ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΜΗΧΑΝΕΣ ΑΝΑΖΗΤΗΣΗΣ
(SEARCH ENGINES)

ΔΙΑΛΕΞΗ 4

ΔΙΔΑΣΚΩΝ
ΚΩΣΤΑΣ ΚΟΛΟΜΒΑΤΣΟΣ

Κατασκευή του Ευρετηρίου

Blocked Sort-based Indexing

- Για την πιο αποδοτική αναπαράσταση των ευρετηρίων αναπαριστούμε τους όρους με Ids αντί για αλφαριθμητικά
- Έτσι έχουμε ζεύγη termID-docID στις postings
- Κάθε termID είναι ένας μοναδικός σειριακός αριθμός
- Η απεικόνιση μεταξύ των όρων και των Ids γίνεται on the fly
- Εναλλακτικά σε δυο φάσεις φτιάχνουμε το λεξικό στο πρώτο στάδιο και στο δεύτερο στάδιο κατασκευάζουμε το inverted index
- Παρακάτω θα δούμε αλγορίθμους που λειτουργούν σε ένα στάδιο
- Οι αλγόριθμοι δύο φάσεων βοηθούν σε κάποιες περιπτώσεις

Blocked Sort-based Indexing

- Μια και η μνήμη δεν είναι αρκετή ώστε να χωρέσει όλα τα έγγραφα χρειαζόμαστε ένα εξωτερικό αλγόριθμο ταξινόμησης (external sorting algorithm)
- Ο αλγόριθμος θα χρησιμοποιεί το δίσκο
- Η βασική απαίτηση είναι να ελαχιστοποιηθούν οι τυχαίες προσβάσεις / αναζητήσεις στο δίσκο ώστε να εξοικονομηθεί χρόνος
- Εστιάζουμε σε ακολουθιακές προσβάσεις στο δίσκο

Blocked Sort-based Indexing

- Μια λύση είναι η τεχνική του blocked sort-based indexing (BSBI)
- Ο αλγόριθμος
  - Τμηματοποιεί τη συλλογή σε δύο τμήματα ίσου μήκους
  - Ταξινομεί τα ζεύγη termID-docID κάθε τμήματος στη μνήμη
  - Αποθηκεύει τα ενδιάμεσα αποτελέσματα
  - Συγχωνεύει τα αποτελέσματα

Blocked Sort-based Indexing

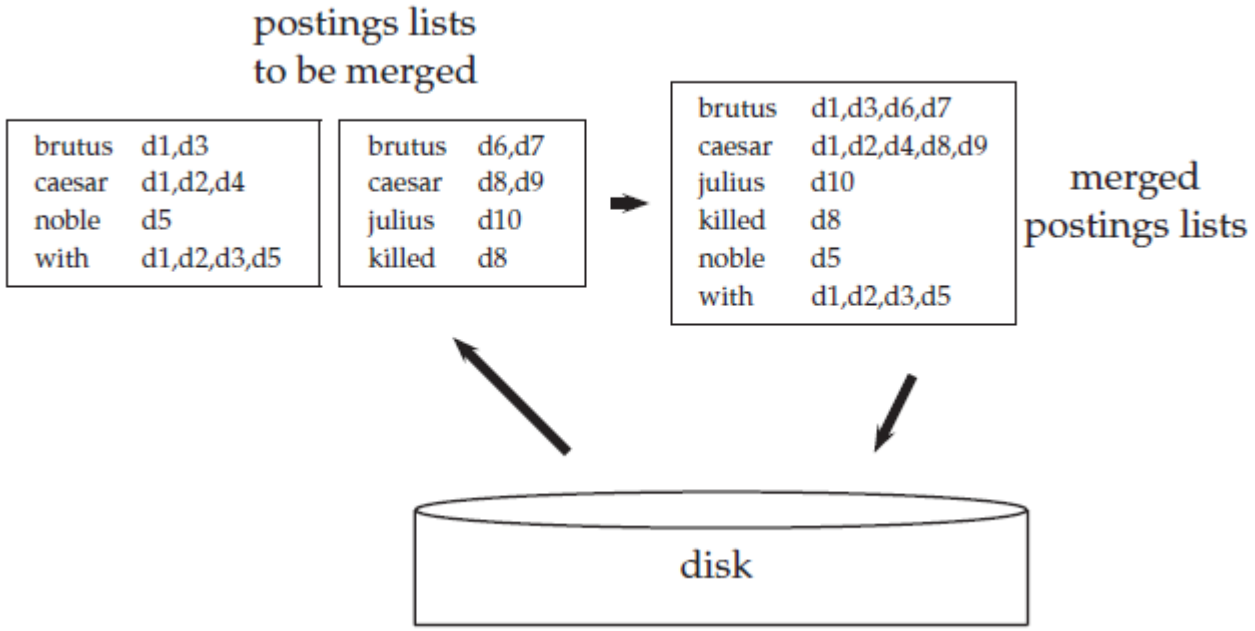
```
BSBINDEXCONSTRUCTION()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← PARSENEXTBLOCK()
5     BSBI-INVERT(block)
6     WRITEBLOCKTODISK(block, f_n)
7  MERGEBLOCKS(f_1, ..., f_n; f_merged)
```

