SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2151

ANALIZA VELIKE KOLIČINE PODATAKA O STANJU GRADSKOG PROMETA UPOTREBOM PROGRAMSKIH ALATA OTVORENOG KODA

Luka Kiseljak

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2151

ANALIZA VELIKE KOLIČINE PODATAKA O STANJU GRADSKOG PROMETA UPOTREBOM PROGRAMSKIH ALATA OTVORENOG KODA

Luka Kiseljak

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

DIPLOMSKI ZADATAK br. 2151

Pristupnik:	**Luka Kiseljak (0036490957)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	izv. prof. dr. sc. Alan Jović

Zadatak: **Analiza velike količine podataka o stanju gradskog prometa upotrebom programskih alata otvorenog koda**

Opis zadatka:

Velika količina podataka zahtijeva programske alate koji su prilagođenu za brzu i učinkovitu predobradu, analizu i pohranu podataka. U ovom diplomskom radu potrebno je ostvariti implementaciju programskog sustava zasnovanog na tehnologijama otvorenog koda Apache Kafka, Apache Spark i Apache Cassandra s primjenom u analizi stanja gradskog prometa. Podatci o gradskom prometu preuzimat će se u stvarnom vremenu sa slobodno dostupnog web sjedišta "Transport for London." Apache Kafka će se koristiti za obradu tokova poruka koji pristižu s web sjedišta. Apache Spark koristit će se za izgradnju i primjenu modela strojnog učenja za predviđanje lokacije i vremena pojave prometnih poremećaja na temelju prikupljenih podataka. Apache Cassandra služit će za učinkovitu pohranu rezultata obrade podataka. U diplomskom radu potrebno je proučiti i opisati značajke navedenih programskih alata te ponuditi implementaciju povezanih alata kako bi se demonstrirala njihova učinkovitost s podacima prikupljenima iz prometa u Londonu. Također, u radu je potrebno izraditi web aplikaciju koja će u sučelju ponuditi korisnicima bitne i ažurne informacije o prometu na temelju dostupnih predobrađenih podataka u bazi Cassandra. U radu se treba osvrnuti na prednosti i nedostatke pojedinih tehnologija u odnosu na druga, jednostavnija rješenja (npr., samo korištenje troslojnog modela s relacijskom bazom podataka). Cilj rada je demonstracija učinkovitog sustava koji će se moći koristiti za povoljan odabir ruta putovanja na temelju predviđenih prometnih poremećaja u odabranom vremenu putovanja.

Rok za predaju rada: 30. lipnja 2020.

SADRŽAJ

1. Uvod

Svako programsko rješenje temelji se na podacima. Korisnici na društvenim mrežama objavljuju sadržaj koji se prikazuje drugim korisnicima. Računalne igre skupljaju interakcije igrača, izračunavaju i prikazuju efekte tih interakcija. Automatizirani sustavi za trgovanje skupljaju informacije o cijenama na tržištu te na temelju toga kupuju i prodaju financijske instrumente. Bankarski sustavi prate korisnikove transakcije i pokušavaju uočiti nepravilnosti koje bi mogle ukazivati na pokušaj krađe. Svaki taj sustav smišljen je tako da uzima podatke, automatizirano ili izravnom interakcijom korisnika, koje zatim pohranjuje i obrađuje, stvarajući time još podataka. U novije vrijeme, osobito od početka masovne upotrebe interneta, količina podataka koju sustavi prikupljaju počela je naglo rasti. Različita programska rješenja izgrađena za potrebe svog vremena, koja su desetljećima dobro obavljala svoje zadaće, počela su dolaziti do svojih gornjih granica.

Početkom tisućljeća velike tvrtke usmjerene na internet počele su ubrzano širiti svoje poslovanje, a time i količinu podataka koju obrađuju. Njihovi timovi inženjera u suradnji s akademskom zajednicom počeli su razmišljati o sustavima nove generacije koji bi mogli pratiti taj rast. Tako je 2003. godine Google objavio znanstveni članak o modelu obrade podataka *map-reduce*, a godinu kasnije započeo s razvojem svog novog mehanizma pohrane podataka, Bigtable. Gotovo istovremeno, Amazon, došavši do granica Oracleove baze podataka koju su koristili, započinje s razvojem svoje baze podataka, Amazon Dynamo. Još jedan veliki igrač u to vrijeme, Yahoo, uključuje se u utrku i 2006. godine izbacuje Hadoop, sustav za obradu velike količine podataka temeljen na idejama iz Googlea o *map-reduce* programskom modelu. Osim velikih tvrtki, inovacije počinju dolaziti i od manjih *start-up* tvrtki koje se probijaju na tržište, a sve veći interes počinju pokazivati i države koje u prikupljanju i analizi velikih količina podataka uviđaju velik potencijal.

Užurbani razvoj ove nove generacije programskih rješenja uvelike je bio potpomognut dijeljenjem ideja i činjenicom da su o njima imali prilike diskutirati inženjeri i znanstvenici iz praktički cijelog svijeta. Većina tehnologija razvijenih u ovom novo-

nastalom području računarstva, području velikih podataka (eng. *big data*), nije iskorištena samo u okruženju za koje su originalno napravljeni nego su postali alati otvorenog koda sa širokim spektrom primjena. Dobar dio njih doniran je američkoj neprofitnoj organizaciji Apache Software Foundation, koja preko decentralizirane mreže suradnika radi na razvoju i održavanju svojih programskih rješenja. Otvorenost njihovih rješenja omogućuje bilo kome na svijetu razvoj i poboljšanje svojih proizvoda i usluga, ali i osigurava da se na samim alatima aktivno radi i unaprjeđuje njihova upotrebljivost, sigurnost i performanse.

Ovaj rad posvećen je alatima otvorenog koda za obradu i pohranu velike količine podataka. Na primjeru aplikacije za analizu stanja gradskog prometa demonstrirana je arhitektura te odabir i upotreba alata u jednom modernom *big data* sustavu fokusiranom na obradu tokova podataka u (skoro) stvarnom vremenu. U nastavku je najprije opisan sustav i odabrani alati koji najbolje odgovaraju postavljenim zahtjevima, imajući na umu moguće buduće nadogradnje. Zatim je za svaki od njih detaljnije pojašnjen način na koji radi, a na kraju je opisan razvijeni sustav i njegove mogućnosti.

2. Arhitektura sustava

2.1. Sustav za analizu stanja gradskog prometa

Rastom popularnosti i upotrebe rješenja iz područja velikih podataka znatno se povećao i broj dostupnih alata namijenjenih razvoju takvih rješenja. Kao što je to obično slučaj, niti jedan alat nije savršen baš za svaku primjenu. Zato je prilikom odabira alata potrebno dobro proučiti kako se uklapaju u konkretan sustav čijim bi dijelom trebali postati. No, bez dovoljno iskustva i znanja o problemima na koje takvi sustavi nailaze, za neke alate nije jednostavno razumjeti koji točno problem rješavaju i zašto su danas u tako širokoj primjeni. Zato se učinilo logičnim u sklopu ovog rada implementirati neko konkretno rješenje, na čijem bi se primjeru mogla bolje upoznati problematika, kao i značajke i primjenjivost dostupnih alata. Osmišljen je sustav koji bi analizirajući javno dostupne podatke o stanju u prometu mogao korisnicima ponuditi njihov pregled i, zanimljivije, mogao predviđati prometne poremećaje za buduća putovanja.

Pretragom javno dostupnih podataka o stanju u prometu pronađeno je nekoliko mogućih izvora podataka. Najzanimljivijim su se pokazali podaci dostupni kroz programsko sučelje Unified API, koji izlaže Transport for London, javna organizacija zadužena za prijevoz u širem području Londona. Unified API u stvarnom vremenu izlaže podatke koji se skupljaju s raznih izvora, od sustava koji vrši naplatu karata pri ulasku u podzemnu željeznicu do sustava kamera raspoređenih po Londonu. Kako su s ovog sučelja u svakom trenutku dostupni samo trenutni podaci, sustav za analizu mora periodički dohvaćati podatke sa sučelja, stvarajući time tok podataka koji se može analizirati. No, takav pristup znači i da je prije same analize potrebno predobraditi podatke. Na primjer, ako se podaci dohvaćaju svakih nekoliko minuta, izgledno je da će jedna nesreća biti prijavljena više puta, iako se radi o istom događaju, pa je potrebno ukloniti duplikate. Kako bi se ostvarila mogućnost predviđanja prometnih poteškoća, koristeći predobrađene podatke potrebno je učiti odabranu vrstu modela strojnog učenja. Uz to, te je podatke potrebno i učinkovito pohraniti kako bi se korisnicima omogućio njihov pregled, ali i ostavio prostor za dodatnu analizu u nekoj budućoj verziji sustava.

2.2. Alat za razmjenu poruka

Gotovo svi današnji programski sustavi sastavljeni su od niza povezanih komponenti. Gledajući neku uslugu poput društvene mreže, može se uočiti da već i korisnici imaju na izbor više aplikacija preko kojih joj mogu pristupiti. Kako bi korisnicima omogućili dodavanje svojeg i dohvaćanje tuđeg sadržaja, te su aplikacije povezane s drugom komponentom, poslužiteljem. No niti poslužitelj nije jedinstvena cjelina. On je povezan s nekom bazom podataka u koju se spremaju podaci, a ako te podatke treba efikasno pretraživati vjerojatno postoji i zasebna komponenta zadužena za to. Na temelju sadržaja kojeg korisnici pregledavaju nude im se personalizirane preporuke, koje izračunava opet neka zasebna komponenta. U novije vrijeme sve su popularnija rješenja u kojem niti glavni poslužitelj nije jedinstven monolit, nego se upotrebom arhitekture mikrousluga poslužitelj ostvaruje kao niz manjih usluga. Slično kao društvena mreža, osmišljeni sustav je sastavljen od više komponenata koje surađuju u pružanju usluge. Korisnicima se nudi barem jedna aplikacija preko koje koriste sustav. Aplikacija šalje zahtjeve poslužitelju, koji je povezan s bazom podataka u kojoj su pohranjeni prikupljeni podaci. Korisnici mogu zatražiti predviđanje poteškoća na odabranoj ruti u odabranom vremenu putovanja, što može biti zadaća zasebne komponente zadužene za učenja modela strojnog učenja. Podaci iz sučelja Unified API periodički se dohvaćaju i zatim predobrađuju, što se može izvesti kao dvije zasebne komponente.

Karakteristika takvih sustava je da dijelovi sustava ovise jedni o drugima. Svaka komponenta prikuplja ili stvara samo dio podataka potrebnih na razini cijelog sustava, zbog čega gotovo svake dvije komponente jednog sustava moraju moći međusobno komunicirati kako bi razmjenjivali podatke. Uzmemo li u obzir činjenicu da zasebne komponente često razvijaju zasebni timovi, možda i u zasebnim tvrtkama, koji koriste različite operacijske sustave, programske jezike i alate, integracija komponenti sustava nije jednostavan zadatak. Kako raste broj komponenata, broj integracija raste još brže, pa je tako već za sustav sastavljen od 10 komponenata koje sve međusobno komuniciraju potrebno ostvariti 35 integracija. Sustav s tako zamršenom arhitekturom teško je pratiti i održavati, a najmanje ažuriranje jedne komponente koje promijeni format ili ograničenja nad podacima vrlo vjerojatno uzrokuje probleme u velikom broju drugih komponenata.

