Izgradnja skalabilne arhitekture za paralelno procesiranje velikih količina podataka

Majnarić, Ivan

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:387654

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-04-25**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Majnarić

Izgradnja skalabilne arhitekture za paralelno procesiranje velikih količina podataka

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Majnarić
Matični broj: 45300/16-R
Studij: Baze podataka

Izgradnja skalabilne arhitekture za paralelno procesiranje velikih količina podataka

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Kornelije Rabuzin

Varaždin, srpanj 2018.

Ivan Majnarić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sadržaj

1. Uvod 1

2. Opis problema 2

2.1. Big Data 3

2.2. Real-time sustavi 4

2.3. Gotovo real-time sustav 6

3. Lambda arhitektura 7

3.1. Batch sloj 9

3.2. Sloj posluživanja 11

3.3. Brzi sloj 12

3.4. Batch sloj i sloj posluživanja zadovoljavaju sva svojstva 15

3.5. Zadnji trendovi u tehnologiji 17

3.6. Batch u odnosu na real-time obradu 18

3.7. Infrastruktura analitike za veliku količinu podataka 21

4. Opis arhitekture 22

4.1. Sistem arhitektura 23

4.2. Arhitektura servisa 23

4.2.1. Confluent 24

4.2.2. Apache Ambari i Hortonworks platforma 25

4.2.3. Apache Kafka 26

4.2.3.1. Teme i logovi 28

4.2.3.2. Distribucija 29

4.2.3.3. Potrošači 30

4.2.3.4. Kafka kao sustav razmjene poruka 30

5. Implementacija arhitekture 32

5.1. Apache Zookeeper 32

5.2. Pokretanje Apache Kafke 35

5.3. Registar shema 36

5.4. Pokretanje Kafka-Connect klastera 38

5.5. Landoop korisnička sučelja 40

5.6. Cassandra 43

5.7. Apache Hadoop 44

5.8. Apache Hive 44

5.9. Trenutni izgled arhitekture 45

6. Obrada podataka 47

6.1. Priprema podataka za streaming sloj 52
6.2. Priprema podataka za batch sloj 53
6.3. Osvrt na krajnje rješenje 54
7. Zaključak 56
8. Literatura 57

1. Uvod

Svako veće poduzeće ne samo da je u doticaju s podacima, već je i vođeno istima. Podaci konstantno pristižu u sustave te se iz istih pokušavaju kreirati informacije koje bi pridonijele što učinkovitijoj analizi. Činjenica je da svaka aplikacija kreira neku vrstu podataka bilo da su to zapisi, informacije o korisničkoj aktivnosti, neka vrsta izlaznih obogaćenih informacija ili slično. Svaki bajt podataka je priča koja govori za sebe, nešto bitno što će na koncu promijeniti određeni tok ili izvijestiti o sljedećoj radnji koju je potrebno odraditi. Kako bi se moglo znati što točno taj bajt želi reći potrebno je isti negdje pohraniti na mjesto gdje će se moći analizirati. Primjer se nalazi na internet stranicama kao što su Booking, Amazon, Ebay. Klikovi korisnika na određene produkte za koje su zainteresirani te koji se nalaze na navedenim stranicama pretvaraju se u niz reklama koje se kasnije prikazuju na nekim društvenim ili sličnim mrežama u čemu je upravo poanta, a i izvor prihoda. Što god se ta obrada podataka brže odradi, organizacija može brže reagirati na moguće anomalije. Također, što se manje vremena provede na premještanje i obradu podataka, više se vremena može iskoristiti na srž poslovnog modela. Upravo zato je jako bitno postaviti arhitekturu čiji će softveri ispunjavati svrhu. Pošto se ne radi samo o jednom bajtu već zapravo o velikoj količini podataka potrebno je imati stabilnu infrastrukturu gdje će se precizno odabrati samo određeni softveri koji će služiti svrsi.

Kao što navodi N. deGrasse Tyson:

„Svaki put kada se kao znanstvenici ne složimo, razlog neslaganja zapravo leži u nedostatku podataka. Tada se složimo o tome kakvi nam podaci zaista trebaju; pribavimo podatke; te se problem s podacima riješi. Ili sam ja u pravu, ili si ti u pravu, ili smo zapravo oboje u krivu te krenemo dalje.“

Ovaj rad se svodi upravo na ono što ovaj citat ističe. U nastavku će se pokušati izgraditi skalabilna arhitektura koja će omogućavati da podaci prođu cijeli ciklus obrade i prijenosa što je moguće brže uz osiguranje od gubitka istih. Cilj će također biti i osiguravanje otpornosti na ispade te mogućnosti balansiranja aplikacija koje se postave na sustav tako da se maksimalno iskoriste resursi koji su joj dani.

2. Opis problema

U ovom radu bit će spomenuto nekoliko pojmova koji u današnjem svijetu počinju biti sve učestaliji te zbog kojih arhitekturalna kao i programska rješenja dobivaju sasvim novu dimenziju. Naime, relacijski sustavi su već dovoljno dugo u opticaju te do sada već na veliku količinu izazova nisu pružili rješenja. Samo neki od izazova na koje nema odgovora su skalabilnost, nestrukturiranost podataka, otpornost na ispade što su ujedno i teme ovoga rada. Količina podataka koja pristiže u sustav počela se povećavati svakodnevno, upiti su postajali sporiji te je ažuriranje podataka počelo sve više kasniti. Upravo navedeni izazovi postali su razlog početka potražnje za tehnologijama čije će mogućnosti pružiti rješenja na iste. Kroz istraživanje tehnologija moglo se zaključiti kako jedna tehnologija nema rješenje za sve izazove kao što ni relacijski sustavi nemaju, no novo arhitekturalno rješenje koje će objediniti veći skup tehnologija možda bi moglo.

Lambda arhitektura je pojam koji se u radu na više mjesta spominje te kao takav predstavlja ključ rada jer rješava gotovo sve izazove koji su ranije navedeni. Kroz rad pokušat će se pomoću Lambda arhitekture izgraditi sustav koji će biti otporan na ispade, biti skalabilan te koji će servirati podatke u stvarnom vremenu (eng. *Real-time*) te tako ostvariti odliku sustava koji pruža podatke u stvarnom vremenu (eng. *Real-time system)*. Potrebno je napomenuti da će u radu biti doticaja ne samo s jednom tehnologijom koja će omogućiti funkcioniranje takvog sustava već veći broj što bi u samom startu moglo biti zbunjujuće no na koncu veoma efektivno.

2.1. Big Data

U zadnje vrijeme „Velika količina podataka“ (eng. *Big Data*) je pojam oko kojeg se počela voditi sve veća polemika. Pojam kao takav ima veliki broj različitih definicija te je doveo do određenih konfuzija. Naime, obično se za navedeni pojam gleda kao na enormne skupine strukturiranih i nestrukturiranih podataka gdje se korištenjem tradicionalnih baza podataka teško procesiraju. Tolika količina podataka obično pomaže velikim firmama u poboljšanju izvođenja pojedinih operacija, donošenju bržih te inteligentnijih rješenja. Upravo ti podaci nakon što ih se strukturira tj. formatira u jednu veliku smislenu cjelinu te potom analizira mogu donijeti određene pogodnosti firmi. (Beal, 2016)

Microsoft je opisao navedeni pojam na sljedeći način:„ Big Data je pojam koji se sve više koristi u svrhu opisivanja računalne snage – osobito po pitanju najnovijeg izdanja strojnog učenja i umjetne inteligencije – do izrazito masivnih te često veoma kompleksnih setova informacija.“ (MIT Technology Review, 2016)

Sve veći broj sustava počinju biti građeni tako da se mogu nositi s velikim volumenom, brzinom te raznolikosti koju donosi Big Data. U konačnici izgradnja takvih sustava može pripomoći u dobivanju novih uvida te donošenju boljih poslovnih odluka. U nastavku bit će prikazan način kako se nositi s volumenom te brzinom simultano unutar jednog arhitekturalnog rješenja kao što je to Lambda arhitektura.

2.2. Real-time sustavi

Real-time sustavi dodiruju nekoliko domena računalne znanosti. Oni mogu biti obrambeni te prostorni sustavi, umreženi multimedijski sustavi kao i ugrađeni u automobilsku elektroniku. U takvim sustavim ispravnost ponašanja sustava ovisi ne samo o logičkom rezultatu nekog izračuna već i o fizičkom trenutku u kojem je rezultat produciran. Isto tako, sustavi kao takvi mogu biti razloženi na set manjih podsustava kao na primjer, kontrolirani objekt, real-time računalni sustav te čovjek kao izvođač. Računalni real-time sustav mora reagirati na podražaje kontroliranog objekta (ili izvođača) unutar određenog vremenskog intervala diktiranog u okruženju kao takvom. Trenutak u kojem je rezultat produciran zovemo rok.

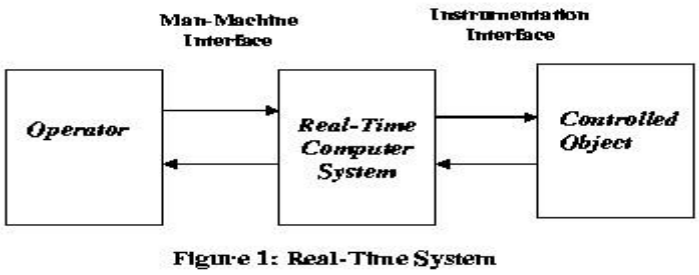


Figure 1: Real-Time System

Slika 1 Real-time sustav (Izvor: Kumar, 2015)

Postoje razni kriteriji za točno definiranje real-time sustava. Kao što je već napomenuto, za sustav kažemo da je real-time ako sav broj ispravnih operacija ne ovisi samo o tim istim logičkim ispravnim operacijama već i o vremenu u kojem su operacije obavljene. Real-time sustav je isto kao što su i rokovi klasificirani prema posljedicama ako se rok ne poštuje:

- Težak – ne poštivanje roka znači cjelokupni neuspjeh sustava kao takvog.
- Solidan – rijetko ne poštivanje roka može biti tolerantno, no isto tako može degradirati kvalitetu servisa sustava. Korisnost kao rezultat nakon roka je jednaka nuli.
- Mekan – korisnost rezultata degradira se nakon roka te zavisno tome dolazi do degradacije kvalitete servisa sustava.

Cilj sustava kao takvog pod kriterijem „težak“ je u osiguravanju da svi rokovi budu poštivani dok je na primjer cilj „mekanog“ u poštivanju nekoliko pod-rokova od kojeg je jedan sastavljen kako bi se optimizirali neki kriteriji specifični za neku aplikaciju.

Kriteriji poput „težak“ za real-time sustave obično se uzimaju kada je imperativ na tome da za neki poseban događaj mora postojati reakcija točno u roku. Potrebu za takva jaka jamstva iskazuju sustavi za koje bi nereagiranje na neki događaj u određenom trenutku značilo veliki gubitak što implicirano tome može značiti i prijetnja ljudskom životu.

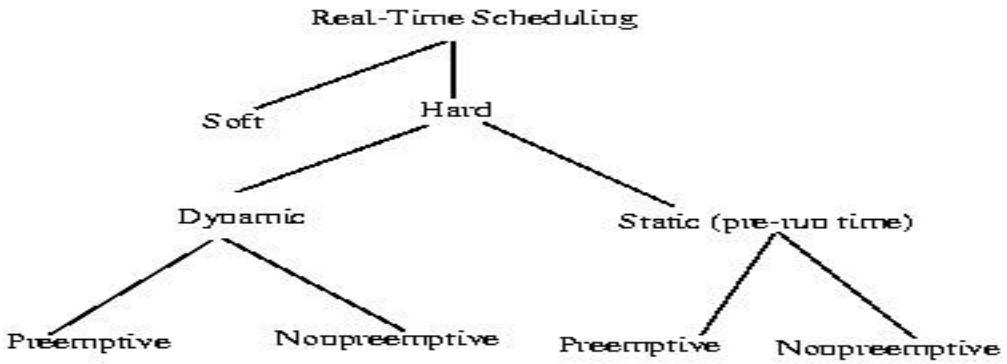
Prikazat ćemo to na jednom primjeru. Naime, motor automobila kontroliran je od strane sustava koji je po kriteriju „težak“ real-time sustav zato što kašnjenje nekog signala može prouzročiti grešku na motoru ili štetu.

Drugi primjeri „teškog“ tipa real-time sustava su pejsmejkeri za srce te industrijski kontrolori procesa. Obično se sustavi takvog tipa mogu naći na hardverima niže razine, ugrađenim sustavima (eng. *embedded systems*) ili sličnima.

Kriterij „mekog“ real-time sustava tipičan je za rješavanje problema istovremenog pristupa te potrebe održavanja broja povezanih sustava ažuriranih u raznim promjenjivima situacijama.

Pravi primjer istog može biti softver koji održava te ažurira planove letova neke aviokompanije.

Planovi letova moraju biti s razlogom ažurirani te moći prikazivati stvarno stanje s mogućim kašnjenjem od nekoliko sekundi. Zvukovni-video sustavi namijenjeni za glazbu uživo također su pod „mekanim“ kriterijem. Kršenjem takvih ograničenja rezultirat će degradiranom kvalitetom zvuka i video zapisa no sustav će u tom trenutku i dalje moći funkcionirati, obavljati započeto te će se također moći oporaviti u budućnosti korištenjem predikcija opterećenja te metodologije ponovne konfiguracije.



Slika 2 Taksonomija real-time sustava (Izvor Juvva, 1999)

2.3. Gotovo real-time sustav

Pojam kao NRT u telekomunikacijama i izračunima odnosi se na vrijeme odgode. Odgoda je uzrokovana automatiziranom obradom podataka ili mrežnom transmisijom između pojavljivanja nekog događaja te potrebe za korištenjem obrađenih podataka kao na primjer prikaz povratne informacije i slično. Na primjer, NRT prikazuje događaj ili situaciju kao da je postojala u tadašnje vrijeme oduzevši vrijeme obrade kao skoro vrijeme događaja koji je tada bio aktualan.

Razlika između pojmova NRT te real-time je relativno mala. Pojam implicira na to kako ne bi trebala postojati značajna kašnjenja u prikazivanju podataka. U mnogo slučajeva, obrada koja se opisuje kao real-time trebala bi biti preciznija te biti opisana kao NRT. NRT se također odnosi na kašnjenje u real-time transmisiji zvuka te videa. Ono dopušta igranje video igrica u približno stvarnom vremenu bez da postoje čekanja kod preuzimanja velikih videa. Razlika između NRT i real-time sustava varira te kašnjenje ovisi o tipu te brzini transmisije. Kod NRT sustava podaci se sakupljaju u istom trenutku kako dođu no nisu nužno odmah i integrirani u neku od baza podataka. Takva integracija može biti programirana tako da se procesiranje podataka izvršava nakon što su podaci sakupljani sat vremena, jedan dan ili svakih par sekundi.

3. Lambda arhitektura

Izvršavanje proizvoljnih upita nad određenim setom podataka s odazivom istog u stvarnom vremenu odnosno velikih kašnjenja je obeshrabrujući izazov. Ne postoji niti jedan alat koji pruža kompletno rješenje za izazov kao takav. Umjesto toga, potrebno je koristiti veliki broj različitih alata te tehnika kako bi se izgradio sustav podložan velikim količinama podataka. Glavna ideja Lambda arhitekture je izgradnja sustava podložnog velikim količinama podataka kao seriju slojeva, a oni su:

1. Brzi sloj (eng. *The Speed Layer*)
2. Sloj posluživanja (eng. *The Serving Layer*)
3. Batch sloj – predstavlja jedinstveni izvor istine podataka u kojem se kreiraju preračunati pogledi ažurirani periodički. (eng. *The Batch Layer*)

Cijela priča zapravo počinje sa

$$query = function(all\ data)$$

izrazom. Idealno, mogli bi pokrenuti funkciju u hodu te prikazati rezultate samo, no, to da je i moguće potrebna bi bila velika količina resursa te bi bilo neopravdano skupo. Zamislite da čitate PB veliki set podataka svaki puta kada želite odgovoriti na izraz, odnosno ako želite dobiti nečiju trenutnu lokaciju. Najočitiji alternativni pristup bio bi tako da se preračunaju izrazi (eng. *Query)* funkcije. Nazovimo preračunati izraz funkcije – obrada velike količine podataka odjednom (eng. *Batch*). Umjesto da se računa izraz u hodu, čitat će se rezultati iz preračunatih pogleda. Preračunati pogledi su indeksirani tako da im se može pristupiti u slučajnim čitanjima. Ovaj bi sustav trebao izgledati na sljedeći način:

$$batch\ view = function(all\ data)$$

$$query = function(batch\ view)$$

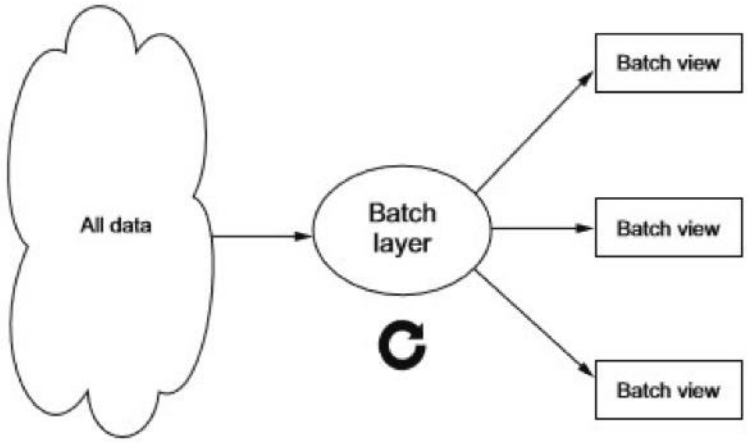
U sustavima kao ovakvim moguće je pokrenuti funkciju nad svim podacima kako bi se dohvatio batch pogled. Batch pogled (eng. *Batch view*) omogućuje da se željene vrijednosti dobiju u relativno kratkom vremenu bez da je potrebno pretražiti sve podatke u njemu. Upravo radi ovoga, rasprava kao takva postaje apstraktna te je krenimo pojednostavljivati.

