SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2804

UPOTREBA TEHNOLOGIJA ZA RASPODIJELJENU OBRADU VELIKIH KOLIČINA PODATAKA U KNJIGOVODSTVU

Filip Pažanin

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2804

UPOTREBA TEHNOLOGIJA ZA RASPODIJELJENU OBRADU VELIKIH KOLIČINA PODATAKA U KNJIGOVODSTVU

Filip Pažanin

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2804

Pristupnik:	**Filip Pažanin (0036500450)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentor:	izv. prof. dr. sc. Alan Jović

Zadatak: **Upotreba tehnologija za raspodijeljenu obradu velikih količina podataka u knjigovodstvu**

Opis zadatka:

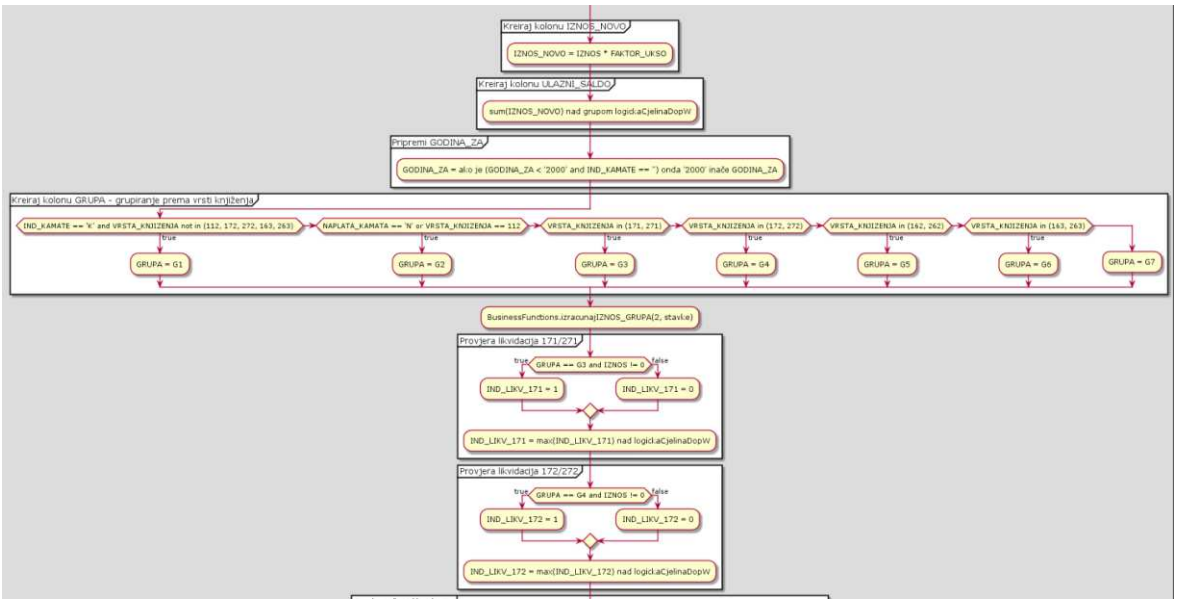
Zbog velike količine podataka i zahtjeva za njihovom brzom obradom, knjigovodstvene usluge izvode se paralelno, u raspodijeljenim okolinama. Da bi se takvo što izvelo potrebno je podatke svesti na oblik koji podržava paralelnu obradu, konfigurirati računalni grozd, instalirati tehnologiju za paralelnu obradu na računalnom grozdu te na kraju upogoniti cijelu obradu te vrednovati rezultate. Paralelna obrada koja se izvršava na raspodijeljenim okolinama predstavlja dodatnu kompleksnost pri izradi u usporedbi s konvencionalnom slijednom obradom na jednom računalu, ali pruža dugotrajne rezultate u obliku skraćenog vremena koje je potrebno da se obrade nad podatcima izvrše. Takva prednost u današnjem svijetu može biti od krucijalnog značaja. Pri konfiguraciji ovakve vrste obrade potrebno je optimalno iskoristiti računalni grozd koji nam je na raspolaganju. Da bi znali kako ga optimalno iskoristiti za pojedine obrade, potrebno je izvršiti ispitivanja s količinom podataka koja odgovara volumenu koji će se uobičajeno koristiti u određenim knjigovodstvenim obradama. U ovom radu razmatrat će se i opisati podaci, poslovna logika i računalni grozd tvrtke Apis IT. Obrada koja će se upogoniti na raspodijeljenoj okolini na paralelan način rada bit će knjigovodstveno saldiranje. U radu će se mjeriti ubrzanja dobivena s obzirom na količinu računalnih resursa koja se koriste te će se usporediti paralelnu obradu sa slijednom obradom, odnosno obradom na jednoj računalnoj jezgri.

Rok za predaju rada: 27. lipnja 2022.

Sadržaj

Uvod

Razlog izrade diplomskog rada na temu knjigovodstvenog saldiranja je zadatak koji sam dobio radeći za tvrtku APIS IT. Moj zadatak bio je pretočiti knjigovodstveni algoritam za saldiranje u programski kod koji podržava paralelnu obradu. Razlog za takvim zahtjevom bilo je predugo vrijeme obrade. U tom trenutku algoritam je bio slijedni te se izvršavao na jednom stroju. Potrebna preinaka bila je pretočiti slijedni algoritam u paralelni te ga pokrenuti na grozdu računala. Isječak prilagođenog knjigovodstvenog algoritma, koji prikazuje početak obrade, moguće je vidjeti na slici 0.1.



Slika 0.1 Isječak knjigovodstvenog algoritma

Možemo primijetiti da algoritam sadrži funkcije kao što su filtriranje, grupiranje i saldiranje koje su pogodne za paralelnu obradu. Zbog toga što je algoritam potrebno provesti na podacima koji su redne veličine nekoliko gigabajta, dolazimo do zaključka da imamo opravdan razlog za upotrebu tehnologija za obradu velikih količina podataka.

Prilagođeni knjigovodstveni algoritam, opisan UML notacijom, bilo je potrebno implementirati u programskom jeziku Scala koristeći pritom radni okvir Spark, a podatke koji se obrađuju prepisani su iz CSV datoteka u Parquet datoteke, format datoteka koji je pogodniji za obradu velikih količina podataka. Takav programski kod, koji podržava

obradu razdijeljenu na više paralelnih procesa, pokrenut je lokalno na malom uzorku podataka da bi se uvjerili da se sam knjigovodstveni algoritam izvršava ispravno. Međutim, prave beneficije ovakvog pristupa obradi podataka postignute su tek kada je programski kod pokrenut na računalnom grozdu koji sadržava računalnu snagu od okvirno deset uobičajenih stolnih računala. Preciznije, računalni grozd koji je bio na raspolaganju sadržavao je 32 jezgre procesora i 134 GB radne memorije. Na računalnom grozdu korišteni su alati i tehnologije za obradu velikih količina podataka, kao što su HDFS, YARN i Spark.

Vođen iskustvom koji sam stekao radom na računalnom grozdu, u ovom radu opisat ću neke od značajnijih tehnologija koje se koriste u svijetu velike količine podataka (engl. *big data*). Za početak je potrebno definirati ovaj pojam. *Big data* je tehnologija koja služi za prikupljanje, obradu i analizu velike količine podataka. Podaci su raznoliki, strukturirani i nestrukturirani, generiraju se i pristižu velikom brzinom i u različitim intervalima (ponekad i u realnom vremenu), što ih čini vrlo složenim za analizu. Međutim, prikupljanje i skladištenje velikih količina podataka nije ono što čini tehnologiju *big data* korisnom. Upravo je mogućnost obrade i analiza tih prikupljenih podataka za daljnju upotrebu ono što ovu tehnologiju čini vrlo vrijednom. Bez mogućnosti analize, bila bi to samo gomila prikupljenih podataka koja bi nam zauzimala prostor na disku [1].

Rad je podijeljen u četiri središnja i četiri općenita poglavlja (uvod, zaključak, literatura, sažetak). Prvo središnje poglavlje bavit će se opisom tehnologije *Apache Hadoop*, koja se smatra pomalo zastarjelom, ali zapravo predstavlja temelj mnogih drugih, novijih, tehnologija. Drugo središnje poglavlje opisuje tehnologiju *Apache Spark* koja je korisniku omogućila sve što mu je potrebno za kvalitetno analiziranje velikih količina podataka, na jednom mjestu, uz jednostavno korištenje i veliku brzinu rada. U trećem poglavlju opisat ću što je to Parquet datoteka i zašto je pogodna kod obrade velikih količina podataka. Za kraj, u četvrtom poglavlju, prikazat ću rezultate ispitivanja performansi na računalnom grozdu i pokušat ću donijeti neke zaključke u vezi podešavanja parametara paralelne obrade koja će se izvoditi na računalnom grozdu.

1. Tehnologija Apache Hadoop

Apache Hadoop je skup programskih rješenja otvorenog koda koji olakšava korištenje više računala u mreži u svrhu rješavanja problema obrade velike količine podataka. Hadoop nam pruža programski radni okvir (engl. *framework*) za raspodijeljenu pohranu i obradu velikih količina podataka pomoću programskih modela MapReduce. Hadoop je izvorno osmišljen za rad na grozdu računala s lošim do osrednjim performansama odnosno računalima namijenjenim „običnim“ korisnicima, no također se može koristiti i na grozdu vrhunskih računala. Svi su moduli u Hadoopu oblikovani s temeljnom pretpostavkom da su kvarovi na sklopovlju česta pojava te da ih programski okvir treba automatski rješavati.

Jezgra Apache Hadoopa sastoji se od dijela za pohranu, poznatog kao Hadoop Distributed File System (HDFS) i dijela za obradu koji je implementiran u obliku programskog okvira MapReduce. Hadoop dijeli datoteke u velike blokove te ih raspoređuje po čvorovima na grozdu. Nakon toga Hadoop prenosi zapakirani kod na čvorove i izvršava paralelnu obradu podataka. Ovakav način obrade podataka koristi prednost lokalnosti podataka, gdje čvorovi manipuliraju samo podacima koji se nalaze na njima. Takav pristup omogućuje bržu i učinkovitiju obradu podataka nego što je to slučaj kod konvencionalne arhitekture superračunala koja se oslanja na paralelni datotečni sustav gdje se računanje i podatci raspodjeljuju brzom mrežnom infrastrukturom. Osnovni okvir Apache Hadoopa sastoji se od sljedećih modula: Hadoop Common, Hadoopov raspodijeljeni datotečni sustav (HDFS), Hadoop YARN, Hadoop MapReduce i Hadoop Ozone (uveden 2020.).

Hadoop Common je modul koji sadrži knjižnice i programe koji su potrebni drugim Hadoopovim modulima.

HDFS je raspodijeljeni datotečni sustav koji pohranjuje podatke te pruža vrlo veliku ukupnu propusnost na grozdu s uglavnom jeftinim računalima.

Hadoop YARN (uveden 2012. godine) je platforma odgovorna za upravljanje računalnim resursima i rasporedom rada korisničkih aplikacija na grozdu.

Hadoop MapReduce je implementacija programskog modela MapReduce za obradu velike količine podataka.

Hadoop Ozone je trgovina aplikacija za Hadoop. Hadoop promatran kao ekosustav može sadržavati razne druge dodatne programske pakete kao što su Apache Pig, Apache Hive, Apache HBase, Apache Spark, Apache Zookeper, Cloudera Impala, Apache Flume, Apache Storm i mnoge druge.

Komponente Apache Hadoop MapReduce i HDFS baziraju se na Googleovim znanstvenim člancima o tehnologiji MapReduce i njihovom datotečnom sustavu, poznatom pod nazivom Google File System.

Sam okvir Hadoop, većim djelom, napisan je u programskom jeziku Java, dok se na nekim mjestima koristio programski jezik C i skripte namijenjene za izravno izvršavanje u naredbenom retku, odnosno u ljusci. Iako se najčešće koristi programski jezik Java da bi se napisale implementacije funkcija *map* i *reduce* u programskom okviru Hadoop MapReduce, mogu se koristiti i razni drugi programski jezici. Veliki broj projekata u ekosustavu Hadoop sadrže bogata i intuitivna korisnička sučelja [1].

