Arhitekture big data sustava stvarnog vremena

Čamagajevac, Alen

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Diplomski studij računarstva

ARHITEKTURE BIG DATA SUSTAVA REALNOG VREMENA

Diplomski rad

Alen Čamagajevac

Osijek, 2018

SADRŽAJ

1. UVOD

Sustavi koji mogu spremiti velike količine podataka bili su prisutni dugo vremena, no problem nastaje kod repliciranja, očuvanja integriteta i ponajviše interaktivnog dohvaćanja takvih podataka. Kompanije su davno prepoznale da bi čuvanje svih podataka koje sakupe dalo veliku korist njihovom poslovanju. No samo čuvanje podataka, u tehničkom smislu, pokazalo se značajno drukčije od same upotrebe takve velike količine podataka. Razni podatkovni centri nudili su mogućnosti skladištenja datoteka ogromnih dimenzija te su na lak način riješili problem hladnog čuvanja podataka. Ovaj način čuvanja odnosi se na pohranjivanje podataka koji su uglavnom ne aktivni, ne mijenjaju se ali najbitnije, kojima se ne pristupa često. Kako bi podaci dali nekakvu vrijednost poslovanju, oni definitivno ne mogu ostati statični te im se treba kontinuirano pristupati. Drugi veliki problem u korištenju podataka nastaje prilikom očuvanja integriteta podataka. Jasno je kako se podatkovna konzistencija i točnosti moraju kontinuirano provjeravati tokom skladištenja. Također, pojavljuje se i problem repliciranja podataka u slučaju kvarova. Upravo ovi problemi nastoje se konsolidirati pod imenom *big data*. Kako bi se definirao pojam *big data,* IT industrija proizišla je s definicijom koja je poznata kao četiri velika V u *big data* svijetu[1]. Volumen (engl. *Volume*) definira kako je količina podataka ogromna. Neke projekcije velikih organizacija ukazuju na to kako se svaki dan generira 2.3 triliona gigabajta novih podataka. Broj ljudi kontinuirano raste. Ukupna svjetska populacija iznosi 7.6 milijardi ljudi. Pretpostavlja se kako će 2020 godine broj aktivnih korisnika raznih pametnih telefona doseći brojku od 2.87 milijardi a broj međusobno spojenih uređaja doseći će 30.73 milijardi[2]. Nije teško za pretpostaviti kako postoji još puno pokazatelja koji nam ukazuju na to da se podaci kreiraju u velikim količinama te kako postoji ogromna potreba za njihovim analiziranjem i čuvanjem. Još jedan pokazatelj je raznolikost (engl. *variety*) podataka. Popularnost raznih internetskih platformi, pametnih uređaja i socijalnih mreža otvara vrata za nekakve jako velike brojke. U 2014 godini procjenjivalo se da postoji 420 milijuna nosivih, pametnih uređaja. Svi oni služe u različite svrhe, od praćenja medicinskih informacija pacijenata do praćenja trenutne vlage u zraku. Brzina (engl. *velocity*) kojom se podaci generiraju također je zapanjujuća. Pretpostavlja se kako moderna autonomna vozila generiraju preko 4000 gigabajta podataka u samo sat vremena vožnje. Istinitost (engl. *veracity*) podataka predstavlja zadnju ključnu točku paradigme velikih V. Samo generiranje i skladištenje podataka nije od velike koristi ako se integritet podataka ne može potvrditi. Tvrtka

IBM tvrdi kako loša kvaliteta čuvanih podataka ošteti američku vladu za 3.1 triliona dolara svake godine. Sve ovo ukazuje nam na jasnu potrebu za korištenjem i savladavanjem tehnologija koje mogu na efikasan način skladištiti, pristupati, očuvati integritet te dohvatiti željene podatke u domeni velike količine podataka[3].

U ovome radu objasniti će se tehnologije vezane uz obradu velike količine podataka te će se poseban naglasak staviti na obradu podataka u realnom vremenu. U drugom poglavlju opisati će se Hadoop programski okvir te njegove najbitnije komponente. Treće poglavlje objašnjava obradu serije podataka, dok se četvrto poglavlje bavi obradom toka podataka. U petom poglavlju opisati će prednosti i mane serijske obrade i obrade toka podataka. Ovdje će se definirati *lambda* arhitektura koja istovremeno nastoji iskoristiti prednosti serijske obrade i obrade toka podatka, kako bi dala korisnicima relevantne rezultate u zahtjevima realnog vremena. Primjena ovakvog pristupa biti će prikazana unutar .NET okvira gdje će se pokazati kako je moguće implementirati *lambda* arhitekturu da bi se omogućila obrada statistika o korištenju aplikacije.

1.1. Zadatak rada

Opisati distribuirane sisteme koji upravljaju većim količinama podataka baziranih na *Hadoop* okviru i tehnologijama koje se vežu za *Hadoop*. Objasniti HDFS i *MapReduce* za pohranu i analiziranje podataka, *Pig* i *Spark* za pisanje kompleksnijih skripti te korištenje baza podataka za spremanje podataka. Pokazati sve na konkretnim primjerima. Posebno objasniti obradu toka podataka i njihovu obradu na *Hadoop* čvorovima pomoću *Kafke* i *Spark* okvira.

2. OSNOVE RADA HADOOP OKVIRA

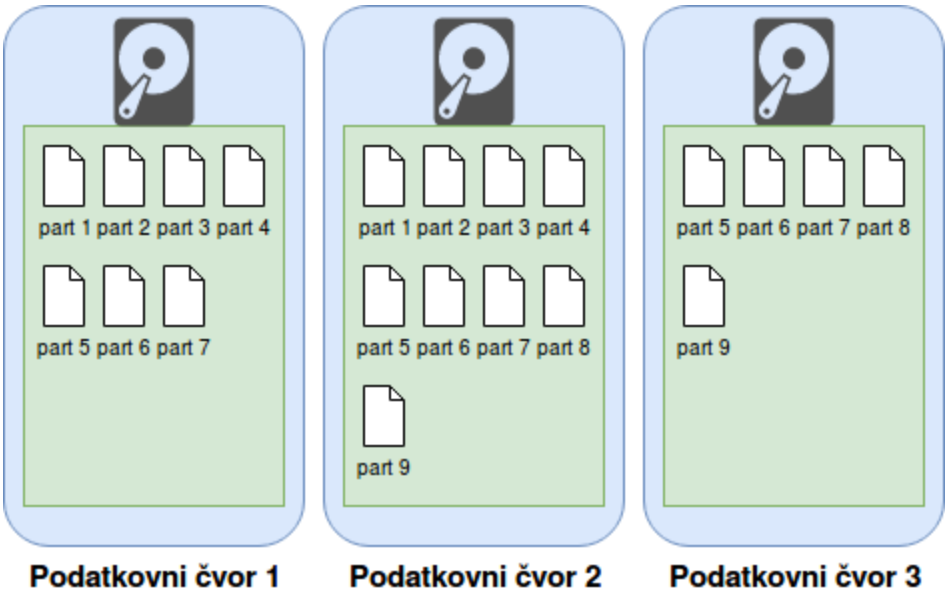
Postoje mnoge definicije Hadoop programskog okvira. Definicija koju nam daje Apache, organizacija koja aktivno razvija i održava projekt, definira Hadoop kao softver otvorenog koda za pouzdano i skalabilno distribuirano upravljanje podatcima[4]. Upravljanje podacima u ovom slučaju može se gledati kroz dvije komponente. Prva je distribuirano spremanje podataka i očuvanje integriteta spremljenih podataka. Druga komponenta je pružanje programskog sučelja

kako bi se podacima moglo upravljati na jednostavan način. Veliki naglasak stavlja se i na skalabilnost koju Hadoop omogućuje. Ovaj okvir dizajniran je tako da se može skalirati horizontalno od jednog računala do više tisuća računala, gdje svako nudi svoj lokalni diskovni prostor i mogućnosti procesiranja podataka. Jedna od najbitnijih komponenti koja omogućuje skaliranje i obradu podataka je Hadoop raspodijeljeni datotečni sustav (engl. *Hadoop distribuded file system*), u daljnjem tekstu referenciran kao *HDFS*.

Kao i većina datotečnih sustava *HDFS* podržava tradicionalnu hijerarhijsku organizaciju datoteka. Korisnici sustava mogu kreirati direktorije te spremati datoteke unutar ovih direktorija. Imenski prostor datotečnog sustava je sličan postojećim datotečnim sustavima. Korisnici mogu kreirati i brisati datoteke, micati datoteke iz jednog direktorija u drugi ili preimenovati datoteku. No *HDFS* nije jedina komponenta *Hadoop* okvira. Uz datotečni sustav *Hadoop* se sastoji i od *YARN* upravitelja resursima te *Hadoop MapReduce* programskog modela.

2.1. Spremanje podataka unutar Hadoop distribuiranog datotečnog sustava

Hadoop datotečni sustav je komponenta sustava koja je zadužena za spremanje podataka. Ova komponenta sposobna je na rastaviti datoteke te rastavljene komponente spremiti na više različitih lokacija na pouzdan i siguran način. Svaka datoteka rastavlja se na više blokova (u osnovnom slučaju 3) veličine 64 megabajta (veličina samog bloka može se konfigurirati). Samim rastavljanjem datoteke rješava se problem ograničenog diskovnog prostora jednog tvrdog diska. Ovakva datoteka može biti spremljena na više različitih diskova u više različitih čvorova unutar grozda. Iz ovog razloga spremanje datoteka koje su veće od slobodnog diskovnog prostora na jednom računalu, postaje moguće. Također, kako je datoteka sada raspodijeljena na više različitih čvorova, svako od njih može procesirati komad datoteke. Nije nužno da se komad datoteke nalazi na čvoru koje je zaduženo za obradu tog dijela datoteke, no datotečni sustav uvijek nastoji naći fizički najbliže računalo za obradu, jer tako olakšava i ubrzava pristup podacima. Operacije koje se izvode nad dijelom datoteke na različitim računalima mogu se izvoditi paralelno. Time se bolje iskorištavaju dostupni računalni resursi unutar grozda.

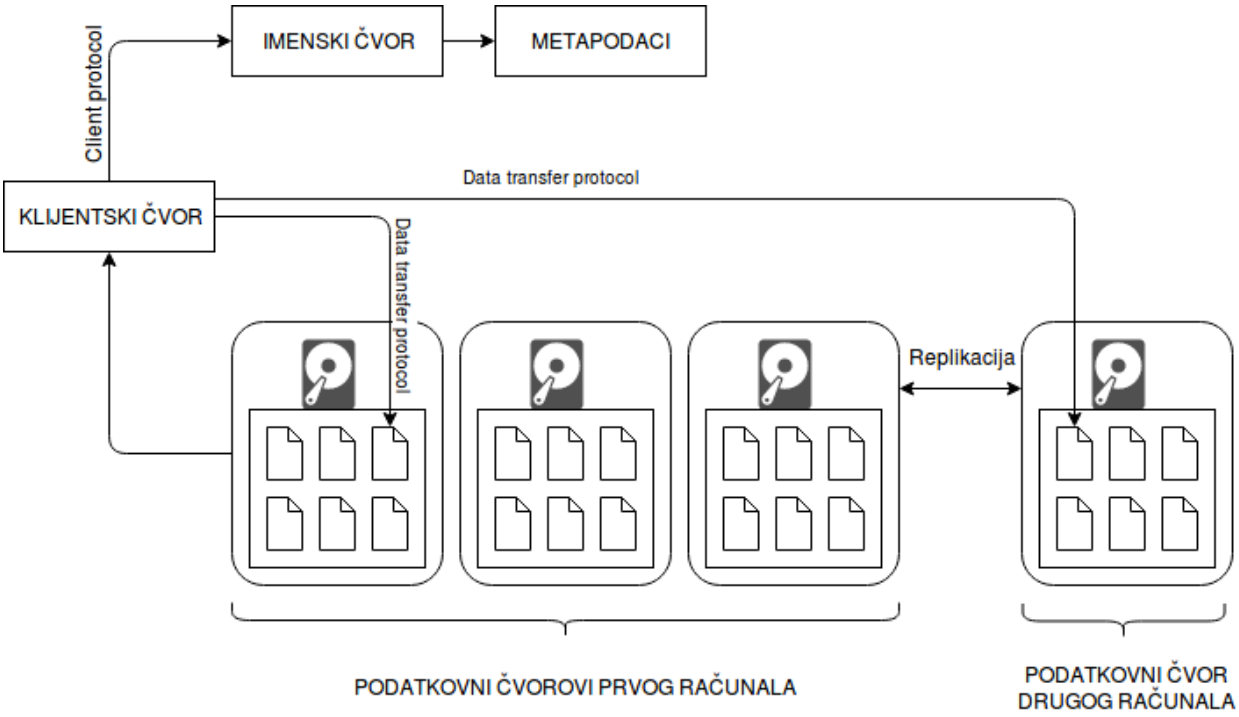


Slika 2.1. Podjela dijelova datoteke na više različitih čvorova u grozdu računala..

Kao što slika 2.1. prikazuje, datotečni sustav vrši i repliciranje pojedinih dijelova datoteke. Jedan dio nikada nije spremljen na samo jednom čvoru. Time se osigurava postojanost podataka i u slučaju da jedan čvor neočekivano ispadne iz rada.

2.2. Arhitektura Hadoop distribuiranog datotečnog sustava

Datotečni sustav Hadoop okvira baziran je na gospodar-rob (engl. *Master-Slave*) arhitekturi. U ovakvoj paradigmi jedan čvor je označen kao gospodar, znan kao imenski čvor (engl. *Name Node*). Svi drugi čvorovi ponašaju se kao robovi i imaju ulogu podatkovnih čvorova (engl. *Data Node*). Imenski čvor zadužen je za čuvanje metapodataka o datotekama spremljenima u datotečni sustav. Ovi podaci uključuju stavke kao što su ime datoteke, ograničenja pristupa, veličina datoteke u koliko blokova je datoteka rastavljena, ali i najvažnije, uključuje i lokaciju pojedinog bloka u grozdu. Također, imenski čvor zadužen je i za brojne druge operacije imenskog prostora kao što su otvaranje, zatvaranje te preimenovanje datoteka i direktorija.



Slika 2.2. Prikaz gospodar-rob arhitekture Hadoop raspodijeljenog datotečnog sustava

Podatkovni čvorovi zaduženi su za čitanje i pisanje podataka u datotečni sustav kada dobiju upute od imenskog čvora. Oni su odgovorni i za kreiranje replika blokova pojedine datoteke te zahtijevanje spremanja replike u druge podatkovne čvorove. Prilikom repliciranja podatkovni čvorovi brinu i o oštećenju podataka. Oni konstantno provjeravaju blokove koji su spremljeni na njih i provjeravaju jesu li podaci iskoristivi. Kompletna arhitektura datotečnog sustava prikazana je na slici 2.2.

Komunikacija sa ovakvim datotečnim sustavom odvija se u četiri koraka[5]:

- HDFS izlaže Java programsko sučelje koje programer može koristiti kako bi komunicirao sa datotečnim sustavom. Aplikacije koje koriste ovo sučelje upućuju zahtjeve klijentskoj biblioteci koja je prisutna na istom računalu na kojem se izvodi program.
- Klijentska biblioteka spaja se na imenski čvor koristeći *RPC* (engl. *Remote Procedure Call*). Komunikacija između klijenta i imenskog čvora odvija se preko klijentskog protokola (engl. *ClientProtocol*). On je zadužen za kreiranje datoteka, dodavanje podataka u datoteku, čitanje iz datoteke, završavanje pisanja u datoteku i drugih operacija nad imenskim prostorom.

