UNIVERSITY OF PIRAEUS

Ονοματεπώνυμο: Σταμάτιος Ορφανός
Αριθμός Μητρώου: E17113
Μάθημα: Ανάκτηση Πληροφοριών

Περιεχόμενα

Δημιουργία Inverted Index

Parser

Το πρώτο βήμα για την δημιουργία ενός Ανεστραμμένου Ευρετηρίου-Inverted Index είναι να λάβουμε τα δεδομένα πάνω στα οποία θα χτίσουμε το ευρετήριο αυτό. Σε αυτή την εργασία τα δεδομένα προέρχονται από την συλλογή Reuters8Kdata. Ξεκινώντας την κλάση Parser ορίσαμε την μέθοδο parseData, με την οποία διατρέχουμε τα δεδομένα σε όλους τους υποφακέλους έτσι ώστε να εισάγουμε όλα τα κείμενα στο ευρετήριο μας.

```
public static void parseData(File dir, DocIDMapper mapper, InvertedIndex invertedIndex) throws IOExcep
        Parser parser = new Parser();
        File[] files = dir.listFiles();

        for (File file : files) {

            if (file.isDirectory()) {
                parseData(file, mapper, invertedIndex);
            } else if (!file.getName().endsWith(".txt")) {
                continue;
            } else {
                nDocId++;
                ArrayList<String> document = parser.parseFile(file.getCanonicalPath());
                HashMap<String, Long> word_freq_mapper = new HashMap<String, Long>();
             // Iterate through the document and give a frequency to each term in that document
                for (int i=0; i<document.size(); i++) {
                    String term = document.get(i);
                    if (!word_freq_mapper.containsKey(term)) {
                        word_freq_mapper.put(term, 1L);
                    } else {
                        long termFrequency = word_freq_mapper.get(term);
                        word_freq_mapper.put(term, ++termFrequency);
                    }
                }
                // Insert the terms of the document with each frequency
                Set<String> stringDoc = word_freq_mapper.keySet();
                Iterator<String> iterator = stringDoc.iterator();
                while(iterator.hasNext()) {
                    String term = iterator.next();
                    long termFrequency = word_freq_mapper.get(term);
                    invertedIndex.insert(term, nDocId, termFrequency);
                }
                mapper.add(nDocId, file.getPath());
            }
        }
}
```

Ξεκινώντας με αυτή την μέθοδο αρχικοποιούμε ένα αντικείμενο Parser και ένα αντικείμενο File[] για να διατρέξουμε τα δεδομένα μας. Τα πρώτα βήματα του βρόχου είναι να ελέγξουμε αν το File object που εξετάζουμε σε κάθε στιγμή είναι είτε directory, όπου θα καλέσουμε αναδρομικά την συνάρτηση για να διατρέξουμε τα περιεχόμενα του είτε ένα File object που δεν έχει την κατάληξη .txt, το οποίο δεν μας προσφέρει δεδομένα για το στόχο μας.

Σε περίπτωση όπου το File object είναι όντως ένα κείμενο τότε σε αυτή την περίπτωση αυξάνουμε το document id και ξεκινάμε την διαδικασία για την εισαγωγή του κειμένου στο ευρετήριο. Το πρώτο βήμα είναι να φτιάξουμε ένα HashMap όπου για κάθε term του κειμένου κάνουμε ένα βασικό έλεγχο. Αν το term δεν υπάρχει στο HashMap τότε το εισάγουμε στο HashMap με frequency 1, ενώ σε περίπτωση που το term υπάρχει το frequency του αυξάνεται κατά ένα και εισάγεται στο HashMap ως νέο στοιχείο.

Με την ολοκλήρωση της παραπάνω διαδικασίας παίρνουμε τις τελικές τιμές του κάθε ζευγαριού term-frequency με την εντολή .keySet() και χρησιμοποιώντας έναν Iterator διατρέχομε τις τιμές αυτές για να τις προσθέσουμε στο Inverted Index. Το τελευταίο βήμα για την ολοκλήρωση της μεθόδου είναι να κάνουμε map-αντιστοιχήσουμε το κείμενο και το id που έχει πάρει με το αντίστοιχο κείμενο στην συλλογή μας, δηλαδή το path του κειμένου τοπικά.

```
public static ArrayList<String> parseFile(String path) throws IOException {
    BufferedReader bfrReader = new BufferedReader(new FileReader(path));
    StringBuilder strBuilder = new StringBuilder();
    String setLine = "";
    while((setLine = bfrReader.readLine()) != null) {
        strBuilder.append(setLine).append(" ");
    }
    bfrReader.close();
    return tokenizer(strBuilder.toString());
}
```

Όπως μπορούμε να καταλάβουμε η μέθοδος parseData, συμπεριλαμβάνει την λειτουργία της συνάρτησης parseFile, όπου διαβάζουμε ένα File με βάση ένα path. Με το κείμενο στην διάθεση μας δημιουργούμε ένα StringBuilder object, το οποίο μας επιτρέπει να προσθέτουμε String δεδομένα χωρίς να κάνουμε overwrite τα προηγούμενα. Με αυτή την λογική στο strBuilder object προσθέτουμε κάθε γραμμή του κειμένου κάθε φορά. Στο τέλος μετατρέπουμε το StringBuilder σε String και το χρησιμοποιούμε ως παράμετρο για την μέθοδο tokenizer έτσι ώστε στο τέλος του parseFile τα δεδομένα που επιστρέφουμε να είναι tokens.

