Contents

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);

Time should be largely independent of database size.
(Unavoidably) proportional to answer size.

Central Data Structure: Inverted file

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

- Collection: document $\rightarrow$ words contained in the document

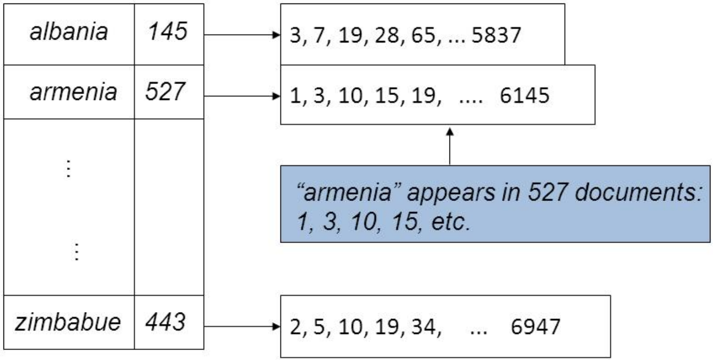
Central Data Structure: Inverted file

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

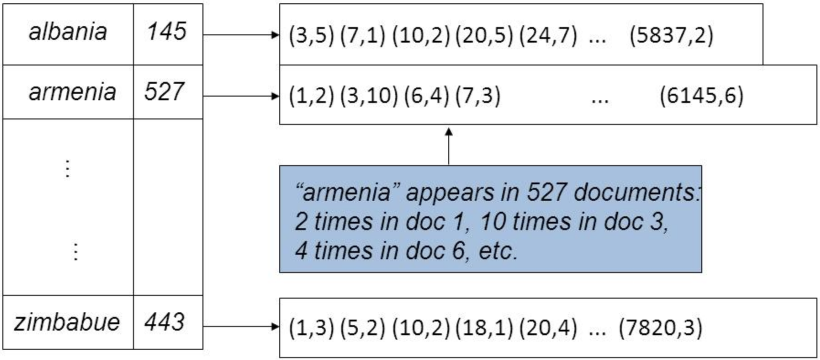
- Collection: document $\rightarrow$ words contained in the document
- Inverted file: word $\rightarrow$ documents that contain the word

Built at preprocessing time, not at query time: can afford to spend some time in its construction.

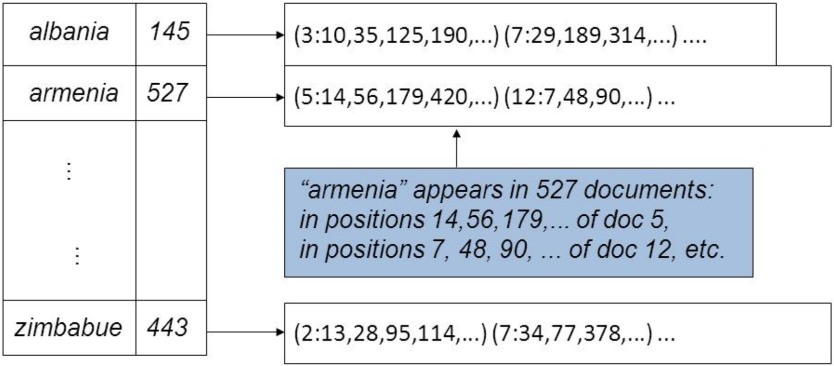
The inverted file: Variant 1



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.
- Posting lists stored in disk for largish collections.
- Almost always sorted by *docid*.
- often compressed: minimize info to bring from disk!

Implementation of the Boolean Model

Simplest: Traverse posting lists

Conjunctive query: a AND b

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

Exercise. Similar algorithms for OR and BUTNOT.

Implementation of the Boolean Model

```
def intersect(L1,L2):
    i = j = 0
    Lres = []
    while i < len(L1) and j < len(L2):
        if L1[i] < L2[j]:
            ++i
        else if L1[i] > L2[j]
            ++j
        else  # L1[i] == L2[j]
            Lres.append(L1[i])
            ++i
            ++j
    return Lres
```

Query Optimization

Query Optimizer $\rightarrow$ evaluation plan for each query:

- Rewriting the query using laws of Boolean algebra
- Choosing other algorithms for intersection and union
- Using more data structures (computed offline)

Query Rewriting

What is the most efficient way to compute a AND b AND c?

