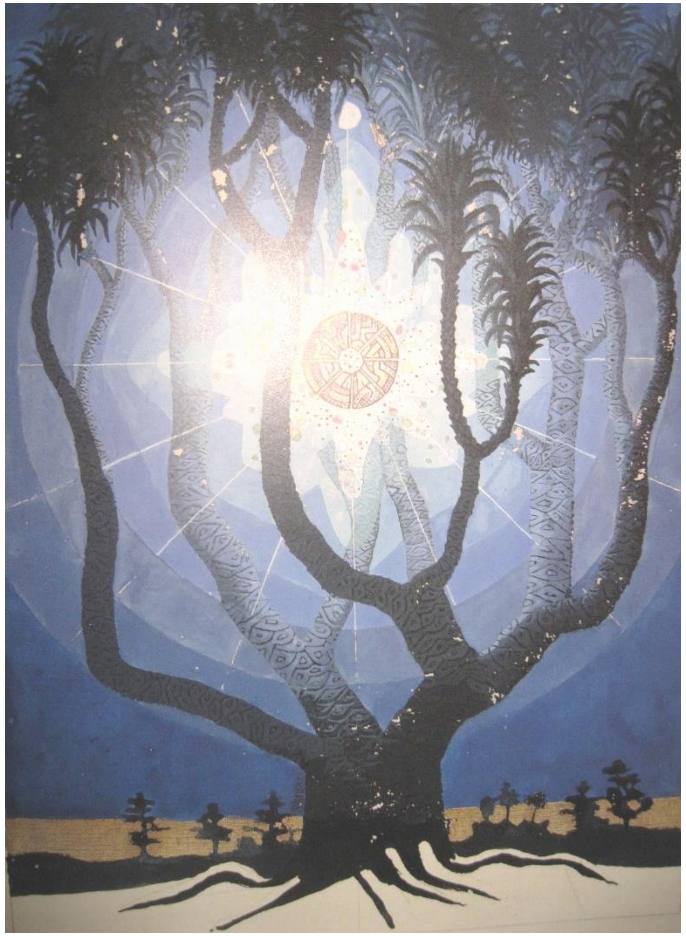
Query processing with skip pointers

```
IntersectWithSkips(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then Add(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                      then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                              do p1 ← skip(p1)
                      else p1 ← next(p1)
              else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                      then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                              do p2 ← skip(p2)
                      else p2 ← next(p2)
return answer
```

Placing skips

- *simple heuristic*: for postings of length L, use $\sqrt{L}$ evenly-spaced pointers [Moffat]
  - ignores the distribution of query terms
  - easy if the index is relatively static; harder if L changes due to updates

Outline



- Phrase and proximity queries
  - Biword indexes
    - extension to long phrase queries
  - Positional indexes
    - processing phrase and proximity queries
    - positional index size
  - Integrative scheme
- **Large-scale libraries**
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - **index compression**

Why compressing our index?

In general…

- use less disk space
- keep more data in memory: increasing index access speed
- increase speed by reducing need to transfer data between disk and memory
  - [*read compressed data* | *decompress*] commonly faster than [*read uncompressed data*]

In IR systems

- compressed dictionary
  - search begins with the dictionary!
  - essential for embedded/mobile devices that may have very little memory
    - memory footprint competition with other applications
  - make it small enough to keep it in main memory, and allow more postings available
- compressed postings
  - postings are the large part of the index, thus even more determinant!
  - large search engines keep a significant part of the postings in memory

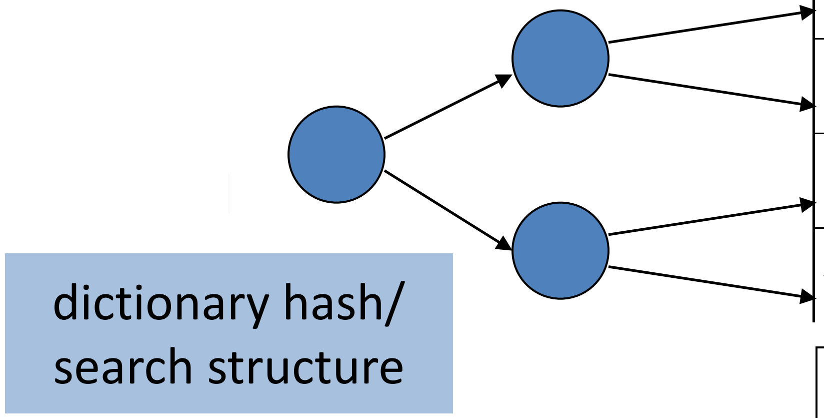
Lossless vs. lossy compression

- **Lossless** compression
  - all information is preserved (*default* option in IR)
- **Lossy** compression
  - discard some information

- Yet…
  - **text processing** steps can be viewed as lossy compression
    - case folding, stop words, stemming, number elimination…
  - **pruning postings entries** is very common (remove documents where term has low relevance)
    - almost no loss in perceived quality for top *k* postings list!