Παραπάνω αναφερθήκαμε στην συνάρτηση tokenizer, η οποία παίρνει ως παράμετρο ένα String object και μας επιστρέφει ένα ArrayList<String>, μετατρέποντας ουσιαστικά το αρχικό κείμενο σε tokens. Σε αυτό το σημείο είναι σημαντικό να αναφερθεί η διαδικασία μετατροπής του κειμένου σε tokens.

```
public static String DELIMITER = " ,.!:;'<>()@*&%${}#\"\t\n";
public static String[] STOPWORDS = {"a", "about", "above", "after", "again"}  *1
```

```
public static ArrayList<String> tokenizer(String line) {
        ArrayList<String> tokens = new ArrayList<String>();
        StringTokenizer strTokenizer = new StringTokenizer(line, DELIMITER);
        String sToken = "";

        while (strTokenizer.hasMoreTokens()){
            sToken = strTokenizer.nextToken();
            sToken = sToken.toLowerCase();
            if (Arrays.stream(STOPWORDS).anyMatch(sToken::equals)) {
                continue;
            } else {
                tokens.add(sToken);
            }
        }
        return tokens;
    }
```

Για την δημιουργία της tokenize μεθόδου αρχικοποιούμε ένα StringTokenizer object έτσι ώστε να αφαιρέσουμε τα γνωστά Delimiters από το κείμενο μας, καθώς δεν προσφέρουν κάποιο σημασιολογικό περιεχόμενο. Επίσης αρχικοποιούμε ένα ArrayList<String> με στο οποίο θα αποθηκεύσουμε τα tokens.

Ξεκινώντας με τον βρόχο μετατρέπουμε κάθε term του strTokenizer σε πεζά έτσι ώστε να αποφύγουμε την εισαγωγή του ίδου σημασιολογικά όρου με δύο διαφορετικές λέξεις στο dictionary του Inverted Index (π.χ. What και what). Στην συνέχεια του while βρόχου έχουμε πλέον ένα κείμενο σε ένα String object χωρίς κάποιο annotation και με πεζά γράμματα, δηλαδή μια πρόταση με terms που μπορούμε να εισάγουμε στο Inverted Index. Ωστόσο για την κατασκευή ενός αποδοτικότερου Inverted Index είναι αναγκαία η χρήση λίστας με Διακόπτουσες Λέξεις-Stopwords στον αλγόριθμο έτσι ώστε να μην αυξάνεται το Ανεστραμμένο Ευρετήριο με λέξεις μικρής σημασιολογικής σημασίας.

*1 Προφανώς οι stopwords είναι περισσότερες, απλά για λόγους εύκολης ανάγνωσης του εγγράφου έχουμε κάποιες ενδεικτικές λέξεις.

Inverted Index

Για την δημιουργία ενός Inverted Index,δημιουργήθηκε η αντίστοιχη κλάση με όλες τις απαραίτητες λειτουργίες. Πιο συγκεκριμένα ξεκινάμε με το γεγονός ότι το Inverted Index έχει υλοποιηθεί με ένα HashMap, δηλαδή ένα πίνακα κατακερματισμού. Το HashMap αποτελείται από ένα String, δηλαδή ένα term του vocabulary και ένα ArrayList<Posting>, δηλαδή μια λίστα με όλα τα κείμενα στα οποία εμφανίζεται και την συχνότητα εμφάνισης. Στην συνέχεια θα αναφερθούμε με περισσότερη λεπτομέρεια για τα Postings, αλλά προς το παρόν τα θεωρούμε ως ένα object που περιέχει το docId και την συχνότητα εμφάνισης.

```
public class InvertedIndex implements Serializable {
    private HashMap<String, ArrayList<Posting>> hashMap;


    public InvertedIndex() {
        hashMap = new HashMap<>();
    }
}
```

Ένα από τα πιο σημαντικά μέρη της δημιουργίας του Inverted Index είναι η εισαγωγή των όρων σε αυτό. Προφανώς η συνάρτηση insert καλύπτει όλες τις ανάγκες εισαγωγής όρων στο ευρετήριο. Πιο συγκεκριμένα σε περίπτωση όπου ένα term υπάρχει ήδη στο ευρετήριο τότε βρίσκουμε και επιστρέφουμε την εγγραφή αυτή και προσθέτουμε στο ArrayList<Posting> το καινούργιο Posting με το αντίστοιχο docId και term frequency σε αυτό το document. Αντίθετα σε περίπτωση όπου το term συναντάται για πρώτη φορά δημιουργούμε ένα νέο ArrayList<Posting> με το docId και το term frequency και προσθέτουμε τελικά στο hashMap το term με το Posting του.

```
    public void insert(String term, int nDocID, long frequency) {
        if(hashMap.containsKey(term)) {
            ArrayList<Posting> postings = hashMap.get(term);
            postings.add(new Posting(nDocID, frequency));
        }
        else {
            ArrayList<Posting> postings = new ArrayList<>();
            Posting posting = new Posting(nDocID, frequency);
            postings.add(posting);
            hashMap.put(term, postings);
        }
    }
```

Παρακάτω έχουμε δύο βασικές συναρτήσεις για την επιστροφή Posting lists με βάση κάποιο term ή ακόμα και την επιστροφή όλων των Posting lists. Στην συνάρτηση get με παράμετρο ένα term επιστρέφουμε το posting list αυτού του term από το Inverted Index. Αντίθετα στην περίπτωση όπου καλούμε την συνάρτηση get χωρίς κάποια παράμετρο τότε επιστρέφεται όλη η συλλογή posting lists όλων των term του λεξιλογίου του ευρετηρίου.

```
public ArrayList<Posting> get(String term) {
    return hashMap.get(term);
}

public Collection<ArrayList<Posting>> get() {
    return hashMap.values();
}
```

Αυξάνοντας περαιτέρω την λειτουργικότητα του ευρετηρίου, ορίσαμε και μια print συνάρτηση για την εκτύπωση του Inverted Index, εκτυπώνοντας κάθε term με τα Posting lists του.

```
public void print() {
    Set<String> keys = hashMap.keySet();
    Iterator<String> it = keys.iterator();

    while(it.hasNext()) {
        String term = it.next();
        ArrayList<Posting> postings = hashMap.get(term);

        System.out.println(term + " (" + postings.size() + "):");

        for(Posting posting : postings) {
            System.out.println("\t" + posting.toString());
        }
    }
}
```

Ένα από τα πιο σημαντικά χαρακτηριστικά μιας αρχικής μηχανής αναζήτησης είναι η αποθήκευση του ευρετηρίου έτσι ώστε να μην χρειάζεται η κατασκευή του κάθε φορά που χρησιμοποιεί κανείς την εφαρμογή. Προφανώς η παραπάνω διαδικασία για το parsing των αρχείων και την δημιουργία του ευρετηρίου είναι υπολογιστικά ακριβή, γεγονός που θα επιβάρυνε την απόδοση της μηχανής αναζήτησης. Με βάση τα παραπάνω δεδομένα ήταν απαραίτητη η αποθήκευση και α ανάκτηση του ευρετηρίου.

