ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

ანა ჯაფარიძე

დიდი მონაცემები - ნაკადების დამუშავება რეალურ დროში

სამაგისტრო პროგრამა - ინფორმაციული ტექნოლოგიები

ნაშრომი შესრულებულია ინფორმაციული ტექნოლოგიების მაგისტრის
აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: მანანა ხაჩიძე

თბილისი
2019

ანოტაცია

მას შემდეგ, რაც ინტერნეტმა მთელი სამყარო ციფრულ სივრცეში დააკავშირა, ორგანიზაციებისათვის ღირებული მონაცემების მოცულობა ძალიან გაიზარდა და კვლავაც იზრდება. ამ მოცულობის ინფორმაციის შენახვასა და დამუშავებას არსებული ტექნოლოგიები მეტწილად ახერხებდნენ იქამდე, სანამ არ გამოჩნდნენ სოციალური ქსელები, საძიებო ქსელები და ელექტრონული კომერციები, რომელთაც, მოკლედ რომ ვთქვათ, მონაცემთა ბუმი მოჰყვა. მომხმარებლებელთა მოთხოვნილებების დასაკმაყოფილებლად, ინოვაციურობისათვის თუ კონკურენციის დასაძლევად, კომპანიებს სჭირდებოდათ უზარმაზარი მოცულობის მონაცემის შეგროვება უამრავი სხვადასხვა წყაროსაგან, გაწმენდა, ტრანსფორმაცია, გაანალიზება და სწრაფი გადაწყვეტილებების მიღება, რაც არსებული საშუალებებით იოლი არ იყო. აქ პრობლემა მხოლოდ ამ ინფორმაციის მოცულობა არ იყო. ამ ფაქტორთან ერთად, ამ დიდ მონაცემთა მთავარი გამოწვევა გახლდა მათი არასტრუქტურირებული ბუნებაც და წარმოქმნის სისწრაფეც.

2005 წლიდან ოფიციალურად დამკვიდრდა ტერმინი დიდი მონაცემები – Big Data. იმავე წელსვე შეიქმნა Hadoop-იც - დიდ მონაცემთა სამყაროს გული. მას შემდეგ ღია პროგრამული უზრუნველყოფების სივრცეში ძალიან აქტიურად მუშაობენ დიდ მონაცემების მიმართულებით. შეიქმნა უამრავი სპეციალიზირებული ტექნოლოგია და საშუალება, რომელთა წყალობითაც შესაძლებელია დიდ მონაცემთა დამუშავება, შენახვა, მართვა, ანალიზი და ა. შ. შედეგად, დღესდღეობით Big Data ეკოსისტემა ძალიან მრავალფეროვანია, არქიტექტურა კი – საკმაოდ კომპლექსური და მრავალი კომპონენტისგან შემდგარი.

წინამდებარე ნაშრომში ზოგადად არის მიმოხილული დიდ მონაცემები, მისი აქტუალობა, მთავარი კონცეფციები თუ პრინციპები, ძირითადი მახასიათებლები და არქიტექტურული თავისებურებები. საკვანძო ნაწილი კი ეხება Big Data სამყაროს ერთ-ერთ აქტუალურ მიმართულებას – მონაცემთა ნაკადების დამუშავება (თითქმის) რეალურ დროში და განხილულია მისი ორი ძირითადი არქიტექტურა. პრაქტიკული ნაშრომის სახით კი წარმოდგენილია მონაცემთა ნაკადების დამუშავებასთან დაკავშირებული კონკრეტული ამოცანის გადაჭრა პროგრამული უზრუნველყოფის კოდის სახით.

Big Data – Real Time Processing of Data Streams

Abstract

Since the Internet digitally connected the world, the amount of data, having high business value for many organizations, has been increasing tremendously. Existing technologies could, mostly, handle the increasing volume of data until the emergence of social media, search engines and e-commerce, which, briefly saying, caused so-called Data Boom. In order to meet customers' demand, to be innovative or to have a competitive advantage, businesses needed to gather the data from many sources, ingest, transform, analyze it and make quick decisions based on it. It was not an easy task, considering the limited capacities of available technologies. The volume of this data was not the only problem here. Along with this factor, the main challenge with processing of the large datasets was related to their various, not necessarily structured nature and velocity.

The term 'Big Data' was officially launched in 2005. Hadoop – the open-source heart of big data universe was created the same year. Since then the open source community has been actively working on and contributing to the Big Data. Numerous technologies and tools have been developed to process, store, manage, analyze large sets of data. As a result, today the Big Data ecosystem is very diverse and its architecture – quite complex.

The following paper overviews the nature of Big Data, its relevance, main concepts and principles along with its key architectural characteristics. The key part of the work has been devoted to one of the most popular topics in the Big Data world – (near) real-time processing of data streams. There are discussed two important architectures of it. As for the practical side of the work, a software solution to a specific task, related to real time data streams processing, has been presented.

შინაარსი

შესავალი

ზოგადი მიმოხილვა

ტერმინ ,,დიდი მონაცემების" სიახლის მიუხედავად, მისი კონცეფცია არც ისე ცოტა ხანია, რაც არსებობს. დღესდღეობით ორგანიზაციათა უმეტესობა აცნობიერებს, რომ თუ ისინი შეძლებენ, პირველ რიგში, ყველა იმ მონაცემს, რომელიც მათ გარშემო უხვად გენერირდება, თავი მოუყარონ და შემდგომში, ეს მონაცემები გაანაალიზონ, ძალიან დიდი სარგებელს მიიღებენ.

მართალია, მონაცემთა ანალიზის მნიშვნელობას კომპანიები ტერმინ ,,დიდ მონაცემთა'' გამოჩენამდე დიდი ხნით ადრეც კარგად ხვდებოდნენ, თუმცა, ამ სფეროს ახალი და მთავარი სარგებელი მასიური, სტრუქტურულად არაერთგვაროვანი მონაცემების სწრაფად და ეფექტურად დამუშავებაა. თუკი რამდენიმე წლის წინ ბიზნესი ჯერ აგროვებდა ინფორმაციას, შემდეგ აანალიზებდა მას და იღებდა ინფორმაციას, რომელზეც სამომავლო გადაწყვეტილებს დააფუძნებდა, დღეს Big Data ტექნოლოგიების გამოყენებით შეუძლიათ, ეს პროცესი უფრო დინამიური და მოქნილი გახადონ და გადაწყვეტილებები თითქმის დაუყოვნებლივ მიიღონ. ეს კი ისეთ კონკურენტულ უპირატესობას უჩენს მათ, რომელიც აქამდე არ ჰქონიათ.

დიდი მონაცემების განმარტება და 3V

მიუხედავად იმისა, რომ დღესდღეობით დიდი მონაცემები ძალიან პოპულარული მიმართულებაა, რეალურად მისი ერთი სრულყოფილად ჩამოყალიბებული და კონკრეტული განმარტება არ არსებობს, განსაკუთრებით, ვიტყოდი, რომ ტექნიკური თვალსაზრისით. 2011 წელს Gartner-ისა და McKinsey Global Institute-ის წარმომადგენლების მიერ გამოქვეყნებულ სტატიებსა და ნაშრომებში მსგავსად არის დიდი მონაცემები განმარტებული. კერძოდ, ავტორები ხსნიან, რომ ტერმინის ქვეშ იგულისხმება იმ მოცულობის მონაცემთა სიმრავლე, რომელიც შეგროვება, მართვა და დამუშავება ტიპიური, ტრადიციული მონაცემთა ბაზებისა და შესაბამის პროგრამების შესაძლებლობებს ბევრად აღემატება. [1]

თუმცა, დიდი მონაცემების განმარტება მხოლოდ მოცულობის ჭრილში არასრულყოფილია. მისი მახასიათებლების შედარებით უფრო ამომწურავად წარმოსაჩენად

მას ხშირად 3 V-ს სახით წარმოადგენენ ხოლმე. ესენია:

- **მოცულობა (Volume):** სახელიდანაც ცხადია, რომ დიდ მონაცემთა სიდიდეს, პირველ რიგში, სწორედ მისი მოცულობა განსაზღვრავს. უკანასკნელ წლებში მთელს მსოფლიოში მონაცემები ზომაში საოცრად გაიზარდა და ისევ სწრაფი ტემპებით განაგრძნობს ზრდას. გამოთვლილია, დღეში დაახლოებით 2.5 კვინტილიონი ბაიტი გენერირდება და ვარაუდობენ, რომ 2020 წლისათვის, 2005 წელთან შედარებით მონაცემთა მოცულობა 300-ჯერ იქნება გაზრდილი; [2]

- **სიჩქარე (Velocity):** სიჩქარე ეწოდება დასამუშავებელ მონაცემთა შემოდინების სიხშირეს. გარკეულ მონაცემთა შემოდინება ნაწილ-ნაწილ, პაკეტებად (batches) ხდება, თუმცა თანამედროვე სამყაროში ინფორმაციის ძალიან დიდი ნაწილი რეალურ დროშიც გენერირდება (მაგალითად, საკრედიტო ბარათების გატარებები, სოც. ქსელებში შეტყობინებები თუ მესიჯები, ლაივ კამერების ვიდეოები და ა. შ.); [2]

- **არაერთგვაროვნება (Variety):** მონაცემთა ტრადიციული ტიპები სტრუქტურირებულია, ფიქსირებული ფორმატისაა და ამიტომ კარგად თავსდებიან რელაციური მონაცემთა ბაზებში. თუმცაღა, უკანასკნელ წლებში იზრდება არასტრუქტურირებულ მონაცემთა რაოდენობაც, რომელთაც მეტა მოდელი (Meta Model) არ გააჩნია. ასეთებია, მაგალითად, ვებ გვერდები და მისი ლოგები, MRI სურათები, tweet-ები სოციალურ ქსელ Twitter-ზე და ა. შ. არასტრუქტურირებული მონაცემები Big Data-ს ფუნდამენტალური კონცეფციაა. ასეთი სახის მონაცემებს, სტრუქტურირებულთან შედარებით, არ გააჩნიათ მკაფიოდ განსაზღვრული წესების ერთობლიობა. მაგალითად, ტიპი Money ყოველთვის იქნება კონკრეტული რიცხვი მძიმის შემდეგ მინიმუმ ორი ციფრით, სახელები ტექსტური ტიპის იქნება და ა. შ. არასტრუქტურირებული მონაცემები კი (მაგალითად, სურათები, ხმოვანი ჩანაწერები ან სოციალურ ქსელების პოსტები შეიძლება სულ სხვადასხვანაირად იყოს წარმოდგენილი და ჩაწერილი. Big Data-ს ერთ-ერთი მიზანი კი სწორედ ისაა, რომ შესაბამისი ტექნოლოგიების გამოყენებით შეძლოს და არაერთგვაროვან

მონაცემებს, ასე ვთქვათ, აზრი შესძინოს. [2]

ზემოთხსენებულ სამ 3V-ს ხშირად ამატებენ მეოთხესაც:

- **სანდოობა (Veracity)** - მონაცემები, სამწუხაროდ, ყოველთვის სანდო არაა. ყველა კარგმა მენეჯერმა იცის, რომ შეგროვებულ ინფორმაციაში ყოველთვის იქნება გარკვეული შეუსაბამობები და თუნდაც ,,ხმაური''. ასე რომ, მონაცემთა დამუშავებისას საჭიროა ამ ფაქტორების გათვალისწინებაც და ,,ბინძური'' მონაცემების გაწმენდა ასეთი ხარვეზებისაგან. [2]

დიდ მონაცემთა გარემოს არქიტექტურის ზოგადი დახასიათება

მონაცემების მხოლოდ შეგროვება, იქნება ეს პატარა თუ დიდი მონაცემები, სრულიად უსარგებლო და ალბათ პირიქით, დანახარჯიც კიაა, თუკი მათ გარკვეული კონტექსტს არ მივუსადაგებთ და არ გამოვიყენებთ. ასე რომ, დიდ მონაცემებთანაც დაკავშირებით ბიზნესის მთავარი კითხვებია: როგორ დავამუშავებთ მას? რა ინფორმაციას მივიღებთ? როგორ გამოვიყენებთ ამ ინფორმაციას ბიზნესისთვის ღირებული გადაწყვეტილებების მისაღებად?

ზედა აბზაცებში დიდ მონაცემების აღწერის შემდეგ ალბათ გასაგებია, თუ რამდენად მდიდარი შეიძლება იყოს იგი კომპანიისათვის ღირებული ინფორმაციით. ეს ძალიან კარგია, მაგრამ სწორედ აქ ჩნდება მთავარი გამოწვევაც: რომელი ტექნოლოგიების და მათი როგორი გამოყენებით უნდა მოვაწყოთ ის გარემო, რომელიც დიდ მონაცემებს ,,ღირსეულად'' გაუმკლავდება და საბოლოოდ ბიზნესს მისცემს საშუალებას, მათგან ღირებული ინფორმაციის შექმნა ეფექტურად შეძლოს?

ვფიქრობ, ნათელია, რომ დიდი მონაცემები არ არის და ვერ იქნება ერთი რომელიმე კონკრეტული ტექნოლოგია ან გარემო (ე. წ. Framework-ი). ის წარმოადგენს იმ პროცესების, პროგრამული საშუალებების, ტექნოლოგიების და სისტემური ინფრასტრუქტურის ერთობლიობას, რომელიც ეხმარება ბიზნესს ზემოთაღწერილი მიზნების მიღწევაში. აქედან გამომდინარე, მისი არქიტექტურა, როგორც წესი, კომპლექსურია.

კიდევ ერთხელ აღვნიშნოთ, რომ დიდ მონაცემთა გარემო ისე უნდა იყოს მოწყობილი, რომ დიდი რაოდენობის განსხვავებული წყაროებიდან შემომავალი მონაცემების დამუშავება

შეძლოს სწრაფად და რესურსების ეფექტურად გამოყენებით. ამისათვის საჭიროა ბევრი არქიტექტურული კომპონენტის ქონა. ძირითადად ოთხ ლოგიკურ ფენას (Layer) გამოყოფენ. მოკლედ რომ დავახასიათოთ, ესენია:

1. დიდ მონაცემთა წყაროების ფენა (Big data sources layer):

დიდ მონაცემთა გარემოსთვის მონაცემები, ფაქტობრივად, ყველგანაა – კომპანიის შიგნითაც (ორგანიზაციის სერვერები, სენსორები, თუნდაც უკვე არსებული მონაცემთა საცავები, როგორიცაა, მაგალითად, Data Warehouse და ა. შ. უამრავი) და შეიძლება მის გარეთაც (მესამე მხარის მიერ შექმნილი მონაცემთა მრავალფეროვანი წყაროები). ასევე, წყარო შეიძლება ნაკადის (Data Stream) სახითაც იყოს. ასე რომ, დიდ მონაცემთა გარემოს უნდა შეეძლოს, როგორც მონაცემთა პაკეტების (Batches), ასევე მონაცემთა ნაკადების (Streams) დამუშავება რეალურ დროში. [3] [4]

2. მონაცემთა დამუშავების და შენახვის ფენა (Data massaging and storage layer):

ეს ფენა არის პასუხისმგებელი წყაროებიდან მონაცემების მიღებაზე. თუ საჭიროა, იგი აკონვერტირებს არასტრუქტურირებულ (ან ნახევრადსტრუქტურირებულ) მონაცემებს ისეთ ფორმატში, რომელსაც შემდეგ კონკრეტული ანალიტიკური პროგრამები აღიქვამენ და გამოიყენებენ. წყაროდან მიღებულ მონაცემებს, მეტადატის ვალიდაციისა და ფორმატის კონვერტაციის გარდა, შეიძლება დასჭირდეთ ,,ხმაურისაგან'' (noise) გაწმენდა და მხოლოდ რელევანტური მონაცემების დატოვება. ასეთი სამუშაოების შემდეგ მონაცემების შენახვა უკვე შეგვიძლია მათი კომპრესია (საჭიროების შემთხვევაში) და ჩვენს მონაცემთა საცავში შენახვა. ეს საცავი, როგორ წესი, HDFS-ია ხოლმე (Hadoop Distributed File System), თუმცა შეიძლება იყოს რომელიმე NoSQL ბაზაც ან ტრადიციული RDBMS-იც. [3] [4]

3. ანალიტიკური ფენა (Analysis layer):

ეს ფენა იმ პროგრამული საშუალებების ერთობლიობაა, რომლებიც უკავშირდებიან საცავში შენახულ მონაცემებს (როგორც სტრუქტურირებულ, ისე ნახევრად ან საერთოდ

არასტრუქტურირებულს) და შემდგომში ამუშავებენ მათ. ასეთი ანალიტიკური პროგრამები უამრავია, ზოგი – შედარებით მარტივი, ზოგიც კი ბევრად უფრო დახვეწილი და ძლიერი. [3] [4]

4. მოხმარების ფენა (Consumption layer):

ამ ფენაში წარმოდგენილია უკვე ანალიზის შედეგები ადამიანებისათვის გასაგები სახით. აქვე მოიაზრება მონაცემთა ვიზუალიზაციაც - უზარმაზარი მოცულობის მონაცემებმა შეიძლება ერთგვარად ,,ინფორმაციული გადატვირთვა'' (Information Overload) გამოიწვიოს. ამიტომ, მონაცემთა ანალიტიკოსებისა და მეცნიერებისათვის ძალიან გამოსადეგი იქნება დიდ მონაცემთა გარემოში ვიზუალიზაციის საშუალებების ქონა (მაგალითად, Tableau, QlikView და სხვები). [3] [4]

CAP თეორემა

Big Data სისტემები გვიქმნიან კომპიუტინგის იმხელა რესურსსა და ხელმისაწვდომობას (Availability), რომელიც, უბრალოდ, წარსულში არ იყო შესაძლებელი. ეს ყველაფერი განაწილებული სისტემების (Distributed Systems) წყალობით. თუმცა, ამ შესაძლებლობებთან ერთად, მათთან მუშაობა უფრო რთულიცაა და მოითხოვს ხოლმე გარკვეულ დათმობებზე (trade-offs) წასვლას. Big Data -სთან მუშაობის ერთ-ერთი მთავარი პრინციპი ისაა, რომ სწორი ტექნოლოგიები და სხვა გადაწყვეტილებები სწორედ ამოცანიდან გამომდინარე იქნეს მიღებული.

დიდ მონაცემებთან ეფექტური მუშაობისათვის ჩამოყალიბებულია ე. წ. CAP თეორემა. ამ კონცეფციის თანახმად, განაწილებულ სისტემას შეუძლია, რომ ერთდროულად ჰქონდეს მხოლოდ ორი შემდეგი სამიდან: მთლიანობა (Consistency), ხელმისაწვდომობა (Availability) და, სისტემის რომელიმე ნაწილის დაზიანების შემთხვევაში, მუშაობის შეუფერხებლად გაგრძელება (Partition Tolerance). ცოტათი უფრო დეტალურად მიმოვიხილოთ, თუ რას გულისხმობს თითოეული მათგანი:

- **High Consistency**

იგულისხმება, რომ სისტემის ყველა კვანძი (node) ერთდროულად ერთსა და იმავე ინფორმაციას ,,უყურებს''. სხვა სიტყვებით, წაკითხვის (read) ოპერაცია დააბრუნებს უკანასკნელი ჩაწერის (write) შედეგს ყველა კვანძისათვის (node). თუკი სისტემას ძლიერი მთლიანობა აქვს, ტრანზაქცია ან ბოლომდე წარმატებულად დასრულდება, ან, შეცდომის შემთხვევაში, მთლიანად დაუბრუნდება საწყისს მდგომარეობას ყოველგვარი ცვლილების გარეშე. [5]

- **Partition Tolerance**

როდესაც სისტემა ამ მდგომარეობაშია, მაშინც მას შეუძლია განაგრძოს მუშაობა ქსელში კვანძებს შორის შეტყობინებების შეფერხებით გადაცემის შემთხვევაშიც. სხვა სიტყვებით, სისტემას შეუძლია, მაქსიმალურად ბევრი დაზიანება თუ ხარვეზი ,,აიტანოს'' და არ გამოიწვიოს მთელი ქსელის მწყობრიდან გამოსვლა. ამისთვის საჭიროა, რომ არსებობდეს მონაცემთა ჩანაწერების საკმარისი რაოდენობის რეპლიკაციები სხვადასხვა კვანძებზე (nodes). [5]

- **High Availability**

მაღალი ხელმისაწვდომობის მდგომარეობა ნიშნავს, რომ სისტემა იქნება მუშა მდგომარეობაში დროის ნებისმიერ მონაკვეთში და ყოველ მომხმარებელს დაუბრუნდება შედეგი (ან წარმატებით, ან წარუმატებლად შესრულებული). ასე რომ, High Availability მდგომარეობაში სისტემა არაა დამოკიდებული ერთ კონკრეტული node-ის მდგომარეობაზე და ყოველთვის შესაძლებელია ჩაწერა-წაკითხვის ოპერაციის განხორციელება. [5]

ზემოთ ნალაპარაკებიდან გამომდინარე, შევეცადე, მეჩვენებინა, რომ Big Data სამყარო ძალიან მრავალფეროვანია, ხოლო არქიტექტურა - უამრავი ტექნოლოგიისა, ფიზიკური მოწყობილობის თუ პროგრამის კომპლექსური ურთიერთკავშირია. ცხადია, ამ ყველა თემის ერთ ნაშრომში ჩატევა და ზედაპირულადაც კი დაფარვა შეუძლებელია. აქედან გამომდინარე,

გადავწყვიტე, ყურადღება გამემახვილებინა დღესდღეობით დიდ მონაცემთა მიმართულების ერთ-ერთ ძალიან აქტუალურ ნაწილზე – მონაცემთა ნაკადების რეალურ დროში დამუშავებასა და ანალიზზე.

