Ανάκληση Πληροφορίας

Information Retrieval

Διδάσκων –
Δημήτριος Κατσαρός

Θεωρία και Ασκήσεις σε εξειδικευμένα ζητήματα σχετικά με τον Inverted Index

Caching σε Μηχανές Αναζήτησης

Πολιτικές caching σε μηχανές αναζήτησης

- Παραδοσιακές πολιτικές αντικατάστασης (replacement policies): *LRU*, *LRU-k*, *LFU*, *FBR*
- *Function-based* πολιτικές αντικατάστασης:
  - c_x: το κόστος φόρτωσης του *x* στην cache (ανάλογο του μεγέθους του x, π.χ., σε bytes)
  - g_x: το κέρδος από την χρήση του cached *x* αντί για την ανάκτησή του από τον δίσκο (π.χ., ο service time του query, ο χρόνος εκτέλεσης list intersection, ο χρόνος για την φόρτωση της postings list στην κύρια μνήμη)
  - p_x: πόσες φορές αναμένουμε να χρησιμοποιηθεί το *x* σε μια χρονική περίοδο (δηλαδή, πιθανότητα χρήσης)
- Θύμα αντικατάστασης (replacement victim) με βάση την εξής σχέση:

$$ENG_x = \frac{p_x \times g_x}{c_x}$$

Άσκηση-01

- Θεωρήστε την πολιτική caching $\mathbf{Q_{TF}D_F}$, η οποία αποθηκεύει στην cache τις postings lists των n top-scoring terms, όπου το score ορίζεται ως εξής:

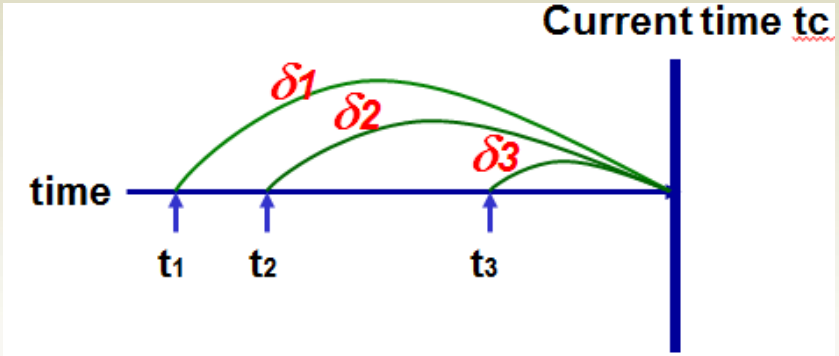
$$score(t) = \frac{f_Q(t)}{N_t}$$

όπου το $f_Q(t)$ είναι ο αριθμός εμφάνισης του t σε ένα query (με βάση το query log), και N_t είναι ο αριθμός των εγγράφων που περιέχουν τον t.

- ΕΡΩΤΗΜΑ: Υπό ποιες προϋποθέσεις η πολιτική είναι ισοδύναμη με την πολιτική **ENB**;

Άσκηση-02

- Έστω μια πολιτική αντικατάστασης για caching που χειρίζεται το caching των postings lists ως εξής:
  - Χρησιμοποιεί μια συνάρτηση F με $F(x)=(1/p)^{\lambda x}$ με $p \geq 2$ και $\lambda \in [0...1]$
  - Αναθέτει σε κάθε στοιχείο (posting list) d που βρίσκεται στην cache μια τιμή val που υπολογίζεται συναρτήσει των χρονικών στιγμών που προσπελάστηκε το στοιχείο όταν βρισκόταν στην cache
  - Για παράδειγμα, για ένα στοιχείο που προσπελάστηκε τρεις φορές, τις στιγμές t1, t2 και t3 όπως φαίνεται στο παρακάτω σχήμα



υπολογίζει την val ως εξής $val(d)=F(tc-t1)+F(tc-t2)+F(tc-t3)$, όπου F() είναι συνάρτηση που ορίστηκε παραπάνω.