Ξεκινώντας με την συνάρτηση write για την αποθήκευση του ευρετηρίου, όπου σαν παράμετρο έχουμε το όνομα και το path του αρχείου, που θα περιλαμβάνει το ευρετήριο. Για την αποθήκευση του ευρετηρίου χρησιμοποιήθηκε το binary file format με κατάληξη .ser .

```
    public void write(String sFile) throws IOException{
        FileOutputStream fileOutput = new FileOutputStream(sFile, true);
        ObjectOutputStream objectOuput = new ObjectOutputStream(fileOutput);

        objectOuput.writeObject(hashMap);

        objectOuput.close();
        fileOutput.close();
    }
```

Ομοίως έχουμε και την συνάρτηση load, η οποία ανακτά τα περιεχόμενα του Inverted Index με βάση το path και το όνομα του ευρετηρίου με στόχο την αποδοτικότερη αναζήτηση.

```
public void load(String path) throws IOException, ClassNotFoundException {
    FileInputStream inputFile = new FileInputStream(path);
    ObjectInputStream objectInput = new ObjectInputStream(inputFile);

    hashMap = (HashMap<String, ArrayList<Posting>>) objectInput.readObject();

    objectInput.close();
    inputFile.close();
}
```

Posting

Σε κάθε Ανεστραμμένο Ευρετήριο το term περιλαμβάνει μια λίστα με τα κείμενα στα οποία συναντάται. Στην συγκεκριμένη περίπτωση μαζί με το id του κειμένου περιλαμβάνεται και η συχνότητα εμφάνισης του συγκεκριμένου term στο κείμενο αυτό.

Με την κλάση Posting έχουμε μοντελοποιήσει την παραπάνω λειτουργικότητα, καθώς αποθηκεύουμε τόσο το docId(nDocId) και το frequency του όρου. Η συνάρτηση toString μας επιτρέπει να εκτυπώνουμε τα δεδομένα του Posting με μια καλύτερη μορφή.

```
public class Posting implements Serializable {
    private int nDocId;
    private long frequency;

    public Posting(int nDocId, long frequency){
        this.nDocId = nDocId;
        this.frequency = frequency;
    }

    public int getnDocId() {
        return nDocId;
    }

    public long getFrequency() {
        return frequency;
    }

    @Override
    public String toString() {
        return "[docId:"+ nDocId + ", frequency: " + frequency + "] ";
    }
}
```

DocIdMapper

Όπως αναφέραμε παραπάνω είναι πολύ σημαντικό να μπορούμε να έχουμε την δυνατότητα αποθήκευσης και να ανάκτησης των δεδομένων του Inverted Index χωρίς την επαναλαμβανόμενη κατασκευή του. Δυστυχώς όμως το Inverted Index μόνο του δεν μας είναι ιδιαίτερα χρήσιμο καθώς δεν γνωρίζουμε το mapping-αντιστοιχία μεταξύ του docId και του text file, από το οποίο παρήχθει.

Με την αρχικοποίηση ενός DocIdMapper object δημιουργούμε ένα πίνακα από n String με τα path προς τα text files. Η συνάρτηση add προσθέτει στον πίνακα το path του αρχείου στο κελί pos (position). Η κλήση αυτής της συνάρτησης γίνεται στην συνάρτηση parseData, που είδαμε παραπάνω αμέσως μετά την εισαγωγή των λέξεων του text file στο Inverted index.

```
public class DocIDMapper {
   private String[] map;


   public DocIDMapper(int size) {
       map = new String[size];
   }

   public void add(int pos, String filename) {
       map[pos] = filename;
   }



   public String get(int pos) {
       return map[pos];
   }
```

Με την συνάρτηση get() επιστρέφουμε το path του αρχείου με docId ίσο με το pos. Η συνάρτηση αυτή χρησιμοποιείται κυρίως στα τελευταία βήματα της διαδικασίας αναζήτησης, αφού έχουμε βρει τα docId's ως αποτέλεσμα.

Ακολουθώντας την λογική αποθήκευσης και ανάκτησης για το Inverted Index, ορίστηκαν οι συναρτήσεις write and load για να πετύχουμε τον στόχο γρηγορότερης αναζήτησης .

Ξεκινώντας με την συνάρτηση write για την αποθήκευση του mapper, όπου σαν παράμετρο έχουμε το όνομα και το path του αρχείου, που θα περιλαμβάνει το ευρετήριο. Για την αποθήκευση του ευρετηρίου χρησιμοποιήθηκε το binary file format με κατάληξη .ser .

```
// Write the mapper file, so that we can use it after we made the inverted index
public void write(String mapperFile) throws IOException{
    FileOutputStream fileOutput = new FileOutputStream(mapperFile, true);
    ObjectOutputStream objectOuput = new ObjectOutputStream(fileOutput);

    objectOuput.writeObject(map);

    objectOuput.close();
    fileOutput.close();
}
```

Ομοίως έχουμε και την συνάρτηση load, η οποία ανακτά τα περιεχόμενα του mapper με βάση το path και το όνομα του mapper με στόχο την αποδοτικότερη αναζήτηση.

```
// Load the mapper file, so that we can use search functions after we have made the
inverted index
public void load(String path) throws IOException, ClassNotFoundException {
    FileInputStream inputFile = new FileInputStream(path);
    ObjectInputStream objectInput = new ObjectInputStream(inputFile);

    map = (String[]) objectInput.readObject();

    objectInput.close();
    inputFile.close();
}
```

Query Parser

Query Parser

Για την κατασκευή ενός Query Parser είναι απαραίτητη μια διαδικασία προεπεξεργασίας του query που εισάγει ο χρήστης. Για την επίτευξη αυτού του στόχου έχουμε την συνάρτηση queryTokenizer όπου μετατρέπουμε το query σε μια μορφή που μπορεί να αναγνωρίσει ο queryParser.

