jamk.fi

Data-analytiikka Apache Sparkilla

Lassi Partamies

Opinnäytetyö
Maaliskuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), Tietotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Partamies, Lassi	Opinnäytetyö, AMK	Maaliskuu 2018
	Sivumäärä 90	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Data-analytiikka Apache Sparkilla

Tutkinto-ohjelma
Tietotekniikan koulutusohjelma

Työn ohjaaja(t)
Antti Häkkinen, Sampo Kotikoski

Toimeksiantaja(t)
IoT:sta liiketoimintaan-projekti, Mika Rantonen

Tiivistelmä

Opinnäytetyö toteutettiin osana IoT:sta liiketoimintaan -projektia. Sen tavoitteena oli tutkia Apache Sparkin arkkitehtuuria ja toimintaa levossa olevan datan sekä reaaliaikaisen datavirran analysoinnissa.

Opinnäytetyön teoreettinen osuus koostuu Apache Sparkin arkkitehtuurin kuvaamisesta, sen käyttämästä Resilient Distributed Datasets (RDD)-toiminnasta ja Pythonin valmiiden data-analytiikkakirjastojen käyttämisestä Apache Sparkissa. Työn käytännön osuudessa Apache Sparkiin kytkettiin useita ulkoisia datalähteitä, jotka yhdistettiin ja data-analytiikan avulla analysoidaan käyttäen Python-ohjelmointikieltä. Työn tarkoituksena oli luoda ympäristö, jolla pystyttiin kokonaisuudessaan suorittamaan kaikki data-analytiikan toimenpiteet.

Data haettiin kahdesta eri lähteestä: Jyväskylän ammattikorkeakoulun Mango-palvelusta ja Ilmatieteenlaitoksen avoimesta datasta. Datavirrat luotiin Apache Kafkalla, johon tieto haettiin käyttämällä Pythonilla luotuja skriptejä. Datavirrat syötettiin Apache Sparkille, jossa tieto lähes reaaliajassa poimittiin ja tallennettiin sisäiseen tietokantaan. Tämän lisäksi haetusta datasta laskettiin korrelaatiota veden sameuden ja sademäärien välillä ja pyrittiin ennustamaan sateen muutoksia käyttäen apuna sääennustedataa.

Opinnäytetyössä päästiin lähes haluttuihin tavoitteisiin, jokseenkin tiettyjen osioiden jääden hieman vajaiksi. Tästä esimerkkinä Pythonin valmiiden data-analytiikkakirjastojen käyttö. Ympäristöstä saatiin toimiva ja tavoitteita vastaava. Itse data-analytiikan osuudesta saatiin halutut tulokset mitattua, mutta ne eivät datan puolesta ole täysin tarkkoja vaan enemmänkin suuntaa-antavia.

Avainsanat (asiasanat)
Data-analytiikka, Apache Spark, Apache Cassandra, Python, Stasmodels

Muut tiedot

jamk.fi

Description

Author(s) Partamies, Lassi	Type of publication Bachelor's thesis	Date March 2018
		Language of publication: Finnish
	Number of pages 90	Permission for web publication: x

Title of publication
Data analysis with Apache Spark

Degree programme
Information Technology

Supervisor(s)
Antti Häkkinen, Sampo Kotikoski

Assigned by
IoT to business project, Mika Rantonen

Abstract

The bachelor's thesis was assigned by IoT to Business project. The goal was to study Apache Spark's architecture and activity as a data analysis tool on stored data and real-time data streams.

The theory part consists of a description of Apache Spark's architecture and its use of Resilient Distributed Datasets (RDD). In addition, the theoretical part contains usage of Python's data analyst libraries as part of the Apache Spark applications. In the practical part, multiple external data sources are connected to Apache Spark, and data will be analysed using Python programming language. One goal is also to create a working environment that can fully complete all data analytical measurements.

Data is retrieved from two different sources. One is a Mango service owned by JAMK University of Applied Sciences and the other is the open data of Finnish Meteorological Institute. Data streams are created by using Apache Kafka, where data is retrieved using Python scripts. Data streams are sent to Apache Spark, where data will be extracted and stored in almost real-time. In addition, Apache Spark calculates the correlation between water turbidity and rainfall from the retrieved data. Apache spark will also be predicting changes of water's turbidity by using a weather forecast.

All desired goals were nearly achieved, however, certain sections were not covered as well as the others. For example, the usage of Python's data analysis is not that well covered. Environment is functional and contains everything that was desired. The results of correlation and predictions are not exact, however, they are very close even though the retrieved data contained some errors.

Keywords/tags (subjects)
Data Analysis, Apache Spark, Apache Cassandra, Python, Stasmodels

Miscellaneous

Sisältö

1 Työn lähtökohdat ... 7
2 Data-analytiikka ... 8
3 Apache Spark ... 8
3.1 Apache Sparkin arkkitehtuuri ... 8
3.2 Spark Core ... 9
3.3 Spark SQL ... 10
3.4 Spark Streaming ... 10
3.5 Discretized stream ... 11
3.6 SparkContext, StreamingContext, SparkSession ... 13
3.7 Toiminta klusterissa ... 13
3.8 Apache Sparkin cluster managerit ... 14
4 RDD ... 15
4.1 Rakenne ... 15
4.2 RDD:n abstraktio ... 16
4.3 RDD:n hyödyt ja heikkoudet ... 16
4.4 RDD:n esitysmuodot ... 17
4.5 Työt, vaiheet ja tehtävät ... 19
5 Apache Kafka ... 20
5.1 Apache Kafkan arkkitehtuuri ... 20
5.2 Kafkan ja Sparkin integraatio ... 20
5.3 Direct- ja Receiver-based-lähestymistapa ... 21
6 Apache Cassandra ... 22
6.1 Yleistä tietokannoista ... 22
6.2 Yleistä Apache Cassandrasta ... 22
6.3 PySpark Cassandra ... 23
7 Suunnitelma ... 23
7.1 Ympäristön suunnittelu ... 23

7.2 Analysoitava data ... 24

7.3 Spark-suunnitelma ... 25

7.4 Kafkan-suunnitelma ... 26

7.5 Tietokantasuunnitelma ... 26

8 Toteutus ... 27

8.1 Sparkin asennus ja klusterointi ... 27

8.2 Kafka-palvelun käyttöönotto ... 32

8.3 Tietokannan toteutus ... 34

8.4 Datan hakeminen Mango-palvelusta ... 37

8.5 Datan hakeminen Ilmatieteenlaitokselta ... 41

8.6 Datan kerääminen Sparkilla ... 43

8.7 Sparkin toiminta datan keräämisessä ... 48

8.8 Korrelaation laskeminen ... 53

8.9 Sameuden ennustaminen ... 57

9 Tulosten analysointi ... 60

9.1 Ympäristö ... 60

9.2 Korrelaation tulosten analysointi ... 62

9.3 Ennusteen tulosten analysointi ... 65

10 Johtopäätökset ... 68

10.1 Parannusehdotukset ... 68

10.2 Pohdinta ... 69

Lähteet ... 71

Liitteet ... 73

Liite 1. db_sameus.py ... 73

Liite 2. db_sade.py ... 74

Liite 3. ilmatiede.py ... 75

Liite 4. stable.py ... 76

Liite 5. stable.py 2 minuutin aikajana ... 78

Liite 6. stable.py workereiden resurssit 79
Liite 7. korrelaatio.py 80
Liite 8. Korrelaation tulokset 82
Liite 9. ennuste.py 83
Liite 10. Korrelaatio kesäkuu 2017 85
Liite 11. ennuste2.py 86
Liite 12. Mitatut sameuden muutokset verrattaen ennustettuun 88

Kuviot

Kuvio 1. Spark-arkkitehtuuri 9
Kuvio 2. Spark Streaming arkitehtuuri 11
Kuvio 3. DStream-ominaisuuden toiminta 12
Kuvio 4. Työnjako klusterissa 14
Kuvio 5. RDD:n toiminta klusterissa. 17
Kuvio 6. Narrow- ja Wide-riippuvuudet 18
Kuvio 7. Spark toiminnon suorittaminen 19
Kuvio 8. Ympäristön topologia 24
Kuvio 9. Tietokannan arkkitehtuuri 27
Kuvio 10. Staattiset laitenimet 27
Kuvio 11. Java 28
Kuvio 12. Python-versio 28
Kuvio 13. PySpark3 alias 28
Kuvio 14. Sparkin interaktiivinen komentokehote 29
Kuvio 15. Slaves-konfiguraatiotiedosto 30
Kuvio 16. Spark-selainkäyttöliittymä 31
Kuvio 17. Spark-Streaming-Kafka 31
Kuvio 18. Kafka otsikko 33
Kuvio 19. Cassandra nodetool status 34
Kuvio 20. Sade-taulun rakenne 35
Kuvio 21. Sameuden-taulun rankenne 36
Kuvio 22. Ennuste-taulun rakenne 36
Kuvio 23. Tulokset-taulun rakenne 37
Kuvio 24. Halutut tunnukset 38
Kuvio 25. Tallennetut sade- ja sameusarvot 38
Kuvio 26. Skriptien todennus 40
Kuvio 27. Haettu sääennuste 43
Kuvio 28. Ajossa olevat applikaatiot 47
Kuvio 29. Sademäärän DataFrame 47
Kuvio 30. Tietue tallennettuna tietokantaan 48
Kuvio 31. Suorittajien lisääminen 49

Kuvio 32. Job 0 ... 50
Kuvio 33. Työn Job 0 tiedot ... 51
Kuvio 34. Tietueen vastaanottaminen ... 52
Kuvio 35. Syklin töiden vaiheiden määrä ... 52
Kuvio 36. Kesäkuun säädatan taulu ... 54
Kuvio 37. Korrelaation tuloksien taulu ... 54
Kuvio 38. Korrelaatio applikaation suorittaminen ... 57
Kuvio 39. Lopullinen topologia ... 61
Kuvio 40. Lopulliset taulut ... 62
Kuvio 41. Sameuden muutos sateen kasvaessa ... 63
Kuvio 42. Sameuden äkillinen nousu sateen alkaessa ... 64
Kuvio 43. Negatiivinen korrelaatio ... 65
Kuvio 44. Ennusteen tulokset ... 67
Kuvio 45. Ennustettu sameus verrattuna mitattuun ... 68

Taulukot

Taulukko 1. Tiedon keräämisen applikaation töiden allokointi ... 53
Taulukko 2. Maaliskuun sameuden ennuste ... 66

Lyhenteet

API	Application Programming Interface
CQL	Cassandra Query Language
CSV	Comma Separated Values
DF	DataFrame
DSM	Distributed Shared Memory
DStream	Discretized stream
FIFO	First in, First Out
HDFS	The Hadoop Distributed File System
RDD	Resilient Distributed Dataset
SQL	Structured Query Language
SSL	Secure Sockets Layer
UI	User Interface
YARN	Yet Another Resource Negotiator

1 Työn lähtökohdat

Opinnäytetyön tarkoituksena oli perehtyä Apache Sparkin arkkitehtuuriin sekä sen toimintaan levossa olevan datan ja reaaliaikaisen datavirran analysoinnissa. Työssä oli tarkoituksena siis luoda ympäristö, jossa liikutetaan ja analysoidaan dataa reaaliajassa sekä jostain tietolähteestä. Työn teoreettisessa osassa kuvattiin Apache Sparkin toimintaa, arkkitehtuuria ja sen käyttämiä toiminteita kuten esimerkiksi Resilient Distributed Datesettejä (RDD). Tämän lisäksi käsiteltiin myös Python-ohjelmointikielen valmiiden data-analytiikkakirjastojen käyttämistä apuna Apache Sparkin kanssa. Työn käytännön osuudessa Apache Sparkiin kytkettiin ulkoisia datalähteitä, joita analysoitiin käyttäen Python-ohjelmointikieltä.

Tavoitteena oli luoda ympäristö, jolla voidaan toimivasti suorittaa data-analytiikan kaikki vaiheet: tähän liittyvät datavirrat ja niiden kytkeminen Apache Sparkiin, data-analytiikan osuus sekä tulosten tallentaminen johonkin tiedostojärjestelmään. Toimivan ympäristön pystytyksen lisäksi tavoitteena oli saada annetusta datasta automaation avulla haluttuja tutkimustuloksia.

2 Data-analytiikka

Data-analytiikalla tarkoitetaan prosessia, jolla pyritään tunnistamaan datasta haluttua informaatiota, käyttäen apuna siihen erikoistuneita toimintatapoja ja ohjelmistoja. Data-analytiikalla pyritään prosessoimaan dataa ja näin etsimään haluttuja tuloksia automaation ja matemaattisten menetelmien avulla. Data-analytiikkaa on yleisesti käytetty esimerkiksi markkinoinnissa ja erinäisillä tieteenaloilla. (Rouse 2016.)

Data-analytiikan prosessi alkaa datan keräämisestä. Tässä vaiheessa kerätään pohjaksi raakadataa, josta myöhemmin pystytään yhdistämällä tai laskelmoimalla saamaan haluttuja tuloksia. Tässä vaiheessa dataa otetaan halutuista lähteistä, muokataan sitä haluttuun formaattiin ja tallennetaan johonkin datan tallennuspaikkaan (esimerkiksi tietokanta). Tätä dataa voidaan syöttää tietopohjaan manuaalisesti, mutta suurissa datamäärissä yleensä luodaan automaattinen datavuo, joka syötetään jollekin datan prosessointiin tarkoitetulle sovellukselle. Tässä tapauksessa datavuosta poimitaan halutut tietueet ja ne tallennetaan myöhempää analysointia varten. (Rouse 2016.)

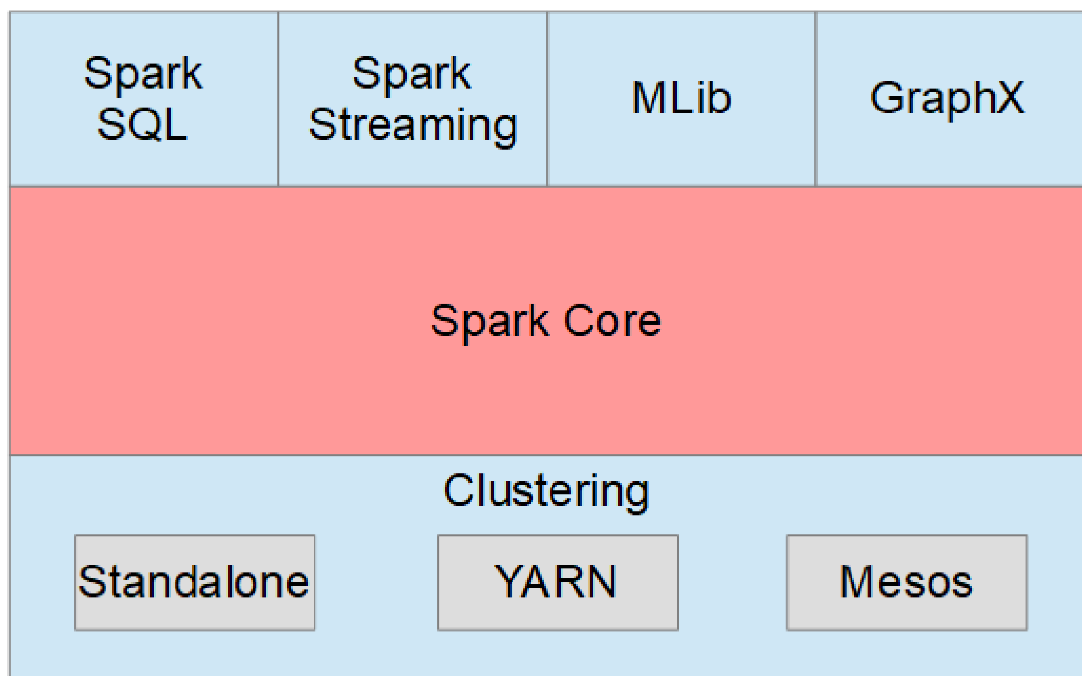
Kun haluttua dataa on saatu varastoitua, tulee seuraavassa vaiheessa tarkistaa datan yhtenäisyys ja korjata mahdolliset analytiikkaa vääristävät tietueet. Tämän jälkeen luodaan malli, jossa tulkitaan mitä datasta halutaan saada lopputulokseksi ja tämän mallin pohjalta siirrytään luomaan automaattista analytiikkaa käyttäen jotain valittua ohjelmointikieltä tai ohjelmistoa. Lopputulokset tallennetaan jälleen tiedostojärjestelmiin ja/tai luodaan tuloksista graafinen esitysmuoto. (Rouse 2016.)

3 Apache Spark

3.1 Apache Sparkin arkkitehtuuri

Apache Spark on avoimen lähdekoodin alusta klusteroituun tietojenkäsittelyyn, joka prosessoi dataa laskevien laitteiden muistissa. Sitä voidaan pääasiassa käyttää erinäisissä datan analyysiä vaativissa tehtävissä, niin levossa olevaan dataan kuin reaaliaikaisiin datavirtoihin. Alustaa voidaan käyttää yksittäisellä koneella paikallisesti tai resursseja voidaan jakaa klusterissa olevien koneiden kesken. Spark on kirjoitettu

käyttäen Scala-ohjelmointikeltä, mutta sillä on mahdollista ajaa sovelluksia, jotka on kirjoitettu Scala-, Python-, Java- ja R-ohjelmointikielillä. Spark tähtää nopeaan tiedon laskemiseen ja helppokäyttöisyyteen ja käyttää apunaan RDD:tä. RDD on Sparkin muuttumaton hajautettu tietorakenne, joka käsitellään tarkemmin luvussa 4. Sparkin arkkitehtuurin pohjana toimii Spark Core, johon voidaan liittää erinäisiä moduuleita käyttötarkoituksen mukaan (ks. Kuvio 1). (Laskowski n.d.b.)



Kuvio 1. Spark-arkkitehtuuri

Tehokkain käyttötapa on luoda jokaisella applikaatiolla RDD jostakin siihen syötetystä datasta. Data muokataan haluttuun muotoon ja lopuksi toteutetaan itse toimenpide, jolla dataa analysoidaan. (Laskowski n.d.b.)

3.2 Spark Core

Spark Core on koko alustan toiminnan periaate, jolle kaikki muut ominaisuudet rakentuvat. Core käsittelee kaikki ajettavat applikaatiot ja huolehtii tehtävien (task) välittämisestä eteenpäin ja töiden aikataulutuksesta. Spark Core pitää sisällään myös interaktiivisen ohjelmointirajapinnan, jolla voidaan kirjoittaa applikaatioita ja niiden

konfiguraatioita Scala- ja Python-ohjelmointikielillä (Spark Shell ja PySpark). Applikaatioita voidaan ajaa joko näiden avulla tai käyttäen Sparkin jakelupaketin mukana tulevaa spark-submit-skriptiä. Myös RDD:t luodaan ja käsitellään Spark Coren sisällä. (Apache Spark - Core Programming n.d.)

3.3 Spark SQL

Spark Structured Query Language (SQL) (Structured Query Language (SQL) n.d.) on Sparkin moduuli jäsenneltyyn datan prosessointiin. Spark SQL pystyy tarjoamaan Sparkille tietoa datan rakenteesta. Spark SQL:ää voidaan käyttää ajamalla SQL-tyyppisiä kyselyitä dataan, mutta yleisin käyttötapa Spark SQL:lle on käyttää sen mukana tulevia DataFrameja ja Datasettejä. (Spark SQL, DataFrames and Datasets Guide n.d.)

Datasetit ovat RDD:n kaltaisia hajautettuja osia datasta, jotka tuovat RDD:n ominaisuudet SparkSQL:ään. Datasettejä voidaan muokata vastaavasti, kuten RDD:itä käyttäen esimerkiksi funktiota `map` tai `filter`. `Map` palauttaa uuden RDD:n, jonka jokaiseen elementtiin tehdään haluttu funktio, kun taas `filter` palauttaa uuden RDD:n, jotka vastaavat annettuja parametrejä (Pyspark Package. n.d). Datasetit toimivat ainoastaan käytettäessä ohjelmointikielenä Javaa tai Scalaa. Python-ohjelmointikieli ei tue Datasettejä, mutta Pythonin dynaamisen luonnon puolesta useat Datasettien edut on saatavilla käyttäen Pythonin muita funktioita. (Spark SQL, DataFrames and Datasets Guide n.d.)

DataFramet ovat käytännössä Datasettejä, jotka on järjestelty nimettyihin sarakkeisiin. Näin ollen DataFrameja voidaan verrata relaatiotietokantojen tauluihin tai Pythonin DataFrameihin, mutta SparkSQL:n DataFramet ovat paremmin optimoituja käytettäessä Apache Sparkia. DataFramet voidaan muodostaa useista eri lähteistä, tärkeimpänä näistä ovat tietyt jäsennellyt tiedostotyypit, ulkoiset tietokannat ja jo olemassa olevat RDD:t. (Spark SQL, DataFrames and Datasets Guide n.d.)

3.4 Spark Streaming

Spark Streaming on Spark Coren laajennus, joka tarjoaa skaalautuvan, vikasietoisen ja suorituskykyisen työkalun reaaliaikaisten datavirtojen analysointiin. Dataa voidaan

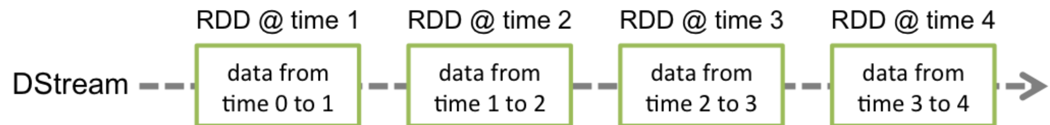
vastaanottaa useista eri lähteistä, mutta yleisimmin käytettyjä sovelluksia ovat esimerkiksi Apache Kafka, Flume ja Twitter. Lisäksi Spark Streaming -laajennuksella voidaan vastaanottaa datavirtoja Transmission Control Protocol (TCP)-kannoista (socket). Käsitellyn datan lopputulokset voidaan tallentaa suoraan tietokantoihin, HDFS-tiedostojärjestelmään (HDFS. N.d) tai dashboard-näkymiin. (Spark Streaming Programming Guide n.d.)

Datan käsittelyssä voidaan käyttää erinäisiä funktioita sen muokkaamiseen ja leikkaamiseen. Data vastaanotetaan lähettäjältä Spark Streaming -moduulin avulla, josta data lähetetään osissa Spark Corelle, joka prosessoi datan sen omien resurssien mukaan ja lähettää prosessoidun datan eteenpäin haluttuun tallennusjärjestelmään (ks. Kuvio 2). (Spark Streaming Programming Guide n.d.)

Kuvio 2. Spark Streaming arkitehtuuri (Spark Streaming Programming Guide n.d.)

3.5 Discretized stream

Discretized stream (DStream) on Spark Streamingin käyttämä ominaisuus, jolla tarkoitetaan jatkuvaa datavuota. DStream voidaan luoda joko sisään tulevasta datasta, jostakin ulkoisesta datalähteestä tai jostain toisesta DStreamista. Sparkin sisällä DStreamit ovat käytännössä RDD-sarjoja. Jokainen RDD sisältää dataa muuttumattomassa hajautetussa muodossa tietyltä ajanjaksolta (ks. Kuvio 3). (Spark Streaming Programming Guide n.d.)



Kuvio 3. DStream-ominaisuuden toiminta (Spark Streaming Programming Guide n.d.)

Jokainen toiminne tai muokkaus, joka suoritetaan koskien DStreamissa kulkevia RDD:itä sijoitetaan taustalla oleviin RDD:hin. Nämä RDD:t laskelmoidaan taas Spark Coressa. Yleinen tapa on, että DStreamissa tapahtuvissa vaiheissa leikataan datasta ylimääräiset tietueet ja osiot pois, jonka jälkeen se lähetetään Spark Corelle siinä muodossa, jossa se halutaan käsitellä. (Spark Streaming Programming Guide n.d.)

