Περιεχόμενα

1 Εισαγωγή

1.1 Μηχανές Αναζήτησης Διαδικτύου

Η μηχανή αναζήτησης διαδικτύου, είναι ένα πολύπλοκο σύστημα το οποίο είναι σχεδιασμένο για να αναζητά πληροφορία στον παγκόσμιο ιστό. Τα αποτελέσματα ενώς ερωτήματος, στην γενική περίπτωση, παρουσιάζονται σε σελίδες αποτελεσμάτων, οι οποίες περιέχουν ιστοσελίδες, εικόνες, video, ή διάφορους άλλους τύπους αρχείων. Ακόμη, μερικές μηχανές αναζήτησης συλλέγουν δεδομένα διαθέσιμα από βάσεις δεδομένων στο διαδίκτυο ή και open directories. Η συλλογή των πληροφοριών αυτών, γίνεται αυτόματα με την χρήση αλγορίθμων πάνω σε web crawlers (ανιχνευτές ιστού).

1.2 Λειτουργία Μηχανών Αναζήτησης Διαδικτύου

Μία τυπική μηχανή αναζήτησης διαδικτύου πραγματοποιεί αυτές τις διαδικασίες σε σχεδόν πραγματικό χρόνο:

- Web crawling
- Indexing
- Searching

Οι μηχανές αναζήτησης ιστού, συλλέγουν την πληροφορία τους από ιστότοπο σε ιστότοπο με ειδικά προγράμματα “web crawler”. Η διαδικασία αυτή, υλοποιείται με την χρήση υπερσυνδέσμων (hyper links) προς άλλες σελίδες και πόρους, ώστε να ανιχνευτούν όλα τα έγγραφα και σελίδες που συνιστούν τον Ιστό. Η πληροφορία που συναντάται αποστέλλεται στου ευρετηριοποιητές (indexers), κάτω από ορισμένα κριτήρια, όπως τίτλο, περιεχόμενο της σελίδας, meta tags της HTML ακόμη και κώδικα Javascript που περιέχονται στην σελίδα αυτή.

Στην φάση της ευρετηριοποίησης (indexing), γίνεται αντιστοίχηση λέξεων ή και λημμάτων τα οποία εντοπίζονται στις σελίδες κατά την φάση του crawling, με τα domain names και τα HTML πεδία των σελίδων αυτών. Η αντιστοίχηση αυτή, γίνεται με την χρήση βάσεων δεδομένων οι οποίες στη συνέχεια χρησιμοποιούνται από μηχανές αναζήτησης για την εκτέλεση ερωτημάτων (“searching”).

Τυπικά ο χρήστης, αναζητά κάποιες "λέξεις κλειδιά" οι οποίες αναζητούνται στα ήδη υπάρχοντα ευρετήρια, επιστρέφοντας πιθανές αντιστοιχίσεις τους με σελίδες, αρχεία κοκ.. Στην συνέχεια, για να παραχθεί η λίστα αποτελεσμάτων εφαρμόζονται τεχνικές κατάταξης ή και εφαρμογής ειδικών φίλτρων, όταν αυτά παρέχονται, για πιο ακριβή και ειδική αναζήτηση.

Η επιτυχία μίας μηχανής αναζήτησης, ορίζεται από την σχετικότητα της λίστας αποτελεσμάτων της, καθώς και το πλήθος αυτών. Η σχετικότητα των αποτελεσμάτων προκύπτει από διάφορες τεχνικές που διαφέρουν από μηχανή σε μηχανή. Επιπρόσθετα οι μέθοδοι αυτοί εξελίσσονται με την πάροδο του χρόνου, ακολουθώντας την πορεία εξέλιξης του διαδικτύου. Τέλος οι μηχανές αναζήτησης οι οποίες είναι δωρεάν, έχουν κέρδος από τις ανακατατάξεις αποτελεσμάτων ή από προώθηση προϊόντων σε κάθε αναζήτηση.

2 Βελτιστοποίηση Indexer

2.1 Εκτέλεση

Αρχικά δημιουργούμε ένα τυχαίο υποσύνολο 5 κειμενικών στοιχείων, από την συλλογή Gutenberg. Στην συνέχεια, εκτελείται η δοθείσα υλοποίηση του *Indexer* με είσοδο το υποσύνολο της συλλογής που δημιουργήσαμε, έτσι ώστε να μετρήσουμε χρόνους εκτέλεσης. Τα αρχεία κειμένου έχουν μεγέθη από 868KB – 4.7MB. Το εξαγόμενο αρχείο ανεστραμμένου ευρετηρίου έχει μέγεθος 10.7MB. Οι χρόνοι εκτέλεσης για το κάθε αρχείο εισόδου παρατίθενται στο σχήμα 2.1 .

```
Processing: ../books/00ws110.txt (232.64 sec)
Processing: ../books/idiot10.txt ( 80.29 sec)
Processing: ../books/nb17v11.txt (162.31 sec)
Processing: ../books/8ataw11.txt ( 89.73 sec)
Processing: ../books/1cahe10.txt (142.23 sec)
```

Σχήμα 2.1 : Αρχική εκτέλεση build index

2.2 Βελτιστοποιήσεις

Όλες οι υλοποιήσεις που θα ακολουθήσουν είναι εξελικτικές και οι χρόνοι που παρουσιάζονται είναι βελτιώσεις των προηγούμενων υλοποιήσεων.

2.2.1 Μετατροπή *lists* σε *sets*

Αυτή η αλλαγή φαντάζει ανούσια, αλλά ο τελεστής *in* χρησιμοποιείται πολλές φορές για την αναζήτηση κάθε λέξης και της συντακτικής της μορφολογίας, μέσα σε σταθερά σύνολα. Έτσι μία μικρή βελτίωση στο χρόνο εκτέλεσης της αναζήτησης, θα επιφέρει συνολικά σημαντικό όφελος.

```
Processing: ../books/00ws110.txt (213.16 sec)
Processing: ../books/idiot10.txt ( 76.61 sec)
Processing: ../books/nb17v11.txt (153.87 sec)
Processing: ../books/8ataw11.txt ( 88.33 sec)
Processing: ../books/1cahe10.txt (135.73 sec)
```

Σχήμα 2.2 : Μετατροπή list σε sets

Ο αρχικός κώδικας χρησιμοποιεί Python *lists* και όχι *sets* για την αναπαράσταση των συνόλων αυτών. Τα *sets* έχουν σταθερό χρόνο αναζήτησης $O(1)$, στην μέση περίπτωση, έναντι του γραμμικού $O(n)$, που διαθέτουν οι λίστες[1]. Μετατρέποντας

[1] `https://wiki.python.org/moin/TimeComplexity`

τις λίστες των stopwords, categories tags, στις οποίες γίνονται οι αναζητήσεις, σε set προκύπτουν οι χρόνοι εκτέλεσης του σχήματος 2.2

Από τα παραπάνω αποτελέσματα προκύπτει $max\ speedup = 1.05$, το οποίο είναι αρκετά μικρό αλλά για τα μεγαλύτερα αρχεία, προσφέρει αισθητή μείωση στο χρόνο εκτέλεσης του *Indexer*.

