SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5569

Obrada toka senzorskih podataka korištenjem platforme Apache Apex

Borna Bezjak

Zagreb, lipanj 2018.

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

Zagreb, 16. ožujka 2018.

ZAVRŠNI ZADATAK br. 5569

Pristupnik: **Borna Bezjak (0036488385)**
Studij: Računarstvo
Modul: Programsko inženjerstvo i informacijski sustavi

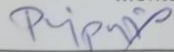
Zadatak: **Obrada toka senzorskih podataka korištenjem platforme Apache Apex**

Opis zadatka:

Vaš zadatak je istražiti i opisati platformu Apache Apex te osmisliti i izvesti u programskom jeziku Java programsko rješenje za obradu senzorskog toka podataka korištenjem platforme ove platforme. Ostvareno rješenje ispitajte na odabranom studijskom slučaju toka senzorskih podataka.
Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

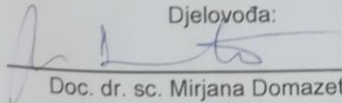
Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 15. lipnja 2018.

Mentor:



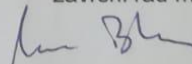
Izv. prof. dr. sc. Krešimir Pripužić

Djelovođa:



Doc. dr. sc. Mirjana Domazet-Lošo

Predsjednik odbora za završni rad modula:



Izv. prof. dr. sc. Ivica Botički

Sadržaj

Uvod 1

1. Apache Apex 2

    1.1. Struktura platforme Apache Apex 3

    1.1. Izvođenje aplikacija na platformi *Apache Apex* 5

2. Osnovni elementi platforme *Apache Apex* 7

    2.1. Operatori 7

    2.2. Vrata 10

    2.3. Tokovi podataka 10

    2.4. Paketi 12

    2.5. Svojstva i atributi 12

    2.6. Kontrolni *tuplovi* 13

    2.7. Pravila afiniteta 13

    2.8. Načini rada 14

    2.9. Validiranje aplikacije 15

3. Otpornost na pogreške i mehanizmi oporavka 16

    3.1. Dijeljenje operatora, skalabilnost 16

    3.2. Statička podjela operatora 17

    3.3. Dinamička podjela 17

    3.4. Kontrolne točke 17

    3.5. Uloga poslužitelja za priručnu pohranu 18

    3.6. Stanje operatora 19

4. Praktična implementacija aplikacije za obradu tokova podataka 20

Zaključak 26

Popis slika 27

Literatura 28

Sažetak 29

Summary 30

Uvod

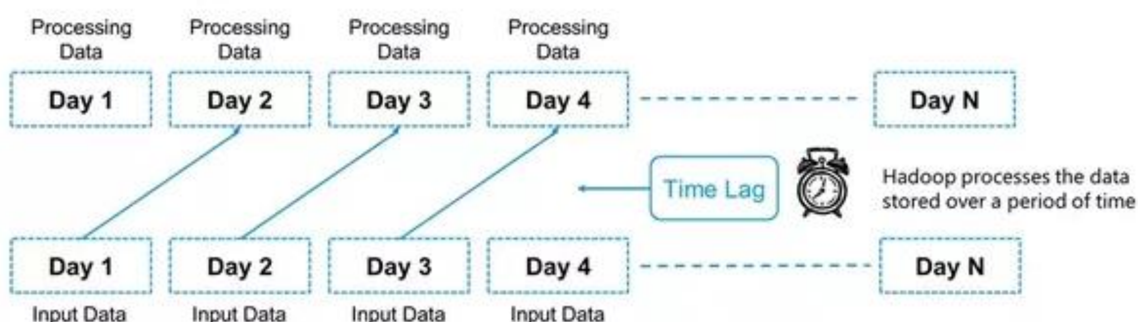
Svaka ljudska aktivnost generira podatke, a u suvremenom svijetu količinu proizvedenih podataka mjerimo terabajtima i petabajtima. U proteklih dvije godine generirano je više podataka nego u cijeloj ljudskoj povijesti, a samo u 2017. godini generirano je više podataka nego u prijašnjih 5000 godina.[9]

Postoje dva načina obrade tako velike količine podataka: obrada podataka u skupinama (*batch processing*) i obrada tokova podataka (*stream processing*). *Batch processing* predstavlja obradu manjeg bloka podataka po isteku nekog perioda vremena koji su već spremljeni na nekom mediju. Nakon isteka nekog perioda vremena, sustav analizira i obrađuje sve podatke koji su uneseni u isteklom periodu vremena, tj. od zadnje obrade podataka. Prednost sustava koji implementira *batch processing* je sposobnost obrade jako velikog skupa podataka odjednom, no mana je da takva obrada traje jako dugo. [5]

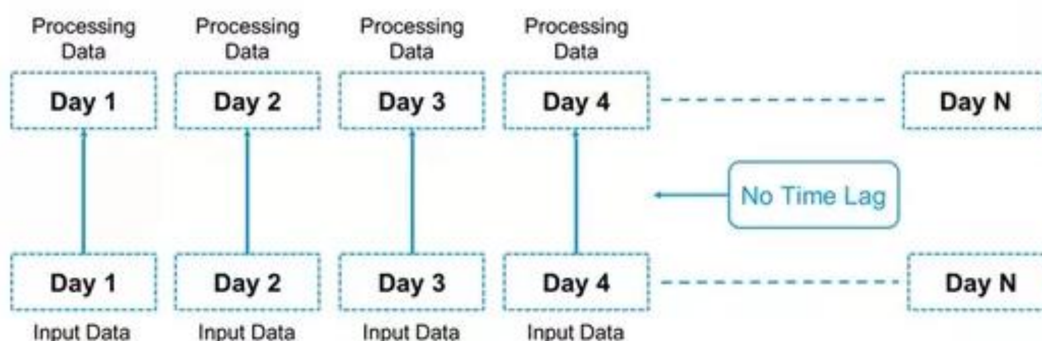
Drugi način je, umjesto obrade podataka nakon isteka većeg vremenskog intervala, podatke obrađivati u realnom vremenu. Takav pristup nazivamo obrađivanje tokova podataka (*stream processing*). Nakon što podatci uđu u sustav prvo se odrađuju, a tek nakon obrade spremaju se na medij. Dobre strane sustava koji implementira obradu tokova podataka je mogućnost dobivanja rezultata obrade u vrlo kratkom vremenu, a rezultati se dinamički osvježavaju kako pristižu novi podaci. [10]

1. Apache Apex

Apache Apex je platforma koja omogućuje stvaranje *batch* i *stream* aplikacija. Dizajnirana je za asinkronu obradu velike količine podataka u realnom vremenu. Napisana je u Javi i Scali, te je neovisna o operacijskom sustavu. Sastoji se od jezgre (*Apex Core*) i biblioteke gotovih komponenti koju zovemo *Apex Malhar*. Razdvajanje funkcionalnih i operacijskih specifikacija olakšava rad s platformom tako što se programer može fokusirati na programiranje konkretnih korisničkih zahtjeva, bez prevelikog razmišljanja o pojedinostima rada platforme. [1]



Slika 1 Batch obrada podataka [2]



Slika 2 Stream obrada podataka [2]

Tvrtka *DataTorrent* započela je s razvojem *Apex*a 2012. godine, a 2015. platforma ulazi u *Apache* inkubaciju. Platforma je otvorenog koda, a cijeli kod *Apex* jezgre i *Apex Malhara*

dostupan je za preuzimanje na *GitHubu*. Trenutna verzija *Apex Corea* je 3.6.0, a *Apex Malhara* je 3.8.0.[4]

