ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΙΤΛΟΣ: *Ζητήματα caching στις μηχανές αναζήτησης*

ΤΕΛΩΝΙΑΤΗΣ ΕΥΑΓΓΕΛΟΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ:
Κατσαρός Δημήτριος

ΒΟΛΟΣ
2011

ΠΕΡΙΕΧΟΜΕΝΑ

Κεφάλαιο 1. Εισαγωγή……………………………………………………........6

1.1 Internet και Web…………………………………………...………6
1.2 Μηχανές Αναζήτησης..…………………………………...……….. 8
1.2.1 Δομή – Περιγραφή………………………………..……...8
1.2.2 Στόχοι μιας μηχανής αναζήτησης……………….…..…….. 11

Κεφάλαιο 2. Caching…………....…………………………………………..12

2.1 Μηχανισμός……………………………………...………..……. 12
2.1.1 Μείωση χρόνου απόκρισης…………………...………..….12
2.1.2 Μείωση φόρτου εργασίας στο search cluster…………..……. 13
2.2 Γιατί caching;……………………………………….……….....… 13
2.3 Πολιτικές αντικατάστασης στην cache ……………………..……..14
2.3.1 LRU και LFU…………….………………………….14
2.3.2 FIFO και Random…….…………………………………..14
2.3.3 FIFO vs. LRU…….…….…………………………….....15
2.3.4 Μέγεθος cache και πολιτική αντικατάστασης…………….15
2.4 Freshness……..……………………………………………………17
2.4.1 Πολύ μεγάλες caches………………………………..……. 17
2.4.2 Πρόβλημα: stale entries…………….…………………… 17
2.4.3 Λύση: cache invalidation…………………..……………... 18
2.5 Refreshing…………………………………………………………18
2.5.1 Κεντρική ιδέα………………………………………….. 18
2.5.2 Πολιτική ανανέωσης……………………………….... 19
2.6 Flushing……………………………………………………………19

Κεφάλαιο 3. Υλοποίηση………………………………………………… 21

3.1 Lucene…………………………………………………………….. 21
3.2 Συλλογή δεδομένων………………………………………………… 23
3.3 Query log…….…...…………………………………………………23
3.4 Cache……………………………………………………………… 25
3.4.1 Χωρητικότητα……………………………………………. 25

Κεφάλαιο 4. Μετρήσεις…………………………………………………………..26

4.1 Κατανομή Zipf ……………………………………………………….26
4.1.1 Zipf και hit ratio…………………………………………27
4.2 TTL Values…………………………………………………………30
4.3 Refreshing…………………………………………………………32
4.3.1 Flushing………..……………….…………………………..34

Κεφάλαιο 5. Συμπεράσματα και μελλοντική δουλειά………………………37

5.1 Συμπεράσματα …………………………………………………37
5.2 Μελλοντική δουλειά………...……………………………...……...38

References…………………………………………………………………39

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου, κ. Κατσαρό Δημήτριο για την καθοδήγηση και πολύτιμη βοήθεια του για την ολοκλήρωση της διπλωματικής μου εργασίας.

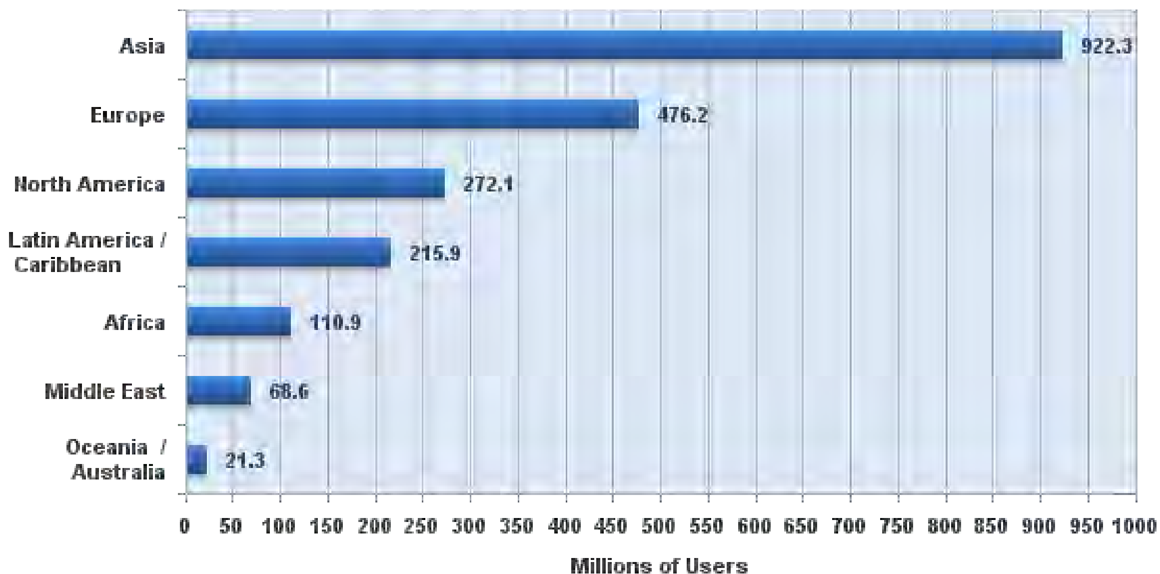
Ευχαριστώ από καρδιάς τους γονείς μου για την αμέριστη συμπαράσταση, την οικονομική στήριξη και την εμπιστοσύνη τους στις επιλογές μου όλα αυτά τα χρόνια των σπουδών μου. Εξίσου ευχαριστώ, για τις συμβουλές και την υποστήριξη του, εδώ και πολλά χρόνια, το θείο μου Αντώνη.

Επιπλέον, ένα μεγάλο ευχαριστώ στους φίλους και συμφοιτητές μου για την αλληλοϋποστήριξη και για τις ωραίες στιγμές που περάσαμε μαζί κατά την διάρκεια των σπουδών μας.

Κεφάλαιο 1ο - Εισαγωγή

1.1 Internet και Web

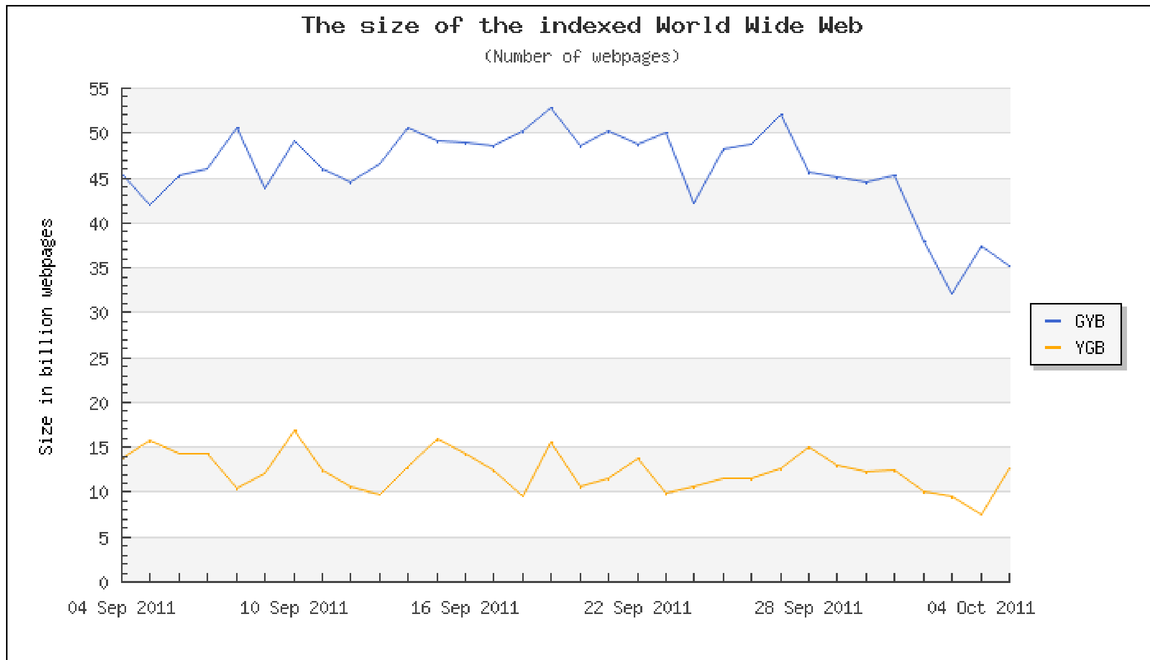
Η πρόσβαση στο **Διαδίκτυο (Internet)** είναι σήμερα πιο εύκολη από ποτέ. Υπολογιστές, και γενικότερα συσκευές υπολογιστικών συστημάτων, συνδεδεμένα στο διαδίκτυο υπάρχουν σχεδόν παντού, ενώ εκτιμάται ότι περίπου 2,1 δισεκατομμύρια άνθρωποι έχουν πρόσβαση σε αυτό σήμερα.



Εικόνα 1: Αριθμός χρηστών του Ιντερνέτ κατά ήπειρο

Ανάμεσα στις υπηρεσίες που προσφέρει, η πιο δημοφιλής, είναι ο **Παγκόσμιος Ιστός (World Wide Web ή απλά Web)**, ένα σύστημα διασυνδεδεμένων αρχείων «υπερκειμένου» (hypertext), κοινώς γνωστές ως **ιστοσελίδες (webpages)**, προσβάσιμες μέσω του Internet. Με ένα πρόγραμμα περιήγησης ιστού (web browser), μπορεί κανείς να δει ιστοσελίδες που μπορεί να περιέχουν κείμενο, εικόνες, βίντεο και άλλα πολυμέσα, και να πλοηγηθεί μεταξύ τους μέσω **υπερσυνδέσμων (hyperlinks)**.

Μια εικοσαετία μετά την επίσημη έναρξη του Web, από τον εφευρέτη του, Tim Berners-Lee, ο αριθμός των ιστοσελίδων παγκοσμίως υπολογίζεται σε εκατοντάδες εκατομμύρια, κανείς όμως δεν μπορεί να ξέρει τον ακριβή αριθμό, καθώς κάθε μέρα προστίθενται εκατοντάδες καινούριες σελίδες, ενώ άλλες εγκαταλείπονται.



Εικόνα 2: Εκτίμηση αριθμού ιστοσελίδων

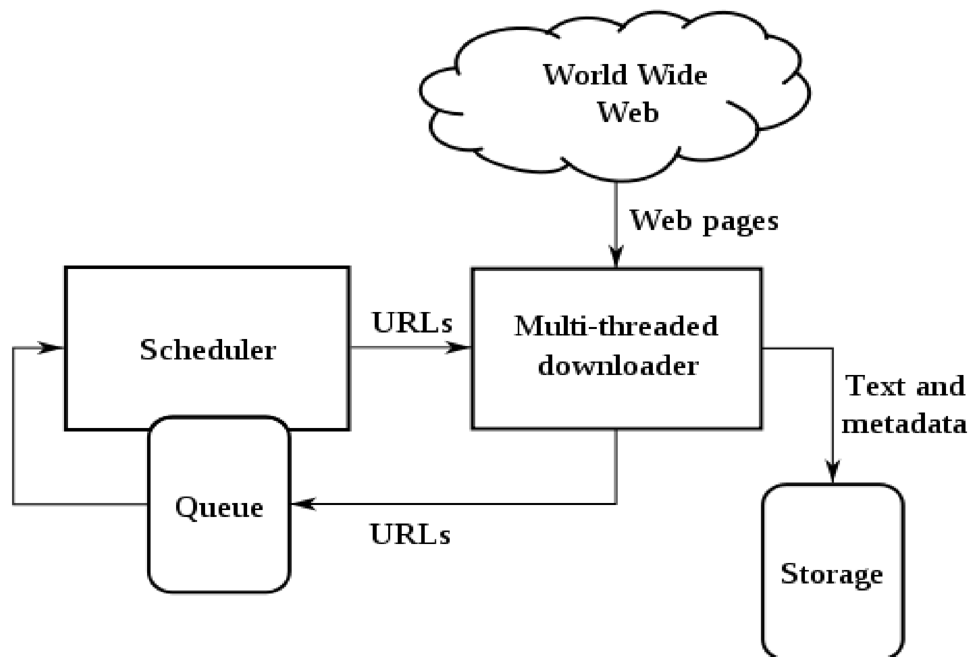
Είναι μια πρόκληση να βρεις τις ιστοσελίδες που σε ενδιαφέρουν σε αυτό το τεράστιο όγκο δεδομένων. Σε αυτό το σημείο, έρχονται οι **μηχανές αναζήτησης ιστού (web search engines)** να αντιμετωπίσουν αυτή την πρόκληση και να προσπαθήσουν να απαντήσουν σε εκατομμύρια **ερώτημα-αιτήματα (queries)** κάθε μέρα.