2.2.2 Precompiled regular expressions

Ως γνωστόν, τα regular expressions είναι ένα εργαλείο με μεγάλες δυνατότητες αλλά και με μεγάλο κόστος υπολογισμού. Στον αρχικό κώδικα, το χρησιμοποιούμενο regex το οποίο κάνει match έναν μη λεκτικό χαρακτήρα, υπολογίζεται συνεχώς από την αρχή, ενώ αυτό χρησιμοποιείται σε κάθε επανάληψη.

Κάνοντας χρήση της μεθόδου compile της βιβλιοθήκης *re* της Python, το regex υπολογίζεται μία φορά και στην συνέχεια χρησιμοποιείται για κάθε αναζήτηση, χωρίς να επιβαρυνόμαστε με το επιπλέον κόστος επαναϋπολογισμού. Σημειώνεται πως και η Python εσωτερικά προσπαθεί να κάνει cache τα συχνά χρησιμοποιούμενα regexes, αλλά παρατηρείται και πάλι μία αισθητή διαφορά στο χρόνο εκτέλεσης στο σχήμα 2.3

```
Processing: ../books/00ws110.txt (208.14 sec)
Processing: ../books/idiot10.txt ( 73.66 sec)
Processing: ../books/nb17v11.txt (152.31 sec)
Processing: ../books/8ataw11.txt ( 85.08 sec)
Processing: ../books/1cahe10.txt (134.11 sec)
```

Σχήμα 2.3 : Χρήση precompiled regex

2.2.3 PoStags closed class categories

Τα closed class categories είναι γραμματικές κατηγορίες για λέξεις άνευ σημασιολογικού περιεχομένου, δηλαδή stopwords. Συνεπώς, είναι επιθυμητό η αφαίρεσή τους έπειτα από την μορφοσυνακτική ανάλυση του κειμένου. Η αλλαγή αυτή έγινε και στην φάση της ευρετηριοποίησης αλλά και σε αυτή της αναζήτησης, δηλαδή στα αρχεία *BuildIndex.py* και *QueryIndex.py* αντίστοιχα.

Η λειτουργία αυτή γινόταν και πρότινος αλλά δεν ήταν πλήρης. Ο τρόπος με τον οποίο διορθώθηκε, είναι με την χρήση του πλήρη πίνακα *Modified Penn Treebank Tag-Set*[2](closed class categories). Οι κατηγορίες αυτές περιλαμβάνουν και ειδικούς χαρακτήρες όπως αριθμούς, σύμβολα λίστας κτλπ.

[2] `http://www.infogistics.com/tagset.html`

```
Processing: ../books/00ws110.txt (208.50 sec)
Processing: ../books/idiot10.txt ( 74.35 sec)
Processing: ../books/nb17v11.txt (148.42 sec)
Processing: ../books/8ataw11.txt ( 79.70 sec)
Processing: ../books/1cahe10.txt (125.46 sec)
```

Σχήμα 2.4 : Χρήση Modified Penn Treebank Tag-Set

Στην πλειοψηφία των περιπτώσεων, παρατηρείται σημαντική επιτάχυνση, πράγμα που οφείλεται στο γεγονός πως περισσότεροι όροι αφαιρούνται, άρα και η πρόσθεση των όρων στο ανεστραμμένο αρχείο γίνεται γρηγορότερα. Το μέγεθος του ευρετηρίου, μάλιστα, έπειτα από την παραπάνω εκτέλεση, μειώθηκε σε μέγεθος στα 10.1MB.

2.2.4 Παραλληλοποίηση Indexer

Το μεγάλο πλήθος αρχείων εισόδου καθώς και το μέγεθός αυτών, απαιτούν μία γρήγορη υλοποίηση, η οποία επιτρέπει την όσο δυνατόν ταχύτερη προσπέλαση και δεικτοδότηση (indexing) αυτών. Η μόνη οδός πλέον, χωρίς να γίνουν αλλαγές στο βασικό μοντέλο, είναι η παραλληλοποίηση.

Για το σκοπό αυτό, υπάρχουν δύο βασικές μέθοδοι: η δημιουργία μιας πολυνηματικής εφαρμογής αλλά και η δημιουργία μιας πολύ-διεργασικής εφαρμογής. Η πρώτη κατηγορία, ενδείκνυνται όταν οι εφαρμογές είναι IO/bound ενώ η τελευταία όταν οι εφαρμογές είναι CPU/bound, δηλαδή ο χρόνος εκτέλεσης εξαρτάται από το χρόνο εισόδου έναντι του χρόνου υπολογισμού αντίστοιχα.

Οι διαδικασίες *tagging*, *lemmatization* είναι CPU/bound, ενώ ακόμη η βιβλιοθήκη *NLTK* δεν είναι thread-safe. Από τα παραπάνω προκύπτει ότι η παράλληλη έκδοση θα εξελιχθεί ως μια πολύ-διεργασική εφαρμογή.

Υλοποίηση: Για την δημιουργία της εφαρμογής, εκμεταλλευτήκαμε το Multiprocessing API[3] που παρέχει η Python. Συγκεκριμένα, δημιουργούμε μία "πισίνα διεργασιών", στις οποίες ανατίθενται η επεξεργασία των αρχείων εισόδου. Με τον τρόπο αυτό, πολλαπλά αρχεία μπορούν να επεξεργάζονται ταυτόχρονα, μειώνοντας έτσι το συνολικό χρόνο εκτέλεσης. Το πλήθος των διεργασιών εξαρτάται από το μηχάνημα που εκτελείται η εφαρμογή. Συγκεκριμένα στο μηχάνημα που πραγματοποιήθηκαν οι μετρήσεις, υπάρχουν μόνο δύο πυρήνες, άρα είναι σκόπιμο οι διεργασίες να είναι δύο και αυτές.

Κάθε διεργασία καλεί την συνάρτηση επεξεργασίας αρχείου *parse_file* και στην συνέχεια ανανεώνει την κοινή δομή ανεστραμμένου ευρετηρίου. Για να επιτευχθεί

[3] `https://docs.python.org/2/library/multiprocessing.html`

διαμοιρασμός του ανεστραμμένου ευρετηρίου μεταξύ διεργασιών (shared memory), χρησιμοποιήθηκε η κλάση *Manager* της Python που διαμοιράζει σε όλες τις διεργασίες παιδιά την ίδια δομή δεδομένων, στην περίπτωση μας ένα Python *dict*.