Kompleksnost višestrukih integracija komponenata u sustavu i posljedična krhkost sustava poznat je problem, a pristup kojim ga se najčešće nastoji riješiti je uvođenjem sustava za razmjenu poruka. Upotrebom tog pristupa, svaka se komponenta integrira samo s centralnim sustavom za razmjenu poruka na koji šalje svoje i prima podatke

sa svih drugih komponenata. Ako je odabrani sustav za razmjenu poruka izveden s redovima poruka, ostvaruje se mogućnost da svaka komponenta sama inicira čitanje i obradu podataka. Takav način rada u skladu je s principima arhitekture u kojoj poslove, ovisno o svojem kapacitetu i trenutnoj zaposlenosti, iniciraju radnici koji te poslove i obavljaju (eng. *pull architecture*). Dodatno, ako odabrani sustav može čuvati poruke i neko vrijeme nakon čitanja, rješava se problem s komponentama koje naiđu na probleme i dio poruka moraju ponovo obraditi.

Premda se upotrebom sustava za razmjenu poruka postiže odvajanje proizvođača i potrošača poruka, oni su i dalje veoma međuovisni. Svaka komponenta mora znati kojim svim drugim komponentama i u kojem formatu mora slati poruke, iako to nije sastavni dio njene poslovne logike. Kako bi se doskočilo tom problemu, sustav za razmjenu poruka može se izvesti kao sustav s objavama i pretplatama (eng. *publish-subscribe messaging system*). U takvom sustavu, pošiljatelj poruku ne mora usmjeriti nekom konkretnom primatelju, nego svaku poruku označava informacijom na što se ona odnosi, odnosno kojoj temi pripada. S druge strane, sve komponente zainteresirane za takve poruke pretplaćuju se na odgovarajuće teme s kojih čitaju pristigle poruke. Time komponente postaju manje međuzavisne i postaje moguće dodavati ili uklanjati komponente iz sustava bez promjena na ostalim komponentama.

Opisanu problematiku i rješenje s uvođenjem sustava za razmjenu poruka temeljenog na objavama i pretplatama razmatrali su i inženjeri iz LinkedIna. Iako je u to vrijeme postojalo nekoliko perspektivnih sustava za razmjenu poruka, niti jedan nije u potpunosti zadovoljavao njihove potrebe pa su odlučili napraviti svoj - Kafku Kreps et al. (2011). Nekoliko godina kasnije projekt su objavili i učinili alatom otvorenog koda, a razvoj i održavanje preuzeo je Apache Software Foundation. Danas je Kafka centralna točka brojnih sustava, a odabrana je i za sustav razvijen u sklopu ovog rada. Njenom upotrebom postignuto je značajno pojednostavljenje arhitekture, ali je i otvoreno dosta prostora za skaliranje komponenti, kao i za bezbolno dodavanje i uklanjanje komponenata. Uz to, kako je razvoj Kafke u posljednje vrijeme snažno fokusiran na obradu tokova podataka preko biblioteke Kafka Streams, Kafka se pokazuje korisnom i za druge zadaće osim same razmjene poruka. U ovom sustavu, biblioteka Kafka Streams je iskorištena kako bi se izravno na Kafki obavila predobrada podataka.

2.3. Baza podataka

Osmišljeni sustav za analizu gradskog prometa treba pohranjivati prikupljene podatke radi evidencije i eventualne kasnije obrade, za što bi se mogla upotrijebiti neka re-

lacijska baza podataka. Njihovom upotrebom jednostavno je modelirati složene domene problema, a promjene nad modelom, kao i unos i ažuriranje podataka ostvaruje se upotrebom relativno jednostavnog i standardiziranog jezika SQL. Uz to, relacijske baze podataka pružaju transakcije, koje garantiraju poželjna svojstva ACID: atomarnost (eng. *atomicity*), konzistentnost (eng. *consistency*), izolaciju (eng. *isolation*) i izdržljivost (eng. *durability*). Također, postoji širok izbor relacijskih baza podataka, od kojih su dobar dio alati otvorenog koda. Zbog svega navedenog, relacijske baze podataka su od svojeg predstavljanja 70-ih godina prošlog stoljeća do danas dominantan način pohrane aplikacijskih podataka.

Premda relacijske baze podataka već dugi niz godina igraju važnu ulogu u raznovrsnim programskim rješenjima, u posljednjih desetak godina sve se više počinju razvijati i upotrebljavati druge vrste baza podataka. Povećanjem količine podataka u relacijskoj bazi podataka, sve više postaje problem činjenica da je potrebno stvarati tablice za entitete koji nisu dio domene, kao što je slučaj s veznim tablicama u vezama više-na-više. Zbog toga upiti zahtijevaju višestruke operacije združivanja (eng. *join*) koje uzimaju dosta vremena. Iako su svojstva koja nude transakcije odlična, povećanjem opterećenja baze podataka, ona postaju sve zahtjevnije za održavati jer rezultiraju dugim redovima čekanja. Problem se nastoji riješiti optimiziranjem upita, uvođenjem novih indeksa, upotrebom brze priručne memorije (eng. *cache*) ili upotrebom boljeg sklopovlja. No sva su ta rješenja privremena, jer kako raste količina pohranjenih podataka i opterećenje na bazu tako se ponovno počinju pojavljivati performansni problemi. Kako bi se povećao kapacitet i rasporedilo opterećenje, počinju se koristiti raspodijeljene baze podataka, ali time se otvara niz novih problema, kao što su očuvanje dostupnosti i konzistentnosti prilikom ispada pojedinih čvorova, osiguranje izdržljivosti podataka i očuvanje konzistentnosti među replikama. Relacijske baze podataka i obećanja koja pružaju veoma su privlačna, ali u području velikih količina podataka nisu sva ostvariva ako se žele održati dobre performanse Hewitt i Carpenter (2020).

Koja su obećanja relacijske baze podataka bitna za sačuvati, a koja je moguće oslabiti ili maknuti, pitanje je iz kojeg danas proizlazi niz različitih vrsta baza podataka. Ne postoji savršeno rješenje, jer nije moguće u isto vrijeme dobiti sva dobra svojstva baza podataka, nego je ovisno o konkretnom sustavu potrebno odabrati odgovarajuće kompromise. Ovo je opisano i teoremom CAP kojeg je formulirao Eric Brewer pod kraj prošlog stoljeća. Teorem se odnosi na tri poželjna svojstva raspodijeljenih baza podataka, a to su konzistentnost (eng. *consistency*), dostupnost (eng. *availability*) i tolerancija na particioniranje (eng. *partition tolerance*). Konzistentnost označava svojstvo da svako čitanje dohvaća zadnje zapisanu vrijednost, dostupnost da na svaki zahtjev

dolazi uspješan odgovor, ali ne nužno sa zadnje zapisanom vrijednosti, a tolerancija na particioniranje je svojstvo da baza nastavlja s radom bez obzira na proizvoljan broj izgubljenih ili zakašnjelih poruka među čvorovima. Prema ovom teoremu, niti jedna raspodijeljena baza podataka ne može (u potpunosti) garantirati sva tri svojstva Gilbert i Lynch (2012).

Raspodijeljene relacijske baze podataka se tipično oslanjaju na protokol 2PC (eng. *two phase commit*) za očuvanje svih obećanih svojstava, garantirajući time konzistentnost i dostupnost, ali ne i toleranciju na particioniranje. Ako do particioniranja dođe, primjerice ako neki od čvorova izgubi pristup mreži, cijela baza podataka postaje nedostupna. Ovo nije dobro rješenje, jer su čvorovi raspodijeljenog sustava povezani mrežom koja je asinkrona, u kojoj se događaju latencije i u kojoj se ponekad zagube paketi, pa je pojava particioniranja neizbježna. Zato se raspodijeljene baze podataka danas rade tako da toleriraju particioniranje, čime iz CAP teorema na odabir ostaje garantirati konzistentnost ili dostupnost. Primjeri raspodijeljenih baza podataka koje garantiraju toleranciju na particioniranje i konzistentnost su Neo4j, Google Bigtable i njegovi nasljednici MongoDB, HBase, Redis i drugi. Čvorovi u tim bazama podataka ne repliciraju svoje podatke na druge čvorove pa kako podaci postoje samo na jednom mjestu klijenti uvijek čitaju najnoviju vrijednost. No, time se žrtvuje dostupnost jer ispadom jednog čvora dio podataka nije moguće čitati ili pisati. S druge strane, Amazon Dynamo i baze podataka koje su temeljene na njoj poput Apache Cassandra, Voldemort i CouchDB, repliciraju podatke s čvorova čime uz toleranciju particioniranja pružaju i garanciju dostupnosti. Konzistentnost nije garantirana jer je moguće da nakon uspješnog pisanja u bazu podataka klijent s replike na kojoj promjena nije još zapisana pročita staru vrijednost.

Ako baza podataka ne garantira svojstvo CAP teorema, to ne znači da ga ne ispunjava barem djelomično. Na primjer, baze podataka koje ne garantiraju konzistentnost su većinu vremena u konzistentnom stanju, ali ne uvijek. Tek u trenutku kad se operacija pisanja propagira svim replikama, baza podataka je u potpuno konzistentnom stanju i više niti jedan klijent ne može pročitati neku staru vrijednost. Termin koji se upotrebljava za opis takvog modela konzistentnosti je konzistentnost koja će u konačnici biti postignuta (eng. *eventual consistency*). Uz to, često konfiguracijski parametri baza podataka mogu mijenjati koja svojstva i u kojoj mjeri će baza podataka ispunjavati, mijenjajući tako i svojstva CAP teorema koja baza podataka ispunjava. Na primjer, Apache Cassandra, koja prema ranije navedenoj podjeli ne garantira konzistentnost, zahtijeva od klijenata da prilikom svakog čitanja ili pisanja navedu koju razinu konzistentnosti je potrebno zadovoljiti, pružajući time svojstvo prilagodljive konzistentnosti

(eng. *tunable consistency*). Klijenti prilikom pisanja u bazu podataka navode koliko čvorova mora potvrdi pisanje prije nego se operacija smatra uspješno završenom, a prilikom čitanja navode koji broj čvorova se mora složiti s vrijednošću podatka prije nego se ona vrati klijentu. Ako je zbroj ova dva broja veći od broja replika, Apache Cassandra postiže strogu konzistentnost, odnosno nije moguće da neki klijent pročita staru vrijednost nakon što je potvrđeno pisanje nove vrijednosti. Time Cassandra garantira toleranciju na particioniranje i konzistentnost, ali ne više i dostupnost zbog velikog broja čvorova koji je potrebno kontaktirati prilikom čitanja i pisanja.

Upravo ta fleksibilnost koju Apache Cassandra pruža jedan je od razloga njene velike popularnosti u svijetu velikih količina podataka. Uz to, Apache Cassandra je od samog početka osmišljena za rad na velikom broju računala u više podatkovnih centara, čiji se broj bez ponovnog pokretanja lako može skalirati. Upotreba više računala u više podatkovnih centara osigurava da ako dođe do problema na nekom čvoru ili pak cijeli podatkovni centar doživi katastrofu, baza podataka ostaje dostupna i čuva podatke koji su u njoj pohranjeni. Dodatno, sličnost s konceptima iz relacijskih baza podataka, kao i dobar izbor materijala i gotovih programskih rješenja, čine Apache Cassandru odličnim odabirom za bazu podataka za sustav za analizu gradskog prometa.

