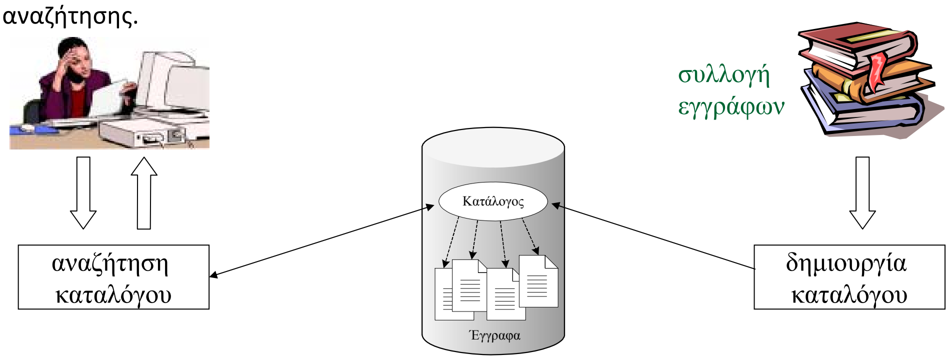
Ευρετηρίαση
ΜΕΡΟΣ ΙΙ

Content

- Processing Boolean Queries
- Faster posting lists with skip pointers
- Phrase and Proximity Queries
  - Biwords
  - Positional Indexes
- Dictionary
- Wild-Card Queries
  - Permutex
  - *k*-gram indexes

Χρήση Καταλόγων/Ευρετηρίων

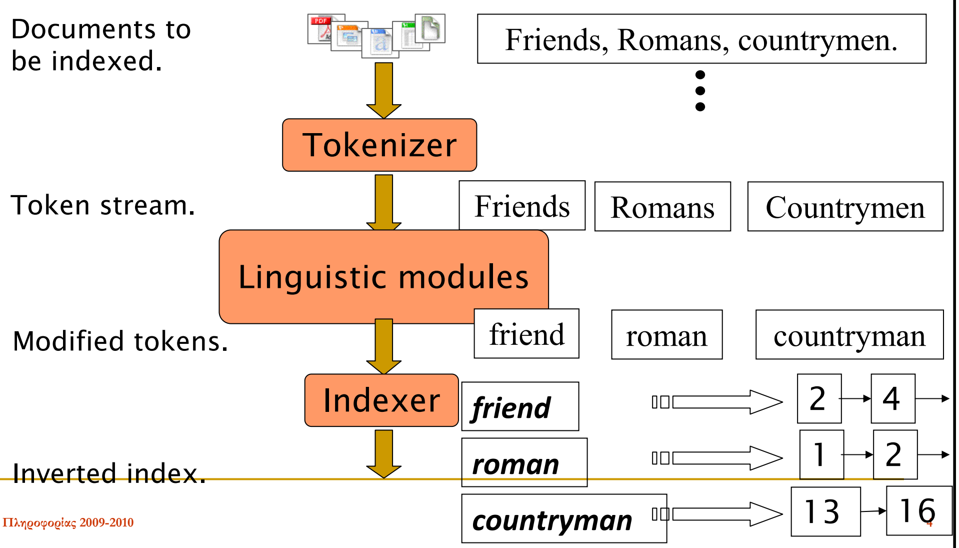
Τα συστήματα ανάκτησης σπάνια αναζητούν την πληροφορία απευθείας στη συλλογή εγγράφων. Συνήθως, χρησιμοποιούνται κατάλογοι οι οποίοι επιταχύνουν τη διαδικασία αναζήτησης.



Σχεδιάζουμε το ευρετήριο ανάλογα με το μοντέλο ανάκτησης και τη γλώσσα επερώτησης

Ανάκτηση Πληροφορίας 2009-2010 3

Inverted index construction



Ανάκτηση Πληροφορίας 2009-2010 4

Γενική (Λογική) μορφή ενός ευρετηρίου

Indexing Items (όροι ευρετηρίου)

Documents

	k_1	k_2	...	k_j	...	k_t
d_1	$c_{1,1}$	$c_{2,1}$	...	$c_{i,1}$	...	$c_{t,1}$
d_2	$c_{1,2}$	$c_{2,2}$	...	$c_{i,2}$	...	$c_{t,2}$
...	...	...	...	...	...	...
d_i	$c_{1,j}$	$c_{2,j}$	...	$c_{i,j}$	...	$c_{t,j}$
...	...	...	...	...	...	...
d_N	$c_{1,N}$	$c_{2,N}$	...	$c_{i,N}$	...	$c_{t,N}$

c_{ij}: το κελί που αντιστοιχεί στο έγγραφο d_i και στον όρο k_j, το οποίο μπορεί να περιέχει:

- ένα w_{ij} που να δηλώνει την παρουσία ή απουσία του k_j στο d_i (ή τη σπουδαιότητα του k_j στο d_i)
- τις θέσεις στις οποίες ο όρος k_j εμφανίζεται στο d_i (αν πράγματι εμφανίζεται)

Ερωτήματα:

- Τι πρέπει να έχει το κάθε c_{ij}
- Πώς να υλοποιήσουμε αυτή τη λογική δομή ώστε να έχουμε καλή απόδοση;

Γλώσσες Επερώτησης για Ανάκτηση Πληροφοριών

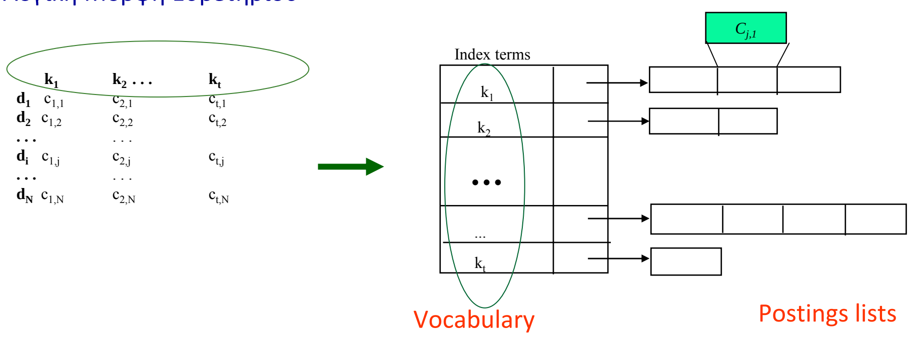
- Επερωτήσεις λέξεων (Keyword-based Queries)
  - Μονολεκτικές επερωτήσεις (Single-word Queries)
  - Επερωτήσεις φυσικής γλώσσας (Natural Language Queries)
  - Boolean Επερωτήσεις (Boolean Queries)
  - Επερωτήσεις Συμφραζομένων (Context Queries)
    - Φραστικές Επερωτήσεις (Phrasal Queries)
    - Επερωτήσεις Εγγύτητας (Proximity Queries)
- Ταίριασμα Προτύπου (Pattern Matching)
  - Απλό (Simple)
  - Ανεκτικές σε ορθογραφικά λάθη (Allowing errors)
    - Levenstein distance, LCS longest common subsequence
  - Κανονικές Εκφράσεις (Regular expressions)
- *Δομικές Επερωτήσεις (Structural Queries)*
  - *(θα καλυφθούν σε επόμενο μάθημα)*
- *Πρωτόκολλα επερώτησης (Query Protocols)*

Ανεστραμμένα Αρχεία (Inverted Files)

Ανεστραμμένο Αρχείο

Λογική Μορφή Ευρετηρίου

Μορφή Ανεστραμμένου Ευρετηρίου



Άρα δεν δεσμεύουμε χώρο για **τα «μηδενικά κελιά» της λογικής μορφής του ευρετηρίου**

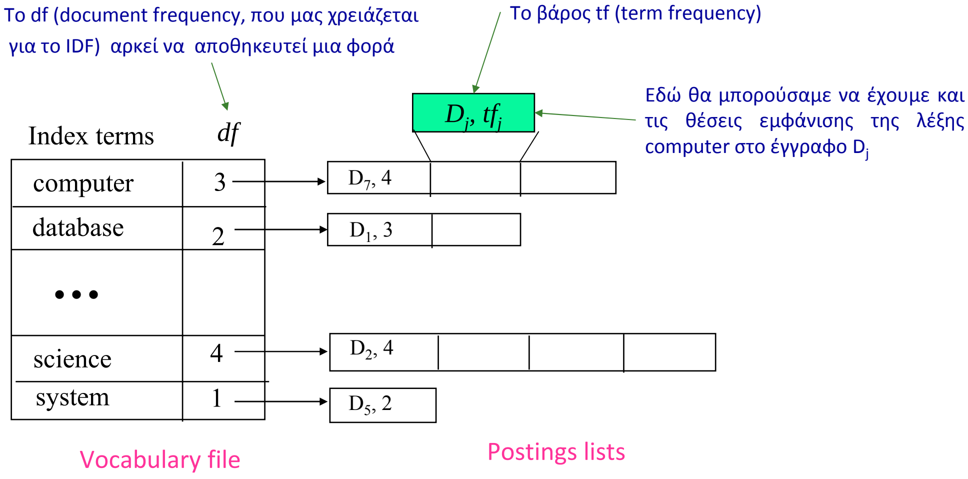
Inverted Files (Ανεστραμμένα αρχεία)

Inverted file = a word-oriented mechanism for indexing a text collection in order to speed up the searching task.