1.1. Arhitektura Apache Hadoopa

Hadoop se sastoji od paketa Hadoop Common, koji pruža apstrakciju na razini operacijskog i datotečnog sustava, stroja MapReduce i HDFS-a. Paket Hadoop Common sadrži razne JAR datoteke i skripte potrebne za pokretanje Hadoopa. Da bi se zadatci mogli učinkovito raspoređivati na grozdu računala, datotečni sustav mora osigurati uvid u lokaciju čvora radnika (engl. *worker node*) odnosno mora pružiti informaciju o tome na kojem se točno skupu računala (engl. *rack*) radnik nalazi te na kojem mu se mrežnom preklopniku (engl. *network switch*) može pristupiti. Ove informacije omogućavaju Hadoopovoj aplikaciji da učinkovito izvršava računalni kod na onim čvorovima na kojima se podatci nalaze. U slučaju da računalni kod nije moguće izvršiti na istom onom računalu na kojem se podatci nalaze, Hadoop će pokušati računalni kod izvršiti na računalima koja su izravno spojena s računalom na kojem se nalaze podatci putem zajedničkog mrežnog usmjernika (engl. *router*). Na taj način Hadoop efikasno smanjuje vrijeme potrebno za izvršavanje računalnog koda tako što smanji ukupnu količinu podataka koja se prenosi preko mreže. HDFS također pri kreiranju podatkovnih replika vodi računa o tome gdje se podatci nalaze. Podatke smješta na različite skupove računala koji su neovisni jedan o drugome po pitanju napajanja. Na taj način ukoliko dođe do sklopovskog kvara na jednom

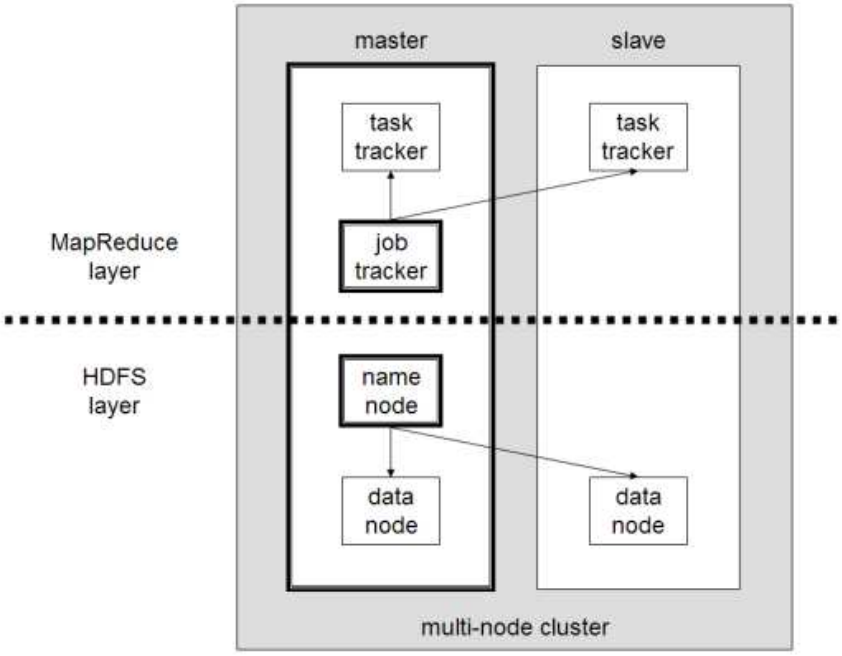
skupu računala u grozdu (npr. nestanak napajanja), podatci će ostati dostupni jer su replicirani na drugi neovisni skup računala.

Grozd na kojem je Hadoop podešen uključuje jedan glavni i više čvorova radnika. Glavni čvor sastoji se od tragača poslova (engl. *Job Tracker*), tragača zadataka (engl. *Task Tracker*), imenskog čvora (engl. *NameNode*) i podatkovnog čvora (engl. *DataNode*). Radnički čvor djeluje i kao podatkovni čvor i kao tragač zadataka iako je moguće imati radne čvorove zadužene samo za podatke ili samo za izračunavanje. Oni se obično koriste samo u nestandardnim aplikacijama.

Hadoop zahtijeva Java Runtime Environment (JRE) verzije 1.6 ili noviju. Standardne skripte za pokretanje i isključivanje zahtijevaju postavljanje sigurnosne ljuske SSH (engl. *Secure Shell*) između čvorova na grozdu.

U većim grozdovima postoje primarni i sekundarni imenski čvorovi. Primarni čvor ima zadatak pružati informacije o indeksima datoteka u datotečnom sustavu, dok sekundarni ima ulogu generiranja snimki memorije imenskog čvora, čime sprječava oštećenje datotečnog sustava i gubitak podataka. Slično tome, samostalni poslužitelj s funkcijom tragača poslova može upravljati raspoređivanjem poslova na čvorovima. Kada se Hadoop MapReduce koristi s nekim alternativnim datotečnim sustavom, imenski čvor, sekundarni imenski čvor i arhitektura podatkovnih čvorova zamjenjuju se s ekvivalentima specifičnima za zadani datotečni sustav [1].

Na slici 1.1 vidi se infrastruktura grozda s instaliranim Hadoopom koji je podijeljen na dva djela: HDFS i MapReduce. Vidimo da je dio HDFS sastavljen od imenskog čvora i više dodijeljenih mu podatkovnih čvorova, dok se na sloju MapReduce nalazi tragač poslova koji komunicira s više tragača zadataka.

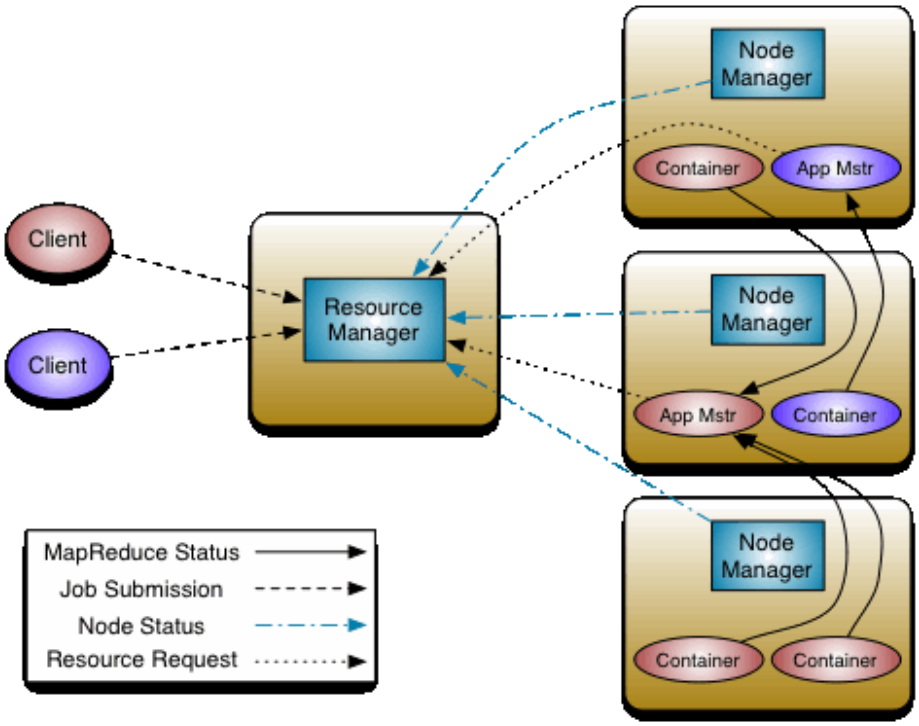


Slika 1.1 Grozd s instaliranim Hadoopom sastavljen od više čvorova [2]

1.2. Platforma YARN

YARN je upravitelj računalnog grozda koji dodjeljuje i upravlja računalnim resursima. Potreba za upraviteljem resursa računalnog grozda došla je s pojavom poslova MapReduce (engl. *MapReduce job*) koji su se izvršavali na svakom pojedinom čvoru u računalnom grozdu gdje su se podaci, koji se obrađuju, nalazili. Prvotnu je ulogu za pronalaženje svih čvorova gdje se nalaze potrebni podaci za obradu imao tzv. JobTracker. JobTracker je imao funkciju pronalaska i zahtijevanja određenih računalnih resursa od čvorova, kao što su broj procesora i veličina radne memorije. Nakon što bi JobTracker identificirao potrebne čvorove te rezervirao određene računalne resurse na njima, posao MapReduce bi se mogao pokrenuti. Svi čvorovi koji sudjeluju u poslu MapReduce imali bi na sebi pokrenut tzv. TaskTracker koji bi prijavljivao stanje čvora JobTrackeru. JobTracker je tako imao dvije funkcije: alociranje resursa za poslove MapReduce i upravljanje zadacima na čvorovima. JobTracker bi tako npr. brinuo što ako neki od čvorova iznenadno prekine sa radom. Ovakvim pristupom, u sustavu je postojala jedinstvena točka kvara (engl. *single point of failure*). Taj nedostatak bio je karakterističan za verziju Hadoop 1. Osim toga, postojao je i problem skalabilnosti, JobTracker bi se vrlo lako zagušio zbog svoje višestruke uloge pri pokretanju više poslova MapReduce. Da bi se popravilo sve navedene

nedostatke i da bi se na računalnom grozdu s HDFS-ovom podatkovnom strukturom moglo pokretati ne samo poslove MapReduce, već i npr. poslove Sparka, Hadoop, u verziji 2, je razvio YARN. Platforma YARN (engl. *Yet Another Resource Negotiator*), za razliku od svog prethodnika JobTrackera, ima samo jednu funkciju, a to je upravljanje računalnim resursima na računalnom grozdu. YARN podržava odnosno ima mogućnosti dodjeljivati računalne resurse različitim radnim okvirima kao što su MapReduce, Spark, Storm i HBase. Upravljanje zadatcima na čvorovima računalnog grozda izdvojeno je u zasebnu stavku koju nazivamo upraviteljem aplikacija (engl. *Aplication Master*). YARN, kada dobije zahtjev da odradi neki posao na računalnom grozdu, utvrđuje sve čvorove koji mogu sudjelovati te na jednom od njih stvara upravitelja aplikacija. Da bi YARN mogao ustvrditi koji računalni čvor može koristiti, YARN mora ispitati odnosno dobiti dopuštenje od upravitelja čvora (engl. *Node Manager*) koji je instaliran na svakom pojedinom čvoru. Upravitelj čvora služi kao predstavnik čvora u računalnom grozdu koji komunicira s menadžerom resursa (engl. *Resource Manager*) i eventualno mu dopušta izvršavanje nekog određenog posla na čvoru. Opisana arhitektura YARN-a prikazana je na slici 1.2. Upravitelj aplikacijom, stvoren na nekom od čvorova koji se mogu upotrebljavati za izvršavanje posla, stvara kontejnere na kojim se zadatak odrađuje. Ukoliko neki od kontejnera iznenadno prekine s radom, upravitelj aplikacije je taj koji upućuje zahtjev upravitelju resursa za dodatnim resursima te ukoliko postoji mogućnost odnosno ukoliko neki od upravitelja čvorova prihvati zahtjev, stvara se novi kontejner na čvoru s raspoloživim resursima.



Slika 1.2 Prikaz arhitekture YARN-a na računalnom grozdu [3]

Upravitelj resursa ima sposobnost zakazivanja posla ukoliko zaprimi neki posao koji se u istom trenutku ne može izvršiti na računalnom grozdu zbog nedostatka resursa. Način na koje raspoređuje poslove može biti FIFO, *Fair* i *Capacitive*. Pristup FIFO označava raspoređivanje poslova u onom redoslijedu po kojem su poslovi došli do upravitelja resursa. Pristup *Fair* daje jednaku šansu za izvršavanjem svakom od poslova koji čekaju na izvršavanje, dok kod pristupa *Capacitive* poslovi dobivaju onoliko resursa koliko je deklarirano da je potrebno te ukoliko u nekom trenutku nije dostupno onoliko resursa koliko je potrebno, posao čeka dok se resursi ne oslobode [4].

