2014

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA

ZAKLJUČNA NALOGA

DATOTEČNI SISTEM HADOOP

JAKOMIN

TOMAŽ JAKOMIN

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga

Datotečni sistem Hadoop

(Hadoop file system)

Ime in priimek: Tomaž Jakomin
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Iztok Savnik

Koper, september 2014

Ključna dokumentacija informacija

Ime in PRIIMEK: Tomaž JAKOMIN

Naslov zaključne naloge: Datotečni sistem Hadoop

Kraj: Koper

Leto: 2014

Število listov: 53 Število slik: 15 Število tabel: 1

Število referenc: 21

Mentor: doc. dr. Iztok Savnik

Ključne besede: Hadoop, HDFS, masovni podatki, gruča, MapReduce.

Izvleček: V velikih podjetjih nastaja ogromna količina podatkov, za katere moramo pri analizi uporabiti prave tehnologije, če želimo pridobiti prave informacije. Za hranjene in obdelavo takšnih velikih količin podatkov, ki jim rečemo tudi masovni podatki, uporabljamo programsko okolje Hadoop. Hadoopov porazdeljen datotečni sistem se imenuje HDFS. Omogoča nam shranjevanje podatkov na več računalnikih in lahko teče na vsakem računalniku, kateri podpira Java programski jezik. Bloki, v katere shranjujemo podatke, so v primerjavi z drugimi datotečnimi sistemi večji in datoteke, ki jih shranjujemo, so lahko večje od trdega diska, saj se lahko bloki določene datoteke porazdelijo po drugih diskih znotraj gruče. V HDFS imamo dva tipa vozlišč in to sta imensko in podatkovno. Imensko vozlišče je nadrejeno in upravlja z imenskim prostorom, medtem ko podatkovno vozlišče shranjuje in posreduje bloke na zahtevo. Za varnost podatkov je prav tako poskrbljeno, saj vsebuje več varnostnih mehanizmov. Programski model znotraj jedra platforme se imenuje MapReduce. Omogoča nam obdelavo podatkov s porazdeljenim delom. Tako lahko uporabimo vse računalnike v gruči, da lahko hitreje pridemo do rezultata. MapReduce je razdeljen na dve operaciji, ki sta Map in Reduce. Map operacija obdela vrednosti po določenem ključu, ki jih nato poda Reduce operaciji, katera združuje vrednosti po določenem ključu. Za lažje pisanje programov, ki temeljijo na tehnologiji MapReduce, je nastal programski jezik Pig Latin. Omogoča nam dodatne operacije za obdelavo podatkov. Nazadnje, kar si lahko preberemo v zaključni nalogi, je praktičen Mapreduce primer, ki sešteva pojavitve besed v zbirki knjig.

Key words documentation

Name and SURNAME: Tomaž JAKOMIN

Title of the final project paper: Hadoop file system

Place: Koper

Year: 2014

Number of pages: 53 Number of figures: 15 Number of tables: 1

Number of references: 21

Mentor: Assist. Prof. Iztok Savnik, PhD

Keywords: Hadoop, HDFS, BigData, cluster, Map, Reduce

Abstract: Big companies generate massive amount of data every day calling it Big Data. For analyzing that amount we need the right technology that will bring us the right information. The technology that can store and analyze Big Data is Hadoop. It uses a distributed file system, which name is HDFS that allows storing data on different machines that support Java. Blocks in HDFS are much bigger in comparison with other file systems. They can be stored on any machine in the cluster. This allows storing files larger than any hard disk in the cluster. HDFS has a master/slave architecture. An HDFS cluster consists of a single Name Node, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are usually one per node Data Nodes in the cluster, which manage the storage attached to the node. In case of any failures, it has many security mechanisms. Inside the core of the programming environment is located the programming model called MapReduce. It is used for processing and generating large data sets with distributed algorithm on a cluster. A MapReduce program is composed of a Map operation and a Reduce operation. Programming language Pig Latin was created for simplifying writing of MapReduce programs. It allows many additional operations for processing data that are not included by default in MapReduce. In the last section is described a simple MapReduce program that counts how often words occur in one collection of books.

ZAHVALA

Želel bi se iskreno zahvaliti svojemu mentorju doc. dr. Iztoku Savniku, ki me je usmerjal in mi dajal napotke skozi celotno izdelavo zaključne naloge.

Posebno se zahvaljujem staršema, ki sta mi dajala finančno in moralno podporo skozi celoten študij.

KAZALO VSEBINE

KAZALO PREGLEDNIC

KAZALO SLIK

SEZNAM KRATIC

API	Application programming interface
FUSE	File system in userspace
GPS	Global positioning system
HDFS	Hadoop Dsitributed File System
HTML	HyperText Markup Language
IBM	IBM podjetje
IDC	International data corporation
JAR	Java Archive
MB	Megabajt
NDFS	Nutch porazdeljen datotečni sistem
NFS	Mrežni datotečni sistem
RAID	Redundant array of independent disc
RPC	Remote procedure call
SQL	Structured query language
STDIN	Standardni vhod
STDOUT	Standardni izhod
STONITH	Shoot the other node in the head
TB	Terabajt
YARN	Yet another resource negotiator

1 UVOD

V zadnjih desetletjih se je kapaciteta diskov povečevala, vendar je prenos podatkov z diska ni mogel dohajati. Za primer vzamemo disk iz leta 1990, ki je lahko shranil 1,370 MB in je imel hitrost prenosa podatkov do 4,4 MB/s. Takšen disk smo lahko prebrali v petih minutah.

Diski v današnjih časih so v povprečju veliki 1 TB in imajo povprečno hitrost branja nekaj več kot 100 MB/s, kar pomeni, da potrebujejo za branje celotnega diska nekaj več kot dve uri in pol. Toliko časa za branje in pregled vseh podatkov s posameznega diska ni sprejemljivo, zato moramo uporabiti drugačne načine, kot je na primer branje z več diskov hkrati. Če vzamemo 100 diskov in 1 TB podatkov enakomerno porazdelimo po vseh diskih in beremo podatke z vseh diskov vzporedno, potem lahko preberemo vse podatke v dveh minutah. Ta način zgleda hiter, vendar je preveč potraten, če za branje ne uporabljamo podatkov vseh 100 diskov hkrati.

Drugi način uporabe velike količine diskov za pohitritev dostopa do podatkov je bil v zadnjem desetletju omogočen s pojavitvijo strežnikov, ki imajo relativno nizko ceno. Takšne strežnike lahko uporabljamo v velikih gručah računalnikov, ki so med seboj povezani s hitrimi podatkovnimi povezavami. Gruče lahko obsegajo od nekaj 10 do nekaj 10.000 strežnikov. 1 TB podatkov v tem primeru porazdelimo po 100 strežniških diskih in uporabnikom omogočimo vzporeden dostop do različnih podatkov, shranjenih v gruči računalnikov. Da bi zagotovili hiter hkraten dostop več uporabnikom do istih podatkov, lahko podatke tudi repliciramo.

Problemi, ki se pojavijo v takšnih sistemih, so naslednji. Prvič, zaradi velikega števila strežnikov obstaja veliko večja verjetnost odpovedi delovanja strojne opreme. Znana rešitev je uporaba redundance: shranjevanje istih podatkov na več medijev. V primeru odpovedi delovanja enega diska lahko podatek dobimo z drugega. Drugi problem masovno vzporednih sistemov se pojavi pri analizi podatkov, ki jih sistem shranjuje. Obstoječa programska orodja niso bila primerna za procesiranje zelo velikih količin podatkov v velikih gručah strežnikov. Zasnovati je bilo treba nove programske paradigme, ki omogočajo učinkovito implementacijo analiz podatkov v gruči računalnikov. Tretji problem, ki ga moramo imeti v mislih pri procesiranju velikih količin podatkov z gručo računalnikov, je možnost razširljivosti sistema. Sistem mora biti zgrajen tako, da omogoča rast, saj se lahko količina podatkov v sistemu poveča.

Eden izmed najbolj uporabljanih sistemov, ki dobro rešuje naštete probleme, je Hadoop. Sistem so razvili leta 2008 v podjetju Yahoo. Hadoop trenutno uporabljajo podjetja, kot so Google, Facebook, Yahoo in drugi. Sistem temelji na programskem modelu MapReduce, ki izvira iz funkcijskih programskih jezikov. Procesiranje podatkov v Hadoopu je zasnovano

na dveh zelo dobro preučenih funkcijah višjega reda Map in Reduce. Med najbolj pomembnimi lastnostmi funkcij Map in Reduce, ki jih uporablja Hadoop, sodi možnost enostavnega snovanja vzporednih programov, ki omogočajo učinkovito uporabo velikih gruč strežnikov.

Poglejmo še načine reševanja prej naštetih problemov. Hadoop vsebuje vrsto mehanizmov, ki zmanjšajo verjetnost izgube podatkov, kot so na primer replikacija, nadomestno delovanje in ograjevanje. Učinkovito analizo podatkov omogoča z uporabo programske tehnologije MapReduce, ki predstavlja jedro Hadoop sistema. Enostavno programiranje za obdelavo zelo velikih količin podatkov je možno z uporabo visoko-nivojskega programskega jezika Pig Latin, ki temelji na prevajanju programov nad tokovi podatkov v programe, izražene z MapReduce. Sistem lahko preprosto povečamo tako, da dodajamo nove strežnike v gručo.

1.1 Zgodovina Hadoop

Hadoop je oblikoval Doug Cutting medtem, ko je bil zaposlen v podjetju Yahoo leta 2008. Začetek projekta bi lahko videli že leta 2002 v zasnovi odprtokodnega preiskovalnika (angl. crawler), ki se imenuje Nutch. Sistem je bil že v prvem letu delovanja zelo uspešen. Nutch je lahko poiskal in indeksiral na stotine milijonov HTML strani. Glavna problema preiskovalnika sta bila vezanost na določen tip strežnikov in nujna prisotnost operaterja zaradi možnih izpadov sistema.

Leta 2003 je bil razvit *Google File System* (okr. GFS), ki omogoča učinkovito upravljanje porazdeljenih podatkov za transakcijsko zahtevne aplikacije. Za delovanje ne potrebuje drage strojne opreme in omogoča delovanje tudi ob strojnih napakah komponent. Pod vplivom GFS so razvijalci Nutch zasnovali odprtokodno implementacijo porazdeljenega datotečnega sistema, imenovanega Nutch Distributed File System (okr. NDFS).

V letu 2004 so v podjetju Google v okolju zelo velikih gruč prvi začeli uporabljati programski model MapReduce. Osnovna namena uporabe MapReduce sta procesiranje in generiranje velikih zbirk podatkov v zelo velikih gručah. Razvijalci Nutch so uporabili tudi to idejo in naredili delujočo implementacijo programskega modela MapReduce za Nutch. Leta 2005 je bila že večina Nutch algoritmov prirejenih za delovanje z uporabo MapReduce in NDFS.

Zaradi zmožnosti uporabe programskega modela Mapreduce in porazdeljenega datotečnega sistema NDFS izven domene preiskovanja (angl. crawling) je februarja 2006 nastal

podprojekt Hadoop.[1] Istega leta je Doug Cutting prešel k podjetju Yahoo!, kjer so mu priskrbeli ekipo in sredstva za razvoj Hadoopa v robusten, prilagodljiv, odziven, varen in poceni sistem.

Že leta 2008 je Yahoo! naznanil, da je bil njihov iskalni indeks generiran z 10.000 jedrno gručo računalnikov, ki je slonela na Hadoopu. Tudi pri Apache so postavili na prvo mesto projekt Hadoop, ki je postajal vse bolj priljubljen. Poleg Yahooja so začeli uporabljati Hadoop še Facebook, Google, New York Times in druga znana podjetja.

1.2 Masovni podatki

Izraz masovni podatki (angl. Big Data) se nanaša na zelo veliko količino podatkov, ki je ne moremo obdelati ali analizirati z uporabo tradicionalnih metod ali pripomočkov. Povečuje se število izzivov, povezanih z masovnimi podatki, s katerimi se organizacije v današnjem času ukvarjajo. Dostop do velikih količin informacij imajo, vendar ne vedo, kako jih uporabiti in pridobiti koristne informacije iz njih.

IBM-ova anketa je pokazala, da skoraj polovica poslovnih vodij nima dostopa do vpogleda v informacije, ki jih potrebujejo za delo. Podjetja so običajno zmožna shranjevati vse podatke, ki jih generirajo, zato se količina podatkov stalno povečuje. Vodilni ljudje v podjetjih imajo vpogled v veliko količino podatkov, ki ima veliko koristno vrednost, če bi jo znajo primerno obdelati in uporabiti. V primerjavi s količino pridelanih podatkov se količina obdelanih podatkov močno zmanjšuje.

Obdobje masovnih podatkov je v polnem zagonu. S pomočjo različnih pripomočkov lahko zaznavamo več dejavnikov. Vse nove dejavnike želimo nato tudi shraniti za nadaljnjo uporabo. Z uporabo naprednih tehnologij in komunikacij so ljudje vedno bolj povezani. Storitve, ki jih ponujajo Google, Twitter, Facebook in ostali, omogočajo hranjenje naših aktivnosti, kot so slike, seznami kontaktov, obiskane spletne strani, dokumenti itd. Količina takšnih podatkov narašča vsako leto.