Pretpostavimo da se izgrađuje web aplikacija za analitiku te se želi pretraživati broj posjeta na stranici za svaki lokator sadržaja (eng. *Uniform Resource Locator* – URL) URL u nekom određenom intervalu.

Ako se računa izraz kao funkciju svih podataka, trebalo bi proći kroz cijeli set podataka za pregled stranica za taj URL unutar intervala te vratiti broj tih istih rezultata.

Batch pogled je pristup koji omogućuje pokretanje funkcija nad svim posjetima stranica koje bi preračunale indeks po ključu [url, dan] do prebrojavanja broja posjeta stranica za taj URL za određeni interval.

Nakon što se izvrši izraz dobivaju se sve vrijednosti tog pogleda za sve dane u tom intervalu. Ovaj pristup prikazan je na sljedećoj slici.

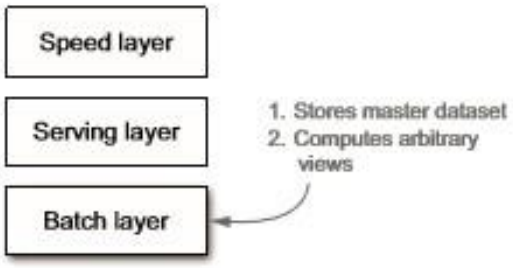


Slika 3 Arhitektura batch sloja(Izvor: Kumar, 2015)

Trebalo bi biti jasno da nešto nedostaje ovakvom pristupu kao što je prijašnje bilo opisano. Kreiranje batch sloja će očito imati problem s operacijama visoke latencije upravo zato što će se izvršavati funkcija nad svim podacima koji se posjeduju. Dok se operacija završi dosta će novih podataka biti sakupljeno koji se ne nalaze u batch pogledu te izrazi neće biti poravnati s datumom već će isti kasniti i do par sati. No, ignorirajmo ovakav problem na trenutak zato što će se isti kasnije moći popraviti. Pravimo se da je sve u redu s izrazom tako da podaci kasne par sati te krenimo istraživati ideju preračunavanja batch pogleda pokretanjem funkcija nad cijelim setom podataka. (Mraz i Warren, 2015)

3.1. Batch sloj

Dio Lambda arhitekture koji implementira *batch view = function(all data)* izraz nazvan je batch sloj. Sloj kao takav sprema glavnu kopiju seta podataka te preračunava batch pogled nad glavnim setom podataka kao što je prikazano na slici ispod.



Slika 4 Batch sloj (Izvor: Mraz i Warren, 2015)

Glavni set podataka može se zamisliti kao velika lista podataka. Batch sloj mora moći raditi dvije stvari:

1. Spremanje nepromjenjivih, konstantno rastućih podataka
2. Računati proizvoljan broj funkcija na tom istom podatkovnom setu.

Takav tip procesiranja je najbolje napravljen korištenjem batch procesirajućeg sustava. Hadoop je primjer batch procesirajućeg sustava te će Hadoop biti alat koji će se koristiti u radu kako bi se prikazali koncepti batch sloja.

Najjednostavniji način prikaza batch sloja može biti reprezentiran pseudo kodom na sljedeći način:

```
01. function runBatchLayer():
02.     while(true):
03.         recomputeBatchView()
```

Slika 5 Primjer koda s izračunom batch pogleda

Batch sloj pokreće se u *while(true)* petlji te neprekidno preračunava batch pogled od početka. U stvarnosti, batch sloj je malo više uključen no s vremenom doći će se i do tog dijela. Ovo je najbolji put u načinu razmišljanja o batch sloju u ovom trenutku. Dobra stvar o batch sloju je ta da je jednostavan za korištenje.

Batch računanja su pisana kao jedno-nitni programi gdje se dobiva paralelizam kao dodatak. Jednostavno je za pisati robusna, jako skalabilna računanja u batch sloju gdje se ista skaliraju tako da se samo dodaju novi čvorovi.

U nastavku se nalazi primjer batch sloj računanja. Nije se potrebno zamarati nerazumijevanjem koda, poanta je u uvidu na to kako paralelni programi izgledaju.

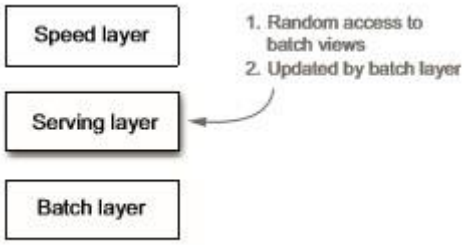
```
01. Api.execute(Api.hfsSeqFile("/tmp/pageeview-counts"),
02.     new SubQuery("?url", "?count")
03.         .predicate(Api.hfsSeqfile("/data/pageview"),
04.             "?url","?user", "?timestamp")
05.         .predicate(new Count(), "?count");
```

Slika 6 Primjer paralelnih programa

Kod kao takav računa broj posjeta na stranici za svaki dani URL kao ulazni parametar podatkovnog seta sirovih podataka o posjetama na stranici. Ono što je zanimljivo u ovome kodu je to da su sve prepreke u rasporedu posla te spajanju rezultata već odrađene. Upravo zato što je algoritam napisan na ovaj način, isti može biti proizvoljno distribuiran putem MapReduce klastera, skalirajući ga do koliko god čvorova u klasteru postoji te do koliko god ih je raspoloživo. Na kraju računanja, izlazni direktorij će sadržavati određen broj datoteka s rezultatima koji su potrebni.

3.2. Sloj posluživanja

Batch sloj emitira batch pogled kao rezultat funkcija. Sljedeći korak je u učitavanju pogleda kako bi isti mogli biti „ispitani“ (eng. *Queried)*. U ovom koraku zapravo nastaje sloj posluživanja. Sloj posluživanja specificiran je kao distribuirana baza podataka u koju se spremaju izlazni podaci brzog i batch sloja te omogućava nasumično čitanje nad istim kao što je prikazano na sljedećoj slici.

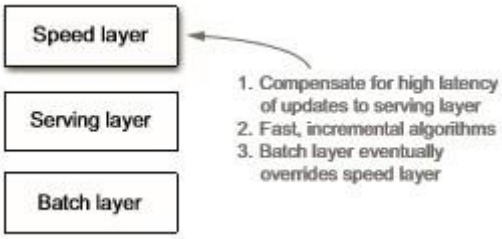


Slika 7 Sloj posluživanja (Izvor: Mraz i Warren, 2015)

Kada je novi batch pogled dostupan, sloj posluživanja se automatski ažurira s istim tako da je više ažuriranih rezultata dostupno. Sloj posluživanja baza podataka pruža potporu za batch izmjene te nasumična čitanja. Ono što se najviše ističe je to da kao takav ne podržava potporu za nasumična pisanja. Ovo je jako važna točka jer nasumična pisanja prouzrokuju najveći dio kompleksnosti baza podataka. S obzirom na to da sloj ne podržava nasumična pisanja, baze podataka koje spadaju u takav sloj su poprilično jednostavne. Takva jednostavnost čini ih robusnim, laganim za konfiguraciju te jednostavnim za upravljanje. ElephantDB je kao takav primjer sloja posluživanja baza podataka.

3.3. Brzi sloj

Sloj posluživanja radi izmjene kada god batch sloj završi s preračunavanjem batch pogleda. To znači da jedini podaci koji se ne prikazuju u batch pogledu su novi podaci koji su došli za vrijeme izvršavanja preračunavanja. Jedino što je preostalo za napraviti kako bi postojale proizvoljne funkcije izvršene nad podacima u stvarnom vremenu je kompenzirati sve ostale sate podataka koji nedostaju. Izgradnjom takvih funkcija koje bi kompenzirale sve ostale sate novih podataka ostvario bi se jedan od ciljeva, a to je izgradnja sustava u stvarnom vremenu. Ovo je svrha brzog sloja. Kao što mu samo ime predlaže, cilj je osigurati da se novi podaci prikazuju u funkcijama upita što je brže moguće za aplikacijske potrebe.



Slika 8 Brzi sloj

Brzi sloj se može zamisliti kao sličan batch sloju jer proizvodi poglede bazirane na podacima koje dobije. Jedna velika razlika je ta da brzi sloj gleda samo zadnje podatke dok batch sloj gleda sve podatke osim zadnje pristiglih od jednom. Druga velika razlika je u latenciji. Da bi se smanjila latencija što je moguće više, brzi sloj ne gleda sve nove podatke odjednom već radi izmjene real-time pogleda kako dobije nove podatke umjesto da preračunava pogled od samog početka kao što to batch sloj radi. Brzi sloj radi inkrementalno računanje umjesto preračunavanja koje se izvodi u batch sloju.

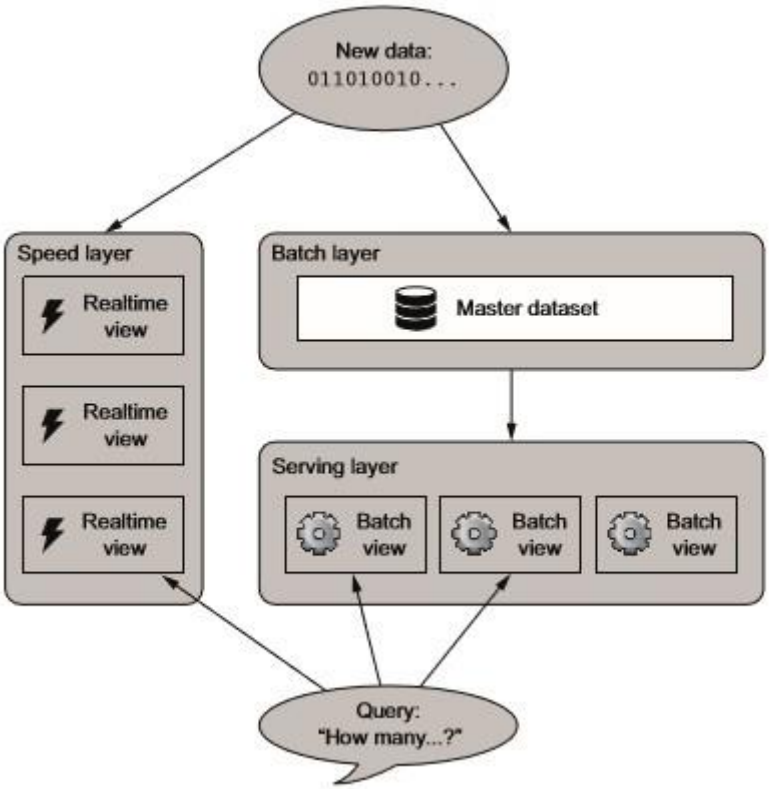
Može se formalizirati tok podataka brzog sloja sa sljedećim izrazom:

```
01. realtime view = function(realtime view, new data)
```

Parametar „*realtime view*“ je izmijenjen bazirano na novim podacima te postojećim real-time pogledom. Lambda arhitektura je u potpunosti sažeta na sljedeća tri izraza:

```
01. batch view = function(all data)
02. realtime view = function(realtime view, new data)
03. query = function(batch view, realtime view)
```

Slikovna reprezentacija ovih ideja je pokazana na sljedećoj slici:



Slika 9 Dijagram lambda arhitekture (Izvor: Shichao, 2017)

Umjesto rješavanja upita izvršavanjem funkcije samo nad batch pogledom, upiti se rješavaju gledanjem na batch kao i na real-time poglede te spajanjem istih.

Brzi sloj koristi bazu podataka koja podržava nasumična čitanja te nasumična pisanja. Zato što takve baze podataka podržavaju nasumična pisanja i čitanja njihovi redovi veličine su više kompleksnije naravni nego kod baza podataka koje se koriste u sloju posluživanja kako u dijelu implementacije tako i u operacijskom dijelu.

Ljepota Lambda arhitekture je ta da jednom kada podaci prođu kroz batch sloj u sloj posluživanja, odgovarajući rezultati u real-time pogledu više nisu potrebni. To znači da se dijelovi real-time pogleda odbacuju.

Ovo je veoma zadovoljavajući te predivan rezultat zato što je brzi sloj daleko kompliciraniji od batch sloja kao i od sloja posluživanja.

Svojstvo Lambda arhitekture zove se kompleksna izolacija što znači da je kompleksnost gurana na sloj čiji su rezultati samo temporalni. Ako išta pođe po zlu, uvijek se može odbaciti stanje cijelog brzog sloja te će sve biti vraćeno u normalu unutar par sati.

Nastavimo s primjerom web aplikacije za analizu koja podržava upite s brojem posjeta stranice u razmaku od par dana. Podsjetimo se da batch sloj proizvodi batch poglede od [url, dan] do broja posjeta na stranicu. Batch sloj preračunava svoje poglede doslovno prebrojavanjem posjeta na stranici dok brzi sloj samo izmjenjuje svoje poglede te inkrementa brojač u pogledu gdje je primio nove podatke. Kako bi se riješio upit, isti se izvršava na oba sloja, tj. nad batch kao i nad brzim slojem. To ne iziskuje puno posla kojeg je obvezno napraviti kako bi se sinkronizirali rezultati te će isto biti pokriveno u sljedećim poglavljima.

Poneki algoritmi su teški za inkrementalno računanje. Odvajanjem batch-a sloja od brzog sloja dobiva se na fleksibilnosti koja omogućuje korištenje određenog algoritma za batch kao i određenog za brzi sloj tako da se aproksimacijom dobije točnost. Računanje unikatnih brojača može biti na primjer izazov kod računanja velikih unikatnih setova. Jednostavno je odraditi prebrojavanje unikatnosti u batch sloju zato što se radi brojanje nad cijelim setom odjednom no u brzom sloju dolazi do potrebe za korištenjem Hyper-LoLog seta za aproksimaciju.

Ono s čim će se završiti je najbolje za oboje odnosno za svijet performansi te robusnosti. Sustav koji odrađuje točno računanje u batch sloju te aproksimativno računanje u brzom sloju izlaže eventualnu točnost jer batch sloj ispravlja ono što je izračunato u brzom sloju. Isto tako i dalje je intencija imati nisku latenciju kod izmjena, no zato što je brzi sloj prolazan, kompleksnost spremanja istog ne utječe na robusnost rezultat. Prolazna priroda brzog sloja omogućuje fleksibilnost da se bude više agresivniji kada u pitanje dođe trampa za performanse. Naravno, za računanja koja mogu biti izvršena točno na inkrementalan način sistem će biti u potpunosti točan. (Mraz i Warren, 2015)

3.4. Batch sloj i sloj posluživanja zadovoljavaju sva svojstva

Batch te sloj posluživanja podržavaju proizvoljne upite nad proizvoljnim setovima podataka tako da se žrtvuje stavka kako će upiti kasniti koji sat. Uzima se novi dio podataka koji se kroz par sati propagira u batch sloju, nakon čega se prenosi u sloj posluživanja nad kojim se tada mogu izvršavati upiti. Važna stvar koju je potrebno napomenuti je ta da usprkos maloj latenciji izmjena, batch te sloj posluživanja zadovoljavaju svako svojstvo koje sustavi podložni velikim količinama podataka traže.

Svojstva kao takva su:

1. Robusnost te tolerancija kvarova – Hadoop je otporan na trenutke kada jedan čvor doživi šok odnosno postane nedostupan. Sloj poslužitelja koristi replikaciju kako bi osigurao dostupnost kada neki od čvorova nisu dostupni. Batch te sloj poslužitelja su također otporni na ljudsku grešku upravo zato što ako je pogreška napravljena može se istu popraviti s algoritmom ili obrisati loše podatke te napraviti novi proračun pogleda otpočetka.
2. Skalabilnost – Oboje, batch te sloj poslužitelja su lagani za skalirati. Oboje su u potpunosti distribuirani sustavi te skaliranje istih je lagano kao dodavanje novih čvorova.
3. Rastezljivost – Dodavanje novih pogleda je jednostavno tako da se dodaju nove funkcije na glavni set podataka. Upravo zato što glavni set podataka koristi proizvoljne podatke, novi tipovi podataka mogu biti jednostavno dodani.
   Ako je želja za uštipnim pogledima, nije potrebno brinuti se o podršci više verzija pogleda u aplikaciji, jednostavno se ponovno izračuna cijeli pogled.
4. Minimalno održavanje - Glavna komponenta za održavanje u sustavu je Hadoop. Hadoop zahtijeva određeno znanje u administriranju no u biti je jednostavan za upravljanje. Kao što je opisano prije, sloj posluživanja baza podataka je jednostavan upravo zato što ne podražava nasumična pisanja. Zato što sloj posluživanja ima relativno malo dijelova, postoji puno manje mjesta da nešto krene po zlu. Zato što postoji puno manje mjesta da nešto krene po zlu sa slojem posluživanja baza podataka, lakši je po pitanju održavanja.
5. Ad hoc upiti – Bach sloj podržava ad hoc upite po prirodi. Svi podaci su dostupni na jednoj lokaciji.

6. Mogućnost otklanjanja neispravnosti – Uvijek će postojati potreba za ulaznim te izlaznim parametrima računanja u batch sloju. U tradicionalnim bazama podataka izlazni podaci mogu biti zamijenjeni originalnim ulaznim podacima kao na primjer kod povećavanja vrijednosti. U batch sloju te sloju posluživanja ulazni dio je glavni set podataka te su izlazni podaci dio pogleda. Jednako tome, postoje ulazi te izlazi za sve naprednije korake. Kako imamo ulaze te izlaze, isti nam daju sve potrebne informacije kako bi mogli otkriti te otkloniti neispravnosti ako nešto krene po zlu.

