Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Melanija Vezočnik

Analiza in uporaba MapReduce za priporočilne sisteme

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite pristope k paralelnemu reševanju problemov. Analizirajte in preučite programski model MapReduce in izvajalno okolje MapReduce. Primerjajte tri izbrane implementacije MapReduce. Opišite koncept priporočilnih sistemov. Z uporabo MapReduce izdelajte dva primera priporočilnih sistemov in sicer iskanje predlogov za prijatelje v socialnem omrežju ter iskanje predlogov za filme.

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Melanija Vezočnik, z vpisno številko **63100338**, sem avtorica diplomskega dela z naslovom:

Analiza in uporaba MapReduce za priporočilne sisteme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 11. septembra 2014 Podpis avtorja:

Najprej se želim zahvaliti mentorju prof. dr. Matjažu Branku Juriču za vodenje pri izdelavi in za strokovni pregled diplomskega dela. Zvonetu Gazvodi se zahvaljujem za pomoč pri praktičnem delu in za strokovni pregled diplomskega dela, dr. Sebastijanu Špragerju pa za strokovni pregled diplomskega dela. Hvala vsem, ki ste mi pomagali pri nastajanju diplomskega dela. Posebno zahvalo namenjam družini, ki mi je ves čas študija stala ob strani.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CentOS	Community Enterprise Operating System	distribucija Linux
CDH 5	Cloudera Distribution Including Apache Hadoop	distribucija Cloudera, ki vključuje Apache Hadoop
GB	gigabyte	gigabajt
GHz	gigahertz	gigaherc
HDFS	Hadoop Distributed File System	porazdeljen datotečni sistem Hadoop
JAR	Java ARchive	datotečni format
MB	megabyte	megabajt
MPI	Message Passing Interface	vmesnik za prenos sporočil
NoSQL	Not Only SQL	skupen pojem za tehnologije, v katerih narava podatkov ne zahteva relacijskega modela
RAM	Random Access Memory	bralno-pisalni pomnilnik
RPC	remote procedure call	klic za oddaljene procedure
VM	virtual machine	navidezna naprava

Povzetek

MapReduce je programski model, namenjen za razvoj skalabilnih paralelnih aplikacij za obdelavo velikih množic podatkov, izvajalno okolje, ki podpira programski model in koordinira izvajanje programov, in implementacija programskega modela in izvajalnega okolja. Cilj diplomskega dela je analizirati MapReduce in ga preizkusiti na dveh primerih priporočilnih sistemov. Cilj smo dosegli, saj smo uspeli realizirati izračun s pomočjo MapReduce na testnih primerih. Najprej smo analizirali programski model in izvajalno okolje ter primerjali tri implementacije MapReduce: Hadoop MapReduce, MongoDB in knjižnico MapReduce-MPI. Ugotovili smo, da je za realizacijo izbranih primerov priporočilnih sistemov najprimernejša implementacija Hadoop MapReduce, saj nudi toleranco za okvare in reproducira podatke, s čimer zagotavlja zanesljivost. Nato smo z uporabo navidezne naprave Cloudera QuickStart VM, ki je gruča Hadoop z enim vozliščem, realizirali izbrana primera priporočilnih sistemov.

Ključne besede: Hadoop MapReduce, knjižnica MapReduce-MPI, MapReduce, MongoDB, priporočilni sistemi.

Abstract

MapReduce is a programming model for developing scalable parallel applications for processing large data sets, an execution framework that supports the programming model and coordinates the execution of programs and an implementation of the programming model and the execution framework. The goal of the thesis is to analyse MapReduce and to use it on two examples of recommender systems. The goal is achieved by developing the computation with MapReduce successfully. At first the programming model and the execution framework are analysed and three implementations for MapReduce: Hadoop MapReduce, MongoDB and MapReduce-MPI Library are compared. It is discovered that Hadoop MapReduce is the most suitable implementation for developing the selected examples of recommender systems as it provides fault tolerance and data reproduction which ensure reliability. Then the selected examples of recommender systems are developed using Cloudera QuickStart VM which is a one node Hadoop cluster.

Keywords: Hadoop MapReduce, MapReduce, MapReduce-MPI Library, MongoDB, recommender systems.

Poglavje 1

Uvod

V informacijski dobi se organizacije spopadajo z velikimi množicami podatkov (*big data*). Repozitoriji privatnih in javnih podatkov definirajo informacijske družbe in oblikujejo množice interesov. Razširjena je tudi analiza podatkov o uporabnikovih aktivnostih, ki jih uspešne spletne strani spremljajo in shranjujejo. Velike množice podatkov se ne pojavljajo samo v poslovni sferi, temveč tudi pri številnih znanstvenih disciplinah, med drugim pri astronomiji, genetiki in fiziki. Ključnega pomena je paralelna obdelava podatkov, s katero zagotovimo, da bo obdelava podatkov zaključena pravočasno.

Leta 2003 je Google razvil programski model MapReduce. Programski model temelji na preprostih konceptih, omogoča skalabilnost in je odporen na okvare, zato so bile od takrat dalje razvite številne implementacije MapReduce. Uporaba MapReduce se je razširila na različna področja, med drugim tudi na bioinformatiko, strojno učenje ter obdelavo besedil in velikih grafov.

V diplomskem delu najprej opišemo pristope k paralelnemu reševanju problemov. Sledi podroben pregled in analiza MapReduce, za tem pa opišemo in primerjamo tri najpopularnejše implementacije MapReduce. Te so Hadoop MapReduce, MongoDB in knjižnica MapReduce-MPI. Nato sledijo kratka predstavitev priporočilnih sistemov in dva njihova izbrana primera.

Poglavje 2

Pristopi k paralelnemu reševanju problemov

Bistveni element pri reševanju problemov z računalnikom je razvoj algoritmov. Paralelni algoritem je način reševanja danega problema z uporabo več procesorjev ali več računalnikov. V nadaljevanju opišemo korake pri načrtovanju paralelnega algoritma.

2.1 Dekompozicijske tehnike

Prvi korak pri načrtovanju paralelnega algoritma je dekompozicija. To je proces, ki algoritem razstavi na manjše dele, ki jih imenujemo naloge. Naloge so računske enote. Njihova velikost je poljubna. Čas za reševanje problema skrajšamo, če več neodvisnih nalog izvajamo paralelno. Dekompozicijske tehnike omogočajo, da algoritem razstavimo na naloge, ki se izvajajo paralelno, in da identificiramo razpoložljiv paralelizem.

V grobem tehnike dekompozicije delimo na rekurzivno, podatkovno, raziskovalno in špekulativno. Rekurzivna in podatkovna dekompozicija sta splošnejši, zato se uporabljata pri širokem naboru problemov, raziskovalna in špekulativna dekompozicija pa sta bolj specializirani, zato sta namenjeni za uporabo pri specifičnih problemih. Omenjene dekompozicijske tehnike razlagamo v nadaljevanju.

2.1.1 Rekurzivna dekompozicija

Rekurzivna dekompozicija je metoda, s katero izkoristimo paralelizem pri problemih, ki jih lahko rešimo z uporabo strategije deli in vladaj (*divide and conquer*). Pri njej se problem najprej razdeli v množico neodvisnih podproblemov, nato pa jih rešimo z uporabo podobne delitve še na manjše podprobleme. Rešitve podproblemov združimo v končno rešitev.

2.1.2 Podatkovna dekompozicija

Podatkovna dekompozicija je metoda, s katero dosežemo paralelizem pri algoritmih, ki delujejo na velikih podatkovnih strukturah [16]. Podatke, nad katerimi izvedemo algoritem, najprej razdelimo na podatkovne podmnožice, ki jih nato uporabimo pri razstavitvi algoritma na naloge. Vsaka naloga nad podatkovno podmnožico izvede operacije, ki so ponavadi enake ali izbrane iz majhne množice operacij.

2.1.3 Raziskovalna dekompozicija

Raziskovalno dekompozicijo uporabljamo za dekompozicijo problemov, pri katerih algoritem rešimo s preiskovanjem prostora za rešitve. Njen potek lahko razdelimo na dva koraka. V prvem koraku prostor za iskanje rešitev razdelimo na manjše dele, v drugem koraku pa vsak del paralelno preiščemo, dokler ne najdemo iskane rešitve.

2.1.4 Špekulativna dekompozicija

Špekulativno dekompozicijo uporabimo, če je v algoritmu več razvejitev (*branches*), odvisnih od vmesnih rezultatov naloge [16]. Če vmesni rezultat naloge, ki ga potrebujemo za izbiro prave razvejitve, še ni znan, lahko začnemo izvajati razvejitve v naslednji fazi izvajanja algoritma. Ena izmed teh razvejitev je tudi prava in takoj, ko bo znan vmesni rezultat naloge, ki jo določajo, opustimo izvajanje vseh razvejitev, razen le-te.

2.2 Karakteristike nalog in interakcij

Naslednji korak pri načrtovanju paralelnega algoritma je dodelitev z dekompozicijo določenih nalog procesom. Nanjo vplivajo karakteristike nalog in interakcij (*interactions*) - komunikacije med procesi, zato pri njeni izbiri pogosto izhajamo iz dekompozicije.

2.2.1 Karakteristike nalog

Generiranje nalog

Naloge generiramo statično ali dinamično. Statično generirane naloge so znane še pred začetkom izvajanja algoritma, dinamično generirane naloge pa niso vnaprej znane.

Velikost nalog

Če poznamo velikosti nalog, jih lahko uporabimo pri dodelitvi nalog procesom. Pomagajo nam oceniti čas izvajanja posamezne naloge. Uniformne (*uniform*) naloge potrebujejo za rešitev približno enak čas, pri neuniformnih (*non-uniform*) nalogah pa je čas za rešitev različen.

Velikost podatkov za naloge

Podatki različnih velikosti, vezani na nalogo, morajo biti na voljo procesom, ki izvajajo nalogo. Lokacija in velikost podatkov določata optimalen proces, ki bo za izvedbo naloge potreboval najmanj dodatnega časa za prenos podatkov (*overhead*).

2.2.2 Karakteristike interakcij med procesi

Če je paralelni algoritem kompleksen, si morajo procesi deliti podatke in informacije za sinhronizacijo. Različni paralelni algoritmi zahtevajo tudi različne tipe interakcij med procesi.

Statične in dinamične interakcije

Statične interakcije se zgodijo ob vnaprej določenih časih, ki so znani še pred izvajanem algoritma. Nasprotno pri dinamičnih interakcijah njihov začetek pred izvajanem algoritma ni znan.

Regularne in neregularne interakcije

Interakcije so regularne (*regular*), če imajo takšno strukturo, da jo lahko izrabimo za učinkovito implementacijo (*implementation*) [16]. Nasprotno neregularne (*irregular*) interakcije takšne strukture nimajo.

Bralne in bralno-pisalne interakcije

Ločimo bralne in bralno-pisalne interakcije. Pri bralnih interakcijah proces zahteva dostop do podatkov, ki si jih deli več procesov, samo za branje. Pri teh dostopih ne pride do konfliktov (*contention*). Bralno-pisalne interakcije zahtevajo bralni in pisalni dostop. Če več procesov dostopa do istih podatkov, so potrebni mehanizmi zaklepanja. Primer takšnega mehanizma je bralno-pisalno zaklepanje (*read/write lock*). Ta mehanizem dovoljuje, da podatke zapisuje samo en proces, medtem ko ostalim procesom dovoljuje samo branje.

Enosmerne in dvosmerne interakcije

Pri enosmernih interakcijah samo en proces začne in konča interakcijo, ne da bi motil druge procese. Pri dvosmernih interakcijah proces ali več procesov preskrbi podatke, ki jih potrebuje drug proces ali več procesov. Bralne interakcije so enosmerne, bralno-pisalne interakcije pa so enosmerne ali dvosmerne [16].

2.3 Porazdelitev obremenitve med procese

Glavni cilj pri dodeljevanju nalog procesom je, da se vse naloge izvedejo v najkrajšem možnem času. Za dosego cilja želimo zmanjšati čas, potreben za paralelno izvedbo nalog. K njemu največ prispevata čas, ki se porabi za interakcijo med procesi in čas, ko morajo procesi čakati, da bodo na vrsti za izvajanje.

Dobra porazdelitev nalog med procese je kompleksen problem, saj želimo doseči čim večjo učinkovitost ter zmanjšati interakcijo med procesi. To dosežemo tako, da naloge, ki si morajo izmenjati podatke, dodelimo istemu procesu. V večini primerov to povzroči neenakomerno obremenitev procesov. Tako so manj obremenjeni procesi že opravili naloge, medtem ko jih bolj obremenjeni še vedno izvajajo.

Porazdelitev nalog med procese delimo na statično in dinamično. Statično dodeljevanje razdeli naloge med procese še pred začetkom izvajanja algoritma, dinamično dodeljevanje razdeli naloge med procese med izvajanjem algoritma. V splošnem je lažje programirati in zasnovati algoritme, ki uporabljajo statično dodeljevanje.

2.4 Metode za obvladovanje dodatnega časa pri interakcijah

V tem razdelku opisujemo nekatere tehnike za obvladovanje dodatnega časa (*overhead*) pri interakciji med procesi. Te tehnike lahko prilagodijo uporabo lokalnih podatkov, zmanjšajo konflikte, optimizirajo izvajanje nalog in reproducirajo podatke. Uporabne so na različnih stopnjah razvoja algoritma.

2.4.1 Uporaba lokalnih podatkov

V paralelnih algoritmih lahko pride do interakcije, če naloge, ki jih izvajajo različni procesi, zahtevajo dostop do skupnih podatkov ali če procesi zahtevajo podatke, ki jih preskrbijo drugi procesi. Dodaten čas pri interakciji lahko zmanjšamo tako, da uporabimo tehnike, ki vzpodbujajo uporabo lokalnih podatkov ali podatkov, ki so trenutno v pomnilniku.

Zmanjšanje obsega izmenjave podatkov

Dodaten čas, potreben za interakcije, lahko zmanjšamo tako, da zmanjšamo celoten obseg skupnih podatkov. To je mogoče storiti na več načinov, in sicer z uporabo ustrezne dekompozicije in porazdelitve ter z uporabo lokalnih virov za shranjevanje vmesnih rezultatov in izvedbo deljenega dostopa do podatkov.

Zmanjšanje frekvence interakcij

Dodaten čas za interakcije lahko prav tako zmanjšamo, če zmanjšamo frekvenco interakcij. Algoritem preoblikujemo tako, da uporablja skupne podatke, ki jih prenaša v večjih blokih, kar zmanjša potrebo za komunikacijo med procesi.