DStreamit tulee muodostaa applikaation koodissa käyttäen StreamingContext-objektia, joka toimii hyvin samalla periaatteella kuin SparkContext-objektiivi. StreamingContext-objektiiviin määritetään konfiguraatio, joka yksinkertaisimmillaan on halutun lähteen osoite, johon ajettaessa StreamingContext muodostaa yhteyden. (Spark Streaming Programming Guide n.d.)

Kaikki DStreamit toimivat yhdessä vastaanottajien (Receivers) kanssa. Vastaanottajat nimensä mukaisesti vastaanottavat datan sen lähteeltä ja tallentavat sen Sparkille prosessoitavaksi. On mahdollista ajaa useita samanaikaisia DStreameja, jolloin tarvitaan myös vastaava määrä vastaanottajia, jotka samanaikaisesti vastaanottavat dataa jokaisesta datavuosta. Tässä vaiheessa onkin huomioitava, että jokainen datavuo varaa yhden ytimen sitä ajavalta koneelta. Näin ollen, mikäli samanaikaisesti ajetaan useampaa datavuota, tulee myös resursseja allokoida suorittaville koneille sen mukaisesti. Ajettaessa useampaa datavuota klusterissa on taas huomioitava, että applikaatiossa tulee olla enemmän ytimiä määritettynä, kuin vastaanottajia. Mikäli vastaanottajia on useampi kuin määritettyjä ytimiä, klusteri saa datan muttei pysty prosessoimaan sitä. (Spark Streaming Programming Guide n.d.)

3.6 SparkContext, StreamingContext, SparkSession

SparkContext-objekti on Sparkin toiminnan lähtökohta. SparkContext muodostaa yhteyden kaikkiin Spark-klusterin osiin ja sen avulla pystytään muodostamaan RDD:itä. Tämän takia SparkContext-objekti tulee olla ensimmäinen asia, joka määritellään ajettavassa applikaatiossa. Toistaiseksi ainoastaan yksi SparkContext voi olla aktiivisena. (Class SparkContext n.d.)

StreamingContext on SparkContextia vastaava objekti, mutta se toimii Spark Streamingin toiminnallisuuksien lähtökohtana. Kuten SparkContext, StreamingContext luo yhteyden klusterin osiin, mutta sen avulla voidaan muodostaa DStreameja. Dstreamin luomisen jälkeen voidaan reaaliaikainen laskelmointi aloittaa käyttäen StreamingContextin komentoa `context.start()` ja lopettaa käyttäen komentoa `context.stop()`. (Pyspark Streaming Module n.d.)

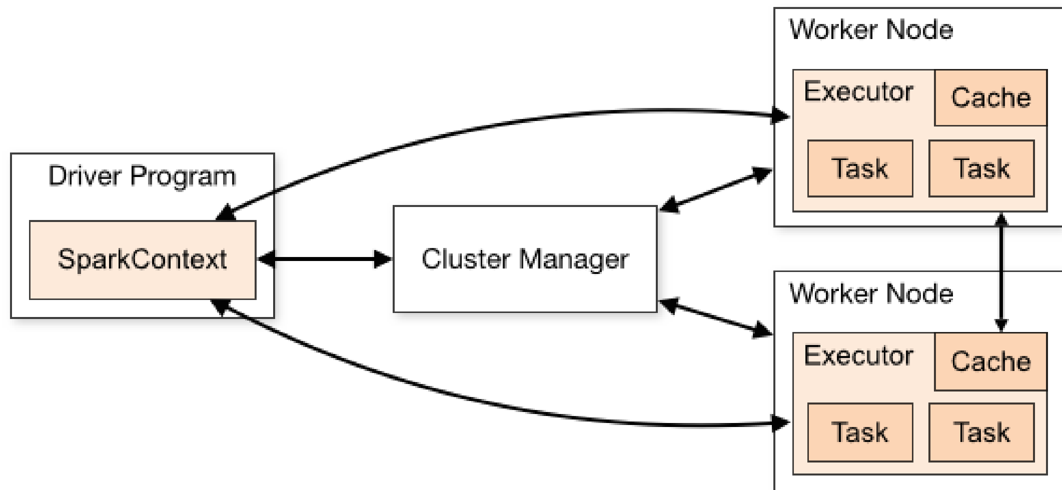
Mikäli applikaatio haluaa käyttää Sparkin DataFrameja tai Datasettejä, tulee käyttää objektia SparkSession. Jälleen SparkSession vastaa toiminnaltaan SparkContextia, mutta toimii myös pohjana Datasetille ja DataFrameille. (Phatak 2016.)

3.7 Toiminta klusterissa

Applikaatiot toimivat itsenäisinä kokoelmina prosesseja Apache Spark -klusterissa, joita koordinoidaan SparkContext-objektilla klusterin pääohjelmassa. Tätä ohjelmaa kutsutaan myös nimellä Driver Program. SparkContext yhdistää klusterin manageriin, joka allokoi resursseja klusterissa olevien suorittajien (Executor) välillä. Suorittajat ovat itse laskelmoinnin suorittava osa. Ne ovat esimerkiksi klusterissa sijaitsevia koneiden prosesseja, joiden optio on ottaa työ vastaan Driver Programilta ja ajavat halutut laskelmat ja tallentavat applikaatioiden tiedot muistiinsa. (Cluster Mode Overview n.d.)

Työt aloitetaan riippumatta mitä Driver Programia käytetään siten, että SparkContext yhdistää klusterin manageriin (Cluster Manager). Tämän jälkeen cluster manager kerää kaikkien suorittajien tiedot ja tilan ja lähettää suorittaville executor-toiminnoille applikaation koodin. Lopuksi SparkContext lähettää suorittajille tehtäviä, eli tiettyjä

osia, jotka tämän suorittajan tulee suorittaa (ks. Kuvio 4). (Cluster Mode Overview n.d.)



Kuvio 4. Työnjako klusterissa (Cluster Mode Overview n.d.)

Jokainen applikaatio saa oman executor-prosessinsa. Tämä prosessi on päällä ja varattuna tälle applikaatiolle koko ajon ajan. Prosessi luo itselleen alaprosesseja, joiden kautta tehtävät ajetaan. Se miten työt jaetaan eri suorittajien välillä, on riippuvainen cluster managerista ja tämän jonotusmekanismeista sekä applikaatioon määritettävistä konfiguraatioista. (Cluster Mode Overview n.d.)

On myös hyvä huomioida, että driver program tarvitsee koko ajan yhteyden kaikkiin muihin klusterin osiin. Tässä saattaa tulla ongelmaksi verkossa oleva viive. Täten on suositeltavaa, että kaikki osat sijaitsisivat samassa verkossa, jottei verkon viiveestä koituisi ylimääräistä viivettä itse applikaatioiden ajossa. (Cluster Mode Overview n.d.)

3.8 Apache Sparkin cluster managerit

Kuten jo aikaisemmin mainittiin, Sparkia voidaan käyttää niin itsenäisenäisesti kuin klusterissakin. Uusimmassa versiossa on tuki kolmelle eri cluster managerille: Spark standalone, Apache Mesos ja Hadoop Yet Another Resource Negotiator (YARN). Näiden lisäksi on myös kehitteillä oleva Kubernetes. (Cluster Mode Overview n.d.)

Spark Standalone on pidetty helpoimpana cluster managerina käyttää ja toteuttaa. Kuten nimestäkin voidaan jo päätellä, tähän ei tarvita mitään ulkoista ohjelmaa, vaan se tulee automaattisesti Sparkin jakelun mukana. Spark Standalonella on hyvä saatavuus ja se pystyy toipumaan hyvin yksittäisten työkoneiden virheistä. Sillä voidaan myös allokoida resursseja applikaatioiden välillä. (Lynn 2016.)

Jonotusmekanismina Standalonella on ainoastaan käytössä First in, First Out (FIFO). Oletuksena jokainen applikaatio otetaan käsittelyyn niiden saapumisjärjestyksessä käyttäen kaikkia käytettävissä olevia verkon osia. Resurssien maksimi määrää applikaatiota tai käyttäjää kohden voidaan kuitenkin rajoittaa. Mikäli jokaisen työkoneen fyysisten resurssien, esimerkiksi prosessorin tai muistin käyttöä, halutaan rajoittaa, tämä onnistuu sisällyttämällä tiedot applikaation SparkConf-objektiin. (Lynn 2016.)

Standalone cluster manager voi virhetilanteissa nostaa itsensä toimintakykyiseksi automaattisesti käyttäen apunaan ZooKeeper-ohjelmistoa. Tämä voidaan myös toteuttaa manuaalisesti käyttäen jotakin tiedostojärjestelmää. (Lynn 2016.)

Standalone tukee autentikaatiota klusterissa olevien koneiden välillä käyttäen ennalta jaettua salausavainta. Laitteiden välillä tapahtuva kommunikaatio voidaan myös salata käyttäen Secure Sockets Layer (SSL) -protokollaa. (Lynn 2016.)

Klusterin tilannetietoa voidaan monitoroida selainpohjaisen käyttöliittymän (UI) avulla, josta voidaan nähdä esimerkiksi ajettavien applikaatioiden ja suorittajien tila sekä resurssien käyttö. Jokaiselle applikaatiolle luodaan oma Web UI heti, kun se on rekisteröity ajettavaksi. (Lynn 2016.)

4 RDD

4.1 Rakenne

RDD on Sparkin muuttumaton hajautettu tietorakenne, joka koostuu erinäisten objektien kokoelmista. Jokainen RDD:n datasetti on jaettu loogisiin osioihin, joita voidaan laskea eri osissa Spark klusteria. RDD:t voivat sisältää mitä tahansa Python-, Java- tai Scala-objekteja ja käyttäjän määrittämiä luokkia (class). RDD:illä pyritään

tekemään datan käsittelystä nopeampaa ja edullisempaa laitteiden resursseille. (Apache Spark – RDD n.d.)

Yleisesti RDD:t ovat jaoteltuja kokoelmia joistakin tietueista. RDD:hen ei voi suoraan kirjoittaa uutta tai korvata tiettyjä arvoja, vaan ne on tarkoitettu ainoastaan luettavaksi. Niitä voidaan luoda määrittäen tietyt arvot applikaation koodissa tai muodostamalle ne jostain ulkoisesta datalähteestä. (Apache Spark – RDD n.d.)

4.2 RDD:n abstraktio

Normaalisti RDD on vain lukuoikeuksilla oleva hajautettu kokoelma tietueita. Niitä voidaan luoda ainoastaan määrittämällä se erikseen, luomalla se jostakin tiedostojärjestelmästä tai muista RDD:istä. Näitä ominaisuuksia kutsutaan muunnoksiksi, joilla erotetaan ne muista toiminteista RDD:eissä. Esimerkkinä muunnoksista ovat `map` ja `filter`. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

Näiden lisäksi käyttäjä voi hallita RDD:n osiointia ja säilyvyyttä. Käyttäjä voi määrittää mitkä RDD:t käytetään uudelleen ja niitä voidaan määrittää varastoitavaksi esimerkiksi välimuistiin. Käyttäjä voi myös sisällyttää tiettyjä elementtejä vastaavat RDD:t tietyille koneille klusterissa. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

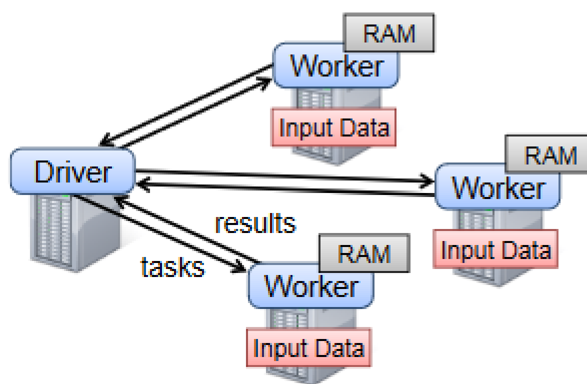
4.3 RDD:n hyödyt ja heikkoudet

Distributed Shared Memory (DSM) applikaatiot lukevat ja kirjoittavat satunnaisiin kohteisiin niiden globaalissa osoiteavaruudessa. DSM on todella yleinen tapa nykypäivän datankäsittelytyökaluissa. Verrattuna RDD:hen DSM on paljon vaikeampi implementoida tehokkaasti ja vikasietoisesti klusteriin. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

Verrattuna DSM:ään RDD:itä voidaan luoda ainoastaan coarse-grained-muutoksilla. Tämä estää RDD:itä tekemästä suuria määriä kirjoittamista, mutta on muuten paljon DSM:ää vikasietoisempi. Tämän lisäksi RDD:llä voidaan ottaa varmuuskopioita hitaista taskeista MapReducessa, kun taas DSM-ohjelmien varmuuskopiointi vaatisi

kaksi kopiota tehtävästä, jotka vaativat pääsyn samaan muistinkohteeseen estäen toistensa päivitykset. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

RDD:llä on kaksi suurinta etua verrattaen DSM:ään. RDD:n massaoperaatioissa voidaan ajoittaa tehtävät datan paikallisuudesta riippuen täten tehostaen suorituskykyä (ks. Kuvio 5). Tämän lisäksi voidaan määrittää tiedot tallennettavaksi levylle välimuistin sijaan, kun ajetaan ainoastaan tietoja skannaavia operaatioita. Tämä auttaa tilanteissa, joissa välimuistia ei ole tarpeeksi. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)



Kuvio 5. RDD:n toiminta klusterissa. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

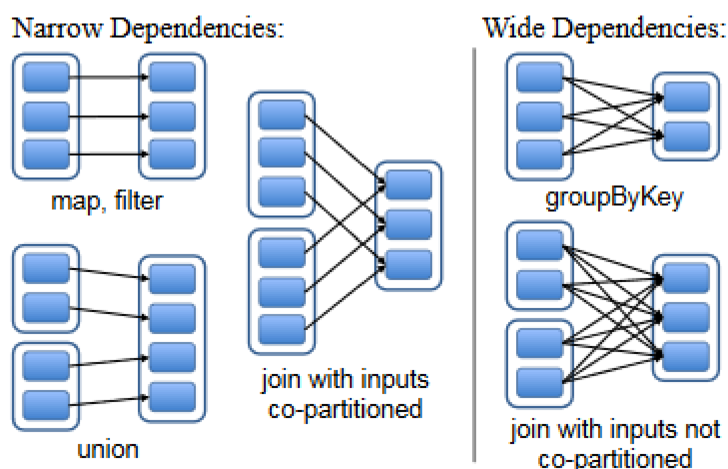
RDD:llä on kuitenki tiettyjä rajoitteita sen arkkitehtuurin luonnon mukaisesti. RDD:t toimivat parhaiten silloin, kun kaikille osioille käsiteltävää dataa tehdään aina samat toimenpiteet ja data tulee aina halutussa formaatissa. Vastaavasti ne eivät sovellu asynkronisille applikaatioille, joiden tulee päivittää jatkuvasti tilatietoa jaettuihin järjestelmiin. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

4.4 RDD:n esitysmuodot

Yhtenä suurimmista haasteista voidaan nähdä RDD:n esitysmuodon valinta. Ideaalisesti halutaan, että käyttäjät pystyvät tekemään muutoksia mahdollisimman

laajalla skaalalla siten, että esitysmuoto on selkeä ja siinä on selkeät eroavuudet eri RDD:iden välillä. Näihin on yleisesti käytetty kahta erityyppistä tapaa esittää riippuvuuksia RDD:iden välillä: narrow- ja wide-riippuvuudet. Narrow-riippuvuudessa käytetään parent RDD:n osioita pienempinä osioina child RDD:inä. Tästä esimerkkinä on `map`-funktio, jolla voidaan muokata dataa haluttuun esitysmuotoon tai ottaa siitä vain valittuja osioita. `Wide depencies` taas on RDD, josta useampi child RDD on riippuvainen. Tästä toimii esimerkkinä funktio `groupByKey`, joka yhdistää useita avainta vastaavia arvoja halutulla tavalla. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

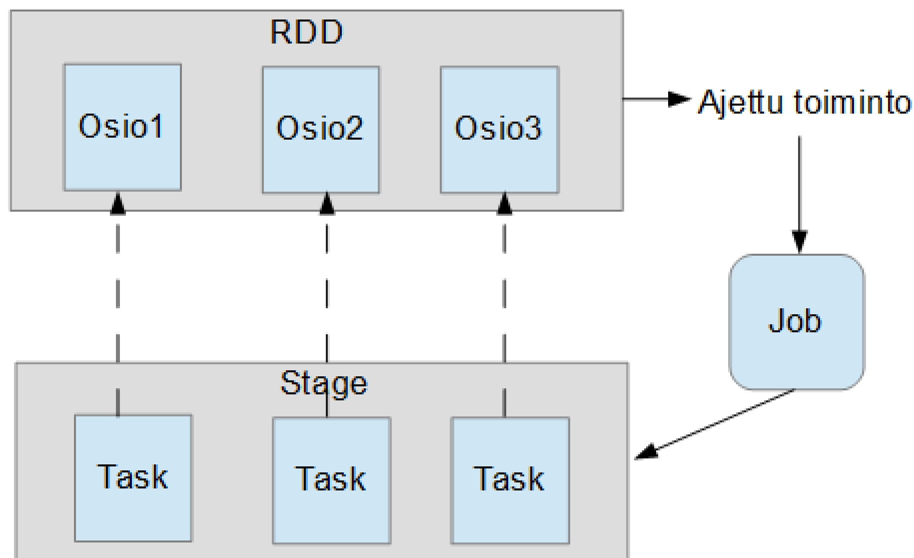
Tämä jaottelu on hyödyllinen kahdesta syystä. Ensimmäisenä narrow-riippuvuuksien avulla voidaan suoritta yhdellä klusterin osalla laskelmointi kaikille parent-osioille, kun taas wide-riippuvuudet tarvitsevat kaiken datan kokoajan saatavilla jokaisella solmulla (node) (ks. Kuvio 6). Toisekseen yksittäisen solmun vikatilanteista korjautuminen on yksinkertaisempaa, koska silloin tulee vain laskea uudelleen ainoastaan vikaantuneella solmulla esiintyvät isäntä osiot. Tämä pystytään suorittamaan muilla klusteriin kuuluvilla solmuilla. (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)



Kuvio 6. Narrow- ja Wide-riippuvuudet (Chowdhury, Das, Dave, Franklin, Ma, McCauley, Shenker, Stoica & Zaharia 2012.)

4.5 Työt, vaiheet ja tehtävät

Spark jakaa applikaation ajon aikana tehtävät toimenpiteet kolmeen hierarkkiseen ryhmään: Työ (job), vaihe (stage) ja tehtävä (task) (ks. Kuvio 7).



Kuvio 7. Spark toiminnon suorittaminen

Korkeimpana näistä ryhmistä ovat työt. Käytännössä jokainen RDD:ssä tapahtuvista toiminteista jaetaan töihin. Töiden osalta lasketaan laskelmoinnin osat, jotka tarvitaan halutun toiminnon suorittamiseen halutussa RDD:ssä. Yleensä työt alkavat ainoastaan yhdestä RDD:stä, mutta voivat lopuksi jakaantua useaan eri RDD:hen sen luonteesta riippuen. (Laskowski n.d.a.)

Käytännössä ottaen työt jakavat itsensä pienempiin osiin, jotka sisältävät osioita tietystä laskelmoinnista. Näitä osia kutsutaan vaiheiksi (stage). Vaiheet ovat rinnakkain toimivien tehtävien (task) sarjoja, yksi tehtävä jokaista osiota kohti. Jokainen vaihe voi liittyä yhteen usean itsenäisen vaiheen kanssa, jotka ovat kohdistuneet toisiin RDD:ihin. (Laskowski n.d.c.)

Tehtävät ovat pienin yksittäinen osio Sparkin toimenpiteen jaossa. Tehtävien avulla voidaan allokoida osiot laskelmoinnista sille sopiville suorittajalle. Tehtävät käynnistetään sille määrätylle suorittajalle, jossa ne ajetaan sopivalla hetkellä. Käytännössä

ottaen tehtävät ovat laskelmointeja RDD:n osioissa olevista tietueista. (Laskowski n.d.d.)

5 Apache Kafka

5.1 Apache Kafkan arkkitehtuuri

Apache Kafka on hajautettu alusta, jonka avulla voidaan lähettää ja vastaanottaa reaaliaikaisia datavirtoja. Pääasiassa Kafkaa käytetään luomaan reaaliaikaisia datavirtoja eri laitteiden ja applikaatioiden välillä. Lisäksi sen avulla voidaan luoda applikaatioita, jotka muokkaavat datavirtoja tai reagoivat niihin halutulla tavalla. Kafka toimii periaatteellisesti hyvin lähellä viestijonoa: se vastaanottaa viestejä halutusta lähteestä ja lähettää niitä eteenpäin halutussa järjestyksessä ja muodossa. Viestit voivat käsittää mitä tahansa haluttuja tietueita erinäisistä kohteista, kuten tiedostoista, tietokannoista tai muista datavirroista. Kafka reagoi ja prosessoi datavirtoja sitä mukaan, kun uusia tietueita saapuu. (Introduction n.d.)

Kafkaa voidaan käyttää klusteroituna joko yhdellä taikka useammalla palvelimella. Se tallentaa datavuon tietueet eri kategorioihin, joita kutsutaan otsikoiksi (topic). Jokainen tietue sisältää avaimen, arvon ja aikaleiman. (Introduction n.d.)

Kafka koostuu neljästä pääohjelmointirajapinnasta: Producer Application Programming Interface (API), Consumer API, Streams API ja Connector API. Producer API:n avulla lähetetään dataa Kafka klusterin otsikoille. Consumer API:n avulla voidaan lukea dataa otsikoista. Streams API muuttaa datavuon sisään tulevasta otsikosta ulosmeneväksi. Kun taas Connect API:n avulla voidaan luoda yhteyksiä muihin laitteisiin tai applikaatioihin, jotka voivat joko vetää dataa Kafkalta tai syöttää dataa Kafkalle. (Introduction n.d.)

5.2 Kafkan ja Sparkin integraatio

Kafkan ja Sparkin välisen integraation voi toteuttaa kahdella eri tavalla. Vanhempi tapa on käyttää vastaanottajia ja uudempi tapa, jossa ei käytetä ollenkaan vastaanot-

tajia. Kumpikin toteutustavoista on edelleen käyttökelpoinen, mutta niillä on eroavaisuuksia ohjelmointimallissa ja suorituskyvyn ominaisuuksissa. (Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher) n.d.)

Tällä hetkellä integraatioon on käytössä kaksi eri versiota paketista: spark-streaming-kafka-0-8 ja spark-streaming-kafka-0-10. Paketti spark-streaming-kafka-0-8 on tuettu Scala-, Java- ja Python-ohjelmointikielillä, ja sen versio on todettu stabiiliksi. Vastaavasti paketti spark-streaming-kafka-0-10 on tuettu vain ohjelmointikielillä Scala ja Java ja tätä kyseistä versiota ei ole vielä laskettu stabiiliksi. (Spark Streaming + Kafka Integration Guide n.d.)

5.3 Direct- ja Receiver-based-lähestymistapa

Vastaanottajapohjainen (Receiver-based) lähestymistapa käyttää nimensä mukaisesti vastaanottajia vastaanottamaan dataa. Vastaanottajat implementoidaan käyttäen Kafkan consumer API:a datan vastaanottamiseksi. Kaikki data, joka vastaanotetaan vastaanottajan kautta Kafkalta, tallennetaan Sparkin executorille, minkä jälkeen Spark käynnistää työn ja alkaa prosessoida dataa. Tässä toteutustavassa on mahdollisuuksia menettää dataa, mikäli prosessin aikana tulee virheitä, mutta koska dataa tallennetaan koko ajan lokeihin, voidaan kadonnut data palauttaa täältä virheen tapahtuessa. (Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher) n.d.)