Ljepota u batch sloju te sloju posluživanja je ta da isti zadovoljavaju skoro sva svojstva koja želimo s pristupom laganim te jednostavnim za razumjeti. Ne postoji konkurencijski problemi s kojima bi se mogli nositi te je skaliranje trivijalno. Jedino svojstvo koje nedostaje je mala latencija kod izmjena. Završni sloj je brzi sloj koji upravo taj problem rješava. (Mraz i Warren, 2015)

3.5. Zadnji trendovi u tehnologiji

Dobro je za razumjeti pozadinu koja se nalazi iza svih alata koje će se koristiti u ovom radu. Postoje brojni trendovi u tehnologiji koji uvelike utječu na način na koji se grade sustavi podložni velikim količinama podataka.

Počinju se dostizati fizičke granice koliko procesori brzo i dobro mogu raditi. To znači da ako se želi skalirati kako bi se moglo podržati više podataka morala bi se paralelno odraditi sva svoja postojeća računanja. Takvo nešto vodi do paralelnih algoritama koji ne dijele resurse za sustave koje odgovaraju istim kao na primjer MapReduce. Umjesto da se trud ulaže u skaliranje kupovanjem jačih servera, što je poznato kao vertikalno skaliranje, sustav će se u ovom radu skalirati dodavanjem novih servera što se još zove horizontalno skaliranje.

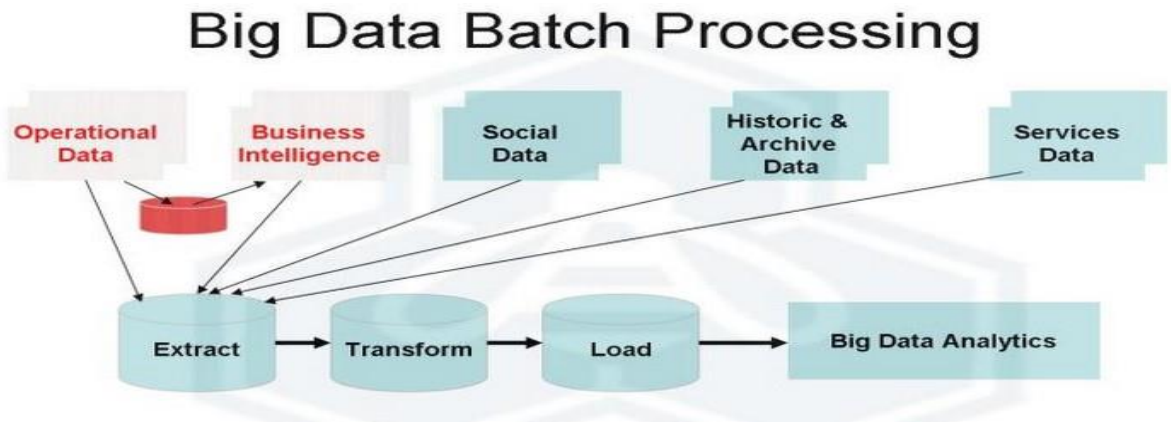
Također, još jedan od trendova u tehnologiji koji se počinje sve više spominjati zove se elastični oblak, poznat još kao i infrastruktura servisa. Amazon Web servis (eng. *AWS*) je najpoznatiji kao takav. El. oblaci nude iznajmljivanje hardvera na zahtjev u odnosu da se kupuje vlastiti hardver te isti postavi na nekoj lokaciji. Također, isti dopušta smanjenje te povećanje klastera skoro istovremeno tako da ako je pokrenuta neka aplikacija u obliku procesa prijenosa podataka (eng. *Streaming*)[1] moguće je alocirati hardver temporalno. Isti također pojednostavnjuju sistem administraciju te pružaju dodatan prostor i opcije alokacije hardvera koje mogu značajno spustiti cijenu infrastrukture. Na primjer, AWS ima svojstvo nazvano točka instance u kojoj se stavljaju ponude na instance radije nego da se plati fiksna cijenu.

Za distribuirani sustav računanja kao MapReduce, jako je dobra opcija jer nudi otpornost na grešku koja je podržana na aplikacijskom sloju.

[1] Streaming – proces kojim se opisuje brzo procesiranje podataka kako bi se dobili bolji uvidi u podatke kao i korisni trendovi iz istog.

3.6. Batch u odnosu na real-time obradu

Kada se govori o obradi gomile podataka tada se isto odnosi na efikasan način obrade velikih količina podataka gdje se grupa transakcija prikuplja u nekom određenom periodu te nakon istog obrađuje. Na takav se način vrše prikupljanja podataka, pristupanja istim te njihova obrada čime se dobivaju batch rezultati (Hadoop[2] je fokusiran na batch obradu podataka). Batch obrada podataka zahtijeva kao ulaz odvojene programe te procese. Primjer toga je platni spisak te sustavi naplate.



Slika 10 Big Data batch obrada podataka (Izvor: Walker, 2013)

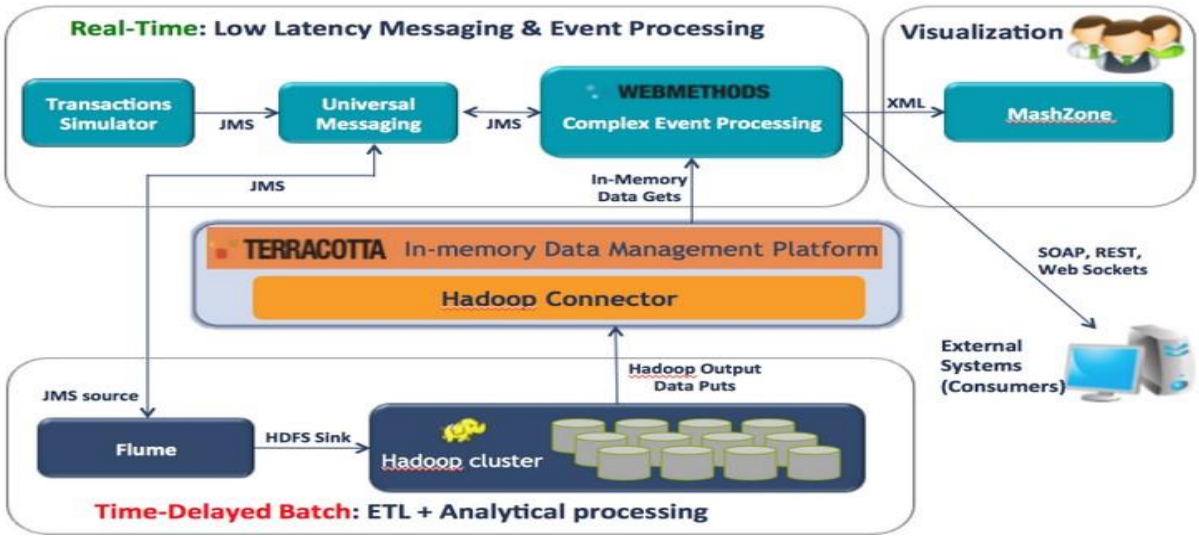
Ono što bi uz sliku trebalo napomenuti je to da tradicionalni sustavi koriste baze podataka koje su privatnom vlasništvu, osigurane lozinkom te izvorni kod nije dostupan čak i ako korisnik kupi prava za korištenje baza podataka kao takvih. Primjer takvih baza podataka je Oracle. Sustavi velikih količina podataka koriste paralelne sustave čije je izvorni kod javno dostupan kao što je to na primjer Hadoop.

Kod batch obrade podataka kada je potrebno kreiranje novog indeksa, proces inicijalnog indeksiranja vrši se nad cijelim sadržajem jednog ili više podatkovnih repozitorija. Obe navedene tehnike su podobne kao i orijentirane ka batch obradi ili analizi podataka.

Također, bitno je napomenuti da je u takvim sustavima gotovo nemoguće dobiti real-time ili analizu gotovo u stvarnom vremenu (eng. *Near Real Time – NRT*).

[2] Programski okvir omogućava distribuiranu obradu velikih setova podataka u klasteru računala koristeći jednostavne programske modele

U odnosu na real-time obradu podataka koja uključuje neprekidan ulaz, proces te izlaz podataka, podaci kod NRT obrade podataka moraju biti procesirani u malim periodima. Primjere istih je već naveden. (The Apache Software Foundation, 2017)



Slika 11 Primjer real-time sustava (Izvor: Walker, 2013)

Dok većina organizacija koristi batch obradu podataka, ponekad je organizacijama potrebna i NRT obrada podataka. NRT obrada podataka kao i analitika nude organizacijama mogućnost neposrednog poduzimanja akcija točno u trenucima kada je djelovanje unutar sekunde ili minute od presudne važnosti. Glavni cilj je dobivanje uvida u to kako promišljeno u pravo vrijeme djelovati.

U Hadoop-ovom okruženju problem u pružanju NRT analize je u skalabilnom sloju gdje se podaci čuvaju u memoriji (eng. *in-memory)* između Hadoop-a te kompleksne obrade događaja (*eng. Complex Event Processing* (CEP)[3]). Storm je distribuirani real-time računalni sustav koji procesira tok podataka (eng. *streams of data)*. Alat kao takav može pomoći s real-time analitikom, strojnim učenjem, neprekidnim računanjem, distribuiranim RPC-om[4] i ETL-om.

[3] CEP – kombinira podatke između više izvora kako bi detektirao uzorke te pokušao identificirati prilike ili prijetnje. Glavni cilj je u identifikaciji značajnih događaja te odgovorima na iste. Primjeri su u prodaji, službi za korisnike i sl.

[4] RPC – eng. *Remote procedure call*, tehnologija koja stvara distribuirane klijent/server programe te omogućava istima, odnosno klijentu i servisu međusobnu komunikaciju.

Hadoop-ov MapReduce[5] obrađuje takozvane „poslove“ (eng. *Jobs*) u batch-u dok Storm obrađuje tokove podataka (eng. *Streams)* u NRT-u. Ideja je u tome da se izravnaju real-time te batch obrada podataka kada su u pitanju veliki setovi podataka.

Primjer istoga je u detekciji prijevara kod transakcija u NRT-u dok se sjedinjuju podaci iz skladišta podataka ili Hadoop-ovog klastera. U nastavku se nalazi lista batch te real-time solucija za obradu podataka:

Tablica 1 Lista batch i real-time solucija za obradu velike količine podataka

Solucije	**Developeri**	**Tip**	**Opis**
Storm	Twitter	*Streaming*	Twitter-ova analitička solucija za prijenos velikih količina podataka
S4	Yahoo!	*Streaming*	Distribuirana platforma za obradu u obliku strujanja podataka
Hadoop	Apache	*Batch*	Prva implementacija MapReduce paradigme
Spark	UC Berkeley AMPLab	*Batch*	Najnovija platforma za analitiku koja podržava *in-memory* setove podataka te elastičnost
Disco	Nokia	*Batch*	Nokia-in distribuirani MapReduce razvojni okvir
HPCC	LexisNexis	*Batch*	HPC klaster za velike količine podataka

[5] MapReduce – razvojni okvir za obrađivanje paralelnih problema kroz velike setove podatka koristeći veliki broj računala kao čvorova koji su kolektivno dodijeljeni jednom klasteru ili nekoj koordinatnoj mreži eng. *grid* (ukoliko se koriste heterogeni hardver)

3.7. Infrastruktura analitike za veliku količinu podataka

Analitika velike količine podataka odnosi se na znanost kao i na analize obiju unutarnjih i vanjskih podataka. Cilj istog je u dobivanju vrijednih te uvida podložnim akcijama koji omogućuju organizaciji donošenje boljih odluka.

Zadnje ankete preporučuju da je najbolje područje investicije kako za privatne tako i za javne organizacije zapravo u dizajnu i izgradnji modernog skladišta podataka/poslovne inteligencije/arhitekture za analitiku podataka koja pruža fleksibilan, više-činjenični analitički ekosustav.

Tradicionalne tehnologije poslovne inteligencije pružaju povijesni, trenutni te opisni pogled na poslovne operacije. U odnosu na tradicionalne tehnologije, analitike velike količine podataka fokusiraju se na podatkovnu znanost, prediktivnu analizu, rudarenje podataka, procesima optimizacije odluka te upravljanje poslovnim performansama.

Pet ključnih pogodnosti koje se dobivaju od znanosti o podacima kao i poslovne analitike:

1. Unaprjeđenje procesa za donošenje odluka
2. Ubrzavanje procesa za donošenje odluka
3. Bolja raspodjela resursa po strategijama
4. Odgovaranje korisničkim potrebama za dostupnost podataka pravovremeno.

Prihvaćanjem znanosti o podacima, procesa optimizacije odluka te analitičkog pristupa, poduzeća mogu identificirati najprofitabilnije kupce/korisnike, ubrzati inovacije po pitanju proizvoda, optimizirati opskrbne lance te cijene kao i identificirati zaslužne za financijsku uspješnost. (Rose Business Technology, 2017)

4. Opis arhitekture

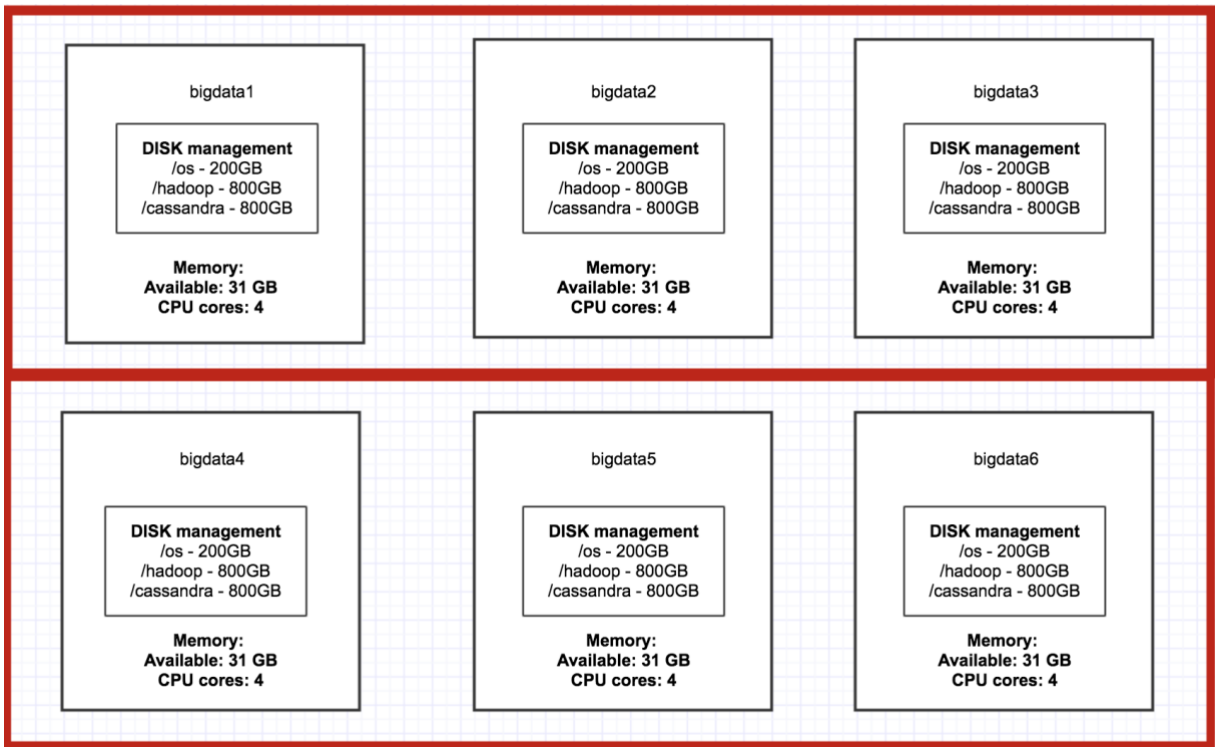
Kako su osnovni pojmovi objašnjeni i cilj postavljen u nastavku ćemo se usredotočiti prvenstveno na arhitekturu te odluke dizajna prema kojima se rade sustavi za procesiranje *stream* aplikacija. Također, bit će prikazano više perspektiva kao i alata kojima će se zaokružiti sustav koji se gradi odnosno sustav koji je zasnovan na prijenosu događaja. Između ostalog prikazat će se izravna usporedba s arhitekturom tradicionalnih baza podataka kao i distribuiranih sustava. Sustavi kao takvi su budućnost upravo zato što pojednostavnjuju kodiranje, povećavaju robusnost, smanjuju latenciju te omogućuju veću fleksibilnosti kada dolazi u obzir manipulacija podacima. No, prilikom izgradnje nekog sustava koji će biti ključan kada dolazi u pitanje prijenos podataka kao i funkcioniranje aplikacija obično se susreće s pojedinim izazovima. Samo neki od izazova koji su riješeni arhitekturom koja će biti predstavljena u nastavku su:

1. Tolerancija na ljudsku pogrešku
2. Tolerancija na hardverske pogreške
3. Skalabilnost
4. Brz odaziv.

U sljedećim poglavljima započet će se sa slaganjem arhitekture zadužene za prijenos događaja. Kao primjer bit će prikazan prijenos od 300 podataka u sekundi uz aplikaciju koja će istu količinu podataka obrađivati te pripremati za sloj posluživanja.

4.1. Sistem arhitektura

Kao osnovu nad kojom će se graditi sustav pripremit će se šest virtualnih strojeva. Operacijski sustav koji će se koristiti bit će Centos verzije sedam te su resursi istih prikazanih na slici ispod.