Poglejmo si nekatere značilnosti masovnih podatkov, da bomo lahko bolje razumeli obseg in naravo teh podatkov. Tri značilnosti so bile identificirane kot najbolj opisne lastnosti masovnih podatkov, in sicer prostornina, raznolikost in hitrost.

Prostornina masovnih podatkov predstavlja skupno količino shranjenih podatkov. V letu 2000 je bilo shranjenih 800,000 petabajtov, leta 2011 je ameriško podjetje IDC izračunalo,

[1] Za zanimivost lahko povem, da je Hadoop dobil ime po igrački v obliki slona, ki jo je imel sin Douga Cuttinga. Menil je namreč, da otroci znajo dobro izbirati imena.

da je obstajalo 1.8 zetabajtov podatkov po celem svetu, do leta 2020 pa naj bi nastalo 35 zetabajtov podatkov. 1 zetabajt je 10^{21} bajtov. Poglejmo si nekaj konkretnih primerov: samo Twitter naredi več kot sedem terabajtov podatkov na dan, Facebook 10 TB, še nekatera podjetja pa naredijo vsako uro en terabajt.

Raznolikost podatkov predstavlja strukturno in tudi imensko raznovrstnost. Z vedno večjo uporabo senzorjev in pametnih naprav v zadnjih letih se raznolikost podatkov povečuje. Podatki v podjetjih so vedno bolj kompleksni, saj so sestavljeni tudi iz nestrukturiranih podatkov, ki nastanejo na spletnih straneh, spletnih dnevnikih, elektronskih poštah, spletnih forumih itd.

Hitrost masovnih podatkov predstavlja količino podatkov, generiranih v časovni enoti. Prav tako, kot sta se velikost in raznolikost spremenila v zadnjem obdobju, se je spremenila tudi hitrost, s katero se generirajo podatki. Večja hitrost je posledica uporabe modernih senzorjev, razcveta uporabe socialnih sistemov, kot tudi napredka pri izdelavi sodobnih informacijskih sistemov.

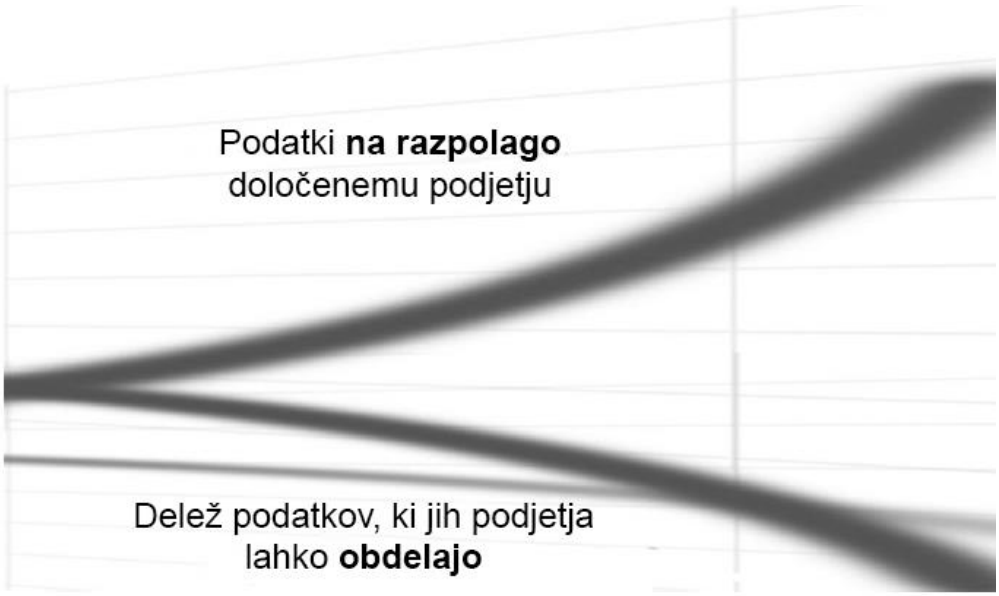
Slika 1 nam prikazuje tri osnovne značilnosti masovnih podatkov v grafični obliki. Prikazane so tudi spremembe pri podatkih. Raznolikost podatkov je pripeljala do tega, da imamo sedaj poleg strukturiranih podatkov, ki so shranjeni v relacijskih tabelah, tudi nestrukturirane podatke, ki pa niso organizirani in jih ne moremo shranit v tradicionalne podatkovne baze. Hitrost generiranih podatkov se spreminja, saj imamo sedaj možnost pridobivanja podatkov v stvarnem času, takoj, ko nastanejo z uporabo tokov. Količina pridelanih podatkov se povečuje iz leta v leto, kar vpliva na to, da podatke merimo že v zetabajtih.[2]

[2] Vir: http://www.theregister.co.uk/2012/12/10/idc_zettabyte_fest/

Slika 1: Razdelitev masovnih podatkov

Predvsem večja podjetja se v zadnjem času spopadajo z ogromno količino podatkov. Tista podjetja, ki ne vedo, kako upravljati s tako količino podatkov, so preobremenjena z njo in izgubljajo vrednost, ki se skriva v njih. S pravo tehnologijo je možno analizirati skoraj vse podatke in pridobiti iz njih tiste informacije, ki so koristne za poslovanje s strankami in razumevanje tržišča. Podjetja lahko najdejo rešitev za določen problem ali vidijo v naprej, kakšen bo novi trend in s tem pridobijo nekaj prednosti pred konkurenco.

Količina podatkov nenehno narašča in orodja za analizo jih ne morejo dohajati. Na Sliki 2 lahko lepo vidimo, kakšna je razlika med podatki, ki jih ima podjetje na razpolago, in deležem, ki ga lahko obdela. Večja razlika med krivuljama pomeni, da podjetje obdela vedno manjši delež podatkov, ki jih ima na voljo. Medtem, ko krivulja s podatki, ki so na razpolago podjetju, stalno narašča, se istočasno krivulja z deležem obdelanih podatkov premika v nasprotno smer. V prihodnje pomeni še večji razpon med razpoložljivimi podatki in deležem obdelanih podatkov v podjetjih.



Slika 2: Prikaz razmerja obdelanih podatkov

S pridobivanjem podatkov na vsakem koraku se povečuje število koristnih informacij, ki jih lahko razberemo iz njih. V prihodnje bo vedno bolj pomembno, kdo bo znal take informacije izkoristiti na pravi način.

2 PORAZDELJEN DATOTEČNI SISTEM HDFS

Ko nam zmanjka prostora na enem strežniku, lahko podatke shranimo tako, da jih porazdelimo po več ločenih strežnikih. Takšen sistem, ki ima ločene strežnike povezane z omrežjem, imenujemo porazdeljen datotečni sistem. Zaradi povezovanja z omrežjem se kompleksnost poveča v primerjavi z navadnim diskovnim datotečnim sistemom.

Hadoop ima svoj porazdeljen datotečni sistem, ki se imenuje HDFS. Narejen je po principih operacijskega sistema UNIX, ki je večopravilni, večuporabniški in prenosljiv operacijski sistem, kateri obstaja v več različicah. V naslednjem poglavju bom natančneje predstavil HDFS.

2.1 Zakaj HDFS

HDFS datotečni sistem je narejen za hrambo zelo velike količine podatkov, kar pomeni več sto megabajtov, gigabajtov ali terabajtov velikosti. Glavna ideja pri nastajanju HDFS je bila ta, da so želeli ustvariti sistem, ki bi bil optimiziran za branje podatkov. V večini primerov se podatkovno zbirko kopira ali ustvari na določenem mestu in se na njej čez čas naredi več analiz podatkov. Vsaka od analiz vključuje del ali celotno podatkovno zbirko, zato je bolj pomemben čas branja celotne podatkovne zbirke kot pa zakasnitev, ki nastane pri branju prvega zapisa.

2.2 Strojna oprema

Za poganjanje Hadoopa ne potrebujemo najdražje in najbolj zanesljive strojne opreme. Narejen je tako, da lahko teče tudi na taki strojni opremi, pri kateri lahko pride do kakšnih napak med delovanjem. Ker je HDFS napisan v Javi, lahko teče na katerem koli sistemu, ki podpira Javo. Napake rešuje tako, da uporabnik sploh ne opazi, da je med delovanje sploh prišlo do napake.

HDFS ima tudi slabe lastnosti, zato za nekatere aplikacije ni uporaben. Aplikacije, ki potrebujejo zelo hiter dostop do podatkov, to je nekje 10 ms, ne bodo delovale dobro, ker je HDFS optimiziran za velik pretok podatkov, kar po drugi strani prinese večjo latenco.

Imensko vozlišče hrani metapodatke datotečnega sistema v spominu, kar pomeni, da je število datotek v datotečnem sistemu omejeno z velikostjo spomina, ki ga ima na voljo imensko vozlišče. Vsaka datoteka, direktorij in blok so predstavljeni kot objekt v imenskem vozlišču znotraj pomnilnika in vsak zasede 150 zlogov. Za primer vzamemo en milijon datotek, kjer vsaka zasede en blok, kar privede do skupnega rezultata 300 MB spomina. Če bi želeli shranjevati milijarde datotek, pa tega s Hadoopom ne bi več mogli. V datoteke lahko piše samo en zapisovalec na enkrat saj v HDFS ni nobene podpore pri pisanju več zapisovalcev hkrati v isto datoteko.

2.3 Osnovne operacije nad datotečnim sistemom

Pogledali si bomo osnovne ukaze za HDFS, ki jih lahko izvedemo znotraj ukazne vrstice. Obstajajo še drugi vmesniki, s katerimi lahko delamo znotraj HDFS, vendar je ukazna vrstica ena izmed najbolj enostavnih in je znana večini razvijalcev.

Za pridobitev natančnejših navodil, kako uporabljati določen ukaz, imamo na voljo ukaz *help*. V ukazno vrstico vpišemo *% hadoop fs –help*. Za kopiranje datoteke iz lokalnega datotečnega sistema v HDFS uporabimo ukaz *copyFromLocal*. Primer ukaza za *copyFromLocal*:

```
% hadoop fs –copyFromLocal input/docs/test.txt hdfs://localhost/user/tom/test.txt
```

Z zgoraj napisanim ukazom *fs* pokličemo lupino Hadoopovega datotečnega sistema, katera podpira dodajanje dodatnih ukazov. V našem primeru smo dodali ukaz *–copyFromLocal*. Lokalno datoteko test.txt smo kopirali v datoteko, ki se nahaja v /user/tom/test.txt na HDFS, ki teče na localhost (lokalni gostitelj). Napisani ukaz bi lahko skrajšali tako, da bi napisali brez *hdfs://localhost*, ker je privzeta nastavitev *hdfs://localhost*, tako da bi ukaz zgledal takole:

```
% hadoop fs –copyFromLocal input/docs/test.txt /user/tom/test.txt
```

Ker je naš domači direktorij prav /user/tom/, lahko še malo skrajšamo ukaz:

```
% hadoop fs –copyFromLocal input/docs/test.txt test.txt
```

Preverili bomo, ali se je datoteka spremenila, ko smo jo kopirali v HDFS. Uporabili bomo ukaz *–copyToLocal*, da jo kopiramo nazaj lokalno.

```
%hadoop fs –copyToLocal test.txt test.copy.txt
% md5 input/docs/test.txt test.copy.txt
```

Algoritem MD5 nam prikaže spremembe v obliki 32 mestnega števila. Spodaj lahko vidimo, da sta obe števili enaki, kar pomeni, da se datoteka ni spremenila na poti v HDFS in nazaj.

```
MD5 (input/docs/test.txt) = a16f231da6b05e2ba7a339320e7dacd9
MD5 (test.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

Poglejmo si še izpisovanje seznama datotek z ukazom *–ls*. Najprej ustvarimo nov direktorij z ukazom *–mkdir* za lažjo primerjavo.

% hadoop fs –mkdir books

Naredili smo nov direktorij z imenom *books.*