Suorassa (Direct) toteutustavassa pyritään varmistamaan mahdollisimman vahva pisteestä pisteeseen yhteys. Vastaanottajien sijasta Spark pystyy määritetyin aika ajoin vetämään viimeisimmän datan Kafkan otsikolta ja näin ollen pystyy prosessoimaan sen erissä. Kun Spark käynnistää työn datan prosessoimiseen, Kafka käyttää consumer API:a lukemaan määritetyn datan siltä halutulta alueelta. (Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher) n.d.)

Suorassa lähestymistavassa on tiettyjä etuuksia verrattaessa vastaanottajapohjaiseen toteutustapaan. Suorassa lähestymistavassa ei tarvitse tehdä useita lähteitä Kafkan datavirroille ja yhdistää niitä, vaan Spark Streaming luo niin monta RDD:n osiota, kuin tulevia osia vastaanotetussa datassa on. Tällöin dataa on helpompi muokata halutulla

tavalla. (Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher) n.d.)

Suora toteutustapa on myös edullisempi laitteistolle. Kun vastaanottajapohjaisessa lähestymistavassa halutaan välttyä datan menettämiseltä kokonaan, tallennetaan data kahteen eri paikaan. Ensimmäinen tallennus tapahtuu Kafkalla ja toinen lokiin, josta Spark saa datan. Tämä vaatii ensinnäkin enemmän tallennustilaa laitteelta, ja tallentaminen useaan kertaan aiheuttaa turhaa viivettä datan prosessoinnissa. Suora toimintatapa vaatii, että data on tallennettuna ainoastaan Kafkalla, josta se voi vikatilanteessa palauttaa datan. (Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher) n.d.)

6 Apache Cassandra

6.1 Yleistä tietokannoista

Tietokanta on kokoelma järjestettyä tietoa, johon on helppo päästä käsiksi, sitä on helppoa muokata ja päivittää. Data järjestetään riveihini, sarakkeisiin ja tauluihin. Tietokantojen tauluihin voidaan tallentaa ja muokata dataa, sekä tauluista voidaan lukea suoraan dataa tekemällä kyselyitä tietokannan kyselykielen mukaisesti. Käytännössä tietokantoihin voidaan varastoida paljon dataa, joka on järjestelty haluttuun esitysmuotoon ja sitä voidaan hakea luettavaksi. Nykyisin tietokantoja on useita erityyppisiä, jotka soveltuvat useisiin eri käyttötarkoituksiin. (Rouse 2017.)

Relaatiotietokannat ovat yksi käytetyimmistä tietokantatyypeistä. Kaikki relaatiotietokantojen taulut liittyvät johonkin ennalta määrättyyn kategoriaan. Lisäksi tauluille luodaan relaatioita keskenään. Pohjana relaatiotietokannoille on ohjelmointirajapinta SQL. Toinen yleisesti käytössä oleva tietokantatyyppi on NoSQL. NoSQL-tietokantoja käytetään erityisen paljon data-analytiikassa, koska ne soveltuvat isojen datamäärien nopeaan käsittelyyn. (Rouse 2017.)

6.2 Yleistä Apache Cassandrasta

Apache Cassandra on hyvin skaalautuva, avoimen lähdekoodin NoSQL-tietokanta, jonka pääasiallisena tavoitteena on tarjota käyttäjälle jatkuva saatavuus, skaalautuva

suorituskyky ja yksinkertainen käyttöliittymä. Cassandra on luotu prosessoimaan suuria määriä dataa mahdollisimman nopeasti, ja tämän takia se on todella yleisesti käytetty erinäisissä töissä, joissa vaaditaan suurien datamäärien keräystä keskitettyyn tietokantaan. Cassandran avulla tietokannat on mahdollista toteuttaa hajautetussa klusterissa, jossa data replikoidaan sen kaikkien laitteiden välillä. (A Brief Introduction to Apache Cassandra n.d.)

Cassandran kanssa toimiminen toimii nykyisin pääasiallisesti Cassandra Query Language (CQL) API:n avulla. CQL vastaa syntaksiltaan hyvin paljon SQL:ää, jonka ansiosta relaatiotietokannoista siirtyminen Cassandraan on saatu mahdollisimman helpoksi. CQL:n avulla voidaan muun muassa luoda, poistaa, tyhjentää ja lukea dataa Cassandran tauluista halutulla tavalla. (A Brief Introduction to Apache Cassandra n.d.)

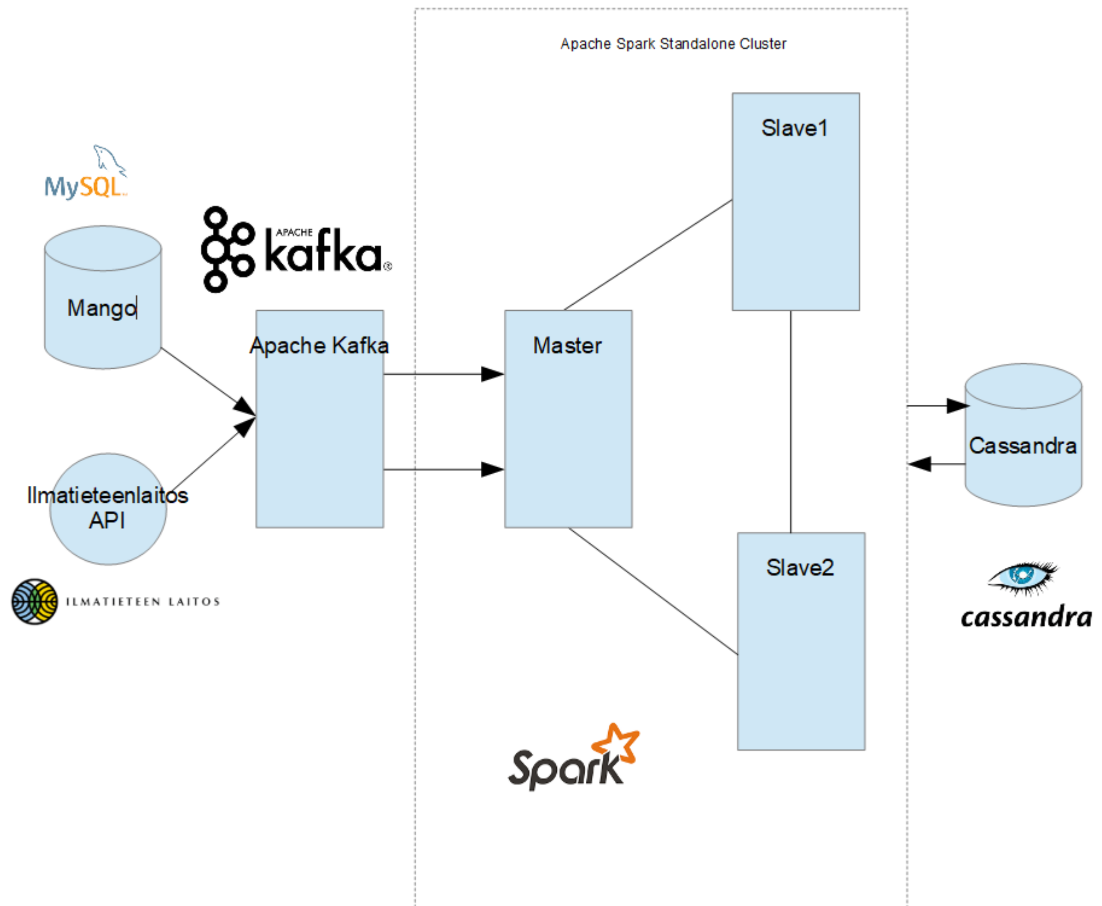
6.3 PySpark Cassandra

PySpark Cassandra on Spark Cassandra Connectorin pohjalta tehty moduuli, jolla Sparkin applikaatiot, jotka on kirjoitettu käyttäen Python-ohjelmointikieltä voivat muokata ja lukea Cassandran tietokannasta haluttua tietoa. Moduuli tarjoaa myös tuen tuoda Cassandran CQL-rivit suoraan RDD:ksi. Pyspark Cassandra toimii sekä PySparkin interaktiivisessa komentokehotteessa ja ajettaessa Sparkin applikaatioissa. (PySpark Cassandra 2018.)

7 Suunnitelma

7.1 Ympäristön suunnittelu

Työssä haluttiin luoda toimiva ympäristö, jossa voidaan toteuttaa Apache Sparkin avulla data-analytiikkaa Python-ohjelmointikielellä. Tämän lisäksi ympäristössä haluttiin toteuttaa reaaliaikaista ja levossa olevan datan analytiikkaa. Tulokset tuli saada tallennetuksi johonkin tiedostojärjestelmään. Työssä pohjana toimi Kuvio 8 mukainen topologia. Kaikki koneet toteutettiin virtuaalisena käyttäen Oracle VM VirtualBox -ohjelmistoa. Käyttöjärjestelmänä laitteille käytettiin Centos 7 -käyttöjärjestelmää.



Kuvio 8. Ympäristön topologia

7.2 Analysoitava data

Itse analysoitava data haettiin kahdesta eri lähteestä. Ensimmäinen lähde oli Jyväskylän ammattikorkeakoulun Mango-palvelu, joka sisälsi eräästä Saarijärvellä sijaitsevasta lammesta otettua mittaridataa. Tämä data sisälsi tietoa muun muassa veden sameudesta, lämpötilasta ja pinnankorkeudesta. Lisäksi palveluun oli tallennettu sää-tietoa koko mittauksen ajalta. Palveluun oli tallennettu dataa alkaen marraskuusta 2016. Datan näytteenottoväli oli yksi tunti. Data oli tallennettu käyttäen MySQL-tietokantaa. Tästä datasta haluttiin poimia reaaliaikaisesti veden sameutta koskevat tiedot, sekä sademäärä.

Toisena datalähteenä käytettiin Ilmatieteenlaitoksen avointa dataa. Ilmatieteenlaitos tarjoaa avoimen verkkopalvelunsa kautta sääennustedataa. Palvelu on maksuton, mutta vaati rekisteröitymisen minkä ohessa hyväksytty käyttäjä saa API-avaimen,

jonka avulla voi hakea haluamaansa dataa käyttäen ennalta tallennettuja parametrejä.

Tavoitteena oli kerätä haluttu data näistä lähteistä reaaliajassa. Datan keräämisen yhteydessä se muokattiin haluttuun formaattiin ja tallennettiin omaan tietokannan tauluun. Tämän jälkeen luotiin applikaatio, jolla pyrittiin etsimään korrelaatiota veden sameuden ja sademäärän välillä, sekä ennustettaisi veden sameuden muutoksia verrannollisesti tähän arvoon.

7.3 Spark-suunnitelma

Sparkin oli tarkoituksena toimia ympäristössä data-analytiikan pohjana. Sparkista käytettiin työtä aloittaessa viimeisintä stabiilia versiota 2.2.0. Spark haluttiin klusteroida suorituskyvyn parantamiseksi ja resurssien laajentamiseksi. Klusterointiin käytettiin Sparkin Standalone klusterointia, koska erillisen driver managerin käyttäminen ei juuri tuo lisäarvoa työn toimeksiantoon, jonka pääasiana oli tutkia Sparkin toimintaa ja arkkitehtuuria. Klusterin koneet olivat identtisiä keskenään, mutta yhdelle niistä ajettiin myös Master-prosessi. Tämän lisäksi jokaisella laitteella haluttiin ajaa neljää worker-prosessia näin helpottaen resurssien allokointia.

Datan keräämistä varten luotiin Sparkille applikaatio, jolla kerätään Ilmatieteenlaitokselta ja Mango-palvelusta tulevaa dataa. Reaaliaikaisten datavirtojen lähteenä toimi Apache Kafka, jonka integraatio toteutettiin käyttäen viimeisintä versiota paketista spark-streaming-kafka-0-8. Kyseinen versio siksi, että se tuki Python-ohjelmointikieltä ja oli versioltaan stabiili. Integraatio Kafkan ja Sparkin välille tehtiin käyttäen suoraa toteutustapaa. Datavuot tallennettiin erilliseen Cassandra-tietokantaan käyttäen PySpark Cassandra -pakettia.

Levossa olevan datan käsittely tapahtui applikaatioilla, jotka ottivat kerätystä datasta halutut tietueet ja laskelmoivat näistä halutun lopputuloksen. Lopputulokset tallennettiin tietokantaan erillisiin tauluihin. Tämä siksi, että Spark Streamingillä oli vain rajallinen määrä toiminteita, joita se pystyy tekemään ja näin pyritään vähentämään reaaliaikaisten applikaatioiden kuormaa tehden näistä varmempia toiminnaltaan. Lisäksi toimeksiannon mukaisesti näin pystyttiin vertaamaan Sparkin toimintaa niin reaaliaikaisessa datan ja levossa olevan datan käsittelyssä.

7.4 Kafkan-suunnitelma

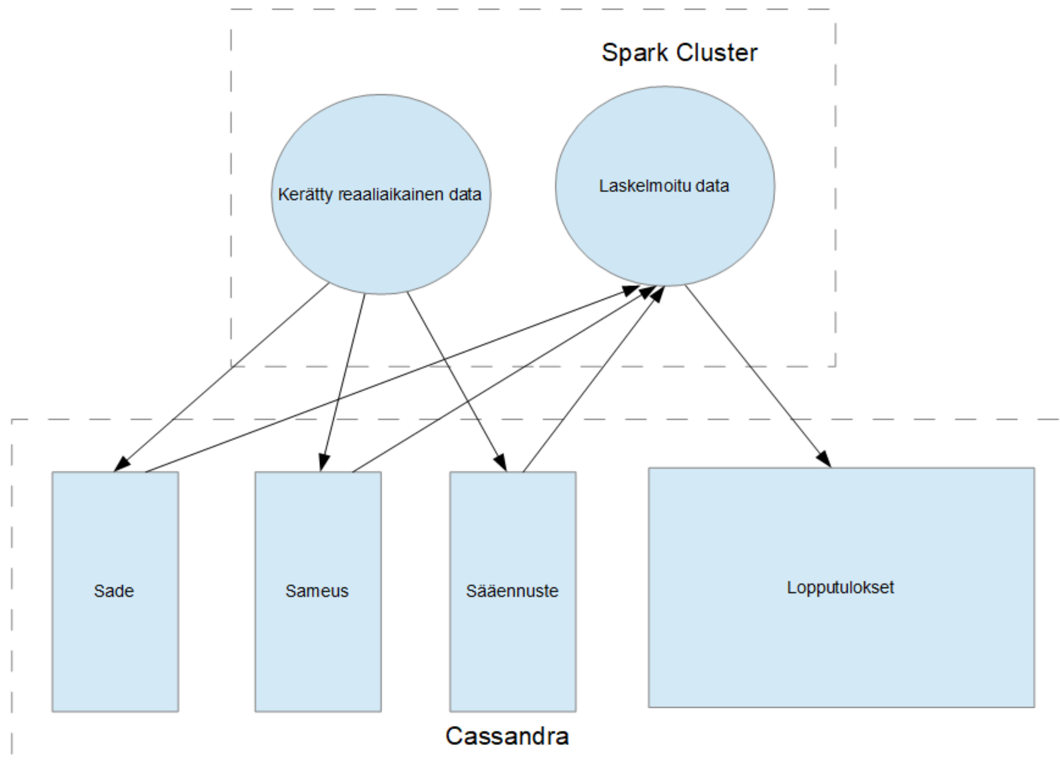
Reaaliaikaisten datavirtojen luonti toteutettiin käyttäen Apache Kafkaa. Tähän toiminteeksi valittiin Kafka sen hyvän suorituskyvyn takia, lisäksi Kafkan ja Sparkin integraatioon löytyi suoraan valmis paketti. Työssä käytettiin työn aloitushetkellä viimeisintä versiota Kafkasta, joka tässä tapauksessa oli versio 2.11-1.0.0. Kafkaa ajetaan yksittäisenä prosessina. Tällä pyrittiin vähentämään isäntäkoneen resurssien käyttöä, jotta Spark-klusterille voitiin allokoida mahdollisimman paljon resursseja.

Datan kerääminen Mango-palvelusta ja Ilmatieteenlaitoksen avoimesta datasta toteutettiin kirjoittamalla Python-skriptejä, jotka ajastetusti hakivat datan lähteistä. Ajastus toteutettiin crontabilla. Näiden applikaatioiden haluttiin myös tallentavan haettu tieto Kafka-koneelle. Data lähetettiin eteenpäin Sparkille käyttäen yhtä otsikkoa.

7.5 Tietokantasuunnitelma

Tietokantana työssä käytettiin Apache Cassandraa. Syynä valintaan toimi NoSQL-kantojen ominaisuuksien sopivuus työnkuvaan. Lisäksi myös Cassandrasta löytyi suora paketti, jolla integraatio Sparkin kanssa onnistui helposti. Myös Cassandrasta käytettiin työn aloitushetkellä uusinta stabiilia versiota, joka oli tässä tapauksessa versio 3.11.1.

Tietokantaan luotiin aluksi yhteensä neljä eri taulua, joista kolmea käytettiin kerätyn datan tallentamiseen ja yhtä lopputulosten tallentamiseen. Datan keräyksen yhteydessä otettiin datan tyypistä kuvaus, aikaleima ja itse mittauksen arvo. Lopputuloksien tauluun oli tarkoitus kerätä tietyllä ajalla tapahtuneiden mittauksien arvot ja arvojen keskinäiset lasketut tulokset (ks. Kuvio 9).



Kuvio 9. Tietokannan arkkitehtuuri

8 Toteutus

8.1 Sparkin asennus ja klusterointi

Ensimmäisenä luotiin jokaiselle ympäristön koneelle staattiset laitenimet. Näin ollen ympäristön jokaisessa osassa ei tarvitsisi käyttää niitä referoidessa laitteiden IP-osoitteita, vaan ennalta määritettyä laitenimeä (ks. Kuvio 10).

```
[root@master ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

# Enviroment hosts
192.168.51.181 master master.local
192.168.51.183 slave1 slave1.local
192.168.51.177 slave2 slave2.local
192.168.51.139 kafka kafka.local
192.168.51.168 cassandra cassandra.local
```

Kuvio 10. Staattiset laitenimet

Spark tarvitsi toimiakseen jokaiselle laitteella Javan asennettuna, joten Sparkin asennus aloitettiin asentamalla ja varmistamalla, että laitteessa on toimiva Javan komentotulkki. Tässä tapauksessa käytettiin laitteella uusinta versiota OpenJDK:sta (ks. Kuvio 11). Tämä tehtiin jokaiselle Spark-klusteriin tulevalle koneelle.

```
[root@master ~]# java -version
openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151-b12)
OpenJDK 64-Bit Server VM (build 25.151-b12, mixed mode)
```

Kuvio 11. Java

Koska työssä käytettiin ohjelmointikielenä Pythonia, tuli myös kaikilla koneilla olla asennettuna Python-komentotulkki. Tässä tapauksessa käytetiin Pythonin versiota 3.4.5, joka oli jokaiselle koneelle määritetty aliakselle ”python3” (ks. Kuvio 12).

```
[root@master ~]# python3 -V
Python 3.4.5
```

Kuvio 12. Python-versio

Tämän jälkeen ladattiin Sparkin asennuspaketti, joka oli saatavilla esimerkiksi Sparkin kotisivuilta. Paketin ladattua tuli sisältö purkaa haluttuun paikkaan. Tässä työssä käytettiin Sparkin hakemistona käyttäjälle ”sparkuser” luotua kotihakemistoa. Koska vakiona PySpark käytti Pythonin versiota 2.7.5, luotiin alias, jonka avulla PySpark (interaktiivinen komentokehote) käytti automaattisesti Python versiota 3.4.5 (ks. Kuvio 13).

```
[root@master ~]# cat /etc/profile.d/pyspark.sh
alias pyspark3="SPARK_PYTHON=python3 /home/sparkuser/spark-2.2.0-bin-hadoop2.7
/bin/pyspark"
```

Kuvio 13. PySpark3 alias

Tällä hetkellä pystyttiin jo käyttämään PySparkin interaktiivista komentoriviä (ks. Kuvio 14) ja todennettiin, että palvelua voitiin jo nyt käyttää tällä tasolla. Oli myös nähtävissä, että PySpark käytti haluttua Python-versiota.

```
[root@master ~]# pyspark3
Python 3.4.5 (default, Dec 11 2017, 14:22:24)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.propertie
s
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLe
vel(newLevel).
18/02/27 19:37:46 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
18/02/27 19:37:54 WARN ObjectStore: Failed to get database global_temp, return
ing NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 3.4.5 (default, Dec 11 2017 14:22:24)
SparkSession available as 'spark'.
>>>
```

Kuvio 14. Sparkin interaktiivinen komentokehote

Klusteroinnin toteutus aloitettiin ensimmäisenä luomalla suojattu yhteys, joka ei vaatinut salasanaa, master-koneen ja worker-koneiden välillä. Tämä toteutettiin luomalla salasanaton avainpari, josta jaettiin julkiset avaimet molemmille worker-koneille. Tämä siksi, että myöhemmässä vaiheessa voitiin worker-prosesseja hallinnoida keskitetysti master-koneelta.

Sparkin asennus sisälsi konfiguraatiotiedoston `spark-env` pohjan, johon voitiin muokata Sparkin ympäristö vastaamaan omia haluttuja ominaisuuksia. Tiedosto löytyi tässä tapauksessa hakemistosta:

/home/sparkuser/spark-2.2.0-bin-hadoop2.7/conf/

Pohja kopioitiin uudeksi tiedostoksi poistamalla tiedostonimestä ”template” osion, nyt tiedostoon lisättiin seuraavat parametrit:

export SPARK_WORKER_INSTANCES=4

export PYSPARK_PYTHON=python3

SPARK_MASTER_HOST=master

Edellä mainituilla parametreillä määritettiin jokaisen koneen käynnistävän neljä worker-prosessia, asetettiin PySpark käyttämään Pythonin versiota 3.4.5 ja määritettiin Sparkin Master-kone. Master-koneen määrityksessä voitiin käyttää laitteen IP-osoitetta tai koneen nimeä. Tässä tapauksessa käytettiin koneen nimeä, joka oli ennalta määritetty staattisesti jokaiselle koneelle. Tämän jälkeen tuli kyseinen konfiguraatiotiedosta kopioida jokaiselle klusterin koneelle samaan hakemistoon.

Tämän lisäksi master-koneelle tuli määrittää slave-koneiden osoitteet. Tämä tapahtui luomalla edellä mainittuun hakemistoon tiedosto nimeltä ”slaves”. Tässä voitiin käyttää myös hakemistosta löytyvää valmista pohjatiedostoa slaves.template. Slaves-tiedostoon riitti määrityksenä ainoastaan klusterin koneiden IP-osoitteet tai koneiden nimet, joille worker-prosessit haluttiin käynnistää (ks. Kuvio 15).

