IRRS: Information Retrieval and Recommender Systems

FIB, Master in Data Science

Slides by Marta Arias, José Luis Balcázar,
Ramon Ferrer-i-Cancho, Ricard Gavaldá
Department of Computer Science, UPC

Fall 2022
http://www.cs.upc.edu/~ir-miri

3. Implementation

Query answering

A bad algorithm:

input query q;
for every document d in database
    check if d matches q;
    if so, add its docid to list L;
output list L (perhaps sorted in some way);

Query processing time should be largely independent of database size.
Probably proportional to answer size.

Central Data Structure

From terms to documents

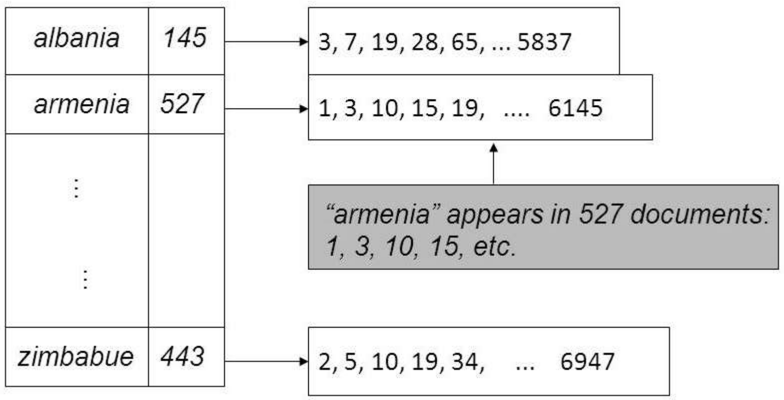
A vocabulary or lexicon or dictionary, usually kept in main memory, maintains all the indexed terms (*set*, *map*. . .); and, besides. . .

The Inverted File

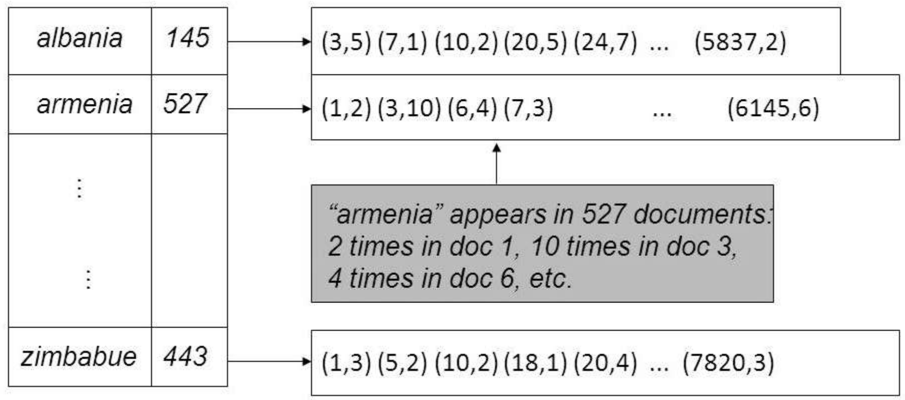
The crucial data structure for indexing.

- A data structure to support the operation:
  - “given term t, get all the documents that contain it”.
- The inverted file must support this operation (and variants) very efficiently.
- Built at preprocessing time, not at query time: can afford to spend some time in its construction.