Αρχικά ορίζουμε ένα StringBuilder object με το query, αφαιρώντας παράλληλα και τα σημεία στίξης. Στην συνέχεια για κάθε token του query ελέγχουμε αν ανήκει σε μια από τις κατηγορίες operator έτσι ώστε να κάνουμε το κατάλληλο mapping για να μειώσουμε την πολυπλοκότητα του queryParser. Πιο συγκεκριμένα αν ένας χρήστης εισάγει ένα query με το “&” ως λογικό operator τότε η συνάρτηση query tokenizer θα το μετατρέψει σε “AND” μειώνοντας τους operators, που πρέπει να ξέρει ο queryParser.

```
public static String DELIMITER = " ,.:;'<>()@*%${}#\"\t\n";
public static String[] ANDOPERATORS = {"&", "&&", "AND"};
public static String[] OROPERATORS = {"|", "||", "OR"};
public static String[] NOTOPERATORS = {"-", "--", "~", "NOT"};
public static String[] STOPWORDS = Parser.STOPWORDS;


public static ArrayList<String> queryTokenizer(String query) {
    ArrayList<String> tokens = new ArrayList<String>();
    StringTokenizer strTokenizer = new StringTokenizer(query, DELIMITER);
    String sToken = "";
    while (strTokenizer.hasMoreTokens()){
        sToken = strTokenizer.nextToken();
        if (Arrays.stream(ANDOPERATORS).anyMatch(sToken::equals)) {
            sToken = "AND";
            tokens.add(sToken);
        }
        else if (Arrays.stream(OROPERATORS).anyMatch(sToken::equals)) {
            sToken = "OR";
            tokens.add(sToken);
        }
        else if (Arrays.stream(NOTOPERATORS).anyMatch(sToken::equals)) {
            sToken = "NOT";
            tokens.add(sToken);
        }
        else {
            sToken = sToken.toLowerCase();
            if (Arrays.stream(STOPWORDS).anyMatch(sToken::equals)) { continue; }
            tokens.add(sToken);
        }
    }
    return tokens;}
```

Ο επόμενος και τελευταίος έλεγχος που υλοποιείται από την queryTokenizer συνάρτηση είναι ο έλεγχος για Stopwords. Ο έλεγχος αυτός είναι απαραίτητος καθώς κατά την διάρκεια του parsing των δεδομένων δεν συμπεριλάβαμε τα stopwords στο Inverted Index και έτσι οποιοδήποτε query που περιλαμβάνει ένα stopword δεν θα επιστρέψει το επιθυμητό αποτέλεσμα. Μια σημαντική παρατήρηση είναι το γεγονός ότι χρησιμοποιούμε ακριβώς την ίδια λίστα με stopwords, που χρησιμοποιήθηκε και στο parsing των δεδομένων. Μια τελική σημείωση είναι ότι μετατρέπουμε όλα τα tokens σε πεζά γράμματα, εφόσον έχουμε κάνει όλους για operators και delimiter.

Ξεκινώντας με την συνάρτηση chooseQueryType αρχικοποιούμε ένα ArrayList<String> με τα tokens, αποτέλεσμα της συνάρτησης queryTokenizer. Με βάση αυτά τα tokens θα ελέγξουμε αν έχουμε ένα Free-text query ή ένα Boolean query. Για τον έλεγχο αυτό χρησιμοποιούμε την μεταβλητή operatorExists, η οποία παίρνει τιμή True όταν υπάρχει κάποιος λογικός operator στο query ενώ παίρνει την τιμή False αν δεν υπάρχει κάποιος operator. Για να βρούμε αν υπάρχει κάποιος operator απλά διατρέχουμε τα tokens και αν συναντήσουμε κάποιο από τα “AND”, “OR”, “NOT” τότε έχουμε ένα Boolean query. Σε αυτό το σημείο φαίνεται η χρησιμότητα του queryTokenizer καθώς ο έλεγχος γίνεται με μια μόνο String τιμή αντί όλων των πιθανών operator μειώνοντας την πολυπλοκότητα αυτής της συνάρτησης.

Ο παραπάνω έλεγχος όμως δεν μπορεί να καλύψει όλες τις περιπτώσεις Free-text και Boolean query. Κατά την διάρκεια πειραματικών δοκιμών query παρατηρήσαμε ότι query του τύπου “*term1 AND/OR term2*” παρουσίασαν προβλήματα με βάση την υπόλοιπη λειτουργικότητα της εφαρμογής. Για να αντιμετωπίσουμε αυτό το πρόβλημα δημιουργήσαμε Boolean queries δύο επιπέδων παράλληλα με το Free-text query.

Ξεκινώντας με την πρώτη περίπτωση όπου το query περιλαμβάνει κάποιο λογικό operator και έχει μήκος μεγαλύτερο από 4 tokens, δηλαδή έχουμε τουλάχιστον 3 terms για τα οποία θέλουμε να κάνουμε ένα search query. Σε αυτή την περίπτωση θα καλέσουμε την συνάρτηση BooleanQueryParser για περαιτέρω ανάλυση του query.

Συνεχίζοντας με την δεύτερη περίπτωση όπου έχουμε operator αλλά ένα query με μικρό μήκος με δύο ακριβώς terms. Όπως μπορούμε να δούμε παρακάτω παίρνουμε τον operator στον αρχικό βρόχο και στην συνέχεια ανάλογα με την τιμή του καλούμε το αντίστοιχο search query.

Στην τρίτη και τελευταία περίπτωση όπου έχουμε free-text query καλούμε την συνάρτηση A_AND_B, καθώς υποθέτουμε ότι σε ένα query free-text θέλουμε να πάρουμε τα κοινά αποτελέσματα των term. Για παράδειγμα το query “family car” είναι ένα free-text query, για το οποίο θα θέλαμε να επιστρέψουμε τα αποτελέσματα για “οικογενειακά” αυτοκίνητα και όχι αποτελέσματα για “οικογενειακά” αυτοκίνητα και για αυτοκίνητα και οικογένεια γενικότερα.