- ΕΡΩΤΗΜΑ: Ποιες πολιτικές αντικατάστασης παράγονται όταν μεταβάλλουμε το λ;

Άσκηση-03

Έστω ο αλγόριθμος caching με όνομα *GreedyDual-Size*, ο οποίος χειρίζεται το caching των postings lists:

- *Αρχικοποίηση x=n=0*
- *Επεξεργασία κάθε αίτησης για postings list σε σειρά*
- *Η τρέχουσα αίτηση είναι για την postings list p*
- *Εάν η p είναι ήδη στην cache, θέστε H(p)=x + c(p)/s(p)*
- *Εάν η p δεν βρίσκεται στην cache*
  - *Όσο δεν υπάρχει αρκετός ελεύθερος χώρος στην cache για την p,*
    - *Έστω n είναι το minimum H(q) για όλες τις postings lists q της cache*
    - *Διώξτε από την cache την q για την οποία ισχύει H(q) = n*
    - *Στεγάστε την p στην cache και θέστε H(p)= x + c(p)/s(p) και x=H(p)*

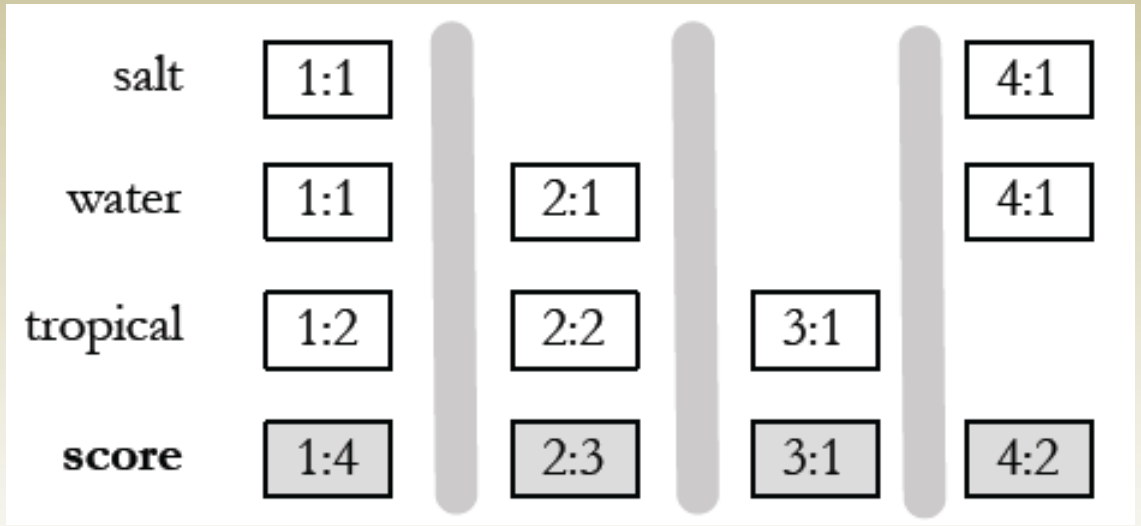
ΕΡΩΤΗΜΑ: Υπο ποιες προϋποθέσεις, ο GreedyDual-Size είναι ισοδύναμος του LRU; Εξηγήστε.