An inverted file consists of:

- Vocabulary: is the set of all distinct words in the text
- Occurrences: lists containing all information necessary for each word of the vocabulary (documents where the word appears, frequency, text position, etc.)
  - Τι είδους πληροφορία κρατάμε στις posting lists εξαρτάται από το λογικό μοντέλο και το μοντέλο ερωτήσεων

Ανεστραμμένο αρχείο για πολλά έγγραφα, και βάρυνση tf-idf



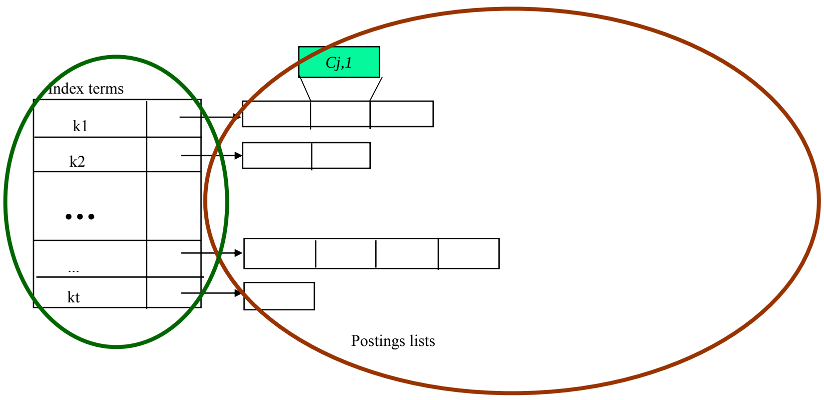
Another example

	term	df		document ids
1	Algorithms	3	:	3 5 7
2	Application	2	:	3 17
3	Delay	2	:	11 12
4	Differential	8	:	4 8 10 11 12 13 14 15
5	Equations	10	:	1 2 4 8 10 11 12 13 14 15
6	Implementation	2	:	3 7
7	Integral	2	:	16 17
8	Introduction	2	:	5 6
9	Methods	2	:	
10	Nonlinear	2	:	9 13
11	Ordinary	2	:	8 10
12	Oscillation	2	:	11 12
13	Partial	2	:	4 13
14	Problem	2	:	6 7
15	Systems	3	:	6 8 9
16	Theory	4	:	3 11 12 17

Ανεστραμμένα Αρχεία: Απαιτήσεις Χώρου

μικρές

μεγάλες



Ανεστραμμένα Αρχεία (Inverted Files)

ΔΥΑΔΙΚΟ ΜΟΝΤΕΛΟ

Boolean Keyword Queries

Term-document incidence

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Brutus** AND **Caesar** BUT NOT **Calpurnia

1 if play contains word, 0 otherwise

Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
  - matrix is extremely sparse.
- What's a better representation?
  - We only record the 1 positions.

Inverted index

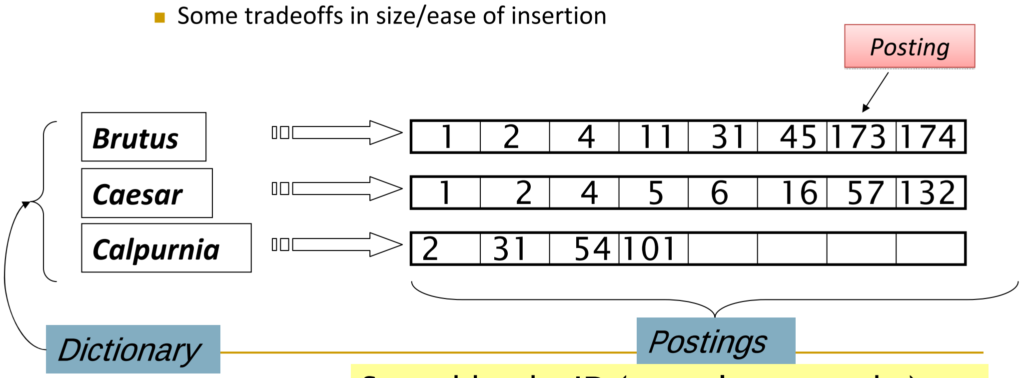
- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each by a **docID**, a document serial number
- Can we used fixed-size arrays for this?

Brutus	1	2	4	11	31	45	173	174
Caesar	1	2	4	5	6	16	57	132
Calpurnia	2	31	54	101				

What happens if the word ***Caesar*** is added to document 14?

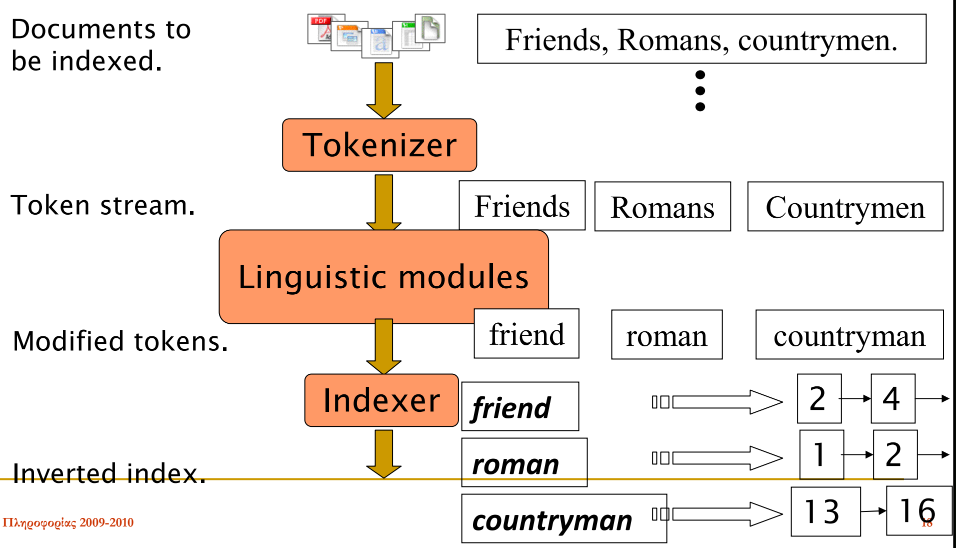
Inverted index

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion



Sorted by docID (more later on why).