Στο τέλος της συνάρτησης ταξινομούμε τα Postings με βάση το term frequency του term στο κάθε document και αμέσως μετά επιστρέφουμε το αποτέλεσμα.

```
public static ArrayList<Posting> chooseQueryType(String query, InvertedIndex invertedIndex) {
   boolean operatorExists = false;
   String operator = "";
   ArrayList<String> tokens = queryTokenizer(query);
   ArrayList<Posting> result = new ArrayList<>();

   for (String token : tokens) {
       if (token.equals("AND") || token.equals("OR") || token.equals("NOT")) {
            operator=token;
            operatorExists = true;
            break;
       }
   }

   if (operatorExists && tokens.size() > 4) {
      result = BooleanQueryParser.booleanQuery(query, invertedIndex);
   }
   else if (operatorExists && !operator.equals("")) {
      if (operator.equals("AND")) {
        result= Search.AND_Boolean(invertedIndex.get(tokens.get(0)),invertedIndex.get(tokens.get(2)),
invertedIndex);
      }
      else if (operator.equals("OR")) {
        result= Search.OR_Boolean(invertedIndex.get(tokens.get(0)),invertedIndex.get(tokens.get(2)),
invertedIndex);
      }
      else {
        result = Search.NOT_A(invertedIndex.get(tokens.get(0)), invertedIndex);
      }
   }
   else {
       result = Search.A_AND_B(tokens, invertedIndex);
   }

   // Rank the results we get from the search queries
   ArrayList<Posting> finalResult = RankedSearch.rankResults(result);
   return finalResult;
}
```

Boolean Query Parser

Για την διαχείριση των boolean queries ήταν αναγκαία η δημιουργία μιας ξεχωριστής κλάσης, καθώς η πολυπλοκότητα ενός boolean query μπορεί να αυξηθεί αρκετά. Δεδομένου ότι το query του χρήστη είναι boolean query, καλούμε την συνάρτηση booleanQuery έτσι ώστε να βρούμε το είδος query και τον τρόπο εκτέλεσης του.

Αρχικά η συνάρτηση λαμβάνει το query του χρήστη, το οποίο κάνει split με βάση το κενό μόνο και εισάγει κάθε ένα από τα tokens του στον String πίνακα tokens. Στην συνέχεια διατρέχουμε αυτόν τον πίνακα έτσι ώστε να βρούμε τους logical operators. Κάθε φορά όπου συναντάμε ένα logical operator, καλούμε την αντίστοιχη συνάρτηση parse για να βρούμε το αποτέλεσμα. Σημαντικό είναι το γεγονός ότι η συνάρτηση αυτή λειτουργεί σχεδόν αναδρομικά καθώς για κάθε καινούργιο operator που βρίσκει στο query συγκρίνει τα Postings που ήδη έχει λάβει με τα καινούργια Postings του νέου operator.

Οι παράμετροι της κάθε parse συνάρτησης είναι ιδιαίτερα σημαντικοί, διότι μας δίνουν την θέση/index του operator (παράμετρος i) στο query έτσι ώστε να πάρουμε τα terms που βρίσκονται αμέσως πριν και μετά από αυτό(πίνακας tokens). Όπως αναφέραμε και νωρίτερα χρησιμοποιούμε το finalPostings σχεδόν αναδρομικά και το InvertedIndex object για να μπορούμε να επιστρέφουμε τα Postings για κάθε term.

```
public static ArrayList<Posting> booleanQuery(String query, InvertedIndex invertedIndex) {
        ArrayList<Posting> finalPostings = new ArrayList<>();

        String[] tokens = query.split(" ");

        for (int i=0; i<tokens.length; i++) {
            String token = tokens[i];

            if (token.equals("AND")) {
                finalPostings = parseANDQuery(finalPostings, invertedIndex, tokens, i);
            }
            else if (token.equals("OR")) {
                finalPostings = parseORQuery(finalPostings, invertedIndex, tokens, i);
            }
            else if (token.equals("NOT")) {
                finalPostings = parseNOTQuery(finalPostings, invertedIndex, tokens, i);
            }
        }
        return finalPostings;
    }
```

Για να γίνει πιο κατανοητή η λειτουργία του booleanQuery, θα αναφερθούμε σε ένα πολύπλοκο παράδειγμα βήμα προς βήμα. Έστω ότι έχουμε το query “Computer Terminal Systems AND sale OR warrants”.

Βήμα 1:
Μετατρέπουμε το query σε ένα πίνακα από String, έχοντας το παρακάτω αποτέλεσμα:

computer	terminal	systems	AND	sale	OR	warrants

Βήμα 2:
Διατρέχουμε τον πίνακα αυτό σε ένα for loop, μέχρι να βρούμε τον πρώτο operator:

computer	terminal	systems	AND	sale	OR	warrants

Βρίσκοντας τον πρώτο operator παίρνουμε όλα τα tokens του πίνακα tokens μέχρι και το AND και όλα τα επόμενα μέχρι τον επόμενο operator ή το τέλος του query. Στην προκειμένη περίπτωση έχουμε νέο operator άρα παίρνουμε το ακριβώς επόμενο term. Αφού εκτελέσουμε το query και ενημερώσουμε το finalPostings συνεχίζουμε με τον επόμενο operator. Οι λεπτομέρειες εκτέλεσης του parseAndQuery αναλύονται παρακάτω.

Βήμα 3:
Βρίσκουμε τον επόμενο operator και τα terms για τα οποία χρησιμοποιείται. Στην προκειμένη περίπτωση έχουμε τον operator OR, με ένα term ακριβώς πριν και μετά, αφού τελειώνει και το query. Φυσικά όλα τα προηγούμενα terms και operators δεν χρησιμοποιούνται αλλά υπάρχουν ακόμα στον αλγόριθμο.

computer	terminal	systems	AND	sale	OR	warrants

Χρησιμοποιώντας την συνάρτηση parseOr παίρνουμε τα posting, που περιλαμβάνουν τις λέξεις sale ή warrants.