Αποτίμηση ερωτήματος: doc-at-a-time & term-at-a-time

Doc-at-a-Time (DaaT) επεξεργασία

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

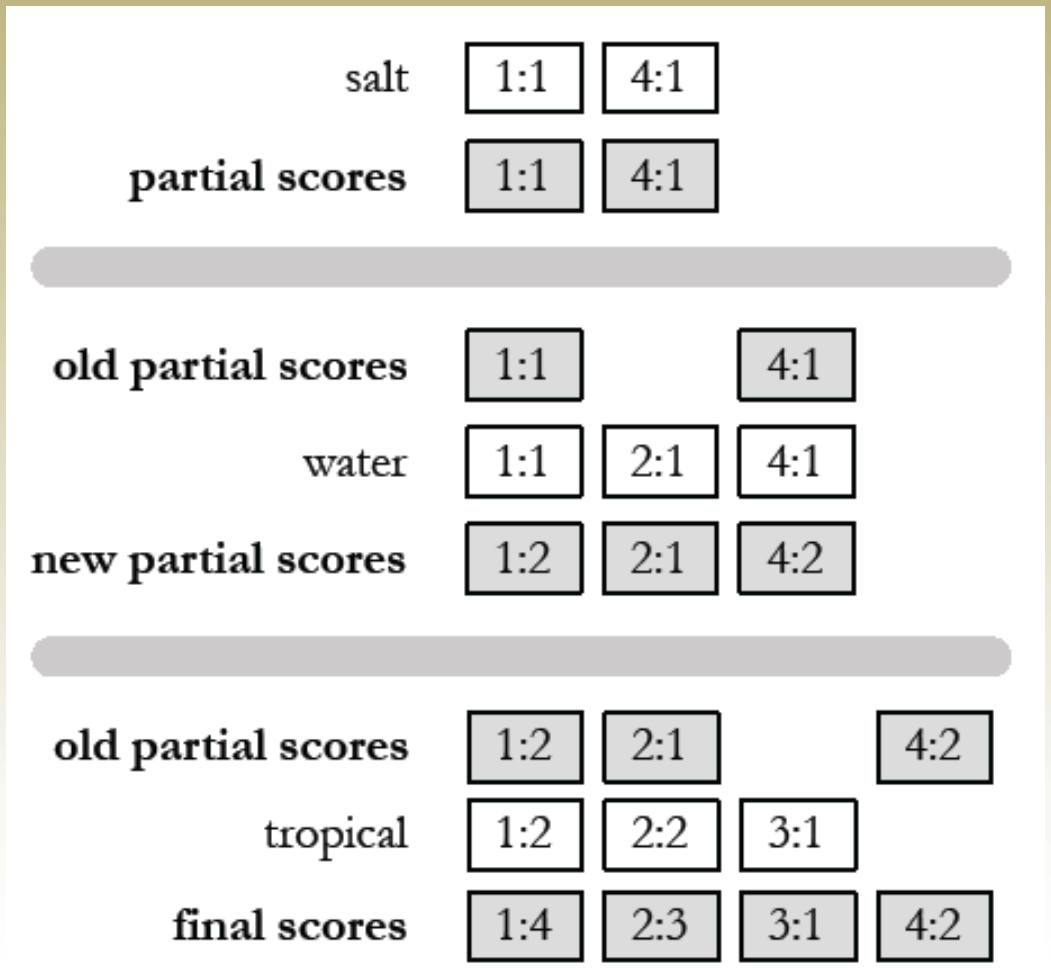
Doc-at-a-Time (DaaT) επεξεργασία



Term-at-a-Time (TaaT) επεξεργασία

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q) f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                    ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Term-at-a-Time (TaaT) επεξεργασία



Άσκηση

- Doc-at-a-time:
  - Παράλληλη σάρωση με ακουλουθιακό τρόπο των postings λιστών
- Term-at-a-time:
  - Ταξινόμηση των query term σε αύξουσα συχνότητα
  - Επαναληπτική συγχώνευση με την επόμενη postings λίστα
- Ποια απο τις δυο τεχνικές είναι πιο αποδοτική στην εξοικονόμηση μνήμης; Γιατί;
- Ποια απο τις δυο τεχνικές είναι πιο αποδοτική στην ελάττωση προσπέλασης στον δίσκο; Γιατί;

- Με βάση την συζήτηση:
  - Τι σημαίνει document-based κατανομή του index σε πολλαπλές μηχανές;
  - Τι σημαίνει term-based κατανομή του index σε πολλαπλές μηχανές;
  - Ποια από τις δυο τεχνικές πλεονεκτεί & σε ποιες καταστάσεις;

