



DRŽAVNI UNIVERZITET
U N O V O M P A Z A R U

Performanse računarskih sistema

Apache Spark na Kubernetes klasteru



kubernetes

Profesor Irfan Fetahović

Aleksandar Milanović

SI 036-018/17

17.04.2021.

Apache Spark

Apache Spark je platforma za klaster računare dizajnirana da bude **brza i opšte namene**.

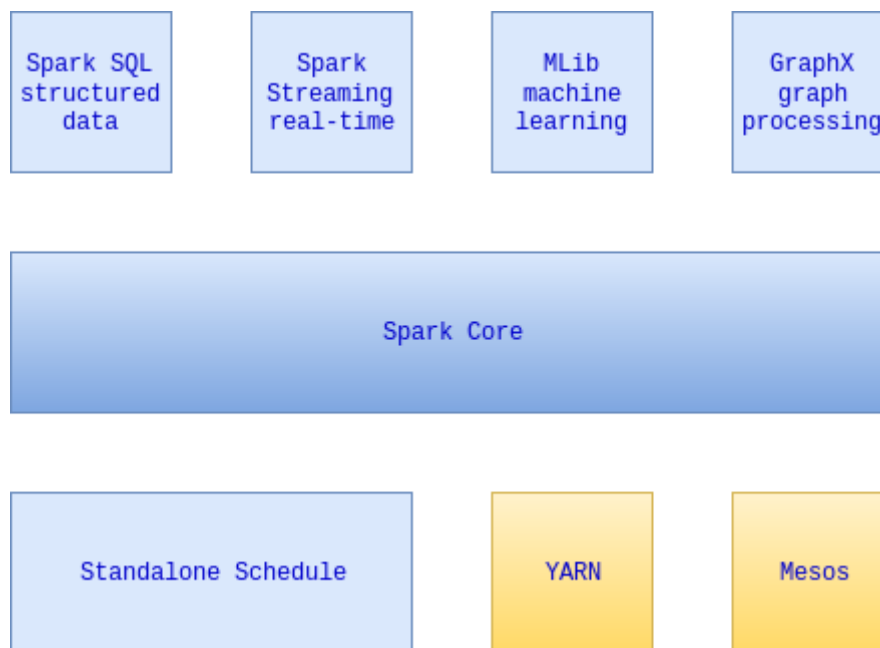
Što se tiče brzine, Spark proširuje popularni MapReduce model da bi podržao više tipova operacija, uključujući interaktivne upite i procesuiranje tokovo podataka (*streaming*). Glavna karakteristika Sparka je mogućnost da se preračunavanje izvode u radnoj memoriji, ali je sistem efikasniji od klasičnog MapReduce modela i u složenim aplikacijama koje koriste podatke sa diska.

Opšta namena Spark-a se ogleda u tome što on pokriva širok opseg poslova koji su prethodno zahtevali različite distribuirane sisteme (paketna obrada, iterativni algoritmi i streaming). Podrškom za različite tipove poslova Spark omogućava da se lako kombinuju različiti tipovi procesuiranja.

Spark projekat sadrži više komponenti. **Jezgro Sparka** je zaduženo za raspoređivanje, distribuiranje i nadgledanje aplikacija koji se sastoje od mnogo zadataka koji se izvršavaju na mnogo mašina. Jezgro pokreće više komponenti višeg nivoa specijalizovanih za različite vrste obrade podataka, kao što su SQL ili mašinsko učenje. Ove komponente su kreirane da blisko sarađuju, pa mogu da se kombinuju kao biblioteke u aplikativnom projektu.

Filozofija takve integracije ima više korisnih posledica. Prvo, sve biblioteke i komponente višeg nivoa imaju korist ako se unapređuju komponente nižeg nivoa. Na primer, kada se Spark jezgro dodatno optimizuje tada i SQL komponenta i komponenta za mašinsko učenje ima korist od te optimizacije. Takođe, troškovi pokretanja celog Sparka su manji nego pokretanje 5 do 10 različitih softverskih rešenja. Ovi troškovi uključuju implementaciju, održavanje, testiranje, podršku i drugo.

Na kraju, najznačajnija prednost bliske integracije je mogućnost pravljenja aplikacije koja kombinuje, naizgled različite, komponente. Na primer, u Spark-u može da se piše jedna aplikacija koja koristi mašinsko učenje za klasifikovanje podataka koji u realnom vremenu koriste podatke sa nekog stream-a podataka. Istovremeno, nad rezultatima mogu da se, preko SQL-a, vrše upiti.



- **Spark Core:** osnovna funkcionalnost Spark-a. Komponente za raspoređivanje, upravljanje memorijom, oporavak od grešaka, interakcija sa fajl sistemom isl. Sadrži api koji definiše *RDD (resilient distributed dataset)*, osnovnu apstrakciju podataka.
- **Spark Streaming:** se može koristiti sa obradu stream-a podataka u realnom vremenu.
- **Spark SQL:** omogućava korišćenje SQL upita u obradi podataka. Uvodi se još jedna apstrakcija podataka (*DataFrame*), što omogućava da izvor podataka bude raznovrsan. Na primer: JSON, Hive, relacione baze podataka, itd..
- **Spark MLlib:** modul se koristi za razvoj alogritama mašinskog učenja na Spark-u i velikim količinama podataka. Omogućava klasifikaciju, regresiju, klasterovanje...
- **Spark GraphX:** omogućava razvoj grafovskih algoritama. Ogoroman broj algoritama koji se često koriste su već implementirani u okviru biblioteke. Jedan od njih je Page Rank algoritam. Koristi još jedan nivo apstrakcije podataka – *Resilient Distributed Property Graph*. Da bi se podržao ovaj tip podataka uvodi se niz akcija (subgraph, joinVertices i aggregateMessages).

Spark poseduje interaktivni shell koji omogućava ad hoc analizu podataka. Za razliku od većine drugih shell-ova, koji manipulišu podacima koristeći memoriju i disk jedne mašine, Spark shell dozvoljava interakciju sa podacima koji su distribuirani na diskovima ili memoriji svih računara na klasteru. Spark se brine o automatskoj distribuciji ovog procesa.

Prvi korak je pokretanje Spark shell-a. Za Python Spark shell se zove **pyspark** i nalazi se u bin poddirektorijumu Spark direktorijuma.

U Spark-u preračunavanja se vrše preko operacija nad distribuiranom kolekcijom podataka koja je automatski paralelizovana na svim čvorovima klastera. Kolekcija se zove **resilient distributed datasets (RDDs)**.

```
>>> lines = sc.textFile("README.md") # Kreira RDD koji se zove lines
>>> lines.count() # Koliko elemenata ima RDD
127
>>> lines.first() # Prvi član RDD-a ili prva linija fajla README.md
u'# Apache Spark'
```

Na visokom nivou, svaka Spark aplikacija sadrži **driver program** koji izvršava različite paralelne operacije na klasteru. Driver program ima main funkciju i definiše RDD-ove na klasteru, a zatim primenjuje operacije nad njima. U prethodnom primeru driver program je Spark shell.