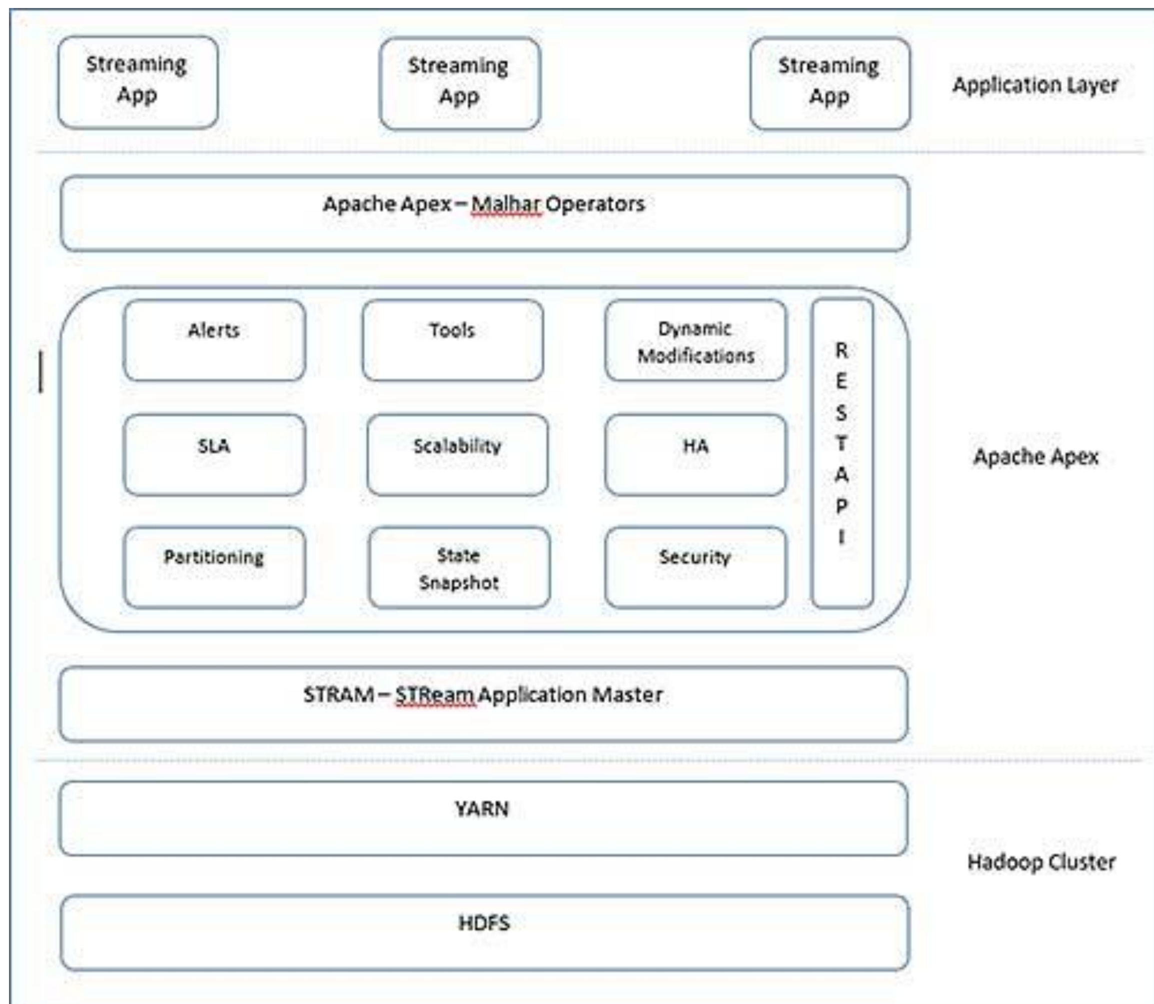
Kako bismo mogli koristit *Apache Apex*, potrebno je na računalu instalirati *Java* JDK, *Apache Maven* i *Git*, podesiti varijable okruženja JAVA_HOME, MAVEN_HOME i M2_HOME, te u varijablu PATH dodati odgovarajuće izvršne datoteke. Na operacijskom sustavu *Windows* potrebno je skinuti datoteku winutils.exe koja se nalazi u paketu hadoop-common-2.2.0-bin-master, i postaviti varijablu okruženja HADOOP_HOME. Nakon navedenih koraka novi projekt moguće je stvoriti pomoću *Apex* CLI-a preko konzole, ili pomoću IDE-a kao *Maven* projekt.[1]

1.1. Struktura platforme Apache Apex

Glavni dijelovi platforme *Apache Apex* su *Apex Core* i *Apex Malhar*.

Apex Core je radni okvir za izgradnju raspodijeljenih sustava koji se temelji na platformi Apache *Hadoop*, a njegove funkcije nadopunjuju *Apex Malhar*.

Apex Malhar je skup gotovih komponenata koji omogućuju brz razvoj aplikacija. *Malhar* nije nadogradnja, već ključan dio *Apexa*, dizajniran kako bi se savršeno uklopio s *Apex* jezgrom. [1]



Slika 3 pregled komponenata Apache Apexa[6]

Apache Apex je namijenjen za rad na platformi Apache *Hadoop*, tj. njegovom upravitelju radnim procesima *Apache Hadoop YARN* i dizajniran je kako bi se integrirao s ostatkom tehnologija koje se temelje na platformi *Apache Hadoop*.

Apache Hadoop je kolekcija raznih alata koji omogućuju raspodijeljenu obradu i raspodijeljeno spremanje velikih količina podataka na računalnim sustavima koji su sastavljeni od mnogo manjih računala koje nazivamo grozdovima.[3] Prvotno dizajniran za raspodijeljene sustave izgrađene od komponenti koje su relativno jeftine, široko dostupne i neovisne o proizvođaču, danas se *Hadoop* može naći i na raspodijeljenim sustavima izgrađenim od komponenti više kategorije. *Apache Hadoop* temelji se na principu da, ukoliko dođe do kvara neke fizičke komponente, sustav treba biti sposoban riješiti problem, bez angažmana krajnjeg korisnika. [1]

Dvije najvažnije komponente *Hadoopa* su HDFS i YARN.

HDFS (*Hadoop Distributed File System*) je raspodijeljeni sustav visoke otpornosti na greške i visoke propusnosti. [1]

YARN (*Yet Another Resource Negotiator*) je sustav za upravljanje resursima i raspoređivanje poslova. Zadaća YARN-a je alociranje sistemskih resursa za izvođenje aplikacija i određivanje kada i na kojem čvoru raspodijeljenog sustava će se izvršiti razni zadatci. Kako YARN ne bi pretjerano koristio resurse, on nije aktivan cijelo vrijeme, već u određenim periodima (*heartbeats*) analizira rad sustava, te odlučuje je li potrebno poduzeti neke akcije. [1]

Kako bi mogao koristiti *Hadoop*, *Apex* na svojoj najnižoj razini posjeduje STRAM. STRAM (*Streaming Application Manager*) je kontroler za *Hadoop* YARN. Kada se aplikacija pokrene, STRAM je prvi proces koji se aktivira. Zadaća procesa STRAM je nadziranje izvođenja aplikacije, alociranje potrebnih resursa, raspoređivanje poslova po čvorovima grozda kako bi se što više operacija odvijalo paralelno na odvojenim komponentama, inicijalizacija operatora, vođenje statistike i različitih bilježenja podataka za vrijeme izvođenja, osiguravanje otpornosti na greške, dinamičko raspoređivanje poslova i briga o sigurnosti sustava. [1]

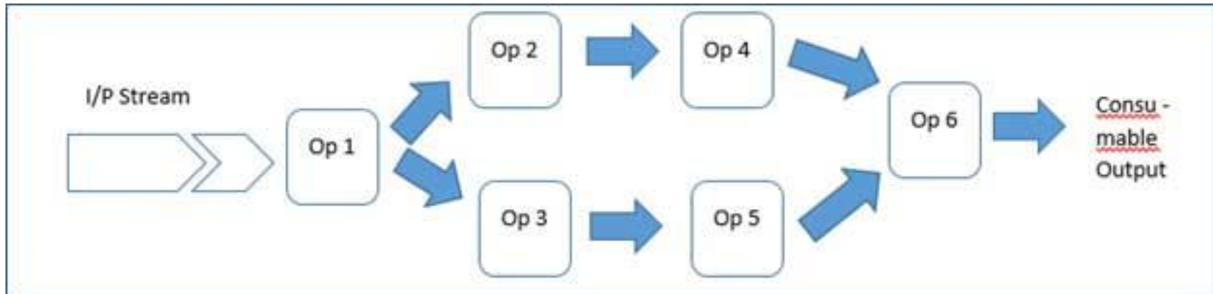
1.1. Izvođenje aplikacija na platformi *Apache Apex*