Inverted index construction



Inverted index construction

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Inverted index construction

- Sort by terms
  - And then docID

Core indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Inverted index construction

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

→

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?

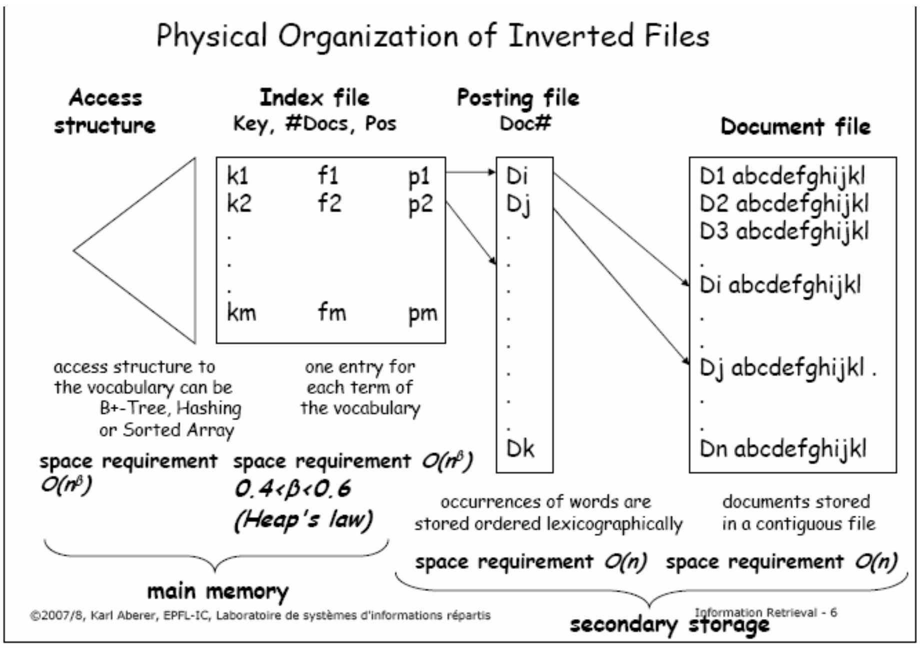
term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

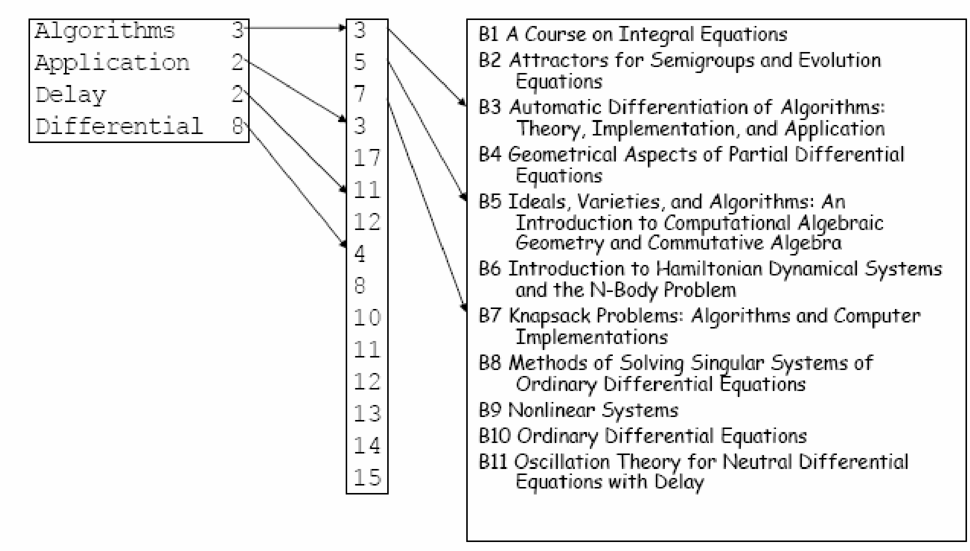
Terms and counts

Lists of docIDs

Pointers

- How do we process a query?