ნაშრომის თეორიული ნაწილის მთავარი კითხვაა, როგორია არქიტექტურა მონაცემების რეალურ დროში დამუშავებისათვის და როგორ უნდა შევარჩიოთ სწორი მათგანი კონკრეტული პროექტისათვის (პოპულარული არაქართული ტერმინით რომ ვთქვათ, use case-სათვის)? აქედან გამომდინარე, ნაშრომში მიმივიხილავ ორ პოპულარულ და ძირითად მოდელებს – ლამბდა (Lambda) და კაპპა (Kappa) არქიტექტურებს და თუ რომელი მათგანის გამოყენებაა უკეთესი გარკვეული სიტუაციებისათვის და ბიზნეს საჭიროებისათვის. პრაქტიკულ ნაწილში კი წარმოგიდგენთ კონკრეტული ამოცანის იმპლემენტაციას კაპპა არქიტექტურის გამოყენებით.

არსებული მდგომარეობა და განხორციელებული სამუშაო

ზოგადი მიმოხილვა

მონაცემთა რეალურ თუ თითქმის რეალურ დროში ანალიზის კუთხით არსებული მდგომარეობის აღწერა - შეფასება Big Data ფარგლებში, ცოტა არ იყოს, არ არის მარტივი ამოცანა ცალსახა პასუხებით. ვგულისხმობ იმას, რომ მისი არქიტექტურული გადაწყვეტა და პროგრამული უზრუნველყოფა დამოკიდებულია კონკრეტულ ბიზნეს საჭიროებაზე და ამ ბიზნესისათვის დამახასიათებელ უნიკალურ გარემოზე. გამომდინარე იქედან, რომ ე. წ. ღია პროგრამული უზრუნველყოფების (Open Source) სივრცეში, მონდომებული დეველოპერების დამსახურებით, როგორც წესი, უფასოდ ხელმისაწვდომია მრავალი ტექნოლოგია. ყველა კომპანია, რომელიც რეალურ დროში ანალიზის პლატფორმის შექმნით დაინტერესდება, თავისი მიზნების მისაღწევად სხვადასხვა საშუალებას შეიძლება იყენებდეს. თუმცა, რა თქმა უნდა, გარკვეული ტიპის ამოცანებისთვის ე. წ. საუკეთესო პრაქტიკები (Best Practices) ჩამოყალიბდა როგორც არქიტექტურული თვალსაზრისით, ისე ზემოთხსენებული ტექნოლოგიების გამოყენების კუთხითაც.

ნაშრომის ამ ნაწილში მოკლედ განვიხილავ და ერთმანეთს ვადარებ ორ ძირითად – ლამბდა და კაპპა არქიტექტურას, შემდგომ ნაწილებში კი გთავაზობთ ლოკალურ რეჟიმში

კონკრეტული ამოცანის მოდელსა და პრაქტიკულ კაპპა არქიტექტურის საფუძველზე გადაწყვეტის დეტალურ აღწერას.

დავიწყოთ დაპირებული ორ ძირითად არქიტექტურის მიმოხილვით და გავარკვიოთ, რა არის თითოეული მათგანის მთავარი მახასიათებელი და რა შემთხვევაში რომლის გამოყენება სჯობს.

უკანასკნელ წლებში აქტიურად განიხილება, თუ, არქიტექტურული გადმოსახედიდან, როგორი უნდა იყოს რეალურ დროში პროცესინგის სწორი დიზაინი. კარგი არქიტექტურა ამ შემთხვევაში გულისხმობს, რომ ეს სისტემა იქნება ხარვეზების მიმართ ტოლერანტული (Fault Tolerant), მასშტაბირებადი (Scalable), საჭიროების შემთხვევაში განვრცობადი (Extensible); ასევე შეეძლება როგორც მონაცემთა დამუშავება როგორც პაკეტებად (Batch Processing), ასევე – ინკრემენტალურად, ანუ ნაკადების პროცესინგი (Stream Processing).

პირველი მნიშვნელოვანი მიღწევა ზემოთხსენებულ დისკუსიებში უკავშირდება Apache Storm-ის შემქმნელს, ნათან მარცს (Nathan Marz), რომელმაც შეიმუშავა ლამბდას (Lambda) სახელით ცნობილი არქიტექტურა. სხვათა შორის, ეს მოდელი დანერგილია და წარმატებით მუშაობს ძალიან ბევრ კომპანიაში, მათ შორის ისეთებშიც, როგორიცაა Yahoo და Netflix. თუმცა, იგი მთლად იდეალური არაა და გარკვეულ კრიტიკას იმსახურებს ამ არქიტექტურაში ზედმეტი კოდის წარმოქმნისა თუ გადუბლირების ნაწილი.

2014 წლის ზაფხულში LinkedIn-ის თანამშრომელმა, ჯეი კრეპსმა (Jay Kreps) გამოაქვეყნა სტატია ლამბდა არქიტექტურის ნაკლოვანებებთან დაკავშირებით და ალტერნატივად შემოგვთავაზა კაპპა (Kappa) მოდელი. მართალია, კაპპა ლამბდას სრულყოფილი ჩამნაცვლებელი არაა, მაგრამ გარკვეულ სიტუაციებში კაპპა უფრო მარტივი და მოსახერხებელი ალტერნატივაა ლამბდასთან შედარებით. [6]

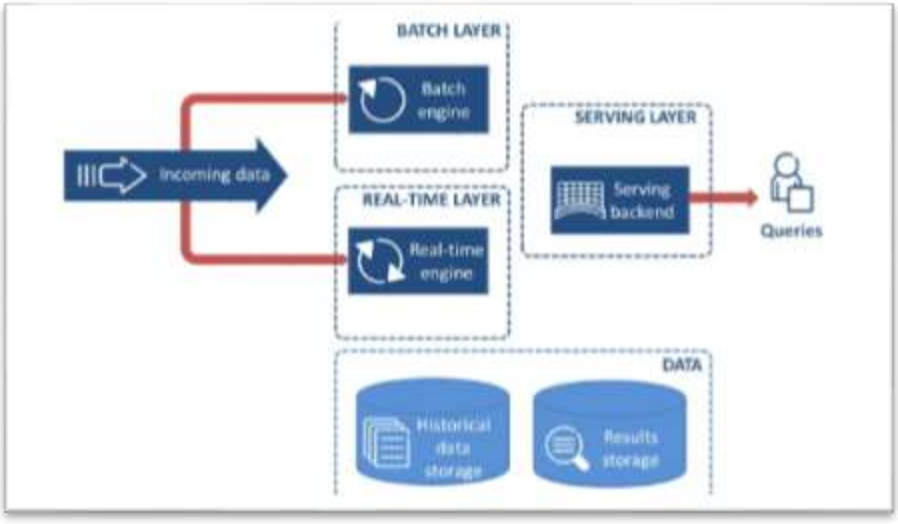
უფრო დეტალურად მიმოვიხილოთ ორივე მათგანი.

ლამბდა არქიტექტურა

ლამბდა არქიტექტურა სამი ფენისაგან შედგება. ესენია: 1. პაკეტების (Batch), 2. ნაკადების (Speed, იგივე Streaming) და 3. მომსახურების (Serving).

1. პაკეტების ფენა (Batch Layer):

ამ ფენას ორი ძირითადი ამოცანა აქვს – ა. ისტორიული მონაცემების მართვა და ბ. შედეგების ხელახლა გამოთვლა (მაგალითად, მანქანური სწავლების მოდელებისათვის). უფრო ზუსტად, ეს ფენა იღებს შემომავალ მონაცემებს, აერთიანებს ისტორიულ მონაცემებთან და კომბინირებულ მონაცემთა სიმრავლეზე ხელახლა აკეთებს გამოთვლებს. ამდენად, იგი მთლიან მონაცემებზე ოპერირებს და მაქსიმალურად აკურატულ შედეგებს იძლევა. თუმცა, რთული მისახვედრი არაა, რომ ამ შედეგების მისაღებად დანახარჯი საკმაოდ დიდია: მონაცემთა სრულ სიმრავლეზე ოპერა-ციების ჩატარებას დიდ დრო და გამომთვლელი რესურსი სჭირდება. ამიტომ ამ ფენისათვის, როგორც წესი, მოლოდინის (დაყოვნების) დრო საკმაოდ მაღალია (High Latency).



ილუსტრაცია 1 ლამბდა არქიტექტურა

2. ნაკადების ფენა (Speed, იგივე Streaming Layer):

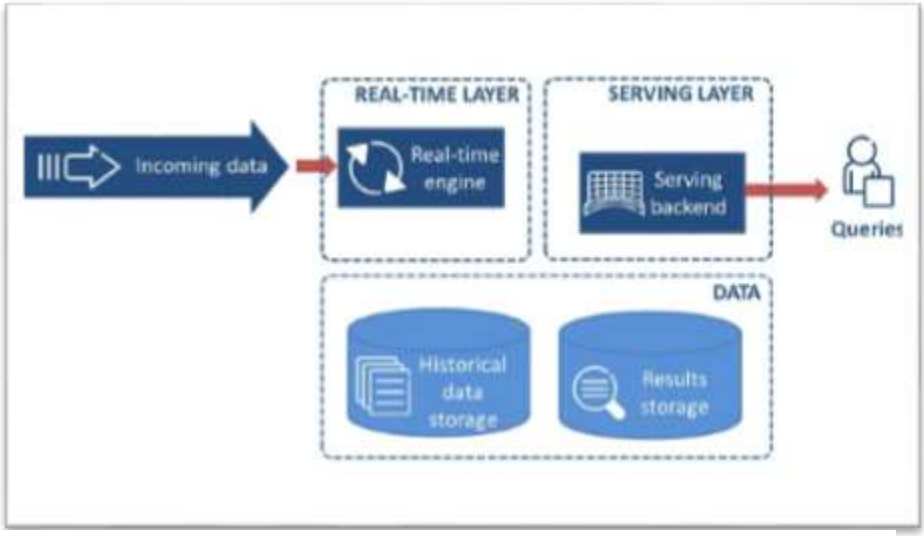
ეს ფენა განკუთვნილია მონაცემების რეალურ (ან თითქმის რეალურ) დროში დასამუშავებლად. შესაბამისად, მოლოდინის დრო (Latency) მინიმალური უნდა იყოს. იგი პასუხისმგებელია შემომავალი ინფორმაციის ნაკადის მალევე დამუშავებაში და ზემოთ აღწერილი პაკეტების ფენის მიერ გამოთვლილი შედეგების ინკრემენტალურად განახლებისათვის.

3. მომსახურების ფენა (Serving Layer):

ამ ფენის ფუნქცია შედარებით მარტივია. მისი დანიშნულებაა, საძიებო მოთხოვნების (query-ების) დამუშავება და ზემოთ აღნიშნული ორი ფენის მიერ მიღებული შედეგების დაბრუნება მომხმარებლებისათვის. [6]

კაპპა არქიტექტურა

რაც შეეხება კაპპა არქიტექტურას, ერთ-ერთი მთავარი მოტივაცია ამ მოდელის შექმნისას იყო ზემოთ აღწერილი ლამბდა არქიტექტურისათვის დამახასიათებელი მონაცემების პაკეტებისა და ნაკადების ფენებისთვის ორი დამოუკიდებელი კოდის ქონა. შესაბამისად, კაპპა მოდელის მთავარი იდეაა, ამ ორი ფუნქციის – მონაცემთა პაკეტების მუდმივი რეპროცესინგისა და რეალურ დროში შემომავალი მონა-ცემთა ნაკადების დამუშავების ერთად მოქცევა და შესაბამისად, ინფორმაციის დამუშავების ორი დამოუკიდებელი ფენის და გადუბლირებული კოდის თავიდან არიდება. შესამისად, იგი მხოლოდ ორი ფენისაგან შედგება. ესენია: 1. ნაკადების პროცესინგის (Stream Processing Layer) და 2. მომსახურების (Serving Layer).



ილუსტრაცია 2 კაპპა არქიტექტურა

პირველი მათგანი უშვებს რეალურ დროში შემომავალი მონაცემების ნაკადების დამუშავებისთვის საჭირო ფუნქციონალს (Job-ებს). მონაცემთა რეპორცესინგი მხოლოდ და მხოლოდ მაშინ არის საჭირო, როცა მონაცემთა დამუშევების ლოგიკა და ფუნქციონალი რამენაირად იცვლება და შესაბამისად, ისტორიული, თავის დროზე დამუშავებული ინფორმაციის ამ ლოგიკით განახლება გვინდა. ასეთი შემთხვევისათვის, ცხადია, დამოუკიდებელი ფუნქციონალი უნდა არსებობდეს, რომელიც საჭირო დროს ისტორიულ მონაცემებზე გამოიძახება.

კაპპა არქიტექტურის მეორე და უკანასკნელი ფენა კი, როგორც ლამბდას შემთხვევაში,

აქაც მომსახურების ფენაა ანალოგიური დანიშნულებით: მისი დანიშნულებაა, საძიებო მოთხოვნების დამუშავება და შედეგების დაბრუნება მომხმარებლებისათვის. [6]

ამ ორივე არქიტექტურის იმპლემენტაცია შესაძლებელია ბიგ დატა სამყაროში კარგად ცნობილი ისეთი Open Source ტექნოლოგიებით, როგორიცაა Apache Kafka, Apache HBase, Apache Hadoop (HDFS, MapReduce), Apache Spark, Apache Drill, Spark Streaming, Apache Storm, Apache Samza თუ სხვები.

მაგალითისთვის, შემომავალი მონაცემები თავდაპირველად ორივე არქიტექტურის შემთხვევაში შეიძლება მოაგროვოს ისეთმა შეტყობინებების განაწილებულმა სისტემამ და ნაკადების პლატფორმამ, როგორიცაა Apache Kafka. მონაცემთა მოდელირება და მათი შენახვა შეიძლება HDFS-ის ან მისი მსგავსი განაწილებული საცავის გამოყენებით. ლამბდა არქიტექტურის batch ფენის უზრუნველოფა შეგვიძლია Hadoop MapReduce ან Apache Spark ტექნოლოგიების გამოყენებით (რაც, მაგალითად, მანქანური სწავლების მოდელების გადასატრენინგებლად არის საჭირო). მონაცემთა ნაკადების დამუშავების ფენისთვის კი შესაფერისია ისეთი ინსტრუმენტები, როგორიცა Apache Storm, Apache Samza და Spark Streaming. ეს უკანასკნელი გამოდგება როგორც ლამბდა, ისე კაპპა არქიტექტურისათვის. ლამბდა არქიტექტურის შემთხვევაში Apache Spark-ის გამოყენება MapReduce-სთან შედარებით ალბათ უკეთესი ვარიანტია ორივე ფენის – batch და speed layer-ების იმპლემენტაციისთვის, ვინაიდან შესაძლებელს ხდის კოდის დიდი ნაწილის გაზიარებას ამ ორ ფენას შორის. საბოლოოდ, რაც შეეხება მომსახურების ფენას, მისი იმპლემენტაციაში გამოიყენება როგორც SQL (Apache Drill, მაგალითად), ისე NoSQL მონაცემთა ბაზები (Apache HBase, Cassandra). [6]

ასე რომ, რა შემთხვევაში რომელი არქიტექტურა უნდა გამოვიყენოთ და რატომ? პირველ რიგში, ამ კითხვაზე პასუხი დამოკიდებულია იმ ამოცანის სპეციფიკაზე, რომლის გადაჭრასაც ვცდილობთ და დამკვეთის საჭიროებებზე. მაგალითად, თუ იდენტური ალგორითმები და, შესაბამისად, კოდების საერთო ბაზა უნდა გამოვიყენოთ მონაცემების ნაკადების თუ ისტორიული ინფორმაციის დამუშავებისას, მაშინ კაპპა არქიტექტურას უნდა მივანიჭოთ უპირატესობა და ტყუილუბრალოდ ორი დამოუკიდებელი ფენის შექმნით საქმე არ გავართულოთ. იმ შემთხვევაში, თუ გვინდა, მონაცემთა ნაკადებზე სხვა ალგორითმი გამოვიყენოთ, ისტორიულ მონაცემები კი სულ სხვანაირად დავამუშავოთ (რეალობაში

ხშირად არის ხოლმე ეს სიტუაცია), მაშინ ლამბდა, დიდი ალბათობით, უკეთესი ვარიანტია.

განხორციელებულ პრაქტიკულ სამუშაოს ამ ნაწილში მოკლედ რომ შევეხო (ქვემოთ, რა თქმა უნდა, დეტალურადაა აღწერილი), იგი კაპპა არქიტექტურაზეა დაფუძნებული. კერძოდ, პირველი ეტაპია, რომ გვაქვს შემომავალ მონაცემთა ნაკადი (კონკრეტულად, მომხმარებლების შეფასებები). ეს ნაკადი უპირველესად იყრება Apache Kafka-ში – ძალიან მოქნილ, სანდო და მოსახერხებელ მესიჯების რიგში (Messaging Queue / Bus. სხვანაირად ნაკადების პლატფორმასაც – Streaming Platform უწოდებენ) JSON შეტყობინებების სახით. ამის შემდეგ Kafka-დან წამოღებული ეს შეტყობინებები გადის რეალურ დროში პროცესინგის ფენას, სადაც კეთდება ამ ინფორმაციის (კონკრეტულად, მომხმარებლების კომენტარების) სენტიმენტ ანალიზი სტენფორდის ერთ-ერთი NLP ბიბლიოთეკის ალგორითმებისა და Apache Spark-ის გამოყენებით. Apache Spark-შივე, სენტიმენტის დადგენისთანავე, თითოეული მესიჯები მუშავდება Apache Cassandra-ში – განაწილებულ NoSQL ბაზაში შექმნილი მოდელის შესაბამისად და იქვე ინახება. ბოლოს კი, Zeppelin-ის გამოყენებით (ან პირდაპირ shell-დან, თუმცა Zeppelin ცალსახად ბევრად მოქნილია) შეგვიძლია ანალიტიკური თუ სხვა სახის query-ების გაშვება და შედეგების რეალურ დროშივე ნახვა.