2.4. Alat za obradu podataka

Upotrebom Kafke ostvarena je učinkovita razmjena i predobrada, a upotrebom Cassandre i učinkovito spremanje prikupljenih podataka. No, svrha sustava nije samo omogućiti pregled prikupljenih podataka o prometnim poremećajima, nego i na temelju njih predviđati buduće poremećaje. Cilj je korisniku omogućiti da nakon odabira rute i vremena svog putovanja sustav upotrebom umjetne inteligencije iznese svoje predviđanje o mogućim poremećajima, poželjno i uz ocjenu pouzdanosti u dano predviđanje. Zato je, uz postojeće, u sustav potrebno dodati komponentu koja će imati pristup prikupljenim podacima i na temelju njih odgovarati na upite korisnika. Zbog količine podataka i računalne zahtjevnosti predviđanja, nije opcija cijeli izračun raditi za svakog korisnika, nego je ideja glavninu posla obaviti i prije nego što prvi korisnik pošalje svoj upit. Konkretno, potrebno je odabrati neki model strojnog učenja i najprije obaviti najzahtjevniji dio posla, učenje tog modela na temelju prikupljenih podataka, nakon čega se svaki upit svodi na relativno jednostavan izračun predviđanja. No, ovaj dio posla nije moguće napraviti samo jednom, nego je zbog činjenice da novi podaci stalno pristižu model konstantno potrebno ažurirati. Kako bi se izbjegla potreba za obradom svih podataka svaki put kad novi podatak stigne, za algoritam strojnog učenja

potrebno je odabrati neki koji omogućavaju učenje kako podaci pristižu (eng. *online learning*).

Inkrementalnu obrada podataka, kao što je ugađanje parametara modela strojnog učenja, moguće je izvesti na dva načina. Prvi je način da se novi podaci neko vrijeme skupljanju u grupe (eng. *batch*), koje se zatim periodički i obrađuju. Ovakav način obrade podataka popularizirao se još 70-ih godina prošlog stoljeća u skladištenju podataka (eng. *data warehousing*) upotrebom procedura ETL. Procedura se sastojala od tri koraka po kojima je dobila i ime: izdvajanja (eng. *extract*) podataka, često iz niza drugih komponenata, transformacije (eng. *transform*) i učitavanja podataka na odredište (eng. *load*). Obično je to višesatni proces koji se ne izvodi pretjerano često, primjerice jednom dnevno, u noćnim satima, kad je smanjeno opterećenje sustava. Upotrebom ovog pristupa na sustav za analizu gradskog prometa, proces ETL bi dohvatio nove podatke iz Cassandre, izveo dodatno učenje modela i zatim pohranio nove rezultate nazad u Cassandru. Danas kad su sklopovlje i programska podrška značajno uznapredovali, popularnije je ipak upotrijebiti drugi pristup – izravnu obradu tokova podataka Majeed et al. (2010). U sustavu za analizu gradskog prometa, pristup bi se ostvario čitanjem toka podataka s Kafke i ugađanjem parametara modela kako podaci pristižu, odnosno skoro u stvarnom vremenu. Kako se u ovoj komponenti radi o učenju algoritma strojnog učenja na temelju velikog broja prikupljenih podataka, nije za očekivati dramatične promjene u predviđanjima pristizanjem novih podataka. Iz tog razloga nije apsolutno nužno obrađivati podatke u stvarnom vremenu, ali postoji niz primjena u kojima je čekanje rezultata obrade do idućeg izvođenja ETL procedure neprihvatljivo. Na primjer, sustavima za automatizirano trgovanje vrijednosnim papirima, koji moraju brzo reagirati na promjene na tržištu i iskoristiti prilike, ključno je da uvijek raspolažu s najnovijim informacijama. Isto tako, sustavi za detekciju prevara s bankovnim karticama moraju što prije uočiti anomalije u potrošnji kako bi spriječili veću štetu. Alati koji se danas koriste za obradu velikih količina podataka uglavnom podržavaju oba pristupa. To znači da se i nakon ulaganja u razvoj i infrastrukturu prilagođenu jednom alatu može prebaciti i na drugi način obrade, ako se za tim pokaže potrebna. Dodatno, alati često omogućavaju da isti programski kod izvodi i jedan i drugi način obrade, čime je trošak prelaska s jednog na drugi pristup minimalan. Također, ovo otvara mogućnost da se tijekom razvoja, grupnim izvođenjem na manjem skupu podataka, isprobava kod i različiti modeli s različitim hiperparametrima, a jednom kad je sve spremno razvijeni kod se može upotrijebiti za obradu toka u produkcijskom okruženju.

Kako bi se unatoč količini podataka zadržale željene performanse komponente za učenje modela, omogućilo horizontalno skaliranje kako se poveća ili smanji opterećenje cijelog sustava i osigurala izdržljivosti podataka stvaranjem sigurnosnih kopija podataka u više čvorova logičan je odabir raspodijeljenog alata. Ovo donosi i dodatnu razinu kompleksnosti pa je povoljno odabrati neki gotovi raspodijeljeni alat za obradu velike količine podataka. Danas postoji dobar broj dostupnih alata, a dva najkorištenija su proizvodi Apache Software Foundationa, Apache Hadoop i Apache Spark. Apache Hadoop, inspiriran rješenjima koje je predstavio Google, implementirao je niz inovativnih rješenja u domeni raspodijeljene obrade velike količine podataka na računalima masovne upotrebe poput HDFS-a, raspodijeljenog datotetečnog sustava, i koordinatora resursa YARN. Hadoop se temelji na programskom modelu *map-reduce*, u kojem je obrada podijeljena na dva koraka, preslikavanje (eng. *map*) koje uključuje učitavanje i pripremu podataka, te smanjivanje (eng. *reduce*) u kojem se rade izračuni na podacima koji se zatim spremaju na disk. Upravo upotreba ovog programskog modela, odnosno njegova ograničenja u nekim primjenama, potaknuli su razvoj novog alata, Apache Sparka. Tako je Spark u obradi podataka koja se sastoji od velikog broj prolaza po podacima, kao što su iterativni optimizacijski postupci kod algoritama strojnog učenja, nekoliko redova veličine brži od Hadoopa Zaharia et al. (2010). Najveće ubrzanje postiže se manipuliranjem podacima izravno u radnoj memoriji, umjesto da svaki prolazak, odnosno svaka iteracija algoritma *map-reduce*, čita i piše podatke na znatno sporiji disk.

Odabir između Apache Hadoopa i Apache Sparka uglavnom se svodi na razmatranje konkretne primjene za koju će alat biti upotrebljen. Iako je Spark brži, budući da njegovi čvorovi zahtijevaju više radne memorije ujedno su i skuplji, no veća brzina znači da je potrebno manje čvorova za obradu iste količine podataka pa može donijeti čak i uštedu. Zato je za opravdanu odluku potrebno razmotriti koliko će podataka sustav obrađivati, kolika su kašnjenja u obradi dozvoljene i druge parametre koji utječu na odluku. Ono što je važno za sustav za analizu stanja gradskog prometa je mogućnost obrade tokova podataka, i to konkretno obrada u vidu učenja algoritma strojnog učenja koji obrađuje podatke kako oni pristižu. I Hadoop i Spark pružaju ove mogućnosti, no u ovom sustavu odabran je noviji alat, Spark, koji je i zbog svojih performansi generalno bolje prilagođen obradi tokova podataka.

3. Apache Kafka

Apache Kafka je alat otvorenog koda namijenjen obradi tokova podataka. U osnovi, Kafka je sustav za razmjenu poruku koji radi na principu objava i pretplata. U sustavu sastavljenom od većeg broja komponenti, Kafka kao središnji čvor omogućava komponentama da objavljuju poruke u teme te da se pretplaćuju na njima zanimljive teme i s njih čitaju poruke. Ovi proizvođači i potrošači poruka mogu biti vlastite komponente, za čiji razvoj postoji dobar izbor gotovih klijentskih biblioteka, ili pak gotovi vanjski sustavi poput baza podataka, spremnika priručne memorije, datotečnih sustava ili alata za pretraživanje. Vanjski sustavi se s Kafkom povezuju preko posrednika u čijem razvoju se koristi Kafkina komponenta Connect. No rijetko postoji potreba za implementacijom vlastitih posrednika jer su dostupna gotova rješenja za većinu vanjskih alata danas u upotrebi. Poruke koje prolaze kroz teme u Kafki čine tokove podataka koje je često potrebno na neki način obrađivati. Iz tog razloga, razvijena je Kafkina komponenta Streams koja pruža jednostavan razvoj rješenja za obradu tokova poruka iz Kafke.

Kafka je namijenjena za upotrebu u sustavima koji zahtijevaju obradu veće količine podataka uz niska kašnjenja, a pri tome inzistiraju na visokoj dostupnosti i izdržljivosti Shapira et al. (2017). Kao središnji čvor, Kafka predstavlja posebno osjetljiv dio sustava jer je ona točka čijim ispadom prestaje rad cijelog sustava (eng. *single point of failure*). Kako bi se ovaj problem zaobišao, Kafka je od samog početka dizajnirana kao raspodijeljeni sustav sastavljen od niza zasebnih čvorova, *brokera*, u jednom podatkovnom centru. Dostupni su i alati koji omogućavaju povezivanje više Kafka instanci u drugim podatkovnim centrima, čime je sustav otporan ne samo na ispade čvorova, nego i cijelih podatkovnih centara. Svojstvo izdržljivosti podataka jednog brokera očuvano je stvaranjem sigurnosnih kopija u drugim brokerima. Uz to, raspodijeljena priroda sustava otvara mogućnost jednostavnog horizontalnog skaliranja dodavanjem i uklanjanjem čvorova, čime se lako osiguravaju performanse sustava na razini zahtijevanih.

Osnovna jedinica podataka koja prolazi kroz Kafku je *poruka*, koja je reprezentirana kao niz bajtova. Kafka ne mora znati što se i u kojem formatu prenosi u porukama,

no kako bi se podacima sigurnije manipuliralo poželjno je uvesti nekakvu strukturu, primjerice XML ili JSON. Budući da se formati podataka ažuriranjima mijenjaju, postoji opasnost da uvođenje promjena na jednom klijentu uzrokuje probleme na drugim klijentima. Zato je pogodno koristiti sustave za serijalizaciju podataka koji podržavaju verzioniranje i koji mogu klijentima koji očekuju stari format dostaviti poruku u starom formatu, premda je ona poslana u novom. Jedan takav sustav za serijalizaciju podataka koji se često koristi u kombinaciji s Kafkom je Avro, također projekt od Apache Software Foundationa.

Svaka poruka pripada nekoj *temi*. Poruke jedne teme ne nalaze se sve na jednom mjestu, nego su podijeljene u *particije*. Prilikom stvaranja nove teme specificira se broj particija i one se ravnomjerno podijele brokerima. Kao što je spomenuto, Kafka osigurava visoku dostupnost i izdržljivost podataka repliciranjem podataka na više brokera. No, za svaku particiju svake teme samo je jedan broker vodeća replika (eng. *leader replica*), a sve druge kopije su replike sljedbenici (eng. *follower replica*). Poruke je dozvoljeno čitati i pisati samo s vodeće replike. Ako vodeća replika postane nedostupna, neka druga replika sljedbenik bit će odabrana kao nova vodeća replika. Isto tako, kada se pokrene novi broker izvest će se reparticioniranje kako bi se dio particija s postojećih brokera dodijelio novom brokeru. Replike sljedbenici redovito prepisuju nove poruke koje se zapisuju u vodeću repliku, uz neku toleriranu razinu kašnjenja. Ako neka replika postane nedostupna ili previše kasni, smatra se da više nije sinkronizirana replika (eng. *in-sync replica*). Prilikom slanja poruka, klijenti specificiraju broj sinkroniziranih replika na koje se podatak mora zapisati kako bi se pisanje smatralo uspješnim, a dostupne su opcije 0, 1 ili sve sinkronizirane replike.