Driver program pristupa Sparku preko **SparkContext** objekta, koji reprezentuje konekciju ka klasteru. Spark Shell automatski kreira SparkContext u promenljivoj **sc**. Funkcijom `sc.textFile()` koristeći SparkContext, kreirali smo RDD koji reprezentuje linije tekstualnog fajla. Nad RDD-om mogu da se izvrše razne operacije npr. `count()`.

Da bi izvršio operacije, driver program obično upravlja većim brojem nodova preko alata koji se zove *executor*. Na primer, ako pokrenemo `count()` operaciju na klasteru, različite mašine mogu da broje linije u različitim delovima fajla.

Veliki deo Spark api-ja je zadužen za posao prosleđivanja funkcija u opratore da bi mogli da se izvrše na klasteru. Na primer mogli bi da proširimo prethodni primer da filtrira linije koje sadrže reč Python.

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
>>> pythonLines.first()
u'## Interactive Python Shell'
```

Poziv `filter()` ili bilo kog drugog operatora koji prima funkciju kao argument, takođe paralelizuje prosleđenu funkciju na klasteru tj. Spark automatski uzima funkciju (`lines.contains("Python")`) i prosleđuje je u executor čvorova. Ovo za posledicu ima da je moguće da jedan driver program ima delove koji se automatski izvršavaju na više čvorova.

Da bi napisali Spark aplikaciju u Python-u dovoljno je napisati Spark program kao Python skriptu i pokrenuti preko `spark-submit` alata koji se nalazi u poddirektorijumu `bin` spark direktorijuma.

\$ `spark-submit my_script.py`

Spark-submit je skripta koja postavlja okruženje tako da Spark Python API može da funkcioniše. Kada je neka naša skripta povezana sa Sparkom potrebno je importovati Spark pakete u program i kreirati `SparkContext`. To se postiže tako što se prvo kreira `SparkConf` objekat koji konfiguriše aplikaciju i onda se kreira `SparkContext` preko `SparkConf` objekta.

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

Primer pokazuje šta je najmanje potrebno da bi se inicijalizovao `SparkContext`.

- `setMaster()` kao argument prima URL klastera. Ako je argument `"local"` Spark se nalazi na lokalnoj mašini.
- `setAppName` kao argument prima ime aplikacije. Ovo ime se koristi kao identifikator aplikacije na klasteru.

Programiranje sa RDD

Sav posao koji Spark obavlja, ugrubo, možemo podeliti u tri kategorije:

1. kreiranje novih RDD-ova,
2. transformisanje postojećih i
3. pozivanje operacija nad RDD-om da bi se preračunao rezultat.

U pozadini, Spark automatski distribuira podatke koji se nalaze u RDD-u na čvorove klastera i paralelizuje operacije koje se izvršavaju nad njim.

Korisnik može da kreira RDD na dva načina – **distribuiranjem kolekcije objekata** (liste ili skupa) ili **učitavanjem spoljnog skupa podataka** u driver program. U prethodnom primeru je prikazano učitavanje podataka iz tekstualnog fajla korišćenjem funkcije `SparkContext.textFile()`.

Jednom kreiran, RDD nudi dve vrste operacija: **transformacije** i **akcije**. Transformacije konstruišu novi RDD iz prethodnog. Primer je korišćenje f-je `filter()`. Akcije preračunavaju rezultat na osnovu RDD i rezultat vraćaju ili u driver program ili u spolji uređaj za smeštanje podataka. Primer je f-ja `first()`.

Transformacije i akcije se razlikuju po načinu na koji ih Spark tretira. Iako je moguće definisati novi RDD u bilo kom trenutku, Spark preračunava RDD u lenjom obliku (**lazy**) tj. kreira se RDD tek kada je nad njim prvi put izvršena akcija. Ovaj pristup, iako čudan na prvi pogled, u stvari ima dosta smisla. Analizirajmo primer koji smo prethodno koristili (učitavanje tekstualnog fajla i filtriranje linija koje sadrže reč Python). Ako bi Spark učitao i uskladištio podatke u trenutku kada napišemo `lines = sc.textFile()` potrošio bi prilično prostora budući da nakon toga odmah koristimo filtriranje koje eliminiše mnogo linija. Spark, jednom kada vidi ceo niz transformacija, može da preračuna koji podaci su potrebni za krajnji rezultat i samo njih da učita. Ako koristimo `first()` akciju, Spark će da skenira fajl samo dok ne pronađe prvu odgovarajuću liniju, ne čitajući ostatak fajla.

Spark RDD po default-u se preračunava svaki put kada se izvrši akcija nad njim. Ako želimo da iskoristimo isti RDD nad više akcija možemo da zamolimo Spark da zadrži RDD u memoriji koristeći `RDD.persist()`. Nakon prve akcije Spark će da zadrži RDD sadržaj u memoriji (particionisan na više mašina u klasteru) i koristiće ga u budućim akcijama. Zadržavanje RDD je moguće i na disku umesto u memoriji.

Najjednostavnije moguće je kreirati RDD iz postojeće kolekcije u programu koju je potrebno proslediti u `parallelize()` f-ju `SparkContext`-a.

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Način koji se više koristi je kreiranje RDD-a učitavanjem podataka iz sekundarne memorije. Spark podržava širok opseg ulaznih i izlaznih izvora. Može da pristupi podacima preko `InputFormat` i `OutputFormat` interfejsa koji koristi i Hadoop MapReduce model i koji koristi mnoge fajl formate i sisteme za skladištenje podataka (S3, HDFS, Cassandra, Hbase isl.).

Za podatke koji se nalaze u lokalnom ili distribuiranom fajl sistemu, npr. HDFS, NFS ili Amazon S3, Spark može da pristupi različitim tipovima fajlova uključujući tekstualne, JSON, `SequenceFiles` i protokol baferima. Spark SQL modul obezbeđuje API za struktuiranje izvora podataka uključujući JSON i Apache Hive. Takođe, za baze podataka i ključ/vrednost skladišta postoje biblioteke koje se povezuju na Cassandra, Hbase, Elasticsearch i JDBC bazu.

Ime formata	Strukturirani	Komentar
Tekstualni fajl	Ne	Linije predstavljaju slogove podataka.
JSON	Delimično	Veći biblioteka zahteva jedan slog po liniji.
CSV	Da	
SequenceFiles	Da	Hadoop fajl format koji koristi key/value podatke
Protocol Buffers	Da	Brz, višejezični format