```
%hadoop fs -ls .
Found 2 items
drwxr-xr-x   -   tom supergroup       0 2009-04-02 22:41  /user/tom/books
-rw-r--r--       1  tom supergroup  118 2009-04-02 22:29  /user/tom/test.txt
```

Informacija, ki smo jo dobili nazaj, je zelo podobna ukazu *ls –l* v Unix sistemu. Razlika je v drugem stolpcu, kjer imamo število kopij določene datoteke. V tem primeru je 1. Na začetku smo v nastavitvah določili tako, ker imamo samo eno podatkovno vozlišče. Pri novo nastalem direktoriju nimamo nobene kopije, ker replikacija ne vpliva na direktorije. Direktorij je obravnavan kot metapodatek, zato ga shranjuje imensko vozlišče. Ostali stolpci so opredeljeni kot v Unix sistemih (lastnik, skupina, velikost, datum spremembe, absolutna pot).

2.4 Dovoljenja datotek v HDFS

Znotraj HDFS imamo model dodeljevanja pravic za datoteke in direktorije, ki je podoben POSIX (zbirka standardov, ki veljajo pri pisanju aplikacij za UNIX in UNIX podobnim sistemom) .

Poznamo tri vrste pravic:

- (r) pravico za branje,
- (w) pravico za pisanje in
- (x) pravico za izvrševanje.

Pravice za branje potrebujemo za branje datotek ali pa za pridobitev seznama datotek znotraj določenega direktorija. Pisalna pravica nam omogoča pisanje v določeno datoteko ali direktorij in oblikovanje ali brisanje datotek/direktorijev. Izvršilna pravica je pri datotekah prezrta, namreč datoteke v HDFS ne moremo izvajati. Za direktorij pa je omogočen dostop do njegovih otrok. Vse datoteke in direktoriji imajo *lastnika*, *skupino* in *način* (angl. mode).

Način je določen na podlagi:

- pravic, ki jih ima uporabnik kot lastnik,
- pravic, ki jih imajo uporabniki določene skupine, in
- pravic, ki jih imajo uporabniki, kateri niso lastniki in niti ne pripadajo skupini.

Za preprečevanje neželenega spreminjana ali brisanja v določenih delih datotečnega sistema se omogoči *preverjanje pravic* (angl. persmission checking). Le-to nam omogoča preverjanje pravic odjemalca, ki se povezuje na HDFS. Pri preverjanju se pogleda, kakšne akcije lahko izvaja kot uporabnik in kateri uporabniški skupini pripada.

3 IMPLEMENTACIJA HDFS

V Hadoopu predstavlja datotečni sistem Javin abstraktni razred, imenovan *org.apache.hadoop.fs.FileSystem*. Znotraj paketa *org.apache.hadoop* najdemo še številne Javine implementacije, kot je na primer razred *hdfs.DistributedFileSystem* implementacija HDFS. *hdfs.HftpFileSy*stem implementacija za HFTP omogoča uporabo branja iz HDFS z uporabo HTTP. Podobno imamo *hdfs.hsftpFileSystem* implementacijo za HTTPS, ki nam omogoča varnejši dostop. Za arhiviranje datotek imamo dodatni datotečni sistem HAR, ki skrbi za arhiviranje datotek in zmanjševanje porabe spomina v imenskih vozliščih. Uporabimo ga lahko z razredom *fs.HarFileSystem*. Datotečni sistem, ki podpira FTP strežnik, je *fs.ftp.FTPFileSystem*. Za večjo varnost podatkov imamo na voljo tudi RAID HDFS različico, ki je optimizirana za arhivsko shranjevanje. Za vsako datoteko se ustvari manjša paritetna datoteka, katera pripomore k temu, da lahko zmanjšamo replikacijo datotek iz 3 na 2. Posledično to pomeni, da zmanjšamo porabo diskovnega prostora. Implementacija *hdfs.DsitributedRaidFileSystem* nam ne zmanjša ali poveča verjetnosti izgube podatkov. Za njegovo delovanje je treba na vsaki gruči računalnikov zagnati v ozadju še prikriti proces *RaidNode*.

3.1 Bločna abstrakcija

Bločna abstrakcija v porazdeljenem datotečnem sistemu prinaša razne prednosti.

Prva najbolj pomembna prednost je ta, da lahko shranimo datoteko, večjo od katerega koli diska v omrežju. Namreč, nikjer ni omejeno, da morajo biti bloki, na katerih je shranjena datoteka, na istem disku. Lahko so shranjeni na katerem koli disku znotraj gruče. Obstaja tudi možnost, da lahko bloki samo ene velike datoteke zasedejo vse diske znotraj HDFS gruče.

Druga prednost je izboljšanje pri upravljanju shranjevanja blokov. Shranjevalni podsistem lahko v naprej izračuna, koliko blokov lahko shranimo na določen disk, saj so vsi bloki iste velikosti. Bloki so skupki podatkov, katere želimo shraniti, zato nam ni treba poleg shranjevati še metapodatkov, kot so informacije o pravicah. Te lahko shranimo posebej na drugem sistemu. Pri poškodovanih blokih ali diskih se lahko prebere kopijo bloka, ki je bila predhodno varnostno shranjena, običajno na drugih dveh računalnikih znotraj gruče, kar omogoča nemoteno delovanje.

Za pregledovanje in preverjanje blokov znotraj sistema uporabimo ukaz *fsck. Fsck* je standardni Unix ukaz.

Uporaba: `% hadoop fsck / -files –blocks`

Zgornji ukaz nam napiše seznam blokov, kateri sestavljajo vsako izmed datotek znotraj datotečnega sistema. S tem ukazom ugotovimo, v kakšnem stanju je datotečni sistem.

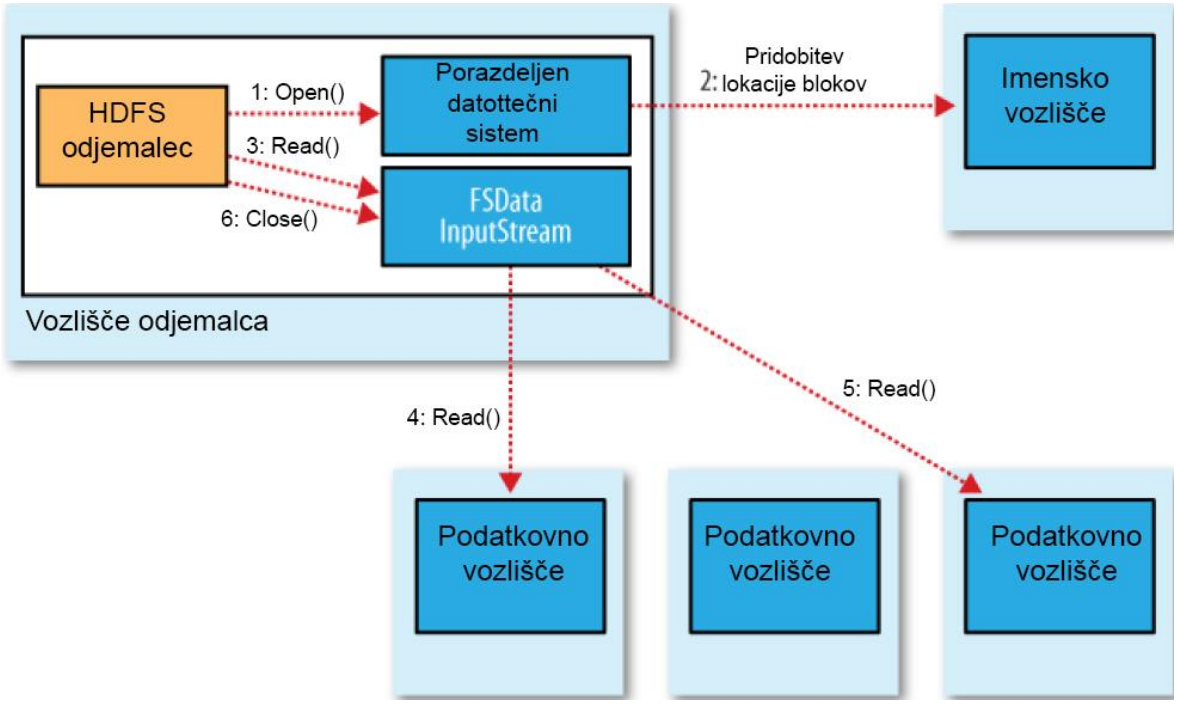
3.2 Blok v HDFS

Trdi diski imajo bloke določene velikosti, ki predstavljajo najmanjšo enoto podatkov, ki jo lahko shranijo. Datotečni sistemi imajo v večini primerov bloke velike 512 zlogov. Za HDFS pa je določeno, da ima bloke velike 64 MB. Datoteke, ki so večje od velikosti bloka, so razbite na več delov velikosti bloka in shranjene kot neodvisne enote. Za razliko od drugih datotečnih sistemov se v primeru, ko datoteka ne zasede celotnega bloka, lahko nezasedeni prostor na bloku še vedno uporabi za nadaljnje shranjevanje.

Bloki v HDFS so večji v primerjavi z diskovnimi bloki običajnih Unix sistemov. Razlog za to tiči v zmanjševanju premikov bralne glave diska. Pri velikih blokih zmanjšamo tudi število interakcij odjemalca z imenskim vozliščem. Enkrat, ko imensko vozlišče poda informacijo o lokaciji bloka, se izvajata branje in pisanje na določenem bloku, kar se pri večjih blokih izvaja dalj časa. Tako nastaja manj interakcij z imenskim vozliščem kot pa pri manjših blokih. Pri večjih blokih se zmanjša tudi poraba omrežja, saj odjemalec izvaja razne operacije na istem bloku z uporabo iste TCP povezave za dalj časa. Zmanjša se tudi velikost metapodatkov, shranjenih v imenskem vozlišču. Manjšo količino metapodatkov lahko posledično shranimo v spomin, kar nam prinese še druge prednosti.

3.2.1 Potek branja blokov

Na spodnji sliki si lahko grafično pogledamo, kako poteka prenos podatkov med odjemalcem (angl. client) in imenskim (angl. namenode) ter podatkovnim vozliščem (angl. datanode).



Slika 3: Potek branja odjemalca iz HDFS

Odjemalec odpre datoteko, ki jo želi brati z ukazom *Open()* na objektu *FileSystem*, ki je za HDFS instanca *Porazdeljenega datotečnega sistema* (1. korak na Sliki 3). Nato *Porazdeljen datotečni sistem* pokliče *imensko vozlišče* z uporabo RPC, ki mu vrne lokacijo za prvih nekaj blokov (2. korak). Imensko vozlišče vrne za vsak blok naslov, na katerem podatkovnem vozlišču se nahaja kopija bloka. Če je odjemalec sam podatkovno vozlišče (v primeru MapReduce opravila), bo prebral kopijo bloka lokalno, če obstaja lokalna kopija.

Porazdeljen datotečni sistem vrne objekt *FSDataInputStream* (vhodni podatki za branje) odjemalcu. Objekt *FSDataInputStream* uporabi objekt *DFSInputStream*, kateri upravlja podatkovna vozlišča in I/O imenskega vozlišča. Odjemalec pokliče *Read()* (3. korak). Objekt *DFSInputStream* ima shranjene naslove podatkovnih vozlišč in se poveže na najbližje podatkovno vozlišče, ki ima prvi blok datoteke. Podatki se pošiljajo iz podatkovnega vozlišča nazaj odjemalcu, ki kliče *Read()*, dokler ne dobi celotnega bloka (4. korak). Ko odjemalec pridobi celoten blok, mu objekt *DFSInputStream* zapre povezavo s podatkovnim vozliščem in odpre novo z naslednjim najbližjim podatkovnim vozliščem, ki vsebuje naslednji blok (5. korak). Z vidika odjemalca se vse skupaj dogaja tako, kot da bi prebral bloke iz samo enega podatkovnega vozlišča.

Objekt *DFSInputStream* pokliče imensko vozlišče za pridobitev naslednje serije blokov za branje. Ko odjemalec konča z branjem vseh blokov, pokliče *Close()* na objektu *FSDataInputStream* (6. korak).

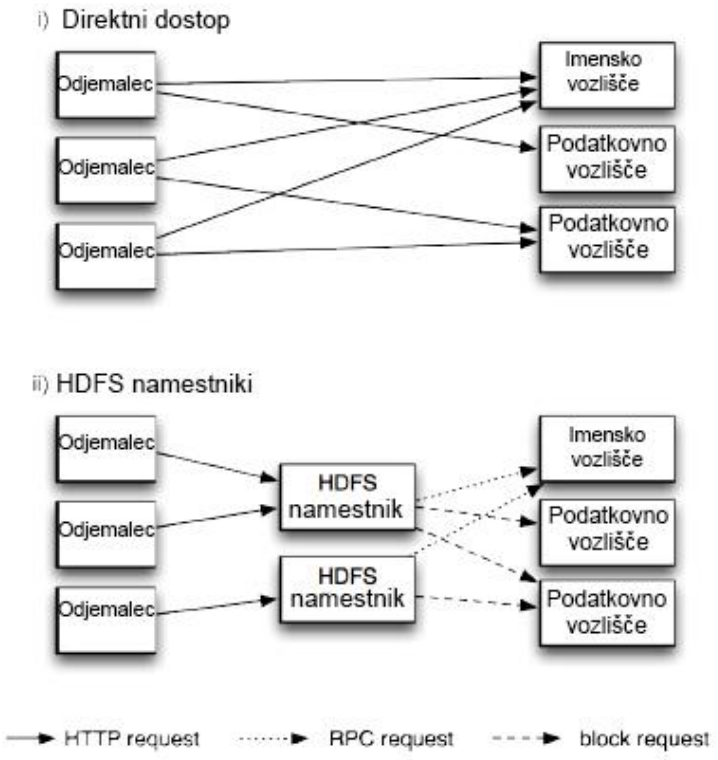
Če med branjem pride do napake pri komunikaciji s podatkovnim vozliščem, se *DFSInputStream* odloči, da poskusi drugo najbližje podatkovno vozlišče za branje bloka. Podatkovno vozlišče, na katerem je prišlo do napake, si zapomni tako, da se naslednjič ne bo spet poskušal povezati z nedelujočim vozliščem. Pri prenosu podatkov iz podatkovnega vozlišča preverja, ali je preneseni blok pokvarjen. Če se najde pokvarjen blok, se takoj obvesti imensko vozlišče. Tak način delovanja HDFS omogoča serviranje velikemu številu odjemalcev, saj se podatkovni promet porazdeli po vseh podatkovnih vozliščih znotraj gruče računalnikov. Imensko vozlišče pa samo vrača zahteve po lokacijah blokov, ki jih ima shranjene v spominu, kar omogoča hitre odgovore.

3.3 Vmesniki

Znotraj Hadoopa najdemo več vmesnikov, ki za identifikacijo datotečnega sistema uporabljajo URI (enotni označevalnik vira). Ker je Hadoop napisan v Javi, so vse interakcije z datotečnim sistemom posredovane preko Java API. Tudi lupina datotečnega sistema je Java aplikacija, ki uporablja Javin *Filesystem* razred za operacije nad datotečnim sistemom. Osnovni vmesnik je napisan v javi, nad katerim lahko delamo še z naslednjimi vmesniki.

3.3.1 Http

Za dostopanje do HDFS z uporabo http obstajata dva načina. Prvi način je neposredni dostop, pri katerem prikriti procesi v HDFS servirajo http zahtevam odjemalcev. Drugi način je z uporabo namestnikov (angl. proxy), ki dostopajo do HDFS po naročilu odjemalca z uporabo DistributedFileSystem API.



Slika 4: Dostopanje neposredno do HDFS z HTTP in z uporabo namestnikov

V prvem primeru (na Sliki 4) se odjemalci povezujejo z imenskim vozliščem z uporabo vgrajenega spletnega strežnika, ki teče na vratih 50070. Odjemalce postreže s seznamom imenikov. Podatki pa se pošiljajo iz podatkovnih vozlišč, ki imajo svoj spletni strežnik, kateri teče na vratih 50075.

HTTP vmesnik je bil namenjen samo branju, nova WebHDFS implementacija pa podpira izvajanje vseh operacij datotečnega sistema, vključno s Kerberos avtentikacijo.

Drugi način dostopanja deluje tako, da se povezujemo s http z uporabo enega ali več namestniških strežnikov. Namestniški strežnik deluje kot vmesnik med zahtevo odjemalca, ki potrebuje neko datoteko z drugega serverja. Preverja zahteve in vrne nazaj odgovor odjemalcu. Celoten promet za določeno gručo gre skozi namestnik, kar omogoča postavljanje strožje varnosti znotraj požarnega zidu na določenem mestu. Namestnike se

uporablja tudi za povezovanje med gručami računalnikov, ki so locirani na ločenih podatkovnih centrih.

Od različice 0.23 je v uporabi nov namestnik, imenovan HttpFS, ki omogoča branje in pisanje, uporablja pa isti HTTP vmesnik kot WebHDFS.

3.3.2 Ostale knjižice

Znotraj Hadoopa lahko najdemo knjižico za programski jezik C, ki se imenuje *libhdfs*. Ta knjižica ima iste lastnosti kot vmesnik Java *FileSystem* in z njo lahko dostopamo do katerega koli datotečnega sistema znotraj Hadoop. Deluje tako, da uporablja *Java Native Interface* (okr. JNI) za klicanje Java datotečnega odjemalca.

Za uporabo raznih Unix orodij imamo na voljo FUSE, kar pomeni datotečni sistem v uporabniškem prostoru. Vsebuje modul Fuse-DFS, ki omogoča vsakemu datotečnemu sistemu iz Hadoopa, da se namesti kot standardni datotečni sistem. Zato lahko uporabimo razna Unix orodja, kot sta *ls* in *cat* za interakcijo. Fuse-DFS je implementiran v C z uporabo *libhdfs* knjižice kot vmesnik za HDFS.

3.4 Imensko in podatkovno vozlišče

Z izrazom gruča (angl. Cluster) poimenujemo več računalnikov, ki sodelujejo med seboj in so navzven videti kot en sistem. Vsak računalnik znotraj gruče ima svoj operacijski sistem in se povezuje z ostalimi preko lokalnega omrežja.

V HDFS gruči najdemo dva tipa vozlišč, ki delujeta po principu *nadrejeni-podrejeni*. Imensko vozlišče je nadrejeno, ker upravlja imenski prostor (angl. name space) in skrbi za metapodatke vseh datotek in direktorijev znotraj drevesa datotečnega sistema. Podatkovno vozlišče pa mu je podrejeno, saj shranjuje in posreduje bloke na zahtevo imenskega vozlišča ali odjemalca, ob določenih intervalih pa poroča imenskemu vozlišču o shranjenih blokih.

Informacije o metapodatkih so shranjene na lokalnem disku v obliki dveh datotek: slika imenskega prostora in dnevnik urejanja. Imensko vozlišče ve, na katerih podatkovnih vozliščih so shranjeni vsi bloki za določeno datoteko, vendar te informacije ne hrani stalno, saj se ta informacija generira ob zagonu sistema. Brez imenskega vozlišča je datotečni sistem neuporaben, kar pomeni, da se ob prenehanju delovanja računalnika, na katerem teče imensko vozlišče, vse datoteke izgubijo in jih ni mogoče rekonstruirati iz blokov v podatkovnih vozliščih.

Zato je pomembno, da je imensko vozlišče odporno na različne izpade in Hadoop ima za ta namen dva mehanizma.

3.4.1 Varnostni mehanizmi

Prvi način za preprečevanje izgube podatkov je narejen tako, da se naredi varnostno kopijo stanja metapodatkov datotek datotečnega sistema. V Hadoopu je mogoče nastaviti, da imensko vozlišče zapisuje svoje trenutno stanje na več datotečnih sistemov, ki se lahko nahajajo na lokalnem disku ali pa na oddaljenem NFS. NFS omogoča oddaljen dostop in delo z datotekami, kot da so na lokalnem disku. Pisanje se izvaja sinhrono in atomarno.

Druga možnost vključuje sekundarno imensko vozlišče, ki ne glede na ime ne opravlja te vloge, ampak je njena naloga, da periodično združuje sliko imenskega prostora z dnevnikom urejanja. Slika imenskega prostora je stanje imenskega vozlišča v spominu ob določenem trenutku. Vsebuje bloke, na katerih se nahajajo datoteka, lastnosti datoteke … Z uporabo te slike lahko vzpostavimo nazaj trenutno stanje. Dnevnik urejanja pa vsebuje nove spremembe od zadnje nastale slike. Z združevanjem slike in dnevnika v novo obnovitveno točko preprečujemo, da bi dnevnik urejanja postal prevelik.

Sekundarno imensko vozlišče potrebuje veliko procesorske moči in spomina velikosti imenskega vozlišča za združevanje slike in dnevnika urejanja. V večini primerov se nahaja na ločenem računalniku. V primeru odpovedi delovanja primarnega imenskega vozlišča nam ne preostane drugega, kot naložiti metapodatke datotek k sekundarnemu vozlišču in ga obravnavati kot novo primarno imensko vozlišče.

3.4.2 HDFS Visoka dostopnost

Pri repliciranju metapodatkov imenskega vozlišča na več datotečnih sistemov in uporabi sekundarnega imenskega vozlišča pripomoremo k preprečevanju izgube podatkov, ne moremo pa zagotoviti stalne dostopnosti datotečnega sistema.

3.4.3 HDFS federacija

Imensko vozlišče beleži za vsako datoteko in blok znotraj datotečnega sistema referenco, ki jo hrani v spominu, kar pomeni, da pri veliki količini datotek v velikih gručah postane spomin ozko grlo.

HDFS Federacija z različico 0.23 omogoča gruči rast z uvedbo novih imenskih vozlišč, vsaka od njih pa vodi določeni del imenskega prostora znotraj datotečnega sistema. Za lažje razumevanje si predstavljamo imensko vozlišče, ki upravlja vse datoteke znotraj imenika /user, drugo imensko vozlišče pa upravlja vse datoteke znotraj imenika /share.