Prije nego klijent može poslati poruku u temu, mora odrediti particiju kojoj poruka pripada. To radi tako da izračuna sažetak poruke i pronađe particiju koja je zadužena za interval sažetaka u koji upada izračunati sažetak. Kafka omogućava upotrebu vlastite implementacije izračuna sažetka i odabira particija, no to se rijetko koristi jer standardna implementacija osigurava ravnomjernu raspodjelu poruka i upotrebljava konzistentan izračun sažetaka (eng. *consistent hashing*). Konzistentan izračun sažetaka znači da će se promjenom broja čvorova morati premještati samo proporcionalan udio ukupnih podataka. Upotreba standardne implementacije do neke mjere omogućava kontrolu nad porukama i particijama. Ako se uz poruku pošalje opcionalni podatak, *ključ*, sažetak poruka izračunava se na temelju ključa, a ne sadržaja poruke. Time je moguće osigurati da dvije poruke završe u istoj particiji ako se uz njih pošalje isti ključ. Ovo je osobito korisno, jer omogućava da sve poruke koje će se kasnije obrađivati zajedno budu spremljene na istom mjestu. U svrhu povećanja učinkovitosti, klijenti ne

šalju poruke jednu po jednu, nego ih šalju u grupama (eng. *batch*) koje sadrže poruke koje pripadaju istoj particiji iste teme. Također, poruke se sažimaju prije slanja kako bi se osigurao učinkovitiji prijenos i kasnije pohrana poruka, na teret veće potrošnje procesorske snage.

Poruke je u particije moguće dodavati samo na kraj, čime se čuva redoslijed poruka unutar particije (ali ne i redoslijed poruka unutar teme). Svakoj poruci koja se doda u particiju dodjeljuje se *odmak* – redni broj poruke u particiji koji koriste potrošači poruka kako bi znali gdje su stali s čitanjem. Za svakog potrošača Kafka u odvojenoj temi prati koji je odmak zadnje pročitane poruke kako bi nakon ponovnog pokretanja potrošač mogao nastaviti s mjesta gdje je stao. Potrošači mogu i sami navesti odmak s kojeg će krenuti slijedno čitati poruke, što se koristi kad dožive neki problem i moraju ponovno obraditi stare poruke.

Poruke u particijama se pohranjuju na disk u datotekama koje se nazivaju segmenti. Za svaku particiju, samo je jedan trenutno aktivan segment u koji se zapisuju nove poruke koje pristižu. Kad se dosegne konfigurirana veličina ili vrijeme isteka segmenta, segment se zatvara i počinje se zapisivati u novi. Na razini teme je moguće konfigurirati maksimalnu veličinu ili maksimalno vrijeme koje će se zatvoreni segmenti sa starim porukama čuvati. Kafka nema performansnih problema s velikim brojem ili veličinom segmenata, tako da se ova vrijednost uglavnom odabire ovisno o dostupnom prostoru za pohranu. Tijekom rada Kafka u svakom trenutku održava sve segmente otvorenima i održava indeks koji omogućava brzo pronalaženje odgovarajućeg segmenta za dani odmak. Kad su pronađene tražene poruke u segmentu, podaci se šalju bez kopiranja (eng. *zero-copy writing*), odnosno prenose se izravno iz datoteka na mrežu, bez međukoraka kopiranja podataka u memoriju. Podaci se između brokera i klijenata prenose upotrebom vlastitog binarnog protokola izgrađenog nad protokolom TCP.

Svaki Kafka klijent koji je pretplaćen na neku temu dio je grupe potrošača (eng. *consumer group*), koja može biti sastavljena samo od njega ili od više potrošača koji svaki čita otprilike jednak dio poruka iz teme. Ravnomjerna raspodjela poruka na potrošače jedne grupe osigurava se tako da se svakom potrošaču dodjeljuje ravnomjerni dio particija teme. Potrošači čitaju poruke samo s particija koje su im dodijeljene, i u kontekstu te grupe nazivaju se vlasnicima te particije. Ako grupa sadrži više potrošača nego ima particija u temi, neki potrošači neće dobiti niti jednu particiju i neće pročitati niti jednu poruku. Zato je prilikom stvaranja teme važno obratiti pozornost da se odabere dovoljno velik broj particija. Mehanizam grupe potrošača bitan je jer omogućava jednostavno horizontalno skaliranje. Ako naraste opterećenje, moguće je dodati

nove potrošače u grupu, a ako opterećenje padne ukloniti. Promjena broja dostupnih potrošača u grupi, bilo namjernim dodavanjem i uklanjanjem, bilo ispadom čvorova, uzrokovat će automatsku preraspodjelu particija koja će osigurati da su sve particije dodijeljene nekom potrošaču, i to otprilike isti broj svakome. Ovo je zaduženje jednog potrošača koji je proglašen koordinatorom grupe i kojem se potrošači moraju periodički javljati kako bi on znao da su i dalje dostupni.

Posljedica preraspodijele particija među potrošačima jedne grupe je činjenica da će neki potrošači izgubiti vlasništvo nad particijama. Ako se dogodi da je prošli vlasnik neke podatke obradio, a nije tražio nove i time uzrokovao uvećanje odmaka kojeg je Kafka zapamtila, novi vlasnik particije ponovno će obraditi iste podatke. Upotrebom Kafke i urednim ponašanjem potrošača (ne traženje novih podataka prije nego što su stari obrađeni) moguće je garantirati da neće biti neobrađenih poruka, ali ne i da će poruke biti obrađene točno jednom. Ako postoji ovakav zahtjev, potrošači će morati zapisivati rezultate obrade i zadnji obrađeni odmak istovremeno u neki vanjski sustav, primjerice u bazu podataka upotrebom transakcija.

Informacije o temama, particijama i replikama, popis aktivnih brokera i trenutno odabranom kontrolirajućem brokeru spremaju se u Zoookeeperu. Svi brokeri dužni su periodički slati poruke instanci Zookeepera kako bi on znao da su oni i dalje aktivni (eng. *heartbeat messages*). Ako neki broker prestane slati ove poruke, Zookeeper će svim brokerima dojaviti da je jedan broker postao nedostupan. Ta informacija doći će i do kontrolirajućeg brokera, koji će donijeti odluku o novom vodećem brokeru i tu informaciju zapisati u Zookeeper. Ako je broker koji je postao nedostupan bio kontrolirajući broker, primitkom poruke o njegovom prestanku rada ostali brokeri će poslati zahtjev da oni postanu kontrolirajući brokeri. Zookeeper će ovo dopustiti samo prvom brokeru čiju je poruku zaprimio, a ostalima će javiti da je kontrolirajući broker već odabran. Isti se postupak odvija i prilikom pokretanja instance Kafke kada treba odabrati prvog kontrolirajućeg brokera. Klijenti ne komuniciraju izravno sa Zookeeperom. Umjesto toga, ažurne informacije iz Zookeepera čuvaju se na svakom brokeru i klijenti ih dohvaćaju s brokera.

Kao što je ranije spomenuto, Kafka omogućava obradu tokova podataka upotrebom komponente Streams. Ona izlaže programsko sučelje niske razine Processor API i programskog sučelje visoke razine DSL API. Potonji omogućava definiranje i kombiniranje jednostavnih operacija kao što su preslikavanje, filtriranje, grupiranje i agregiranje, izravno kroz kôd kao niz lambda operacija. Osim toga, moguće je i spajanje sa statičnim podacima poput podataka iz baze podataka, ali i spajanje s dinamičnim podacima poput drugih tokova podataka.

Operacije koje se definiraju nad tokovima podataka prikazuju se *topologijom*, acikličkim usmjerenim grafom u kojem su bridovi prijelazi, a čvorovi procesori. Većina procesora su implementacije baznih operacija koje izlaže programsko sučelje, ali svaka topologija sadrži i barem jedan izvorni čvor koji dohvaća elemente toka te barem jedan odvodni čvor koji zapisuje obrađene elemente toka. Definirana topologija se prije izvođenja dijeli na niz *zadataka*, osnovnih jedinica paralelizacije koje se izvode neovisno na zasebnim nitima. Svaki se zadatak sastoji od izvornog čvora koji čita s jedne particija teme, niza čvorova koji izvode operacije te odvodnog čvora, koji zapisuje rezultate u temu s rezultatima. Ponekad su operacije u topologiji takve da zadaci moraju čitati s više od jedne particije pa se takvim zadacima, kako bi se izbjegla međuovisnost zadataka, dodjeljuje više od jedne particije. Drugi način na koji nastaju ovisnosti je situacija kad je podatke grupirane prema jednom ključu potrebno grupirati prema drugom ključu. U toj situaciji, podatke iz svih zadataka je potrebno skupiti i ponovno podijeliti prema drugom ključu (eng. *shuffle*), što iziskuje koordinaciju svih zadataka koji obrađuju dijelove ulaznih podataka. Ovisnost zadataka se izbjegava tako da se topologija podijeli na dvije manje podtopologije. Jedna podtopologija čita s originalne teme i izvodi grupiranje tako da poruke zapisuje u novu temu u kojoj su podaci particionirani prema novom ključu. Zatim iz te teme zatim druga podtopologija čita ispravno grupirane podatke i izvršava daljnje potrebne operacije.

Kafka Streams sam brine o dostupnim računalima i nitima koje izvode zadatke. Ako neki čvor postane nedostupan, zadaci koji su se izvodili na nitima tog čvora raspoređuju se ravnomjerno ostalim dostupnim čvorovima. U slučaju da se pridruži novi čvor, zadaci se također preraspodjeljuju tako da svaki čvor završi s podjednakim brojem zadataka. Kako Kafka drži odmak koji je pročitan u svakoj particiji, zadatak koji je prebačen na neki novi čvor može jednostavno nastaviti tamo gdje je obrada na prethodnom čvoru stala. U topologijama koje izvode operacije u kojima se mora čuvati stanje, primjerice kod agregiranja podataka u nekom vremenskom intervalu, zadatak na novom čvoru vratit će se nazad do poruke kojom započinje neobrađeni vremenski interval i obraditi cijeli interval od početka do kraja.

4. Apache Cassandra

Apache Cassandra je nerelacijska baza podataka namijenjena pohrani velikih količina podataka. Kombiniranjem raspodijeljenog načina rada iz baze podataka Amazon Dynamo i podatkovnog modela iz baze podataka Google Bigtable, Cassandra osigurava visoku dostupnost, izdržljivost podataka te impresivne performanse Hewitt i Carpenter (2020). Cassandra je decentralizirani raspodijeljeni sustav sastavljen od niza čvorova u više podatkovnih centara. Zbog decentraliziranosti, ali i niza drugih odluka, pruža svojstvo razmjernog rasta (eng. *linear scalability*), odnosno omogućava da dodavanjem novih čvorova sustav može pohranjivati i obrađivati proporcionalno veću količinu podataka. Zbog toga, za razliku od relacijskih baza podataka čije su granice dostignute, Cassandra u teoriji nudi mogućnost beskonačnog rasta dodavanjem novih čvorova. Tako trenutno najveću instancu Cassandre ima Apple, koja s više od 75 tisuća čvorova pohranjuje preko 10 petabajta podataka.

Svi podaci koje sadrži jedna instanca Cassandre dio su grozda (eng. *cluster*), odnosno prstena (eng. *ring*). Prsten je podijeljen na više *prostora ključeva* koji imaju svoje ime i za koje se definiraju konfiguracijski parametri poput strategije replikacije. Prostor ključeva otprilike odgovara jednoj bazi podataka u relacijskom modelu. Prostori ključeva sadrže *tablice*, sortirane kolekcije *redova* koje su podijeljene na particije. Na razini tablice definiraju se *stupci* koje redovi tablice sadržavaju, ali se ne zahtijeva da svaki red ima svaki stupac, za razliku od relacijskih baza podataka gdje svaki stupac ima vrijednost, iako ona može biti posebna vrijednost koja označava nedostajuću ili nepoznatu vrijednost. Na razini tablice također se definira koji podskup stupaca čini primarni ključ. Stupci primarnog ključa dijele se na ključ za particioniranje, ključ za grupiranje te statične stupce. Stupci ključa za particioniranje služe određivanju particije kojoj red pripada. Unutar jedne particije, redovi su sortirani prema vrijednostima stupaca koji čine ključ za grupiranje. Statični stupci su stupci koji su jednaki za sve redove koji se nalaze unutar iste particije pa se oni spremaju samo na jednom mjestu, umjesto da se sprema ista vrijednost za svaki red. Stupac nekog retka sastoji se od imena stupca i njegove vrijednosti, koja mora biti tipa koji je definiran za taj stupac na

razini tablice. Osim toga, svaki stupac retka sadrži i dvije dodatne informacije, vrijeme kad je vrijednost zapisana te vrijeme isteka. Vrijeme zapisivanja služi razrješavanju konflikata, na način da se najnovija vrijednost uvijek smatra ispravnom. Vrijeme isteka, koje ne mora biti postavljeno, označava kad stupac ističe, odnosno do kojeg je trenutka vrijednost tog stupca moguće čitati.

