SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6202

Usporedba različitih inačica platforme Apache Spark

Filip Herceg

Zagreb, srpanj 2019.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 13. ožujka 2019.

ZAVRŠNI ZADATAK br. 6202

Pristupnik: **Filip Herceg (0036502567)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Usporedba različitih inačica platforme Apache Spark**

Opis zadatka:

Vaš zadatak je detaljno proučiti i opisati razlike između inačica 1.x i 2.x platforme Apache Spark. Nakon toga u obje inačice tehnologije implementirajte jednostavne aplikacije za obradu statičkih podataka te toka podataka kojima ćete demonstrirati prethodno opisane razlike. Razvijene aplikacije usporedite na stvarnom računalnom grozdu u laboratoriju.
Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Zadatak uručen pristupniku: 15. ožujka 2019.
Rok za predaju rada: 14. lipnja 2019.

Mentor:
Pripužić
Izv. prof. dr. sc. Krešimir Pripužić

Djelovođa:
Izv. prof. dr. sc. Tomislav Hrkać

Predsjednik odbora za završni rad modula:
Marko Čupić
Doc. dr. sc. Marko Čupić

SADRŽAJ

1. Uvod

1.1. Uvod

U proteklih sto godina tehnologija je iznimno napredovala. To je uzrokovalo potrebu za obradom sve većih količina podataka. Ipak tehnologija ne napreduje dovoljno brzo koliko je podataka potrebno obraditi na jednom računalu kako bi se pružila dovoljno kvalitetna usluga stoga su se počeli razvijati sustavi za raspodijeljenu obradu podataka kao što je platforma *Apache Spark*.

Razvoj platforme *Apache Spark* potaknut je nedostatcima i ograničenjima već postojećih tehnologija kao što je iznimno popularan Hadoop koji nudi mogućnost obrade podataka na principu map i reduce operacija. Osim ograničenih mogućnosti obrade podataka jedan od glavnih nedostataka bila je i brzina izvođenja. Hadoop prije svake operacije čita podatke iz stalne memorije te nakon svake operacije zapisuje rezultate izvođenja u stalnu memoriju. Platforma *Apache Spark* ubrzala je vrijeme izvođenja čak do sto puta smanjivanjem broja čitanja iz memorije i pisanja u memoriju tako što podatke koji se obrađuju učita u radnu memoriju te ih ne sprema u stalnu memoriju sve dok se ne završi obrada podataka, a jedini trošak vremena osim na samu obradu je prijenos podataka između Izvršitelja nakon obrade akcije. Još jedan razlog zašto je *Apache* Spark brz i pouzdan je *DAG* (engl. *Directed Acyclic Graph*), koji omogućava optimizaciju izvođenja napisanog koda, koji ću detaljnije obraditi kasnije u ovom radu.

Pisanje aplikacija platforme *Apache Spark* moguće je u više jezika, a to su *Scala*, *Java*, *Python* i drugi. Provođenjem raznih optimizacija moguće je postići jednaku brzinu izvođenja bez obzira u kojem jeziku je program napisan.

Ono što programeri žele je lakše pisanje koda, brže izvođenje, stabilnost i kompatibilnost između verzija. U nastavku ću objasniti i pokazati na primjerima usporedbu verzija 1.x.x i 2.x.x platforme *Apache Spark*.

2. Osnovni koncepti platforme Apache Spark

2.1. Platforma *Apache Spark*

Platforma *Apache Spark* namijenjena je za raspodijeljenu obradu velikih skupova podataka. Platforma Apache Spark pruža programiranje računalnog grozda implicitnim paralelizmom te automatski oporavak od pogreške što ju čini jednostavnom za uporabu, a pouzdanom. [1] [2]

Primarni jezik za pisanje aplikacija platforme Apache Spark je *Scala*, ali se također mogu pisati i aplikacije u *Javi*, *Pythonu* i jeziku *R*. Platformu Apache Spark moguće je pokrenuti pomoću upravitelja resursima *YARN* i *Apache Mesos*, samostalno ili u oblaku.

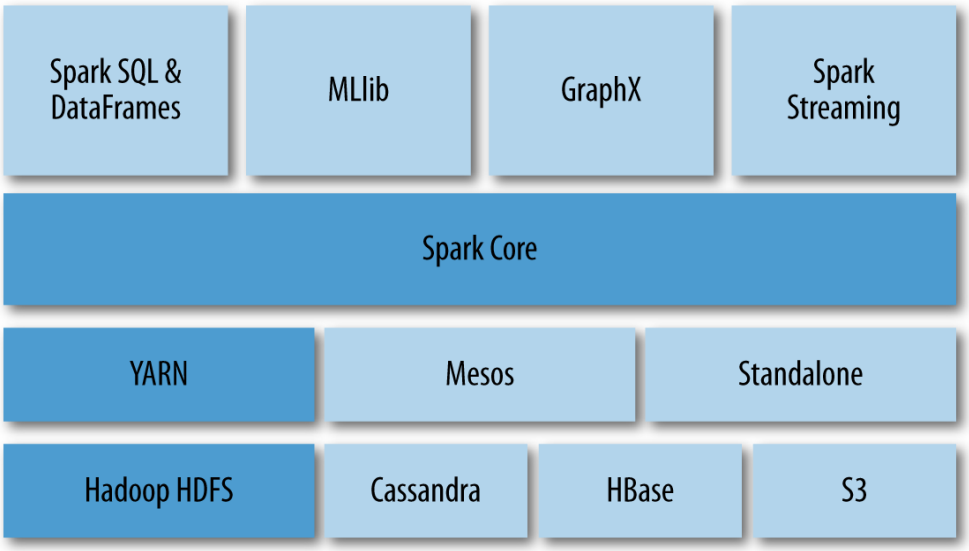
Podatci se mogu pohraniti lokalno prilikom lokalnog pokretanja ili se može koristit jedan od sustava za raspodijeljenu pohranu velikih količina podataka kao što su: *HDFS*, *HBase*, *Hive*, *Cassandra* ili u oblaku koristeci *Amazon S3* ili *Google Dataproc*.

Složaj platforme Apache Spark sastoji se od 5 dijelova:

- Spark Core - osnovna komponenta koja služi za raspodjelu poslova i upravljanje resursima
- Spark SQL - komponenta koja za rad sa strukturiranim podatcima, omogućava obradu podataka pomoću SQL interpretera
- Spark Streaming - komponenta za rad s tokovima podataka
- Spark MLlib - komponenta koja nudi mnoštvo već implementiranih algoritama strojnog učenja
- Spark GraphX - komponenta za rad s velikim grafovima

Na slici 2.1 možemo vidjeti arhitekturu aplikacije platforme Apache Spark. Najniži sloj predstavlja sustave za pohranu podataka. Sloj iznad podatkovnog sloja predstavlja upravitelje resursima koji su zaduženi za komunikaciju između Upravitelja

(engl. *Driver*) i Izvršitelja (engl. *Executora*). Najbitniji sloj je *Spark Core* koji raspodijeljuje posao na zadatke, koordinira raspoređivanje posla na *Executore* te vrši raspoređivanje poslova na izvodače temeljeno na prostornoj lokalnosti. U najvišem sloju arhitekture platforme Apache Spark nalaze se komponente koje pružaju API visoke razine za jednostavan razvoj aplikacija i brzu obradu podataka. [2]

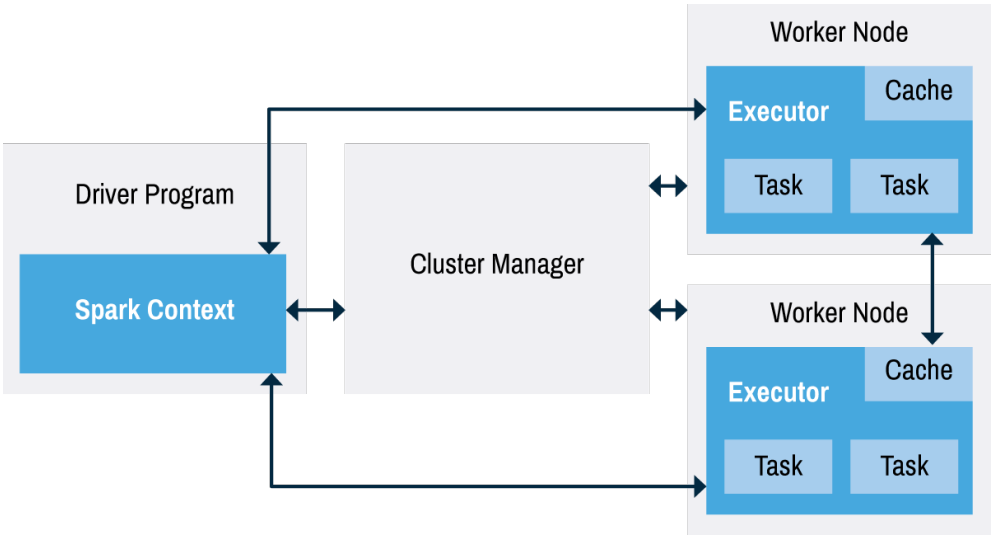


Slika 2.1: Arhitektura aplikacije platforme Apache Spark, izvor: [3]