1.2 Μηχανές αναζήτησης

1.2.1 Δομή - Περιγραφή

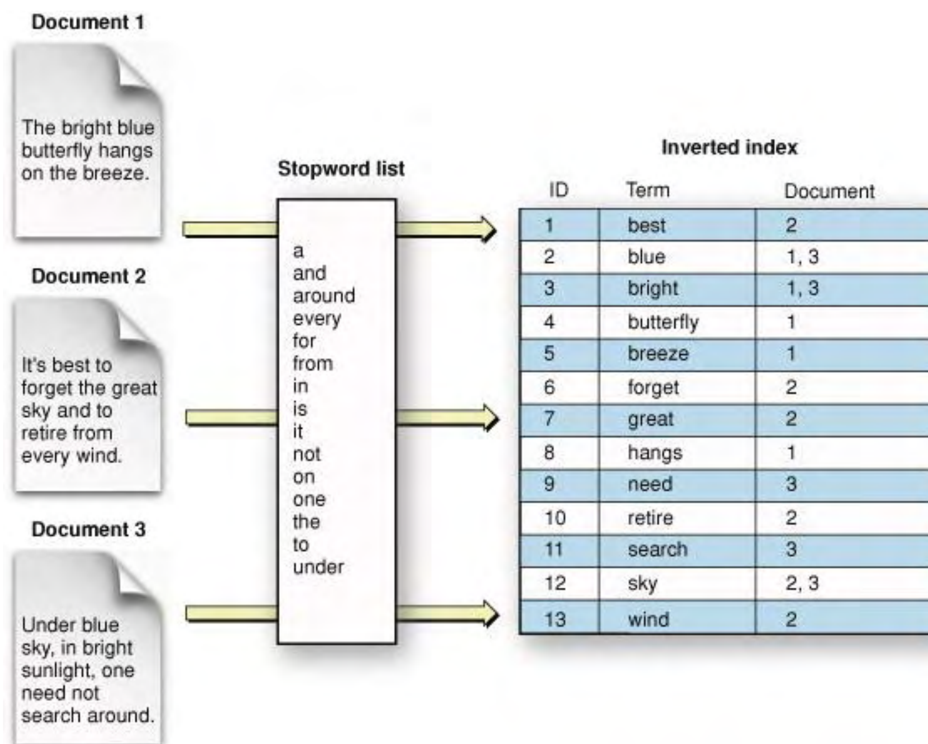
Οι μηχανές αναζήτησης είναι προγράμματα που επιτρέπουν την αναζήτηση με **λέξεις-κλειδιά** (***keywords***) σε τεράστιες βάσεις δεδομένων. Αυτές οι βάσεις δεδομένων περιέχουν αντίγραφα εκατομμυρίων ιστοσελίδων του World Wide Web που συλλέγονται αυτόματα από ειδικά προγράμματα «αράχνες», τα οποία έχουν διάφορες ονομασίες (spider, crawler, robot), αλλά εκτελούν ουσιαστικά την ίδια εργασία. Έτσι λοιπόν, μία μηχανή αναζήτησης αποτελείται από τρία βασικά μέρη:

- ***Crawler (ή Spider ή Robot):*** ένα ειδικό αυτόματο πρόγραμμα (λογισμικό πράκτορας) που επισκέπτεται ιστοσελίδες, συλλέγει και αναλύει το περιεχόμενο τους, επιστρέφει τα αποτελέσματα, και έπειτα ακολουθεί τις τυχόν **υπερσυνδέσεις** (***hyperlinks***) των ιστοσελίδων αυτών προς άλλες ιστοσελίδες. Η παραπάνω διαδικασία γίνεται ανά τακτά χρονικά διαστήματα, ώστε να είναι διαθέσιμο πάντα το πιο πρόσφατο δυνατό περιεχόμενο του web. Πάνω στη συλλογή αποτελεσμάτων που επιστρέφει ο crawler χτίζεται **ευρετήριο (index).**



Εικόνα 3: Αρχιτεκτονική ενός Web crawler

- ***Ευρετήριο (Index):*** μία τεράστια βάση δεδομένων, που περιέχει διάφορες πληροφορίες για τις ιστοσελίδες που επισκέφτηκε και σύλλεξε ο crawler, γι' αυτό και στις μεγάλες εμπορικές μηχανές αναζήτησης, χρησιμοποιούνται χιλιάδες υπολογιστές για να αποθηκεύσουν και να επεξεργαστούν μια τέτοια συλλογή. Συγκεκριμένα, χρησιμοποιείται μια δομή **ανεστραμμένου ευρετηρίου (inverted index)**, η όποια έχει δειχθεί ότι είναι η καλύτερη λύση στα συστήματα ανάκτησης πληροφοριών (information retrieval).

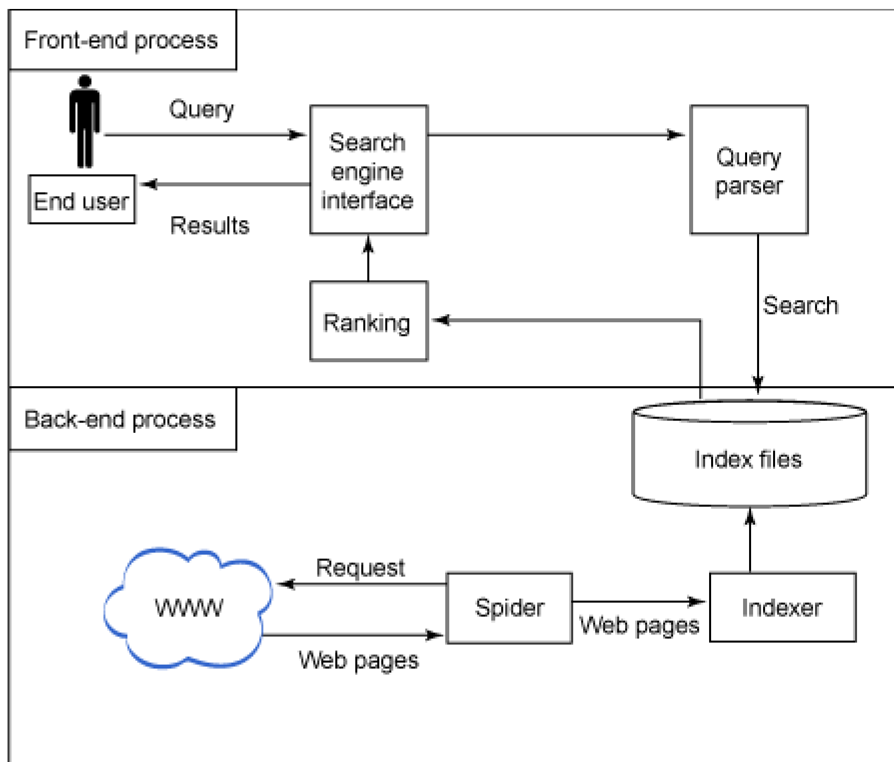


Εικόνα 4: Ανεστραμμένο ευρετήριο

- ***Τον μηχανισμό αναζήτησης:*** το πρόγραμμα που ψάχνει στο ανεστραμμένο ευρετήριο για να βρει ιστοσελίδες που ταιριάζουν στις **λέξεις-κλειδιά** του **ερωτήματος (query)** που έθεσε ο χρήστης. Οι μηχανές αναζήτησης χρησιμοποιούν ένα **αλγόριθμο αναζήτησης-κατάταξης (ranking)** για να εντοπίσουν γρήγορα τη ζητούμενη πληροφορία. Ο αλγόριθμος υπολογίζει το βαθμό (σκορ) για τα έγγραφα που σχετίζονται

περισσότερο με το ερώτημα και αυτά με τα υψηλότερα σκορ, είναι τα αποτελέσματα (results) που επιστρέφονται στο χρήστη.

Τέλος, οι μηχανές αναζήτησης έχουν δικές τους ιστοσελίδες στο διαδίκτυο, ουσιαστικά αποτελούν τη **διεπαφή (user interface)** για τους χρήστες. Κύρια πηγή εσόδων τους είναι οι διαφημίσεις και έτσι συνήθως προσφέρουν τις υπηρεσίες τους στους χρήστες δωρεάν. Σε ειδικά **πεδία (search forms)** ο χρήστης μπορεί να πληκτρολογεί τις λέξεις-κλειδιά προς αναζήτηση και η μηχανή αναζήτησης εμφανίζει μια σελίδα με τα καλύτερα αποτελέσματα, τα οποία είναι συγκεκριμένα στον αριθμό, πχ. τα 10 πρώτα. Ένα αποτέλεσμα συνήθως είναι ο τίτλος της ιστοσελίδας, μία υπερσύνδεση (URL) που οδηγεί σε αυτήν, και ένα μικρό απόσπασμα του κειμένου της ιστοσελίδας.



Εικόνα 5: Αρχιτεκτονική μια μηχανής αναζήτησης

1.2.1 Στόχοι μιας μηχανής αναζήτησης

Αν και όλα τα μέρη της μηχανής αναζήτησης είναι σημαντικά και απαραίτητα για την λειτουργίας, αυτό που παίζει σημαντικό ρόλο στην επιτυχία της είναι ο μηχανισμός αναζήτησης, καθώς από αυτόν εξαρτάται η αποτελεσματικότητα και αποδοτικότητα της.

Η μηχανή αναζήτησης πρέπει να είναι *αποτελεσματική* έτσι ώστε τα επιστρεφόμενα αποτελέσματα να είναι σχετικά με τα ερωτήματα των χρηστών και *αποδοτική* προκειμένου να ανταποκριθεί σε κάθε χρήστη μέσα σε δευτερόλεπτα, σχεδόν αμέσως.

Για το πρώτο, κάθε μηχανή αναζήτησης χρησιμοποιεί κάποιον αλγόριθμο κατάταξης (ranking algorithm) που υπολογίζει τα πιο σημαντικά και σχετικά έγγραφα σύμφωνα με την αναζήτηση του χρήστη. Όσο καλύτερος αλγόριθμος, τόσο καλύτερα αποτελέσματα επιστρέφονται στους χρήστες.

Σε αυτήν την εργασία θα ασχοληθούμε με την αποδοτικότητα των μηχανών αναζήτησης και πως αυτή εξαρτάται από το **caching (προσωρινή αποθήκευση)**.

Κεφάλαιο 2ο – Caching

Η **προσωρινή αποθήκευση (caching)** έχει μελετηθεί εκτενώς και εφαρμόζεται σε διαφορετικές περιοχές της επιστήμης των υπολογιστών με επιτυχία εδώ και πολλά χρόνια. Η θεμελιώδης αρχή του caching είναι να αποθηκεύονται τα στοιχεία που θα ζητηθούν στο εγγύς μέλλον.

Χρησιμοποιείται σε λειτουργικά συστήματα, σε βάσεις δεδομένων, σε εξυπηρετητές Web, έτσι και στις μηχανές αναζήτησης του Παγκόσμιου Ιστού, συναντάει κανείς caches σε διάφορα μέρη της, όπως caches αποτελεσμάτων (results), caches όρων ευρετηρίου (posting lists) και caches εγγράφων (document).

Εμείς εδώ μελετάμε **caches αποτελεσμάτων**, όπου ένα **entry (στοιχείο)** στη cache είναι το ζευγάρι (query, results).