Dictionary: fixed-width terms are wasteful

- Most of the bytes in the *term* column are wasted (28 bytes per term)
  - 1M terms; 28 bytes/term = 28 MB (discounting statistics such as document frequency, etc.)
  - and we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*!
- Written English averages ~4.5 characters/word (short words dominate token counts)
  - exercise: why is/is not this the number to use for estimating the dictionary size?
- Average dictionary word in English: <8 characters
  - How do we use 8 characters per dictionary term?



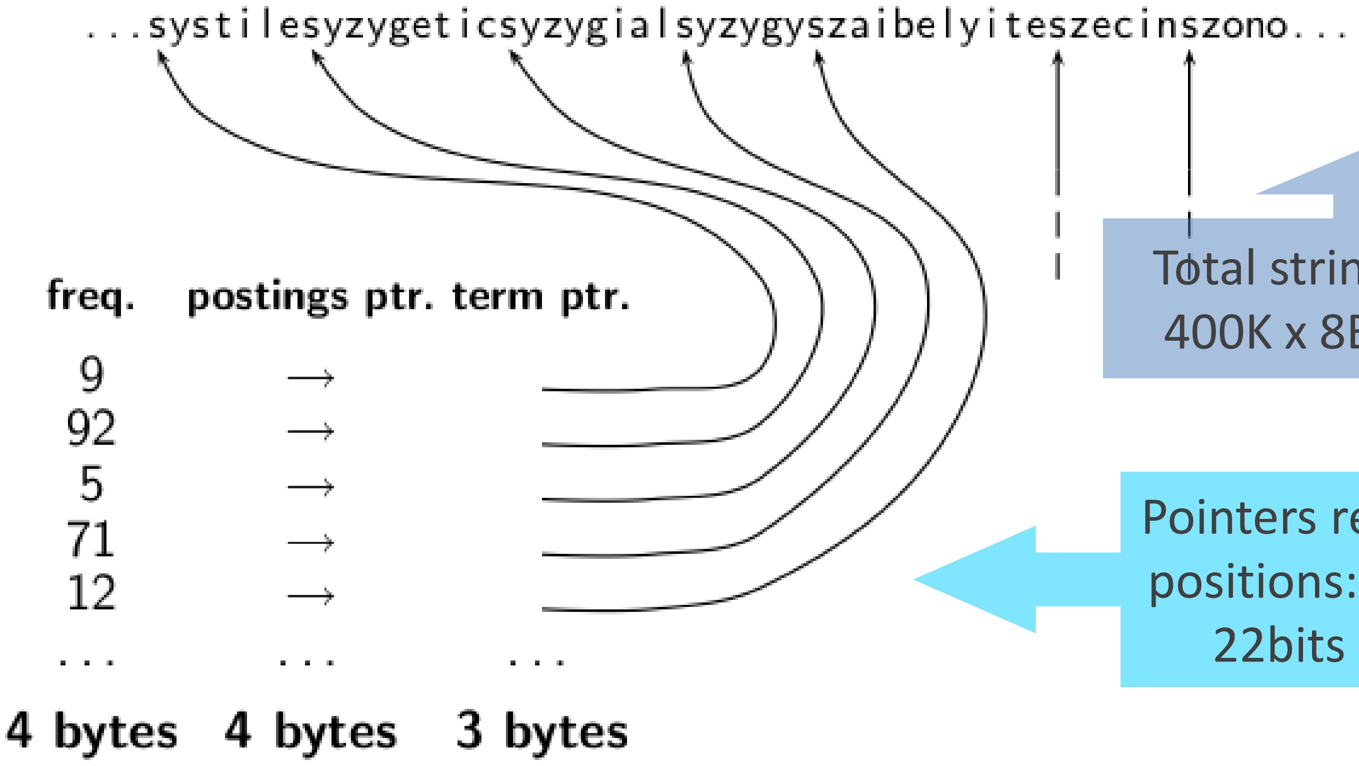
terms	freq.	postings ptr.
a	656,265	→
aachen	65	→
….	….	
zulu	221	→

28 bytes

4 bytes each

Compressing term list: dictionary as a *string*

- Store dictionary as a (long) string of characters
  - pointer to next word shows end of current word
  - save up to 60% of dictionary space



Total string length = 400K x 8B = 3.2MB

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

Postings compression: efficient typing

- The postings file is much larger than the dictionary (factor of at least 10)!
- How to store each posting compactly? Let a posting for *current purposes* be a docID
- We commonly use 64 bits per docID when using 8 byte integers
  - yet, considering a collection with 1 million documents…

- **Savings using efficient types**
  - alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID
  - this means 3 bytes! Reduce size for more than half!
  - and we can still use far fewer than 20 bits per docID!

