Experis Case

November 7, 2021

Indlæsning og strukturering af data

Før jeg afdækker, hvordan selve de stillede problemer kan løses, må første skridt nødvendigvis være en gennemgang af data indlæsning, og hvordan data'en efterfølgende bliver sat i struktur. Casen kommer med en række .txt filer, som indeholder `Product` og `User` data'en, som ligger til grund for de to anbefalingsfunktioner.

Indlæsningen af data'en sker i denne løsning gennem brug af Java klassen `BufferedReader`, der læser en linje af gangen fra filen den bliver givet. Data'en rengøres for mellemrum (whitespaces) og efterfølgende deles data'en op i kategorier ved brug af string operationen `split(Regex)`, hvor løsningen udnytter formatet data'en kommer i, til at dele hver kategori af data atomisk op. Data'en gemmes efterfølgende i enten et `Product` object eller et `User` object, hvor data typen string parses til den valgte data type for hver kategori.

User klasse

`User` klassen er designet til at holde al `User` data i sine field variabler. Hver variable har en `.get()` metode implementeret, så data'en i en enkelt kategori kan hentes, ved blot et kald til relevante `.get()` metode. Dette er særligt relevant, når vi skal sammenligne og søge efter data i de stillede problemer.

Et `User` objects fields kan udelukkende modificeres, når et object instantieres, altså kan data'en kun gemmes, når et nyt `User` object skabes ved brug af dets constructor.

Det er et konkret design valg at bruge set til at holde *viewed* og *purchased* data. Dette for at sikre hurtig lookup på de enkelte stykker data, samtidig med at holde data'en fri for kopier.

`User` klassen kan ses herunder:

```
1  class User {
2  int id;
3  String name;
4  Set<Integer> viewed;
5  Set<Integer> purchased;
6      public User(int i, String n, Set<Integer> v, Set<Integer> p) {
7          id = i;
8          name = n;
9          viewed = v;
10         purchased = p;
11     }
12 [...] getter metoder for hvert field.
13 }
```

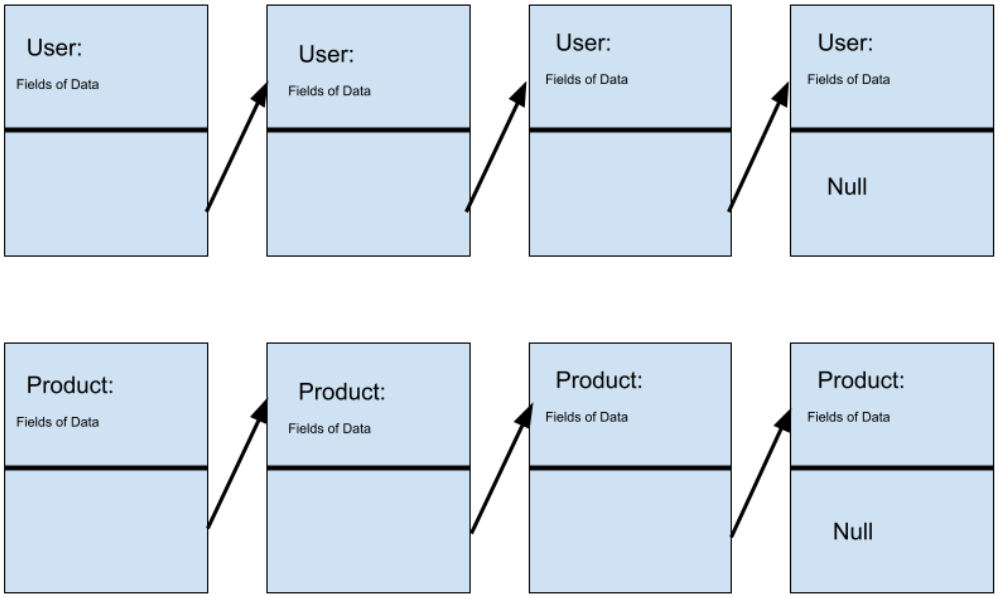
Product klasse

`Product` klassen er designet efter de samme principper som `User` klassen. Al data er gemt atomisk i fields, og data'en kan kun modificeres ved instantiering af et nyt `Product` object gennem dets constructor. Genre/keyword data om hver enkelt `Product` er her også gemt i et set, af samme årsager, som beskrevet under `User` klassen.