Slika 2.2 prikazuje organizaciju računalnog grozda platforme Apache Spark. Upravitelj sadrži kontekst aplikacije, odnosno informacije o dostupnim resursima, poslovima koje je potrebno obaviti te informacije o izvorima podataka. Upravitelj grozdom (engl. *Cluster manager*) komunicira sa Spark Upraviteljem te pregovara o dostupnim resursima koje može dodijeliti Izvršitelju. Radni čvorovi (engl. *Worker Node*) povezani su na Upravitelja grozdom s kojim izmjenjuju informacije o dostupnosti vlastitih izvršitelja koji vrše obradu podataka. [4]

2.1.1. Upravitelj

Upravitelj je glavni dio aplikacije platforme Apache Spark. U njemu se pokreće kod iz kojeg Upravitelj kreira *SparkContext* koji je zadužen za koordinaciju obrade podataka. SparkContext (ili *SparkSession*) upravlja zadatcima toka posla, raspoređuje zadatke i dostupan je cijelo vrijeme dok se aplikacija izvodi (tj. Program upravljačkog programa mora slušati i prihvaćati dolazne veze od svojih izvršitelja tijekom cijelog njegovog vijeka trajanja. Kao takav, upravljački program mora biti dostupan mreži na koju su spojeni radnički čvorovi). [6]



Slika 2.2: Prikaz komunikacije komponenti aplikacije platforme Apache Spark, izvor: [5]

2.1.2. Izvršitelj

Izvršitelj je proces koji se nalazi unutar radnog čvora. Jedan radni čvor može sadržavati jedan ili više executora, ali se preporuča korištenje samo jednog executora na jednom radnom čvoru kako bi se na svakom čvoru izvodila jedna instanca *JVM*-a. Izvršitelj od Upravljača dobiva zadatke koje treba obaviti te rezultate obrade šalje Upravljaču. [6]

2.2. Komponente platforme Apache Spark

2.2.1. Spark Core

Spark Core je temelj cjelokupnog projekta. Pruža osnovne funkcionalnosti potrebne za raspodijeljenu obradu a to su:

- rezervacija resursa
- podjela posla na zadatke
- raspodijeljivanje zadataka
- osnovne ulazno izlazne operacije

Spark Core API izložen je putem sučelja programskih jezika Scala, Java, Python i R. Operacije kao što su mapiranje, filtriranje ili reduciranje pozivane nad RDD objektima u main metodi Driver programa se paralelno izvršavaju na računalnom grozdu. RDD nije moguće promijeniti, nego se stvara novi RDD te se koraci pretvorbe prate kako bi se podatci mogli reproducirati u slucaju gubitka podataka.

Uz RDD postoji još jedan Spark Core tip podataka, a to je Akumulator (engl. *Accumulator*). Obzirom da promjena vrijednosti lokalnih varijabli na executorima ne utjece na lokalne varijable drivera potreban je posebni tip podataka čija se vrijednost instance Upravitelja ažurira prilikom korištenja na Izvršiteljima. [1]

2.2.2. Spark SQL

Spark SQL podržava raspodijeljenu *in-memory* obradu podataka. Nestrukturirane RDD objekte možemo pretvoriti u strukturirane DataFrame objekte pomoću Spark SQL sučelja. Strukturirane podatke je lakše obrađivati jer imamo više informacija o samim podatcima stoga je moguće vršiti razne optimizacije prilikom obrade. Glavni aspekt Spark SQL-a je podržavanje izvršavanja SQL upita nad podatcima. Također Spark SQL se može koristiti za dohvat podatka iz SQL baze (npr. Hive) koji se učitaju u memoriju radi brže obrade.

Rezultati korištenja Spark SQL sučelja nisu nestrukturirani RDD objekti, nego strukturirani DataFrame/Dataset objekti nad kojima se vrši daljnja obrada.

Spark SQL osigurava apstrakciju DataFrame-a u Pythonu, Javi i Scali. To pojednostavljuje rad sa strukturiranim skupovima podataka. U Sparku, SQL DataFrameovi su isti kao i tablice u relacijskoj bazi podataka stoga je moguće izvršavanje SQL upita nad DataFrame objektima kao i pretvaranje DataFrame objekata u tablicu i obrnuto.

Spark SQL može čitati i pisati podatke u različitim strukturiranim formatima, kao što su JSON, Hive tablice i parquet.[1]

2.2.3. Spark Streaming

Spark Streaming je skalabilan sustav za obradu toka podataka koji ima automatski oporavak greške i koji izvorno podržava i obradu malih serija i obradu kontinuiranog toka podataka. Spark Streaming je komponenta koja omogućuje obradu podatka u stvarnom vremenu iz različitih izvora uključujući *Kafku*, *Flume* i *Amazon Kinesis*. Ovi obrađeni podatci mogu se spremati u lokalnu memoriju ili baze podataka. Njegova ključna apstrakcija je diskretizirani tok ili, ukratko, *DStream*, koji predstavlja tok podataka podijeljenih u male serije. DStream se temelji na RDD-ovima, Sparkovoj apstrakciji temeljnih podataka. To omogućuje neprimjetnu integraciju Spark Streaminga s drugim komponentama platforme Apache Spark kao što su MLlib i Spark SQL. [1]

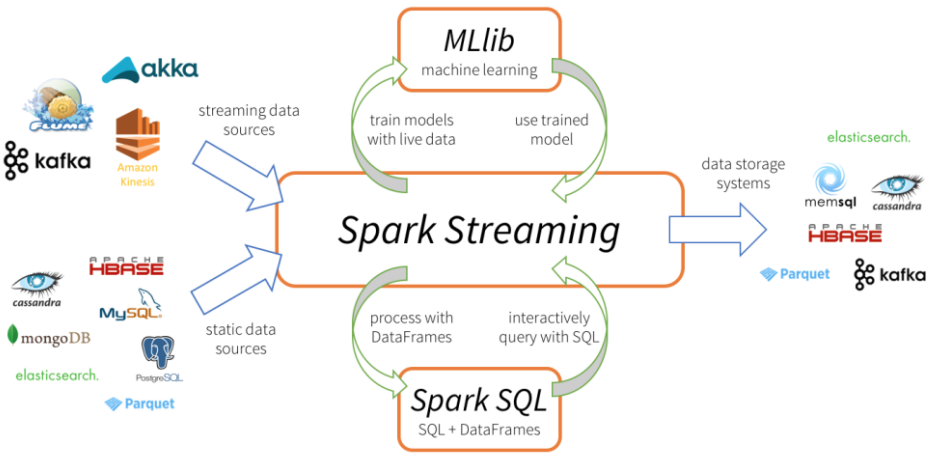
Na slici 2.3 prikazan je primjer obrade toka podataka pomoću malih serija. Tok podataka se diskretizira te se podatci periodički obrađuju prema zadanom vremenskom intervalu obrade.

Slika 2.3: Prikaz obrade toka na principu malih serija, izvor: [7]

Slika 2.4 prikazuje arhitekturu sustava koji koristi Spark Streaming. S lijeve strane su prikazani izvori iz kojih se podatci preuzimaju tokovima podataka i statički izvori podataka. U sredini je prikazana mogućnost koristenja Spark MLlib komponente za treniranje modela korištenjem dolazećih podataka iz tokova ili za postavljanje upita na već istreniranim modelima. Također je prikazana i mogućnost obrađivanja upita nad bazama podataka i spremanja podataka pomoću Spark SQL komponente. Konačno s desne strane su prikazane mogućnosti spremanja podataka u trajne baze podataka.

2.2.4. Spark MLlib

Spark MLlib je Spark komponenta za distribuirano treniranje modela strojnog učenja. Zasnovan na Spark arhitekturi na temelju distribuirane memorije, Spark MLlib je čak



Slika 2.4: Prikaz arhitekture sustava koji koristi Spark Streaming, izvor: [8]

devet puta brži od implementacije temeljene na disku koju koristi Apache Mahout. Spark MLlib sučelje nudi implementirane metode čestih algoritama strojnog učenja i statističke analize. Spark MLlib moguće je koristiti uz Spark SQL i Spark Streaming. [1]

2.2.5. Spark GraphX

GraphX je Spark komponenta za distribuiranu obradu grafova. Temelji se na RDD-ovima, koji su nepromjenjivi, stoga su grafovi koje Spark GraphX obrađuje nepromjenjivi, pa je GraphX neprikladan za grafove koje je potrebno ažurirati, a kamoli na transakcijski način kao što je baza podataka temeljena na grafovima. GraphX nudi dva zasebna API-ja za implementaciju masivno paralelnih algoritama (kao što je PageRank): Pregelova apstrakcija i općenitiji API za MapReduce stil. Za razliku od svog prethodnika Bagela, koji je formalno odbačen u Sparku 1.6, GraphX ima punu podršku za grafikone svojstava (grafikoni gdje se svojstva mogu pridružiti rubovima i vrhovima).