Object fajl	Da	Koristan za pamćenje iz Spark poslova koji koriste zajednički kod. Oslanja se na Java Serializaciju.

Više o kreiranju RDD-ova, akcijama i transformacijama, možete pronaći na sledećem linku:

<https://www.tutorialspoint.com/pyspark/index.htm>

Kubernetes

Kubernetes je alat otvorenog koda koji služi za upravljanje visoko skaliranih kontejnerskih okruženja (u ovom slučaju Docker kontejnera), razmeštanje aplikacija, isporučivanje aplikacija, automatsko skaliranje, monitoring itd. Kubernetes je napravljen od strane Google inženjera kako bi upravljali ogromnim brojem aplikacija i njihovih kontejnera, kao i za primenu automatizacije procesa.

Nakon nekog vremena, dodeljen je Cloud Native Computing Foundation (CNCF), delu Linux fondacije, radi daljih unapređenja.

Kubernetes je u osnovi sistem koji se bavi svim resursima u klasteru i pomaže programerima i administratorima sistema da upravljaju razvojem aplikacija i kontejnerima u Cloud okruženju.

Iako su ključni koncepti i konfiguracije Kubernetes-a dobro dokumentovani na zvaničnom sajtu, svakom početniku nije baš lak zadatak da konfiguriše i upravlja Kubernetes klasterima na svom lokalnom okruženju, kao na primer kroz upotrebu Virtuelnih mašina i Minikuba.

Kubernetes i Minikube

Minikube je lokalni Kubernetes, koji se fokusira na to da Kubernetesu olakša učenje i razvoj.

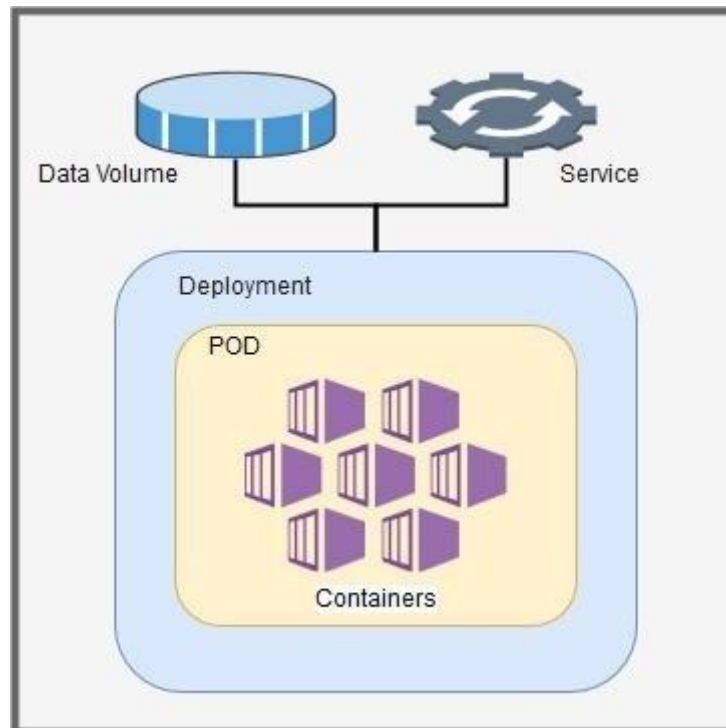
Sve što vam treba je Docker (ili slično kompatibilan) kontejner ili okruženje virtuelne mašine, a Kubernetes je udaljen samo jednom naredbom: **minikube start**.

Zbog složenosti konfiguracije i upravljanja samog Kubernetesa, velike hosting i cloud kompanije poput [Google Cloud Platform](#), [Amazon Web Services \(AWS\)](#), [Microsoft Azure Cloud](#), pokrenuli su sopstvene servise kako bi se bavili Kubernetes klasterima.

Ovi servisi su vrlo korisni za upravljanje Kubernetes konfiguracijama i klasterima.

U nekom od narednih tekstova objasnićemo kako Kubernetes radi na AWS Cloudu.

Osnovni pojmovi Kuberbetesa



Kao što je prikazano na dijagramu, čvorovi (Nodes), količine podataka (Volume), usluge (Service), razmeštanja (Deployment), podovi (Pod) i kontejneri su neki od najvažnijih delova Kuberbetes klastera.

Node: Čvor ili Node je radna mašina (npr na Cloud Serveru “AWS Elastic Compute Cloud – EC2”) Kuberbetes klastera, koja pokreće vaše kontejnere i aplikacije.

Container: Kontejner je da kažemo lagani softverski skup s potrebnim paketima i programskim zavisnicima za pokretanje aplikacije.

Ukratko, kontejner je mesto gde se nalaze sve aplikacije koje trenutno razvijate, a koje bi trebalo da se prebace na server ili da se pokrenu na neki drugi način.

Pod: Pod je jedna od najvažnijih i osnovnih jedinica kojom Kuberbetes upravlja. Pod upravlja grupom jednog ili više kontejnera raspoređenih na radnom čvoru (Node) klastera. Kuberbetes klasteri mogu imati više Pod-ova i svaki Pod ima jedinstvenu IP adresu za povezivanje.

Service: Konfigurišući servise na klasteru, možemo otvoriti pristup aplikacijama i kontejnerima koji rade na različitim Pod-ovima na klasteru ili dodati pristup kontejnerima aplikacija izvan mreže klastera.

Deployment: raspoređivanje fajlova je skup uputstava za kreiranje Pod-ova i drugih potrebnih stvari u vašoj aplikaciji. Datoteke koje se raspoređuju su slične datotekama tzv. komposer fajlova iz Docker okruženja.

Ovo ćemo objasniti u sledećem tekstu, kako ona izgleda i koliko je bitna za rad na Kubernetesu.

Takođe, možete koristiti Deployment za specifičan broj Pod-ova, odnosno njihovih replika tako što ćete sve uredno upisati u fajl, kako bi se dale instrukcije za implementaciju na Kubernetesu.

Deployment takođe nadzire klaster radi održavanja definisanih specifikacija za Pod-ove, čak i ako se ručno izbriše bilo koji Pod, konfigurisana implementacija će se pobrinuti da se odmah stvori nova struktura kako je definisana u fajlu.


Volume: Kubernetes volume je jedinica koja može biti konfigurisana da obezbedi pristup vašoj kontejnerskoj aplikaciji koja je pokrenuta unutar Poda na klasteru.

Ovim volumenima može se priložiti količina radnih čvorova (Nodova), koji se skladište na Cloudu poput Amazon Elastic Block Store (EBS), Elastični sistem datoteka (EFS) itd., kao i na listi Kubernetes servisa za skladištenje podataka, koje možete pronaći na Kubernetes veb lokaciji.

Kubectl: Kubectl je komandna linije za upravljanje okruženjem klastera. To može biti bilo koji sistem van okruženja klastera. Potrebno je samo da se instaliraju osnovni paketi i konfiguracije da biste komunicirali sa klasterom. Jednom kada je vaš sistem konfigurisan, možete izvršiti sve operacije u vašem klasteru iz komandne linije **kubectl**.

Svrha, i problemi koje Kubernetes rešava

Tehnologija kontejnerizacije aplikacija je u velikoj meri promenila oblast IT infrastrukture i isporuke softvera u prethodnih nekoliko godina. Pomenutoj temi i prednostima koju je doneo ovakav način pakovanja i distribucije aplikacija je u prethodnom periodu posvećeno više tekstova u *Business&IT* časopisu, gde su između ostalog obrađivane teme Docker-a (br. 5) i Windows kontejnera (br. 7). Dok su Docker i druge kontejner platforme rešile problem pakovanja aplikacija i pokretanja pojedinačnih kontejnera na jednom serveru, pojavila se potreba za platformom koja omogućava upravljanje većim brojem kontejnera na klasteru servera – tzv. orkestratorom kontejnera.



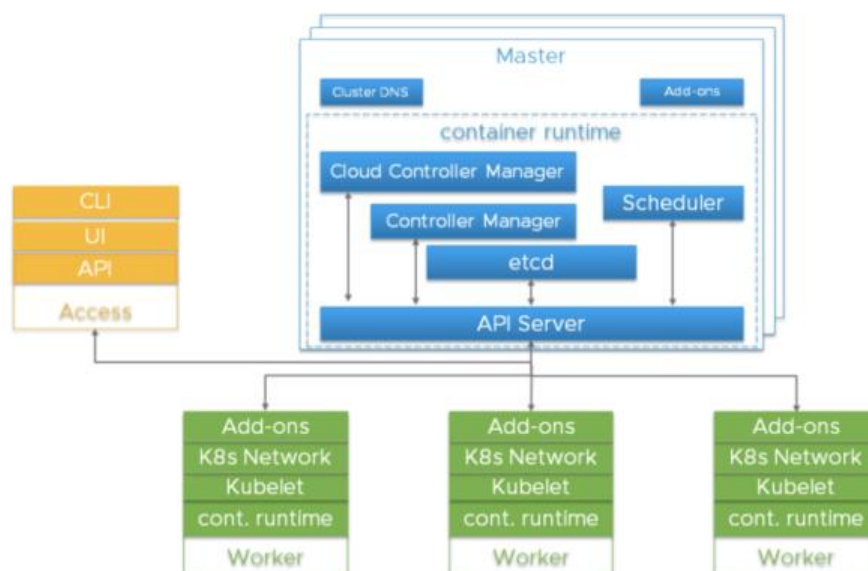