Aplikacije u *Apache Apexu* grafički prikazujemo kao usmjereni aciklički graf (*Data Aycilic Graph*). Vrhove grafa zovemo operatorima, a stranice grafa zovemo tokovima podataka. Operatori sadrže programsku logiku aplikacije, a stranice predstavljaju putove kroz koje podaci putuju od operatora pošiljatelja do operatora primatelja. Podatak koji putuje kroz tokove naziva se *tuple*. Obrada podataka odvija se u memoriji, a ukoliko je potrebno, rezultate je moguće spremiti na neki medij (HDFS). [1]

Ključan koncept izgradnje aplikacija temeljenih na tokovima je podjela podataka u manje blokove podataka koje nazivamo pomičnim prozorima (*streaming window*). *Streaming* aplikacije su zapravo *batch* aplikacije, samo što su periodi obrade podataka vrlo mali (uobičajeno taj period iznosi 500 ms). Takav koncept naziva se *micro-batching*, a razlog korištenja ovakvog principa je optimizacija zapisa o proteklim operacijama nad *tuplovima*. Kada se ovaj princip ne bi koristio, sustav bi morao zapisivati svaku operaciju provedenu

nad svakim *tupleom*, što bi narušilo propusnost, povećalo vrijeme potrebno da se platforma oporavi nakon pogreške te smanjilo skalabilnost. [1]

Uz prozor postoji i aplikacijski prozor koji je skup više običnih prozora koji dolaze jedan za drugim. Aplikacijski prozori koriste se za agregaciju podataka na razini operatora.



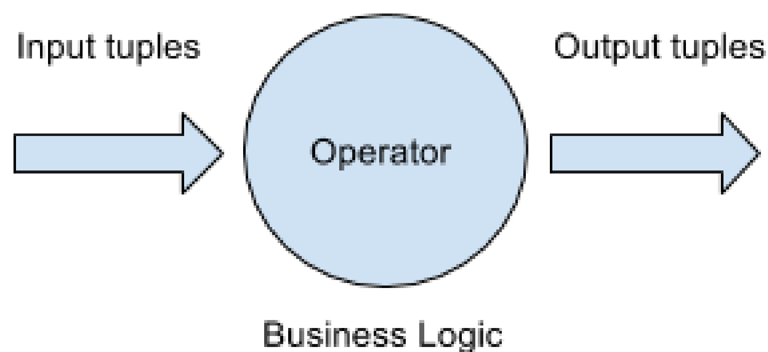
Slika 4 Primjer usmjerenog acikličkog grafa[6]

Kolekcija operatora *Apex Malhar* sadrži veliki broj već gotovih operatora spremnih za korištenje, no ako operator za traženu funkciju ne postoji, moguće je napisati i novi operator. Tok izvođenja aplikacije u *Apache Apexu* može se usporediti s principom ulančavanja operacija koji koriste moderni procesori (*pipelining*), čime se ostvaruje mogućnost paralelizacije operacija nad *tuplovima*. [1]

2. Osnovni elementi platforme *Apache Apex*

2.1. Operatori

Operatori su osnovne jedinice za obradu podataka. U operator ulaze *tuplovi*, obrađuju se i zatim šalju dalje na obradu ostalim operatorima. Koncept operatora uveden je kako bi se iskoristila prednost raspodijeljenog sustava. Svaki operator zamišljen je kao logička jedinica koja je potpuno nezavisna o drugim operatorima i koja asinkrono obrađuje *tuplove*. [1]



Slika 5 Grafički prikaz operatora[1]

U teoriji, jedan operator dovoljan je da bi se ostvarila cijela funkcionalnost aplikacije, no to nije poželjno rješenje. Operatori bi trebali biti oblikovani prema načelu jedinstvene odgovornosti, što znači da bi svaki operator trebao bit zadužen za točno određen zadatak. Na taj način potiče se ponovna upotreba operatora, a određeni operator lako je dodijeliti određenom čvoru u raspodijeljenom sustavu, neovisno o ukupnom broju čvorova. Svaki operator ima svoje jedinstveno ime i identifikacijski broj. [1]

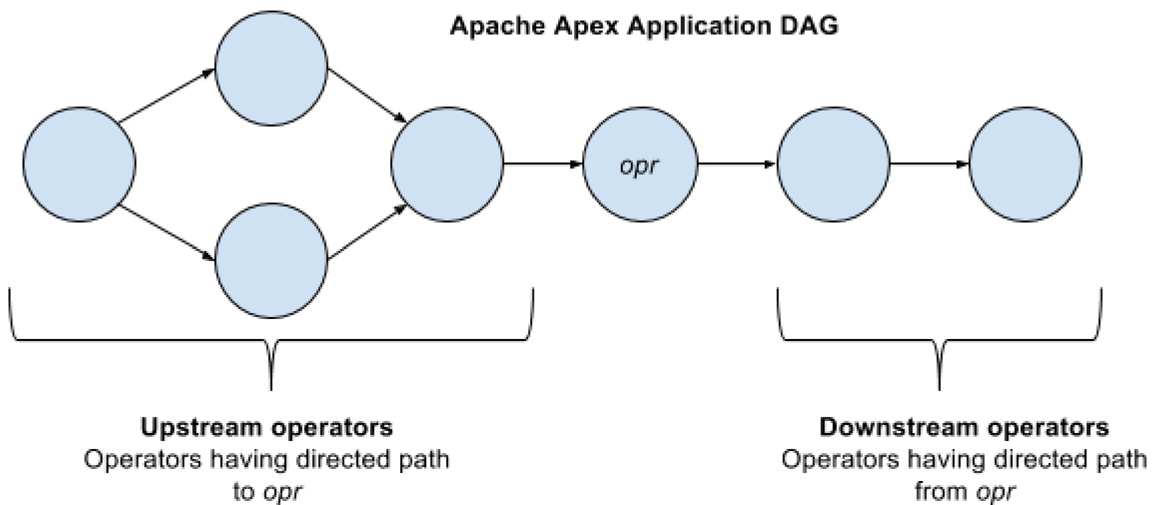
Postoje tri vrste operatora:

- Ulazni operatori (ulazni adapter) – čine početnu točku aplikacije, tj. prvi vrh u logičkoj shemi grafa aplikacije, uobičajeno generira ili učitava podatke s vanjskog izvora

- Generički operator – operator koji prihvaća ulazne *tuplove*, obrađuje ih te šalje *tuplove* operatorima koji slijede nakon njega
- Izlazni operatori (izlazni adapteri) – čine završnu točku aplikacije, tj. zadnji vrh u grafu, zapisuje podatke na neko vanjsko odredište

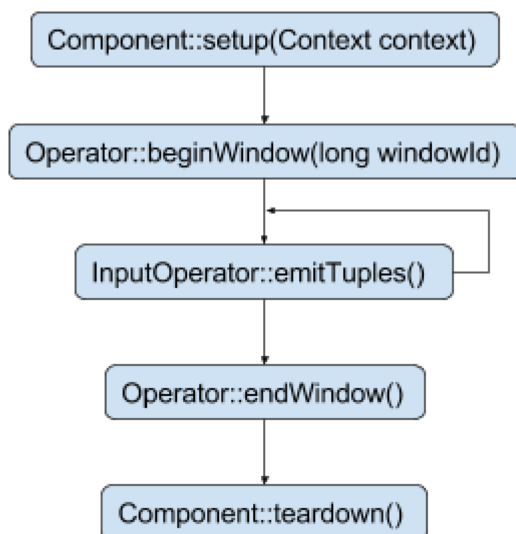
S obzirom na položaj operatora u ovisnosti o položaju jednog određenog operatora, definiramo 2 skupine operatora:

- Operatori prethodnici (*upstream operators*) – skup operatora koji se nalaze prije određenog operatora, tj. od kojih postoji usmjerena veza prema određenom operatoru
- Operatori sljedbenici (*downstream operators*) - skup operatora koji se nalaze nakon određenog operatora, tj. postoji usmjerena veza od određenog operatora prema skupu operatora

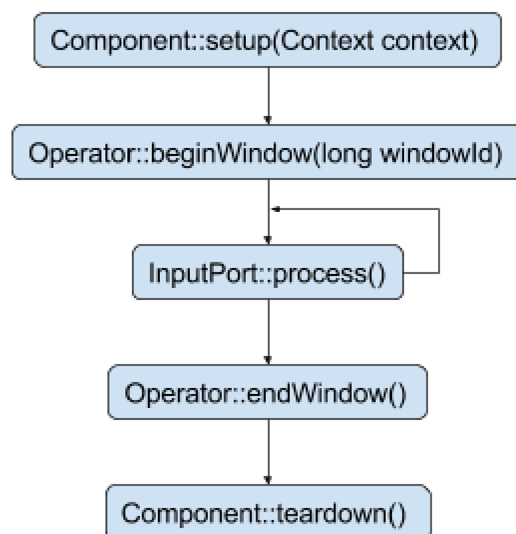


Slika 6 Operator i njegovi *upstream* i *downstream* operatori[1]

Od stvaranja do uništenja svaki operator prolazi kroz određene faze. Za prelazak operatora u novu fazu brine STRAM. Svaka faza započinje pozivom određene metode operatora.



Slika 7 Dijagram toka ulaznog adaptera[1]



Slika 8 Dijagram toka generičkog operatora i izlaznog operatora[1]

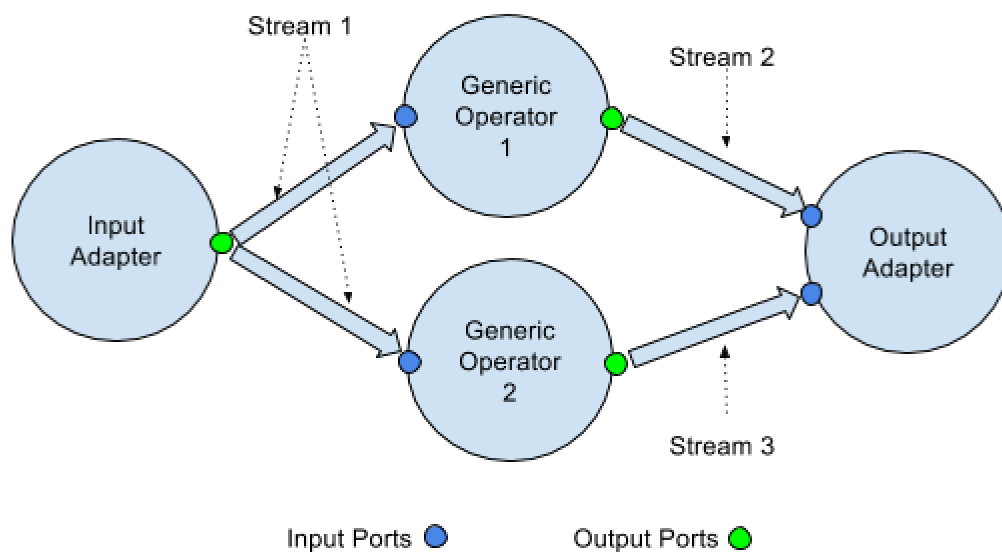
Metode u životnom ciklusu operatora:

- *setup*() – inicijalizacija operatora i priprema za početak obrade *tuplova*
- *beginWindow*() – označava početak aplikacijskog prozora
- *process*() – obrada pojedinog *tupla* koji je u operator stigao preko ulaznih vrata, svaka ulazna vrata imaju metodu *process*() u kojoj se pristigli operatori obrađuju, specifična metoda za generičke operatora i izlazne adaptere
- *endWindow*() – označava kraj aplikacijskog prozora
- *teardown*() – operator se uništava i oslobađa se sva memorija koju je operator koristio

Biblioteka *Malhar* sadrži veliki broj gotovih operatora spremnih za korištenje, ali programeri mogu, ako im je potrebna dodatna funkcionalnost, postojećem operatoru dodati nova svojstva ili napisati potpuno novi operator. Svaki stvoreni operator može se dodati već postojećim operatorima sadržanim u *Malharu*. [1]

2.2. Vrata

Svaki operator *Apache Apexa* sadrži određni broj vrata (*port*) kojim se povezuje s ostalim operatorima. Vrata se dijele na ulazna vrata i izlazna vrata. Kroz skup ulaznih vrata operator dobiva *tuplove*, a kroz skup izlaznih vrata operator šalje *tuplove* operatorima koji slijede nakon njega. Uobičajeno je da generički operator sadrži jedna ili više ulazna i jedna ili više izlazna vrata, ulazni adapter sadrži samo izlazna vrata, a izlazni adapter sadrži samo ulazna vrata. Ako ulazni adapter prima podatke iz više izvora može istovremeno posjedovati i ulazni skup vrata[1]



Slika 9 Prikaz ulaznih i izlaznih vrata svih tipova operatora[1]

2.3. Tokovi podataka

Tokovi podataka predstavljaju put kroz koji se *tuplovi* šalju od izlaznih vrata jednog operatora do ulaznih vrata nekog drugog operatora. Kroz jedna izlazna vrata *tuplove* je moguće poslati na ulazna vrata jednog ili više operatora, ali više izlaznih operatora ne mogu slati *tuplove* kroz tok.

Kroz tok podataka *tuplovi* se šalju u redoslijedu u kojem ih je operator propustio kroz svoja izlazna vrata. *Tuplovi* se ne šalju stalno, već se u određenim vremenskim trenutcima više *tuplova* emitira na izlazna vrata. Za svaki *tuple* koji jedan operator emitira za vrijeme jednom tokovnog prozora bilježi se da taj *tuple* pripada tom tokovnom prozoru. Na taj

način, ako je potrebno ponovno reproducirati prozor, svi *tuplovi* biti će reproducirani redoslijedom kojim su izvorno bili poslani. Na taj način *Apache Apex* garantira da će redoslijed *tuplova* u toku biti očuvan u bilo kojem slučaju.[1]

Tokovima podataka možemo dodijeliti dodatne atribute koji služe kao naputci STRAM.

Tokovima podataka imaju 4 načina rada:

- THREAD_LOCAL (IN-LINE) – tokovi između operatora u jednoj dretvi, ovaj način rada moguće je koristiti samo ako operator koji slijedi ima samo jedna ulazna vrata
- CONTAINER_LOCAL (IN-CONTAINER) – tokovi između operatora u jednom spremniku, tj. u jednom procesu (*inter-process*)
- NODE_LOCAL (IN-NODE) – tokovi između operatora u jednom čvora *Hadoop* arhitekture, ali ne nužno u jednom procesu (*intra-process*)
- RACK_LOCAL (IN-RACK) – tokovi između operatora u istom *rack*-u raspodijeljenog sustava.
- Nedefinirani način rada – tokovi između operatora za koje ne znamo gdje su

Načine rada je moguće nadjačati kada je to potrebno (npr. kada ne postoji dovoljno *containera*, ili ih nemamo uopće). [1]

Poslužitelj za priručnu pohranu je mehanizam *Apache Apexa* za prijenos podataka između operatora koji nisu na istom čvoru raspodijeljenog sustava. Podatci koje operator emitira prvo se spremaju na poslužitelj za priručnu pohranu koji se nalazi u istom čvoru kao i operator koji šalje podatke.[8] Operator sljedbenik se pretplaćuje na poslužitelj operatora koji šalje podatke.[11]

In-line i *in-container* tokovi ne koriste poslužitelj za priručnu pohranu prilikom izvođenja aplikacije.