RankBM25 DaaT επεξεργασία ερωτήματος

Θυμηθείτε τον ορισμό του term frequency κατά Okapi BM25:

$$Score_{BM25}(q,d) = \sum_{t \in q} \left(\frac{N}{N_q} \right) \times TF_{BM25}(t,d)$$

$$TF_{BM25}(t,d) = \frac{f_{t,d} \times (k_1 + 1)}{f_{t,d} + k_1 \times ((1-b) + b \times (l_d / l_{avg})}$$

Έχει δυο ανεξάρτητες παραμέτρους:

- k_1 (εξ ορισμού τιμή k_1 =1.2): ελέγχει πόσο γρήγορα ο παράγων TF γίνεται "επίπεδος" (saturates)
- b (εξ ορισμού τιμή b =0.75): ελέγχει τον βαθμό της κανονικοποίησης του μήκους εγγράφου

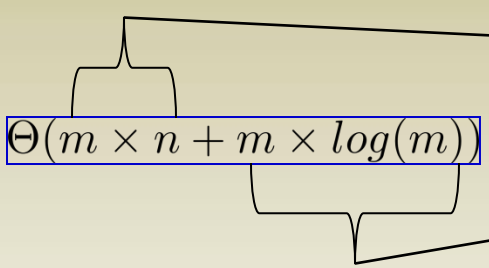
Παρατηρήστε ότι: $\lim_{f_{t,d} \to \infty} TF_{BM25}(t,d) = k_1 + 1$

- Για k_1=1.2, η συνεισφορά του TF για οποιονδήποτε query term:

Δεν υπερβαίνει το 2.2

- Εξ αιτίας του ορίου αυτού, ένα έγγραφο που περιέχει δυο διαφορετικούς όρους είναι πολύ πιο πιθανό να διαταχθεί υψηλότερα από ένα έγγραφο που περιέχει μόνο έναν query term, ακόμη και εάν τόν περιέχει πολλές φορές

RankBM25 DaaT επεξεργασία ερωτήματος




```
    rankBM25_DocumentAtATime (⟨t_1, ..., t_n⟩, k) ≡
1       m ← 0   // m is the total number of matching documents
2       d ← min_{1≤i≤n}{nextDoc(t_i, −∞)}
3       while d < ∞ do
4           results[m].docid ← d
5           results[m].score ← Σ_{i=1}^{n} log(N/N_{t_i}) · TF_BM25(t_i, d)
6           m ← m + 1
7           d ← min_{1≤i≤n}{nextDoc(t_i, d)}
8       sort results[0..(m − 1)] in decreasing order of score
9       return results[0..(k − 1)]
```

Πηγές αναποτελεσματικότητας του αλγορίθμου:

- Οι υπολογισμοί στις Γραμμές 5–7 γίνονται για όλους τους *n* όρους, ακόμη και εάν δεν εμφανίζονται στο τρέχον έγγραφο.
  - Για μεγάλα n, αυτό είναι πρόβλημα. Σκεφτείται όταν *n*=10 και κάθε έγγραφο περιέχει μόνο έναν από αυτούς
- Το βήμα ταξινόμησης (Γραμμή 8) ταξινομεί όλα τα έγγραφα στον πίνακα αποτελεσμάτων
  - Η ταξινόμηση ολόκληρου του πίνακα είναι περιττή, αφού μάς ενδιαφέρουν μόνο τα top-k αποτελέσματα. Η *Θ(m· log(m))* πολυπλοκότητα ίσως φαίνεται αποδεκτή, αλλά εάν σκεφτούμε ότι χειριζόμαστε μερικά εκατομύρια από έγγραφα, τότε ο όρος *m* x *log(m)* μπορεί να υπερκεράσει κατά πολύ το γινόμενο *m* x *n*