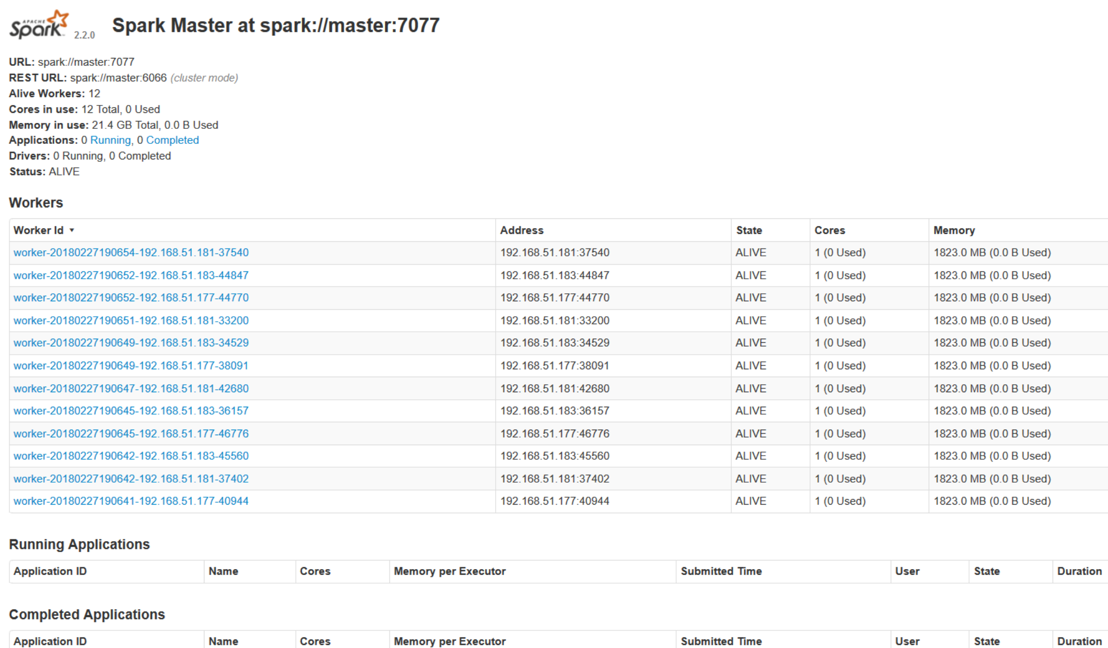
```
[root@master conf]# cat slaves
#
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

# A Spark Worker will be started on each of the machines listed below.
localhost
slave1
slave2
```

Kuvio 15. Slaves-konfiguraatiotiedosto

Nyt klusteri voitiin käynnistää käyttäen Sparkin paketin mukana tullutta ohjelmaa, jolla voitiin käynnistää kaikki klusterin prosessit samanaikaisesti:

/home/sparkuser/spark-2.2.0-bin-hadoop2.7/sbin/start-all.sh

Nyt voitiin todentaa klusterin toimivuus tarkistamalla Sparkin selainpohjainen käyttöliittymä (ks. Kuvio 16). Vakiona selaimen käyttöliittymä käytti osoitteenaan master-koneen osoitetta ja TCP-porttia 8080, mutta tämä voitaisiin muokata halutuksi määrittämällä se `spark-env`-konfiguraatiotiedostoon.


Spark 2.2.0 Spark Master at spark://master:7077

URL: spark://master:7077
REST URL: spark://master:6066 *(cluster mode)*
Alive Workers: 12
Cores in use: 12 Total, 0 Used
Memory in use: 21.4 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id ▾	Address	State	Cores	Memory
worker-20180227190654-192.168.51.181-37540	192.168.51.181:37540	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190652-192.168.51.183-44847	192.168.51.183:44847	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190652-192.168.51.177-44770	192.168.51.177:44770	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190651-192.168.51.181-33200	192.168.51.181:33200	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190649-192.168.51.183-34529	192.168.51.183:34529	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190649-192.168.51.177-38091	192.168.51.177:38091	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190647-192.168.51.181-42680	192.168.51.181:42680	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190645-192.168.51.183-36157	192.168.51.183:36157	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190645-192.168.51.177-46776	192.168.51.177:46776	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190642-192.168.51.183-45560	192.168.51.183:45560	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190642-192.168.51.181-37402	192.168.51.181:37402	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)
worker-20180227190641-192.168.51.177-40944	192.168.51.177:40944	ALIVE	1 (0 Used)	1823.0 MB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Kuvio 16. Spark-selainkäyttöliittymä

Tässä vaiheessa voitiin ladata jo Kafkan integraatiota varten tarvittava paketti. Paketti sijoitettiin Sparkin kotihakemiston `jars`-alihakemistoon, josta se voitiin sisällyttää applikaatioiden käynnistykseen (ks. Kuvio 17).

```
[root@master ~]# ls /home/sparkuser/spark-2.2.0-bin-hadoop2.7/jars/ | grep kafka
spark-streaming-kafka-0-8-assembly 2.11-2.2.0.jar
```

Kuvio 17. Spark-Streaming-Kafka

8.2 Kafka-palvelun käyttöönotto

Kafka-palvelun rakentaminen alkoi vastaavalla tavalla kuin Sparkinkin. Aluksi ladattiin uusin versio Apache Kafkasta, joka tässä tapauksessa oli versio 2.11-1.0.0. Paketin sisältö purettiin käyttäjälle ”kafkauser” luotuun kotihakemistoon. Kafka vaati toimiakseen usean prosessin, jotka kaikki ajettiin erillisissä ikkunoissa. Näin ollen näiden alasajo ja uudelleenkäynnistys sekä virheiden raportoinnin seuranta oli helpompaa. Ennen prosessien käynnistämistä, siirryttiin Kafkan kotihakemiston juureen, jotta prosessien käynnistyksessä ei tarvinnut käyttää absoluuttisia polkuja. Ensimmäisenä tuli käynnistää ZooKeeper komennolla:

bin/zookeeper-server-start.sh config/zookeeper.properties

Käynnistys tapahtui ajamalla paketin mukana tullut skripti, joka käynnisti ZooKeeper-prosessin käyttäen määritettyä konfiguraatiotiedostoa. Tässä toteutuksessa käytettiin ZooKeeperille porttia 2181 ja ikkunaa nimellä ”zk”.

Seuraavana käynnistettiin itse palvelun prosessi vastaavalla tavalla. Tämän prosessin käynnistykseen käytettiin myös paketin mukana tulevaa konfiguraatiotiedostoa, johon muokattiin seuraavat optiot vastaamaan ympäristöä:

broker.id=0

zookeeper.connect=localhost:2181

Myös palvelu käynnistettiin omassa ikkunassaan nimellä ”server”. Prosessi käynnistettiin seuraavalla komennolla:

bin/kafka-server-start.sh config/server.properties

Seuraavana luotiin tarvittava otsikko, jonka kautta data lähetetään eteenpäin (ks. Kuvio 18). Otsikolle käytettiin nimeä ”mango”.

```
[root@kafka kafka_2.11-1.0.0]# bin/kafka-topics.sh --describe --zookeeper localhost:2181 --
topic mango
OpenJDK 64-Bit Server VM warning: If the number of processors is expected to increase from
one, then you should configure the number of parallel GC threads appropriately using -XX:Pa
rallelGCThreads=N
Topic:mango     PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: mango    Partition: 0    Leader: 1       Replicas: 1     Isr: 1
```

Kuvio 18. Kafka otsikko

Tämän jälkeen luotiin otsikolle producer. Producerille luotiin oma ikkuna nimeltä ”it_produ”. Seuraavana tuli muokata käytettävät konfiguraatiotiedostot. Pohjana kaikille konfiguraatiotiedostoille oli paketin mukana tulleet tiedostot:

config/connect-standalone.properties

config/connect-file-source.properties

config/connect-file-sink.properties

Ensimmäisenä muokattiin Mangosta tulevan datan konfiguraatioita. Tiedostoon connect-file-source.properties lisättiin seuraavat rivit:

name=mango

file=/home/h9553/temp.csv

topic=mango

Näissä määritettiin producerille haluttu nimi, mitä otsikkoa käytetään ja mistä tiedostosta luetaan saapuva tieto. Seuraavana määriteltiin tiedostoon connect-file-sink.properties seuraavat optiot:

name=mango-file-sink

file=test.sink.txt

topics=mango

Näissä määrityksissä tuli esille, mitä tiedostoa käytettiin väliaikaisesti datan tallentamiseen, määritettiin filesink:in nimi ja sitä vastaava otsikko. Nyt kaikki uusi data, joka lisätään edellä mainittuun tiedostoon, lähetetään eteenpäin kuuntelijoille, jotka tässä tapauksessa tulevat olemaan Sparkin applikaatiot.

8.3 Tietokannan toteutus

Tietokantana tässä työssä käytettiin Apache Cassandraa. Cassandrasta käytettiin versiota 3.11.1 ja kuten aikaisemminkin toteutuksissa, Cassandran kotisivuilta löytyi ladattava paketti, joka sisälsi kaiken tarvittavan. Cassandra tarvitsee toimiakseen Pythonin version 2.7 ja Java tai OpenJDK:n version 8 valmiiksi asennettuna palvelimelle. Paketin sisältö sijoitettiin käyttäjän ”cuser” kotihakemistoon. Tämän jälkeen Cassandra-prosessi voitiin käynnistää palvelimella käyttäen paketin mukana tullutta käynnistysskriptiä:

bin/cassandra

Tässä vaiheessa voitiin tietokannan toiminta tarkastaa käyttämällä nodetool-työkalua (ks. Kuvio 19).

```
[root@cassandra apache-cassandra-3.11.1]# bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID
        Rack
UN  192.168.51.168  589.31 KiB  256          100.0%            2cb526c4-859d-4ff3-828e-f4ff
4cfd48ac  rack1
```

Kuvio 19. Cassandra nodetool status

Tämän jälkeen päästiin luomaan taulut käyttäen CQL:ää. CQL-komentokehote käynnistettiin käyttäen paketin mukana tullutta skriptiä:

bin/cqlsh cassandra

Nyt voitiin aloittaa tietokannan taulujen luominen. Ensimmäisenä luotiin `keyspace` nimeltä ”analytiikka”. Tämän luomisessa käytettiin optiota siten, että se luodaan ainoastaan yhdelle prosessille, koska tässä toteutuksessa ei tietokantaa klusteroitu säästäen näin isäntäkoneen resursseja. Tämä siksi, että tietomäärät olivat työssä kohtuullisen pienet, eikä tietokannalle tarvitse varata tästä syystä paljoa suoritustehoa. `Keyspace` luotiin käyttäen komentoa:

CREATE KEYSPACE analytiikka WITH replication = {'class': 'SimpleStrate gy', 'replication_factor': 1 } ;

Seuraavana siirryttiin arkkitehtuurissa asteelle, jossa voitiin luoda tauluja juuri luodun `keyspacen` sisälle:

USE analytiikka ;

Seuraavana luotiin ensimmäinen taulu ”sade”, johon tulee Mango-palvelusta saatu mitattu sateen määrä viimeiseltä tunnilta. Taulussa käytettiin sarakkeita aika, info ja arvo (ks. Kuvio 20). Sarakkeeseen aika tallennetaan otetun datan aikaleimatekstinä ja tätä käytetään oletusarvoisena avaimena. Saraketta info käytetään lyhyenä kuvauksena otetusta datasta tekstinä ja arvo sen hetken mitattuna arvona. Arvo tallennetaan desimaalina.

```
cqlsh:analytiikka> DESCRIBE sade;

CREATE TABLE analytiikka.sade (
    aika text PRIMARY KEY,
    arvo float,
    info 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.SizeTieredComp
actionStrategy', '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';
```

Kuvio 20. Sade-taulun rakenne

Käyttäen aikaisempaa toimintatapaa luotiin vastaava taulu myös sameudelle (ks. Kuvio 21). Sameuden taulussa käytettiin sarakkeina sade-taulua vastaavia sarakkeita.

```
cqlsh:analytiikka> DESCRIBE sameus;

CREATE TABLE analytiikka.sameus (
    aika text PRIMARY KEY,
    arvo float,
    info 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.SizeTieredComp
actionStrategy', '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';
```

Kuvio 21. Sameuden-taulun rankenne

Viimeinen tarvittava taulu datan keräykseen oli Ilmatieteenlaitokselta tuleva sääennusteen datan taulu. Myös tässä käytettiin edellisiä tauluja vastaavaa rakennetta (ks. Kuvio 22).

```
cqlsh:analytiikka> DESCRIBE ennuste ;

CREATE TABLE analytiikka.ennuste (
    aika text PRIMARY KEY,
    arvo float,
    info 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.SizeTieredComp
actionStrategy', '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';
```

Kuvio 22. Ennuste-taulun rakenne

Datan keräykseen tarvittavat taulut oli nyt luotu, joten seuraavana tarvittiin taulu, johon yhdistettäisiin tiettynä ajankohtana tallennetut tulokset ja niistä analytiikalla lasketut arvot. Tauluun tuotaisiin mittaushetkellä ollut veden sameus, mitattu sademäärä, ennuste sekä aikaisempi sameuden arvo ja sameudessa tapahtunut muutos (ks. Kuvio 23).

```
cqlsh:analytiikka> DESCRIBE tulokset ;

CREATE TABLE analytiikka.tulokset (
    aika text PRIMARY KEY,
    edellinen float,
    ennuste float,
    info text,
    muutos float,
    sade float,
    sameus float
) 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.SizeTieredComp
actionStrategy', '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';
```

Kuvio 23. Tulokset-taulun rakenne

Nyt toistaiseksi kaikki tarvittavat taulut oli luotu. Tauluja pystyttiin toteutuksen tarpeen mukaan vielä muokkaamaan ja luomaan uusia, mikäli näin tarve vaati.

8.4 Datan hakeminen Mango-palvelusta

Data tuli hakea Mango-palvelun tietokannasta, josta se lähetettäisiin eteenpäin Sparkille analysoitavaksi. Data oli tallennettu Mango-palvelun MySQL-tietokantaan. Tätä varten luotiin Python-ohjelmointikielellä ohjelma, joka hakee halutut tietueet tietokannasta ja kirjoittaa ne tiedostoon, josta Kafka poimii ne ja lähettää eteenpäin heti uuden tiedon saapuessa.

Tutkiessa Mangon tietokantaa huomattiin, että laitenimet ja kuvaukset oli tallennettu eri tauluun kuin näiden kuvaukset. Näin ollen oli SQL-kyselyssä haettava tietoa molemmista taulusta ja yhdistettävä ne kyselyssä. Laitenimi ja kuvaus löytyivät taulusta ”dataPoints”. Tutkimalla taulua saatiin selville, että sademäärän tunnus oli 465 ja sameuden 535 (ks. Kuvio 24).

```
MySQL [mango]> select id, xid, deviceName, name from dataPoints where id = '465' or id = '535';
+-----+-----------+----------------+------------------------+
| id  | xid       | deviceName     | name                   |
+-----+-----------+----------------+------------------------+
| 465 | DP_684389 | BTI sääasema   | Sademäärä [mm/tunti]   |
| 535 | DP_109195 | BTI EXO 1      | Sameus [FTU]           |
+-----+-----------+----------------+------------------------+
2 rows in set (0.00 sec)
```

Kuvio 24. Halutut tunnukset

Itse arvot oli tallennettu tauluun ”pointValues”, josta voitiin hakea dataa edellä mainituilla tunnuksilla (ks. Kuvio 25). Oli myös huomattava, että aikaleima oli kannassa määritetty Unixin aikaleimalla.

```
MySQL [mango]> select * from pointValues where dataPointId = '535' or dataPointId = '465' o
rder by id desc limit 10;
+----------+-------------+----------+------------+---------------+
| id       | dataPointId | dataType | pointValue | ts            |
+----------+-------------+----------+------------+---------------+
| 23052780 |         535 |        3 |       6.41 | 1519858800000 |
| 23052748 |         465 |        3 |          0 | 1519859949668 |
| 23051397 |         535 |        3 |       6.13 | 1519855200000 |
| 23051363 |         465 |        3 |          0 | 1519856349668 |
| 23050012 |         535 |        3 |       6.25 | 1519851600000 |
| 23049979 |         465 |        3 |          0 | 1519852749668 |
| 23048629 |         535 |        3 |       6.62 | 1519848000000 |
| 23048595 |         465 |        3 |          0 | 1519849149668 |
| 23047241 |         535 |        3 |       6.44 | 1519844400000 |
| 23047209 |         465 |        3 |          0 | 1519845549668 |
+----------+-------------+----------+------------+---------------+
10 rows in set (0.00 sec)
```

Kuvio 25. Tallennetut sade- ja sameusarvot

Kun oli selvitetty, miten haluttua tietoa voitiin hakea, voitiin kirjoittaa skripti, joka hakee tiedon automaattisesti ja tallentaa sen tiedostoon, josta data lähetetään eteen-

päin analysoitavaksi. Datan aikaleimoista oli huomattavissa, että sade- ja sameustietueet saapuvat kantaan eri aikoina. Tämän takia nähtiin parhaaksi, että molemmille luotiin oma skripti, jotta ne voitiin ajastaa ajettavaksi eri aikaan.

Ensimmäisenä luotiin skripti, joka haki sameuden arvon. Tästä käytettiin nimeä ”db_sameus.py”. Skripti on kokonaisuudessaan esitetty liitteessä 1. Skriptissä käytettiin Pythonin kirjastoja MySQLdb, csv ja os. MySQLdb-kirjaston avulla pystytään muodostamaan yhteys MySQL-tietokantaan ja sen eri funktioilla voidaan hakea ja muokata SQL-tauluja. Kirjastolla csv voidaan avata ja kirjoittaa Comma Separated Values (CSV) -tiedostoja. Kirjastoa os käytetään hakemaan tietoa käyttöjärjestelmän muuttujista tai halutusta lähteestä. Näin ollen koodiin itsessään ei tarvinnut tallentaa selkokielisenä muun muassa tietokannan käyttäjätunnuksia, vaan ne tallennettiin eri kohteeseen.

Ensimmäisenä määritettiin SQL-kysely, jolla haettiin oleelliset tiedot:

```
dbQuery_sameus = 'SELECT dataPoints.deviceName, dataPoints.name,
from_unixtime(floor(pointValues.ts/1000), "%Y/%m/%d
%H:%i:%s") as time, pointValues.pointValue, pointValues.dataType
FROM pointValues, dataPoints WHERE pointValues.dataPointId = 535
AND dataPoints.id = 535 ORDER BY pointValues.id DESC LIMIT 1;'
```

Samalla muutettiin Unixin aikaleima helpommin luettavaan päivämäärämuotoon. Tämän jälkeen haettiin tietokannan tunnukset ja määritettiin yhteyden muodostaminen tietokantaan:

```
db_host = os.environ.get('DB_HOST')

db_user = os.environ.get('DB_USER')

db_pass = os.environ.get('DB_PASS')

db_name = os.environ.get('DB_NAME')

connection=MySQLdb.connect(host=db_host, user=db_user,
passwd=db_pass, db=db_name)
```

Nyt määritettiin MySQLdb-kirjastosta kursori, jolla tehtiin aikaisemmin määritetty kysely tietokantaan. Tämän jälkeen tallennettiin juuri ajetun kyselyn tulokset muuttujaan ”tulos”:

kursori=connection.cursor()

kursori.execute(dbQuery_sameus)

tulos=kursori.fetchall()

Viimeisenä avattiin tiedosto, johon haluttiin tulokset tallennettavan. Tulokset lisätiin tiedoston loppuun rivi kerrallaan. Lopuksi katkaistiin yhteys tietokantaan.

file = csv.writer(open("/home/h9553/temp.csv","a"))

for i in tulos:

file.writerow(i)

connection.close()

Tämän pohjalta luotiin toinen skripti nimellä ”db_sade.py”, joka vastasi täysin edellä mainittua skriptiä, mutta kyselyllä haettiin sademäärän tietueita. Skripti on kokonaisuudessaan esitetty liitteessä 2.

Nyt skriptejä ajettaessa voitiin todentaa niiden toimivuus (ks. Kuvio 26). Ne hakivat viimeisimmän tietueen tietokannasta ja tallensivat ne haluttuun tiedostoon.

```
[root@kafka h9553]# ./db_sameus.py
[root@kafka h9553]# ./db_sade.py
[root@kafka h9553]# tail -2 temp.csv
DP_109195,BTI EXO 1,Sameus [FTU],2018/03/01 02:00:00,6.41,3
DP_684389,BTI sääasema,Sademäärä [mm/tunti],2018/03/01 02:19:09,0.0,3
```

Kuvio 26. Skriptien todennus

Datasta oli huomattavissa, että tietueet saapuivat tietokantaan tunnin välein. Sameuden tietue saapui jokaisella tasatunnilla ja sademäärä joka tunti noin 20 minuuttia yli. Näin ollen ajastettiin skriptit ajettavaksi hieman näiden aikojen jälkeen. Oli myös muistettava, että crontabiin ajastettaessa tuli komentoon sisällyttää kohde, josta se

haki ympäristön muuttujiin määritetyt tiedot. Lisättiin siis crontabiin seuraavat tiedot:

*30 * * * * root . /var/jail/enviroment && /home/h9553/db_sade.py*

*5 * * * * root . /var/jail/enviroment && /home/h9553/db_sameus.py*

8.5 Datan hakeminen Ilmatieteenlaitokselta

Ilmatieteenlaitos tarjoaa avointa säätietodataa oman API:nsa kautta. Palvelusta haluttiin hakea sademäärän ennustetta ja lähettää se Sparkille analysoitavaksi. Tätä dataa voitiin käyttää esimerkiksi apuna sameuden ennustukseen. Tieto haettiin taas Kafkalle käyttäen apuna Python-skriptiä.

Ennen tätä tuli rekisteröityä käyttäjäksi Ilmatieteenlaitoksen sivuilla. Kun rekisteröinti oli varmistettu, sai käyttäjä API-avaimen, jonka avulla hakuja datasta voitiin suorittaa. Avoimeen dataan oli tallennettu useita valmiita hakuja ja tässä tapauksessa käytettiin tallennettua hakua:

fmi::forecast::hirlam::surface::point::simple

Haku sisälsi 36 tunnin sääennustetiedon yksinkertaisessa esitysmuodossa. Tämän jälkeen pystyttiin aloittamaan itse skriptin kirjoittaminen. Skripti löytyy kokonaisuudessaan liitteestä 3. Skriptissä käytettiin Pythonin kirjastoja requests, xml.etree.ElementTree, csv ja os. Requests-kirjaston avulla voitiin tehdä haluttuja kyselyitä ja tulostaa niitä. Koska data on tallennettu XML:n kaltaisessa puumaisessa rakenteessa, tuli sen parsimiseen käyttää kirjastoa xml.etree.ElementTree.