Βήμα 4:
Σε αυτό το σημείο βρίσκουμε τα κοινά Postings άρα και τα κοινά έγγραφα των παραπάνω δύο βημάτων, δίνοντας το τελικό αποτέλεσμα. Προφανώς η παραπάνω διαδικασία μπορεί να επεκταθεί για πολλαπλούς operators.

Ξεκινώντας αυτή την συνάρτηση ελέγχουμε αν το finalPostings είναι κενό έτσι ώστε να εισάγουμε τα Postings των πρώτων terms ή του πρώτου term. Έτσι αρχικοποιούμε ένα ArrayList<String> για τα tokens/terms πριν τον logical operator. Για παράδειγμα το ArrayList αυτό για το query “Computer Terminal Systems AND sale OR warrants” θα είναι {“computer”, “terminal”, systems“}. Σε άλλες περιπτώσεις φυσικά το ArrayList θα μπορούσε να αποτελείται από ένα μόνο token/term, χωρίς να έχουμε κάποιο πρόβλημα με τα postings που επιστρέφουμε χάρη στην συνάρτηση setTokens, που θα αναλύσουμε παρακάτω.

```
private static ArrayList<Posting> parseANDQuery(ArrayList<Posting> finalPostings, InvertedIndex
invertedIndex, String[] tokens, int index){
       ArrayList<String> postOperator = new ArrayList<>();
       ArrayList<Posting> andList = new ArrayList<Posting>();

       if (finalPostings.size() == 0) {
           ArrayList<String> initialTokens = new ArrayList<String>();
           for (int i=0; i<index; i++) {
               initialTokens.add(tokens[i]);
           }
           finalPostings = setTokens(initialTokens, invertedIndex);
       }

       for (int i=index + 1; i<tokens.length; i++) {
           String token = tokens[i];
           if (token.equals("AND") || token.equals("OR") || token.equals("NOT")) {
               if (token.equals("NOT") && index == i-1) {
                   andList = parseNOTQuery(finalPostings, invertedIndex, tokens, i);
               }
               break;
           }
           else { postOperator.add(token); }
       }
       andList = setTokens(postOperatorString, invertedIndex);
       return Search.AND_Boolean(finalPostings, andList, invertedIndex);
   }
```

Σε αυτό το σημείο έχουμε Posting δεδομένα στο finalPostings, άρα μπορούμε να χρησιμοποιήσουμε το finalPostings ArrayList στη συνέχεια. Αφού βρήκαμε όλους τους όρους πριν από τον operator θα βρούμε και όλα τα terms μετά τον operator. Ξεκινώντας αμέσως μετά το index/θέση του operator στον πίνακα tokens μέχρι και το τέλος του πίνακα ή μέχρι να βρούμε κάποιον άλλο operator προσθέτουμε όλα τα terms/tokens στο ArrayList postOperator. Αφού βρούμε όλα τα terms μετά τον operator τότε καλούμε την setTokens συνάρτηση για να επιστρέψουμε τα κατάλληλα Postings στο ArrayList andList.

Ένας ακόμα βασικός έλεγχος αφορά τα query του τύπου “term1 AND NOT term2” όπου καλούμε άμεσα την συνάρτηση parseNotQuery για να πάρουμε το αποτέλεσμα.

Στο τέλος θα πάρουμε το finalPostings με τα Postings πριν τον operator και το andList με τα Postings μετά τον operator και θα πάρουμε τα κοινά Postings αυτών των δύο ArrayList με το return statement.

Όπως μπορούμε να καταλάβουμε η συνάρτηση setTokens είναι ιδιαίτερα σημαντική για όλες τις συναρτήσεις, καθώς μας δίνει την δυνατότητα να προσεγγίζουμε δυναμικά κάθε query. Πιο συγκεκριμένα μας δίνει τις εξής δύο επιλογές:

1. Σε περίπτωση όπου έχουμε ένα μόνο token πριν ή μετά τον operator τότε επιστρέφουμε τα Postings αυτού του term με την μέθοδο get του invertedIndex object.
2. Σε περίπτωση όπου έχουμε πολλά tokens, τότε αντιμετωπίζουμε τα tokens αυτά ως ένα free-text query επιστρέφοντας τα κοινά Postings με το A_AND_B query.

```
private static ArrayList<Posting> setTokens(ArrayList<String>tokens,InvertedIndex invertedIndex) {
    ArrayList<Posting> result = new ArrayList<>();
    if (tokens.size() <=1 ) {
        result = invertedIndex.get(tokens.get(0));
    }
    else {
        result = Search.A_AND_B(tokens, invertedIndex);
    }
    return result;
}
```

Για καλύτερη κατανόηση της συνάρτησης έχουμε το γνωστό παράδειγμα: “Computer Terminal Systems AND sale OR warrants”.

computer	terminal	systems	AND	sale	OR	warrants

-> initialTokens = {“computer”, “terminal”, “systems”}
-> finalPostings = A_AND_B(initialTokens) // πολλά tokens
-> postOperator = {“sale”}
-> andList = invertedIndex.get(sale)

--> finalPostings = AND_Boolean(finalPostings, andList)

Ομοίως λειτουργούν οι μέθοδοι parseOrQuery και parseNotQuery με τις μόνες διαφορές το return statement, το οποίο θα γίνει OR_Boolean ή Not_Boolean.

Queries

Search Queries

Για την ικανοποίηση όλων των αναγκών της εργασίας έπρεπε να εμπλουτιστεί σημαντικά ο κώδικας για την υποστήριξη free-text και boolean queries. Ένας από τους πιο σημαντικούς στόχους για αυτή την εργασία ήταν η υποστήριξη ερωτημάτων με πολλά terms. Δεδομένου αυτού του στόχου κρίθηκε απαραίτητη η δημιουργία μιας γενικότερης συνάρτησης Search, για την διαχείριση του ερωτήματος.