2. Projekt Apache Spark

Apache Spark je projekt otvorenog koda koji predstavlja programski stroj za analizu i obradu velike količine podataka. Spark pruža sučelje za stvaranje čitavih grozdova s karakteristikom paralelne obrade podataka i tolerancijom na pogreške. Izvorno je razvijen na Sveučilištu Kalifornija, Berkeley, u laboratoriju AMPLab te je kasnije doniran zakladi Apache Software Foundation koja ga i dan danas održava i unaprjeđuje.

Apache Spark svoju arhitekturu temelji na otpornom raspodijeljenom skupu podataka zvanom RDD. RDD (engl. *Resilient Distributed Dataset*) predstavlja skup podataka koji je raspoređen po čitavom grozdu računala na način da podatke nije moguće mijenjati već samo čitati te se podatke održava na način da budu dostupni i uz pojavu kvarova na računalnom grozdu. U kasnijim verzijama Apache Sparka stvarani su API-ji *Dataframe* i *Dataset*, namijenjeni olakšanom korištenju ove tehnologije i približavanju korisnicima koji su navikli na rad s relacijskim bazama podataka. Oba API-ja samo predstavljaju apstrakciju rada s RDD-ovima te je i dalje omogućeno korisnicima da izravno rade s RDD-ovima ukoliko se to želi.

Spark i tehnologija RDD stvoreni su 2012. godine kao odgovor na ograničenja u računalnoj paradigmi MapReduce koja se koristila na grozdovima računala, a koja inzistira na određenom linearnom protoku podataka u raspodijeljenim programima. Taj linearni protok sastoji se od čitanja ulaznih podataka s diska, preslikavanja podataka zadanom funkcijom, reduciranja rezultata funkcije *map* funkcijom *reduce* te spremanja dobivenih rezultata trajno da disk. Sparkovi RDD-ovi funkcioniraju kao radni skup za raspodijeljene programe koji predstavljaju namjerno ograničeni oblik raspodijeljene zajedničke memorije.

Spark olakšava implementaciju iterativnih algoritama, koji koriste svoj skup podataka više puta u okviru petlje, ali i interaktivnih algoritama koji služe za analizu podataka, kao što je npr. postavljanje višestrukih upita na bazu podataka. Latencija takvih algoritama može se smanjiti za nekoliko redova veličine korištenjem Sparka u usporedbi s implementacijom pomoću Apache Hadoop MapReducea. Među iterativne algoritme spadaju i algoritmi za učenje sustava strojnog učenja koji su inicijalno i bili jednim od razloga za stvaranjem Apache Sparka.

Apache Spark zahtijeva upravitelja grozda te raspodijeljeni sustav za pohranu podataka. Za upravljanje grozdom, Spark nudi mogućnosti samostalnog upravljanja, upravljanja pomoću Hadoop YARN-a, Apache Mesosa ili pomoću Kubernetesa. Od sustava za raspodijeljenu pohranu podataka, Spark podržava mnoge, neki od njih su: Alluxio, HDFS, MapR File System, Cassandra, OpenStack Swift, Amazon S3, Kudu. Spark također podržava pseudo-raspodijeljeni lokalni način rada, koji se obično koristi u svrhu razvoja ili ispitivanja, gdje raspodijeljena pohrana nije potrebna već se koristi lokalni datotečni sustav. U tom slučaju Spark se pokreće na jednom stroju s jednim izvršiteljem po jezgri CPU-a [5].

2.1. Temelj Sparka – Spark Core

Spark Core, kao i što sam naziv govori, temelj je čitavog Sparka. On omogućuje raspodijeljeno dijeljenje podataka, raspoređivanje i osnovne funkcionalnosti I/O izložene kroz sučelja programskih jezika kao što su Java, Python, Scala, .Net i R. Ta sučelja podržavaju funkcionalni model programiranja gdje upravljački program (engl. *driver*) poziva operacije koje se mogu izvršavati paralelno nad RDD-om, kao što su preslikavanje, filtriranje i reduciranje, na način da definira funkcije te ih prosljeđuje Sparku koji ih raspoređuje i paralelno izvršava na grozdu. Sve navedene operacije, kao i operacije spajanja, uzimaju RDD kao ulaz i stvaraju novi RDD. RDD su nepromjenjivi i operacije nad njima su lijene, što znači da se ne izvršavaju sve dok nam rezultat istih nije potreban. Tolerancija na pogreške postiže se praćenjem izmjena svakog RDD-a, odnosno bilježenjem svake operacije koja ga je izmijenila počevši od inicijalnog stanja. Na taj način RDD možemo obnoviti u bilo koje od prijašnjih stanja u slučaju gubitka podataka. RDD-ovi mogu sadržavati bilo koju vrstu objekata u Pythonu, .NET-u, Javi ili Scali.

Osim funkcionalnog stila programiranja usmjerenog na RDD, Spark nudi dva ograničena oblika zajedničkih varijabli, varijable emitiranja (engl. *broadcast*) koje predstavljaju podatke koji su samo za čitanje te moraju biti dostupni na svim čvorovima na grozdu i varijable akumuliranja (engl. *accumulators*) koje služe za reduciranje skupa podataka koji se početno nalazi na više čvorova.

Na slici 2.1 prikazan je tipičan primjer funkcionalnog programiranja usmjerenog na RDD u programskom jeziku Scala. Navedeni program izračunava frekvenciju svih riječi koje se pojavljuju u skupu tekstnih datoteka i ispisuje one najčešće. Svaka funkcija *map*, *flatMap*

(posebna inačica funkcije *map* koja vraća rezultate u obliku nadređenog skupa podataka) i *reduceByKey* uzima anonimnu funkciju koja provodi jednostavnu operaciju na pojedinoj podatkovnoj stavci (ili paru stavki) i primjenjuje ju nad svojim argumentom da bi zadani RDD pretvorila u novi RDD [5].

```
val conf = new SparkConf().setAppName("wiki_test") // create a spark
config object
val sc = new SparkContext(conf) // Create a spark context
val data = sc.textFile("/path/to/somedir") // Read files from "somedir"
into an RDD of (filename, content) pairs.
val tokens = data.flatMap(_.split(" ")) // Split each file into a list
of tokens (words).
val wordFreq = tokens.map((_, 1)).reduceByKey(_ + _) // Add a count of
one to each token, then sum the counts per word type.

wordFreq.sortBy(s => -s._2).map(x => (x._2, x._1)).top(10) // Get the
top 10 words. Swap word and count to sort by count.
```

Slika 2.1 Primjer Scalinog programa za brojanje frekvencije pojave riječi u dokumentima [5]

2.2. Komponenta Spark SQL

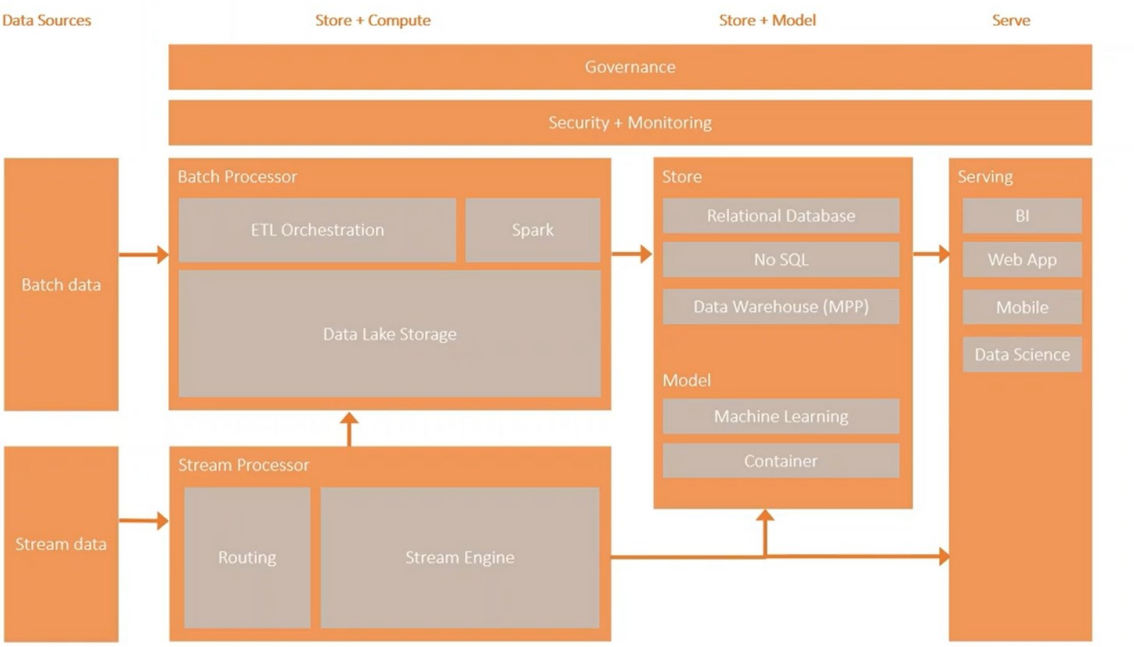
Spark SQL je komponenta koja se nalazi iznad Spark Corea, a uvela je apstrakciju podataka nazvanu podatkovni okviri (engl. *DataFrames*). Oni nam pružaju podršku za strukturirane i polu-strukturirane tipove podataka. Spark SQL nudi jezik specifičan za domenu (engl. *domain specific language*, DSL) koji služi za upravljanje podatkovnim okvirima u jezicima Scala, Java, Python i .NET. Osim toga, Spark SQL nudi podršku za pisanje naredbi u jeziku *SQL*, sučelje za naredbeni redak te mogućnost spajanja na poslužitelj tipa JDBC/ODBC. Iako u slučaju podatkovnih okvira ne postoji mogućnost provjere koda prije samog izvršavanja koda (engl. *compile-time type checking*) kao što je to slučaj kod RDD-ova, od Sparkove inačine 2.0 uvodi se novi tip podataka, podatkovni skup (engl, *DataSet)*, strogo tipiziran podatak (engl. *strongly typed*) kojemu je potrebno definirati strukturu prije samog izvođenja. Na slici 2.2 možemo vidjeti primjer korištenja podatkovnih okvira gdje se spajamo na bazu podataka, uzimamo podatke iz tablice “people“ i računamo broj ljudi s obzirom na njihovu starost.

```
import org.apache.spark.sql.SparkSession

val url =
"jdbc:mysql://yourIP:yourPort/test?user=yourUsername;password=yourPas
sword" // URL for your database server.
val spark = SparkSession.builder().getOrCreate() // Create a Spark
session object

val df = spark
  .read
  .format("jdbc")
  .option("url", url)
  .option("dbtable", "people")
  .load()

df.printSchema() // Looks the schema of this DataFrame.
val countsByAge = df.groupBy("age").count() // Counts people by age

//or alternatively via SQL:
//df.createOrReplaceTempView("people")
//val countsByAge = spark.sql("SELECT age, count(*) FROM people GROUP
BY age")
```

Slika 2.2 Primjer korištenja Sparkovih podatkovnih okvira

2.3. Komponenta Spark Streaming

Spark Streaming koristi sposobnosti brzog raspoređivanja koje nudi Spark Core u svrhu obavljanja analize toka podataka (engl. *stream*). Podaci se konzumiraju u malim obrocima (engl. *mini-batches*) na način da se nad njima provode RDD-ove transformacije. Ovakav način oblikovanja omogućuje da se isti računalni kod koji je napisan za analizu obroka podataka (engl. *baches*) koristi i za analizu toka podataka, omogućavajući nam time jednostavnu implementaciju arhitekture lambda. Arhitektura lambda označava način obrade velikih količina podataka na način da se prvenstveno izvršava obrada obroka podataka i obrada toka podataka, zatim se rezultati obrade spremaju i po potrebi modeliraju te sa naposljetku rezultati obrade poslužuju korisniku u formatu izvještaja ili aplikacija. Slika 2.3 detaljno ilustrira arhitekturu lambda.