RankBM25 DaaT επεξεργασία ερωτήματος

- Φυσικά, με βάση το πόσο συχνά οι query terms συνεμφανίζονται σε ένα έγγραφο, το m μπορεί να πάρει τιμή μεταξύ N_q/n και N_q, όπου $Nq = N_{t1} + \cdot\ \cdot\ \cdot + N_{tn}$ είναι ο συνολικός αριθμός των postings όλων των query terms
- Στην χείριστη περίπτωση, κάθε matching έγγραφο περιέχει μόνο έναν query term
- Τότε η υπολογιστική πολυπλοκότητα του προηγουμένου αλγορίθμου είναι:

$$\Theta(N_q \text{ x } n + N_q \text{ x } log(N_q))$$

Μπορούμε να σχεδιάσουμε έναν διαφορετικό αλγόριθμο, στον οποίο χρησιμοποιούμε δυο heaps:

- Ο ένας heap χειρίζεται τους query terms και, για κάθε term t, δείχνει στο επόμενο έγγραφο το οποίο περιέχει τον t
- Ο άλλος heap χειρίζεται το σύνολο των top-k αποτελεσμάτων μέχρι στιγμής

RankBM25 DaaT με binary heaps

Sort in increasing order of score

Gets all docs with the query terms

Gets the docs with the lowest ID

Process one doc

Replace the worst doc

Sort in decreasing order of score

```
rankBM25_DocumentAtATime_WithHeaps (⟨t_1, ..., t_n⟩, k) ≡
1   for i ← 1 to k do   // create a min-heap for the top k search results
2       results[i].score ← 0
3   for i ← 1 to n do   // create a min-heap for the n query terms
4       terms[i].term ← t_i
5       terms[i].nextDoc ← nextDoc(t_i, −∞⟩
6   sort terms in increasing order of nextDoc   // establish heap property for terms
7   while terms[0].nextDoc < ∞ do
8       d ← terms[0].nextDoc
9       score ← 0
10      while terms[0].nextDoc = d do
11          t ← terms[0].term
12          score ← score + log(N/N_t⟩ · TF_BM25(t, d)
13          terms[0].nextDoc ← nextDoc(t, d)
14          reheap(terms⟩   // restore heap property for terms
15      if score > results[0].score then
16          results[0].docid ← d
17          results[0].score ← score
18          reheap(results)   // restore heap property for results
19  remove from results all items with score = 0
20  sort results in decreasing order of score
21  return results
```

RankBM25 DaaT με binary heaps

- Ο terms heap περιέχει το σύνολο των query terms, διατεταγμένων με βάση το επόμενο έγγραφο στο οποίο εμφανίζεται ο όρος (nextDoc)
    - Μάς επιτρέπει να εκτελέσουμε multiway merge στις *n* postings λίστες
- Ο results heap περιέχει τα top-k έγγραφα μέχρι στιγμής, διατεταγμένα με βάση το score τους
    - Ο ριζικός κόμβος του results (δηλ., results[0]) δεν περιέχει το καλύτερο έγγραφο μέχρι στιγμής, αλλά το k^{οστό} –καλύτερο μέχρι στιγμής
    - Αυτό μάς επιτρέπει να δαιτηρούμε και ν' ανανεώνουμε συνεχώς τα top-k αποτελέσματα της αναζήτησης αντικαθιστώντας το lowest-scoring έγγραφο στα top k (και αποκαθιστώντας την ιδιότητα του heap) οποτεδήποτε βρίσκουμε ένα νέο έγγραφο καλύτερο από το κορυφαίο του σωρού

➢ Η χείριστη πολυπλοκότητα του προηγουμένου αλγορίθμου είναι:

$$\Theta(N_q \text{ x } \log(n) + N_q \text{ x } log(k))$$

Ο όρος (N_q x $log(n)$) αντιστοιχεί στις λειτουργίες Reheap πάνω στον terms heap (αποκατάσταση της ιδιότητας του σωρού μετά από κάθε posting). Ο όρος (N_q x $log(k)$) αντιστοιχεί στις λειτουργίες Reheap πάνω στον results heap, (αποκατάσταση της ιδιότητας του σωρού οποτεδήποτε ένα νέο έγγραφο προστίθεται στο σύνολο των top-k αποτελεσμάτων