2.1 Μηχανισμός

Σε υψηλό επίπεδο, μια μηχανή αναζήτησης που λαμβάνει ένα ερώτημα από ένα χρήστη, επεξεργάζεται το ερώτημα στο ευρετήριο των εγγράφων του, και επιστρέφει ένα μικρό σύνολο των σχετικών αποτελεσμάτων στο χρήστη. Αν ένα σύνολο των αποτελεσμάτων, που έχουν υπολογιστεί προηγουμένως, είναι cached, δηλαδή έχουμε **cache hit**[1], το ερώτημα μπορεί να εξυπηρετηθεί άμεσα από την cache, εξαλείφοντας την ανάγκη για την επεξεργασία του ερωτήματος. Αυτό έχει δύο πλεονεκτήματα:

2.1.1 Μείωση χρόνου απόκρισης

Μία πρόκληση για τις μεγάλες μηχανές αναζήτησης είναι η διεκπεραίωση του ερωτήματος σε πολύ σύντομο χρονικό διάστημα, σχεδόν αμέσως. Για το σκοπό αυτό, οι μηχανές αναζήτησης χρησιμοποιούν έναν αριθμό βασικών μηχανισμών για την αντιμετώπιση αυτών των προκλήσεων. Η προσωρινή αποθήκευση (caching) αποτελεσμάτων είναι μια από αυτές τις μεθόδους για να αντιμετωπιστούν αυτές οι υψηλές απαιτήσεις.

1.έχουμε cache hit όταν το στοιχείο που ζητάμε βρίσκεται στην cache, διαφορετικά έχουμε *cache miss*

2.1.2 Μείωση φόρτου εργασίας στο search cluster

Καθώς η επεξεργασία ερωτημάτων (queries) σε ένα μεγάλο ανεστραμμένο ευρετήριο που είναι χωρισμένο σε χιλιάδες υπολογιστές **(search cluster)** είναι μια σύνθετη λειτουργία, το caching συμβάλει σημαντικά στην αποφυγή της. Στο πρώτο στάδιο της επεξεργασίας, ένα ερώτημα διαβιβάζεται στους **εξυπηρετητές ευρετηρίου (index servers)** που περιέχουν ένα μέρος του ανεστραμμένου ευρετηρίου, δεδομένου ότι δεν είναι δυνατόν να φιλοξενηθεί το πλήρες ευρετήριο σε έναν server. Κάθε index server πρέπει να αποκτήσει πρόσβαση στο δίσκο για να προσκομίσει τα σχετικά τμήματα του ευρετηρίου για τους όρους του ερωτήματος. Ο αλγόριθμος κατάταξης (ranking) εκτελείται, προκειμένου να βρει τα πιο σχετικά έγγραφα με το ερώτημα. Τέλος, τα αποτελέσματα από κάθε server συγκεντρώνονται και η σελίδα αποτελεσμάτων είναι έτοιμη.

2.2 Γιατί caching;

Μία πολύ γνωστή παρατήρηση από το πεδίο έρευνας της ανάκτησης πληροφοριών (information retrieval) είναι ότι οι συχνότητες εμφάνισης των ερωτημάτων χαρακτηρίζονται από μία **power-law κατανομή (distribution)**. Αυτό σημαίνει ότι λίγα ερωτήματα έχουν πολύ υψηλές συχνότητες και πολλά εμφανίζονται σπάνια, ορισμένα μόνο μία φορά (singleton queries).

Επίσης έχει παρατηρηθεί ότι τα ερωτήματα έχουν την ιδιότητα της **χρονικής τοπικότητας (temporal locality)**, που σημαίνει ότι, ένα ποσό των ερωτήσεων που έχουν υποβληθεί προηγουμένως από τους χρήστες, υποβάλλονται και πάλι στο εγγύς μέλλον.

Τα παραπάνω δείχνουν γιατί το caching για τα αποτελέσματα των queries είναι σημαντικό. Ως εκ τούτου, τα αποτελέσματα του ερωτήματος μπορούν να αποθηκευτούν σε έναν αποκλειστικό χώρο προσωρινής αποθήκευσης και τα ερωτήματα μπορούν να απαντηθούν από τη μνήμη cache, χωρίς να απαιτούν περαιτέρω επεξεργασία.

2.3 Πολιτικές αντικατάστασης στην cache

Οι caches, γενικότερα, χαρακτηρίζονται από τη **χωρητικότητα** (**capacity**) τους και τις **πολιτικές αντικατάστασης ή εκδίωξης** (**replacement or eviction policies**) που χρησιμοποιούν για να διώξουν entries όταν γεμίσει η cache.

2.3.1 LRU και LFU

Οι δύο πιο κοινές πολιτικές αντικατάστασης είναι η **LRU** που διώχνει τα λιγότερα πρόσφατα χρησιμοποιούμενα (Least Recently Used), και η **LFU** που διώχνει τα λιγότερα συχνά χρησιμοποιούμενα (Least Frequently Used) στοιχεία από τη μνήμη cache. Ενώ υπάρχουν και αρκετές τροποποιήσεις των παραπάνω πολιτικών, καθώς και άλλες που συνδυάζουν τα πλεονεκτήματα των δύο πολιτικών.

Όπως είπαμε προηγουμένως τα ερωτήματα που φτάνουν στην μηχανή αναζήτησης εμφανίζουν την ιδιότητα της χρονικής τοπικότητας, δηλαδή ένα query, και τα αντίστοιχα αποτελέσματα, που έχουν γίνει cached, έχουν μεγάλη πιθανότητα να ζητηθούν στο κοντινό μέλλον, με την πιθανότητα αυτή να μειώνεται εκθετικά καθώς καταφθάνουν νέα queries. Οπότε η πολιτική LRU εκμεταλλεύεται πλήρως το γεγονός αυτό, διώχνοντας το query που ζητήθηκε παλαιότερα, του όποιου η πιθανότητα να ξαναεμφανιστεί, σύμφωνα με τα παραπάνω, είναι πλέον πολύ μικρή.

2.3.2 FIFO και Random

Στην εργασία μας πέρα από τις παραπάνω δύο, δοκιμάζουμε επίσης την πολιτική **FIFO** (**First In First Out**), η οποία διώχνει το πρώτο entry που μπήκε στη cache, και μία πολιτική **Random** που διώχνει ένα τυχαίο entry της cache. Μπορεί και οι δύο αυτές πολιτικές να είναι πολύ πιο απλές στην υλοποίηση τους από τις LRU και LFU, αλλά είναι και λιγότερο αποδοτικές. Στην επόμενη παράγραφο με ένα μικρό παράδειγμα θα το δούμε αυτή τη διαφορά μεταξύ FIFO και LRU.

2.3.3 FIFO vs. LRU

Έστω ότι έχουμε 6 διαφορετικά queries, με αναγνωριστικά (id's) 1 έως 6. Αν η cache μας έχει μέγεθος 4 (χωράει 4 queries με τα αποτελέσματα τους) και η σειρά που έρχονται τα queries είναι: **1 2 3 4 6 3 2 5 6 3 1 6 3 4 2 5**, τότε η συμπεριφορά των δύο αλγορίθμων αντικατάστασης για την παραπάνω ακολουθία φαίνεται στην Εικόνα 6.

FIFO

queries	1	2	3	4	6	3	2	5	6	3	1	6	3	4	2	5
cache	1	1	1	1	6	6	6	6	6	6	6	6	6	4	4	4
		2	2	2	2	2	2	5	5	5	5	5	5	5	2	2
			3	3	3	3	3	3	3	3	1	1	1	1	1	5
				4	4	4	4	4	4	4	4	4	3	3	3	3

LRU

queries	1	2	3	4	6	3	2	5	6	3	1	6	3	4	2	5
cache	1	1	1	1	6	6	6	6	6	6	6	6	6	6	6	5
		2	2	2	2	2	2	2	2	2	1	1	1	1	2	2
			3	3	3	3	3	3	3	3	3	3	3	3	3	3
				4	4	4	4	5	5	5	5	5	5	4	4	4

Εικόνα 6: Πως εξελίσσονται τα περιεχόμενα της cache όταν τα queries έρθουν με τη σειρά στη πρώτη γραμμή του πίνακα

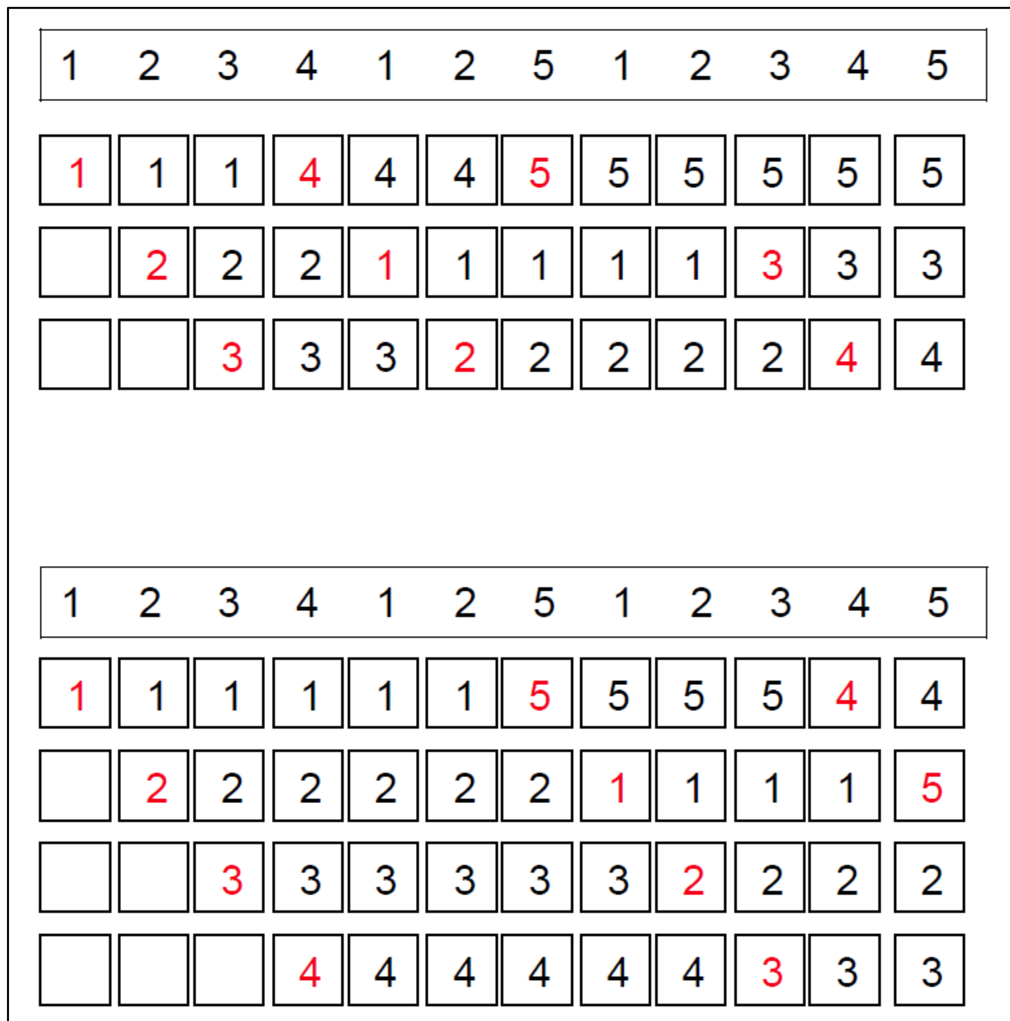
Στην παραπάνω εικόνα με πράσινο σημειώνουμε όποτε έχουμε cache hit. Παρατηρούμε ότι ο αλγόριθμος LRU σημειώνει ένα cache hit περισσότερο από τον FIFO.

2.3.4 Μέγεθος cache και πολιτική αντικατάστασης

Διαισθητικά, μπορεί να σχηματιστεί η εντύπωση ότι όσο μεγαλύτερη είναι μία cache, τόσο λιγότερα cache misses θα προκύψουν για μία δεδομένη ακολουθία από queries, και πράγματι αυτό συμβαίνει στις περισσότερες των περιπτώσεων, αλλά όμως όχι πάντα. Συγκεκριμένα, για την πολιτική FIFO, υπάρχει το εξής **παράδοξο του Belady (Bélády's anomaly),** που λέει, ότι είναι πιθανό να υπάρχουν περισσότερα misses καθώς αυξάνεται τον αριθμός των entries που μπορούμε να φιλοξενηθεί στην cache. Αυτό που συμβαίνει

μεγαλώνοντας το μέγεθος της cache, είναι ότι αλλάζει η σειρά με την οποία εκδιώχνονται τα entries χρησιμοποιώντας πολιτική FIFO, και σε ορισμένες περιπτώσεις αυτό συμβάλει στην αύξηση των misses.