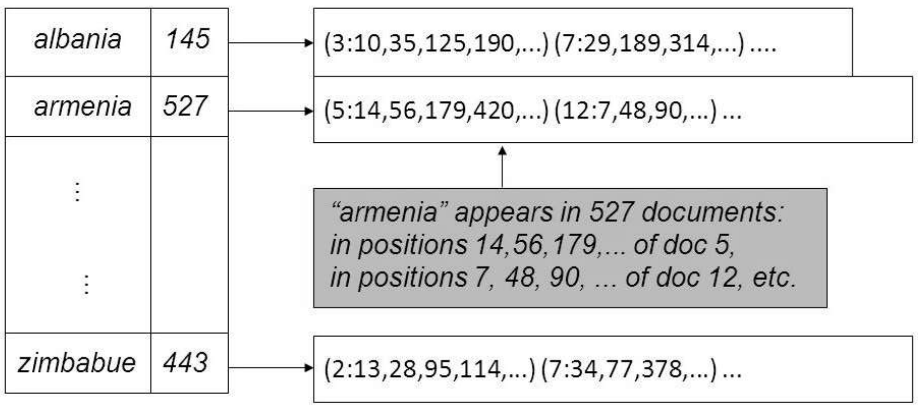
The inverted file: Variant 1



The inverted file: Variant 2



The inverted file: Variant 3



Postings

The inverted file is made of incidence/posting lists

We assign a *document identifier*, docid to each document.
The dictionary may fit in RAM for medium-size applications.

For each indexed term
a posting list: list of docid's (plus maybe other info) where the term appears.

- Wonderful if it fits in memory, but this is unlikely.
- Additionally: posting lists are
  - almost always sorted by *docid*
  - often compressed: minimize info to bring from disk!

Implementation of the Boolean Model, I

Simplest: Traverse posting lists

Conjunctive query: a AND b

- intersect the posting lists of a and b;
- if sorted: can do a merge-like intersection;
- time: order of the sum of the lengths of posting lists.

intersect(input lists $L1$, $L2$, output list L):
  while (not $L1$.end() and not $L2$.end())
    if ($L1$.current() $<$ $L2$.current()) $L1$.advance();
    else if ($L1$.current() $>$ $L2$.current()) $L2$.advance();
    else { L.append($L1$.current());
        $L1$.advance(); $L2$.advance();   }

Implementation of the Boolean Model, II

Simplest

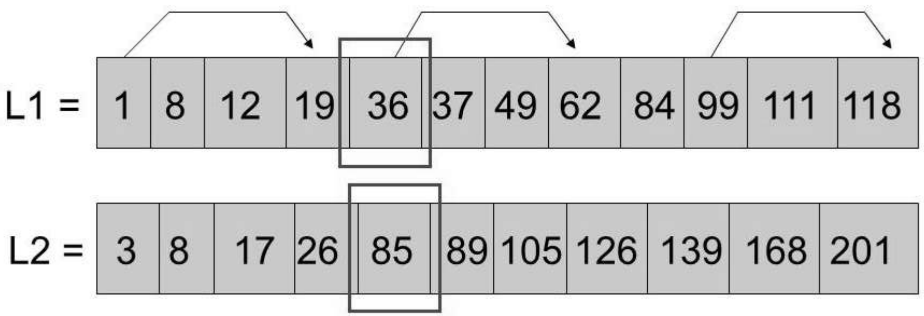
- Similar merge-like union for OR.
  - Time: again order of the sum of lengths of posting lists.
- Alternative: traverse one list and look up every docid in the other via binary search.
  - Time: length of shortest list times log of length of longest.

Example:

- $|L1| = 1000, |L2| = 1000$:
  - sequential scan: 2000 comparisons,
  - binary search: $1000 * 10 = 10,000$ comparisons.
- $|L1| = 100, |L2| = 10,000$:
  - sequential scan: $10,100$ comparisons,
  - binary search: $100 * \log(10,000) = 1400$ comparisons.

Implementation of the Boolean Model, III

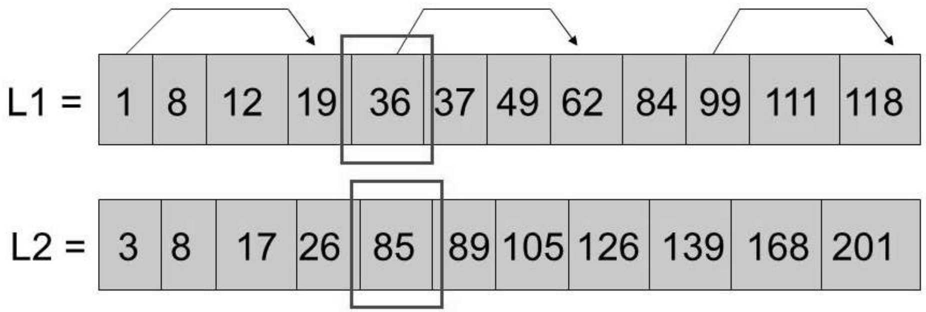
Sublinear time intersection: Skip pointers



- We've merged 1. . . 19 and 3. . . 26.
- We are looking at 36 and 85.
- Since pointer(36)=62 < 85, we can jump to 84 in L1.