Ακόμη, δεσμεύτηκε μία δομή *Manager.Lock*, η οποία διαφυλάττει την κρίσιμη περιοχή, δηλαδή την επεξεργασία του κοινού ευρετηρίου, και επιτρέπει την ανανέωση του από μία διεργασία κάθε φορά. Στην πρώτη αυτή υλοποίηση, επιλέχτηκε το ευρετήριο να διαμοιράζεται μεταξύ διεργασιών ώστε να χρειάζεται η ελάχιστη δυνατόν μνήμη, με επιβάρυνση βέβαια στην επικοινωνία των διεργασιών. Αργότερα θα σχολιαστούν και άλλες μέθοδοι.

Για την διαμοίραση των εργασιών στις διεργασίες παιδιά, χρησιμοποιήθηκε η μέθοδος *map_async* της κλάσης *Pool*. Στην τελευταία, μία λίστα από στοιχεία - δεδομένα διαμοιράζονται σε όλες τις διεργασίες παιδιά της "πισίνας", μαζί με μία αναφορά στην συνάρτηση επεξεργασίας, ενώ τελικά τα αποτελέσματα αυτών επιστρέφονται σε τυχαία σειρά στην διεργασία γονέα. Τα αποτελέσματα αυτά, στην υλοποίηση μας, είναι οι μετρητές πλήθους λέξεων που αφαιρέθηκαν ή προστέθηκαν στο ευρετήριο.

Εναλλακτικά, αντί για χρήση της *map_async* θα μπορούσαμε να χρησιμοποιήσουμε την συνάρτηση *map*. Η διαφορά των δύο, είναι πως η πρώτη δεν σταματά την ροή εκτέλεσης στην main. Για την υλοποίηση αυτή, δεν υπάρχει κάποια διαφορά, μιας και μεταξύ των κλήσεων της συνάρτησης *map* και υπολογισμού του score *tfidf*, δεν παρεμβάλλεται κάποια ανεξάρτητη πράξη. Βέβαια, παρατηρήθηκε πως η πρώτη έχει καλύτερη απόδοση έναντι της δεύτερης, λόγο του ότι δεν χρειάζεται να αναδιατάξει τα αποτελέσματα.

Ένα ακόμη σημαντικό στοιχείο αυτής της υλοποίησης, είναι ο τρόπος που το κοινό ευρετήριο διαμοιράζεται μεταξύ διεργασιών. Κάθε προσπέλαση γίνεται προστατευόμενη από ένα mutex (Lock), το οποίο διαφυλάττει πως μία διεργασία μεταβάλλει τις τιμές του ευρετηρίου κάθε φορά. Ακόμη, παρατηρήθηκε λόγο της υλοποίησης της κλάσης *Manager* πως η απευθείας προσπέλαση είναι απαγορευτικά αργή. Για τον λόγο αυτό, το ανεστραμμένο ευρετήριο, αντιγράφεται τοπικά στην μνήμη της διεργασίας παιδιού και εφόσον τροποποιηθεί αποθηκεύεται πίσω στην κοινή μνήμη.

Τέλος, έγινε μία μικρή επέκταση του argument parser, ώστε αυτός να δέχεται και ένα επιπλέον όρισμα, αυτό του πλήθους των διεργασιών. Με τον τρόπο αυτή η εκτέλεση του *BuildIndex.py* είναι παραμετροποιημένη, ώστε να υποστηρίζονται και άλλα πολυπύρηνα συστήματα. Το νέο όρισμα δίνεται με το χαρακτήρα P, ενώ εξ ορισμού η εκτέλεση γίνεται σε ένα πυρήνα (default $P = 1$).

Αποτελέσματα: Στο σχήμα 2.5 παρατίθενται τα αποτελέσματα εκτέλεσης σε μία διεργασία παιδί ($P = 1$), ενώ στο σχήμα 2.6 για δύο διεργασίες παιδιά ($P = 2$). Σημειώνεται, πως οι μετρήσεις αυτές έγιναν σε διαφορετικό μηχάνημα από το αρχικό, το οποίο διαθέτει δύο πυρήνες, έναντι τους ενός που εκτελέστηκαν οι προηγούμενες.

Όπως θα περίμενε κάποιος, ο χρόνος εκτέλεσης για δύο διεργασίες θα έπρεπε να είναι ο μισός του αρχικού, αλλά αυτό δεν είναι εφικτό αν αναλογιστούμε τα μη παραλληλοποιήσημα τμήματα του κώδικα, καθώς και την επιβάρυνση της επικοινωνίας των διεργασιών / κρίσιμες περιοχές κτλπ.. Παρόλα αυτά, η παραλληλοποίηση είχε $speedup = 1.5$, το οποίο είναι αρκετά ικανοποιητικό.

```
memas@debian:~/Desktop/inf_ret$ time python BuildIndex.py -I ../books/ -0 . -P 1
Pid: 1978 processing: ../books/nb17v11.txt ( 43.81 sec)
Pid: 1978 processing: ../books/idiot10.txt ( 25.11 sec)
Pid: 1978 processing: ../books/00ws110.txt (114.62 sec)
Pid: 1978 processing: ../books/1cahe10.txt ( 48.18 sec)
Pid: 1978 processing: ../books/8ataw11.txt ( 33.72 sec)

real    4m40.033s
user    4m32.596s
sys     0m7.240s
```

Σχήμα 2.5 : Εκτέλεση BuildIndex για P=1

```
memas@debian:~/Desktop/inf_ret$ time python BuildIndex.py -I ../books/ -0 . -P 2
Pid: 2021 processing: ../books/idiot10.txt ( 27.86 sec)
Pid: 2020 processing: ../books/nb17v11.txt ( 51.98 sec)
Pid: 2020 processing: ../books/1cahe10.txt ( 35.94 sec)
Pid: 2020 processing: ../books/8ataw11.txt ( 25.54 sec)
Pid: 2021 processing: ../books/00ws110.txt (142.67 sec)

real    3m4.922s
user    4m48.472s
sys     0m7.344s
```

Σχήμα 2.6 : Εκτέλεση BuildIndex για P=2

2.2.5 Προσπάθειες Βελτιστοποίησης Παραλληλοποίησης

Στο σημείο αυτό, θα περιγραφεί μία σειρά από προσπάθειες που έγιναν ώστε να βελτιστοποιηθεί η μέθοδος της παραλληλοποίησης, στο συγκεκριμένο μηχάνημα:

Το μέγεθος της κρίσιμης περιοχής, αποτελεί ένα μεγάλο πρόβλημα, καθώς πολλές διεργασίες προσπαθώντας να εισέλθουν, ενώ κάποια άλλη βρίσκεται ήδη μέσα, σπαταλούν υπολογιστικό χρόνο περιμένοντας ανενεργές μέχρις ότου να αποκτήσουν το Lock. Το δε πρόβλημα, είναι εντονότερο όταν αντί για $P = 2$ διεργασίες εργάτες, υπάρχουν αρκετές παραπάνω. Στην περίπτωση αυτή, τα race conditions είναι εντονότερα.