Στην Εικόνα 7 βλέπουμε πως ισχύει το παράδοξο του Belady για δύο caches, μία τριών και μία τεσσάρων θέσεων, για 5 διαφορετικά queries, που έρχονται με της εξής σειρά: **1 2 3 4 1 2 5 1 2 3 4 5**.



Εικόνα 7: Πως εξελίσσονται τα περιεχόμενα των δύο caches.
Με κόκκινο τα cache misses.

Βλέπουμε λοιπόν, πως η 3-θέσια cache είχε 1 λιγότερο miss (9 αντί 10) από την 4-θέσια!

2.4 Freshness

2.4.1 Πολύ μεγάλες caches

Σε σύγκριση με το σύνηθες πρόβλημα της προσωρινής αποθήκευσης σε λειτουργικά συστήματα, το πρόβλημα της προσωρινής αποθήκευσης σε μηχανές αναζήτησης δεν είναι ο χώρος. Οι μηχανές αναζήτησης, είναι δυνατόν να αποθηκεύσουν εκατομμύρια καταχωρήσεις αποτελεσμάτων χρησιμοποιώντας και σκληρούς δίσκους πέρα από τη μνήμη RAM και πάλι να βελτιώσουν τη καθυστέρηση απάντησης του ερωτήματος. Κατά μέσο όρο, χρειάζονται δεκάδες χιλιοστά του δευτερολέπτου για την επεξεργασία ενός ερωτήματος στο search cluster, ενώ να τα φέρει από το δίσκο παρουσιάζει μια ανάλογη καθυστέρηση, συχνά μικρότερη. Επιπλέον, χρησιμοποιώντας δίσκους για την αποθήκευση προηγουμένως υπολογιζόμενων αποτελεσμάτων παρέχει μία ευκαιρία για να εξαλειφθούν τα misses σε μικρές results caches που χρησιμοποιούν μόνο μνήμη RAM.

2.4.2 Πρόβλημα: stale entries

Όμως ένα σημαντικό μειονέκτημα των μεγάλων caches αποτελεσμάτων είναι η **freshness** (‘φρεσκάδα’). Πιο συγκεκριμένα, επειδή ο inverted index των μηχανών αναζήτησης αλλάζει συχνά λόγω των νέων παρτίδων από έγγραφα που φέρνει ο crawler, είναι πιθανό να υπάρχει αναντιστοιχία μεταξύ του cached εγγράφου και του ανανεωμένου (updated) εγγράφου στο index. Ως εκ τούτου, ένα σημαντικό τμήμα των προηγουμένως υπολογιζόμενων αποτελεσμάτων στη μνήμη cache μπορεί να γίνει “μπαγιάτικο” **(stale)** με την πάροδο του χρόνου, με αποτέλεσμα να υπάρχει υποβάθμιση της ποιότητας των αποτελεσμάτων που επιστρέφονται στους χρήστες. Στην πραγματικότητα, το πρόβλημα της freshness γίνεται πιο σοβαρό, καθώς η χωρητικότητα της cache αυξάνεται.

Εκτός της παραπάνω μορφή του (αναντιστοιχία μεταξύ cache και index), το πρόβλημα της freshness στις μηχανές αναζήτησης εμφανίζεται και μεταξύ index και του πραγματικού εγγράφου (ιστοσελίδας), εξ αιτίας καθυστερημένης ανακάλυψης του από τον crawler. Εδώ δε μας ενδιαφέρει αυτή η περίπτωση.

2.4.3 Λύση: cache invalidation

Μια λύση στο πρόβλημα της freshness είναι να ακυρώνονται καταχωρήσεις με τη πάροδο του χρόνου. Υπάρχουν δύο πιθανές προσεγγίσεις για την **cache invalidation (ακύρωση)** στο πλαίσιο αυτό: *συνδεδεμένη (coupled)* και *αποσυνδεδεμένη (decoupled).* Η συνδεδεμένη προσέγγιση παρέχει στη cache πληροφορίες σχετικά με τις αλλαγές στο index. Αυτή η προσέγγιση είναι δύσκολη να υλοποιηθεί στην πράξη λόγω της πολυπλοκότητας και του υπολογιστικού κόστους, για τον ακριβή προσδιορισμό αλλαγών στο index και διάδοση τους στη cache αποτελεσμάτων. Η εν λόγω συνδεδεμένη προσέγγιση είναι έξω από το πεδίο εφαρμογής της παρούσας εργασίας. Η αποσυνδεδεμένη προσέγγιση, την οποία έχουμε υιοθετήσει στο έργο αυτό, ακυρώνει προσωρινά αποθηκευμένες καταχωρήσεις, χωρίς συγκεκριμένη γνώση για το αν έχει γίνει αλλαγή στο δείκτη (index). Ένας απλός τρόπος για να επιτευχθεί αυτός ο στόχος είναι να χρησιμοποιηθεί μία **time-to-live τιμή (TTL)** και να καταγράφονται οι καταχωρήσεις ως **ληγμένες (expired)** εφόσον υπάρχουν στη cache για μεγαλύτερο χρονικό διάστημα από το TTL. Μόλις οι καταχωρήσεις είναι άκυρες, αντιμετωπίζονται πλέον σαν caches misses, το οποίο σημαίνει ο μηχανισμός με το TTL λύνει το πρόβλημα της freshness αλλά μειώνεται η απόδοση της cache αφού πέφτει το **hit ratio**[1].

2.5 Refreshing

2.5.1 Κεντρική ιδέα

Μια σημαντική βελτίωση στο μηχανισμό του TTL, είναι η χρήση ενός μηχανισμού **refreshing,** που ανανεώνει τα expired entries, με το να τα επεξεργάζεται το αποθηκευμένο query εκ νέου στο search cluster.

Έχοντας προηγούμενη γνώση της ημερήσιας ή εβδομαδιαίας κίνησης των queries, υπάρχουν συνήθως περίοδοι χαμηλής δραστηριότητας (μειώνεται ο ρυθμός με τον οποίο έρχονται queries), μπορούμε να εκμεταλλευτούμε αυτά τα διαστήματα για να χρησιμοποιήσουμε το search cluster για να κάνουμε το refresh.

1. hit ratio: ο λόγος του αριθμού των queries που εξυπηρετούνται από την cache προς συνολικό αριθμό queries

Οπότε η προσθήκη ενός μηχανισμού για να ανανεώνει την cache, πάνω στο μηχανισμό του TTL, έχει το πλεονέκτημα της αύξησης του hit ratio ως αποτέλεσμα ελαχιστοποίησης των cache misses που οφείλονται σε expired entries.

2.5.2 Πολιτική ανανέωσης

Ένας μηχανισμός ανανέωσης απαιτεί μια πολιτική για να επιλέξετε τις καταχωρήσεις προς ανανέωση. Ιδανικά, κρατάμε ένα σύνολο από entries fresh, έτσι ώστε το hit ratio να μεγιστοποιείται και η μέση ηλικία των entries να ελαχιστοποιείται.

Υπάρχουν διάφορα πιθανά κριτήρια για την επιλογή των καταχωρήσεων, για ανανέωση, όπως η συχνότητα του query, το πόσο πρόσφατο είναι, το κόστος της επεξεργασίας του στο search cluster κτλ.

Στην εργασία αυτή, χρησιμοποιούμε το πόσο πρόσφατο είναι ένα entry σαν κριτήριο επιλογής, ενώ εφαρμόζουμε το refreshing περιοδικά.

2.6 Flushing

Μία απλοϊκή λύση στο πρόβλημα του freshness είναι να καθαρίζεται περιοδικά το περιεχόμενο της μνήμης cache και να γεμίζει εκ νέου με νέα ερωτήματα **(flushing)**. Η προσέγγιση αυτή εγγυάται ότι όλα τα «μπαγιάτικα» αποτελέσματα θα απορριφθούν, αλλά αυτό επιβάλλει την επεξεργασία κάθε μελλοντικού query στο cluster.

Αν και το flushing λύνει ως ένα βαθμό το freshness, μπορεί να οδηγήσει σε σημαντική πτώση των ποσοστών επιτυχίας εξαιτίας των πολλών υποχρεωτικών αστοχιών.

Σε γενικές γραμμές το ποσοστό επιτυχίας φαίνεται να ανακτάται στο χρονικό διάστημα ανάμεσα σε 2 flushes. Ωστόσο, η απότομη πτώση hit ratio μετά από κάθε flush δεν είναι κάτι αποδεκτό σε μια μηχανή αναζήτησης που εξυπηρετεί χιλιάδες queries το δευτερόλεπτο.

Μια τέτοια ξαφνική και απότομη πτώση του ποσοστού επιτυχίας μπορεί να οδηγήσει σε υψηλό query traffic στο search cluster, σε τέτοιο βαθμό που ίσως ξεπεράσει την μέγιστη επεξεργαστική δυνατότητα του συστήματος. Μια τέτοια κατάσταση οδηγεί σε μείωση της ποιότητας των αποτελεσμάτων ή και κάποια queries απορρίπτονται εντελώς από επεξεργασία.

Κεφάλαιο 3ο - Υλοποίηση

Σε αυτό το κεφάλαιο, θα αναφέρουμε διάφορα θέματα, που αφορούν την υλοποίηση και το σχεδιασμό της μηχανής αναζήτησης, όπου θα διεξάγουμε τα πειράματα μας.

3.1 Lucene

Το λογισμικό που χρησιμοποιήσαμε για να στήσουμε μια μηχανή αναζήτησης για διεξάγουμε τα πειράματα μας είναι το **Lucene**, που είναι δωρεάν διαθέσιμο από την Apache Software Foundation.

Το Apache Lucene είναι μια βιβλιοθήκη (library) δωρεάν λογισμικού (open source), γραμμένη σε Java, που παρέχει δυνατότητες μιας μηχανής αναζήτησης υψηλής απόδοσης, με πλήρης υποστήριξη δημιουργίας ευρετηρίου (indexing) και αναζήτησης κειμένου (searching). Δε περιλαμβάνει μηχανισμό crawling, ωστόσο υπάρχουν αρκετά projects (πχ. Apache Nutch) που κάνουν αυτή τη δουλειά, και συμπληρώνουν το Lucene.

Τα βασικά στοιχεία στο Lucene είναι: το ευρετήριο (index), το έγγραφο (document), το πεδίο (field) και οι όροι (term).

- Το ευρετήριο περιέχει έγγραφα, πχ. Μία ιστοσελίδα είναι ένα έγγραφο.
- Ένα έγγραφο περιέχει πεδία, πχ. Πεδία σε μια ιστοσελίδα είναι ο τίτλος της, το κείμενο που περιέχει, η διεύθυνση URL κτλ.
- Ένα πεδίο περιέχει όρους, πχ. Οι λέξεις του κειμένου της ιστοσελίδας είναι όροι.

Στην επόμενη σελίδα, στην Εικόνα 8, δείχνουμε μια εκδοχή, σε κώδικα Java, για το πως δημιουργούμε το index στο Lucene:

```java
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;


public class Indexer {

    public static void main(String[] args) throws IOException {


        IndexWriter indexWriter = new IndexWriter(FSDirectory.open(new File("index")),
                                                  new StandardAnalyzer(Version.LUCENE_30),
                                                  true,
                                                  IndexWriter.MaxFieldLength.LIMITED);

        Scanner scan = new Scanner(new File("files/ohsumed.dat"));

        String line, data[];
        Document doc;

        while (scan.hasNextLine()) {
            line = scan.nextLine().trim();
            data = line.split("\t");


            doc = new Document();
            doc.add(new Field("id", data[0], Field.Store.YES, Field.Index.NO));
            doc.add(new Field("content", data[1], Field.Store.NO, Field.Index.ANALYZED));

            indexWriter.addDocument(doc);
        }

        indexWriter.optimize();
        indexWriter.close();

        scan.close();
    }

}
```

Εικόνα 8: Κώδικας Java για τη δημιουργία του index στο Lucene

3.2 Συλλογή δεδομένων

Όπως είπαμε το Lucene δεν δίνει τη δυνατότητα crawling, οπότε πρέπει να του παρέχονται τα δεδομένα που μας ενδιαφέρουν για indexing και searching. Η συλλογή δεδομένων (dataset) που χρησιμοποιήσαμε για να χτίσουμε το index, είναι μια συλλογή εγγραφών, που ονομάζεται OHSUMED και χρησιμοποιήθηκε στο TREC-9 Filtering Track, και είναι διαθέσιμη online.