Blocked Sort-based Indexing

- Το block size επιλέγεται να είναι τέτοιο ώστε να χωράει στη μνήμη
- Στη συνέχεια καταγράφεται στο δίσκο
- Αρχικά, ταξινομούμε όλα τα ζεύγη termID-docID με το ίδιο termID σε μια λίστα όπου βάζουμε μόνο το docID
- Το αποτέλεσμα θα είναι ένα block για τα ζεύγη που μόλις επεξεργαστήκαμε
- Καταγράφουμε το αποτέλεσμα στο δίσκο
- Παράδειγμα:
  - Για το dataset του reuters-RCV1 με 10,000,000 ζεύγη καταλήγουμε σε 10 blocks
- Στο τελευταίο βήμα, συγχωνεύουμε όλα τα blocks

Blocked Sort-based Indexing

- Διατηρούμε buffers για την ανάγνωση από τα blocks και ένα buffer για την εγγραφή
- Σε κάθε επανάληψη επιλέγουμε το μικρότερο termID με τη βοήθεια μιας ουράς με προτεραιότητα ή μια άλλη κατάλληλη δομή δεδομένων



Single Pass in-Memory Indexing

- Το μειονέκτημα του BSBI είναι ότι απαιτεί μια δομή / τεχνική για την απεικόνιση των όρων σε IDs
- Για πολύ μεγάλες συλλογές δεδομένων, η απαιτούμενη δομή δεν χωρά στη μνήμη
- Εναλλακτικά μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο Single-pass in-Memory Indexing (SPIMI)
- Ο SPIMI δεν χρησιμοποιεί termIDs αλλά τους όρους
- Γράφει κάθε block του λεξικού στο δίσκο και ξεκινά ένα νέο λεξικό για το επόμενο block

Single Pass in-Memory Indexing

- Ο αλγόριθμος δέχεται μια ροή (stream) εγγράφων και τα μετατρέπει σε ζεύγη term-docID
- Η συνάρτηση SPIMI-INVERT καλείται επαναληπτικά μέχρι να τελειώσει η είσοδος
- Επεξεργαζόμαστε τα ζεύγη ένα-ένα
- Όταν ένας όρος απαντάνται για πρώτη φορά προστίθεται στο λεξικό και δημιουργείται μια νέα postings

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Single Pass in-Memory Indexing

- Ο αλγόριθμος προσθέτει απ' ευθείας ένα posting στην κατάλληλη postings list
- Είναι πιο γρήγορη τεχνική επειδή δεν ταξινομεί πριν προσθέσει στις λίστες
- Απαιτεί λιγότερο χώρο στη μνήμη
- Στις γραμμές 8-9 αυξάνουμε το χώρο για μια λίστα αν είναι απαραίτητο

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Single Pass in-Memory Indexing

- Αν η μνήμη γεμίσει, γράφουμε το ευρετήριο στο δίσκο αφού ταξινομήσουμε τις λίστες σε λεξικογραφική σειρά
- Η ταξινόμηση πρέπει να γίνει ώστε να διευκολυνθεί η συγχώνευση σε μεταγενέστερο στάδιο
- Τελευταίο βήμα είναι η συγχώνευση των λιστών

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Distributed Indexing

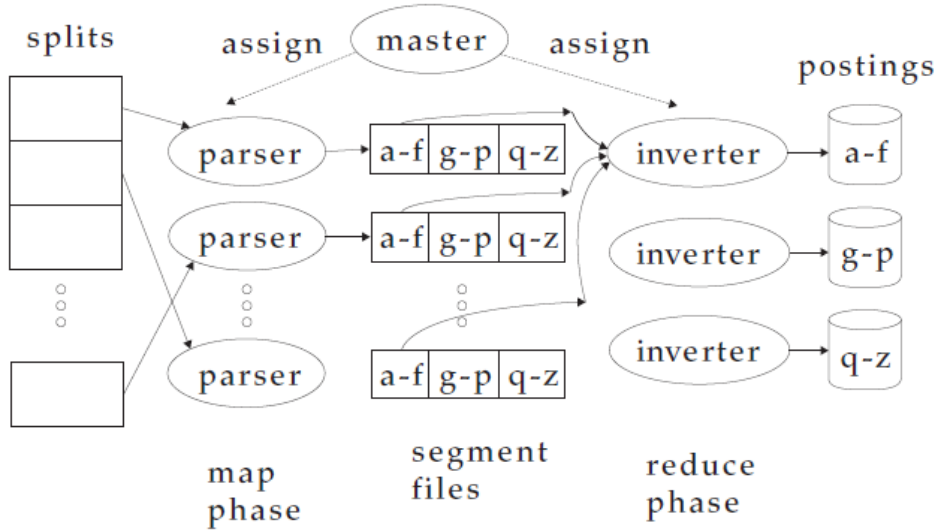
- Πολλές φορές λόγω του μεγάλου όγκου των δεδομένων, η κατασκευή του ευρετηρίου σε μια μηχανή δεν είναι δυνατή
- Υιοθετούμε ένα σύνολο από συστάδες (clusters)
- Υιοθετούμε μια κατανεμημένη τεχνική (distributed indexing)
- Το ευρετήριο κατανέμενεται σε ένα σύνολο μηχανών
- Η κατανομή γίνεται είτε σε σχέση με τους όρους ή σε σχέση με τα έγγραφα
- Οι μεγάλες μηχανές αναζήτησης προτιμούν την κατανομή με βάση τα έγγραφα