Slika 2.3 Arhitektura lambda [7]

Međutim, pristup analizi toka podataka na način konzumiranja malih obroka dolazi s nedostatcima, a to su kašnjenja koja iznose onoliko koliko iznosi trajanje jednog malog obroka. Za razliku od Spark Streaminga, postoje mehanizmi za obradu toka podataka koji obrađuju događaj po događaj, a ne male obroke podataka. Neki od njih su Storm i Flink. Spark Streaming ima ugrađenu podršku za konzumaciju podataka iz Kafke, Fluma, Twittera, ZeroMQa, Kinesisa i TCP/IP-ovih utičnica (engl. *sockets*).

U verziji Sparka 2.x., uz Spark Streaming uvedena je dodatna tehnologija temeljena na podatkovnim skupovima (engl. *Datasets*) nazvana Structured Streaming, koja osigurava korisniku još višu razinu apstrakcije pri obradi toka podataka.

2.4. Radni okvir Spark MLLib

Spark MLlib (od engl. *Machine Learning Library*) raspodijeljeni je radni okvir korišten u svrhu strojnog učenja, a radi na temelju Spark Corea. Zahvaljujući arhitekturi koju koristi, raspodijeljenu memorijsku strukturu, Spark MLlib čak je do devet puta brži od Apache Mahouta koji je implementiran korištenjem diska [5]. Mnogi često korišteni algoritmi za strojno učenje i statistiku implementirani su i isporučeni s MLlibom što uvelike

pojednostavljuje kreiranje cjevovoda velikih razmjera u procesima strojnog učenja. Neki od ugrađenih algoritama su statističko ispitivanje hipoteze, klasifikacija, regresija, suradničko filtriranje, algoritam *k*-srednjih vrijednosti te mnogi algoritmi za redukciju dimenzionalnosti, transformaciju funkcija i optimizaciju.

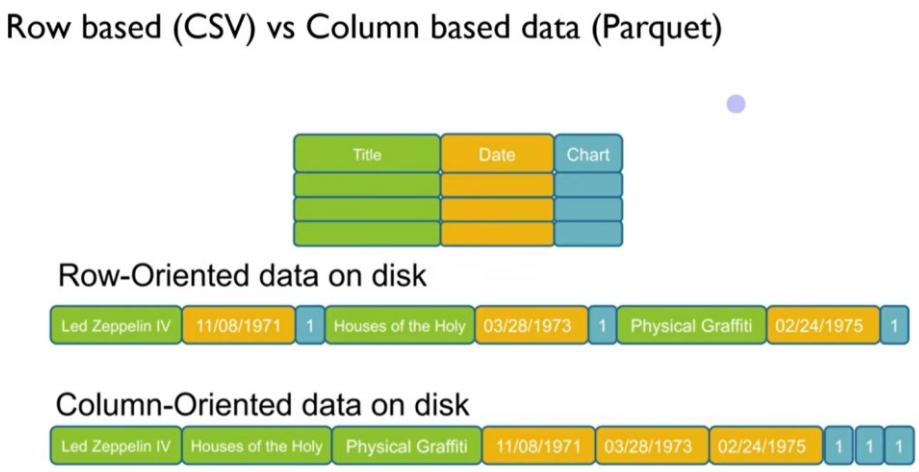
2.5. Komponenta GraphX

GraphX je Sparkov radni okvir za raspodijeljenu obradu grafova. Budući da se temelji na RDD-ovima, koji su nepromjenjivi, grafovi su također nepromjenjivi pa je stoga GraphX neprikladan za grafove koje je potrebno ažurirati tijekom vremena. Stoga GraphX možemo promatrati kao bazu podataka u kojoj su podaci zapravo grafovi. GraphX nudi dva odvojena API-ja za implementaciju masovno paralelnih algoritama (kao što je *PageRank*). To su Pregelova apstrakcija (engl. *Pregel abstraction*) i drugi, općenitiji API, nalik *MapReduceu*. GraphX, za razliku od svog pretka Bagela, koji je bio aktivan do verzije Sparka 1.6, ima mogućnost korištenja tzv. grafova svojstava, grafova u kojima svojstva mogu biti dodijeljena rubovima i vrhovima grafa.

3. Datotečni format Apache Parquet

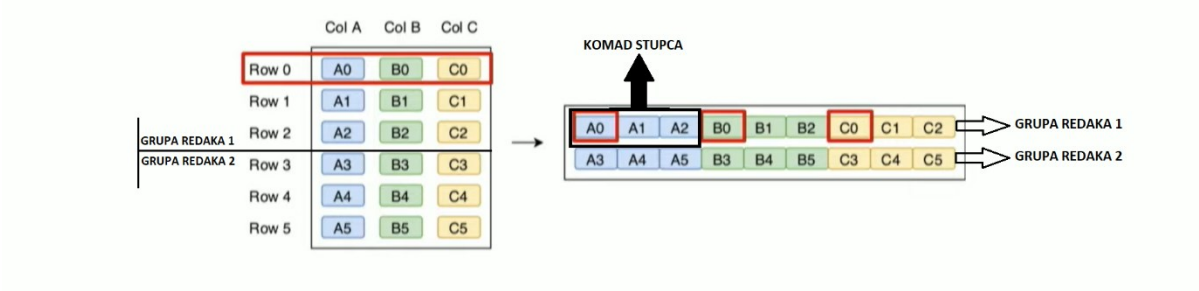
Parquet je naziv za binarni datotečni format koji je pogodan za korištenje u analitici velikih količina podataka. Ovaj datotečni format omogućuje efikasno spremanje i izvršavanje upita nad velikim količinama podataka. Parquet je moguće integrirati u mnoge sustave odnosno radne okvire analitike podataka kao što su Azure, Google Cloud, AWS i Apache Spark.

Za razliku od CSV datoteka, koje spremaju podatke redak po redak, Parquet datoteka podatke sprema stupac po stupac kao što je prikazano na slici 3.1.



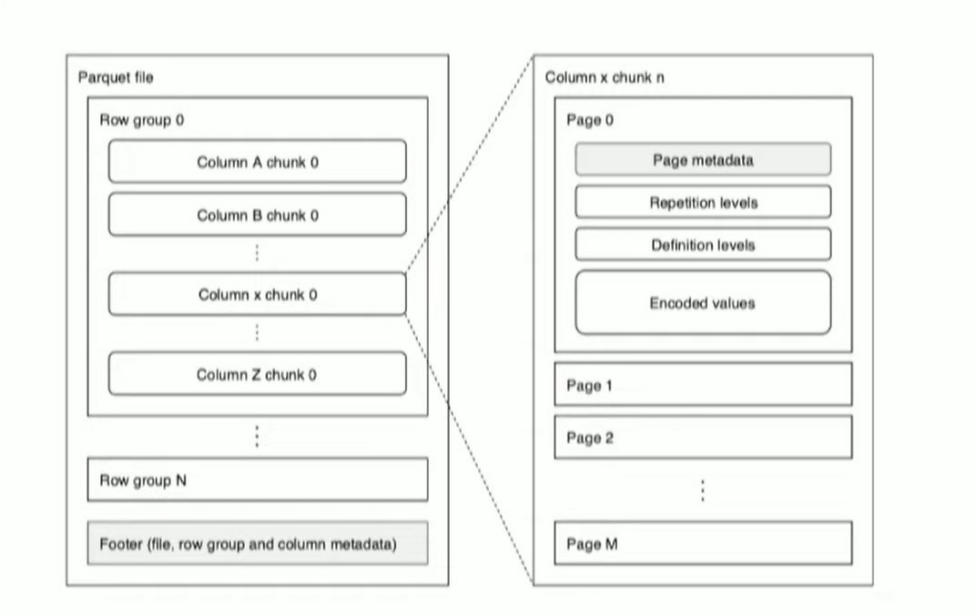
Slika 3.1 Usporedba spremanja na disk redoslijedom redak po redak odnosno stupac po stupac [8]

Preciznije, Parquet datoteka sprema podatke na hibridan način, dijeli datoteku na grupe redaka pa zatim grupe redaka sprema na način stupac po stupac. Kako izgleda hibridni način pohrane i kako izgledaju grupe redaka i komadi stupaca u tabličnim podatcima predočeno je na slici 3.2.



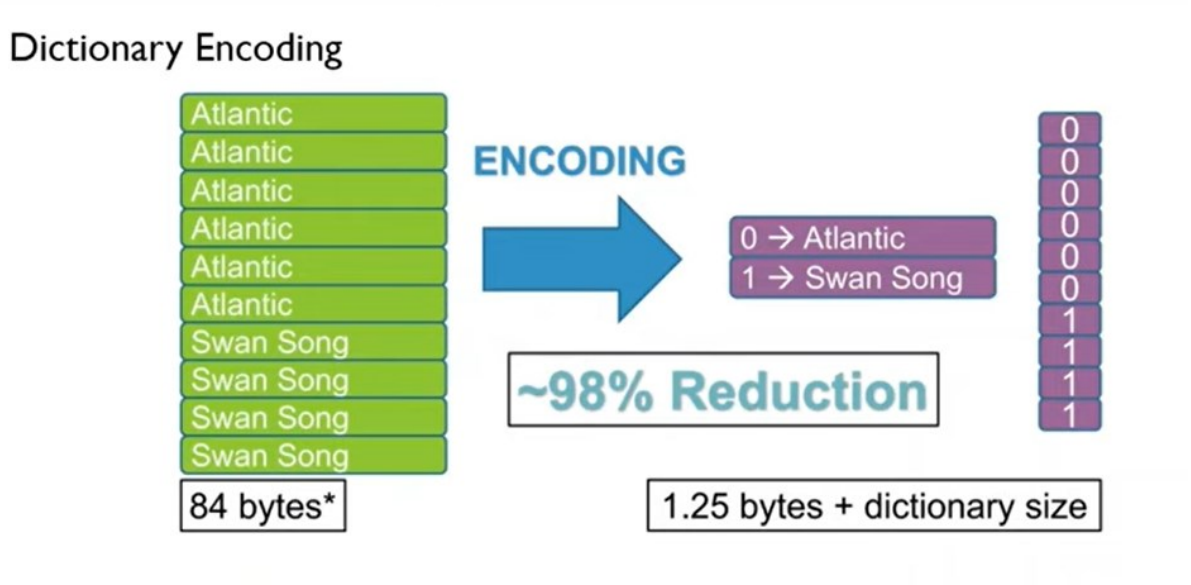
Slika 3.2 Prikaz podjele podataka na grupe redaka kod hibridnog načina pohrane

Na slici 3.3 možemo vidjeti kako izgleda struktura Parquet datoteke. Svaka Parquet datoteka sadrži meta podatke koji se nalaze u podnožju datoteke. Ti meta podatci sadrže informacije o lokacijama ostalih meta podataka u datoteci, meta podataka stupca i meta podataka stranica. U Parquet datoteci podaci su podijeljeni u grupe redaka (engl. *row group*), grupe redaka su ilustrirane na prethodnoj slici, slici 3.2. Parquet datoteka može sadržavati više grupa redaka, uobičajena veličina grupe redaka je 128 MB. Grupe redaka sastoje se od komada stupaca (engl. *Column chunk*) koji predstavljaju dijelove stupaca koji su obuhvaćeni određenom grupom redaka. Za kraj, sami podaci zapisani su unutar komada stupaca u strukturi nazvanoj stranica (engl. *page*), uobičajene veličine 1 MB. Svaka od tih stranica, osim kodiranih podataka, sadrži i meta podatke o podacima koji su sadržani u toj stranici kao što su minimalna i maksimalna vrijednost podataka i broj zapisa. Čitanjem tih meta podataka, prije samih vrijednosti podataka, značajno ubrzavamo način filtriranja jer, ukoliko želimo, recimo, dohvatiti sve zapise koji u određenoj koloni imaju vrijednost veću od 5 i ako znamo da je u stranici koja sadrži podatke o toj zatraženoj koloni maksimalni broj 4, onda sa sigurnošću možemo preskočiti cijelu tu stranicu s 1 MB podataka koje ona sadržava jer unutar nje ne postoji broj veći od 5. Još jedna tajna uspješnosti Parquet datoteka jest ta da su podaci koji se spremaju u stranicama kodirani te da je na kraju cijela Parquet datoteka komprimirana [8].