GraphX se može promatrati kao Spark in-memory verzija Apache Giraph koji koristi MapReduce koncept temeljen na Hadoop disku. [1]

2.3. Načini izvođenja aplikacije na platformi Apache Spark

2.3.1. Lokalni način izvođenja

Aplikaciju platforme Apache Spark moguće je pokrenuti lokalno na jednom računalu. Ako se aplikacije pokrene lokalno tada se izvodi samo jedan proces koji je ujedno i Upravljač i Izvršitelj. Ovakav način pokretanja najčešće se koristi prilikom razvoja aplikacije te se najčešće koristi lokalna memorija sustava.

2.3.2. Pseudoraspodijeljeni način izvođenja

Upravitelj računalnog grozda Spark Standalone jednostavan je upravitelj računalnog grozda dostupan kao dio distribucije Spark. Ima visoku raspoloživost za Upravljački čvor, otporan je na neuspjehe radnih cvorova, ima mogućnosti za upravljanje resursima ovisno o konfiguraciji aplikacije i može se izvoditi uz postojeću implementaciju Hadoop-a i pristupati podatcima HDFS-a (engl. *Hadoop Distributed File System*). Spark distribucija uključuje skripte koje olakšavaju postavljanje na lokalnoj razini ili u oblaku na Amazon EC2.

Samostalni Spark koristi jednostavan *FIFO Scheduler* za aplikacije. Prema zadanoj konfiguraciji, svaka aplikacija koristi sve dostupne čvorove u računalnom grozdu. Broj čvorova može biti ograničen po aplikaciji, po korisniku ili globalno. Ostali resursi, kao što su memorija, procesori itd., mogu se kontrolirati putem objekta SparkConf u aplikaciji.

Spark Standalone upravitelj podržava automatski oporavak Upravljačkog čvora pomoću ZooKeeper sustava. Također podržava ručni oporavak pomoću datotečnog sustava. Računalni grozd je otporan na neuspjehe radnih čvorova bez obzira na to je li omogućen oporavak Upravljačkog čvora.

Spark Standalone podržava provjeru autentičnosti putem zajedničke tajne sa svim upraviteljima računalnog grozda. Pokretanje u Standalone načinu zahtijeva da korisnik konfigurira svaki od čvorova sa zajedničkom tajnom, podatci se mogu kodirati pomoću SSL-a za komunikacijske protokole.

Svaka aplikacija platforme Apache Spark ima Web sučelje za nadzor aplikacije. Web sučelje prikazuje informacije o zadatcima koji se izvode u aplikaciji, izvršiteljima i korištenju pohrane. Osim toga, Sparkov samostalni upravitelj računalnog grozda ima web sučelje za pregled statistike računalnog grozda i poslova, kao i detaljnog izlaza za

svaki posao.[9]

2.3.3. Raspodijeljeni način izvođenja na platformi Hadoop YARN

Hadoop YARN je distribuirani radni okvir za raspoređivanje poslova i upravljanje resursima računalnog grozda, ima visoku dostupnost za upravljačke programe i izvršitelje, podršku za *Docker* kontejnere, ali samo u nezaštićenom načinu rada.

Apache Hadoop YARN ima *ResourceManager* s dva dijela, Scheduler i *ApplicationsManager*. Scheduler je komponenta koja se može priključiti. Navedene su dvije implementacije, *CapacityScheduler*, koristan u raćunalnom grozdu koji dijeli više organizacija, i *FairScheduler*, koji osigurava da sve aplikacije u prosjeku dobiju jednak broj resursa. Oba raspoređivača stavljaju aplikacije u redove i svaki red dobiva resurse koji se dijele jednako među redovima. Unutar reda, resursi se dijele između aplikacija. ApplicationsManager je odgovoran za prihvaćanje zahtjeva za obrađivanje poslova i pokretanje aplikacije *ApplicationMaster*. U ovom slučaju, ApplicationsMaster je aplikacija Spark. U aplikaciji Spark, resursi su specificirani u SparkConf objektu aplikacije.

Apache Hadoop YARN podržava ručni oporavak pomoću naredbenog retka i podržava automatski oporavak pomoću Zookeeper komponente koja je ugrađena u ResourceManager stoga, za razliku od Mesos i Standalone upravitelja, nema potrebe za pokretanjem zasebnog ZooKeeper kontrolera. ZooKeeper se koristi samo za snimanje stanja resursa za upravljanje resursima.

Hadoop YARN ima sigurnost za provjeru autentičnosti, ovlaštenje na razini usluge, provjeru autentičnosti za web konzole i povjerljivost podataka. Provjera autentičnosti koristi *Kerberos* za provjeru autentičnosti svakog korisnika i servisa. Pristup Hadoop uslugama može se precizno kontrolirati putem popisa za kontrolu pristupa.

Hadoop YARN ima web sučelje za ResourceManager i NodeManager. ResourceManager sučelje pruža metrike za računalni grozd dok NodeManager pruža informacije za svaki čvor te aplikacije i spremnike koji se izvode na čvoru.

Na slici 2.5 je primjer konfiguracije YARN upravitelja gdje prvi redak predstavlja fizičku memoriju jednog YARN kontejnera, prvi redak predstavlja minimalnu memoriju koja će biti dodjeljena jednom kontejneru, treći redak predstavlja maksimalnu memoriju koja će biti dodjeljena jednom kontejneru, četvrti redak predstavlja ukupni broj jezgri kojima YARN upravlja, peti redak predstavlja minimalni broj jezgri koje treba dodijeliti jednom kontejneru i posljednji redak predstavlja maksimalni broj jezgri koje YARN može dodijeliti jednom kontejneru.

Property Name	Configured Value	Meani
yarn.nodemanager.resource.memory-mb	8192 MB	Physical memory can for containers.
yarn.scheduler.minimum-allocation-mb	1024 MB	Minimum container
yarn.scheduler.maximum-allocation-mb	8192 MB	Maximum container
yarn.nodemanager.resource.cpu-vcores	3	Number of vcores tha allocated for contain
yarn.scheduler.minimum-allocation-vcores	1	Minimum container
yarn.scheduler.maximum-allocation-vcores	3	Maximum container

Slika 2.5: Primjer YARN konfiguracije, izvor: [10]

2.3.4. Raspodijeljeni način izvođenja na platformi Mesos

Apache Mesos, kernel distribuiranih sustava, ima visoku raspoloživost za Upravljački čvor i radne čvorove, može upravljati resursima svake aplikacije i ima podršku za Docker spremnike. Može pokrenuti Spark poslove, Hadoop MapReduce ili bilo koju drugu aplikaciju usluge. Sučelje podržava Javu, Python i C++.

Apache Mesos ima glavne i podređene procese. Upravljački čvor nudi resurse aplikaciji koja može ili ne mora prihvatiti. Prema tome, zauzimanje raspoloživih resursa i pokretanje poslova određeno je samom aplikacijom. Apache Mesos omogućuje precizno upravljanje resursima u sustavu kao što su CPU, memorija, diskovi i portovi. Apache Mesos također nudi kontrolu nad resursima gdje Spark dodjeljuje fiksni broj CPU-a svakom izvršitelju unaprijed koji se ne oslobađaju sve dok aplikacija ne završi s radom.

Apache Mesos upravitelj računalnog grozda također podržava automatsko obnavljanje Upraviteljskog čvora pomoću Apache ZooKeepera. Zadatci koji se trenutno izvršavaju nastavljaju raditi u slučaju greške.

Mesos osigurava provjeru autentičnosti za bilo koji entitet u interakciji s računalnim grozdom. To uključuje radne čvorove koji se registriraju s Upravljačem, okviri (to jest, aplikacije) koji se podnose računalnom grozdu i operatore koji koriste krajnje točke kao što su HTTP krajnje točke. Svakom od tih entiteta može se omogućiti korištenje provjere autentičnosti ili ne. Mesosov zadani modul za provjeru autentičnosti, *Cyrus SASL*, može se zamijeniti vlastitim modulom. Popisi kontrole pristupa koriste se za odobravanje pristupa uslugama u Mesosu. Prema zadanim postavkama, komunikacija između modula u Mesosu je nekodirana, ali se može postaviti tako da bude kodirana.