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

The following are equivalent. Which is cheapest?

- (a AND b) OR (a AND c)?
- 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 Rewriting

What is the most efficient way to compute a AND b AND c?

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

The following are equivalent. Which is cheapest?

- (a AND b) OR (a AND c)?
- a AND (b OR c)?

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

Worst cases:

- $|L1 \cap L2| \leq \min(|L1|, |L2|)$
- $|L1 \cup L2| \leq |L1| + |L2| - |L1 \cap L2| \leq |L1| + |L2|$

Query Rewriting

a AND b AND $c \rightarrow$ (a AND b) AND c

Assume: $|L_a| = 1.000$, $|L_b| = 2.000$, $|L_c| = 300$.

Minimum comparisons if using sequential scanning = $1.000 + 2.000 + 300 = 3.300$.

Query Rewriting

a AND b AND $c \rightarrow$ (a AND b) AND c

Assume: $|L_a| = 1.000$, $|L_b| = 2.000$, $|L_c| = 300$.

Minimum comparisons if using sequential scanning = $1.000 + 2.000 + 300 = 3.300$.

Instruction	Comparisons	Result $\leq$
1. $L_{a \cap b} =$ intersect(L_a, L_b)	1.000 + 2.000 = 3.000	1.000
2. $L_{res} =$ intersect$(L_{a \cap b}, L_c)$	1.000 + 300 = 1.300	300
Total comparisons	3.000 + 1.300 = **4.300**	–

Query Rewriting

a AND b AND $c \rightarrow$ (a AND c) AND b

Assume: $|L_a| = 1.000$, $|L_b| = 2.000$, $|L_c| = 300$.

Minimum comparisons if using sequential scanning = $1.000 + 2.000 + 300 = 3.300$.

Query Rewriting

a AND b AND $c \rightarrow$ (a AND c) AND b

Assume: $|L_a| = 1.000$, $|L_b| = 2.000$, $|L_c| = 300$.

Minimum comparisons if using sequential scanning = $1.000 + 2.000 + 300 = 3.300$.

Instruction	Comparisons	Result $\leq$
1. $L_{a \cap c} =$ intersect(L_a, L_c)	1.000 + 300 = 1.300	300
2. $L_{res} =$ intersect$(L_{a \cap c}, L_b)$	300 + 2.000 = 2.300	300
Total comparisons	1.300 + 2.300 = **3.600** < 4.300	–

Query Rewriting

a AND b AND $c \rightarrow$ (a AND c) AND b

Assume: $|L_a| = 1.000$, $|L_b| = 2.000$, $|L_c| = 300$.

Minimum comparisons if using sequential scanning = $1.000 + 2.000 + 300 = 3.300$.

Instruction	Comparisons	Result $\leq$
1. $L_{a \cap c} =$ intersect(L_a, L_c)	1.000 + 300 = 1.300	300
2. $L_{res} =$ intersect$(L_{a \cap c}, L_b)$	300 + 2.000 = 2.300	300
Total comparisons	1.300 + 2.300 = **3.600** < 4.300	–

Heuristic for AND-only queries: Intersect from shortest to longest.

Query Rewriting

a AND (b OR c)

Assume: $|L_a| = 300$, $|L_b| = 4.000$, $|L_c| = 5.000$.

Minimum comparisons if using sequential scanning =
$300 + 4.000 + 5.000 = 9.300$.

Query Rewriting

a AND (b OR c)

Assume: $|L_a| = 300$, $|L_b| = 4.000$, $|L_c| = 5.000$.

Minimum comparisons if using sequential scanning = $300 + 4.000 + 5.000 = 9.300$.