Den endelige data struktur

Efter at hver enkel linje er indlæst og sat i sit respektive object, gemmes hvert object i en liste. Konsekvensen af dette er, at efter indlæsning af al `Product` og `User` data ender programmet med to separate lister af henholdsvis `User` objects og `Product` objects, som nu kan søges i, og data'en kan fra hvert enkelt field nemt hentes ud, ved brug af getter metoder, og sammenlignes. Brugen af lister opfordrer til en naiv gennemgang af hvert object. Dette vil ikke skabe de hurtigst mulige algoritmer, men som proof of concept fungerer denne tilgang godt, da tilgangen brugt i programmet lettere lader sig læse, ved implementeringen af naive, "full traversal" algoritmer. Effektivisering kan let lade sig indarbejde på et senere punkt, særligt vil parallelisering af algoritmerne i denne løsning give en forholdsvist stor forbedring af running time.

Den endelige data struktur kan visualiseres således:



Populære produkter

Med den ovenfor beskrevne data struktur på plads, kan det første problem, hvor en generel liste af anbefalinger af de bedste produkter i `Product` basen skal udarbejdes. Denne løsning tilgår dette problem ved først at implementere to algoritmer. En til at udregne købsrate af hvert `Product`, en anden til at udregne det højest rangerede `Product` blandt en række af `Products`. Den sidste algoritme skal særligt fungere, med en hvilken som helst liste af `Product` objects, så vi når som helst kan udregne det bedst rangerede `Product`. Med disse to algoritmer på plads, kan den generelle liste af anbefalinger udregnes ved at samle alle `Product` objects, som deler en genre kategori (et keyword), og gemme den højest vurderede inden for hver genre. Dernæst kan den returnerede liste af højest vurderede `Product` objects reduceres ved kun at beholde de `Product` objects, som har en købsrate over n, i denne løsning er $n = 0$, pga. den forholdsvis lille mængde data i casen. Dette giver en endelig liste af anbefalinger med de bedst vurderede `Products`, som allerede er købt af mindst en `User`.

De centrale algoritmer kan ses herunder:

- En algoritme til udregning af købsraten (hvor mange gange er et `Product` købt af `User` basen) på et enkelt `Product`:

```
1 private static int productPurchaseRate(int itemId, List<User> users){
2     int res = 0;
3         for (User user : users) {
4             for (int id : user.getPurchased()) {
5                 if (id == itemId) res++;
6             }
7         }
8     return res;
9 }
```

- En algoritme til at udregne det højest vurderede `Product` i en liste af `Product`'s:

```
1 private static Product highestRatedProduct(List<Product> products){
2     Product res = products.get(0);
3         for (Product product : products) {
4             if (product.getRating() > res.getRating()) res = product;
5         }
6     return res;
7 }
```

- En algoritme der returnerer højest vurderede `Product` fra hver genre:

```
1 private static List<Product> genreHRPpopList(List<Product> products){
2     List<Product> res = new LinkedList<>();
3     Set<String> genres = getCategorySet(products);
4              for (String string : genres) {
5                  List<Product> movies = new ArrayList<>();
6                      for (Product product : products) {
```

```
                            if (product.getKeywords().contains(string)) movies.add
                                (product);
                        }
                res.add(highestRatedProduct(movies));
                }
      return res;
}
```

- En algoritme der reducerer listen ovenfra til kun at have `Products`, som har en købsrate over 0, hvorefter den gemmer de tilbageværende `Product`'s data i strings, så den endelige liste er klar til at blive sendt som output, printet i konsollen eller pipelined til et andet program.

```
private static Set<String> popList(List<Product> products, List<User> users){
    Set<String> res = new HashSet<>();
    List<Product> genreHRP = genreHRPpopList(products);
    String st;
        for (Product HRP : genreHRP) {
            if (productPurchaseRate(HRP.getId(), users) > 0){
                st = "Name: " + HRP.getOGName() + "\nRating: " + HRP.getRating()
                     + "\nPrice: " + HRP.getPrice();
                res.add(st);
            }
        }
    return res;
}
```

Listen, printet til konsollen i denne løsning, af generelle/populære anbefalinger, ser således ud:

```
Problem 1 Solution:


Name:  War and Peace
Rating: 4.2
Price: 20

Name:  Popeye
Rating: 4.7
Price: 15

Name:  Shanghai Noon
Rating: 3.7
Price: 15

Name:  Cloudy with a Chance of Meatballs 2
Rating: 4.5
Price: 20
```

Individuelle anbefalinger

Ved problemet omkring individuelle anbefalinger skal programmet læse yderligere data fra en .txt fil. I denne sammenhæng gælder det nuværende `User` sessioner. Her kan ses ID og nuværende session (hvilket `product`, som den relevante `User` i øjeblikket kigger på), for hver online `User`. I stedet for at lave yderligere objects, hentes det relevante `User` object ved hjælp af ID, og gemmes i et map som key, med det `Product` objects ID, som kigges på, som value. Dermed opnås et map med hver online `User` og ID på det relevante `Product` object.