ამოცანის დასმა

როგორც ნაშრომის შესავალში აღვნიშნე, როცა მონაცემთა ნაკადების რეალურ (ან თითქმის რეალურ) დროში ანალიზზე ვსაუბრობთ, მართალია, შინაარსობრივად ერთი ტიპის ამოცანაზე გვაქვს საუბარი, მაგრამ ვფიქრობ, პრაქტიკაში ძალიან მრავალფეროვან თავსატეხებს ვაწყდებით გამომდინარე ბიზნესის უნიკალური მოთხოვნებიდან, საჭიროებებიდან, გარემოდან და ა. შ. სწორედ ამიტომ, რახან რთულია ერთ უნივერსალურ მიდგომაზე საუბარი, გადავწყვიტე, უფრო კონკრეტული მიზანი დამესახა ჩემი ნაშრომისათვის გარკვეული დაშვებებით (ვგულისხმობ რაიმე ერთ ბიზნეს მოთხოვნაზე კონცენტრირებას) და პრაქტიკულ ნაშრომში ხელმისაწვდომი რესურსებით გადამეჭრა. კერძოდ, ინტერნეტში არსებობს გარკვეული წყაროები, სადაც თავმოყრილია სხვადასხვა ტიპისა და ზომის მონაცემები (მათ შორის ბევრია რეალური კომპანიების რეალური ინფორმაციაც). ერთ-ერთი ასეთი წყაროდან მოვიპოვე აშშ-ის სხვადასხვა მხარეში მდებარე სხვადასხვა სასტუმროს შეფასებები და მომხმარებელთა განხილვები .csv ფაილის სახით. რახან ნაკადის სახით არ იყო ხელმისაწვდომი მსგავსი ინფორმაცია, მიწევს, რომ ამ ფაილიდან მონაცემთა ნაკადის სიმულაცია შევქმნა. დავუშვათ, რომ ბიზნესის მიზანია, ჰქონდეს ისეთი გარემო, სადაც შეეძლება, ეს მონაცემები რეალურ დროში, დაგენერირებისთანავე დაამუშავოს, გაანალიზოს და ბაზებში უკვე დამუშავებული ინფორმაცია შეინახოს. დავუშვათ, ამ სასტუმროების შეფასება-განხილვების მაგალითზე ბიზნესს აინტერესებს, რამდენად კმაყოფილები არიან მათი მომხმარებლები და, შესაბამისად, გააკეთოს მათი კომენტარების სენტიმენტ ანალიზი რეალურ დროში. კერძოდ, ამ შეფასებების დაგენერირებისთანავე უნდა დამუშავდეს და გაანალიზდეს მათი სენტიმენტი ისე, რომ რეალურ დროში, ისე ისტორიულადაც მიუწვდებოდეს ხელი ამ ინფორმაციაზე ანალიტიკოსებს. ჩემი ნაშრომის მიზანია, ამ კონკრეტული ამოცანის გადაჭრა და ბიზნესისათვის შესაბამისი გარემოს შესაქმნელად კონკრეტული გადაწყვეტილების შეთავაზება.

ზოგად დონეზე, ამ ამოცანას შემდეგ კომპონენტებად ვყოფ :

- მონაცემთა ნაკადის შექმნა (სიმულაცია);

- ნაკადებად შემომავალი მონაცემების მესიჯებად შენახვა შესაბამის სტრუქტურაში (მესიჯების რიგში. კერძოდ, Apache Kafka-ში). შეიძლება Staging-იც ვუწოდოთ ამ ნაწილს;

- Apache Kafka-დან შემდგომში ამ მესიჯების დამუშავება Apache Spark-ის გამოყენებით NLP-ის მზა ბიბლიოთეკებით (Stanford CoreNLP) და სენტიმენტის მინიჭება;

- სენტიმენტის დაგენერირების შემდეგ მონაცემების პროცესინგი და NoSQL ბაზის შესაბამისად მოდელირებულ სტქუტურაში შენახვა (კერძოდ, Apache Cassandra-ში);

- Apache Zeppelin-ის გამართვა და იქედან ანალიტიკური query-ების გაშვება ზემოთხსენებულ NoSQL ბაზაში, Cassandra-ში, იქნება ეს კარგად ნაცნობი SQL-ის სახით თუ Spark-ის DataFrame-ების და Scala-ს სინტაქსის გამოყენებით. რა თქმა უნდა, ამ Query-ების საფუძველზე შესაძლებელი უნდა იყოს როგორც ისტორიული, ისე რეალურ დროში შემომავალი მონაცემების შედეგების მიღება.

მიმაჩნია, რომ დასახული ამოცანის მთავარი გამოწვევა ამ კომპონენტების ერთმანეთთან დაკავშირება და პროცესის აწყობაა. ასევე ისიც, თუ შუალედურ კომპონენტებში მონაცემებს რა მოდელს შევურჩევთ და როგორ შევინახავთ. უფრო დეტალურად კი ამ ტექნოლოგიებზე და იმპლემენტაციაზე ნაშრომის შემდეგ ნაწილებში ვისაუბრებ:

- სისტემური მოდელში აღწერილია არქიტექტურა და მონაცემთა დინება კომპონენტებს შორის. აღწერილია ის ტექნოლოგიები, რომლებიც უნდა გამოვიყენო რეალურ იმპლემენტაციაში;
- პრობლემის გადაწყვეტისა და დანართის ნაწილებში ყურადღება ეთმობა პრაქტიკულ ნაწილს და ძირითადად გარჩეულია კოდები.

სისტემური მოდელი

მოდელის მიმოხილვა

არქიტექტურა და მოდელი შემდეგია:

- ნაკადის სიმულაციაზე დიდ ყურადღებას არ გავამახვილებ – რეალურად ბიზნესს სიმულაცია არაფერში დასჭირდება, რეალურ წყაროსთან ექნება საქმე. ისე კი სიმულაციას Spark-ის გარემოში ვაკეთებ. აქ ჩვენთვის საინტერესო ისაა, თუ როგორ ვეპყრობით მონაცემთა ნაკადს ჩვენს სისტემაში შემოსვლისთანავე: წყაროდან მიღებული მესიჯები მუშავდება, გადაგვყავს JSON ფორმატში და იყრება Kafka-ს შესაბამის ტოპიკში producer-ის (Kafka Producer) მეშვეობით;

- Kafka-დან Spark-ის გარემოში consumer-ის (Kafka Consumer) მეშვეობით ვკითხულობთ მესიჯებს, ვპარსავთ JSON-ს, ვიღებთ უშუალოდ კომენტარს და ამ უკანასკნელს ვაანალიზებთ სტენფორდის NLP ბიბლიოთეკის გამოყენებით. ვანიჭებთ 5 შესაძლებელი სენტიმენტიდან – Very Negative, Negative, Neutral, Positive, Very Positive ერთ-ერთს;

- ვუკავშირდებით Cassandra-ს პროცესებს და ვიძახებთ სერვისს, რომელიც გადაცემულ მესიჯს Cassandra-ს ბაზებში მიერ შექმნილი მოდელის მიხედვით დაამუშავებს და თუ საჭიროა, INSERT-ებსაც გააკეთებს. ერთ-ერთი ველი, ცხადია, წინა ეტაპზე დაგენერირებული სენტიმენტია;

- Zeppelin-დან ვუკავშირდებით Spark-სა და Cassandra-ს სერვისებს, საიდანაც შეგვიძლია გავუშვათ ანალიტიკური query-ები როგორც Spark DSL-ზე (Domain-Specific Language), ისე Spark SQL-ზე ან პირდაპირ Cassandra CQL (Cassandra Query Language) ენაზე.

სქემატურად ეს ყველაფერი შემდეგნაირად გამოიყურება:

ილუსტრაცია 3 სისტემური მოდელი

აქვე მიმოვიხილოთ ზემოთხსენებული ძირითადი ტექნოლოგიები:

Apache Spark

Apache Spark არის უნიფიცირებული გამოთვლითი ძრავა და ბიბლიოთეკების ერთობლიობა კლასტერებზე მონაცემების პარალელური პროცესინგისათვის. ამ მომენტისათვის, სპარკი არის ღია პროგრამული უზრუნველყოფების (Open Source) სივრცეში დეველოპერების მიერ ყველაზე აქტიურად განვითარებადი პროდუქტი, რის გამოც დიდ მონაცემებთან მომუშავე თითქმის ყველა დეველოპერისა თუ მონაცემთა მეცნიერისათვის იგი

ილუსტრაცია 4 სპარკის ეკოსისტემა

სტანდარტულ ტექნოლოგიად იქცა.

სპარკს რამდენიმე პოპულარული პროგრამირების ენის მხარდაჭერა აქვს - Python, Java, Scala და R. იგი შედგება უამრავი ბიბლიოთეკისაგან, რომლებიც განკუთვნილია მრავალფეროვანი ამოცანების გადასაჭრელად, სტანდარტული SQL-დან დაწყებული მონაცემთა ნაკადებით თუ მანქანური სწავლებით დამთავრებული. Spark-ის გაშვება შეიძლება როგორც ლეპტოპზე, ისე ათასობით სერვერიდან შემდგარ კლასტერზეც. ამის წყალობით, Spark არის საკმაოდ მოსახერხებელი სისტემა დიდი მონაცემების გარემოს აწყობის დასაწყებად და თანდათან მასშტაბების უსაზღვროდ გასაზრდელად.

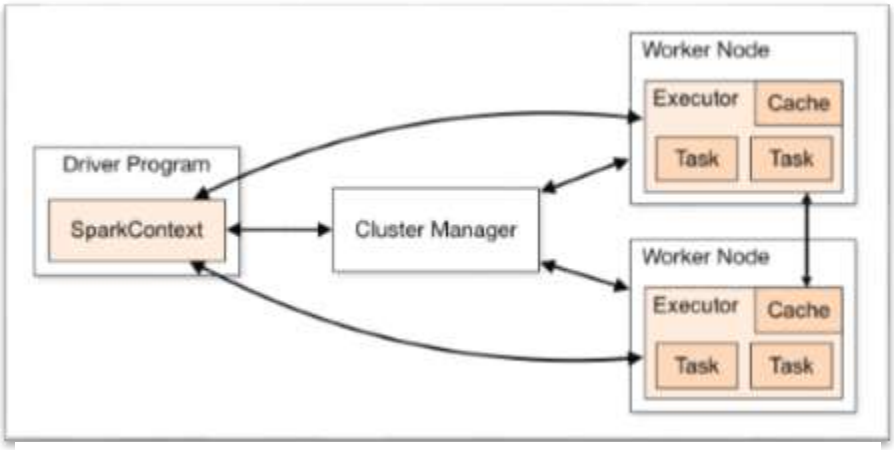
სპარკის გამოყენება შეიძლება როგორც მონაცემთა პერსისტენტული საცავების სისტემებთან (როგორიცაა Azure Storage და Amazon S3), ასევე განაწილებულ ფაილურ სისტემების (Apache Hadoop, მაგალითისთვის), გასაღები-მნიშვნელობის (Key-value) წყვილებზე დაფუძნებული საცავებსა (Apache Cassandra) და თუნდაც მესიჯების შემნახველ სისტემაში (როგორიცაა Apache Kafka). სპარკი არასდროს ინახავს მონაცემებს დიდი ხნით თავის სისტემაში და არც უპირატესობას ანიჭებს ერთს მეორესთან შედარებით. მისთვის მთავარი არის ის, რომ მონაცემები უკვე ინახება შესაბამის საცავში (ან რამდენიმე განსხვავებულ საცავში). მონაცემთა ქსელში მოძრაობა და ერთი ადგილიდან მეორეზე გადატანა ძალიან ძვირი ოპერაციაა, ამიტომ სპარკი ფოკუსირდება მონაცემების ადგილზევე დამუშავებაზე, განურჩევლად იმისა, თუ სადაა ისინი შენახული.

ერთეულ მანქანებს, რაც არ უნდა მძლავრები იყვნენ, ჯერჯერობით არ აქვთ იმხელა რესურსი, რომ უზარმაზარი მოცულობის ინფორმაცია მომხარებლებისათვის მისაღებ დროში დაამუშავონ. კლასტერი, ანუ კომპიუტერების ჯგუფი, თავს ერთად უყრის მრავალი მანქანის რესურსს ისე, თითქოს ერთი, ძალიან მძლავრი კომპიუტერი იყოს. თუმცა ამ სიმძლავრეს არანაირი სარგებელი არა აქვს, თუკი არ გვექნება რაიმე framework-ი, რომელიც შეძლებს ამ

კლასტერზე დატვირთვის სწორად განაწილებას და პროცესების მართვას. სწორედ ამას აკეთებს სპარკი – მონაცემებზე ჩასატარებელ სამუშაოებს ხლეჩს ამოცანებად, ანაწილებს კლასტერზე და უწევს კოორდინაციას.

კლასტერი და მასზე რესურსების განაწილება, რომელსაც სპარკი გამოიყენებს ამოცანების გასაშვებად და შესასრულებლად, იმართება გარკვეული პროცესებით, რაზეც შესაბამისი მენეჯერია პასუხისმგებელი. ეს მენეჯერი შეიძლება იყოს საკუთრივ სპარკისიც (Spark's Standalone Cluster Manager), ან სულაც YARN (Yet Another Resource Negotiator Hadoop-ის ეკოსისტემიდან), ან Apache Mesos. სპარკის აპლიკაციები გადაეცემა ამ სამიდან რომელიმე მენეჯერს, ეს უკანასკნელი კი გადაწყვეტს, რა რესურსები ესაჭიროება ამ აპლიკაციას შესასრულებლად და შესაბამისად გამოუყოფს მას.

სპარკის აპლიკაციის სტრუქტურაზე რომ ვილაპარაკოთ, იგი შედგება ორი მთავარი ნაწილისაგან: დრაივერი (Driver) და შემსრულებელი პროცესებისაგან. პირველი მათგანი ეშვება კლასტერის რომელიმე კვანძზე (node) და ზრუნავს გადაცემული აპლიკაციის main() ფუნქციის შესრულებაზე. მისი სამი მთავარი მოვალეობაა: 1. აპლიკაციის შესახებ ინფორმაციის შენახვა; 2. მომხმარებლის პროგრამაზე ან მის რაიმე აქტივობაზე საპასუხო მოქმედება და 3. ამოცანის ანალიზი, მისი დანაწევრება ქვე-ამოცანებად, კლასტერის შემსრულებელ კომპონენტებზე გადანაწილება და მათი გაშვების დაგეგმვა. მიუხედავად იმისა, რომ დრაივერ პროცესი უშუალოდ საქმეს თვითონ არ აკეთებს, სპარკისათვის საციცოხლოდ მნიშვნელოვანია გამომდინარე იქედან, რომ მასზეა დამოკიდებული ამ აპლიკაციის ბედი, თუ სად, როგორ, როდის და რანაირად უნდა გაეშვას და ასევე, რა თქმა უნდა, ამ ყველაფრის ოპტიმიზაციაც მისი მოვალეობაა. ამ პროცესს ტექნიკურად SparkSession ეწოდება. სპარკის სესიაა სწორედ ის საშუალება, რომლითაც სპარკთან ვურთიერთობთ და კლასტერზე ვაშვებინებთ ჩვენ მიერ განსაზღვრულ მანიპულაციებსა თუ



ილუსტრაცია 5 სპარკის აპლიკაციის სტრუქტურა

მოქმედებებს მონაცემებზე.

შემსრულებლების ძირითადი ფუნქცია არის დრაივერ პროცესის მიერ მათთვის განაწილებული ქვე-ამოცანების უშუალოდ შესრულება, რომლის დასრულების შემდეგ კი მდგომარეობა უნდა ,,მოახსენოს“ თავად დრაივერს და დაუბრუნოს შედეგი. სამუშაოს ამგვარად დანაწილების მთავარი მოტივაცია პროცესების დაპარალელებაა, რამდენადაც ეს ამოცანის ხასიათიდან და არსებული რესურსებიდან გამომდინარეა შესაძლებელი.

სპარკის შესაძლებლობების სწორად გამოყენებისათვის აუცილებელია, რომ შემდეგი კონცეფციები კარგად გვესმოდეს:

1. DataFrames – დატაფრეიმები

დატაფრეიმი უბრალოდ წარმოადგენს მონაცემების ცხრილს, თავისი სტრიქტონებითა და სვეტებით. სიას, რომელიც ამ სვეტებს და მათ ტიპებს განსაზღვრავს, სქემა ეწოდება. დატაფრეიმზე როცა ვსაუბრობთ, შეგვიძლია, ექსელის ცხრილებიც წარმოვიდგინოთ, უბრალოდ სხვაობა იმაშია, რომ ექსელის ფაილი მხოლოდ ერთ კომპიუტერზეა, დატაფრეიმი კი განაწილებულია უამრავ კომპიუტერზე კლასტერში.

2. Partitions – პარტიციები

როგორც ზემოთ ვახსენეთ, დრაივერი მთავარ ამოცანას ქვე-ამოცანებად ყოფს და შემსრულებლებს გადასცემს პარალელიზმის მისაღწევად. რეალურად ამის უკან იგულისხმება, რომ სპარკი მონაცემებს გარკვეული პრინციპით ჰყოფს პორციებად, რომელსაც პარტიციები ეწოდებათ. დატაფრეიმის პარტიციები რეალურად წარმოადგენს იმას, თუ როგორ არის მონაცემები რეალურად გადანაწილებული კლასტერის ყველა მანქანაზე აპლიკაციის შესრულების პროცესში. მხოლოდ ერთი პარტიციის ქონა ნიშნავს პარალელიზმის დათმობას მიუხედავად იმისა, რომ ამ დროს ათასობით შემსრულებელი შეიძლება გვქონდეს. პირიქითაც, თუ ბევრი პარტიცია გვაქვს, მაგრამ მხოლოდ ერთი შემსრულებელი, პარალელიზმი ვერც ამ შემთხვევაში გვექნება.

3. ტრანსფორმაციები (transformations) და მოქმედებები (actions); ზანტი გამოთვლა (lazy evaluation)

ტერმინი ,,ზანტი გამოთვლა'' ნიშნავს, რომ სპარკი მოიცდის უკანასკნელ წუთამდე, რომ ფიზიკურად შეასრულოს გრაფის სახით დაგენერირებული გამოთვლების ინსტრუქციები. იმის ნაცვლად, რომ მონაცემების პროცესინგი და შედეგის გამოთვლა მოხდეს მათზე რაიმე ტრანსფორმაციის გამოძახებისთანავე, სპარკში უბრალოდ გენერირდება ამ მონაცემების ტრანსფორმაციების ლოგიკური გეგმა ისე, რომ ამ მომენტში ფიზიკურად მონაცემებს საერთოდ არ ეხება. ამის შემდეგ იგი იცდის აპლიკაციის იმ კონკრეტულ მომენტამდე, როცა ამ ტრანსფორმაციების შედეგის დაბრუნება იქნება გამომძახებლისათვის საჭირო. ამ დროს კი სპარკი ზემოთხსენებულ ლოგიკურ გეგმას უკვე ისეთ ფიზიკურ გეგმად გარდაქმნის და თან დააოპტიმიზირებს, რომელიც მაქსიმალურად ეფექტურად გაეშვება და იმუშავებს კლასტერზე. ამის ნიმუშია ე. წ. Predicate pushdown – მაგალითად, გვაქვს დიდი მონაცემები, რომლიდანაც კონკრეტული ერთი ჩანაწერი ჩანაწერი გვაინტერესებს, რისთვისაც ეს მონაცემები უნდა დავფილტროთ. სპარკში ამ მონაცემებს DataFrame-ად თუ წავიკითხავთ, მნიშვნველობა არ აქვს, ჩვენს კოდში სასურველ ფილტრს DataFrame-ზე რა მომენტში გამოვიძახებთ. იგულისხმება, რომ შეიძლება, ამ ცხრილს აპლიკაციის დასაწყისში ვკითხულობდეთ DataFrame-დ და ამ დროს სასურველ ფილტრს აპლიკაციის დასასრულს ვადებდეთ. ტრადიციული SQL ბაზების შემთხვევაში ეს ძალიან ცუდი იქნებოდა, ვინაიდან კლიენტი ჯერ მთელს ცხრილს წაიკითხავდა და ბოლოს დაფილტრავდა წაკითხულ მონაცემებს. სპარკის ზანტი გამოთვლების, predicate pushdown-ებისა და ტრანსფორმაციების ოპტიმიზირებული გეგმების შექმნის დამსახურებით კი სპარკი ჯერ კარგად გაანაალიზებს აპლიკაციის კოდს და შემდეგ წაკითხვისთანავე ეფექტურად დაფილტრავს ამ მონაცემებს ისე, რომ მთლიანი ცხრილის წაკითხვა და შემდგომში დაფილტრვა თავიდან აგვარიდოს.