Ξεκινώντας με την συνάρτηση A_AND_B, στην οποία ως παραμέτρους παίρνουμε ένα ArrayList με τα query tokens και φυσικά το InvertedIndex object, με το οποίο θα κάνουμε την αναζήτηση. Δεδομένου ότι θα έχουμε τουλάχιστον δύο tokens, παίρνουμε τις Posting lists των δύο πρώτων token, βρίσκοντας τα κοινά documents και τα προσθέτουμε στο result. Σε περίπτωση όπου έχουμε παραπάνω από δύο terms τότε συγκρίνουμε το result με τα Postings του τρίτου και συνεχίζουμε με την ίδια λογική έως ότου συμπεριλάβουμε όλα τα tokens.

Άρα όπως μπορούμε να καταλάβουμε η συνάρτηση A_AND_B, είναι η βοηθητική συνάρτηση, που διαχειριζεται το query που εισάγει ο χρήστης.

```
public static ArrayList<Posting> A_AND_B(ArrayList<String> queryTokens, InvertedIndex invertedIndex){
       ArrayList<Posting> result = new ArrayList<>();
       if (queryTokens.size() < 2) {
           System.out.println("In order to make a query we need at least two arguments");
       }
       ArrayList<Posting> firtsPosting = invertedIndex.get(queryTokens.get(0));
       ArrayList<Posting> secondPosting = invertedIndex.get(queryTokens.get(1));
       result = AND_Boolean(firtsPosting, secondPosting , invertedIndex);
       for (int i=2; i < queryTokens.size(); i++) {
           ArrayList<Posting> morePostings = invertedIndex.get(queryTokens.get(i));
           result = AND_Boolean(result, morePostings, invertedIndex);
       }
       return result;
   }
```

Συνεπώς η συνάρτηση για την εύρεση των κοινών εγγράφων μεταξύ δύο tokens είναι η συνάρτηση AND_Boolean, η οποία παίρνει ως παραμέτρους τα Posting lists των δύο παραμέτρων που εξετάζουν κάθε φορά. Η συνάρτηση αυτή προφανώς είναι αρκετά απλή καθώς μετατρέπουμε τα ArrayLists σε πίνακες, τους οποίους και διατρέχουμε παράλληλα σε χρόνο O(x+y) κρατώντας όλα τα document id's που βρίσκονται και στους δύο πίνακες.

```
public static ArrayList<Posting> AND_Boolean(ArrayList<Posting> aPost, ArrayList<Posting> bPost,
InvertedIndex invertedIndex) {
       Posting[] aArray = aPost.toArray(new Posting[0]);
       Posting[] bArray = bPost.toArray(new Posting[0]);
       int sizeA = aArray.length;
       int sizeB = bArray.length;
       ArrayList<Posting> result = new ArrayList<>();
       int aIndex = 0, bIndex = 0;

       while(aIndex < sizeA && bIndex < sizeB) {
           if (aArray[aIndex].getnDocId() < bArray[bIndex].getnDocId()) {
               aIndex++;
           }
           else if (aArray[aIndex].getnDocId() > bArray[bIndex].getnDocId()) {
               bIndex++;
           }
           else {
               result.add(aArray[aIndex]);
               aIndex++;
               bIndex++;
           }
       }
       return result;
   }
```

Με τον ίδιο τρόπο έχουν αναπτυχθεί και οι συναρτήσεις A_OR_B και OR_Boolean για την κάλυψη των or queries, που μπορεί να εισάγει ένας χρήστης. Η μόνη διαφορά όπως μπορούμε να καταλάβουμε βρίσκεται στο βρόχο με την σύγκριση των στοιχείων των δύο πινάκων. Επιπλέον αναπτύχθηκαν οι συναρτήσεις για ερωτήματα A_NOT_B και AND_NOT_Boolean, οι οποίες συναρτήσεις δεν χρησιμοποιήθηκαν κάπου δυστυχώς.

Η επόμενη και τελευταία συνάρτηση για τα queries είναι η NOT_A, η οποία έχει ως παραμέτρους ένα ArrayList από Postings για το term που μας ενδιαφέρει και το InvertedIndex για να φέρουμε τα απαραίτητα αποτελέσματα.

Αρχικά επιστρέφουμε όλα τα postings του InvertedIndex και τα αποθηκεύουμε σε ένα Collection από ArrayList τύπου Posting. Στην συνέχεια μετατρέπουμε το Collection σε ένα ArrayList του παραπάνω τύπου έτσι ώστε να μπορούμε να συγκρίνουμε τα δύο collections και να αφαιρέσουμε όλα τα κοινά Postings.

Μέχρι αυτό το σημείο όμως έχουμε ένα ArrayList<ArrayList<Posting>>, το οποίο δημιουργεί compatibility προβλήματα με όλες τις υπόλοιπες συναρτήσεις. Για να αντιμετωπίσουμε αυτό το πρόβλημα χρησιμοποιήθηκε ένας βρόχος για την μετατροπή του ArrayList<ArrayList<Posting>> σε ένα ArrayList<Posting> object.

```
public static ArrayList<Posting> NOT_A(ArrayList<Posting> aPost, InvertedIndex invertedIndex) {
      Collection<ArrayList<Posting>> allPostings = invertedIndex.get();
      ArrayList<ArrayList<Posting>> newList = new ArrayList<ArrayList<Posting>>(allPostings);

      for (int i=0; i< newList.size(); i++) {
          if (newList.get(i) == aPost) {
              newList.remove(i);
          }
      }
      ArrayList<Posting> result = new ArrayList<Posting>();
      for (int i = 0; i < newList.size(); ++i) {
          result.addAll(newList.get(i));
      }
      return result;
  }
```

Δημιουργία Εφαρμογής

Runner Manager

Ένας βασικός στόχος της εργασίας ήταν η δημιουργία ενός περιβάλλοντος για την εισαγωγή query και την εμφάνιση των αποτελεσμάτων αναζήτησης. Για να πετύχουμε τον στόχο αυτό δημιουργήσαμε την συνάρτηση runner_manager, η οποία διαχειρίζεται τις παραπάνω απαιτήσεις.

