CAI: Cerca i Anàlisi de la Informació
Grau en Ciència i Enginyeria de Dades, UPC

Implementation

September 30, 2021

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

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);

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

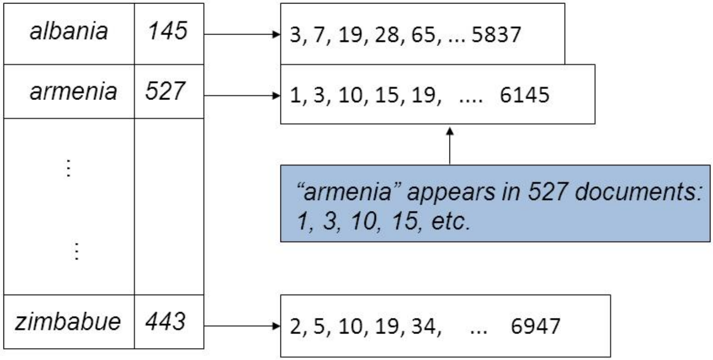
Central Data Structure: Inverted file

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

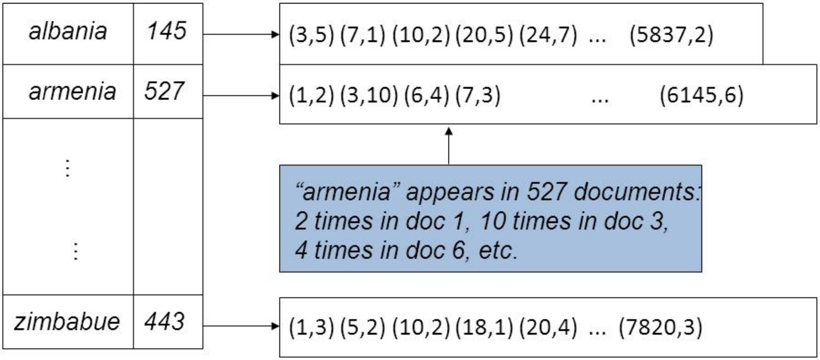
- Collection: document $\rightarrow$ words contained in the document
- Inverted file: word $\rightarrow$ documents that contain the word

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

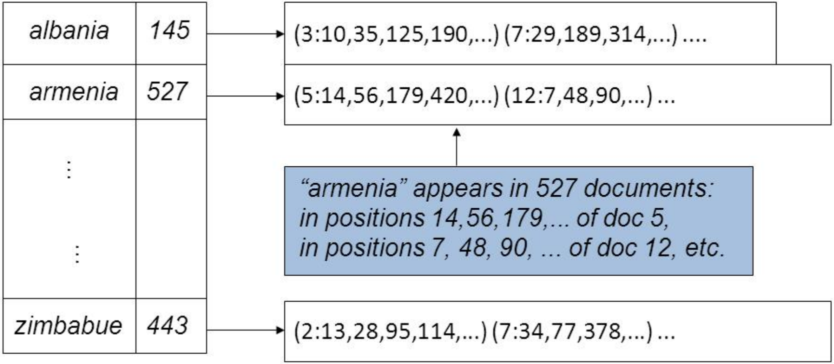
The inverted file: Variant 1



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.

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?

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 $|L1| \ll |L2|$

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

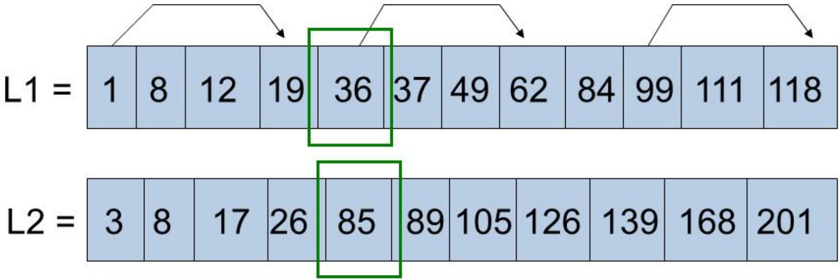
Query Optimization

Query Optimization

Example:

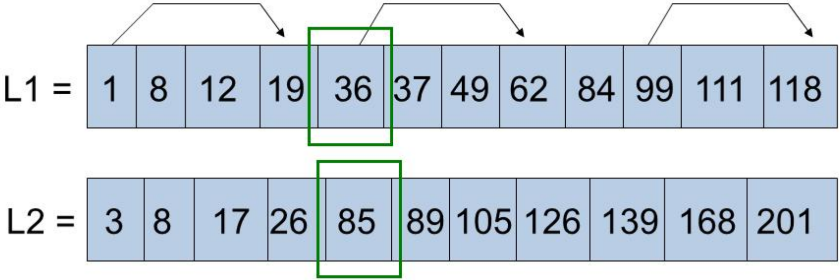
- $|L1| = 1000$, $|L2| = 1000$:
  - sequential scan: 2000 comparisons,
  - binary search: $1000 * 10 = 10,000$ comparisons.
- $|L1| = 100$, $|L2| = 10,000$:
  - sequential scan: $10,100$ comparisons,
  - binary search: $100 * \log(10,000) = 1400$ comparisons.

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

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

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:

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

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 $\overbrace{111 \ldots 1}^{x \text{ times}}$

- 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

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).

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., $r = 10$.

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
```

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.

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: $O(r)$
  - (recal 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)

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

Getting fast the top r results

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

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

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

  $= \ldots$ (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 $O(R)$.

How to build the Index (offline)

Given document collection D, build the inverted file F

How to build the Index (offline)

Given document collection D, build the inverted file F

In python - in RAM:

```
F = {}
for doc in D:
    d = docid(doc)
    for w in doc:
        if w not in F:
            F[w] = {}
        if d not in F[w]:
            F[w][d] = 0
        F[w][d] += 1
```

How to build the Index (offline)

Given document collection D, build the inverted file F

In python - in RAM:

```
F = {}
for doc in D:
    d = docid(doc)
    for w in doc:
        if w not in F:
            F[w] = {}
        if d not in F[w]:
            F[w][d] = 0
        F[w][d] += 1
```

Large indices must go to disk, not RAM

Writing indices to disk

Without going to many details . . .

```
Initialize F to empty in disk
for docid in D in increasing order:
    for word in D[docid]:
       L = retrieve list F(word) from disk
       if (docid,c) in L:
          replace (docid,c) with (docid,c+1)
          # in disk list!
       else:
          append (docid,1) at the end of F(word)
          # this keeps lists sorted by docid
```

Writing indices to disk

Without going to many details . . .

```
Initialize F to empty in disk
for docid in D in increasing order:
    for word in D[docid]:
        L = retrieve list F(word) from disk
        if (docid,c) in L:
            replace (docid,c) with (docid,c+1)
            # in disk list!
        else:
            append (docid,1) at the end of F(word)
            # this keeps lists sorted by docid
```

Perhaps can be optimized to not read/write all of L, only parts

Writing indices to disk

Without going to many details ...

```
Initialize F to empty in disk
for docid in D in increasing order:
    for word in D[docid]:
        L = retrieve list F(word) from disk
        if (docid,c) in L:
            replace (docid,c) with (docid,c+1)
            # in disk list!
        else:
            append (docid,1) at the end of F(word)
            # this keeps lists sorted by docid
```

Perhaps can be optimized to not read/write all of L, only parts

But the real problem is another: access to lists is random!

Disk technology

Traditional hard disks with moving parts

- Seek time veery slow - head movement.
- Once head is in place, sequential access is fast.
- = reads/writes with consecutive, large chunks of bits.
- N random read/writes muuuch slower than N sequential read/writes.
- Like $50\times$, easily.

Disk technology

Traditional hard disks with moving parts

- Seek time veery slow - head movement.
- Once head is in place, sequential access is fast.
- = reads/writes with consecutive, large chunks of bits.
- N random read/writes muuuch slower than N sequential read/writes.
- Like $50\times$, easily.

[Things are different with new SSD drives - no moving parts, little difference between sequential and random access. They may have problems with small writes, as they read/write full pages. And they use large page sizes. Impact of this still not well studied.]

More efficient

1. Initialize disk index to be empty
2. Build index in RAM, up to allocated memory M
3. When RAM full:
    - append each list in RAM to end of corresponding list in disk
    - sequential writes to disk! fast!
    - clear the RAM index
    - goto to 2 to process more documents

More efficient

1. Initialize disk index to be empty
2. Build index in RAM, up to allocated memory M
3. When RAM full:
    - append each list in RAM to end of corresponding list in disk
    - sequential writes to disk! fast!
    - clear the RAM index
    - goto to 2 to process more documents

Observations:

- a RAMful of index is sometimes called a “barrel”
- many barrels can be built in concurently with a cluster
- merging barrels into disk index is done by a single machine