2.4.2 Zmanjšanje konfliktov

Če več procesov poskuša istočasno dostopati do istih virov, pride do konfliktov. Povzročijo jih na primer procesi, ki istočasno pošiljajo sporočila istim procesom ali istočasni dostopi do skupnih podatkov. Ker se naenkrat lahko obdela samo ena zahteva, so le-te razvrščene v vrsto in obdelane zaporedno. Konflikte lahko zmanjšamo, če algoritem spremenimo tako, da ni več sočasne potrebe po podatkih oziroma je teh dostopov čim manj.

2.4.3 Optimizacija izvajanja nalog

Procesi lahko čakajo na prihod skupnih podatkov ali na prejem dodatne naloge po začetku interakcije. Ta čas lahko zmanjšamo, če nekatere naloge izvedemo med čakanjem ali pa z interakcijo začnemo dovolj zgodaj, tako da se ta zaključi, še preden jo proces potrebuje pri izvedbi naloge.

2.4.4 Reproduciranje podatkov

Če v paralelnem algoritmu več procesov zahteva pogoste bralne dostope do skupnih podatkov, je najbolje reproducirati skupno podatkovno strukturo za vsak proces. Paziti moramo, da pomnilniške zahteve niso prevelike, saj se z reproduciranjem podatkov linearno povečajo glede na število paralelnih procesov.

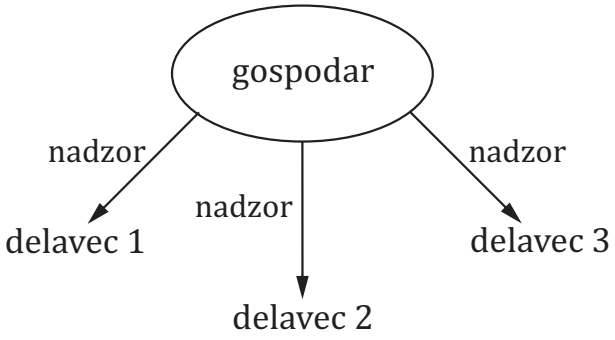
2.5 Modeli za paralelne algoritme

Paralelne algoritme lahko oblikujemo tako, da uporabimo več oblik organizacije paralelnih procesov, zajete v modelih za paralelne algoritme. V splošnem izberemo dekompozicijo, porazdelitev in uporabimo primerno strategijo, da zmanjšamo inte-

rakcije. V nadaljevanju opisujemo najbolj pogoste in uporabne modele za paralelne algoritme.

Model gospodar-delavec

V modelu gospodar-delavec (*master-worker*) ima gospodar-proces glavno funkcijo in generira naloge, ki jih dodeli ostalim procesom za izvajanje. Te procese imenujemo delavci. Omenjen model prikazuje slika 2.1.



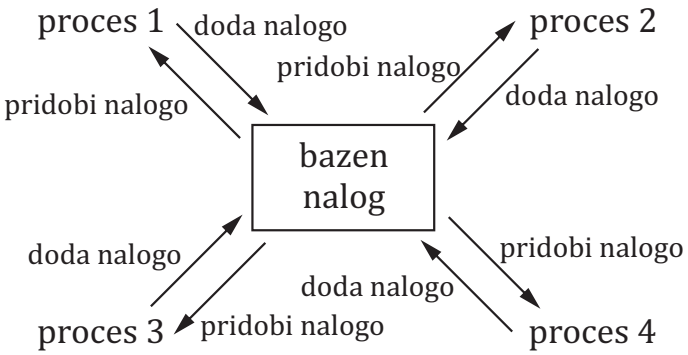
Slika 2.1: Model gospodar-delavec.

Model bazen nalog

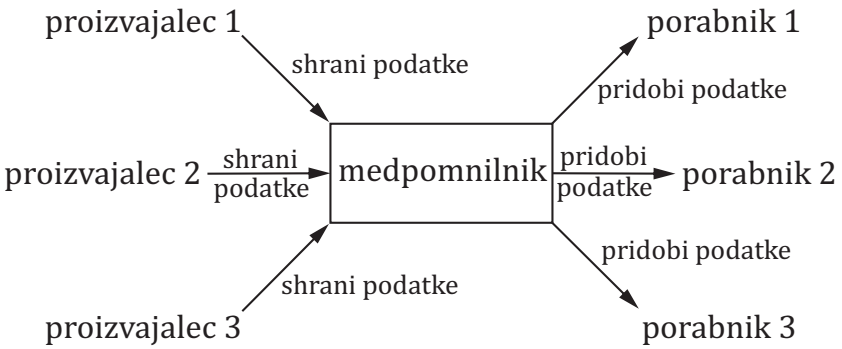
Bazen nalog (*task pool*) je podatkovna struktura, ki hrani naloge za obdelavo. Procesi lahko iz bazena nalog pridobijo naloge za obdelavo. Med obdelavo naloge lahko procesi v bazen nalog dodajo nove naloge. Dostop do bazena nalog mora biti usklajen, da se izognemo konfliktom. Izvedba paralelnega algoritma se zaključi, ko je bazen prazen in vsi procesi končajo z obdelavo nalog. Model bazen nalog je prikazan na sliki 2.2.

Model proizvajalec-porabnik

Proizvajalec-porabnik (*producer-consumer*) je model, sestavljen iz procesov porabnikov in procesov proizvajalcev. Porabnik kot vhod uporabi podatke, ki so izhod proizvajalca. Delovanje je prikazano na sliki 2.3.



Slika 2.2: Model bazen nalog.



Slika 2.3: Model proizvajalec-porabnik.

Cevovodni model

Cevovodni model (*pipelining*) je model, sestavljen iz verige modelov proizvajalec-porabnik. Cevovod ni nujno linearna veriga, ampak je usmerjen graf.

Poglavje 3

MapReduce - pregled in analiza

MapReduce je definiran kot:

1. programski model, ki omogoča enostaven razvoj skalabilnih paralelnih aplikacij, namenjenih za obdelavo velikih množic podatkov [19, 10];

2. izvajalno okolje (*framework*), ki podpira programski model in koordinira izvajanje programov;

3. implementacija programskega modela in izvajalnega okolja [21]. Primer implementacije je Googlova implementacija [10].

3.1 Programski model MapReduce

MapReduce idejno izhaja iz funkcijskega programiranja, in sicer iz funkcij višjega reda *map* in *reduce* [21], ki se pogosto uporabljata v funkcijskih jezikih, kot je *Lisp* [19]. Glavna značilnost funkcij višjega reda je, da lahko za argument navedemo funkcijo.

Programski model MapReduce je zasnovan na preprostih konceptih. Ti so iteracija čez vhodne podatke, izračun vmesnih parov ključ-vrednost za vsak vhodni podatek in njihovo grupiranje glede na ključ ter iteracija po skupinah in njihova skrčitev. Programski model omogoča paralelizacijo in lahko rešuje veliko različnih problemov, pri katerih je potrebno obdelati velike količine podatkov. Je enostaven, zato se je uporaba algoritmov, ki so razviti z MapReduce, razširila na številna

področja, med drugim na obdelavo velikih grafov, statistično strojno prevajanje, obdelavo besedil, strojno učenje, bioinformatiko, kemijo in okoljske znanosti [11]. MapReduce zagotovi abstrakcijo, ki programerju skrije veliko sistemskih podrobnosti na skalabilen, robusten in učinkovit način ter mu posledično omogoča, da se osredotoči na razvoj algoritmov, ne pa na njihovo implementacijo.

Programer definira funkciji *map* in *reduce*. Funkcija *map* vzame kot vhod par ključ-vrednost ter vrne kot izhod množico vmesnih parov ključ-vrednost. Izvajalno okolje združi vse vmesne vrednosti z istim vmesnim ključem v skupino in jih posreduje funkciji *reduce*. Ta sprejme vmesni ključ in množico vrednosti, ki mu pripadajo, ter jo združi v manjšo množico, sestavljeno iz nič ali več elementov. Faza *map* ustreza uporabi funkcije *map* na vseh vhodnih podatkih, faza *reduce* pa ustreza uporabi funkcije *reduce* na vseh vmesnih ključih in pripadajočih množicah vrednosti.

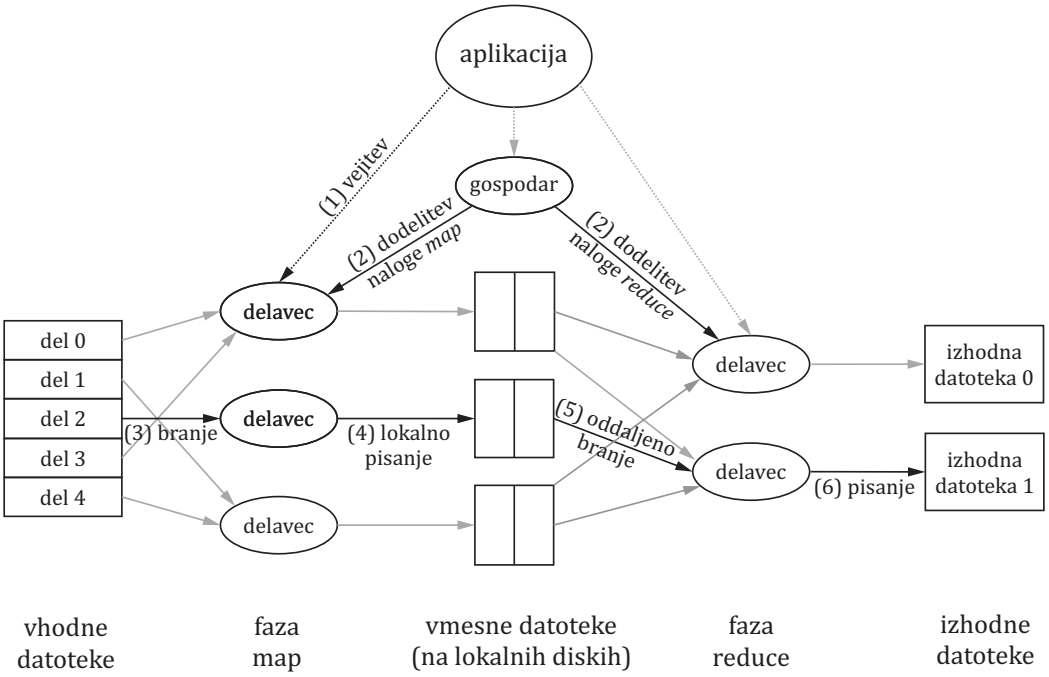
3.1.1 Izvajanje

Opis izvajanja je povzet iz Googlove implementacije [10] in se od izvajanj pri drugih implementacijah lahko razlikuje. Izvajanje prikazuje slika 3.1 in ga sestavljajo naslednji koraki:

1. Podatki v vhodni datoteki aplikacije se razdelijo na več delov. Zažene se veliko kopij programa v gruči, sestavljeni iz množice naprav.

2. Ena kopija programa je izbrana za gospodarja, ostale za delavce. Gospodar izbere delavce in jim dodeli naloge za funkcijo *map* ali funkcijo *reduce*, v nadaljevanju naloga *map* in naloga *reduce*.

3. Delavec, ki mu je dodeljena naloga *map*, prebere vsebino dodeljenega dela podatkov, iz njega razčleni pare ključ-vrednost ter vsak par posreduje funkciji *map*. Funkcija *map* prejme kot vhod par ključ-vrednost ter vrne nič ali več vmesnih parov ključ-vrednost, ki se shranijo v pomnilnik. Vse naloge *map* se izvajajo paralelno. To pomeni, da različne naprave, ki sestavljajo gručo, vsak del vhodnih podatkov obdelujejo istočasno. Če je več nalog *map*, kolikor jih lahko obvladuje gruča, so te razvrščene v vrsto in izvedene v tistem vrstnem redu, ki ga izvajalno okolje določi za najboljšega.



Slika 3.1: Izvajanje MapReduce.

4. V časovnih presledkih se vmesni pari zapišejo na lokalni disk. Pri tem jih particijska funkcija dodeli particijam. Gospodar prejme lokacije vmesnih parov na lokalnem disku in jih dodeli delavcem, zadolženim za opravljanje naloge *reduce*.

5. Ko gospodar opomni delavca, ki je zadolžen za opravljanje naloge *reduce*, o lokacijah, na katerih so shranjeni vmesni pari na lokalnem disku, le-ta prebere shranjene podatke delavcev, ki so zadolženi za nalogo *map*. Ti podatki so nato razporejeni glede na vmesne ključe, tako da so vse ponovitve istega ključa združene.

6. Delavec, ki je zadolžen za opravljanje naloge *reduce*, posreduje ključ in pripadajoč seznam vmesnih vrednosti funkciji *reduce*. Njen izhod je pripet izhodni datoteki te particije.

7. Po zaključku vseh nalog gospodar zažene aplikacijo.

3.1.2 Ravnanje pri napakah in odpovedih

Delavec v okvari

Gospodar periodično preverja, če se vsak delavec odziva. Če se delavec v določenem času ne odzove, gospodar predpostavi, da je delavec v okvari.

Če je delavec zadolžen za opravljanje nalog *map*, mu gospodar vse opravljene naloge odvzame in jih razvrsti med druge delavce. Izhod nalog *map* je namreč shranjen na lokalnem disku okvarjene naprave in je posledično nedostopen. Prav tako se razveljavijo tudi naloge, ki jih delavec trenutno opravlja, in se razvrstijo med druge delavce. Delavci, zadolženi za opravljanje nalog *reduce*, so o okvari obveščeni, tako da lahko vsaka naloga *reduce*, ki še nima podatkov, podatke pridobi od novega delavca.

Če je delavec v okvari zadolžen za opravljanje nalog *reduce*, mu gospodar odvzame samo tiste naloge, ki jih ta delavec trenutno opravlja. Izhod končanih nalog je shranjen v globalnem datotečnem sistemu, zato ponovna izvedba ni potrebna.

Programski model omogoča tudi redundantno izvajanje. To pomeni, da se naenkrat lahko izvaja več enakih nalog. Če je naprava v okvari, če pride do izgube podatkov ali če je izvajanje počasno, naloge ni potrebno prerazporediti, saj je zanjo zadolžen drug delavec ali več delavcev.

Gospodar v okvari

Gospodar periodično zapisuje točke kontrole. Če pride do okvare, se ustvari nova kopija gospodarja od zadnjega stanja točke kontrole.

3.1.3 Izzivi paralelizacije

Paralelna preslikava vhodnih podatkov

Obdelava vhodnih podatkov poteka tako, da so pari ključ-vrednost obdelani zaporedno. Takšna seznamska preslikava je prilagodljiva na paralelizacijo vseh podatkov [19], zato je lahko izvedena paralelno na nivoju posameznih elementov. Funkcija *map* mora biti čista (*pure*). Ne glede na to, kolikokrat je čista funkcija z istimi argumenti poklicana, da vedno isti rezultat [17], torej vrstni red obdelave parov ključ-vrednost ne vpliva na rezultat faze *map*.