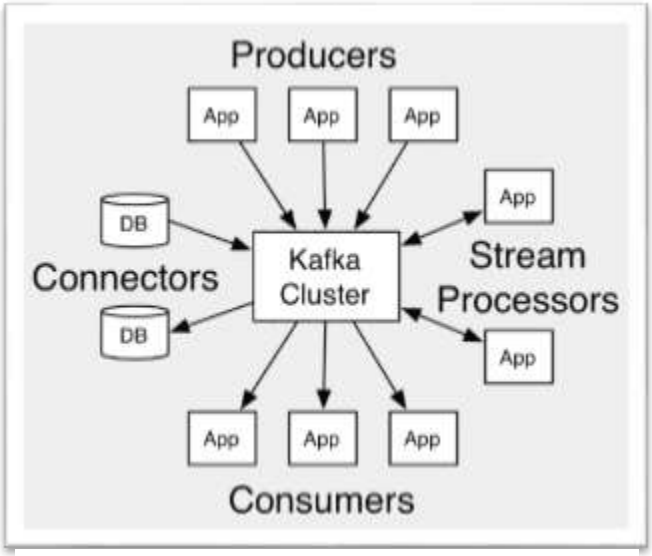
როგორც ზემოთ ვახსენეთ, მონაცემებზე ტრანსფორმაციების გამოძახების შედეგად ლოგიკურ გეგმას ვიღებთ და არა უშუალოდ ტრანსფორმირებულ მონაცემებს. იმისთვის, რომ სპარკი ,,ვაიძულოთ“, გეგმა კი არა, გამოთვლის შედეგი დაგვიბრუნოს, უნდა გამოვიძახოთ გარკვეული მოქმედება. მოქმედება სპარკს უბიძგებს, რომ ზემოთხსენებული გეგმა ფიზიკურად შეასრულოს და შედეგი გამოთვალოს.

აღსანიშნავია, რომ სპარკში მონაცემებთან ურთიერთობის ფუნქციონალი მხოლოდ DataFrame-ების გამოყენებით არ შემოიფარგლება. Spark SQL ბიბლიოთეკის დამსახურებით, ნებისმიერი DataFrame შეგვიძლია დავარეგისტრიროთ დროებით ცხრილად ან წარმოდგენად (View), ხოლო მათგან შედეგი მივიღოთ კარგად ნაცნობ SQL-ზე დაწერილი ბრძანებების გამოყენებით. ეფექტური მუშაობის თვალსაზრისით, მნიშვნელობა არ აქვს, აპლიკაციის ლოგიკას DataFrame-ებზე დავწერთ თუ Spark SQL ბიბლიოთეკას გამოვიყენებთ – სპარკი ორივე შემთხვევაში იდენტურ გეგმებს აგენერირებს. [7]

Apache Kafka

Apache Kafka ე. წ. publish/subscribe messaging სისტემების ერთ-ერთი წარმომადგენელია. ასეთი სისტემებისათვის დამახასიათებელია შემდეგი მოდელი: გამგზავნი (Publisher) აგენერირებს მონაცემებს (მესიჯებს), რომელიც რომელიმე კონკრეტული მიმღებისათვის არაა განკუთვნილი. კერძოდ, იგი როგორღაც აკლასიფიცირებს შეტყობინებებს და, როგორც წესი, თავს უყრის მათ ერთ ცენტრალურ წერტილში (ბროკერი, Broker). მიმღები (გამომწერი, Subscriber) კი ამ ბროკერიდან კითხულობენ შეტყობინებების მათთვის საინტერესო კატეგორიებს.

Apache Kafka-ც სწორედ ასეთი სისტემაა. მას ხშირად ახასიათებენ, როგორც ნაკადების განაწილებულ პლატფორმას ან ლოგს. მონაცემები კაფკაში არის შენახული საიმედოდ და თანმიმდევრულად. შეტყობინებების წაკითხვა კაფკადან ხდება დეტერმინისტულად. დამატებით, მონაცემები განაწილებულია კაფკაშიც ისე, რომ, პირველ რიგში, დამატებითი საფრთხეები არიდებულია თავიდან (რეპლიკაცია ხდება) და ასევე, სისტემის მასშტაბირებადობაც იოლია საჭიროების შემთხვევაში.

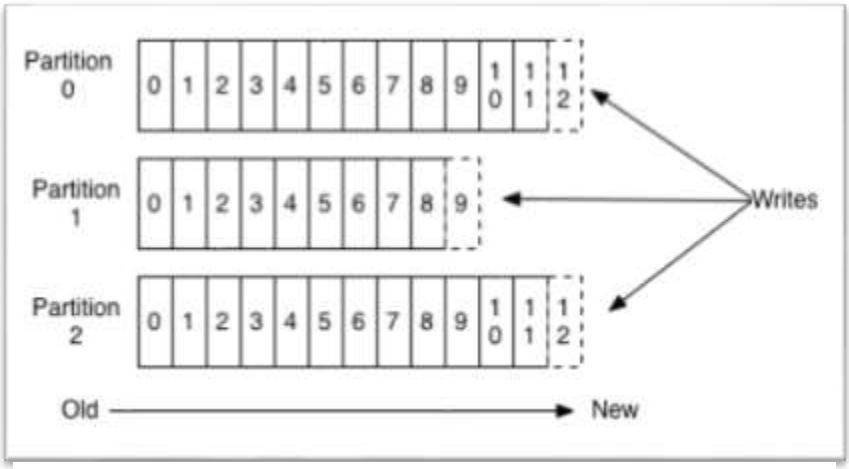


ილუსტრაცია 6 კაფკას ზოგადი მოდელი

კაფკაში მონაცემების ,,საზომ“ (უმცირეს) ერთეულს შეტყობინება (მესიჯი) ეწოდება. იგი მსგავსია SQL ბაზაში სტრიქონის ან ჩანაწერის. შეტყობინება ამ სისტემისათვის მხოლოდ და მხოლოდ ბაიტების მასივია, ასე რომ, რეალურად რა წერია ამ მესიჯში და როგორი ფორმატით, კაფკასათვის სულერთია.

შეტყობინებები კაფკაში ორგანიზებულია თემების (ტოპიკების, topics) სახით. SQL ბაზაში მისი ანალოგი ცხრილია, ფაილურ სისტემაში – საქაღალდე (folder). ტოპიკები დაყოფილი შეიძლება იყოს პარტიციებად. ერთი ტოპიკის განსხვავებული პარტიციები შეიძლება, რომ სხვადასხვა სერვერზე იყოს მოთავსებული, რაც იმას ნიშნავს, რომ ცალკეული ტოპიკის ჰორიზონტალური მასშტაბირებადობაც არის შესაძლებელი და რამდენიმე სერვერის რესურსების გამოყენება ეფექტური და სწრაფი მუშაობისათვის. [8]

როცა კაფკაზე ან მის მსგავს სისტემებზე ვსაუბრობთ და ტერმინ ,,ნაკადს“ ვახსენებთ, როგორც წესი, ვგულისხმობთ, რომ ერთი ნაკადი ერთ ტოპიკს შეესაბამება, განურჩევლად პარტიციების ოდენობისა. ნაკადიდან შეტყობინებები კაფკას შესაბამის ტოპიკების პარტიციებში თანმიმდევრობით იყრება, უკვე არსებულ მესიჯებს ბოლოში ემატება და წაკითხვის დროს ყოველთვის FIFO პრინციპი გამოიყენება. იმ შემთხვევაში, თუ ტოპიკს რამდენიმე პარტიცია აქვს, მაშინ ტოპიკის ფარგლებში მესიჯების ქრონოლოგიური დალაგება შეიძლება დაგვერღვეს.



ილუსტრაცია 7კაფკას topic-ის ანატომია

ზემოთ ვახსენეთ, რომ კაფკა ე. წ. publish/subscribe messaging სისტემა. Publish-ერს აქ producer-ი ეწოდება, ხოლო subscriber-ს – consumer-ი პროდიუსერები ქმნიან მესიჯებს და შემდეგ მათ შესაბამის ტოპიკებში ყრიან. producer-ებს არ აინტერესებთ, რომელ პარტიციაში წერენ ამ მესიჯებს და ნაგულისხმევი მნიშვნელობებით, შეეცდებიან, ტოპიკის ყველა პარტიციაში შეტყობინებების რაოდენობა დააბალანსონ. consumer-ებს რაც შეეხებათ, ისინი კონკრეტულ ტოპიკს არიან მიმაგრებულები და კითხულობენ შეტყობინებებს იმ თანმიმდევრობით, რა თანმიმდევრობითაც ისინი პროდიუსერმა ჩაწერა. კონსიუმერი აკონტროლებს, რომელი

მესიჯები აქვს უკვე წაკითხული და საიდან უნდა განაგრძოს წაკითხვა. ამისთვის იმახსოვრებს ოფსეტს – მეტაინფორმაციას, რომელიც არის მუდმივად ზრდადი რიცხვი. კაფკა ერთით ზრდის მას პარტიციაში ყოველი ახალი შეტყობინების ჩამატებისას. ამდენად თითოელ პარტიციაში თითოეულ მესიჯს უნიკალური ოფსეტი (offset) აქვს (ფაქტობრივად, ინდექსი). კონსიუმერები კი ბოლო წაკითხულ მესიჯს სწორედ მისი ოფსეტის შენახვის გზით იმახსოვრებენ. ბოლო წაკითხული შეტყობინების დამახსოვრება თუნდაც იმიტომ არის სასარგებლო, რომ სისტემის მწყობრიდან გამოსვლის შემთხვევაში კონსიუმერებმა შეძლონ, კითხვა სწორედ იქედან განაგრძონ, სადაც გაწყვიტეს. [9]

Apache Cassandra

Apache Cassandra განაწილებული ბაზაა ძალიან დიდი მოცულობის სტრუქტურიზებული მონაცემების სამართავად. Facebook-ში შექმნილი ეს Open Source სისტემა, ზემოთ აღწერილი ტექნოლოგიების მსგავსად, ისიც არის მასშტაბირებადი (scalable), ხარვეზების მიმართ ტოლერანტული (fault tolerant) და მდგრადი (consistent). იგი მკვეთრად განსხვავდება რელაციურ მონაცემთა ბაზების მართვის სისტემებისაგან. Cassandra-ს იყენებენ ისეთი მსხვილი კომპანიები, როგორიცაა Facebook, Twitter, Cisco, ebay, Netflix და ა. შ.

კასანდრას მთავარი გამოწვევა, არქიტექტურული თვალსაზრისით, არის გაუმკლავდეს დიდი მოცულობის მონაცემებით დატვირთვას მრავალი კვანძის (Node) მეშვეობით ისე, რომ არც ერთი შეფერხების წერტილი (Single Point of Failure) არ ჰქონდეს. კასანდრა თავის node-ებს ე. წ. Peer-to-peer განაწილებულ სისტემაში აერთიანებს და მონაცემებს კლასტერის ყველა node-ზე ანაწილებს. კასანდრას კლასტერში ყველა კვანძი ერთსა და იმავე როლს ასრულებს. თითოეული მათგანი დამოუკიდებელია, თუმცა, ამავე დროს, დაკავშირებულია სხვა node-ებთან. თითოეულ მათგანს შეუძლია წაკითხვის და ჩაწერის მოთხოვნების მიღება, მიუხედავად იმისა, თუ მონაცემები რეალურად კლასტერის რა ნაწილშია ლოკალიზებული. როცა რომელიმე კვანძი ფერხდება და ავარიულად ითიშება, ეს მოთხოვნები შესასრულებლად ქსელში სხვა node-ებს ეგზავნებათ. ამასთანავე, კლასტერში ერთი ან რამდენიმე კვანძი მონაცემების კონკრეტული ნაწილისათვის რეპლიკების ფუნქციას ასრულებს. თუ როგორმე კასანდრამ აღმოაჩინა, რომ მომხმარებლისათვის შედეგის დაბრუნებისას რატომღაც ამ node-ებიდან ერთ-ერთი მაინც მოთხოვნილი მონაცემების უახლეს მნიშვნელობას არ აბრუნებს,

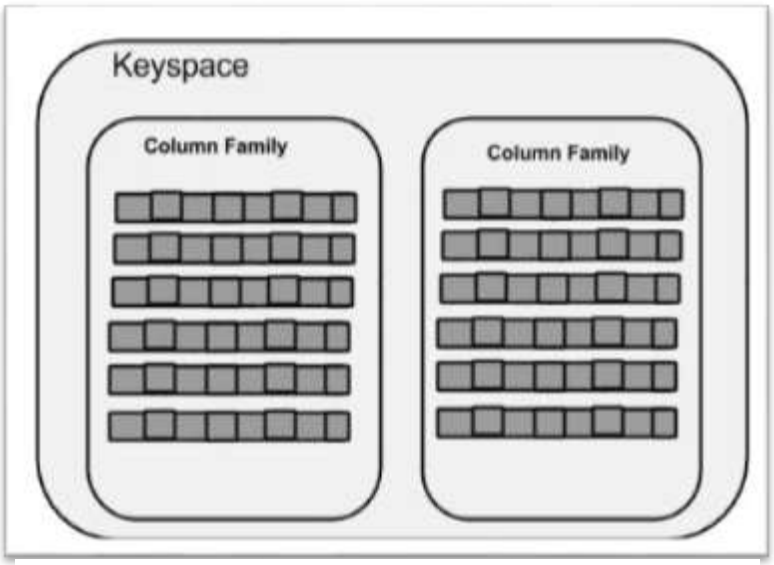
კასანდრა შეამოწმებს, ხომ ნამდვილად უახლესს მონაცემებს უბრუნებს კლიენტს, ამის შემდეგ კი ,,ვადაგასულ'' მონაცემებს მოუბრუნდება და განაახლებს read repair პროცესით.

რაც შეეხება კასანდრაში მონაცემების შენახვის მოდელს, იგი შემდეგი ძირითადი კომპონენტებისგან შედგება:

1. Keyspace:

Keyspace კასანდრაში მონაცემებისთვის ყველაზე გარეთა, ასე ვთქვათ, კონტეინერია. მისი ძირითადი ატრიბუტებია:

- რეპლიკაციის ფაქტორი - რაოდენობა კლასტერში იმ მანქანებისა, რომლებიც ამ Keyspace-ის მონაცემების ასლებს შეინახავან;
- რეპლიკების განთავსების სტრატეგია – განსაზღვრავს, თუ როგორ იქნებიან წრიული ტოპოლოგიის ქსელში რეპლიკები განთავსებული;
- სვეტების ოჯახები (Column Families) – Keyspace შედგება ერთი ან რამდენიმე სვეტების ოჯახისაგან.



ილუსტრაცია 8 Keyspace-ების და Column Family-ების სქემატური წარმოდგენა

2. Column Families:

სვეტების ოჯახები არის სტრიქონების კოლექციის კონტეინერი. თითოეული სტრიქონი შედგენილია დალაგებული სვეტებისაგან. სხვა სიტყვებით, სვეტების ოჯახი მონაცემების სტრუქტურას წარმოადგენენ. იგი რელაციური ბაზების ცხრილების მსგავსია, თუმცა არა იდენტური. რელაციური ცხრილებისაგან განსხვავებით, Cassandra-ს ცხრილები უფრო მოქნილია, არ აქვთ ფიქსირებული სქემა. ასევე, მნიშვნელობებად შეიძლება გამოვიყენოთ მონაცემთა არაპრიმიტიული ტიპები, როგორიცაა სიები, სიმრავლეები და ა. შ. რელაციური ცხრილები თუ განსაზღვრავენ მხოლოდ სვეტებს და მომხმარებელს შეუძლია ამ ცხრილის წინასწარ განსაზღვრული ტიპის მიხედვით კონკრეტული მნიშვნელობებით

შევსება. კასანდრაში შესაძლებელია ე.წ. სუპერ სვეტის (Super Column) შექმნაც, რომელშიც ქვე-სვეტების ჩამონათვალს გავაკეთებთ. Super Column არის განსაკუთრებული ხასიათის სვეტი. იგი განსაზღვრულია გასაღებისა და მნიშვნელობების წყვილის სახით. გასაღები არის მისი სახელი, მნიშვნელობებში კი ქვე-სვეტების ჩამონათვალია (map სტრუქტურა, უფრო კონკრეტულად).

როგორც წესი, სვეტების ოჯახები დამოუკიდებელ ფაილებად ინახება დისკზე. შესაბამისად, კასანდრას მუშაობის ოპტიმიზაციისათვის, უკეთესი იქნება, თუ ერთად, ერთ სვეტების ოჯახში შევინახავთ სვეტებს, რომლებიც, დიდი ალბათობით, მომხმარებელს ერთდროულად, ერთ საძიებო მოთხოვნაში დასჭირდება. აქ კი შეიძლება, სუპერ სვეტები (Super Columns) სასარგებლო აღმოჩნდეს.

3. Columns:

სვეტი კასანდრას ყველაზე საბაზისო მონაცემთა სტრუქტურაა. იგი სამი მნიშვნელობისაგან შედგება: გასაღებისაგან (Key, რომელიც არის სვეტის სახელი), მნიშვნველობისაგან (value) და დროის შტამპისაგან (timestamp).

Cassandra Query Language

კასანდრასთან სამუშაოდ გამოიყენება SQL-ის თითქმის იდენტური სინტაქსი, რომელსაც ამ ტექნოლოგიის თავისებურებების გათვალისწინებით, CQL (Cassandra Query Language) ეწოდება. CQL-ის მეშვეობით, მომხმარებელი თავის საძიებო მოთხოვნას უგზავნის ერთ-ერთ რომელიმე node-ს. ეს უკანასკნელი ამ შემთხვევაში კოორდინატორის როლში გვევლინება და ასრულებს შუამავლის როლს კლიენტსა და იმ ნოუდებს შორის, რომლებზეც უშუალოდ მონაცემებია განთავსებული. [10]

დასასრულისკენ, შევაჯამოთ ძირითადი განსხვავებები კასანდრასა და ტრადიციულ რელაციურ ბაზას შორის:

RDBMS	Cassandra
აქვს ფიქსირებული სქემა. კომპლექსურ სტრუქტურებს მნიშვნელობებში ვერ გამოვიყენებთ (სიებს, სიმრავლეებს, მაგალითად)	აქვს მოქნილი სქემა. მნიშვნელობებში შეგვიძლია გვქონდეს ისეთი სტრუქტურები, როგორიცაა list-ები, set-ები, map-ები და ა. შ.
ცხრილი არის მასივების მასივი (სტრიქონი x სვეტი)	ცხრილი არის სია ერთმანეთში ჩადგმული გასაღებისა და შესაბამისი მნიშვნელობის წყვილებისა (სტრიქონი x სვეტის გასაღები x სვეტის მნიშვნელობა)
ბაზა არის მონაცემების ყველაზე გარეთა კონტეინერი	Keyspace არის მონაცემების ყველაზე გარეთა კონტეინერი
ბაზის ობიექტები არიან ცხრილები	Keyspace-ის ობიექტები არიან სვეტების ოჯახები
სტრიქონი არის ერთი ინდივიდუალური ჩანაწერი	სტრიქონი არის კასანდრას რეპლიკაციის ერთეული
სვეტი წარმოადგენს რელაციის ატრიბუტებს	სვეტი არის კასანდრაში შენახვის ერთეული
არსებობს foreign key-ების, join-ების კონცეფციები	--

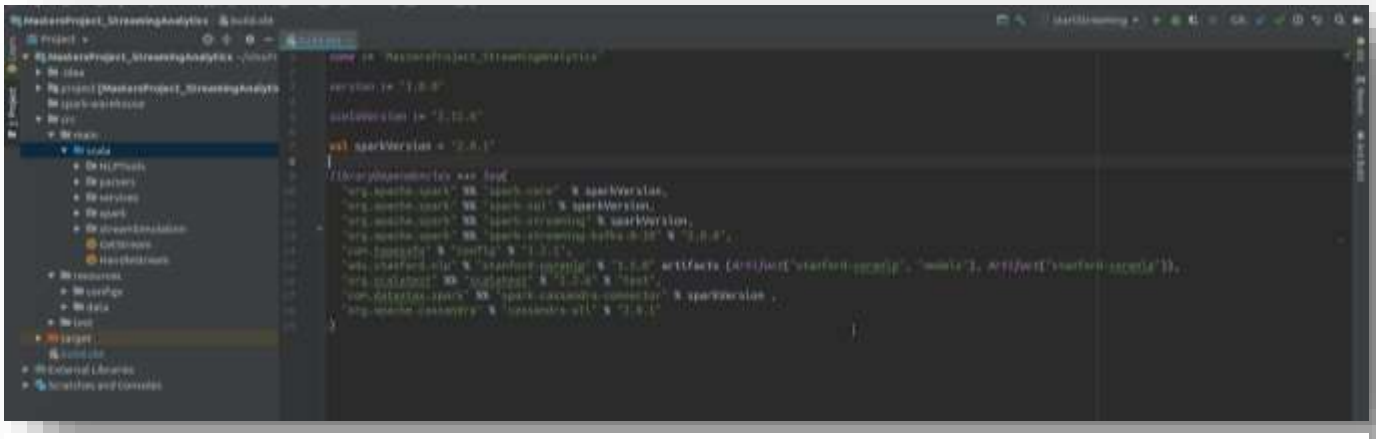
ამოცანის პრაქტიკული რეალიზაცია

პირველ რიგში, დასახული ამოცანის შესასრულებლად, ცხადია, გვჭირდება ყველა ზემოთხსენებული კომპონენტი – Apache Spark, Kafka, Cassandra და Zeppelin დავაინსტალიროთ და გავმართოთ. როგორც უკვე ვახსენეთ, ყველა ამ ტექნოლოგიის მთავარი უპირატესობება მასშტაბირებადობა და კლასტერზე განაწილებული კომპიუტინგია, თუმცა ასევე შესაძლებელია მათი ლოკალურად, პერსონალურ კომპიუტერზე ან ლეპტოპზე გაშვებაც. ნაშრომის მიზნებისათვის და სადემონსტრაციო რეალიზაციისათვის სწორედ ამ შესაძლებლობას გამოვიყენებ და ყველაფერს ლოკალურად, Ubuntu-ს გარემოში გავმართავ. IDE-დ ვიყენებ IntelliJ-ს, ხოლო შერჩეული ენაა Scala (ვერსია 2.11.6) – ფუნქციონალური და ობიექტზე ორიენტირებული JVM ენა. Spark მთლიანად Scala-ზეა დაწერილი და, როგორც წესი, სპარკში მუშაობა სწორედ ამ ენაზეა ყველაზე ოპტიმალური.