Για την αντιμετώπιση του παραπάνω προβλήματος, έγιναν αρκετές απόπειρες. Αρχικά δοκιμάστηκε η επιστροφή επιμέρους ανεστραμμένων ευρετηρίων από κάθε worker process και σύνθεση αυτών στο master process. Αν και δοκιμάστηκε και η σύγχρονη αλλά και ασύγχρονη σύνθεση, ο χρόνος μεταφοράς και σύνθεσης ήταν μεγαλύτερος από τον χρόνο που σπαταλιέται στα κρίσιμα σημεία.

Το ίδιο φαινόμενο παρατηρήθηκε στην διατήρηση επί μέρους ιδιωτικών ανεστραμμένων ευρετηρίων, τα οποία διατηρεί κάθε διεργασία σε κοινό χώρο μνήμης, ώστε να μην σπαταλείται χρόνος για μεταφορά στο χώρο μνήμης της διεργασίας γονέας. Η κάθε διεργασία εργάτης επεξεργάζεται ατομικά το ευρετήριο της και στο τέλος όλα τα τμήματα, συνθέτονται σε ένα ανεστραμμένο ευρετήριο μέσω του master process.

Ακόμη, δοκιμάστηκαν αναδιατάξεις και ταξινομήσεις στην σειρά με την οποία επεξεργάζονται τα αρχεία εισόδου, με βάση το μέγεθός τους, έτσι ώστε να υπάρχουν όσο το δυνατόν λιγότερα race conditions κατά την είσοδο στην κρίσιμη περιοχή, αλλά και ο φόρτος εργασίας να είναι ίσο-κατανεμημένος, χωρίς κάποιο ουσιαστικό όφελος.

Πιθανώς μία υλοποίηση χωρίς κρίσιμη περιοχή θα ήταν απαραίτητη για περισσότερες από $P = 2$ διεργασίες εργάτες, όπου εκεί θα υπήρχε εντονότερο το πρόβλημα της αναμονής για απόκτηση του Lock, αλλά για το μηχάνημα μας το οποίο διαθέτει δύο πυρήνες μόνο, είναι ιδανικότερη, καθώς δεν επιβαρύνεται με χρόνους συλλογής και σύνθεσης των επιμέρους αποτελεσμάτων.

2.2.6 Αποθήκευση ανεστραμμένου ευρετηρίου

Η αποθήκευση του ανεστραμμένου ευρετηρίου, είναι μία πολύ βασική διαδικασία, καθώς αυτό πρέπει να είναι διαθέσιμο κατά την διαδικασία των ερωτημάτων, ενώ στην συνέχεια, μπορεί να ανανεωθεί και να επεκταθεί. Στην συγκεκριμένη υλοποίηση το ευρετήριο αποθηκεύεται σε μορφή αρχείου κειμένου ASCII, το οποίο σπαταλάει πολύ χώρο.

Επίσης, η μέθοδος αποθήκευσης της βιβλιοθήκης JSON, η οποία και χρησιμοποιείται στην αρχική υλοποίηση, περιορίζει την δομή του ευρετηρίου σε Python *dict* (ταυτόσημο με JSON). Η υλοποίηση που προτείνεται, είναι αυτή των αρχείων Pickle[4] είτε η χρήση κάποιου συστήματος κατανεμημένης βάσης / δομής δεδομένων (π.χ. Redis[5]). Στην τελευταία περίπτωση η αξιοπιστία αποθήκευσης / ανάγνωσης ενός μεγάλου ανεστραμμένου ευρετηρίου από/σε ένα ή πολλαπλά μηχανήματα είναι εγγυημένη. Στην απλή υλοποίηση αυτή, θα χρησιμοποιηθούν τα

[4] `https://docs.python.org/2/library/pickle.html`

[5] `https://redis.io/`

αρχεία Pickle.

Τα αρχεία Pickle, είναι μία μέθοδος σειριοποίησης αντικειμένων στην Python με δυνατότητα δυαδικής αποθήκευσης. Έτσι τα αρχεία των ανεστραμμένων ευρετηρίων απαιτούν τον ελάχιστο δυνατό χώρο, ενώ εύκολα μπορούν να υποστηριχτούν και άλλες δομές ανεστραμμένων αρχείων όπως B+ δένδρα. Για την υλοποίηση αυτή, χρειάζεται να μεταβληθεί και το αρχείο *QueryIndex* ώστε να υποστηρίζεται η φόρτωση της νέας δομής αρχείου, κατά την φάση των ερωτημάτων.

Με την μέθοδο αυτή, όχι μόνο το ανεστραμμένο αρχείο είναι σημαντικά μικρότερο, αλλά και η εκτέλεση του *BuildIndex.py* είναι γρηγορότερη. Το νέο μέγεθος είναι περίπου στα 7MB έναντι των 10.1MB της προηγούμενης μας υλοποίησης, δηλαδή μείωση αποθηκευτικού χώρου της τάξης του 30%.

2.2.7 Συμπίεση ανεστραμμένου ευρετηρίου

Εκτός από την σειριοποίηση του ευρετηρίου και αποθήκευση του ως δυαδικό αρχείο, θα μπορούσαμε να μειώσουμε και άλλο το μέγεθός του εφαρμόζοντας κάποια βιβλιοθήκη συμπίεσης της Python (π.χ. *gzip*), για ακόμη καλύτερα αποτελέσματα. Στο σημείο αυτό όμως, περιγράφεται ο τρόπος με τον οποίο μειώσαμε το μέγεθος του ευρετηρίου εφαρμόζοντας μεθόδους συμπίεσης ευρετηρίων και όχι κάποιο γενικό αλγόριθμο συμπίεσης. Η μέθοδος που ακολουθήθηκε, είναι αυτή της κωδικοποίηση κενών *d-gap*.

Συγκεκριμένα, κάθε λήμμα στο ευρετήριο, περιέχει ένα σύνολο από document ids τα οποία με την σειρά τους περιλαμβάνουν μία μεγάλη λίστα από σημεία που εντοπίζεται το λήμμα στο αντίστοιχο κειμενικό στοιχείο, ώστε να υποστηρίζονται phrase queries. Τα σημεία αυτά, αναπαριστώνται από φυσικούς αριθμούς, οι οποίοι κωδικοποιούν την θέση των λημμάτων στο κείμενο. Για τα μεγαλύτερα έγγραφα, με αρκετές χιλιάδες λέξεις, οι αριθμοί αυτοί μπορούν να μεγαλώσουν αρκετά γρήγορα. Επίσης σε κάθε κείμενο, τα ίδια λήμματα συναντώνται αρκετές φορές, άρα οι λίστες αυτές είναι αρκετά μεγάλες. Έτσι θα ήταν ωφέλιμο να ακολουθηθεί αναπαράσταση των θέσεων με κωδικοποίηση d-gaps.