Kao dominantno rešenje na ovom polju se vremenom izdvojio Kubernetes, projekat koji je potekao iz Google-a koji je 2014. godine doneo odluku da ga javno objavi u vidu otvorenog koda. Google je u međuvremenu predao Kubernetes na upravljanje Cloud Native Computing Fondaciji (CNCF), osnovanoj sa ciljem koordinisanja svih projekata i integracija u cloud native ekosistemu, kojih trenutno ima preko 1,000. Istorija je pokazala da su obe odluke bile odlične, jer danas Kubernetes svom uspehu može da zahvali upravo jakoj organizaciji koja stoji iza njega i snazi *open source* zajednice koja ga razvija.

Kako objasniti šta je Kubernetes u jednoj rečenici? Ukratko, Kubernetes omogućava da se u programskom kodu kompletno opišu svi elementi jedne aplikacije spakovane u kontejnere i infrastruktura potrebna za njeno funkcionisanje. Pomenuti kod pisan u YAML jeziku se izdaje Kubernetes klasteru, koji čine Linux ili Windows serveri podeljeni u dve grupe: master nodovi koji se brinu o stanju klastera i kontejnera koji su mu povereni na izvršavanje, i worker nodovi (radnici) na kojima se izvršavaju sami kontejneri. Zadatak klastera je da se brine o kontejnerima i pratećoj infrastrukturi: da obezbedi svakom kontejneru nod na kome će se izvršavati i zahtevane resurse, monitoriše njihovo stanje i restartuje ih po potrebi, smanjuje i povećava njihov broj u zavisnosti od opterećenja, konfiguriše mrežu kako bi kontejneri mogli da komuniciraju međusobno i sa spoljnim svetom itd.

Pre Kubernetes-a, infrastruktura nikada nije bila tako blizu developerima, koji sada mogu direktno konzumirati i konfigurisati resurse klastera. Kubernetes im omogućava da definišu sve gradivne elemente jedne aplikacije, počev od

pojedinačnih kontejnera, preko diskovnog prostora za smeštanje perzistentnih podataka, sve do pravila za mrežno oglašavanje i balansiranje saobraćaja. Nakon inicijalnog deployment-a aplikacije, mogu nadalje upravljati njenim životnim vekom – Kubernetes omogućava automatizaciju procesa objavljivanja nove verzije aplikacije u “kotrljajućem” maniru, gde se postojeći kontejneri postepeno zamenjuju novim. I sve to kroz jezik koji se uklapa u savremene trendove definisanja infrastrukture u kodu, kao što su deklarativnost (kod taksativno definiše željeno krajnje stanje koje je potrebno postići) i idempotentnost (svako izvršavanje istog koda daje isto krajnje stanje), omogućavajući efikasnu integraciju Kubernetes platforme u postojeće CI/CD procese.

Kao što nas je tehnologija virtualizacije zauvek oslobodila zavisnosti od fizičkog hardvera, tako Kubernetes ide korak dalje i praktično nas oslobađa zavisnosti od virtualizacionih i cloud platformi. Šta se nalazi ispod više nije toliko važno, jer se aplikacija definisana Kubernetes jezikom može izvršavati gde god da postoji Kubernetes, što garantuje portabilnost aplikacija i konzistentnost u njihovom radu na različitim infrastrukturama. A Kubernetes je danas moguće pokrenuti praktično svuda – od IoT uređaja i fizičkih servera, preko virtualizacionih platformi poput VMware-a, do cloud provajdera kao što su AWS i Microsoft Azure.



Postoji više načina kako doći do operativnog Kubernetes klastera, pri čemu je njegova manuelna instalacija od nule rezervisana samo za najhrabrije. Na jednoj strani spektra su *open source* alati kao što su *Kubespray* i *Rancher* koji olakšavaju kreiranje i upravljanje klasterom na različitim on-premise i cloud platformama. Na potpuno drugoj strani su hostovana Kubernetes rešenja kod

kojih je kompletna automatizacija upravljanja životnim vekom klastera odgovornost cloud provajdera, poput Azure Kubernetes Service (AKS) i Amazon Elastic Kubernetes Service (EKS).

I negde između stoje enterprise distribucije Kubernetes-a: *Red Hat OpenShift* i *VMware vSphere with Kubernetes*. VMware je dakle doneo odluku da integriše Kubernetes u svoju vSphere virtualizacionu platformu, čime su ESXi hostovi postali i radnici Kubernetes klastera, a kontejneri ravnopravni virtuelnim mašinama. Tako će od vSphere-a verzije 7 biti moguće koristiti Kubernetes jezik ne samo za opisivanje servisa u kontejnerima, već i VMware virtuelnih mašina, što predstavlja još jedan dokaz tvrdnje da je Kubernetes postao defakto standard za infrastrukturni API.

Ako je ovo infrastruktura sadašnjosti i budućnosti, postavlja se pitanje da li treba krenuti u migraciju postojećih aplikacija na Kubernetes, kao i da li su uopšte sve aplikacije i servisi dobri kandidati za Kubernetes. Treba imati na umu da unapređenja i mogućnosti koja Kubernetes donosi imaju za cenu njegovu kompleksnost u pogledu upravljanja i korišćenja, koju nije uvek opravdano plaćati. Jednostavne aplikacije i servisi koji ne poseduju distribuiranu arhitekturu (npr. tradicionalne relacione baze podataka) ne dobijaju mnogo prelaskom na novu platformu, dok najviše koristi od Kubernetes-izacije imaju aplikacije koje su distribuirane po svojoj prirodi i dizajnirane kao skup mikroservisa.

Projektni zadatak – pokretanje Apache Spark posla na Kubernetes klusteru

Minikube priprema

Sada kada smo se upoznali sa osnovama Apache Spark-a i Kubernetes-a, možemo preći na praktičnu implementaciju. U ovom primeru koristićemo lokalni Kubernetes klaster, **Minikube**, koji smo ranije pomenuli.

Pre nego što predjemo na pokretanje Spark posla, potrebno je instalirati Minikube i neko izolovano okruženje (Docker ili drugo po izboru).

Instalacije i uputstva možete pronaći na sledećim linkovima:

<https://minikube.sigs.k8s.io/docs/start/>

<https://docs.docker.com/docker-for-windows/install/>

Nakon instalacije, vršimo proveru da li je ista uspela, tako što u komandnoj liniji unesemo:

1. `docker - -version`
2. `minikube version`
3. `kubectl version - -client`

Kubectl je alatka komandne linije koja služi za upravljanje Kubernetes klasterima. U našem slučaju, ona se automatski instalirala prilikom instalacije Minikube-a.

Sledeći korak jeste da podesimo i pokrenemo naš Minikube klaster. Možemo naglasiti na koliko memorije i jezgara želimo da radi. U komandnoj liniji unesemo sledeće komadne:

```
minikube config set memory 8192
```

```
minikube config set cpus 4
```

Nakon toga, dozvolimo podrazumevanom korisničkm nalogu: default pristup podrazumevanom prostoru imena (default namespace):

```
kubectl create clusterrolebinding default --clusterrole=edit --serviceaccount=default:default --namespace=default
```

Konačno, možemo da pokrenemo Minikube:

```
minikube start
```

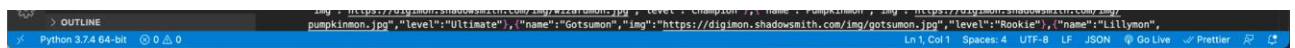
Prisetimo se, prilikom pokretanja Apacher Spark posla, potrebno je da naglasimo URL master-a, odnosno klastera na kojem će se posao izvršiti (ili “local”). Da bismo znali adresu našeg Minikube klastera, pokrenućemo komandu:

```
minikube ip
```

PySpark program

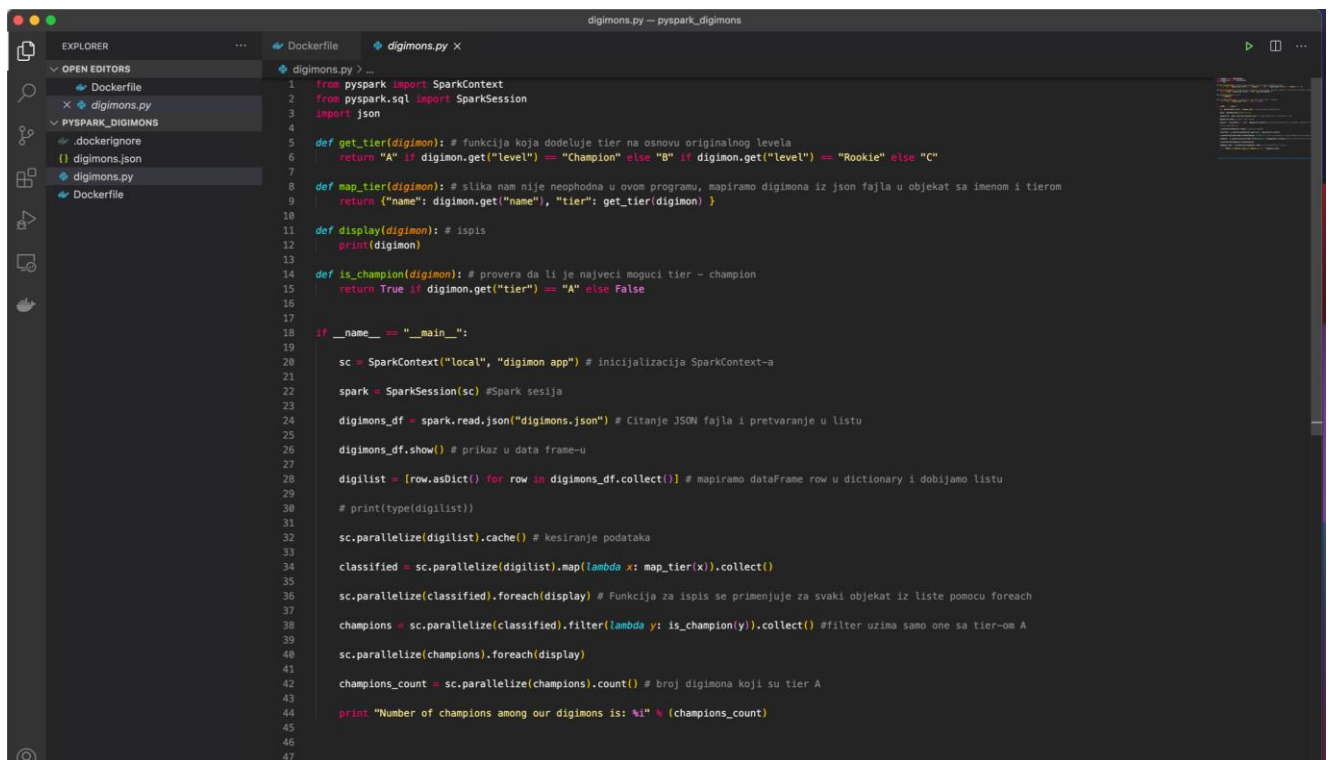
Sada ćemo pogledati našu Python skriptu koja će izvršiti Apache Spark posao koristeći PySpark biblioteku.

Najpre imamo fajl “**digimons.json**” koji sadrži JSON niz, to jest podatke o velikom broju digimona, i za svakog od njih čuva ime, link do slike, i nivo (level).



Kada pogledamo ovaj skup podataka, ono što bi najviše zanimalo svakog od nas jeste da vidimo koji su to “najjači” digimoni. Atribut *level* je nominalne prirode i svrstava sve digimone u neku od kategorija “In Training”, “Rookie” ili “Champion”. Ovo je na prvi pogled u redu, ali bi bilo lepše kada bi se te klase mogle lakše pročitati ili zapisati.

Cilj našeg PySpark programa je da pomoću RDDs-a koje smo ranije pomenuli, akcija i transformacija mapira sve digimone iz skupa u “A”, “B” ili “C” kategoriju, ukloni atribut slika i filtrira niz, tako da nam na kraju ostani samo najmoćniji digimoni, oni koji pripadaju “A” klasi.