Slika 12 Izgled infrastrukture

4.2. Arhitektura servisa

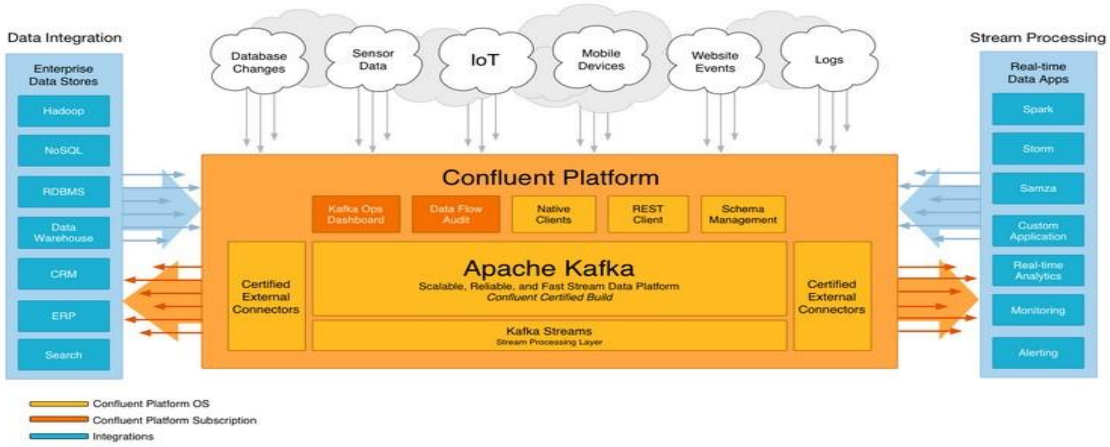
Postoji niz tehnologija kojima je svrha dohvaćanje podataka iz niza relacijskih baza podataka te slanja istih na neko mjesto. Projekt je mogao biti u startu napisan u Spring Boot-u[6] kao jedna aplikacija za sve, bez dodavanja novih servisa, no kako bi dodatno pojednostavili stvari u vezi arhitekture kreće se od projekta nazvanog Confluent.

[6] Spring Boot – platforma koja omogućuje kreiranje samostalnih (eng. *stand-alone*) aplikacija sa minimalnim utroškom vremena na konfiguraciju. Postoje veliki broj biblioteka koji omogućuju lagani početak projekta te nema nepotrebnih generiranja kodova koji mogu dodatno otežati radi (Spring Boot, 2017)

4.2.1. Confluent

Confluent je platforma za prijenos podataka koja omogućuje organizaciju te upravljanje velikim količinama podataka koji pristižu svaku sekundu iz nekog izvora ka sustavu. Podaci obično dolaze u nestrukturiranom obliku no bez obzira na to, ti isti podaci nisu zanemarivi već mogu biti od velike važnosti.

Kao takav, omogućuje batch analizu velikih količina podataka u integraciji sa Hadoop-om te punjenja nekog real-time sustava s podacima. Također nudi nadgledanje sustava te izvršavanja tradicionalnih integracijskih poslova po pitanju velikih količina podataka koji zahtijevaju veliku propusnost, industrijski jake transformacije ili učitavanje ETL-a. Kako zapravo izgleda dio arhitekture ekosustava kada je u pitanju Confluent prikazano je na *Slika 13*.



Slika 13 Confluent (Izvor: Confluent, 2017)

Kao što je vidljivo iz slike u samoj jezgri Confluent-a kao platforme nalazi se Apache Kafka koja će u nastavku biti objašnjena. Valja napomenuti da je Confluent naime platforma koju koristi LinkedIn te koja je nastala tamo u potražnji rješenja određenih izazova. Na temelju Confluent-a riješeni su veliki izazovi po pitanju optimizacije LinkedIn-a kao internet stranice. Ono za što će se koristiti Confluent u ovom projektu je zbog sheme koju kao takav posjeduje, konektora pomoću kojeg će se prihvaćati podaci iz izvora te u konačnici algoritma koji su razvili za prihvaćanje podataka svakih *n* sekundi. U samom početku potrebno je preuzeti jednu od verzija Confluent platforme. U radu će biti korištena verzija 4.0.0.

```
wget http://packages.confluent.io/archive/4.0/confluent-4.0.0-2.11.7.tar.gz
tar -xzf confluent-4.0.0-2.11.7.tar.gz
cd confluent-4.0.0
```

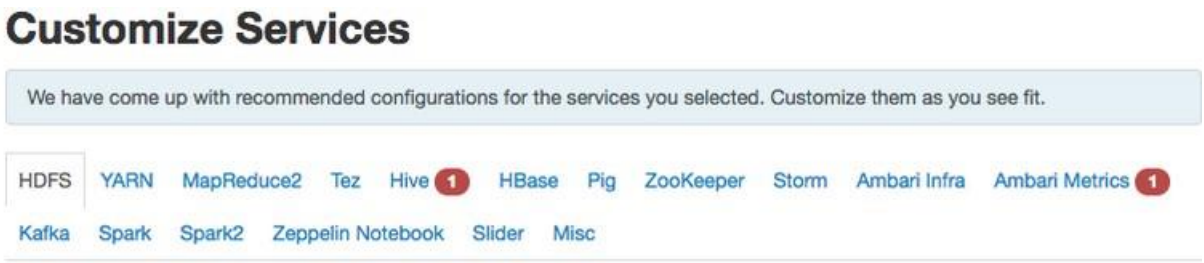
Nadalje, u direktoriju *confluent-4.0.0/etc* nalazi se datoteka imena *server.properties.* Navedena datoteka je primjer konfiguracijske datoteke u kojoj je potrebno namjestiti imena ili IP[7] adrese računala koji se koriste u mreži kao distribuiranom sustavu, tip kompresije, Zookeeper-e koji će kasnije biti objašnjeni te Kafka servere. Pošto nisu još instalirane ključne stvari koje su potrebne krenut će se s objašnjavanjem istih.

4.2.2. Apache Ambari i Hortonworks platforma

Apache Ambari je projekt čija je svrha izgradnja sustava za upravljanjem Hadoop-om. Također, softver omogućava jednostavniji način praćenja promjena, mogućnost upravljanja te praćenja klastera Apache Hadoop-a.

No, Apache Ambari nudi još pregršt programa koje je moguće instalirati te također pratiti i upravljati istima. Ono što je za ovaj projekt važno i zanimljivo je da podržava servise kao Apache Zookeeper, Apache Kafka, Apache Spark. Kako nudi takve mogućnosti na jedan, glavni čvor *node1* instalirat će se Apache Ambari kao i njegovog agenta. Isto tako na svaki sljedeći od *node2* do *node6,* instalirat će se agent kako bi zajedno mogli biti spojeni u jedan klaster. Verzija koja će se koristiti za Apache Ambari bit će najnovija, a ona je 2.6.3.

Naime, agenti su posebni procesi čija je uloga instaliranje, brisanje, promjena svih paketa ili servisa koji se izaberu na glavnom čvoru.



Slika 14 Prikaz mogućih servisa

[7] IP adresa (eng. *Internet Protocol address)* - odnosi se na jedinstveni broj računala koji se nalazi u mreži.

4.2.3. Apache Kafka

Apache Kafka je distribuirana streaming platforma. Kada se kaže distribuirana streaming platforma tada se misli na tri bitne stavke:

- Omogućuje objavljivanje te pretplaćivanje na stream-ove podataka. Takav aspekt je sličan redu poruka ili nekom velikom sustavu za slanje poruka.
- Omogućuje spremanje procesa prijenosa podataka tako da pruža sustav koji je otporan na prestanak rada jedne ili više komponenti, što su u ovom slučaju čvorovi.
- Omogućuje obradu podataka točno u trenutku kada se isti dogode odnosno novi podaci pojave.

Apache Kafka se koristi za dvije glavne svrhe te je po pitanju toga među najboljim alatima, a one su:

1. Izrada cjevovoda za real-time streaming podataka koje pouzdano prenose podatke između sustava i aplikacija.
2. Izrada real-time streaming aplikacija koje transformiraju ili reagiraju na prijenose podataka.

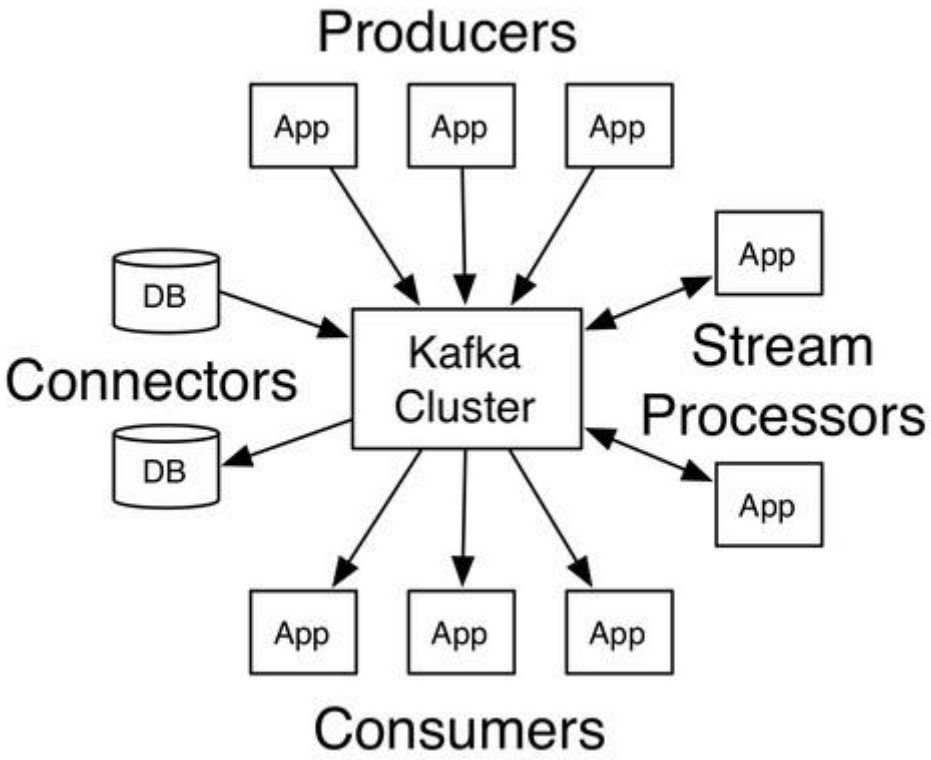
Koncepti nad kojima Apache Kafka počiva:

- Kafka je pokrenuta u klasteru na jednom ili više čvorova.
- Kafka-in klaster sprema procese prijenosa podataka pod određene kategorije koje se zovu još „*teme*“.
- Svaki zapis sastoji se od ključa, vrijednosti te vremenske oznake.

Verzija Apache Kafke koja će se instalirati preko Confluent platforme bit će 1.0.0. Svaki server Kafka-e sprema prijenose podataka u obliku tema te se svaki redak u temi sastoji od vrijednosti, ključa te vremenske oznake. (Apache Kafka, 2017)

Isto tako Kafka sadrži četiri glavna aplikacijsko programska sučelja (*eng. Application Programming Interface – API)*:

1. Proizvođač – omogućava aplikacijama objavljivanje zapisa jednoj ili više Kafka-inih tema.
2. Potrošač – omogućava aplikacijama da se pretplate na jednu ili više tema te obrađuju zapise koji su poslani istima.
3. Streams API – omogućuje aplikacijama da se ponašaju kao procesori tako da prikupljaju podatke od jednog ili više čvorova te proizvode izlazne tokove jednoj ili više izlaznih.
4. Konektor – omogućuje izgradnju te pokretanje proizvođača i potrošača koji se mogu koristiti u više svrha tako da se spoje na Kafka-ine teme, postojeće aplikacije ili podatkovne sustave. Na primjer, konektor za relacijske baze podatka koji može uhvatiti svaku promjenu koja se u tablici dogodila. (P. Damodaran, 2017)



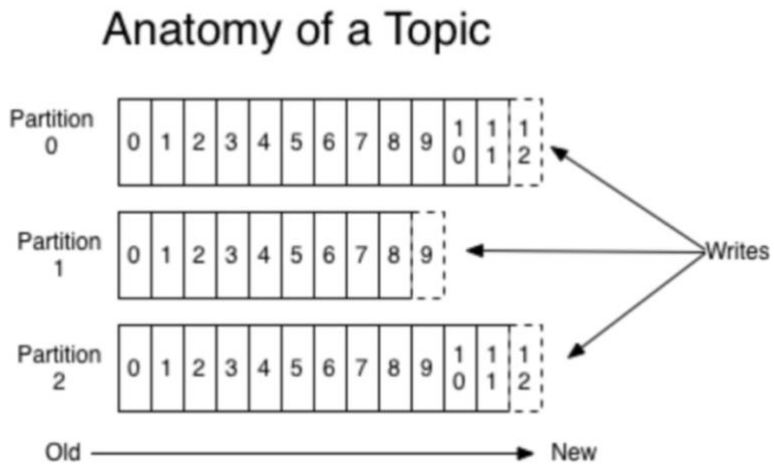
Slika 15 Izgled Kafkine arhitekture (Izvor: Apache Kafka, 2017)

U Kafka-i, komunikacija između klijenata i servera napravljena je preko TCP[8] protokola. Protokol kao takav održava kompatibilnost sa starijim verzijama.

4.2.3.1. Teme i logovi

Kako bi mogli razumjeti i krenuti s izradom projekta potrebno je objasniti srž apstrakcije koju Kafka pruža za prijenos podataka, a to je tema. Tema (eng. *Topic*) je kategorija ili općenito ime o tome kakvi se podaci u istu spremaju. Kao takva, Kafka uvijek podržava višestruko pretplaćivanje, odnosno tema kao takva može imati niti jednog, jednog ili više potrošača koji su u isto vrijeme pretplaćeni na čitanje podataka u istoj.

Za svaku temu posebno Kafka-in klaster generira te održava particionirani zapisnik koji izgleda kao na sljedećoj slici:

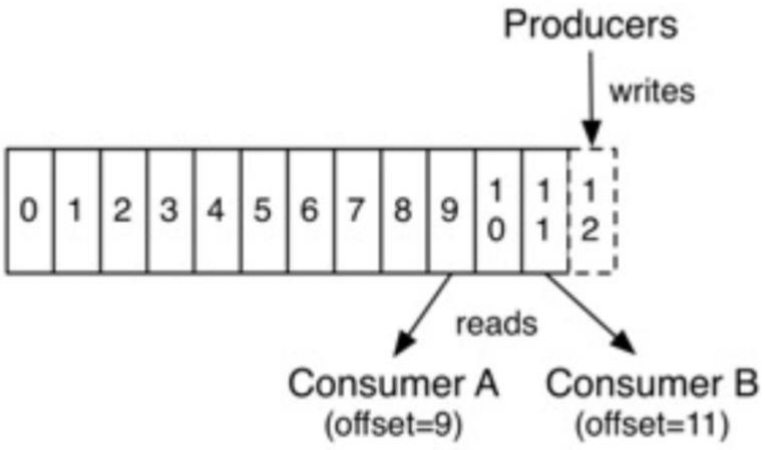


Slika 16 Anatomija tema (Izvor: Apache Kafka, 2017)

Svaka particija je sortirana, nepromjenjiva sekvenca podataka koja se kontinuirano dodaje strukturiranom zapisniku. Svim podacima unutar particije dodan je sekvencijalni „id“ nazvan ofset koji unikatno označuje svaki podataka unutar particije. Kafka klaster zadržava sve proizvedene podatke, bili oni korišteni od strane nekog potrošača ili ne tako da se također može konfigurirati period zadržavanja podataka. Na primjer, ako je konfigurirano da je period zadržavanja podataka postavljen na dva dana tada svaki redak koji je produciran može biti korišten sljedeća dva dana nakon čega se briše kako bi se oslobodio prostor.

Performansa Kafka-e je efektivno konstantna s obzirom na količinu podataka tako da spremanje podataka na više vremena ne predstavlja nikakav problem.

[8] TCP – (*eng. Transmission Control Protocol)* – jedan od osnovnih protokola IP grupe protokola.



Slika 17 Zapisivanje podataka u zapisnik (Izvor: Apache Kafka, 2017)

U stvari, jedini metapodatak na bazi potrošača ponaosob je ofset ili pozicija istog potrošača u zapisniku. Taj ofset kontroliran je od strane potrošača. Normalno bi potrošač trebao povećavati svoj ofset linearno kako čita podatke no, upravo zato što je ofset kontroliran od strane potrošača on može čitati podatke u redoslijedu kako on želi. Na primjer, potrošač može resetirati stari ofset kako bi ponovno procesirao podatke iz prošlosti ili dapače može preskočiti određeni ofset kako bi mogao krenuti od najnovijih podataka ako mu stari nisu potrebni.

Ovakva kombinacija znači da su Kafka-ini potrošači zapravo relativno ”jeftini”, odnosno, isti mogu doći ili otići bez da naprave preveliki utjecaj na klaster ili ostale potrošače.

Particije u zapisniku imaju višestruku svrhu. Prva je ta da dopuštaju zapisniku skaliranje iznad veličine koja bi stala na jedan server. Svaka individualna particija mora se moći spremiti na server s kojeg se poslužuje no, tema može imati više particija tako da može podržavati proizvoljan broj podataka. Druga je ta da se ponaša kao jedinica paralelizma.