Apex garantira da će *tuplovi* poslani u istom tokovnom prozoru stići na odredište u nepromijenjenom poretku. Jedan tokovni prozor vezan je samo za jedan tok, pa nažalost aplikacija ne može garantirati da će *tuplovi* iz dva ili više toka stići u operator u nepromijenjenom poretku. Konceptualno rješenje tog problema je da operatori ne ovise o *tuplovima* koji dolaze kroz više ulaznih vrata. U situacijama kada je to nužno potrebno,

Apex će prvo pričekati da se obrade svi *tuplovi* koju trebaju doći u operator, a rezultat emitira prilikom poziva *endWindow*() metode. [1]

2.4. Paketi

Apache Apex aplikacijski paket je skup svih datoteka potrebnih za izvođenje aplikacije. Postoje četiri vršne datoteke u svakom aplikacijskom paketu: *app*, *lib*, *conf*, *META-INF* i *resources*.

- *app* sadrži *jar* datoteke izvornog koda aplikacije
- *lib* sadrži sve *jar* datoteke o kojima aplikacija ovisi.
- *conf* sadrži konfiguracijske datoteke u XML formatu.
- *META-INF* sadrži *properties.xml* datoteku
- *resources* sadrži bilo koju drugu datoteku.

Svaki projekt u *Apache Apexu*, osim Java koda, sadrži još dvije datoteke, *pom.xml* i *properties.xml*.

Datoteka *pom.xml* služi za određivanje *Maven* paketa o kojima aplikacija ovisi. Za svaku *Apache Apex* aplikaciju najvažnije su 3 ovisnosti, *malhar-library*, *apex engine* i *junit*.

Datoteka *properties.xml* služi za definiranje posebnih konfiguracijskih parametara. Unutar nje možemo definirati konfiguracijske parametre za svojstva i atribute aplikacije, pojedinih operatora, pojedinih vrata i tokova. [1]

2.5. Svojstva i atributi

Svojstva omogućavaju prilagođavanje funkcionalnih svojstava operatora. Definiraju se kao ne-tranzijentni objekti koji se instanciraju unutar operatora. Iako su članske varijable operatora tranzijente, vrlo je važno da operator pamti svoja svojstva kako bi se operator mogao ponovno instancirati ako dođe do kvara u sustavu. Operator može, ali i ne mora imati definirana svojstva.

Atributi određuju kako će se aplikacija ponašati za vrijeme izvođenja, no ne utječu na funkcionalnost aplikacije. Korisnici ne mogu dodavati nove atribute, nego samo koristiti već postojeće.

Atribute možemo dodijeliti na razini aplikacije, ali i svakom operatoru zasebno. Atributima na razini aplikacije moguće je odrediti ime aplikacije, postaviti veličinu prozora toka i slično, a atributima na razini operatora moguće je odrediti broj particija i slično.

Razdvajanje svojstava od atributa je korisno u situacijama kada određenu aplikaciju želimo prenijeti na neki drugi raspodijeljeni sustav. U tom slučaju nije potrebno mijenjati funkcionalne postavke operatora definiranih preko svojstava, nego je dovoljno samo promijeniti određene atribute. [1]

2.6. Kontrolni *tuplovi*

Kako bi operatori znali kada prijeći iz jednog stanja u drugo, platforma definira kontrolne *tuplove*. Najvažniji predefinirani kontrolni *tuplovi* su BEGIN_WINDOW i END_WINDOW. Kontrolni *tuple* služi kako bi se pokrenula neka posebna akcija u operatorima sljedbenicima.

Kontrolni *tuplovi* uglavnom se koriste u slijedećim slučajevima:

- *Batch* potpora – dojava operatorima kada počinje, a kada završava tokovni prozor
- *Watermark* – kontrolni *tuple* koji služi za identifikaciju zakašnjelih prozora
- Promjena svojstava operatora – promjena svojstava operatora za vrijeme izvođenja aplikacije
- Snimanje *tuplova*

Kako bi operator mogao primati i emitirati kontrolne tuplove definirana su posebna vrsta ulaznih i izlaznih vrata, *ControlAwareDefaultInputPort* i *ControlAwareDefaultOutputPort*. Ulazna vrata kroz koja mogu biti poslani kontrolni *tuplovi*, uz uobičajenu metodu *process*(), imaju i metodu *processControl*(). [1]

2.7. Pravila afiniteta

Jedna od mogućnosti korištenja atributa je specificiranje pravila afiniteta. Pravila afiniteta su naputci kako bi operatori trebali biti raspoređeni po čvorovima raspodijeljenog sustava.

Postoje dvije vrste pravila afiniteta, pravila i anti-pravila. Pravila služe kao naputak koji operatori trebaju biti alocirani zajedno, dok anti-pravila govore koji operatori ne bi smjeli biti alocirani zajedno.

Drugi parametar pravila afiniteta je način rada (lokalnost). Da bi operatorima mogla biti dodijeljena pravila afiniteta, zasad oni moraju bit povezani tokovima čiji je način rada IN_LINE, IN_NODE ili IN_CONTAINER. IN_RACK još uvijek nije implementiran.

Pravila afiniteta mogu se definirati i u dokumentu *properties.xml* u JSON formatu. [1]

2.8. Načini rada

Apache Apex aplikacija može se izvoditi u 3 načina rada: lokalni način, rad na grozdu s jednim čvorom i rad na grozdu s više čvorova.

Lokalni način rada predstavlja okruženje u kojem aplikacija radi kao jedan proces koji sadrži više dretvi. U lokalnom načinu rada nema ovisnosti o *Hadoop* grozdu i *Hadoop* servisima, a umjesto HDFS-a aplikacija koristi datotečni sustav platforme na kojoj se izvodi. Lokalni način rada koristi se za ispravljanje logičkih grešaka u aplikaciji. Mana rada aplikacije u lokalnom načinu je manja količina dostupnih resursa, što dovodi do manje skalabilnosti i pojave „uskih grla" u aplikaciji.

Način rada u jednočvornom grozdu donekle je sličan radu u lokalnom načinu. Svi potrebni elementi (upravljač resursa, *NameNode*, *DataNode* i upravljač čvorova) smješteni su na istom računalu, no svaki element zauzima jedan zaseban proces, a ne jednu dretvu dijeljenog procesa. Način rada u jednočvornom grozdu pogodan je za poboljšanje razine optimizacije, opću analizu performansi i otkrivanje problema s integracijom na *Hadoopu*. Kada se aplikacija izvodi na *Hadoop* grozdu, ona ovisi o Hadoop okruženju, pa na vidjelo izlaze problemi koje ne možemo otkriti u lokalnom načinu rada. Prednost rada u jednočvornom grozdu je mogućnost iskorištavanja višeprocesne arhitekture računala, ali broj čvorova koje aplikacija može koristiti puno je manji nego u načinu rada na grozdu s više čvorova.

Prilikom rada na *Hadoop* grozdu s više čvorova elementi aplikacije raspodijeljeni su kroz čvorove grozda. Uobičajeno je na odvojenom čvoru pokrenut proces STRAM. [1]

2.9. Validiranje aplikacije

Provjera ispravnosti aplikacije u *Apache Apexu* provodi se u 3 faze :

1) Za vrijeme prevođenja
2) Prilikom inicijalizacije
3) Za vrijeme izvođenja

Provjeru ispravnosti prilikom prevođenja obavlja Java prevoditelj. To je prva faza validacije koja je za programera najpovoljnija. Provjera prilikom prevođenja uključuje:

- Validaciju shema u tokovima – provjera da li sva vrata na nekom toku emitiraju, tj. primaju istu vrstu *tuplova.*
- Provjeru ispravnosti vrata – provjera postoji li u toku samo jedna izlaznih vrata i barem jedna ulazna vrata
- Imenovanje – provjera jesu li svi operatori i tokovi ispravno imenovani

Provjera ispravnosti prilikom inicijalizacije uglavnom se odnosi na postavke konfiguracija aplikacije. Neke od provjera uključuju validaciju pomoću *JavaBeansa*, provjeru da li je na jedna vrata priključen samo jedan tok podataka, ispravnost aplikacijskog prozora i drugo.