Η εφαρμογή τρέχει σε ένα while loop, το οποίο σταματάει όταν ο χρήστης εισάγει την τιμή q. Η τιμή αυτή επιλέχθηκε καθώς τιμές quit ή exit θα μπορούσαν να χρησιμοποιηθούν σε query, αν και έχουμε φροντίσει να ενημερώνουμε τον χρήστη κάθε φορά που ζητείται να εισάγει δεδομένα.

Ξεκινώντας την εφαρμογή η εφαρμογή θα ζητήσει από τον χρήστη να εισάγει το query με τα terms. Σε αυτό το σημείο ο χρήστης δεν μπορεί να φύγει από την εφαρμογή και πρέπει να εισάγει το query.

Δοθέντος ενός query από τον χρήστη καλούμε την συνάρτηση chooseQueryType, η οποία επεξεργάζεται το query, και καλεί την κατάλληλη συνάρτηση search έτσι ώστε να βρούμε τα κατάλληλα αποτελέσματα. Επιστρέφοντας το αποτέλεσμα έχουμε ένα βασικό έλεγχο, όπου αν δεν υπάρχει κάποιο κείμενο που ικανοποιεί το query τότε δίνεται η επιλογή στο χρήστη να συνεχίσει με ένα νέο query ή να σταματήσει την εφαρμογή με το q.

Σε περίπτωση όπου υπάρχουν αποτελέσματα, ο χρήστης έχει την δυνατότητα είτε να συνεχίσει για νέο query εισάγοντας c είτε να επιλέξει κάποιο αποτέλεσμα του query. Στην συνέχεια θα εμφανιστεί το κείμενο και θα ζητηθεί από τον χρήστη να εισάγει enter για να συνεχίσει ή q για να σταματήσει την αναζήτηση.

Για την εμφάνιση των κειμένων χρησιμοποιούμε την συνάρτηση getDocument, όπου σαν παράμετρο έχουμε το ArrayList<Integer> με τα docId’s των κειμένων που περιέχουν τα terms των queries, τον αριθμό σελίδας που θέλει να δει και το mapper αρχείο που θα αντιστοιχίσει το docId με το κείμενο. Για να εμφανίσουμε το κείμενο χρησιμοποιούμε το Buffer reader object, με το οποίο εμφανίζεται κάθε σειρά του κειμένου.

```
public static void runner_manager(InvertedIndex invertedIndex, DocIDMapper mapper)throwsIOException {
       Scanner scanner = new Scanner(System.in);
       String running = "";
       System.out.println("\n\nWelcome to the browser");

       while (!running.equals("q")) {
           System.out.println("Please enter the query:");
           String query = scanner.nextLine();
           ArrayList<Posting> finalResult = QueryParser.chooseQueryType(query, invertedIndex);
           if (finalResult.size() == 0) {
               System.out.println("\n\n\nUnfortunately there are no documents to match your query");
               System.out.println("\n\n\nEnter q to exit the browser or hit enter to continue\n\n");
               running = scanner.nextLine();
           } else {
               for(int i=0; i<finalResult.size(); i++) {
                   System.out.println(i + "." + mapper.get(finalResult.get(i).getnDocId()));
               }
               System.out.println("Please type the number of the page you would like to visit or 'c'
to continue");
               String choice = scanner.nextLine();

               if (choice.equals("c")) {
                   System.out.println("\n\n\nEnter q to exit the browser or hit enter to continue");
                   running = scanner.nextLine();
               } else {
                   int page = Integer.parseInt(choice);
                   getDocument(finalResult, page, mapper);
                   System.out.println("\n\n\nEnter q to exit the browser or hit enter to continue");
                   running = scanner.nextLine();
               }
           }
       }
       scanner.close();
   }


public static void getDocument(ArrayList<Posting> result, int choice, DocIDMapper mapper) throws
IOException {
       String path = mapper.get(result.get(choice-1).getnDocId());
       BufferedReader bfrReader = new BufferedReader(new FileReader(path));
       String setLine = "";
       while((setLine = bfrReader.readLine()) != null) {
           System.out.println(setLine);
       }
       bfrReader.close();
   }
```

Runner

Η κλάση Runner είναι η κλάση με την οποία τρέχουμε την εφαρμογή και είναι υπεύθυνη για την δημιουργία των Boolean εγγράφων για το Inverted Index και το Mapper. Ο λόγος για την ύπαρξη της δημιουργίας των Inverted Index και Mapper είναι η ευκολία αλλαγής αυτών των path, χωρίς να χρειαστεί ιδιαίτερη αναζήτηση για το setup της εφαρμογής σε διαφορετικό υπολογιστή.

Σε περίπτωση όπου δεν έχουν δημιουργηθεί τα έγγραφα αυτά (Inverted Index, Mapper) τότε δημιουργούνται για μια φορά και μετά μπορούμε να τα φορτώσουμε για να εκτελέσουμε τα queries.

Τέλος καλούμε την συνάρτηση runner_manager για την λειτουργικότητα, που αναφέραμε παραπάνω, παρέχοντας έτσι μια καθαρή εφαρμογή και ευκολία στην παραμετροποίηση της.

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

    InvertedIndex invertedIndex = new InvertedIndex();
    String indexPath = "/home/stamatiosorfanos/git/SearcghEngine/src/invertedIndex.ser";
    File invertedIndexFile = new File(indexPath);

    DocIDMapper mapper = new DocIDMapper(8001);
    String mapperPath = "/home/stamatiosorfanos/git/SearcghEngine/src/mapper.ser";
    File mapperFile = new File(mapperPath);

    if (!invertedIndexFile.exists() || !mapperFile.exists()) {
        File data = new File("/home/stamatiosorfanos/git/SearcghEngine/Reuters8Kdata");
        Parser.parseData(data, mapper, invertedIndex);
        invertedIndex.write(indexPath);
        mapper.write(mapperPath);
    }
    else {
        invertedIndex.load(indexPath);
        mapper.load(mapperPath);
    }

    RunnerManager.runner_manager(invertedIndex, mapper);
}
```