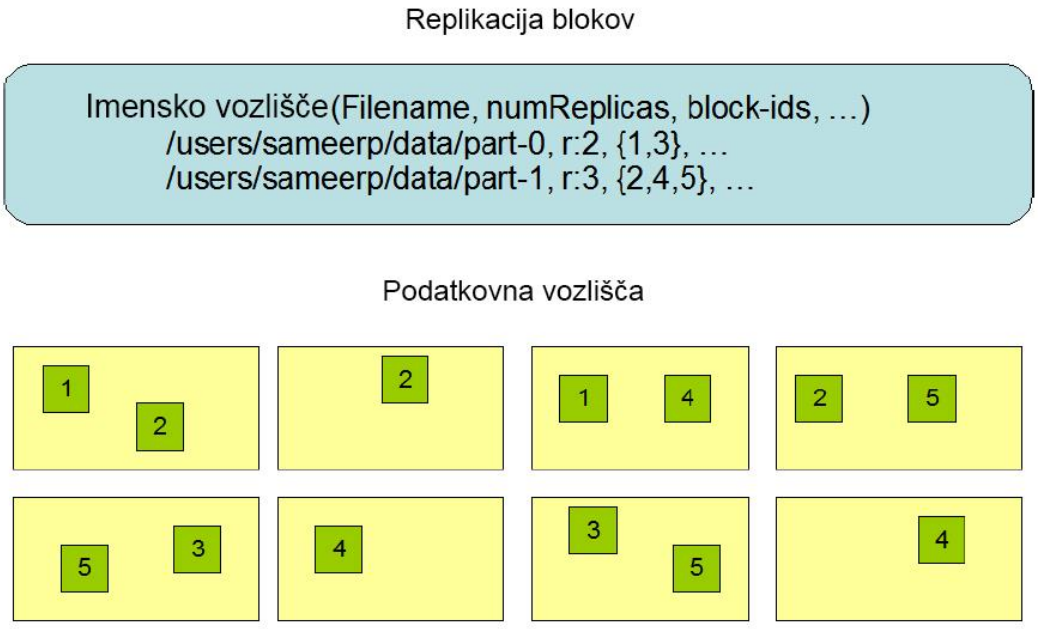
Vsako imensko vozlišče upravlja meta podatke za določen imenski prostor in bazen blokov, ki vsebuje vse bloke za datoteke v tem imenskem prostoru. Prednost takega delovanja je neodvisnost imenskih vozlišč med seboj. Ob napaki določenega imenskega vozlišča ne

ogrožamo delovanja v drugih imenskih prostorih. Podatkovna vozlišča lahko komunicirajo z več imenskimi vozlišči znotraj gruče in shranjujejo bloke iz več bločnih bazenov.

3.5 Replikacija podatkov

HDFS shranjuje datoteke v neko zaporedje blokov, kateri so iste velikosti, razen zadnjega. Zaradi zagotavljanja večje varnosti pred izgubo podatkov so bloki replicirani. Koliko krat bo blok repliciran, je odvisno od nastavitev. Ob nastanku datoteke lahko aplikacije same določijo število kopij določene datoteke, ki se lahko spremeni tudi kasneje.

Odločitve za repliciranje blokov sprejema imensko vozlišče na podlagi *srčnega utripa* (angl. Heartbeat) in *bločnega poročila* (angl. Blockreport), ki jih prejema od podatkovnih vozlišč. *Srčni utrip* nosi podatke o tem, ali podatkovno vozlišče deluje pravilno, *bločno poročilo* pa vsebuje seznam vseh blokov na vozlišču. Na Sliki 5 lahko vidimo, kako deluje replikacija. V prikazanem primeru imamo datoteko part-0, za katero je določeno, da mora imeti dve kopiji blokov 1 in 3. Vidimo lahko, da sta nastali dve kopiji blokov 1 in 3 na podatkovnih vozliščih.



Slika 5: Prikaz replikacije blokov

3.5.1 Namestitev replikacije

Nastavljanje repliciranja je zelo pomembno za stabilnost in zmogljivost sistema. Optimiziranje repliciranja loči HDFS od ostalih porazdeljenih datotečnih sistemov. Za določanje optimalnih nastavitev je potrebno veliko časa, testiranja in predvsem veliko

izkušenj. Z nastavitvami repliciranja znotraj strežniškega stojala (angl. rack) vplivamo na stabilnost, dostopnost in porabo omrežja.

Velike HDFS instance tečejo znotraj gruče računalnikov, katera združuje več strežniških stojal. Komunikacija med vozlišči iz različnih strežniških stojal mora potekati skozi stikalo, ki povezuje strežnike v omrežje. V večini primerov je pasovna širina znotraj enega strežniškega stojala večja kot pa tista v povezavi z drugim.

HDFS uporablja enostavno in učinkovito politiko lociranja kopij blokov. Ko proces, ki teče na enem izmed vozlišč HDFS gruče, odpre datoteko za pisanje bloka, se ustvari kopija bloka na strežniškem stojalu, kjer teče proces. Druga kopija se shrani na strežniškem stojalu, ki je drugačen od tistega, na katerem je bila ustvarjena prva kopija. Tretja kopija se shrani na istem strežniškem stojalu kot druga, ampak na drugem računalniku. Poudariti moramo, da ni nobene povezave med replikami blokov iste datoteke. Vsak blok je lociran neodvisno od ostalih.

Takšen algoritem je dober glede razširljivosti in dostopnosti.

3.5.2 Izbor kopije

Za zmanjševanje globalne porabe povezave preko omrežja in latence branja se lahko uporabi kopijo bloka, ki je najbližja bralcu. HDFS dobi zahtevek po branju določenega bloka in pogleda, če obstaja kakšna kopija na istem strežniškem stojalu, na katerem je bralec. V takem primeru bralec prebere kopijo, ki mu je najbližja in s tem prihranimo pri povezavi in času branja.

3.5.3 Stanje varen način

Ob zagonu preide imensko vozlišče v posebno stanje, imenovano *varen način* (angl. safe mode). Repliciranje podatkovnih blokov ne poteka medtem, ko je imensko vozlišče v *varnem načinu*. Takrat prejme *srčni utrip* in *bločno poročilo* iz podatkovnih vozlišč. *Bločno poročilo* vsebuje seznam blokov, ki so na določenem podatkovnem vozlišču. Vsak blok ima predpisano minimalno število kopij. Za blok smatramo, da je varen, ko je minimalno število doseženo. Nato imensko vozlišče zapusti *varen način* in določi seznam podatkovnih blokov, kateri imajo premalo kopij. Takoj za tem se bloki s seznama replicirajo na podatkovnih vozliščih.

3.6 Reševanje iz napak pri odpovedi vozlišča

Ustvarjanje obnovitvene točke na sekundarnem vozlišču preprečuje izgubo podatkov, vendar pa nam ne omogoča visoke dostopnosti. V vsakem primeru se ob odpovedi delovanja imenskega vozlišča prekine normalno delovanje sistema (pisanje, branje …), dokler ne začne delovati novo imensko vozlišče. Vsem odjemalcem, kot je na primer MapReduce, ne bo omogočeno branje, pisanje ali pridobitev seznama datotek, ker edino imensko vozlišče hrani metapodatke in preslikavo blokov, na katerih je shranjena datoteka.

Za obnovitev sistema ob napaki imenskega vozlišča moramo zagnati novo primarno imensko vozlišče. Pri vzpostavitvi novega imenskega vozlišča je treba nanj naložiti kopijo meta podatkov datotečnega sistema in nastaviti podatkovna vozlišča ter odjemalce na novo imensko vozlišče.

Za začetek delovanja pa mora imensko vozlišče še:

- naložiti sliko imenskega prostora v spomin,
- prebrati dnevnik urejanja in
- pridobiti dovolj poročil o blokih iz podatkovnih vozlišč.

Na velikih gručah računalnikov z veliko datotekami in bloki lahko traja celoten proces tudi 30 minut in več. Pri različici Hadoopa 0.23 je dodan nov način reševanja problemov za *HDFS visoka dostopnost* (angl. high availability). Namesto enega aktivnega imenskega vozlišča sta sedaj dva in v primeru izpada enega prevzame drugi njegovo delo in servisira naprej zahtevke brez prekinitve.

Za tak način delovanja je potrebnih nekaj arhitekturnih sprememb, kot so :

- Imenska vozlišča morajo uporabljati visoko dostopno skupno hrambo za delitev dnevnika urejanja (uporaba NFS), zato da si lahko ob menjavi drugo pasivno imensko vozlišče prebere konec dnevnika urejanja in sinhronizira njegovo stanje z aktivnim. Tako lahko nadaljuje z branjem novih vnosov.
- Podatkovna vozlišča morajo pošiljati poročila vsakemu imenskemu vozlišču posebej, ker si stanje blokov shranjujeta v svoj spomin, ne pa na disk.
- Odjemalci morajo biti nastavljeni za menjavanje imenskih vozlišč ob napakah, brez da bi s tem vplivali na delovanje za uporabnike.

Celotna zamenjava se zgodi v nekaj sekundah, saj ima sekundarno vozlišče že v spominu vse, kar potrebuje (zadnji dnevnik urejanja in bločno mapo).

3.6.1 Nadomestno delovanje in ograjevanje

V vsakem imenskem vozlišču teče proces *nadomestni krmilnik* (angl. failover), ki pregleduje, ali prihaja do napak. Ob napaki se sproži nadomestno vozlišče, ki zamenja imensko vozlišče. *Nadomestni krmilnik* torej skrbi za zamenjavo med aktivnim in pasivnim imenskim vozliščem. Nadomestni način delovanja se lahko zažene tudi ročno (postopno nadomeščanje), npr.: pri vzdrževanju.

Pri nenadnem nadomeščanju se ne moremo prepričati, ali je zamenjano imensko vozlišče res nehalo teči. Zato ima implementacija *visoka dostopnost* metodo *ograjevanje* (angl. fencing), ki preprečuje delovanje dveh aktivnih imenskih vozlišč hkrati, kar bi lahko privedlo do okvar.

Mehanizmi *ograjevanja* lahko ubijejo proces imenskega vozlišča, preprečijo dostop do skupne shrambe in onemogočijo mrežni port. Kot zadnji mehanizem pa lahko uporabi tudi tehniko, imenovano STONITH, ki uporabi posebno enoto za napajanje, katera na silo zaustavi sistem, na katerem teče imensko vozlišče.

4 MAPREDUCE

MapReduce je programski model, ki je vgrajen v jedro platforme Hadoop in je prilagojen za uporabo skupaj s HDFS. Povezan je z implementacijo pri obdelavi in generiranju velikih podatkovnih nizov.

Uporabniki določijo map funkcijo, katera obdela par vrednost/ključ in generira nabor parov vmesnih vrednosti/ključev. Reduce funkcija združuje vse vmesne vrednosti, povezane z istim vmesnim ključem. Programi, napisani v Mapreduce, so avtomatsko prirejeni za porazdeljeno delo na velikih gručah. Izvajalni sistem poskrbi za porazdelitev vhodnih podatkov na več strežnikov in za notranjo komunikacijo med njimi. Tak način omogoča programerjem brez kakršnih koli izkušenj pri delu s porazdeljenimi sistemi, da zlahka izkoristijo zmožnosti velikih porazdeljenih sistemov.

MapReduce podpira programe, napisane v Javi, z uporabo *pretakanja* (angl. streaming) pa lahko tudi Ruby, Python in C++. Pretočni API (angl. Streaming API) služi kot vmesnik med Hadoopom in napisanim programom. Pomembno vlogo pri MapReduce igra vzporedno programiranje. Pri vzporednem programiranju si pomagamo pri reševanju določenega problema tako, da ga razdelimo na več delov, katere procesiramo sočasno na več procesorjev. S takim početjem zmanjšamo čas izvedbe.

4.1 Osnove MapReduce

MapReduce je srce Hadoopa, ker omogoča združevanje velikega števila strežnikov v Hadoop gručo. Pomen besede MapReduce je razdeljen na dve operaciji, kateri se izvajata znotraj Hadoop programa. Prva operacija je Map posel, ki vzame nek niz podatkov in ga pretvori v drug niz podatkov formata (par ključ/vrednost). Reduce operacija pa vzame izhodne podatke od map in jih uporabi kot vhodne podatke. Z združevanjem podatkov po ključu generira manjše nize podatkov.

4.2 Osnovni koncepti

Za lažje razumevanje delovanja MapReduce si bomo pogledali enostaven primer.

Predpostavimo, da imamo štiri datoteke in vsaka datoteka vsebuje dva stolpca (ključ in vrednost), ki, recimo, predstavljata mesto in število ljudi v tem mestu. Meritve števila ljudi so zapisane večkrat za isto mesto, ker se izvajajo v različnih obdobjih. Torej je mesto ključ in število ljudi je vrednost.

Vsaka datoteka zgleda tako, vendar lahko ima druge vrednosti:

Tabela 1: Primer datoteke zapisa ključ/vrednost

Koper	23.000
Ljubljana	274.000
Maribor	94.000
Kranj	37.000
Ljubljana	270.000
Koper	21.000

Iz vseh danih podatkov želimo razbrati največje število prebivalcev vsakega mesta. V kateri datoteki je zapisano največje število prebivalstva za določeno mesto, ne vemo, zato uporabimo MapReduce. Delo razdelimo med štiri Mapperje (Map funkcije), vsakemu določimo eno datoteko. Vsak pregleda svojo datoteko in vrne rezultate: največje število prebivalcev za določeno mesto.

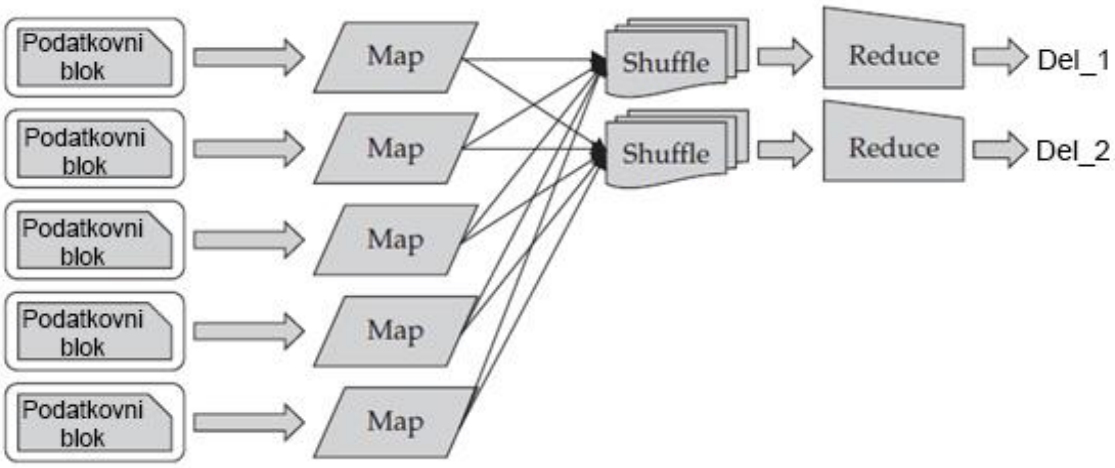
Primer na naši datoteki po Map operaciji: (Koper, 23.000) (Ljubljana, 274.000) (Maribor, 94.000) (Kranj, 37.000).

Ostali trije Map naredijo tako tudi z njihovo datoteko. Rezultate vseh štirih Mapperjev bo operacija Reduce vzela kot vhodne podatke in bo za vsako mesto posebej izpisala največje število prebivalcev. Tako bodo nastali samo štirje nizi podatkov, eden za vsako mesto.

V Hadoop gruči se MapReduce program nanaša na *posel* (angl. job). Vsak posel se razbije na več koščkov, kateri se imenujejo *opravila* (angl. tasks). Aplikacija pošlje *posel* določenemu vozlišču znotraj Hadoop gruče, na katerem teče prikriti proces, imenovan *JobTracker*. *Jobtracker* komunicira z imenskim vozliščem, da ugotovi, kje se znotraj gruče nahajajo potrebni podatki za izvajanje. Ko najde vse podatke, razbije/razdeli posel na Map in Reduce opravila za vsako vozlišče znotraj gruče. Opravila so porazdeljena tako, da vsako vozlišče dobi opravilo, za katero lahko dobi podatke lokalno. Tak koncept imenujemo lokalnost podatkov in je zelo pomemben pri delu z velikimi podatki.

V Hadoop gruči imamo *TaskTracker* agente, ki pregledujejo stanje vsakega opravila. Če pride do napake pri določenem opravilu, se stanje napake posreduje storitvi *JobTracker*, katera dodeli opravilo drugemu vozlišču znotraj gruče. Število ponovitev določenega opravila lahko nastavimo, ob prekoračitvi se celoten posel prekine. Slika 5 prikazuje primer poteka MapReduce. Večje število Reduce opravil poveča porazdeljenost in hitrost izvedbe. Na Sliki 5 lahko vidimo, da so izhodni podatki operacije Map preusmerjeni po vrednosti ključa na primerno Reduce opravilo. Če se spomnimo našega primera, se mora vsak ključ,

ki ima vrednost Koper, usmeriti na isto Reduce opravilo (Reduce mora dobiti vse temperature za določeno mesto, če hoče izpisati največje).



Slika 6: Prikaz poteka MapReduce

Usmerjanje zapisov v pravo Reduce opravilo se imenuje *Shuffle*. Hadoop omogoča tudi lokalno združevanje izhodnih podatkov iz Map operacije pred pošiljanjem v Reduce z uporabo *kombinirnika* (angl. combiner) (ni prikazan na sliki). Pri zelo velikih podatkovnih zbirkah se pozna hitrejše delovanje z uporabo več Reduce opravil kot pa z uporabo *kombinirnika*. Vsi MapReduce programi, ki se izvajajo znotraj Hadoopa, so napisani v programskem jeziku Java. JobTracker je tisti, ki dodeljuje Java arhivske datoteke (jar) vozliščem znotraj Hadoop gruče za izvajanje Map in Reduce opravil. Jar arhivska datoteka je paket, ki vsebuje MapReduce kodo za izvajanje.

5 PIG

Pig je nastal z namenom, da zmanjša čas, ki ga porabimo za pisanje mapper in reducer programov. S tem se lahko programer bolj osredotoči na analizo podatkovnih zbirk. Razvoj Piga se je začel pri podjetju Yahoo! za lažje rudarjenje po veliki količini podatkovnih zbirk.

Sestavljen je iz dveh komponent:

- jezika za predstavitev podatkovnih tokov, ki se imenuje *Pig Latin* in
- izvajalnega okolja, kjer lahko izvajamo Pig Latin programe.

5.1 Programski jezik

Pig Latin je podatkovno dirigiran jezik, pri katerem se opisuje, kako se vzporedno prebere ali obdela podatke iz enega ali več vhodnih podatkovnih tokov in se nato rezultat shrani vzporedno na enega ali več izhodnih podatkovnih tokov. Od ostalih programskih jezikov se razlikuje po tem, da nima *if* stavkov in *for* zanke. To je zato, ker večina postopkovno in objektno usmerjenih programskih jezikov opisuje krmiljenje toka, medtem ko je podatkovni tok postavljen na stran, Pig Latin pa se osredotoča prav na podatkovni tok.

V naslednjem poglavju si bomo pogledali, kako zgleda programski jezik Pig Latin in kako lažje je programiranje takih programov kot pisanje mapper in reducer programov. Prvi korak pri Pig programih je LOAD, ki naloži podatke iz HDFS, nad katerimi bomo opravljali razne manipulacije. Nato gredo podatki skozi različne transformacije, ki so prevedene v mapper in reducer opravila, brez da bi uporabnik opazil spremembo. Na koncu se z uporabo DUMP prikažejo podatki na zaslonu ali pa z uporabo STORE shranijo v določeno datoteko.

Ukaz LOAD

Podatke znotraj HDFS moramo naložiti v program, zato da jih lahko Pig program uporabi. LOAD ukaz naredi prav to, da naloži določen direktorij ali samo datoteko v program. Če so podatki, do katerih dostopamo, shranjeni v takšnem formatu, da jih Pig ne prepozna, lahko uporabimo določeno funkcijo, ki te podatke prepozna. Z uporabo USING in imena funkcije pred ukazom LOAD lahko naložimo datoteke, ki jih Pig privzeto ne prepozna. Osnovna uporaba ukaza LOAD: *LOAD* '*ime_datoteke* ali *direktorija*'.

Transformacije nad podatki

Logika za transformacijo poskrbi za vse manipulacije nad podatki. Na voljo imamo veliko ukazov, ki jih lahko uporabimo, kot na primer:

- ukaz FILTER nam izloči vrstice, ki nas ne zanimajo,
- ukaz JOIN nam združi dve zbirki podatkov in
- ukaz ORDER nam razvrsti rezultat po nekem merilu.

Prikazan je primer Pig programa, ki vzame podatke iz Twiterja (socialno omrežje) in izbere samo tiste zapise, ki so v angleškem jeziku. Nato rezultate grupira po uporabniku, ki je tvital, in izpiše skupno število odgovorov na njegovo kratko sporočilo.