დავიწყოთ Apache Spark-ით. სპარკის სესიის გაშვება პირდაპირ IntelliJ-დან შეგვიძლია, თუკი შესაბამის ბიბლიოთეკებს დავამატებთ. ამისათვის Spark-ის სასურველი ვერსიისათვის (ჩვენს შემთხვევაში, ვერსიაა 2.0.1) ვამატებთ შესაბამის დამოკიდებულებას (dependency) build.sbt ფაილში (sbt – scala build tool), სადაც, ასე ვთქვათ, ვუთითებთ არტეფაქტის იდენტიფიკატორებს. აპლიკაციის building-ის დროს შესაბამისი jar ფაილები იძებნება Maven Repository-ში და ემატება external library-ებში. ასე რომ, ჩვენი აპლიკაციის და-build-ვის შემდეგ სპარკის ბიბლიოთეკები ხელმისაწვდომი გახდება. კონკრეტულად ეს დამოკიდებულებები გამოიყურება შემდეგნაირად:

```
val sparkVersion = "2.0.1"

libraryDependencies ++= Seq(
 "org.apache.spark" %% "spark-core"  % sparkVersion,
 "org.apache.spark" %% "spark-sql" % sparkVersion,
 "org.apache.spark" %% "spark-streaming" % sparkVersion
 ...)
```

სრული ნიმუშისათვის, გთხოვთ, იხილოთ თანდართული სქრინის (ილუსტრაცია 9, სადაც მოცემულია build.sbt მთლიანად) მე-10 - მე-12 ხაზები.



ილუსტრაცია 9 – build.sbt ფაილი

Kafka-სთან და Cassandra-სთან დაკავშირებისთვის ასევე გვჭირდება სათანადო ბიბლიოთეკები, რისთვისაც ვამატებთ შემდეგ დამოკიდებულებებს:

```
val sparkVersion = "2.0.1"
libraryDependencies ++= Seq(
 ...
"org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.0.0",
 ...
"com.datastax.spark" %% "spark-cassandra-connector" % sparkVersion ,
"org.apache.cassandra" % "cassandra-all" % "2.0.1"
)
```

თუმცა, რასაკვირველია, ამ დამოკიდებულებებს Kafka-სა და Cassandra-ს დაყენების და გაშვების გარეშე ვერაფერში გამოვიყენებთ.

Kafka-ს დასაყენებლად www-eu.apache.org-დან გადმოვწერთ შესაბამის არქივს და განვაარქივებთ სასურველ საქაღალდეში. კომფორტისათვის, შეგვიძლია გარემოს ცვლადებში (Enviroment Variable) დავამატოთ ცვლადი KAFKA_HOME. ამისათვის bash ფაილში დავამატე შემდეგი ხაზები:

```
export KAFKA_HOME=/usr/local/kafka

PATH=$KAFKA_HOME/bin:$PATH
```

აღსანიშნავია, რომ Kafka-ს მუშაობისთვის აუცილებლად სჭირდება Apache Zookeeper-ი. ეს უკანასკნელი არის ცენტრალიზებული ინფრასტრუქტურა და სერვისების ერთობლიობა, რომელიც გვეხმარება კლასტერზე განაწილებული სისტემების პროცესების მართვასა და სინქრონიზაციაში. Kafka-ს ლოკალურ რეჟიმში სამუშაოდაც Zookeper-ის სერვისების ჩართვა სავალდებულოა. ანალოგიურად ვიწერთ Zookeeper-ის არქივსაც Apache-ს საიტიდან, განვაარქივებთ სასურველ საქაღალდეში და გარემოს ცვლადებში ვამატებთ მნიშვნელობებს:

```
export ZOOKEEPER_HOME=/usr/local/zookeeper

PATH=$ZOOKEEPER/bin:$PATH
```

ასე რომ, bash ფაილის შესაბამისი ნაწილი შემდეგნაირად გამოიყურება (იხ. ილუსტრაცია 10):

```
export ZOOKEEPER_HOME=/usr/local/zookeeper
PATH=$ZOOKEEPER/bin:$PATH

export KAFKA_HOME=/usr/local/kafka
PATH=$KAFKA_HOME/bin:$PATH
```

ილუსტრაცია 10 გარემოს ცვლადები bash ფაილიდან

შევამოწმოთ, რომ Kafka ნამდვილად ეშვება. ამისთვის ჯერ გადავინაცვლოთ Zookeeper დირექტორიაში და ჩავრთოთ შესაბამისი პროცესები, შემდეგ გადავინაცვლოთ Kafka-ს დირექტორიაში და ამჟამად თავად Kafka გავუშვათ. შევამოწმოთ, რომ ორივე შესაბამის პორტებზე ნამდვილად გაეშვა (Zookeeper-ის ნაგულისხმევი პორტია 2181, Kafka-სი – 9092) და შევქმნათ კაფკას საცდელი topic-ი სახელად testtopic. წარმატებული დაყენების შემთხვევაში, უნდა მივიღოთ შემდეგი სურათი (ილუსტრაცია 11):

ილუსტრაცია 11კაფკას ინიციალიზაცია და ტოპიკის შექმნა

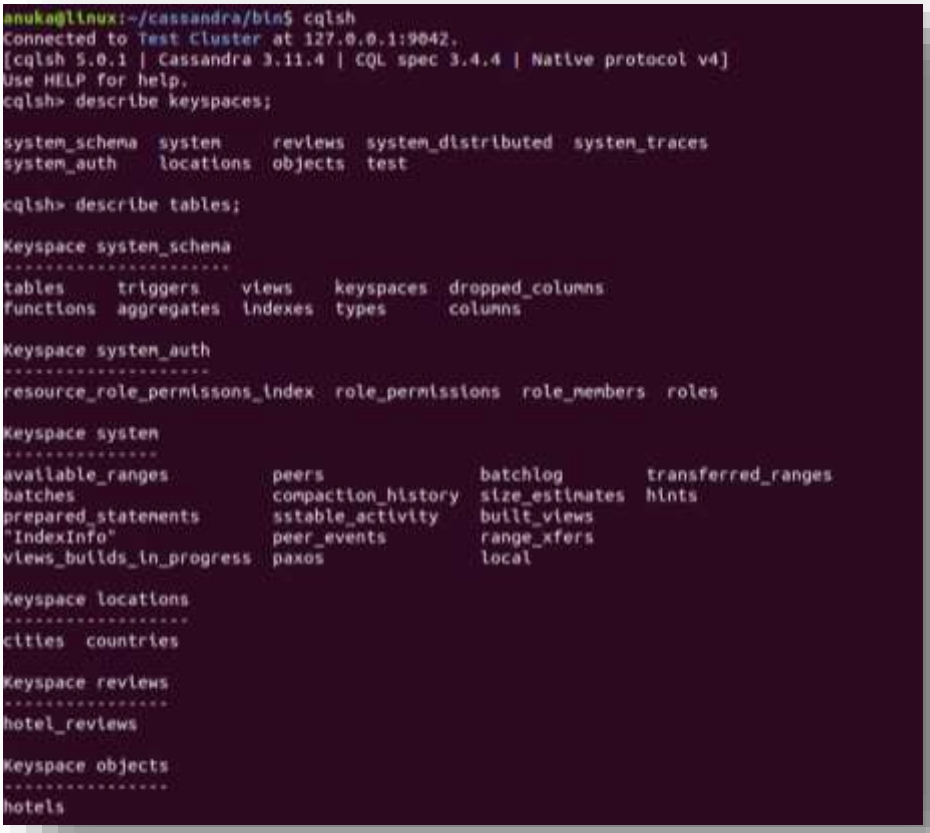
ანალოგიურად ვაყენებთ კასანდრასაც. ვიწერთ სასურველი ვერსიის არქივს www-eu.apache.org-დან და განვაარქივებთ რომელიმე საქაღალდეში. შემდეგ გარემოს ცვლადებში ვამატებთ ამ საქაღალდის მისამართს, რაც ჩვენს შემთხვევაში გამოიყურება შემდეგნაირად:

```
export CASSANDRA_HOME=~/cassandra

PATH=$PATH:$CASSANDRA_HOME/bin
```

ამის შემდეგ შეგვიძლია გავუშვათ კასანდრას პროცესები $CASSANDRA_HOME/bin დირექტორიაში cassandra ბრძანებით. წარმატების შემთხვევაში, სურათი მსგავსად გამოიყურება (ილუსტრაცია 12. და ეს ლოგების სრული ჩამონათვალი არაა):

ილუსტრაცია 12კასანდრას ინიციალიზაცია

ამის შემდეგ შეგვიძლია, cqlsh ბრძანების გაშვებით cql-ის გამოყენებით ,,ვეურთიერთოთ“ Cassandra-ს. მაგალითისთვის, ვნახოთ keyspace-ების და ცხრილების აღწერები (ილუსტრაცია 13):



ილუსტრაცია 13 ბრძანებას keyspace-ების და ცხრილების აღწერა

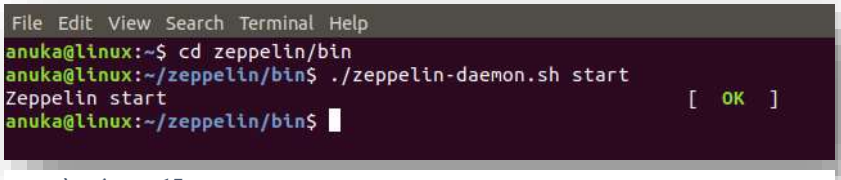
ასევე, შეგვიძლია, დავა-select-ოთ მონაცემები ცხრილებიდან (ილუსტრაცია 14):

```
cqlsh> select * from locations.countries;

 country_key | country_name
-------------+--------------
           1 |           US

(1 rows)
cqlsh> select * from locations.cities;

 city_key | city_name          | country_key
----------+--------------------+-------------
       23 |         Southfield |           1
       53 |              Mosca |           1
       91 |        Schenectady |           1
       55 |       Indianapolis |           1
       33 |               Bend |           1
        5 |            Anaheim |           1
       28 |           New York |           1
       42 |             Gorham |           1
       50 |            Gresham |           1
       95 |          Blythewood |           1
       88 |           Palmerton |           1
```

ილუსტრაცია 14

Zeppelin Notebook-საც მსგავსად ვიწერთ www-eu.apache.org-დან და ვაარქივებთ სასურველ საქაღალდეში. ვინაიდან Zeppelin-იდან სპარკისა და კასანდრას პროცესებს უნდა დავუკავშირდეთ, Zeppelin-ს ესაჭიროება გარკვეული დამაკავშირებლები, ე. წ. ინტერპრეტატორები. მათი ხელით დაყენება რომ არ მოგვიწიოს, სჯობს, გადმოვწეროთ Zeppelin-ის all interpreters ვერსია.

Zeppelin-ს ვუშვებთ ბრძანებით ./zeppelin-daemon.sh start თავის Home დირექტორიაში შემდეგნაირად (ილუსტრაცია 15):

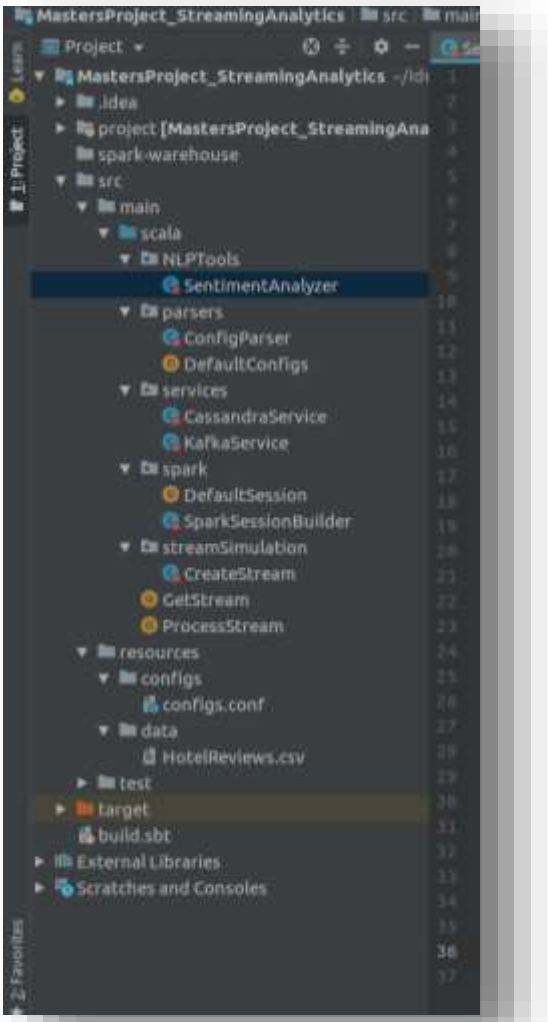


ილუსტრაცია 15

Zeppelin-თან მუშაობა შესაძლებელია ვებ ინტერფეისით. ნაგულისხმევი მნიშვნელობით, იგი ავტომატურად ეშვება 8080 პორტზე. ზეპელინის წარმატების გაშვების შემთხვევაში, ამ პორტზე შესვლის დროს უნდა დაგვხვდეს მსგავსი სურათი (ილუსტრაცია 16):

ილუსტრაცია 16 Zeppelin Home Page

მთავარი კომპონენტების აღწერის და გამართვის შემდეგ, გადავინაცვლოთ უშუალოდ სკალაზე შესრულებული პროექტის გარჩევაზე. დავიწყებ სტრუქტურის (ილუსტრაცია 17) მიმოხილვით.



ილუსტრაცია 17 პროგრამული კოდის სტრუქტურა

პროექტის main ნაწილი შედგება როგორც ჩემ მიერ შექმნილი კლასებისა და ობიექტებისაგან (მათ შორის, ორი მთავარია – GetStream და ProcessStream, ყველა სხვა დანარჩენი – დამხმარე), ასევე რესურსებისაგან. რესურსებში გაერთიანებულია:

1. ის მონაცემები (HotelReviews.csv ფაილი), რომლებისგანაც ნაკადის სიმულაცია უნდა შევქმნათ GetStream ობიექტის მეშვეობით. მონაცემები შინაარსობრივად წარმოადგენს აშშ-ის სხვადასხვა ქალაქში მდებარე სხვადასხვა სასტუმროს მომხმარებლების შეფასებებს. ფაილი შემდეგი ველებისაგან შედგება:

address – სასტუმროს მისამართი

city – სასტუმროს ქალაქი

country – სასტუმროს ქვეყანა

latitude - განედი

longitude - გრძედი

name – სასტუმროს დასახელება

review_rating – მომხმარებლის შეფასება (ქულა)

review_text – მომხმარებლის კომენტარი

review_title – შეფასების დასახელება

reviewer_username – მომხმარებლის სახელი

2. კონფიგურაციის ფაილი – configs.conf, რომელშიც თავმოყრილია ყველა ის პარამეტრი, რომელიც, ამ შემთხვევაში, ნაგულისხმევად უნდა გამოიყენოს პროექტის სხვადასხვა კომპონენტებმა და, საჭიროების შემთხვევაში, იოლად გადაკეთებადი იყოს მომხმარებლის მიერ. configs.conf ფაილის შიგთავსი შემდეგნაირია:

ილუსტრაცია 18 config.conf ფაილი

ობიექტი GetStream პასუხისმგებელია HotelReviews.csv ფაილიდან ნაკადის სიმულაციაზე. ამისათვის იგი უბრალოდ ქმნის streamSimulation package-ის CreateStream კლასის ობიექტს და მას გადასცემს config.conf ფაილს პარამეტრების ასაღებად. გადაცემულ სპარკის სესიაში CreateStream კლასი, პირველ რიგში, კითხულობს კონფიგურაციით

განსაზღვრულ ფაილს (HotelReviews.csv), რომლის თითოეული ჩანაწერი გადაჰყავს JSON ფორმატში და KafkaProducer ობიექტის მეშვეობით აგზავნის Kafka-ს შესაბამის Topic-ში (ეს უკანასკნელიც განსაზღვრულია კონფიგურაციით. კერძოდ, მისამართზე KafkaConfigs.defaultTopic = “reviews”).

ობიექტი ProcessStream პასუხისმგებელია ნაკადის დამუშავებაზე, კერძოდ კი Kafka-დან KafkaConsumer-ის მეშვეობით reviews ტოპიკიდან კითხულობს მესიჯებს (ამას უზრუნველყოფს services package-ის კლასი KafkaService), პარსავს JSON-ს და გარდაქმნის map სტრუქტურად, იღებს ველს review_text და ატარებს სენტიმენტის ანალიზს (ამას უზრუნველყოფს NLPTools package-ის კლასი SentimentAnalyzer), მიღებულ სენტიმენტს ამატებს map-ში, ისევ JSON-ში გარდაქმნის და გადასცემს Cassandra-ს სერვისის კლასს (services package-ის კლასი CassandraService), რომელიც ამ მესიჯის ბაზაში შექმნილი მოდელის მიხედვით გარდაქმნის მესიჯს და ჩაწერს. ამ ნაწილზე ცოტა ვრცლად ვისაუბროთ.

უხეშად რომ ვთქვათ, კასანდრა სწრაფ querying-ს იმის ხარჯზე აკეთებს, რომ აქ არ ხდება join-ები და მონაცემებს წინასწარ ისე ვინახავთ, რომ მაქსიმალურად კარგად მოერგოს ხშირად გამოყენებად query-ებს. აქედან გამომდინარე, Cassandra-ში, როგორც წესი, დენორმალიზებული ცხრილები გვხვდება ხოლმე. შეიძლება, რომ Kafka-სგან მიღებული მესიჯები ერთ ცხრილში ჩავყაროთ და სულაც არ ვიდარდოთ ინფორმაციის გადუბლირებაზე, თუმცა, ვინაიდან კასანდრას სპარკთანაც ვაინტეგრირებთ და შეგვიძლია, join-ების ნაწილი სპარკს მივანდოთ, მთლიანი მესიჯის ერთ ცხრილში ჩაწერა არ მიმაჩნია მიზანშეწონილად. ამიტომ კაფკადან წაკითხული მესიჯის დაპარსვის მერე იგი კასანდრას სხვადასხვა ცხრილში ჩავწეროთ.