Posljednji korak u validaciji aplikacije je provjera ispravnosti za vrijeme izvođenja. U ovoj fazi provjeravaju se pogreške do kojih dolazi prilikom rada sa stvarnim podatcima, kao što su iznimke zbog logičkih pogrešaka u aplikaciji. [1]

3. Otpornost na pogreške i mehanizmi oporavka

U slučaju neočekivanog događaja (na primjer kvar čvora raspodijeljenog sustava) čija je posljedica nedostupnost operatora (ili više njih), aplikacija se mora moći prilagoditi novonastaloj situaciji. Mehanizam za oporavak od pogreške platforme *Apache Apex* uzima tokovne prozore kao atomarne vrijednosti i definira tri načina oporavka od pogreške:

- procesiranje prozora najmanje jednom – svi prozori obrađuju se najmanje jednom, te nema gubitka podataka, podrazumijevani način
- procesiranje prozora najviše jednom – svi prozori obrađuju se najviše jednom, gubitak podataka je moguć, najefikasniji mehanizam
- procesiranje prozora točno jednom – prozori se obrađuju samo jednom, smatra se da neće biti gubitka podataka, najneefikasniji mehanizam [1]

Svaki operator ima ograničenu količinu resursa koji može koristiti, a razlog tome su ograničenja na resurse koja nameće *Apache Hadoop*. Ta ograničenja odnose se na propusnost, vrijeme kašnjenja i agregirano korištenje resursa u čvoru. Kada operator dosegne gornju granicu resursa koje smije koristiti, a potrebno mu je još resursa do kojih ne može doći, operator postaje usko grlo aplikacije. Problem uskih grla rješava mehanizam podjele operatora u particije. [1]

3.1. Dijeljenje operatora, skalabilnost

U situaciji kada operator ima dovoljno resursa za normalan rad *tuplovi* dolaze u operator, te se nakon obrade prosljeđuju na izlazna vrata. Kada zahtjevi porastu i operator postane usko grlo, *Apache Apex* dijeli operator na particije. Postoje dva načina podjele operatora u particije: raspodjela opterećenja po instanci operatora i ljepljivi ključ. [1]

Prilikom korištenja podjele operatora po istanci operatora *Apache Apex* stvara više particija (fizičkih instanci) operatora, razdjeljuje ulazni tok podataka u operator na više dijelova i svakoj particiji dodjeljuje jedan dio izvornog ulaznog toka. Koji podatak (*tuple*) odlazi u koji operator određuje se algoritmom *round-robin* ili sličnim. Ovakav način obrade nesvjestan je podataka s kojima radi.

Ljepljivi ključ podrazumijeva da će *tuple* s određenim ključem uvijek obrađivati određeni operator, bez obzira koliko puta taj *tuple* prošao kroz tok podataka. Ljepljivost operatora i *tuplova* se nastavlja čak i kada broj particija raste velikom brzinom (dinamički). Ovakav način obrade je svjestan s kojim podacima radi.

Na particije također možemo primijeniti pravila i anti-pravila afiniteta.

3.2. Statička podjela operatora

Statička podjela operatora odnosi se na unaprijed definiran broj particija. Nakon što je logički graf aplikacije dizajniran, STRAM pretvara logički graf u fizički graf. Ideja statičke podjele operatora je da se operatori podijele samo jednom. [1]

3.3. Dinamička podjela

U aplikacijama za obradu toka podataka opterećenje na aplikaciju se dinamički mijenja kroz vrijeme. Povećanje opterećenja može dovesti do povećanog kašnjenja, dužeg vremena izvođenja i slično, dok premalo opterećenje dovodi do bespotrebnog zauzimanja resursa na sustavu. Dinamičku prilagodbu aplikacije na promjenjive uvjete ostvarujemo dinamičkom podjelom operatora. Kako bismo omogućili operatoru da se dinamički konfigurira, potrebno mu je predati novi objekt koji implementira sučelje `Partitioner`, ili postaviti podrazumijevani `Partitioner`. [1]

Podrazumijevani `Partitioner` podržava podjelu operatora pomoću metode ljepljivog ključa. Uobičajeno operator koji radi s podrazumijevani `Partitioner` ima samo jedna ulazna vrata. Razlog tome je što podrazumijevani `Partitioner` dijeli samo prva vrata u operatoru, dok ostala vrata ostaju nepodijeljena. [1]

3.4. Kontrolne točke

Mehanizam koji tokovne prozore tretira kao mikro *batcheve* daje nam mogućnost nakon određenog broja obrađenih tokovnih prozora napravimo kontrolnu točku. Kontrolne točke su vrlo važne prilikom oporavka od pogreške. Za vrijeme kontrolne točke cijeli objekt operatora zapisuje se na HDFS, a poslužitelj za priručnu pohranu čvora na kojem se nalazi operator se prazni. Nakon kontrolne točke u poslužitelj za priručnu pohranu će se upisivati svi *tupovi* koje operator obradi dok slijedeće kontrolne točke. [1]

Prilikom inicijalizacije STRAM predaje inicijalne parametre tokovnom procesu (*StreamingContainer*). Na kraju kontrolnog intervala šalje se kontrolni *tuple* koji prolazi kroz tokove i aktivira svaki tokovni proces na koji naiđe. Takav mehanizam omogućava da svi operatori naprave kontrolnu točku u istom vremenskom ograničenju. Jedino kašnjenje uzrokuje vrijeme potrebno da kontrolni *tuple* stigne do svih tokovnih procesa. Kontrolni *tuple* neće aktivirati procese koji imaju različit kontrolni interval. Stvaranje kontrolne točke izvodi se između poziva metoda *endWindow*() i *beginWindow*(). U nekom trenutku dovoljno je imati samo jednu kontrolnu točku. Kada kontrolni *tuple* obiđe cijeli graf, u sustavu se kao kontrolnu točku postavlja ID prozora u kojem je poslan kontrolni *tuple*, a stara vrijednost se briše jer više nije potrebna.

Ako operator koristi aplikacijski prozor, potrebno je podesiti operator da se zapisivanje kontrolne točke radi na kraju aplikacijskog prozora, a ne tokovnog prozora. To znači da je potrebno namjestiti da je broj prozora nakon kojih se radi kontrolna točka jednak aplikacijskom prozoru. Za vrijeme zapisivanja kontrolne točke operator se na kratko pauzira. [1]

3.5. Uloga poslužitelja za priručnu pohranu

Ako operatori razmjenjuju *tuplove* preko poslužitelja za priručnu pohranu i dođe do kvara jednog od operatora, uzvodni operator u svom poslužitelju sadrži *tuplove* koje je pokvareni operator trebao obraditi. Nakon što se operator oporavi potrebno je samo da se *tuplovi* u poslužitelju ponovno pošalju oporavljenom operatoru na obradu. Ako operator nema svoj uzvodni operator (npr. ulazni adapter), tada je on sam odgovoran za reproduciranje toka podataka, a to je moguće ostvariti na dva načina. Prvi način je da ulazni adapter koristi vanjski sustav za reprodukciju toka, a drugi je da čuva podatke u sebi između dvije kontrolne točke. Ako dođe do kvara poslužitelja za priručnu pohranu, svi operatori u procesu (*containeru*) koji je koristio poslužitelj i svi nizvodni operatori moraju se resetirati na stanje zabilježeno u zadnjoj kontrolnoj točki. [7]