4.2.3.2. Distribucija

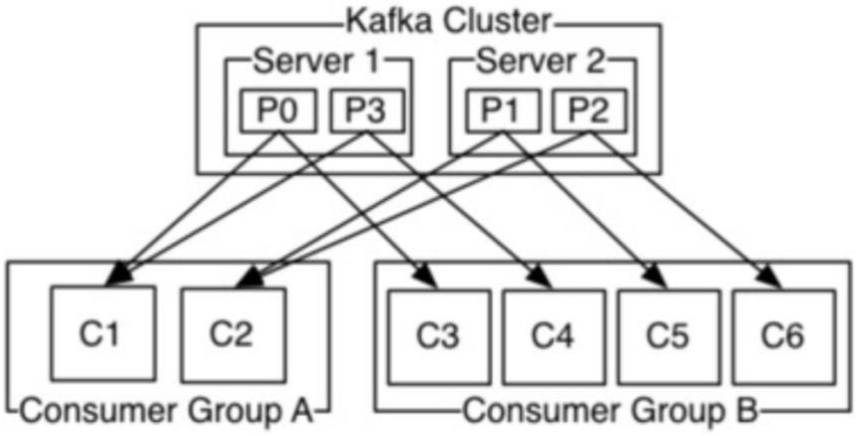
Particije zapisnika su distribuirane preko svih servera u Kafka-inom klasteru gdje svaki server rukuje s podacima kao i zahtjevima za dijeljenjem particija. Svaka particija je replicirana u konfiguriranom broju servera tako da omogućuje toleranciju na greške. Isto tako svaka particija ima jedan server koji se predstavlja kao vođa te nula ili više servera kao njegovi sljedbenici.

Vođa je taj koji upravlja svim zahtjevima za pisanjem ili čitanjem particije dok sljedbenici pasivno repliciraju vođu. Ako vođa prestane biti dostupan jedan od sljedbenika će automatski postati novim vođom.

Svaki server se ponaša kao vođa za poneki dio svojih particija te za ostatak kao sljedbenik. Razlog tomu je taj da se tako balansira opterećenje unutar klastera.

4.2.3.3. Potrošači

Potrošači se unutar Kafka-inog klastera označavaju kao potrošačka grupa (eng. *Consumer group*) te svaki redak koji je proizveden unutar teme također je spremljen u jednu potrošačku instancu unutar svake pretplaćene potrošačke grupe. Potrošačka instanca može biti odvojeni proces na istom ili na odvojenim strojevima. Ako sve potrošačke instance imaju isto ime potrošačke grupe tada će podaci biti efektivno raspoređeni preko svih potrošačkih skupina. U obrnutom slučaju svaki redak bit će emitiran svim potrošačkim procesima.



Slika 18 Potrošačke grupe (Izvor: Apache Kafka, 2017)

4.2.3.4. Kafka kao sustav razmjene poruka

Sustav razmjena poruka obično se sastoji od dva modela:

1. Čekanje u redu
2. Objavi-pretplati.

U modelu redova, bazen potrošača može čitati od jednog servera te svaki red se prosljeđuje pojedinom potrošaču dok je u drugom modelu taj redak proslijeđen svim potrošačima odjednom. Svaki od ova dva modela ima svoje prednosti i mane.

Prednost čekanja u redovima je ta što omogućuje da se procesiranje podataka može podijeliti u više potrošačkih instanci što omogućuje bolju skalabilnosti u procesiranju.

Nažalost, redovi ne omogućavaju postojanje višestrukog pretplaćivanja na određenu temu, jednom kada je proces čitanja gotov tada i podaci nestaju.

Druga metoda omogućuje emitiranje podataka više procesa no ne postoji šansa da se poboljša skalabilnost pošto svaka poruka odlazi svakom pretplatniku. Koncept potrošačke grupe u Kafka-i generalizira ova dva koncepta. S redovima, potrošačka grupa dopušta dijeljenje procesiranja nad kolekcijama procesa. S metodom objavi-pretplati, Kafka omogućuje emitiranje poruka više potrošačkih grupa. Kafka također ima jači i bolji način garancije sortiranja u odnosu na tradicionalne sustave razmjena poruka. Tradicionalni redovi čekanja zadržavaju podatke u redu na serveru te ako više pretplatnika krene crpiti podatke iz reda tada server krene slati podatke u onom redu na način na koji su spremljeni u početku. Kako god, iako server pruža podatke u redu, podaci su dostavljeni asinkrono potrošačima tako da oni mogu doći totalno ne sortirani različitim potrošačima.

Efektivno, ovo znači da je red podatak izgubljen u trenutku paralelne potrošnje. Sustavi za razmjenu poruka obično zaobilaze ovaj problem tako da naznačuju „ekskluzivnog potrošača“ koji dopušta samo jednom procesu da pribavlja podatke iz reda što naravno znači da više ne postoji paralelizam u procesiranju. Način na koji to Kafka radi je bolji. Imajući pojam paralelizma particija unutar teme, Kafka pruža kako garanciju reda tako i balansiranje opterećenja nad bazenom potrošačkih procesa. Ovakvo nešto je postignuto dodjeljivanjem particije u temi potrošaču u potrošačkoj grupi tako da je svaka particija potrošena od točno jednog potrošača unutar grupe. Radeći takvo nešto osigurava se da će potrošač biti jedini koji čita tu particiju te crpi podatke u redu. Kako postoji više particija ovaj način još uvijek balansira opterećenje nad više potrošačkih instanci. Ono što je bitno napomenuti je to da ne može biti više potrošačkih instance u potrošačkoj grupi nego particija.

5. Implementacija arhitekture

Kako bi mogli početi raditi i provjeriti sve mogućnosti što Kafka kao takva pruža te koje su navedene, potrebno je prvo postaviti Apache Kafka-u. Naime, postoje dvije verzije Apache Kafke, a to su Apache Kafka te Confluent.

Pošto će u nastavku biti potrebne dodatne značajke odnosno servisi koje Confluent pruža koristit će se Confluent-ov paket s kojim dolazi Apache Kafka koji je ranije preuzet[9].

5.1. Apache Zookeeper

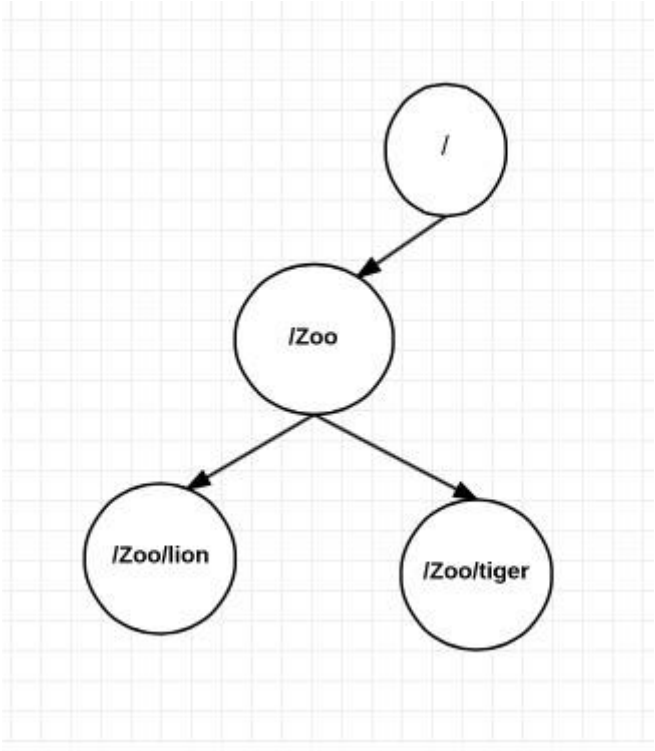
Prije nego se krene s radom trebalo bi napomenuti kako svaki servis, softver i sve što će se koristiti izravno surađuje sa servisom zvanim Zookeeper. Pošto niti jedan servis ne može funkcionirati bez instanci Zookeeper-a isti će predstavljati sam početak rada.

Apache Zookeeper je centralizirani servis koji održava konfiguracijske informacije, imenovanje, pruža distribuirano sinkroniziranje kao i grupiranje servisa. U nastavku će biti objašnjena svrha Zookeeper-a jer će isti biti potreban za održavanje raznih servisa kao što su Apache Kafka, Apache Hadoop, Apache YARN te slični.

Na najvećoj razini kao što je napomenuto Zookeeper predstavlja servis na koji se klijenti spajaju. Isti pruža pristup klijentima u strukturi oblika stabla ili kako još u dokumentaciji za Zookeeper-e navedeno, strukturi hijerarhijskih imenskih prostora. Stablo kao takvo potrebno je kako bi mogli izvršavati veliki set operacija brisanja, pisanja, ažuriranja te kreiranja. Isto tako omogućuje operacije dohvaćanja (eng. *GET)* te postavljanja (eng. *SET)* za manipuliranje podacima. Kao standard se koristi UNIX notacija za datotečni sustav putanja.

[9] https://www.confluent.io/download/

Na primjer, recimo da koristimo /A/B/C kako bi označili putanje do čvora C, gdje C ima roditelja B te B kao svojeg roditelja A. U nastavku slijedi primjer:



Slika 19 Hijerarhijsko stablo

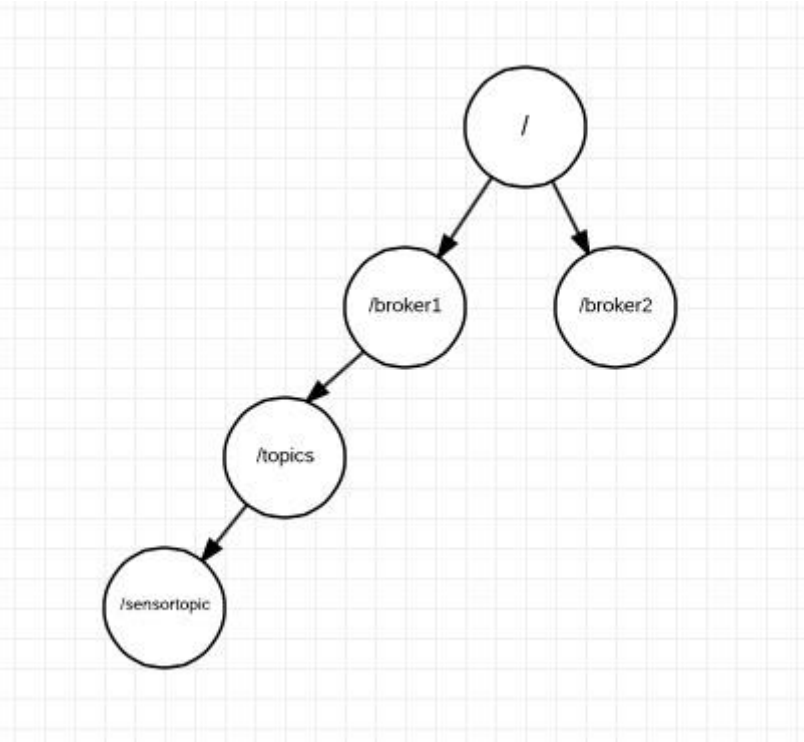
Nadalje, terminologija koja će nadalje koristiti je sljedeća:

- Svaki čvor stabla zove se z*Node*
- Svaki z*Node* u stablu identificiran je putanjom
- z*Node* tipovi su – perzistentan i kratkotrajan
- Svaki z*Node* sprema vrijednost ili podatak te može predstavljati čvor-dijete
- Ne može se dodati ili izbrisati „sat[10]“ z*Node-a*.

Način na koji Kafka koristi Zookeeper je za spremanje različitih konfiguracija u obliku ključ-vrijednost na Zookeeper-ovo podatkovno stablo. (Landoop, 2017)

[10] Sat – predstavlja neku oznaku za putanju z*Node-a* da te obavijesti ukoliko se nešto promijenilo. Primjer toga može biti pretplata na promjene u putanji.

Primjer dva Kafka-ina čvora koji se još zovu „*brokeri*“ gdje jedan čvor sadrži teme dok drugi trenutno ne:



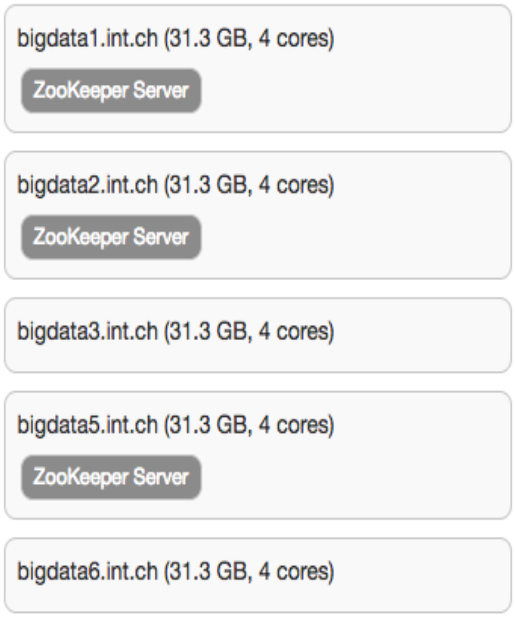
Slika 20 Odnos Zookeepera i Kafke

Nakon što su objašnjene pojedinosti potrebno je krenuti s pokretanjem Zookeeper-ovih servisa. Kako bi se napravio klaster puno elastičnijim, a kada se kaže elastičnijim misli se na otpornijim na ispade, postavit će se tri Zookeeper instance na tri različita servera. Zookeeper radi na temelju kvoruma što bi značilo da većina instanci mora raditi kako bi servis mogao biti stabilan. Formula po kojoj se računa je sljedeća

$$2n + 1$$

Oznaka *n* predstavlja broj instanci koje smo spremni izgubiti. U ovom slučaju je to broj jedan što bi značilo da su potrebne tri instance kako bi se mogao pretrpjeti gubitak jedne instance te kako bi sustav i dalje bio stabilan. Zookeeper servis instalirat će se i konfigurirati preko Hortonworks platforme koja je u početku postavljena.

Trenutni izgled klastera je sljedeći:



Slika 21 Trenutno postavljeni servisi

5.2. Pokretanje Apache Kafke

Nakon što su napokon pokrenuti ključni servisi o kojima manje-više cijela arhitektura ovisi, može se krenuti s postavljanjem ostatka servisa. Kafka brokere će se postaviti po čvorovima nazvanim *node1, node3, node4.*

Kako bi se pokrenuli Kafka brokeri mora se dobro pripaziti na konfiguriranje datoteke imena *server.properties* koja se nalazi u direktoriju */confluent-4.0.0/etc/kafka/*. Da se ne prikazuje cijela datoteka sa svim svojstvima koja su dostupna, dalje u tekstu bit će prikazana najbitnija svojstva na koja je potrebno obratiti pozornost.

```
1.  # Brokerov id.Potrebno je da bude unikatan.
2.  broker.id=1
3.
4.  delete.topic.enable=true
5.  default.replication.factor=3
6.  compression.type=lz4
7.
8.  # Lista direktorija koji su odvojeni zarezima gdje će se spremati Kafkini logovi
9.  log.dirs=/hadoop/kafka-logs
10. zookeeper.connect=node1:2181,node2:2181,node5: 2181
```

Nakon što se pripremila datoteka sa svojstvima potrebno je napraviti servis koji će pokrenuti Kafka-in broker, održavati ga te ako dođe do bilo kakvog ispada automatizirano ga pokrenuti kao da se ništa dogodilo nije.

Oblik je to *systemd* skripte koja će se smjestiti u */etc/systemd/system* gdje se i nalaze servisi slični istome.

```
1.  [Service]
2.
3.  Environment = KAFKA_HEAP_OPTS = -Xmx6g -Xms6g
4.  Environment = JMX_PORT = 9000
5.  Environment = KAFKA_JVM_PERFORMANCE_OPTS = -XX: MetaspaceSize = 96 m - XX: +UseG1GC -
     XX: MaxGCPauseMillis = 20 - XX: InitiatingHeapOccupancyPercent = 35 -
     XX: G1HeapRegionSize = 16 M - XX: MinMetaspaceFreeRatio = 50 -
     XX: MaxMetaspaceFreeRatio = 80
6.  ExecStart = /confluent-4.0.0/bin/kafka-server-start /confluent-
    4.0.0/etc/kafka/server.properties
7.  Restart = always
8.  RestartSec = 3
9.
10. [Install]
11. WantedBy = default.target
```

Nakon što je napravljena skripta potrebno je osvježiti listu te pokrenuti servis naredbom:

```
1.  systemctl daemon-reload & systemctl start kafka-server-v1.service
```

Isti postupak treba napraviti na svakom od čvorova gdje je rečeno da će se postaviti Kafka broker.

5.3. Registar shema

Registar shema (eng. *Schema registry)* pruža sloj posluživanja metapodacima. U registru kao takvom postoji upravljanje verzijama shema tako da je moguće vidjeti koje su se sheme do sada koristile ili se koriste. Glavni zadatak registra shema je taj da pruža RESTful[11] sučelje za spremanje/dohvaćanje AVRO shema kao i serijalizaciju poruka koje pristižu u AVRO formatu u Kafka-u.

[11] RESTful (eng. *Representational state transfer*) – akronim za protokol koji omogućuje komunikaciju na Internetu pomoću programskih jezika

Način pokretanja registara sheme je sličan pokretanju brokera. Nakon što se uredi datoteka */confluent-4.0.0/etc/schema-registry/schema-registry.properties*, potrebno je napraviti servis te pokrenuti instancu registra shema.

Registra shema radi na principu gospodar – rob te će se iz razloga kao takvoga pokrenuti dvije instance. Instance će se nalaziti na čvorovima *node1* te *node4*.