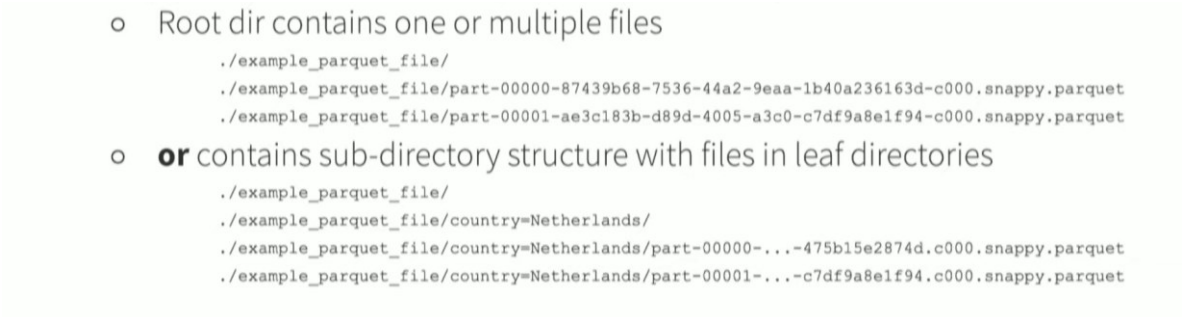
Slika 3.3 Prikaz strukture Parquet datoteke [8]

Kodirati podatke isplati se kada se podaci unutar komada stupca ponavljaju, jer u tom slučaju podatke možemo stvaranjem rječnika zamijeniti s nečim što sadrži istu informaciju, ali zauzima manje prostora. Na slici 3.4 možemo vidjeti kako kodiranjem upotrebom rječnika možemo uštediti čak 98% prostora potrebnog za pohranu podataka. Osim kodiranja, Parquet datoteka koristi i princip sažimanja (kompresije) koje provodi nad cijelom datotekom, uobičajeno metodom Snappy. Metoda sažimanja Snappy pomaže smanjenju veličine Parquet datoteke u odnosu na CSV datoteku do 8 puta i smanjuje vrijeme koje je potrebno za izvršavanje upita nad podatcima čak do 100 puta u odnosu na upite koje se izvršavaju nad CSV datotekom [6].



Slika 3.4 Prikaz kodiranja podataka upotrebom rječnika [6]

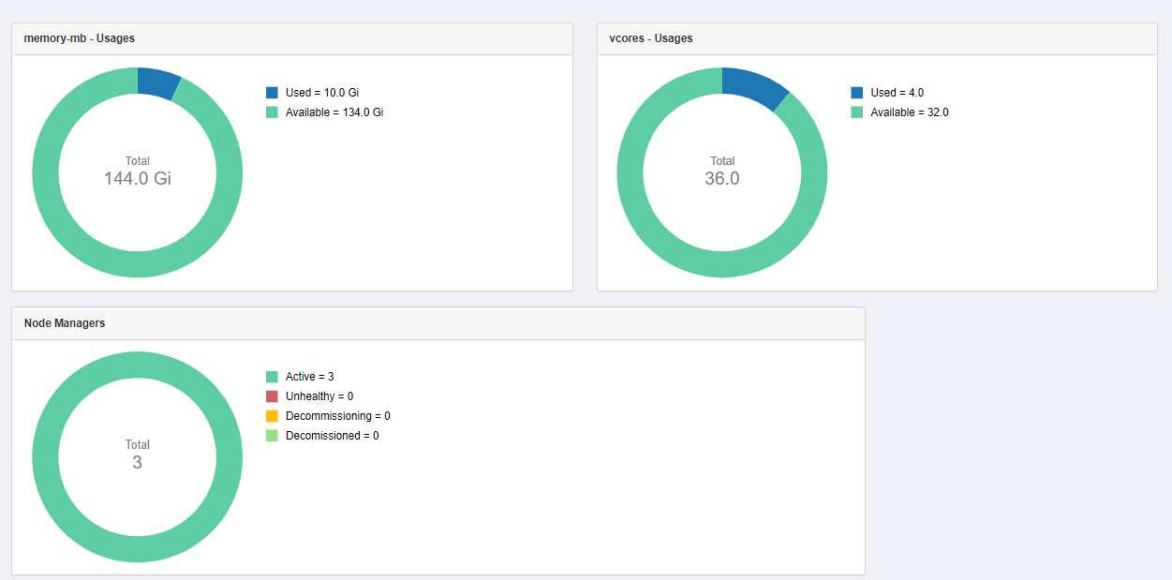
Podaci koji se nalaze u Parquet datotečnom formatu uglavnom ne dolaze u formatu jedne datoteke. Parquet datoteka na disku uglavnom ima onoliko koliko ima particija na koje podatke želimo podijeliti. Ukoliko želimo paralelno izvršavati obradu nad podatcima, s recimo pet paralelnih procesa, onda nam je korisno podatke podijeliti na pet ili deset manjih Parquet datoteka. Također, Parquet datoteku možemo particionirati s obzirom na neku vrijednost u podatcima, tako na slici 3.5 vidimo da smo podatke koje želimo pohraniti u Parquet datoteku grupirali po stavci država. Na taj način, određene će Parquet datoteke imati samo podatke koji se odnose na državu Nizozemsku.



Slika 3.5 Prikaz Parquet datoteka na disku

4. Analiza vremena obrade na dostupnom računalnom grozdu

Kao što je navedeno u uvodu, računalni grozd na kojem se obrada knjigovodstvenih podataka izvodi sastoji se od tri računala, na zasebnim IP adresama, od kojih svaki ima 12 jezgri procesora i 48 GB radne memorije. Računalni grozd ukupno sadržava računalne resurse od 36 jezgri procesora i 144 GB radne memorije, dok je od toga za korištenje slobodno 32 jezgre procesora i 134 GB radne memorije. Kako grozd izgleda i koliko ima računalne moći moguće je vidjeti na slikama 4.1 i 4.2.



Slika 4.1 Prikaz stanja grozda računala

Node Address	Node HTTP Address	Containers	Mem Used	Mem Available	VCores Used	VCores Available
plr011.lidm.gzaop.local:45454	plr011.lidm.gzaop.local:8042	1	3 GB	45 GB	1	11
plr010.lidm.gzaop.local:45454	plr010.lidm.gzaop.local:8042	1	3 GB	45 GB	1	11
plr012.lidm.gzaop.local:45454	plr012.lidm.gzaop.local:8042	2	4 GB	44 GB	2	10

Slika 4.2 Prikaz računala od kojih se sastoji računalni grozd

Na grozdu je instaliran HDFS na koji smo učitali 7 CSV datoteka ukupne veličine 7,4 GB. Prije same obrade, sve podatke čitamo iz CSV datoteka te prepisujemo u Parquet format i to na način da, ukoliko radimo obradu podataka s faktorom paralelizacije 5, onda podatke raspoređujemo u pet Parquet datoteka. Na taj ćemo način obradu podataka moći optimalno rasporediti na pet procesa. Programski kod koji je zadužen za pripremu podataka nazvan je PripremiPodatke. Programski kod koji je zadužen za obradu podataka, a izvršava se nakon PripremiPodatke, nazvan je StanjeDoprinosi. PripremiPodatke i StanjeDoprinosi su Sparkove aplikacije koje se pokreću na računalnom grozdu pomoću Airflowa. Airflow je platforma otvorenog koda za programsko kreiranje, zakazivanje i praćenje radnih tokova. Zadatci se na platformi Airflow definiraju pomoću DAG-a (engl. *directed acyclic graph*) odnosno računalnim kodom u Pythonu koji opisuje jednosmjerni aciklički postupak obrade podataka (bez programskih petlji). Isječak iz konfiguracije Airflow DAGa prikazan je na slici 4.3. Sparkove aplikacije se pokreću u načinu rada YARN-grozd (engl. *YARN-cluster*), što znači da se Sparkov pokretač, kontejner koji orkestrira cijelom Sparkovom aplikacijom odnosno svim izvršiteljima, nalazi unutar YARN-ovog upravitelja aplikacije. Aplikacije se pokreću s odabranim parametrima koji definiraju koliko Sparkova aplikacija sadrži izvršitelja, drugim riječima, koliko će YARN stvoriti kontejnera na grozdu koji će biti zaduženi za izvršavanje obrade. Svakog izvršitelja definiramo s brojem jezgri koje će sadržavati i veličinom radne memorije. Također, s obzirom koliko definiramo da će Sparkova aplikacija sadržavati izvršitelja odnosno jezgri, tako definiramo i faktor paralelizacije. Ukoliko imamo 5 izvršitelja i svakom od njih damo 5 jezgri, onda možemo definirati paralelizaciju kao faktor 25.

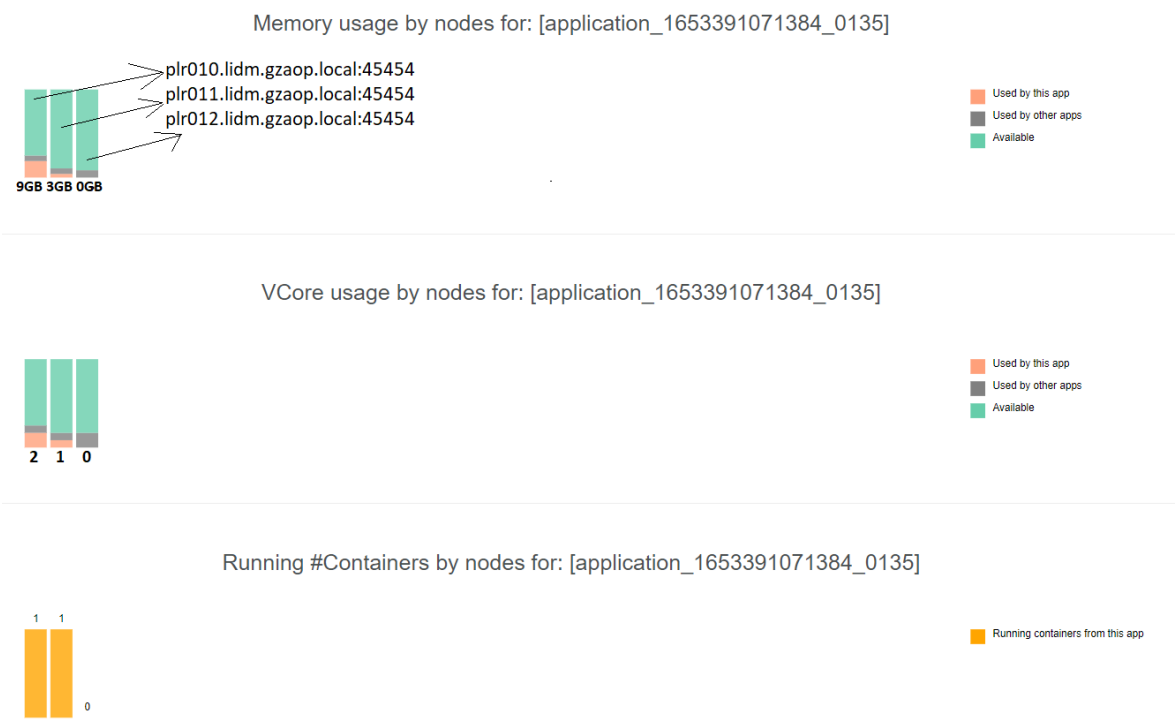
U slučaju da Sparkove aplikacije PripremiPodatke i StanjeDoprinosi pustimo s konfiguracijom koja je prikazana na slici 4.3, s jednim izvršiteljem s dvije jezgre i 8 GB radne memorije, resursi slični stolnom računalu, vrijeme koje će biti potrebno da se završi aplikacija PripremiPodatke iznosit će 50 minuta, dok će za aplikaciju StanjeDoprinosi biti potrebno 32 minute.

```
spark_confs_6 = {
    'spark.hadoop.fs.hdfs.impl.disable.cache':'true',
    'spark.eventLog.enabled':'true',
    'spark.cleaner.referenceTracking.blocking':'false',
    'spark.yarn.appMasterEnv.ENVIRONMENT':'eprod',
    'spark.yarn.executorEnv.ENVIRONMENT':'eprod',
    'spark.yarn.appMasterEnv.PROJECT_ROOT_DIR':'/projects/razv/djelokrug-fin/f04-knjigovodstvo/f04-load-formiranje-pocetnih-stanja',
    'spark.yarn.executorEnv.PROJECT_ROOT_DIR':'/projects/razv/djelokrug-fin/f04-knjigovodstvo/f04-load-formiranje-pocetnih-stanja',
    'spark.memory.fraction':'0.8',
    'spark.sql.shuffle.partitions':'2'
}

dag = DAG(dag_id='f04-load-razv-StanjaDoprinosi-fpazanin', default_args=default_args, catchup=False, schedule_interval=None)

sparkJob =  SparkSubmitOperator(
    task_id='f04-load-razv-StanjaDoprinosi-fpazanin-task',
    conn_id='spark_default',
    application=app_loc,
    java_class='hr.apisit.f04.PripremiPodatke',
    num_executors='1',
    executor_cores='2',
    executor_memory='8g',
    name='f04-load-razv-StanjaDop-fpazanin',
    keytab=spark_keytab,
    principal=spark_principal,
    conf=spark_confs_6,
    verbose=False,
    dag=dag
)
```