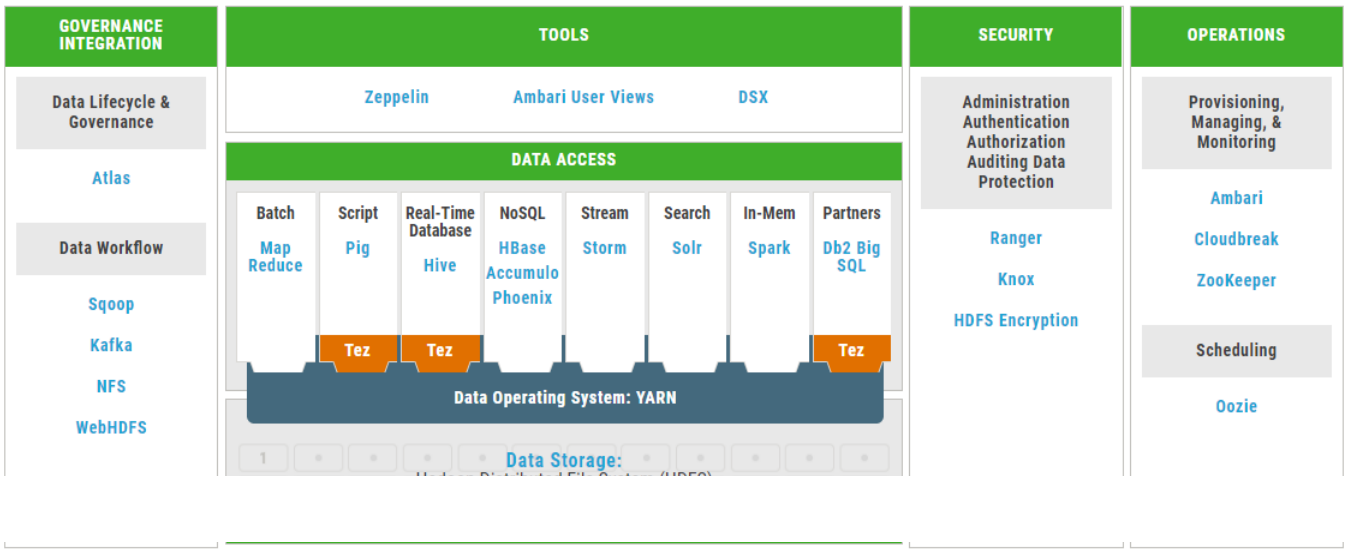
- Klijentska biblioteka spaja se na podatkovne čvorove direktno koristeći protokol razmijene podataka (engl. *DataTransferProtocol*). Ovaj protokol definira operacije kao što su pisanje u blok, čitanje iz bloka, provjere kontrolnog zbroja bloka i repliciranje bloka.
- U zadnjem koraku odvija se interakcija između podatkovnog čvora i imenskog čvora. Podatkovni čvor svaki puta započinje interakciju. Najčešće podatkovni čvor zahtijeva svoju registraciju u imenski čvor, šalje statusne podatke o blokovima, šalje notifikacije o pohrani repliciranih blokova sa drugih podatkovnih čvorova ili dobivanju bloka datoteke od klijenta.

Kao što se može zaključiti iz sučelja koje je otvoreno klijentima prema *HDFS* sustavu, *HDFS* je napisan koristeći Java programski jezik. Svaki računalni sustav koji podržava Javu može pokrenuti softver imenskog čvora ili podatkovnog čvora. Tipično je da u grozdu postoji jedno računalo na kojemu je pokrenut imenski čvor, dok sva ostala računala pokreću podatkovne čvorove. Preporučeno je da računala na kojima se izvršava ovaj softver koriste GNU/Linux kao operacijski sustav. Bitno je spomenuti kako ovo nije jedini način komuniciranja i upravljanja *HDFS* sustavom. Kao alternative pružaju se interaktivna korisnička sučelja kao što su *Ambari, Command-Line interface, HTTP/HDFS Proxies te NFS Gateway*.

2.3. Moduli Hadoop okvira

Hadoop se sastoji od tri glavna modula poznata kao *Hadoop Core*. Ovi moduli dio su projekta koji razvija Apache i uključene su u sve distribucije *Hadoop* okvira. Prvi modul je datotečni sustav koji je opisan u prethodnim poglavljima. Drugi modul je *YARN* (engl. *Yet Another Resource Negotiator*). *YARN* omogućuje zakazivanje poslova i upravljanje resursima dostupnima u grozdu računala. Treći modul projekta je *Hadoop MapReduce*. Ovo je implementacija *Map-Reduce* programske paradigme koja iskorištava *YARN* za optimizaciju poslova te služi za obradu i analiziranje velikih skupova podataka. Kao poseban modul izdvaja se *Hadoop Common*. On definira brojne korisne skripte koje podupiru i olakšavaju rad ostalih modula.

Postoje i mnogi drugi moduli koji su vezani za *Hadoop* sistem. Neki od njih razvijani su od strane Apache organizacije, no mnogi su razvijani i od strane drugih organizacija te imaju ulogu proširiti mogućnosti platforme, olakšati pristup podacima, poboljšati sigurnost platforme ili pak postojeće funkcionalnosti implementirati na drukčiji način te time pružiti alternativu korisnicima platforme.



Slika 2.3. Prikaz arhitekture Hadoop distrubucije bazirane na Hortonworks HD platformi

Bitno je za naglasiti kako tehnologijama prikazanima na slici 2.3[5]. nije nužno da se izvode unutar *Hadoop* okvira. Mnoge od njih mogu se izvršavati na jednom računalu te ne ovise o *YARN* upravitelju resursa.

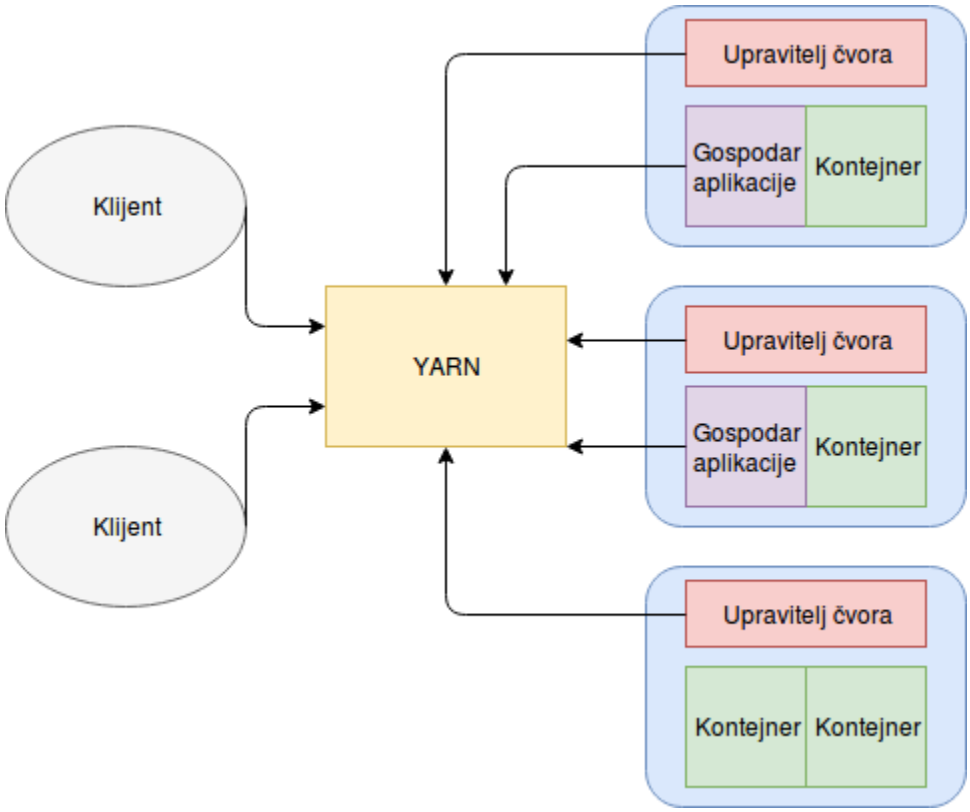
2.4. Yarn

U arhitekturi *Hadoop* sustava *YARN* se nalazi na sloju između *HDFS* i komponenata za pristup podatcima. *YARN* kombinira centralno upravljanje resursima sa implementacijama različitih modula za pristup podataka. Arhitektura ovog modula prikazana je na slici 2.4.

YARN modul definiran je kroz četiri glavne komponente:

- Globalni upravitelj resursima koji prima zahtjeve za odrađivanje poslova od klijenata te zakazuje poslove i dodjeljuje im resurse

- Upravitelj čvorom koji je instaliran na svaki čvor i služi za promatranje i objavljivanje statusa i događaja upravitelju resursa
- Gospodar aplikacije (engl. *Application Master*) koji je kreiran za svaku aplikaciju kako bi mogao pregovarati o dostupnim resursima i poslu koji se treba obaviti sa upraviteljem čvora. Upravitelj čvora na temelju ovih pregovora izvodi poslove.
- Kontejner resursa kojeg kontroliraju upravitelj čvora. Ovom kontejneru su dodijeljeni resursi koji su alocirani individualnoj aplikaciji.



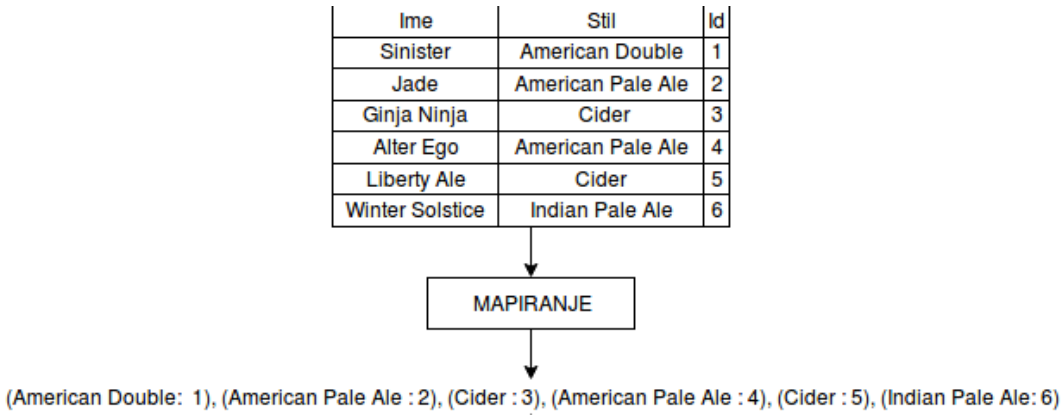
Slika 2.4. Uloga YARN upravitelja resursa u Hadoop okviru

2.5. Hadoop MapReduce

Hadoop MapReduce je programski okvir koji omogućuje laku obradu velike količine podataka. Obrada podataka unutar ovog okvira izvršava se paralelno na grozdu računala. Posao koji *MapReduce* treba obaviti definira se programski te se izvodi isključivo nad podatcima strukturiranim u parove. Jedan par čine ključ i pripadajuća vrijednost.

Potrebno je definirati razliku između programske paradigme znane kao *MapReduce* te programskog modela koji *Hadoop* koristi za obradu podataka. *Hadoop* koristi paradigmu *MapReduce* u svojem programskom modelu, no ove dvije stvari nisu istovjetne. Postoje i drugi programski modeli koji se također mogu izvršavati unutar Hadoop okvira koji također koriste *MapReduce* paradigmu. Apache *Spark* i Apache *Storm* koriste *MapReduce* paradigmu kao i *Hadoop MapReduce* u obradi velike količine podataka. U ovome poglavlju opisati će se osnove rada *MapReduce* programske paradigme. Konkretne implementacije i sučelje koje *Hadoop* pruža prema svojoj implementaciji ovakvog programskog modela biti će opisane u kasnijim poglavljima.

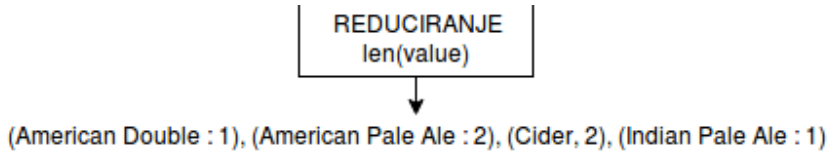
U osnovi *MapReduce* paradigma sastoji se od dvije glavne procedure. Prva procedura naziva se maper te je zadužena za sortiranje i filtriranje ulaznih podataka. Konačni rezultat ove metode su ulazni podaci razvrstani prema nekom ključu. Ključ može biti bilo koji podatak iz ulaznog skupa nad kojim podatke možemo agregirati. Izlaz se tako može opisati listom parova, gdje svaki par ima svoj ključ i listu vrijednosti koje mu pripadaju. Slika 2.5. prikazuje mapiranje ulaznih podataka.



Slika 2.5. Prikaz mapiranja podataka

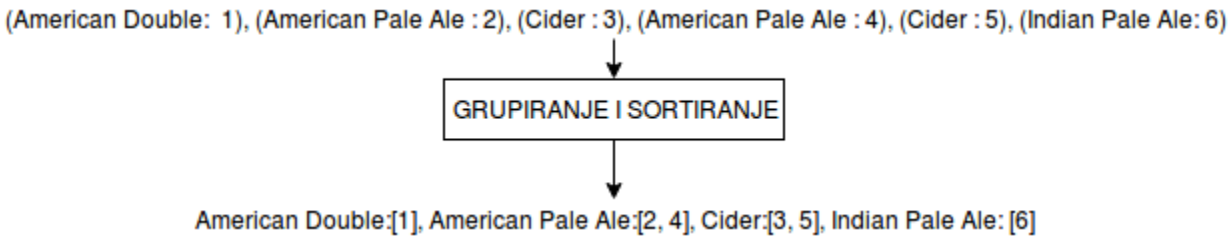
Procedura koja se izvršava nakon mapiranja naziva se reducer. Ona je zaslužna za obradu mapiranih podataka te agregiranja podataka. Izlaz ove metode može se opisati listom parova gdje se svaki par sastoji od ključa i vrijednosti koja mu pripada. Vrijednost se najčešće dobiva agregiranjem liste podataka koji su proizašli iz mapera. Postoji niz funkcija koje možemo upotrijebiti za agregiranje podataka no neke od najkorištenijih su prebrojavanje podataka, traženje srednje vrijednosti, traženje maksimalne i minimalne vrijednosti te traženje ukupne sume unutar

liste podataka. Sve ove funkcije kao konačni rezultat daju nekakvu diskretnu vrijednost tj. Jedan broj. Slika 2.6. prikazuje izgled reducer procedure.



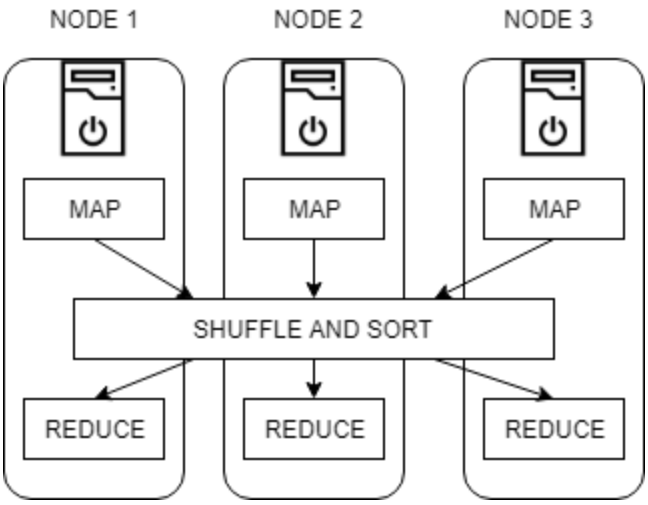
Slika 2.6. Reduciranje podataka funkcijom prebrojavanja

Postoji i treći korak koji se izvršava nakon mapiranja a prije reduciranja podataka, koji je u implementacijama ove paradigme najčešće skriven programeru, a naziva se miješanje i sortiranje podataka (engl. *Shuffle and Sort*). Ova procedura zadužena je sortiranje podataka prema ključu. Slika 2.7. prikzuje izgled procedure miješanja i sortiranja.



Slika 2.7. Grupiranje i sortiranje podataka

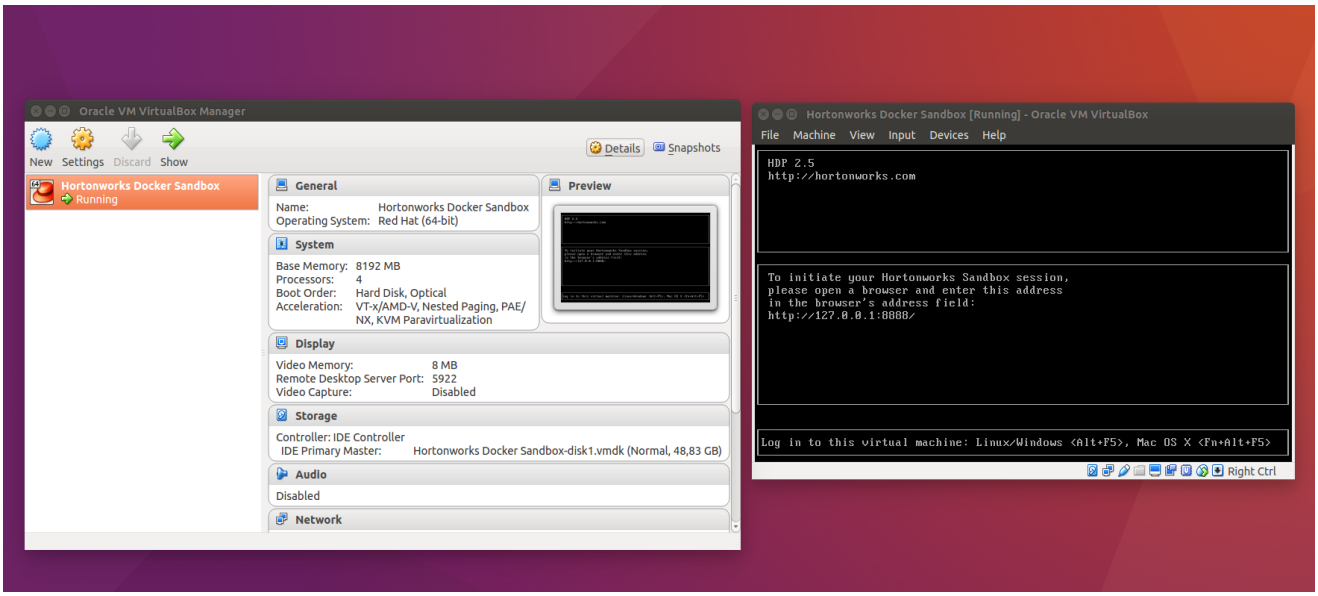
Sve ove procedure idealne su za izvođenje na grozdu računala jer svaki čvor unutar grozda može zasebno mapirati podatke. Procedura miješanja i sortiranja zatim može redistribuirati podatke tako da svi podaci vezani uz jedan ključ budu na jednom čvoru. Reduer procedure tada moraju obraditi samo podatke vezane uz ključeve koji se nalaze na čvoru gdje se procedura izvršava. Upravo ovo omogućuje obradu velikih količina podataka, ali i skaliranje dodavanjem novih čvorova u grozd. Slika 2.8. prikazuje ovu paradigmu u potpunosti.



