Lauri Uotila

Reaaliaikainen lokijärjestelmä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

21.9.2022

Tiivistelmä

Tekijä:	Lauri Uotila
Otsikko:	Reaaliaikainen lokijärjestelmä
Sivumäärä:	25 sivua
Aika:	21.9.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander
	Full-Stack Developer Lauri Raivio
	Full-Stack Developer Eero Vehmanen

Insinöörityön aiheena oli suunnitella ja toteuttaa lokijärjestelmä, joka reaaliajassa kerää tietoa ja tilastoja eri palvelimilla ja ympäristöissä ajettavista ohjelmakoodeista. Järjestelmän toteutuksen dokumennoinnin ohella oli tarkoitus vertailla projektin eri toteutustapoja sekä punnita vaihtoehtoisten teknologioiden hyötyjä ja haittoja.

Tehtävän suunnittelun tuloksena toteutuksessa päädyttiin käyttämään Googlen pilvipalveluja sekä data-analytiikka maailman uutta ja lupaavaa tiedon prosessoinnin työkalua, Apache Beam-ohjelmointimallia ja sen ohjelmakirjastoa. Työn tuloksena saatiin järjestelmä, joka toimii reaaliaikaisesti toimivana dataputkena, joka kerää ja eristää hyödyllistä tietoa lähtöympäristöjen ohjelmakoodien lokeista ja tallentaa ne lopuksi tietovarastoon tulkittavaksi. Järjestelmän kyky käsitellä lokeja lähellä niiden luontihetkeä teki ohjelmakoodien ajojen tarkasta monitoroinnista mahdollista. Monitoroinnin lisäksi järjestelmälle kehitettiin toiminnallisuus luoda koosteita yksittäisistä lokeista, joka tuli avuksi lokeista kerätyn tiedon tulkintaan.

Avainsanat: data-analytiikka, pilviympäristö, reaaliaikainen tiedon prosessointi

Abstract

Author:	Lauri Uotila
Title:	Real-time Logging System
Number of Pages:	25 pages
Date:	21 September 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Lauri Raivio, Full-Stack Developer Eero Vehmanen, Full-Stack Developer

---

The aim of the study was to design and implement a system that gathers run-time logs from scripts run on different server environments loading and transforming data. The system was also supposed to parse this log data and extract useful information and statistics from it. The goal was also to document and go over potentially useful technologies and different implementation methods that could be utilised in creating the system, while weighing their pros and cons.

The implementation ended up using Google cloud services and the Apache Beam Model for data processing and its software libraries. The resulting system functions as a real-time data pipeline that gathers and extracts useful information from the logs generated by the monitored server environments, finally saving this useful information to a data warehouse where it could be analysed. The ability of the system to process data close to the creation time of the logs made it possible to accurately monitor the current state of the run scripts. In addition to monitoring capabilities, the system was given functionality to aggregate information from the logs to help in the interpretation and understanding of the gathered data.

Keywords: data analytics, cloud services, real-time data processing

Sisällys

Lyhenteet

REST: *Representational state transfer.* Arkkitehtuurityyli ohjelmointirajapintojen toteuttamiseen.

BI-työkalu: *Business intelligence -työkalu.* Ohjelmisto, jolla voi tehdä tiedon latausta, muokkausta sekä graafista esittelyä. Yleensä käytetään yrityksissä niiden toimintaan liittyvän tiedon tulkinnan apuna.

ELT: *Extract, Load, Transform.* Tiedon prosessoinnissa käytetty toimintaperiaate, jossa tieto ensin louhitaan lähteestä, ladataan tietovarastoon ja lopuksi muutetaan haluttuun muotoon.

GCP: *Google Cloud Platform.* Googlen pilviympäristö, joka tarjoaa monia eri palveluilta ja liittyy prosessointiin ja tiedon säilytykseen.

1 Johdanto

Nykypäivänä isojen tietomäärien keräys ja analytiikka ovat koko ajan tärkeämmissä osissa yritysten päätöksentekoa. Mahdollisia tietolähteitä on REST-rajapintojen, uusien tietokantojen ja datan prosessoinnin kehityksen kautta enemmän kuin koskaan. Moni liiketoiminnan tehostus perustuukin tiedosta kaivettuihin oivalluksiin. Tiedon keräyksen ja analysoinnin alalla on myös tämän insinöörityön toimeksiantaja, QuickBI Oy, vuonna 2018 perustettu helsinkiläinen startup-yritys. QuickBI myy dataintegraatioita, tiedonvarastointia sekä näiden avulla tehtävää raportointia Business Intelligence (BI) -työkalujen avulla.

Tietoa ladattaessa ja tallennettaessa useiden palvelinten avulla moniin eri tietovarastoihin myös ohjelmakoodien ajoista syntyvistä lokitiedoista alkaa laajassa mittakaavassa saada hyödyllistä tietoa yrityksen teknisen toiminnan tilasta. Insinöörityön aihe, reaaliaikainen lokijärjestelmä, valittiin, koska QuickBI:llä oli tarvetta kerätä ohjelmakoodeista syntyneitä lokiviestejä ja tehdä niistä analyysiä. Luotava lokijärjestelmä toimisi omana jatkuvasti ajassa pysyvänä tietolähteenä, joka ideaalisesti vastaisi kysymyksiin kuten kuinka useasti tietyt ohjelmakoodit epäonnistuvat, onko niillä jotain yhteisiä tekijöitä tai kuinka ohjelmakoodin ajon kesto muuttuu ajan kuluessa.

Toteutettavan lokijärjestelmän ohella oli myös tarkoitus tarkastella, mitä moderneja datamaailman työkaluja tehtävässä oikeastaan kannattaisi käyttää ja pohtia niiden hyötyjä ja-tai haittoja toteutuksen kannalta. Tarkoituksena siis on lokijärjestelmän lisäksi pohdinnan kautta päätyä parempaan yleiskuvaan datamaailman työkaluista ja toimintatavoista ja siitä, miten niitä kannattaisi hyödyntää muissa tulevaisuuden projekteissa.