Instruction	Comparisons	Result $\leq$
1. $L_{b \cup c} =$ union(L_b, L_c)	4.000+5.000 = 9.000	9.000
2. $L_{res} =$ intersect$(L_a, L_{b \cup c})$	9.000 + 300 = 9.300	300
Total comparisons	9.000 + 9.300 = **18.300**	–

Query Rewriting

a AND (b OR c) $\rightarrow$ (a AND b) OR (a AND c)

Assume: $|L_a| = 300$, $|L_b| = 4.000$, $|L_c| = 5.000$.

Minimum comparisons if using sequential scanning = $300 + 4.000 + 5.000 = 9.300$.

Query Rewriting

a AND (b OR c) $\rightarrow$ (a AND b) OR (a AND c)

Assume: $|L_a| = 300$, $|L_b| = 4.000$, $|L_c| = 5.000$.

Minimum comparisons if using sequential scanning = $300 + 4.000 + 5.000 = 9.300$.

Instruction	Comparisons	Result $\leq$
1. $L_{a \cap b} =$ intersect(L_a, L_b)	300+4.000 = 4.300	300
2. $L_{a \cap c} =$ intersect(L_a, L_c)	300+5.000 = 5.300	300
3. $L_{res} =$ union$(L_{a \cap b}, L_{a \cap c})$	300 + 300 = 600	600
Total comparisons	4.300 + 5.300 + 600 = **9.900** < 18.300	–

Query Rewriting

The combinatorics may get complicated . . .

$$(a \text{ AND } b \text{ AND } d) \text{ OR } (a \text{ AND } (c \text{ OR } d) \text{ AND } e)$$
$$\equiv$$
$$((a \text{ AND } d) \text{ AND } b) \text{ OR } (a \text{ AND } c \text{ AND } e) \text{ OR } ((a \text{ AND } d) \text{ AND } e)$$

Query Rewriting

The combinatorics may get complicated . . .

$$(a \text{ AND } b \text{ AND } d) \text{ OR } (a \text{ AND } (c \text{ OR } d) \text{ AND } e)$$
$$\equiv$$
$$((a \text{ AND } d) \text{ AND } b) \text{ OR } (a \text{ AND } c \text{ AND } e) \text{ OR } ((a \text{ AND } d) \text{ AND } e)$$

Consider distributing so that we can compute intersect(L_a, L_d) once and store for reuse.

Exercise: Write the new plan as a sequence of instructions.
Exercise: Find cases where the new plan is more efficient.

Sublinear time intersection: Binary Search

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.

Sublinear time intersection: Binary Search

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.

If $|L1| \ll |L2|$

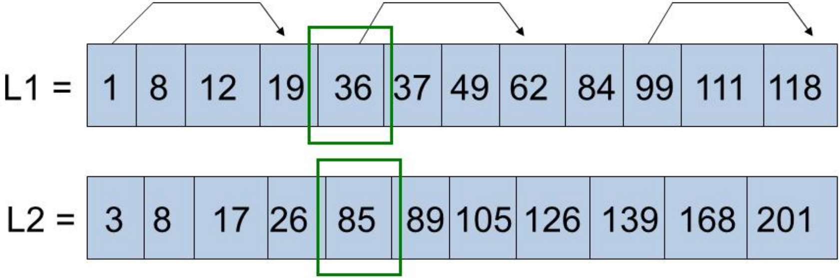
$$|L1| \cdot \log(|L2|) < |L1| + |L2|$$

Query Optimization

Example:

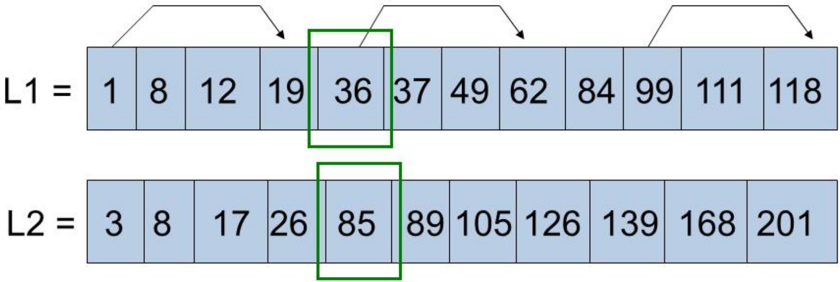
- $|L1| = 1.000$, $|L2| = 1.000$:
  - sequential scan: 2.000 comparisons,
  - binary search: $1.000 * 10 = 10.000$ comparisons.
- $|L1| = 100$, $|L2| = 10.000$:
  - sequential scan: 10.100 comparisons,
  - binary search: $100 * \log(10.000) = 1.400$ comparisons.