Distributed Indexing

- Μια αποδοτική τεχνική βασίζεται στην εφαρμογή του MapReduce μοντέλου
- Βασιζόμαστε σε μια συλλογή κόμβων (nodes) που έχουν συγκεκριμένα χαρακτηριστικά (επεξεργαστή, μνήμη, κ.λπ.)
- Οργανώνουμε την επεξεργασία που θέλουμε να κάνουμε σε τμήματα (chunks)
- Ένας master node 'οργανώνει' τους υπόλοιπους

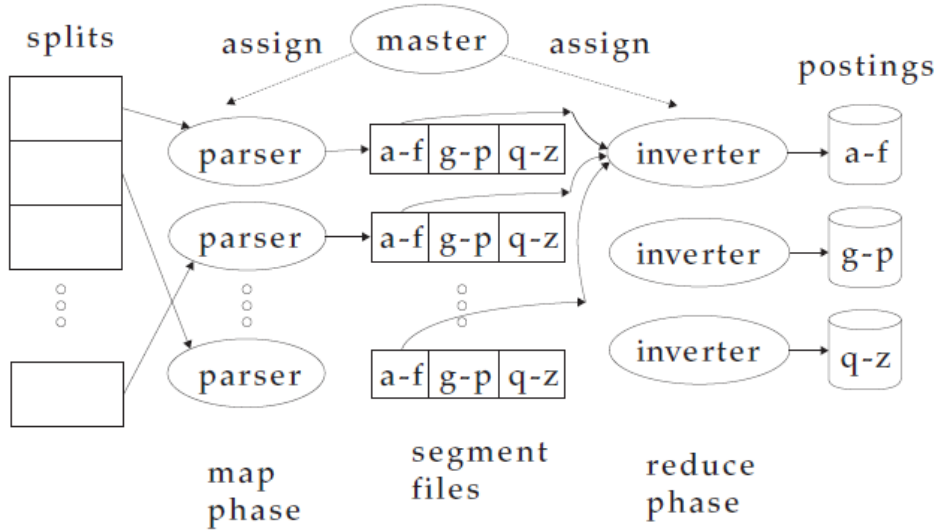
Distributed Indexing

- Τα δεδομένα εισόδου ‘σπάνε’ σε τμήματα ένα από το καθένα θα ανατεθεί σε ένα κόμβο για γρήγορη επεξεργασία
- Το μέγεθος του τμήματος επιλέγεται έτσι ώστε να έχουμε ισοκατανομή
- 16-64MB είναι ένα κατάλληλο μέγεθος
- Ο master node αναθέτει ένα τμήμα σε κάθε κόμβο
- Όταν ένας κόμβος τελειώσει τότε προχωρά στο επόμενο τμήμα
- Αν ένας κόμβος καταρρεύσει τότε το τμήμα ανατίθεται σε άλλον