Postings compression: gaps

- Until now: we store the list of docs containing a term in increasing order of docID
  - e.g. **COMPUTER** : 283154, 283159, 283202, . . .
- What if we store ***gaps***?
  - 283159-283154=5, 283202-283154=43
  - example postings list using gaps: **COMPUTER**: 283154, 5, 43, . . .
  - Gaps for frequent terms are small!
    - we can encode small gaps with fewer than 20 bits

	encoding	postings list									
THE	docIDs	. . .		283042		283043		283044		283045	. . .
	gaps				1		1		1		. . .
COMPUTER	docIDs	. . .		283047		283154		283159		283202	. . .
	gaps				107		5		43		. . .
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

- **Two conflicting forces**
  - a term like ***arachnocentric*** occurs in maybe one document out of a million
    - we need approximately $\log_2$ 1M ≈ 20 bits/gap entry
  - a term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive
    - in this case we just need ~1 bit/gap entry

- *Intuition*: if the average gap for a term is G, we want to use $\log_2 G$ bits/gap entry
- *Key challenge*: encode every integer (gap) with about as few bits as needed for that integer
  - requires a *variable length encoding*
    - instead of fixing 3 bytes for every integer, we use only 3 bytes for non-frequent terms
  - principle: use few bits for small gaps and more bits for large gaps

Index compression on positional indexes

- Can we extend these **compression principles** to handle:
  - **weighted retrieval?** where postings are accompanied by statis
    - e.g. relevance of term in the document (TF-IDF)
  - **positional information?** where each posting has a list of positi
  - **dynamic collections?** where posting changes

- Yes! The same exact principles
  - efficient typing for statistics
  - variable length positional encoding sensitive to document size
  - change expectations, mixed types and re-compression for dynamic collections

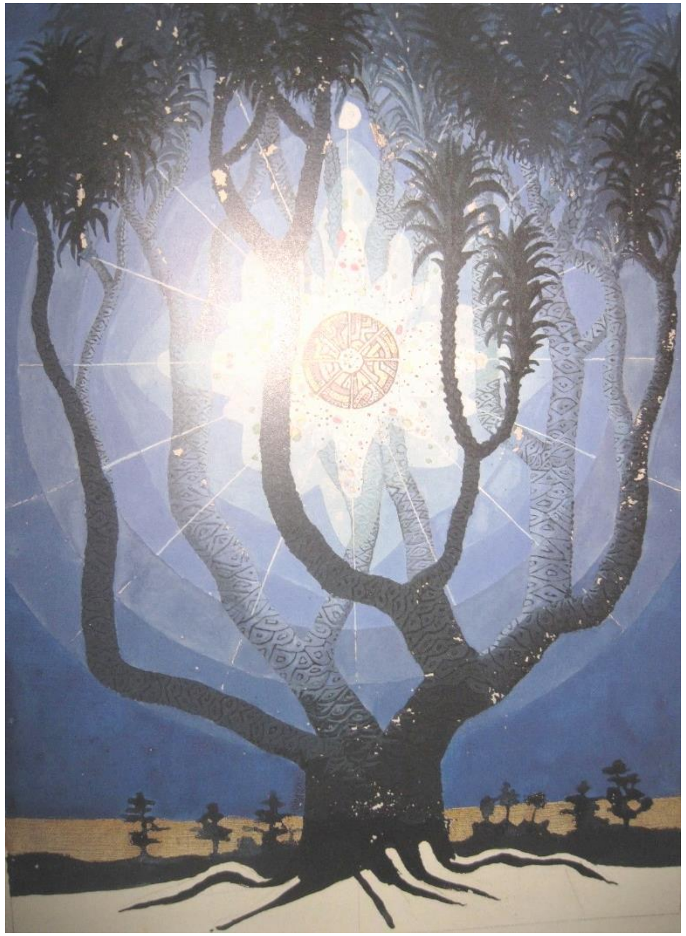
Index compression: last remarks

- We can now create a compact index for highly efficient Boolean retrieval
  - approximately 10% of the total *text size* in the collection
  - example for the RCV1 collection (800.000 documents)

data structure	size in MB
dictionary, fixed-width	11.2
dictionary as a string	**7.6**
collection (text)	1000.0
T/D incidence matrix	40,000.0
postings, uncompressed (64-bit words)	**800.0**
postings, uncompressed (20 bits)	**250.0**
postings, variable byte encoded	**116.0**

- Some pointers on index compression:
  - Introduction to IR, chapter 5
  - F. Scholer, H.E. Williams, J. Zobel. 2002. *Compression of Inverted Indexes For Fast Query Evaluation*
  - V. N. Anh, A. Moffat. 2005. *Inverted Index Compression Using Word-Aligned Binary Codes*

Outline



- Phrase and proximity queries
  - **Biword indexes**
    - extension to long phrase queries
  - **Positional indexes**
    - processing phrase and proximity queries
    - positional index size
  - Integrative scheme
- **Large-scale libraries**
  - disk-aware and distributed indexing
  - boosting querying: skip pointers
  - index compression

Thank You

rmch@tecnico.ulisboa.pt