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.

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 $\sqrt{|L|}$.
- Needs random access - not so easy if in disk.

Implementation of the Vector 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 Vector Model

Obvious non-solution

```
for each d in D:
    sim(d,q) = 0
    get vector representing d
    for each w in q:
        sim(d,q) += tf(d,w) * idf(w)
    normalize sim(d,q) by |d|*|q|
sort results by similarity
```

idf_w and $|d|$ can be precomputed and stored in the index.
$|q|$ computed now.

…too inefficient for large D

Implementation of the Vector Model

Towards a faster algorithm

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.

Inverted file available!

Implementation of the Vector Model

Idea: Invert the loops, use inverted file

```
for each w in q:
    L = posting list for w, from inverted file
    for each d in L:
        if d seen for first time:
            sim(d,q) = 0
        sim(d,q) += tf(d,w) * idf(w)
for each d seen:
    normalize sim(d,q) by |d|*|q|
sort results by similarity
```

Implementation of the Vector Model

Idea: Invert the loops, use inverted file

After a few outer loops:

- Instead of having all of $sim(d, q)$ for some d's
- We have partially computed $sim(d, q)$ for all d's

= scan the document-term matrix by columns, not by rows

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

Topic for self-study. At least:

- Unary self-delimiting code.
- Gap compression + Elias Gamma code.
- Continuation bit.
- Typical compression ratios.

E.g. books listed in the Presentation part of these notes.

Getting fast the top r results

The last line of the algorithm was

```
sort the documents in answer by value of sim(d,q)
```

Time $O(R \log R)$, where R = #docs with $sim(d, q) > sim_{min}$.
Noticeable if R is large (milions).

User usually wants really fast the top-r, where $r \ll R$.
E.g., $r = 10$.

Example with $R = 10$ and $r = 3$

$$L = [5, 8, 3, 6, 4, 1, 10, 2, 7, 9]$$

docid	sim(docid, q)
1	0.53
2	0.26
3	0.72
4	0.32
5	0.25
6	0.36
7	0.52
8	0.88
9	0.50
10	0.19

Getting fast the top r results

Let $L = [d_1 ... d_R]$ be the answer (random order)

```
put [d_1, ..., d_r] in a minheap
for i = r+1..R:
    min_val = sim(d,q) for d = top of the heap
    if sim(d_i,q) > min_val:
        replace smallest element in heap with d_i
        reorganize heap
```

Claim: After any iteration, the heap contains the top-r documents among the first i.

Claim: If the similarities in L are randomly ordered, the expected running time of this algorithm is $O(R + r \cdot \ln(r) \cdot \ln(R/r))$.

Getting fast the top r results

Let $L = [d_1, \ldots, d_R]$ be the answer

- Time to put r elements in heap: $O(r)$
  - (recall why it is better than the obvious $O(r \log r)$)
- $Pr(d_i$ enters the heap$) =$

  $= Pr[d_i$ among r largest in $d_1, \ldots, d_i] = r/i$
- $E[$time to process $d_i] = \frac{r}{i} O(\log r) + \frac{i-r}{i} O(1)$
- E[Running time] = $O(r) + \sum_{i=r+1}^{R} \left(\frac{r}{i} O(\log r) + \frac{i-r}{i} O(1) \right)$

  $= \ldots$ (use $H(n) \simeq \ln(n)$, H harmonic function)

  $= O(R) + O(r \ln(r) \ln(R/r))$

For $r \ll R$, we go from $O(R \log R)$ to $O(R)$.