Pruning με MaxScore

- Επανερχόμαστε τώρα στο μέγιστο score που οφείλεται σε έναν μόνο όρο, και αυτό είναι:

$$\mathbf{2.2 \times} \mathit{log}(N/N_t)$$

- Αυτό το άνω φράγμα αποκαλείται το **MaxScore** του όρου
- Θεωρήστε τώρα το query

Q = ‹ “greek”, “philosophy”, “stoicism” ›

και έστω ότι ισχύουν τα κάτωθι:

Term	N_t	$\log_2(N/N_t)$	MaxScore
“greek”	4,504	6.874	15.123
“philosophy”	3,359	7.297	16.053
“stoicism”	58	13.153	28.936

Pruning με MaxScore

- Υποθέστε ότι κάποιος χρήστης που υπέβαλλε το προηγούμενο ερώτημα ενδιαφέρεται για τα top-k = 10 αποτελέσματα
- Μετά τον υπολογισμό των scores για μερικές εκατοντάδες έγγραφα, μπορεί να προκύψει η κατάσταση στην οποία το 10^{ο}-καλύτερο αποτέλεσμα που ευρέθη μέχρι στιγμής υπερβαίνει το μέγιστο maximum score του όρου “greek”. Δηλαδή:

results[0].score > MaxScore(“greek”) = 15.123

- Όταν συμβεί αυτό, γνωρίζουμε ότι κάθε έγγραφο που περιέχει μόνο τον όρο “greek”, αλλά ούτε τον όρο “philosophy” ούτε το όρο “stoicism”, δεν μπορεί ποτέ να μπει στην λίστα με τα top-10 αποτελέσματα
- Συνεπώς, δεν υπάρχει ανάγκη να υπολογίζουμε το score για εκείνα τα έγγραφα που περιέχουν μόνο τον όρο “greek”
- Μπορούμε να διαγράψουμε τον term από τον terms heap και να κοιτάζουμε στα postings του μόνο όταν βρίσκουμε ένα έγγραφο το οποίο περιέχει έναν από τους άλλους δυο όρους

Pruning με MaxScore

- Καθώς συνεχίζουμε τον υπολογισμό των scores για περισσότερα έγγραφα, μπορεί να φτάσουμε σε ένα σημείο όπου ισχύει το εξής:

  results[0].score > MaxScore(“greek”)+MaxScore(“philosophy”) = 31.176
- Στο σημείο αυτό, γνωρίζουμε ότι ένα έγγραφο μπορεί να μπει στην λίστα των top-10 εάν και μόνο εάν περιέχει τον όρο “stoicism”, και άρα μπορούμε να διαγράψουμε τον όρο “philosophy” από τον terms heap

- Η στρατηγική αυτή ονομάζεται MaxScore
- Η MaxScore εγγυάται ότι θα παράξει το ίδιο σύνολο των top-k αποτελεσμάτων όπως και ο απλοϊκός αλγόριθμος, αλλά πολύ γρηγορότερα, διότι αγνοεί όλα τα έγγραφα για τα οποία είναι γνωστό εκ των προτέρων ότι δεν μπορούν να είμαι μέλη του συνόλου των τελικών top-k

Pruning με MaxScore

- Σημειώστε ότι παρόλο που η τεχνική MaxScore διαγράφει μερικούς όρους από τον heap, εξακολουθεί να τους χρησιμοποιεί για το scoring
  - Αυτό γίνεται με την χρήση δυο δομών δεδομένων, μια για τους terms οι οποίοι βρίσκονται ήδη στον heap, και η άλλη για τους όρους που έχουν διαγραφεί από τον heap
  - Οποτεδήποτε βρίσκουμε ένα έγγραφο d το οποίο περιέχει έναν από τους όρους που βρίσκονται ήδη στον heap, ψάχνουμε στο σύνολο των όρων οι οποίοι διαγράφησαν από τον heap, και για κάθε έναν από αυτούς t καλούμε την nextDoc(t, $d-1$) ώστε να προσδιορίσουμε εάν ο t εμφανίζεται στο d. Εάν εμφανίζεται, υπολογίζουμε την συνεισφορά του t στο score και την αθροίζουμε στο score του d