Κατά την φάση εισαγωγής, η λίστα με τις θέσεις είναι ήδη ταξινομημένη, εφόσον κάθε αρχείο επεξεργάζεται από μία διεργασία μόνο, και οι λέξεις εισάγονται με την σειρά που εντοπίζονται στο κείμενο. Άρα η προσθήκη της εν λόγω κωδικοποίησης αποδεικνύεται αρκετά εύκολη διαδικασία, η οποία και εκτελείται όταν ανανεώνονται τα λήμματα στο ευρετήριο, δηλαδή στην συνάρτηση *update_inverted_index*.

Μεταβάλλοντας την αναπαράσταση των λιστών θέσης, απαιτούνται και οι κατάλληλες αλλαγές στο αρχείο *QueryIndex.py* και συγκεκριμένα στα ερωτήματα

φράσεων. Δηλαδή στην συνάρτηση *phrase_query*, πρέπει να γίνεται αποκωδικοποίηση της αναπαράστασης d-gaps σε απόλυτη θέση λήμματος για το κάθε document id.

Στην υλοποίηση που ακολουθήθηκε, η αποκωδικοποίηση γίνεται σε κάθε ερώτημα και όχι με την φόρτωση του ανεστραμμένου αρχείου στην μνήμη. Το τελευταίο απαιτεί την φόρτωση ολόκληρου του ευρετηρίου στην μνήμη, ενώ ο χρήστης μπορεί να κάνει μερικά ερωτήματα τα οποία δεν χρησιμοποιούν καν παραπάνω από μερικά λήμματα. Άρα η παραπάνω επιβάρυνση ειδικά σε μεγάλα ευρετήρια είναι απαγορευτική.

Έπειτα από μετρήσεις στο ίδιο σύνολο κειμένων, προκύπτει πως το νέο αρχείο ανεστραμμένου ευρετηρίου μειώθηκε στα 5.6MB από τα 7MB. Η μείωση αυτή δεν είναι μεγάλη αλλά εξαρτάται από το πόσο συχνά εμφανίζεται ένα λήμμα σε ένα κείμενο. Όσο συχνότερα, τόσο τα διαστήματα θα είναι μικρότερα άρα και η κωδικοποίηση αυτή θα έχει καλύτερα αποτελέσματα.

2.3 Τελικά Αποτελέσματα

Για τα τελικά αποτελέσματα, έγινε σύγκριση της αρχικής υλοποίησης του αρχείου *BuildIndex.py* που δόθηκε με την εκφώνηση της εργασίας, με την τελική βελτιστοποιημένη έκδοση. Τα αποτελέσματα χρόνων εκτέλεσης παρατίθενται στο σχήμα 2.7 ενώ τα μεγέθη των εξαγόμενων ανεστραμμένων ευρετηρίων στο σχήμα 2.8 :

```
memas@debian:~/Desktop/inf_ret$ time python bindex.py -I ../books/ -O ./
Processing: ../books/nb17v11.txt ( 49.83 sec)
Processing: ../books/idiot10.txt ( 27.69 sec)
Processing: ../books/00ws110.txt (127.27 sec)
Processing: ../books/1cahe10.txt ( 55.98 sec)
Processing: ../books/8ataw11.txt ( 40.05 sec)

real    5m4.134s
user    5m1.528s
sys     0m2.740s
memas@debian:~/Desktop/inf_ret$ time python BuildIndex.py -I ../books/ -O ./ -P
2
Pid: 1797 processing: ../books/idiot10.txt ( 23.78 sec)
Pid: 1796 processing: ../books/nb17v11.txt ( 41.40 sec)
Pid: 1796 processing: ../books/1cahe10.txt ( 23.57 sec)
Pid: 1796 processing: ../books/8ataw11.txt ( 13.50 sec)
Pid: 1797 processing: ../books/00ws110.txt ( 81.78 sec)

real    1m53.141s
user    3m3.940s
sys     0m5.512s
```

Σχήμα 2.7 : Χρόνοι εκτέλεσης υλοποιήσεων

Από τα παραπάνω αποτελέσματα, παρατηρείται σημαντική επιτάχυνση εκτέλεσης, ίση με $speedup = 2.68$ από την αρχική υλοποίηση, για παράλληλη εκτέλεση σε

```
memas@debian:~/Desktop/inf_ret$ ls -l | grep inverted_file.
-rw-r--r-- 1 memas memas  5793794 Oct  1 13:06 inverted_file.pickle
-rw-r--r-- 1 memas memas 10704210 Jul 25 08:45 inverted_file.txt
```

Σχήμα 2.8 : Μεγέθη ανεστραμμένων ευρετηρίων

$P = 2$ διεργασίες. Επίσης, το εξαγόμενο αρχείο ανεστραμμένου ευρετηρίου έχει μέγεθος 5.6MB έναντι των 10.7MB του αρχικού, δηλαδή μείωση της τάξης του 52%.

3 Recommender System

Ένα σύστημα προτάσεων[6], είναι το σύστημα το οποίο προβλέπει την προτίμηση του χρήστη για ένα προϊόν ή υπηρεσία. Τα συστήματα αυτά είναι αρκετά δημοφιλή τα τελευταία χρόνια και έχουν εισαχθεί σε μία πληθώρα σελίδων, όπως βάσεις για ταινίες, μουσική, νέα, βιβλία, μηχανές αναζήτησης κ.α..

Στα πλαίσια αυτής της εργασίας, δημιουργήθηκε ένα μικρό υποσύστημα το οποίο μπορεί να προστεθεί στην βασική μηχανή αναζήτησης, και συγκεκριμένα στο τμήμα εφαρμογής ερωτημάτων. Συγκεκριμένα, εφόσον ο χρήστης αναζητά κείμενα από την συλλογή Gutenberg, να λαμβάνει προτάσεις για άλλα βιβλία που μπορεί να τον ενδιαφέρουν.

Τυπικά υπάρχουν δύο τρόποι παραγωγής προτάσεων:

- Collaborative filtering
- Content-based filtering

Στην πρώτη μέθοδο, κατασκευάζεται ένα μοντέλο από παλαιά συμπεριφορά χρηστών (αγορές, αξιολογήσεις προϊόντων), αλλά και παρόμοιες αποφάσεις άλλων χρηστών. Στο content-based-filtering, χρησιμοποιούνται τα διακριτά χαρακτηριστικά ενός αντικειμένου, έτσι ώστε να προταθούν αντίστοιχα προϊόντα με παρόμοια χαρακτηριστικά.

Στα πλαίσια της εργασία αυτής, αναπτύχθηκε ένα collaborative filtering σύστημα, στο οποίο υποθετικά οι χρήστες βαθμολογούν βιβλία της συλλογής Gutenberg και στην συνέχεια παρέχονται προτάσεις για αυτά, με βάση τις προτιμήσεις άλλων χρηστών.