```
L = LOAD 'hdfs//node/tweet_data';
FL = FILTER L BY iso_language_code EQ 'en';
G = GROUP FL BY from_user;
RT = FOREACH G GENERATE group, SUM(retweets);
```

Ukaza DUMP in STORE

Za generiranje rezultatov iz Pig programa moramo uporabiti vsaj enega izmed ukazov STORE ali DUMP. Pri čiščenju napak Pig programov se pogosto uporablja DUMP ukaz, ki nam izpiše izhodne podatke na zaslon. Pri končni različici programa pa se DUMP zamenja za STORE ukaz, ki rezultate programov shranjuje v datoteke.

Pri poganjanju Pig programov z uporabo izvajalnega okolja Pig znotraj Hadoopa imamo na voljo tri načine:

- zagon Piga iz skripte,
- zagon Piga iz Javinega programa in
- zagon iz ukazne vrstice, imenovane Grunt.

Ne glede na to, kateri način uporabimo, se bo Pig program z uporabo Pig izvajalnega okolja prevedel v niz Map in Reduce opravil. Tak način poenostavi delo in omogoča, da lahko razvijalci posvetijo več časa analizi podatkov, kot pa da se ukvarjajo z opravili Map in Reduce.

5.2 MapReduce primer wordcount

Za primer bomo vzeli MapReduce wordcount program. V Map fazi bo program prebral vsako vrstico tekstovne datoteke, vsako besedo ločil posebej in zapisal poleg 1. Shuffle faza bo uporabila besedo kot ključ in jo posredovala Reducerju, Reducer pa bo seštel pojavitve besed in izpisal rezultat.

Za primer je uporabljena tekstovna datoteka, ki vsebuje naslednji zapis:

```
Mary had a little lamb
  its fleece was white as snow
  and everywhere that Mary went
  the lamb was sure to go.
```

Pig uporablja MapReduce za procesiranje vseh podatkov. Celotno Pig Latin skripto prevede v serijo enega ali več MapReduce poslov, ki so nato izvedeni.

Primer Pig Latin wordcount:

```
// naloži tekstovno datoteko po vrsticah
input = load 'mary' as (line);
// TOKENIZE razdeli vrstico na vsako besedo posebej
// flatten vzame vse, kar vrne TOKENIZE, in shrani vsak zapis posebej v polje word
words = foreach input generate flatten(TOKENIZE(line)) as word;
// grupira vse zapise po vsaki besedi
grpd  = group words by word;
// sešteje besede
cntd  = foreach grpd generate group, COUNT(words);
// vrne rezultat
dump cntd;
```

Pri Pigu nam ni treba skrbeti za Map, shuffle in Reduce faze, saj bo sam poskrbel za pravilno razporeditev v primerno MapReduce fazo.

5.3 Primerjava povpraševalnega jezika in programskega jezika tokov podatkov

SQL je povpraševalni jezik, pri katerem opisujemo vprašanje, na katero želimo odgovor. V primerjavi s SQL je Pig Latin bolj osredotočen na to, kako natančno obdelati vhodne podatke za izračun odgovora. Še ena razlika se pojavi pri odgovarjanju na več vprašanj. Če želimo pri SQL odgovore na več vprašanj, moramo napisati ločena povpraševanja, shranjevati vmesne rezultate v začasne tabele ali pa napisati znotraj enega povpraševanja še

podpovpraševanje. Pig je načrtovan prav za take namene, ko moramo opraviti več podatkovnih operacij. Tako ne potrebujemo začasnih tabel ali pa podatkovnih cevovodov.

Poglejmo si primer primerjave SQL in Pig Latin povpraševanja. V tem primeru želimo grupirati tabelo po določenem ključu in nato pridružiti še drugo tabelo. Združevanje v SQL se zgodi pred grupiranjem, zato moramo napisati z uporabo podpovpraševanja ali dvema povpraševanjima ter z rezultatom, shranjenim v začasni tabeli. V naslednjem primeru je uporabljena začasna tabela.

#SQL primer

```
CREATE TEMP TABLE t1 AS
SELECT customer, sum(purchase) AS total_purchases
FROM transactions
GROUP BY customer;

SELECT customer, total_purchases, zipcode
FROM t1, customer_profile
WHERE t1.customer = customer_profile.customer;
```

#Pig primer:

// najprej se naloži transactions datoteka
```
txns    = load 'transactions' as (customer, purchase);
```
// grupiramo po polju customer
```
grouped = group txns by customer;
```
// seštejemo skupno vrednost nakupov po stranki
```
total   = foreach grouped generate group, SUM(txns.purchase) as tp;
```
// naložimo customer_profile datoteko
```
profile = load 'customer_profile' as (customer, zipcode);
```
// združimo grupiran rezultat iz transactions in podatke iz customer_profile
```
answer  = join total by group, profile by customer;
```
// izpišemo rezultat
```
dump answer;
```

Kot smo videli, je Pig načrtovan tako, da lahko sprejme tudi datoteko, ki vsebuje nekonsistentno bazo podatkov in ne normiranih podatkov. Zato ni pomembno, da imamo podatke naložene v tabelah, ampak je pomembno, da so shranjeni znotraj HDFS.

5.4 Pig in MapReduce

Cilj razvijalcev Piga je, da postane Pig materni jezik za okolja, ki uporabljajo vzporedno obdelavo podatkov. Pig omogoča uporabniku dodatne operacije za obdelavo podatkov, kot so join, filter, group by, union ... Pri MapReduce pa nekaterih od teh operacij ni in jih mora uporabnik napisati sam. Določene kompleksne implementacije standardnih operacij v Pigu omogočajo nadzor nad operacijami Reduce, katerim se porazdeli podatke. Tako ne pride do tega, da ima en Reducer 10-krat več podatkov za obdelati kot pa ostali, ampak se podatki pravično porazdelijo. Pig lahko analizira Pig Latin kodo in razume pretok podatkov, ki ga opisuje uporabnik. Tako početje omogoča zgodnje odkrivanje napak in optimizacijo, kar je še ena prednost pred MapReduceom. Z vsem tem, kar je bilo napisano, lahko trdimo, da je Pig Latin preprostejši za uporabo kot pisanje Java kode za MapReduce.

5.5 Uporaba Piga

Uporaba Piga se deli na tri kategorije: podatkovni cevovodi (angl. data pipelines), raziskave na neobdelanih podatkih in iterativno analiziranje. V naslednjih nekaj vrsticah bom opisal primer podatkovnih cevovodov, ki je najbolj uporabljan. Podatkovni cevovodi služijo za analizo in premikanje podatkov iz enega strežnika na drugega. Običajen primer uporabe podatkovnih cevovodov sta čiščenje in priprava dnevniških datotek, ki nastanejo na raznih spletnih strežnikih. Iz njih odstranjuje poškodovane podatke in združuje znane podatke o uporabnikih iz podatkovne baze z novimi. Z uporabo Piga lahko ustvarimo model obnašanja, v katerem napovemo, kako se bodo določeni uporabniki odzvali na določen element, ki ga objavimo na spletni strani. To zgleda tako, da Pig razišče vse interakcije uporabnika z določeno spletno stranjo in nato razvrsti uporabnike v različne segmente. Nato se za vsak segment naredi matematični model, kateri napove, kako se bodo člani segmenta odzvali na določeno reklamo ali določen članek na spletni strani. Tako lahko na določeni spletni strani prikazujemo za vsakega uporabnika posebej določene vsebine, ki bodo verjetno pritegnile njegovo pozornost. Za druga dva primera si lahko več preberemo v knjigi.[3]

[3] Alan Gates, Programming Pig, "O'Reilly Media, Inc, 2011

6 PRIMER MAPREDUCE PROGRAMA

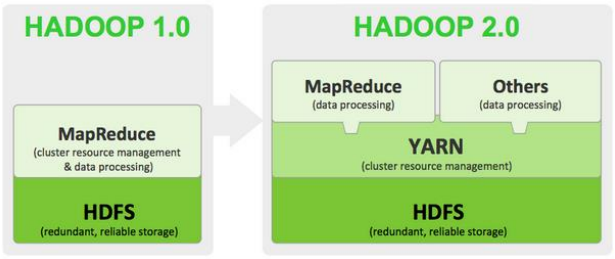
Prikazal bom osnovni program za Mapreduce, ki je napisan v programskem jeziku Python. Za tak primer sem se odločil, ker je prikazano, kako lahko s pomočjo Pretočnega API-ja zaženemo tudi skripte, ki niso napisane v Javi. Edina pogoja, ki ju morajo izpolnjevati skripte, sta branje vhodnih podatkov iz STDIN in izpisovanje izhodnih podatkov v STDOUT.

Program bom imenoval wordCount. Napisal bom dve skripti v Pythonu, kateri bosta obdelali besedila v določenem imeniku. Cilj MapReduce skript bo, da izpiše skupno število pojavitev besed neke zbirke knjig.

6.1 Namestitev Hadoop

Za prikaz programa wordCount sem namestil Hadoop na enem računalniku tako, da sta imensko in podatkovno vozlišče tekla oba na enem računalniku.

Za ta primer sem uporabil Hadoop 2.2.0, kateri sodi med zadnje različice sistema Hadoop in v kateremu se že pojavlja Hadoop Yarn (od različice 2.0 naprej). Hadoop Yarn je tehnologija, ki skrbi za nadziranje sistemskih virov. Za lažjo predstavo je tukaj Slika 7.



Slika 7: Hadoop Yarn

Vse skupaj bom izvajal na operacijskem sistemu windows 7 professional znotraj programa Oracle VM VirtualBox, kjer sem namestil operacijski sistem linux ubuntu 14.04, ki je izšel aprila leta 2014.

Znotraj operacijskega sistema Ubuntu sem namestil JDK Javo, brez katere Hadoop ne more delovati. Ustvaril sem novega uporabnika, ki bo poganjal Hadoop (v mojem primeru je to hduser). Naslednja pomembna nastavitev je delujoči SSH dostop.

S spletne strani[4] sem prenesel različico 2.2.0, katero sem razširil v direktorij /usr/local/hadoop.

Ta korak je zelo pomemben, saj lahko z nepravilnimi nastavitvami onemogočimo pravilno delovanje. Nastaviti moramo sledeče spremenljivke. V mojem primeru sem v .bashrc dodal spodnje nastavitve:

```
export JAVA_HOME=/usr/lib/jvm/jdk/

export HADOOP_INSTALL=/usr/local/hadoop

export PATH=$PATH:$HADOOP_INSTALL/bin

export PATH=$PATH:$HADOOP_INSTALL/sbin

export HADOOP_MAPRED_HOME=$HADOOP_INSTALL

export HADOOP_COMMON_HOME=$HADOOP_INSTALL

export HADOOP_HDFS_HOME=$HADOOP_INSTALL

export YARN_HOME=$HADOOP_INSTALL
```

Pri glavni konfiguraciji Hadoopa moramo spreminjati nastavitve v štirih datotekah, ki se nahajajo v /usr/local/hadoop/etc/hadoop. To so:

```
core-site.xml
yarn-site.xml
mapred-site.xml.template.
Hdfs-site.xml
```

Pred prvim zagonom moramo formatirati imensko vozlišče, kar naredimo z ukazom *hdfs namenode -format*. Sedaj lahko zaženemo Hadoop storitve z ukazoma:

`start-dfs.sh in start-yarn.sh`. Delovanje gruče ustavimo s `stop-dfs.sh in stop-yarn.sh`. Z ukazom *jps* dobimo seznam storitev, ki tečejo na trenutnem računalniku.

[4] Vir: http://www.apache.si/hadoop/common/

Primer jps ukaza:

```
hduser@tomaz-VirtualBox:/usr/local/hadoop$ jps

18509 Jps
17107 NameNode
17170 DataNode
17252 ResourceManager
17384 NodeManager
17458 SecondaryNameNode
```

6.2 Python in MapReduce

Kot sem že prej omenil, bom za ta primer uporabil Hadoopov Pretočni API. To je pripomoček, ki ga dobimo pri namestitvi Hadoopa. Omogoča nam ustvarjanje in zaganjanje MapReduce poslov z uporabo raznih skript. To pomeni, da lahko uporabimo skripto, ki smo jo napisali in nad njo poženemo MapReduce opravilo. V našem primeru bomo uporabili Pretočni API zato, da si bomo lahko podali rezultate iz map v reduce preko STDIN in STDOUT. V Pythonu bomo preprosto določili, naj skripta uporabi *sys.stdin* za branje vhodnih podatkov in *sys.stdout* za pisanje izhodnih podatkov. Za vse ostalo bo poskrbel Hadoopov Pretočni API.

Za vsak MapReduce posel dobimo par ključ in vrednost, tako velja tudi pri Pretočnem API-ju. Pri uporabi *pretakanja* so vhodni in izhodni podatki vedno predstavljeni kot tekst. Za ločevanje ključa in vrednosti je uporabljen *tab* (\t) znak.

Primer :

```
ključ1 \t vrednost1 \n
ključ 2 \t vrednost2 \n
ključ 3 \t vrednost3 \n
```

Vsaka vrstica vsebuje po en par ključ in vrednosti, kateri so združeni po istem ključu in sprejeti kot vhodni podatki za Reducer.

6.2.1 Skripti mapper in reducer

Naredil sem novo datoteko z imenom mapper.py, v katero sem napisal map program. Koda je zelo preprosta. Najprej preberemo vhodne podatke iz STDIN in jih nato ločimo po besedah z *line.split()* ter v STDOUT izpišemo vsako besedo (ključ) v svojo vrstico in zraven izpišemo še številko 1 (vrednost).

```
#!/usr/bin/env python

import sys
# prebere vrstico
for line in sys.stdin:
# odstrani prazne prostore
    line = line.strip()
#loči besede
    words = line.split()
#za vsako besedo izpiše besedo in vrednost 1
    for word in words:
        print '%s\t%s' % (word, 1)
```

Reducer prebere iz STDIN podatke, združene po določenem ključu, katere je predhodno obdelal mapper. Prebere vsako vrstico posebej in obdela zapise. Rezultate shranjuje v polje na tak način, da je beseda ključ v polju in vrednost število pojavitev besede. Nato z uporabo STDOUT izpiše besedo in skupno število pojavitev besede.

```
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# prebere vrstico
for line in sys.stdin:
# odstrani prazne prostore
    line = line.strip()
# razdeli vrstico na word in count
    word, count = line.split('\t', 1)

    try:
#besedo v spremenljivki count spremenimo v število
        count = int(count)
    except ValueError:
        continue
#če beseda že obstaja, seštejemo vrednost
    if current_word == word:
        current_count += count
    else:

        if current_word:
#izpiše rezultat stdout
            print '%s\t%s' % (current_word, current_count)
        current_count = count
```

```
        current_word = word
```

#izpiše zadnjo besedo

```
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

6.3 Testiranje delovanja skripte

Preden zaženemo naši skripti v Hadoopu, ju moramo testirati. V mojem primeru sem skripti testiral v konzoli znotraj Linux operacijskega sistema. Enostavno vpišemo nek vhodni niz v konzolo, katerega z uporabo cevovoda pošljemo kot vhodni parameter za našo skripto.

Primer samo za mapper.py skripto:

```
tomaz@tomaz-VirtualBox:~/Desktop$ echo "test map reduce mapper tester reducer map map te
st" | /home/tomaz/Desktop/mapper.py
test    1
map     1
reduce  1
mapper  1
tester  1
reducer 1
map     1
map     1
test    1
```

Slika 8: Test mapper.py skripte

Izhodni podatki naše skripte so pari ključ/vrednost, in sicer je vsak par napisan v svoji vrstici. Vse besede, ki smo jih vpisali kot vhodni parameter, so se izpisale. Z veliko verjetnostjo lahko sklepamo, da naša skripta pravilno deluje in lahko testiramo naprej. V naslednjem testu bomo vključili še reducer.py skripto. Poleg mapper.py bomo postavili cevovod, ki bo služil za vhodne podatke reducer.py skripti, ki jih bo nato izpisala. Pred skripto reducer.py sem dodal sort, ki sortira besede po prvem stolpcu.

```
tomaz@tomaz-VirtualBox:~/Desktop$ sudo echo "test map reduce mapper tester reducer map m
ap tester test reduce" | /home/tomaz/Desktop/mapper.py | sort -k1,1 | /home/tomaz/Deskto
p/reducer.py
map     3
mapper  1
reduce  2
reducer 1
test    2
tester  2
```

Slika 9: Test reducer.py skripte

Iz izpisanega rezultata lahko vidimo, da reducer.py skripta deluje pravilno, saj je seštevek pojavitve besed pravilen. Sedaj lahko zaženemo skripti znotraj Hadoopa.

6.4 Zagon skripte v HDFS

Za naš primer sem uporabil knjige, zapisane v txt formatu, ki so prosto dostopne na spletnem naslovu.[5]

Izbrane knjige sem prenesel na svoj računalnik in poskrbel, da so shranjene v utf-8 kodiranju. Datoteke sem nato preimenoval v besedilo.txt, besedilo1.txt in besedilo3.txt za lažje ločevanje datotek.

Če želimo pognati naš MapReduce posel nad prenesenimi datotekami, jih moramo najprej prenesti v datotečni sistem Hadoop. To naredimo z uporabo ukaza *copyFromLocal*. Primer je podan za eno datoteko. Postopek ponovimo za vsako datoteko posebej.

```
hduser@tomaz-VirtualBox:/usr/local/hadoop$ bin/hadoop dfs -copyFromLocal
/home/hduser/besedilo.txt /wordscount/
```

Z ukazom *dfs –ls* lahko pogledamo seznam datotek ali map znotraj določenega direktorija.

```
hduser@tomaz-VirtualBox:/usr/local/hadoop$ bin/hadoop dfs -ls /wordscount/
Found 3 items
-rw-r--r--   1 hduser supergroup     594933 2014-06-29 15:23 /wordscount/besedilo.txt
-rw-r--r--   1 hduser supergroup     502760 2014-08-05 17:16 /wordscount/besedilo2.txt
-rw-r--r--   1 hduser supergroup     421884 2014-08-05 17:16 /wordscount/besedilo3.txt
```

Slika 10: Seznam datotek znotraj HDFS

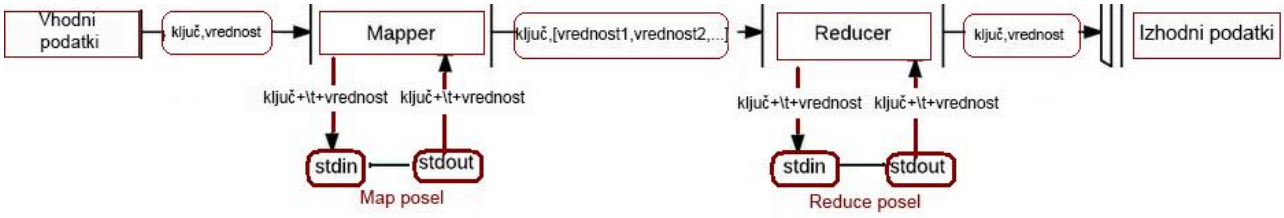
Sedaj imamo vse pripravljeno za zagon naših skript znotraj Hadoopa. Kot sem že prej omenil, bom uporabil Hadoopov Pretočni API za zaganjanje Python skript. Pri podajanju podatkov med skriptama mapper in reducer bom uporabil STDIN in STDOUT. Za začetek se postavimo v namestitveni direktorij Hadoopa, kjer bomo pognali skripti z ukazom:

```
hduser@tomaz-VirtualBox:/usr/local/hadoop$ bin/hadoop jar
./share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar -file /home/hduser/mapper.py
-mapper /home/hduser/mapper.py
-file /home/hduser/reducer.py -reducer /home/hduser/reducer.py
-input /wordscount/* -output /wordscount/izpis.out
```

Napisali smo, da naj znotraj orodij najde lib direktorij, kjer se nahaja Pretočni API. Ker se mapper.py in reducer.py ne nahajata znotraj HDFS, uporabimo *–file* opcijo, ki naredi povezavo na trenutnem delovnem direktoriju iz našega lokalnega direktorija tako, da ga lahko -mapper in -reducer funkciji uporabita. Na koncu podamo še direktorij, kjer se nahajajo naše knjige in datoteka za izpis izhodnih podatkov.

[5] Vir: http://www.gutenberg.org/ebooks/search/

Na spodnji sliki lahko vidimo potek pretočnega procesa. Mapper opravilo vzame vhodne podatke in jih pretvori v vrstice v STDIN. Mapper.py skripta naredi svoje delo in vrne rezultat v STDOUT, nato Mapper pobere vse rezultate in jih razvrsti v pare ključ/vrednost. Ključ in vrednost loči tako, da je prva vrednost do 'tab' znaka ključ in za tem znakom je vrednost. Reducer opravilo pretvori pare ključ/vrednost v vrstice in jih postavi v STDIN od procesa reducer.py. Skripta opravi svoje delo in vrne rezultat v STDOUT. Reduce opravilo pobere izhodne vrstice in pripravi ključe/vrednosti za končni izpis.



Slika 11: Potek pretočnega procesa

Spodnja slika prikazuje končni izpis MapReduce posla. Vidimo lahko, da je za naš primer Hadoop dodelil štiri mape opravila. Število map opravil določi Hadoop glede na število DFS blokov, ki jih zasedejo vhodne datoteke, lahko pa tudi ročno nastavimo minimalno število Map opravil. Merilo, s katerim si lahko pomagamo, je 10 do 100 map opravil na vozlišče. Za ta primer je nastalo samo eno Reduce opravilo, saj je privzeta vrednost pri *pretakanju* 1. Lahko bi nastavili večje število z ukazom *–D mapreduce.reduce.tasks=število*.

```
14/08/05 17:22:04 INFO mapreduce.Job: Counters: 43
        File System Counters
                FILE: Number of bytes read=2846932
                FILE: Number of bytes written=6108057
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=1678003
                HDFS: Number of bytes written=320540
                HDFS: Number of read operations=15
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=4
                Launched reduce tasks=1
                Data-local map tasks=4
                Total time spent by all maps in occupied slots (ms)=130849
                Total time spent by all reduces in occupied slots (ms)=12124
        Map-Reduce Framework
                Map input records=47687
                Map output records=302419
                Map output bytes=2242088
                Map output materialized bytes=2846950
                Input split bytes=395
                Combine input records=0
                Combine output records=0
                Reduce input groups=29925
                Reduce shuffle bytes=2846950
                Reduce input records=302419
                Reduce output records=29925
                Spilled Records=604838
                Shuffled Maps =4
                Failed Shuffles=0
                Merged Map outputs=4
                GC time elapsed (ms)=883
                CPU time spent (ms)=6640
                Physical memory (bytes) snapshot=854347776
                Virtual memory (bytes) snapshot=3985440768
                Total committed heap usage (bytes)=578633728
        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=1677608
        File Output Format Counters
                Bytes Written=320540
14/08/05 17:22:04 INFO streaming.StreamJob: Output directory: /wordscount/izpis.out
```

Slika 12: Končni izpis Hadoop ukaza

Lahko pogledamo, ali je nastala izhodna datoteka v /wordscount/izpis.out, kot smo določili. Uporabimo ukaz *dfs –ls*, ki nam prikaže vsebino direktorija.