Apache Mesos nudi brojne podatke za glavne i podređene čvorove dostupne putem URL-a. Te metrike uključuju, primjerice, postotak i broj dodijeljenog procesora,

ukupno upotrijebljenu memoriju, postotak korištene dostupne memorije, ukupni prostor na disku, dodijeljeni prostor na disku, vrijeme neprekidnog rada glavnog računala, podređene registracije, povezane radne čvorove itd. Podržano je praćenje i izolacija mreže po kontejneru. [9]

2.3.5. Raspodijeljeni način izvođenja u računalnom oblaku

Spark se može postaviti u tradicionalnom lokalnom podatkovnom centru, kao i u oblaku. Oblak omogućuje organizacijama da implementiraju Spark bez potrebe za stjecanjem hardverske ili specifične stručnosti za postavljanje, nego to dobiju davatelja usluga kao što su Amazon Web Services, Google Cloud Platform i Microsoft Azure koji nude jednostavno pokretanje Spark rješenja. Glavne razloge koje korisnici vide u pokretanju u oblaku kao prednost za Spark su brže vrijeme uvođenja, bolja dostupnost, češća ažuriranja značajki/funkcionalnosti, više elastičnosti, više geografske pokrivenosti i troškovi povezani s stvarnim korištenjem.

2.4. Spark tipovi podataka

2.4.1. Resilient Distributed Dataset

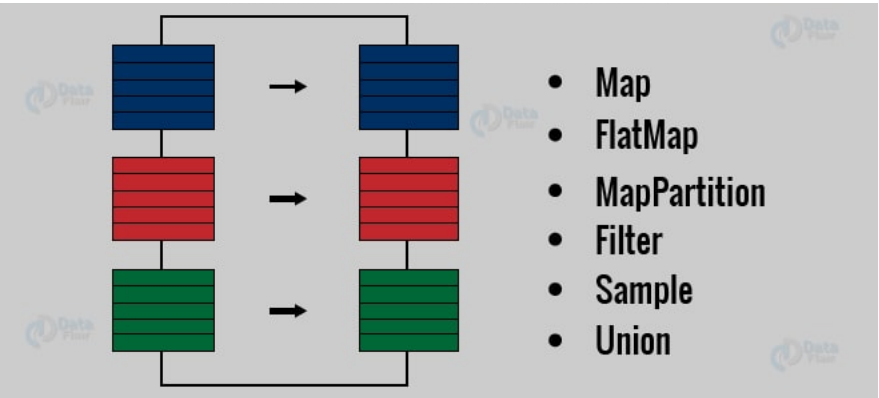
RDD (engl. *Resilient Distributed Dataset*) je osnovna apstrakcija za rad s podatcima u aplikacijama platforme Apache Spark. Nastaje učitavanjem podataka iz vanjskog izvora ili iz postojeće kolekcije objekata u memoriji.

RDD predstavlja cijeli skup podataka koji je particioniran na čvorovima izvodačima te mu se može paralelno pristupati. Vrijednost RDDa se ne može promijeniti već se prilikom transformacija stvara novi RDD. Otporan je na pogreške jer se može ponovno rekonstruirati.

Dvije vrste operacija koje RDD podržava su transformacije i akcije. Transformacije su operacije koje stvaraju novi RDD. Obrada transformacija odvija se lijeno (tek ukoliko dode do akcije), a akcije su operacije koje konzumiraju RDD.

2.4.1.1. Transformacije

Transformacija je oblik operacije u kojoj se svi potrebni podatci za izvršavanje nalaze u memoriji izvršitelja te nije potrebno izmjenjivati podatke s drugim izvršiteljima. [11] Slika 2.6 prikazuje primjer obavljanja transformacije gdje retci iste boje predstavljaju podatke koji su se nalazili na istom čvoru prije izvršavanja. Iz slike je vidljivo da nije potrebno izmjenjivati podatke izmedu čvorova.



Slika 2.6: Očuvanje podataka prilikom transformacije, izvor: [11]

Glavne transformacije nad RDD-om:

– *rdd.map(func)* - Naredba pretvara RDD jednog tipa u RDD drugog tipa izvodenjem funkcije *func* nad svakim podatkom u RDD-u. Iz jednog elementa početnog RDDa nastaje jedan element konačnog RDDa

```
1 val data =
    spark.sparkContext.parallelize(Seq('abc', 'def',
    'ab', 'cdef'))
2 val mapFile = data.map(v => (v, v.length))
```

Rezultat izvođenja biti će novi RDD sa vrijednostima (('abc',3), ('def',3), ('ab',2), ('cdef',4))

– *flatMap(func)* - Naredba pretvara RDD jednog tipa u RDD drugog tipa izvođenjem funkcije *func* nad svakim podatkom u RDD-u. Iz jednog elementa početnog RDDa nastaje jedan ili više elemenata konačnog RDDa.

```
1 val data =
    spark.sparkContext.parallelize(Seq('abc', 'def',
    'ab', 'cdef'))
2 val mapFile = data.flatMap(v => v.split(""))
```

Rezultat izvođenja biti će novi RDD sa vrijednostima ('a','b','c','d','e','f','a', 'b','c','d','e','f')

– *filter(func)* - Naredba filtrira one elemente za koje izvođenje funkcije *func* vrati vrijednost *False*.

```
1 val data =
    spark.sparkContext.parallelize(Seq('abc', 'def',
    'ab', 'cdef'))
2 val mapFile = data.filter(v => v.length == 3)
```

Rezultat izvođenja biti će novi RDD sa vrijednostima ('abc','def')

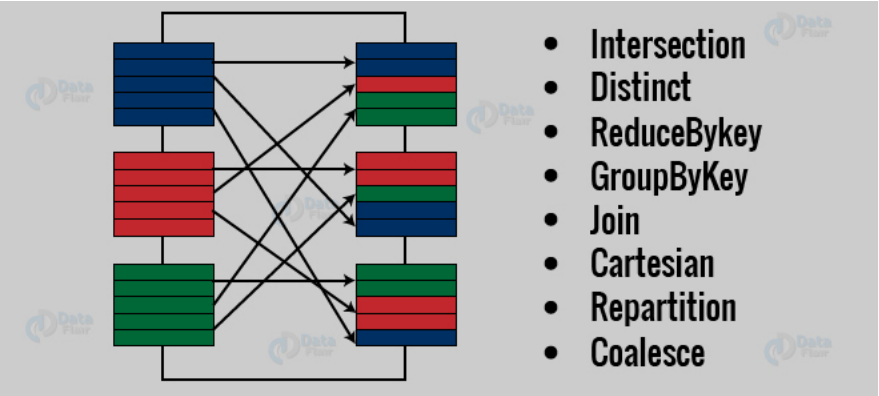
– *distinct()* - Naredba vraća RDD sa jedinstvenim elementima

```
1 val data =
    spark.sparkContext.parallelize(Seq('abc', 'def',
    'ab', 'cdef', 'def'))
2 val mapFile = data.distinct()
```

Rezultat izvođenja biti će novi RDD sa vrijednostima ('abc', 'def', 'ab', 'cdef')

2.4.1.2. Akcije

Akcija je oblik operacije kojoj su potrebni podatci koji se ne moraju nužno nalaziti u memoriji istog izvršitelja. [11] Slika 2.7 prikazuje primjer izvršavanja akcije gdje svi retci iste boje predstavljaju podatke koji se nalaze na istom čvoru prije izvršavanja.



Slika 2.7: Razmještaj podataka prilikom akcije, izvor: [11]

Glavne akcije nad RDD-om:

– *rdd.reduce(func)* - Naredba vrši agregaciju nad elementima RDD-a. Funkcija koja se poziva prima 2 argumenta i vraća 1.

```
1  val data =
       spark.sparkContext.parallelize(Seq('abc',
       'def', 'ab', 'cdef'))
2  val mapFile = data.reduce((a,b) ->
       a.append(b))
```

Rezultat izvodenja biti će jedan RDD koji predstavlja jedan niz.

– *rdd.foreach(func)* - Naredba koja nad svakim elementom obavlja naredbu *func*. Za razliku od map metode, metoda foreach ne vraća rezultat

```
1  val data =
       spark.sparkContext.parallelize(Seq('abc',
       'def', 'ab', 'cdef'))
2  val mapFile = data.foreach(println)
```

Rezultat izvodenja biti će ispis svih elemenata RDD-a

– *rdd.collect()* - Naredba sakuplja sve elemente na Upravljaču u obliku liste

```
1        val data =
             spark.sparkContext.parallelize(Seq('abc',
             'def', 'ab', 'cdef'))
2        val mapFile = data.collect()
```

Rezultat izvodenja biti će skupljeni svi podatci na Upravljaču u obliku liste

– *rdd.take(n)* - Naredba vraća prvih *n* elemenata RDD-a, ili ako se *n* ne zada vraća sve elemente, u obliku liste

```
1        val data =
             spark.sparkContext.parallelize(Seq('abc',
             'def', 'ab', 'cdef'))
2        val mapFile = data.take(2)
```

Rezultat izvođenja biti će lista koja sadrži elemente 'abc' i 'def'

2.4.2. DataFrame

DataFrame se pojavio u Spark verziji 1.3.0. DataFrame možemo definirati kao skup podataka organiziran u imenovane stupce. DataFrame je sličan tablici u relacijskoj bazi podataka. Može se reći da je to relacijska tablica s dobrom tehnikom optimizacije.