Oblik datoteke koju je potrebno namjestiti da bi servis kao takav radio izgleda na sljedeći način:

```
kafkastore.connection.url = node1:2181, node2:2181, node5:2181

# Ime teme gdje će biti spremljene sheme
kafkastore.topic = _schemas

host.name = node1
access.control.allow.methods = GET, POST, OPTIONS
access.control.allow.origin = *
```

Izgled servisa koji je napravljen te smješten u */etc/systemd/system* je:

```
#[Unit]

[Service]
 ExecStart = /confluent-4.0.0/bin/schema-registry-start
                    /confluent-4.0.0/etc/schema-registry/schema-registry.properties
 Restart = always
 RestartSec = 3

[Install]
WantedBy = default.target
```

Nakon izvršavanja zadnje skripte te pokretanje servisa isti je moguće pronaći na ulazu 8081 na oba čvora gdje je pokrenut. Važno je napomenuti kako je samo jedan registar od dva podignuta gospodar te samo gospodar može odgovarati na zahtjeve.

5.4. Pokretanje Kafka-Connect klastera

Kafka-Connect je programski okvir koji dolazi kao takav u paketu s Apache Kafka-om te pruža integraciju Apache Kafka-e s drugim sustavima. Poanta okvira kao takvog je u omogućavanju skalabilnosti procesa prijenosa podataka prema bazama podataka kao što su MySQL Server, Microsoft SQL Server, Cassandra te mnogih drugih.

Kako bi se podaci mogli kopirati između Kafka-e te drugih sustava potrebno je koristiti Kafka-ine konektore koji omogućuju takav način povlačenja ili puštanja podataka u/iz Kafka-e.

Postoje dvije vrste konektora, a to su:

1. Izvorišni konektori – omogućuju puštanje podataka iz drugih sustava u Kafka-u
2. Odredišni konektori – omogućuju povlačenje podataka iz Kafka-e ka drugim sustavima.

Kako bi se mogli koristiti ti isti konektori potrebno ih je preuzeti[12]. Nakon preuzimanja istih, raspakirat će se u */confluent-4.0.0* direktorij te će se u nastavku objasniti primjena.

Postoje dva načina rada, a to su samostalni te distribuirani. Razlika u načinima rada je što se u distribuiranom načinu rada svakim novim članom Kafka-Connect-a povećava paralelizam kao i otpornost na ispad. Pošto je cilj izgraditi skalabilnu arhitekturu jasno je da će se iskoristiti prednosti koje dolaze s distribuiranim načinom rada.

Članovi Kafka-Connect-a zovu se još radnici. Klaster koji se izgrađuje sadržavat će dva radnika gdje će svaki čvor imati jednog, a oni su *node1* te *node4*. Način pokretanja radnika je sličan pokretanju brokera te registara shema samo naravno drugačijih svojstava.

U datoteci koja se nalazi na putanji *confluent-4.0.0/etc/schema-registry/connect-avro-distributed.properties* moguće je pronaći predložak kojeg je potrebno namjestiti tako da odgovara sustavu koji se izgrađuje.

[12] https://lenses.stream/connectors/download-connectors.html#downloadconnectors

```
# Kafka brokeri
bootstrap.servers = node1:9092, node3:9092, node4:9092

# Unikatno ime grupe radnika klastera
group.id = connect-cluster-main

key.converter = io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url = http://node1:8081
value.converter = io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url = http://node1:8081
config.storage.topic = connect-configs-main
offset.storage.topic = connect-offsets-main
status.storage.topic = connect-statuses-main
rest.port = 8083
plugin.path = /confluent-4.0.0/stream-reactor-1.0.0-1.0.0/libs,
/confluent-4.0.0/share/java/kafka-connect-jdbc
```

Jedina je razlika što ovaj put treba paziti da se i konektori koju su preuzeti dodaju u klaster s kojim se radi. Naime, upravo ti konektori kao što je bilo rečeno dat će mogućnost protoka podataka između različitih sustava.

Uz konektore koji su preuzeti potrebno je preuzeti i upravljački program za MySQL Server pošto će se isti koristiti kao izvorište podataka.

Isto tako i servis koji će biti smješten smjestiti u */etc/systemd/system* direktorij:

```
[Service]
Environment = CLASSPATH=/confluent-4.0.0/calcite-linq4j-1.13.0.jar
:/confluent-4.0.0/share/java/kafka-connect-jdbc/mysql-connector-java-6.0.6- bin.jar
:/confluent-4.0.0/share/java/kafka-connect-jdbc/sqljdbc4-3.0.jar

Environment = KAFKA_HEAP_OPTS = -Xmx9g - Xms9g
Environment = KAFKA_JVM_PERFORMANCE_OPTS = -server - XX: MetaspaceSize = 96 m -
 XX: G1HeapRegionSize = 16 M - XX: +UseG1GC - XX: MinMetaspaceFreeRatio = 50 -
 XX: MaxMetaspaceFreeRatio = 80 - XX: MaxGCPauseMillis = 20 -
 XX: InitiatingHeapOccupancyPercent = 35 - XX: +DisableExplicitGC -
 Djava.awt.headless = true

ExecStart = /confluent-4.0.0/bin/connect-distributed
 /confluent-4.0.0/etc/schema-registry/connect-avro-distributed.properties

Restart = always
RestartSec = 3

[Install]
WantedBy = default.target
```

5.5. Landoop korisnička sučelja

Svaki servis koji se nalazi u sklopu arhitekture do sada je predstavljen kroz konzolni način rada. U svrhu lakšeg praćenja rada, snalaženja u arhitekturi kao i dobivanja šireg spektra, predstavit će se Landoop kao projekt koji pruža korisnička sučelja za servise povezane s Kafka-om.

Naime, tri su korisnička sučelja koja će se koristiti. Prethodno je bilo govora o temama u Kafka-i, registrom shema te Kafka-Connect-om. Za svaki od navedenih servisa postoji korisničko sučelje te će u nastavku svako biti predstavljeno i opisano.

Kada se Kafka kao servis pokrenuo mogle su se pratiti teme tako da se upisala naredba izravno na čvoru gdje je servis instaliran kao:

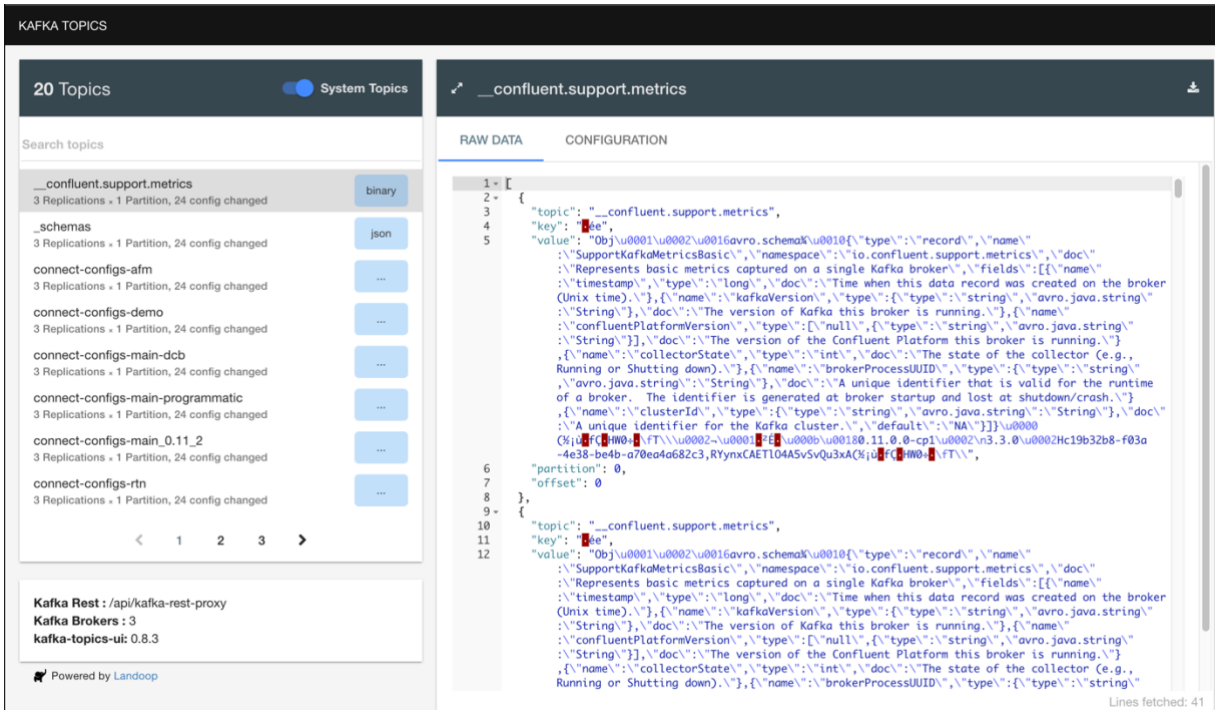
```
1. bin/kafka-topics --list --zookeeper zookeeper-node-ip:2181
```

Kako bi se izbjeglo konstantno upisivanje naredba te olakšao pristup informacijama preko korisničkog sučelja, Landoop je predstavio projekt nazivom „Kafka-Topics-UI“. Način na koji će se instalirati je koristeći *docker*.

Docker je platforma koja kao arhitekturalni pristup omogućuje virtualizaciju u obliku takozvanih kontejnera prilikom čega se dolazi do bolje iskoristivosti kako postojećih tako i novih resursa. (Docker Inc., 2018)

Nakon što se na platformi instalirao docker, potrebno je upisati sljedeću naredbu koja će preuzeti s vanjskog repozitorija kontejner na kojem se nalazi korisničko sučelje koje je potrebno.

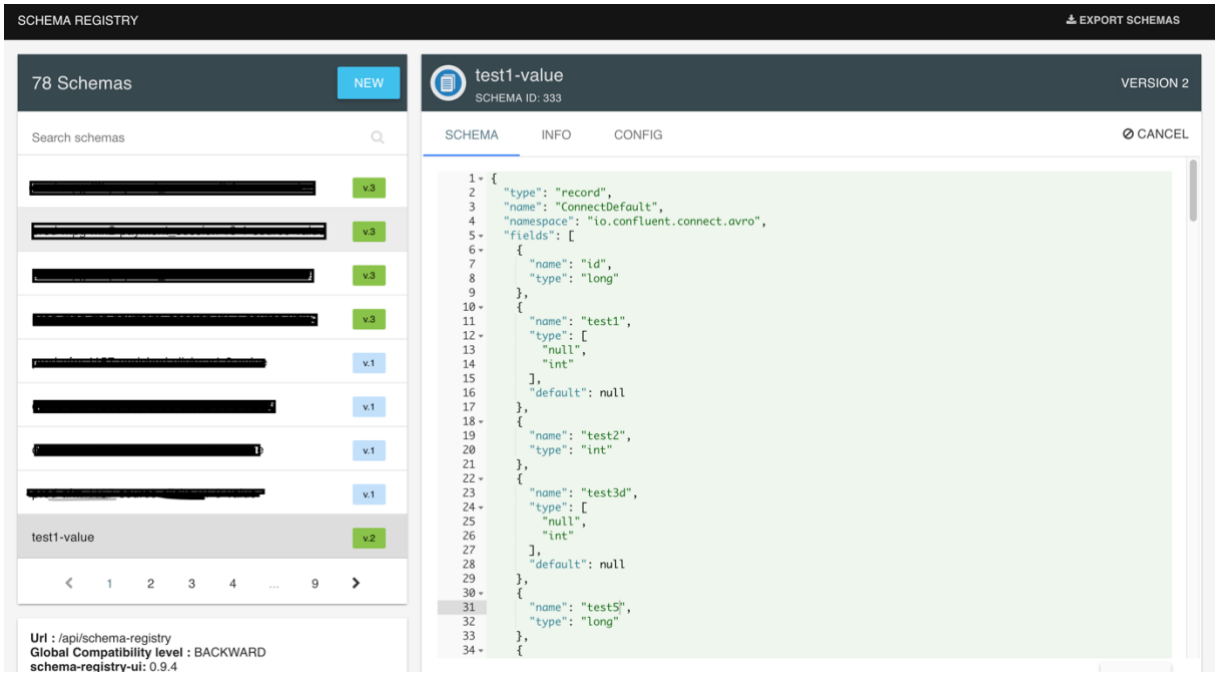
```
1. bin/docker run    --rm -i -p 8000:8000 -e    "KAFKA_REST_PROXY_URL=http://node3:8082"     -
   e "SCHEMAREGISTRY_URL=http://node1:8081" - e "PROXY=true"
   docker.io/landoop/kafka-topics-ui:0.9.3
```



Slika 22 Prikaz korisničkog sučelja tema

Nadalje, kako se konzolni pristup ne bi morao ponavljati za registar shema kao i za Kafka-Connect, preko docker-a će se pokrenuti korisničko sučelje za oba. Za registar shema potrebno je upisati sljedeću naredbu (Landoop, 2017):

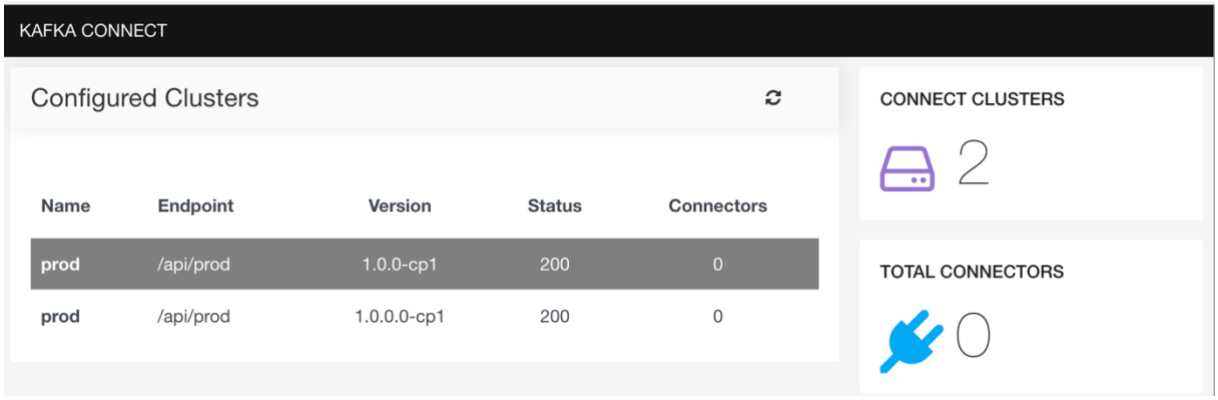
```
1. bin/docker run--rm -i -p 8000: 8000 -e "SCHEMAREGISTRY_URL=http://node1:8081"
- e "PROXY=true"  docker.io/landoop/schema-registry-ui:0.9.4
```



Slika 23 Prikaz korisničkog sučelja registara shema

Za provjeru Kafka-Connect potrebno je upisati sljedeću naredbu (Landoop, 2017):

```
1.  bin/docker run --rm -i -p 8000:8000
      -e "CONNECT_URL=node1:8083;prod,node4:8083;prod“ docker.io/landoop/kafka-connect-
    ui:0.9.4
```



Slika 24 Prikaz korisničkog sučelja Kafka-Connect-a

5.6. Cassandra

Cassandra je distribuirani sustav baza podataka kojoj je glavna uloga upravljanje s velikom količinom strukturiranih podataka na više servera te pritom pružanje visoko dostupne usluge bez postojanja dijela sustava gdje može doći do onesposobljavanja cijelog sustava. Ujedno u navedenoj definiciji se može doći do odgovora zašto je upravo ovaj tip baza podataka odabran kao najbolji primjer distribuirane baze podataka. Samo neke od prednosti po kojima se Cassandra kao sustav baza podataka ističe su:

1. Kontinuirana dostupnost
2. Linearno skaliranje performansi
3. Jednostavno izvršavanja operacija
4. Jednostavna distribucija podataka preko više podatkovnih centara
5. Zone dostupnosti u obliku oblaka (eng. *Cloud*)

Upravo zato što je Cassandra-ina arhitektura složena na netradicionalan način odnosno ne u gospodar-rob (eng. *Master-slave*) konfiguraciji već u konfiguraciji bez glavnog čvora nazvanoj „prsten“ razlog je u kojem se kriju sposobnosti gdje se omogućava skaliranje, djelovanje te kontinuirani odnosno neprekidni rad. Konfiguracija u obliku „prstena“ je ta koja Cassandra-i daje dodatnu eleganciju te jednostavnost u radu. Kod takvog tipa konfiguracije distribuirane baze podataka, svaki čvor sustava ima identičnu ulogu gdje svaki komunicira sa svim čvorovima sustava na jednaki način. Cassandra-ina arhitektura još se zove i „građena za skaliranje“ što znači da može jednostavno rukovati s ogromnim količinama podataka te tisućama korisnika i operacija istovremeno čak i preko više podatkovnih centara u istoj mjeri koliko i s malom količinom podataka. Kao pravi pokazatelj koliko se zapravo Cassandra u stvarnosti koristi primjer su velike organizacije koje su je izabrale kao temelj gradnje njihovih sustava, a to su: Apple, eBay, Spotify, Über te slični. (Planet Cassandra 2015)

Svaki čvor u sustavu ima tri odvojene particije odnosno diska. Na čvorove *node2* do *node6* postavit će se Cassandra-in takozvani prsten što znači da će arhitektura sada sadržavati Cassandra-u kao servis skaliranu na pet čvorova. Upravo količina od pet čvorova kao i replikacijski faktor tri koji će kasnije biti postavljen osigurava funkcioniranje sustava i do ispada od čak dva čvora.