Slika 2.8. Paralelno izvođenje MapRecude paradigme

2.6. Pokretanje Hadoop okvira

U ovome radu koristiti će se *Hortonworks* podatkovna platforma (engl. *Hortonworks Data Platform*), U daljnjem tekstu *HDP*, za pokretanje i uporabu Hadoop okvira[6]. *HDP* je dostupan kao slika virtualnog stroja (engl. *Virtual Machine*). Ova slika biti će pokrenuta koristeći *OracleVM* virtualizacijski alat. Slika je dostupna kao datoteka sa .vhd ekstenzijom koja je skraćenica za virtualni tvrdi disk (engl. *Virtual Hard Drive*). Ova slika služi kao predložak za kreiranje virtualnog stroja bazirane na *Fedora Core* distribuciji linuksa sa instaliranim *Hadoop* okvirom kao i brojnim drugim tehnologijama vezanim za Hadoop. Slika 2.9. Prikazuje izgled *OIracleVM* alata sa učitanom *HDP* slikom. Bitno je za naglasiti kako se u ovom slučaju programski kod *Hadoop* okvira izvršava unutar podatkovne platforme koja se izvršava unutar *HDP* slike.



Slika 2.9.. Pokretanje HDP slike unutar OracleVM alata

Na virtualni stroj moguće se spojiti korištenjem *SSH* (engl. *Secure Shell*) protokola. Ovaj protokol nam pruža zaštićeni kanal u nezaštićenoj mreži u klijent-server arhitekturi. U ovom slučaju klijent je sistem koji pokreče *OracleVM* alat, a server je virtualni stroj (u ovom slučaju *HDP* slika) koja je pokrenuta unutar alata. Slika 2.10. prikazuje korištenje *SSH* protokola za spajanje na virtualni stroj.

```
maria_dev@sandbox:~
alen@alen-desktop:~$ ssh maria_dev@127.0.0.1 -p 2222
maria_dev@127.0.0.1's password:
Last login: Sun Aug  5 10:37:32 2018 from 10.0.2.2
[maria_dev@sandbox ~]$
```

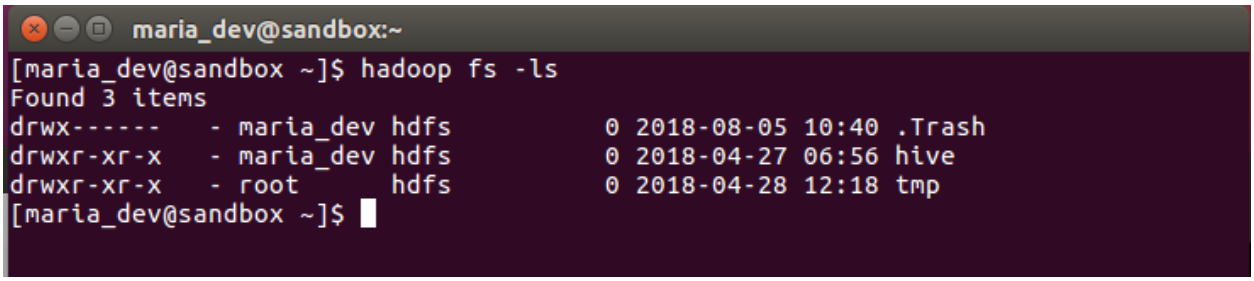
Slika 2.10.. Korištenje SSH protokola

HDP je konfiguriran da izvršava ulogu podatkovnog i imenskog čvora. Ovi čvorovi su programi koji se izvode u pozadini te pružaju sučelje prema Hadoop okviru. Kako bi se neka datoteka spremila unutar *Hadoop* programskog okvira potrebno je prvo kopirati datoteku na virtualni stroj. Slika 2.11.8 Prikazuje kopiranje datoteke na virtualni stroj koristeći *SCP* protokol.

```
alen@alen-desktop:~$ scp -P 2222 -r /home/alen/HDFS/Datasets/craft-cans maria_dev@127.0.0.1:
maria_dev@127.0.0.1's password:
breweries.csv                                 100%   23KB  22.5KB/s   00:00
beers.csv                                     100%  142KB 142.0KB/s   00:00
alen@alen-desktop:~$
```

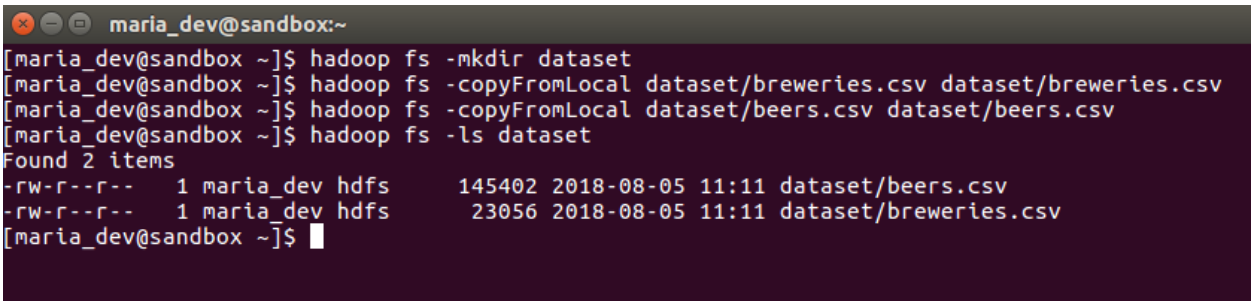
Slika 2.11. Kopiranje datoteke na virtualni stroj

Nakon što je određena datoteka kopirana na virtualni stroj, moguće ju je spremiti unutar *Hadoop* datotečnog sustava. Postoje mnogi načini za komuniciranje sa *Hadoop* okvirom navedeni u poglavlju 2.2. U ovome slučaju koristiti će se komandna linija za interakciju sa *Hadoop* okvirom. Sve naredbe koje su upućene *Hadoop* datotečnom sustavu moraju sadržavati prefiks *'hadoop fs'*. Nakon stavljanja ovog prefiksa *moguće* je koristiti većinu naredbi koje su podržane unutar linux operacijskog sustava za upravljanje i navigaciju datotečnim sustavom. Na slici 2.12 može se vidjeti korištenje *'ls'* naredbe kako bi se ispisao sadržaj direktorija koji se nalazi na *Hadoop* datotečnom sustavu.



Slika 2.12. Korištenje naredbe 'hadoop fs -ls'

Slika 2.13. Prikazuje kreiranje direktorija korištenjem naredbe *'mkdir'* te kopiranje datoteke na *Hadoop* datotečni sustav korištenjem naredbe *'copyFromLocal'*.

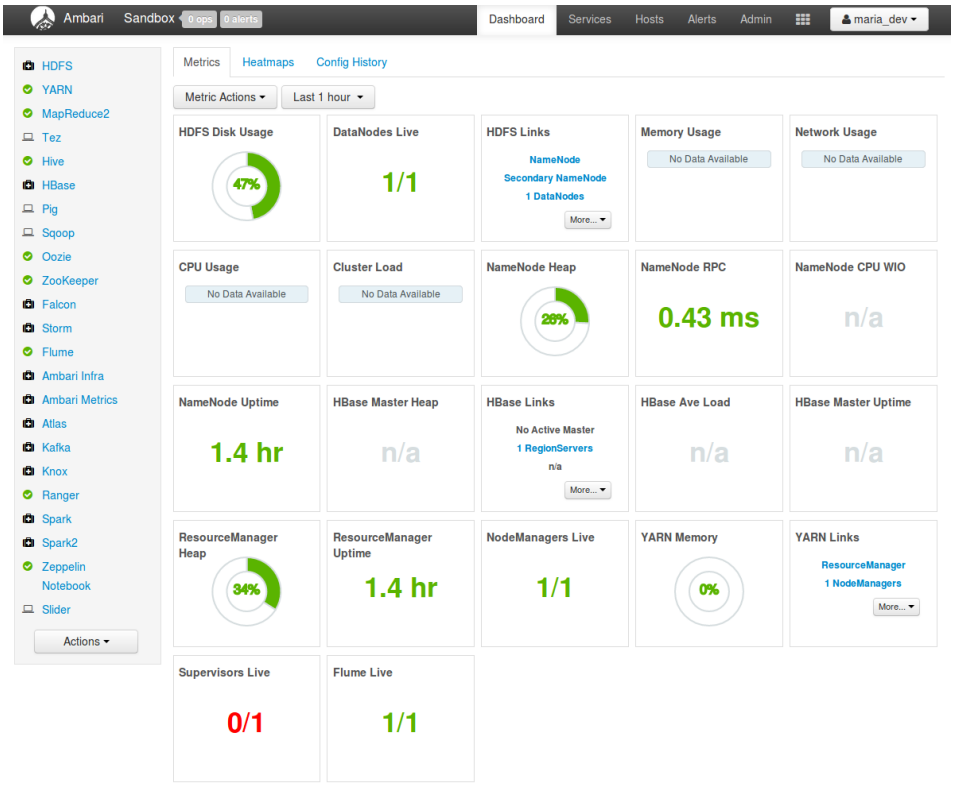


Slika 2.13. Korištenje naredbe 'copyFromLocal'

2.7. Upravljanje Hadoop okvirom koristeći Ambari korisničko sučelje

Upravljanje *Hadoop* datotečnim sustavom nije jedina stvar koja se može izvršiti preko komandne linije. Postoje i naredbe za konfiguriranje brojnih parametara te dobivanja raznih metrika. Međutim, korištenje komandne linije može biti sporo te zahtijeva pamćenje velikog broja naredbi.

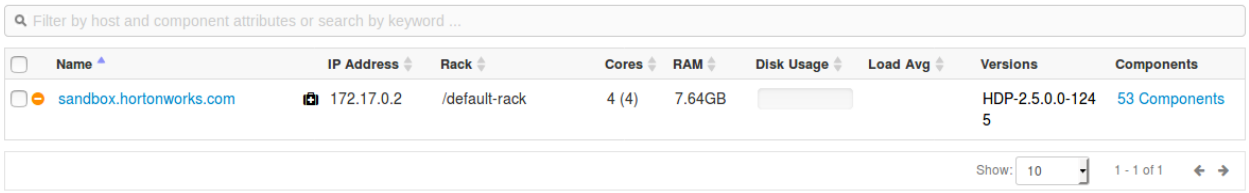
Tehnologije kao *Ambari* omogućuju upravljanje *Hadoop* okvirom korištenjem grafičkog sučelja. Grafičko sučelje znatno olakšava pregled statistika, kao i postavljanje određenih parametra. Uz sam *Hadoop* postoje i mnoge druge tehnologije koje se izvršavaju na čvoru na kojemu je pokrenut program podatkovnog ili imenskog čvora. Tehnologije kao što su *Pig*, *Hive*, *Spark* i druge, sve imaju svoje postavke i konfiguracijske datoteke koje se mogu uređivati. Unutar *Ambari* sučelja te tehnologije nazivaju se servisi te nam grafičko sučelje pruža lak način uređivanja konfiguracijskih datoteka pojedinog servisa, kao i praćenje stanja u kojem se određeni servis nalazi. Pregled datotečnog sustava također je omogućen. Slika 2.11. prikazuje izgled *Ambrai* korisničkog sučelja za *Hadoop* okvir pokrenut u prethodnom poglavlju.



Slika 2.14. Ambari korisničko sučelje

Na prethodnoj slici možemo vidjeti da *Ambari* prati metrike kao što su Iskorištenost diskovnog prostora, broj podatkovnih čvorova u grozdu, iskorištenost procesora, vrijeme koje je imenski čvor radio ispravno te brojne druge. S lijeve strane može se vidjeti i status određenih servisa koji su pokrenuti unutar grozda. *Ambari* pruža i detalje o svim čvorovima koji se nalaze u grozdu. Na slici 2.15. može se vidjeti kako u pokrenutom grozdu postoji jedan čvor nazvan *'sandbox.hortonworks.com'*, koji je dostupan na IP adresi 172.17.0.2, ima četiri procesorske jezgre

te 8GB dostupne memorije. Sve ove postavke moguće je konfigurirati prilikom pokretanja virtualnog stroja. U ovome slučaju pokrenut je samo jedan čvor, no lako je zamisliti kako ovo mogu biti korisni podaci u slučaju da se u grozdu nalazi puno čvorova.

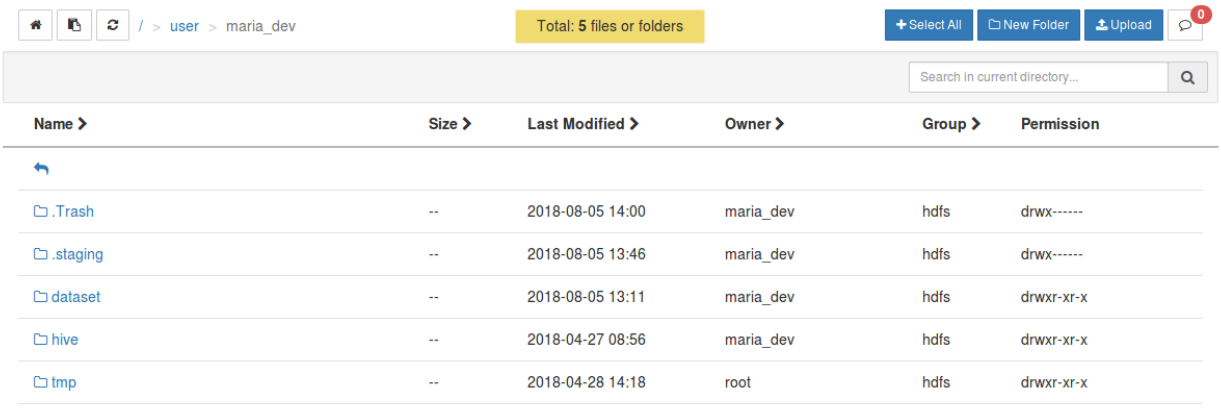


Slika 2.15. Detalji čvorova unutar grozda računala

Kako *Hadoop* okvir zahtijeva značajne resurse za pokretanje, *Ambari* pruža mogućnost konfiguriranja dostupnih resursa pojedinom podatkovnom čvoru. Ovo se izvršava mijenjanjem postavki YARN upravitelja resursa. Sve postavke mogu se promijeniti na željene veličine te time osigurati bolju iskorištenost resursa unutar podatkovnih čvorova.

Hadoop okvir nudi i mogućnost postavljanja specifičnih postavki za određeni čvor. U prvom redu ovo se odnosi na konfiguraciju sustava koji će dizati znakove uzbune kada određena metrika izađe van postavljenih granica. Ne dobivanje signala od podatkovnog čvora, popunjenost kapaciteta spremanja čvora ili ispad čvora iz rada samo su neki od znakova koji se mogu konfigurirati.

Ambari nudi i grafički prikaz datotečnog sustava. Datotečni sustav prikazan na slici 2.16. isti je onaj iz poglavlja 2.6. Uočava se kako je navigacija, kreiranje novih datoteka i prijenos datoteka na datotečni sustav znatno lakša korištenjem grafičkog sučelja.



Slika 2.16. Izgled HDFS unutar Ambari korisničkog sučelja

Postoje mnoga grafička sučelja za upravljanje grozdom računala, no *Ambari* se pokazuje kao sučelje sa mnogo dobrih odlika, koje pokriva širok raspon potreba korisnika.

3. OBRADA SERIJE PODATAKA UNUTAR HADOOP OKVIRA

Serijska obrada podataka odnosi se na proces obrade podataka koji su već spremljeni. Podaci nad kojima se odvija obrada spremljeni su u nekom vremenskom periodu. Ovaj period može definirati količinu podataka koji se trebaju obraditi, ali može i ograničiti obradu na samo dio podataka koji su pristigli u točno određenom vremenskom intervalu. Kako je u ovome radu riječ o velikim podatkovnim skupovima, obrada se često ne može izvršiti u realnom vremenu sa dostupnim resursima. Ipak, koristeći određene programske paradigme poput *MapReduce* i optimizacijom upravljanja čvorova i njihovih resursa unutar grozda računala mogu se postići znatna ubrzanja obrade podataka.