Tipovi podataka koje Cassandra podržava uglavnom su poznati iz relacijskih baza podataka, uz nekoliko dodataka. Cassandra nudi tip podataka brojač, koji je 64-bitni cijeli broj koji je moguće samo uvećavati ili umanjivati. Stupac primarnog ključa ne može biti brojač, a kad se brojač koristi drugi stupci koji nisu dio primarnog ključa mogu biti samo brojači. Ova ograničenja postoje kako bi se u raspodijeljenom okruženju osigurala ažuriranja vrijednosti bez pojave utrka među čvorovima. No, važno je i spomenuti da zato što operacije nad brojačima nisu idempotentne, moguće je da u prisutnosti smetnji na mreži neki čvor pokuša ponovo izvesti operaciju za koju nije dobio potvrdu, a koja je uspjela i prvi put, pa se ona izvede više puta. Iz tog razloga brojač se koristi uglavnom za praćenje podataka u kojima nije bitna apsolutna točnost, primjerice broj posjeta neke web-stranice. Osim brojača, Cassandra podržava i kolekcije koje nisu standardno podržane u relacijskim bazama podataka. Konkretno, dostupni tipovi kolekcija su setovi, liste i mape. Setovi su kolekcije elemenata nekog tipa čiji elementi nisu zapisani nekim posebnim redoslijedom, ali se prilikom čitanja uvijek vraćaju sortirani. Za razliku od setova, elementi liste su zapisani redoslijedom. Setovi se preferiraju nad listama jer je umetanje znatno brže: umetanje u set je jednostavno dodavanje na kraj, dok umetanje u listu zahtijeva čitanje cijele liste, dodavanje elementa na pravo mjesto i ponovno zapisivanje cijele liste. Mape su kolekcije parova ključ-vrijednost. Ključevi i vrijednosti mogu biti bilo kojeg tipa, osim brojača. Osim predefiniranih tipova, u Cassandri je moguće definirati i vlastite, korisnički definirane tipove (eng. *user defined types – UDT*). To su tipovi podataka koji se sastoje od više imenovanih atributa nekog tipa. Primjerice, moguće je definirati tip podataka adresa, koji sadrži tekstualnu vrijednost za ime ulice, cjelobrojnu pozitivnu vrijednost za kućanski broj te tekstualnu vrijednost za grad. Korisnički definirani tipovi mogu se iskoristiti i kao elementi kolekcija, ali moraju biti označeni kao "zamrznuti" (eng. *frozen*). Ovime se označava da su podaci u korisnički definiranom tipu cjelina i serijaliziraju se kao binarni podaci. Zbog toga nije moguće pristupati pojedinačnim atributima korisnički definiranog tipa, nego je moguće čitati samo cijelu vrijednost tipa.

Gledajući opisani model i tipove Cassandra se čini poprilično sličnom relacijskim bazama podatka, ali postoji i niz bitnih razlika. Cassandra ne podržava operacije spajanja (eng. *join*), zbog čega nije moguće modelirati vezu više-prema-više klasičnim

pristupom s veznom tablicom. Cassandra ne poznaje koncept stranih ključeva pa nije moguće održavati referencijalni integritet prema drugim tablicama. Prilikom dohvata nije dopušteno odabrati redoslijed na temelju nekih stupaca – podaci se vraćaju redoslijedom kako su zapisani. Ako se pokuša izvesti upit koji ograničava vrijednosti stupaca rezultata, a nije u potpunosti specificiran ključ za particioniranje, Cassandra će odbiti izvođenje takvog upita. Ove funkcionalnosti jednostavno nisu podržane u Cassandri jer su to skupe operacije. Referencijalni integritet zahtijeva provjeru vezanih entiteta, a njihov pronalazak i dohvat košta vremena, osobito u raspodijeljenom okruženju. Isto je i sa spajanjem s drugim tablicama. Sortiranje podataka u količini kojom Cassandra barata također uzima značajnu količinu resursa. Dohvat podataka bez vrijednosti ključa za particioniranje zahtijeva prolazak cijele tablice, nešto što je pri ovolikoj količini podataka neprihvatljivo Chang et al. (2006).

Premda Cassandra namjerno izostavlja neke mogućnosti kako bi izbjegla izvođenje skupih operacija, sustavi koji ju koriste i dalje postižu sve funkcionalnosti kao i oni koji koriste relacijske baze podataka. Nedostatak mogućnosti u Cassandri zaobilazi se drugačijim pristupom modeliranju podataka Chebotko et al. (2015). Kad se domena problema modelira ustaljenim načinom izgradnje relacijskih baza podataka, izdvoje se entiteti i njihovi atributi te se za svakog definira tablica, a zatim i vezne tablice za veze više-na-više. Nakon toga, obično upotrebom SQL-a, pišu se upiti kako bi se iz stvorenih tablica izvukli podaci koje aplikacije trebaju. S druge strane, kad se domena problema modelira u Cassandri, najprije se promatraju i izdvajaju upiti koje će aplikacija izvoditi. Ako je potrebno pretraživati na temelju nekog stupca, tada će on biti odabran kao ključ za particioniranje. Ako je potrebno podatke vraćati nekim posebnim redoslijedom, tada će stupci na temelju kojih se izračunava redoslijed činiti ključ za grupiranje. Ako u sustavu postoje entiteti koji su u odnosu više-na-više, najprije će se pogledati postoje li i upiti koji dohvaćaju entitete s jedne strane sa svim pripadnima s druge strane i obratno. Ako postoje, model će se morati udaljiti od normalnih forma naslijeđenih iz relacijskih baza podataka. Denormalizacijom podataka u jednoj tablici će se čuvati jedni entiteti sa svim pripadnim drugim entitetima, a u drugoj tablici svi drugi entiteti sa svim pripadnim prvima. Podaci će se duplicirati, zbog čega će aplikacija morati posebno paziti na održavanje konzistentnosti u te dvije tablice, ali to je kompromis koji se u sustavima ovakve veličine i prihvaća. Denormalizacijom će se zauzeti i više prostora za pohranu, ali to danas ne predstavlja toliko veliki problem. Nakon što su otkriveni svi upiti koji će biti potrebni aplikacijama koje koriste Cassandru, stvorit će se i same tablice. Novije verzije Cassandre podržavaju definiranje i manipuliranje podacima upotrebom jezika Cassandra Query Language – CQL.

CQL je veoma sličan standardiziranom SQL-u, i to je napravljeno namjerno kako bi se olakšala prilagodba novim korisnicima Cassandre.

Kako bi se podržala decentralizacija i tolerancija na particioniranje sustava, svi su čvorovi jednaki DeCandia et al. (2017). Informacije o stanju drugih čvorova prenose se upotrebom "protokola glasina" (eng. *gossip protocol*). Svake sekunde, svaki čvor nasumično odabere jedan drugi čvor i pokuša s njim razmijeniti glasine o drugim čvorovima. Ako ne uspije, kod sebe označava taj čvor nedostupnim i počinje širiti tu glasinu. Na razini cijele Cassandre, čvor se u tom trenutku ipak ne odbacuje u potpunosti. Umjesto binarne klasifikacije dostupnosti čvora, upotrebom ϕ detektora ispada s prirastima (eng. ϕ *Accrual Failure Detector*), za svaki se čvor na temelju glasina izračunava razina sumnje ϕ da čvor nije dostupan, što umanjuje mogućnosti lažne detekcije ispalog čvora Hayashibara et al. (2004).

Čvorovi Cassandre su poslužitelji smješteni u ormarima podatkovnih centara (eng. *datacenter racks*). Cassandra koristi informacije o smještaju čvorova kako se sigurnosne kopije istih podataka ne bi smještale u iste ormare i iste podatkovne centre te time ugrozila dostupnost i izdržljivost podataka. Također, te je informacije pogodno koristiti i za učinkovito usmjeravanje zahtjeva između čvorova. Na primjer, ako klijent zatraži podatak od čvora na kojem se podatak ne nalazi i zahtijeva da se podatak potvrdi s još nekoliko sigurnosnih kopija, taj će čvor pitati jednog susjeda za podatke, a ostale susjede samo za sažetak podataka koji će usporediti sa sažetkom izračunatim iz dobivenih podataka. Kako su sažetci u pravilu mnogo manji od podataka, isplati se održavati informacije o blizini susjeda i time omogućiti da se podaci zatražuju samo od najbližeg susjeda, a sažeci od udaljenijih. Najbliži čvor se određuje prema relativnoj bliskosti, koja se osim sa statičnim informacijama o smještaju, izračunava na temelju opaženih trajanja zahtjeva do drugih čvorova. Mehanizmi koji Cassandri pružaju informacije o mrežnoj topologiji, poput opažanja trajanja zahtjeva i konfiguracijskih parametara o ormarima i podatkovnim centrima, nazivaju se "cinkaroši" (eng. *snitches*).

Redovi tablica ravnomjerno se raspodjeljuju u particije upotrebom funkcije za izračun sažetaka. Svi mogući sažeci koji se odabranom funkcijom sažimanja mogu dobiti dijele se na intervale, a svakom intervalu se pridružuje jedna particija. Prije zapisivanja, ažuriranja ili brisanja reda iz tablice izračunava se sažetak i pronađe odgovarajući interval na temelju čega se određuje particija kojoj red pripada. U svrhu očuvanja dostupnosti i izdržljivosti podataka, particije se kopiraju na više čvorova. Pri tome su sve kopije jednako vrijedne, odnosno ne postoji koncept glavne ili primarne replike. Kako je Cassandra decentralizirana, klijenti mogu kontaktirati bilo koji čvor za pisanje ili

čitanje, koji se za tu operaciju naziva koordinatorom. Njegova je dužnost u slučaju čitanja podatke potvrditi s dovoljnim brojem čvorova kako bi se udovoljilo željenoj razini konzistencije. U slučaju pisanja, koordinator je dužan poslati nove podatke na sve čvorove, ali prije slanja uspješnog odgovora klijentu, dužan je pričekati odgovor samo dovoljnog broja čvorova, u skladu sa zahtijevanom razinom konzistencije. Ako neki od čvorova na koji je potrebno zapisati nove podatke nije dostupan, koordinator će to zabilježiti i kad čvor ponovno postane dostupan poslati podatke. Ovaj mehanizam zabilježavanja primopredaja (eng. *hinted handoff*) smanjuje količinu vremena u kojem će ispali čvor imati nekonzistentne podatke i pomaže održavanju protočnosti podataka. Zabilježba se u pravilu ne smatra uspješnim pisanjem i ne doprinosi ispunjenju razine konzistentnosti, osim kad je klijent koji zapisuje podatke odabrao da želi maksimalnu razinu konzistentnosti, odnosno da se novi podaci zapišu svim drugim čvorovima. Ovo je izvedeno tako jer u protivnom pisanje ne bi bilo moguće sve dok nisu svi čvorovi dostupni. U slučaju da je neki čvor nedostupan dulje vrijeme zabilježbe će se nakupljati na drugim čvorovima, a kad se jednom ponovno priključi ostatku na njega će pristići velik broj zahtjeva, u trenutku kad je ranjiv jer se tek pokreće. Kako bi se ovo izbjeglo, kroz konfiguraciju je mehanizam zabilježavanja primopredaja moguće u potpunosti isključiti.