3.1 Pearson Correlation

Για την εύρεση συσχέτισης μεταξύ δύο χρηστών χρησιμοποιήθηκε ο συντελεστής συσχέτισης Pearson[7], καθώς αποδίδει καλύτερα αποτελέσματα από την απλή ευκλείδεια απόσταση, ενώ είναι αρκετά απλός στον υπολογισμό.

$$r = r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \rightarrow$$
$$r_{xy} = \frac{\sum x_i y_i - \frac{1}{n}\sum x_i \sum y_i}{\sqrt{\sum x_i^2 - \frac{1}{n}(\sum x_i)^2}\sqrt{\sum y_i^2 - \frac{1}{n}(\sum y_i)^2}} \quad (3.1.1)$$

[6] https://en.wikipedia.org/wiki/Recommender_system

[7] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

Στην σχέση 3.1.1, το n αποτελεί το πλήθος των κοινών αξιολογήσεων των χρηστών x,y ενώ τα x_i,y_i τις αξιολογήσεις των χρηστών, σε i αντικείμενα αντίστοιχα.

3.2 Περιγραφή υλοποίησης

Αρχικά χρησιμοποιήθηκε το αρχείο *masters_list.csv*, το οποίο βρίσκεται εντός της συλλογής *Gutenberg* και περιέχει όλα τα κείμενα αυτής, με πληροφορίες για τον συγγραφέα, έτος έκδοσης, κωδικό βιβλίου κτλπ..

Αφού έγινε parsing του αρχείου αυτού, με την βοήθεια της βιβλιοθήκης *csv*[8] της Python, δημιουργήθηκε ένα ευρετήριο (Python dict) με τους κωδικούς, τον συγγραφέα και τίτλο του βιβλίου. Το ευρετήριο αυτό, αποθηκεύεται ως αρχείο Pickle για γρήγορη και εύκολη ανάκτηση σε κάθε κλήση του υποσυστήματος προτάσεων.

Στην συνέχεια, εισήχθηκαν μερικές τυχαίες αξιολογήσεις από χρήστες για μερικά από αυτά τα βιβλία. Ως κλειδί για την κάθε αξιολόγηση ορίζεται το αναγνωριστικό του κάθε χρήστη ενώ ακολουθεί ένα ευρετήριο, με κλειδί το αναγνωριστικό βιβλίων και τιμή την βαθμολογία που έχει δώσει ο κάθε χρήστης για το βιβλίο αυτό. Αυθαίρετα ορίστηκε η βαθμολογία αυτή να είναι ένας αριθμός στο διάστημα (0..5.0] με βήμα 0.5 μονάδων.

```
reviews = {
    'Alice': {'00ws110.txt': 5.0, '8ataw11.txt': 4.5},
    'John' : {'8ataw11.txt': 2.0, 'idiot10.txt': 3.5},
    'Bob'  : {'idiot10.txt': 4.0, '8ataw11.txt': 2.0,
      ↪ '1cahe10.txt': 3.0, '00ws110.txt': 4.0}
}
```

Listing 1: "Δομή αξιολογήσεων χρήστη"

Για κάθε χρήστη της εφαρμογής, υπολογίζεται ο συντελεστής συσχέτισης με τον χρήστη μας (χρήστης για τον οποίο θέλουμε προτάσεις). Αν ο συντελεστής είναι θετικός, τότε υπάρχει συσχέτιση και για τα βιβλία τα οποία δεν έχει γίνει ήδη αξιολόγηση από τον χρήστη μας, (άρα δεν τα έχει διαβάσει ακόμη), υπολογίζεται ο βεβαρημένος μέσος όρος ή η μέση προβλεπόμενη βαθμολογία.

Για τον υπολογισμό της προβλεπόμενης μέσης βαθμολογίας, τίθενται ως βάρη η ομοιότητα των δύο χρηστών, κατά την συσχέτιση Pearson, και ως τιμές οι βαθμολογίες των χρηστών στα βιβλία αυτά. Στο τέλος της εκτέλεσης, για το κάθε βιβλίο, γίνεται φθίνουσα ταξινόμηση με βάση την προβλεπόμενη βαθμολογία. Επίσης, ορίστηκε ένα κατώφλι ίσο με $threshold = 2.0$, κατά το οποίο βιβλία με μικρότερη

[8] https://docs.python.org/2/library/csv.html

προβλεπόμενη βαθμολογία να μην επιστρέφονται στον τελικό χρήστη καθώς δεν ανταποκρίνονται στα "γούστα του".

Στο σχήμα 3.9 παρατίθεται ένα παράδειγμα εκτέλεσης του υποσυστήματος το οποίο εμφανίζει προτάσεις που ανταποκρίνονται στο χρήστης “John” του συστήματος. Παρατηρείται πως η έξοδος είναι το αναγνωριστικό του βιβλίου, το οποίο μπορεί να είναι μία υπερσύνδεση για το βιβλίο στην συλλογή μας, καθώς και ο τίτλος αυτού.

```
memas@debian:~/Desktop/inf_ret$ python Recommender.py
These books are highly recommended for you:
00ws110.txt: The Complete Shakespeare's First Folio, by Shakespeare William
1cahe10.txt: Critical and Historical Essays, by Macaulay
```

Σχήμα 3.9 : Recommendation System execution

4 Δημιουργία Crawler για την συλλογή Gutenberg

Στην δοθείσα υλοποίηση, υπάρχει ένας απλός web crawler για τον οποίο θα εξηγήσουμε την βασική του λειτουργία και στην συνέχεια θα επιδείξουμε τρόπους με τους οποίους μπορεί να βελτιωθεί.

Ο crawler αυτός, δέχεται ως όρισμα την σελίδα εκκίνησης και το πλήθος σελίδων που θα επισκεφτεί. Όταν επισκεφθεί κάποια σελίδα, τότε εντοπίζει και εξάγει όλες τις υπερσυνδέσεις με την βοήθεια της βιβλιοθήκης *BeautifulSoup* της Python. Εφόσον έχει την διεύθυνση στην οποία δείχνει η υπερσύνδεση αυτή, προσπαθεί να την κανονικοποιήσει, π.χ. φτιάχνοντας το domain name και να την προσθέσει στην λίστα με τις νέες σελίδες, αν δεν υπάρχει ήδη ή αν δεν είναι διεύθυνση ηλεκτρονικού ταχυδρομείου. Στην επόμενη επανάληψη του crawler η νέα διεύθυνση εξέρχεται της λίστας και όλα τα παραπάνω βήματα επαναλαμβάνονται μέχρις ότου να φτάσουμε το όριο σελίδων, που τέθηκε ως αρχικός περιορισμός ή να μην υπάρχουν νέες σελίδες.