Ensimmäisenä avattiin tiedosto, johon haun tulokset haluttiin kirjoittaa ja haettiin käyttöjärjestelmän muuttujista sinne ennalta tallennettu API-avain:

file = open("/home/h9553/temp.csv", "a")

api = os.environ.get('IT_API')

Seuraavaksi luotiin kysely, johon liitettiin aikaisemmin haettu API-avain. Lisäksi kyselyyn sisällytettiin mitä hakua käytetään, mitä parametrejä haettiin ja miltä paikkakun-

nalta. Tässä tapauksessa käytettiin aiemmin mainittua hakua, haettiin ainoastaan sademäärä ja käytettiin paikkakuntana Saarijärveä, josta myös Mango-palvelusta saatu data oli mitattu.

```
r = requests.get('http://data.fmi.fi/fmi-apikey/' + api + '/wfs?re
quest=getFeature&storedquery_id=fmi::forecast::hirlam::sur
face::point::simple&parameters=PrecipitationAmount&place=saarijär
vi')
```

Jotta dataa voitiin leikata, muodostettiin siitä yksi ElementTree-elementti:

```
root = ET.fromstring(r.content)
```

Tämän jälkeen poimittiin kyselyn tuloksesta kaikki halutut parametrit ja yhdistettiin ne yhdeksi kokonaisuudeksi.

```
t1 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}Parame
terName')]
t2 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}Time')]
t3 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}Parame
terValue')]
list = zip(t1,t2,t3)
```

Lopuksi muokattiin dataa hieman helpommin käsiteltävään muotoon, tallennettiin tiedostoon rivi kerrallaan ja suljettiin tiedosto.

```
for i in list:
i = ("Ilmatieteenlaitos," + ",".join(map(str, i)) + ',,')
i = i.replace('-', '/').replace('T', ' ').replace('Z', '')
file.write(i + '\n')
file.close()
```

Nyt tiedostoa ajettaessa voitiin todentaa sen toimivan halutulla tavalla saaden haluttu tuloste (ks. Kuvio 27).

```
[root@kafka h9553]# ./ilmatiede.py
[root@kafka h9553]# tail -36 temp.csv
Ilmatieteenlaitos,PrecipitationAmount,2018/03/01 21:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/01 22:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/01 23:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 00:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 01:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 02:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 03:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 04:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 05:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 06:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 07:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 08:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 09:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 10:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 11:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 12:00:00,0.0,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 13:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 14:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 15:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 16:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 17:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 18:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 19:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 20:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 21:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 22:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/02 23:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 00:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 01:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 02:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 03:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 04:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 05:00:00,0.1,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 06:00:00,0.2,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 07:00:00,0.2,,
Ilmatieteenlaitos,PrecipitationAmount,2018/03/03 08:00:00,0.2,,
```

Kuvio 27. Haettu sääennuste

Lopuksi vielä voitiin ajastaa myös tämä skripti ajettavaksi joka päivä tasan kello 12.00. Tämä toteutettiin lisäämällä seuraava rivi crontabiin:

*00 12 * * * root . /var/jail/enviroment && /home/h9553/ilmatiede.py*

8.6 Datan kerääminen Sparkilla

Kun data oltiin saatu haluttuun lähteeseen, josta se voitiin lähettää eteenpäin, oltiin valmiita lähettämään se kerättäväksi tietokantaan. Oli kirjoitettava applikaatio, jolla data kerätään Kafkan otsikosta, muokataan se haluttuun muotoon ja tallennetaan

haluttuihin tauluihin. Spark-applikaatio luotiin nimellä ”stable.py” ja se on kokonaisuudessaan esitetty liitteessä 4.

Ensimmäisenä luotiin itse pääohjelma, jossa aluksi määriteltiin konfiguraatiomuuttuja, johon tässä tapauksessa liitettiin applikaation nimi.

```
conf = SparkConf()\
.setAppName("stable")\
```

Konfiguraatioon voitaisiin myös asettaa esimerkiksi masterin osoite ja käytettävien resurssien määrät, mutta tässä tapauksessa nämä määritettiin applikaation ajossa, jotta niitä voitiin tarvittaessa muokata halutusti. Tämän jälkeen määritettiin CassandraSparkContext objektille edellä mainitut konfiguraatiot käytettäväksi:

```
sc = CassandraSparkContext(conf=conf)
```

Koska skriptissä haluttiin käyttää SparkSQL:n ominaisuuksia ja toiminteita tietojen tallentamiseen tietokantaan, määritettiin SparkSession, johon sisällytettiin CassandraSparkContext.

```
spark = SparkSession(sc)
```

Reaaliaikaisen datavuon luomiseksi tuli määritellä SparkStreamingille vielä StreamingContext, johon sisällytettiin aikaisempi CassandraSparkContext ja määritettiin aikaväli, jolla applikaatio ajaa sen pääohjelmaa. Tässä tapauksessa käytettiin alkuun 60 sekuntia.

```
ssc = StreamingContext(sc, 60)
```

DStreamin luomiseksi tarvittiin myös Kafkan osoitetieto ja otsikko, josta dataa kerätään. Myöskin tässä vaiheessa tiedot voitaisiin suoraan tallentaa muuttujaan. Näitä tietoja on kuitenkin helpompi muokata applikaatiota ajettaessa, joten käytettiin Pythonin kirjastoa sys. Nämä tiedot poimittiin applikaation ajokomennosta.

```
kohde, aihe = sys.argv[1:]
```

Itse DStreamin luominen tapahtui käyttäen KafkaUtils-kirjaston komentoa, johon sisällytettiin aikaisemmin määritetyt optiot.

stream = KafkaUtils.createDirectStream(ssc, [aihe],{'metadata.bro ker.list': kohde})

Nyt DStream oli luotu ja dataa voitiin vastaanottaa. Dataa muokattiin siten, että ensimmäisenä poimittiin datasta pelkästään hyötykuorma ja tämän jälkeen eroteltiin sen osiot käyttäen erottelumerkkinä pilkkua.

lines = stream.map(lambda x: x[1])

split = lines.map(lambda x: x.split(','))

Koska Spark luo jokaisesta saadusta tietueesta sen saapuessa oman RDD:nsä, määritettiin sen ajavan jokainen RDD aliohjelmaan nimeltä ”save”:

split.foreachRDD(save)

Pääohjelman lopuksi käynnistettiin StreamingContext ja määritettiin sen jatkavan, kunnes se sammutetaan.

ssc.start()

ssc.awaitTermination()

Nyt pääohjelma oli valmis. Tämän jälkeen luotiin aliohjelma, jonka tarkoituksena oli erotella data ja tallentaa se sille haluttuun tietokannan tauluun.

def save(time, split):

Aluksi luotiin selkeä erotin aikaleimalla, jotta applikaation ajon aikana voitaisiin seurata vastaanotettua dataa helpommin.

print("========= %s =========" % str(time))

Seuraavana nimettiin tulevan datan osiot käyttäen otsikkona nimiä, jotka vastasivat tallennettavan taulun saraketta. Kun tietueesta muutetaan Sparkin DataFrame, käytetään näitä nimiä sarakkeina.

rdd = split.map(lambda a: Row(info=a[3], aika=a[4], arvo=a[5]))

df = spark.createDataFrame(rdd)

Viimeisenä tarkistettiin, oliko vastaanotetuissa tietuissa sameutta, sadetta vai ennustetta koskevaa dataa. Tämä tapahtuu suodattamalla tulleesta datasta kaikki rivit, joiden ”info”-sarake vastaa halutun tietotyypin kuvausta. Tämän jälkeen laskettiin suodatettujen rivien lukumäärä. Esimerkkinä sameuden arvojen määrän tarkistus:

```
sam = df.filter(col('info').like("%Sameus%")).count()
```

Mikäli tietueet sisältävät sameutta koskevaa dataa, on laskettu lukumääräksi muuta kuin arvo 0. Tällöin siirryttäisiin valintarakenteeseen, jossa datasta suodatetaan halutut rivit ja tallennetaan haluttuun tietokannan tauluun. Mikäli tietueessa ei ole tätä tyyppiä koskevia tietueita, siirrytään ohjelman seuraavaan osaan.

```
if sam > 0:
dfsam = df.filter(col('info').like("%Sameus%"))
dfsam.write\
.format("org.apache.spark.sql.cassandra")\
.mode('append')\
.options(table="sameus", keyspace="analytiikka")\
.save()
```

Aikaisempaa toimintatapaa soveltaen luotiin valintarakenteet myös sademäärälle ja sen ennusteelle. Nyt applikaatio voitiin suorittaa käyttäen Sparkin asennuksen mukana tullutta spark-submit skriptiä. Käynnistyksessä oli kuitenkin myös määritettävä käytettäväksi aikaisemmin asennettu Kafkan integraatiopaketti, joka aikaisemmin tallennettiin Sparkin `jars`-hakemistoon. Tämän lisäksi määritettiin käytettäväksi myös Cassandran integraatiota varten tarvittava paketti ja tälle konfiguraatio, joka sisälsi tiedon Cassandran osoitteesta. Tämän jälkeen komennossa tuli esille missä Spark klusterin koneessa sijaitsee cluster manager -prosessi. Tämän jälkeen määritettiin missä polussa ajettava applikaatio oli. Lopuksi määritettiin Kafkan osoitetieto ja otsikon nimi.

```
/home/sparkuser/spark-2.2.0-bin-hadoop2.7/bin/spark-submit --jars
/home/sparkuser/spark-2.2.0-bin-hadoop2.7/jars/spark-streaming-
kafka-0-8-assembly_2.11-2.2.0.jar --packages anguenot:pyspark-cassan
```

dra:0.6.0 --conf spark.cassandra.connection.host="192.168.51.168" --master spark://master:7077 /home/h9553/stable.py 192.168.51.139:9092 mango

Nyt applikaatio oli käynnistetty halutuilla parametreillä. Ajettavia applikaatiota ja niiden tietoja voitiin nyt seurata Sparkin selainkäyttöliittymässä (ks. Kuvio 28).

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20180301232428-0012 (kill)	stable	12	1024.0 MB	2018/03/01 23:24:28	root	RUNNING	40 s

Kuvio 28. Ajossa olevat applikaatiot

Applikaation toimivuus voitiin nyt todentaa. Uusi sade tietue haettiin jokaisen tunnin 30:llä minuutilla, jonka jälkeen se lähetettiin heti eteenpäin. Haluttu tietue oli nyt saapunut perille ja Spark oli käsitellyt ja muokannut sen asianmukaisella tavalla (ks. Kuvio 29).

```
18/03/01 23:31:05 INFO CodeGenerator: Code generated in 567.372397 ms
+-------------------+----+--------------------+
|               aika|arvo|                info|
+-------------------+----+--------------------+
|2018/03/01 23:19:09| 0.0|Sademäärä [mm/tunti]|
+-------------------+----+--------------------+
```

Kuvio 29. Sademäärän DataFrame

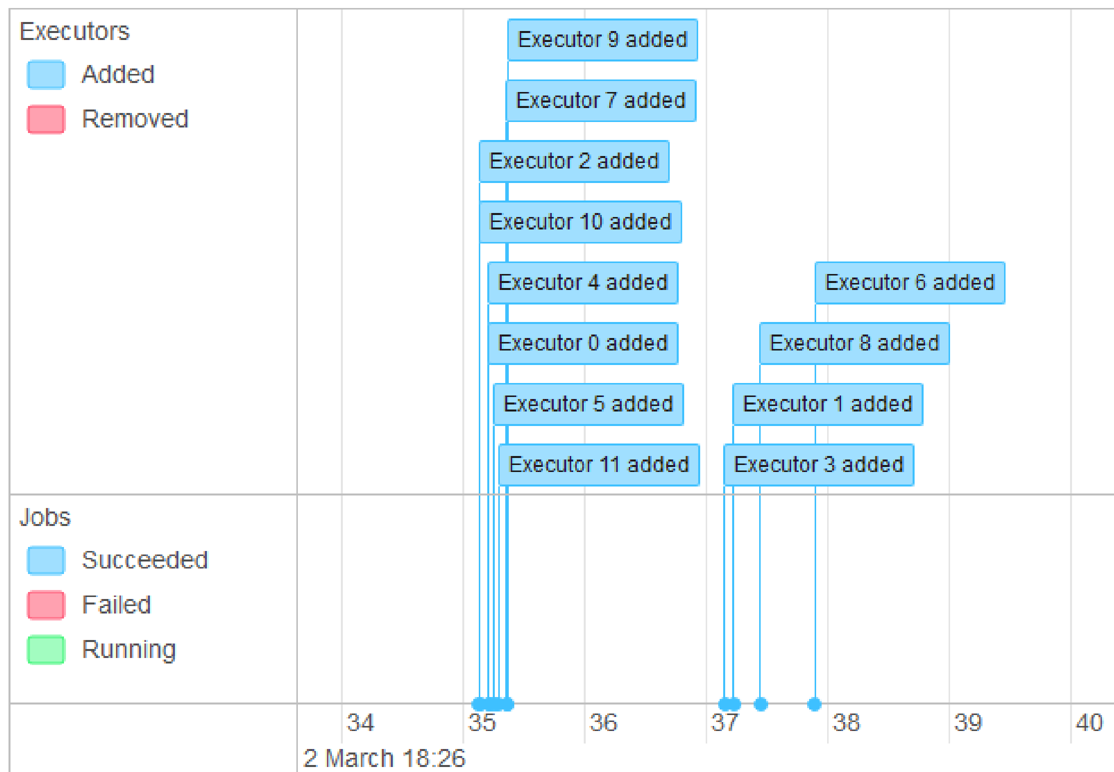
Tämä tietue oli myös tallentunut tietokannan tauluun onnistuneesti (ks. Kuvio 30). Näin ollen voitiin olettaa applikaation toimivan odotetusti.

```
cqlsh:analytiikka> SELECT * FROM sade;

 aika                | arvo | info
---------------------+------+----------------------
 2018/03/01 23:19:09 |    0 | Sademäärä [mm/tunti]

(1 rows)
```

Kuvio 30. Tietue tallennettuna tietokantaan

8.7 Sparkin toiminta datan keräämisessä

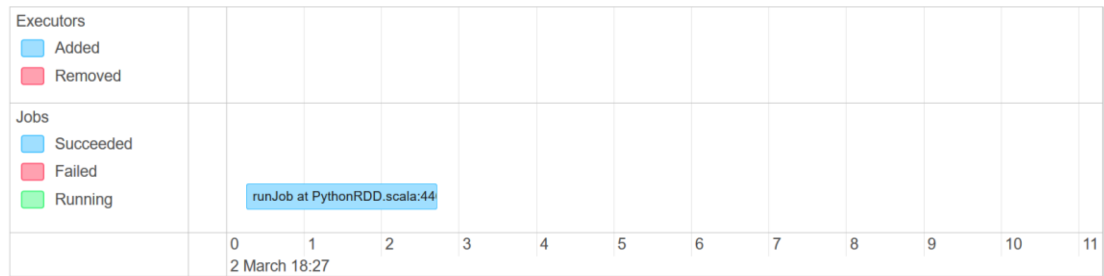
Sparkin selainkäyttöliittymästä voitiin nyt nähdä applikaatioiden toiminta sen eri vaiheissa. Kahden minuutin aikajana on nähtävissä liitteessä 5. Ajettaessa applikaatiota ”stable” voitiin seurata sen toimintaa. Ensimmäisenä Spark lisäsi jonotusmekanismin ja resurssien allokoinnin mukaisesti vapaana olevat worker-prosessit suorittajiksi (executor) (ks. Kuvio 31). Jokainen worker numeroitiin lisäyksen yhteydessä. Koska Sparkin standalone-klusterissa ei ollut muuta jonotusmekanismia, kuin FIFO ja työtä ajaessa ei ollut rajoitettu resurssien allokointia, oli applikaation käsittelyyn varattu kaikki workerit.



Kuvio 31. Suorittajien lisääminen

Ajon aikana voitiin myös tarkastella aktiivisesti jokaisen workkerin toimintaa ja resurssien käyttöä reaaliaikaisesti. Oli nähtävissä, että lukuun ottamatta driver progra-mia applikaatio oli käyttänyt jokaista worker-prosessia. Workereiden resurssit ja käyttö on nähtävissä liitteessä 6.

Tämän jälkeen applikaatio oli ajanut ensimmäisen työn. Job 0 oli suoritettu onnistuneesti ja sen prosessointiin oli kulunut aikaa 2 sekuntia (ks. Kuvio 32).

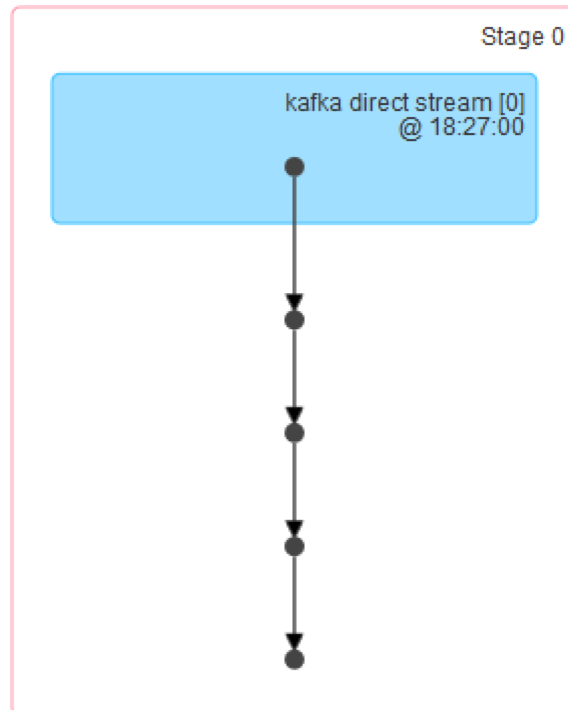


Completed Jobs (9)

Job Id ▲	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at PythonRDD.scala:446	2018/03/02 18:27:00	2 s	1/1	1/1

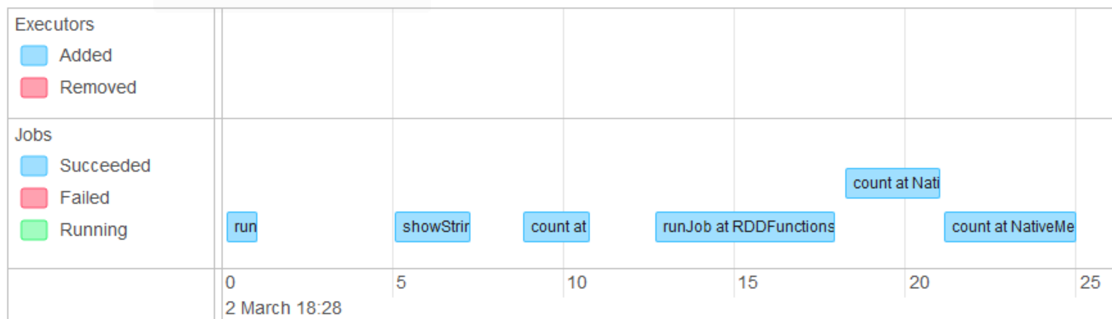
Kuvio 32. Job 0

Kun työtä tarkasteltiin tarkemmin, voitiin nähdä kyseessä olevan DStreamin muodostaminen Kafkalle (ks. Kuvio 33). Toisin sanottuna, applikaatio oli muodostanut yhteyden Kafkaan, mutta sillä hetkellä otsikko ei ollut tarjonnut uusia tietueita. Työ oli jaettu ainoastaan yhteen vaiheeseen.



Kuvio 33. Työn Job 0 tiedot

Tämän jälkeen applikaatio oli odottanut StreamingContext:ssa määritetyn 60 sekuntia, jonka jälkeen se oli muodostanut uudelleen yhteyden (ks. Kuvio 34). Nyt otsikossa oli uusi tietue, joka oltiin otettu käsittelyyn ja tehty sille applikaation koodissa määritetyt toimenpiteet. Uusin tietue oli manuaalisesti lähetetty sameuden arvo.



Kuvio 34. Tietueen vastaanottaminen

Toisinsanottuna tämän hetkisellä applikaation koodilla Spark jakoi toimenpiteet kuuteen eri työhön. Työt oli suoritettu yhdessä tai kahdessa eri vaiheessa (ks. Kuvio 35). Spark ei merkinnyt kaikkea koodissa tapahtuvaa aikajanalle. Tässä tapauksessa nähtiin DStreamin luominen, tietueesta saadun DataFramen tulostaminen, tietueen tyypin lukumäärien laskeminen ja saadun sameuden arvojen suodattaminen ja tietokantaan tallentaminen.

1	runJob at PythonRDD.scala:446	2018/03/02 18:28:00	0.9 s	1/1	1/1
2	showString at NativeMethodAccessorImpl.java:0	2018/03/02 18:28:05	2 s	1/1	1/1
3	count at NativeMethodAccessorImpl.java:0	2018/03/02 18:28:08	2 s	2/2	2/2
4	runJob at RDDFunctions.scala:36	2018/03/02 18:28:12	5 s	1/1	1/1
5	count at NativeMethodAccessorImpl.java:0	2018/03/02 18:28:18	3 s	2/2	2/2
6	count at NativeMethodAccessorImpl.java:0	2018/03/02 18:28:21	4 s	2/2	2/2

Kuvio 35. Syklin töiden vaiheiden määrä

Tarkastellessa näitä töitä voitiin nähdä, kuinka driver manager oli allokoinut tehtäviä klusterille. Eri vaiheet ja ne suorittaneet suorittajat löytyvät taulukosta 1.

Taulukko 1. Tiedon keräämisen applikaation töiden allokointi

<table>
<tr><th>Job</th><th>Toiminto</th><th>Stage</th><th>Executor / Host</th></tr>
<tr><td>1</td><td>Yhteyden muodos-taminen</td><td>1</td><td>2 / slave1</td></tr>
<tr><td>2</td><td>DataFramen tulos-taminen</td><td>2</td><td>6 / master</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">Sameuden arvojen lukumäärän laske-minen</td><td>3</td><td>6 /master</td></tr>
<tr><td>4</td><td>1 / master</td></tr>
<tr><td>4</td><td>Tietueiden suodat-taminen ja tallen-taminen</td><td>5</td><td>4 / slave1</td></tr>
<tr><td rowspan="2">5</td><td rowspan="2">Sateen arvojen lu-kumäärän laskemi-nen</td><td>6</td><td>3 / master</td></tr>
<tr><td>7</td><td>6 / master</td></tr>
<tr><td rowspan="2">6</td><td rowspan="2">Ennusteen arvojen lukumäärän laske-minen</td><td>8</td><td>7 / slave2</td></tr>
<tr><td>9</td><td>9 / slave2</td></tr>
</table>

8.8 Korrelaation laskeminen

Työn yhtenä osa-alueena oli tutkia sademäärän vaikutusta veden sameuteen. Korrelaation laskemiseen käytettiin aikaisempaa dataa, joka oli kerätty kesäkuulta 2017. Data tallennettiin tietokantaan tauluun ”saa_kesa”, käyttäen erillistä keyspacea ”legacy” (ks. Kuvio 36). Datassa oli yhteensä 716 riviä ja jokainen rivi sisälsi aikaleiman, sameuden- ja sateenmäärän arvon.

```
cqlsh:legacy> DESCRIBE saa_kesa ;

CREATE TABLE legacy.saa_kesa (
    aika text PRIMARY KEY,
    sade float,
    sameus float
) 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.SizeT
ieredCompactionStrategy', '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';
```

Kuvio 36. Kesäkuun säädatan taulu

Lisäksi tietokantaan tehtiin uusi taulu, johon lisättäisiin tulokset. Taululle käytettiin nimeä ”korrelaatio” (ks. Kuvio 37).

```
CREATE TABLE legacy.korrelaatio (
    pvm int PRIMARY KEY,
    korrelaatio float
) 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.SizeT
ieredCompactionStrategy', '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';
```

Kuvio 37. Korrelaation tuloksien taulu

Tämän datan pohjalta voitiin luoda skriptin, joka laskee korrelaation sateen ja sameuden muutosten välillä. Koska sameuden muutokset eivät tapahdu kovinkaan nopeasti ja data oli otettu näytteinä tunnin välein, laskettiin skriptillä kokonaisen kuukauden

korrelaatio, sekä erikseen jokaisen päivän korrelaatio. Kokonaisuudessaan applikaatio on nähtävissä liitteessä 7.