Kako je u slučaju odabira nižih vrijednosti konzistencije prilikom pisanja moguće da dio čvorova Cassandre ostane u nekonzistentnom stanju, koristi se protokol sprječavanja entropije (eng. *anti-entropy protocol*). Protokol se može aktivirati na dva načina. U prvom, ako koordinator neke operacije čitanja primijeti da postoji kopija s nekonzistentnim podacima, on će izdati zahtjev za popravkom. Ako je koordinator uspio zadovoljiti traženu razinu konzistencije popravak će biti odrađen u pozadini, a ako razina nije zadovoljena, pričekat će se popravak pa zatim klijentu vratiti podatke. Drugi način je ručnim pokretanjem popravaka od strane administratora baze podataka. Svaki čvor od susjednih replika zatraži razmjenu sažetaka podataka organiziranih u strukturu podataka poznatu kao Merkleovo stablo. Ta se stabla zatim uspoređuju i ako se naiđe na nekonzistentne podatke, podaci se popravljaju.

Podatke koje Cassandra pohranjuje nalaze se u radnoj memoriji i na disku. Kad čvor dobije nove podatke, najprije ih zapisuje u dnevnik promjena (eng. *commit log*), koji se nalazi na disku. Ovo je sustav oporavka od pogreške jer se pisanje potvrđuje tek nakon zapisa u dnevnik, a ako čvor prestane s radom odmah nakon toga, pregledom dnevnika uspjet će spasiti podatke. Nakon zapisa u dnevnik, podaci se zapisuju u radnu memoriju, u posebnu strukturu podataka zvanu *memtable*, koja se održava za svaku tablicu posebno. Kad se dosegne određena veličina ili broj objekata u njoj,

ta se struktura zapisuje na disk u obliku datoteka koje čine strukturu zvanu *SSTable*. Jednom kad je zapisana, tu strukturu na disku nije moguće mijenjati. Ako se promijene ili obrišu redovi sadržani u njoj, te će se informacije nadodati u memorijsku strukturu i kasnije zapisati u novi *SSTable* na disku. Ovime se osigurava visoka brzina pisanja jer umjesto pronalaska reda na disku, čitanja, ažuriranja i ponovnog zapisivanja Cassandra samo nadodaje nove informacije u memoriju. No vremenom ovakav način postaje problem za čitanja jer Cassandra mora čitati *memtable* i više *SSTable* struktura kako bi pronašla prave vrijednosti retka, a i zauzima se više prostora na disku zbog toga što stare vrijednosti redaka ostaju zapisane u starijim *SSTable* strukturama. Zato Cassandra povremeno vrši sažimanje (eng. *compaction*), odnosno uzima sve podatke o redcima i zapisuje samo aktualnu vrijednost u novi *SSTable*, dok se stari brišu. *SSTable* osim podataka sadrži i Bloomov filter, probabilističku strukturu koja služi ispitivanju nalazi li se neki redak u danom *SSTable*. Bloomov filter može sa sigurnošću utvrditi da se neki redak ne nalazi u *SSTable*, ali ako kaže da se neki redak nalazi, moguće je da to nije istina, odnosno da će Cassandra morati pogledati na disk i provjeriti. Ovo je ponašanje prihvatljivo, jer u većini slučajeva izbjegava relativno skupo čitanje s diska, a nikad neće uzrokovati da zapis nekog retka bude propušten. Redci unutar *SSTable* su sortirani, a osim Bloomovog filtera zapisuje se i indeks preko kojeg se brzo može pronaći traženi redak u datoteci s podacima.

5. Apache Spark

Apache Spark je razvojni okvir otvorenog koda namijenjen raspodijeljenoj obradi velikih količina podataka. Spark je nastao na Berkeleyju, na Sveučilištu u Kaliforniji, a projekt je kasnije doniran Apache Software Foundation, koji se od tada brine za njegovo održavanje. Spark se sastoji od više modula. Modul Spark Core osnovni je modul koji služi primanju zadataka, raspoređivanju i optimizaciji obrade na čvorovima. Iznad tog modula izgrađeni su i drugi moduli, kao što je Spark SQL za specificiranje zadataka kao SQL upita, MLlib koji sadrži implementacije niza algoritama iz područja strojnog učenja ili modul GraphX koji se koristi za operacije nad grafovima. Osim onih koji dolaze sa samim Sparkom, razvijaju se i drugi moduli koji implementiraju razne druge funkcionalnosti. Apache Spark je napisan u programskom jeziku Scala, ali preko programskih sučelja omogućava zadavanje zadataka koristeći i Javu, Python i R bez većih razlika u performansama Chambers i Zaharia (2018). Spark se u svom radu oslanja na voditelja raspodijeljene grupe računala i na raspodijeljeni mehanizam pohrane. Postoji vlastita implementacija voditelja, ali je moguće odabrati i nekog drugog, primjerice Hadoopov YARN. Za raspodijeljeni mehanizam pohrane postoji niz opcija, poput HDFS-a, raspodijeljenog datotečnog sustav razvijenog u sklopu Apache Hadoopa, baze podataka poput Apache Cassandre ili pak rješenja u oblaku poput Amazonovog S3.

Zadaci koje Apache Spark izvodi zadaju se koristeći dostupna programska sučelja. Većina tih programska sučelja dostupna je za sve programske jezike koje Spark podržava, osim sučelja čije značajke zbog ograničenja programskog jezika nije moguće implementirati. Dostupna programska sučelja dijele se na sučelja visoke i niske razine. Sučelja visoke razine su preferirana jer apstrahiraju niže operacije koje Spark izvodi poput podjele zadataka čvorima, ali kada se za tim pokaže potreba upotrebom sučelja niske razine moguće je pristupiti svim značajkama Sparka. Sučelja visoke razine su DataFrame, Dataset i SQL. DataFrame je dostupan u svim jezicima i koristi se tako da se u programskom kodu kroz niz poziva metoda specificiraju operacije koje je nad podacima potrebno napraviti. Dataset je vrlo sličan DataFrameu, ali je strogo tipizi-

ran. Za ulazne podatke mora se specificirati shema kojom se deklariraju tipovi ulaznih podataka, a specificranjem operacija prati se tip ulaznih i izlaznih podataka čime se osiguravaju ispravni tipovi već za vrijeme prevođenja. Dataset je dostupan samo u jezicima koji se izvode na Javinom virtualnom stroju, jer drugi jezici koji se koriste sa Sparkom (Python i R) ne podržavaju koncept strogog tipiziranja, odnosno tipove razrješavaju za vrijeme izvođenja. Visoko programsko sučelje SQL omogućava specificiranje operacija upotrebom jezika Spark SQL, koji je sintaksno skoro identičan SQL standardu. Programska sučelja također je moguće kombinirati. Na primjer, moguće je na učitanim podacima specificirati SQL upit, a zatim povratni tip pretvoriti u DataFrame i zvati metode DataFrame sučelja. Programska sučelja niske razine su RDD i raspodijeljene varijable. Sučelje RDD omogućava manipulacije na razini raspodijeljenih izdržljivih skupova podataka (eng. *resilient distributed dataset – RDD*), koji su osnovni blokovi podataka nad kojima Spark izvršava operacije. Upotrebom raspodijeljenih varijabli moguće je jednostavno održavati i manipulirati zajedničke vrijednosti kojima pristupaju svi čvorovi.

Instanca Apache Sparka sastoji se od više čvorova koje koordinira vanjski voditelj raspodijeljene grupe računala. Kao što je spomenuto ranije, postoji više opcija za voditelja, koji svi imaju svoje specifičnosti. Čvorovi su organizirani u arhitekturu s jednim gospodarom i više radnika. Gospodar (eng. *driver*) drži kontekstni objekt SparkContext, koji predstavlja aktivnu vezu s ostalim čvorovima i preko kojeg se zadaju zadaci i dijele zajedničke varijable. Ostali čvorovi su radnici (eng. *executor*) koji čekaju da im gospodar dodijeli zadatak, a kada ga dobiju i izvedu javljaju gospodaru rezultate. Kod izvođenja nekog posla, ulazni podaci se dijele na particije i svaki radnik izvodi iste operacije na podjednakom broju particija. Ponekad posao zahtijeva da se dogodi reparticioniranje, pa radnici razmjenjuju particije, koordinirani od strane voditelja.

Zadatke je ovisno o smještaju gospodara i radnika moguće izvoditi na tri načina. Prvi i najčešći je izvođenje s grupom (eng. *cluster mode*), u kojem se posao pokreće tako da se voditelju predaje prevedena JAR datoteka, Python ili R skripta. Voditelj na nekom od čvorova u grupi pokreće dobivenu datoteku, a pokrenuti proces je gospodar za taj posao koji odmah uspostavlja vezu s voditeljem. Po uspostavi veze između gospodara i voditelja, gospodar dobiva pristup kontekstualnom objektu SparkContext, a voditelj na drugim čvorovima pokreće radnike i relevantne informacije o njima šalje gospodaru. Kad je posao gotov, proces gospodar prestaje s radom, voditelj gasi sve radnike, a rezultati posla ostaju dostupni preko voditelja. Drugi način izvođenja poslova je klijentski način (eng. *client mode*), u kojem se na nekom vanjskom računalu koje nije čvor u grupi pokrene proces gospodar, na primjer na lokalnom računalu. Nakon

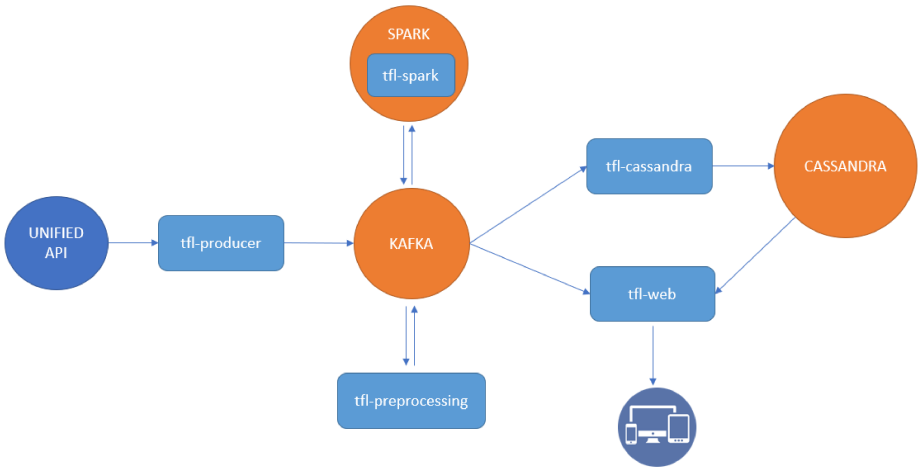
pokretanja taj proces uspostavlja vezu s voditeljem i dalje je izvođenje identično kao u prvom načinu. Ovaj se način obično koristi kad je gospodar neki alat koji omogućava interaktivno izvođenje posla na Sparku. Posljednji način je lokalni (eng. *local mode*), u kojem se paralelizam gospodara i radnika ostvaruje upotrebom više dretvi na istom računalu. Ovaj način je osmišljen kako bi se tijekom razvoja omogućilo izvođenje na lokalnom računalu i nije namijenjen izvođenju na produkcijskoj razini.