4.1 Προβλήματα υλοποίησης

Από τα παραπάνω, με μία καλύτερη μελέτη του κώδικα, βλέπουμε η υλοποίηση αυτή έχει αρκετά βασικά προβλήματα. Αρχικά, εύκολα παρατηρεί κάποιος, πως οι σελίδες που επισκέπτεται ο crawler δεν χρησιμοποιούνται κάπου, αλλά ούτε αποθηκεύονται στο δίσκο για έπειτα χρήση.

Ακόμη, η κανονικοποίηση των διευθύνσεων είναι αρκετά ελλιπής, καθώς δεν γίνεται ουσιαστικός έλεγχος της δομής κάθε διεύθυνσης. Συγκεκριμένα δεν γίνεται έλεγχος για άλλες διαδεδομένες διευθύνσεις όπως magnet-urls, ftp κ.α., αλλά μόνο για διευθύνσεις ηλεκτρονικού ταχυδρομείου. Επιπρόσθετα, οι διευθύνσεις που προσπελάζονται είναι μόνο τύπου html, ενώ αγνοούνται άλλοι τύποι όπως αρχεία κειμένου, pdf κτλπ. περιορίζοντας έτσι την λειτουργικότητα του crawler.

4.2 Περιγραφή Νέας Υλοποίησης

Για την υλοποίηση που θα ακολουθήσουμε θα χρησιμοποιηθεί το *scrapy*[9] web crawling framework για Python. Το *scrapy* είναι ένας εύκολα επεκτάσιμος web crawler, ο οποίος μπορεί να προσαρμοστεί εύκολα για λειτουργία σε κάθε σελίδα. Αρχικά προσπελάσει τα ορισμένα url εκκίνησης, επιστρέφοντας τα αποτελέσματα του αιτήματος στον χρήστη, αδιαφανώς.

Η διαδικασία αυτή γίνεται με την κλήση ενός ορισμένου callback function από το χρήστη, η οποία κάνει και την εκτέλεση του αιτήματος ασύγχρονη. Το πλε-

[9] `https://scrapy.org/`

ονέκτημα αυτής της διαδικασίας, είναι πως πολλά αιτήματα μπορούν να γίνονται ταυτόχρονα, εξοικονομώντας έτσι σημαντικό χρόνο αλλά και προσφέροντας ανοχή σε σφάλματα που προκύπτουν από "σπασμένα αιτήματα", τα οποία και αγνοούνται, επιτρέποντας στην εφαρμογή να συνεχίζει ανεξάρτητα.

Από τα παραπάνω προκύπτει πως το *scrapy*, χειρίζεται σφάλματα σε επίπεδο δικτύου χωρίς να ανησυχεί ο προγραμματιστής και να μπερδεύεται στην λογική της υλοποίησης του. Σε αντίθεση με την αρχική υλοποίηση στην οποία τα HTTP status codes πρέπει να ελέγχονται από το προγραμματιστή για το αν το αίτημα απαντήθηκε επιτυχώς από το server (HTTP status code 200).

Ακόμη, παραμετροποιήσιμος είναι και ο τρόπος με τον οποίο γίνεται η προσπέλαση σε έναν ιστότοπο. Συγκεκριμένα, μπορούν να οριστούν χρόνο-καθυστερήσεις ή και ο user-agent που χρησιμοποιείται στα αιτήματα, ώστε το μηχάνημα μας να απεικονίζεται ρεαλιστικά ως κάποιος "πραγματικός χρήστης" και όχι ως κάποια κακόβουλη επίθεση. Ειδικότερα αρκετοί ιστότοποι δεν επιτρέπουν την προσπέλαση τους από μηχανές αναζήτησης, είτε παραθέτουν κάποιους κανόνες περιορισμούς προς αυτές. Οι περιορισμοί αυτοί συνήθως αναφέρονται σε ένα αρχείο με όνομα *robots.txt*.

Ο *scrapy* επιπλέον, υποστηρίζει μία πληθώρα τρόπων για αποθήκευση των σελίδων που προσπελάζονται. Για την βελτίωση της τωρινής λειτουργίας θα χρησιμοποιηθεί μία βάση δεδομένων για την αποθήκευση των σελίδων επιτυγχάνοντας έτσι την ευκολότερη προσπέλαση των σελίδων οποιαδήποτε στιγμή, τόσο για εγγραφή αλλά και ανανέωση περιεχομένου.

Ειδικότερα, ο πίνακας αποθήκευσης προσφέρει και άλλα πλεονεκτήματα όπως την δυνατότητα αποθήκευσης μεταδεδομένων (metadata) αλλά και εξοικονόμηση χώρου αποθήκευσης στο δίσκο. Το τελευταίο προκύπτει από το γεγονός πως η αποθήκευση μικρού μεγέθους αρχείων στο δίσκο οδηγεί σε κατακερματισμό και στην κατανάλωση μεγαλύτερου αποθηκευτικού χώρου από τον απαιτούμενο.

4.3 Υλοποίηση Spider

Η δημιουργία του crawler γίνεται με την χρήση της εντολής *scrapy startproject crawler* η οποία δημιουργεί ένα νέο φάκελο, ονόματος crawler, με όλα τα απαραίτητα αρχεία που χρειάζεται για την λειτουργία του το *scrapy*. Από τα πιο σημαντικά αρχεία, είναι το settings.py, το οποίο περιέχει όλα τα configurations του crawler. Στην συνέχεια τα στοιχεία σύνδεσης της βάση δεδομένων αλλά και άλλες παράμετροι που υπαγορεύουν την λειτουργία του crawler θα οριστούν στο αρχείο αυτό (όπως ο user-agent καθώς και χρόνοι καθυστέρησης προσπέλασης κάθε σελίδας).

Στην συνέχεια, δημιουργείται η πρώτη αράχνη (Spider) μέσω της εντολής *scrapy*

genspider mydomain mydomain.com, η οποία δημιουργεί μία αράχνη για ένα συγκεκριμένο ιστότοπο και αποθηκεύεται στο φάκελο spiders/. Συνεχίζοντας την λογική των προηγούμενων κεφαλαίων, θα δημιουργηθεί μία αράχνη για την συλλογή Gutenberg, η οποία και θα αποθηκεύει όλα τα ηλεκτρονικά αντίτυπα των βιβλίων της συλλογής. Εκτελώντας την εντολή *scrapy genspider gutenberg www.gutenberg.org* δημιουργείται μία boilerplate αράχνη στον κατάλογο spiders/.

Στην βασική της λειτουργία, η αράχνη αυτή θα πρέπει να εντοπίζει όλα τα βιβλία και να δημιουργεί ένα αίτημα για λήψη τους τοπικά. Στην συνέχεια έχουμε την δυνατότητα αποθήκευσης, είτε ως αρχεία *.txt τα οποία θα είναι και συμβατά με την ήδη υπάρχουσα υλοποίηση του *BuildIndex.py*, είτε να τα αποθηκεύσουμε σε βάση δεδομένων. Όπως αναφέρθηκε και παραπάνω, θα ακολουθηθεί η λογική αποθήκευσης σε βάση δεδομένων. Για την επιτυχίας της λειτουργίας αυτής θα πρέπει να γραφτούν τα κατάλληλα item pipelines[10] στο αρχείο *pipelines.py*.