Ensimmäisenä määriteltiin jälleen CassandraSparkContext ja tämän konfiguraatio. Määritelmät tehtiin noudattaen aikaisemmin luodun ”stable.py”-applikaation mukaisia metodeja. Tämän jälkeen määritettiin muuttujat ”num” ja ”dict”:

```
num = int(0)
dict = {}
```

Muuttujaan ”num” alustettiin numero, jota myöhemmin käytettiin halutun päivämäärän valitsemisessa. Muuttujaa ”dict” taas käytettiin korrelaation arvojen tallentamiseen. Seuraavana luotiin DataFrame tietokannan kesäkuun säädatan taulusta, joka lisäämisen jälkeen järjestettiin päivämäärän mukaiseen järjestykseen nousevassa järjestyksessä:

```
df = spark.read\
.format("org.apache.spark.sql.cassandra")\
.options(table="saa_kesa", keyspace="legacy")\
.load()
df = df.orderBy(df.aika.asc())
```

Tämän jälkeen laskettiin koko kuukauden korrelaation arvo. Korrelaation laskemiseen käytettiin SparkSQL:n objektia, joka laskee korrelaation määritettyjen sarakkeiden välillä. Tuloksien laskennan jälkeen lisättiin laskettu arvo sanakirjaan ”dict”:

```
korrelaatio_kk = df.stat.corr('sameus','sade')
dict.update({int(0) : korrelaatio_kk})
```

Tämän jälkeen luotiin lista päivämääristä käyttäen Pythonin numpy-kirjastoa. Lista muutettiin RDD:ksi käyttäen Sparkin ”parallelize”-objektia, ja kerättiin muuttujaan sen tulokset.

```
pvm = np.arange(1, 31)
pvm = sc.parallelize(pvm).collect()
```

Nyt luotiin toistorakenne, joka toistuu 30 kertaa, eli yhtä monta kertaa kuin kesäkuussa on päiviä. Toistorakenteen sisälle tehtiin optio, jonka toiminteet suoritetaan, jos muuttujan ”num” arvo on vähemmän kuin 9:

```
for paiva in range(1, 30):
if num < 9:
```

Valintarakenteen sisällä ensimmäisenä valittiin minkä päivän arvot valitaan ja tämän jälkeen laskettiin korrelaatio siltä päivältä ja tallennettiin arvo sanakirjaan. Tämän jälkeen lisättiin ”num”-muuttujan arvoon 1.

```
korrelaatio_paiva = df.filter(col('aika').like('2017/06/0' + str(pvm[num]) + '%'))
cor = korrelaatio_paiva.stat.corr('sameus','sade')
dict.update({int(pvm[num]) : cor})
num = num + 1
```

Muuttujan arvon lisääntyessä siirryttiin toisen valintarakenteen sisälle, kun arvo oli yli 8. Tämä siksi, että muuttujan ollessa enemmän kuin 8, päivämäärät muuttuvat kaksinumeroisiksi. Tämän jälkeen suoritettiin aikaisemman valintarakenteen toiminteet.

```
korrelaatio_paiva = df.filter(col('aika').like('2017/06/' + str(pvm[num]) + '%'))
cor = korrelaatio_paiva.stat.corr('sameus','sade')
dict.update({int(pvm[num]) : cor})
num = num + 1
```

Kun kaikki päivät oli käyty läpi, muokattiin data sanakirjamuotoon, jossa se voitiin muuttaa DataFrameksi. Tässä tapauksessa siten, että jokainen rivi sisältää päivän tunnin ja sen korrelaation. Tämän jälkeen sanakirjasta luotiin DataFrame, jonka jälkeen tulokset tallennettiin tietokantaan.

Applikaatio oli nyt valmis ja se voitiin suorittaa. Kuten aikaisemminkin suorittamisen yhteydessä määritettiin tietokannan osoitetieto ja masterin tiedot, sekä käytettiin Cassandra - Spark -pakettia.

/home/sparkuser/spark-2.2.0-bin-hadoop2.7/bin/spark-submit –packages anguenot:pyspark-cassandra:0.6.0 --conf spark.cassandra.connection.host="192.168.51.168" --master spark://master:7077 /home/h9553/korrelaatio.py

Applikaatio käynnistyi onnistuneesti ja suoriutui ilman virheitä. Tämä näkyi myös Sparkin selainkäyttöliittymässä tilamerkinnällä ”finished” (ks. Kuvio 38).

Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time ▾	User	State	Duration
app-20180303220116-0001	korrelaatio	12	1024.0 MB	2018/03/03 22:01:16	root	FINISHED	2.6 min

Kuvio 38. Korrelaatio applikaation suorittaminen

Lisäksi tulokset näkyivät nyt tietokannassa halutulla tavalla. Tulokset ovat nähtävissä liitteessä 8.

8.9 Sameuden ennustaminen

Sameuden muutoksia haluttiin myös ennustaa aikaisemmin kerätyn datan pohjalta, käyttäen ennusteessa apuna Ilmatieteenlaitokselta saatua sademäärän ennustetta. Tapoja ennusteen tekemiseen on useita, mutta tässä toteutuksessa käytettiin Pythonin pakettia ”statsmodels.formula.api”. Datana käytettiin Mango-palvelusta ja Ilmatieteenlaitokselta kerättyä dataa. Sameuden ja sateen data oli otettu aikaväliltä 23.02.2018 kello 17:00 – 02.03 kello 10:00 ja kerätty sade-ennuste aikaväliltä 02.03.2018 kello 11:00 – 03.03.2018 kello 11:00. Data oli tallennettuna tietokantaan käyttäen ”stable.py”-applikaation mukaisia tauluja. Applikaatiolle käytettiin nimeä ”ennuste.py” ja se on kokonaisuudessaan liitteessä 9.

Ensimmäisenä määritettiin aikaisempia metodeja käyttäen SparkSession ja CassandraSparkContext. Tämän jälkeen luotiin tauluista sade, sameus ja ennuste omat DataFraminsa ja järjestettiin ne nousevassa järjestyksessä ajan mukaisesti. Tämän jälkeen laskettiin rivimäärät sameuden ja ennusteen DataFrameista.

ennuste_count = df_ennuste.count()

sameus_count = df_sameus.count()

Tämän jälkeen luotiin numerolista, jossa oli numeroituna yhtä monta arvoa, kun sameuden DataFramessa oli rivejä. Tätä käytettiin apuna myöhemmässä vaiheessa merkitsemään päivämääriä helpommin laskettavassa muodossa. Listan luomiseen käytettiin numpy-kirjaston `arange`-objektia.

num = np.arange(0, sameus_count)

Tämän lisäksi alustettiin muuttuja ”ennuste_num”, jota käytettiin myöhemmin apuna ennusteen arvojen poimimiseen.

ennuste_num = 0

Koska tauluja joudutaan muokkaamaan paljon ja Statsmodels-sivuston yleisessä ohjeistuksessa oli kauttaaltaan käytetty Pandas-kirjaston DataFrameja, muutettiin Sparkin DataFramet Pandasin DataFrameiksi.

pdf_sade = df_sade.toPandas()

pdf_sameus = df_sameus.toPandas()

pdf_ennuste = df_ennuste.toPandas()

Tämän jälkeen muutettiin sameuden arvon sarakkeen nimi sameudeksi:

pdf_sameus['sameus'] = pdf_sameus['arvo']

Tämän jälkeen tallennettiin sameuden DataFrame uuteen muuttujanimeen ”pdf” ja pudotettiin pois rivit, joita ei tässä tapauksessa ollut käytössä.

pdf = pdf_sameus.drop(['info', 'arvo'], axis=1)

Nyt liitettiin samaan DataFrameen myös sateen arvo nimellä ”sade” ja aikaisemmin luotu numeroitu lista. Tässä tapauksessa käytettiin numeroitua listaa ajan määrittämiseen. Tämä toteutettiin koodin kirjoittamisen helpottamiseksi, koska tiedettiin, että näytteet oli otettu ainoastaan tunnin välein. Jokainen ’num’-luku vastasi tiettyä päivämäärää ja tuntia, jolloin näyte oli otettu. Jokaisen rivin välillä oli siis kulunut yksi tunti.

```
pdf['sade'] = pdf_sade['arvo']
pdf['num'] = num
```

Seuraavana luotiin toistorakenne, joka käydään läpi yhtä monta kertaa, kun ennusteessa oli rivejä.

```
for i in range(1, ennuste_count):
```

Toistorakenteen ensimmäisenä toimenpiteenä alustettiin data, josta ennuste tultiin tekemään.

```
data = {"sameus": pdf['sameus'], "sade": pdf['sade'], "num": pdf['num']}
```

Tämän jälkeen määritettiin kaava ennusteelle. Tässä tapauksessa määritettiin laskettavaksi sameus suhteessa sateen määrään ja aikaan. Datana käytettiin aikaisemmin määritettyä sanakirjaa ”data”.

```
tulos = sfm.ols(formula='sameus ~ sade + num', data=data).fit()
```

Tämän jälkeen otettiin arvoista ennuste. Ensimmäinen arvo oli vakio 1. Toisena otettiin ennusteesta halutun rivin arvo käyttäen aikaisemmin määritettyä muuttujaa ” ennuste_num”. Ensimmäisellä kierroksella tämä siis poimi muuttujalle alustetun rivin 0, josta Pandas aloittaa indeksoinnin. Kolmantena määritettiin ajanjakso, eli tässä tapauksessa seuraava tunti. Jälleen koska Pandas aloittaa indeksoinnin numerosta 0 oli DataFramen ”pdf” viimeinen indeksi yhden alle lasketun rivimäärän. Näin ollen voitiin suoraan käyttää laskettua rivimäärää määrittämään seuraava ajankohta.

```
ennuste = tulos.predict([1, pdf_ennuste['arvo'][ennuste_num], sa
meus_count], transform=False)
```

Lopuksi lisättiin DataFramen (”pdf”) loppuun saadut tiedot, jotta ne voitiin ottaa seuraavalla kierroksella huomioon ennustetta laskettaessa.

pdf = pdf.append({"sameus" : ennuste[0], "sade": pdf_ennuste['ar vo'][ennuste_num], "num": int(sameus_count)}, ignore_index=True)

Lopuksi kasvatettiin muuttujien ”sameus_count” ja ” ennuste_num” arvoa yhdellä, jotta seuraavalla kierroksella saatiin halutut seuraavan ajankohdan arvot. Lopuksi ennuste tulostettiin komentokehotteeseen.

sameus_count = sameus_count + 1

ennuste_num = ennuste_num + 1

print(ennuste)

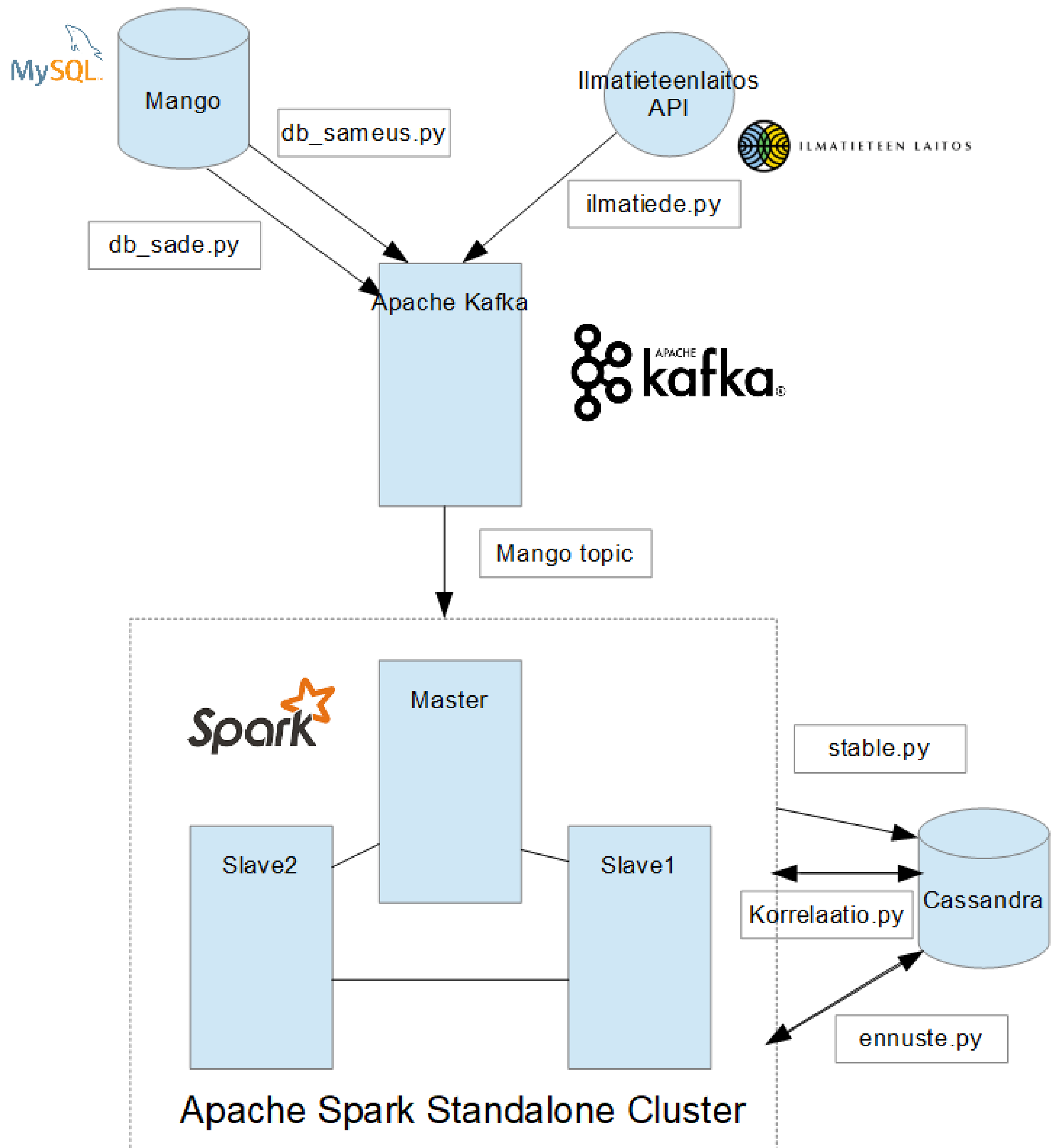
Nyt applikaatio voitiin suorittaa. Suorituksessa tuli jälleen käyttää Cassandra - Spark -pakettia ja määrittää Cassandran osoitetiedot.

/home/sparkuser/spark-2.2.0-bin-hadoop2.7/bin/spark-submit –packa ges anguenot:pyspark-cassandra:0.6.0 --conf spark.cassandra.connec tion.host="192.168.51.168" --master spark://master:7077 / home/h9553/ennuste.py

9 Tulosten analysointi

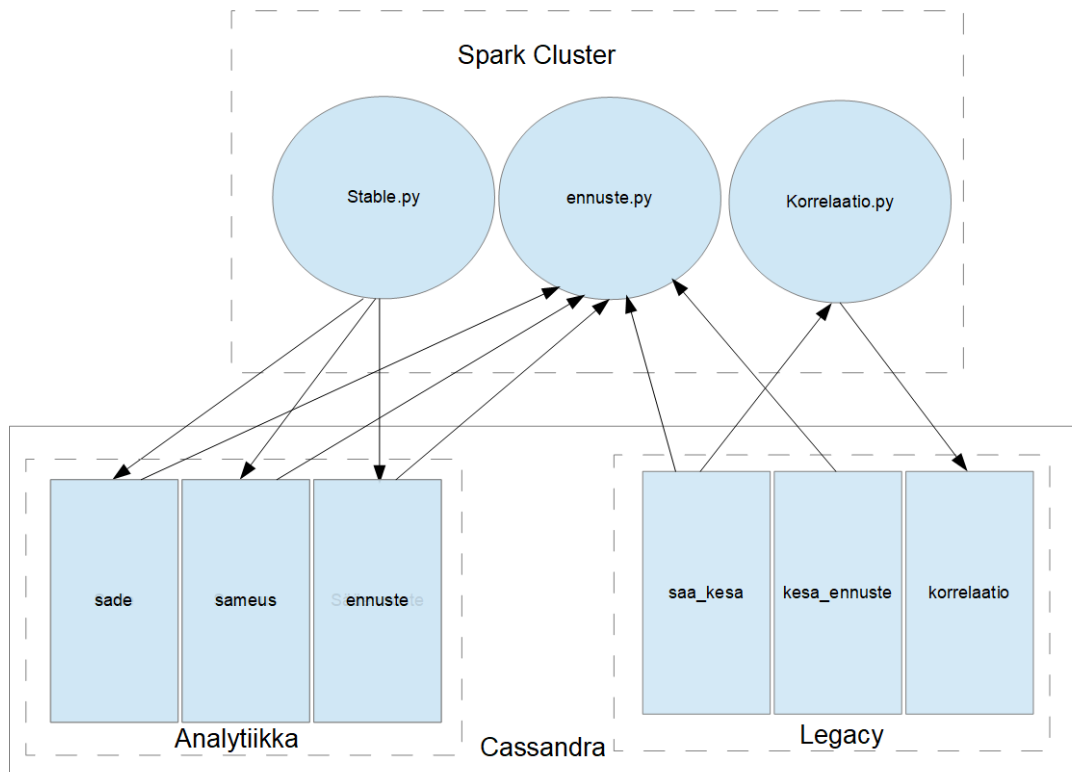
9.1 Ympäristö

Lopulliseen ympäristöön tuli joitakin muutoksia verrattuna alkuperäiseen suunnitelmaan, vaikka pääasiallisesti aikaisemmin mainitut komponentit eivät olleet muuttuneet (ks. Kuvio 39). Alkuperäisestä kahdesta Kafkalta tulevasta datavuosta luovuttiin, koska sekä Mango-palvelusta tuleva data, että Ilmatieteenlaitokselta tuleva data saatiin implementoitua samaan datavuohon. Näin ollen säästettiin resursseja niin Kafkalta kun Sparkiltakin, koska nyt ei tarvinnut ajaa erikseen useampaa applikaatiota datan keräämiseen.



Kuvio 39. Lopullinen topologia

Suurimmat muutokset olivat tapahtuneet tietokannan tauluihin, joita oli tarvinnut lisätä suhteellisen paljon (ks. Kuvio 40). Myös koska analytiikassa oltiin jouduttu käyttämään aikaisempaa dataa, oli se eristettävä erilliseen keyspaceen ja uusiin tauluihin.

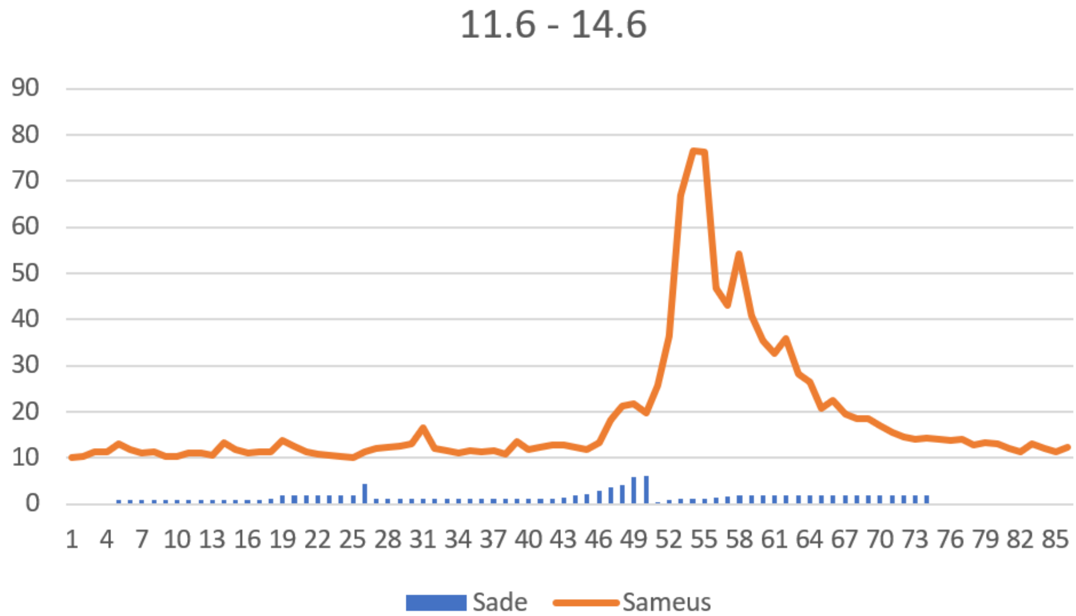


Kuvio 40. Lopulliset taulut

9.2 Korrelaation tulosten analysointi

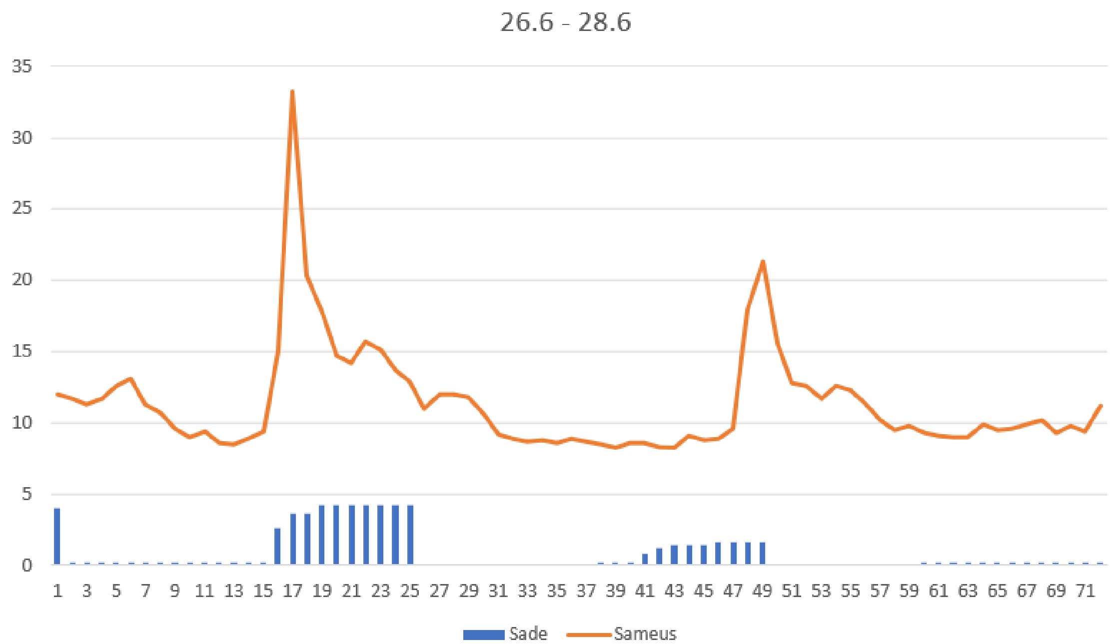
Laskettu korrelaation arvo oli sameuden muutoksen suhde sateen määrään määritettyinä aikajaksoina. Korrelaatio on positiivinen, mikäli toisen arvon kasvaessa kasvaa myös toinenkin arvo. Negatiivinen tulos tarkoittaa sitä, että kun toinen arvo kasvaa, toinen vähenee. Korrelaation on suurempi mitä lähempänä sen arvo on yhtä. Aluksi tuloksia tarkastellessa sai kuvan, ettei korrelaatiota ole, koska arvot heittelevät melko radikaalisti. Osa päivistä oli myös saanut korrelaatioksi arvon 0, mutta näistä oli nähtävissä, että kyseisenä päivänä ei ollut satanut. Liitteessä 10 on nähtävissä graafinen esitys koko kuukauden korrelaatiosta.

Kuitenkin tarkemmassa tarkastelussa nähtiin syy miksi korrelaation arvot ovat päiväkohtaisesti vaihdelleet näin radikaalisti. Kun tarkasteltiin ajankohtia, jolloin oli satanut paljon, voitiin nähdä, miten sameus oli hetkellisesti kasvanut, kun sade oli alkanut. Esimerkkinä ajanjakso 11-14.6 (ks. Kuvio 41).