Bitno je napomenuti kako serijska obrada podataka ne može zamijeniti obradu toka podataka ako je brzina odziva sustava bitna metrika te ako se rezultati obrade očekuju u realnom vremenu. Serijska obrada može, s druge, strane pružiti uvid u veći volumen podataka, prikupljen u puno većem vremenskom periodu, i tako dati točnije i preciznije rezultate obrade. Upravo iz ovog razloga ovaj način obrade ostaje bitna komponenta svih sustava koji nastoje obraditi velike količine podataka u realnom vremenu.

3.1. Obrada podataka Hadoop MapReduce programskim okvirom

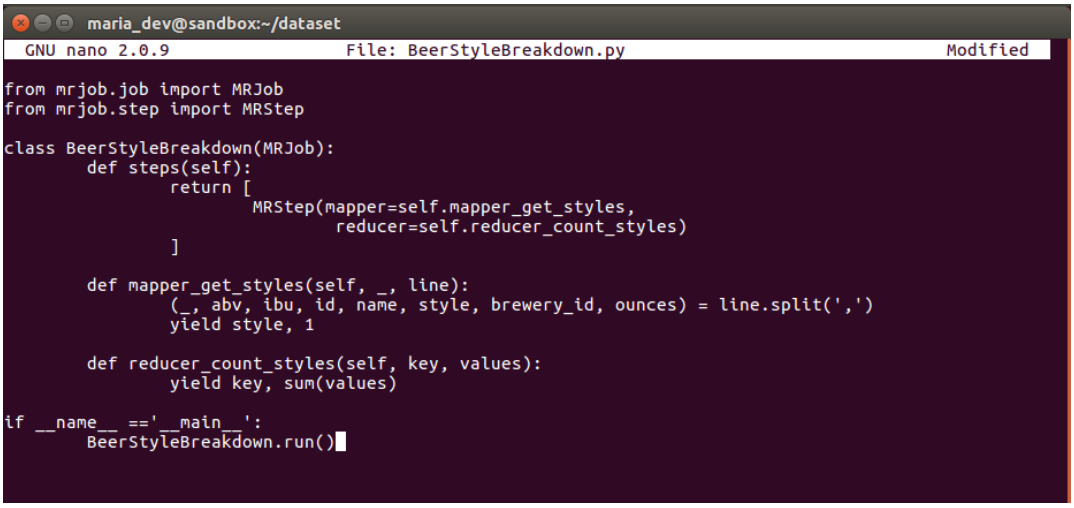
U poglavlju 2.5 opisana je *MapReduce* paradigma te je utvrđeno kako postoji *Hadoop* implementacija ove paradigme, koju *Hadoop* koristi kako bi serijski obradio velike količine podataka. U ovom poglavlju opisati će se korištenje *mrjob* biblioteke koja nam pruža sučelje prema *Hadoop MapReduce* implementaciji u python programskom jeziku.

Ključna karakteristika *mrjob* biblioteke je ta što je neovisna o platformi na kojoj se izvodi. Ovo znači da se ukoliko se programi napisani koristeći ovu biblioteku pokušaju izvesti na jednom računalu gdje nije dostupan *YARN* ili nekakav drugi upravitelj resursa, *mrjob* biblioteka će koristiti

lokalne resurse za obradu podataka. Ovo omogućuje lako testiranje i iteraciju verzija programskog koda prilikom razvijanja programske podrške. Ukoliko se program nastoji izvršiti u okruženju u kojem je *YARN* dostupan te na grozdu računala, *mrjob* biblioteka ga može uspješno iskoristiti kako bi izvela kod usporedno na više čvorova i time znatno ubrzala obradu podataka. Generalno pravilo je da se prilikom razvoja koristi manji podskup podatkovnog skupa koji se treba obraditi. Ovime se rasterećuju lokalni resursi računala na kojemu se programska podrška razvija te se dodaje određena doza sigurnosti u to da će programska podrška moći uspješno obraditi čitav skup podataka.

Kako bi se dobio detaljniji uvid u način na koji *mrjob* biblioteka omogućuje interakciju sa Hadoop *MapReduce* implementacijom, uz ovaj rad priložena je i aplikacija te podatkovni skup kojega ta aplikacija obrađuje. Čitav kod može se vidjeti na slici 3.1.

Kod je napisan u python programskom jeziku, koristeći verziju 2.6. U prve dvije linije koda dodaju se biblioteke o kojima daljnji dijelovi koda ovise. U ovome slučaju to su *MRJob* te *MRStep* klase iz *mrjob* biblioteke. Kako bi definirali *MapReduce* posao koji se treba obaviti, potrebno je definirati klasu koja nasljeđuje svoja svojstva od *MRJob* klase. U ovome primjeru definirana je klasa *BeerStyleBreakdown* unutar koje će se nadglasati implementacija metode *steps(self)* dostupne u roditeljskoj klasi. Metoda *steps* definira korake koji će se izvršavati u definiranom *MapReduce* poslu. Jedan korak mora sadržavati funkciju mapera i funkciju reducera te opcionalno može sadržavati vlastitu implementaciju funkcije za sortiranje i grupiranje rezultata. U primjeru koda, korak je definiran pomoću maper funkcije nazvane *mapper_get_styles* te reducer funkcije nazvane *reducer_count_style*. Bitno je naglasiti kako su klasi *MRStep* proslijeđene reference na ove dvije metode. Upravo ovo omogućuje definiranje vlastitih implementacija reducer te maper metoda. Iz programskog koda uočljivo je kako je izlaz *steps* metode polje koje sadrži jedan ili više objekata klase *MRStep*. Ukoliko izlaz funkcije sadrži više od jednog objekta, poslovi definirani tim objektom će se ulančati te će izlaz reducer metode jednog objekta, postati ulaz maper metode idućeg objekta.



Slika 3.1. Programski kod BeerStyleBreakdown skripte

Maper funkcija prima dva argumenta, prvi je izlaz reducer metode prethodnih *MapReduce* poslova. U primjeru koda, parametar je nazvan '_' te se time označava kako ne postoji prijašnji *MapReduce* posao te će taj parametar ostati prazan (Kao *'None'* konstanta unutar python programskog jezika definirana unutar ugrađenog imenskog prostora *'types.NoneType'*). Drugi parametar maper funkcije nazvan je *line* te sadrži jednu liniju ulaza *MapReduce* posla. Jedna linija ulaza može predstavljati jedan redak unutar baze podataka, jednu liniju tekstovne datoteke, ili pak jednu liniju *.csv* (*comma separated value*) ili *.tsv* (*tab separated value*) datoteke. Dani podatkovni skup formatiran je koristeći *csv* princip, gdje svaka linija datoteke predstavlja jedan zapis te se polja unutar tog zapisa odvajaju pomoću znaka zarez. U prvoj liniji metode ovaj format ulaza razdvaja se na temelju znaka zarez te se dobivene vrijednosti spremaju u zasebne varijable. Izlaz funkcije je varijabla *style* te vrijednost jedan. Ovakav izlaz omogućiti će nam da u sljedećem koraku grupiramo podatke prema vrijednosti varijable *style*. Očekivani ulaz u reducer metodu su parovi podataka, gdje jedan par sadrži jedinstveni ključ, u ovome slučaju jedinstvenu vrijednost *style* varijable te listu vrijednosti popunjenu sa konstantom jedan.

Reducer metoda kao ulaz prima ključ i vrijednosti koje se vežu za taj ključ. Implementacija funkcije sastoji se od vračanja vrijednosti ključa te agregiranja vrijednosti pridružene ključu koristeći funkciju *sum(values)*. Ovime se zbrajaju sve vrijednosti unutar liste, a kako su sve vrijednosti bile konstantne te su imale vrijednost jedan, ovaj podatak na kraju predstavlja dužinu liste. Ako pridružimo semantičko značenje ovim podacima, ključ predstavlja stil određenog piva, dok funkcija *sum(values)* vrača koliko puta se ovaj stil pojavio u podatkovnom skupu.

U primjeru koda može se uočiti kako *mrjob* biblioteka koristi asinkroni pristup u definiranju maper i reducer funkcija. Ove dvije funkcije u asinkronom pristupu pisanja koda znane su kao generatori. Generatori su zaduženi za asinkrono iteriranje podatka te omogućuju iteraciju podatkovnog skupa samo jedanput. Oni nemaju mogućnost spremanja vrijednosti u pričuvnu memoriju te s su stoga zaduženi za generiranje podataka. Pozivanjem funkcije maper ili reducer vrača se objekt koji predstavlja generator. Svaki korak iteracije nad ovim objektom vrača jednu vrijednost. Ova vrijednost unutar generatora vrača se koristeći *yield* ključnu riječ. Prednosti ovog pristupa su u mogućnosti oslobađanja resursa podatkovnog čvora na kojemu se obrađuju podaci koristeći *await* ključnu riječ. Ovim pristupom resursi podatkovnog čvora oslobađaju se za izvođenje drugih procesa, dok se čeka da generator generira idući podatak. U primjeru koda obrada koju generatori izvršavaju nije toliko komplicirana, no moguće je da izlaz generatora bude definiran složenom transformacijom podataka koja zahtjeva puno više vremena za obradu. Kada se ne bi koristio asinkroni pristup u definiranju i korištenju maper i reducer metoda, resursi podatkovnog čvora samo bi čekali bi iduće vrijednosti te tako trošili resurse na ne optimalan način. Korištenjem asinkronog pristupa, za vrijeme čekanja podatka, resursi se mogu iskoristiti za druge zadače dok se izvršava kompliciran transformacija ulaznih podataka.

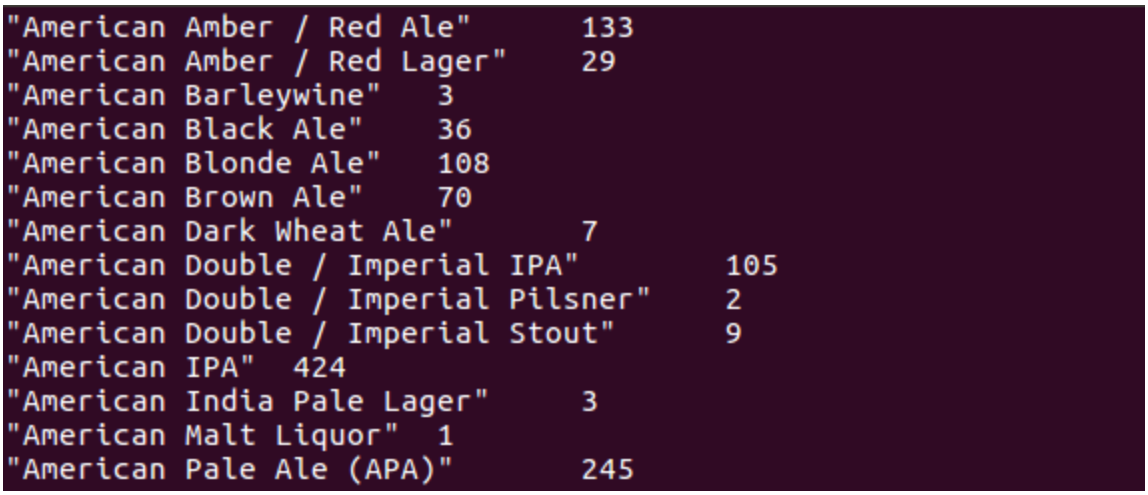
Pokretanje programskog koda iz primjena može se vidjeti na slici 3.2.

```
[maria_dev@sandbox dataset]$ python BeerStyleBreakdown.py -r hadoop --hadoop-streaming-jar /usr/hdp/c
urrent/hadoop-mapreduce-client/hadoop-streaming.jar beers.csv
No configs found; falling back on auto-configuration
Looking for hadoop binary in $PATH...
Found hadoop binary: /usr/bin/hadoop
Using Hadoop version 2.7.3.2.5.0.0
Creating temp directory /tmp/BeerStyleBreakdown.maria_dev.20180805.114622.811859
Copying local files to hdfs:///user/maria_dev/tmp/mrjob/BeerStyleBreakdown.maria_dev.20180805.114622.
811859/files/...
```

Slika 3.2. Pokretanje skripte koja koristi Hadoop MapReduce okvir

Potrebno je naglasiti kako ovaj program želimo izvršiti unutar *Hadoop* okvira te uz to predati datoteku koja sadrži podatkovni skup koji se obrađuje. U slučaju da se podatkovna datoteka ne nalazi na Hadoop datotečnom sustavu, *Hadoop MapReduce* će ju prije izvršavanja *MapReduce* posla tamo prebaciti.

Slika 3.3. Prikazuje rezultat izvršavanja koda.



Slika 3.3. Rezultati izvršavanja skripte

Moguće je uočiti kako su rezultati posloženi prema ključu abecedno. Ovo potvrđuje da se metoda sortiranja i grupiranja izvršila iako se nije eksplicitno definirala. Iz dobivenih rezultata možemo vidjeti kako je u podatkovnom skupu bilo najviše piva stila 'American IPA'.

Uz rezultate obrade, *Hadoop MapReduce* također ispisuje i statistike o poslu kojeg je odradio. Ove statistike uključuju generalne podatke o okolini u kojoj se *MapReduce* izvršio posao, broj poslova koji se odradio te podatke o neuspjelim koracima izvršavanja. Iz prikazanih statistika na slici 3.1.4. da se zaključiti kako je korak sortiranja i grupiranja zadužen za praćenje uspješnih maper i reducer koraka. Također, može se vidjeti i da *MapReduce* okvir bilježi podatke o veličini podatkovnog skupa koji je obrađivao te iskorištenosti resursa dostupnih podatkovnih čvorova.

```
  Job job_1533465132315_0001 completed successfully
  Output directory: hdfs:///user/maria_dev/tmp/mrjob/BeerStyleBreakdown.maria_dev.20180805.114622.811
859/output
Counters: 49
        File Input Format Counters
                Bytes Read=218103
        File Output Format Counters
                Bytes Written=2159
        File System Counters
                FILE: Number of bytes read=58820
                FILE: Number of bytes written=564291
                FILE: Number of large read operations=0
                FILE: Number of read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=218465
                HDFS: Number of bytes written=2159
                HDFS: Number of large read operations=0
                HDFS: Number of read operations=9
                HDFS: Number of write operations=2
        Job Counters
                Data-local map tasks=2
                Launched map tasks=2
                Launched reduce tasks=1
                Total megabyte-milliseconds taken by all map tasks=2035750
                Total megabyte-milliseconds taken by all reduce tasks=646250
                Total time spent by all map tasks (ms)=8143
                Total time spent by all maps in occupied slots (ms)=8143
                Total time spent by all reduce tasks (ms)=2585
                Total time spent by all reduces in occupied slots (ms)=2585
                Total vcore-milliseconds taken by all map tasks=8143
                Total vcore-milliseconds taken by all reduce tasks=2585
        Map-Reduce Framework
                CPU time spent (ms)=1950
                Combine input records=0
                Combine output records=0
                Failed Shuffles=0
                GC time elapsed (ms)=314
                Input split bytes=362
                Map input records=2411
                Map output bytes=53992
                Map output materialized bytes=58826
                Map output records=2411
                Merged Map outputs=2
                Physical memory (bytes) snapshot=512217088
                Reduce input groups=101
                Reduce input records=2411
                Reduce output records=101
                Reduce shuffle bytes=58826
                Shuffled Maps =2
                Spilled Records=4822
                Total committed heap usage (bytes)=265814016
                Virtual memory (bytes) snapshot=5826760704
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
```

Slika 3.4. Dobivene statistike prilikom izvršavanja skripte

3.2. Obrada serije podataka korištenjem Apache Pig okvira

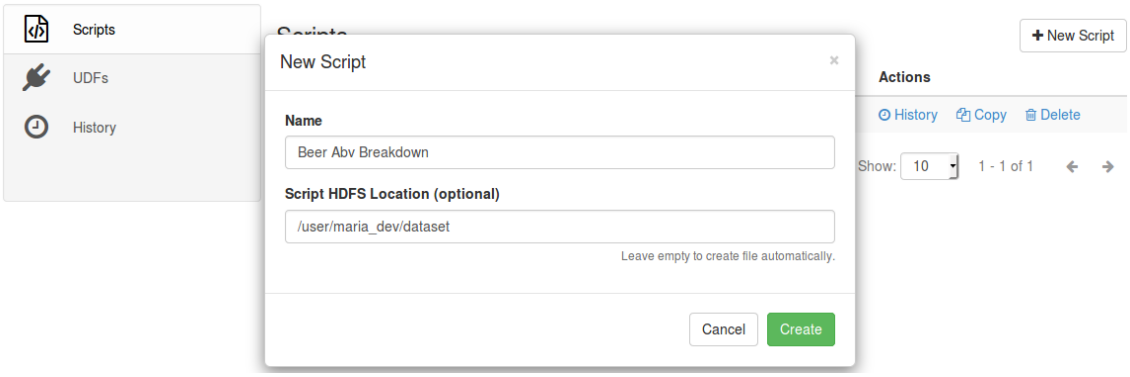
Apache *Pig* definiran je kao platforma za analiziranje i obradu velikih podatkovnih skupova. Ovaj okvir razlikuje se od *MapReduce* okvira u tome što jezik za obradu podataka koji *Pig* koristi sakriva brojne složene funkcije koje *MapReduce* definira te tako omogućava lakše i brže pisanje određenih *MapReduce* poslova.