Slika 4.3 Konfiguracija Airflow DAG-a za Sparkovu aplikaciju

Sparkova aplikacija PripremiPodatke pokrenuta konfiguracijom prikazanoj na slici 4.3 na računalnom grozdu zauzela je računalne resurse u iznosu od 3 jezgre i 11 GB radne memorije. Slika 4.4 prikazuje korištenje resursa Sparkove aplikacije. Možemo vidjeti da je aplikacija pokrenula dva kontejnera, jedan od njih je upravitelj aplikacije koji je ujedno i Sparkov upravljač dok je drugi kontejner izvršitelj, onaj jedan koji smo i definirali u DAG-u na slici 4.3. Također, vidimo da upravitelj aplikacije, osim što je zauzeo jedan procesor, zauzeo je i 2 GB radne memorije. Još jedna zanimljivost je da izvršitelj, iako je definirano da zauzima samo 8 GB radne memorije, zauzima 9 GB, što je uobičajeno i nešto na što moramo računati kada konfiguriramo Sparkovu aplikaciju. Moramo računati da je moguće da će se zauzeti oko 10% više radne memorije nego što je to definirano.

Na slici 4.4 moguće je vidjeti kako izgleda zauzeće resursa na grozdu gore navedene aplikacije s definiranim jednim izvršiteljem s dva procesora i 8 GB radne memorije.



Slika 4.4 Prikaz zauzeća resursa Sparkove aplikacije

4.1. Maksimalno iskorištenje resursa računalnog grozda

U slučaju da želimo maksimalno iskoristiti raspoložive resurse na grozdu, učinili bi to sljedećim izračunom. Uzimamo u obzir da je optimalni broj jezgri po izvršitelju manji od 5, zbog propusnosti u datotečnom sustavu HDFS, naime prevelik broj procesora po procesu može dovesti do kontra učinka. Izračun je sljedeći:

Ukupni broj čvorova u našem računalnom grozdu = 3

Broj jezgri po čvoru = 10 (iako ih zapravo ima 12 po čvoru, 4 od ukupno 36 na računalnom grozdu nisu slobodne, pa ćemo računati da svaki čvor ima 10 slobodnih jezgri).

Količina radne memorije po čvoru = 44 GB (iako svaki čvor ima zapravo 48 GB, 10 GB od ukupno 144 GB na računalnom grozdu nije slobodno, pa ćemo računati da svaki čvor ima 44 GB slobodne radne memorije).

Odlučiti broj radnika (izvršitelja) →

Broj radnika po čvoru: 10 / 5 = 2 (5 je najbolji broj jezgri po radniku)

Ukupni broj radnika = 3 čvora * 2 radnika = 6 radnika

Ukupni broj slobodnih radnika = 5 (1 radnika ćemo ostaviti za funkciju voditelja odnosno za upravitelja aplikacijom

Odlučiti količinu radne memorije po radniku →

Memorija po radniku = 44 / 2 = 22 GB (2 radnika po čvoru)

Preljev memorije = 10% od 22 GB = 3 GB

Konačna memorija po radniku = 19 GB

Tako konačno imamo ukupno 5 radnika s 5 jezgri i 19 GB radne memorije.

Međutim, vrlo slične performanse grozda računala pri obradi podataka dobili bi i da smo naš grozd računala podijelili na 8 radnika s 3 jezgre i 12 GB radne memorije ili na 11 radnika s 2 jezgre i 9 GB radne memorije ili pak na 14 radnika s 2 jezgre i 7 GB radne memorije. U tablicama 4.1 i 4.2 možemo vidjeti koliko je bilo vrijeme potrebno za izvršavanje PripremiPodatke i StanjeDoprinosi za svaku od gore navedenih konfiguracija te koliko je svaka od navedenih kombinacija konačno zauzimala resursa računalnog grozda (vidjeti stupce MEMORIJA ZAUZETA NA GROZDU i JEZGRE ZAUZETE NA GROZDU).

	IZVRŠITELJI	JEZGRE PO IZVRŠITELJU	MEMORIJA PO IZVRŠITELJU	MEMORIJA ZAUZETA NA GROZDU	JEZGRE ZAUZETE NA GROZDU
1.	5+1AM	5	19	107	26
2.	8+1AM	3	12	114	25
3.	11+1AM	2	9	112	23
4.	14+1AM	2	7	114	29

Tablica 4.1 Prikaz zauzeća resursa na grozdu pri pokretanjem Sparkovih aplikacija različitih konfiguracija

	VRIJEME ZA PRIPREMIPODATKE	VRIJEME ZA OBRADIPODATKE
1.	11m 12s 189ms	7m 12s 255ms
2.	11m 83ms	7m 26s 6ms
3.	10m 15s 118ms	7m 46s 638ms
4.	9m 48s 219ms	7m 33s 544ms

Tablica 4.2 Prikaz vremena obrade podataka pri različitim konfiguracijama

Pretvarajući vrijeme u decimalni broj i uspoređujući sva vremena svih navedenih kombinacija konfiguracije obrade vidimo da ne postoji neko pravilo po kojem možemo reći da je neka konfiguracija bolja od druge. Tako možemo iz tablice 4.3 zaključiti da su u ovom slučaju sve konfiguracije legitimne.

	IZVRŠITELJI	PRIPREMIPODATKE	OBRADIPODATKE	GROZD %
1.	5	11,2	7,2	74.30556
2.	8	11	7,43	79.16667
3.	11	10,25	7,78	77.77778
4.	14	9,8	7,57	80.55556

Tablica 4.3 Tablica s rezultatima ispitivanja konfiguracija navedenih na slici 4.1

Napomena: stupac GROZD % kako u tablici 4.3 tako i u ostalim tablicama, označava maksimalnu vrijednost od dviju vrijednosti za programe: postotnog zauzeća jezgri i postotnog zauzeća radne memorije. Postotno zauzeće jezgri označava ukupni broj zauzetih jezgri podijeljen s ukupnim brojem jezgri na računalnom grozdu dok postotno zauzeće radne memorije označava ukupnu količinu zauzete radne memorije podijeljenu s ukupnom količinom radne memorije koja postoji na računalnom grozdu.

4.2. Analiza utjecaja broja procesora na obradu podataka

Iako u literaturi [9] možemo pročitati da je optimalni broj jezgri koji izvršitelju u Sparkovoj aplikaciji možemo zadati iznosi 5, odnosno broj manji od 5, u nastavku je proveden niz ispitnih slučajeva koji pokazuju da je ta hipoteza uistinu ispravna.

U svrhu ispitivanja, pokrenute su Sparkove aplikacije PripremiPodatke i StanjeDoprinosi s deset različitih konfiguracija. Fiksni parametri u konfiguraciji bili su broj radnika (3) i radna memorija po radniku (16 GB), dok je parametar broj jezgri po radniku mijenjan u rasponu od 1 do 10. Tako sukladno s promjenom broja jezgri po radniku, paralelizacija je rasla s 3, što je bio slučaj kada smo imali 1 jezgru po radniku, do čak 30, u slučaju kad smo imali 10 jezgri po radniku. Detaljni rezultati ispitivanja prikazani su u tablicama 4.4 i 4.5.

	JEZGRE PO IZVRŠITELJU	MEMORIJA PO IZVRŠITELJU	MEMORIJA ZAUZETA NA GROZDU	JEZGRE ZAUZETE NA GROZDU	GROZD %
1.	10	16	56	31	86.11111
2.	9	16	56	28	77.77778
3.	8	16	56	25	69.44444
4.	7	16	56	22	61.11111
5.	6	16	56	19	52.77778
6.	5	16	56	16	44.444447
7.	4	16	56	13	38.88889
8.	3	16	56	10	38.88889
9.	2	16	56	7	38.88889
10.	1	16	56	4	38.88890

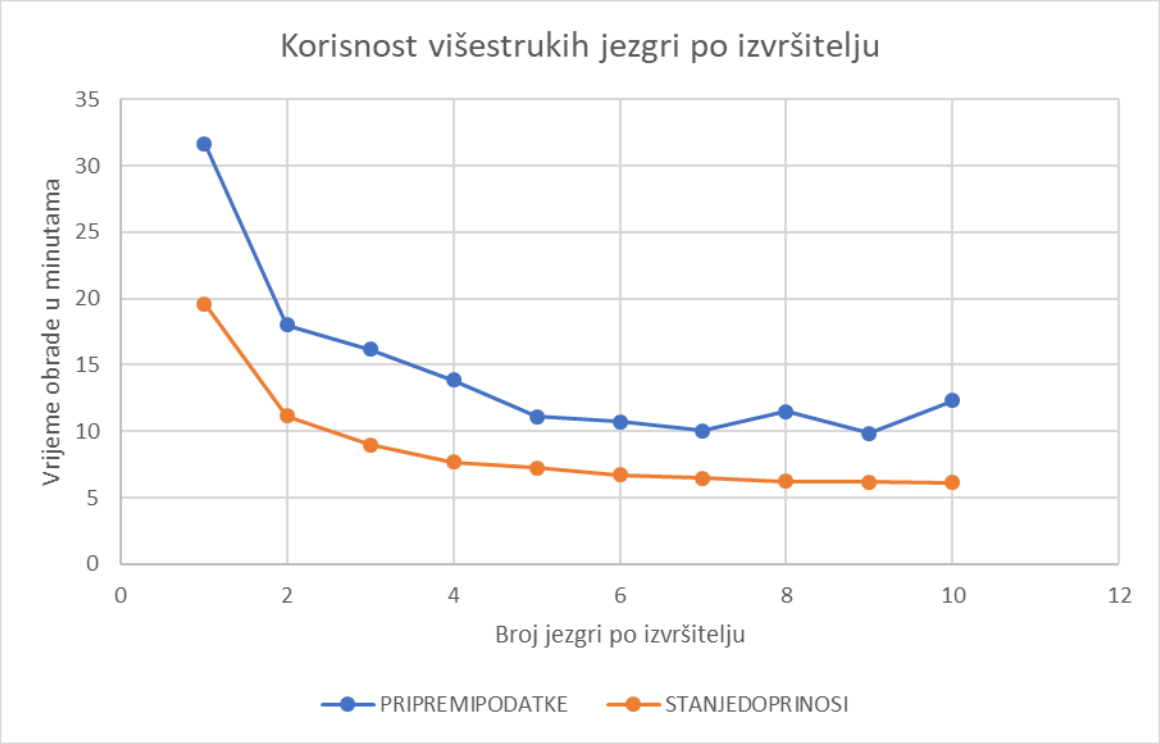
Tablica 4.4 Prikaz zauzeća resursa na grozdu pri pokretanjem Sparkovih aplikacija s konfiguracijom različitog broja jezgri po izvršitelju (broj izvršitelja je konstantan, 3)

Napomena, stupac GROZD % u retku 7 i na dalje označava postotak ukupne radne memorije na računalnom grozdu koji je zauzet pokretanjem Sparkovih aplikacija s navedenim parametrima. U redcima prije retka 7, zauzetost jezgri na grozdu je bila prevladavajuća nad zauzetosti radne memorije na grozdu. Pretvarajući vremena iz tablice 4.4 u decimalne brojke dobivamo rezultate zapisane u tablici 4.5. Brojke navedene u tablici 4.5 prikazane su grafički na slici 4.5.