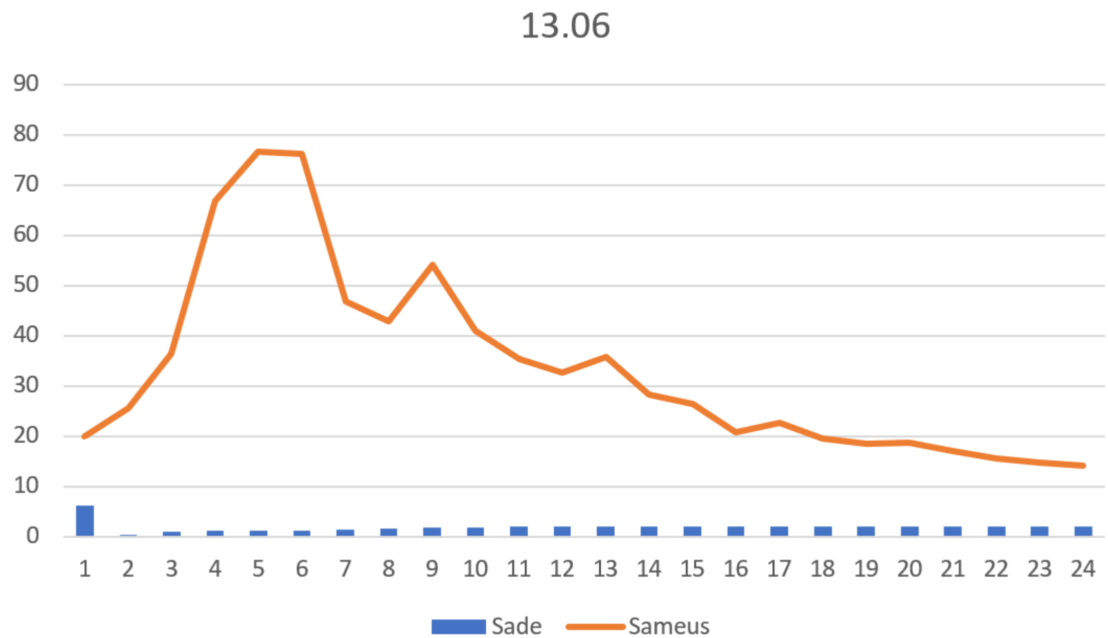
Kuvio 41. Sameuden muutos sateen kasvaessa

Korrelaatio oli ollut päivämäärällä 12.06. 0.73, sillä sade oli noussut äkillisesti suureen määrään. Tämä oli toistaiseksi nostanut sameuden arvon, mutta oli siitä hetken kuluttua tippunut, vaikkakin sade ei ollut loppunut, mutta vähentynyt huomattavasti. Oli myös nähtävissä, kuinka sateen loppumisen jälkeen sameus oli jatkanut laskemistaan. Sama huomio voitiin tehdä myös toiselta ajanjaksolta, jolloin korrelaatio oli ollut korkealla (ks. Kuvio 42).



Kuvio 42. Sameuden äkillinen nousu sateen alkaessa

Kun taas lähdettiin tarkastamaan mistä johtuu tuloksissa olevat negatiiviset arvot, nähtiin syy myös tälle käytökselle. Tarkasteltiin esimerkkinä päivää 13.6., jonka korrelaatio oli ollut -0.367 (ks. Kuvio 43). Aluksi oli nähtävissä aikaisempaa vastaava käytös: korkea sademäärä oli nostanut sameuden arvon korkeaksi, jonka jälkeen se oli lähtenyt laskuun. Syy miksi korrelaatio oli negatiivinen, johtuu selkeästi näytteenottovälistä. Sameus oli laskenut koko päivän ajalla, kun taas samalla sademäärä oli hiljalleen noussut. Näin ollen applikaatio oli huomioinut sameuden laskevan samalla, kun sademäärä oli hieman noussut. Sademäärän nousu ei ollut suuri, mutta sameuden laskeminen oli. Tämä oli aiheuttanut mittaustuloksessa negatiivisen arvon.



Kuvio 43. Negatiivinen korrelaatio

Yhteenvetona korrelaation tuloksista oli nähtävissä, että suuret sademäärät kasvattivat sameuden arvoa todella nopeasti ja todella suureksi. Kuitenkin äkillisen nousun jälkeen sameus oli laskenut aina. Korrelaation arvosta saatiin suuntaa sameuden käytöksestä ja voitiin osoittaa kohdat, joissa sameus oli muuttunut ja millä tavalla. Tulos ei kuitenkaan ollut tarkka, koska siinä oli staattinen näytteenottoväli, joka saattoi aiheuttaa varianssin tutkimustuloksissa. Mitä pienempi laskelmointiväli on, sitä selkeämmäksi tämä varianssi tulee. Lisäksi mikäli laskelmointiväliä kasvatetaan liikaa, isommat muutokset voivat jättää pienemmät muutokset huomiotta.

9.3 Ennusteen tulosten analysointi

Ensimmäinen huomattava asia ennusteen tuloksista oli, että applikaatio oletti veden sameuden arvon ainoastaan nousevan tasaisesti ajan myötä, vaikka sääennusteen mukaan tulisi sataa jonkin verran (ks. Taulukko 2).

Taulukko 2. Maaliskuun sameuden ennuste

Aika	Ennustettu sademäärä	Mitattu sademäärä	Ennustettu sameus	Mitattu sameus
02.03.2018 11:00	0.0	0	6.51719345	6.04
02.03.2018 12:00	0.1	0	6.5238484	6.34
02.03.2018 13:00	0.1	0	6.53050336	5.76
02.03.2018 14:00	0.1	0	6.53715831	5.85
02.03.2018 15:00	0.1	0	6.54381326	6.19
02.03.2018 16:00	0.1	0	6.55046821	6.1
02.03.2018 17:00	0.1	0	6.55712317	5.95

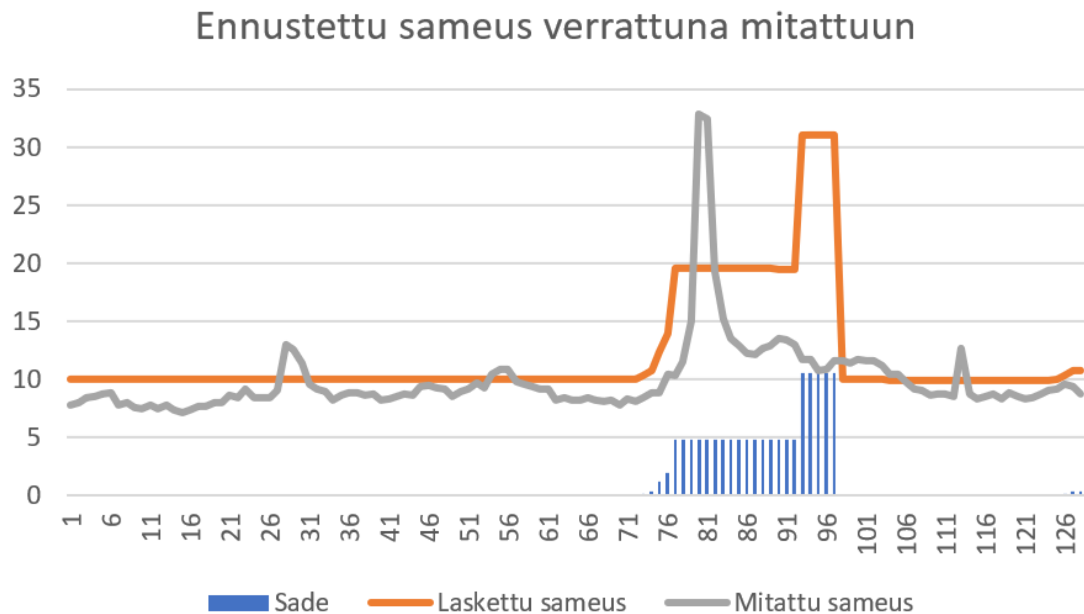
Tässä vaiheessa tarkistettaessa ennusteen tiedoissa oli nähtävissä varoitus, joka viittasi kaavaan syötettyyn dataan (ks. Kuvio 44).

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 sameus   R-squared:                       0.496
Model:                            OLS   Adj. R-squared:                  0.490
Method:                 Least Squares   F-statistic:                     94.30
Date:                Thu, 08 Mar 2018   Prob (F-statistic):           2.96e-29
Time:                        02:06:30   Log-Likelihood:                -86.969
No. Observations:                 195   AIC:                             179.9
Df Residuals:                     192   BIC:                             189.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      5.4391      0.059     92.327      0.000       5.323       5.555
sade       -8.962e-13      0.416  -2.16e-12      1.000      -0.820       0.820
num            0.0067      0.001     10.788      0.000       0.005       0.008
==============================================================================
Omnibus:                       20.374   Durbin-Watson:                   0.593
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               26.202
Skew:                           0.686   Prob(JB):                     2.04e-06
Kurtosis:                       4.158   Cond. No.                     1.71e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.71e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Kuvio 44. Ennusteen tulokset

Tämä johti syötetyn datan tarkastukseen, josta voitiin heti huomioida virhe. Mango-palvelusta saatu sademäärä oli ollut yli kuukauden aikana ainoastaan 0. Tästä syystä ennuste ei tältä ajalta ollut realistinen. Tästä syystä laskelmoitiin uusi ennuste käyttäen aikaisempaa dataa, joka oli mitattu kesäkuulta 2017. Koska Ilmatieteenlaitos ei tarjonnut aikaisempaa sääennustedataa, käytettiin ennusteena mitattua arvoa verratakseen lasketun ennusteen paikkansapitävyys. Näin ollen applikaatioon tehtiin muutamia muokkauksia. Muokattu versio on nähtävissä liitteessä 11.

Nyt ajettaessa muokattu applikaatio saatiin haluttu tulos sille määritetyistä arvoista (ks. Kuvio 45). Pääosin ennuste mukaili mitattuja arvoja suhteellisen lähellä, mutta oli paljon hillitympi muutoksissa. Ennustettu arvo myös seurasi paljon tarkemmin sademäärää. Pääosin ennuste oli suhteellisen lähellä, mutta ei osannut ennustaa sateen alussa tapahtuvaa piikkiä, vaan ennusti sen tapahtuvan kohdassa, jossa sade oli suurimmillaan. Muuten arvot pysyivät suhteellisen samassa tasossa. Kaavalle syötetty rivimäärä oli suhteessa todella pieni ja laskelmointiin saataisiin varmasti lisää tarkkuutta tätä kasvattamalla.



Kuvio 45. Ennustettu sameus verrattuna mitattuun

10 Johtopäätökset

10.1 Parannusehdotukset

Jatkoa ajatellen ympäristöstä voisi parantaa lähinnä koneiden resurssien lisäämisellä ja lisäksi sekä Kafka että Cassandra voitaisiin klusteroida suorituskyvyn parantamiseksi. Lisäksi Mango-palvelun tietokantaan voitaisiin lisätä liipaisut, jotka automaattisesti uuden arvon saapuessa lähettäisivät sen Kafkan vastaanottavaan datavuohon. Tätä ei voitu nyt toteuttaa sillä kyseiseen tietokantaan oli ainoastaan lukuoikeudet. Sparkin osalta, jonkin ulkoisen driver managerin asentaminen monipuolistaisi työskentelyä. Lisäksi kaikille ympäristön koneille tulisi panostaa suuresti tietoturvan kehittämiseen, joka on tässä työssä jokseenkin laiminlyöty, koska se ei olisi tuonut lisäarvoa toimeksiantoon.

Suurimmat parannusehdotukset tulevat ehdottomasti applikaatioiden koodiin. Koodi ei ole optimoitua ja on jokseenkin hidasta ajaa. Tämän lisäksi siinä on mitä todennäköisimmin turhia vaiheita, jotka voitaisiin kiertää ja näin nopeuttaa sen ajamista.

Applikaatioiden osalta ennusteapplikaatio tarvitsisi ehdottomasti lisää dataa sen kaavaan. Tässä hankaluuden tuottaa sameuden vaihtelun ero kesäisin ja talvella, sekä aikaisemmin kerätyn datan laatu. Yrittäessäni syöttää dataa vuoden ajalta ennusteelle sen laatu tippui huomattavasti. Tähän syynä on mitä todennäköisimmin vuodenaikojen vaihtelu, jota applikaatio ei osaa ottaa huomioon, sekä datassa olevat selvästi virheelliset rivit. Korrelaation laskemisessa tulisi aikaväli kohdentaa tarkemmin tilanteeseen sopivammaksi. Näiden lisäksi korrelaation ja ennusteen applikaatiot tulisi tuotantoympäristössä automatisoida.

10.2 Pohdinta

Työn tavoitteina oli käsitellä Sparkin arkkitehtuuria ja sen sisäisiä toiminteita, sekä sen lisäksi suorittaa sen avulla data-analytiikkaa levossa olevaan dataan ja lähes reaaliaikaisiin datavirtoihin, käyttäen Python-ohjelmointikieltä ja sen valmiita data-analytiikkakirjastoja. Pääasiallisesti tavoitteisiin päästiin, joskin jotkut aiheet jäivät hieman suppeammaksi, kuten esimerkiksi Pythonin muiden kirjastojen käyttö. Lisäksi itse analytiikkaan olisi voinut keskittyä enemmän, koska työn toteutuksessa meni myös paljon aikaa ympäristön muiden komponenttien hallitsemiseen.

Työn pääasiallisena työkaluna toiminut Apache Spark on todella laaja ja potentiaalinen. Tässä työssä käsitelty osuus on vasta pintaraapaisu sen sisäiseen toimintaan, eikä myöskään kaikkeen voida perehtyä osaamatta muita ohjelmointikieliä. Spark on kuitenkin todella toimiva hajautettu laskentaympäristö, jossa on valtava määrä toimivia ominaisuuksia ja työkaluja datan analysointiin ja laitteiden resurssien allokoimiseen klusteroitujen laitteiden välillä. Lisäksi selkeä applikaatioiden toiminnan visualisointi ja hyvä käyttöliittymä niiden seurantaan tarjoaa hyvät lähtökohdat applikaatioiden optimoimiseen.

Pääasiallisesti ympäristö oli toimiva ja stabiili ja se saatiin käytännössä mutkitta pystyyn. Ympäristö oli käytännössä koko prosessin ajan toiminnassa. Toki ongelmitta tästä ei selvitty, vaan kuten kaikissa IT-infrastruktuureissa toimivien osien välillä on toimenpiteitä, jotka tulee tehdä niiden ylläpitämiseksi. Suurin osa näistä ongelmista tuli resurssivajeen takia. Kaikki ympäristön koneet olivat toiminnassa yhdellä fyysisellä koneella, jossa oli niukasti resursseja ympäristön pyörittämiseen. Lisäksi pyrin

optimoimaan koko ajan muistin allokointia Sparkin klusterille. Tämän takia, jotkin Kafkan palveluista kaatuivat muutamaan otteeseen, koska isäntäkone ei pystynyt allokoimaan niille muistia.

Lisäksi jokaisella palvelimella on todella rajallinen määrä levytilaa. Tämä ei koitunut ongelmaksi muualla, kuin Sparkin koneilla. Spark tallentaa yllättävän paljon lokeja ja tietoja levylle, jonka ansiosta applikaatiot joutuivat ajamaan itsensä alas.

Suurimmat ongelmat ja eniten aikaa vienyt osa-alue oli Python-koodin tuottaminen. Omat lähtökohtani koodaukseen ovat käytännössä perusteet ja olen opinnoiltani keskittynyt IT-palveluiden hallintaan. Sen takia ympäristön pystyttämiseen meni murto-osa käytetystä ajasta. Näen itse tämän syyksi miksi työn tekoprosessi kesti näinkin kauan, koska kaikkien applikaatioiden opettelun lisäksi, minun tuli opetella vielä tuottamaan sujuvaa koodia. Tästä syystä aluksi erehdyin keskittymään vääriin asioihin ja tekemään paljon turhaa työtä ja tuotin käyttökelvotonta koodia satoja rivejä. Lisäksi analysoitavan datan vaihtuminen työn alkumetreillä hidasti työn etenemistä.

Analysoitava data oli pääasiallisesti ihan hyvää, mutta siinä oli myös paljon virheitä. Vuodenajan vaihtelut vaikeuttavat analysointia ja toistaiseksi näyttää siltä, ettei sademäärän mittari toimi oikein talvella. Lisäksi datassa oli useita kohtia, jotka olivat selvästi virheellisiä. Joissain kohdissa arvot saattoivat nousta päiväksi alle yhdestä tuhansiin. Tämän lisäksi joissain mittauksissa oli selkeitä taukoja, joissa arvoja ei oltu saatu. Myöskin näytteenottovälin ollessa ainoastaan yksi tunti, on parempi kutsua toteutusta lähes reaaliaikaiseksi.

Kehitys koodin tuottamisessa työn aikana on kuitenkin ollut valtava verrattuna olemattomiin lähtökohtiin ja nyt huomaan jo valtavaa kehitystä tämän saralta. Työ itsessään oli kohtuullisen työläs ja haastava, mutta lopputulema on toimiva ja olen suhteellisen tyytyväinen saatuihin tuloksiin.

Lähteet

A Brief Introduction to Apache Cassandra. N.d. DataStax-sivusto Viitattu 20.02.2018. https://academy.datastax.com/resources/brief-introduction-apache-cassandra

Apache Spark - Core Programming. N.d. Tutorialpoint-sivusto Viitattu 24.10.2017. https://www.tutorialspoint.com/apache_spark/apache_spark_core_programming.htm

Apache Spark – RDD. N.d Tutorialpoint-sivusto Viitattu 8.11.2017. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

Chowdhury, M., Das, T., Dave, A., Franklin, M., Ma, J. McCauley, M. Shenker, S. Stoica, I. Zaharia, M. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Viitattu 23.11.2017. https://cs.stanford.edu/~matei/papers/2012/nsdi_spark.pdf

Class SparkContext. N.d. Apache Spark objektiivien-sivusto. Viitattu 19.02.2018. https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/SparkContext.html

Cluster Mode Overview. N.d. Apache Spark-sivusto. Viitattu 23.10.2017. https://spark.apache.org/docs/latest/cluster-overview.html

HDFS. N.d Viitattu 8.11.2017. https://www.ibm.com/analytics/us/en/technology/hadoop/hdfs/

Introduction. N.d. Apache Kafka-sivusto. Viitattu 19.02.2018. https://kafka.apache.org/intro

Laskowski, J. N.d.a. ActiveJob. Viitattu 5.3.2018. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dagscheduler-jobs.html

Laskowski, J. N.d.b. Apache Spark. Viitattu 20.10.2017. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-overview.html

Laskowski, J. N.d.c. Stage — Physical Unit Of Execution. Viitattu 5.3.2018. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dagscheduler-jobs.html

Laskowski, J. N.d.d. Tasks. Viitattu 5.3.2018. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-taskscheduler-tasks.html

Lynn, D. 2016. Apache Spark Cluster Managers: YARN, Mesos, or Standalone?. Viitattu 23.10.2017. http://www.agildata.com/apache-spark-cluster-managers-yarn-mesos-or-standalone/

Phatak, M. 2016. Introduction to Spark 2.0 - Part 1 : Spark Session API. Viitattu 19.02.2018. http://blog.madhukaraphatak.com/introduction-to-spark-two-part-1/

Pyspark Cassandra. 2018. PySpark Cassandra-paketin kuvaus. Viitattu 20.2.2018. https://github.com/anguenot/pyspark-cassandra

Pyspark Package. n.d. Viitattu 13.03.2018.
http://spark.apache.org/docs/2.2.0/api/python/pyspark.html

Pyspark Streaming Module. N.d. Apache Spark objektiivien-sivusto. Viitattu 19.02.2018.
https://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.StreamingContext

Rouse, M. 2016. Data analytics (DA). Viitattu 19.02.2018.
http://searchdatamanagement.techtarget.com/definition/data-analytics

Rouse, M. 2017. Database (DB). Viitattu 20.02.2018.
http://searchsqlserver.techtarget.com/definition/database

Spark SQL, DataFrames and Datasets Guide. N.d. Apache Spark-sivusto. Viitattu 8.11.2017. https://spark.apache.org/docs/latest/sql-programming-guide.html

Spark Streaming + Kafka Integration Guide. N.d. Apache Spark-sivusto. Viitattu 20.02.2018. https://spark.apache.org/docs/2.2.0/streaming-kafka-integration.html

Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher). N.d. Apache Spark-sivusto. Viitattu 20.02.2018.
https://spark.apache.org/docs/2.2.0/streaming-kafka-0-8-integration.html

Spark Streaming Programming Guide. N.d. Apache Spark-sivusto. Viitattu 19.02.2017.
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Structured Query Language (SQL). N.d. Techopedia-sivusto. Viitattu 13.03.2018.
https://www.techopedia.com/definition/1245/structured-query-language-sql