Ideja DataFrame-a je da omogući obradu velike količine strukturiranih podataka. DataFrame sadrži retke sa shemom. Shema je ilustracija strukture podataka.

DataFrame u platformi Apache Spark prevladava nad RDD-om, ali sadrži i značajke RDD-a. Značajke zajedničke RDD-u i DataFrame-u su nepromjenjivost, *in-memory* obrada, otpornost, distribuirana računalna obrada. To korisniku omogućuje nametanje strukture distribuiranom prikupljanju podataka. Time se osigurava apstrakcija više razine.

DataFrame možemo izgraditi iz različitih izvora podataka. Za primjer strukturiranu datoteku podataka, vanjske baze podataka ili postojeće RDD-ove. Aplikacijsko programsko sučelje u DataFrameu dostupno je na različitim jezicima. Primjeri uključuju Scala, Java, Python i R.

I u Scali i u Javi, predstavljamo DataFrame kao skup podataka redaka. U Scala API-ju, DataFrames su tip pseudonima Dataset [Row]. U Java API-ju, korisnik koristi skup podataka <Row> za predstavljanje DataFrame-a. [12]

2.4.3. DataSet

Dataset je struktura podataka u SparkSQL-u koja je strogo tipizirana i predstavlja presliku relacijskoj shemi. To je proširenje DataFrame API-a. Spark Dataset pruža i sigurnost prilikom prevođenja jer su svi podatci tipizirani i objektno orijentirano sučelje. Dataset je postao dostupan u verziji 1.6.

Koder je primarni koncept u okviru serijalizacije i deserizacije (*SerDes*) u Spark SQL-u. Enkoder prevodi između JVM objekata i Sparkovog unutarnjeg binarnog formata. Spark ima ugrađene enkodere koji su vrlo napredni. Oni generiraju bajt-kod za interakciju s off-heap podacima. Enkoder omogućuje pristup pojedinačnim atributima na zahtjev bez potrebe za deserijalizacijom cijelog objekta. Kako bi vrijeme i prostor unosa bili učinkovitiji, Spark SQL koristi SerDe okvir. Budući da koder poznaje shemu zapisa, može postići serijalizaciju i deserijalizaciju.

Spark Dataset je strukturiran i prijelazne operacije se lijeno izvode tek kada dođe do provođenja akcije. Interni skup podataka predstavlja logičan plan. Logički plan govori računskom upitu koji trebamo proizvesti. logički plan je osnovni plan upita katalizatora za logičkog operatora da oblikuje logički plan upita. Kada to analiziramo i riješimo, možemo oblikovati fizički plan upita.

[13]

2.4.4. Accumulator

Akumulator (engl. *Accumulator*) je struktura podataka koja služi kao zajednička varijabla za sve Izvršitelje. Objašnjenje zašto su nam potrebni akumulatori prikazano je u slijedećem primjeru. Kada bi imali isječak koda kao na slici 2.8, prilikom pokretanja varijabla *blankLines* se kreira unutar konteksta Upravljača, proslijedi se na Izvršitelje, ali se na svakom izvršitelju stvara njegova instanca te varijable te promjenom vrijednosti u Izvršiteljima vrijednost varijable na Upravljaču ostaje ista. Zato trebamo koristiti akumulatore kojima Izvršitelji mogu dodavati vrijednost, ali ju ne mogu čitati. Samo Upravljač moze čitati vrijednost akumulatora, a razlog zašto izvršitelji ne mogu čitati vrijednost akumulatora je taj što to nije potrebno. Akumulator ne možemo koristiti kao parametar o kojem će ovisiti obrada podataka jer će neki izvršitelji brže odraditi svoje zadatke stoga bi operacije bile nekonzistentne. Akumulator mozemo koristiti za analiziranje izvršavanja obrade, ali ne i za samu obradu. Ispravan primjer korištenja dijeljenje varijable pomoću akumulatora prikazan je na slici 2.9.

Ukoliko dode do pogreške prilikom obrade, operacija će se ponovno izvesti. Takvo ponašanje bi moglo akumulatore učiniti ne konzistentnima kada bi se za ponovljenu

```
var blankLines: Int = 0

sc.textFile("some log file", 4)
    .foreach { line =>
        if (line.length() == 0) blankLines += 1
    }

println(s"Blank Lines=$blankLines")
```

Slika 2.8: Primjer korištenja obične varijable, izvor: [14]

```
val blankLines = sc.accumulator(0, "Blank Lines")

sc.textFile("some log file", 4)
    .foreach { line =>
        if (line.length() == 0) blankLines += 1
    }

println(s"\tBlank Lines=${blankLines.value}")
```

Slika 2.9: Primjer korištenja akumulatora, izvor: [14]

operaciju također ponovno proveli i naredba nad akumulatorom. Međutim *Apache Spark* garantira konzistentnost prilikom izvođenja akcija, ali isto ne vrijedi i za transformacije stoga se preporuča korištenje akumulatora isključivo tokom akcija.

3. Razlike između inačica 1 i 2 platforme Apache Spark

Svaka nova verzija Apache Sparka donosi razne optimizacije koje ubrzavaju izvođenje. Ipak, postoje veće promjene koje je potrebno detaljnije objasniti, a to su:

- Promjena osnovne podatkovne strukture
- Spark SQL
- Spark Streaming
- Akumulatori

3.1. Promjena osnovne strukture podataka

RDD API je u Sparku od izdanja 1.0. Iz perspektive razvojnog programera, RDD je jednostavno skup objekata koji predstavljaju podatke.

RDD API pruža mnoge metode transformacije, kao što su map(), filter() i reduce() za izvođenje izračuna podataka. Svaka od ovih metoda rezultira novim RDD-om koji predstavlja transformirane podatke. Međutim, ove metode su samo definiranje operacija koje se izvode, a transformacije se ne izvode dok se ne pozove akcija. Primjeri metoda akcije su collect() i saveAsObjectFile().

Glavna prednost RDD-a je u tome što su jednostavni i dobro razumljivi jer se bave konkretnim klasama, pružajući poznati objektno orijentirani stil programiranja s sigurnošću pri prevođenju. Primjerice, s RDD-om koji sadrži primjerke osobe možemo filtrirati prema dobi pomoću referencije na atribut dobi svakog objekta Person:

```
rdd.filter(person -> person.age > 18)
```

Glavni nedostatak RDD-a je u tome štp se oni ne obrađuju posebno dobro. Kad god Spark mora distribuirati podatke unutar grozda ili zapisivati podatke na disk, to se čini pomoću Java serijalizacije. Opterećenje serijalizacije pojedinih objekata je skupa i

zahtijeva slanje podataka i strukture između čvorova (svaki serijski objekt sadrži strukturu klase kao i vrijednosti). Postoji i opterećenje skupljanja smeća koje proizlazi iz stvaranja i uništavanja pojedinačnih objekata.

Spark 1.3 je predstavio novi DataFrame API kao dio inicijative Project Tungsten koja nastoji poboljšati performanse i skalabilnost Sparka. DataFrame API uvodi koncept sheme za opisivanje podataka, omogućujući Sparku da upravlja shemom i prosljeđuje samo podatke između čvorova, na mnogo učinkovitiji način nego pomoću serializacije cijelih objekata sa shemom. Postoje i prednosti u izvođenju računanja u pojedinom procesu, jer Spark može serijski pretvoriti podatke u binarnom formatu, a zatim izvršiti mnoge transformacije izravno na ovoj memoriji, izbjegavajući troškove prikupljanja smeća povezanih s izgradnjom pojedinačnih objekata za svaki redak skupa podataka. Budući da Spark razumije shemu, nema potrebe koristiti Java serializaciju za kodiranje podataka.

API DataFrame drastično se razlikuje od RDD API-ja jer je to API za izgradnju relacijskog plana upita koji Sparkov Catalyst optimizator može izvršiti. API je prirodan za programere koji su upoznati s planovima upita za izgradnju, ali nisu prirodni za većinu razvojnih programera. Plan upita može se izgraditi iz SQL izraza u nizovima ili iz funkcionalnijeg pristupa korištenjem *fluent-style* API-ja.