	PARALELIZAM	JEZGRE PO IZVRŠITELJU	VRIJEME ZA PRIPREMIPODATKE	VRIJEME ZA STANJEDOPRINOSI
1.	30	10	12m 18s 777ms → 12,32	6m 8s 158ms → 6,13
2.	27	9	9m 50s 914ms → 9,83	6m 9s 805ms → 6,17
3.	24	8	11m 27s 982ms → 11,47	6m 15s 86ms → 6,25
4.	21	7	10m 3s 166ms → 10,05	6m 28s 756ms → 6,48
5.	18	6	10m 41s 654ms → 10,7	6m 43s 292ms → 6,72
6.	15	5	11m 5s 333ms → 11,08	7m 15s 422ms → 7,25
7.	12	4	13m 52s 349ms → 13,87	7m 40s 412ms → 7,67
8.	9	3	16m 10s 812ms → 16,18	8m 57s 536ms → 8,97
9.	6	2	18m 1s 97ms → 18,03	11m 9s 330ms → 11,15
10.	3	1	31m 38s 775ms → 31,65	19m 35s 345ms → 19,58

Tablica 4.5 Tablica s rezultatima ispitivanja konfiguracija navedenih na slici 4.4

Iz grafa slike 4.5 vidljivo je da smo povećavanjem broja jezgri do brojke od 5 jezgri po radniku imali značajna ubrzanja izvršavanja Sparkovih aplikacija PripremiPodatke i StanjeDoprinosi. Za konfiguracije od 5 i više jezgri po radniku ne možemo tvrditi da su isplative odnosno takve konfiguracije neopravdano troše više resursa računalnog grozda.



Slika 4.5 Grafički prikaz tablice 4.5, prikaz veze između broja jezgri radnika i vremena obrade podataka

4.3. Analiza utjecaja količine memorije na obradu podataka

Svaki Sparkov izvršitelj opisan je s brojem jezgri i količinom radne memorije koju ima na raspolaganju. Prethodnim ispitivanjem opisanim u poglavlju 4.2. uvjerili smo se da optimalni broj jezgri po izvršitelju iznosi 5. Sljedećim ispitivanjem provjerit ćemo koliko iznosi optimalna količina radne memorije po izvršitelju za izvršavanje Sparkove aplikacije StanjeDoprinosi. Tablice 4.6 i 4.7 prikazuju rezultate ispitivanja obrade podataka varijabilnom količinom radne memorije s fiksnim brojem radnika, 3, i fiksnim brojem jezgri po radniku, 5.

	JEZGRE PO IZVRŠITELJU	MEMORIJA PO IZVRŠITELJU	MEMORIJA ZAUZETA NA GROZDU	JEZGRE ZAUZETE NA GROZDU	GROZD %
1.	5	1	8	16	44.444447
2.	5	2	11	16	44.444447
3.	5	4	17	16	44.444447
4.	5	8	29	16	44.444447
5.	5	16	56	16	44.444447
6.	5	32	110	16	76.38889

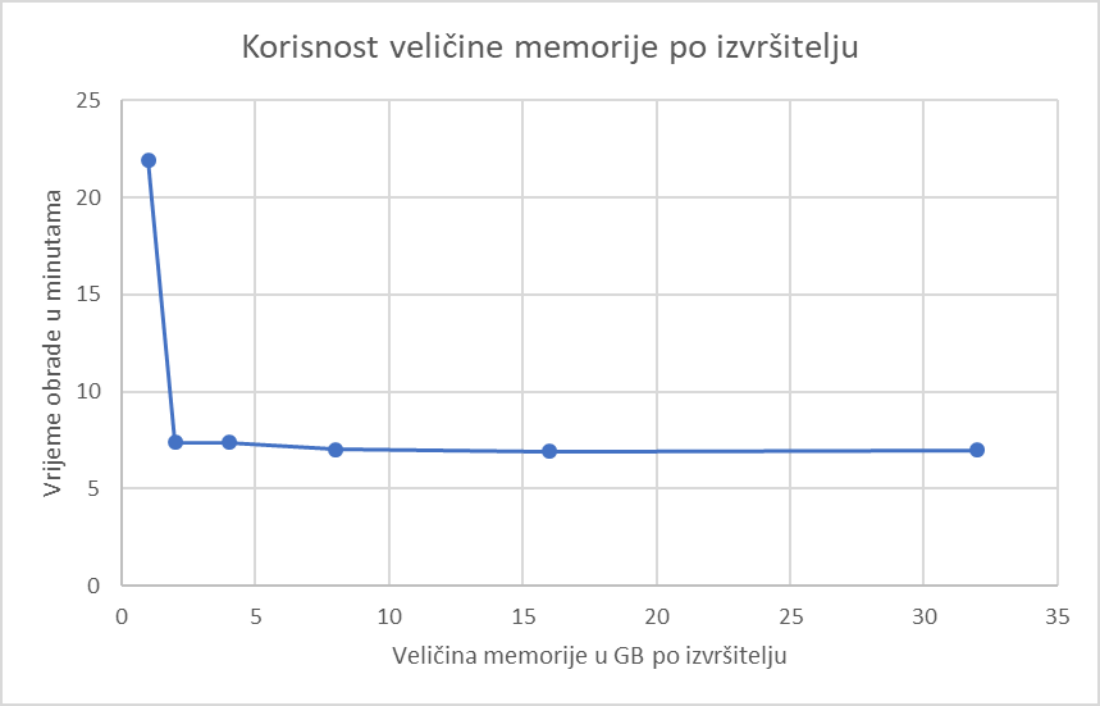
Tablica 4.6 Prikaz zauzeća resursa na grozdu pri pokretanjem Sparkovih aplikacija s konfiguracijom različite količine radne memorije po radniku (broj radnika je fiksan, 3)

Napomena, kolona GROZD % ukazuje na to da su konfiguracije od 1. do 5. postotno crpile više procesorske snage računalnog grozda naspram postotne količine radne memorije računalnog grozda. Rezultati vremena obrade podataka prikazani su u tablici 4.7. Budući da se svaka obrada izvršava s faktorom paralelizacije 15, bilo je potrebno samo jednom izvršiti algoritam PripremiPodatke koji je podatke podijelio na 15 Parquet datoteka. Na taj način učinili smo podatke pogodnima za obradu s 3 izvršitelja od po 5 jezgara svaki.

	PARALELIZAM	MEMORIJA PO IZVRŠITELJU	VRIJEME ZA STANJEDOPRINOSI
1.	15	1	21m 55s 438ms → 21,92
2.	15	2	7m 23s 273ms → 7,38
3.	15	4	7m 21s 944ms → 7,37
4.	15	8	7m 1s 717ms → 7,03
5.	15	16	6m 54s 851ms → 6,92
6.	15	32	6m 58s 564ms → 6,98

Tablica 4.7 Rezultati ispitivanja konfiguracija navedenih u tablici 4.6

Tablica 4.7 grafički je prikazana na slici 4.6. Vidimo da u našem primjeru paralelne obrade nešto više od 450 MB podataka raspoređenih u 15 Parquet datoteka, gdje koristimo 3 izvršitelja od po 5 jezgara, nema smisla konfigurirati obradu na način da izvršiteljima dodjeljujemo više od 2 GB radne memorije.



Slika 4.6 Prikaz kretanja vremena obrade podataka povećavanjem veličine radne memorije predane izvršiteljima

4.4. Analiza utjecaja paralelizacije na obradu podataka

Za kraj, provedeno je ispitivanje koje utvrđuje koliko je optimalni faktor paralelizacije kod pripreme i obrade podataka kojih imamo. Za podsjetnik, radi se o nešto više od 7 GB podataka, zapisanih u CSV formatu koje je potrebno prebaciti u Parquet format pa zatim obraditi. Računalni grozd koji imamo na raspolaganju sadrži nešto više od 30 jezgri i 130 GB radne memorije. Imajući na umu da je optimalni broj jezgri po radniku 5 te da je 16 GB radne memorije više nego dovoljno dodijeliti izvršitelju da radna memorija ne bude usko grlo kod izvođenja obrade (u prethodnom poglavlju uvjerili smo se da bi zapravo 2 GB bilo sasvim dovoljno), mijenjali smo broj radnika i time faktor paralelizacije te dobili rezultate prikazane u tablici 4.8.

	IZVRŠITELJI	JEZGRE PO IZVRŠITELJU	MEMORIJA PO IZVRŠITELJU	MEMORIJA ZAUZETA NA GROZDU	JEZGRE ZAUZETE NA GROZDU	GROZD %
1.	1	5	16	20	6	16.666668
2.	2	5	16	38	11	30.555555
3.	3	5	16	56	16	44.444447
4.	4	5	16	74	21	58.333332
5.	5	5	16	92	26	72.22222
6.	6	5	16	110	31	86.11111

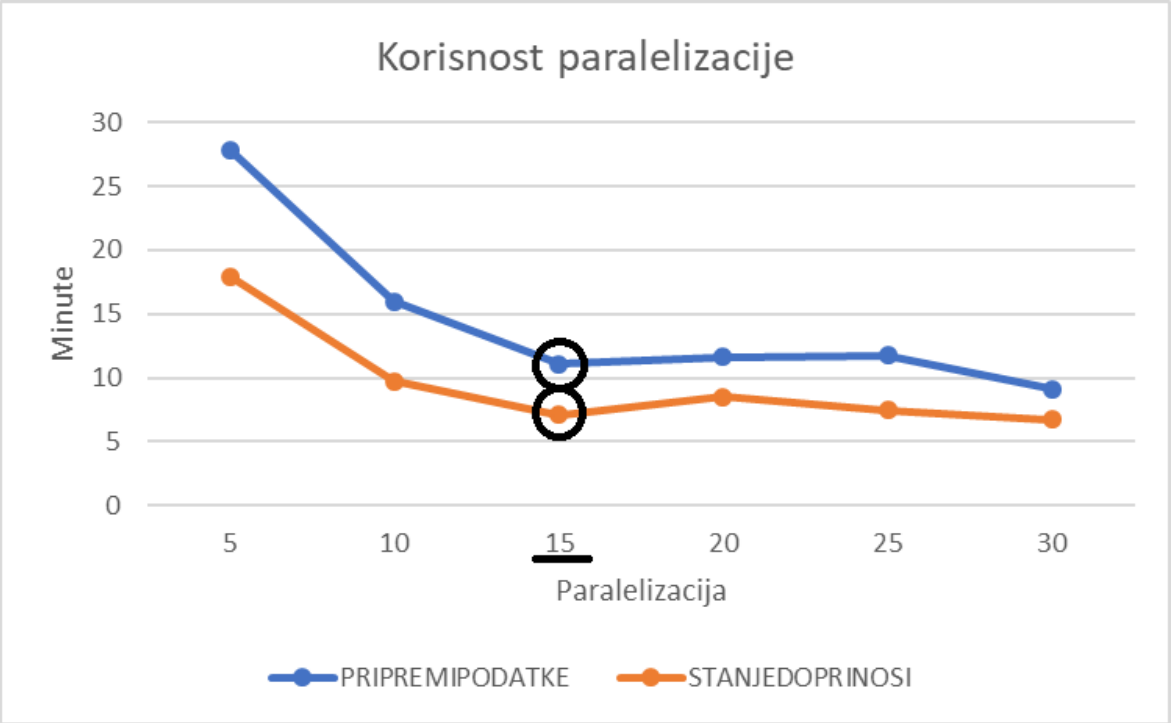
Tablica 4.8 Prikaz zauzeća resursa na grozdu pri pokretanjem Sparkovih aplikacija s različitim brojem izvršitelja

Vrijeme potrebno za izvršavanje Sparkovih aplikacija PripremiPodatke i StanjeDoprinosi, konfiguracijom prikazanoj u tablici 4.8, navedeno je u tablici 4.9.