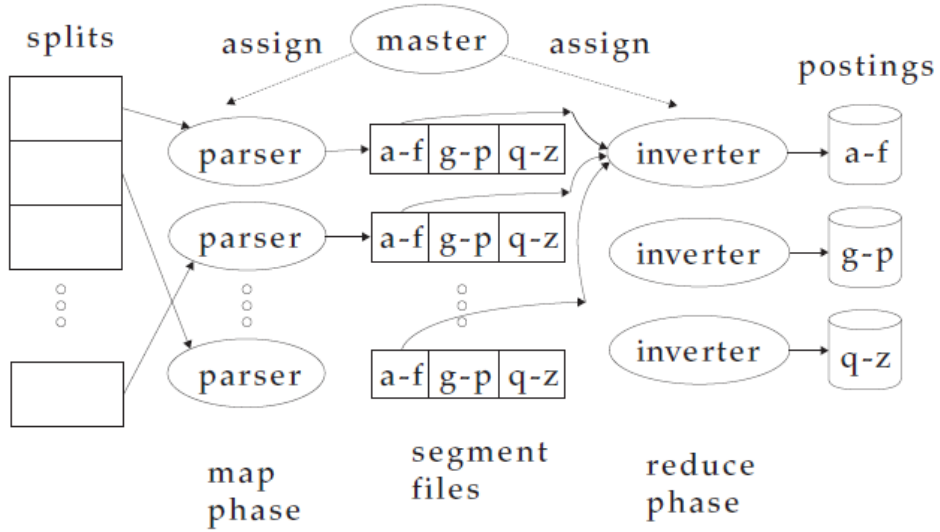
Distributed Indexing

- Το MapReduce μοντέλο αντιλαμβάνεται την είσοδο σαν ζεύγη key-value
- Στις μηχανές αναζήτησης το ζεύγος απεικονίζει termID-DocID
- Η αντιστοίχιση των termIDs γίνεται επίσης κατανεμημένα
- Επειδή η κατανεμημένη αντιστοίχιση είναι πιο πολύπλοκη κρατάμε τους πιο συχνούς όρους



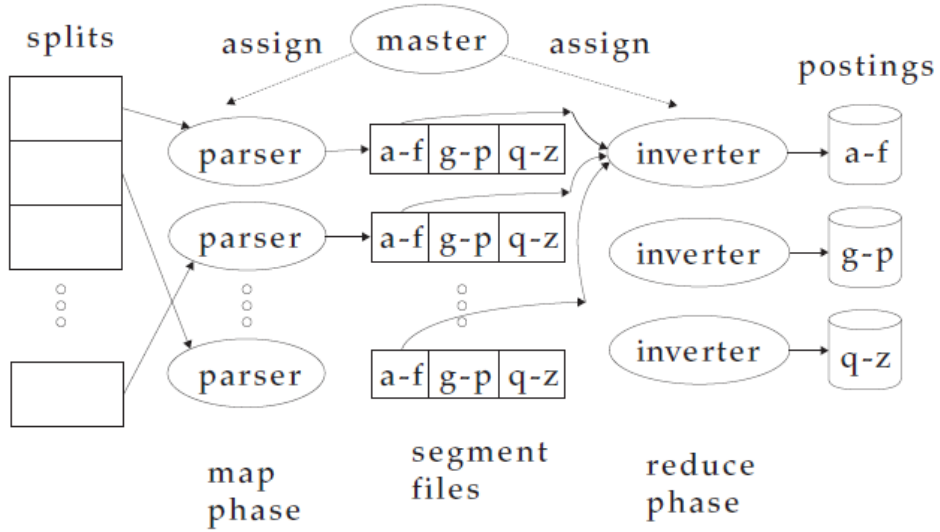
Distributed Indexing

- Η Map φάση έχει να κάνει με την αντιστοίχιση των εισόδων σε ζεύγη termID-docID (parser)
- Κάθε parser γράφει την έξοδο σε τοπικά αρχεία (segment files)
- Στην Reduce φάση θέλουμε όλες οι τιμές για ένα κλειδί να αποθηκευτούν σε κοντινές θέσεις
- Για αυτό το λόγο χωρίζουμε τα κλειδιά σε j partitions
- Οι parsers γράφουν τα ζεύγη για κάθε partition σε ξεχωριστό segment file



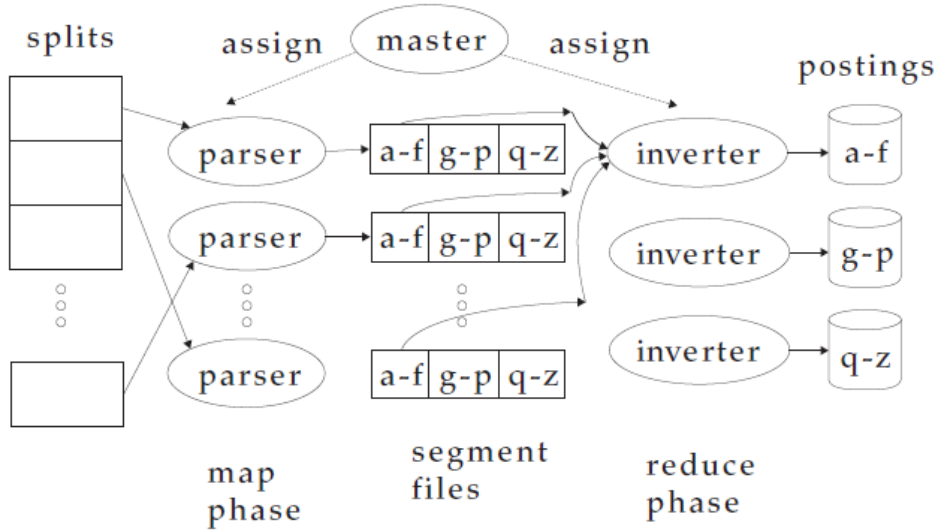
Distributed Indexing

- Στην εικόνα τα segment files έχουν χωριστεί με βάση το πρώτο γράμμα
- Το πως θα γίνουν τα partitions εξαρτάται από τους developers
- Όσοι είναι οι parsers τόσα θα είναι και τα αρχεία στα οποία θα υπάρχουν οι όροι