2 Vaatimukset

Toteutettava lokijärjestelmä tulisi käyttöön yritykselle, jonka toimintaan kuuluu suurien tietomäärien liikuttelu ja lataus eri tietokantojen ja api-rajapintojen kautta. Tärkeää järjestelmän hyödyllisyyden kannalta on, että se kykenisi käsittelemään monista eri dataa lataavista palvelimista tulevia lokiviestejä ja eriteltyä viesteistä hyödyllistä tietoa. Tällaista tietoa voisi olla esimerkiksi se, onko jokin palvelimelle ajastettu ohjelmakoodi ajettu onnistuneesti tai kuinka paljon tietoa monitoroitava ohjelmakoodi on eristänyt REST-rajapintaisesta tietolähteestä. Haluttiin myös, että järjestelmä voi tehdä tilastollisia laskelmia pidemmältäkin aikaväliltä. Näin voitaisiin tarkistella ei pelkästään yksittäisten ohjelmakoodien ajoja, mutta myös saada yleinen kuva niiden suoritusajoista ja keskimääräisistä onnistumisprosenteista. Käsiteltävän tiedon määrä saattaisi myös vaihdella laajasti, joten järjestelmän tulisi kyetä suoriutumaan myös samaan aikaan tulevista suurista tietomääristä ilman, että siitä aiheutuisi latenssia kerätyssä lopullisessa datassa. Lokiviesteistä kerätty hyödyllinen tieto tallennettaisiin tietokantaan, josta sitä voitaisiin tarkastella esimerkiksi BI-työkalujen avulla.

Yksi pohdintaa herättäneistä kysymyksistä oli, kuinka suuri viive lokin luontiajan ja sen käsittelyajan välillä olisi hyväksyttävää. Jos aikaväli ei ole tärkeä ja hyväksyttävä viive, olisi mitattavissa tunneissa. Tällöin mahdollinen vaihtoehto olisi kerryttää lokit tiedostojärjestelmään. Kerrytyksen jälkeen tiedot voitaisiin käsitellä kooltaan ennestään tiedettyinä erinä, eli käyttäen eräprosessointia. Dataanalytiikan maailmassa eräprosessoinnin lisäksi puhutaan tiedon suoratoistosta. Toisin kuin eräprosessointi, tiedon suoratoistossa ei tiedetä, kuinka suuri käsiteltävä datasetti on tai, onko datasetti rajaton (Sherif Sakr, Albert Y. Zomaya 2019, 611, 648). Datan käsittely erinä on yksinkertaista ja käytännöllistä, mutta sen käyttö aiheuttaisi latenssia lopullisen datasetin ilmestymisessä tarkasteltavaksi. Esimerkiksi jos tiedostojärjestelmään kerrytettyjä lokeja käytäisiin läpi 10 minuutin välein ja haluttaisiin tietää, onko tietty ohjelmakoodin ajo onnistunut, saataisiin tämä tieto huonoimmassa tapauksessa 10 minuuttia ajon onnistumista viestivän lokitapahtuman luonnin jälkeen. Toisaalta jos tieto ohjelmakoodien

tilasta käsitellään suoratoistona, ja lokiviestit käsitellään aina lähellä niiden luontihetkeä, voitaisiin lokijärjestelmän tietokantaa käyttää reaaliaikaisuuden vuoksi ohjelmakoodien ajojen monitorointiin niin, että tietokannan tilaa voidaan aina pitää viiveettömänä ja ajantasaisena. Ohjelmakoodien ajoista luotavien tilastolaskelmien kannalta reaaliaikaisuudesta ei olisi kuitenkaan juurikaan apua - niiden koko hyöty perustuu pidemmän ajan tiedon kerrytyksestä saatuihin oivalluksiin. Reaaliaikaisten tiedon prosessoinnin työkalujen avulla olisi kuitenkin mahdollista toteuttaa molemmat, ajantasalla pysyvän tiedon raportointi sekä pidemmän ajan tilastointi. Tämän vuoksi järjestelmää oli luontevaa lähteä kehittämään tiedon suoratoistoa käyttäen.

3 Ohjelmistoarkkitehtuuri ja käytetyt ohjelmistoteknologiat

Ohjelmistoteknologioiden valinnassa kriteereinä oli, että käytettäisiin ohjelmointikieltä, joka on yleisesti hyvin tunnettu ja käytetty. Muut käytettyyn teknologiaan liittyvät päätökset tehtäisiin sen ympärille. Käytetylle ohjelmointikielelle tuli myös olla tehtävään soveltuvia, hyvin dokumentoituja vapaan lähdekoodin ohjelmakirjastoja, joita hyödynnetään järjestelmän toteutuksessa. Näiden kriteerien perusteella projektin pääohjelmointikieleksi valittiin tulkattava, korkeantason ohjelmointikieli Python, jolla tuli toteuttaa kaikki mahdolliset toiminnallisuudet. Pelkkä ohjelmointikielen valinta ei kuitenkaan ollut projektin kannalta riittävä, sillä oikein toimiva lokijärjestelmä tulisi sisältämään eri osia, jotka eivät ole toteutettavaa ohjelmakoodia, vaan valmiita ohjelmistotyökaluja. Ne on erityisesti suunniteltu niille rajattuun tehtävään. Valmiiden työkalujen avulla projektin osa-alueita voitiin jakaa toteutettavaksi tehtäviin ominaisille työkaluille, jotka yksinkertaistavat lopullista ohjelmistoarkkitehtuuria. Erityisesti ohjelmistotyökaluista tuli saada helpotusta järjestelmän kolmeen loogiseen osaan: lokien siirtelyyn palvelimilta, tiedon jäsentely datasta sekä tallennus. Työkalujen valinnassa olennaista oli, että niitä voitaisiin mahdollisimman vaivattomasti yhdistää Python-ohjelmointikielen avulla ja että nämä työkalut toimisivat järkevästi yhteen toistensa kanssa, mikä säästää aikaa ohjelmakoodin kehitykseen ja varmistaa, että lopullinen val-

mis lokijärjestelmä olisi toiminnaltaan vakaa, luotettava sekä helposti ymmärrettävä kokonaisuus.

3.1 Tiedon prosessoinnin työkalut

Projektia aloitettaessa ei tiedetty tarkalleen, kuinka suuria määriä tietoa lokijärjestelmän tulisi voida käsitellä tai kuinka raskaita operaatioita sen tarvitsisi kyetä suorittamaan tulevaisuudessa. Tämän vuoksi oli hyvä valita teknologioita, jotka pystyvät tarpeen tullen skaalautumaan käsiteltävän tiedon määrän kasvaessa, mikä takaa että järjestelmä kykenee prosessoimaan tarvittaessa suuriakin tietomääriä. Oli siis syytä harkita Big data -käyttöön soveltuvia työkaluja.

Suurien tietomäärien prosessointiin käytettävien ohjelmistojen toimintaperiaate on, että ne kykenevät tarvittaessa skaalaamaan prosessointitehoaan, yleensä käyttäjän hallinnoimalle klusterille. Ohjelmiston suorituskoneisto taas hallitsee tehtävän prosessointityön jakamista klusterin sisällä. Klusterien omatoiminen asennus ja hallinnointi vie kuitenkin aikaa ja on virhealtista, joten käytetyn ohjelmiston valinnassa laitettiin painoarvoa sille, että prosessointitehon skaalaus voitaisiin tehdä mahdollisimman vähäisellä asennustyöllä kuten käyttämällä pilvilaskenta-alustan tarjoamaa versiota ohjelmistosta, joka skaalautuisi tarpeen vaatiessa automaattisesti. Tämä ratkaisu säästäisi aikaa lokijärjestelmän ohjelmalogiikan koodaukseen, joka muuten menisi klusterin hallinnoimiseen. Tapauksissa, joissa hyödynnetään pilvipalveluita, oli lokijärjestelmää luotavalle yritykselle tapana käyttää Googlen Google Cloud Platform -pilvipalvelualustaa (lyhyemmin GCP), joten valittua prosessointiohjelmistoa tuli voida käyttää kyseisen alustan kautta.