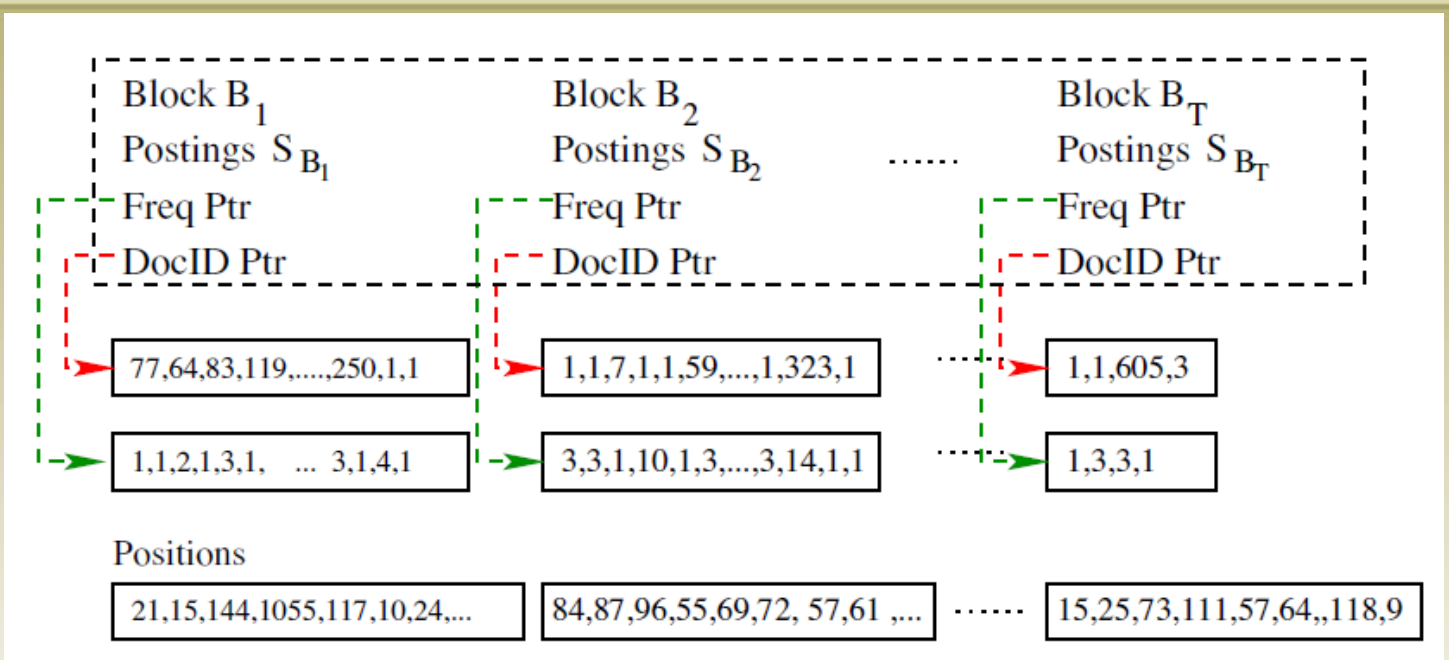
Pruning με MaxScore

	Without MaxScore		
	Wall Time	CPU	Docs Scored
OR, k=10	400 ms	304 ms	$4.4 \cdot 10^6$
OR, k=100	402 ms	306 ms	$4.4 \cdot 10^6$
OR, k=1000	426 ms	329 ms	$4.4 \cdot 10^6$

	With MaxScore		
	Wall Time	CPU	Docs Scored
OR, k=10	188 ms	93 ms	$2.8 \cdot 10^5$
OR, k=100	206 ms	110 ms	$3.9 \cdot 10^5$
OR, k=1000	249 ms	152 ms	$6.2 \cdot 10^5$