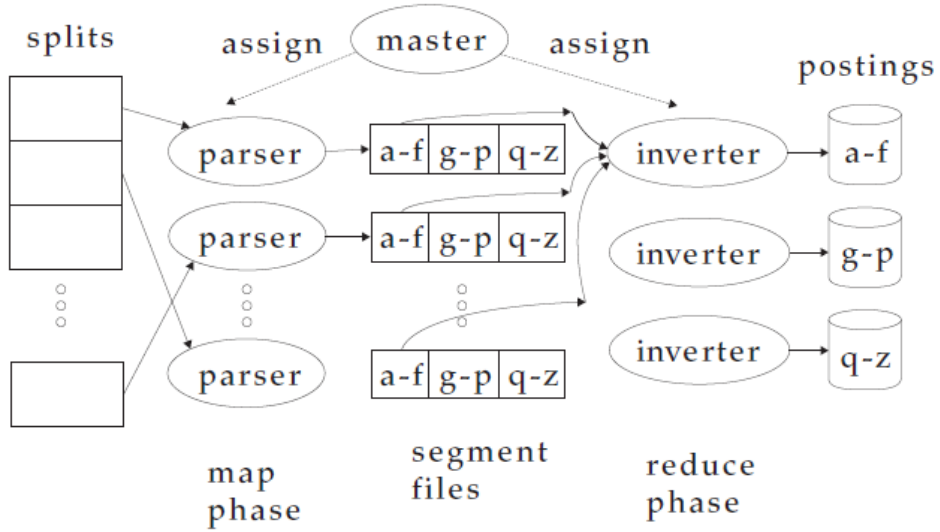
Distributed Indexing

- Στην reduce φάση οι inverters συλλέγουν όλα τα docIDs για κάθε κλειδί termID
- Κάθε partition τυγχάνει επεξεργασίας από ένα inverter
- Η λίστα των τιμών ταξινομείται για κάθε κλειδί και γράφεται στην τελική postings list



Distributed Indexing

- Η ίδια μηχανή μπορεί να είναι ένας parser στη Map φάση ή ένας inverter στην Reduce φάση
- Κάθε parser γράφει τα segment files στον τοπικό δίσκο για να ελαχιστοποιήσουμε τις εγγραφές πριν οι inverters 'μειώσουν' τα δεδομένα
- Κάθε segment file απαιτεί μια ακολουθιακή ανάγνωση αφού όλα τα δεδομένα που το αφορούν πάνε σε ένα συγκεκριμένο inverter



Dynamic Indexing

- Τι θα γίνει όταν η συλλογή των εγγράφων δεν είναι στατική;

- Νέοι όροι πρέπει να προστεθούν στο λεξικό
- Ο πιο απλός τρόπος είναι να ανακατασκευάζουμε περιοδικά το ευρετήριο από την αρχή
- Πρόκειται για καλή λύση όταν οι αλλαγές δεν είναι συχνές και η καθυστέρηση στην αναζήτηση στα νέα έγγραφα δεν είναι απαγορευτική

Dynamic Indexing

- Μια λύση για την εμπλοκή νέων εγγράφων είναι να διατηρήσουμε δύο ευρετήρια
  - Ένα μεγάλο βασικό
  - Ένα μικρό βοηθητικό (auxiliary index) που θα χειρίζεται τα νέα έγγραφα
- Το βοηθητικό ευρετήριο αποθηκεύεται στη μνήμη
- Οι αναζητήσεις γίνονται και στα δύο ευρετήρια και συγχωνεύονται
- Οι διαγραφές αποθηκεύονται σε διανύσματα και φιλτράρουμε τα αποτελέσματα
- Τα έγγραφα ενημερώνονται με διαγραφές και στη συνέχεια με νέα εισαγωγή

Dynamic Indexing

- Κάθε φορά που το βοηθητικό ευρετήριο γίνεται μεγάλο το συγχωνεύουμε με το κύριο ευρετήριο
- Αν έχουμε ένα αρχείο για κάθε postings list απλά συγχωνεύουμε τις λίστες του κύριου λεξικού με τις λίστες του βοηθητικού λεξικού
- Όμως αυτή η προσέγγιση δεν είναι εφικτή λόγω του μεγάλου πλήθους των αρχείων
- Η εναλλακτική είναι να αποθηκεύσουμε το λεξικό σαν ένα μεγάλο αρχείο
- Πρόκειται για τη συγχώνευση όλων των λιστών

Συμπίεση του Ευρετηρίου

Εισαγωγή

- Θα αναλύσουμε μεθόδους συμπίεσης του λεξικού
- Ο στόχος είναι να ελαχιστοποιήσουμε τις προσβάσεις στο δίσκο όταν επεξεργαζόμαστε διάφορα ερωτήματα
- Η συμπίεση στοχεύει στο να παράξει μια μορφή του λεξικού που θα χωρά στη μνήμη