Ensimmäiseksi harkinnan alle osui Apache Spark -ohjelmistokehys, joka tarjoaa suorituskoneiston lisäksi ohjelmointirajapinnan, jonka kautta lokijärjestelmän toiminnallisuuksia voitaisiin kehittää. Apache Sparkia pidettiin käytännöllisenä vaihtoehtona, koska GCP-pilvipalvelualusta tarjoaa Dataproc-palvelunsa kautta Apache Spark -klusteria (Google, 2022a). Dataproc-palvelun kautta hallinnoituna Apache Spark -ohjelmistokehys näkyisi käyttäjälle "palvelittomana", eli käyt-

täjän tarvitsisi tehdä vähäistä hallinnointia ja suurimman osan klusterin asennuksesta Dataproc hoitaisi automaattisesti.

Toinen harkittava tiedon prosessoinnin työkalu oli Apache Beam -ohjelmointimalli. Apache Beam on yksi toteutus Dataflow-mallista (Akidau Tyler, Schmidt Eric, Whittle Sam, Bradshaw Robert, Chambers Craig, Chernyak Slava, Fernández-Moctezuma Rafael J., Lax Reuven, McVeety Sam, Mills Daniel & Perry Frances, 2015), jonka tarkoitus on olla yleinen abstraktio datan käsittelyyn. Beam ja Spark eivät ole toisiaan poissulkevia - Apache Beamia voisi verrata enemmänkin Apache Sparkin ohjelmointirajapintaan. Apache Beam -ohjelmakoodia voidaan ajaa Sparkin suorituskoneiston kautta. Apache Beamin eduksi voidaan laskea se, että sillä kehitettyä ohjelmakoodia voidaan ajaa ajoympäristöstä riippumattomasti kunhan alustalle on luotu Beam-ajuri (The Apache Software Foundation, 2022). Apache Beam:ia käyttävää ohjelmakoodia voidaan ajaa lokaalisti kehitysvaiheessa sekä valitulla pilvilaskenta-alustalla, ajuri taas mahdollistaa, että ohjelmakoodin ajossa tarvittava prosessointiteho skaalautuu käyttöympäristön resurssien mukaan. Kuten Apache Spark, Apache Beam on hyvin tuettu GCP-alustalla ja Googlella on oma pilvilaskentaympäristö Apache Beam -ohjelmakoodin ajamiseen Dataflow-palvelun kautta (Google, 2022b).

Taulukko 1.Tiedon prosessoinnin työkalujen erot.

Ominaisuus	**Apache Spark**	**Apache Beam**
Arkkitehtuuri	Ohjelmistokokonaisuus isojen datamäärien käsittelyyn.	Toteutus ohjelmointimallista, joka pyrkii tekemään datan käsittelystä ajoalustasta riippumatonta.
Käyttöönotto	Vaatii klusterin. Käyttö vaatii mahdollisesti ylläpitoa, hallinnointia ja hienosäätöä.	Valmis ohjelmakoodi suoritetaan pilvialustan kautta.
Edut	Antaa käyttäjälle enem-	Helppo käytettävyys.

	män mahdollisuutta muokata klusterin toimintaa, näin säästäen kuluja. Laajasti käytetty ja suuri dokumentaatio.	Mahdollisuus vaihtaa ajoympäristöä.
Haitat	Käyttöönoton ja ylläpidon mahdollinen monimutkaisuus ja hienosäätö.	Kohtuullisen uusi teknologia. Ei laajaa dokumentaatiota.

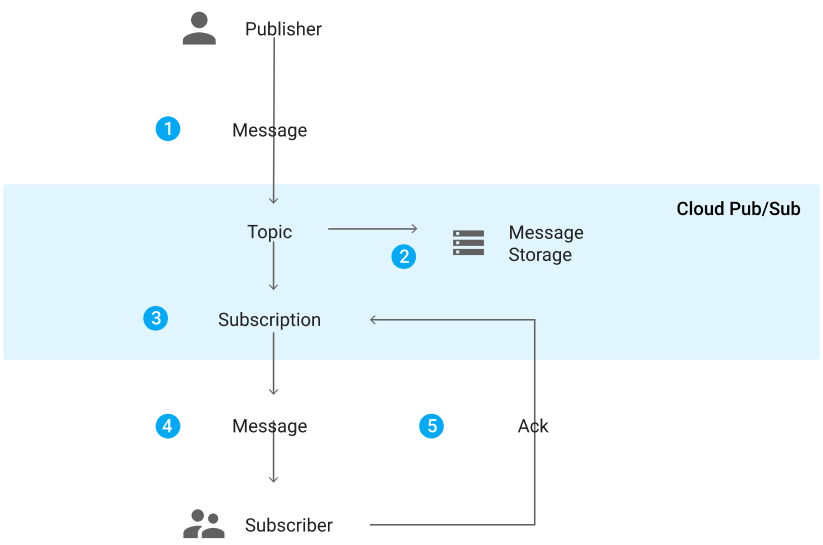
Molemmat Apache Spark ja Apache Beam vaikuttivat käytännöllisiltä vaihtoehdoilta, joiden ympärille lokijärjestelmä voitaisiin kehittää. Apache Beamin arvioitiin kuitenkin olevan käyttäjäystävällisempi, sillä sen käyttö vaatisi vain ohjelmakoodin kehityksen, ja valmis koodi voitaisiin yksinkertaisesti siirtää suoritettavaksi Googlen Dataflow'n ympäristöön. Tarpeen vaatiessa Beam-ohjelmakoodi voitaisiin myös siirtää toiseen pilviympäristöön suoritettavaksi, jos esimerkiksi Googlen Dataflow -palvelun käytön hinta kasvaisi liian suureksi. Arvioitujen etujen perusteella lokijärjestelmää lähdettiin kehittämään Apache Beam -ohjelmointimallin kanssa.