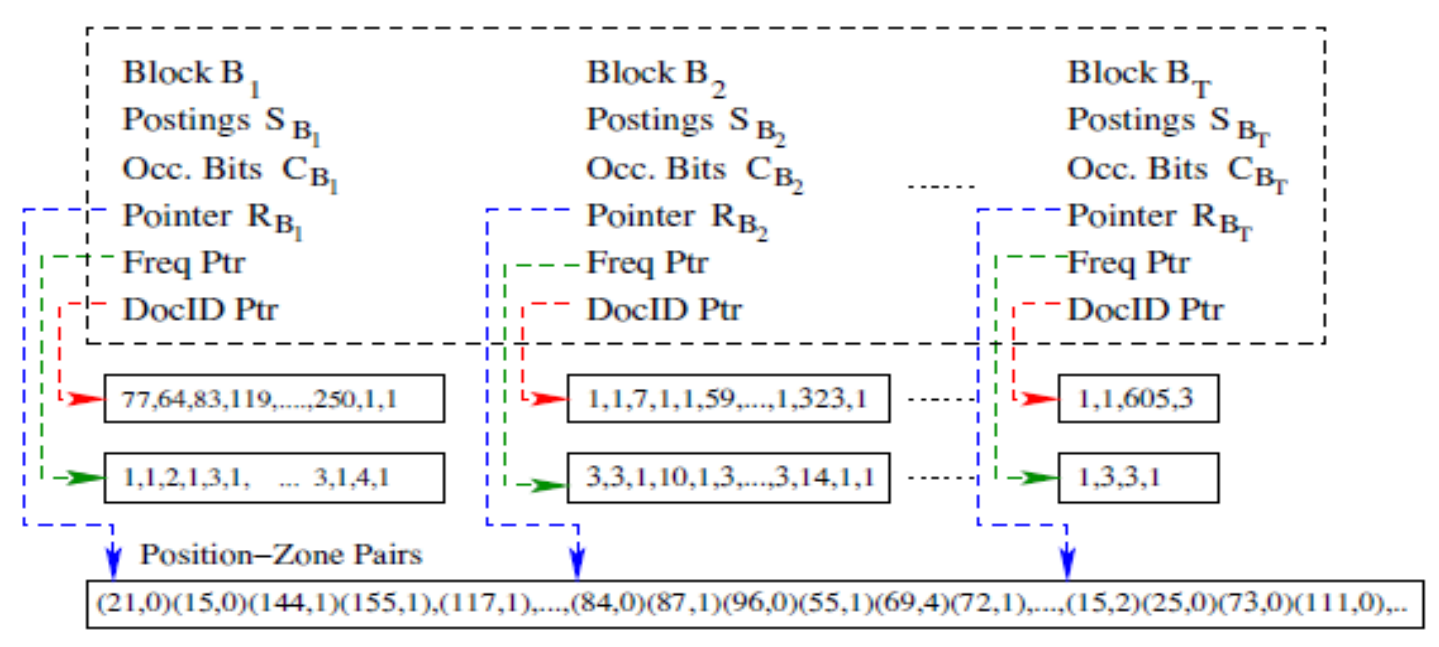
Η πραγματική δομή ενός (positional) inverted index

Inverted index οργανωμένος σε blocks



- Interleaving:
  - positions μετά από docIDs και μετά από frequencies του block
- Δημιουργία μιας χωριστής δομής για τα positions
  - (Μια δι-επίπεδη) Indexed list: το επάνω επίπεδο περιέχει ptrs σε δεδομένα αποθηκευμένα στο κατώτερο επίπεδο
    - Γρήγορη προσπέλαση, αλλά storage overhead (ένας ptr ανά posting)

Inverted index με ζώνες & οργανωμένος σε blocks



- Ζώνες

Zone	ZoneID	HTML
Body (normal text)	0	<body>... </body>
Anchor text	1	<a>...
Title text	2	<title>... </tile>
Document's URL	3	-
Headings	4	... , ... ,...
Page description	5	<meta name=' 'Description' ' Content="...">
Image description	6	<img alt="..." ...
Label text	7	<label>... </label>