Χρήση Αλφαριθμητικού

- Η πιο απλή περίπτωση είναι να ταξινομήσουμε το λεξικό και να το αποθηκεύσουμε σε ένα πίνακα

	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	…	…	…
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

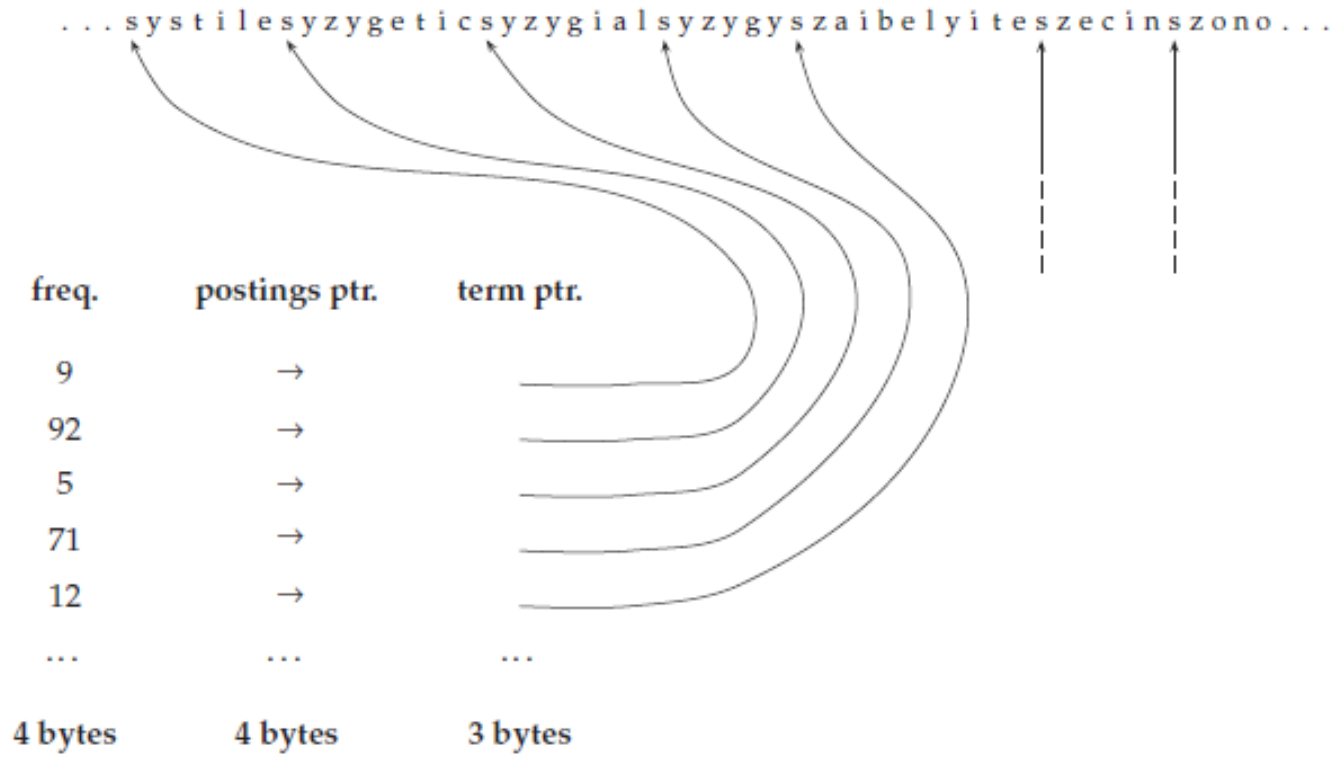
- Χρειαζόμαστε 20 bytes για κάθε όρο, 4 bytes για τη συχνότητα και 4 bytes για τους δείκτες προς τις postings lists
- Με χρήση δεικτών 4 bytes μπορούμε να ‘δείξουμε’ σε χώρο 4GB
- Για το Web χρειάζονται περισσότερα bytes για τους δείκτες

Χρήση Αλφαριθμητικού

- Όμως η χρήση σταθερού μεγέθους οδηγεί σε σπατάλη χώρου
- Ο μέσος όρος των όρων στα Αγγλικά είναι 8 χαρακτήρες
- Δεσμεύοντας όμως 20 bytes χωρίς λόγο χάνουμε τα 12 (κατά μέσο όρο)
- Επιπλέον δεν μπορούμε να αποθηκεύσουμε όρους με περισσότερα από 20 bytes