Implementation of the Boolean Model, IV

Sublinear time intersection: Skip pointers



- Forward pointer from some elements.
- Either jump to next segment, or search within next segment (once).
- Optimal: in RAM, $\sqrt{|L|}$ pointers of length $\sqrt{|L|}$.
- Difficult to do well, particularly if the lists are on disk.

Query Optimization and Cost Estimation, I

Queries can be evaluated according to different plans
E.g. a AND b AND c as

- (a AND b) AND c
- (b AND c) AND a
- (a AND c) AND b

E.g. (a AND b) OR (a AND c) also as

- a AND (b OR c)

The cost of an execution plan depends on the sizes of the lists and the sizes of intermediate lists.

Query Optimization and Cost Estimation, II

Example

Query: (a AND b) OR (a AND c AND d).

Assume: $|La| = 3000$, $|Lb| = 1000$, $|Lc| = 2500$, $|Ld| = 300$.

- Three intersections plus one union, in the order given: up to cost 13600.
- Instead, ((d AND c) AND a): reduces to up to cost 11400.
- Rewrite to a AND (b OR (c AND d)): reduces to up to cost 8400.

Implementation of the Vectorial Model, I

Problem statement

Fixed similarity measure $sim(d, q)$:

Retrieve
documents d_i which have a similarity to the query q

- either
  - above a threshold sim_{min}, or
  - the top r according to that similarity, or
  - all documents,
- sorted by decreasing similarity to the query q.

Must react very fast (thus, careful to the interplay with disk!), and with a reasonable memory expense.

Implementation of the Vectorial Model, II

Obvious nonsolution

Traverse all the documents, look at their terms in order to compute similarity, filter according to sim_{min}, and sort them. . .

. . . will not work.

Implementation of the Vectorial Model, III
Observations

Most documents include a small proportion of the available terms.

Queries usually include a humanly small number of terms.

Only a very small proportion of the documents will be relevant.

A priori bound r on the size of the answer known.

Inverted file available!

Implementation of the Vectorial Model, IV
Idea

Invert the loops:

- Outer loop on the terms t that appear in the query.
- Inner loop on documents that contain term t.
  - the reason for inverted index!
- Accumulate similarity for visited documents.
- Upon termination, normalize and sort.

Many additional subtleties can be incorporated.

Index compression, I
Why?

A large part of the query-answering time is spent

bringing posting lists from disks to RAM.

Need to minimize amount of bits to transfer.

Index compression schemes use:

- Docid's sorted in increasing order.
- Frequencies usually very small numbers.
- Can do better than e.g. 32 bits for each.

Index compression, II
Why?

A large part of the query-answering time is spent bringing posting lists from disks to RAM.

- Need to minimize amount of bits to transfer.

Easiest is to use "`int` type" to store docid's and frequencies

- 8 bytes, 64 bits per pair
- ... but want/can/need to do much better!

Index compression schemes use:

- Docid's sorted in increasing order.
- Frequencies usually very small numbers.

Index compression, III

Posting list is:

$$term \to [(id_1, f_1), (id_2, f_2), ..., (id_k, f_k)]$$

Can we compress frequencies f_i?:

Yes! Will use *unary self-delimiting* codes because frequencies typically very small

Can we compress docid's id_i?:

Yes! Will use *Gap compression* and *Elias Gamma* codes because docid's are sorted

Index compression, IV

Compressing frequencies

The distribution of frequencies is very biased towards small numbers, i.e., most f_i are very small

- Exercise: can you quantify this using Zipf's law?
- E.g. in files for lab session 1: 68 % is 1, 13 % is 2, 6 % is 3, <13 % is >3, <3 % is >10, 0.6 % is >20.