Στο listing 2 παρουσιάζεται η δομή του πίνακα *books*, ο οποίος περιλαμβάνει βασικά πεδία όπως αναγνωριστικό βιβλίου, τίτλο, όνομα συγγραφέα καθώς και το περιεχόμενο του βιβλίου. Για το τελευταίο επιλέχτηκε τύπος δεδομένων *mediumtext* καθώς αυτό μπορεί να είναι μέχρι και μερικά *MB*.

```
CREATE TABLE `books` (
  `id` INT NOT NULL,
  `title` VARCHAR(80) NOT NULL,
  `author` VARCHAR(80),
  `text` MEDIUMTEXT NOT NULL,
  `last_updated` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Listing 2: Δομή πίνακα βιβλίων

Η εύρεση των βιβλίων μπορεί να γίνει από πολλαπλά σημεία του ιστότοπου, μερικά παρέχοντας περισσότερη πληροφορία για τον συγγραφέα και το βιβλίο ενώ άλλα λιγότερες. Βέβαια στην τελευταία περίπτωση υπάρχει το κόστος δημιουργίας περισσότερων αιτημάτων στον εξυπηρετητή (server). Για την υλοποίηση αυτή, θα χρησιμοποιηθεί η διεύθυνση gutenberg.org/ebooks/search με ορίσματα φθίνουσας ταξινόμησης κατά δημοτικότητα βιβλίων.

Συγκεκριμένα, εφόσον οριστεί η παραπάνω διεύθυνση ως η αρχική σελίδα της αράχνης, θα ακολουθηθούν οι υπερσυνδέσεις “Next” για την φόρτωση των επόμενων βιβλίων κ.ο.κ. Για κάθε βιβλίο, δημιουργείται δυναμικά η διεύθυνση λήψης

[10] `https://doc.scrapy.org/en/latest/topics/item-pipeline.html`

του με βάση το αναγνωριστικό του και στην συνέχεια γίνεται ένα αίτημα για λήψη και αποθήκευση στην βάση δεδομένων. Η διαδικασία αυτή ολοκληρώνεται μετά από ορισμένο πλήθος σελίδων που προσπελάθηκαν. Ο ορισμός αυτός γίνεται με το attribute maxpages και ορίζεται με την εκτέλεση της εντολής *scrapy crawl gutenberg -a maxpages=3*, στην οποία οι μέγιστες σελίδες είναι $N = 3$.

```
2017-08-25 07:50:38 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://ww
w.gutenberg.org/cache/epub/2600/pg2600.txt> (referer: http://www.gutenberg.o
rg/ebooks/search/?sort_order=downloads)
2017-08-25 07:50:39 [scrapy.core.scraper] DEBUG: Scraped from <200 http://ww
w.gutenberg.org/cache/epub/2600/pg2600.txt>
Item 2600 updated
2017-08-25 07:50:40 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://ww
w.gutenberg.org/cache/epub/1184/pg1184.txt> (referer: http://www.gutenberg.o
rg/ebooks/search/?sort_order=downloads)
2017-08-25 07:50:40 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://ww
w.gutenberg.org/cache/epub/345/pg345.txt> (referer: http://www.gutenberg.org
/ebooks/search/?sort_order=downloads)
2017-08-25 07:50:40 [scrapy.core.scraper] DEBUG: Scraped from <200 http://ww
w.gutenberg.org/cache/epub/1184/pg1184.txt>
Item 1184 updated
```

Σχήμα 4.10 : Παράδειγμα εκτέλεσης crawler

Στο σχήμα 4.10 παρουσιάζεται ένα τμήμα της εξόδου από την εκτέλεση του crawler. Παρατηρείται πως τα βιβλία εισέρχονται διαδοχικά στην βάση και εκτυπώνεται κατάλληλο μήνυμα με την ολοκλήρωση τους. Επίσης τα download links των κειμένων είναι διαφορετικά από αυτά που προσπαθεί ο crawler να προσπελάσει. Αυτό συμβαίνει, γιατί όλα τα αρχεία δεν έχουν την ίδια δομή ονομασίας, αλλά ο εξυπηρετητής εντοπίζει αυτόματα σε ποιο αρχείο αναφερόμαστε και επιστρέφει αίτηση ανακατεύθυνσης (HTTP status code 302), το οποίο αυτόματα ο *scrapy* ακολουθεί, για να κατεβάσει το αρχείο.

Τέλος στο σχήμα 4.11 παρουσιάζεται η εκτέλεση ενώς ερωτήματος *SQL* στην βάση των βιβλίων. Είναι προφανές πως το αρχείο *BuildIndex.py* μπορεί πλέον να χρησιμοποιεί ερωτήματα προς την βάση για να κατασκευάζει το ευρετήριο, καθώς και να χρησιμοποιεί το πεδίο *last_updated* του πίνακα *books*, έτσι ώστε να ανιχνεύει τυχών αλλαγές και να ανακατασκευάζει το ευρετήριο.

```
mysql> select id, author, title from books;
+-------+------------------------------+-----------------------------------------------------+
| id    | author                       | title                                               |
+-------+------------------------------+-----------------------------------------------------+
|    11 | Lewis Carroll                | Alice's Adventures in Wonderland                    |
|    16 | J. M. Barrie                 | Peter Pan                                           |
|    74 | Mark Twain                   | The Adventures of Tom Sawyer                        |
|    76 | Mark Twain                   | Adventures of Huckleberry Finn                      |
|    84 | Mary Wollstonecraft Shelley  | Frankenstein; Or, The Modern Prometheus             |
|    98 | Charles Dickens              | A Tale of Two Cities                                |
|   135 | Victor Hugo                  | Les Misérables                                      |
|   158 | Jane Austen                  | Emma                                                |
|   174 | Oscar Wilde                  | The Picture of Dorian Gray                          |
|   345 | Bram Stoker                  | Dracula                                             |
|  1184 | Alexandre Dumas              | The Count of Monte Cristo, Illustrated              |
|  1232 | Niccolò Machiavelli          | Il Principe. English                                |
|  1342 | Jane Austen                  | Pride and Prejudice                                 |
|  1400 | Charles Dickens              | Great Expectations                                  |
|  1661 | Arthur Conan Doyle           | The Adventures of Sherlock Holmes                   |
|  1952 | Charlotte Perkins Gilman     | The Yellow Wallpaper                                |
|  2500 | Hermann Hesse                | Siddhartha                                          |
```

Σχήμα 4.11 : Linsting βιβλίων