Ανάκτηση Πληροφορίας 2009-2010 25

Searching an inverted index

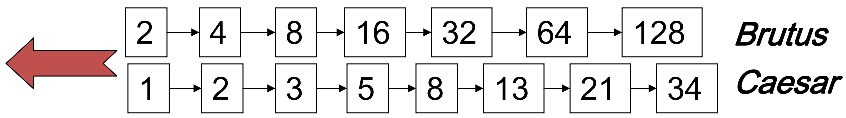
General Steps:

1. Vocabulary search:
   the words present in the query are searched in the vocabulary
2. Retrieval occurrences:
   the lists of the occurrences of all words found are retrieved
3. Manipulation of occurrences:
   The occurrences are processed to solve the query

Ανάκτηση Πληροφορίας 2009-2010 26

Query processing: AND

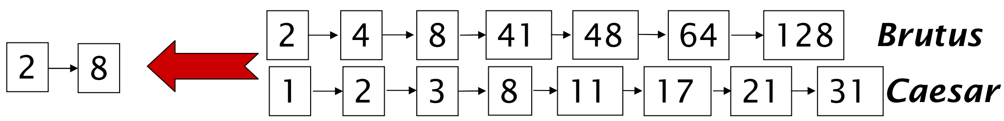
- Consider processing the query: ***Brutus*** *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:



Query processing: AND

The 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.

Crucial: postings sorted by docID.

Query processing: **merge**

```
INTERSECT(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 p1 ← next(p1)
             else p2 ← next(p2)
return answer
```


Boolean queries: More general merges

- Exercise: Adapt the merge for:
  Brutus *AND NOT* ***Caesar***

Can we still run through the merge in time O(*x*+*y*)?
What can we achieve?

- Exercise: Adapt the merge for:
  Brutus *OR NOT* ***Caesar***

Can we still run through the merge in time O(*x*+*y*)?
What can we achieve?

Merging

What about an arbitrary Boolean formula?

(Brutus OR Caesar) *AND NOT* ***(Antony*** *OR* ***Cleopatra)***

- Can we always merge in “linear” time?
  - Linear in what?
- Can we do better?

Query optimization

What is the best order for query processing?

- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.

Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

Query optimization example

- <u>Process in order of increasing freq</u>:
  - *start with smallest set, then keep cutting further.*

This is why we kept document freq. in dictionary

Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

More general optimization

e.g., *(**madding** OR **crowd**) AND (**ignoble** OR **strife**)*

- Get doc. freq.'s for all terms.
- Estimate the size of each *OR* by the sum of its doc. freq.'s (conservative).
- Process in increasing order of *OR* sizes.

Exercise

- Recommend a query processing order for

*(**tangerine** OR **trees**) AND (**marmalade** OR **skies**) AND (**kaleidoscope** OR **eyes**)*

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

Query processing exercises

- Exercise: If the query is ***friends** AND **romans** AND (NOT **countrymen**)*, how could we use the freq of ***countrymen***?
- Exercise: Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?
- Hint: Begin with the case of a Boolean *formula* query: in this, each query term appears only once in the query.

Ειδικές Μορφές του Ανεστραμμένου Ευρετηρίου

Recap

Key step in construction: Sorting

BRUTUS	⟶	1	2	4	11	31	45	173	174
CAESAR	⟶	1	2	4	5	6	16	57	132	...
CALPURNIA	⟶	2	31	54	101				

Boolean query processing
- Intersection by linear time "merging"
- Simple optimizations

Αποτίμηση Boolean επερωτήσεων με χρήση ανεστραμμένων αρχείων

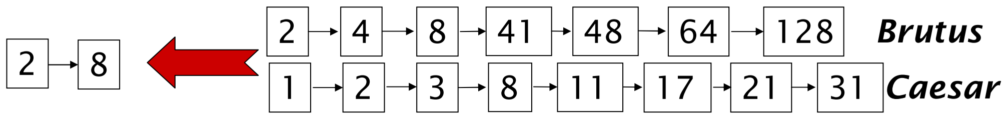
Αποτίμηση με χρήση ανεστραμμένων αρχείων

- Single keyword: Retrieve containing documents using the inverted index.
- OR: Recursively (by merge) retrieve e_1 and e_2 and take union of results.
- AND: Recursively retrieve e_1 and e_2 and take intersection of results.
- BUT: Recursively retrieve e_1 and e_2 and take set difference of results.

FASTER POSTINGS MERGES:
SKIP POINTERS/SKIP LISTS

Recall basic merge

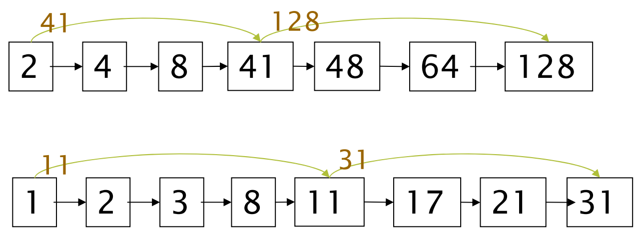
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



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

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

Augment postings with skip pointers (at indexing time)

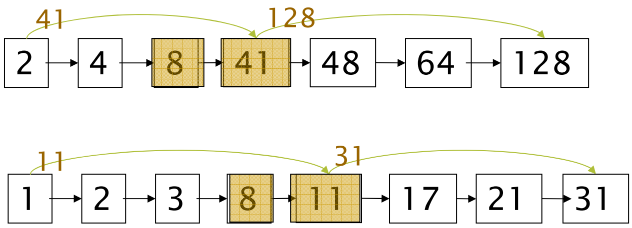


Why?