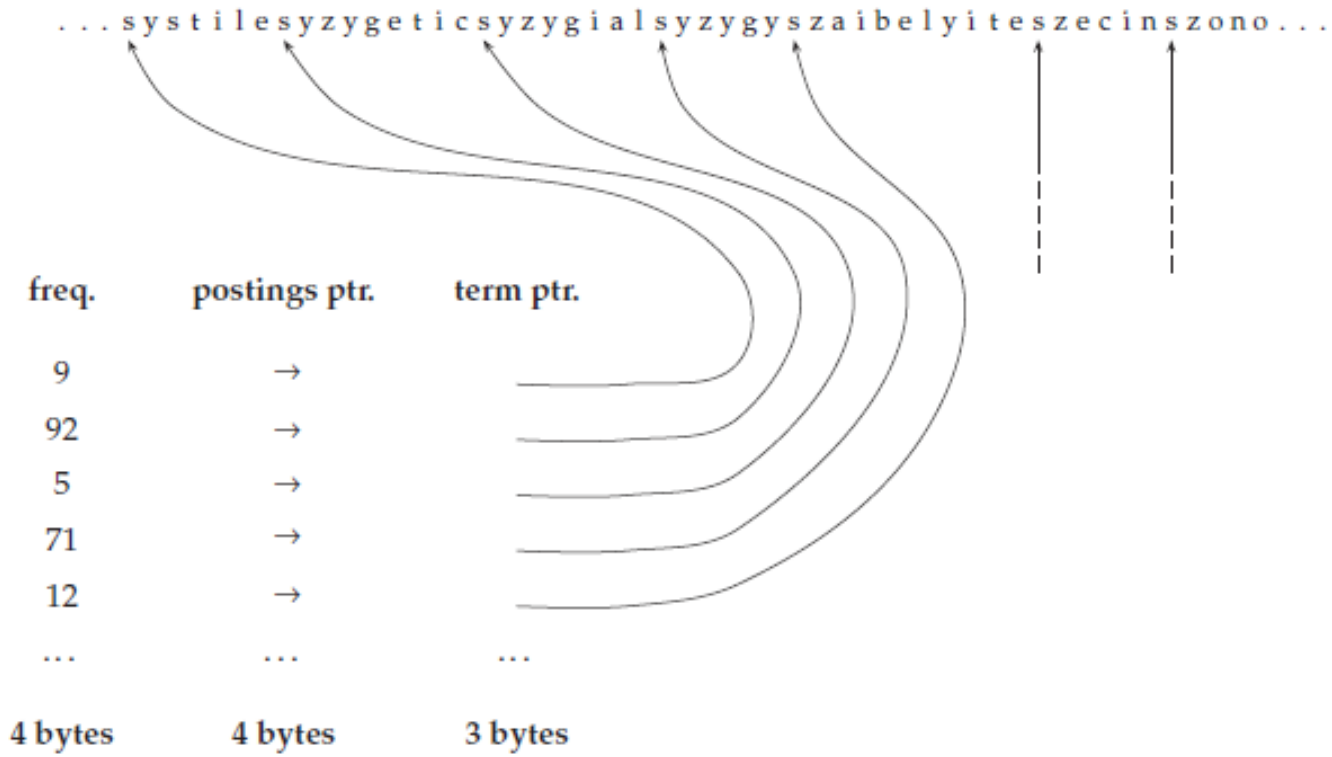
Χρήση Αλφαριθμητικού

- Μια λύση είναι να αποθηκεύσουμε το λεξικό σαν ένα μεγάλο αλφαριθμητικό
- Επιπλέον, ένας δείκτης προς τον επόμενο όρο υιοθετείται για το διαχωρισμό
- Η εύρεση μπορεί να υλοποιηθεί με δυαδική αναζήτηση
- Αυτό το σχήμα εξοικονομεί 60% λιγότερο χώρο σε σχέση με την αρχική λύση