Jezik koji *Pig* koristi za definiranje poslova zove se *Pig Latin*, koji je sličan SQL jeziku. Jednostavnost programiranja glavna je odlika *Pig* Latin jezika. Jednostavno je razviti programsku

podršku koja koristi paralelni pristup, ulančava *MapReduce* poslove te transformira podatke. Ovo dovodi i do lakšeg razumijevanja te održavanja postojeće programske podrške. *Pig Latin* također pruža i mnoge načine optimizacije. Ponekad je teško sve zadaće i transformacije podataka zamisliti i izraziti pomoću *MapReduce* paradigme. *Pig* Latin nastoji sakriti ovu kompleksnost i omogućiti programerima da pokušaju semantički izraziti način na koji žele transformirati podatke, bez zahtijevanja poznavanja detalja *MapReduce* paradigme.

Za razliku od *mrjob* biblioteke *Pig* je ovisan o *YARN* upravitelju resursa, no moguće je izvršiti poslove definirane *Pig Latin* skriptom i nad nekim drugim upraviteljem. *Tez* se nudi kao alternativa te u nekim slučajevima nudi značajno bolje vrijeme izvršavanja obrade podataka od *YARN* upravitelja resursa.

Ambari korisničko sučelje nudi razvojno okruženje za pisanje *Pig Latin* skripti te pokretanje *MapReduce* poslova koristeći *Pig* okvir. Kreiranje nove skripte prikazano je na slici 3.5.



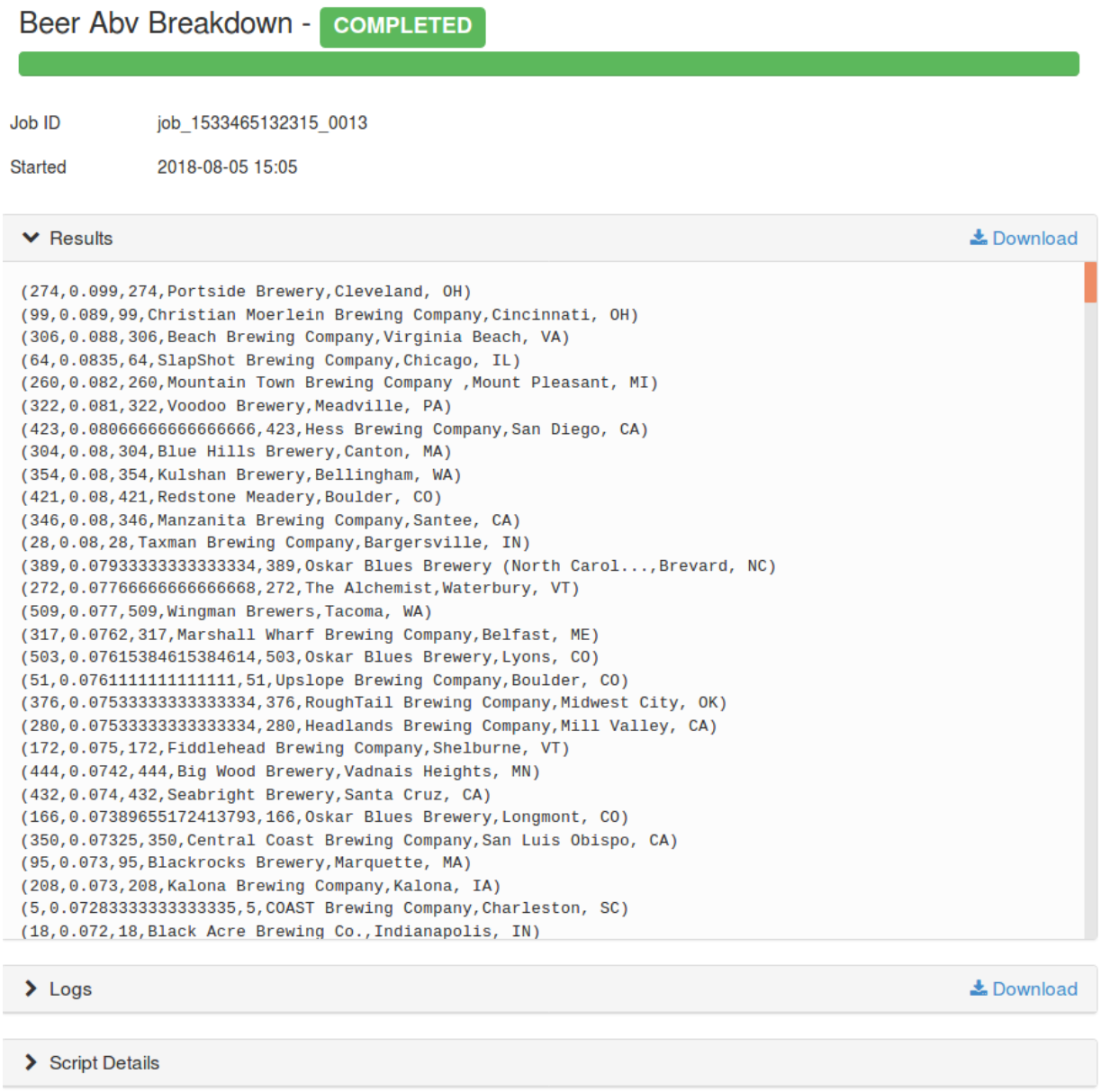
Slika 3.5. Kreacija Pig skripte unutar Ambari korisničkog sučelja

Skripti je moguće dodijeliti ime te mjesto gdje će biti spremljena unutar Hadoop datotečnog sustava. U primjeru skripte koja se nalazi na slici 3.6. prikazan je posao definiran *Pig Latin* jezikom, koji pronalazi pivovaru koja proizvodi piva sa najvećim prosječnim postotkom alkohola.

Slika 3.6. Pig latin skripta

Iz skripte *Beer Abv Breakdown* vidi se kako se u liniji 2 i 5 učitavaju podatkovni skupovi u *.csv* formatu. *PigStorage* funkcija omogućuje automatsko rastavljanje podatkovnog skupa u polja. U linijama 8 do 14 definiraju se transformacije nad ulaznim podacima. Očito je kako je naglasak na semantičkom izražavanju gdje su pojedine linije koda jasno čitljive. Naredba *'FOREACH dataset GENERATE group AS group_details*' tako prolazi kroz čitav podatkovni skup, nazvan *dataset*, i pravi parove ili grupe od podataka koji zadržavaju polja sadržana u *group_details* dijelu. Ovaj korak predstavlja mapiranje ulaznih podataka izražen na jednostavan i lako čitljiv način. Spajanje više različitih podatkovnih skupova nad nekakvim ključem jednako je lako. Upotrebom *JOIN* ključne riječi možemo definirati koji podatkovni skup želimo nadodati te nad kojim ključevima želimo podatke spojiti. Također, primjer pokazuje kako je podatke moguće filtrirati prema određenom kriteriju te poslagati prema određenom polju u uzlaznom ili silaznom smjeru.

Slika 3.7. Prikazuje ispis koji se dobije pokretanjem skripte iz primjera. Bitno je za naglasiti kako je označeno da se prethodni posao obavi upotrebom *TEZ* upravitelja resursa.



Slika 3.7. Rezultat pokretanja Pig latin skripte

Ispis uz rezultate obrade sadrži i interne zapise *Pig* okvira te dodatne detalje o pokrenutoj skripti. Ova polja sadrže brojne informacije koje mogu pomoći u otkrivanju problema prilikom izvršavanja *MapReduce* posla, ali i optimizacije pojedinih poslova.

3.3. Obrada serije podataka koristeći Apache Spark okvir

Apache *Spark* okvir je za obradu velike količine podataka. Ovaj okvir izgrađen je tako da bude otporan na greške te da iskoristi mogućnosti paralelne obrade podataka na više čvorova unutar grozda računala[7]. Modularnost ovog okvira omogućuje mu da se izvodi na grozdu računala gdje može iskoristiti dostupne resurse koristeći *YARN* upravitelj resursa, no postoje i drugi upravitelji koje *Spark* okvir može iskoristiti. Također, postoje i mnogi načini učitavanja podataka unutar *spark* okvira. Podaci koji se nalaze na *Hadoop* raspodijeljenom datotečnom sustavu mogu biti obrađeni koristeći *Spark* okvir, kao i podaci koji se nalaze unutar *Cassandra* i Apache *HBase* baza podataka. Sam *Spark* okvir sadrži mnoge module koji se grade nad *Spark common* modulom. Ovo uključuje *SparkSQL, Spark streaming, Spark MLib* te *GraphX* module.

Sučelje *Spark* okvira dostupno je unutar mnogih programskih jezika. *Spark* okvir izvorno je napisan u skala programskom jeziku no podrška za javu, python, R te SQL također postoji.

maria_dev@sandbox:~/dataset

GNU nano 2.0.9 File: LowestIbuToAbvBeerStyle.py

```
from pyspark import SparkConf, SparkContext
from itertools import islice

# Parse line of input to return ratio of bitternes over abv
def parseBeerInput(beerRddRow):
    columns = beerRddRow.split(',')
    # if ibu or abv is missing, substitute it with avereage value for column
    if not columns[2]:
        columns[2] = 42.713
    if not columns[1]:
        columns[1] = 0.055

    return (columns[5], (float(columns[2]) / float(columns[1]), 1.0))

if __name__ == "__main__":
    # Create spark config
    conf = SparkConf().setAppName("BeerTypeBitternesBreakdown")
    sc = SparkContext(conf = conf)

    # Load beer data and remove first line
    beerRdd = sc.textFile("hdfs:///user/maria_dev/dataset/beers.csv", use_unicode = False).map(lambda x: x.decode('utf-8'))

    beerRddNoHeader =  beerRdd.mapPartitionsWithIndex(
        lambda idx, it: islice(it, 1, None) if idx == 0 else it
    )

    # (beerStyle, ibu/abv)
    beerStylesBitternes = beerRddNoHeader.map(parseBeerInput)

    # Reduce to (beerStyle, (sumOfRatings, totalRatings))
    beerStylesTotalBiterness = beerStylesBitternes.reduceByKey(lambda beer1, beer2: ( beer1[0] + beer2[0], beer1[1] + beer2[1] ) )

    # Map to (rating, averageRating)
    averageBitternes = beerStylesTotalBiterness.mapValues(lambda totalAndCount : totalAndCount[0] / totalAndCount[1])

    # Sort by average rating
    sortedBitternes = averageBitternes.sortBy(lambda x: x[1])

    # Take the top 10 results
    results = sortedBitternes.take(10)

    # Print them out:
    for result in results:
        print(result[0], result[1])
```

Slika 3.8. programski kod LowestIbuToAbvBeerStyle skripte

Slika 3.8. prikazuje programski kod napisan koristeći *pyspark* biblioteku. *Pyspark* biblioteka omogućuje korištenje *Spark* okvira unutar python programskog jezika. Kao što je vidljivo iz programskog koda *Spark* okvir zahtijeva definiciju *SparkContext* objekta. Ovaj objekt zaslužan je za definiranje postavki okvira te pruža mogućnosti učitavanja podataka te obrade podataka korištenjem apstrakcije *RDD*. *RDD* (engl, *resilient distributed dataset*) je glavna apstrakcija podataka koju *Spark* okvir nudi. Ona predstavlja kolekciju elemenata raspodijeljenu u čvorovima računalnog grozda. Ovim elementima *Spark* okvir upravlja paralelno. U gornjem primjeru *RDD* je kreiran iz datoteke koja se nalazi na Hadoop raspodijeljenom datotečnom sustavu.

Spark okvir tretira *RDD* apstrakciju kao jedinstven skup podataka iako taj skup može biti raspodijeljen u grozdu računala. Ovo znači kako podatci ne moraju nužno biti na jednome mjestu kako bi ih *Spark* doživio kao jedinstvenu kolekciju. Prednost ovog pristupa je mogućnost korištenja operacija mapiranja, reduciranja i grupiranja i sortiranja nad čitavim skupom podataka.

```
maria_dev@sandbox:~/dataset
[maria_dev@sandbox dataset]$ nano LowestIbuToAbvBeerStyle.py
[maria_dev@sandbox dataset]$ spark-submit LowestIbuToAbvBeerStyle.py
Multiple versions of Spark are installed but SPARK_MAJOR_VERSION is not set
Spark1 will be picked by default
(u'Other', 333.33333333333331)
(u'Scotch Ale / Wee Heavy', 345.87704634223314)
(u'Flanders Oud Bruin', 365.07936507936506)
(u'Quadrupel (Quad)', 365.81271701388891)
(u'Wheat Ale', 400.0)
(u'Tripel', 416.1883627059982)
(u'American Malt Liquor', 431.44444444444446)
(u'Grisette', 446.42857142857144)
(u'Abbey Single Ale', 448.9795918367347)
(u'Belgian Strong Pale Ale', 451.3063788735713)
[maria_dev@sandbox dataset]$
```

Slika 3.9. pokretanje skripte koja koristi spark okvir

Slika 3.9. prikazuje pokretanje skripte napisane pomoću *pyspark* biblioteke unutar *Hadoop* okvira te dobivene rezultate pokretanjem programskog koda koji se nalazi na slici 3.3.1.

Korištenjem *Spark SQL* modula programski kod može se poboljšati tako da se podaci učitaju strukturirano. Također, moguće je i bolje upravljati vrijednostima koje nedostaju u podatkovnom skupu, tako da se za nepostojeće vrijednosti izračuna srednja vrijednost stupca. Ovo omogućuje *MLib* modul *Spark* okvira.

```
  GNU nano 2.0.9                                File: LowestIbuToAbvBeerStyleDf.py

from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions
from pyspark.sql.types import StructType, StructField, FloatType, StringType

def blankAsNull(x):
    return functions.when(functions.col(x) != "", functions.col(x)).otherwise(None)

def fillWithMean(df, include=set()):
    # Set null values for columns with empty string as value
    reduce(lambda d, x: d.withColumn(x, blankAsNull(x)), include, df)
    # Remove all missing values
    removeAllDf = df.na.drop()
    # Impute missing values for specific columns
    imputeDf = df
    for x in filter(lambda column: column in include, imputeDf.columns):
        meanValue = removeAllDf.agg(functions.avg(x)).first()[0]
        print(x, meanValue)
        imputeDf = imputeDf.na.fill(meanValue, [x])
    return imputeDf

if __name__ == "__main__":

    spark = SparkSession.builder.appName("BeerTypeBitternesBreakdown").getOrCreate()

    # Define beer data schema
    beerSchema = StructType([
    StructField("lineNo", FloatType(), True),
    StructField("abv", FloatType(), True),
    StructField("ibu", FloatType(), True),
    StructField("id", FloatType(), True),
    StructField("name", StringType(), True),
    StructField("style", StringType(), True),
    StructField("breweryId", FloatType(), True),
    StructField("ounces", FloatType(), True)])

    # Load beer data and remove first line
    beerDf = spark.read.schema(beerSchema).option("header", "true").csv("hdfs:///user/maria_dev/dataset/beers.csv")

    # Fill missing values with mean
    beerDfFilled = fillWithMean(beerDf, ["abv", "ibu"])

    # Find average ration for each style
    averageRatio = beerDfFilled\
                        .groupBy("style")\
                        .agg(functions.avg(beerDfFilled.ibu/beerDfFilled.abv).alias('avgRatio'))

    # Get top results
    topTen = averageRatio.orderBy("avgRatio").take(10)

    # Print them out, converting movie ID's to names as we go.
    for beer in topTen:
        print (beer.style.encode("utf-8"), beer.avgRatio)

    # Stop the session
    spark.stop()
```

Slika 3.10. Programski kod koji koristi Spark SQL i MLib module

Slika 3.10. prikazuje programski kod koji izvršava isti posao kao programski kod prikazan na slici 3.9. napisan koristeći *SQL* i *MLib* module *Spark* okvira. Glavna razlika je u načinu učitavanja podataka. Umjesto *SparkConfig* objekta definira se *SparkSession* objekt koji je zaslužan za učitavanje datoteke sa *Hadoop* raspodijeljenog računalnog sustava. *SQL* modul omogućuje definiranje sheme ulaznih podataka što dodatno olakšava rad sa učitanim podatkovnim skupom. Funkcija *FillWithMean(df, include=set())* zaslužna je za pozivanje funkcija *imputer* objekta koji se nalazi unutar *MLib* modula.

4. OBRADA TOKA PODATAKA

Obrada toka podataka uključuje obradu, transformaciju i ekstrakciju bitnih podataka sa izvora podataka. Ovaj način obrade odvija se kontinuirano te omogućuje dobivanje relevantnih rezultata u kratkom vremenskom periodu. Ovdje je bitno razlikovati dvije vrijednosti. Prva definira kada se ulazni podatak pojavio na izvoru te se naziva vrijeme događaja. Ova vremenska oznaka predstavlja trenutak kada podatak postaje dostupan za obradu te ga aplikacije koje se bave obradom podataka mogu konzumirati. Drugo bitno vrijeme je vrijeme obrade. Ono predstavlja vrijeme koje je potrebno kako bi podatak bio učitan u tok podataka, i primljen od strane aplikacija koje ga dalje mogu obraditi ili transformirati.