3.2 Tiedon siirto

Lähtökohtaisesti lokijärjestelmän tiedon prosessoinnin tuli tapahtua keskitetysti Dataflow'ssa ajettavan ohjelmakoodin kautta, ja tuli miettiä, että miten lokeja lähettävät palvelimet tulisi yhdistää tiedon prosessointiin. Suoraviivaisin tapa toteuttaa tiedon siirto palvelimilta olisi luoda suora yhteys lokeja käsittelevään pilviympäristöön. Tarkemmalla pohdinnalla suora kytkentä palvelinten ja lokiprosessin välille osoittautui virhealttiiksi. Suurin ongelma asetelmassa on, että siinä ei ole takeita, että lokiviestit tulevat prosessoiduiksi. Jos lokiviestin lähetyksessä tai prosessoinnissa tapahtuu jokin häiriö tai virhe, niin tieto lokiviestistä menetetään. Tällaisessa asetelmassa täytyisi olla jokin edistyneempi kommunikointiprotokolla, jonka avulla voitaisiin varmistaa, että tieto on lähetetty ja prosessoitu oikein ja että tiedon siirron virheen sattuessa tieto osattaisiin lähettää uudes-

taan. Tämä osoittautui liian epäkäytännölliseksi, ja päädyttiin tilanteen ratkaisemiseksi käyttämään viestijonoa.

Viestijonossa merkittävää on, ettei tietoa välittävien ja käyttävien osapuolien välille tarvitse luoda suoraa kommunikointia. Viestijonon käyttö luo tiedonsiirrosta asynkronista - osapuolien ei tarvitse tietää muista viestijonoon osallistujista vaan voivat lukea tai vastaanottaa tietoa omaan tahtiinsa. Lokijärjestelmässä käytettäväksi viestijonoksi valittiin Googlen Pub/Sub. Monien muiden GCP-ympäristössä toimivien palveluiden tavoin Pub/Sub-viestijono on pilvipalvelu, jonka etuna on, että käyttäjän ei tarvitse huolehtia palvelun ylläpidosta tai skaalautuvuudesta (Google, 2022c).



Kuva 1. Pub/Sub-viestijonon toiminta (Google, 2022c).

Käyttäjän näkökulmasta Pub/Sub-viestijonossa viestintä tapahtuu jakamalla sitä käyttävät ohjelmistot julkaisijoihin ja tilaajiin. Kuvassa 1 voidaan nähdä, kuinka julkaisijat lähettävät viestejä tietyn otsikon alle ja otsikon tilaajat vastaanottavat viestit tätä kautta. Viestijonon kautta palvelimet voivat julkaista viestejä saman otsikon alle ja otsikon tilaajana lokiprosessointi käsittelee viestit niiden ilmestyttyä viestijonoon. Pub/Sub-viestijonossa käytetty viestintäprotokolla varmistaa,

että tilaaja saa tilaamastaan otsikosta tarvitsemansa tiedon, ja näin voidaan vähentää mahdollisia tilanteita, joissa lokijärjestelmästä putoaisi pois lokiviestejä esimerkiksi erilaisten verkkohäiriöiden sattuessa.

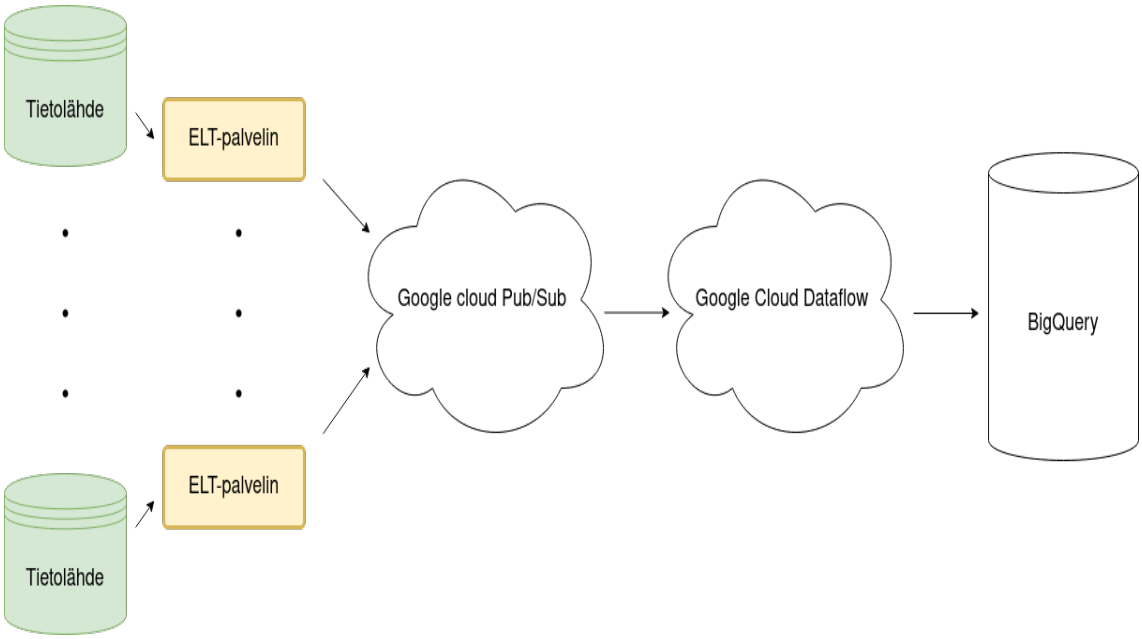
3.3 Lokitiedon tallennus ja kerryttäminen

Lokeista saatua tietoa oli tarkoitus tallentaa paikkaan, josta sitä voitaisiin helposti tarkastella esimerkiksi BI-työkalujen avulla. Lähtökohtaisesti lokijärjestelmän tarkoitus on käsitellä suuria määriä lokeja. Sen ei pitäisi kerryttää samalla tahdilla dataa, jota varastoida, vaan enemmänkin tehdä hyödyllisiä tiedon yhdistelyjä ja varastoida ne. Relaatiotietokannoille tyypillisiä ominaisuuksia kuten tietorivien päivitystä ja poistoa ei pidetty lokijärjestelmälle tarpeellisena. Muussa lokijärjestelmän toiminnallisuuden kehityksessä oli jo sitouduttu käyttämään GCP-ympäristön kanssa yhteensopivia työkaluja, joten oli luontevaa valita työkalu myös tiedon tallentamiseen tältä kannalta. Näiden valossa tiedon tallennuspaikaksi valittiin Googlen BigQuery, joka on pilvipohjainen tietovarasto. BigQueryn pilvipohjaisuuden ansiosta vältyttäisiin perinteisen relaatiotietokannan asennukselta ja ylläpidolta. BigQueryn skaalautuvuutta pidettiin myös pienenä etuna, sillä se on luotu suurien tietomäärien käsittelyyn. Jos tulevaisuudessa lokeista kerätyn tiedon määrä kasvaisikin suureksi, BigQueryn käyttö antaa hieman takeita, että tietokannan tutkinnassa käytettävät SQL-haut suoriutuvat tehtävästään koohtuullisessa ajassa.