- To skip postings that will not figure in the search results.

1. How?
2. Where do we place skip pointers?

How? Query processing with skip pointers



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

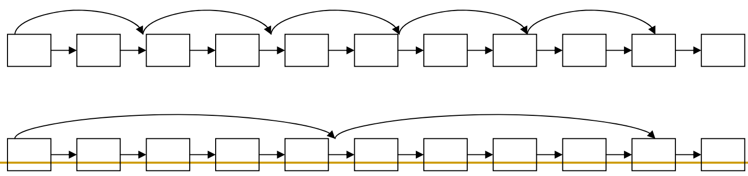
We then have **41** and **11** on the lower. **11** is smaller.

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 √*L* evenly-spaced skip pointers.

- This ignores the distribution of query terms.
- 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 (Bahle et al. 2002) unless you're memory-based
  - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!

PHRASE QUERIES AND POSITIONAL INDEXES

Phrase queries

- Want to be able to answer queries such as "***stanford university***" – as a phrase
- Thus the sentence *"I went to university at Stanford"* is not a match.
  - The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works -- 10% explicit phrase queries ("")
  - Many more queries are *implicit phrase queries (such as person names)*

Phrase queries

- For this, it no longer suffices to store only

  <*term* : *docs*> entries

Είδαμε στο προηγούμενο μάθημα ότι μπορούμε να κρατάμε τη θέση κάθε όρου στο κείμενο ή να χωρίσουμε το κείμενο σε blocks

(θα το δούμε ποιο αναλυτικά σήμερα)

A first attempt: Biword indexes

Index every consecutive pair of terms in the text as a phrase

- For example the text "***Friends, Romans, Countrymen***" would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Ανάκτηση Πληροφορίας 2009-2010 49

Longer phrase queries

stanford university palo alto

can be broken into the Boolean query on biwords:

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

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!

Ανάκτηση Πληροφορίας 2009-2010 50

Extended biwords

1. Parse the indexed text and perform part-of-speech-tagging (POST).
2. Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
3. Call any string of terms of the form NX*N an extended biword
4. Each such extended biword is now made a term in the dictionary.

Example: ***catcher in the rye***

N X X N

Query processing: parse it into N's and X's

- Segment query into enhanced biwords
- Look up in index: ***catcher rye***

Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

Biword indexes are not the standard solution (for all biwords) but can be **part of** a compound strategy

Solution 2: Positional indexes

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

<***term***, number of docs containing ***term***;
doc1: position1, position2 … ;
doc2: position1, position2 … ;
etc.>

Ας θεωρήσουμε ότι position είναι η θέση του token

Positional index example

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Processing a phrase query

- 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***".
  - ***to***:
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - ***be***:
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...

Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  - Again, here, /*k* means "within *k* words of".
- Clearly, positional indexes can be used for such queries; biword indexes cannot.

Proximity queries

- 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
  - See Figure 2.12 of IIR
  - There's likely to be a problem on it!

Positional index size

- You can compress position values/offsets

  Nevertheless, a positional index expands postings storage *substantially*

- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system.

Positional index size

- 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
- Consider a term with frequency 0.1%

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

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- (compressed) Positional index size 35–50% of volume of original text
- Caveat: all of this holds for "English-like" languages

The number of items to check Θ(N) -> Θ(T), where
N:number of documents, T: number of tokens

Combination schemes

- These two approaches can be profitably combined
  - For particular phrases (***"Michael Jackson", "Britney Spears"***) it is inefficient to keep on merging positional postings lists
    - Even more so for phrases like ***"The Who"***

In general:

Good queries to include: common (based on recent query behavior) and expensive

Combination schemes

- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme (+ a partial next word index)
  - 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

Evaluating **Phrasal** Queries with Inverted Indices

- Phrasal Queries (summary)
  - Must have an inverted index that also stores ***positions*** of each keyword in a document.
  - Retrieve documents and positions for each individual word,
    intersect documents, and
    then finally **check** for ordered contiguity of keyword positions.

  Best to start contiguity check with the *least common word* in the phrase.

Evaluating **Proximity** Queries with Inverted Indices

- Proximity Queries (summary)
  - Use approach similar to phrasal search to find documents in which all keywords are found in a context that satisfies the proximity constraints -- a list (in increasing positional order) is generated for each one
  - The lists of all elements are traversed in synchronization to find places where all the words appear close enough (for proximity).
  - During binary search for positions of remaining keywords, find closest position of k_i to p and check that it is within maximum allowed distance.