Ovaj način obrade podataka omogućuje brzo i efikasno dobivanje bitnih rezultata, no pojavljuju se i očite mane ovakvog pristupa. Nedostatak ovog pristupa je taj što svi podaci koji se čitaju sa nekakvog izvora moraju biti učitani u memoriju. Ova činjenica značajno smanjuje vremenski period vremena događaja koji se efektivno može obraditi u sustavu. Kako podaci u sustavima koji rade nad velikim skupovima podataka pristižu brzo i u velikim količinama, pričuvna memorija u koju se učitavaju podaci sa izvora popunjava se velikom brzinom. Da bi se zahtjevi za obradom u realnom vremenu ispunili često je potrebno odbaciti starije podatke čime se i značajno smanjuje točnost i relevantnost informacija dobivenih obradom toka. Također, ulazni tok podataka može značajno promijeniti intenzitet slanja ulaznih podatka i njihovu strukturu. Ovo može dovesti do nemogućnosti sustava da obradi novonastalu, veću količinu podataka, ili pak nemogućnosti razumijevanja promijenjene strukture podataka. Uz sve ove mane obrada toka podataka nezamjenjiv je dio svih arhitektura sustava koji obrađuju velike količine podataka jer uspijeva davati relevantne informacije sa malim vremenskom odzivom.

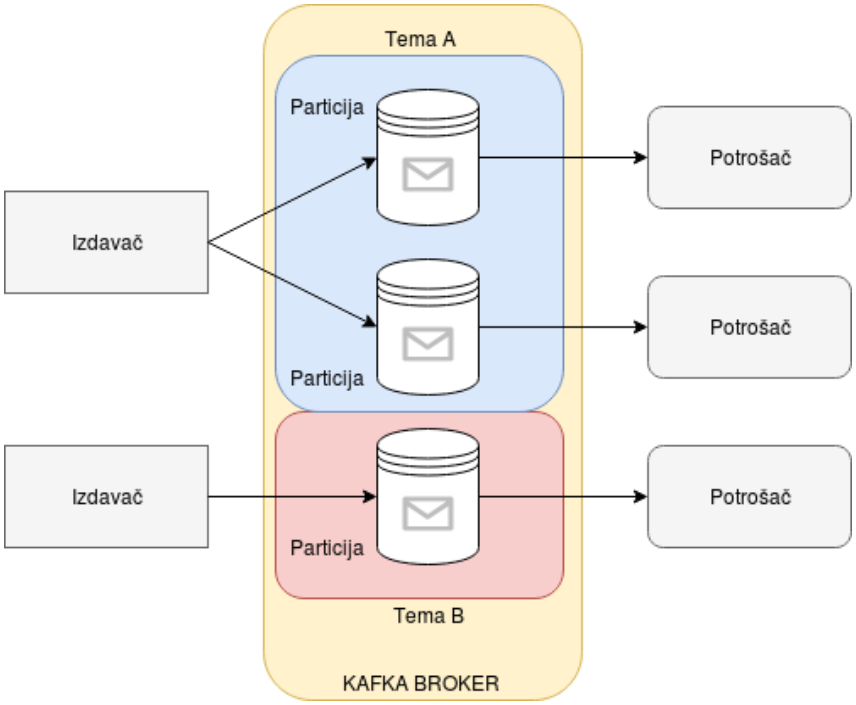
4.1. Kreiranje toka podatka koristeći Apache Kafka okvir

Apache *Kafka* okvir je temeljen na obradi transakcijskih zapisa[8]. Transakcijski zapis je sortirana podatkovna struktura koja podržava samo dodavanje na kraj. Zapisi koji su jednom dodani u ovu listu ne mogu se više uređivati, niti brisati ni na koji način. Iskorištavanjem ove strukture Apache *Kafka* dobiva značajne prednosti. Čitanje i pisanje odvija se u konstantnom vremenu te čitanje

podataka ne utječe na pisanje podataka. Ovime se postiže lako distribuiranje liste u grozdu računala te paralelna obrada više podataka od jednom.

Sučelje koje je otvoreno prema korisnicima Apache *Kafka* okvira dozvoljava slanje poruka u čvor u kojem se izvršava program *Kafka* čvora. Ovaj čin zove se objavljivanje poruke te se aplikacija koja objavljuje poruku zove izdavač. Poruke se objavljuju na određenu temu unutar *Kafka* čvora. Tema predstavlja skup poruka koje nose informacije o sličnom domenskom problemu. Kada se poruka objavi *Kafka* okvir sprema ju u podatkovnu strukturu transakcijskih zapisa i time garantira da su sve poruke spremljene onim redom kojim su i objavljene. Porukama se zato može pristupiti definiranjem odstupa od trenutne poruke, što zapravo predstavlja indeks polja poruka, gdje trenutna poruka ima indeks postavljen na vrijednost broja poruka unutar liste. Kako se u ovom radu govori o velikim količinama podataka, lako je za pretpostaviti kako se veličina ove strukture može značajno povećati u malom vremenskom periodu. *Kafka* okvir omogućuje razdvajanje poruka jedne teme u particije manje veličine. Time se dodatno osigurava bolje izvođenje i veća skalabilnost sustava. Particije mogu biti razdvojene prema proizvoljnom kriteriju, no najčešće se razdvajaju prema karakteristikama poruka koje su vezane za temu. Također, svaka poruka ima određen vremenski period tokom kojeg se može pročitati. Nakon isteka ovog vremenskog perioda, poruka se briše i postaje nedostupna.

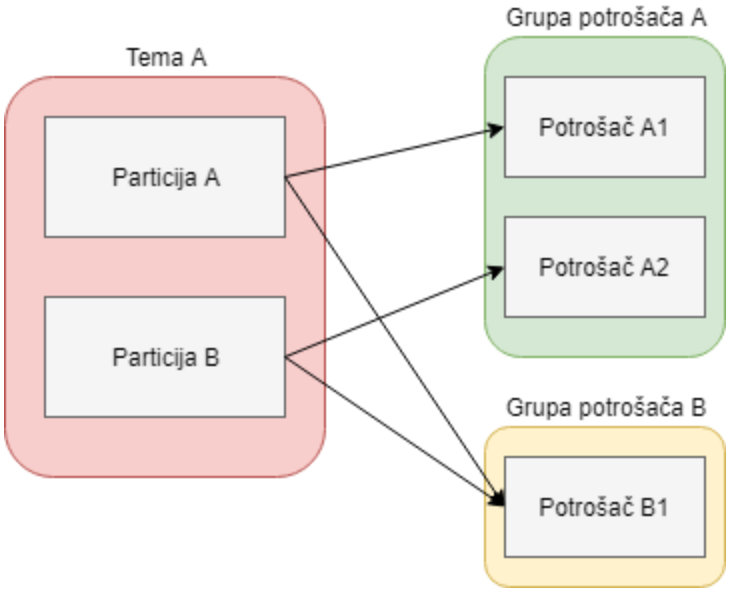
S druge strane postoje potrošači. Potrošači određuju koje poruke žele primiti definiranjem teme na koju se žele pretplatiti te definiranjem ofseta od trenutne poruke. Sve poruke koje su objavljene na temu na koju je potrošač pretplaćen i koje se nalaze unutar definiranog ofseta biti će dostavljene potrošaču. Potrošači dobivaju poruke sa svih particija definiranih u određenoj temi.



Slika 4.1. Arhitektura Apache Kafka okvira

Slika 4.1. prikazuje arhitekturu sustava Apache Kafka okvira

Lako je za pretpostaviti kako se na pojedine teme može objaviti više poruka nego na ostale. *Kafka* okvir omogućuje kreiranje grupa potrošača kako bi rasteretio rad potrošača koji su zaduženi za obradu objavljenih poruka. U jednoj grupi potrošača može se nalaziti jedan ili više potrošača. Grupa može definirati koje poruke želi primiti na jednak način na koji to može učiniti i pojedinačni potrošač. Razlika je u dostavljanju poruka, točnije u tome da svaki potrošač unutar grupe dobiva poruke sa određenog podskupa particija te teme, a ne sa čitavog skupa particija teme. Ovo znači da ukoliko postoje dvije particije teme i dva potrošača u grupi koja je pretplaćena na temu, svaki potrošač će dobivati poruke sa jedne particije. Slika 4.2. pokazuje ovaj princip detaljnije.



Slika 4.2. Pretplata potrošača na određenu temu

Iz definicije i principa rada Apache *Kafaka* okvira može se zaključiti kako je on primjer okvira koji koristi paradigmu izdavača i pretplatnika.

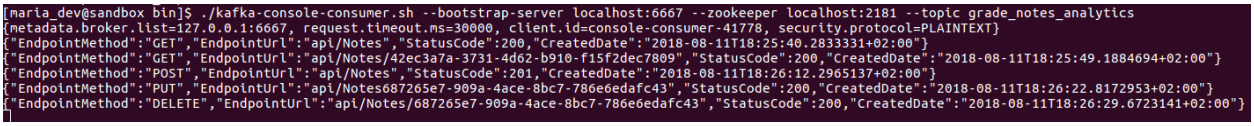
Apache *Kafka* okvir koristi servis *Zookeeper* kako bi držao podatke o izdavačima i potrošačima, postojećim temama, particijama određene teme te porukama koje se čitaju i pišu u određenu temu. *Zookeeper* servis je distribuirana baza podataka sposobna za čitanje i pisanje podatkovnih struktura koje se sastoje od ključa i vrijednosti. *Zookeeper* je još jedan primjer okvira koji koristi paradigmu izvođača i pretplatnika. Pretplatnici na određene teme *zookeper* servisa dobivaju obavijesti kada se dogodi promjena u bazi. Apache *Kafaka* uvelike ovisi o ovom servisu te ga koristi za spremanje metapodataka navedenih ranije.

Apache *Kafka* okvir dostupan je i na *HDP* platformi. Kreiranje nove teme moguće je izvođenjem naredbe prikazane na slici 4.3. Može se uočiti kako je potrebno specificirati *Zookeper* instancu koja je zadužena za spremanje metapodataka o temi te *Kafka* broker koji je zadužen za upravljanje temom.

```
[maria_dev@sandbox bin]$ ./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test_topic
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.
Created topic "test_topic".
[maria_dev@sandbox bin]$
```

Slika 4.3. Stvaranje teme unutar Apache Kafka okvira

Pretplata na određenu temu također je moguća. Slika 4.4. prikazuje pretplatu na temu. Pretplaćivanjem na temu stvara se ulazni tok podataka. Svi potrošači koji dobivaju podatke sa određenog toka, mogu samostalno obrađivati, transformirati i ekstraktirati bitne informacije iz poruka koje dobivaju u ulaznom toku podataka.



Slika 4.4. Pretplata na temu

4.2. Obrada toka podataka koristeći spark streaming modul spark okvira

Spark sadrži modul za obradu toka podataka, koji na jednostavan i efektivan način može obraditi veliku količinu ulaznih podataka. Postoje dvije verzije ovog modula a u ovom radu koristiti će se novija verzija znana kao *Spark* strukturirana obrada toka podataka. Ovaj modul izgrađen je na *Spark SQL* modulu te omogućava korisnicima da izraze operacije nad tokom podataka na jednak način kao da obrađuju seriju podataka. Ovo dovodi do značajnog olakšanja u korištenju *Spark* okvira i ubrzanju pisanja programske podrške.

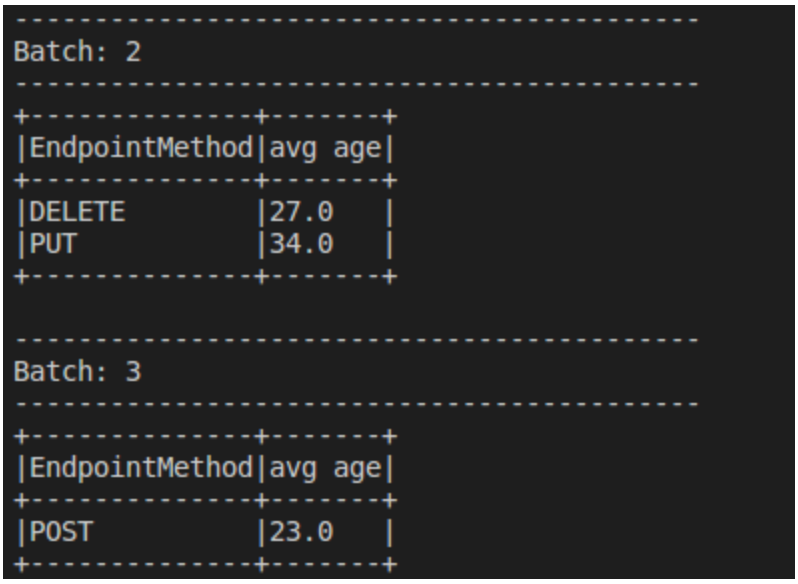
Spark okvir u svojem modulu strukturirane obrade toka podataka nudi dvije bitne paradigme oko kojih je okvir izgrađen. Prva je sam ulazni tok podataka. On može biti definiran na razne načine, neki od kojih uključuju čitanje podataka iz datoteke koja se nalazi na *Hadoop* raspodijeljenom datotečnom sustavu ili se pak ulazni tok može predstaviti kao niz poruka koje se dobiju pretplatom na određenu temu Apache *Kafka* okvira. Ovaj ulazni tok može se transformirati na razne načine koristeći operacije identične onima za obradu serije podataka. Definicija ulaznog toka podataka definiranog sa Apache *Kafka* temom može se vidjeti na slici 4.2.1.

```
// Define kafka stream
val analyticsStream = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "localhost:6667")
  .option("subscribe", "grade_notes_analytics")
  .load()
  .selectExpr("CAST(value AS STRING)", "CAST(timestamp AS TIMESTAMP)").as[(String, Timestamp)]
  .select(from_json($"value", endpointSchema).as("data"), $"timestamp")
  .select("data.*", "timestamp")
  .withWatermark("timestamp", "10 seconds")
  .groupBy("EndpointMethod")
  .agg(avg("Age").alias("avg_age"))
```

Slika 4.5. Definicija ulaznog toka podataka

U programskom kodu na slici 4.5. vidi se i obrada ulaznog toka podataka. U prvome redu ekstraktiraju se podaci same poruke te vremenska oznaka koja predstavlja vrijeme događaja poruke. Ovdje je vrijeme događaja definirano kao vrijeme kada je Apache *Kafka* okvir objavio poruku svim pretplaćenim potrošačima. Nadalje, definiranjem ulazne sheme podataka omogućuje se strukturirano čitanje toka. Ovaj pristup značajno smanjuje probleme koji se mogu pojaviti promjenom strukture poruka koje se objavljuju na određenu temu te koje se zatim propagiraju do potrošača. Definiranjem sheme podataka, samo oni podaci koji su u shemi će biti učitani. Dodatno, moguće je obaviti i validaciju podataka unutar sheme te odbaciti podatke koji se odgovaraju postavljenim kriterijima prije same obrade, značajno smanjujući količinu podataka koja prolazi kroz sustav i koju je potrebno obraditi. Ulazni tok zatim se grupiraju nad određenim podatkom, u ovom slučaju poljem označenim sa imenom *EndpointMethod* te se izračunava srednja vrijednost polja *Age*.

Druga paradigma koju *Spark* okvir pruža je koncept podatkovnog ponora (engl. *Data sink*). Podatkovni ponor pruža mogućnost ispisa ili izvoza transformiranog ulaznog toka podataka. *Spark* okvir podržava ispis transformiranog toka unutar komandne linije, izvoz podataka u smislu objave na određenu temu Apache *Kafka* okvira ili dodavanje podataka u određenu datoteku. Slika 4.6. prikazuje ispis podatkovnog toka unutar komandne linije.



Slika 4.6. Ispis podatkovnog toka unutar komandne linije

Spark okvir nudi i mogućnost određivanja vremenskog perioda objave podataka. U ovome slučaju podatci transformirani podatci objavljivati će se na temu prema definiranom vremenskom periodu. Potrebno je naglasiti kako će *Spark* okvir držati sve podatke koji pristignu na ulazni tok u memoriji unutar definiranog perioda. Prilikom objave, pričuvna memorija se prazni te se počinju prikupljati i transformirati novi podaci. Podatkovni ponor također može definirati jedan od tri načina rada. Način ispisa definira koji podatci se ispisuju u izlaznoj tablici. Načini ispisa prikazani su u tablici 4.1.

Tablica 4.1. Načini ispisa podatkovnog toka

Način rada	Objašnjenje
Append	Samo oni podatci koji su se dodali u tablicu nakon zadnjeg ispisa se ispisuju.
Complete	Kompletna tablica ispisuje se svaki puta kada se rezultat ispisuje
Update	Samo redovi koji su promijenjeni od zadnjeg ispisa se ispisuju.

Ispis unutar komandne linije ili ispis u memoriju računala može biti koristan prilikom razvoja programske podrške.