3.6. Stanje operatora

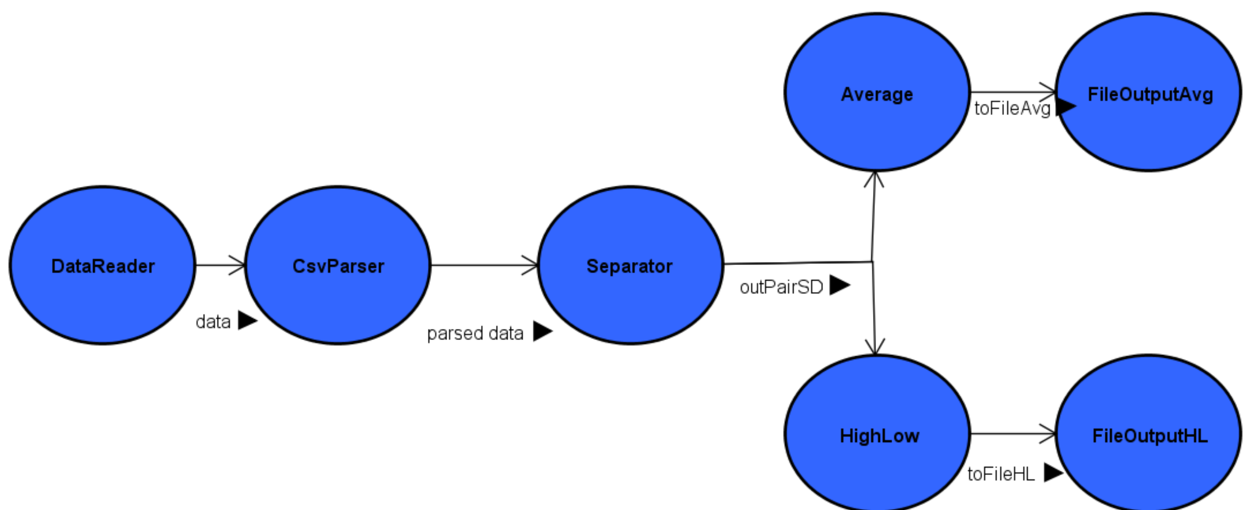
Stanje operatora definiramo kao skup podataka koji se prenosi iz jednog tokovnog prozora u slijedeći. Operator može imati stanje (*statefull*) ili biti bez stanja (*stateless*). Operator sa stanjem zahtijeva očuvanje vrijednosti jedne ili više varijabli na kraju tokovnog prozora. Sačuvani podatci potrebni su za obavljanje operacija nad *tuplovima* koji tek trebaju doći. Primjer operatora sa stanjem je operator koji računa sumu pristiglih podataka kroz više tokovnih prozora.

Operator bez stanja ne zahtijeva čuvanje vrijednosti varijabli između više tokovnih prozora. Primjer operatora bez stanja je `CsvParser`, koji prima tekst u CSV formatu, i pretvara ga u POJO objekt. Nakon pretvorbe i odašiljanja novostvorenog POJO objekta, operator ne mora znati ništa o obrađenom tekstu i stvorenom objektu. [1]

Operatori bez stanja otporniji su na greške te se brže i lakše mogu rekonstruirati. Dovoljno je stvoriti novi operator (na novom čvoru, ako je došlo do kvara čvora) i ponovno poslati spremljene prozore novostvorenom operatoru. [1]

4. Praktična implementacija aplikacije za obradu tokova podataka

Primjer praktične implementacije aplikacije biti će objašnjen kroz aplikaciju za obradu tokova podataka temperaturnih senzora.



Slika 10 Grafički prikaz aplikacije za obradu tokova senzorskih podataka

Aplikacija će primati podatke od senzora za temperaturu s različitih senzorskih lokacija u CSV formatu i pretvoriti ih u POJO (*Plain Old Java Object*) razreda `Mjerenje`. Izvor podataka bit će datoteka s CSV zapisima senzorskih mjerenja.

Kao izvor podataka bit će korištena CSV datoteka s raznim senzorskim mjerenjima. Najbitnije članske varijable razreda `Mjerenje` su identifikacijska oznaka senzorske lokacije, tj. mjesto gdje se nalazi senzor za temperaturu, vrijeme kada je obavljeno mjerenje i vrijednost temperature očitane sa senzora. Za potrebe aplikacije biti će dovoljni podaci o identifikaciji senzorske lokacije, nekoj očitanoj temperaturi (na primjer temperature na površini mora), te datumu i vremenu kada je obavljeno mjerenje. Odabrane podatke spremat će se u objekt razreda `Mjerenje`. Primjer jednog zapisa u datoteci izgleda ovako:

-18.190001,27.440001,23,13611,2018-06-12T09:28:00.000Z,BUOY,,,27.44,-18.19,0,0,,,,,,,,,,,,,,,,69,,,,,,,,,,,,,,1,2018-06-12T09:50:00.000Z,2018-06-

12T08:20:00.000Z,2018-06-12T09:50:00.000Z,2018-06-12T12:50:00.000Z,-180,2018-06-12T12:50:00.000Z,-180.

Nakon učitavanja podaci se trebaju pretvoriti u POJO, a za služi operator `CsvParser`. Da bi `CsvParser` uspješno pretvarao CSV zapise u objekte potrebno je dati shemu koja služi kao predložak za pretvorbu. Shemu definiramo u zasebnoj JSON datoteci.

Primjer jednog objekta razreda `Mjerenje` s najbitnijim (ne svim) članskim varijablama u JSON formatu je:

```
{
    "locid": "13611",
    "sst1_f": 80,
    "starttime": "2018-06-12T12:50:00.000Z"
}
```

Novostvoreni objekt razreda `Mjerenje` sadrži dosta podataka koji nam nisu potrebni (zapisi iz ostalih senzora na senzorskoj lokaciji), stoga je potrebno iz svakog objekta izvući samo one podatke koji nam trebaju. Za to je stvoren razred `MjerenjeSeparator` s jednim ulaznim vratima i pet izlaznih vrata.

```
public class MjerenjeSeparator extends BaseOperator {

        public final transient DefaultInputPort<Object>
pojoIn = new DefaultInputPort<Object>() {
            @Override
            public void process(Object tuple) {
                if (tuple instanceof Mjerenje) {
                    Mjerenje mjer = (Mjerenje) tuple;
                    outName.emit(mjer.locid.toString());
                    outVal.emit(mjer.sst1_f);
                    outTime.emit(mjer.starttime);
                    KeyValPair<String, Date> tupSDate = new
KeyValPair<>(mjer.locid.toString(), mjer.starttime);
                    outpairSDate.emit(tupSDate);
                    KeyValPair<String, Double> tupSD = new
KeyValPair<>(mjer.locid.toString(), mjer.sst1_f);
                    outpairSD.emit(tupSD);
                    KeyValPair<String, String> tupSS = new
```

```
KeyValPair<>(mjer.locid.toString(), mjer.provider);
                    outPairSS.emit(tupSS);
                }
            }
        };
        @OutputPortFieldAnnotation(optional = true)
        public final transient DefaultOutputPort<String>
outName = new DefaultOutputPort<>();

        @OutputPortFieldAnnotation(optional = true)
        public final transient DefaultOutputPort<Double>
outVal = new DefaultOutputPort<>();

        @OutputPortFieldAnnotation(optional = true)
        public final transient DefaultOutputPort<Date>
outTime = new DefaultOutputPort<>();

        @OutputPortFieldAnnotation(optional = true)
        public final transient
DefaultOutputPort<KeyValPair<String, Date>> outPairSDate =
new DefaultOutputPort<>();

        @OutputPortFieldAnnotation(optional = true)
        public final transient
DefaultOutputPort<KeyValPair<String, Double>> outPairSD = new
DefaultOutputPort<>();
```

Separator će kroz tri vrata emitirati članske varijable, a kroz ostala vrata emitirat će parove vrijednosti u obliku ključ, vrijednost. U trenutnoj verziji aplikacije biti će korištena samo izlazna vrata `outPairSD`, ali ostala vrata su implementirana kako bi se omogućila nadogradnja u budućnosti. Slijedeći korak je pronaći najveću i najmanju temperaturu za svaku postaju, a za to služi operator `RangeKeyVal` koji prima podatke s izlaznih vrata `outPairSD`. Za izračun srednje vrijednosti temperatura koristi se operator `SimpleMovingAverage` koji također prima podatke s izlaznih vrata `outPairSD`. `RangeKeyVal` i `SimpleMovingAverage` na ulazna vrata `data` primaju objekte parametrizirane klase `KeyValPair`<K, V>, gdje K predstavlja ključ, a V predstavlja vrijednost. V može biti bilo koja klasa koja nasljeđuje klasu `Number`. `RangeKeyVal` za svaki ključ pamti najmanju i najveću vrijednost pristiglu za vrijeme jednog aplikacijskog

prozora, te se prilikom poziva metode *endWindow*() vrijednosti emitiraju na izlazna vrata. SimpleMovingAverage funkcionira na sličan način, samo što prilikom poziva metode *endWindow*() računa aritmetičku sredinu svih pristiglih podataka, te se prilikom poziva metode *endWindow*() izračunata aritmetička sredina emitiranja na izlazna vrata. Nakon separacije i prikupljanja, dobivene rezultate potrebno je zapisati u datoteke, a za taj zadatak korišten je operator FileOutputOperator.