Paralelno grupiranje vmesnih podatkov

Za fazo *reduce* je potrebno grupiranje vmesnih podatkov po ključu. Ta problem spada med probleme sortiranja, za katere obstajajo številni paralelni modeli. Iz porazdeljenosti faze *map* izhaja povezava grupiranja s porazdeljeno preslikavo. Za vsak del vmesnih podatkov se lahko izvede grupiranje. Rezultati porazdeljenega grupiranja so lahko združeni centralno.

Paralelna preslikava skupin

Skupina je sestavljena iz ključa in seznama vrednosti. Za vsako skupino posebej se izvede skrčitev. Spet se pojavi seznamska preslikava. Paralelizacija celotnih podatkov je upoštevana za fazo *reduce* tako, kot je upoštevana pri fazi *map*.

Paralelna skrčitev skupin

V funkcijskem programiranju je *reduce* operacija, ki pretvori seznam v eno samo vrednost s pomočjo asociativne operacije in njene enote. Njene lastnosti lahko prenesemo tudi na MapReduce [19] in skupine paralelno skrčimo.

3.2 Izvajalno okolje MapReduce

Opravilo MapReduce je sestavljeno iz programske kode funkcij *map* in *reduce* ter konfiguracijskih parametrov. Programer predloži opravilo strežniku-vozlišču v gruči. Izvajalno okolje poskrbi za razporejanje, obstoj podatkov in programske kode, sinhronizacijo ter ravnanje pri napakah in odpovedih.

3.2.1 Razporejanje

Opravilo MapReduce se razdeli na manjše naloge, običajno več tisoč, ki jih je potrebno razdeliti med vozlišča v gruči. Če je opravilo MapReduce obsežno, lahko število nalog preseže število vozlišč v gruči in se vse naloge ne morejo izvajati istočasno, zato jih razvrščevalnik razvrsti. Razvrščevalnik vodi tudi evidenco nalog, ki se trenutno izvajajo, tako da se lahko razpoložljivim vozliščem dodelijo čakajoče naloge. Drugi vidik razporejanja vključuje koordinacijo med nalogami, ki pripadajo različnim opravilom.

3.2.2 Obstoj podatkov in programske kode

Za izvedbo naloge je potrebno dostaviti podatke do programske kode. Lokalnost podatkov dosežemo tako, da razvrščevalnik zažene nalogo na vozlišču, ki hrani del podatkov, ki ga potrebuje naloga, na primer na lokalnem disku. Tako se programska koda prenese k podatkom. Če to ni mogoče, se nekje drugje zaženejo nove naloge, potrebni podatki pa se prenesejo preko omrežja.

3.2.3 Sinhronizacija

Sinhronizacija se nanaša na mehanizme, pri katerih se mnogo sočasno izvajajočih nalog združi, da bi si na primer delile vmesne rezultate ali kakšne druge podatke [21]. Pri MapReduce je sinhronizacija dosežena med fazama *map* in *reduce*. Takrat poteka grupiranje parov ključ-vrednost glede na ključ. To dosežemo z velikim porazdeljenim sortiranjem, ki vključuje vsa vozlišča, ki izvajajo eno izmed nalog *map* in *reduce*. Vmesni podatki se kopirajo preko omrežja.

3.2.4 Ravnanje pri napakah in odpovedih

Izvajalno okolje MapReduce mora opraviti vse naloge, četudi so napake in okvare v okolju pogosto prisotne. Pojavljajo se okvare diskov in bralno-pisalnih pomnilnikov, v podatkovnih centrih pride do izpadov, na primer vzdrževanje in nadgradnja sistema, izpad napajanja in povezave. Poleg napak na strojni opremi so tu še napake v programski opremi. Pri problemih z veliko podatki se v programski kodi, za katero velja, da je brez hroščev, odkrijejo kakšne napake. Vsaka velika množica podatkov lahko vsebuje podatke, iz katerih izhajajo napake.

3.2.5 Prednosti izvajalnega okolja MapReduce

Izvajalno okolje MapReduce prinaša številne prednosti:

- **Preprosta in enostavna uporaba:** Programer definira samo funkciji *map* in *reduce*. Ni mu treba specificirati distribucije opravila preko vozlišč.
- **Prilagodljivost:** MapReduce programerju omogoča lažjo obravnavo nepravilnih ali nestrukturiranih podatkov.

- **Neodvisnost od podatkovnega skladišča:** MapReduce lahko sodeluje z različnimi podatkovnimi skladiščnimi plastmi, saj je praktično neodvisen od njih.
- **Neobčutljivost na okvare:** Poskusi, ki so bili izvedeni pri Googlu, kažejo, da je MapReduce dokaj neobčutljiv na okvare [10].

3.2.6 Slabosti izvajalnega okolja MapReduce

Izvajalno okolje MapReduce ima med drugim naslednje slabosti:

- **Ni visokonivojskih programskih jezikov:** MapReduce ne podpira nobenega visokonivojskega programskega jezika in nobenih tehnik za optimizacijo poizvedb, zato jih mora programer kodirati v funkcijah *map* in *reduce*.
- **Ni shem in indeksov:** MapReduce je brez shem in indeksov. Opravilo MapReduce lahko prične z delom takoj za tem, ko je vhod prenesen v podatkovno skladišče. Zahteva tudi razčlenitev vsakega elementa pri branju vhoda in njegovo transformacijo v podatkovne objekte za obdelavo podatkov, kar zmanjša zmogljivost [20].
- **Samo en fiksen tok podatkov:** MapReduce ima en fiksen tok podatkov, zato je veliko kompleksnih algoritmov težko rešiti samo s funkcijama *map* in *reduce*. Poleg tega algoritmi z več vhodi niso dobro podprti, saj je bil MapReduce prvotno zasnovan, da prebere en vhod in generira en izhod.
- **Slaba učinkovitost:** Operacije MapReduce niso vedno optimizirane za učinkovit vhod-izhod, saj sta primarna cilja toleranca do okvar in skalabilnost. Ker se faza *reduce* ne more pričeti, dokler se ne zaključi faza *map* in ker se opravilo ne more zaključiti, dokler se ne zaključi faza *reduce*, se cevovodna paralelizacija ne more uporabiti [20]. MapReduce nima specifičnih načrtov za izvajanje in jih tudi ne optimizira. Prav tako ima izvajalno okolje težave z zakasnitvijo, ki se pojavi zaradi narave podedovane paketne obdelave.

Poglavje 4

Pregled implementacij MapReduce

V razdelku so predstavljene tri pogosto uporabljene implementacije MapReduce. To so Hadoop MapReduce, MongoDB in knjižnica MapReduce-MPI.

4.1 Hadoop MapReduce

Hadoop MapReduce [5] je implementacija programskega modela MapReduce in računska komponenta projekta Apache Hadoop [23]. Apache Hadoop je računalniško okolje. Zagotavlja pragmatično, cenovno učinkovito in skalabilno infrastrukturo za razvijanje različnih tipov aplikacij. Napisan je v programskem jeziku Java. Poleg Hadoop MapReduce je pomembna komponenta, ki sestavlja Apache Hadoop, tudi porazdeljen datotečni sistem HDFS, ki je s Hadoop MapReduce neločljivo povezan.

4.1.1 HDFS

HDFS je datotečni sistem, namenjen za shranjevanje zelo velikih datotek in teče na diskovnih gručah, sestavljenih iz potrošniške diskovne opreme [24].

HDFS je datotečni sistem v uporabniškem prostoru, saj koda datotečnega sistema teče zunaj jedra kot procesi operacijskega sistema, kar je za implementacijo bolj prilagodljivo, enostavno in varno. HDFS je prav tako porazdeljen datotečni

sistem. To pomeni, da vsaka naprava v gruči hrani podmnožico podatkov, ki tvorijo celoten datotečni sistem. Če želimo shraniti več podatkov, preprosto dodamo več naprav z več diski.

Bloki

HDFS tako kot disk uporablja koncept blokov. Velikost bloka določa minimalno količino podatkov, ki se lahko bere ali piše. Pri HDFS je privzeto 64 MB.

Datoteke so razbite v dele velikosti bloka. Shranjene so kot neodvisne enote. Če so manjše od velikosti bloka, ne zavzamejo celotnega bloka. HDFS reproducira vsak blok v datoteki na več naprav v gruči, privzeto trikrat. Kljub temu spremembe datotek niso možne, saj se datoteke pišejo le enkrat, berejo pa večkrat. Zaradi reprodukcije se več okvar na napravah lažje tolerira, pa tudi podatki se lahko berejo iz naprave, ki je najbližje aplikaciji. HDFS hrani podatke o lokacijah reproduciranih blokov. Če se njihovo število zmanjša pod privzeto število, datotečni sistem samodejno reproducira nov blok iz preostalih reprodukcij.

Tipi vozlišč

V gruči HDFS poznamo tri tipe vozlišč. Ti so imensko vozlišče (*namenode*), sekundarno imensko vozlišče (*secondary namenode*) in podatkovno vozlišče (*datanode*). Imensko vozlišče hrani metapodatke datotečnega sistema in datoteko, ki vsebuje podatke o preslikavah blokov, ter preskrbi globalno sliko datotečnega sistema. Sekundarno imensko vozlišče izvaja notranje beleženje transakcij imenskega vozlišča za točke kontrole. Podatkovno vozlišče hrani vsebino datotek, razdeljeno v bloke. Imensko ali sekundarno imensko vozlišče je lahko samo eno, podatkovnih pa je lahko več.

4.1.2 Lastnosti Hadoop MapReduce

Hadoop MapReduce je podoben tradicionalnim porazdeljenim računalniškim sistemom. Ima izvajalno okolje in aplikacijo. Vozlišče, ki je gospodar, usklajuje vire v gruči. Delavci opravljajo naloge *map* ali *reduce*. Za komunikacijo med procesi se uporabljajo klici za oddaljeno proceduro (*RPC*).

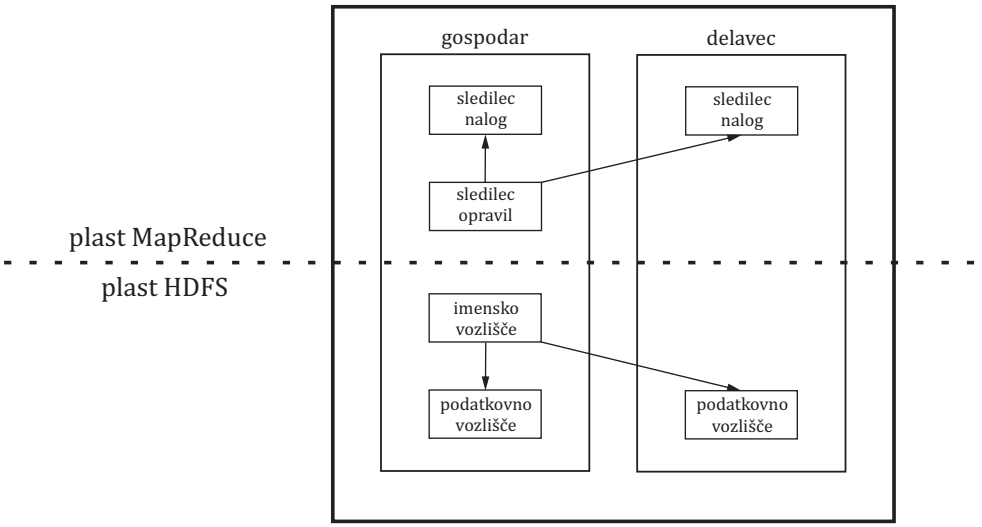
Hadoop MapReduce lahko med razporejanjem nalog uporabi imensko vozlišče, da čim bolj razporedi naloge med naprave, kjer je mogoč lokalen dostop do podatkov. Posledično se zmanjša čas za prenos podatkov preko omrežja. Obdelava poteka na isti napravi, kjer so shranjeni podatki. Delavec je vozlišče za obdelavo in shranjevanje podatkov. Ker HDFS hrani več reproduciranih blokov, se poveča verjetnost, da ima naprava, ki hrani reproducirane bloke, prosto kapaciteto za izvajanje nalog in tudi v primeru okvar je blok razpoložljiv.

Hadoop MapReduce vodi evidenco ključev. Poskrbi, da vsak delavec, ki opravlja nalogo *reduce*, vidi ključe v urejenem vrstnem redu, prav tako vidi njihove vrednosti. Vsak delavec naenkrat obdela en ključ.

Hadoop MapReduce nudi programske vmesnike, ki omogočajo kodiranje funkcij *map* in *reduce* ne samo v programskem jeziku Java, temveč tudi v drugih programskih jezikih, na primer C++, Ruby in Python.

4.1.3 Demoni

Demon je program, ki teče kot proces v ozadju. V Hadoop MapReduce tečeta dva večja demona. To sta sledilec opravil (*jobtracker*) in sledilec nalog (*tasktracker*). Gruča Hadoop z več vozlišči je prikazana na sliki 4.1.



Slika 4.1: Gruča Hadoop z več vozlišči.

Sledilec opravil je gospodar, odgovoren za sprejemanje programerjevih predložitev opravil in razporejanje nalog delavskim vozliščem. Zagotavlja tudi administrativne funkcije, na primer stanje delavcev in spremljanje napredka nalog v gruči. V vsaki gruči MapReduce je samo en sledilec opravil.

Drugi pomemben demon v Hadoop MapReduce je sledilec nalog. Ta je odgovoren za sprejemanje nalog, ki mu jih dodeli sledilec opravil, izvedbo dodeljenih nalog in periodično sporočanje napredka sledilcu opravil. Na vsakem delavskem vozlišču je en sledilec nalog. Vsak sledilec nalog je konfiguriran, da opravlja določeno število nalog *map* in *reduce* paralelno.

4.1.4 Okvare

Najhujša okvara je odpoved sledilca opravil, saj ne obstaja mehanizem za obravnavanje te okvare. Predstavlja kritično točko odpovedi, vendar je verjetnost, da pride do odpovedi, zelo majhna. Ko se sledilec opravil ponovno zažene, morajo biti vse naloge, ki so se izvajale ob okvari, ponovno posredovane. Obstaja nastavitev, ki omogoča ponovno vzpostavitev trenutnih opravil, vendar je znano, da ni zanesljiva.

Tudi okvara datotečnega sistema HDFS je resna. Če so vsa podatkovna vozlišča, ki vsebujejo blok, v okvari in če imensko vozlišče ne more najti podatkovnih vozlišč, na katere bi zapisalo blok, se naloga ne zaključi uspešno. V primeru odpovedi imenskega vozlišča bodo naloge neuspešne pri poskusu vzpostavitve naslednjega stika z njim. Izvajalno okolje bo ponovno poskusilo zagnati te naloge, vendar bodo poskusi neuspešni in opravilo bo neuspešno. Nadaljnje nova opravila ne bodo posredovana v gručo.