Dernæst udregnes alle relaterede `Product` objects, med baggrund i, at de er relaterede, hvis de deler et keyword med det nuværende `Product` object, som bliver kigget på, af den respektive `User`. I denne proces fravælges `Product` objects, hvis ID er indeholdt i det respektive `User` object's viewed og purchased set. Dernæst udvælges det højest vurderede `Product` inden for hvert keyword, hvert tilbageværende `Product` parses til en præsentabel string, sættes i en liste over relaterede `Products`, og den respektive `User` og listen af anbefalinger gemmes i et map med `User` som key, listen af anbefalinger som value og returneres.

De centrale algoritmer kan ses herunder:

- En algoritme til indlæsning af Current `Users` og det respektive `Product` object, som hver `User` kigger på.

```
1  private static Map<User, Integer> currentUserRequests(List<User> users) throws
       IOException {
2      Map<User, Integer> res = new HashMap<>();
3      [...] File reading
4      String st;
5          while ((st = br.readLine()) != null) {
6                 String[] currentUser = removeWS(st).split(",");
7                       for (User user: users) {
8                             if (Integer.parseInt(currentUser[0]) == user.getId())
                                  res.put(user, Integer.parseInt(currentUser[1]));
9                       }
10         }
11     br.close();
12     return res;
13 }
```

- En algoritme til at finde det højest vurderede `Product` inden for hver genre/keyword.

```
1  private static List<Product> genreHRPUserRec(List<Product> products, Set<String>
       keywords){
2      List<Product> res = new LinkedList<>();
3          for (String string : keywords) {
4              List<Product> movies = new LinkedList<>();
5                     for (Product product : products) {
6                         if (product.getKeywords().contains(string)) movies.add(
                              product);
7                      }
```

```
8               res.add(highestRatedProduct(movies));
9           }
10      return res;
11  }
```

- Algoritmen der laver den egentlige søgning i data og udvælgelse.

```
1  private static Set<Product> handleRequest(User user, int id, List<Product>
       products, List<User> users){
2           Set<Product> res = new HashSet<>();
3           Set<String> keywords = new HashSet<>();
4                   for (Product product : products) {
5                       if (id == product.getId()) keywords = product.getKeywords();
6                   }
7           List<Product> relatedProducts = new LinkedList<>();
8                   for (String string : keywords) {
9                       for (Product product : products) {
10                          if (((product.getKeywords().contains(string)) && !(user
                                .getViewed().contains(product.getId()))) && !(user.
                                getPurchased().contains(product.getId())))
                                relatedProducts.add(product);
11                      }
12                  }
13          res.addAll(genreHRPUserRec(relatedProducts, keywords));
14          return res;
15  }
```

- Algoritmen der samler relevant `User` og listen af anbefalede `Products`.

```
1  private static Map<User, List<String>> collectUserRec(List<User> users, List<
       Product> products) throws IOException {
2               Map<User, List<String>> res = new HashMap<>();
3               Map<User, Integer> userRequests = currentUserRequests(users);
4                   for (Map.Entry<User, Integer> entry : userRequests.entrySet()) {
5                        res.put(entry.getKey(), userRecList(handleRequest(entry.
                             getKey(), entry.getValue(), products, users)));
6                   }
7               return res;
8  }
```

Listen, printet til konsollen i denne løsning, af individuelle anbefalinger, ser således ud:

```
Problem 2 Solution:



Username: Ida || user id 3

recommendations:

Name:  GoldenEye
Rating: 4.8
Price: 25

Name:  Ghost in the Shell: Solid State Society
Rating: 4.3
Price: 15

Name:  La La Land
Rating: 4.9
Price: 25


Username: Mia || user id 5

recommendations:

Name:  GoldenEye
Rating: 4.8
Price: 25

Name:  Dr. Terror's House of Horrors
Rating: 5.0
Price: 10


Username: Olav || user id 1

recommendations:

Name:  Dr. Terror's House of Horrors
Rating: 5.0
Price: 10

Name:  Insomnia
Rating: 4.1
Price: 15

Name:  The thing
Rating: 4.2
Price: 20

Name:  La La Land
Rating: 4.9
Price: 25


Username: Tage || user id 2

recommendations:

Name:  GoldenEye
Rating: 4.8
Price: 25

Name:  La La Land
Rating: 4.9
Price: 25
```