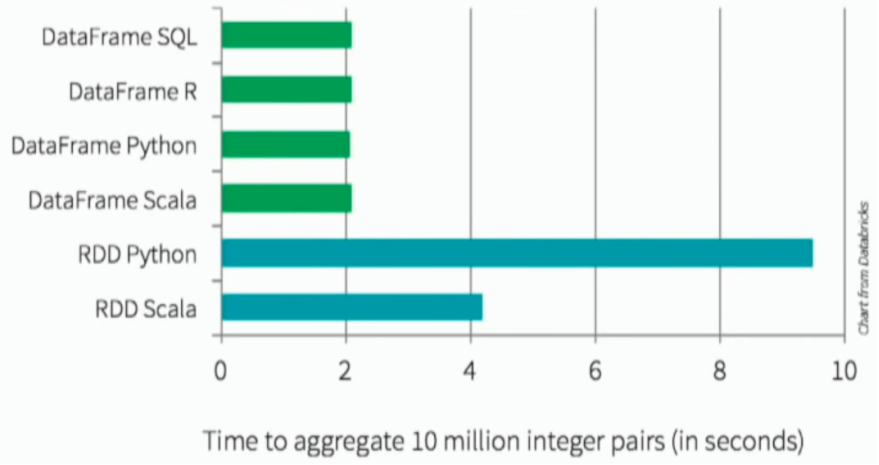
Budući da se kod odnosi na atribute podataka po imenu, kompajler ne može prepoznati nikakve pogreške. Ako su imena atributa netočna, tada će se pogreška otkriti tek tijekom izvođenja, kada se kreira plan upita. Također iako DataFrame dobro radi u Scali, u Javi ima problema prilikom kreiranja DataFrame objekta iz RDD objekta. Na primjer, prilikom kreiranja DataFramea iz postojećeg Java RDD objekta, Sparkov Catalyst optimizator ne može izvesti shemu i pretpostavlja da bilo koji objekt u DataFrame-u implementira *scala.Product* sučelje.

Dataset API, dostupan od verzije 2.0, napravljen je s ciljem pružanja najboljeg od oba svijeta. Poznati stil objektno orijentiranog programiranja i tipska sigurnost prilikom kompajliranja RDD API-ja, ali s prednostima performansi koje nudi Catalyst optimizatora upita. Dataset također upotrebljava isti učinkoviti mehanizam za pohranu podataka kao i DataFrame API.

Kada je riječ o serijaliziranju podataka, Dataset API ima koncept enkodera koji prevodi između JVM reprezentacija (objekata) i Sparkovog unutarnjeg binarnog formata. Spark ima ugrađene enkodere koji su vrlo napredni u tome što generiraju bajtni kod za interakciju s *off-heap* podacima i osiguravaju pristup pojedinačnim atributima na zahtjev bez potrebe za deserijalizacijom cijelih objekata. Spark još ne nudi API za implementaciju prilagođenih enkodera, ali je planirano za buduća izdanje.

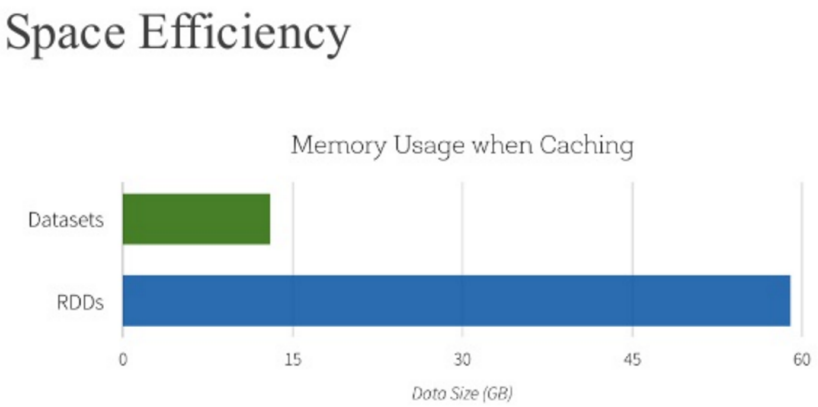
U konačnici preostali su RDD i Dataset API, a DataFrame API je zadržan kao apstrakcija Dataset objekta radi zadržavanja kompatibilnosti unatrag. [15]

Na slici 3.1 prikazan je odnos vremena izvođenja obrade nad DataFrame objektima koristeći Catalyst optimizator i RDD objekata nad kojima nije moguće koristiti Catalyst optimizator.



Slika 3.1: Usporedba vremena izvođenja upita uz pomoću *Catalyst* optimizatora u odnosu na izvođenje bez optimizatora, izvor: [16]

Na slici 3.2 vidimo drastičnu razliku količine potrebne memorije za obradu RDD objekata i Dataset objekata.



Slika 3.2: Usporedba zauzeća memorije između *Dataseta* i *RDD*-a, izvor: [17]

3.2. Spark SQL

Spark 2.x.x je znatno popravio SQL funkcionalnosti. Implementiran je nativni SQL parser, mogućnost podupita i API za automatsko pretvaranje JSON i CSV formata u Dataset objekte te Dataset objekata u JSON i CSV datoteke, izvršavanje SQL upita nad učitanim JSON datotekama bez potrebnog zadavanja sheme.

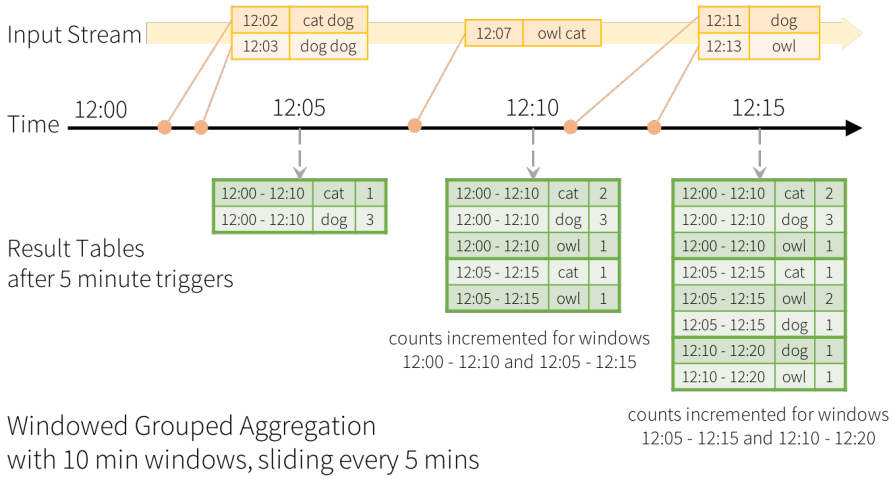
Dodana je i podrška za čitanje JSON i CSV objekata koji se protezu kroz više redaka jedne datoteke. Te je kreirana i funkcija koja pretvara stari zapis spremljene tablice u novi zapis kako bi tranzicija bila što lakša.

Uz to u posljednjoj većoj verziji 2.4.0 dodano je preko 30 funkcija visoke razine za lakše upravljanje podatcima. [18] [19] [20]

3.3. Spark Streaming

Spark Streaming je Spark komponenta koja nudi mogućnost obrade podataka koje dobivamo kroz tokove podataka. Obrada se odvija u obliku manjih grupa umjesto jedan po jedan podatak.

U verziji 2.0.0 dodana je *alpha* verzija nadogradnje *Spark Structured Streaming* koja je bazirana na Dataset API-u koja ne koristi obradu na principu manjih grupa već obradu na principu neograničene tablice. Slika 3.3 prikazuje kontinuirani ulazni tok podataka koji se sprema u neograničenu tablicu nad kojom se vrši obrada. Spark Structured Streaming je bolji odabir za obradu tokova podataka od Spark Streaminga koji se temelji na RDD-ovima zbog jednostavnijeg, a moćnijeg Dataset API-a te optimizacija koje on nudi uz jednostavniji i napredniji Structured Streaming API. U verzijama 2.x.x je sa Spark Streaminga okrenut na Spark Structured Streaming na čijoj nadogradnji se konstatno radi. [21] [7] [22]



Slika 3.3: Grafički prikaz kontinuirane obrade na principu klizećeg prozora, izvor: [23]

3.4. Accumulator

U verziji 1.3.0 implementirani su samo akumulatori tipa Long, dakle "globalni" brojači dok se akumulatori drugih tipova podataka trebaju samostalno implementirati nasljeđivanjem klase AccumulatorParam[generalizirani tip]. Prilikom kreiranja akumulatora potrebno je zadati početnu vrijednost te je moguće zadati ime akumulatora. Ukoliko se zada ime akumulatora, biti će prikazane u *web* sučelju. U verziji 2.x.x akumulatori su značajno promijenjeni. Osnovna uporaba je postala jednostavnija. Nije moguće zadati početnu vrijednost, ali je nuzno zadati ime. Postoji više preddefiniranih akumulatora (za tipove *int*, *double* te kao kolekcija). A prilikom kreiranja vlastitog akumulatora u novoj verziji se nasljeđuje klasa *AccumulatorV2* koja nudi metode *reset*, *add*, *merge*, *copy* i druge. [14] [18] [19] [20]

3.5. Manje promjene

Bitno je spomenuti i veliki napredak postignut u slijedećim radnim okvirima:

- *MLlib*
- *Spark Streaming*
- *GraphX*

U navedenim radnim okvirima nije bilo velikih promjena kao što je slučaj za radni okvir *Spark SQL*, ali ipak je bitno spomenuti da iako nije bilo promjena koje bi utjecale na pisanje samog koda, postignut je veliki napredak što se tiče brzine izvođenja i stabilnosti programskih rješenja.