```
1 from pyspark import SparkContext
2 from pyspark.sql import SparkSession
3 import json
4
5 def get_tier(digimon): # funkcija koja dodeljuje tier na osnovu originalnog levela
6     return "A" if digimon.get("level") == "Champion" else "B" if digimon.get("level") == "Rookie" else "C"
7
8 def map_tier(digimon): # slika nam nije neophodna u ovom programu, mapiramo digimona iz json fajla u objekat sa imenom i tierom
9     return {"name": digimon.get("name"), "tier": get_tier(digimon)}
10
11 def display(digimon): # ispis
12     print(digimon)
13
14 def is_champion(digimon): # proverava da li je najveći mogući tier - champion
15     return True if digimon.get("tier") == "A" else False
16
17 if __name__ == "__main__":
18
19     sc = SparkContext("local", "digimon app") # inicijalizacija SparkContext-a
20
21     spark = SparkSession(sc) # Spark sesija
22
23     digimons_df = spark.read.json("digimons.json") # Citanje JSON fajla i pretvaranje u listu
24
25     digimons_df.show() # prikaz u data frame-u
26
27     digilist = [row.asDict() for row in digimons_df.collect()] # mapiramo dataframe row u dictionary i dobijamo listu
28
29     # print(type(digilist))
30
31     sc.parallelize(digilist).cache() # keširanje podataka
32
33     classified = sc.parallelize(digilist).map(lambda x: map_tier(x)).collect()
34
35     sc.parallelize(classified).foreach(display) # Funkcija za ispis se primenjuje za svaki objekat iz liste pomocu foreach
36
37     champions = sc.parallelize(classified).filter(lambda y: is_champion(y)).collect() # filter uzima samo one sa tier-om A
38
39     sc.parallelize(champions).foreach(display)
40
41     champions_count = sc.parallelize(champions).count() # broj digimona koji su tier A
42
43     print("Number of champions among our digimons is: %i" % (champions_count))
44
45
46
47
```

Na početku uvozimo sve biblioteke koje su nam potrebne – *SparkContext* iz *pyspark*, *SparkSession* iz *pyspark.sql* i *json*.

Zatim, imamo definisane 4 funkcije:

- **get_tier** – prima digimon objekat kao argument, i na osnovu njegovog level-a vraća “A”, “B” ili “C” - nove klase koje smo definisali.
- **map_tier** - prima digimon objekat kao argument i vraća objekat (dictionary) koji sadrži ime digimona, i generisani atribut “tier” na osnovu prve funkcije.
- **display** – ispisuje digimona
- **is_champion** – proverava da li je dati digimon “A” klase

Sada prelazimo u *main* funkciju. Inicijalizujemo *SparkContext*, a zatim *SparkSession* koja prima kontekst kao paramtera.

Na liniji 24 čitamo “*digimons.json*” datoteku i njen sadržaj čuvamo u *DataFrame* objektu, a na liniji 25. Prikazujemo taj *DataFrame*.

Na liniji 28 Konvertujemo *DataFrame* objekat u listu, kako bi kasnije primenili *Spark* akcije i transformacije nad njom.

Na liniji 32 implementirano je keširanje, tako da je svaki sledeći pristup ovim podacima brži.

Na liniji 34. koristimo **map()** transformaciju Apache Spark-a, kako bi svakog digimona iz datoteke transformisali u oblik koji nam odgovara pomoću već pomenute funkcije **map_tier**.

Sada, na liniji 36 prikazujemo objekte dobijene transformacijom.

Na 38. liniji primenjena je druga transformacija – **filter()**, koja će izdvojiti sve digimone “A” klase iz liste, pomoću naše funkcije **is_champion**.

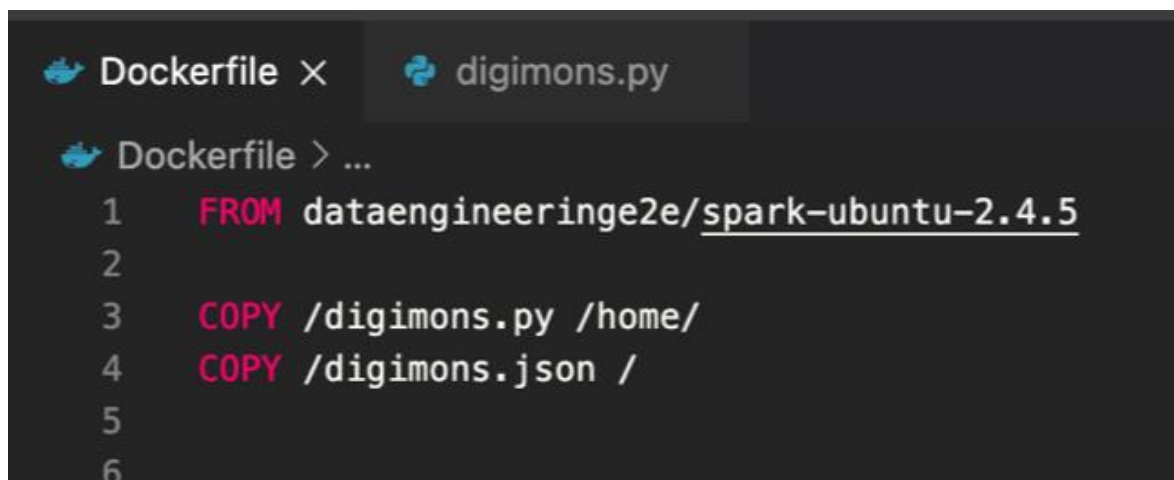
Linija 40 je ista kao 36, gde se prikazuju objekti u konzoli, ovaj put novi filtrirani podaci.

Na liniji 42. primenjujemo akciju **count()** i kao rezultat dobijamo broj digimona koji su klase “A”, a na 44. Ispisujemo isti.

Napomena: Prilikom svake transformacije nad listom, neophodno je pozvati funkciju **collect()** koja formira listu od transformisanih podataka.

Izolovanje aplikacije – Docker

Da bismo mogli nas program da pokrenemo kao Apache Spark posao na Minikube klasteru, potrebno je da izradimo Docker “image” (sliku) - [Dockerfile](#).