როგორც ზემოთ ვახსენეთ, სპარკში ნაკადიდან შემომავალი მესიჯი მუშავდება და ემატება სენტიმენტის აღმწერი ველი. შესაბამისად, როცა დამუშავებულ მესიჯს უკვე CassandraService-ს გადავცემთ ბაზაში ჩასაწერად, გვაქვს შემდეგი ველების მქონე JSON შეტყობინება:

address – სასტუმროს მისამართი
city – სასტუმროს ქალაქი
country – სასტუმროს ქვეყანა

latitude - განედი

longitude - გრძედი

name – სასტუმროს დასახელება

review_rating – მომხმარებლის შეფასება (ქულა)

review_text – მომხმარებლის კომენტარი

review_title – შეფასების დასახელება

reviewer_username – მომხმარებლის სახელი

sentiment – review_text ველის სეტიმენტი

ისეთი ველები, როგორიცაა ქვეყანა, ქალაქი, გრძედი, განედი, სასტუმროს შესახებ ინფორმაცია დამოუკიდებელ ცხრილებში (ცნობარებში) გავიტანოთ და უნიკალური იდენტიფიკატორები მივანიჭოთ. მაგალითად, პირველი ცხრილი, სახელად countries, არის ქვეყნების ცნობარი, სადაც გვაქვს ველები [country_key - იდენტიფიკატორი, country_name – ქვეყნის დასახელება]; მომდევნო ცხრილი, cities, ქალაქების ცნობარია ველებით [city_key - იდენტიფიკატორი, city_name – ქალაქის დასახელება, country_key – იმ ქვეყნის იდენტიფიკატორი, სადაც ეს ქალაქი მდებარეობს]; ცალკე ცხრილი, სახელად, hotels, გვაქვს სასტუმროებისთვისაც ველებით [hotel_key - სასტუმროს იდენტიფიკატორი, city_key – ქალაქის იდენტიფიკატორი, hotel_address – სასტუმროს მისამართი, hotel_name – სასტუმროს დასახელება, latitude – განედი, longitude – გრძედი]. ბოლო ცხრილი, hotel_reviews, კი უშუალოდ მომხმარებელთა შეფასებისაა ველებით [review_key – შეფასების იდენტიფიკატორი, hotel_key – სასტუმროს იდენტიფიკატორი, rating – შეფასება (ქულა), review_date – შეფასების თარიღი, review_text – მომხმარებლის კომენტარი, review_title - სათაური, reviewer_name – შემფასებლის სახელი, sentiment – სენტიმენტი].

კასანდრაში ზემოთხსენებული ცხრილები სამ სხვადასხვა keyspace-შია განაწილებული. კერძოდ, ცხრილები countries და cities მდებარეობს locations keyspace-ში, ცხრილი hotels – objects keyspace-ში (ეს ზოგადი სახელი იმიტომ გადავწყვიტე, რომ არაა აუცილებელი, მარტო სასტუმროები იყოს. შეიძლება რაიმე სხვა ბაზაში და, პირველ რიგში, ბიზნესში სხვა ობიექტებიც იყოს, მაგალითად, რესტორნები და ა. შ.), შეფასებები კი – reviews-ს keyspace-ში.

კასანდრას describe ბრძანების მეშვეობით შეიძლება keyspace-ების თუ ცალკეული ცხილების აღწერა. describe keyspaces ბრძანების გაშვების შემდეგ ზემოთხსენებული keyspace-ები, სხვა სისტემურ თუ სატესტოდ შექმნილ keyspace-ებთან ერთად შემდეგნაირად გამოიყურება კასანდრაში:

```
cqlsh> describe keyspaces;

system_schema  system     reviews  system_distributed  system_traces
system_auth    locations  objects  test
```

ილუსტრაცია 19 კასანდრაში Keyspace-ების აღწერა

describe tables ბრძანების მეშვეობით ვნახავთ, თითოეულ keyspace-ში რა ცხრილები შედის:

```
cqlsh> describe tables;

Keyspace system_schema
----------------------
tables     triggers    views    keyspaces  dropped_columns
functions  aggregates  indexes  types      columns

Keyspace system_auth
--------------------
resource_role_permissons_index  role_permissions  role_members  roles

Keyspace system
---------------
available_ranges          peers              batchlog        transferred_ranges
batches                   compaction_history  size_estimates  hints
prepared_statements       sstable_activity    built_views
"IndexInfo"               peer_events         range_xfers
views_builds_in_progress  paxos              local

Keyspace locations
------------------
cities  countries

Keyspace reviews
----------------
hotel_reviews

Keyspace objects
----------------
hotels

Keyspace system_distributed
---------------------------
repair_history  view_build_status  parent_repair_history

Keyspace test
-------------
kv

Keyspace system_traces
----------------------
events  sessions
```

ილუსტრაცია 20 ცხრილების აღწერა კასანდრაში

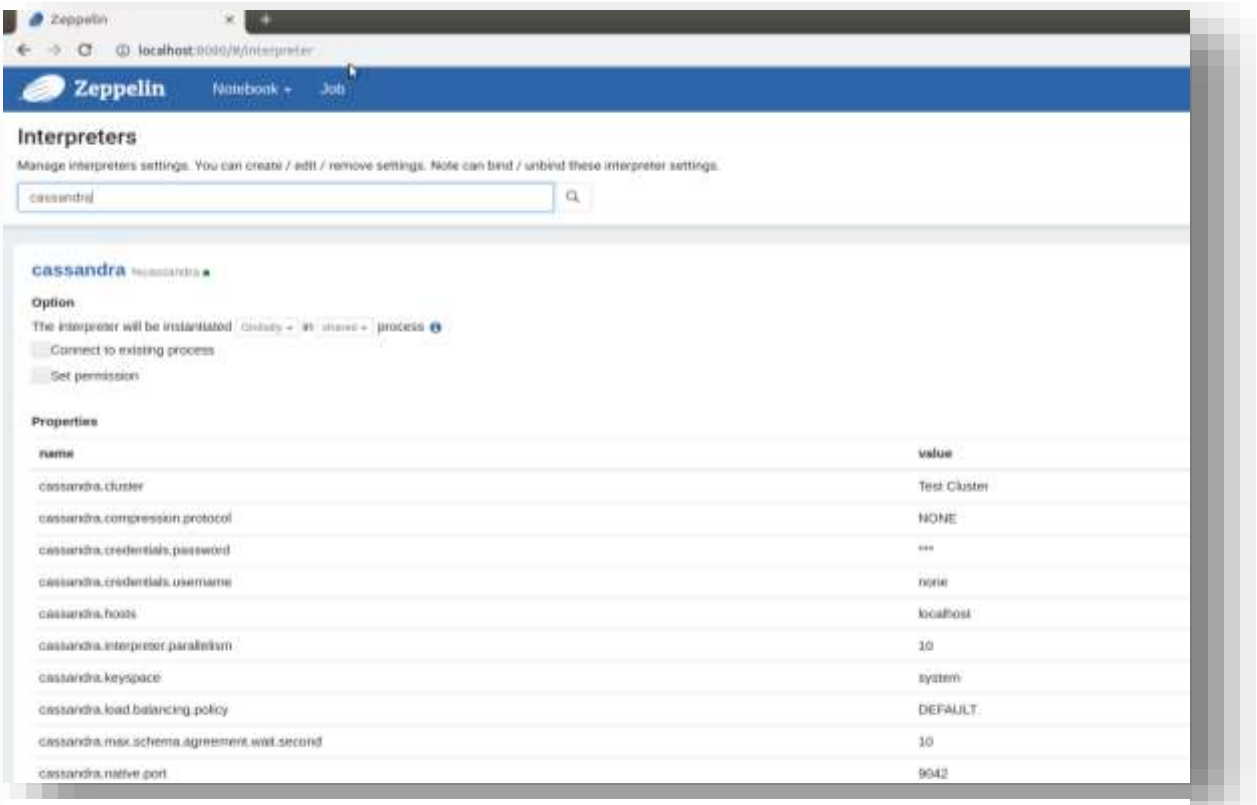
ბრძანებით describe <keyspace_name>-თი კი შესაძლებელია კონკრეტული keyspace-ში შემავალი ცხრილების უფრო დეტალური აღწერა, კერძოდ, CREATE TABLE DDL ბრძანების სახით. მაგალითად, ბრძანება describe locations-ზე კასანდრას დაბრუნებული შედეგი შემდეგნაირად გამოიყურება:

```
cqlsh> describe locations;

CREATE KEYSPACE locations WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}  AND durable_writes = true;

CREATE TABLE locations.cities (
    city_key int PRIMARY KEY,
    city_name text,
    country_key int
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';

CREATE TABLE locations.countries (
    country_key int PRIMARY KEY,
    country_name text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

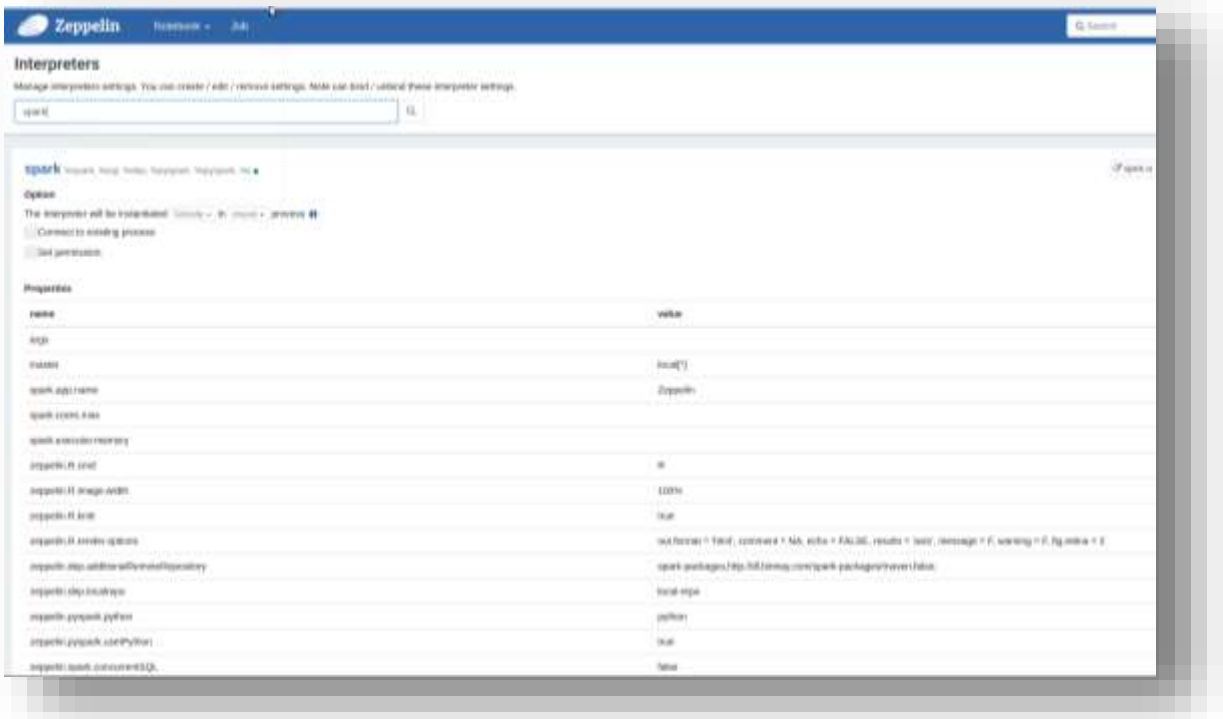
ილუსტრაცია 21 ცხრილების დეტალური აღწერა კასანდრაში

ისევ პროგრამულ კოდს რომ დავუბრუნდეთ, კლასი CassandraService მიღებულ მესიჯს პარსავს და აღწერილ ცხრილებში წერს. ცნობარების შემთხვევაში, ჯერ ამ ცნობარებს კითხულობს ბაზიდან და ამოწმებს, შემომავალ მესიჯში ნახსენები ობიექტი, მაგალითად, ქვეყანა, ქალაქი ან სასტუმრო უკვე ხომ არ არსებობს. თუ კი, მაშინ ახალი ჩანაწერების ჩამატება საჭირო არაა და შესაბამისი მეთოდი აბრუნებს ამ ობიექტის იდენტიფიკატორს. უშუალოდ მომხმარებლის შეფასებებს კი ცხრილ hotel_reviews-ში პირდაპირ ამატებს მას შემდეგ, რაც სასტუმროს იდენტიფიკატორს განსაზღვრავს. ამ კლასის მთელი დანიშნულება სწორედ ესაა.

ბოლოს, მოკლედ Zeppelin-ის შესახებაც. ნაშრომის წინა ნაწილში ვახსენეთ, რომ მასთან ვებ ინტერფეისით ვურთიერთობთ. ნაშრომის მიზნებისთვის, გვჭირდება, რომ დაინსტალირებული გვქონდეს კასანდრასა და სპარკის ინტერპრეტერები. ამაში დასარწმუნებლად, ზეპელინის interpreter-ს გვერდზე გადავიდეთ და შევამოწმოთ:



ილუსტრაცია 23 კასანდრას ინტერპრეტერი ზეპელინში



ილუსტრაცია 22 სპარკის ინტერპტერი ზეპელინში

ორივე ინტერპრეტერის გასწვრივ პატარა წრის სიმწვანე ნიშნავს, რომ ინტერპრეტერი გამართულია და სესია ხელმისაწვდომია კასანდრას შემთხვევაში, %cassandra, ხოლო სპარკის შემთხვევაში - %spark (%sql, %pyspark, ...) ბრძანებებით.

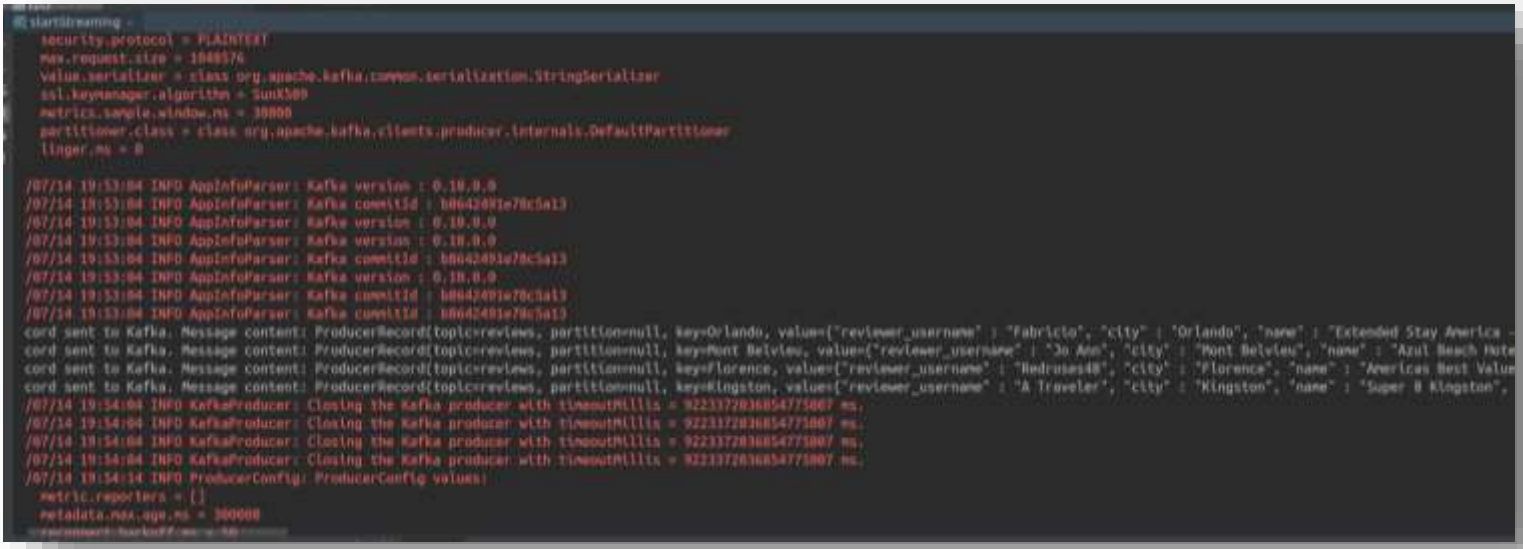
რახან უკვე აღვწერე ნაშრომის ყველა ძირითადი კომპონენტი და პროგრამული კოდი (რომლის ძირითადი კლასები დანართში შეგიძლიათ იხილოთ), შევეცდები, ქვემოთ მოცემული ილუსტრაციების (ე. წ. Screen-ების) სახით გაჩვენოთ იგი მოქმედებაში:

1. **ნაკადის სიმულაცია და მესიჯების ჩაწერა Kafka-ში:**

1.1 შეიქმნა Spark სესია მონაცემების .csv ფაილიდან წასაკითხავად. შეგვიძლია შევამოწმოთ Spark UI-ს მეშვეობით:

ილუსტრაცია 24 სპარკის UI -ის სქრინი ნაკადების სიმულაციის დროს

1.2 კონსოლის Log-ებიდან დავრწმუნდეთ, რომ მესიჯები ნამდვილად იკითხება და იგზავნება Kafka-ში. ამისთვის კონსოლში, ბევრ სხვა ინფორმაციასთან ერთად, თითოეული მესიჯისათვის იბეჭდება შეტყობინება - ,,Record sent to Kafka. Message content: …'' და მესიჯის შინაარსი. მაგალითად:



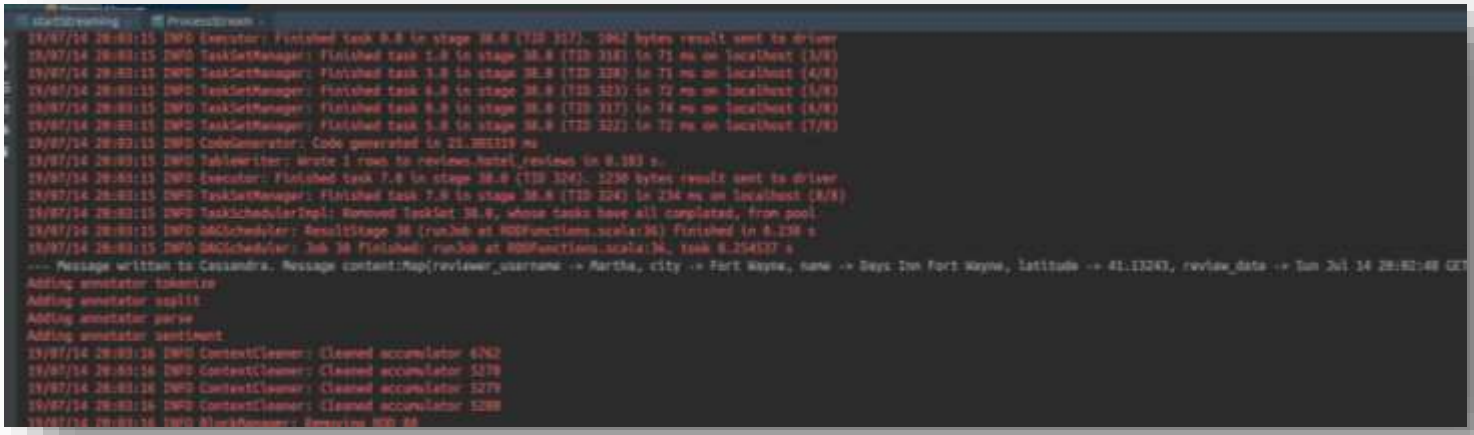
ილუსტრაცია 25 მესიჯების ჩაწერა კაფკაში

1.3 ტერმინალიდან გადავამოწმოთ, რომ მესიჯები Kafka-ში ნამდვილად იყრება:

ილუსტრაცია 26 მესიჯები კაფკაში

2. ნაკადის დამუშავება, სენტიმენტ ანალიზი და Cassandra-ში ჩაწერა:

2.1 მონაცემების წაკითხვა Kafka-დან Spark-ში, სენტიმენტის ანალიზი და კასანდრაში ჩაწერა. წარმატების შემთხვევაში, კონსოლში უნდა ვხედავდეთ შეტყობინებას ,,Message written to Cassandra. Message Content: …" და შემდეგ მესიჯის შინაარსი.



ილუსტრაცია 27 მესიჯის ჩაწერა კასანდრაში

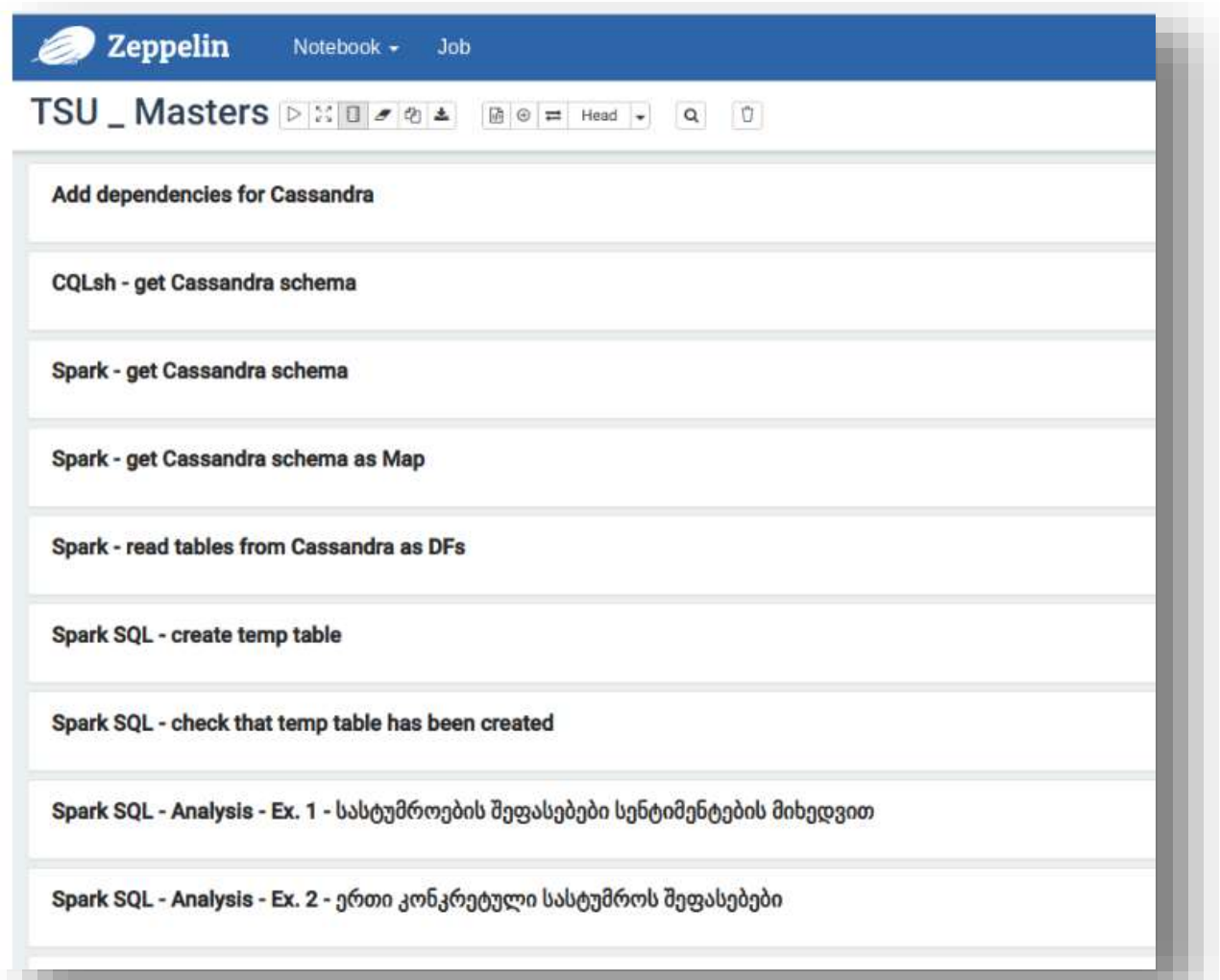
2.2 დავრწმუნდეთ, რომ Cassandra-ს ბაზა ნამდვილად ივსება მონაცემებით. მაგალითად, რამდენიმეწამიანი შუალედებით დავა-select-ოთ რაოდენობა ცხრილიდან hotel_reviews:

```
cqlsh> select count(*)  from reviews.hotel_reviews;

 count
-------
   158

(1 rows)

Warnings :
Aggregation query used without partition key

cqlsh> select count(*)  from reviews.hotel_reviews;

 count
-------
   176

(1 rows)

Warnings :
Aggregation query used without partition key

cqlsh> select count(*)  from reviews.hotel_reviews;

 count
-------
   181
```

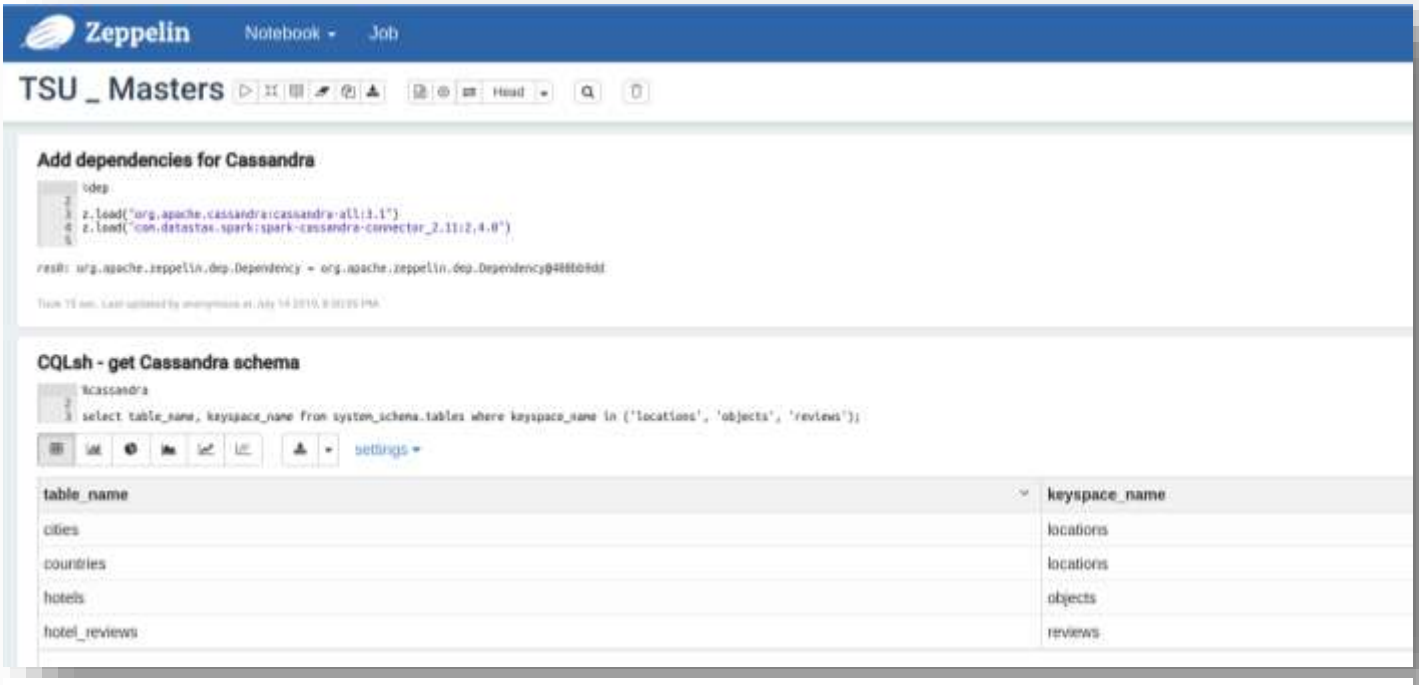
ილუსტრაცია 28

3. **Zeppelin-დან მუშაობა:**

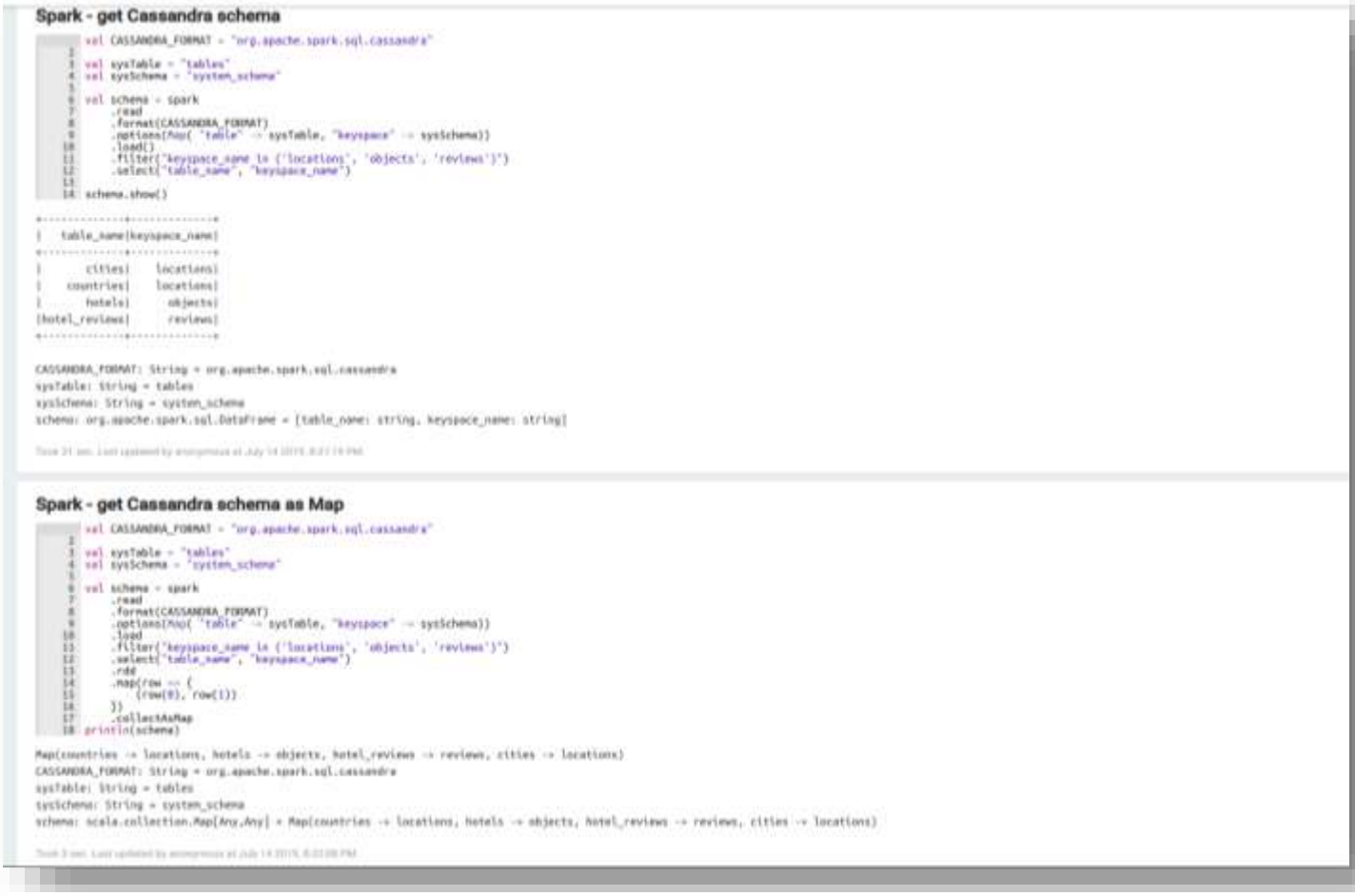
3.1 როგორც მოცემული სურათიდან ჩანს, ზეპელინში წინასწარ მაქვს რამდენიმე ბრძანება გამზადებული. პირველი მათგანი საჭიროა Spark-იდან Cassandra-სთან დასაკავშირებელი ბიბლიოთეკების ჩამოსაწერად. მომდევნო ბრძანებები კი უბრალოდ იმის გასატესტავად, რომ ნამდვილად ვუკავშირდებით Cassandra-ს ბაზას და სქემასაც სწორად ვკითხულობთ (ილუსტრაციები 29-31):



ილუსტრაცია 29



ილუსტრაცია 31



ილუსტრაცია 30

3.2 როგორც ნაშრომის ზედა ნაწილში ვთქვით, კასანდრაში join-ები არ კეთდება, მაგრამ ეს შესაძლებელია სპარკში. სადემონსტრაციოდ, წავიკითხოთ თითოეული ცხრილი დატაფრეიმად, დავაჯოინოთ, მივიღოთ ერთი დატაფრეიმი სახელად reviewsDF და ზეპელინში გამოვიტანოთ შედეგი. ვხედავთ, რომ აქვე არის ველი sentiment-იც:

ილუსტრაცია 32

3.3 თუკი დატაფრეიმებთან მუშაობა სპარკში ასე თუ ისე უხერხულობას გვიქმნის, შესაძლებლობა გვაქვს, sql-ზეც ვწეროთ. ამისთვის გვჭირდება, რომ სასურველი დატაფრეიმი დროებითი ცხრილის სახით დავარეგისტრიროთ. მაგალითად, ავიღოთ დატაფრეიმი reviewsDF და დავარქვათ სახელი tempReviews, რასაც ვაკეთებთ შემდეგნაირად:

Spark SQL - create temp table

```
reviewsDF.registerTempTable("tempReviews")
```

```
warning: there was one deprecation warning; re-run with -deprecation for details
```

Took 2 sec. Last updated by anonymous at July 14 2019, 9:06:25 PM.

Spark SQL - check that temp table has been created

```
%sql
select *
from tempReviews
limit 1
```

country_key	city_key	hotel_key	review_key	rating	review_date	review_text	review_title	reviewer_name	sentiment
1	31	32	206	4	Sun Jul 14 20:21:40 GET 2019	You may think you're lost as you drive the long and winding road to the Meadow Farm. Stay the course and you'll be rewarded. Have stayed here about 4 times over the past few years visiting our children at summer camp. The owners, Janet and	Wonderful, warm, comfortable stay!	CoachBrez	Very Positive

ილუსტრაცია 33

3.4 დასასრულს, შეგვიძლია სტანდარტული sql სინტაქსის სახით ვიმუშავოთ tempReviews ცხრილთან. მაგალითად, შემდეგნაირად:

Spark SQL - Analysis - Ex. 1 - სასტუმროების შეფასებები სენტიმენტების მიხედვით

```
%sql

select
    hotel_key,
    hotel_name,
    sentiment,
    count(1) as count
from tempReviews
group by
    hotel_key,
    hotel_name,
    sentiment
order by
    1, 2, 3
```

hotel_key	hotel_name	sentiment	count
1	Red Roof Inn Erie	Negative	3
2	Americas Best Value Inn	Negative	4
3	Centennial Bed and Breakfast	Negative	2
4	Fiesta Inn and Suites	Negative	1
4	Fiesta Inn and Suites	Neutral	4
5	Howard Johnson Inn - Newburgh	Negative	11
5	Howard Johnson Inn - Newburgh	Neutral	1

ილუსტრაცია 34

დასკვნა

დიდ მონაცემების სამყარო საკმაოდ მრავალფეროვანია და კონკრეტული ტექნოლოგიური ამოცანის გადასაწყვეტად სპეციალიზირებულ ინსტრუმენტთა დიდ არჩევანს გვთავაზობს. მათ აბსოლუტურ უმეტესობას ღია პროგრამული უზრუნველყოფები წარმოადგენენ, რომლებიც დიდ მონაცემებს ძალიან ეფექტურად და, ამასთანავე, ნაშრომის დასაწყისში აღწერილი CAP თეორემის შესაბამისად, საიმედოდ უმკლავდებიან. თუმცა, ვინაიდან თითოეული მათგანი, შეიძლება ითქვას, რომ საკმაოდ ვიწრო მიზანს ემსახურება, ორგანიზაციაში დიდ მონაცემთა გარემოს აწყობის მთავარი გამოწვევა, არქიტექტურული თუ უშუალოდ იმპლემენტაციის თვალსაზრისით, სწორედ მრავალი კომპონენტის ერთმანეთთან დაკავშირებაა ისე, რომ ბიზნესის საჭიროებებსაც სრულად ფარავდეს და ამასთან, მაქსიმალურად დაცული იყოს ხარვეზებისაგან.

ის, რაც ნაშრომში თეორიულად იქნა განხილული და პრაქტიკულად განვახორციელებული, ერთი დიდი სურათის მნიშვნელოვანი, თუმცა პატარა ნაწილია: ცხადია, მხოლოდ მონაცემთა ნაკადების რეალურ დროში დამუშავების ფენა ვერ იქნება რომელიმე ორგანიზაციის მთელი Big Data გარემო. მიუხედავად ამისა, ვფიქრობ, საკმაოდ საფუძვლიანად განვიხილეთ ის ძირითადი კომპონენტები, რომლებზეც აღნიშნული არქიტექტურული ფენა შეიძლება დაფუძნდეს და რომლებიც, ჩემ მიერ მიმოხილული ლიტერატურის და მცირე სამუშაო გამოცდილების საფუძველზე, მგონია, რომ საუკეთესო პრაქტიკადაც შეიძლება ჩაითვალოს. აღსანიშნავია, რომ ნაშრომის პრაქტიკული ნაწილი გაკეთდა კონკრეტული ამოცანის მაგალითზე. აქედან გამომდინარე, მომავალში ვხედავ შესაძლებლობას, უფრო სიღრმისეულად შევისწავლოთ განსხვავებული ბიზნესების საჭიროებები და მოთხოვნები მონაცემების რეალურ დროში დამუშავებასთან მიმართებაში, შეძლებისდაგვარად განვაზოგადოთ, დავაჯგუფოთ გარკვეული პრინციპის მიხედვით, შევქმნათ გარკვეული მოდელები და უკვე ამ მოდელებიდან გამომდინარე ორგანიზაციების შევთავაზოთ მზა არქიტექტურული გადაწყვეტილებები კონკრეტული პროგრამული უზრუნველყოფების გათვალისწინებით (ასე ვთქვათ, შუალედური შედეგი ნაშრომში მიმოხილულ ორ არქიტექტურას შორის და პრაქტიკულ ნაშრომში წარმოდგენლი კონკრეტული ამოცანის კონკრეტულ რეალიზაციას შორის). ვფიქრობ, ეს მიღწევადია სათანადო რესურსების არსებობისა და კვლევის განხორციელების შემთხვევაში.

ბიბლიოგრაფია

[1] B. Franks, Taming the Big Data Tidal Wave: Finding Opportunities in Huge Data Streams with Advanced Analytics, New Jersey: John Wiley & Sons, Inc., 2012.

[2] Big Data Framework, "The 4 Characteristics of Big Data," Big Data Framework, 12 March 2019. [Online]. Available: https://www.bigdataframework.org/four-vs-of-big-data/. [Accessed June 2019].

[3] D. Mysore, K. Shrikant and J. Shweta, "Understanding the architectural layers of a big data solution," IBM, 15 October 2013. [Online]. Available: https://developer.ibm.com/articles/bd-archpatterns3/. [Accessed June 2019].

[4] RCV Academy, "Big Data Layers – Data Source, Ingestion, Manage and Analyze Layer," RCV Academy, 22 November 2016. [Online]. Available: https://www.rcvacademy.com/big-data/big-data-layers/. [Accessed June 2019].

[5] S. S. Nazrul, "CAP Theorem and Distributed Database Management Systems," Towards Data Science, 25 April 2018. [Online]. Available: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e. [Accessed June 2019].

[6] J. Forgeat and E. Feller, "Data processing architectures – Lambda and Kappa," ericcson, 19 November 2015. [Online]. Available: https://www.ericsson.com/en/blog/2015/11/data-processing-architectures--lambda-and-kappa. [Accessed June 2019].

[7] B. Chambers and M. Zaharia, Spark: The Definitive Guide, Sebastopol: O'Reilly Media, 2018.

[8] Apache Kafka, "Apache Kafka," Apache Kafka, [Online]. Available: https://kafka.apache.org/. [Accessed May 2019].

[9] N. Narkhede, G. Shapira and T. Palino, Kafka: The Definitive Guide - Real-Time Data and Stream Processing at Scale, Sebastopol: O'Reilly Media, 2018.

[10] tutorialspoint, "Cassandra Tutorial," tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/cassandra/. [Accessed June 2019].

დანართი

პროგრამული კოდი

Package: NLPTools

Class/object Name: SentimentAnalyzer

Code:

```
1.  package NLPTools
2.
3.  import java.util.Properties
4.
5.  import edu.stanford.nlp.ling.CoreAnnotations
6.  import edu.stanford.nlp.neural.rnn.RNNCoreAnnotations
7.  import edu.stanford.nlp.pipeline.StanfordCoreNLP
8.  import edu.stanford.nlp.sentiment.SentimentCoreAnnotations
9.  import edu.stanford.nlp.util.CoreMap
10.
11. import scala.collection.JavaConversions._
12.
13. // სენტიმენტის დეტექტორი კლასი. გადაცემული ტექსტისთვის აბრუნებს ხუთი შესაძლებელი სე
    ნტიმენტიდან ერთ-ერთს:
14. // "Very Negative", "Negative", "Neutral", "Positive", "Very Positive"
15. class SentimentAnalyzer{
16.
17.     def detectSentiment(text: String): String = {
18.         var mainSentiment = 0
19.         var longest = 0
20.         val sentimentText = Array("Very Negative", "Negative", "Neutral", "Positive", "
    Very Positive")
21.         val props = new Properties()
22.         props.setProperty("annotators", "tokenize, ssplit, parse, sentiment")
23.
24.
25.         new StanfordCoreNLP(props).process(text).get(classOf[CoreAnnotations.SentencesA
    nnotation]).foreach((sentence: CoreMap) => {
26.             val sentiment = RNNCoreAnnotations.getPredictedClass(sentence.get(classOf[S
    entimentCoreAnnotations.AnnotatedTree]))
27.             val partText = sentence.toString
28.             if (partText.length > longest) {
```