TREC (Text REtrieval Conference) είναι μια σειρά από ετήσιες συνδιασκέψεις που ασχολούνται με έρευνες πάνω στην ανάκτηση πληροφοριών (information retrieval) για διάφορους επιστημονικούς τομείς. Οι συλλογές δεδομένων που χρησιμοποιούν συνήθως δεν είναι διαθέσιμες δωρεάν, με εξαίρεση την παραπάνω.

Η συλλογή OHSUMED είναι ένα σύνολο 348.566 αναφορές από το MEDLINE, μια online βάση δεδομένων με βιο-ιατρικές πληροφορίες, συνολικού μεγέθους ~ 400MB κειμένου, που αποτελείται από τίτλους ή / και περιλήψεις από 270 ιατρικά περιοδικά κατά τη διάρκεια μιας πενταετίας περίοδο (1987-1991).

Από τα διάφορα πεδία των 348.566 εγγραφών της συλλογής, με ένα script σε python, κρατήσαμε μόνο τα πεδία, μοναδικό αναγνωριστικό (ID), τίτλο και περιγραφή της αναφοράς (σε ορισμένες αναφορές υπάρχει μόνο τίτλος), τα οποία χρησιμοποιήσαμε για να χτίσουμε το index.

Στην περίπτωση μας, κάθε εγγραφή από τη συλλογή OHSUMED είναι ένα document στο Lucene, με δύο fields, το ID και το κείμενο της εγγραφής (τίτλος και περιγραφή μαζί).

3.3 Query log

Μιας και δεν έχουμε πραγματικά queries για να κάνουμε τις δοκιμές μας, δημιουργήσαμε τυχαία συνθετικά queries από τους υπάρχοντες όρους (terms) του ευρετηρίου.

Για να φτιάξουμε ένα query, επιλέγουμε τυχαία το μήκος του, ομοιόμορφα (uniformly) μεταξύ ενός και τεσσάρων όρων, και έπειτα, επίσης ομοιόμορφα, επιλέγουμε τυχαίους όρους (1-4) από το ευρετήριο. (βλ. Εικόνα 9)

Ο παραπάνω τρόπος δημιουργίας των queries δεν είναι αρκετός, αφού για να δούμε την λειτουργία της cache χρειαζόμαστε queries που επαναλαμβάνονται. Έτσι λοιπόν το query log που δημιουργήσαμε, κάναμε τα εξής: συνθέσαμε αρχικά 2000 μοναδικά queries με τον τρόπο που περιγράφεται παραπάνω. Έπειτα από αυτά τα 2000 queries, επιλέξαμε κατά τέτοιο τρόπο αυτά που θα βάζαμε στο query log μας, υποθέτοντας ότι ακολουθούν μια power-low κατανομή, συγκεκριμένα την κατανομή **Zipf** (βλ. Εικόνα 10). Δηλαδή ορισμένα queries είναι πολύ δημοφιλή και εμφανίζονται συχνά, ενώ τα περισσότερα εμφανίζονται μόνο μία φορά. Τέλος, δίνουμε τυχαίους χρόνους άφιξης στα query, έτσι ώστε ο χρόνος μεταξύ των αφίξεων να είναι εκθετικά κατανεμημένος.

```java
public Query randomQuery(int maxQueryLength) {

    int queryLength;
    int rInt[];

    // epilegoume tuxaio mikos query
    queryLength = rGen.nextInt(maxQueryLength) + 1;

    rInt = new int[queryLength];

    // epilegoume tuxaia terms gia ti sin8esi tou query
    for(int i=0; i<queryLength; i++){
        rInt[i] = rGen.nextInt(terms.size());

        // elegxos gia monadikotita twn tuxaiwn ari8mwn
        for(int j=0; j<i; j++){
            if(rInt[j] == rInt[i]){
                i--;
                break;
            }
        }
    }

    // sun8etoume to query
    BooleanQuery query = new BooleanQuery();
    for(int j=0; j<queryLength; j++){
        query.add(new TermQuery(terms.get(rInt[j])), BooleanClause.Occur.SHOULD);
    }

    return query;
}
```

Εικόνα 9: Κώδικας Java για τη δημιουργία τυχαίου query από τα terms του index

```java
// arxika monadika queries
String uniqueQueries[] = qGen.getNStringQueries(2000);

// to teliko query log 8a exei 5000 queries
LinkedList<String> queryLog = new LinkedList<String>();

// ek8etis tis Zipf katanomis
double skew = 1.0;
ZipfGenerator zipf = new ZipfGenerator(numOfUniqueQueries, skew);

int queryId;
for(int i=0; i<5000; i++){

    // zipf.next() returns a rank id between 0 and numOfUniqueQueries
    // The frequency of returned rank id follows the Zipf distribution
    queryId = zipf.next();

    queryLog.add(uniqueQueries[queryId]);
}
```

Εικόνα 10: Ενδεικτικός κώδικας Java για τη δημιουργία του query log

3.4 Cache

Όπως αναφέρθηκε, το results caching είναι σημαντικός παράγοντας για την αποδοτικότητα της μηχανής αναζήτησης, καθώς οδηγεί σε μικρότερους χρόνους απόκρισης για τους χρήστες αλλά και σε λιγότερο φόρτο για το search cluster. Για να μετρήσουμε αυτήν την αποδοτικότητα, υπάρχουν ορισμένες μετρικές όπως: το **hit ratio (ή miss ratio),** ο χρόνος απόκρισης ή η διεκπεραιωτική ικανότητα (throughput). Παρακάτω θα μας χρησιμοποιήσουμε εναλλακτικά και το hit και το miss ratio. Ως γνωστόν, *hit ratio + miss ratio = 1*.

3.4.1 Χωρητικότητα

Για τις caches που δοκιμάζουμε, μετράμε τη **χωρητικότητα (capacity)** τους σε αριθμό entries (ζευγάρια query-results) που μπορεί να φιλοξενήσει, πχ. 100 έως 2000.

Όπως είναι αναμενόμενο, υπάρχει αναλογική σχέση μεταξύ cache capacity και hit ratio, με το ποσοστό να σταθεροποιείται αν έχουμε μια πολύ μεγάλη cache (άπειρη), που μπορεί να χωρέσει όλα τα μοναδικά queries (2000).

Κεφάλαιο 4^{ο} – Μετρήσεις

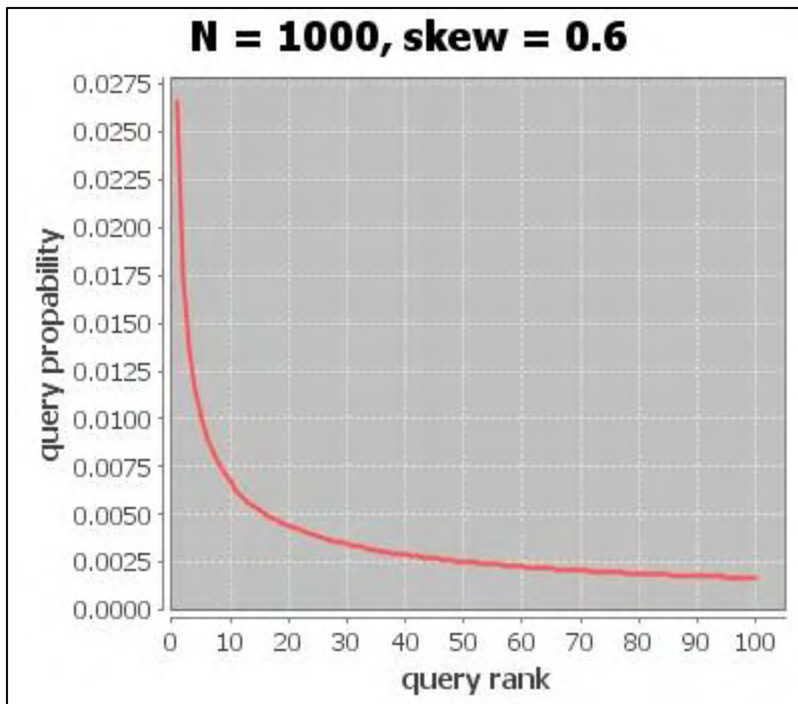
Σε αυτό το κεφάλαιο, παρουσιάζουμε, μέσω γραφημάτων, και σχολιάζουμε τις πειραματικές μετρήσεις που κάναμε.

4.1 Κατανομή Zipf

Αναφέραμε ότι χρησιμοποιήσαμε την **κατανομή Zipf** για να φτιάξουμε το query log, οπότε θα ήταν χρήσιμο να πούμε λίγα λόγια για αυτήν, που είναι μια από τις πιο δημοφιλείς κατανομές. Για πολλά είδη διαγωνισμών δημοτικότητας, όπως οι ενοικιαζόμενες ταινίες, τα βιβλία που δανειζόμαστε από μια βιβλιοθήκη, οι ιστοσελίδες του διαδικτύου, οι λέξεις της αγγλικής γλώσσας που χρησιμοποιούνται σε μια νουβέλα, και ο πληθυσμός των μεγαλύτερων πόλεων, μια λογική προσέγγιση της σχετικής δημοτικότητας ακολουθεί ένα περιέργως προβλέψιμο πρότυπο. Το πρότυπο αυτό ανακαλύφθηκε από τον καθηγητή γλωσσολογίας στο Χάρβαρντ Georg Zipf και ονομάζετε νόμος του Zipf. Ο νόμος αυτός ορίζει ότι, αν έχουμε N αντικείμενα τα οποία μπορούμε να τα διατάξουμε σύμφωνα με την δημοτικότητα τους, έτσι ώστε x1>x2>.....>xN, τότε στην γενική περίπτωση η πιθανότητα εμφάνισης του κ-πιο δημοφιλούς αντικειμένου ισούται με:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}.$$

Από την εξίσωση παρατηρούμε ότι όσο πιο μεγάλος είναι ο εκθέτης **s (skewness),** τόσο πιο «σκεβρωμένη» και εκθετική είναι η κατανομή, όπως φαίνεται και στην Εικόνα 11. Ενώ όσο περισσότερο πλησιάζει το 0, τόσο περισσότερο απλώνει και γίνεται πιο ομοιόμορφη (για s=0, ταυτίζεται με την ομοιόμορφη κατανομή).