3.4 Lopullinen arkkitehtuuri

Yhteenvetona lokijärjestelmän ohjelmistoarkkitehtuuri ottaa vaikutteita mikropalveluiden käsitteestä, jossa toteutus jaetaan toisistaan riippumattomiin osiin, tässä tapauksessa hyödynnetään GCP-pilvipalvelualustan teknologioita. Hyötyjä mikropalvelumallia matkivalla toteutuksella on useita. Selkeä rajaus tehtävien välillä tekee järjestelmän toiminnan ymmärtämisestä helpompaa. Samalla järjestelmän koko virheensietokyky paranee, koska palvelut eivät ole selkeästi kietoutuneet toisiinsa. Jos virheitä esiintyy tietyssä osassa järjestelmän luomaa

dataputkea, häiriön ei pitäisi levitä muihin osiin niiden ollessa toisistaan riippumattomia. Ideaalisesti palvelut eivät olisi ollenkaan tietoisia toisistaan, vaan ne toimisivat kuin funktiot, jotka ottavat jotain tietoa sisään ja puskevat muutettua tietoa eteenpäin seuraavaan järjestelmän osaan.



Kuva 2. Lokijärjestelmän arkkitehtuuri

Kuvassa 2 näkyy lokijärjestelmän oleelliset osat. Osien välissä olevat nuolet viittaavat tiedon siirron suuntaan. Tietolähteistä kuten REST-rajapinnat ja tietokannat tieto ladataan ELT-palvelinten avulla. Latauksista syntyvät lokiviestit siirtyvät osaksi lokijärjestelmää, kun palvelimet lähettävät lokinsa viestijonoon. Dataflow'ssa ajettava ohjelmakoodi käsittelee viestijonon kautta tulevat viestit lajitellen niitä onnistumistilan mukaan ja tehden niistä tilastolaskelmia. Lopulta hyödyllinen eristetty tieto tallennetaan BigQuery-tietovarastoon.

4 Lokijärjestelmän toteutus

4.1 Viestijonon käyttöönotto

Ennen kuin palvelimilta tulevia lokeja voitiin prosessoida ohjelmakoodin avulla, oli syytä ottaa käyttöön Pub/Sub-viestijono, jonka kautta dataa voitiin siirtää. Google tarjoaa GCP-alustan projektinhallintaan ja asetusten konfiguroitiin gcloud-komentorivityökalun, jonka kautta voidaan suorittaa samoja toimintoja kuin alustan web-sovelluksen kautta. Työkalun käyttö vaatii ohjelmiston asennuksen lisäksi omien GCP-alustan tunnusten ja projektitietojen lisäystä sen konfiguraatioon. Gcloud-työkalu konfiguroitiin lokijärjestelmän projektia varten, jonka jälkeen sitä voitiin hyödyntää viestijonon käyttöönotossa.

```
$ gcloud pubsub topics create LOGGING
$ gcloud pubsub subscriptions create LOGGING_SUBSCRIPTION \
   --topic=LOGGING
```

Esimerkkikoodi 1. Gcloud-työkalun avulla luodaan logging-niminen Pub/Sub-aihe sekä aiheen tilaaja.

Gcloud-työkalun avulla viestijonon käyttöönotto on yksinkertaista. Esimerkkikoodi 1:ssä viestijonossa käytettävän aiheen nimeksi asetetaan LOGGING. Pub/Sub-viestijonon mallin sisällä aihe merkitsee kanavaa, jota kautta lokitietoa voidaan siirtää käsiteltäväksi. Samalla luodaan aiheelle tilaaja, LOGGING_SUBSCRIPTION, joka kuluttaa jonon kautta siirrettyjä viestejä. Pub/Sub-viestijonon käytössä tilaajia täytyy luoda ja määrittää erikseen, sillä tilaajien määrä ja asetukset vaikuttavat viestijonon sisäiseen toimintaan. Esimerkiksi jos samaa tietoa tarvittaisiin monessa eri lähteessä, voidaan viestijonon aiheelle luoda monia eri tilaajia. Tilaajien määrän avulla viestijono säilyttää viestejään, kunnes kaikki tilaajat ovat onnistuneesti lukeneet jonossa olevat viestit tai kunnes viestit ylittävät asetetun säilytysajan, jolloin viestit niin sanotusti vanhentuvat ja täten poistetaan jonosta.

4.2 Lokien tuonti viestijonoon

Lokeja tuli lähettää Pub/Sub-viestijonoon monesta eri palvelinympäristöstä sekä niissä ajettavista ohjelmakoodeista. Tämän valossa oli hyvä miettiä lähestymistapa, joka ei vaatisi, että jokaista ajettavaa ohjelmakoodia tarvitsisi muokata sisältämään lokiviestien lähetykseen liittyviä ominaisuuksia. Ajon aikana ohjelmakoodit palauttavat tulosteena lokeja tapahtumista, joten päätettiin tehdä toteutus käyttäen Unix-putkea. Unix-putki on alkuperäisesti Unix-pohjaisista käyttöjärjestelmistä lähtöisin oleva prosessien välinen kommunikointitapa, jossa ohjelman tuloste voidaan putkittaa toisen ohjelman syötteeksi (Free Software Foundation, 2022). Tällä tavalla lokeja tulostaviin ohjelmakoodeihin ei tarvitse tehdä ollenkaan muutoksia, vaan ohjelmakoodien tulosteet voidaan siirtää erilliseen ohjelmaan, jonka tehtävä on lähettää ne viestijonoon. Ohjelmakoodeja ajetaan palvelinympäristöissä, joissa käyttöjärjestelmät ovat Linux-pohjaisia ja joissa on käytettävissä Bash-komentotulkki. Bash-komentotulkin kautta monet eri käyttöjärjestelmän ominaisuudet, kuten Unix-putki, ovat käytettävissä.

```
$ python seurattava_ohjelmakoodi.py | python \ lokien_pusku_viestijonoon.py
```

Esimerkkikoodi 2. Havainnollistus Bash-ohjelmakoodista, jossa kaksi ohjelmakoodia koostetaan toimimaan yhdessä komentotulkin putki-operaattorin avulla.

Esimerkkikoodi 2:ssa havainnollistetaan, miten ensimmäisen seurattava_ohjelmakoodi.py-tiedoston Python-ohjelmakoodin ajon tuloste saadaan siirrettyä lokien_pusku_viestijonoon.py-tiedoston ajon syötteeksi, jonka tarkoitus on siirtää lokit viestijonoon. Unix-putkea käyttävää kommunikointitapaa voitiin soveltaa kaikkien tarkkailtavien ohjelmakoodien osalta tehden uusien ohjelmakoodien lokien lähetyksestä viestijonoon suoraviivaista ja vaatien vain vähän manuaalista työtä. Kun käytetty Bash-ohjelma oli selvitetty, sitä voitiin automatisoida cron-automaatiotyökalun avulla. Cron on komentoriviltä käytettävä tyypillisesti Unix-pohjaisten käyttöjärjestelmien työkalu, jolla erilaisia komentoja voidaan ajastaa ajettavaksi tietyin väliajoin. Käytetyt ohjelmakoodien putkitukseen liittyvät ohjel-

makoodit sijoitettiin Cron-automaatiotyökalun asetustiedostoon ja niille spesifoitiin tietyt säännölliset ajoajat.