```
                mainSentiment = sentiment
                longest = partText.length
            }
        })

        sentimentText(mainSentiment)
    }
}
```

Package: parsers

Class/object Name: ConfigParser

Code:

```
package parsers

import java.io.File

// კონფიგურაციის ფაილების პარსერი. გადაეცემა კონფიგ ფაილის მისამართი
class ConfigParser(configFile: String) extends Serializable {
    import com.typesafe.config.ConfigFactory

    // პარსავს გადმოცემულ მისამართზე მდებარე კონფიგ ფაილს
    private val parsedConfigFile = ConfigFactory.parseFile(new File(configFile))

    // გადაეცემა path. აბრუნებს კონფიგ ფაილიდან ამ path-ის შესაბამის მნიშვნელობას
    def getAt(path: String): String = {
            parsedConfigFile.getString(path)
    }

    def confToString(): String = {
```

```
18.         parsedConfigFile.toString
19.     }
20.
21. }
```

Package: parsers

Class/object Name: DefaultConfigs

Code:

```
1.  package parsers
2.
3.  // ნაგულისხმევი მნიშვნელობით მითითებული მისამართიდან კითხულობს კონფიგურაციის ფაილ
    ს და ქმნის ConfigParser კლასის ობიექტს
4.  object DefaultConfigs {
5.      val confFilePath = "/home/anuka/IdeaProjects/MastersProject_StreamingAnalytics/src/
    resources/configs/configs.conf"
6.      val defConf = new ConfigParser(confFilePath)
7.  }
```

Package: services

Class/object Name: KafkaService

Code:

```
1.  package services
2.
3.  import java.util
4.  import java.util.Properties
5.
6.  import org.apache.kafka.clients.consumer.{ConsumerRecord, KafkaConsumer}
7.  import parsers.{ConfigParser, DefaultConfigs}
8.
```

```
9.  import scala.collection.JavaConverters._
10. 
11. // ეს კლასი უზრუნველყოფს Kafka-
    სთან ურთიერთობისთვის (კერძოდ, მესიჯების წაკითხვისთვის) საჭირო ფუნქციონალს.
12. // კონფიგურაციები Kafka-სთან დასაკავშირებლად გადაეცემა არგუმენტების სახით.
13. class KafkaService(conf: ConfigParser  = DefaultConfigs.defConf){
14. 
15.     // ქმნის properties ფაილს გადმოცემული კონფიგურაციების მიხედვით
16.     val props = new Properties()
17.     props.put("bootstrap.servers", conf.getAt("KafkaConfigs.brokers"))
18.     props.put("key.deserializer", conf.getAt("KafkaConfigs.keyDeserializer"))
19.     props.put("value.deserializer", conf.getAt("KafkaConfigs.valueDeserializer"))
20.     props.put("auto.offset.reset", conf.getAt("KafkaConfigs.offsetResetMode"))
21.     props.put("group.id", conf.getAt("KafkaConfigs.consumerGroup"))
22. 
23.     // ქმნის  Consumer-
    ს String ტიპის მესიჯების წასაკითხავად (შენიშვნა: Key და Value - ორივე String-
    ია) ზემოთ აღწერილი properties ფაილის მიხედვით
24.     val consumer: KafkaConsumer[String, String] = new KafkaConsumer[String, String](pro
    ps)
25. 
26.     // გადაეცემა იმ ტოპიკის სახელი, რომელიც ზემოთ შექმნილმა consumer-
    მა უნდა ,,გამოიწეროს'' (subscribe)
27.     def subscribeToTopic(topic: String): Unit = {
28.         consumer.subscribe(util.Arrays.asList(topic))
29.     }
30. 
31.     // Consumer-
    მა პარამეტრად გადაცემული timeout დროის შუალედებში უნდა მოაგროვოს და დააბრუნოს შეტყ
    ობინებები Kafka-დან
32.     def pollMessages(pollingTimeout: Int = conf.getAt("KafkaConfigs.streamingInterval")
    .toInt): Iterable[ConsumerRecord[String, String]] = {
33.         val records = consumer.poll(pollingTimeout).asScala
34. 
35.         records
36.     }
37. 
38. }
```

Package: services

Class/object Name: CassandraService

Code:

```
1.  package services
2.
3.  import org.apache.spark.sql.{DataFrame, SaveMode, SparkSession}
4.  import org.apache.spark.sql.functions.max
5.
6.  import scala.util.parsing.json.JSON
7.
8.  class CassandraService(spark: SparkSession) {
9.
10.     import spark.implicits._
11.     private val CASSANDRA_FORMAT = "org.apache.spark.sql.cassandra"
12.
13.     // კასანდრას ბაზიდან DataFrame-
    ად კითხულობს პარამეტრებში მითითებულ ცხრილს (პარამეტრებად გადაეცემა keyspace და tab
    le).
14.     def cassandraTableAsDF(keyspace: String, table: String): DataFrame = {
15.         val df = spark
16.             .read
17.             .format(CASSANDRA_FORMAT)
18.             .options(Map("table" -> table, "keyspace" -> keyspace))
19.             .load()
20.
21.         df
22.     }
23.
24.     // გადაეცემა DataFrame, რომელსაც წერს კასანდრას ბაზაში პარამეტრებად მითითებული ke
    yspace-ის table-ში.
25.     def saveDFtoCassandraTable(df: DataFrame, keyspace: String, table: String): Unit =
    {
26.         df
27.             .write
28.             .format(CASSANDRA_FORMAT)
29.             .options(Map("keyspace"-> keyspace,"table"-> table))
30.             .mode(SaveMode.Append)
```

```
31.             .save()
32.     }
33.
34.     // გადმოეცემა მესიჯი JSON ფორმატის სახით, რომელსაც პარსავს და დამხმარე მეთოდების საშუალებით ავსებს ცხრილებს:
35.     // locations.countries, locations.cities, objects.hotels და reviews.hotel_reviews

36.     def saveMessageToCassandra(message: String): Unit = {
37.         val parsedMessage = JSON.parseFull(message).get.asInstanceOf[Map[String, String]]
38.         getCountryAndCityKeys(parsedMessage("city"), parsedMessage("country"))
39.
40.         val hotelKey = getHotelKey(parsedMessage("name"), parsedMessage("address"), parsedMessage("city"), parsedMessage("country"), parsedMessage("latitude").toDouble, parsedMessage("longitude").toDouble)
41.         loadReviewsToCassandra(hotelKey, parsedMessage("review_rating").toDouble, parsedMessage("review_date"), parsedMessage("review_text"), parsedMessage("review_title"), parsedMessage("reviewer_username"), parsedMessage("sentiment"))
42.
43.         println("--- Message written to Cassandra. Message content:" + parsedMessage)
44.     }
45.
46.     // წერს შეფასებებს კასანდრას ბაზაში, ცხრილში reviews.hotel_reviews
47.     def loadReviewsToCassandra(hotelKey: Int, rating: Double, reviewDate: String, reviewText: String, reviewTitle: String, reviewerName: String, sentiment: String): Unit = {

48.         var reviewKey = 0
49.         val hotelsDF = cassandraTableAsDF("reviews", "hotel_reviews")
50.
51.         val maxKeyDF = hotelsDF
52.             .select(max("review_key")).alias("maxKey")
53.
54.         if (!maxKeyDF.rdd.isEmpty)
55.             reviewKey = maxKeyDF.first.get(0).asInstanceOf[Int] + 1
56.
57.         val record = spark
58.             .sparkContext
59.             .parallelize(Seq((reviewKey, hotelKey, rating, reviewDate, reviewText, reviewTitle, reviewerName, sentiment)))
```

```
60.             .toDF("review_key", "hotel_key", "rating", "review_date", "review_text", "r
    eview_title", "reviewer_name", "sentiment")
61.
62.         saveDFtoCassandraTable(record, "reviews", "hotel_reviews")
63.     }
64.
65.     // გადაეცემა სასტუმროს შესახებ ინფორმაცია. პირველ რიგში, ამოწმებს, ეს სასტუმრო არსე
    ბობს
66.     // ბაზაში თუ არა. თუ კი, მაშინ შესაბამის Key-
    ებს აბრუნებს. თუ არ არსებობს,  უგენერირებს ახალ Key-ს
67.     // და ბაზაში, ცხრილში objects.hotels, ახალ ჩანაწერს აკეთებს. ამ შემთხვევაში აბრუნებ
    ს ახალდაგენერირებულ Key-ს
68.     def getHotelKey(hotelName: String, hotelAddress: String, hotelCity: String, hotelCo
    untry: String, latitude: Double, longitude: Double): Int = {
69.         val cityKey = getCountryAndCityKeys(hotelCity, hotelCountry)("cityKey")
70.
71.         var hotelKey = 0
72.         val hotelsDF = cassandraTableAsDF("objects", "hotels")
73.
74.         hotelsDF.persist()
75.
76.         val existingHotelKey = hotelsDF.
77.             filter(hotelsDF("hotel_name") === hotelName &&
78.                         hotelsDF("hotel_address") === hotelAddress &&
79.                         hotelsDF("city_key") === cityKey &&
80.                         hotelsDF("latitude") === latitude &&
81.                         hotelsDF("longitude") === longitude)
82.         .select("hotel_key")
83.
84.         if (existingHotelKey.rdd.isEmpty()) {
85.             val maxKeyDF = hotelsDF
86.             .select(max("hotel_key")).alias("maxKey")
87.
88.             if (!maxKeyDF.rdd.isEmpty)
89.                 hotelKey = maxKeyDF.first.get(0).asInstanceOf[Int] + 1
90.
91.             val record = spark
92.                 .sparkContext
93.                 .parallelize(Seq((hotelKey, cityKey, hotelAddress, hotelName, latitude,
     longitude)))
```

```
94.                 .toDF("hotel_key", "city_key", "hotel_address", "hotel_name", "latitude
    ", "longitude")
95.
96.             saveDFtoCassandraTable(record, "objects", "hotels")
97.         }
98.         else
99.             hotelKey = existingHotelKey.first.get(0).asInstanceOf[Int]
100.
101.                hotelsDF.unpersist()
102.                hotelKey
103.            }
104.
105.            // გადაეცემა ქვეყნის სახელი და აბრუნებს შესაბამის Key-
    ს. პირველ რიგში, ამოწმებს,  უკვე არსებობს
106.            // ბაზაში თუ არა. თუ კი, მაშინ პირდაპირ Key-
    ს აბრუნებს, წინააღმდეგ შემთხვევაში, ახალ Key-ს აგანერირებს,
107.            // ჩანაწერს ამატებს ბაზაში და შემდეგ აბრუნებს Key-ს.
108.            def getCountryKey(countryName: String): Int = {
109.                var countryKey = 0
110.                val countriesDF = cassandraTableAsDF("locations", "countries")
111.
112.                countriesDF.persist()
113.
114.                val existingCountryKey = countriesDF
115.                    .filter(countriesDF("country_name") === countryName)
116.                    .select("country_key")
117.
118.                if (existingCountryKey.rdd.isEmpty) {
119.                    val maxKeyDF = countriesDF
120.                        .select(max("country_key").alias("maxKey"))
121.
122.                    if (!maxKeyDF.rdd.isEmpty)
123.                        countryKey = maxKeyDF.first.get(0).asInstanceOf[Int] + 1
124.
125.                    val record = spark
126.                        .sparkContext
127.                        .parallelize(Seq((countryKey, countryName)))
128.                        .toDF("country_key", "country_name")
129.
130.                    saveDFtoCassandraTable(record, "locations", "countries")
```

```
131.                 }
132.                 else
133.                     countryKey = existingCountryKey.first.get(0).asInstanceOf[Int]
134.
135.                 countriesDF.unpersist()
136.                 countryKey
137.             }
138.
139.             // გადაეცემა ქვეყნისა და ქალაქის სახელები. პირველ რიგში ორივესთვის ამოწმებს, ეს ქვეყანა და ამ ქვეყნის ეს ქალაქი უკვე არსებობს
140.             // ბაზაში თუ არა. თუ კი, მაშინ შესაბამის Key-ებს აბრუნებს ქვეყნისთვისაც და ქალაქისთვისაც Map-ის სახით. თუ არ არსებობს,  უგენერირებს ახალ Key-ებს
141.             // და ბაზაში, შესაბამის ცხრილებში, ახალ ჩანაწერებს აკეთებს. ამ შემთხვევაში აბრუნებს ახალდაგენერირებულ Key-ებს Map-ის სახით.
142.             def getCountryAndCityKeys(cityName: String, countryName: String): Map[String, Int] = {
143.                 val countryKey = getCountryKey(countryName)
144.
145.                 var cityKey = 0
146.                 val citiesDF = cassandraTableAsDF("locations", "cities")
147.
148.                 citiesDF.persist()
149.
150.                 val existingCityKey = citiesDF
151.                     .filter(citiesDF("city_name") === cityName && citiesDF("country_key") === countryKey)
152.                     .select("city_key")
153.
154.                 if (existingCityKey.rdd.isEmpty) {
155.                     val maxKeyDF = citiesDF
156.                         .select(max("city_key")
157.                         .alias("maxKey"))
158.
159.                     if (!maxKeyDF.rdd.isEmpty)
160.                         cityKey = maxKeyDF.first.get(0).asInstanceOf[Int] + 1
161.
162.                     val record = spark
163.                         .sparkContext
164.                         .parallelize(Seq((cityKey, cityName, countryKey)))
```

```
                    .toDF("city_key", "city_name", "country_key")

                saveDFtoCassandraTable(record, "locations", "cities")
            }
            else
                cityKey = existingCityKey.first.get(0).asInstanceOf[Int]

            citiesDF.unpersist()

            Map("countryKey" -> countryKey, "cityKey" -> cityKey)
        }

    }
```

Package: spark

Class/object Name: SparkSessionBuilder

Code:

```
package spark

import org.apache.spark.sql.SparkSession
import parsers.{ConfigParser, DefaultConfigs}

// ქმნის სპარკის სესიას. გადაეცემა ConfigParser-
ის ობიექტი. ამ არგუმენტის არგადაცემის შემთხვევაში, ნაგულისხმევი
// მნიშვნელობაა DefaultConfigs ობიექტი, რომელიც, თავის მხრივ, default-
ად იღებს რესურსებში მდებარე config.conf ფაილს
class SparkSessionBuilder(conf: ConfigParser = DefaultConfigs.defConf) {

    private val sparkAppName = conf.getAt("SparkSession.appName").toString
    private val master = conf.getAt("SparkSession.master").toString

    def buildSession(): SparkSession = {

        val spark = SparkSession.builder()
            .appName(sparkAppName)
            .master(master)
            .getOrCreate()

```

```
        spark
    }

}
```

Package: spark

Class/object Name: DefaultSession

Code:

```
package spark

import org.apache.spark.sql.SparkSession
import parsers.DefaultConfigs

object DefaultSession {
    private val conf = DefaultConfigs.defConf
    private val sbs = new SparkSessionBuilder(conf)

    val spark: SparkSession = sbs.buildSession()
}
```

Package: streamSimulation

Class/object Name: CreateStream

Code:

```
import java.util.{Calendar, Properties}

import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
import org.apache.spark.sql.{Row, SparkSession}
import parsers.{ConfigParser, DefaultConfigs}

import scala.collection.mutable._
import scala.util.parsing.json._

```

```
10. class CreateStream(spark: SparkSession, conf: ConfigParser = DefaultConfigs.defConf) ex
    tends Serializable {
11.
12.     // Spark-ის სესიის შექმნა
13.     val df = spark
14.         .read
15.         .option("inferSchema", "true")
16.         .option("header", "true")
17.         .csv(conf.getAt("StreamSimulation.dataFilePath"))
18.
19.     var fields = ListBuffer[String]()
20.
21.     df.schema.foreach(u => {
22.         fields += u.name.toString
23.     })
24.
25.     // ნაკადის სიმულაცია
26.     df.foreach(d => {
27.
28.         val message = rowToJsonString(d, fields)
29.         val props = new Properties()
30.
31.         props.put("key.serializer", conf.getAt("KafkaConfigs.keySerializer"))
32.         props.put("value.serializer", conf.getAt("KafkaConfigs.valueSerializer"))
33.         props.put("bootstrap.servers", conf.getAt("KafkaConfigs.brokers"))
34.
35.         val producer = new KafkaProducer[String, String](props)
36.         val record = new ProducerRecord(conf.getAt("KafkaConfigs.defaultTopic"), d.get(
    1).toString, message)
37.         producer.send(record)
38.
39.         println("Record sent to Kafka. Message content: " + record)
40.
41.         producer.flush()
42.         producer.close()
43.
44.         Thread.sleep(conf.getAt("StreamSimulation.timeout").toInt)
45.     })
46.
47.     // სტრიქონის JSON სტრინგად გარდაქმნა
```

```
48.     def rowToJsonString(row: Row, fields: ListBuffer[String]): String = {
49.         val map = HashMap[String, String]()
50.
51.         fields.indices.foreach(i => {
52.             map(fields(i)) = if (!row.isNullAt(i)) row(i).toString else ""
53.         })
54.
55.         map("review_date") = Calendar.getInstance.getTime().toString
56.
57.         JSONObject(map.toMap).toString()
58.     }
59.
60. }
```

Class/object Name: GetStream

Code:

```
1.  import parsers.DefaultConfigs
2.  import spark.DefaultSession
3.
4.  // ნაგულისხმევი მნიშვნელობებით ქმნის სპარკის სესიას. პასუხისმებელია კონფიგურაციაში მითი
    თებული ფაილიდან
5.  // ნაკადის შექმნაზე (სიმულაციაზე)
6.  object GetStream extends App {
7.      val conf = DefaultConfigs.defConf
8.      val spark = DefaultSession.spark
9.
10.     new CreateStream(spark, conf)
11. }
```

Class/object Name: ProcessStream

Code:

```
1.  import NLPTools.SentimentAnalyzer
2.  import org.apache.kafka.clients.consumer.ConsumerRecord
3.  import parsers.DefaultConfigs
4.  import services.{CassandraService, KafkaService}
5.  import spark.DefaultSession
6.
7.  import scala.util.parsing.json.{JSON, JSONObject}
8.
9.  // კითხულობს ნაკადს Kafka-დან, აანალიზებს სენტიმენტს და წერს Cassandra-ს ბაზაში
10. object ProcessStream extends App{
11.     val conf = DefaultConfigs.defConf
12.     val spark = DefaultSession.spark
13.
14.     val consumer = new KafkaService(conf)
15.     consumer.subscribeToTopic(conf.getAt("KafkaConfigs.defaultTopic"))
16.
17.     val analyzer = new SentimentAnalyzer
18.
19.     val cassandra = new CassandraService(spark)
20.
21.     // კითხულობს მესიჯებს KafkaConsumer-
    დან, ადგენს სენტიმენტს და იძახებს კასანდრას სერვისებს, რომელიც
22.     // პასუხისმგებელია მესიჯების ბაზაში ჩაწერაზე
23.     while (true) {
24.         val messages = consumer.pollMessages()
25.
26.         messages.foreach(m => {
27.                     val withSentiment = getSentiment(m, "review_text", analyzer)
28.                     cassandra.saveMessageToCassandra(withSentiment)
29.         })
30.     }
31.
32.     // აბრუნებს მესიჯს გაანალიზებულ სენტიმენტთან ერთად JSON სტრინგის სახით
33.     def getSentiment(message: ConsumerRecord[String, String], fieldName: String, sentim
    entAnalyzer: SentimentAnalyzer) = {
34.
```

```
35.         val parsedMessage = JSON.parseFull(message.value()).get.asInstanceOf[Map[String
    , String]]
36.         val sentiment = sentimentAnalyzer.detectSentiment(parsedMessage(fieldName).toSt
    ring)
37.         val withSentiment = parsedMessage + {"sentiment" -> sentiment}
38.
39.         val result = JSONObject(withSentiment).toString()
40.
41.         result
42.     }
43. }
```