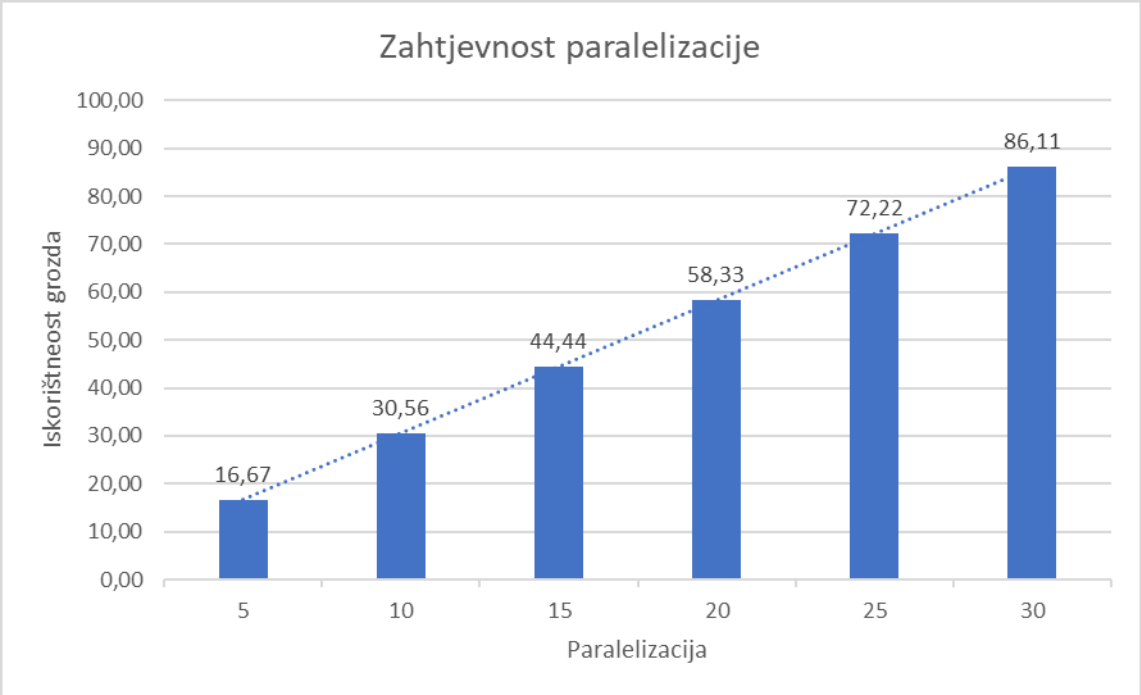
	PARALELIZAM	VRIJEME ZA PRIPREMIPODATKE		VRIJEME ZASTANJEDOPRINOSI		GROZD %
1.	5	27m 51s 466ms →	27,85	17m 56s 972ms →	17,95	16,67
2.	10	16m 121ms →	16	9m 42s 223ms →	9,7	30,56
3.	15	11m 4s 625ms →	11,07	7m 6s 749ms →	7,12	44,44
4.	20	11m 38s 682ms →	11,63	8m 28s 19ms →	8,47	58,33
5.	25	11m 43s 546ms →	11,72	7m 27s 238ms →	7,45	72,22
6.	30	9m 5s 837ms →	9,1	6m 44s 242ms →	6,73	86,11

Tablica 4.9 Rezultati ispitivanja konfiguracija navedenih u tablici 4.8

Grafički prikaz rezultata prikazan je na slici 4.7. Vidimo da smo optimalnu kombinaciju između vremena obrade i zauzeća resursa računalnog grozda imali u slučaju faktora paralelizacije iznosa 15, gdje smo Sparkove aplikacije pokrenuli s tri radnika te svakom pridijelili 5 procesorskih jezgri. Na slici 4.8 prikazan je razmjer zauzeća resursa računalnog grozda ovisno o faktoru paralelizacije odnosno ovisno o konfiguraciji koje su navedene u tablicama 4.8 odnosno 4.9. Zaključujemo da smo prilikom faktora paralelizacije 15, odnosno stvaranjem 3 izvršitelja (kontejnera) od po 5 jezgri i 16 GB radne memorije zauzeli nešto manje od 44% ukupne količine računalnih resursa našeg grozda računala.



Slika 4.7 Grafički prikaz tablice 4.9, prikaz veze između faktora paralelizacije i vremena izvršavanja Sparkovih aplikacija



Slika 4.8 Prikaz postotka ukupne količine resursa računalnog čvora koji se zauzima ovisno o faktoru paralelizacije, konfiguracijama prikazanima u tablici 4.8

Zaključak

Često možemo čuti uzrečicu da su podatci zlato 21. stoljeća. Moje je mišljenje, u prenesenom značenju, da su podaci samo običan kamen koji bez obrade ostaje samo kamen. Obradom podaci postaju zlato. Obradom iz podataka stvaramo zaključke koji poslovanje mogu usmjeriti u pravom smjeru, u smjeru efikasnijeg poslovanja. U ovom radu prikazan je stvarni primjer iz industrije gdje sam knjigovodstveno poslovanje unaprijedio koristeći znanja iz područja računarske znanosti. Važnost računarstva, u vremenu kada je obrada podataka od iznimne važnosti za poslovanje bilo koje industrije, je očigledna. Način na koji ćemo podatke obraditi stvara razliku. Ispitivanjem performansi obrade podatka na računalnom grozdu uvjerio sam se da veće ne znači nužno i bolje. Zauzećem nepotrebnih količina resursa nećemo postići ništa osim kontraefekta, obradu ćemo učiniti skupljom, a u isto vrijeme i sporijom. Zato nam je kao inženjerima računarstva posao pokušati optimalno iskoristi resurse koji su nam dani za zadani posao. Tek kada postoji donekle dobar omjer uloženih resursa i vremena obrade, posao se smatra uspješno završenim. Smanjenjem trajanja obrade podataka za 5 puta koristeći nekoliko osrednjih računala umreženih u grozd i paradigmu paralelne obrade, postignuto je uspješno unaprjeđenje poslovanja tvrtke APIS IT. Iako su ovi rezultati zadovoljavajući, oni naravno ostavljaju mjesta za napredak i svaka dodatna analiza je dobrodošla. Kako bi se obrada izvodila da nisu u pitanju gigabajti podataka, već terabajti ili petabajti te kako bi se pristupilo obradi nestrukturiranih podataka može biti tema rada koji bi se nadovezao na ovaj. Također, zanimljiv slučaj za analizu bio bi slučaj u kojem želimo da se obrada obavlja u stvarnom vremenu, kako podatci pristižu. Obrada u stvarnom vremenu nosi puno izazova, ali danas mnoge tvrtke biraju takav pristup obradi podataka u želji što veće konkurentnosti na tržištu.

Literatura

[1] Big Data Technology, (2019., lipanj). Poveznica: https://ec.europa.eu/croatia/basic/everything_you_need_to_know_about_big_data_technology_hr; pristupljeno 1. lipnja 2019.

[2] Wikipedia, Apache Hadoop, (2019., lipanj). Poveznica: https://en.wikipedia.org/wiki/Apache_Hadoop; pristupljeno 1. lipnja 2022.

[3] Apache Hadoop, (2022., lipanj). Poveznica: Apache Hadoop 3.3.3 – Apache Hadoop YARN; pristupljeno 14. lipnja 2022.

[4] LimeGuru, Learn Hadoop YARN Resource Manager In 20 Minutes | Yarn Tutorial For Beginners, (2022., lipanj). Poveznica: (2) Learn Hadoop YARN Resource Manager In 20 Minutes | Yarn Tutorial For Beginners - YouTube; pristupljeno 1. lipnja 2022.

[5] Wikipedia, Apache Spark, (2019., lipanj). Poveznica: https://en.wikipedia.org/wiki/Apache_Spark; pristupljeno 1.lipnja 2022.

[6] Databricks, The Parquet Format and Performance Optimization Opportunities Boudewijn Braams (Databricks), (2022., lipanj). Poveznica: The Parquet Format and Performance Optimization Opportunities Boudewijn Braams (Databricks) - YouTube; pristupljeno 1. lipnja 2022.

[7] Riz Ang, Lambda Architecture tutorial under 10 minutes, (2022., lipanj) Poveznica: (1) Lambda Architecture tutorial under 10 minutes - YouTube; pristupljeno 1. lipnja 2022.

[8] Riz Ang, What is Apache Parquet file?, (2022., lipanj). Poveznica: What is Apache Parquet file? - YouTube; pristupljeno 1. lipnja 2022.

[9] DataEngineer, Apache Spark: How to decide number of Executor & Memory per Executor?, (2022., lipanj). Poveznica: https://dataengineer1.blogspot.com/2021/04/apache-spark-how-to-decide-number-of.html; pristupljeno, 1. lipnja 2022.

Sažetak

Naziv: Upotreba tehnologija za raspodijeljenu obradu velikih količina podataka u knjigovodstvu

Sažetak: U tvrtki APIS IT zadatak je bilo smanjiti vrijeme potrebno za knjigovodstvenu obradu podataka. Podatci su bili veličine nekoliko gigabajta te se obrada trebala prevesti na paralelnu paradigmu i zatim pokrenuti na računalnom grozdu. Grozd je imao resurse nekolicine osrednjih stolnih računala. Proučavanjem tehnologija velikih količina podataka kao što su HDFS, YARN, Spark i Parquet zadatak je uspješno izvršen te se upotrebom niti polovice resursa dostupnog računalnog grozda smanjilo vrijeme obrade za 5 puta. U radu su opisane sve gore navedene tehnologije te su detaljno prikazani rezultati ispitivanja raznih konfiguracija obrade podataka na računalnom grozdu. Neke od konfiguracija odabrane su kao optimalne, a taj je izbor obrazložen u radu.

Ključne riječi: veliki podaci, knjigovodstvo, računalni grozd, Hadoop, YARN, Spark, Parquet, testiranje performansi, paralelizacija

Summary

Title: Big Data Technologies in Bookkeeping

Summary: At APIS IT, the task was to reduce the time required for bookkeeping data processing. The data was a few gigabytes in size, so processing had to be translated to a parallel paradigm and then run on a computer cluster. The cluster had the resources of a few average desktop computers. By studying big data technologies such as HDFS, YARN, Spark and Parquet, the task was completed successfully. The solution included threading of half the resources of the available computer cluster, which reduced the processing time 5 times. The paper describes all the above technologies and presents in detail the results of various data processing configurations testing on the computer cluster. Some of the configurations were chosen as optimal and the choice was explained.

Keywords: big data, bookkeeping, computer cluster, Hadoop, YARN, Spark, Parquet, performance testing, parallelization