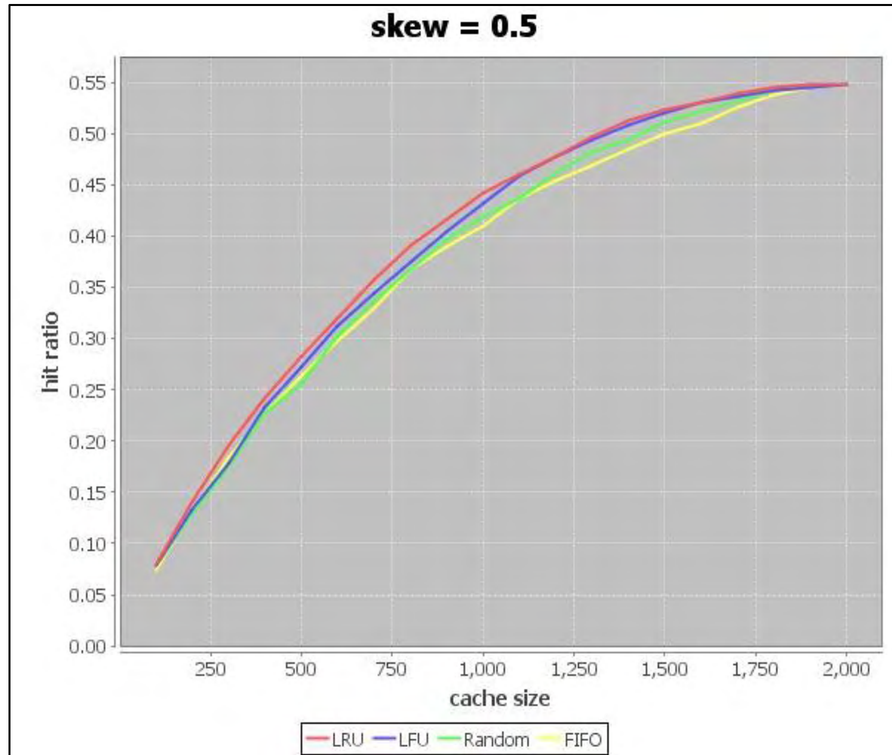


Εικόνα 11: Οι πιθανότητες των 100 δημοφιλέστερων στοιχείων

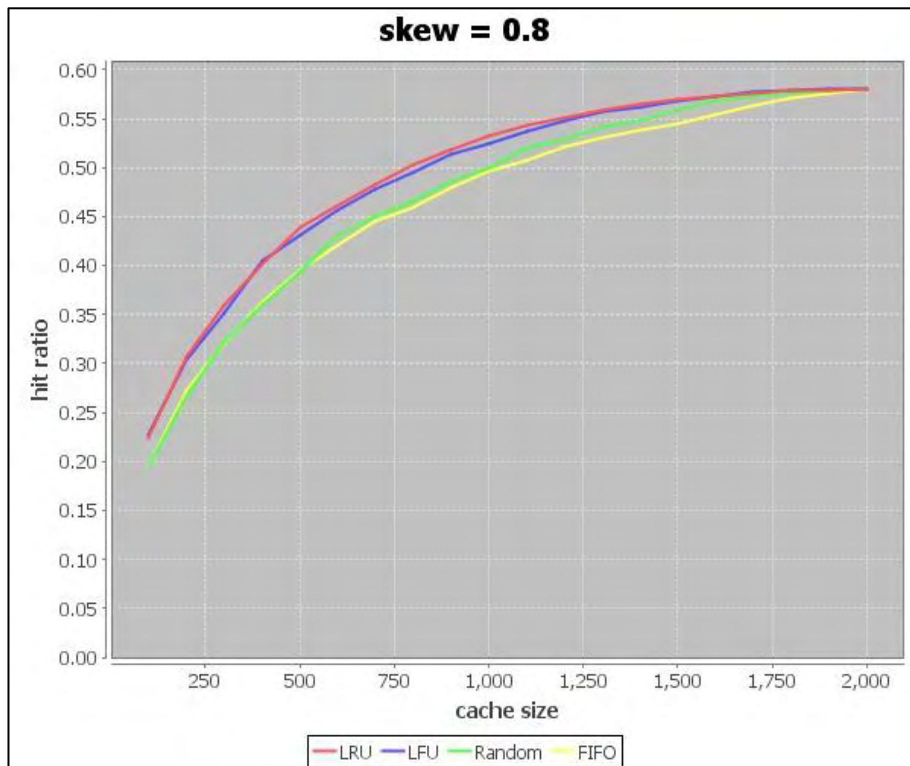
Επίσης παρατηρούμε ότι η κατανομή πέφτει αρκετά γρήγορα στην αρχή, ενώ εμφανίζει μεγάλη ουρά και μάλιστα όσο προχωράμε προς την ούρα, η διαφορά ποσοστού ανάμεσα σε διαδοχικά αντικείμενα γίνεται όλο και πιο μικρή.

4.1.1 Zipf και hit ratio

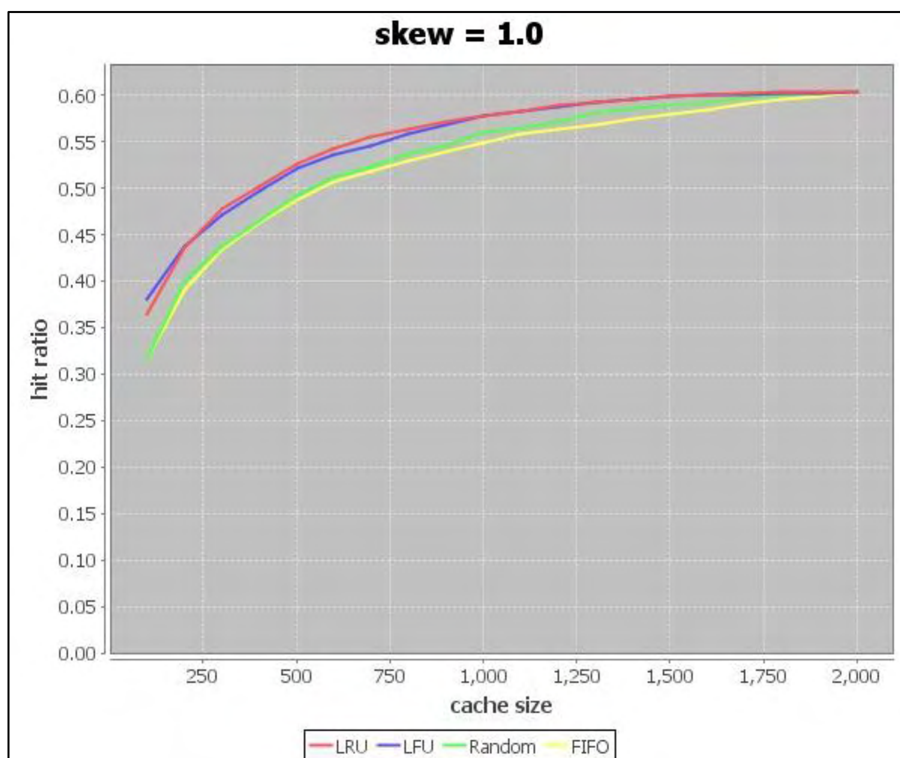
Στα παρακάτω γραφήματα βλέπουμε πως ο εκθέτης στη Zipf κατανομή, που χαρακτηρίζει το query log, επηρεάζει το hit ratio, για διάφορες πολιτικές εκδίωξης από την cache.



Εικόνα 12



Εικόνα 13



Εικόνα 14

Αυτό που παρατηρούμε είναι ότι όσο μεγαλύτερο είναι το s, δηλαδή όσο πιο σκεβρωμένη είναι ή όσο πιο μεγάλη είναι η ουρά, το hit ratio αυξάνεται. Αυτό οφείλεται στο ότι, για το query log, έχουμε να επιλέξουμε μεταξύ λιγότερων queries, μιας και τα περισσότερα έχουν πολύ μικρή πιθανότητα, λόγω της μεγάλης ουράς, και άρα δεν επιλέγονται ποτέ.

Επίσης, από τα παραπάνω γραφήματα, βλέπουμε ότι η LRU παρουσιάζει παρόμοια συμπεριφορά με την LFU, αλλά είναι συνολικά η καλύτερη πολιτική. Το ίδιο συμβαίνει και μεταξύ FIFO και Random πολιτικής, με τη FIFO να αποδεικνύεται η χειρότερη απ’ όλες. Επίσης, όσο μικρότερος ο εκθέτης s στην Zipf, τόσο πιο παρόμοια συμπεριφορά παρουσιάζουν και οι 4 πολιτικές.

Στα γραφήματα που θα ακολουθήσουν θα χρησιμοποιήσουμε το query log, με skew = 1.0, και μόνο τις πολιτικές LRU και FIFO.

4.2 TTL values

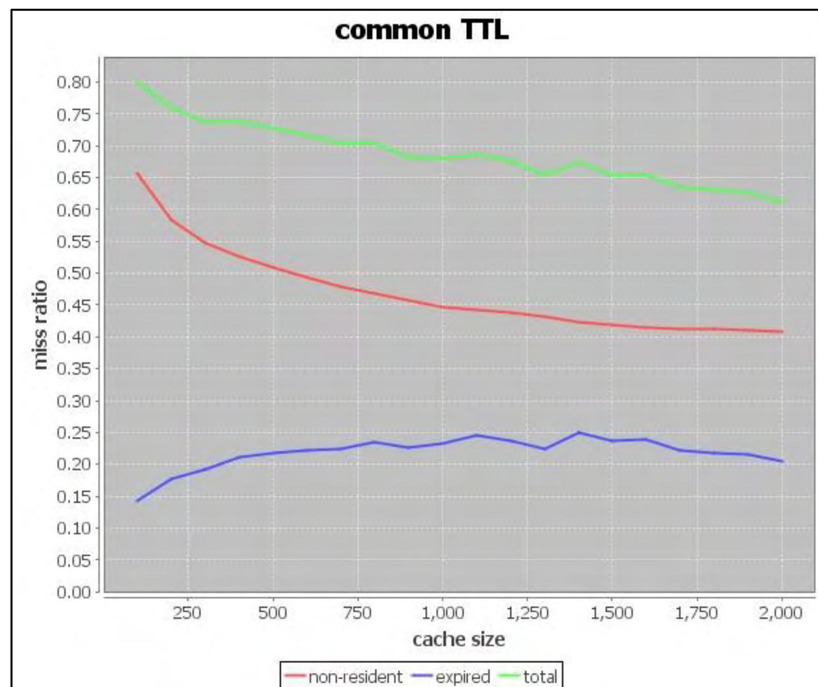
Όπως είπαμε χρησιμοποιούμε μια τιμή TTL για να θέσουμε άκυρα (expired) όσα entries βρίσκονται στη cache για περισσότερο από TTL. Στα γραφήματα που ακολουθούν αναθέσαμε τυχαίες τιμές για το TTL στα cache entries με 3 διαφορετικούς τρόπους:

1. Κοινό TTL για όλα τα entries. Ανάλογο του cache size και του μέσου χρόνου μεταξύ αφίξεων των queries. (Το κάνω αυτό για να μην έχω ένα πολύ μεγάλο TTL σε μια μικρή cache, θα ήταν μη λειτουργικό.)

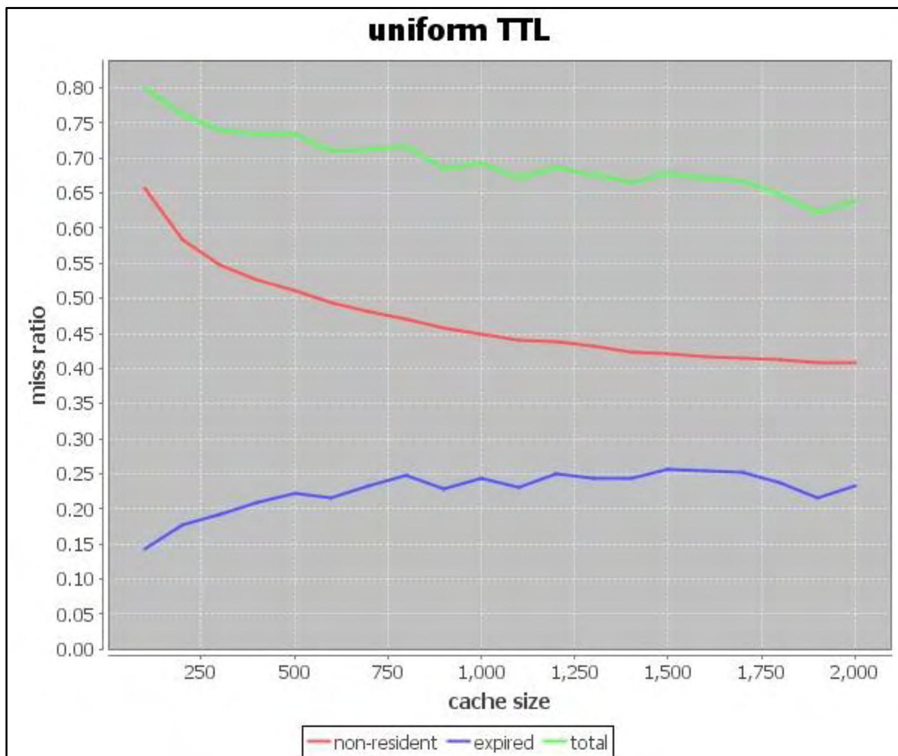
2. Διαφορετικό TTL για κάθε entry, ομοιόμορφα κατανεμημένο σε ένα εύρος τιμών, με μέση τιμή το παραπάνω κοινό TTL.