Če je sledilec nalog nekaj določenih časovnih intervalov neodziven, sledilec opravil predpostavi, da je neaktiven skupaj z dodeljenimi nalogami. Naloge so prerazporejene. Izvedejo se na drugem sledilcu nalog. Na aplikacijo ta okvara nima posebnega vpliva, le opravilo traja dalj časa.

4.2 MongoDB

MongoDB je dokumentno orientirana podatkovna baza NoSQL [8, 9]. MongoDB tako kot običajna splošno namenska podatkovna baza nudi kreiranje, branje, po-

sodabljanje in brisanje podatkov. Poleg tega nudi še indeksiranje, agregacijo in posebne tipe zbirk. Znotraj agregacije je zajeta tudi podpora za MapReduce.

NoSQL je pojem, ki pokriva različne tipe tehnologij za shranjevanje podatkov, ki se uporabljajo takrat, ko poslovni model ne ustreza klasičnemu relacijskemu podatkovnemu modelu [9]. Tehnologije NoSQL nudijo višjo razpoložljivost, skalabilnost in zmogljivost ter omogočajo delo z velikimi množicami podatkov.

4.2.1 Predstavitev MongoDB

Pri MongoDB je v nasprotju z relacijsko podatkovno bazo koncept vrstice zamenjan s prilagodljivejšim modelom, dokumentom, ki omogoča predstavitev kompleksnih hierarhičnih razmerij z enim zapisom. Dokument je urejena množica ključev, ki vsebuje njihove pripadajoče vrednosti. Je osnovna enota za podatke in se uporablja tudi pri komunikaciji med procesi. Ključi in vrednosti dokumenta niso fiksnih tipov ali velikosti. Skupina dokumentov tvori zbirko. Pri MongoDB ni vnaprej definiranih shem, zato je dodajanje ali odstranjevanje polj preprostejše.

Dokumentno orientiran podatkovni model omogoča lažjo delitev podatkov med več strežnikov. MongoDB samodejno poskrbi za naslednje stvari: izenačevanje podatkov in obremenitev preko gruče, ponovno samodejno porazdelitev dokumentov in usmerjanje zahtev pravim napravam. Programerji se lahko posledično osredotočijo izključno na programiranje aplikacij. Če moramo gruči povečati kapaciteto, vanjo dodamo nove naprave. MongoDB samodejno ugotovi, kako naj bodo med naprave razdeljeni obstoječi podatki.

V MongoDB poznamo številne tipe indeksov. Indeksi se lahko ustvarijo na vsakem polju znotraj dokumenta. Priporočljivo je ustvariti indekse, ki podpirajo pogoste poizvedbe [3]. Vsi indeksi v MongoDB so indeksi B-drevesa in se uporabljajo pri branju. Za branje se uporablja naključni in zaporedni dostop. Vsak strežnik ima globalno zaklepanje za pisanje. Za pisanje se uporabljajo dnevniški (*journaled*) zapisi v indeksne in podatkovne bloke [12]. Podatki so shranjeni v pomnilniku vsakega vozlišča.

MongoDB nudi programske vmesnike za najpopularnejše programske jezike, med drugim za programske jezike C, C++, Java in Python.

4.2.2 Uporaba MapReduce v MongoDB

V MongoDB je MapReduce prilagodljivo okolje za agregacijo podatkov [8]. Z njim lahko rešimo probleme, ki so preveč kompleksni, da bi jih lahko izrazili z agregacijskim okoljem povpraševalnega jezika. MapReduce kot povpraševalni jezik uporablja programski jezik JavaScript. Z njim lahko izrazimo poljubno kompleksno logiko. Agregacijsko okolje omogoča, da dokumente v zbirki preoblikujemo in kombiniramo.

MongoDB ne vodi evidence ključev v vsakem dokumentu, saj predpostavlja, da je shema dinamična. Najboljši način, da najdemo vse ključe v vseh dokumentih v zbirki, je uporaba MapReduce.

Toleranco pri okvarah in zanesljivost lahko zagotovimo s točkami kontrole, tako da vsak delavec periodično poroča osrednjemu strežniku MongoDB svoje stanje [12]. Za reproduciranje poskrbimo tako, da ustvarimo množico replik (*replica set*). Množica replik je skupina strežnikov, sestavljena iz primarnega (*primary*) strežnika in sekundarnih (*secondary*) strežnikov. Primarni strežnik sprejema zahteve, sekundarni strežniki pa hranijo kopijo podatkov primarnega strežnika.

4.3 Knjižnica MapReduce-MPI

Knjižnica MapReduce-MPI je odprtokodna programska oprema in implementacija MapReduce na podlagi standarda za prenašanje sporočil MPI [6]. MPI je standardna knjižnica, ki temelji na soglasju foruma MPI Forum [14]. Forum želi postaviti temelje za prenosen, učinkovit in prilagodljiv standard, ki bi se uporabljal v različnih programskih jezikih. MPI se šteje za industrijski standard.

4.3.1 Lastnosti knjižnice MapReduce-MPI

Knjižnica MapReduce-MPI zagotavlja enostaven vmesnik, ki programerjem omogoča pisanje programov MapReduce. Zasnovana je za paralelno izvedbo na enem procesorju ali več računalniških okoljih s porazdeljenim pomnilnikom. Knjižnica je napisana v programskem jeziku C++ in nudi vmesnike za programske jezike C++, C, Fortran in Python. Sledijo lastnosti knjižnice MapReduce-MPI.

Uporaba MPI za komunikacijo med procesi

Knjižnica uporablja MPI za komunikacijo med procesi. Uporaba MPI omogoča nadzor nad pomnilnikom, ki je dodeljen med izvajanjem opravila MapReduce.

Majhnost in prenosljivost

Celotna knjižnica obsega nekaj tisoč vrstic programske kode C++ v nekaj datotekah. To omogoča namestitev na vsaki napravi, ki vsebuje prevajalnik C++. Pri paralelnih operacijah se knjižnica MPI-MapReduce poveže s knjižnico MPI, pri zaporednih operacijah pa je knjižnica MPI nadomeščena z navidezno.

Operacije v jedru in zunaj jedra

Procesorju dodeli vsak ustvarjen objekt MapReduce strani (*pages*) v pomnilniku. Velikost strani določi programer. Množico podatkov sestavljajo pari ključ-vrednost. Če množica podatkov ne presega velikosti strani, knjižnica opravi operacije v jedru. V nasprotnem primeru procesorji podatke po potrebi zapišejo v začasne datoteke na disk in potem berejo iz njih. Dostop do teh datotek je zaporeden.

4.3.2 Omejitve

Knjižnica MapReduce-MPI nima tolerance za okvare. MPI pri okvarah ravna tako, da prekine opravilo. Če je procesor v okvari, se paralelni program, ki kliče knjižnico MapReduce-MPI, prekine. Knjižnica MapReduce-MPI ne poskrbi za reprodukcijo podatkov [22].

Funkcija *reduce* prinaša veliko omejitev, ki zajemajo tudi vhodne podatke:

- funkcija *reduce* mora biti asociativna, saj za redukcijske operacije MPI v splošnem predpostavljamo, da so asociativne [18],
- vsak proces mora poznati število različnih ključev, vrednosti vseh ključev morajo biti fiksne velikosti in
- če vsi procesi nimajo vrednosti za vsak ključ, potrebujemo element za identifikacijo zaradi funkcije *reduce*, ki ga morajo zagotoviti ti procesi brez vrednosti.

4.4 Primerjava implementacij MapReduce

V tabeli 4.1 so zbrane karakteristike implementacij MapReduce.

KARAKTERI-STIKE	**Hadoop MapReduce**	**MongoDB**	**Knjižnica MapReduce-MPI**
reproduciranje	da	da	ne
shranjevanje podatkov	porazdeljen datotečni sistem HDFS	porazdeljena podatkovna baza brez shem, podatki so shranjeni v pomnilniku vsakega vozlišča	strani v pomnilniku; če podatki presežejo velikost strani, jih procesor zapiše v diskovne datoteke na lokalni disk ali paralelni datotečni sistem
branje	zaporedni dostop do blokov	naključni in zaporedni dostop, indeksi B-drevesa	zaporedni dostop do diskovnih datotek izven jedra
pisanje	podatki so shranjeni lokalno, nato so poslani imenskemu vozlišču	dnevniški zapisi v indeksne in podatkovne bloke, vsak strežnik ima globalno zaklepanje za pisanje	zaporedni dostop do diskovnih datotek izven jedra
toleranca za okvare	da	da	ne

vodenje evidence ključev	da	ne	da
komunikacija med procesi	klici za oddaljene procedure	dokumenti	MPI
programski vmesniki	Java, C++, Ruby, Python	C, C++, Java, Python	C++, C, Python

Tabela 4.1: Primerjava implementacij MapReduce.

Pri implementaciji Hadoop MapReduce je HDFS optimiziran za zaporedna branja in pisanja podatkov. Na drugi strani je MongoDB optimiziran za naključni in paralelni dostop. To so predvsem podatkovne poizvedbe. Rezultati so pokazali, da MongoDB pokaže slabo izvedbo pri paralelnih zapisih zaradi globalnega zaklepanja za pisanje [12]. Pri knjižnici MapReduce-MPI se za branje in pisanje datotek izven jedra uporablja zaporedni dostop.

Knjižnica MapReduce-MPI ne reproducira podatkov. Na drugi strani HDFS in MongoDB reproducirata podatke in s tem zagotavljata zanesljivost. HDFS reproducira podatke v številu, ki je privzeto ali pa ga definira programer. Pri MongoDB programer definira število reprodukcij s tem, da določi število elementov v množici replik.

Implementaciji Hadoop MapReduce in knjižnica MapReduce-MPI vodita evidenco ključev. Na drugi strani MongoDB ne vodi evidence ključev, saj predpostavlja, da je shema dinamična. Hadoop MapReduce uporablja za komunikacijo med procesi klice za oddaljene procedure, MongoDB dokumente in knjižnica MapReduce-MPI MPI.

Implementaciji Hadoop MapReduce in MongoDB nista občutljivi na okvare. MongoDB ima točke kontrole, Hadoop MapReduce pa razne načine ravnanja ob odpovedih. Problematični sta le odpoved sledilca opravil in okvara HDFS. Nasprotno je knjižnica MapReduce-MPI občutljiva na okvare. Če je procesor v okvari, se paralelni program, ki kliče knjižnico MapReduce-MPI, prekine.

Vse implementacije imajo programske vmesnike za najpopularnejše programske jezike.

Poglavje 5

Priporočilni sistemi

V poglavjih 6 in 7 smo se osredotočili na primere priporočilnih sistemov, zato bomo to področje tudi na kratko predstavili. V prvem izbranem primeru priporočilnega sistema iščemo predloge za nove prijatelje v socialnem omrežju. V drugem izbranem primeru priporočilnega sistema iščemo filme za uporabnike, ki jih še niso videli, in so najbolj podobni tistim, ki so jih že gledali.

Priporočilni sistemi so programska orodja in tehnike, ki zagotavljajo predloge za predmete, ki jih bo uporabnik uporabljal [15]. Uporabniku med drugim lahko predlagamo, katere filme naj gleda, kakšno glasbo naj posluša in katere knjige naj bere. Predlogi so povezani s procesi odločanja. Predmeti so objekti z različno kompleksnimi karakteristikami in vrednostjo, ki se priporočajo. Uporabniki priporočilnega sistema so si lahko med sabo zelo različni. Priporočilni sistem za prilagoditev priporočil uporablja informacije o uporabniku, vendar je količina uporabljenih informacij odvisna od tehnike.

Poznamo več tipov priporočilnih sistemov. Ločimo jih glede na tehnike, ki jih uporabljajo za določanje predlogov. Danes sta najbolj razširjeni dve tehniki. Prvo imenujemo izbiranje s sodelovanjem (*collaborative filtering*), druga pa temelji na vsebini (*content-based*).

5.1 Izbiranje s sodelovanjem

Izbiranje s sodelovanjem uporabniku priporoča predmete, ki so jih v preteklosti najbolje ocenili uporabniki s podobnimi preferencami. Podobnost v preferencah

med dvema uporabnikoma temelji na njuni zgodovini ocen. Če sta si zgodovini ocen podobni, sta si podobna tudi uporabnika.

Metode izbiranja s sodelovanjem lahko razdelimo v dva razreda. V prvi razred spadajo metode, ki temeljijo na uporabniku (*user-based methods*), in v drugi metode, ki temeljijo na modelu (*model-based methods*). Metode, ki temeljijo na uporabniku, poskušajo najti množico uporabnikov z enakimi preferencami kot ciljni uporabnik. Ko se takšna množica uporabnikov najde, se njihove preference kombinirajo, da se predvidi preference ciljnega uporabnika. Metode, ki temeljijo na modelu, najprej razvijejo model uporabnikovih ocen. Pogosto za določitev uporabnikovih preferenc uporabljajo verjetnostni pristop.

Izbiranje s sodelovanjem pogosto apliciramo tudi na MapReduce, saj ni skalabilno in ima visoko računsko zahtevnost, če je množica podatkov velika. Pri izbiranju s sodelovanjem lahko skalabilnost dosežemo z uporabo MapReduce. Še več, če pri metodi, ki temelji na uporabnikih, vhodne podatke (podatke o uporabnikih) razdelimo v enako velike skupine in jih beremo v skupinah, se naloge končajo ob približno istem času, čas izvajanja pa se povečuje linearno s številom uporabnikov [25]. Izbiranje s sodelovanjem na MapReduce apliciramo tako, da v fazi *map* iz vhodnih podatkov izračunamo podobnosti in predvidene ocene uporabnikov za predmete. V fazi *reduce* izračunane predvidene ocene le še uredimo.

Portal za iskanje novic Google News uporablja za generiranje priporočil izbiranje s sodelovanjem, aplicirano na MapReduce [13]. Priporočila temeljijo na zgodovini pregledanih člankov aktivnega uporabnika in zgodovini pregledanih člankov celotne skupnosti uporabnikov.

5.1.1 Prednosti

Izbiranje s sodelovanjem prinaša številne prednosti:

- **Enostavnost:** Metode, ki temeljijo na izbiranju s sodelovanjem, so intuitivne. Prav tako jih ni težko izvesti.
- **Utemeljenost predvidene ocene:** Vse izračunane predvidene ocene so izračunane iz ocen predmetov, ki so bili v preteklosti ocenjeni. Uporabniku ocene predmetov utemeljijo izračunano predvideno oceno, kar mu omogoča lažje razumevanje priporočila in njegove pomembnosti.

- **Učinkovitost:** Metode, ki temeljijo na izbiranju s sodelovanjem, ne zahtevajo zahtevnih učnih faz. Kljub temu da je faza, v kateri se iščejo priporočila, zahtevna, lahko podobnosti med uporabniki izračunamo predčasno. Njihovo shranjevanje zavzame malo prostora. To zagotavlja skalabilnost.