Inverted Index: Κατακλείδα

- Is probably the most adequate indexing technique
- Appropriate when the text collection is large and semi-static
- If the text collection is volatile online searching is the only option
- Some techniques combine online and indexed searching

Resources for today's lecture

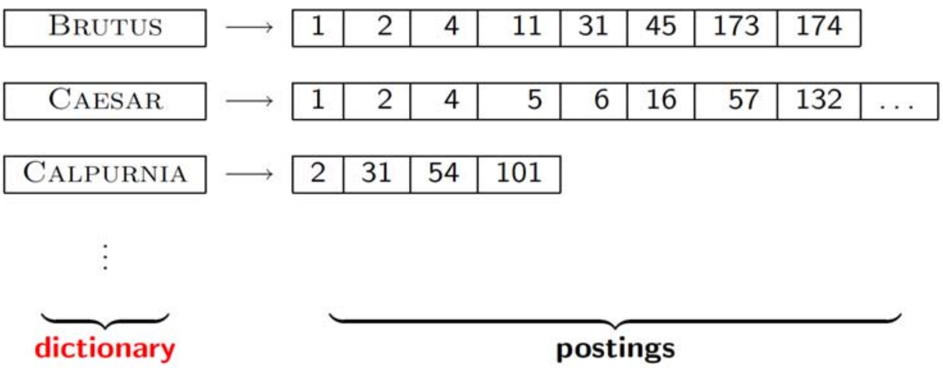
- Skip Lists theory: Pugh (1990)
  - *Multilevel skip lists give same O(log n) efficiency as trees*
- H.E. Williams, J. Zobel, and D. Bahle. 2004. "Fast Phrase Querying with Combined Indexes", ACM Transactions on Information Systems.

  http://www.seg.rmit.edu.au/research/research.php?author=4
- D. Bahle, H. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. SIGIR 2002, pp. 215-221.

Vocabulary search

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … *in what data structure?*



A naïve dictionary

- An array of struct:

  char[20] int Postings *

  20 bytes 4/8 bytes 4/8 bytes
- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
...	...	...
zulu	221	⟶

Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees

Vocabulary search

As each searching task on an inverted file always starts in the vocabulary, it is better to **store the vocabulary in a separate file**

- this file is not so big so it is possible to keep it at main memory at search time

Suppose we want to search for a word of length *m.*

The structures most used to store the vocabulary are ***hashing, tries*** or ***B-trees.***

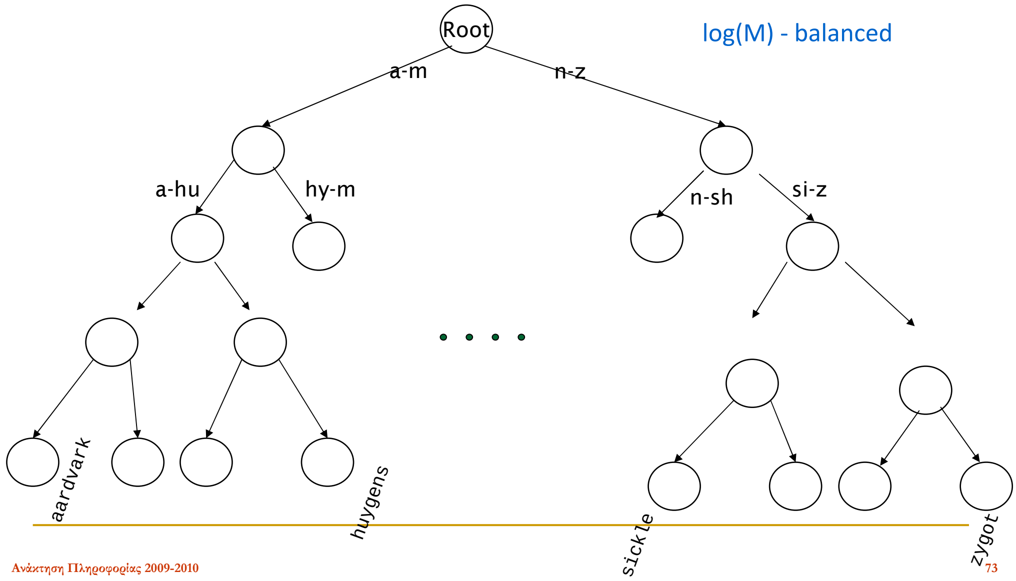
Options:

- Cost of searching a sequential file: O(V)
- Cost of searching assuming hashing: O(m)
- Cost of searching assuming tries: O(m)
- Cost of searching assuming the file is ordered (lexicographically): O(log V)
  - this option is cheaper in space and very competitive

Hashes

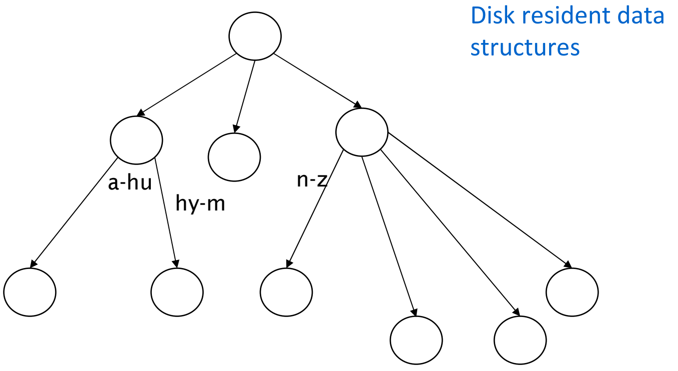
- Each vocabulary term is hashed to an integer
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree



Tree: B-tree

- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].



Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we standardly have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

WILD-CARD QUERIES

Wild-card queries: *

Wildcard queries are used in any of the following situations:

(1) the user is uncertain of the spelling of a query term

(e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney);

(2) the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants

(e.g., color vs. colour);

(3) the user seeks documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming

(e.g., judicial vs. judiciary, leading to the wildcard query judicia*);

(4) the user is uncertain of the correct rendition of a foreign word or phrase

(e.g., the query Universit* Stuttgart).

Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning "mon".
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ****mon:*** find words ending in "mon": harder

Trialing wildcards

Wild-card queries: *

- ***mon:*** find words ending in "mon": harder
  - Maintain an additional B-tree for terms *backwards.*

  Can retrieve all words in range: ***nom ≤ w < non***.

Reverse B-tree (suffix B-tree)

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

(in general, any query with a single wildcard)

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.

- E.g., consider the query:

  se*ate** AND **fil*er

  This may result in the execution of many Boolean *AND* queries.

B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ****tion*** in a B-tree and intersect the two term sets
  - Expensive

B-trees handle *'s at the end of a query term

- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

Κατασκευάζουμε επιπρόσθετη δομή (πλέον του dictionary + inverted index)

Permuterm index

- A special symbol $ to indicate the end of a word
  - ***hello -> hello$***
- Construct a permuterm index, in which the various rotations of each term (augmented with $) all link to the original vocabulary term.
  - ***hello$, ello$h, llohe, lohel, o$hell***

Permuterm vocabulary (the vocabulary consists of all such permutations)

Ουσιαστικά, θεωρούμε όλα τα πιθανά suffix

Permuterm index

A query with one wildcard

Rotate so that the wildcard (*) appears at the end of the query

Lookup the resulting string in the permuterm index (prefix query – trailing wildcard) and get all words in the dictionary

Query = ***hel*o***
X=***hel***, Y=***o***
Lookup ***o$hel****

Permuterm index

Example

Permuterm vocabulary for **magic** and **music**

Query **m*ic**

m*n matches **man** and **moron**

We lookup these terms in the standard inverted index to retrieve matching documents

Permuterm index

- Queries:
  - **X** lookup on **X$** — **X*** lookup on $**X***
  - ***X** lookup on **X$*** — ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X*** — **X*Y*Z** ??? Exercise!

Query = ***hel*o***
X=*hel*, Y=*o*
Lookup ***o$hel****

Permuterm index

Example

fi*mo*er

fi*mo*er

1. Enumerate all terms in the dictionary that are in the permuted index of er$fi*
2. Then, filter out (exhaustive search) those that do not have mo in the middle
3. Run surviving terms through the standard inverted index

Ανάκτηση Πληροφορίας 2009-2010 87

Permuterm query processing (summary)

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem: ≈ quadruples lexicon size (tenfold increase of the dictionary)*

Empirical observation for English.

Είναι παρόμοιο με το να εισάγουμε όλους τα suffix σε ένα B-tree (SUFFIX TREES)

Ανάκτηση Πληροφορίας 2009-2010 88

Bigram (*k*-gram) indexes

- A *k*-gram is a sequence of *k* characters
- Use as special character $ to denote the beginning or the end of a term
- In a *k*-gram index, the dictionary contains all k-grams that occur in any term in the vocabulary

Example 3-grams for ***music***

Bigram (*k*-gram) indexes

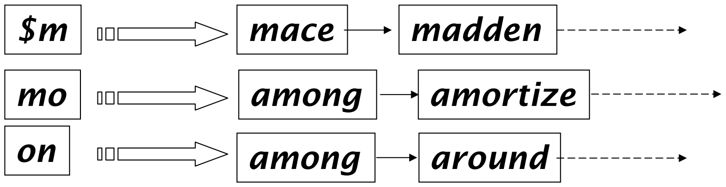
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

  a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
  ue,el,le,es,st,t$, m,mo,on,nt,h

  - $ is a special word boundary symbol
- Maintain a *second* inverted index *from bigrams to dictionary terms* that match each bigram.

Bigram index example

- The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



Similar to the postings in the inverted index (ordered)

Processing wild-cards

- Query ***mon**** can now be run as (assume 2-grams)
  - ***$m** AND **mo** AND **on***
- Gets terms that match AND version of our wildcard query.

Example ***re*ve*** and 3-grams

Processing wild-cards

- Query ***mon**** can now be run as (assume 2-grams)
  - ***$m** AND **mo** AND **on***
- Gets terms that match AND version of our wildcard query.

But we'd enumerate ***moon***.

1. Must *post-filter* these terms against query. (the terms enumerated by the Boolean query on the k-gram are checked individually against the original query
2. Surviving enumerated terms are then looked up in the term-document inverted index.

- Fast, space efficient (compared to permuterm).

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage "laziness" people will respond!

- Which web search engines allow wildcard queries?

[ ] Search

Type your search terms, use '*' if you need to.
E.g., Alex* will match Alexander.