4.3 Lokien jäsennys

Pub/Sub-viestijonon huolehtiessa, että kaikki halutut lokiviestit saadaan keskitettyä yhdeksi suoratoistettavaksi datasetiksi, voitiin keskittyä siihen, mitä tietoa kertyneistä lokitiedoista voitaisiin eristää ja minkälaisia päätelmiä siitä voitaisiin tehdä. Lokiviestien parsiminen, muokkaus ja tulkinta keskitettiin yhteen Apache Beam -ohjelmointimallia käyttävään Python-ohjelmakoodiin, jota tuli voida ajaa Dataflow-palvelussa.

Saapuessaan käsiteltäväksi lokitieto on merkkijono ilman selvää rakennetta. Yleisimmin tiedon kommunikointi laitteiden välillä noudattaa jonkinlaista määritettyä rakennetta, kuten JSON:ää tai XML:ää, jota molempien laitteiden ohjelmakoodit osaavat välittää ja tulkita. Lokijärjestelmän toteutuksessa pyrittiin siihen, että lähtöjärjestelmien lokien luontiin ei tarvitsisi tehdä muutoksia, joita määrätyn rakenteen saavuttaminen vaatisi. Lokitiedossa saattoi myös olla pieniä eroja vaihdellen ohjelmakoodien mukaan, mikä olisi vaikeuttanut järkevän kommunikoitavan rakenteen löytämistä. Merkkijonona olevan lokitiedon parsimisessa päädyttiin käyttämään säännöllisiä lausekkeita hyödyllisen tiedon etsimiseen.

```
(1. ) Palvelin1 (2. ) - 2022-07-27 14:12:31 INFO - Task: (3. ) Extract
Google Analytics: Starting task run state: (5. ) Start
(1. ) Palvelin1 (2. ) 2022-07-27 14:13:23 INFO - Task: (3. ) Extract
Google Analytics: Meta data: (4. )  ...
(1. ) Palvelin1 (2. ) 2022-07-27 14:15:30 INFO - Task: (3. ) Extract
Google Analytics: Finished task run state: (5. ) Success
```

Esimerkkikoodi 3. Esimerkkejä kolmesta jäsennettävästä lokirivistä

Esimerkkikoodissa 3 nähdään kolme saman ohjelmakoodin ajon tulostamista lokiriveistä. Esimerkin lokiriveihin on havainnollistamisen vuoksi lisätty numerointi tekstiosien kohdalle, jotka halutaan parsia säännöllisten lauseiden avulla. Numeroidut kohdat viittaavat seuraaviin tietoihin:

- lokin tulostaneen palvelimen nimi
- lokin tapahtuma-aika
- ohjelmakoodin nimi
- meta data, saattaa sisältää ohjelmakoodin lataaman datan määrän
- ohjelmakoodin tila.

Tiedon eristäminen lokista toimii myös turhan datan suodattimena - jos tiettyjä asioita ei löydetä lokitekstin rivistä, se jätetään pois seuraavista datan muokkauksen vaiheista. Lokien jäsentämisen ansiosta siitä voitiin muodostaa tiedonkäsittelyn kannalta käytännöllisiä, yhtenäiseen malliin sopivia Python-ohjelmointikielen dictionary-tietorakenteita. Kyseinen tietorakenne on käytännössä hajautustaulu, jossa löydetyt hyödylliset tiedot voidaan rajata avain ja arvo -pareiksi.

4.4 Tiedon kerrytys ja tulkinta

Lokitiedon jäsennys -toiminnallisuuden ansiosta saatiin käyttöön tietorakenne, jota voitiin käyttää tiedon tulkintaan ja monimutkaisempaan prosessointiin. Jäsennetystä tiedosta haluttiin selvittää seuraavia asioita:

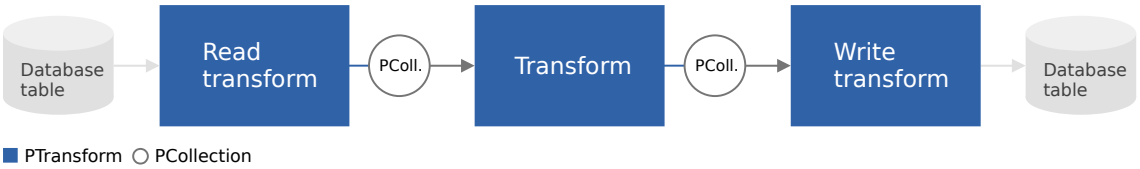
- kuinka paljon tietoa ajettu ohjelmakoodi on ladannut tai tallentanut

- tietyn ohjelmakoodin ajonaika

- asiakkaan päivän ohjelmakoodien ajoaikojen kokonaissumma.

Tässä vaiheessa oli hyvä selvittää tarkemmin Apache Beam -ohjelmointimallin toiminnallisuuksia, ja sitä, miten niitä voitaisiin käyttää hyödyksi yllämainittujen tietojen selvittämiseen.

4.4.1 Apache Beam -ohjelmointimalli



Kuva 3. Apache Beam -ohjelmointimalli (The Apache Software Foundation, 2022b)

Apache Beam -ohjelmointimallissa halutulle tiedon käsittelylle määritetään data-putki. Dataputki on suunnattu syklitön verkko, jonka solmut ovat datalle tehtäviä transformaatioita ja kaaret kuvaavat putkessa kulkevaa dataa. Rakenne tarkoittaa käytännössä, että datalla on tietty suunta, eikä sitä voida ohjata johonkin aiempaan transformaation vaiheeseen esimerkiksi jonkin ehdon avulla. On myös hyvä huomioida, että ohjelmointimallin tietorakenteet ovat muttumattomia. Transformaatiot siis aina palauttavat uuden tietorakenteen, joten malli voidaan sanoa olevan hyvin lähellä funktionaalista ohjelmointia. Kuvassa 3 on yleinen havainnollistus mallin yksinkertaisuudesta: tietoa luetaan datan kokoelmaan (PCollection), muokataan (PTransform) ja lopulta tallennetaan tietokantaan. Apache Beam -ohjelmakirjastossa PCollection ja PTransform ovat yleisiä luokkia ohjelmointimallin tietorakenteille ja transformaatioille. Luokat mahdollistavat tiedon muutosten suorituksen rinnakkaisesti sekä sarjallisuuden, jota tarvitaan esimerkiksi, kun tietoa tai transformaatioita siirretään hajautetun ajoalustan työläisille.

```
import apache_beam as beam
from apache_beam import (io, Pipeline)

with Pipeline() as pipeline:
     log_lines = (
          pipeline
          | "Lue lokeja Pub/sub-viestijonosta" >>
               io.ReadFromPubSub(subscription=subscription_path)
          | "Muunna Bytes-objektit UTF-8-merkkijonoksi" >>
               beam.Map(lambda b: b.decode('utf-8'))
          | "Jäsennä lokiriveistä tarvittu tieto dict-objekteiksi" >>
               beam.ParDo(ParseLogString())
     )
```

Esimerkkikoodi 4. Lokien luku ja jäsennys Apache Beam -ohjelmointimallin avulla.