- **Stabilnost:** Na metode, ki temeljijo na tem pristopu, malo vplivajo dodajanje predmetov, uporabnikov in ocen. Podobnosti, ki so bile izračunane pred tem, lahko uporabimo pri izračunu predvidenih ocen. Na novo je potrebno izračunati le podobnosti med novim uporabnikom in predhodnimi uporabniki v sistemu.

5.1.2 Slabosti

Izbiranje s sodelovanjem prinaša med drugim tudi slabosti:

- **Omejena pokritost:** Pri izračunu predvidene ocene za predmet upoštevamo le podobnosti uporabnikov, ki so ocenili isti predmet, vendar lahko imata uporabnika enake preference, četudi sta ocenila samo nekaj ali nič enakih predmetov. Pokritost metod je lahko omejena, saj so lahko priporočeni samo predmeti, ki so jih ocenili drugi uporabniki.

- **Občutljivost na redke podatke:** Če je na voljo malo ocen, je natančnost metod slaba. Redki podatki so problem pri večini priporočilnih sistemov, saj uporabniki običajno ocenijo le manjši del predmetov, ki so na voljo. V tem primeru je malo verjetno, da imata dva uporabnika ali predmeta pogoste ocene. V izračun predvidene ocene je vključeno omejeno število uporabnikov. Podobno je tudi v izračun podobnosti vključeno malo število ocen, zato priporočila niso reprezentativna. Če so predmeti dodani na novo, niti nimajo ocen. Ta problem imenujemo hladen začetek (*cold-start*).

5.2 Tehnika, ki temelji na vsebini

Pri tehniki, ki temelji na vsebini, so uporabnikovi interesi opisani s karakteristikami predmetov in uporabnikovim profilom [13]. Priporočila so opredeljena z določitvijo predmetov, ki se najbolj ujemajo z uporabnikovimi interesi. Proces

določitve predmetov potrebuje dodatne informacije o uporabnikih in predmetih, a ne zahteva obstoja velike skupnosti uporabnikov ali zgodovine ocen.

Sistemi, ki uporabljajo tehniko, ki temelji na vsebini, najprej analizirajo množico opisov predmetov, ki jih je uporabnik že ocenil. Nato zgradijo profil uporabnikovih interesov. Profil temelji na karakteristikah predmetov, ki jih je predhodno ocenil uporabnik, in se uporablja pri priporočanju novih predmetov. Novi predmeti se priporočijo tako, da se karakteristike uporabnikovega profila primerjajo s karakteristikami predmeta. Ta primerjava da kot rezultat uporabnikovo raven interesa za predmet.

5.2.1 Prednosti

Tehnika, ki temelji na vsebini, prinaša naslednje prednosti:

- **Uporabnikova neodvisnost:** Tehnike, ki temeljijo na vsebini, za izgradnjo uporabnikovega profila izkoriščajo le uporabnikove ocene, kar je nasprotno kot pri izbiranju s sodelovanjem, kjer so za izgradnjo uporabnikovega profila potrebne ocene drugih uporabnikov.

- **Jasnost:** Delovanje priporočilnega sistema lahko opišemo z navajanjem opisov, ki so povzročili, da se je predmet pojavil v seznamu priporočil. Ti opisi so pokazatelji, ali lahko priporočilu zaupamo. Nasprotno pri izbiranju s sodelovanjem pojasnitev za priporočilo izhaja izključno iz ocen uporabnikov s podobnimi preferencami.

- **Priporočanje novega dodanega predmeta:** Tehnike, ki temeljijo na vsebini, lahko priporočijo predmet, ki ga še ni ocenil noben uporabnik. Posledično niso občutljive na oceno prvega uporabnika, kar je nasprotno kot pri izbiranju s sodelovanjem.

5.2.2 Slabosti

Tehnika, ki temelji na vsebini, prinaša tudi nekatere slabosti.

- **Površna vsebinska analiza:** Površna vsebinska analiza je prisotna takrat, ko pri določanju priporočil upoštevamo premalo karakteristik predmetov.

Na primer, če so predmeti spletne strani in če upoštevamo samo njihovo vsebino, vemo, da na njihovo zanimivost ne vpliva samo vsebina, temveč tudi estetika, uporabnost, pravočasnost in pravilnost povezav.

- **Prevelika podobnost priporočenih predmetov:** Pri tehnikah, ki temeljijo na vsebini, se lahko hitro zgodi, da priporočijo samo predmete, ki so zelo podobni predmetom, ki jih je uporabnik že pozitivno ocenil. Takšen učinek je nezaželen, saj so lahko priporočeni predmeti preveč podobni tistim, ki jih uporabnik že pozna. Primer takšnega priporočilnega sistema je priporočilni sistem, ki filtrira novice, in priporoči časopisni članek, ki pokriva enako zgodbo, kot jo je uporabnik že videl v drugačnem kontekstu. Rešitev za ta problem je vključitev nepričakovanih predmetov, ki bi uporabnika morda zanimali, v priporočila poleg pričakovanih, za uporabnika nezanimivih predmetov.

- **Oteženo pridobivanje ocen:** Tudi pri tehnikah, ki temeljijo na vsebini, se pojavlja problem hladnega začetka, vendar v drugačni obliki. Te tehnike zahtevajo najmanj začetno množico uporabnikovih ocen. Lahko se zgodi, da uporabnik v veliko domenah ne bo želel oceniti minimalnega števila predmetov. Da se temu izognemo, od uporabnika zahtevamo, da z izbiranjem ali vnašanjem besed zagotovi seznam ključnih besed.

Poglavje 6

Primer priporočilnega sistema - iskanje predlogov za prijatelje v socialnem omrežju

6.1 Opredelitev problema

Napišimo program MapReduce za implementacijo algoritma, ki predlaga prijatelje v socialnem omrežju [2]. Sistem naj predlaga, da se dve osebi povežeta, če imata veliko skupnih prijateljev.

Dana je vhodna datoteka[1], ki ima 49995 vrstic in vsebuje seznam sosednosti. V vsaki vrstici je najprej naveden uporabnik, za njim pa seznam uporabnikovih prijateljev, ki je od uporabnika ločen s tabulatorjem. Uporabnik je predstavljen z enoličnim identifikatorjem. Vsak prijatelj v seznamu uporabnikovih prijateljev je prav tako predstavljen z enoličnim identifikatorjem. Prijatelji v seznamu so med sabo ločeni z vejico.

Želimo napisati enostaven algoritem, ki za vsakega uporabnika U predlaga deset prijateljev. Predlagani uporabniki z uporabnikom U še niso povezani, z njim pa imajo največ skupnih prijateljev.

Izhodna datoteka naj za vsakega uporabnika vsebuje eno vrstico. V vsaki

[1]Datoteka je dostopna na povezavi http://snap.stanford.edu/class/cs246-data/hw1q1.zip.

vrstici naj bo najprej naveden uporabnik. Za njim naj bo seznam priporočil, ki je od uporabnika ločen s tabulatorjem. Seznam priporočil vsebuje enolične identifikatorje uporabnikov, ki jih je algoritem predlagal. Urejeni naj bodo po padajočem vrstnem redu glede na število skupnih prijateljev in če jih je manj kot deset, naj bodo po istem kriteriju prav tako izpisani vsi. V primeru, da imajo predlagani uporabniki enako število skupnih prijateljev, naj bodo le-ti razvrščeni po naraščajočem vrstnem redu glede na enolične identifikatorje. Elementi seznama priporočil naj bodo med sabo ločeni z vejico.

6.2 Izbira implementacije MapReduce

Za reševanje problema je pri obeh izbranih primerih priporočilnih sistemov izbrana implementacija Hadoop MapReduce, saj nudi toleranco za okvare in reproducira podatke. Na drugi strani knjižnica MapReduce-MPI ne nudi tolerance za okvare in ne reproducira podatkov, s čimer ne zagotavlja zanesljivosti, zato ni najprimernejša izbira. MongoDB tako kot Hadoop MapReduce nudi toleranco za okvare in reproducira podatke, vendar pa v primerjavi s Hadoop MapReduce pokaže slabo izvedbo pri paralelnih zapisih zaradi globalnega zaklepanja za pisanje.

Uporabljena je navidezna naprava QuickStart VM [1], ki predstavlja gručo Hadoop z enim vozliščem. Obdelava poteka v psevdoporazdeljenem načinu, kar pomeni, da je porazdeljena na vse procesorje v napravi. QuickStart VM ima nameščeno odprtokodno računalniško okolje Hadoop CDH 5, ki vključuje MapReduce 2.0.

6.3 Rešitev problema

Problem rešimo z enim opravilom MapReduce. Za branje podatkov smo nastavili razred KeyValueTextInputFormat, ki poskrbi, da funkcija *map* dobi iz vsake vrstice v vhodni datoteki v obdelavo pare ključ-vrednost tipa Text. Ključ predstavlja uporabnikov enolični identifikator, vrednost pa seznam njegovih prijateljev, ločenih z vejico.

Funkcija *map* iz prejetega seznama prijateljev določi predloge za nove prijatelje za vsakega uporabnika v seznamu. Vsak uporabnik v seznamu prijateljev

ima z vsakim drugim uporabnikom v seznamu prijateljev enega skupnega prijatelja. Ta je ključ, ki ga funkcija *map* prejme. Za vsakega uporabnika v seznamu prijateljev so torej predlogi za nove prijatelje vsi drugi uporabniki v seznamu prijateljev. Tako so vmesni ključi vsi enolični identifikatorji uporabnikov v seznamu prijateljev, pripadajoče vmesne vrednosti pa so vsi enolični identifikatorji uporabnikov iz seznama, brez tistega, ki predstavlja vmesni ključ. Za vmesni ključ določi tudi uporabnikov enolični identifikator, ki je ključ, in za vmesno vrednost njegov označeni seznam prijateljev. Označen seznam prijateljev je namenjen filtriranju že obstoječih prijateljev iz predlogov v fazi *reduce*.

Funkcija *reduce* prejme vmesni ključ in seznam vmesnih vrednosti. Vmesni ključ predstavlja uporabnikov enolični identifikator. Funkcija *reduce* iz seznama vmesnih vrednosti prešteje priporočila in izloči uporabnike, ki so že povezani. Priporočila se ustrezno uredijo glede na to, kolikokrat se pojavijo. Funkcija *reduce* jih deset skupaj z uporabnikovim enoličnim identifikatorjem zapiše v izhodno datoteko.

6.3.1 Funkcija *map*

Funkcija *map* prejme par ključ-vrednost. Ključ predstavlja uporabnikov enolični identifikator, vrednost pa seznam njegovih prijateljev, ki so med sabo ločeni z vejico. Seznam prijateljev razčleni. Iz njega določi vmesne ključe in vmesne vrednosti.

Vsak uporabnik v seznamu prijateljev ima z vsakim drugim uporabnikom v seznamu prijateljev enega skupnega prijatelja. Ta je ključ, ki ga funkcija *map* prejme. Za vsakega uporabnika v seznamu prijateljev so torej predlogi za nove prijatelje vsi drugi uporabniki v seznamu prijateljev. Recimo, da je v seznamu prijateljev n uporabnikov. Vsak vmesni ključ je i-ti uporabnik v seznamu prijateljev, kjer i teče od $i = 1, ..., n$, pripadajočo vmesno vrednost pa sestavljajo vsi j-ti uporabniki v seznamu prijateljev za $j = 1, ..., n$ in $j \neq i$. Izsek programske kode odstavka je prikazan na sliki 6.1.

Za vmesni ključ določi tudi uporabnikov enolični identifikator, ki predstavlja ključ, in za vmesno vrednost njegov označeni seznam prijateljev.

```
if(friends != null && friends.length > 1){
   for(int i=0; i<friends.length; i++){
      String intermediateValue = "";
      for(int j=0; j<friends.length; j++){
         if(i != j){
            intermediateValue += friends[j] + ",";
         }
      }
      int intermediateKey = Integer.parseInt(friends[i]);
      intermediateValue = intermediateValue.substring(0,
                                    intermediateValue.length() - 1);
      context.write(new IntWritable(intermediateKey),
                                       new Text(intermediateValue));
   }
}
```

Slika 6.1: Izsek programske kode za določanje predlogov za nove prijatelje.

6.3.2 Funkcija *reduce*

Funkcija *reduce* prejme vmesni ključ in seznam vmesnih vrednosti. Vmesni ključ predstavlja uporabnikov enolični identifikator, seznam vmesnih vrednosti pa tvorijo nizi, ki vsebujejo predloge. Funkcija *reduce* v zanki vse vmesne vrednosti v seznamu, razen označene, shrani v nov seznam. Označeno vmesno vrednost shrani ločeno, saj je namenjena izločevanju tistih uporabnikov, ki so z uporabnikom, ki predstavlja vmesni ključ, že povezani.

Funkcija *reduce* v zanki razčleni uporabnike iz vsakega elementa v seznamu. Za vsakega uporabnika preveri, če je v označeni vmesni vrednosti. V primeru, da ni, ga shrani v nov seznam in na pripadajočo pozicijo v drug seznam število ena. Prvi seznam torej hrani priporočila, drugi pa pripadajoča števila skupnih prijateljev. Če se uporabnik ponovi večkrat, se v seznamu s števili skupnih prijateljev povečuje njihovo pripadajoče število. Izsek programske kode odstavka je prikazan na sliki 6.2.

Funkcija *reduce* seznam priporočil in seznam s števili skupnih prijateljev pre-

```
ArrayList<String> recommendations = new ArrayList<String>(900);
ArrayList<Integer> numberOfMutualFriends =
                                          new ArrayList<Integer>(900);
for(i=0; i<suggestions.size(); i++){
   String[] suggestionsTmp = suggestions.get(i).split(",");
   for(int j=0; j<suggestionsTmp.length; j++){
       if(!friends.contains(suggestionsTmp[j])){
         if(recommendations.contains(suggestionsTmp[j])){
            int index = recommendations.indexOf(suggestionsTmp[j]);
            int element = (int) numberOfMutualFriends.get(index) + 1;
            numberOfMutualFriends.set(index, element);
          }
          else{
            recommendations.add(suggestionsTmp[j]);
            numberOfMutualFriends.add(1);
          }
       }
    }
}
```

Slika 6.2: Izsek programske kode za shranjevanje uporabnikov in števil skupnih prijateljev v nova seznama.

tvori v tabeli. Če tabeli vsebujeta več kot en element, funkcija *reduce* kliče drugo funkcijo, ki tabelo s skupnimi števili prijateljev uredi z algoritmom za urejanje z zlivanjem po velikosti od največje vrednosti proti najmanjši vrednosti, skladno z njo pa preuredi tudi tabelo priporočil. Nato funkcija *reduce* preveri, če so med prvimi desetimi elementi v tabel is skupnimi števili prijateljev enaki vnosi. Pri tem prav tako upošteva tudi elemente, ki sledijo desetemu, samo če so enaki desetemu. Funkcija *reduce* kliče funkcijo, ki uporabnike v tabeli, ki so na enakih pozicijah kot enaki elementi v tabeli s skupnimi števili prijateljev, uredi po velikosti od najmanjše proti največji vrednosti z algoritmom za urejanje z zlivanjem. Izsek programske kode odstavka je prikazan na sliki 6.3.

```
recommendations.toArray(users);
numberOfMutualFriends.toArray(numberOfFriends);
if(users.length > 1){
   reverseMergeSort(users, numberOfFriends,
                                        0, numberOfFriends.length-1);
   int firstValue = numberOfFriends[0];
   int index = 1;
   for(i=1; i<users.length; i++){
      int currentValue = numberOfFriends[i];
      if(firstValue == currentValue){
         index++;
         if(i == users.length-1){
            mergeSort(users, i-index+1, i);
         }
      }
      else{
         if(index > 1){
            mergeSort(users, i-index, i-1);
         }
         if(i > 9){
            break;
         }
         index = 1;
         firstValue = currentValue;
      }
   }
}
```

Slika 6.3: Izsek programske kode za urejanje tabele priporočil in tabele s števili skupnih prijateljev.