```
Dockerfile x digimons.py
Dockerfile > ...
1 FROM dataengineeringe2e/spark-ubuntu-2.4.5
2
3 COPY /digimons.py /home/
4 COPY /digimons.json /
5
6
```

Za ovaj zadatak korišćićemo postojeću Docker sliku - **dataengineeringe2e/spark-ubuntu-2.4.5**, koja u sebi sve potrebne zavisnosti kako bi se izvršio neki Apache Spark posao.

Jedini dodatak koji trebada uradimo jeste kopiranje JSON datoteke i Python skripte u odgovarajuće direktorijum slike.

Pre nego što od ove slike izradimo Docker *container* (izolovano okruženje), neophodno je da predjemo u Minikube okruženje. To radimo sledećom komandom:

```
eval $(minikube docker-env)
```

```
docker image ls
```

Sada, iz direktorijuma gde se nalazi naš Dockerfile, pokrenimo sledeću komandu:

```
$ docker build -t digimons --no-cache .
```

Ovim smo pokrenuli Docker *container* (izolovano okruženje), a proveru možemo izvršiti komandom: *docker image ls*.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
digimons	latest	092e56ce8e98	10 seconds ago	2.1GB
k8s.gcr.io/kube-proxy	v1.20.2	43154ddb57a8	3 months ago	118MB
k8s.gcr.io/kube-controller-manager	v1.20.2	a27166429d98	3 months ago	116MB
k8s.gcr.io/kube-apiserver	v1.20.2	a8c2fdb8bf76	3 months ago	122MB
k8s.gcr.io/kube-scheduler	v1.20.2	ed2c44fbdd78	3 months ago	46.4MB
kubernetesui/dashboard	v2.1.0	9a07b5b4bfac	4 months ago	226MB
gcr.io/k8s-minikube/storage-provisioner	v4	85069258b98a	4 months ago	29.7MB
k8s.gcr.io/etcd	3.4.13-0	0369cf4303ff	7 months ago	253MB
dataengineeringe2e/spark-ubuntu-2.4.5	latest	ae827496ae3e	8 months ago	2.1GB
k8s.gcr.io/coredns	1.7.0	bfe3a36ebd25	10 months ago	45.2MB
kubernetesui/metrics-scraper	v1.0.4	86262685d9ab	13 months ago	36.9MB
k8s.gcr.io/pause	3.2	80d28bedfe5d	14 months ago	683kB

Prikazana je lista svih Docker slika, a na vrhu se nalazi upravo kreirana **digimons**.

Sve pripreme se završene, sada možemo pokrenuti naš Apache Spark program na Minikube klasteru.

Predjimo u direktorijum gde je instaliran Spark lokalno. (npr. */Users/Username/spark*)

Sada predjimo u **bin** direktorijum komandom: **cd bin**

Ovo je neophodno da bi nas komanda **spark-submit** bila dostupna.

Sada treba da pokrenemo sledeću komandu:

```
./spark-submit \  
--master k8s://$MINIKUBE_IP \  
--deploy-mode cluster \  
--name pyspark-digimons \  
--class org.apache.spark.examples.SparkPi \  
--conf spark.executor.instances=2 \  
--conf spark.kubernetes.container.image=digimons:latest \  
local:///home/digimons.py
```

\$MINIKUBE_IP predstavlja IP adresu Minikube klastera koju smo ranije naveli.

Kao što je prikazano na slici, to je IP adresa klastera (master, k8s – Kubernetes oznaka), a tu su takodje atributi poput imena posla (pyspark-digimons), klase (podrazumevana Apache Spark klasa za primere), broj instanci (2), slika (image=digimon:latest) koju smo ranije prikazali i putanja do naše skripte (digimons.py).

Ako pokrenemo komandu: **kubectl get pods**, rezultat ce biti poput ovog na slici ispod:

NAME	READY	STATUS	RESTARTS	AGE
pyspark-digimons-3fa84078f0251eae-driver	0/1	Completed	0	7m4s

Vidimo ime našeg posla, i status – Completed, kao i vreme kada se izvršio.

Komandom **kubectl logs pyspark-digimons-3fa84078f0251eae-driver** dobijamo izlaz našeg programa, kao i dodatne informacije o izvršavanju Apache Spark posla:

```
bin -- -bash -- 137x70
{'name': 'Kuwagamon', 'tier': 'A'}
{'name': 'Greymon', 'tier': 'A'}
{'name': 'Shellmon', 'tier': 'A'}
{'name': 'Garurumon', 'tier': 'A'}
{'name': 'Seadramon', 'tier': 'A'}
{'name': 'Monochromon', 'tier': 'A'}
{'name': 'Birdramon', 'tier': 'A'}
{'name': 'Meramon', 'tier': 'A'}
{'name': 'Kabuterimon', 'tier': 'A'}
{'name': 'Togemon', 'tier': 'A'}
{'name': 'Numemon', 'tier': 'A'}
{'name': 'Ikkakumon', 'tier': 'A'}
{'name': 'Unimon', 'tier': 'A'}
{'name': 'Leomon', 'tier': 'A'}
{'name': 'Ogremon', 'tier': 'A'}
{'name': 'Devimon', 'tier': 'A'}
{'name': 'Frigimon', 'tier': 'A'}
{'name': 'Mojyamon', 'tier': 'A'}
{'name': 'Sukamon', 'tier': 'A'}
{'name': 'Centarumon', 'tier': 'A'}
{'name': 'Bakemon', 'tier': 'A'}
{'name': 'Angemon', 'tier': 'A'}
{'name': 'Whamon', 'tier': 'A'}
{'name': 'Drimogemon', 'tier': 'A'}
{'name': 'Kokatorimon', 'tier': 'A'}
{'name': 'Tyrannomon', 'tier': 'A'}
{'name': 'Vegiemon', 'tier': 'A'}
{'name': 'Gekomon', 'tier': 'A'}
{'name': 'Flymon', 'tier': 'A'}
{'name': 'Gatomon', 'tier': 'A'}
{'name': 'Nanimon', 'tier': 'A'}
{'name': 'Devidramon', 'tier': 'A'}
{'name': 'Dokugumon', 'tier': 'A'}
{'name': 'Gesomon', 'tier': 'A'}
{'name': 'Raremon', 'tier': 'A'}
21/04/20 16:37:58 INFO PythonRunner: Times: total = 44, boot = -27, init = 70, finish = 1
{'name': 'Wizardmon', 'tier': 'A'}
{'name': 'DarkTyrannomon', 'tier': 'A'}
{'name': 'Tuskmon', 'tier': 'A'}
{'name': 'Snimon', 'tier': 'A'}
{'name': 'Kiwimon', 'tier': 'A'}
{'name': 'RedVegiemon', 'tier': 'A'}
{'name': 'Mekanorimon', 'tier': 'A'}
{'name': 'Tankmon', 'tier': 'A'}
{'name': 'Vilemon', 'tier': 'A'}
{'name': 'Musyamon', 'tier': 'A'}
{'name': 'Starmon', 'tier': 'A'}
{'name': 'Hanumon', 'tier': 'A'}
{'name': 'Revolvermon', 'tier': 'A'}
{'name': 'BlueMeramon', 'tier': 'A'}
{'name': 'Gorillamon', 'tier': 'A'}
{'name': 'Veedramon', 'tier': 'A'}
{'name': 'Guardromon', 'tier': 'A'}
{'name': 'PlatinumSukamon', 'tier': 'A'}
{'name': 'ModokiBetamon', 'tier': 'A'}
{'name': 'Saberdramon', 'tier': 'A'}
{'name': 'Icemon', 'tier': 'A'}
{'name': 'Airdramon', 'tier': 'A'}
{'name': 'Akatorimon', 'tier': 'A'}
{'name': 'Geremon', 'tier': 'A'}
{'name': 'FlareRizamon', 'tier': 'A'}
{'name': 'Thunderballmon', 'tier': 'A'}
{'name': 'Soulmon', 'tier': 'A'}
{'name': 'Piddomon', 'tier': 'A'}
{'name': 'ExVeemon', 'tier': 'A'}
{'name': 'Stingmon', 'tier': 'A'}
{'name': 'Aquillamon', 'tier': 'A'}
{'name': 'Ankylomon', 'tier': 'A'}
21/04/20 16:37:58 INFO Executor: Finished task 0.0 in stage 6.0 (TID 6). 1418 bytes result sent to driver
21/04/20 16:37:58 INFO TaskSetManager: Finished task 0.0 in stage 6.0 (TID 6) in 48 ms on localhost (executor driver) (1/1)
```

```
bin -- -bash -- 137x70
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 147
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 156
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 188
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 189
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 150
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 145
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 179
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 174
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 165
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 155
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 148
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 141
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 158
21/04/20 16:37:59 INFO SparkContext: Starting job: count at /home/digimons.py:42
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 178
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 182
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 171
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 161
21/04/20 16:37:59 INFO DAGScheduler: Got job 7 (count at /home/digimons.py:42) with 1 output partitions
21/04/20 16:37:59 INFO DAGScheduler: Final stage: ResultStage 7 (count at /home/digimons.py:42)
21/04/20 16:37:59 INFO DAGScheduler: Parents of final stage: List()
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 162
21/04/20 16:37:59 INFO DAGScheduler: Missing parents: List()
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 176
21/04/20 16:37:59 INFO DAGScheduler: Submitting ResultStage 7 (PythonRDD[21] at count at /home/digimons.py:42), which has no missing parents
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 154
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 183
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 163
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 169
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 153
21/04/20 16:37:59 INFO MemoryStore: Block broadcast_10 stored as values in memory (estimated size 5.7 KB, free 413.6 MB)
21/04/20 16:37:59 INFO MemoryStore: Block broadcast_10_piece0 stored as bytes in memory (estimated size 3.9 KB, free 413.6 MB)
21/04/20 16:37:59 INFO BlockManagerInfo: Added broadcast_10_piece0 in memory on pyspark-digimons-3fa84078f0251eae-driver-svc.default.svc:7079 (size: 3.9 KB, free: 413.9 MB)
21/04/20 16:37:59 INFO SparkContext: Created broadcast 10 from broadcast at DAGScheduler.scala:1163
21/04/20 16:37:59 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 7 (PythonRDD[21] at count at /home/digimons.py:42) (first 15 tasks are for partitions Vector(0))
21/04/20 16:37:59 INFO TaskSchedulerImpl: Adding task set 7.0 with 1 tasks
21/04/20 16:37:59 INFO TaskSetManager: Starting task 0.0 in stage 7.0 (TID 7, localhost, executor driver, partition 0, PROCESS_LOCAL, 9824 bytes)
21/04/20 16:37:59 INFO Executor: Running task 0.0 in stage 7.0 (TID 7)
21/04/20 16:37:59 INFO BlockManagerInfo: Removed broadcast_8_piece0 on pyspark-digimons-3fa84078f0251eae-driver-svc.default.svc:7079 in memory (size: 3.2 KB, free: 413.9 MB)
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 187
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 149
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 144
21/04/20 16:37:59 INFO BlockManagerInfo: Removed broadcast_9_piece0 on pyspark-digimons-3fa84078f0251eae-driver-svc.default.svc:7079 in memory (size: 4.3 KB, free: 413.9 MB)
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 181
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 164
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 173
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 159
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 167
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 170
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 172
21/04/20 16:37:59 INFO ContextCleaner: Cleaned accumulator 190
21/04/20 16:37:59 INFO PythonRunner: Times: total = 43, boot = -33, init = 76, finish = 0
21/04/20 16:37:59 INFO Executor: Finished task 0.0 in stage 7.0 (TID 7). 1418 bytes result sent to driver
21/04/20 16:37:59 INFO TaskSetManager: Finished task 0.0 in stage 7.0 (TID 7) in 50 ms on localhost (executor driver) (1/1)
21/04/20 16:37:59 INFO TaskSchedulerImpl: Removed TaskSet 7.0, whose tasks have all completed, from pool
21/04/20 16:37:59 INFO DAGScheduler: ResultStage 7 (count at /home/digimons.py:42) finished in 0.057 s
21/04/20 16:37:59 INFO DAGScheduler: Job 7 finished: count at /home/digimons.py:42, took 0.058831 s
67
21/04/20 16:37:59 INFO SparkContext: Invoking stop() from shutdown hook
21/04/20 16:37:59 INFO SparkUI: Stopped Spark web UI at http://pyspark-digimons-3fa84078f0251eae-driver-svc.default.svc:4040
21/04/20 16:37:59 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
21/04/20 16:37:59 INFO MemoryStore: MemoryStore cleared
21/04/20 16:37:59 INFO BlockManager: BlockManager stopped
21/04/20 16:37:59 INFO BlockManagerMaster: BlockManagerMaster stopped
```

Literatura:

<https://spark.apache.org/docs/latest/running-on-kubernetes.html>

<https://www.tutorialspoint.com/pyspark/index.htm>

<https://digimon-api.herokuapp.com>

<https://medium.com/@andreyonistchuk/how-to-run-spark-2-4-on-osx-minikube-f0e5fdeb27be>