Esimerkkikoodi 4:ssa nähdään, miten viestijonosta lokien lukeminen ja haluttujen tietojen jäsennys voidaan määrittää Apache Beam -ohjelmointimallissa. Ensin määritetään dataputki, pipeline, joka esittää mallissa muodostettavaa verkkorakennetta. Ohjelmointimallin toiminnallisuuksia esitetään Python-pohjaisen ohjelmakirjaston sisällä, kun käytetään yleensä bittioperaatioita kuvaavia OR (|) ja oikealle siirto (>>) -merkkejä. Ohjelmakirjastossa merkkien käytön tarkoitus on luoda mallin toiminnalle sopiva ja selkeä syntaksi. Käytännössä OR-bittioperaattori kuvaa yhtä dataputken osaa, jokaisen putken jälkeen tulee selitys tehtävästä transformaatiosta, jonka jälkeen oikealle siirto -bittioperaattori ja suoritettava transformaatio. Esimerkkikoodissa luetaan lokivestejä viestijonosta rivi kerrallaan käyttäen ohjelmakirjaston ReadFromPubSub-transformaatiota. Ohjelmointimallissa dataa käsitellään aina kokoelmina, joten luetuille riveille tehdään Map-transformaatio, jossa jokainen Bytes-objekti muunnetaan UTF-8-merkkijonoksi käyttäen anonyymiä funktiota. Lopuksi lokiriveistä halutun tiedon jäsennyksessä käytetään ohjelmakirjaston ParDo-transformaatiota yhdistettynä erilliseen ParseLogString-funktioon, jonka määritystä ei ole esimerkkikoodissa. ParDo eroaa Map-transformaatiosta siinä, että se palauttaa nolla tai enemmän alkiota jokaista argumentin kokoelman alkiota kohden. ParseLogString-funktion käyttö ParDo-transformaatiossa purkaa lokirivin helposti käsiteltäväksi hajautustauluksi, jos siitä löydetään halutut tiedot. Lopuksi log_lines-muuttujaan tallennetaan jäsennetyt lokitiedot, joka on käytännössä PCollection-luokka, jota voidaan jatkokäsitellä seuraavissa dataputken vaiheissa.

4.4.2 Tiedon rajaus aikaväleihin

Tilanteissa, joissa halutaan yhdistää yksittäisiä lokirivejä ja luoda yhdistetystä datasta yleisiä päätelmiä, tarvitaan, että datalla on jokin yhteinen tekijä, jonka mukaan sitä voidaan koota ja ryhmitellä. Apache Beam -ohjelmointimallissa yksi yleinen tapa tehdä yhdistelyä ovat aikaleimat. Aikaleimoja käyttävässä rajauksessa vaihtoehtoina on rajata tietoa prosessointiajan mukaan tai tiedon mukana

tulleen tapahtumahetken avulla. Yleisesti tiedon yhdistely tapahtumahetken mukaan on Apache Beam -ohjelmointimallissa varmempi tapa, sillä itse tiedon prosessointiaika voi vaihdella. Lokijärjestelmän prosessoitaville lokitiedoille tuli lisätä aikaleima ohjelmakirjaston vaatimalla tavalla.

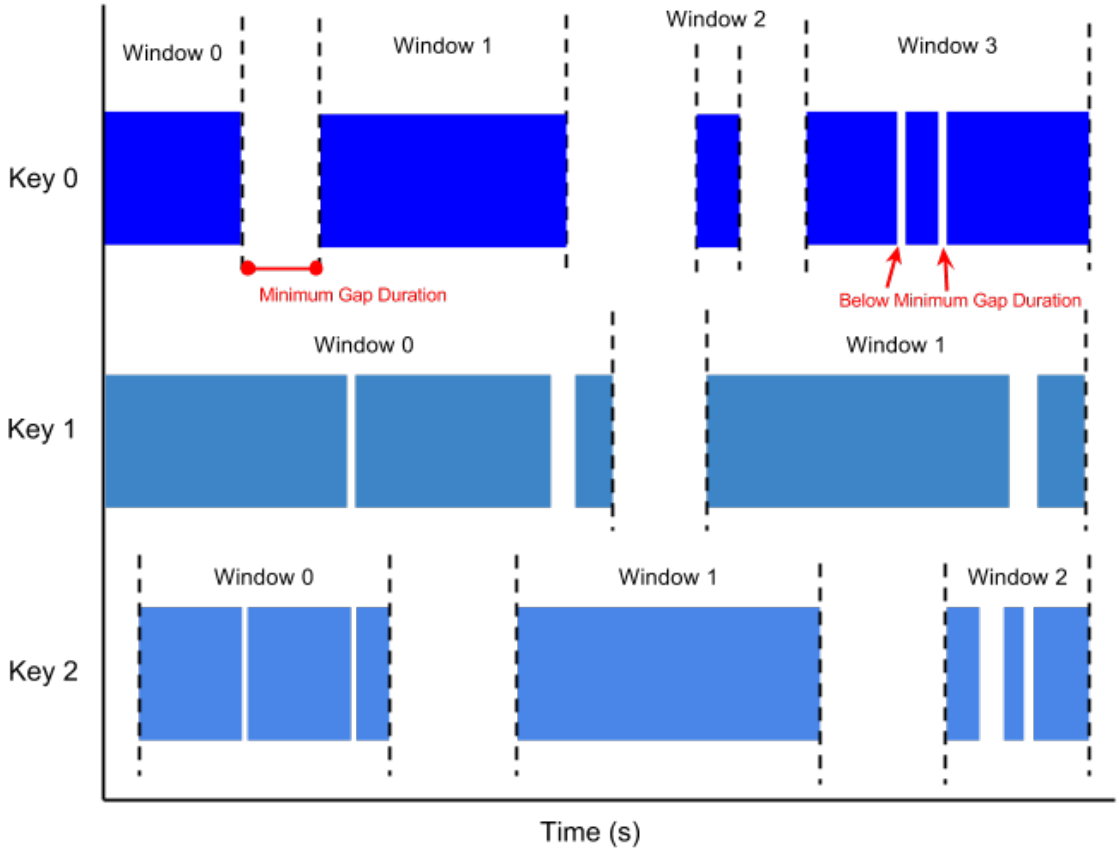
```
import time
from apache_beam.transforms.window import TimestampedValue

def to_unix_time(time_str: str,
    time_format="%Y-%m-%d %H:%M:%S") -> int:
        struct_time = time.strptime(time_str, time_format)
        return int(time.mktime(struct_time))

log_lines_timestamped = (
    log_lines
    | "Lisää Unix-ajallinen aikaleima" >> beam.Map(
        lambda log_line: TimestampedValue(
            log_line, to_unix_time(log_line["timestamp"]))
      )
  )
```