Svaka aplikacija mora imati razred koji implementira sučelje StreamingAplication. Za potrebe testiranja aplikacija će se izvoditi u lokalnom načinu rada.

```
public class csvParserApplication implements
StreamingApplication
{

  @Override
  public void populateDAG(DAG dag, Configuration conf)
  {


    FSRecordReaderModule input = dag.addModule("input",
FSRecordReaderModule.class);


    CsvParser parserOperator = dag.addOperator("csvParser",
new CsvParser());

parserOperator.setSchema(SchemaUtils.jarResourceFileToString(
"schema.json"));

    MjerenjeSeparator sep = dag.addOperator("mySeparator",
MjerenjeSeparator.class);


    RangeKeyVal rangeKeyVal = dag.addOperator("rangeKeyVal",
new RangeKeyVal<String, Double>());


    SimpleMovingAverage movingAverage =
SimpleMovingAverage<String, Double> oper =
```

```
dag.addOperator("average", new SimpleMovingAverage<String,
Double>());


    FileOutputOperator FileOutputAvg =
dag.addOperator("fileOutputAvg", new FileOutputOperator());
    FileOutputOperator FileOutputHL =
dag.addOperator("fileOutputHL", new FileOutputOperator());

    //streamovi
    dag.addStream("data", input.records, parserOperator.in);

    dag.addStream("parsedData", parserOperator.out,
sep.pojoIn);

    dag.addStream("sepStringDouble", sep.outpairSD,
movingAverage.data, rangeKeyVal.data);

dag.addStream("toFileAvg", movingAverage.doubleSMA,
FileOutputAvg.input);

dag.addStream("toFileHL", rangeKeyVal.range,
FileOutputHL.input);
```

Nakon obrade podataka zapis u datoteci s minimalnim i maksimalnim vrijednostima izgledao bi slično slijedećem zapisu:

13568=(75,22)

14509=(24,18)

15670=(90,50)

13568=(60,22)

14509=(30,16)

15670=(90,30)

Zapis u datoteci sa srednjom vrijednošću temperatura izgledao bi slično slijedećem zapisu:

13568=(50)

14509=(20)

15670=(70)

13568=(30)

14509=(25)

15670=(50)

U oba slučaja imamo ponavljanje identifikacijskih oznaka senzorskih lokacija, a razlog tome je što se podaci za svaku senzorsku lokaciju upisuju nakon svakog aplikacijskog prozora operatora `RangeKeyVal` i `SimpleMovingAverage.` Kada bismo u sustavu imali 10 senzorskih lokacija, i za vrijeme izvođenja aplikacije metoda *endWindow()* se pozove 10 puta za svaki od operatora koji podatke zapisuju u datoteku, u svakoj datoteci bilo bi zapisano 100 unosa.

Zaključak

Apache Apex pruža mogućnost brzog stvaranja aplikacija za obradu tokova podataka. Instanciranjem operatora i spajanjem operatora tokovima podataka možemo implementirati bilo koju aplikacijsku logiku na relativno jednostavan način. Korištenjem platforme *Apache Hadoop Apache Apex* eliminira potrebu za postojanjem snažnog računala bez kojeg ne bi bilo moguće obrađivati podatke u realnom vremenu, već koristi mnogo manjih računala koja surađuju. Dodatna prednost korištenja više računala je veća otpornost na pogreške i veća skalabilnost. *Apache Malhar* dodatno ubrzava izradu aplikacija svojom ponudom gotovih komponenata. U radu je prikazana aplikacija koja omogućuje pronalazak maksimalne, minimalne i srednje vrijednosti podataka koji se očitavaju na senzorima za temperaturu. Podaci se obrađuju u tokovima, pa su dobiveni rezultati uvijek ažurni. Ovakva aplikacija može se koristiti za promatranje promjena u temperaturi na nekom određenom području.

Popis slika

Slika 1 Batch obrada podataka [2] 2

Slika 2 Stream obrada podataka [2] 2

Slika 3 pregled komponenata Apache Apexa[6] 4

Slika 4 Primjer usmjerenog acikličkog grafa[6] 6

Slika 5 Grafički prikaz operatora[1] 7

Slika 6 Operator i njegovi *upstream* i *downstream* operatori[1] 8

Slika 7 Dijagram toka ulaznog adaptera[1] 9

Slika 8 Dijagram toka generičkog operatora i izlaznog operatora[1] 9

Slika 9 Prikaz ulaznih i izlaznih vrata svih tipova operatora[1] 10

Slika 10 Grafički prikaz aplikacije za obradu tokova senzorskih podataka 20

Literatura

[1] Apache, Apex
http://apex.apache.org
[2] Medium, Big data battle batch processing vs stream processing
https://medium.com
[3] Wikipedia, Apache Hadoop
https://en.wikipedia.org
[4] Wikipedia, Apache Apex
https://en.wikipedia.org
[5] Wikipedia, Batch processing
https://en.wikipedia.org
[6] Edupristine, Apache Apex intruduction
https://www.edupristine.com
[7] StackOverflow, How does apache apex handle back pressure
https://stackoverflow.com
[8] SlideShare, Intro to Apache Apex women in big data
https://www.slideshare.net
[9] Forbs, Big data 20 mind boggling facts everyone must read
https://www.forbes.com
[10] Wikipedia, Stream processing
https://en.wikipedia.org

Sažetak

Obrada toka senzorskih podataka korištenjem platforme *Apache Apex*

U radu je opisana platforma za obradu velike količine podataka Apache Apex. Prikazana je integracija s platformom Apache Hadoop, osnovni koncepti rada s platformom Apache Apex. Opisan je proces validiranja aplikacije, te kako se platforma Apache Apex nosi s ograničenim resursima i kako se oporavlja od pogrešaka. Uz opis platforme prikazan je proces izrade aplikacije za obradu tokova podataka s temperaturnih senzora. Aplikacija čita zapise iz datoteke i dinamički računa maksimalnu, minimalnu i srednju vrijednost pristiglih podataka. Dobiveni rezultati zapisuju se u pripadne datoteke.

Ključne riječi: *Java, Apache*, *Apex*, *Malhar*, CSV, *Hadoop*, Obrada tokova podataka, *Maven*, senzori temperature

Summary

Sensor Datastream Processing with Apache Apex Platform

In this paper is described *Apache Apex*, platform for big data processing. Reader can see how *Apache Apex* integrates with *Hadoop* and basic concepts of *Apache Apex*, as well as how Apache Apex handles situations when computer resources are limited and how Apex recovers from errors. Reader can also see the process of constructing an application for processing data from temperature sensors. Application reads CSV file with collected data, and dynamically calculates minimum, maximum and mean value of processed values. After processing, results are stored in associated files.

Key words: *Java, Apache*, *Apex*, *Malhar*, CSV, *Hadoop*, stream processing, *Maven*, temperature sensors