Nato se v izhodno datoteko zapiše ključ, ki je enak vmesnemu ključu, ter niz prvih desetih vrednosti iz tabele priporočil. Vrednosti so v nizu med sabo ločene

z vejico.

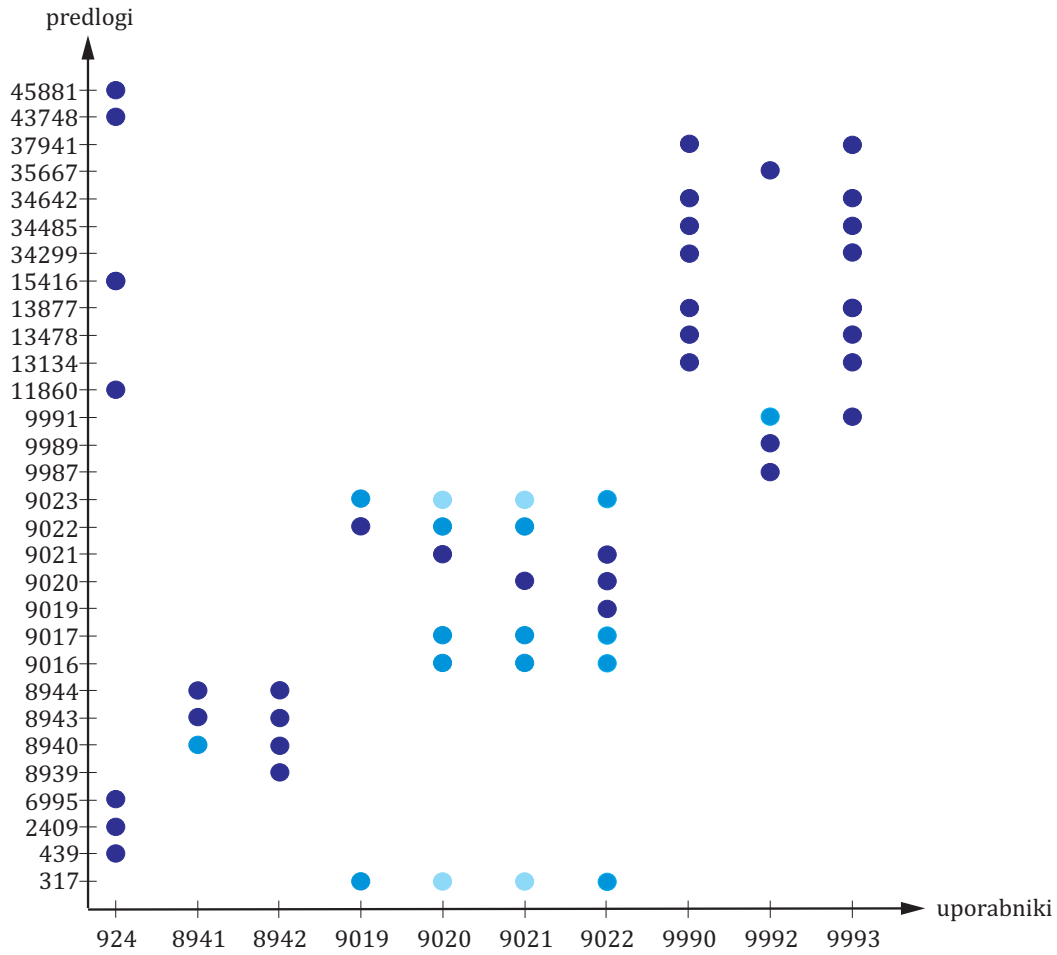
6.3.3 Validacija

Avtorji naloge v navodilih za nalogo [2] navajajo enolične identifikatorje 10 uporabnikov, ki naj se uporabijo za validacijo. Enolični identifikatorji teh uporabnikov so 924, 8941, 8942, 9019, 9020, 9021, 9022, 9990, 9992 in 9993. Uporabnike smo poiskali v izhodni datoteki in njihove predloge za prijatelje primerjali s predlogi za prijatelje, ki jih avtorji naloge navajajo v rešitvah [4]. Rezultati se v celoti ujemajo in so predstavljeni z grafom na sliki 6.4. Enak odtenek modre barve pri enem uporabniku označuje, da imajo predlogi za prijatelje enako število skupnih prijateljev z uporabnikom. Temnejši odtenek modre označuje večje število skupnih prijateljev in pomeni, da se predlogi za prijatelje v izhodni datoteki nahajajo v seznamu predlogov prej, svetel odtenek modre pa označuje manjše število skupnih prijateljev in pomeni, da se predlogi v izhodni datoteki nahajajo v seznamu predlogov za prijatelje kasneje.

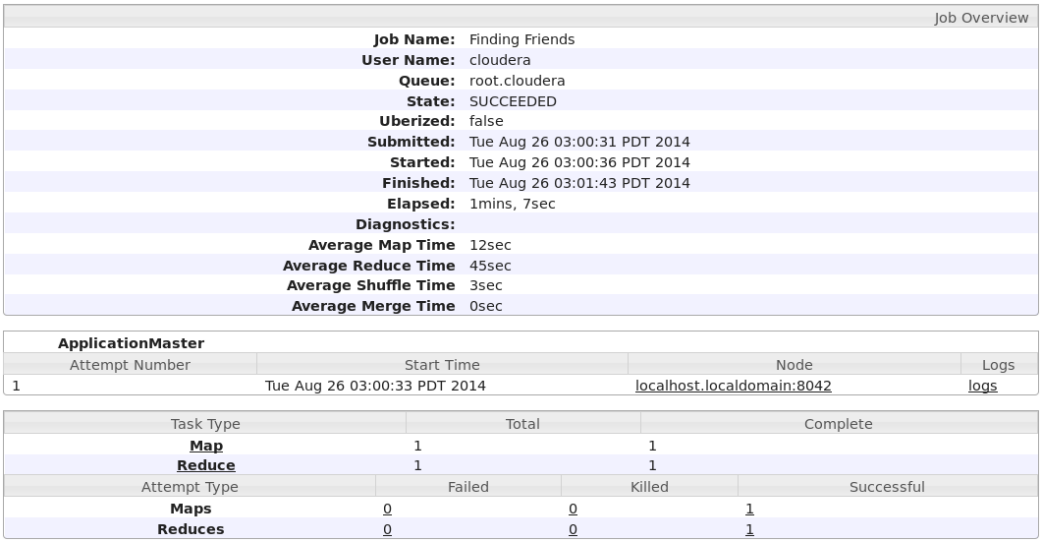
6.3.4 Izvajanje

Za izvajanje programa poskrbi poseben razred. V njem se iz argumentov prebereta ime vhodne datoteke in ime mape, ki se bo ustvarila in v katero bo shranjena izhodna datoteka. Ustvari se še opravilo, kateremu se nastavijo parametri. Ti so ime, datoteka JAR glede na razred, pot do vhodne datoteke, pot do mape za izhod, razred KeyValueTextInputFormat za branje vhodnih podatkov, razred, ki vsebuje funkcijo *map*, razred, ki vsebuje funkcijo *reduce*, tip vmesnega ključa in vmesne vrednosti za funkcijo *map* ter tip ključa in vrednosti za funkcijo *reduce*.

Program smo izvajali v virtualni napravi Cloudera QuickStart VM z operacijskim sistemom CentOS. Dodeljena sta ji 2 procesorja z 1 jedrom in 5.6 GB bralno-pisalnega pomnilnika (RAM). Virtualna naprava je nameščena na računalniku z operacijskim sistemom Windows, ki ima bralno-pisalni pomnilnik s kapaciteto 8 GB, procesor Intel, ki ima največjo hitrost 2.40 GHz, 2 jedri in 4 logične procesorje. Kopija zaslona po uspešno zaključenem opravilu je prikazana na sliki 6.5, na sliki 6.6 pa so prikazani natančni podatki o izvajanju opravila.



Slika 6.4: Grafični prikaz rezultatov.



Job Overview

Job Name:	Finding Friends
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Tue Aug 26 03:00:31 PDT 2014
Started:	Tue Aug 26 03:00:36 PDT 2014
Finished:	Tue Aug 26 03:01:43 PDT 2014
Elapsed:	1mins, 7sec
Diagnostics:	
Average Map Time	12sec
Average Reduce Time	45sec
Average Shuffle Time	3sec
Average Merge Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Aug 26 03:00:33 PDT 2014	localhost.localdomain:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Slika 6.5: Slika zaslona po uspešno zaključenem opravilu.

```
File System Counters
   FILE: Number of bytes read=137463388
   FILE: Number of bytes written=207972331
   FILE: Number of read operations=0
   FILE: Number of large read operations=0
   FILE: Number of write operations=0
   HDFS: Number of bytes read=4156297
   HDFS: Number of bytes written=2952721
   HDFS: Number of read operations=6
   HDFS: Number of large read operations=0
   HDFS: Number of write operations=2
Job Counters
   Launched map tasks=1
   Launched reduce tasks=1
   Data-local map tasks=1
   Total time spent by all maps in occupied slots (ms)=3662336
   Total time spent by all reduces in occupied slots (ms)=13062912
```

```
   Total time spent by all map tasks (ms)=14306
   Total time spent by all reduce tasks (ms)=51027
   Total vcore-seconds taken by all map tasks=14306
   Total vcore-seconds taken by all reduce tasks=51027
   Total megabyte-seconds taken by all map tasks=3662336
   Total megabyte-seconds taken by all reduce tasks=13062912
Map-Reduce Framework
   Map input records=49995
   Map output records=707316
   Map output bytes=140103637
   Map output materialized bytes=69779942
   Input split bytes=116
   Combine input records=0
   Combine output records=0
   Reduce input groups=49995
   Reduce shuffle bytes=69779942
   Reduce input records=707316
   Reduce output records=49995
   Spilled Records=2121948
   Shuffled Maps =1
   Failed Shuffles=0
   Merged Map outputs=1
   GC time elapsed (ms)=1369
   CPU time spent (ms)=60160
   Physical memory (bytes) snapshot=475095040
   Virtual memory (bytes) snapshot=1788420096
   Total committed heap usage (bytes)=328204288
Shuffle Errors
   BAD_ID=0
   CONNECTION=0
   IO_ERROR=0
   WRONG_LENGTH=0
   WRONG_MAP=0
```

```
   WRONG_REDUCE=0
File Input Format Counters
   Bytes Read=4156181
File Output Format Counters
   Bytes Written=2952721
```

Slika 6.6: Natančni podatki o izvajanju opravila.

Poglavje 7

Primer priporočilnega sistema - iskanje predlogov za filme

7.1 Opredelitev problema

Želimo napisati program MapReduce za implementacijo priporočilnega sistema, ki temelji na izbiranju s sodelovanjem in uporabnikom predlaga filme, ki jih še niso videli in so najbolj podobni tistim, ki so jih že gledali.

Vhodni podatki so zbrani v datoteki zip[1] in vsebujejo 100000 ocen za 1682 filmov od 943 uporabnikov, podatke o uporabnikih ter podatke o filmih.

Želimo napisati enostaven priporočilni sistem, ki vsakemu uporabniku predlaga deset filmov, ki jih še ni videl in so najbolj podobni tistim, ki jih je že gledal.

Izhodna datoteka naj za vsakega uporabnika vsebuje eno vrstico. V vsaki vrstici naj bo najprej naveden uporabnik. Za njim naj bo seznam filmov, ki je od uporabnika ločen s tabulatorjem. Uporabnik je predstavljen z enoličnim identifikatorjem, prav tako film. Seznam filmov vsebuje enolične identifikatorje filmov, ki jih je algoritem predlagal. Urejeni naj bodo po padajočem vrstnem redu glede na predlagane ocene. V primeru, da ima več filmov enako oceno, naj bodo le-ti razvrščeni po naraščajočem vrstnem redu glede na enolične identifikatorje. Elementi seznama filmov naj bodo med sabo ločeni z vejico.

[1]Datoteka je dostopna na povezavi http://grouplens.org/datasets/movielens/.

7.2 Rešitev problema

Rešitev temelji na izbiranju s sodelovanjem, in sicer na metodi, ki temelji na uporabnikih [25]. Metoda, ki temelji na uporabnikih, poskuša najti množico uporabnikov, v kateri imajo uporabniki podobne preference kot ciljni uporabnik [7]. Ko je določena množica sosednjih uporabnikov, se njihove preference kombinirajo, da se predvidijo preference ciljnega uporabnika. Torej ciljni uporabnik, ki je podobno ocenil enake filme kot drugi uporabniki, ima rad podobne filme kot drugi uporabniki in bo podobno kot oni ocenil filme, ki jih bo gledal.

Problem je rešen z enim opravilom MapReduce. Izvajalno okolje najprej kliče funkcijo *setup*, ki sestavi matriko ocen in v tabelo shrani aritmetično sredino ocen filmov za vsakega uporabnika. Vsaka celica v matriki ocen vsebuje uporabnikovo oceno za specifičen film. Če uporabnik filma še ni gledal, je celica prazna.

Funkcija *map* dobi iz vhodne datoteke v obdelavo po meri določene pare ključ-vrednost. Ključ je tipa NullWritable. Vrednost je niz, ki vsebuje enolične identifikatorje uporabnikov iz datoteke, ki vsebuje podatke o uporabnikih. Funkcija *map* izračuna ocene za vse filme, ki jih uporabnik še ni gledal, po naslednjem postopku.