4.3. Korištenje operacija vremenskog prozora nad vremenom događaja

Glavna stavka gotovo svih sustava koji se bave obradom toka podataka je izračunavanje određenih metrika unutar određenog vremenskog okvira. Ovi agregacijski podaci služe kao osnova gotovo svih sustava te se upotrebljavaju u brojnim statistikama, sučeljima za promatranje rada sistema te poslovnom odlučivanju. No spremanje svih podataka sa ulaznog toka u pričuvnu memoriju je nemoguće. *Spark* okvir nudi mogućnost rada sa prozorima koji definiraju vremenski okvir unutar kojeg se podatci sa ulaznog toka čuvaju i nad kojima se izvode transformacije definirane ulaznim tokom.

Spark okvir nudi mogućnost upravljanja podacima koji pristižu u ulazni tok automatskim upravljanjem zakašnjelih podataka. Zakašnjelim podatkom smatra se svaki podatak čije je vrijeme obrade, veće od vrijeme događaja. Ovo znači da ukoliko se nekakva poruka objavila u vremenu *T*, a poruka dolazi na ulazi tok i postaje spremna za obradu u vremenu *t*, vrijeme kašnjenja poruke može se definirati kao razlika ova dva vremena. *Spark* okvir sposoban je uočiti ovo te u pozadini poruku obraditi unutar vremenskog perioda kojemu poruka pripada gledajući vrijeme događaja.

Druga mogućnost upravljanja podacima unutar ulaznog toka jest dodavanje vodenog žiga poruke. Označavanjem vremena događaja svake poruke *Spark* okvir može pokušati odbaciti poruke koje imaju vrijeme događaja starije od konfiguriranog vremena iz pričuvne memorije i ažurirati vrijednosti agregacijskih polja

Zadnja mogućnost upravljanja ulaznim tokom podataka koju *Spark* okvir nudi su operacije nad vremenskim prozorom. Operacija nad vremenskim prozorom definirana je sa dva parametra. Prvi parametar predstavlja vrijeme koje definira koji vremenski period vremena događaja će se uzeti u obzir prilikom izračuna agregacijskih polja. Drugi parametar je vrijeme koje definira koliko često će se izvršavati izračun agregacijskih polja. Drugim riječima ukoliko navedene konstante imaju vrijednosti deset i pet, vremenski prozor pomicati će se svakih pet sekundi te će prilikom transformacije ulaznih podataka uzimati u obzir samo one podatke koji imaju vrijeme kašnjenja

manje od 10 sekundi. Slika 4.7. prikazuje definiciju ulaznog toka nad kojim se izvršava operacija unutar vremenskog prozora.

```
// Define kafka stream
val analyticsStream = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "localhost:6667")
  .option("subscribe", "grade_notes_analytics")
  .load()
  .selectExpr("CAST(value AS STRING)", "CAST(timestamp AS TIMESTAMP)").as[(String, Timestamp)]
  .select(from_json($"value", endpointSchema).as("data"), $"timestamp")
  .select("data.*", "timestamp")
  .withWatermark("timestamp", "10 seconds")
  .groupBy(
    window($"timestamp", "10 seconds", "5 seconds"),
    $"EndpointMethod"
  )
  .agg(avg("Age").alias("avg_age"))
```

Slika 4.7. Korištenje operacija vremenskog prozora

Slika 4.8. prikazuje izlaz toka podataka koji je obrađen pomoću vremenskih prozora. Može se uočiti kako je vremenski prozor definiran sa dvije vremenske oznake, početak vremenskog prozora te njegov kraj.

```
-------------------------------------------
+------------------------------------------+--------------+-------+
|window                                    |EndpointMethod|avg age|
+------------------------------------------+--------------+-------+
|[2018-08-15 11:09:00, 2018-08-15 11:09:01]|PUT           |56.0   |
|[2018-08-15 11:09:01, 2018-08-15 11:09:02]|PUT           |70.0   |
+------------------------------------------+--------------+-------+


-------------------------------------------
Batch: 13
-------------------------------------------
+------------------------------------------+--------------+-------+
|window                                    |EndpointMethod|avg age|
+------------------------------------------+--------------+-------+
|[2018-08-15 11:09:03, 2018-08-15 11:09:04]|POST          |44.5   |
|[2018-08-15 11:09:04, 2018-08-15 11:09:05]|DELETE        |65.0   |
+------------------------------------------+--------------+-------+
```

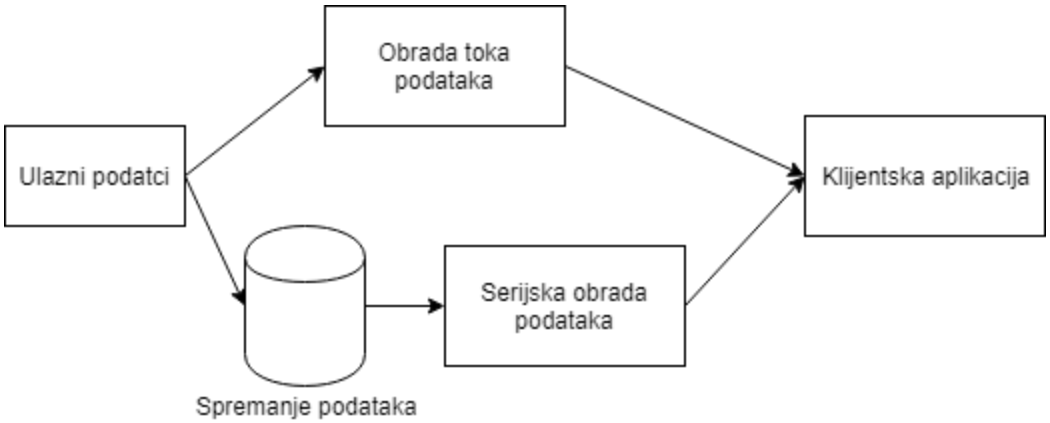
Slika 4.9. Ispis podatkovnog toka

5. USPOSTAVA LAMBDA ARHITEKTURE

Lambda arhitektura jedna je od arhitektura sistema koji obrađuju velike količine podataka a imaju zahtjeve za realnim vremenom. Sastoji se od dvije odvojene grane koje obrađuju iste ulazne podatke na dva različita načina[9]. Prva je grana koja podatke obrađuje serijski. Ona se još naziva i sporom granom ili povijesnom granom. Ovo ime dobila je radi toga što podatke obrađuje sporije, nedovoljno brzo da dobije rezultate u realnom vremenu, no upravo radi toga ima mogućnost spremanja velike količine podataka i analizom većeg podatkovnog opsega. Najčešće zadaće ove grane uključuju spremanje ulaznih podataka i obradu podataka u seriji. Za spremanje se često iskorištava datotečni sustav sposoban za pohranu velikih datoteka kao što je Hadoop raspodijeljeni datotečni sustav ili pak baze podataka dizajnirane tako da budu skalabilne i raspodijeljene. Za obradu podataka iskorištavaju se okviri dizajnirani za rad sa velikim količinama podataka, koji su skalabilni i otporni na greške, kao što su *Hadoop MapReduce* okvir, *Pig* ili Apache *Spark*.

Druga grana naziva se brzom granom. Njena zadaća je predstaviti ulazne podatke kao tok podataka te transformirati i obraditi ulazni tok u realnom vremenu. Postoje brojne tehnologije koje mogu izvršiti ovaj posao, no u ovome radu predstavljena je Apache *Spark* strukturirana obrada podataka.

Zahtjev za rezultatom obrade na ovakav sistem uključuje kombiniranje rezultata obje grane. Grana koja je zadužena za serijsku obradu podataka daje točne i pouzdane informacije o puno većem vremenskom periodu. Zahtjev za rezultate ove grane može se dobiti trenutno ukoliko postoje već unaprijed izračunate vrijednosti potrebne za definiranje odgovora. Upravo zato se nakon svakog određenog vremenskog perioda, obrada podataka ponovo izvršava s ciljem davanja točnijih informacija krajnjem korisniku sustava. No kako je definirano ranije, serijska obrada velikih količina podataka je skup proces koji se izvodu duže vrijeme te tako dobivene informacije ne uključuju najkasnije pristigle podatke u sustav. Ovdje dolazi do iskorištavanja grane koja se bavi obradom toka podataka. Ona je u mogućnosti dobiti informacije iz ulaznih podataka u realnom vremenu. Spajanjem rezultata ove dvije grane korisnicima se može pružiti rezultat koji je dobiven obradom ogromne količine povijesnih podataka i podataka koji ulaze u sustav u realnom vremenu. Ovo dovodi do znatnog poboljšanja točnosti dobivenih rezultata. Prikaz lambda arhitekture može se vidjeti na slici 5.1.



Slika 5.1. Shema lambda arhitekture

5.1. Definiranje izvora podataka unutar GradeNotes aplikacije

Svaki sistem koji se bavi obradom velike količine podataka mora imati izvor podataka koji obrađuje. Unutar *GradeNotes* aplikacije izvor podataka je tema unutar Apache *Kafka* okvira. Pozivanjem određene HTTP metode API sučelja, na Apache *Kafka* temu objavljuje se podatkovna struktura koja sadrži informacije o tome koja je putanja do HTTP metode, koji HTTP glagol je korišten, kada je poziv upućen i koliko godina ima korisnik koji je poziv uputio. Unutar tablice 5.1. definirano je REST API sučelje *GradeNotes* aplikacije. Ovo sučelje deinirano je tako da prati standardne operacije koje su definirane CRUD (engl. Create, Read, Update, Delete) operacijama. Dostupne operacije uključuju dohvačanje jedne ili više bilješki, stvaranje jedne bilješke, ažuriranje jedne postojeće bilješke i brisanje jedne bilješke. Operacije koje se izvršavaju nad jednom bilješkom zahtijevaju identifikator bilješke kao argument koji se prosljeđuje u putanji. Čitavo sučelje napisano je unutar ASP.NET Core 2.1. okvira.

Tablica 5.1. REST API sučelje GradeNotes aplikacije

HTTP glagol	Putanja	Opis
GET	/api/note/{noteId}	Dohvaća bilješku sa specificiranim identifikatorom
GET	/api/note	Dohvaća sve bilješke
POST	/api/note	Kreira novu bilješku
PUT	/api/note/{noteId}	Ažurira postojeću bilješku sa specificiranim identifikatorom
DELETE	/api/note/{noteId}	Briše bilješku sa specificiranim identifikatorom

Kako bi se podaci objavili potrebno je integrirati Sučelje Apache *Kafka* okvira unutar .NET okvira. Slika 5.2. prikazuje programski kod zaslužan za objavu poruka na određenu temu.

```
public async Task PublishAsync<TEvent>(TEvent @event, string topic) where TEvent : BaseEvent
{
    // convert event to string value
    var payload = JsonConvert.SerializeObject(@event);

    // send message
    var message = await _producer.ProduceAsync(topic, null, payload);
}
```

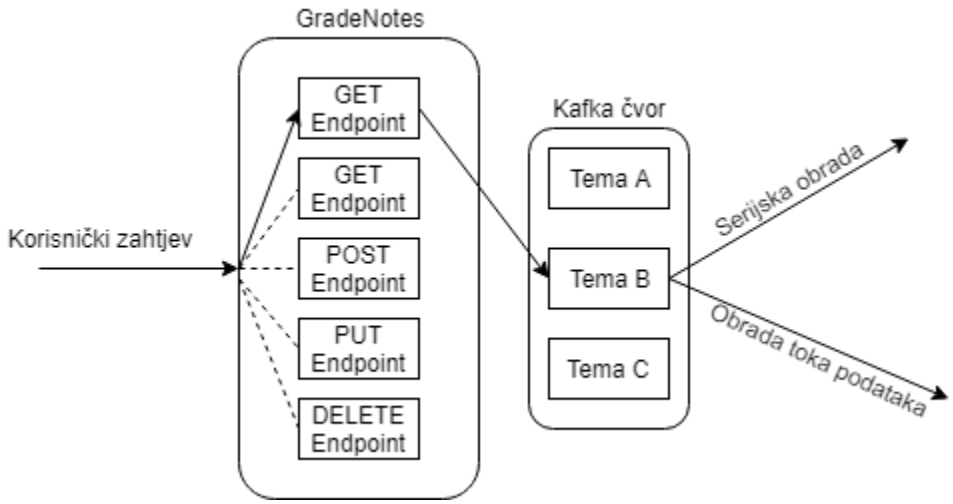
Slika 5.2. Programski kod za objavu poruke na temu

Slika 5.3. prikazuje programski kod metode kontrolera koji je zadužen za rukovanje određenim zahtjevom REST API sučelja. Kao što je vidljivo nakon obrade zahtijeva u kojem se podaci spremaju u bazu podataka, definira se poruka koja će biti poslana na određenu temu.

```
// GET single note
[HttpGet("{noteId}")]
public async Task<IActionResult> GetNote([FromRoute] string noteId)
{
    try
    {
        var note = await _unitOfWork.Notes.SingleOrDefaultAsync(n => n.Id == noteId);

        await _eventBus.PublishAsync(new EndpointHitAnalitycEvent("GET", $"api/Notes/{noteId}", 200),
            _eventBusTopics.WebLogGradeNotes);

        return Ok(_mapper.Map<Note, NoteDTO>(note));
    }
    catch (EntityNotFoundException ex)
    {
        await _eventBus.PublishAsync(new EndpointHitAnalitycEvent("GET", $"api/Notes/{noteId}", 404),
            _eventBusTopics.WebLogGradeNotes);

        return NotFound(new BaseResponse
        {
            StatusCode = AppStatusCode.NoEntitiesFound,
            Description = ex.Message
        });
    }
}
```

Slika 5.3. Metoda kontrolera zadužena za dohvaćanje bilješke

Slika 5.4. prikazuje shemu objave podataka unutar sustava.



Slika 5.4. Objava podataka unutar sustava

5.2. Definiranje Serijske obrade podataka povijesne grane lambda arhitekture

Kako bi se podaci mogli serijski obrađivati oni se moraju spremiti. U ovome radu istraženo je spremanje podataka na *Hadoop* raspodijeljeni računalni sustav, no spremanje poruka koje se objavljuju na *grade-notes-analytics* temu biti će ostvareno uz pomoć *Cassandra* baze podataka. *Cassandra* baza podataka je distribuirana baza koja je lako skalabilna te je dizajnirana tako da podržava veliku količinu strukturiranih podataka. Postoje tri karakteristike svake baze podataka, konzistencija, dostupnost te mogućnost particioniranja. Konzistencija predstavlja svojstvo baze podataka da svaki zapis, nakon što je zapisan u bazu, može biti pročitan ili da će spremanje podatka u bazu rezultirati pogreškom. Ukoliko je ovo svojstvo zadovoljeno za bazu se može reći da je konzistentna. Postoje i eventualno konzistentne baze kojima je potreban određeni vremenski period prije nego što postanu konzistentne. Kod nekonzistentnih baza podataka, zapisivanje podatka u bazu nije nužna garancija da će podatak ikada biti dostupan za čitanje. Dostupnost baze definira vremenski period unutar kojega baza može odgovoriti na zahtjeve korisnika. Mogućnost particioniranja definira se kao sposobnost baze da nastavi izvršavati svoju funkciju čak i kada se podaci nalaze na različitim čvorovima unutar grozda računala. Ovo svojstvo uključuje otpornost na pogreške koje se mogu pojaviti u komunikaciji između čvorova na kojima se nalaze podaci. Prema CAP teoremu svaka baza podataka može zadovoljavati najviše dva od navedenih tri svojstava. *Cassandra* baza podataka dizajnirana je tako da pruži jako veliku mogućnost particioniranja podataka i veliku raspoloživost pod cijenu manje konzistencije podataka. Ovo znači da je potrebno određeno vrijeme dok se podatak ne propagira kroz potrebne čvorove unutar *Cassandra* baze.

Upiti na *Cassandra* bazu izvode se jezikom nazvanim *CQL* (engl. *Cassandra Query Language*). Ovaj jezik sličan je *SQL* jeziku i omogućuje gotovo identičan skup upita. Sve podatkovne tablice unutar *Cassandra* baze podataka definirane su unutar ključnog prostora. Definicija ključnog prostora može se vidjeti na slici 5.5.

```
cqlsh> CREATE KEYSPACE grade_notes WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': '1'} AND durable_writes = true;
```

Slika 5.5. Definiranje ključnog prostora

Prilikom kreiranja ključnog prostora moguće je definirati faktor repliciranja. Ovaj parametar definira koliko će biti kopija podataka unutar tablica definiranih unutar ovog prostora. *Cassandra* se brine za repliciranje podataka u pozadini te od korisnika očekuje jedino definiranje strategije repliciranja. Postavljanjem ovog parametra definira se koji čvorovi su uključeni u repliciranje podataka.