Liitteet

Liite 1. db_sameus.py

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-


import MySQLdb

import csv

import os


dbQuery_sameus = 'SELECT dataPoints.deviceName, dataPoints.name, from_unixtime(floor(pointVa-
lues.ts/1000), "%Y/%m/%d %H:%i:%s") as time, pointValues.pointValue, pointValues.dataType FROM
pointValues, dataPoints WHERE pointValues.dataPointId = 535 AND dataPoints.id = 535 ORDER BY
pointValues.id DESC LIMIT 1;'


db_host = os.environ.get('DB_HOST')

db_user = os.environ.get('DB_USER')

db_pass = os.environ.get('DB_PASS')

db_name = os.environ.get('DB_NAME')


connection=MySQLdb.connect(host=db_host, user=db_user, passwd=db_pass, db=db_name)

kursori=connection.cursor()


kursori.execute(dbQuery_sameus)

tulos=kursori.fetchall()


file = csv.writer(open("/home/h9553/temp.csv","a"))

for i in tulos:

    file.writerow(i)


connection.close()
```

Liite 2. db_sade.py

```
#!/usr/bin/python3

# -*- coding: utf-8 -*-

import MySQLdb

import sys

import csv

import os

dbQuery_sade ='SELECT dataPoints.deviceName, dataPoints.name, from_unixtime(floor(pointVa-
lues.ts/1000), "%Y/%m/%d %H:%i:%s") as time, pointValues.pointValue, pointValues.dataType FROM
pointValues, dataPoints WHERE pointValues.dataPointId = 465 AND dataPoints.id = 465 ORDER BY
pointValues.id DESC LIMIT 1;'

db_host = os.environ.get('DB_HOST')

db_user = os.environ.get('DB_USER')

db_pass = os.environ.get('DB_PASS')

db_name = os.environ.get('DB_NAME')

connection=MySQLdb.connect(host=db_host, user=db_user, passwd=db_pass, db=db_name)

kursori=connection.cursor()

kursori.execute(dbQuery_sade)

tulos=kursori.fetchall()

file = csv.writer(open("/home/h9553/temp.csv","a"))

for i in tulos:

   file.writerow(i)

connection.close()
```

Liite 3. ilmatiede.py

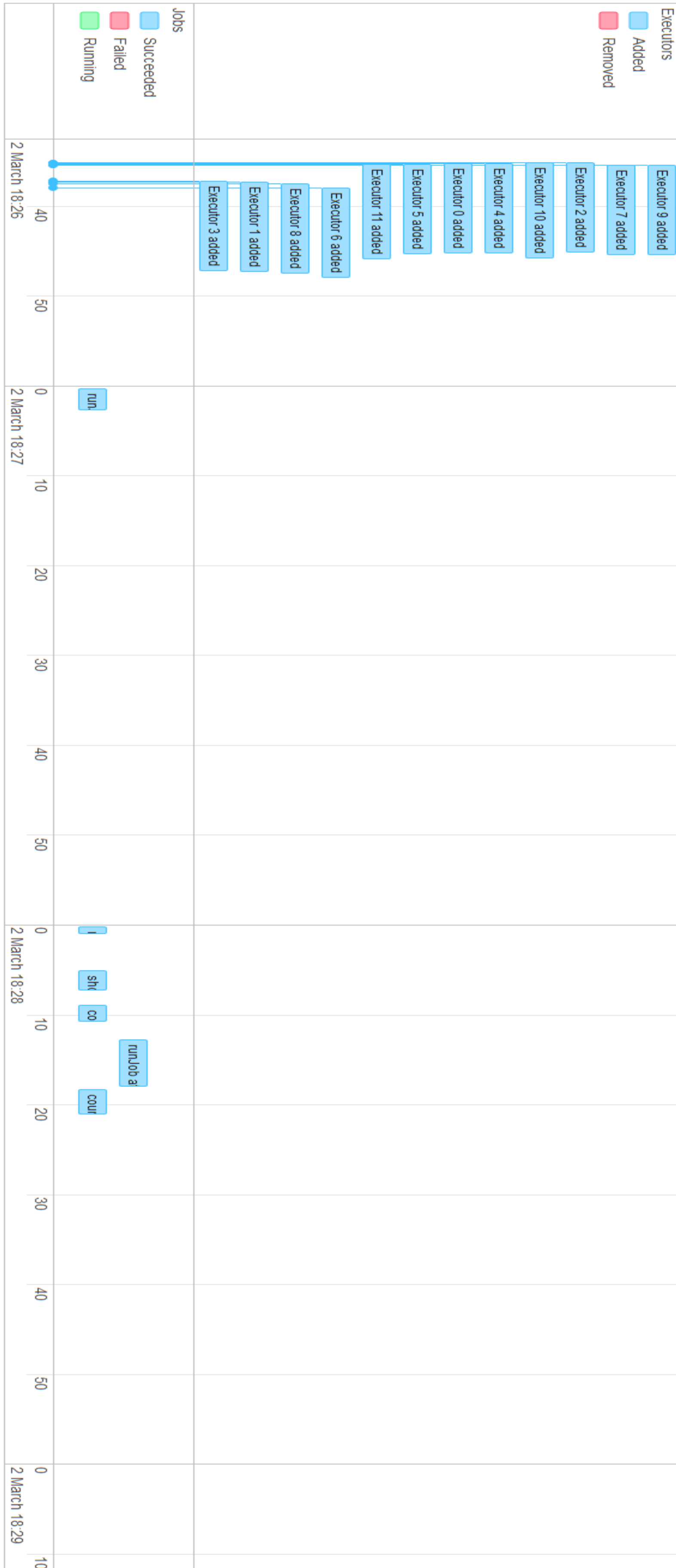
```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import requests
import xml.etree.ElementTree as ET
import csv
import os

file = open("/home/h9553/temp.csv", "a")
api = os.environ.get('IT_API')

r = requests.get('http://data.fmi.fi/fmi-apikey/' + api + '/wfs?request=getFeature&sto-
redquery_id=fmi::forecast::hirlam::surface::point::simple&parameters=Precipitati-
onAmount&place=saarijärvi')

root = ET.fromstring(r.content)

t1 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}ParameterName')]
t2 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}Time')]
t3 = [i.text for i in root.iter('{http://xml.fmi.fi/schema/wfs/2.0}ParameterValue')]

list = zip(t1,t2,t3)

for i in list:
    i = ("Ilmatieteenlaitos," + ",".join(map(str, i)) + ',,')
    i = i.replace('-', '/').replace('T', ' ').replace('Z', '')
    file.write(i + '\n')

file.close()
```

Liite 4. stable.py

```
# -*- coding: utf-8 -*-

import sys
from pyspark_cassandra import CassandraSparkContext
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import col

def save(time, split):
    print("========= %s =========" % str(time))
    try:
        rdd = split.map(lambda a: Row(info=a[3], aika=a[4], arvo=a[5]))
        df = spark.createDataFrame(rdd)
        df.show()

        sam = df.filter(col('info').like("%Sameus%")).count()
        if sam > 0:
            dfsam = df.filter(col('info').like("%Sameus%"))
            dfsam.write\
                .format("org.apache.spark.sql.cassandra")\
                .mode('append')\
                .options(table="sameus", keyspace="analytiikka")\
                .save()

        sade = df.filter(col('info').like("%Sadem%")).count()
        if sade > 0:
            dsade = df.filter(col('info').like("%Sadem%"))
            dsade.write\
                .format("org.apache.spark.sql.cassandra")\
                .mode('append')\
                .options(table="sade", keyspace="analytiikka")\
```

```
                .save()

        ennuste = df.filter(col('info').like("%PrecipitationAmount%")).count()
        if ennuste > 0:
            dfen = df.filter(col('info').like("%PrecipitationAmount%"))
            dfen.write\
                .format("org.apache.spark.sql.cassandra")\
                .mode('append')\
                .options(table="ennuste", keyspace="analytiikka")\
                .save()



    except:
        pass
if __name__ == '__main__':

    conf = SparkConf()\
        .setAppName("stable")\

    sc = CassandraSparkContext(conf=conf)
    spark = SparkSession(sc)
    ssc = StreamingContext(sc, 10)

    kohde, aihe = sys.argv[1:]
    stream = KafkaUtils.createDirectStream(ssc, [aihe],{'metadata.broker.list': kohde})
    lines = stream.map(lambda x: x[1])
    split = lines.map(lambda x: x.split(','))
    split.pprint()
    split.foreachRDD(save)

    ssc.start()
    ssc.awaitTermination()
```

Liite 5. stable.py 2 minuutin aikajana



Liite 6. stable.py workereiden resurssit

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	192.168.51.181:46766	Active	0	0.0 B / 434 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
0	192.168.51.177:38009	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	1	1	2 s (84 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
1	192.168.51.181:41941	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	6	6	3 s (86 ms)	0.0 B	115 B	0.0 B	stdout stderr	Thread Dump
2	192.168.51.183:41049	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	12	12	9 s (0.2 s)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	192.168.51.181:41335	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	4	4	3 s (0.1 s)	0.0 B	56 B	56 B	stdout stderr	Thread Dump
4	192.168.51.183:38972	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	5	5	6 s (0.2 s)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
5	192.168.51.183:42427	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	3	3	2 s (86 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
6	192.168.51.181:45481	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	11	11	5 s (0.1 s)	0.0 B	56 B	230 B	stdout stderr	Thread Dump
7	192.168.51.177:43513	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	7	7	4 s (91 ms)	0.0 B	0.0 B	56 B	stdout stderr	Thread Dump
8	192.168.51.181:46698	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	6	6	4 s (0.1 s)	0.0 B	59 B	56 B	stdout stderr	Thread Dump
9	192.168.51.177:35585	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	8	8	4 s (95 ms)	0.0 B	112 B	59 B	stdout stderr	Thread Dump
10	192.168.51.177:33922	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	9	9	4 s (79 ms)	0.0 B	0.0 B	56 B	stdout stderr	Thread Dump
11	192.168.51.183:37360	Active	0	0.0 B / 434 MB	0.0 B	1	0	0	6	6	3 s (97 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Liite 7. korrelaatio.py

```
import numpy as np
from pyspark_cassandra import CassandraSparkContext
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import col

if __name__ == '__main__':
    conf = SparkConf()\
        .setAppName("korrelaatio")\
    sc = CassandraSparkContext(conf=conf)
    spark = SparkSession(sc)

    num = int(0)
    dict = {}

    df = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="saa_kesa", keyspace="legacy")\
        .load()
    df = df.orderBy(df.aika.asc())

    korrelaatio_kk = df.stat.corr('sameus','sade')
    dict.update({int(0) : korrelaatio_kk})
    pvm = np.arange(1, 31)
    pvm = sc.parallelize(pvm).collect()
    for paiva in range(1, 30):
        if num < 10:
            korrelaatio_paiva = df.filter(col('aika').like('2017/06/0' + str(pvm[num]) + '%'))
            cor = korrelaatio_paiva.stat.corr('sameus','sade')
            dict.update({int(pvm[num]) : cor})
            num = num + 1
        if num > 9:
            korrelaatio_paiva = df.filter(col('aika').like('2017/06/' + str(pvm[num]) + '%'))
```

```
        cor = korrelaatio_paiva.stat.corr('sameus','sade')
        dict.update({int(pvm[num]) : cor})
        num = num + 1

dict_map = [(k,)+(v,) for k,v in dict.items()]
df_cor = spark.createDataFrame(dict_map, ['pvm','korrelaatio']).na.fill(0)
df_cor.write\
    .format("org.apache.spark.sql.cassandra")\
    .mode('append')\
    .options(table="korrelaatio", keyspace="legacy")\
    .save()
```

Liite 8. Korrelaation tulokset

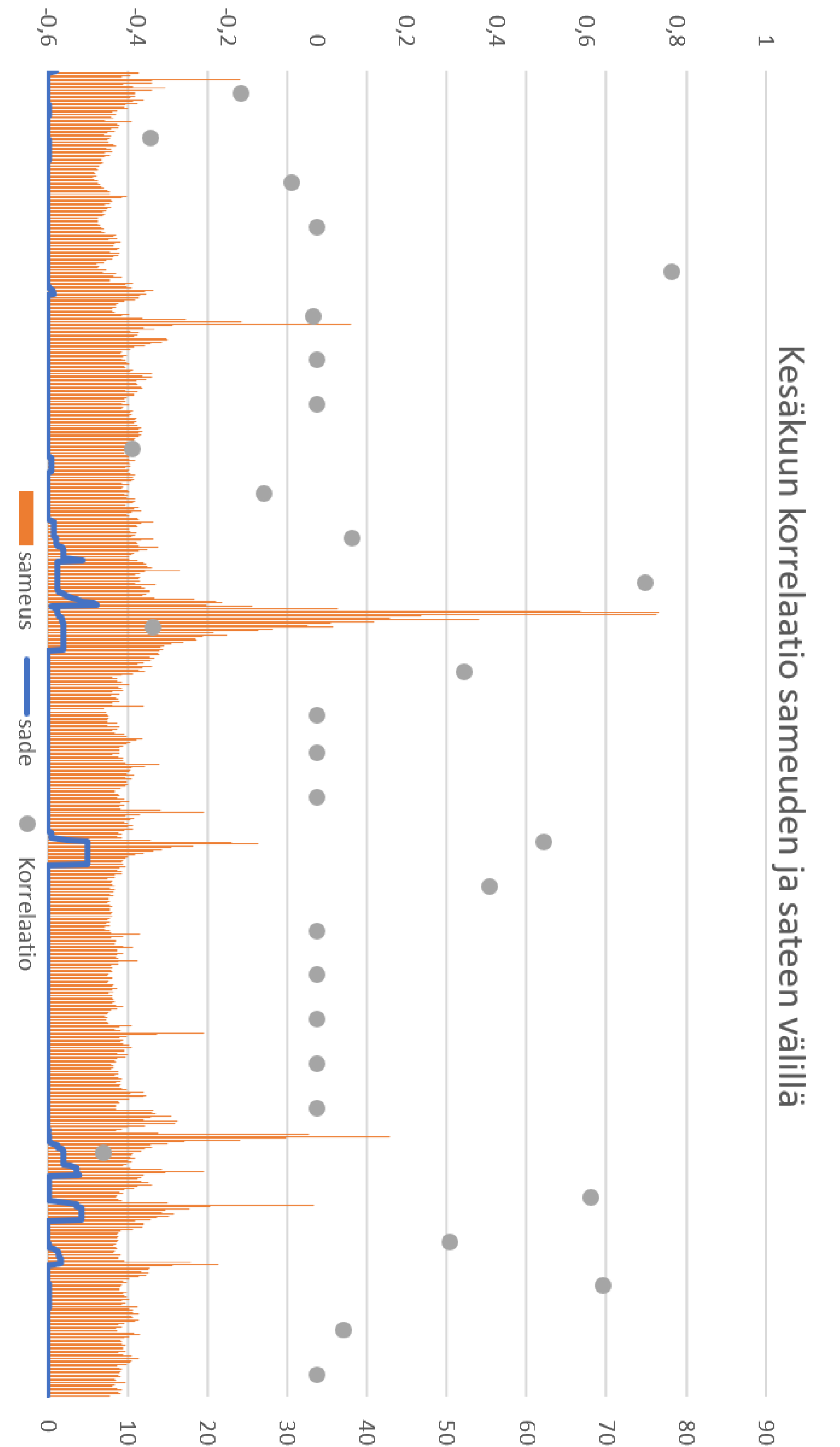
```
cqlsh:legacy> SELECT * FR
OM korrelaatio ;

 pvm | korrelaatio
-----+-------------
  23 |           0
   5 |    0.788854
  28 |    0.637321
  10 |   -0.117222
  16 |           0
  13 |   -0.366133
  30 |           0
  11 |    0.077505
   1 |   -0.169704
  19 |    0.384503
   8 |           0
   0 |    0.325417
   2 |   -0.372198
   4 |           0
  18 |    0.505957
  15 |           0
  22 |           0
  27 |    0.294218
  20 |           0
   7 |           0
   6 |   -0.007655
  29 |    0.059481
   9 |   -0.410925
  14 |    0.328908
  26 |    0.610831
  21 |           0
  17 |           0
  24 |           0
  25 |   -0.475361
  12 |    0.730614
   3 |   -0.055532

(31 rows)
```

Liite 9. ennuste.py

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as sfm
from pyspark_cassandra import CassandraSparkContext
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import col

if __name__ == '__main__':

    conf = SparkConf()\
        .setAppName("ennuste")\
    sc = CassandraSparkContext(conf=conf)
    spark = SparkSession(sc)

    df_sade = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="sade", keyspace="analytiikka")\
        .load()

    df_sameus = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="sameus", keyspace="analytiikka")\
        .load()

    df_ennuste = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="ennuste", keyspace="analytiikka")\
        .load()

    df = pd.DataFrame(columns = ['sameus', 'sade', 'num'])

    df_sade = df_sade.orderBy(df_sade.aika.asc())
```

```
df_sameus = df_sameus.orderBy(df_sameus.aika.asc())
df_ennuste = df_ennuste.orderBy(df_ennuste.aika.asc())

ennuste_count = df_ennuste.count()
sameus_count = df_sameus.count()
num = np.arange(0, sameus_count)
ennuste_num = 0

pdf_sade = df_sade.toPandas()
pdf_sameus = df_sameus.toPandas()
pdf_ennuste = df_ennuste.toPandas()

pdf_sameus['sameus'] = pdf_sameus['arvo']
pdf = pdf_sameus.drop(['info', 'arvo'], axis=1)
pdf['sade'] = pdf_sade['arvo']
pdf['num'] = num

for i in range(1, ennuste_count):
    data = {"sameus": pdf['sameus'], "sade": pdf['sade'], "num": pdf['num']}
    tulos = sfm.ols(formula='sameus ~ sade + num', data=data).fit()
    ennuste = tulos.predict([1, pdf_ennuste['arvo'][ennuste_num], sameus_count], trans-
form=False)
    pdf = pdf.append({"sameus" : ennuste[0], "sade": pdf_ennuste['arvo'][ennuste_num], "num":
int(sameus_count)}, ignore_index=True)
    sameus_count = sameus_count + 1
    ennuste_num = ennuste_num + 1
    print(ennuste)
```

Liite 10. Korrelaatio kesäkuu 2017



Liite 11. ennuste2.py

```
import numpy as np
import pandas as pd

import statsmodels.formula.api as sfm
from pyspark_cassandra import CassandraSparkContext
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import col


if __name__ == '__main__':
    conf = SparkConf()\
        .setAppName("ennuste")\
    sc = CassandraSparkContext(conf=conf)
    spark = SparkSession(sc)

    df_saa = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="saa_kesa", keyspace="legacy")\
        .load()

    df_ennuste = spark.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table="kesa_ennuste", keyspace="legacy")\
        .load()

    df = pd.DataFrame(columns = ['sameus', 'sade', 'num'])
    df_saa = df_saa.orderBy(df_saa.aika.asc())
    df_ennuste = df_ennuste.orderBy(df_ennuste.aika.asc())
    ennuste_count = df_ennuste.count()
    saa_count = df_saa.count()
    num = np.arange(0, saa_count)
    ennuste_num = 0
```

```
pdf_saa = df_saa.toPandas()

pdf_ennuste = df_ennuste.toPandas()

pdf_saa['num'] = num

pdf = pdf_saa


for i in range(1, ennuste_count):

    data = {"sameus": pdf['sameus'], "sade": pdf['sade'], "num": pdf['num']}

    tulos = sfm.ols(formula='sameus ~ sade + num', data=data).fit()

    ennuste = tulos.predict([1, pdf_ennuste['arvo'][ennuste_num], saa_count], transform=False)

    pdf = pdf.append({"sameus" : ennuste[0], "sade": pdf_ennuste['arvo'][ennuste_num], "num":
int(saa_count)}, ignore_index=True)

    saa_count = saa_count + 1

    ennuste_num = ennuste_num + 1

    print(ennuste)

print(tulos.summary())
```

Liite 12. Mitatut sameuden muutokset verrattaen ennustettuun

Aika	Sade	Laskettu sameus	Mitattu sameus
1.7.2017 0:00	0	10,0253	7,78
1.7.2017 1:00	0	10,0247	7,96
1.7.2017 2:00	0	10,0242	8,42
1.7.2017 3:00	0	10,0237	8,54
1.7.2017 4:00	0	10,0232	8,76
1.7.2017 5:00	0	10,0227	8,82
1.7.2017 6:00	0	10,0222	7,81
1.7.2017 7:00	0	10,0217	8,06
1.7.2017 8:00	0	10,0211	7,57
1.7.2017 9:00	0	10,0206	7,47
1.7.2017 10:00	0	10,0201	7,78
1.7.2017 11:00	0	10,0196	7,47
1.7.2017 12:00	0	10,0191	7,78
1.7.2017 13:00	0	10,0186	7,41
1.7.2017 14:00	0	10,0180	7,11
1.7.2017 15:00	0	10,0175	7,38
1.7.2017 16:00	0	10,0170	7,72
1.7.2017 17:00	0	10,0165	7,72
1.7.2017 18:00	0	10,0160	7,96
1.7.2017 19:00	0	10,0155	8,02
1.7.2017 20:00	0	10,0150	8,61
1.7.2017 21:00	0	10,0144	8,42
1.7.2017 22:00	0	10,0139	9,13
1.7.2017 23:00	0	10,0134	8,48
2.7.2017 0:00	0	10,0129	8,39
2.7.2017 1:00	0	10,0124	8,48
2.7.2017 2:00	0	10,0119	9,06
2.7.2017 3:00	0	10,0113	12,95
2.7.2017 4:00	0	10,0108	12,58
2.7.2017 5:00	0	10,0103	11,39
2.7.2017 6:00	0	10,0098	9,64
2.7.2017 7:00	0	10,0093	9,19
2.7.2017 8:00	0	10,0088	8,97
2.7.2017 9:00	0	10,0083	8,24
2.7.2017 10:00	0	10,0077	8,64
2.7.2017 11:00	0	10,0072	8,85
2.7.2017 12:00	0	10,0067	8,91
2.7.2017 13:00	0	10,0062	8,64
2.7.2017 14:00	0	10,0057	8,73
2.7.2017 15:00	0	10,0052	8,24
2.7.2017 16:00	0	10,0046	8,3
2.7.2017 17:00	0	10,0041	8,51
2.7.2017 18:00	0	10,0036	8,79
2.7.2017 19:00	0	10,0031	8,67

2.7.2017 20:00	0	10,0026	9,4
2.7.2017 21:00	0	10,0021	9,49
2.7.2017 22:00	0	10,0016	9,31
2.7.2017 23:00	0	10,0010	9,16
3.7.2017 0:00	0	10,0005	8,54
3.7.2017 1:00	0	10,0000	8,97
3.7.2017 2:00	0	9,9995	9,22
3.7.2017 3:00	0	9,9990	9,71
3.7.2017 4:00	0	9,9985	9,28
3.7.2017 5:00	0	9,9979	10,44
3.7.2017 6:00	0	9,9974	10,84
3.7.2017 7:00	0	9,9969	10,87
3.7.2017 8:00	0	9,9964	9,8
3.7.2017 9:00	0	9,9959	9,55
3.7.2017 10:00	0	9,9954	9,34
3.7.2017 11:00	0	9,9949	9,22
3.7.2017 12:00	0	9,9943	9,19
3.7.2017 13:00	0	9,9938	8,27
3.7.2017 14:00	0	9,9933	8,42
3.7.2017 15:00	0	9,9928	8,27
3.7.2017 16:00	0	9,9923	8,24
3.7.2017 17:00	0	9,9918	8,48
3.7.2017 18:00	0	9,9912	8,21
3.7.2017 19:00	0	9,9907	8,15
3.7.2017 20:00	0	9,9902	8,27
3.7.2017 21:00	0	9,9897	7,84
3.7.2017 22:00	0	9,9892	8,3
3.7.2017 23:00	0	9,9887	8,09
4.7.2017 0:00	0,2	10,3859	8,45
4.7.2017 1:00	0,4	10,7832	8,88
4.7.2017 2:00	1,2	12,3738	8,91
4.7.2017 3:00	2	13,9645	10,44
4.7.2017 4:00	4,8	19,5329	10,35
4.7.2017 5:00	4,8	19,5324	11,54
4.7.2017 6:00	4,8	19,5319	15,05
4.7.2017 7:00	4,8	19,5314	32,84
4.7.2017 8:00	4,8	19,5309	32,45
4.7.2017 9:00	4,8	19,5303	19,36
4.7.2017 10:00	4,8	19,5298	15,24
4.7.2017 11:00	4,8	19,5293	13,5
4.7.2017 12:00	4,8	19,5288	12,88
4.7.2017 13:00	4,8	19,5283	12,3
4.7.2017 14:00	4,8	19,5278	12,15
4.7.2017 15:00	4,8	19,5272	12,67
4.7.2017 16:00	4,8	19,5267	12,88
4.7.2017 17:00	4,8	19,5262	13,5
4.7.2017 18:00	4,8	19,5257	13,37
4.7.2017 19:00	4,8	19,5252	12,98

4.7.2017 20:00	10,6	31,0604	11,75
4.7.2017 21:00	10,6	31,0599	11,69
4.7.2017 22:00	10,6	31,0594	10,75
4.7.2017 23:00	10,6	31,0589	10,87
5.7.2017 0:00	10,6	31,0584	11,6
5.7.2017 1:00	0	9,9753	11,57
5.7.2017 2:00	0	9,9748	11,36
5.7.2017 3:00	0	9,9742	11,75
5.7.2017 4:00	0	9,9737	11,66
5.7.2017 5:00	0	9,9732	11,57
5.7.2017 6:00	0	9,9727	11,17
5.7.2017 7:00	0	9,9722	10,44
5.7.2017 8:00	0	9,9717	10,44
5.7.2017 9:00	0	9,9711	9,77
5.7.2017 10:00	0	9,9706	9,16
5.7.2017 11:00	0	9,9701	9,09
5.7.2017 12:00	0	9,9696	8,64
5.7.2017 13:00	0	9,9691	8,76
5.7.2017 14:00	0	9,9686	8,76
5.7.2017 15:00	0	9,9681	8,54
5.7.2017 16:00	0	9,9675	12,67
5.7.2017 17:00	0	9,9670	8,79
5.7.2017 18:00	0	9,9665	8,33
5.7.2017 19:00	0	9,9660	8,54
5.7.2017 20:00	0	9,9655	8,76
5.7.2017 21:00	0	9,9650	8,33
5.7.2017 22:00	0	9,9645	8,85
5.7.2017 23:00	0	9,9639	8,54
6.7.2017 0:00	0	9,9634	8,33
6.7.2017 1:00	0	9,9629	8,48
6.7.2017 2:00	0	9,9624	8,79
6.7.2017 3:00	0	9,9619	9,09
6.7.2017 4:00	0	9,9614	9,13
6.7.2017 5:00	0,2	10,3586	9,61
6.7.2017 6:00	0,4	10,7559	9,37
6.7.2017 7:00	0,4	10,7554	8,7