Programski kod koji koristi programska sučelja Sparka ne izvodi operacije nad podacima, nego samo služi specificiranju operacija koje je nad podacima potrebno obaviti. Na primjer, programski kod koji se sastoji od poziva funkcije za učitavanje podataka iz neke datoteke, poziva funkcije za sortiranje tih podataka i poziva funkcije koja uzima prvi zapis iz sortiranih podataka, nakon izvođenja drugog poziva nije još niti otvorio datoteku u kojoj se nalaze podaci. Svaka operacija, osim onih koje dohvaćaju podatke, uzrokuju jedino da programsko sučelje Sparka zapamti da ju je potrebno napraviti. Kad opisani kod pozove funkciju za učitavanje, programsko sučelje zapamti da je potrebno napraviti učitavanje i vraća se iz funkcije. Nakon toga, kad opisani kod pozove funkciju za sortiranje, programsko sučelje zapamti da je potrebno sortirati dohvaćene podatke i vrati se na izvođenje korisnikovog koda. Tek kad korisnik pozove treću funkciju, koja dohvaća podatke, Spark će započeti izvođenje, najprije učitavanje, zatim sortiranje i konačno dohvat najvećeg podatka koji vraća pozivatelju. Na taj način pozivima operacija Spark u pozadini izgrađuje aciklički usmjereni graf transformacija, koji se izvodi tek kad korisnik zatraži rezultat izračuna. Ovo je izvedeno na takav način kako bi Spark mogao optimizirati operacije upotrebom optimizatora Catalyst. Kad korisnički kod zatraži podatak, Catalyst analizira izgrađeni graf na temelju kojeg stvara logički, zatim fizički plan izvođenja te konačno generira kod koji provodi taj plan. Prilikom izgradnje logičkog plana Catalyst na temelju ugrađenih pravila evaluira različite moguće načine izvođenja operacija koje je korisnik specificirao i odabire najpovoljniji. Na temelju tog logičkog plana stvara se jedan ili više fizičkih planova izvođenja koji odgovaraju različitim mogućim generiranim izvedivim kodovima. Njih se također evaluira izračunavanjem troškova te se na temelju toga odabire najpovoljniji fizički plan izvođenja. Konačno, na temelju odabranog fizičkog plana izvođenja se generira izvedivi kod. Generirani kod sastoji se od niza zadataka kao što su preslikavanja i filtriranja, koji se izvode paralelno na zasebnim dretvama procesa radnika. Svi zadaci koji se izvode samo na jednoj particiji, odnosno ne zahtijevaju reparticioniranje i razmjenu podataka s drugim čvorovima, dio su iste faze. Svaki radnik izvodi cijelu fazu neovisno o drugima, a kad faza završi razmijenjuje potrebne podatke s drugim radnicima prije nego nastavi s izvođenjem iduće faze s novodobivenim particijama.

Činjenica da radnici izvode generirani kod, a ne kod koji je napisan u jeziku u kojem se koristi programsko sučelje, razlog je zašto neovisno o jeziku u kojem se specificiraju zadaci performanse Sparka ostaju konzistentne.

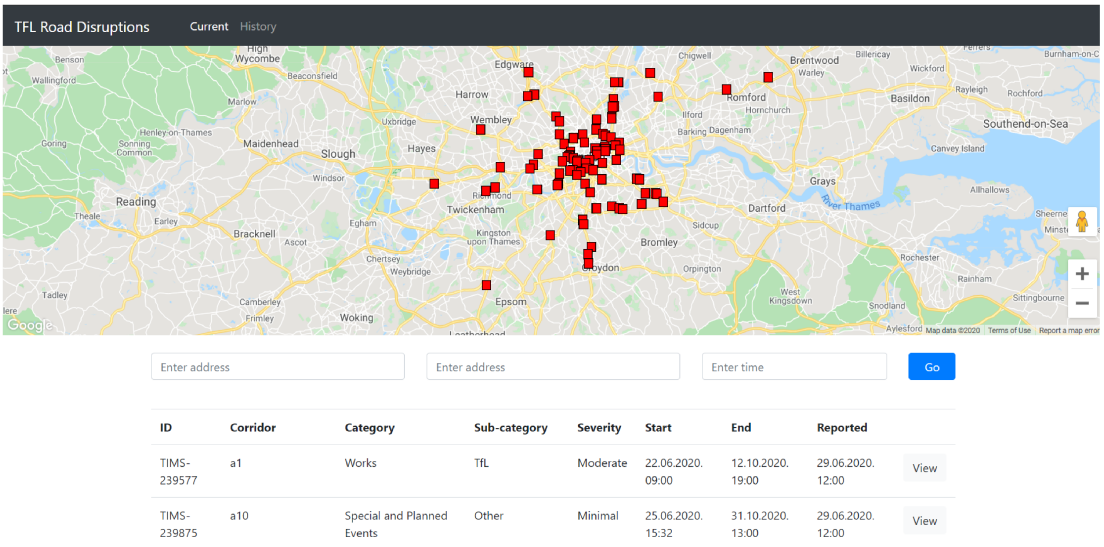
6. Implementirano rješenje

Implementirano rješenje se, osim opisanih alata, sastoji od više vlastitih komponenata. Za implementaciju je odabran programski jezik Java, a gdje je bilo pogodno iskorištena je i razvojna okolina Spring. Sve komponente koriste Apache Maven alat za automatiziranu izgradnju projekata i dohvat potrebnih biblioteka. Kako se ista struktura podataka proteže kroz više komponenata najprije je implementiran jednostavan Mavenov modul *tfl-model* koji sadrži klase koje opisuju podatke. S mrežnih stranica Traffic for Londona preuzeta je definicija programskog sučelja Unified API u obliku Swagger datoteke, standardiziranog formata opisa programskih sučelja. Iz te datoteke je koristeći alat za automatsko generiranje koda generiran Mavenov modul *tfl-api*, koji sadrži klase koje predstavljaju podatke koji se razmijenjuju, kao i klase potrebne za ostvarenje komunikacije s programskim sučeljem. Zatim je napravljen Mavenov modul *tfl-producer* koji upotrebom *tfl-api* modula periodički dohvaća podatke s programskog sučelja Unified API i podatke prosljeđuje u Apache Kafku. Periodički dohvat i povezivanje s Kafkom je ostvareno upotrebom biblioteka dostupnih u razvojnom okruženju Spring. Poslani podaci se najprije predobrađuju, što je zadaća implementiranog Mavenovog modula *tfl-preprocessing*, koji čita s teme na koju stižu neobrađeni podaci, predobrađuje ih upotrebom biblioteke Kafka Streams te objavljuje na drugu temu u Kafki. Iz te teme predobrađene podatke čitaju dvije komponente. Prva komponente, Mavenov modul *tfl-cassandra*, pročitane podatke izravno sprema u Cassandru. Druga komponenta, *tfl-spark*, na temelju dobivenih podataka uči model strojnog učenja i njegove parametre šalje nazad u Kafku. Modul *tfl-spark* je jednostavan Mavenov modul koji se pakira u izvedivu *jar* datoteku i ne pokreće se kao samostalna aplikacija. Umjesto toga, datoteka se šalje Spark voditelju, koji ju na jednom čvoru pokreće kao gospodara tog posla, a na drugim čvorovima stvara procese radnike kojima pokrenuti gospodar šalje zadatke, u obliku serijaliziranih klasa iz pokrenute *jar* datoteke. Klijenti sustav koriste preko web-aplikacije izložene iz modula *tfl-web*, koji je povezan s Kafkom iz koje dohvaća parametre modela te Cassandrom iz koje dohvaća trenutne i povijesne podatke o poremećajima. Opisana arhitektura sustava prikazana je na slici 6.1.

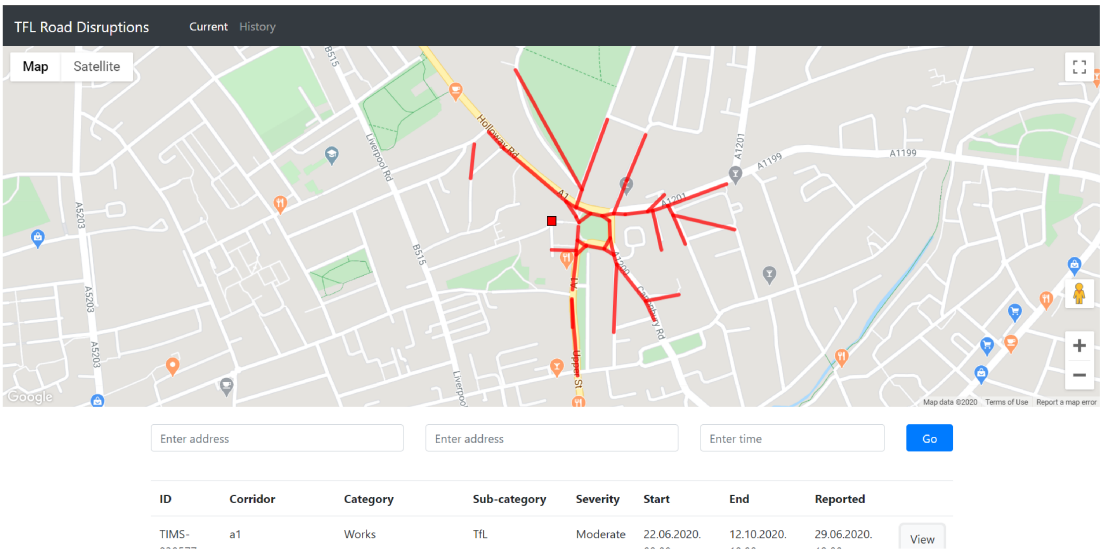


Slika 6.1: Arhitektura implementiranog rješenja.

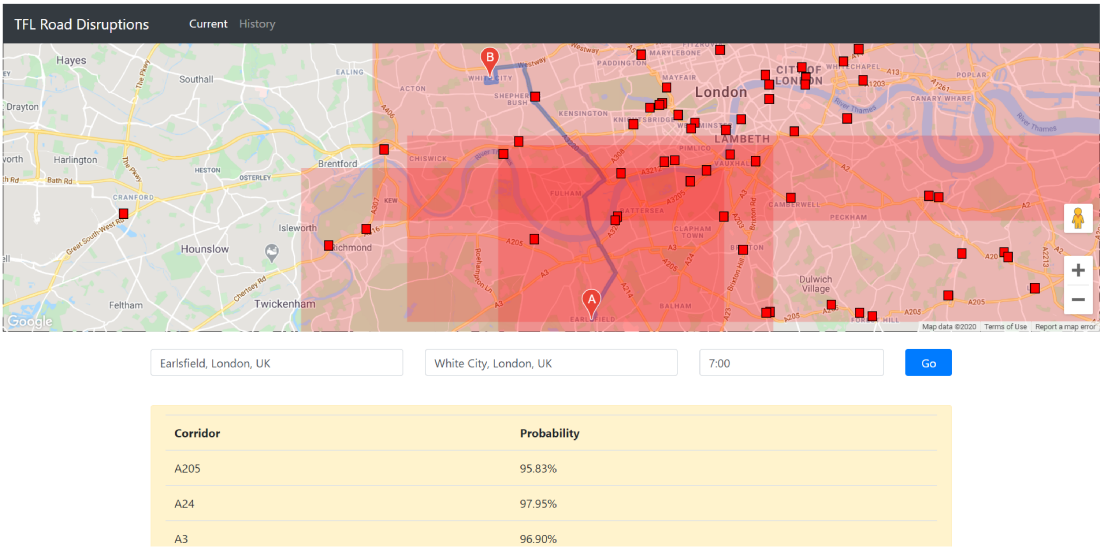
Izložena web-aplikacija sastoji se od serverskog djela, ostvarenog upotrebom razvojnog okvira Spring, i klijentskog djela ostvarenog upotrebom HTMLa, CSSa uz biblioteku Bootstrap za stiliziranje i JavaScripta uz biblioteku jQuery za manipulaciju elementima na stranici. Korisnicima se upotrebom programskog sučelja Google Maps prikazuje karta na kojoj su markeri koji označavaju trenutne prometne poremećaje, a oni su uz nešto više detalja popisani i u tablici ispod karte, kao što je prikazano na slici 6.2. Klikom na marker ili gumb unutar tablice, korisnicima se prikazuju zahvaćeni segmenti zahvaćenih ulica za odabrani poremećaj, kao što je vidljivo na slici 6.3. Između karte i tablice nalaze se tri unosna polja u koja korisnici unose mjesto polaska, odredište i vrijeme njihovog putovanja. Klikom na gumb uz polja korisnicima se na karti prikazuje preporučena ruta njihovog putovanja, dobivena koristeći programsko sučelje Directions API, dio Google Mapsa. Za rutu se određuje kroz koje prometne pravce prolazi i za te prometne pravce se upotrebom naučenog modela izračunava vjerojatnost pojave prometnih poremećaja u odabrano vrijeme putovanja. Ako su na nekom prometnom pravcu poremećaji veoma izgledni, na karti se područje prometnog pravca označava crvenim pravokutnikom, ako su srednje izgledni žutim pravokutnikom, a ako poremećaji nisu izgledni prometni pravac se ne uzima u obzir. Dobivene vjerojatnosti pojave prometnih nesreća po prometnim pravcima se u obliku postotka prikazuju u tablici ispod unosnih polja. Primjer predviđanja za neku rutu prikazan je na slici 6.4.