3.5.1. MLlib

MLlib je nadogradnja koja služi za raspodijeljeno treniranje modela umjetne inteligencije. *MLlib* se konstantno nadograđuje dodavanjem poznatih algoritama za umjetnu inteligenciju i analizu podataka stoga nije potrebno samostalno pisati te algoritme već je dovoljno samo učitati podatke kako bi se pokrenulo treniranje. U implementaciju je moguće dodati i *Spark Streaming* kako bi se model mogao trenirati pomoću tokova podataka.

3.5.2. GraphX

GraphX je nadogradnja služi za provođenje raznih analiju grafova. emphGraphX iako je baziran na RDD-ovima nudi vlastiti *API* visoke razine (*npr.* dohvaćanje podgrafova, spajanje vrhova, *itd.*). Također kao i *MLlib*, *GraphX* se konstantno nadograđuje dodavanjem novih algoritama kako bi se olakšalo provođenje visoko performantnih analiza nad grafovima.

4. Praktična demonstracija razlika

4.1. Primjer korištenja Spark Core komponente

U ovom primjeru prikazati ću vrijeme izvršavanja jednostavnog koda za brojanje riječi.

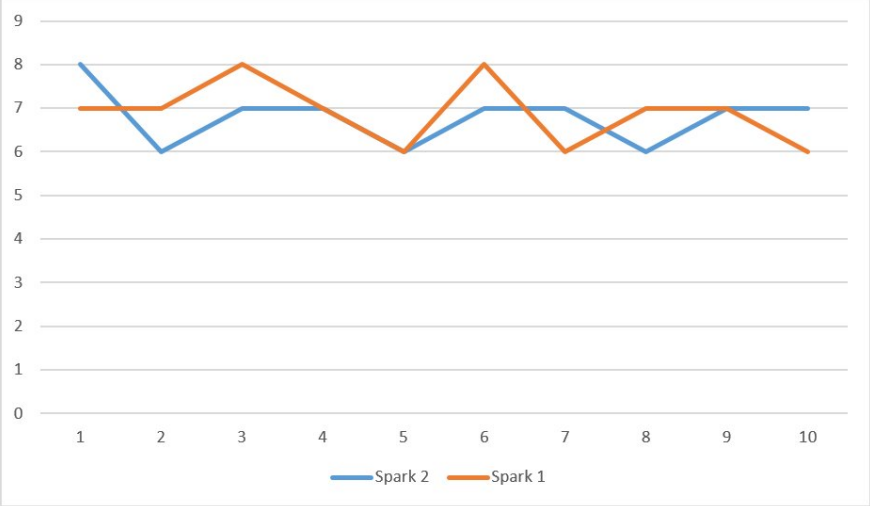
```
1 val data = sc.textFile("wordsInput.txt")
2 val words = data.flatMap(_.split(" ")).map((_,1))
3 val aggr = words.reduceByKey(_+_).collect()
```

Pomoću Spark Contexta učitamo datoteku u varijablu data. Redove datoteke podijelimo po razmacima te ih pretvorimo u parove (ključ, 1) kako bi naredbom reduceByKey zbrajali vrijednosti po ključevima. Na kraju ćemo za svaku riječ izbrojanu vrijednost koliko puta se ta riječ pojavila u tekstu. Ovaj kod moguće je pokretati pomoći inačice 1 i 2.

Mjerenje je pokrenuto 10 puta za obje inačice sa istim skupom podatka. Kao što možemo vidjeti na grafu 4.1 rezultati su vrlo slični. Razlog tomu je što se u oba slučaja koristila struktura RDD. Pri ovakvom skupu ulaznih podataka niti se ne mogu podatci drugačije obraditi. Prosječno vrijeme trajanja obrade podataka korištenjem inačice 2 je 6.8 sekundi, a korištenjem inačice 1 u prosjeku obrada traje 6.9 sekundi.

4.2. Usporedba izvršavanja sql upita pomoću Dataset i Dataframe API-a

U slijedećem kodu prikazan je primjer učitavanja JSON datoteke u DataFrame objekt. Obzirom da učitavamo JSON file koji je strukturiran, nije potrebno zadatavi imena stupaca DataFrame objekta. U kodu se koriste pojednostavljeni SQL upiti pomoću API-a visoke razine.



Slika 4.1: Usporedba brzine izvođenja verzije 1 i 2

```
1 import spark.implicits._
2 val df = spark.read.json("people.json")
3 val filtered = df.filter($"name" rlike "A" )
4 val grouped = filtered.groupBy("name")
5 val average = grouped.avg("age").show()
```

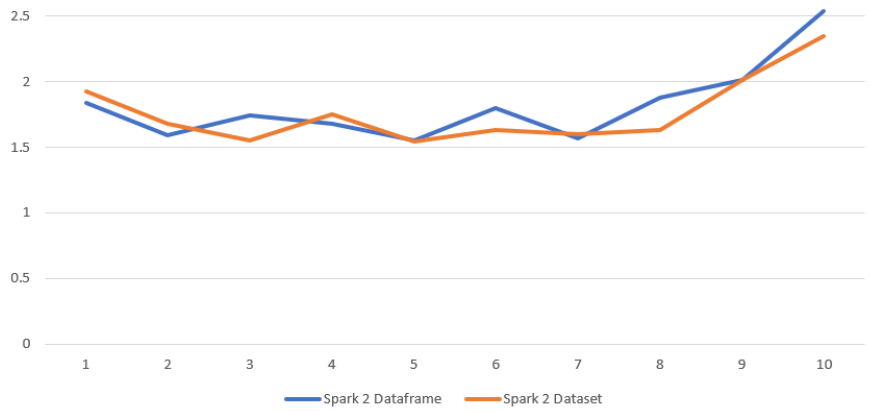
Idući kod prikazuje primjer učitavanja JSON datoteke u Dataset objekt te isti u DataFrame objekt nad kojim se izvršavaju upiti.

```
1 import spark.implicits._
2 case class Person(name: String, age: Long)
3 val df = spark.read.json("people.json").as[Person].toDF()
4 val filtered = df.filter($"name" rlike "A" )
5 val grouped = filtered.groupBy("name")
6 val average = grouped.avg("age").show()
```

Slika 4.2 Vrijeme izvođenja je identično što je i očekivano obzirom da se upiti izvode pomoću istog SQL interpretera.

4.3. Primjer izvršavanja Spark Streaming aplikacije

U slijedećem kodu prikazan je primjer otvaranja sučelja za prihvaćanje i obradu toka podataka. Prvo je potrebno stvoriti Streaming Context koji prima općenitu Spark konfiguraciju i broj sekundi koliko često će se obrađivati pristigli podatci. U trećoj liniji



Slika 4.2: Usporedba vremenskog izvođenja SQL upita nad DataFrame i Dataset API-em

se otvara sučelje za prihvat podataka. Linija 4. i 5. predstavljaju primjer jednostavne obrade podataka.

Bitno je naglasiti da ukoliko se ne pozove naredba ssc.start() podatci se neće obrađivati. Poziv naredbi transformacija i akcija je priprema nacrta po kojem će se podatci obrađivati ali samu obradu je prilikom obrađivanja tokova podataka potrebno ručno pokrenuti.

```
1  val conf = new
       SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
2  val ssc = new StreamingContext(conf, Seconds(1))
3  val lines = ssc.socketTextStream("localhost", 9999)
4
5  val words = lines.flatMap(_.split(" "))
6  val wordCounts = pairs.reduceByKey(_ + _)
7
8  wordCounts.print()
9
10 ssc.start()
11 ssc.awaitTermination()
```

4.4. Primjer izvršavanja MLlib aplikacije inačice 1 i 2

U slijedećem kodu prikazan je primjer korištenja MLlib implementacije naivnog Bayesovog klasifikatora. Slijedeći kod prikazuje primjer aplikacije inačice 1 za treniranje modela naivnog Bayesovog klasifikatora. Podatci se pretvaraju u tip podataka Labe-

ledPoint koji sadrži oznaku akcije te listu parametara povezanih uz akciju. Model se trenira vrlo jednostavno postavljanjem priređenih podataka te nije potrebno brinuti o statistikama ili znati samostalno napraviti implementaciju algoritma.