Unary code

Want encoding scheme that uses few bits for small frequencies

Index compression, V

Compressing frequencies: unary encoding

Unary encoding of x is $\overbrace{111\ ...\ 1}^{x \text{ times}}$

- E.g. $unary(15) = 111111111111111$
- $|unary(x)| = x$
  - typical binary encoding: $|binary(x)| = \log_2(x)$
- variable length encoding

But..

want to encode *lists* of frequencies, where do we cut?

Index compression, VI

Compressing frequencies: self-delimiting unary encoding

- Make 0 act as a separator
- Replace last 1 in each number with a 0
- Example: $[3, 2, 1, 4, 1, 5]$ encoded as $110\ 10\ 0\ 1110\ 0\ 11110$
- This is a *self-delimiting* code: no prefix of a code is a code
- Self-delimiting *implies* unique decoding

Index compression, VII
Compressing frequencies: self-delimiting unary encoding

Recall example from lab session 1: 68 % is 1, 13 % is 2, 6 % is 3, <13 % is >3, <3 % is >10, 0.6 % is >20, the expected length would be (approx)

$$1 * 0.68 + 2 * 0.13 + 3 * 0.06 + 6^{1} * 0.13 = 1.91$$

Unary code works very well

- 1 bit when $f_i = 1$
- 1.3 to 2.5 bits per f_i on real corpuses
- 1 bit per term occurrence in document
  - Easy to estimate memory used!

[1] I put it something greater than 3 as an approximation

Index compression, VIII
Compressing docid's

Gap compression

Instead of compressing $[(id_1, f_1), (id_2, f_2), ..., (id_k, f_k)]$
Compress $[(id_1, f_1), (id_2 - id_1, f_2), ..., (id_k - id_{k-1}, f_k)]$

Example:

$(1000, 1), (1021, 2), (1037, 1), (1056, 4), (1080, 1), (1095, 3)$
compressed to:

$$(1000, 1), (21, 2), (16, 1), (19, 4), (24, 1), (15, 3)$$

Index compression, IX
Compressing docid's

- Fewer bits if gaps are small
- E.g.: $N = 10^6$, $|L| = 10^4$, then average gap is 100
  - So, could use 8 bits instead of 20 (or 32)
- .. but .. this is only on average! *Large gaps do exist*
  - Will need a *variable length, self-delimiting* encoding scheme
- Gaps are not biased towards 1, so unary not a good idea
  - Will use need a *variable length, self-delimiting, binary* encoding scheme

Index compression, X
Compressing docid's: Elias-Gamma code (self-delimiting binary code)

IDEA:
First say how long x is in binary, then send x

Pseudo-code for Elias-Gamma encoding:

- let $w = binary(x)$
- let $y = |w|$
- prepend $y - 1$ zeros to w, and return

Examples:

$EG(1) = 1, EG(2) = 010, EG(3) = 011, EG(4) = 00100, EG(20) = 000010100$

Index compression, XI

Compressing docid's: Elias-Gamma code (self-delimiting binary code)

- Elias-Gamma code is self-delimiting
  - Exercise: think how to decode uniquely
- Length of a code for x is about $2\log_2(x)$
  - Exercise: why?

Index compression, XII

Compressing docid's: easier alternative, *variable byte* codes

Easier alternative: byte-wise (8 bits) or nibble-wise (4 bits) encoding that make use of first bit to say whether it is the last byte or not (*continuation* bit).

- Encoding is also variable length, but much simpler
- Waste is not that much
- Better use of CPU by reading bytes instead of single bits
- First bit of byte is continuation bit, other 7 bits used to encode in binary
  - if 0, then last byte
  - if 1, number continues

Example:

$10101001\ 11100111\ 01100111$ is code for
$0101001\ 1100111\ 1100111$ (continuation bits in red)

Index compression, XIII

Bottom line

- Ratios of 20 % to 25 % routinely achieved
- Translates to similar speed-up at query time