Naj bo $r_{i,j}$ ocena i-tega uporabnika za j-ti film. Predvideno oceno ciljnega uporabnika a za j-ti film $\widehat{r}_{a,j}$ izračunamo z enačbo:

$$\widehat{r}_{a,j} = \overline{r_a} + \tau \sum_{i=1}^{n} s_{a,i}(r_{i,j} - \overline{r_i}), \tag{7.1}$$

kjer $\overline{r_a}$ predstavlja aritmetično sredino ocen ciljnega uporabnika a, $\overline{r_i}$ aritmetično sredino ocen i-tega uporabnika, $s_{a,i}$ podobnost med uporabnikom a in uporabnikom i, pri čemer konstanto τ izračunamo z enačbo

$$\tau = (\sum_{i=1}^{n} |s_{a,i}|)^{-1}. \tag{7.2}$$

Aritmetična sredina se izračuna za filme, ki jih je uporabnik ocenil. Vsota se nanaša samo na tiste uporabnike, ki so ocenili j-ti film. $s_{a,i}$ meri podobnost preferenc med uporabnikoma a in i in je pozitivna, če uporabnika rada gledata enake filme, sicer je negativna.

Za predvideno oceno ciljnega uporabnika $\widehat{r}_{a,j}$ se kombinirajo relativne ocene vseh uporabnikov, ki so ocenili j-ti film. Prispevek vsakega uporabnika pa je drugačen, saj ga določa podobnost med uporabnikom a in uporabnikom i $s_{a,i}$.

Če sta si uporabnika podobna, je prispevek večji, sicer pa je manjši, tako da uporabnik z drugačnim okusom ne prispeva veliko k predvideni oceni filma ciljnega uporabnika. Podobnost določa Pearsonov korelacijski koeficient in ga izračunamo z enačbo

$$s_{a,i} = \frac{\sum_j (r_{a,j} - \overline{r_a})(r_{i,j} - \overline{r_i})}{\sqrt{\sum_j (r_{a,j} - \overline{r_a})^2 \sum_j (r_{i,j} - \overline{r_i})^2}}. \tag{7.3}$$

Pearsonov korelacijski koeficient se računa samo pri tistih filmih, ki sta jih ocenila oba uporabnika. Predvidena ocena filma je lahko nezanesljiva, če je film ocenilo malo uporabnikov.

Funkcija *map* za vmesni ključ določi enolični identifikator uporabnika, za vmesno vrednost pa niz, sestavljen iz enoličnega identifikatorja filma, dvopičja in predvidene ocene. Funkcija *reduce* ustrezno uredi podatke, dobljene iz vmesnih vrednosti, in v izhodno datoteko v vsako vrstico zapiše uporabnika ter seznam s predlogi desetih filmov.

7.2.1 Branje podatkov

Pare ključ-vrednost, namenjene funkciji *map*, ustvarimo po meri. Ključ je tipa NullWritable. Vrednost je niz, sestavljen iz enoličnih identifikatorjev uporabnikov, ločenih z vejico.

Iz razreda RecordReader, ki podatke razdeli v pare ključ-vrednost, smo izpeljali razred MyReader. Povozili smo metode razreda RecordReader. Te so close, getCurrentKey, getCurrentValue, getProgress, initialize in nextKeyValue. Med naštetimi metodami sta pomembni predvsem slednji. Metoda initialize je klicana enkrat pri inicializaciji. Metoda nextKeyValue določi naslednjo vrednost. Prebere 50 vrstic iz vhodne datoteke, ki vsebuje podatke o uporabnikih, in iz vsake vrstice razčleni uporabnikov enolični identifikator ter ga shrani v niz. Izsek programske kode odstavka je prikazan na sliki 7.1.

Iz razreda FileInputReader smo izpeljali razred MyInputReader. Povozili smo funkcijo createRecordReader, ki ustvari objekt RecordReader in jo razred FileInputFormat deduje iz razreda InputFormat.

```
if(position > 942){
   return false;
}
value.clear();
Text line = new Text();
String string = "";
for(int i=0; i<50; i++){
   int bytesRead = in.readLine(line);
   if(bytesRead == 0){
      break;
   }
   String line2 = line.toString();
   String[] data = line2.split("\\|");
   string += data[0] + ",";
   position++;
}
string = string.substring(0, string.length()-1);
Text string2 = new Text(string);
value.append(string2.getBytes(), 0, string2.getLength());
return true;
```

Slika 7.1: Izsek programske kode za branje podatkov.

7.2.2 Funkcija *setup*

Izvajalno okolje najprej kliče funkcijo *setup*. Funkcija *setup* sestavi matriko ocen iz vhodne datoteke, ki vsebuje podatke o ocenah filmov uporabnikov. Iz vrstice razčleni uporabnikov enolični identifikator, enolični identifikator filma in uporabnikovo oceno za film. Ker enolični identifikatorji uporabnikov tečejo od 1 do 943 in enolični identifikatorji filmov od 1 do 1682, uporabnikovo oceno za film shrani na mesto v matriki ocen, ki je na preseku vrstice, ki je enaka za ena zmanjšanemu uporabnikovemu enoličnemu identifikatorju, in stolpca, ki je enak za ena zmanjšanemu enoličnemu identifikatorju filma. Za vsakega uporabnika sproti sešteva ocene ter njihovo število. Ko je matrika ocen sestavljena, izračuna še aritmetične sredine

ocen filmov uporabnikov in jih shrani v tabelo. Do tabele z aritmetičnimi sredinami ocen filmov in do matrike ocen lahko med izvajanjem dostopa funkcija *map*.

7.2.3 Funkcija *map*

Funkcija *map* prejme par ključ-vrednost. Ključ je tipa NullWritable. Vrednost je niz, ki vsebuje enolične identifikatorje uporabnikov, ločene z vejico. Funkcija *map* razčleni niz in v zanki za vsakega uporabnika izračuna predvidene ocene za filme.

Najprej določi pozicijo uporabnikove vrstice v matriki ocen s tem, da od njegovega enoličnega identifikatorja odšteje ena. V zanki najde filme, ki jih uporabnik še ni ocenil. Ko naleti na prvi neocenjeni film, njegovo predvideno oceno izračuna v treh korakih.

V prvem koraku iz matrike, ki vsebuje podobnosti, pridobi vse izračunane podobnosti uporabnika s tistimi uporabniki, ki imajo manjši enolični identifikator, torej tistimi, ki so v matriki ocen in matriki, ki vsebuje podobnosti med uporabniki, nad njim in so ocenili ciljni film. Za tem sešteje absolutne vrednosti podobnosti med uporabniki ter produkte podobnosti med uporabniki z razliko ocene filma in aritmetične sredine ocen filmov tistega uporabnika, ki je film že ocenil. Izsek programske kode odstavka je prikazan na sliki 7.2.

```
for(int j=0; j<positionOfUser; j++){
   if(matrix[j][k] != 0){
      absSum += Math.abs(similarities[positionOfUser][j]);
      sumOfProducts += (matrix[j][k] - average[j]) *
                       similarities[positionOfUser][j];
   }
}
```

Slika 7.2: Izsek programske kode za seštevanje absolutnih vrednosti podobnosti med uporabniki ter produktov podobnosti med uporabniki z razliko ocene filma in aritmetične sredine ocen filmov.

Funkcija *map* v drugem koraku z enačbo (7.3) izračuna še podobnosti ciljnega

uporabnika z uporabniki, ki so v matriki ocen pod njim. Izračunane podobnosti med ciljnim uporabnikom in drugimi uporabniki shrani v matriko podobnosti. Če je drugi uporabnik ocenil film, k vsoti absolutnih vrednosti prišteje absolutno vrednost podobnosti in k vsoti produktov prišteje produkt podobnosti z razliko ocene filma in aritmetične sredine ocen filmov drugega uporabnika. Izsek programske kode odstavka je prikazan na sliki 7.3.

V tretjem koraku izračuna uporabnikovo predvideno oceno za film in jo ustrezno zaokroži. Če je predvidena ocena manjša od 1.5, jo zaokroži na 1, če je večja ali enaka 4.5, jo zaokroži na 5, sicer pa na najbližje celo število. Za vmesni ključ določi uporabnikov enolični identifikator, za vmesno vrednost pa niz, sestavljen iz enoličnega identifikatorja filma, dvopičja in zaokrožene predvidene ocene. Izsek programske kode odstavka je prikazan na sliki 7.4.

Če funkcija *map* računa predvideno uporabnikovo oceno za film, ki sledi prvemu neocenjenemu filmu, sta za izračun predvidene ocene potrebna le dva koraka. Prvi korak je podoben prvemu koraku pri izračunu ocene za prvi neocenjeni film s to razliko, da v zanki števec teče od prve do zadnje vrstice v matriki, saj so podobnosti že izračunane. Drugi korak pa je enak tretjemu koraku pri izračunu ocene za prvi neocenjeni film.

7.2.4 Funkcija *reduce*

Funkcija *reduce* prejme vmesni ključ in seznam vmesnih vrednosti. Vmesni ključ je uporabnik. Seznam vmesnih vrednosti tvorijo nizi oblike <film>:<ocena>, kjer <film> predstavlja enolični identifikator filma in <ocena> pripadajočo oceno za film.

Funkcija *reduce* v zanki vmesne vrednosti iz seznama loči glede na dvopičje. Filme shrani v nov seznam, ocene filmov pa v drug seznam, tako da so na enakih pozicijah v seznamih filmi in njihove ocene.

Seznam filmov in seznam ocen pretvori v tabeli. Če tabeli vsebujeta več kot en element, funkcija *reduce* kliče drugo funkcijo, ki tabelo ocen uredi z algoritmom za urejanje z zlivanjem po velikosti od največje vrednosti proti najmanjši vrednosti, skladno z njo pa preuredi tudi tabelo filmov. Nato funkcija *reduce* preveri, če so med prvimi desetimi elementi v tabeli ocen enaki vnosi. Pri tem prav tako upošteva tudi elemente, ki sledijo desetemu, samo če so enaki desetemu. Funkcija

```
for(int j=positionOfUser+1; j<matrix.length; j++){
   double sum = 0;
   double sumOfSquares1 = 0;
   double sumOfSquares2 = 0;
   for(int l=0; l<1682; l++){
      if(matrix[j][l] != 0 && matrix[positionOfUser][l]!=0){
         double part1 = matrix[positionOfUser][l] -
                        average[positionOfUser];
         double part2 = matrix[j][l] - average[j];
         sum += part1*part2;
         sumOfSquares1 += part1*part1;
         sumOfSquares2 += part2*part2;
      }
   }
   double similarity = sum/Math.sqrt(sumOfSquares1*sumOfSquares2);
   similarities[positionOfUser][j] = similarity;
   similarities[j][positionOfUser] = similarity;
   if(matrix[j][k] != 0){
      absSum += Math.abs(similarities[positionOfUser][j]);
      sumOfProducts += (matrix[j][k] - average[j]) *
                        similarities[positionOfUser][j];
   }
}
```

Slika 7.3: Izsek programske kode za izračun podobnosti med uporabniki, seštevanje absolutnih vrednosti podobnosti med uporabniki ter produktov podobnosti med uporabniki z razliko ocene filma in aritmetične sredine ocen filmov.

reduce kliče funkcijo, ki filme v tabeli, ki so na enakih pozicijah kot enaki elementi v tabeli ocen, uredi po velikosti od najmanjše proti največji vrednosti z algoritmom za urejanje z zlivanjem.

V izhodno datoteko zapiše ključ, ki je enak vmesnemu ključu, ter niz prvih

```
double rating = average[positionOfUser] + sumOfProducts/absSum;
int rating2 = 0;
if(rating >= 4.5){
   rating2 = 5;
}else if(rating < 1.5){
   rating2 = 1;
}
else{
   rating2 = (int) Math.round(rating);
}
String string = Integer.toString(k+1) + ":" +
                Integer.toString(rating2);
context.write(new IntWritable(positionOfUser+1), new Text(string));
```

Slika 7.4: Izsek programske kode za izračun in zaokrožitev predvidene ocene.

desetih vrednosti iz tabele filmov. Vrednosti so v nizu med sabo ločene z vejico.

7.2.5 Validacija

Validacijo pravilnosti delovanja smo preverili na primeru iskanja predlogov za filme. Kot referenco smo vzeli ročno rešen primer. Njegovo rešitev smo primerjali z rešitvijo primera, rešenega z računalnikom.

V tabeli 7.1 so zbrane ocene petih uporabnikov za deset filmov. Iščemo ocene za filme, ki jih uporabniki še niso gledali in so najbolj podobni tistim, ki so jih že gledali.

Najprej izračunamo aritmetične sredine ocen filmov uporabnikov:
$\overline{r_1} = \frac{5+3+4+5+5+1+5}{7} = 4$, $\overline{r_2} = \frac{2+1+1+2+3+2+4+2}{8} = 2,125$, $\overline{r_3} = \frac{4+3+3+2+4+3+2}{7} =$
$= 3$, $\overline{r_4} = \frac{4+1+3+2+4+3+2+2}{8} = 2,625$ in $\overline{r_5} = \frac{3+1+3+2+1+3+1}{7} = 2$.

Nato izračunamo podobnosti med uporabniki z uporabo Pearsonovega korelacijskega koeficienta. Izračunati moramo le $s_{m,n}$, kjer m teče od 1 do 4 in n teče od $m+1$ do 5, saj je $s_{m,n} = s_{n,m}$. Velja $s_{n,n} = 1$. Slednje podobnosti pri nadaljnjih izračunih ne potrebujemo.

Uporabniki \ Filmi	1	2	3	4	5	6	7	8	9	10
1	5	3		4	5		5	1		5
2	2	1	1		2		3	2	4	2
3		4	3	3		2	4		3	2
4	4			1	3	2	4	3	2	2
5	3	1	3		2	1		3	1	

Tabela 7.1: Ocene uporabnikov za filme.

Označimo $s_{m,n} = \frac{a_{m,n}}{\sqrt{b_{m,n} \cdot c_{m,n}}}$, kjer je $a_{m,n} = \sum_j (r_{m,j} - \overline{r_m}) \cdot (r_{n,j} - \overline{r_n})$, $b_{m,n} = \sum_j (r_{m,j} - \overline{r_m})^2$ in $c_{m,n} = \sum_j (r_{n,j} - \overline{r_n})^2$.