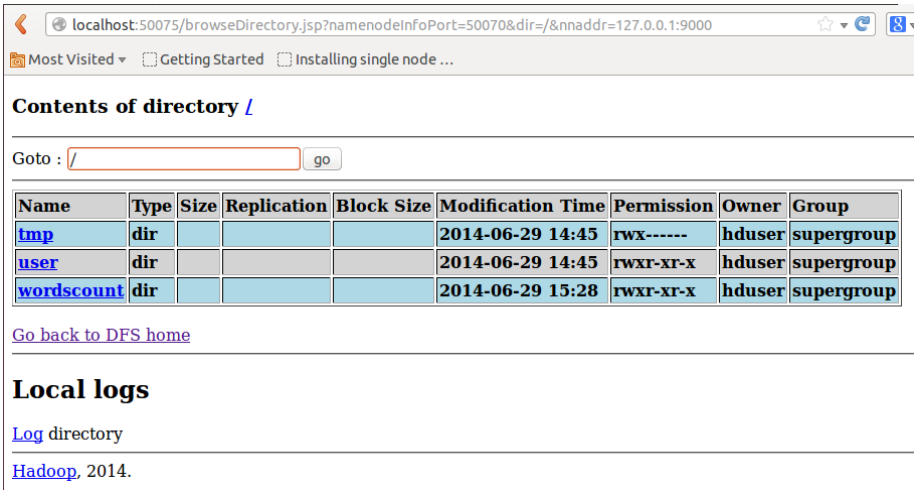
```
hduser@tomaz-VirtualBox:/usr/local/hadoop$ bin/hadoop dfs -ls
/wordscount/izpis.out

Found 1 items
/wordscount/wc.out/part-00000     &lt;r 1&gt;   903193  2014-06-29 13:00
```

Vsebino izhodne datoteke lahko pogledamo z uporabo konzole in ukaza *cat*, ki nam izpiše vsebino datoteke na zaslon.

```
you_."  1
young   154
young)  1
young,  8
young--only     1
young.  5
younger 14
youngster       3
your    733
your'n, 1
yourn,  1
yourn.  1
yours   11
yours,  10
yours,' 2
yours.  4
yours." 4
yours?  1
yourself        50
yourself!       1
yourself!"      4
yourself,       13
yourself,"      1
yourself.       1
yourself."      4
yourself.'      2
yourself?       1
yourself?"      4
yourselves      7
yourselves,     2
yourselves.     1
youth   6
youth,  5
youth.  2
youth.' 1
youth?" 2
youthful        1
zeal    2
zealous 1
zebras--all     1
zenith  1
zephyr  1
zero,   2
zero-point,     2
zest    2
zigzag  2
 I      1
 and    1
        1
Project 3
hduser@tomaz-VirtualBox:/usr/local/hadoop$ bin/hadoop dfs -cat /wordscount/izpis.out/part-00000
```

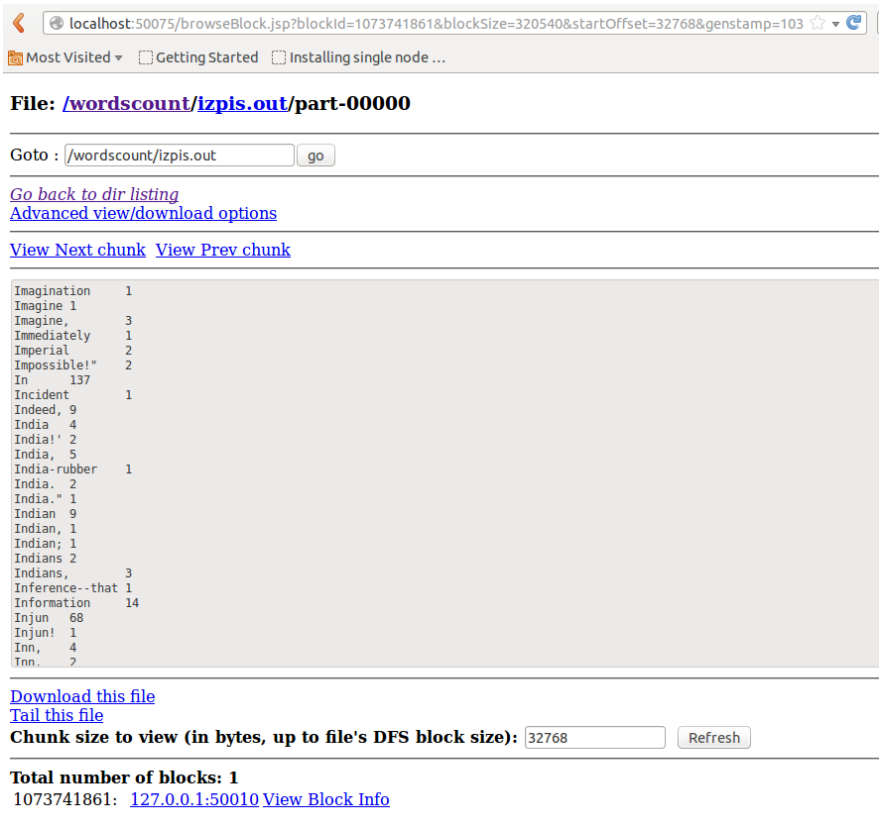
Slika 13: Delni izpis rezultata z uporabo konzole

Drugi način za vpogled v vsebino izhodne datoteke in ostale vsebine datotečnega sistema je z uporabo osnovnega spletnega vmesnika, ki ga ponuja Hadoop. Dostop je omogočen na *http://localhost:50075*. Spodnja slika prikazuje izgled vmesnika.



Slika 14: Izgled Hadoop spletnega vmesnika

Z uporabo spletnega vmesnika lahko enostavno pogledamo nastalo izhodno datoteko za našo MapReduce operacijo. Na Sliki 15 je prikazan samo določen del izpisa. Vidimo lahko, da je poleg vsake besede zapisano število pojavitev besede tako, kot smo pričakovali.



Slika 15: Prikaz delnega rezultata v spletnem vmesniku

6.5 Mnenje o napisanem primeru

Kot smo videli na primeru, je pisanje skripte za MapReduce v različnih programskih jezikih zelo enostavno, saj za komunikacijo med skriptama poskrbi Pretočni API. S takim pristopom omogočimo razvijalcem, ki se spoznajo na programske jezike, ki ni Java, da lahko pišejo programe tudi za MapReduce opravila. Vse, kar mora njihov program podpirati, sta branje podatkov iz STDIN in pisanje podatkov v STDOUT.
Zagon skripte je enostaven, saj moramo poleg napisanih skript napisati še, kje so vhodni podatki in kam naj se zapiše rezultat. Pretočni API nam omogoča nastavljanje še dodatnih nastavitev, kot je število Reduce opravil za boljšo optimizacijo izvajanja glede na določen primer, za pretvorbo Python skripte v ustrezne MapReduce operacije pa poskrbi Pretočni API. Po mojih izkušnjah je najtežji del nastaviti vse parametre in inštalirati vse komponente za Hadoop. Velikokrat se zgodi, da Hadoop nima nastavljenih pravih poti za določene spremenljivke in takrat se za iskanje takih napak izgubi največ časa.

7 ZAKLJUČEK

V zaključni nalogi smo spoznali, kaj so masovni podatki in kje nastajajo. Pri obdelavi in shranjevanju takih podatkov potrebujemo primeren sistem, zato sem natančneje opisal datotečni sistem Hadoop, ki je prav temu namenjen. Spoznali smo, da je Hadoop narejen za delo na sistemu, ki ima povezanih več računalnikov v omrežje. Prednost takega sistema je prav ta, da se delo razdeli na več delov in vsak računalnik znotraj omrežja opravi določeno delo. Tako se celotno delo izvede hitreje. Strojna oprema, na kateri teče Hadoop, ima za zahtevo le to, da na njej lahko teče Java. Pomembna lastnost, ki smo jo spoznali, je ta, da ima večje bloke v primerjavi s standardnimi datotečnimi sistemi. Namen tega je optimizirati branje, saj se bralna glava diska manj premika in izgublja čas. Za shranjevanje in posredovanje blokov poskrbi podatkovno vozlišče, ki dela na zahtevo imenskega vozlišča. V imenskem vozlišču se hranijo informacije, na katerem podatkovnem vozlišču so shranjeni bloki. Brez te informacije je sistem neuporaben, zato vsebuje Hadoop več varnostnih mehanizmov, ki poskrbijo, da ne pride do izgube podatkov.

Glavno vlogo pri obdelavi podatkov pa ima programski model MapReduce, ki je vgrajen v jedro platforme Hadoop. Spoznali smo, da je MapReduce razdeljen na dve opravili Map in Reduce. Predstavljen je bil tudi praktičen primer delovanja MapReduce, ki je prikazal razne korake do končnega rezultata. Poudariti je treba, da Mapreduce izkorišča porazdeljeni sistem tako, da porazdeli Map opravila po raznih vozliščih tako, da se delo čim prej zaključi.

Za lažje pisanje MapReduce programov sem izpostavil uporabo programskega orodja, imenovanega Pig. Njegov prvotni namen je zmanjšanje časa, ki ga porabimo pri pisanju MapReduce programov. Omogoča nam dodatne operacije pri manipulaciji podatkov in lažje odpravljanje napak v kodi.

Kot zadnje poglavje pa sem predstavil osnovni program za MapReduce. Namen tega primera je bil prikazati, kako lahko enostavno v izbranem programskem jeziku napišemo skripto in jo z uporabo pripomočka Pretočni API zaženemo v MapReduce. Natančno sem opisal postopek, kako napisati skripti za map in reduce, ki kot končni rezultat izpišeta število pojavitev besed v zbirki knjig. Primer je bil uspešno izveden in prikazan je tudi delni rezultat. Celoten proces se je odvijal na enem računalniku, na katerem sta bila nameščena podatkovno in imensko vozlišče. S predstavljenim primerom sem tudi sam boljše spoznal delovanje MapReduce in njegovo enostavno uporabo.

Sam stvaritelj Hadoopa, Doug Cutting, zatrjuje [1] »In the future we'll be able to store and process more data than we can now«. Če verjamemo njegovim besedam, lahko trdimo, da se bo Hadoop z leti še izboljšal in se uporabljal na več področjih.

8 LITERATURA IN VIRI

[1] J. Koetsier, „venturebeat.com,“ april 2014. [Elektronski]. Available: http://venturebeat.com/2014/04/10/hadoop-founder-says-future-of-big-data-looks-like-well-hadoop/.

[2] T. White, Hadoop the definitive guide, tretja ured., O'Reilly Media / Yahoo Press, 2012.

[3] M. Olson, „HADOOP: Scalable, flexible Data Storage an Analysis,“ 2010. [Elektronski]. Available: http://www.cloudera.com/content/dam/cloudera/Resources/PDF/Olson_IQT_Quarterly_Spring_2010.pdf.

[4] J. Dean in S. Ghemawat, „Mapreduce: Simplified Data Processing on Large Clusters,“ 2014. [Elektronski]. Available: http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[5] E. Dumbill, „http://radar.oreilly.com/,“ 2012. [Elektronski]. Available: http://radar.oreilly.com/2012/02/what-is-apache-hadoop.html.

[6] Apache, „Apache Hadoop,“ [Elektronski]. Available: http://hadoop.apache.org/.

[7] R. Chansler, H. Kuang, S. Radia, K. Shvachko in S. Srinivas, „aosabook,“ [Elektronski]. Available: http://www.aosabook.org/en/hdfs.html.

[8] D. Borthakur, „http://hadoopblog.blogspot,“ september 2009. [Elektronski]. Available: http://hadoopblog.blogspot.com/2009/09/hdfs-block-replica-placement-in-your.html.

[9] M. G. Noll, „michael-noll,“ 2011. [Elektronski]. Available: http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/.

[10] S. S. A. jaffri, „sharafjaffri.blogspot,“ 2014. [Elektronski]. Available: http://sharafjaffri.blogspot.com/2014/05/running-your-example-on-hadoop-220.html.

[11] „bigdatahandler,“ 2013. [Elektronski]. Available: http://bigdatahandler.com/hadoop-hdfs/installing-single-node-hadoop-2-2-0-on-ubuntu/.

[12] Zhi, „codesfusion.blogspot,“ 2013. [Elektronski]. Available: http://codesfusion.blogspot.com/2013/10/setup-hadoop-2x-220-on-ubuntu.html.

[13] „hortonworks,“ hortonworks, [Elektronski]. Available: http://hortonworks.com/hadoop/yarn/.

[14] A. Gates, Programming Pig: Dataflow Scripting with Hadoop, O'Reilly Media, 2011.

[15] „Wikipedia,“ [Elektronski]. Available: http://en.wikipedia.org/wiki/Glavna_stran.

[16] SAS, „sas.com,“ [Elektronski]. Available: http://www.sas.com/en_us/insights/big-data/hadoop.html.

[17] E. Baldeschwieler, „Hortonworks,“ julij 2012. [Elektronski]. Available: http://hortonworks.com/blog/thinking-about-the-hdfs-vs-other-storage-technologies/.

[18] R. L. Hornbeck, „http://datatactics.blogspot.com/,“ februar 2013. [Elektronski]. Available: http://datatactics.blogspot.com/2013/02/batch-versus-streaming-differentiating.html.

[19] V. Beal, „webopedia,“ [Elektronski]. Available: http://www.webopedia.com/TERM/U/unstructured_data.html. [Poskus dostopa 2014].

[20] R. Barrett, B. de Supinski, E. Dube, C. Edwards, P. Henning, S. Langer, P. McCormick in A. McPherson, „Programming Models,“ 2011. [Elektronski]. Available: https://asc.llnl.gov/exascale/exascale-pmWG.pdf.

[21] D. deRoos, T. Deutsch, C. Eaton, G. Lapis in P. C. Zikopoulos, Understanding Big Data, McGraw-Hill, 2012.