3. Διαφορετικό TTL για κάθε entry, κατανεμημένο κατά Zipf, με skew=0.7, μεταξύ N=10 τιμών TTL που έχουν επιλεχθεί ομοιόμορφα, με τον 2° τρόπο παραπάνω. (Αρχικοποιώ ένα πίνακα με 10 τυχαίες τιμές TTL με το 2° τρόπο, και μετά επιλέγω από αυτό το πίνακα με Zipf(0.7))

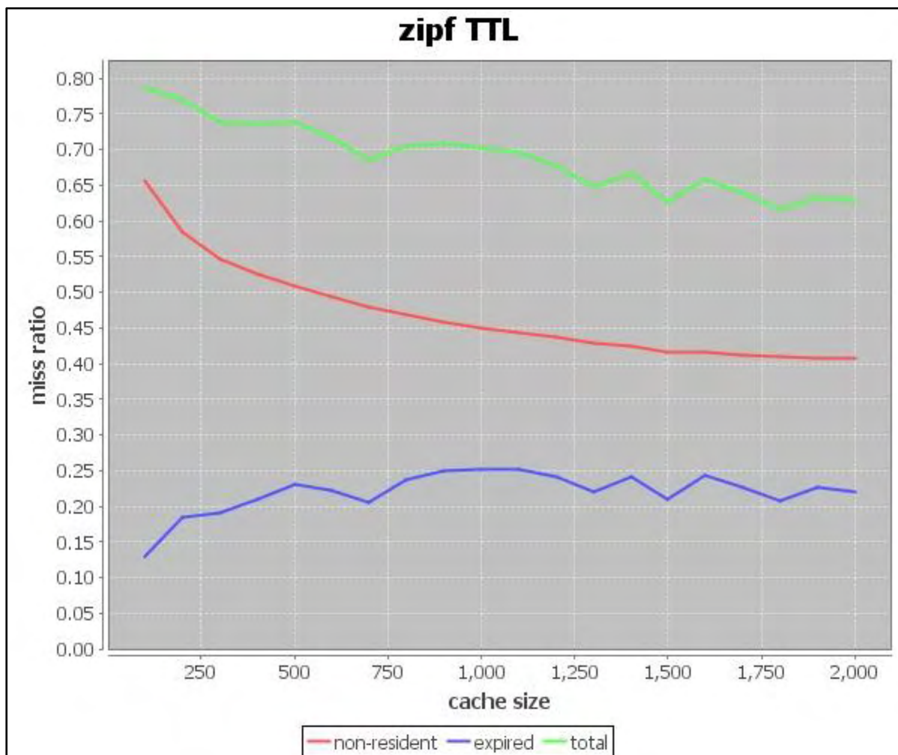
Ακολουθούν τα γραφήματα για κάθε μία από τις περιπτώσεις. Η πολιτική αντικατάστασης είναι η LRU.



Εικόνα 15



Εικόνα 16



Εικόνα 17

Τα παραπάνω γραφήματα δείχνουν το miss ratio αυτή τη φορά, που οφείλεται αποκλειστικά σε expired entries και που οφείλεται σε κανονικά cache misses (non-resident entries), όπως και το συνολικό.

Τα δύο πρώτα μοιάζουν πολύ γιατί όπως είπαμε η αναμενόμενη μέση τιμή TTL για τα cache entries είναι ίδια στις δύο περιπτώσεις. Στο τρίτο βλέπουμε μεγαλύτερες διακυμάνσεις διότι το TTL επιλέγεται μεταξύ λίγων τιμών, που μπορεί να έχουν μεγάλη απόκλιση μεταξύ τους

Παρατηρούμε ότι το miss ratio λόγω expired entries είναι περισσότερο από το ένα τρίτο του συνολικού miss ratio, και έστω με μικρό ρυθμό αυξάνεται καθώς μεγαλώνει το μέγεθος της cache. Όπως είχαμε τονίσει το πρόβλημα του freshness είναι σοβαρότερο για μεγαλύτερες caches, αφού είναι πιθανότερο ένα entry να μείνει για αυθαίρετα μεγάλο χρονικό διάστημα μέσα στη cache, και άρα περισσότερο χρόνο από TTL και να έχουμε cache miss.

4.3 Refreshing

Παρακάτω θα δούμε πως βελτιώνονται τα πράγματα εφαρμόζοντας το μηχανισμό του refreshing, ο οποίος όπως περιγράψαμε, συμβάλει στη βελτίωση του hit ratio, κρατώντας «φρέσκια» την cache.

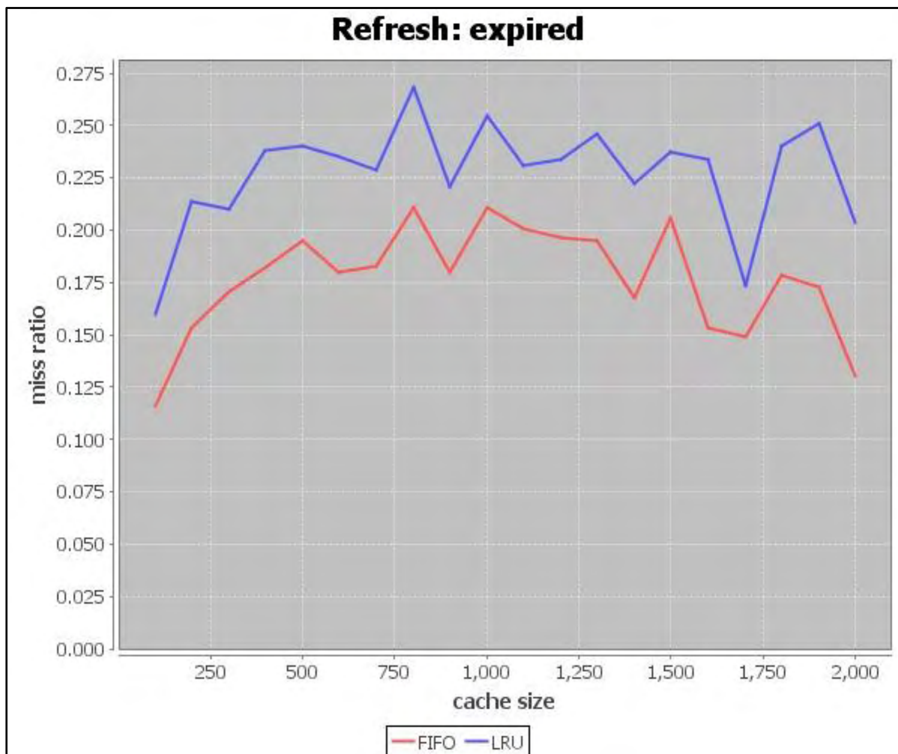
Στην υλοποίηση μας αυτό που λάβαμε υπόψη για να κάνουμε refresh ένα entry είναι η ηλικία του, δηλαδή πόσο χρόνο βρίσκεται στην cache. Να σημειώσουμε ότι εφαρμόζουμε τον κανόνα σε καθορισμένη στιγμή κατά τη διάρκεια των αφίξεων των queries, παρόλο που όπως τονίσαμε στο **2.5**, υπάρχει καλύτερη λύση (βλ. και **5.2**).

Συγκεκριμένα κάνουμε τον εξής έλεγχο για κάθε entry: αν η διαφορά της χρονικής στιγμής, που έγινε τελευταία φορά update[1], από την τρέχουσα χρονική στιγμή είναι μεγαλύτερη από ένα όριο **RT (refresh threshold)** (ενδεικτικές τιμές: TTL/2, TTL/4, TTL/8), τότε το κάνουμε refresh, δηλαδή ζητάμε από το index να μας απαντήσει εκ νέου το query του συγκεκριμένου entry, και ανανεώνουμε τα results. Σε ψευδοκώδικα:

```
if(curTime – timeUpdated(entry[i]) > RT){
    refresh(entry[i]);
}
```

1. update για ένα entry είναι η πρώτη φορά που θα μπει στην cache ή όταν γίνει refresh

Εικόνα 18: miss ratio due to expiration (uniform TTL)

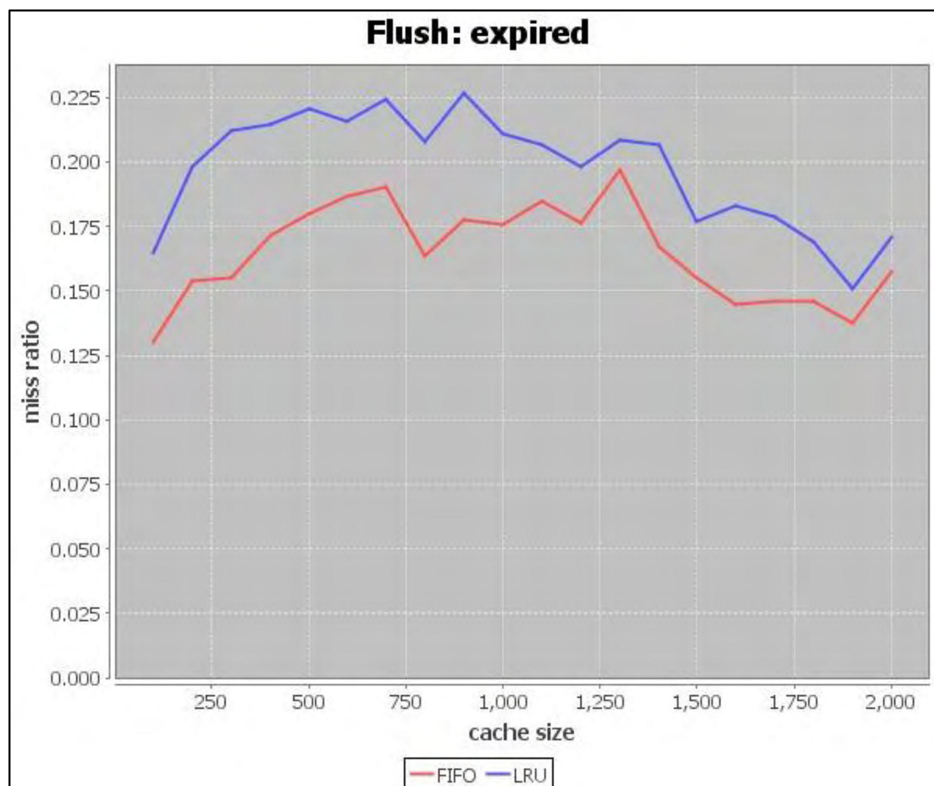


Εικόνα 19: refreshing miss ratio due to expiration (uniform TTL)

Οι εικόνες 18 και 19, δείχνουν τη διαφορά μεταξύ τη χρήση και μη του μηχανισμού refreshing, που εδώ χρησιμοποιείται στην πιο απλή μορφή του, δηλαδή μία και μόνο καθορισμένη στιγμή. Βλέπουμε, ότι για ορισμένα caches sizes ρίχνει το miss ratio και πάνω από 5%. Επίσης, όσο πιο μικρή είναι η cache (μέχρι 500 entries), τόσο μικρότερη επιρροή έχει.