5.7. Apache Hadoop

Apache Hadoop je projekt koji je napravljen kao programsko sučelje koje omogućuje distribuirano procesiranje velikih količina podataka preko cijelog klastera koristeći jednostavan programski model. Dizajniran je tako da se skalira od jednog do tisuću čvorova gdje svaki omogućuje lokalno računanje kao i skladištenje. Umjesto da se oslanja na hardver kako bi se ostvarila visoka dostupnost, projekt kao takav je dizajniran je tako da sam detektira te rukuje s greškama na aplikacijskom sloju. (Apache Hadoop, 2018)

Slika 25 Instalacija Hadoop-a

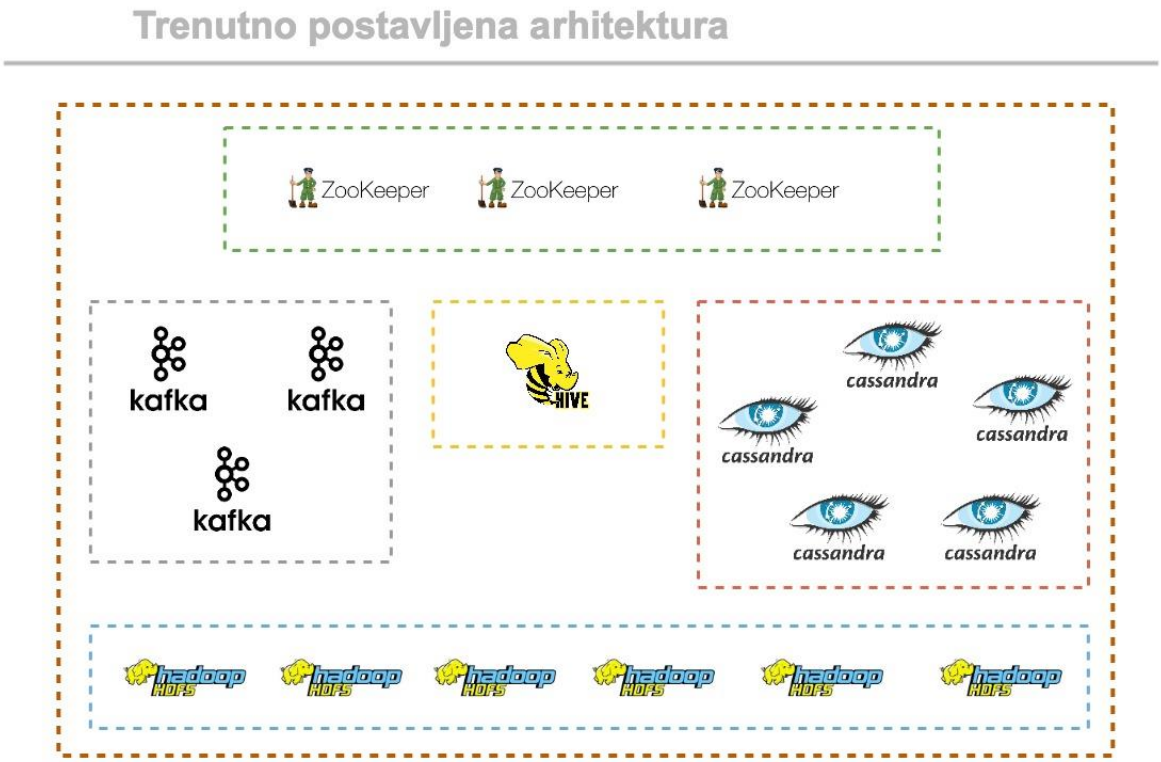
Kao što je vidljivo instalirana su dva *NameNode-a*. Naime, *NameNode* je servis koji održava Hadoop kao sustav. U ovom slučaju jedan je aktivan dok je drugi postavljen kao servis u pripravnosti koji se uključuje kao aktivan ako aktivan zakaže.

5.8. Apache Hive

Apache Hive je projekt kreiran za skladištenje podataka usmjeren ka čitanju, pisanju te upravljanju velikim količinama podataka koji se nalaze u distribuiranom načinu skladištenja putem strukturiranog jezika za upite (eng. *Structured Query Language – SQL-style)*. Isti je postavljen na čvoru „*node6*“. (Apache Hive, 2014)

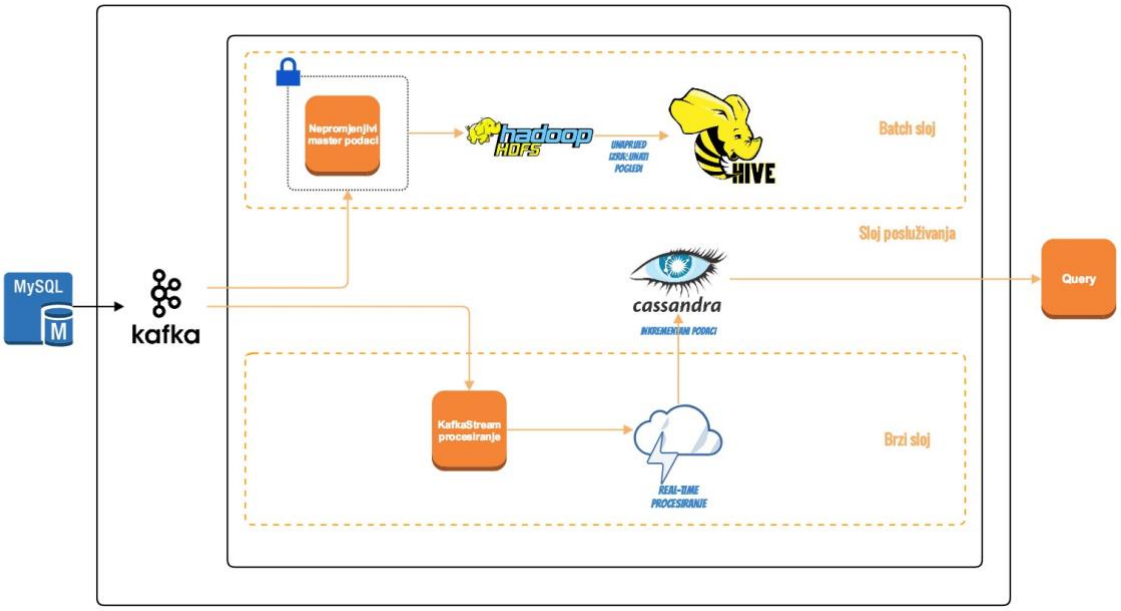
5.9. Trenutni izgled arhitekture

Nakon što su postavljeni svi servisi koji su potrebni za funkcioniranje skalabilne arhitekture koja može procesirati velike količine podataka u nastavku će biti prikazan trenutni izgled iste.



Slika 26 Trenutni izgled arhitekture

Isto tako, prethodno je bilo govora o Lambda arhitekturi kao glavnom osloncu i konceptu po kojemu se izgrađivala ista. Na sljedećoj slici prikazana su tri sloja o kojima je ranije bilo govora, samo ovoga puta sa servisima koji su ranije postavljeni.



Slika 27 Prikaz slojeva

Vidljivo je kako batch i brzi sloj nisu u potpunosti spojeni u sloju posluživanja. Razlog je što za ovu svrhu isto nije potrebno te je Cassandra uzeta kao predstavnik sloja posluživanja. Hadoop u ovom slučaju služi kao jedinstveni izvor istine (eng. *Single Source of Truth – SSOT*).

6. Obrada podataka

U samom početku spomenuto je više Kafka-inih koncepata te je jedan od njih bio potrošač. U ovom poglavlju obradit će se jedan tip potrošača koji će konstantno biti u doticaju s jednom izvorišnom točkom, na primjer nekom relacijskom bazom podataka. Također, ovo poglavlje je ujedno i uvod u implementaciju brzog sloja. Zadatak takvog koncepta bit će konstantno ili uz zadano vrijeme traženja promjena u tablici neke relacijske baze podataka. Svaki pokušaj proizvodnje podataka odnosno izvlačenja podataka iz tablice zapisat će se u dvije teme istodobno. Jednu temu koja će pratiti vremenske oznake kada se upit prema relacijskoj bazi podataka ispunio te drugu temu u koju će povući podatke i spremiti ih. Ako se nije dogodila niti jedna promjena u relacijskoj bazi podataka zapisat će se samo jedan zapis u temu koja zapisuje ofset te vremensku oznaku kako bi sljedećim pokušajem konektor mogao nastaviti s točkom gdje je stao. Na ovaj način se osigurava da će svaki podatak čim stigne u tablicu neke relacijske baze podataka ili u nekom zadanom okviru biti zabilježen i u Kafka-i.

Primjer takvog konektora koji će služiti danjem radu je:

```
1. connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
2. mode=incrementing
3. incrementing.column.name=id
4. topic.prefix=demo-clicks-v1-0-0,
5. tasks.max=1
6. poll.interval.ms=1000
7. query=select id, cl.affiliate_id,cl.offer_id, cl.revenue_type, cl.payout_type,
   cl.snippet_code, ifnull(cl.user_agent,'Unknown') as user_agent, ifnull(cl.creative_id,-1)
   as creative_id, cl.fake_click,cl.time_expired as time_expired,
   REPLACE(cl.enrichment_parameters,SUBSTRING_INDEX(cl.enrichment_parameters, '', 1),'') as
   enrichment_parameters, CAST(DATE_FORMAT(time_clicked, '%Y%m%d%H') AS SIGNED) as datekey
   FROM click_raw cl",
8. name=demo-clicks-v1-0-0
9. connection.url=jdbc:mysql://[(host=domain,port=3306, user=some_user, password=password)]/db
   ?useLegacyDatetimeCode=false&serverTimezone=Europe/Zagreb
```

Naime, u gore navedenoj logici navedeno je osam najbitnijih parametara koji se moraju zadovoljiti kako bi neki konektor mogao uopće biti pokrenut te će u nastavku biti objašnjen svaki pojedinačno.

Parametri:

1. **connector.class** - služi za čitanje klase koja će obraditi sve ostale parametre
2. **mode** - opis načina praćenja ključa u upitu. Osigurava da upiti koji se budu izvršavali prate definirani ključ slijedno, kao što je definirano u bazi podataka. Obično se za takav ključ postavlja primarni ključ tablice i vremensku oznaku. Parametar se strogo nadovezuje na sljedeći parametar
3. **incrementing.column.name** - kao što je navedeno u parametru iznad, za ovaj parametar se postavlja ključ koji se nalazi u upitu te preko kojeg će se pratiti izvršavanje i dohvaćanje podataka
4. **task.max** - postavlja se maksimalni broj instanci koje će istovremeno asinkrono raditi i dohvaćati podatke, obično se odnosni na paralelizam
5. **poll.interval.ms** - broj milisekundi kada će instanca ponovno pokušati prikupiti podatke iz izvorišne tablice
6. **query** - upit kojeg instanca šalje u izvorišnu tablicu
7. **name** - ime konektora
8. **connection.url** - URL do izvorišne tablice.

No, postoji jedan izazov koji se događa prilikom kreiranja konektora unutar jednog klastera na Kafka-Connect-u. Kada se kaže klaster Kafka-Connect-a tada se misli na dva ili više radnika (eng. *Workers*) koji imaju iste konfiguracije te obnašaju iste dužnosti. Trenutno u arhitekturi postoji samo jedan klaster koji je sačinjen od dva čvora. Povećanjem broja čvorova unutar jednog klastera povećava se broj radnika što znači da se može postići veći paralelizam prilikom definiranja konektora i *„task.max“* parametra. Nakon što se kreira konektor ili se izmjeni istom neka opcija, sustav ponovno učita konfiguraciju svih konektora koji postoje. Ako klaster ima recimo nekoliko desetaka konektora te je potrebno jednom promijeniti opcije ili jednostavno dodati novi konektor, svih nekoliko desetaka konektora će prestati raditi na otprilike desetak sekundi dok sustav ne pročita sve otpočetka.

Takav način rada potencijalno narušava rad produkcijskih sustava jer aplikacije koje moraju biti u toku sa stvarnim vremenom počinju kaskati te nemaju dotok podataka.

Katastrofalno može biti ako jedan konektor prestane raditi te je potrebno utvrditi zašto ne radi, mijenjati više puta konfiguracije i slično. U tom trenutku svi produkti koji se nalaze na tom klasteru krenu kaskati te takav način rada može biti poguban za poslovanje.

No, postoji način kako da se istom izazovu doskoči te se rješenje zove *izolacija*. Rješenje je veoma jednostavno te jedino što iziskuje kao kompenzaciju je dodatno resursa. Naime, svaki puta kada se gradi novi klaster učitavaju se i svi priključci koji su potrebni kako bi se mogli kreirati konektori kao na primjer priključci za Cassandru, Hadoop te slični. Svaki taj priključak sam po sebi zauzima kao mala aplikacija određeni postotak memorije no cijena je mala u odnosu na rezultat do kojeg se dolazi. Izolacija se odrađuje tako da svaki radnik unutar novog klastera koji se kreira mora imati nove vrijednosti za sljedeće varijable:

1. group-id
2. config.storage.topic
3. offset.storage.topic
4. status.storage.topic
5. rest.port

Nakon što se kreira novi klaster gdje svi radnici imaju iste opcije moguće je kreirati i nove konektore na istom. Izolacija se može vršiti u odnosu na više segmenata. Može biti u odnosu na projekte tako da svaki projekt ima odvojen svoj sloj te se ne isprepleće s ostalim projektima ili u odnosu na verzije priključaka tako da se projekti nalaze na istim ili različitim verzijama. Rezultat izolacije može se primijetiti tek nakon što se na jednom od konektora dogodi neka greška, isti prestane raditi, a na desetine drugih konektora se niti ne osjeti promjena ili greška koja se dogodila.

Nakon što je kreiran konektor te su podaci vidljivi kao na *Slika 28*, potrebno je napraviti mikroservis koji će uređivati odnosno obogaćivati na svojevrstan način podatke te iste spremati u novokreiranu temu.

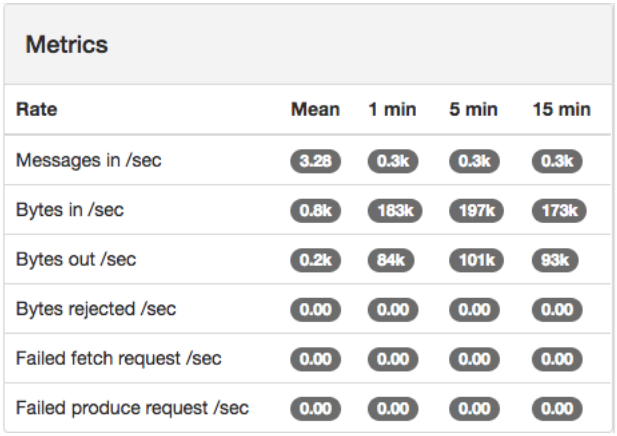
Mikroservis je pisan u programskom jeziku Scala. Prilikom kreiranja mikroservisa potrebno je znati par svojstava koji će biti potrebni za dohvaćanje podataka. Naime, potrebno je znati URL do sheme registara, Kafka brokera te u konačnici iz koje će se teme dohvaćati podaci. Glavni zadatak mikroservisa bit će raščlanjivanje kolone „enrichment_parameters“ koja je tekstualnog oblika te koja trenutno sadrži ugniježđene parametre. U kodu koji slijedi prikazan je način na koji se raščlanjuje kolona „enrichment_parameters“.

```
1.  private void parseParams(String params) {
2.      Stream < String > lines = Pattern.compile("\\n").splitAsStream(params);
3.      lines.forEach(z - > {
4.          String[] y = z.split("=");
5.          if (y.length > 1) {
6.              switch (y[0]) {
7.                  case "device_brand":
                        setDeviceBrand(truncate(y[1], 30));
8.                      break;
9.                  case "device_model":
                        setDeviceModel(truncate(y[1], 30))
10.                     break;
11.                 case "device_os":
                        setDeviceOs(truncate(y[1], 25));
12.                     break;
13.                 case "device_type":
                         setDeviceType(truncate(y[1], 20));
14.                     break;
15.                 case "country_name":
                         setCountry(truncate(y[1], 64));
16.                     break;
17.                 case "region_name":
                         setRegion(truncate(y[1], 64));
18.                     break;
19.                 case "isp_name":
                        setIsp(truncate(y[1], 70));
20.                     break;
21.                 case "isp_type":
                         setIspType(truncate(y[1], 20));
22.                     break;
23.             }
24.         }
25.     });
26. }
```

Nadalje, prikazan je odlomak koda koji prikazuje postavljanje svih kolona te kreiranje nove teme u kojoj se nalaze nove kao i obogaćene stare kolone.

```
1.  private GenericRecord eventRecord(GenericRecord v, String installationId) {
2.
3.      Parameters params = new Parameters(v.get("enrichment_parameters").toString());
4.      GenericRecord g = createRecordValue();
5.      g.put("ClickId", v.get("id"));
6.      g.put("AffiliateId", v.get("affiliate_id"));
7.      g.put("OfferId", v.get("offer_id"));
8.      g.put("RevenueType", v.get("revenue_type"));
9.      g.put("SnippetCode", v.get("snippet_code"));
10.     g.put("UserAgent", v.get("user_agent"));
11.     g.put("PayoutType", v.get("payout_type"));
12.     g.put("CreativeId", v.get("creative_id"));
13.     g.put("FakeClick", v.get("fake_click"));
14.     g.put("TimeExpired", v.get("time_expired"));
15.     g.put("EnrichmentParameters", v.get("enrichment_parameters"));
16.     g.put("DateKey", v.get("datekey"));
17.     g.put("Country", params.getCountry());
18.     g.put("Region", params.getRegion());
19.     g.put("DeviceOs", params.getDeviceOs());
20.     g.put("DeviceManufacturer", params.getDeviceBrand());
21.     g.put("DeviceModel", params.getDeviceModel());
22.     g.put("DeviceOsVersion", params.getDeviceOsVersion());
23.     g.put("DeviceType", params.getDeviceType());
24.     g.put("Isp", params.getIsp());
25.     return g;
26.   }
27. }
```

Ako se prebaci pogled na analizu može se uvidjeti broj poruka koje pristižu u Kafka-u trenutno. Na slici je prikazana analiza po minutama i sekundama. Kako podaci tek pristižu može se vidjeti da broj poruka koje dođu u sekundi u Kafka-u je otprilike „0.3k“ odnosno 300.