Χρήση Αλφαριθμητικού

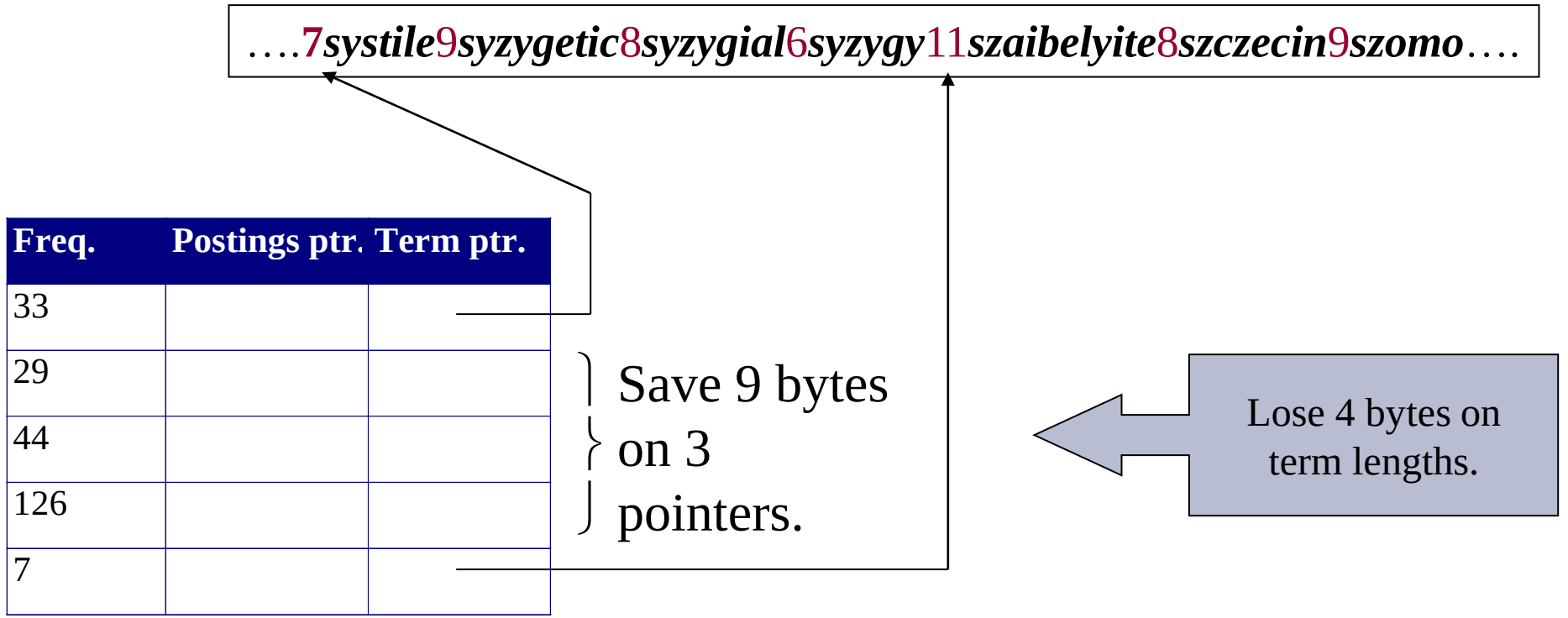
- Χρειαζόμαστε όμως περισσότερο χώρο για την αποθήκευση των δεικτών
- Χρειαζόμαστε περίπου 3 bytes για τους δείκτες προς τους επόμενους όρους



Blocked Storage

- Μπορούμε να συμπιέσουμε περαιτέρω το λεξικό αν ομαδοποιήσουμε το αλφαριθμητικό σε ομάδες των k όρων
- Χρειάζεται μόνο ένας δείκτης προς τον πρώτο όρο κάθε ομάδας
- Αποθηκεύουμε επίσης το μήκος των όρων στο αλφαριθμητικό με ένα επιπλέον byte στην αρχή κάθε όρου
- Με αυτό τον τρόπο αποφεύγουμε τη χρήση k-1 δεικτών
- Όμως, χρειαζόμαστε k bytes για το μήκος των όρων

Blocked Storage



Blocked Storage

- Αν αυξήσουμε το k θα πάρουμε καλύτερη συμπίεση
- Όμως υπάρχει ένα trade off ανάμεσα στο λόγο συμπίεσης και στην ταχύτητα αναζήτησης
- Η αναζήτηση των όρων στο μη-συμπιεσμένο λεξικό γίνεται με τη δυαδική αναζήτηση
- Στο συμπιεσμένο λεξικό αρχικά αναζητούμε και εντοπίζουμε το block πάλι με χρήση δυαδικής αναζήτησης
- Στη συνέχεια βρίσκουμε τη θέση του όρου με γραμμική αναζήτηση

Blocked Storage

- Υπάρχει πλεονασμός στο λεξικό στο γεγονός ότι κάθε συνεχόμενος όρος μπορεί να μοιράζεται κοινό πρόθεμα με τους επόμενους
- Η τεχνική του front coding επιλύει το συγκεκριμένο πλεονασμό
- Το κοινό πρόθεμα μπορεί να αναγνωριστεί για ένα σύνολο όρων
- Αναφερόμαστε σε αυτό με ένα ειδικό χαρακτήρα

One block in blocked compression ($k = 4$) . . .
8automata8automate9automatic10automation

⇓

. . . further compressed with front coding.
8automat∗a1⋄e2 ⋄ ic3⋄ion

Άλλες Τεχνικές

- Υπάρχει η πιθανότητα να μην χωρά το λεξικό στη μνήμη ακόμα και έπειτα από συμπίεση
- Σε αυτή την περίπτωση μπορούμε να χωρίσουμε το λεξικό σε σελίδες
- Χρησιμοποιούμε δείκτες προς τον πρώτο όρο κάθε σελίδας υιοθετώντας ένα B-δένδρο
- Το σύστημα προσπελαύνει το δίσκο για να πάρει τις postings lists
- Επιπλέον, αποκτούμε πρόσβαση στην κατάλληλη σελίδα του λεξικού στο δίσκο

Στατιστικά

Statistic	Value
documents	800,000
avg. # tokens per document	200
terms	400,000
avg. # bytes per token (incl. spaces/punct.)	6
avg. # bytes per token (without spaces/punct.)	4.5
avg. # bytes per term	7.5
tokens	100,000,000

Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9