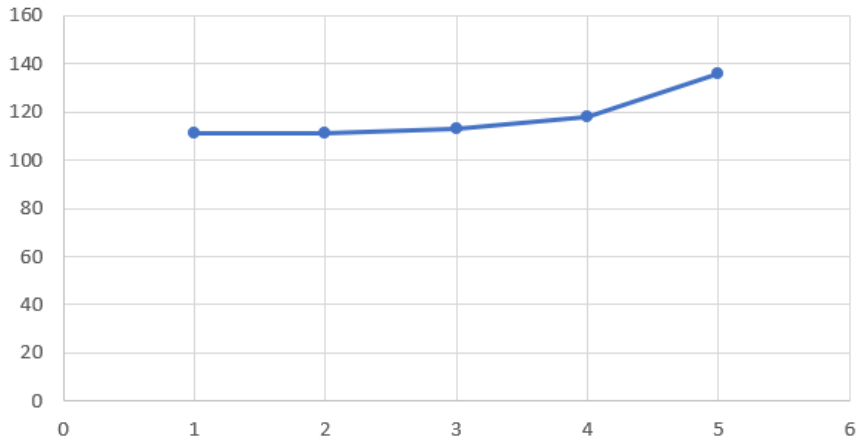
```
1  val data = sc.textFile("input.txt")
2  val parsedData = data.map { line =>
3      val parts = line.split(',')
4      LabeledPoint(parts(0).toDouble,
           Vectors.dense(parts(1).split('
           ').map(_.toDouble)))
5  }
6
7  val splits = parsedData.randomSplit(Array(0.6, 0.4))
8  val training = splits(0)
9  val test = splits(1)
10 val model = NaiveBayes.train(training, lambda = 1.0)
11 val prediction = model.predict(test.map(_.features))
12
13 val predictionAndLabel =
     prediction.zip(test.map(_.label))
14 val accuracy = 1.0 * predictionAndLabel.filter(x =>
     x._1.equals(x._2)).count() / test.count()
15
16 println(accuracy)
```

Kod u nastavku prikazuje implementaciju istog algoritma napisanog pomoći API-a inačice 2. Podatci se priređuju na identičan način, ali se model postavlja drugačije te pruža više mogućnosti postavljanja modela.

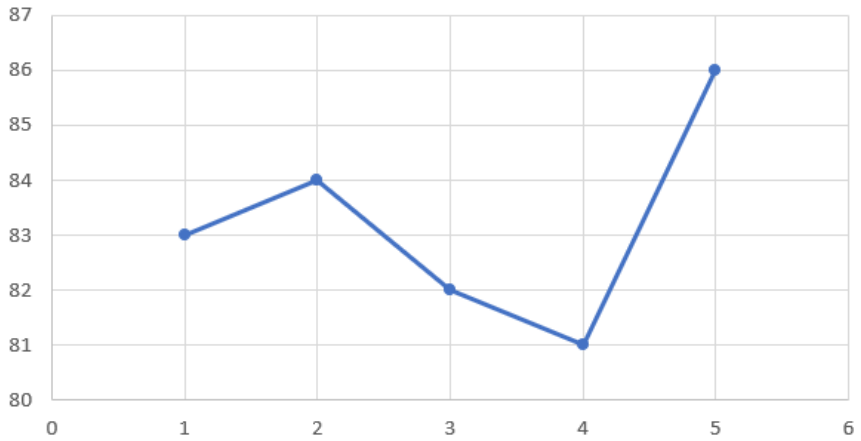
```
1  val data = sc.textFile("input.txt")
2  val parsedData = data.map { line =>
3    val parts = line.split(',')
4    LabeledPoint(parts(0).toDouble,
       Vectors.dense(parts(1).split(' ').map(_.toDouble)))
5  }
6
7  val Array(training, test) =
     parsedData.randomSplit(Array(0.6, 0.4))
```

```
8
9  val model = NaiveBayes.train(training, lambda = 1.0,
       modelType = "multinomial")
10 val predictionAndLabel = test.map(p =>
       (model.predict(p.features), p.label))
11 val accuracy = 1.0 * predictionAndLabel.filter(x => x._1
       == x._2).count() / test.count()
12
13 println(accuracy)
```

Implementacija inačice 2 izvodi se u dvije trećine vremena potrebnog implementaciji inačice 1. Što možemo vidjeti iz u grafova 4.3 i 4.4



Slika 4.3: Prikaz vremena potrebnog za treniranje modela koristeći inačicu 1



Slika 4.4: Prikaz vremena potrebnog za treniranje modela koristeći inačicu 1

5. Zaključak

Apache Spark je od platforme koja nudi raspodijeljenu obradu postala platforma koja uz to nudi mogućnosti povezivanja na bazu, treniranja modela umjetne inteligencije, obrade tokova podataka, obrade grafova, optimizacije upita i druge. Također je moguće vrlo jednostavno istovremeno koristit više komponenti odjednom bez postavljanja posebnih konfiguracija kako bi međusobno besprijekorno surađivale. Iako je u početku imao poteškoća sa stabilnošću i u nekim slučajevima bio nepouzdan, danas je alat koji u potpunosti ispunjava svoju zadaću.

Pisanje programskog rješenja za platformu *Apache Spark* vrlo je slično pisanju objektno orijentirane aplikacije te je zahvaljujući jednostavnom i naprednom *API*-u iznimno jednostavno za savladati. Prilikom pisanja programskog rješenja nije potrebno poznavati detalje rada radnog okvira kako bi se aplikacija optimalno izvodila zahvaljujući postojećim optimizacijama.

Svakom novom inačicom dodaju se nove funkcionalnosti, pritom zadržavajući postojeće, uz iznimku prelaska sa inačice 1.6 na inačicu 2.0, ali kako bi se maksimalno olakšao prijelaz dokumentirani su svi koraci koje je potrebno napraviti kako bi prijelaz bio uspješan uz što manje preinaka u kodu.

LITERATURA

[1] “Wikipedia - apache spark.”

[2] “Literatura kolegija "raspodijeljena obrada velike količine podatak" sa fakulteta elektrotehnike i računarstva, sveučilište u zagrebu.”

[3] “The spark stack - o’reilly.”

[4] “What are the cluster managers supported in apache spark.”

[5] “Spark streaming framework and processing models.”

[6] “Spark službena dokumentcija verzije 2.3.0.”

[7] “Spark streaming programming guide.”

[8] “Databricks - what is spark streaming?.”

[9] “Apache spark cluster managers: Yarn, mesos, or standalone?.”

[10] “Apache spark on yarn: Resource planning.”

[11] “Spark rdd operations-transformation and action with example.”

[12] “Spark sql dataframe tutorial – an introduction to dataframe.”

[13] “Spark dataset tutorial – introduction to apache spark dataset.”

[14] “Spark accumulators explained: Apache spark.”

[15] “Apache spark: Rdd, dataframe or dataset?.”

[16] “A tale of three apache spark apis: Rdds, dataframes, and datasets - jules damji.”

[17] “A tale of three apache spark apis: Rdds vs dataframes and datasets.”

[18] “Spark službena dokumentcija verzije 1.3.0.”

[19] “Spark službena dokumentcija verzije 2.0.0.”

[20] “Spark službena dokumentcija verzije 2.4.0.”

[21] “Spark streaming vs. structured streaming.”

[22] “Structured streaming programming guide.”

[23] “Structured streaming programming guide.”

Usporedba različitih inačica platforme Apache Spark

Sažetak

Sažetak na hrvatskom jeziku.

U ovom radu opisani su osnovi koncepti platforme Apache Spark, komponente platforme Apache Spark (Spark Core, Spark Streaming, Spark SQL, Spark MLlib, Spark GraphX), mogućnosti pokretanja aplikacije te upravljanja resursima. Objašnjene su temeljne strukture podataka platforme Apache Spark. Napravljena je usporedba inačica 1 i 2 te su razlike prikazane na primjerima.

Ključne riječi: Apache Spark, Spark SQL, MLlib, Spark Streaming, Spark Structured Streaming, GraphX, Scala, Upravljač, Izvršitelj, RDD, Dataset, DataFrame, raspodijeljena obrada podataka, transformacije, akcije, Akumulator, Hadoop YARN, Mesos, Raspodijeljena obrada u oblaku

Title

Abstract

Abstract.

This paper describes basic concepts of Apache Spark platform, components of Apache Spark platform (Spark Core, Spark Streaming, Spark SQL, Spark MLlib, Spark GraphX), possibilities of running a Spark application and possibilites of resource management, defines fundamental data structures of Apache Spark platform. Finally this paper shows a comparison of versions 1 and 2 which are also show in the examples.

Keywords: Apache Spark, Spark SQL, MLlib, Spark Streaming, Spark Structured Streaming, GraphX, Scala, Driver, Executor, RDD, Dataset, DataFrame, distributed computing, transformations, actions, Accumulator, Hadoop YARN, Mesos, Distributed cloud computing