Sledi postopek za izračun podobnosti med uporabnikoma 1 in 2. Vsi nadaljnji rezultati izračunov so v primeru, da obsegajo več kot pet decimalnih mest, zaokroženi na pet decimalnih mest natančno.

$$\begin{aligned}
a_{1,2} &= (5-4)\cdot(2-2,125) + (3-4)\cdot(1-2,125) + (5-4)\cdot(2-2,125) + \\
&\quad + (5-4)\cdot(3-2,125) + (1-4)\cdot(2-2,125) + (5-4)\cdot(2-2,125) = \\
&\quad = 2 \\
b_{1,2} &= (5-4)^2 + (3-4)^2 + (5-4)^2 + (5-4)^2 + (1-4)^2 + (5-4)^2 = 14 \\
c_{1,2} &= (2-2,125)^2 + (1-2,125)^2 + (2-2,125)^2 + (3-2,125)^2 + \\
&\quad + (2-2,125)^2 + (2-2,125)^2 = 2,09375 \\
s_{1,2} &= \frac{a_{1,2}}{\sqrt{b_{1,2} \cdot c_{1,2}}} = \frac{2}{\sqrt{14 \cdot 2,09375}} \doteq 0,36941
\end{aligned}$$

Podobno izračunamo druge podobnosti. Njihove vrednosti so zbrane v spodnji matriki.

$$s_{i,j} = \begin{bmatrix}
1 & 0,36941 & -0,33333 & 0,14318 & -0,16667 \\
0,36941 & 1 & -0,02762 & -0,03406 & -0,38497 \\
-0,33333 & -0,02762 & 1 & 0,63462 & 0 \\
0,14318 & -0,03406 & 0,63462 & 1 & 0,87287 \\
-0,16667 & -0,38497 & 0 & 0,87287 & 1
\end{bmatrix}$$

Uporabniki \ Filmi	1	2	3	4	5	6	7	8	9	10
1	5	3	3	4	5	5	5	1	5	5
2	2	1	1	2	2	3	3	2	4	2
3	4	4	3	3	3	2	4	4	3	2
4	4	2	3	1	3	2	4	3	2	2
5	3	1	3	1	2	1	2	3	1	2

Tabela 7.2: Ocene in izračunane predvidene ocene uporabnikov za filme.

Za tem izračunamo predvidene ocene uporabnikov za filme, ki jih še niso gledali. Pri računanju uporabimo enačbi (7.2) in (7.1). Sledi izračun predvidene ocene šestega filma za prvega uporabnika.

$$\begin{aligned}
\tau &= (|s_{1,3}| + |s_{1,4}| + |s_{1,5}|)^{-1} = \\
&= (|-0,33333| + |0,14318| + |-0,16667|)^{-1} \doteq \\
&\doteq 1,55477 \\
\widehat{r}_{1,6} &= \overline{r_1} + \tau \sum_{i=3}^{5} s_{1,i}(r_{i,6} - \overline{r_i}) = \\
&= 4 + 1,55477 \cdot (-0,33333 \cdot (2-3) + 0,14318 \cdot (2-2,625) - \\
&- 0,16667 \cdot (1-2)) \doteq \\
&\doteq 4,63825
\end{aligned}$$

Izračunano predvideno ocena $\widehat{r}_{1,6}$ zaokrožimo na 5. Podobno izračunamo še manjkajoče predvidene ocene, ki so zbrane v tabeli 7.2 in izpisane z modro barvo.

Na koncu določimo še sezname s predlogi filmov. Uporabniku 1 predlagamo filme 6, 9 in 3, uporabniku 2 filma 6 in 4, uporabniku 3 filme 1, 8, in 5, uporabniku 4 filma 3 in 2 ter uporabniku 5 filme 7, 10 in 4.

Isti primer smo reševali še z računalnikom. Na sliki 7.5 je prikazana vsebina izhodne datoteke. Obe rešitvi primera se v celoti ujemata.

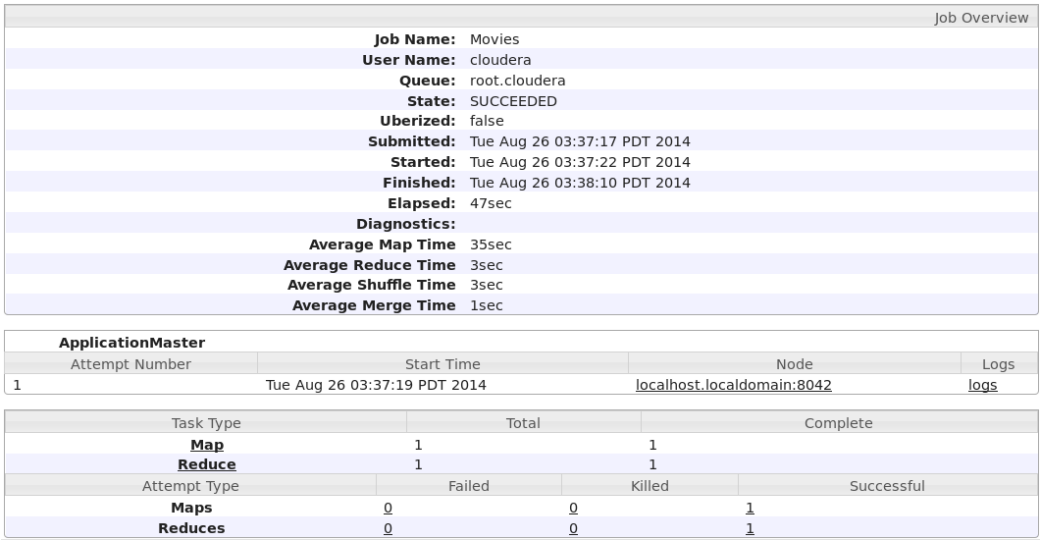
```
1    6,9,3
2    6,4
3    1,8,5
4    3,2
5    7,10,4
```

Slika 7.5: Vsebina izhodne datoteke.

7.2.6 Izvajanje

Za izvajanje programa poskrbi poseben razred. V njem se iz argumentov prebereta ime vhodne datoteke in ime mape, ki se bo ustvarila in v katero bo shranjena izhodna datoteka. Ustvari se še opravilo, kateremu se nastavijo parametri. Ti so ime, datoteka JAR glede na razred, pot do vhodne datoteke, pot do mape za izhod, po meri napisan razred za branje vhodnih podatkov, razred, ki vsebuje funkcijo *map*, razred, ki vsebuje funkcijo *reduce*, tip vmesnega ključa in vmesne vrednosti za funkcijo *map* ter tip ključa in vrednosti za funkcijo *reduce*.

Program smo izvajali v virtualni napravi Cloudera QuickStart VM z operacijskim sistemom CentOS. Dodeljena sta ji 2 procesorja z 1 jedrom in 5.6 GB bralno-pisalnega pomnilnika. Virtualna naprava je nameščena na računalniku z operacijskim sistemom Windows, ki ima bralno-pisalni pomnilnik s kapaciteto 8 GB, procesor Intel, ki ima največjo hitrost 2.40 GHz, 2 jedri in 4 logične procesorje. Kopija zaslona po uspešno zaključenem opravilu je prikazana na sliki 7.6. Na sliki 7.7 so prikazani natančni podatki o izvajanju opravila.



Job Overview

Job Name:	Movies
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Tue Aug 26 03:37:17 PDT 2014
Started:	Tue Aug 26 03:37:22 PDT 2014
Finished:	Tue Aug 26 03:38:10 PDT 2014
Elapsed:	47sec
Diagnostics:	
Average Map Time	35sec
Average Reduce Time	3sec
Average Shuffle Time	3sec
Average Merge Time	1sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Aug 26 03:37:19 PDT 2014	localhost.localdomain:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Slika 7.6: Slika zaslona po uspešno zaključenem opravilu.

```
File System Counters
   FILE: Number of bytes read=6592890
   FILE: Number of bytes written=13369483
   FILE: Number of read operations=0
   FILE: Number of large read operations=0
   FILE: Number of write operations=0
   HDFS: Number of bytes read=2001914
   HDFS: Number of bytes written=32793
   HDFS: Number of read operations=7
   HDFS: Number of large read operations=0
   HDFS: Number of write operations=2
Job Counters
   Launched map tasks=1
   Launched reduce tasks=1
   Data-local map tasks=1
   Total time spent by all maps in occupied slots (ms)=9147392
   Total time spent by all reduces in occupied slots (ms)=1859072
```

```
   Total time spent by all map tasks (ms)=35732
   Total time spent by all reduce tasks (ms)=7262
   Total vcore-seconds taken by all map tasks=35732
   Total vcore-seconds taken by all reduce tasks=7262
   Total megabyte-seconds taken by all map tasks=9147392
   Total megabyte-seconds taken by all reduce tasks=1859072
Map-Reduce Framework
   Map input records=19
   Map output records=1486126
   Map output bytes=15412070
   Map output materialized bytes=6592886
   Input split bytes=113
   Combine input records=0
   Combine output records=0
   Reduce input groups=943
   Reduce shuffle bytes=6592886
   Reduce input records=1486126
   Reduce output records=943
   Spilled Records=2972252
   Shuffled Maps =1
   Failed Shuffles=0
   Merged Map outputs=1
   GC time elapsed (ms)=848
   CPU time spent (ms)=38100
   Physical memory (bytes) snapshot=508039168
   Virtual memory (bytes) snapshot=1805029376
   Total committed heap usage (bytes)=317718528
Shuffle Errors
   BAD_ID=0
   CONNECTION=0
   IO_ERROR=0
   WRONG_LENGTH=0
   WRONG_MAP=0
```

```
    WRONG_REDUCE=0
File Input Format Counters
    Bytes Read=22628
File Output Format Counters
    Bytes Written=32793
```

Slika 7.7: Natančni podatki o izvajanju opravila.

Poglavje 8

Zaključek

Cilj diplomskega dela je analizirati MapReduce in ga preizkusiti na dveh primerih priporočilnih sistemov. Cilj smo dosegli, saj smo uspeli realizirati izračun s pomočjo MapReduce na testnih primerih v dveh korakih. V prvem koraku smo analizirali programski model in izvajalno okolje ter primerjali tri implementacije MapReduce: Hadoop MapReduce, MongoDB in knjižnico MapReduce-MPI. Ugotovili smo, da je za realizacijo izbranih primerov priporočilnih sistemov najprimernejša implementacija Hadoop MapReduce, saj nudi toleranco za okvare in reproducira podatke, s čimer zagotavlja zanesljivost. V drugem koraku smo z uporabo navidezne naprave Cloudera QuickStart VM, ki je gruča Hadoop z enim vozliščem, realizirali izbrana primera priporočilnih sistemov.

Programski model MapReduce omogoča paralelizacijo, je neobčutljiv na okvare in enostaven. Programer izrazi izračun s funkcijama *map* in *reduce*. Funkcija *map* vzame kot vhod par ključ-vrednost ter vrne kot izhod množico vmesnih parov ključ-vrednost. Izvajalno okolje združi vse vmesne vrednosti z istim vmesnim ključem v skupino in jih posreduje funkciji *reduce*, ki sprejme vmesni ključ in pripadajočo množico vmesnih vrednosti ter jo združi v manjšo množico, ki ima nič ali več elementov.

Obstajajo še možnosti za izboljšave in nadaljnje raziskovanje. Izbrana primera priporočilnih sistemov bi lahko preizkusili na gruči z več vozlišči in rešili tudi z implementacijama MongoDB in knjižnice MapReduce-MPI. Pri primeru priporočilnega sistema, v katerem iščemo predloge za filme, bi lahko izboljšali natančnost napovedovanja. Za izračun podobnosti bi lahko med drugim uporabili

kosinusno podobnost ali Spearmanov korelacijski koeficient ter poskusili določiti optimalno število elementov v množici uporabnikov s podobnimi preferencami, da bi bila natančnost napovedovanja še učinkovitejša.

Literatura

[1] Cloudera QuickStart VM. `http://www.cloudera.com/content/dev-center/en/home/developer-admin-resources/quickstart-vm.html`. Zadnjič dostopano 6. 7. 2014.

[2] Navodila za praktični primer. `http://web.stanford.edu/class/cs246/homeworks/hw1.pdf`. Zadnjič dostopano 20. 3. 2014.

[3] Priročnik za MongoDB 2.6. `http://docs.mongodb.org/manual/`. Zadnjič dostopano 19. 5. 2014.

[4] Rešitve za praktični primer. `http://web.stanford.edu/class/cs246/homeworks/hw1_solutions.zip`. Zadnjič dostopano 20. 3. 2014.

[5] Spletna stran implementacije Hadoop. `http://hadoop.apache.org/`. Zadnjič dostopano 21. 8. 2014.

[6] Uporabniški priročnik knjižnjice MapReduce-MPI. `http://mapreduce.sandia.gov/doc/Manual.pdf`. Zadnjič dostopano 18. 5. 2014.

[7] Robert C. Blattberg, Byung-Do Kim, and Scott A. Neslin. *Database Marketing: Analyzing and Managing Customers*. Springer, 2008.

[8] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly, O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, Second Edition, 2013.

[9] Hector Cuesta. *Practical Data Analysis*. Packt Publishing, Packt Publiching Ltd., Livery Place, 35 Livery Street, Birmingham B32PB, UK, First Edition, October 2013.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design an Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, October 2004.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Communications of the ACM - Amir Pnueli: Ahead of His Time*, 53(1):72–77, January 2010.

[12] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, Science Cloud '13, pages 13–20, New York, NY, USA, 2013. ACM.

[13] Alexander Felfernig Gerhard Friedrich Dietmar Jannach, Markus Zanker. *Recommender Systems: An Introduction*. Cambridge University Press, Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2011.

[14] Rob Farber. *CUDA Application Design and Development*. Morgan Kaufman, 225 Wyman Street, Waltham, MA 02451, USA, 2011.

[15] Bracha Shapira Francesco Ricci, Lior Rokach and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

[16] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, Second Edition, January 2003.

[17] Christian Gross. *Beginning C# 2008: From Novice to Professional*. Apress, Second Edition, 2008.

[18] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards Efficient MapReduce Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th PVM/MPI Users' Group Meeting*, pages 492–499, Esspo, Finland, September 2009. Springer.

[19] Ralf Lämmel. Google's MapReduce Programming Model - Revisited. *Science of Computer Programming*, 70(1):1–30, January 2008.

[20] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel Data Processing with MapReduce: a Survey. *ACM SIGMOD Record*, 40(4):11–20, December 2011.

[21] Jimmy Lin and Chris Dyer. *Data-Intensitive Text Processing with MapReduce.* Synthesis Lectures on Human Language Technologies. Morgan & Clayton Publishers, 2010.

[22] Steven J. Plimpton and Karen D. Devine. Mapreduce in {MPI} for Large-Scale Graph Algorithms. *Parallel Computing*, 37(9):610 – 632, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[23] Eric Sammer. *Hadoop Operations.* O'Reilly, O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, First Edition, September 2012.

[24] Tom White. *Hadoop: The Definitive Guide.* O'Reilly, O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, Second Edition, 2011.

[25] Zhi-Dan Zhao and Ming-Sheng Shang. User-Based Collaborative-Filtering Recommendation Algorithms on Hadoop. In *Knowledge Discovery and Data Mining, 2010. WKDD '10. Third International Conference on*, pages 478–481, Januar 2010.