4.3.1 Flushing

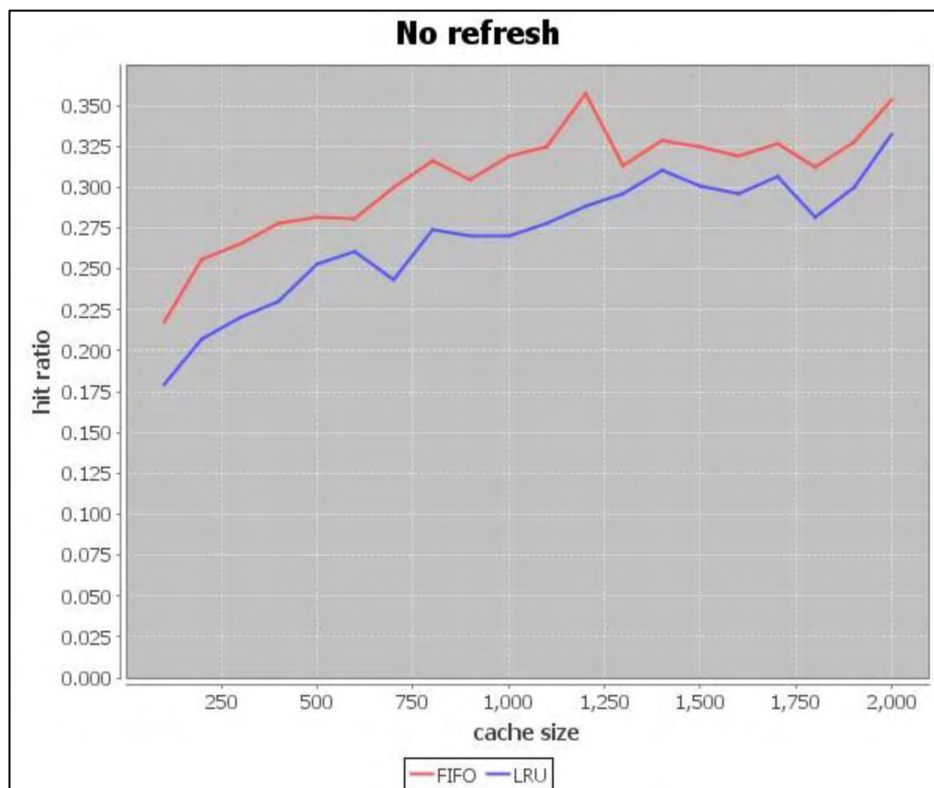
Τώρα πρώτα θα δούμε τη επίδραση έχει το flushing στο miss ratio εξ αιτίας λήξης, εφαρμόζοντας το, τις ίδιες καθορισμένες χρονικές στιγμές που κάναμε και το refreshing. Κι έπειτα, θα δούμε τα hit ratios για τις τρεις επιλογές: όχι refresh, refresh και flushing.



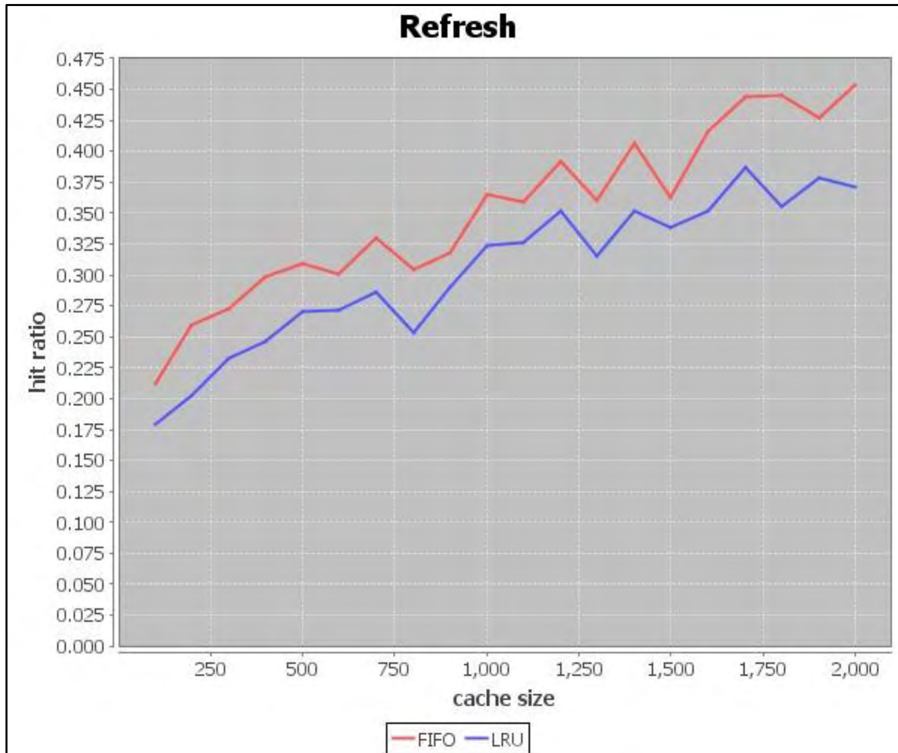
Εικόνα 20: flushing miss ratio due to expiration (uniform TTL)

Όσο και να δείχνει περίεργο, στο flushing το miss ratio είναι αρκετά μικρότερο από την περίπτωση που έχουμε απλά το μηχανισμό του TTL (όχι refresh), αλλά είναι μικρότερο, και απ’ όταν έχουμε refresh. Αυτό είναι λογικό διότι όταν γίνεται το flushing, και αδειάζει πλήρως η cache, όσα ήταν ήδη μέσα πολλά από αυτά δε πρόλαβαν να λήξουν (και άρα να έχουμε misses), ενώ το ίδιο ισχύει και γι αυτά που θα μπουν μιας και η εισροή queries στη μηχανή αναζήτησης σταματάει (μη ξεχνάμε ότι τρέχουμε 5000 queries)

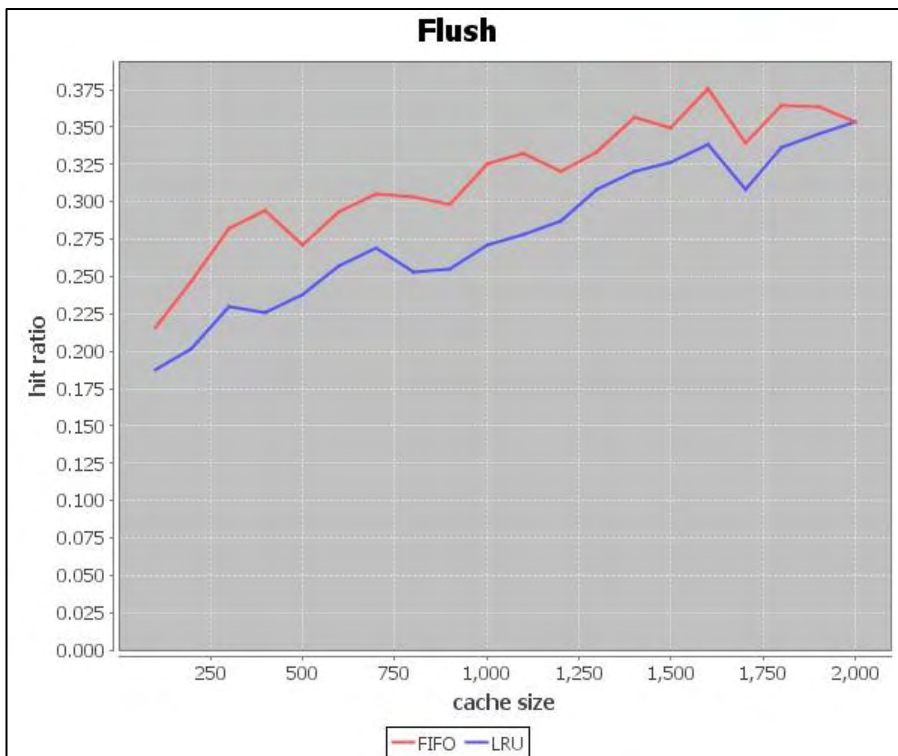
Τέλος, ακολουθούν τα γραφήματα με το hit ratio για: όχι refresh, refresh και flushing. Βλέπουμε όπου για μεγέθη cache μεγαλύτερα από 500, πόσο υπερτερεί ο μηχανισμός του refreshing, όπου τελικά, όπως περιμέναμε, σημειώνει 10% μεγαλύτερο hit ratio από τους άλλους δύο μηχανισμούς.



Εικόνα 21: hit ratio (uniform TTL)



Εικόνα 22: refreshing hit ratio (uniform TTL)



Εικόνα 23: flushing ratio (uniform TTL)

Κεφάλαιο 4° - Συμπεράσματα και μελλοντική δουλειά

4.1 Συμπεράσματα

Μιλήσαμε για ότι το results caching είναι ένας πολύ σημαντικός παράγοντας για την απόδοση της μηχανής αναζήτησης. Επειδή, συνήθως τα συνήθως τα query streaming τα χαρακτηρίζει μια power-low κατανομή, εκμεταλλευόμαστε την ιδιότητα αυτή και αποθηκεύοντας στη cache τα πιο συχνά queries, πετυχαίνουμε καλύτερους χρόνους απόκρισης για το χρήστη, και δεύτερον μειωμένη επεξεργαστικό φόρτο στη μηχανή αναζήτησης. Μάλιστα, όσο πιο «σκεβρωμένη» είναι η κατανομή (μεγαλύτερος εκθέτης στη Zipf), τόσο μεγαλύτερο hit ratio επιτυγχάνεται.

Μετά σχολιάσαμε το πρόβλημα του freshness, που αντιμετωπίζουν οι μηχανές αναζήτησης, και είναι σημαντικότερο από την επιλογή πολιτική αντικατάστασης, μιας και στις τεράστιες caches που έχουν οι εμπορικές μηχανές αναζήτησης, δεν παίζει ιδιαίτερο ρόλο η πολιτική (χωράνε τόσα πολλά που δεν εκτελείται τόσο συχνά ο αλγόριθμος αντικατάστασης), ενώ όσο μεγαλύτερη cache, τόσο περισσότερα non fresh entries. Το πρόβλημα αυτό επηρεάζει άμεσα τους χρήστες αφού μπορεί να τους επιστρέφονται ανακριβή αποτελέσματα.

Προτείναμε λύση στο παραπάνω πρόβλημα την χρήση μιας τιμής TTL για να θέτουμε expired τα cache entries που βρίσκονται στη cache για περισσότερο από TTL. Όμως αυτό με τη σειρά του οδηγεί σε περισσότερα miss caches, αφού έτσι αντιμετωπίζονται τα «ληγμένα» entries, και μάλιστα το miss ratio εξ αιτίας expiration είναι αναλογικό με το μέγεθος της cache.

Οπότε μια ακόμα βελτίωση είναι η χρήση ενός μηχανισμού refreshing για τα expired entries, ώστε να είναι όσο το δυνατόν λιγότερα μέσα στη cache και άρα να έχουμε πιο «φρέσκια» cache. Ο μηχανισμός βασίζεται στην ηλικία των entries και ανανεώνει τα παλαιότερα.

4.2 Μελλοντική δουλειά

Πως θα βελτιώσουμε περισσότερο το μηχανισμό του refreshing

- λαμβάνοντας υπόψη, πέρα από το **age**, και τη **frequency** των cache entries
- κάθε πότε πρέπει να κάνουμε refresh;
- και πόσα entries κάθε φορά;

References

1. http://www.internetworldstats.com/stats.htm
2. http://www.worldwidewebsize.com/
3. http://en.wikipedia.org/wiki/World_Wide_Web
4. http://en.wikipedia.org/wiki/Search_engine
5. http://en.wikipedia.org/wiki/Web_crawling
6. http://en.wikipedia.org/wiki/Cache_algorithms
7. http://en.wikipedia.org/wiki/Belady's_anomaly
8. http://lucene.apache.org/java/docs/index.html
9. http://en.wikipedia.org/wiki/Text_Retrieval_Conference
10. http://trec.nist.gov/
11. http://ir.ohsu.edu/ohsumed/ohsumed.html
12. http://en.wikipedia.org/wiki/Zipf's_law
13. B. B. Camabzoglu, F. P. Junqueira, V. Plachouras et al. A Refreshing Perspective of Search Engine Caching, 2010
14. E. P. Markatos. On caching search engine query results. Comp. Commun., 24(2):137–143, 2001.
15. R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In 30th SIGIR Conference, pages 183–190, 2007.
16. E. A. Brewer. Lessons from giant-scale services. IEEE Internet Computing, 5(4):46–55, 2001.
17. Y. Xie and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In INFOCOM, pages 1238–1247, 2002.

Τέλος
Ευχαριστώ