Kako bi se poruke objavljene sa Apache *Kafka* okvirom mogle spremiti potrebno je kreirati tablicu unutar ključnog prostora. Definicija sheme ove tablice podudara se sa definicijom sheme poruke koja se objavljuje na temu. Slika 5.6. prikazuje stvaranje tablice unutar ključnog prostora unutar koje će biti spremljene sve objavljenje poruke.

```
cqlsh:grade_notes> CREATE TABLE endpoint_data (message_id int, method text, url t
ext, status_code text, user_age int, timestamp date, PRIMARY KEY (message_id));
cqlsh:grade_notes>
```

Slika 5.6. Stvaranje tablice unutar ključnog prostora

Kako bi se sve objavljene poruke spremile unutar *Cassandra* baze podataka, potrebno je pokrenuti programski kod prikazan na slici 5.7.

```
import sys
from kafka import KafkaConsumer
import json
from cassandra.cluster import Cluster
consumer = KafkaConsumer(
    'grade_notes_analytics', bootstrap_servers="localhost:6667")
cluster = Cluster(['127.0.0.1'])
session = cluster.connect('gradenotes')
# start the loop
try:
    for message in consumer:
        print(message.value)
        entry = json.loads(message.value)
        print("EventId: {} EndpointMethod: {} EndpointUrl: {} StatusCode: {} CreatedDate: {} Ager: {}".format(
            entry['EventId'],
            entry['EndpointMethod'],
            entry['EndpointUrl'],
            entry['StatusCode'],
            entry['CreatedDate'],
            entry['Age']))
        print("-------------------------------------------------")
        session.execute(
            """
INSERT INTO analytics_stream_data (eventid, age, createddate, endpointmethod, endpointurl, statuscode)
VALUES (%s, %s, %s, %s, %s, %s)
""",
            (entry['EventId'],
             entry['Age'],
             entry['CreatedDate'],
             entry['EndpointMethod'],
             entry['EndpointUrl'],
             entry['StatusCode']))
except KeyboardInterrupt:
    sys.exit()
```

Slika 5.7. Programski kod za spremanje poruka u Cassandra bazu podataka

Ovaj program koristi programsko sučelje za python jezik koje pružaju Apache *Kafka* okvir i *Cassandra* baza podataka. Potrebno je uočiti kako sve vrijednosti koje Cassandra prima moraju biti formatirane kao niz znakova. Programsko sučelje *Cassandra* baze nije u mogućnosti raditi sa drugim tipovima podataka. Objavljivanjem poruka na temu podaci iz poruke, spremiti će se unutar novokreirane tablice. Slika 5.8. prikazuje podatke iz tablice nakon nekoliko objavljenih poruka.

```
cqlsh:gradenotes> SELECT * FROM analytics_stream_data;

 eventid                              | age | createddate                       | endpointmethod | endpointurl                                          | statuscode
--------------------------------------+-----+-----------------------------------+----------------+------------------------------------------------------+------------
 565a84a0-d945-4e47-8499-98cc9662e7b4 |  32 | 2018-08-15T19:43:39.1435746+02:00 |         DELETE | api/Notes/0661b358-e53f-40a4-b9e9-ff4e71cb9db2 |        200
 f78939a2-1433-43b8-b2ad-7fddcdf5e88b |  64 | 2018-08-15T19:43:20.2980378+02:00 |            GET |                                      api/Notes |        200
 5726a174-6d54-41a6-87e2-a5899797a598 |  55 | 2018-08-15T19:43:17.5448135+02:00 |            GET |                                      api/Notes |        200
 cb8e70cd-2f5b-4657-ab16-07be48d74a48 |  47 | 2018-08-15T19:38:27.8411822+02:00 |            GET |                                      api/Notes |        200
 c359a46a-16eb-4bbb-8a1b-67c9a30ef947 |  38 | 2018-08-15T19:43:28.3951429+02:00 |           POST |                                      api/Notes |        201
 5a5c7782-f651-4afb-994d-2a93795e2227 |  31 | 2018-08-15T19:43:26.1036085+02:00 |            PUT | api/Notes/687265e7-909a-4ace-8bc7-786e6edafc43 |        404
 702262c5-646c-47f7-b807-630957cf0d4e |  57 | 2018-08-15T19:43:21.3596963+02:00 |            GET |                                      api/Notes |        200
```

Slika 5.8. Dohvačanje podataka tablice Cassandra baze podataka

Obrada velike količine poruka koje se nalaze u ovoj tablici obavlja se korištenjem *Spark* okvira i *PySpark* biblioteke. Male izmjene potrebne su kako bi se podaci učitali iz *Cassandra* baze te spremili u bazu. Slika 5.9. prikazuje učitavanje podataka u *Spark* podatkovni okvir

```
historical_info = spark.read
    .format("org.apache.spark.sql.cassandra")
    .options(table="endpoint_data", keyspace="grade_notes")
    .load()
```

Slika 5.9. Programski kod za čitanje podataka iz Cassandra baze podataka

Izvršavanje izračuna potrebnih informacija nad ovim podacima izvršava se periodično. Ovaj period trebao bi biti dovoljno velik kako se jedan posao izračuna informacija ne bi preklopio sa drugim, ali ne prevelik tako da grana obrade toka podataka mora držati previše podataka u pričuvnoj memoriji. Spremanje obrađenih podataka u *Cassandra* bazu podataka može se izvršiti definiranjem programskog koda na slici 5.10.

```
historical_info_transformed.write
    .format("org.apache.spark.sql.cassandra")
    .mode('append')
    .options(table="analytics_data", keyspace="grade_notes")
    .save()
```

Slika 5.10. Programski kod za pisanje podataka unutar Cassandra baze podataka

5.3. Definiranje obrade toka podataka brze grane lambda arhitekture

Obrada toka podataka odvija se pomoću Apache *Spark* strukturirane obrade toka podataka kao što je definirano u poglavlju 4. Ova grana u mogućnosti je obraditi velike količine ulaznih podataka te ih grupirati po određenom vremenskom okviru.

Nakon obrade toka podataka potrebno je objaviti nove podatke koristeći Apache *Kafka* okvir. Ovo se postiže definiranjem podatkovnog ponora koji objavljuje podatke na određenu temu. Definicija izlaznog toka podataka može se vidjeti na slici 5.11.

```
analyticsStream
  .selectExpr("to_json(struct(*)) AS value")
  .writeStream
  .queryName("realtime_endpoint_analytics")
  .outputMode("update")
  .format("kafka")
  .option("kafka.bootstrap.servers","localhost:6667")
  .option("topic", "speed_lane_result")
  .option("checkpointLocation", "/home/alen/HDFS/sparkCheckpoints")
  .start()
  .awaitTermination()
```

Slika 5.11. Definicija izlaznog toka podataka

Poruke se objavljuju na temu *speed_lane_result*. Korisnička aplikacija može se pretplatiti na ovu temu i u realnom vremenu pratiti statistike dobivene obradom objavljenih poruka. Kod za pretplatu korisničke aplikacije *GradeNotes* na određenu temu može se vidjeti na slici 5.12. Moguće je vidjeti kako sučelje prema Apache *Kafka* okviru pruža mogućnost definiranja metoda koje će se izvršiti ukoliko se poruka uspješno primi te ukoliko dođe do pogreške prilikom primanja poruke. Ove dvije metode definirane su koristeći upravitelje događajima koji su nazvani *onMessage* te *onError*. Pretplatom na određenu temu svaka poruka koja se dohvati pozivati će jednu od ove dvije metode. Bitno je za naglasiti kako se nit koja izvodi pretplatu na temu blokira prilikom čekanja na poruku. Iz ovog razloga se svaka pretplata, tj. Čekanje na poruke određene teme, odvija u posebnoj niti.

```
public void Subscribe<TEvent>(
    string topic,
    CancellationToken token,
    EventHandler<Message<Null, string>> onMessage = null,
    EventHandler<Error> onError = null)
    where TEvent : BaseEvent
{
    using (var consumer = new Consumer<Null, string>(_config, null, new StringDeserializer(Encoding.UTF8)))
    {
        // Define callbacks
        consumer.OnMessage += onMessage;
        consumer.OnError += onError;

        // Subscribe to topic
        consumer.Subscribe(topic);

        // Define a new thread for consumer so it doesn't block
        var poolingThread = new Thread(_ =>
        {
            while (!token.IsCancellationRequested)
            {
                consumer.Poll(TimeSpan.FromMilliseconds(100));
            }
        });

        // Start fetching messages on topic every 100 miliseconds
        poolingThread.Start();
    }
}

public void Dispose()
{
    _producer?.Dispose();
}
```

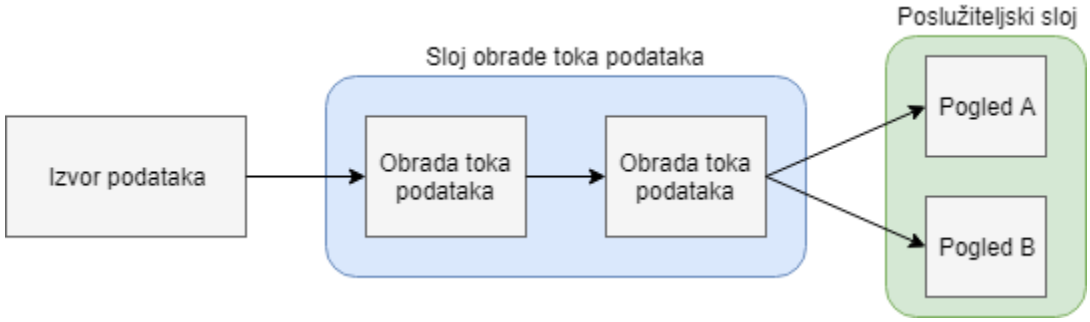
Slika 5.12. Programski kod za pretplatu na temu

5.4. Usporedba lambda i kappa arhitekture

Postoje i druge arhitekture, osim lambda arhitekture, sustava koje pokušavaju u realnom vremenu obraditi velike količine podataka. Jedna od najpoznatijih i najkorištenijih je kappa arhitektura. Ona se temelji samo na obradi toka podataka te tako pokušava smanjiti kompleksnost upravljanja, održavanja i orkestriranja obrade serije podataka.

Kappa arhitektura sadrži samo jednu granu, granu u kojoj se izvršava obrada toka podataka. Ova grana je podijeljena u dva sloja. Prvi sloj zove se sloj realnog vremena te se ne razlikuje od grane obrade toka podataka unutar lambda arhitekture. Drugi sloj zove se poslužiteljski sloj. Njegova zadaća je akumulirati rezultate dobivene obradom toka u nepromjenjive podatkovne zapise. Ovi zapisi podržavaju samo operacije dodavanja i čitanja. Oni se mogu poistovjetiti sa rezultatima koje daje grana obrade serije podataka unutar lambda arhitekture. Korisnički upiti na ovakav sustav dati će rezultate koji se nalaze unutar postojećeg generiranog zapisa. Upravo zato

ovaj zapis može se nazvati pogledom (engl. *View*). Iz ovoga se može zaključiti kako se kappa arhitektura može efektivno iskoristiti u onim slučajevima gdje se svaki red ili poruka koja se nađe unutar ulaznog toka podataka mora obraditi i predstaviti korisniku sustava kao red ili poruku nad kojim se izvršila nekakva funkcija. Ova obrada najčešće se izvršava u više koraka, pa se tako može reći kako je konačni rezultat dobiven korištenjem kompozicije funkcija. Slika 5.13. daje prikaz kappa arhitekture.



Slika 5.13. Shema kappa arhitekture

Unutar ovoga rada predstavljen je podatkovni okvir Apache *Kafka* koji uspješno koristi ovakvu arhitekturu. Svaka objavljena poruka na nekakvu temu obrađuje se te se dodaje u određeni pogled prema temi na koju je poruka objavljena. Korisnici mogu dohvatiti poruke iz određene teme, tj. Iz određenog pogleda. Uočljivo je kako kappa arhitektura pati od istih problema od kojih pati i grana obrade toka podataka lambda arhitekture. Količina akumuliranih podataka koji se stvaraju obradom toka može postati prevelika da se efektivno čuva unutar pričuvne memorije. Sustavi koji koriste ovakvu paradigmu u svom dizajnu rješavaju ovaj problem na dva načina. Prvi je brisanje starijih podataka. Ovo je najlakša metoda za implementirati ukoliko zahtjevi sistema dopuštaju ovakav pristup. Apache *Kafka* koristi ova pristup, gdje se svakoj poruci može odrediti vrijeme života. Istekom ovog vremena poruka se briše iz pogleda u kojem se nalazi te postaje nedostupna korisnicima. Drugi način rješavanja ovog problema je korištenje baza podataka koje se izvode u memoriji računala. Primjer ovakve baze je *Zookeeper*, kojega također koristi Apache *Kafka*.

Bitno je za naglasiti kako kappa arhitektura nije zamjena za lambda arhitekturu[10]. Obje arhitekture imaju svoje prednosti i mane. Kappa arhitektura postavlja se kao bolji pristup dizajniranja sustava ukoliko ne postoji potreba za dugoročnim čuvanjem podataka te se svaki izlazni podatak sustava može predstaviti kao ulazni podatak nad kojim je primijenjen niz određenih transformacija. Lambda arhitektura gradi se nad ovom arhitekturom te uvodi granu serijske obrade

podataka. Ova grana pruža mogućnost dugoročnog čuvanja veće količine podataka te efektivnu upotrebu spremljenih podataka. Odabir arhitekture uvelike ovisi o zahtjevima sistema te podacima koji se obrađuju.

6. ZAKLJUČAK

U ovome radu predstavljena su dva pristupa obrade velike količine podataka. Prvi pristup uključivao je serijsku obradu podataka, dok se drugi pristup bazirao na obradi toka podataka. Serijska obrada podataka izvršila se korištenjem *Hadoop MapReduce* okvira, *Pig* okvira te *Spark* okvira. Obrada toka podataka bazirala se na *Spark* strukturiranoj obradi toka podataka koja se pokazala kao najefektivniji način obrade toka. Prikazane su mane i prednosti oba načina obrade. U zadnjem dijelu rada pokazano je kako na efektivan način spojiti ova dva načina obrade te na temelju njih izgraditi sustav koji je baziran na lambda arhitekturi. Ovdje je dana i usporedba sa kappa arhitekturom. Primjena lambda arhitekture dana je unutar aplikacije *GradeNotes* gdje se uvela obrada statistika o korištenju aplikacije. Korištenjem pristupa opisanih u radu pokazalo se kako obraditi velike količine podataka pritom uzimajući u obzir mogućnost horizontalnog skaliranja dodavanjem novih čvorova u grozd računala.

LITERATURA

[1] http://gennet.com/big-data/big-data-important/, 15.06.2018

[2] http://www.ibmbigdatahub.com/infographic/four-vs-big-data, 16.06.2018

[3] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 16.06.2018

[4] http://hadoop.apache.org/, 15.06.2018

[5] https://hortonworks.com/tutorial/learning-the-ropes-of-the-hortonworks-sandbox/, 18.06.2018

[6] https://hortonworks.com/ecosystems/, 18.06.2018

[7] https://hortonworks.com/apache/spark/, 01.09.2018

[8] https://kafka.apache.org/documentation/#producerapi, 01.09.2018

[9] https://mapr.com/developercentral/lambda-architecture/, 02.09.2018

[10] https://www.oreilly.com/ideas/questioning-the-lambda-architecture, 02.09.2018

SAŽETAK

Arhitekture big data sustava realnog vremena

Korištenjem principa serijske obrade podataka te obrade toka podataka moguće je dizajnirati sustave koji mogu obraditi veliku količinu podataka i korisnicima dati rezultate u realnom vremenu. Lambda arhitektura se u ovom slučaju osobito ističe. Prikazana je primjena ove arhitekture te se naglasak stavio na skalabilnost koju ovakva arhitektura pruža.

Ključne riječi: lambda arhitektura, obrada serije podataka, obrada toka podataka, Hadoop, MapReduce, Apache Spark, Apache Kafka

ABSTRACT

Real time big data architectures

By using principles of batch data processing and data stream processing, it is possible to design systems that can handle large volume of data and give users results in real time. Lambda architecture can be effectively utilized in this case. Usage of this architecture is shown and scalability that it can offer is highlighted.

Keywords: lambda architecture, batch data processing, stream data processing, Hadoop, MapReduce, Apache Spark, Apache Kafka

ŽIVOTOPIS

Alen Čamagajevac rođen je 13.02.1995 godine u Osijeku. Završava osnovnu školu Ivana Kukuljevića Belišće te zatim upisuje Gimnaziju u Srednjoj školi Valpovo. Godine 2013 završava srednjoškolsko obrazovanje te upisuje preddiplomski studij računarstva na Elektrotehničkom fakulteu u Osijeku. Preddiplomski studij završava 2016. godine, te odmah potom upisuje diplomski studij računarstva gdje odabire smjer Informacijske i podatkovne znanosti. Tokom ovog studija pohađa praksu u Gideon Brothers firmi te nakon uspješno odrađene prakse ostaje tamo raditi i steče iskustvo na području razvoja web aplikacija i strojnog učenja.

PRILOZI

[1] GradeNotesAPI – Web aplikacija napisana koristeći *ASP.NET Core 2.1* okvir

[2] GradeNotesAnalytics – Aplikacija napisana koristeći *Spark* okvir i skala programski jezik

[3] Examples – Direktorij u kojem se nalazi niz python skripti za serijsku i obradu toka podataka

[4] Dataset – Direktorij u kojem se nalaze podatkovni skupovi koji se obrađuju u skriptama priloženima u direktoriju *'Examples*'