Slika 6.2: Prikaz trenutnih prometnih poremećaja.



Slika 6.3: Prikaz detalja prometnog poremećaja.



Slika 6.4: Prikaz predviđanja prometnih poremećaja.

Podaci koje sustav prikuplja sastoje se od vremenske oznake, prometnog pravca, popisa zahvaćenih segmenata zahvaćenih ulica, razine ozbiljnosti i drugih informacija. Te se informacije korisnicima prikazuju prilikom pregleda poremećaja, ali nisu sve relevantne za učenje modela. Kako su ulazni podaci na temelju kojih se radi predviđanje sastavljeni od vremenske oznake i prometnog pravca, ulazni primjeri na kojima se model uči također moraju sadržavati samo te informacije, a ostale je potrebno ukloniti. Dodatno, potrebno je napraviti neke transformacije nad preostalim podacima. Vremenska oznaka, koja je reprezentirana kao numerička vrijednost koja označava broj milisekundi proteklih od fiksnog trenutka u prošlosti, predstavlja vrijednost iz koje bi model mogao imati problema s izlačenjem korisnih zaključaka. Na primjer, za očekivati je da je vjerojatnost pojave prometnih poremećaja danas i za tjedan dana vrlo slična, iako su vremenske oznake dosta različite. S druge strane, vjerojatnost pojave poremećaja sigurno je viša u jutarnjim satima kad su prometnice zagušenije zbog putovanja na posao nego u podne, iako su te vremenske oznake bliže. Iz tog je razloga prije učenja potrebno napraviti izdvajanje značajki, u kojem se proučavanjem prikupljenih podataka i na temelju znanja o domeni uvode transformacije nad podacima, kako bi se modelu olakšalo predviđanje. U ovom primjeru, vremenska oznaka izdvojena je u dvije značajke, u sat u danu i dan u tjednu. Dodatno, kako su te dvije značajke i oznaka prometnog pravca kategoričke varijable, a odabrani model logističke regresije radi samo s numeričkima, potrebno ih je na neki način kodirati u numeričke. To se najjednostavnije ostvaruje tako da se svakoj opciji pridijeli neka numerička vrijednost, primjerice ponedjeljku vrijednost jedan, utorku dva, srijedi tri i tako dalje. No takav

način može dovesti do neželjenih posljedica kad se model počne prilagođavati pridijeljenim oznakama, primjerice tako da ponedjeljak i utorak smatra sličnijima nego ponedjeljak i petak jer su manje razlike između njima pridijeljenim vrijednostima Alpaydin (2009). Iz tog razloga, obično se upotrebljava *one-hot* kodiranje, u kojem se kategorička varijabla s n mogućih ishoda kodira s n značajki koje poprimaju vrijednost 1 ako taj ishod ostvaren, odnosno 0 ako nije. Upravo to kodiranje iskorišteno je za kodiranje svih kategoričkih značajki primjera u ovom sustavu.

Pretvorba ulaznih podataka u primjere s odgovarajućim značajkama kao zajednička poslovna logika dio je modula *tfl-model*. Njegovom upotrebom modul *tfl-spark* dobivene podatke pretvara u odgovarajuće primjere i prosljeđuje ih algoritmu strojnog učenja. Spark preko svoje biblioteke MLlib nudi velik broj gotovih algoritama strojnog učenja, no kako je u ovom sustavu bilo potrebno koristiti algoritam koji može učiti kako podaci pristižu izravno iz toka podataka, odabir je bio značajno sužen. Odabran je MLlib algoritam *StreamingLogisticRegressionWithSGD*, koji koristi model logističke regresije i gradijentni spust kao optimizacijski postupak. Učenje modela se događa inkrementalnom obradom manjih skupova primjera, a nakon obrade svakog ažurirani parametri modela se šalju u Kafku. Kako bi se naučeni parametri sačuvali između pokretanja posla na Sparku, oni se također spremaju na datotečni sustav. Prilikom pokretanja, posao koji se izvodi potraži ovu datoteku i ako ona postoji model se inicijalizira s tim parametrima, a inače s proizvoljnim početnim vrijednostima.

7. Zaključak

Početkom široke upotrebe interneta potkraj prošlog stoljeća količina podataka koju programski sustavi prikupljaju počela je značajno rasti. Prijenos, pohrana i analiza tih količina podataka gurnula je tadašnje alate do njihovih granica, čime se pred inženjere tehnoloških divova, *start-up* tvrtki u uzletu i akademsku zajednicu postavio zadatak izgradnje nove generacije alata. Činjenica da je velik dio tih alata postao alatima otvorenog koda omogućila je inženjerima i znanstvenicima sa svih krajeva svijeta da svojim radom sudjeluju u njihovom razvoju. Velik dio njih doniran je američkoj tvrtki Apache Software Foundation, koja danas aktivno radi na njihovom daljnjem razvoju i održavanju preko svoje decentralizirane mreže suradnika.

U ovom radu detaljnije su analizirana tri alata otvorenog koda koji su proizvodi Apache Software Foundationa namijenjeni velikim količinama podataka. Apache Kafka, sustav za razmjenu poruka temeljen na principu objava i pretplata, koristi se jer značajno pojednostavljuje arhitekturu sustava, oslobađa komponente od brige za druge dijelove sustava te otvara mogućnost jednostavnog dodavanja i uklanjanja komponenati. Baza podataka Apache Cassandra izvrstan je izbor za *big data* sustave jer bez obzira na pohranjenu količinu podataka održava visoke performanse, a klijentima pruža fleksibilnost u odabiru željenih razina izdržljivosti, dostupnosti i konzistentnosti. Apache Spark je alat namijenjen učinkovitoj raspodijeljenoj obradi velikih skupova podataka koji zbog svoje modularne građe, programskih sučelja različitih razina i mogućnosti upotrebe iz različitih okruženja otvara mogućnost napredne obrade podataka u raznovrsnim kontekstima.

Upotrebom ova tri alata ostvarena je implementacija sustava za analizu gradskog prometa. Ostvarena je web-aplikacija preko koje korisnici mogu interaktivno pregledavati trenutne i prošle prometne poremećaje. Uz to, mogu unijeti podatke o svom budućem putovanju na temelju čega dobivaju rutu i upotrebom umjetne inteligencije procjenu mogućih poremećaja na koje trebaju obratiti pozornost. U implementiranom sustavu Apache Kafka je omogućila jednostavno povezivanje svih komponenta sustava: komponente koja dohvaća podatke, komponente koja radi predobradu, kom-

ponente zadužene za pohranu podataka, komponente koja uči model strojnog učenja i same web-aplikacije. Upotrebom Apache Cassandre omogućena je pohrana svih prikupljenih podataka kojima korisnici mogu pristupati bez performansnih problema. Konačno, upotrebom Apache Sparka i njegove biblioteke MLlib ostvareno je učenje parametara modela strojnog učenja izravno na toku podataka, upotrebom kojih korisnici dobivaju predviđanja budućih poremećaja.

LITERATURA

Ethem Alpaydin. *Introduction to Machine Learning*. 2009.

Bill Chambers i Matei Zaharia. *Spark: The Definitive Guide: Big Data Processing Made Simple*. 2018.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, i Robert E. Gruber. *Bigtable: A Distributed Storage System for Structured Data*. 2006.

Artem Chebotko, Andrey Kashlev, i Shiyong Lu. *A Big Data Modeling Methodology for Apache Cassandra*. 2015.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, i Werner Vogels. *Dynamo: Amazon's Highly Available Key-value Store*. 2017.

Seth Gilbert i Nancy A. Lynch. *Perspectives on the CAP Theorem*. 2012.

Naohiro Hayashibara, Xavier Defago, Rami Yared, i Takuya Katayama. *The Phi Accrual Failure Detector*. 2004.

Eben Hewitt i Jeff Carpenter. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. 2020.

Jay Kreps, Neha Narkhede, i Jun Rao. *Kafka: a Distributed Messaging System for Log Processing*. 2011.

Fiaz Majeed, Muhammad Sohaib Mahmood, i Mujahid Iqbal. *Efficient data streams processing in the real time data warehouse*. 2010.

Gwen Shapira, Neha Narkhede, i Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. 2017.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, i Ion Stoica. *Spark: Cluster Computing with Working Sets*. 2010.

Analiza velike količine podataka o stanju gradskog prometa upotrebom programskih alata otvorenog koda

Sažetak

Porastom količina podataka koje programski sustavi prikupljaju i obrađuju, osobito početkom masovne upotrebe interneta, mnogi alati u upotrebi pokazali su se nedostatnima. Iz tog razloga, velik broj inženjera i znanstvenika uključio se u razvoj alata otvorenog koda namijenjenih upotrebi u novonastalom području velikih podataka. U ovom radu prikazana su tri takva alata, Kafka, Cassandra i Spark, proizvodi Apache Software Foundationa. Apache Kafka je alat za razmjenu poruka koji služi pojednostavljenju arhitekture sustava i predstavlja središnju točku preko koje se razmijenjuju svi podaci u sustavu. Apache Cassandra je raspodijeljena baza podataka koja omogućava čitanje i pisanje velikih količina podataka uz održavanje visokih performansi. Apache Spark je alat za raspodijeljenu obradu podataka poput napredne analitike ili upotrebe umjetne inteligencije. Na primjeru sustava koji analizira stanje u prometu demonstrirane su funkcionalnosti i upotreba navedenih alata u ostvarenju web-aplikacije koja korisnicima nudi pregled zapaženih prometnih poremećaja i predviđanje budućih.

Ključne riječi: Big data, Apache Kafka, Apache Cassandra, Apache Spark

City Traffic Big Data Analysis Using Open Source Software Solutions

Abstract

With the dramatic increase in the quantities of data collected and processed by software systems, especially after massive adoption of internet, many tools in use have shown themselves as insufficiently capable. To address this issue, engineers and scientists from around the world have started building open source software tools intended for use in the newly established field of big data. In this thesis, three such tools are presented, Apache Software Foundation's Kafka, Cassandra and Spark. Apache Kafka is a messaging system which greatly simplifies big data system's architecture and presents a central information exchange point in these systems. Apache Cassandra is a distributed database which retains high performance despite data volumes stored in it. Apache Spark is a distributed tool for data processing such as advanced analytics and artificial intelligence usages. This thesis demonstrates features and use cases for these three tools using an example system for traffic data analysis, which exposes a web-aplication that allows users to browse collected traffic disruption data and find predictions for possible future such events.

Keywords: Big data, Apache Kafka, Apache Cassandra, Apache Spark