Slika 28 Prikaz metrike brokera

6.1. Priprema podataka za streaming sloj

Nakon što su podaci obrađeni i pripremljeni iste je potrebno prenijeti u Cassandra-u. Na isti način kao što su podaci dohvaćani iz MySQL Servera te preneseni u Kafka-u tako će biti preneseni preko konektora iz Kafka-e u Cassandra-u. Prije nego se kreira konektor potrebno je naravno pripremiti tablicu u koju će podaci biti spremani. U sljedećem primjeru prikazan je izgled tablice u Cassandra-i.

```
CREATE TABLE demo.demo_enriched_clicks
 (clickid bigint, datekey int, affiliateid int, offerid int,
 revenuetype text, payouttype text, snippetcode text, useragent text,
 timeexpired bigint, creativeid bigint, fakeclick int,
 enrichementparameters text, region text, country text,
 deviceos text, devicemanufacturer text,
 devicemodel text, devicetype text, deviceosversion text,
 isp text, isptype text,
PRIMARY KEY(deviceos, fakeclick), datekey, clickid));
```

Kako je tablica pripremljena, sljedeći korak je priprema konektora čiji će zadatka biti real-time prebacivanje podataka kako je prikazano na *Slika 27.*

```
connector.class=com.datamountaineer.streamreactor.connect.cassandra.sink.CassandraSinkConnector
connect.cassandra.key.space=demo
connect.cassandra.contact.points=node2,node3,node4,node5,node6
topics=demo-clicks-enriched-v1-0-0
tasks.max=1
connect.cassandra.kcql=INSERT INTO demo_enriched_clicks SELECT ClickId, DateKey, AffiliateId, OfferId, RevenueType, PayoutType, SnippetCode, UserAgent, TimeExpired, CreativeId, FakeClick, Region, Country, DeviceOs, DeviceManufacturer, DeviceModel, DeviceType, DeviceOsVersion, Isp, IspType FROM demo-clicks-enriched-v1-0-0
connect.cassandra.port=9042
name=CassandraSinkConnector
connect.cassandra.password=password
connect.cassandra.username=user
```

Kako je tablica u Cassandra-i modelirana samo za specifične upite, do podataka je moguće doći samo sa specifičnim upitima. Primjer upita vidljiv je u sljedećem primjeru.

```
SELECT * FROM demo_enriched_clicks WHERE deviceos = 'Windows' AND fakeclick = 0 AND datekey
 = 2018060519 AND clickid = 10476510271;
```

Nažalost zbog zaštite podataka neće biti prikazane sve vrijednosti podataka, no većina je vidljiva na sljedećoj slici.

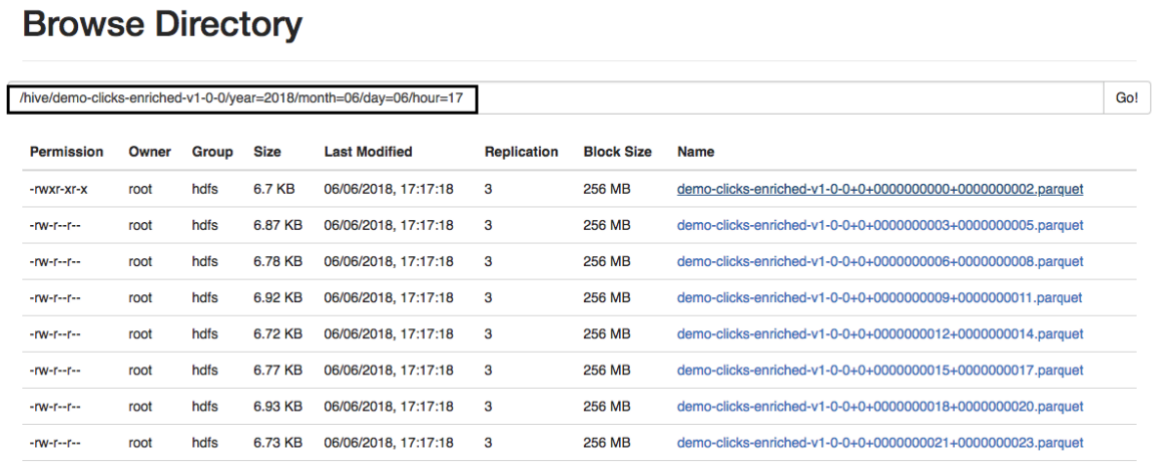
Slika 29 Prikaz podataka upita nad Cassandra-om

6.2. Priprema podataka za batch sloj

Nakon što su podaci obogaćeni iste će se spremati na Hadoop kao SSOT kao što je ranije navedeno. Princip spremanja podataka je isti kao i u prethodnom primjeru s Cassandra-om samo što ovoga puta nije potrebno kreirati tablicu već su podaci spremani u obliku posebnih datoteka zvanih Parquets[13]. Izgled konektora koji će u određenim intervalima prebacivati podatke na Hadoop je prikazan u sljedećem primjeru.

```
1.  connector.class=io.confluent.connect.hdfs.HdfsSinkConnector
2.  schema.compatibility=BACKWARD
3.  flush.size=3
4.  topics=demo-clicks-enriched-v1-0-0
5.  tasks.max=2
6.  timezone=CET
7.  hdfs.url=hdfs://node1:8020
8.  hive.metastore.uris=thrift://node6:9083
9.  locale=en-eu
10. format.class=io.confluent.connect.hdfs.parquet.ParquetFormat
11. hive.integration=true
12. partitioner.class=io.confluent.connect.hdfs.partitioner.HourlyPartitioner
13. name=demo-enriched-clickshdfs-sink
```

Naime, podaci se particioniraju na bazi sata, što znači da se za svaki sat kreira novi direktorij u kojem se nalazi niz Parquet datoteka od kojih svaka ima po tri zapisa. U sljedećem primjeru bit će prikazana hijerarhija direktorija koju je konektor napravio na Hadoop-u.



Slika 30 Hijerarhija direktorija

[13] Apache Parquet – je stupičasti način na koji se spremaju podaci (Apache Software Foundation, 2018)

6.3. Osvrt na krajnje rješenje

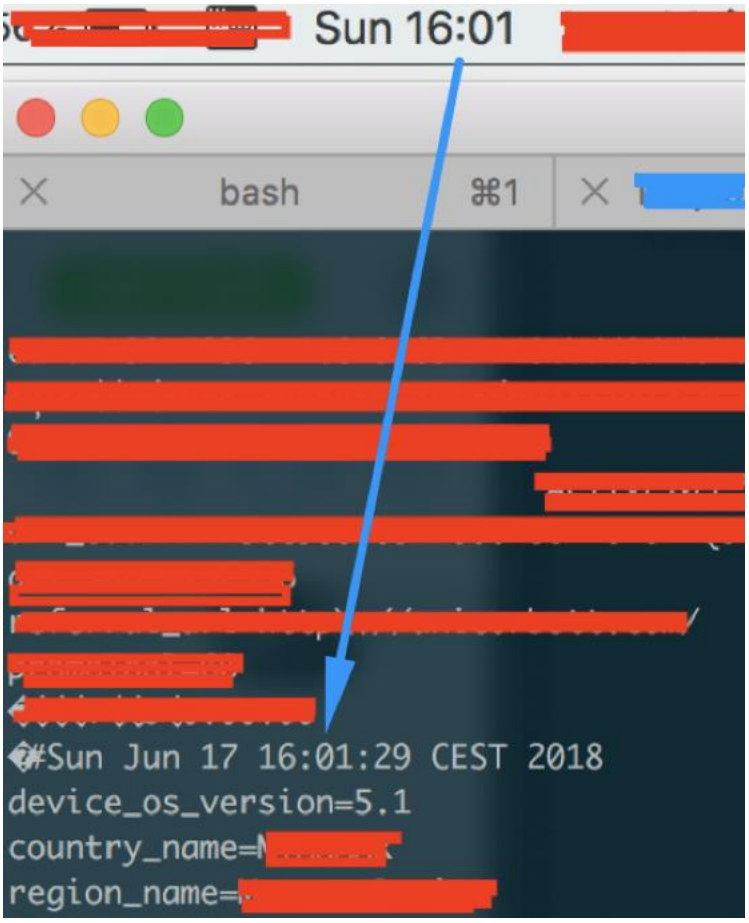
Odlukom korištenja Lambda arhitekture riješeno je podosta izazova s kojima se razni timovi susreću te koji su ranije navedeni. No, Lambda arhitektura ne mora biti i idealno rješenje. Rješenje koje je zamijenjeno Lambda arhitekturom sastojalo se od MySQL – MSSQL Servera te klasičnog ETL-a. Glavni nedostatak bilo je kašnjenje podataka prilikom izvršavanja ETL-a i do par sati. Uvođenjem Lambda arhitekture uveden je novi način izvršavanja klasičnog ETL-a putem mikroservisa. Također uvedene su i nove baze podataka kojima su se riješili i dodatni izazovi na strani baze podataka. Primjer takvog izazova koji nije bio toliko nužan no koji je postojao te bio konstantna prijetnja je problem ispada.

U trenutnoj arhitekturi korištenjem Cassandra-e kao primarne baze podataka riješen je problem ispada. Na aplikacijskoj razini gdje se za primjer mikroservisa uzela Kafka Stream aplikacija također je riješen problem ispada. Naime, mikroservis kao takav moguće je pokrenuti na više servera odjednom prilikom čega će samo jedan mikroservis na jednom serveru zaista raditi dok će ostali biti u stanju pripravnosti. Isto tako, moguće je na strani Kafka-e povećati broj particija na temama u koje dolaze podaci te samim tim omogućiti paralelizam na aplikacijskoj razini gdje će u tom slučaju više mikroservisa moći raditi te ostatak biti u stanju pripravnosti ako se dogodi ispad.

Kako bi se prikazalo vrijeme odaziva na strani Cassandra-e iskoristit će se alat koji dolazi u paketu s Cassandra-om naziva *nodetool*. Alatom je moguće provjeriti stanje klastera, stanje svakog čvora pojedinačno, brzina odaziva upita nad tablicama te slično. Kako bi se provjerilo vrijeme odaziva koristit će se funkcija naziva *tablestats*. Uz pretpostavku da se u tablici nalazi desetak milijuna podataka te da se upiti koriste kako bi se dobili najnoviji podaci rezultati po pitanju brzine odaziva su sljedeći:

```
Total number of tables: 104
-- -- -- --
Keyspace: demo
Table: demo_enriched_clicks
 Read Latency: 0.032 ms
 Write Latency: 0.058817 ms.
```

Iz čega je moguće zaključiti da upiti koji dolaze do Cassandra-e su izvršeni u poprilično malim intervalima. Na sljedećoj slici prikazana je izlazna tema u koju mikroservis upisuje podatke nakon same obrade te koji se prenose u Cassandra-u.



Slika 31 Obrada podataka

Ako se uzme u obzir da podaci u izlaznoj temi dolaze gotovo u stvarnom vremenu te da se iz iste teme podaci prenose direktno u Cassandra-u gdje je latencija zapisivanja 0.0588 milisekundi može se zaključiti kako se kašnjenje uspjelo smanjiti s nekoliko sati na gotovo nekoliko sekundi čime je cilj ispunjen. Dodatni razlog izabira baš Lambda arhitekture je podržavanje navedena tri sloja. U slučaju da je potreban bio samo brzi sloj, Kappa arhitektura[14] bi vjerojatno bila najpogodnija. Naime, potrebno je napomenuti da Lambda arhitektura u ovom projektu nije prikazana u svojem punom svjetlu odnosno da nisu apsolutno sve mogućnosti arhitekture kao takve prikazane. Zbog nedostatka resursa kao i potrebe nije spojen batch sloj s brzim slojem u sloj posluživanja već je brzi sloj poslužio kao sloj posluživanja te batch sloj kao SSOT. Kada bi se krenulo razmišljati o spajanju slojeva, HBase kao baza podataka bi bila najbolji izbor ili eventualna zamjena cijelog batch sloja koji naveden u ovom projekta s Apache Kudu-om[15].

[14] Kappa arhitektura – pojednostavljenje Lambda arhitekture gdje je batch sloj jednostavno izbrisan, odnosno isti je zamijenjen na način da krajnji podaci dolazi samo iz brzog sloja.
[15] Apache Kudu – stupičasti tip baza podataka razvijen za integraciju sa Hadoop platformom.

7. Zaključak

Razvijanje aplikacija oduvijek nije predstavljalo preveliki problem ako postoji jedna baza podataka koja može zadovoljiti sve potrebe pohrane podataka, pristupa te procesiranja. U ovom radu, istražen je jedan arhitekturalni pristup koji omogućava izgradnju visoko skalabilne arhitekture bazirane na stream-ovima u obliku nepromjenjivih događaja. Ovaj pristup korištenja višeslojne arhitekture od kojih je jedan brzi sloj već je opće prihvaćen te se uvelike primjenjuje u svrhu analitike, otkrivanje prijevara ili jednostavno upozoravanje ako dođe do anomalija.

U trenutnom pogledu na svijet, dnevnik događaja (eng. *Event Log*) se gleda kao SSOT te su druge baze podataka derivirane od istog kroz određene stream transformacije kao što su na primjer spajanje, agregiranje te slične. U radu je prikazan još jedan dodatni sloj apstrakcije gdje su podaci bili derivirani od temporalnog SSOT-a (Kafka-e) te u konačnici bili spremani u Hadoop što je ujedno i konačni prikaz SSOT-a.

U radu je prikazano te se prošlo kroz ogromnu količinu informacija no u konačnici ako se napravi suma mogu se izdvojiti neke najbitnije stvari. Važno je napomenuti da servisi koji su spomenuti u radu kao na primjer Kafka veoma skalabilni te ih je isto tako veoma lagano skalirati. Ponajviše zaslužno tome je jednostavna struktura podataka koja omogućuje particioniranje, replikaciju preko više strojeva te visoku propusnost na disku pošto je ulaz/izlaz obično sekvencijalan. Upravo ako su podaci dostupni u sustavu u obliku dnevnika postaje puno lakše za integrirati kao i sinkronizirati podatke različitih servisa. Moguće je također lako izbjeći oporavke od kvarova.

Naravno, prikazana tehnika odnosno stil arhitekture kao i servisi koji su korišteni nisu jedini. Svakim danom počinju na površinu isplivavati novi stilovi, novi servisi te dan danas se pokušava otkriti koja je prava tehnika izgradnje skalabilne arhitekture za procesiranje velikih količina podataka. Na sreću, ovo nije znanstvena fantastika već realnost koja se upravo događa. Ljudi rade na različitim problemima u nadi da pronađu rješenje koje im je prikladno. Količina alata svakim danom se rapidno povećava kao što i rapidno postaje sve bolja što čini ovo vrijeme veoma uzbudljivim kada u pitanju dođe izgradnja arhitekture prikladne za rješavanja problema skalabilnosti te procesiranja podataka.

8. Literatura

[1] What is RPC?, Microsoft, https://technet.microsoft.com/en-us/library/cc787851(v=ws.10).aspx, [19.05.2018]

[2] MapReduce, Nepoznati autor, https://en.wikipedia.org/wiki/MapReduce, [22.05.2018]

[3] Big Data Analytics Infrastructure, Rose Business Technology, http://www.rosebt.com/big-data-analytics-infrastructure.html, [23.05.2018]

[4] Spring Boot, Nepoznati autor, https://projects.spring.io/spring-boot/, [24.06.2018]

[5] Introduction, Nepozanti autor, https://kafka.apache.org/intro, [26.05.2018]

[6] What is actual role of Zookeeper in Kafka, P. Damodaran, https://www.quora.com/What-is-the-actual-role-of-ZooKeeper-in-Kafka, [30.05.2018]

[7] Kafka-topics-ui, Landoop, https://github.com/Landoop/kafka-topics-ui, [27.05.2018]

[8] What is Docker, Docker Inc, https://www.docker.com/what-docker, [31.05.2018]

[9] Schema-registry-ui, Landoop, https://github.com/Landoop/schema-registry-ui, [27.05.2018]

[10] Kafka-connect-ui, Landoop, https://github.com/Landoop/kafka-connect-ui, [27.05.2018]

[11] Planet Cassandra 2015, What is Cassandra, dostupno na: http://www.planetcassandra.org/what-is-apache-cassandra/, [10.05.2018]

[12] Apache Hadoop 2018, What is Apache Hadoop, dostupno na: http://hadoop.apache.org/, [31.05.2018]

[13] Apache Hive TM, Apache Hive, https://hive.apache.org/, [01.06.2018]

[14] Apache Software Foundation 2018, Parquet, dostupno na: https://parquet.apache.org/documentation/latest/, [04.06.2018]

[15] Lambda Architecture using Apache Spark – with Java Code examples, S. Kumar, http://blogs.quovantis.com/lambda-architecture-using-apache-spark-with-java-code-examples/, [10.05.2018]

[16] Mraz S., Warren J. (2015) Principles and best practices of scalable real-time dana systems. Shelter Island: MANNING

[17] A new paradigm for Big Data, A. Shichao, https://notes.shichao.io/bd/ch1/, [20.05.2018]

[18] Batch vs Real Time Data Processing, M. Walker, https://www.datasciencecentral.com/profiles/blogs/batch-vs-real-time-data-processing [05.05.2018]

[19] MIT Technology Review 2016, The Big Data Conundrum: How to Define It?, dostupno na: https://www.technologyreview.com/s/519851/the-big-data-conundrum-how-to-define-it/, [15.06.2018]