Esimerkkikoodi 5. Aikaleiman lisääminen käsiteltäviin lokitietoihin

Lokijärjestelmän läpi käytävissä lokeissa tulee mukana aikaleima lokin tapahtuma-ajasta. Esimerkkikoodi 5:ssä jatketaan esimerkkikoodi 4:n log_lines-muuttujan sisältävien jäsennettyjen lokitietojen käsittelyä. Timestamp-avaimen alla oleva aikaleima muunnetaan ensin Unix-ajaksi käyttäen to_unix_time-funktiota. Funktion vakioparametrina on annettu käytettävä ajan formaatti, time_format, jota Pythonin time-ohjelmakirjasto tarvitsee osatakseen lukea ajan merkkijonona olevasta time_str-parametrista strptime-funktion avulla. Funktio lopulta palauttaa Unix-ajan kokonaislukuna. Laskettu Unix-aika ja lokirivin hajautustaulu annetaan argumentteina TimestampedValue-luokan luontiin. Tämä prosessi suoritetaan kaikille käsiteltäville lokeille käyttäen Map-transformaatiota. Luotava TimestampedValue-luokka on käytännössä päällystys mille tahansa arvolle, jolle halutaan tehdä aikaväleihin rajattuja transformaatiota. Aikaleimojen avulla lokeja voidaan rajata aikaikkunoihin, mikä auttaa vastaamaan kysymyksiin, jotka riippuvat datan kokonaismäärästä. Tällaisia ovat esimerkiksi, kuinka paljon tietoa tietty ohjelmakoodi on ladannut yhteensä tai kuinka kauan päivän aikana ajetut ohjelmakoodit ovat kestäneet.

4.4.3 Sessioitu tieto

Ohjelmakoodit, joiden lokeja järjestelmän on tarkoitus kerätä, lähettävät ajonaikana säännöllisesti lokiviestejä tehtävien ELT-operaatioiden tilasta. Säännöllisesti ajettavien ohjelmakoodien ajot voidaan erottaa toisistaan niiden ajoaikojen perusteella. Tyypillisesti yksi ohjelmakoodi saattaa lähettää kymmeniä ellei satoja lokiviestejä lyhyen ajan sisällä, ja sitten kyseiselle saman nimiselle ohjelmakoodille tulee tunneissa tai päivissä mitattava väli riippuen siitä, kuinka usein ajettavaksi ohjelmakoodi on ajastettu. Tätä aikaväliä voidaan käyttää määrittämään ajon sessio. Sessio kuvaa tässä tapauksessa siis yhden ohjelmakoodin ajon tapahtumaa, ja tarkoitus on eritellä kaikki yksittäiset lokirivit saman session alle, jotta voidaan määrittää esimerkiksi, kuinka kauan tietyn ohjelmakoodin ajo on kestänyt tai kuinka paljon dataa se on ladannut.



Kuva 4. Havainnollistus sessioista. (The Apache Software Foundation, 2022b)

Kuvassa 4 kuvataan kolmeen eri avaimeen liittyvien tietoalkioiden esiintymistä dataputkessa, jotka jaetaan window-sanalla merkittyihin aikaikkunoihin. Asetetun aikaikkunan minimivälin, minimum gap duration, ylittyessä tietyn avaimen kohdalla, havaitaan sessio päättyneeksi ja aloitetaan uusi sessio. Päättyneestä sessiosta tiedetään, ettei sen alle enää tule lisää tietoa, joten siitä voidaan luoda kooste ja tallentaa se esimerkiksi tietokantaan. Lokijärjestelmän sisältämien aikaleimojen avulla sessiointia voitiin käyttää, kunhan oli selvillä, kuinka pitkä aikaikkunan minimivälin tulisi olla.

```
from apache_beam.transforms.window import Sessions

total_rows_saved_per_session = (
    log_lines_timestamped
    | "Luo avain, joka erottaa ohjelmakoodien lokit toisistaan" >>
        beam.Map(
```

```
            lambda log: (
                f"{log['customer_id']}{log['task_name']}",
                log
            )
        )
    | "Määritä sessio 10 minuutin minimivälillä" >>
        beam.WindowInto(Sessions(60 * 10))
    | "Laske päättyneen session ladattu rivimäärä" >>
        beam.CombinePerKey(CombineRowCount())
)
```

Esimerkkikoodi 6. Aikaleimatusta lokitiedosta luodaan yhdiste, joka kertoo sessiossa tallennetun datan summan.

Esimerkkikoodi 6:ssa pyritään selvittämään, kuinka paljon tietyn ohjelmakoodin ajo on ladannut dataa tietolähteestä. Ensin jokaiselle aiemmin määritetylle lokille luodaan uniikki avain, jonka avulla tieto voidaan laskea tietyn session alle. Uniikki avain luodaan yhdistämällä asiakkaan tunniste ja ohjelmakoodin nimi, esimerkkikoodissa customer_id ja task_name, yhdeksi merkkijonoksi. Tämä toimii lokien erottamisessa toisistaan olettaen, ettei samalla asiakkaalla ole kahta samannimistä ohjelmakoodia suorituksessa yhdenaikaisesti. Uniikilla avaimella eroteltu data voidaan seuraavaksi ikkunoida sessioksi käyttäen Apache Beam -ohjelmakirjaston WindowInto-transformaatiota ja Sessions-funktiota. Sessions-funktio ottaa argumenttina session minimivälin sekunteina, joka on tässä määritetty 10 minuutin pituiseksi. Kun yhden avaimen sessio havaitaan päättyneeksi ja sen alle kuuluvat lokit ovat selvillä, voidaan laskea kyseisen session aikana ladattujen tietorivien summa. Apache Beam -ohjelmakirjaston CombinePerKey-transformaatio erottaa session avaimen alle kuuluvat alkiot omaksi setiksi, ja yhdistää ne käyttäen CombineRowCount-funktiota, jota ei ole määritetty esimerkkikoodissa. Kyseinen funktio purkaa loki-hajautustauluista rivien määrät, laskee ne yhteen ja luo yhden sessiota viittaavan alkion, joka voidaan tallentaa BigQuery-tietovarastoon.

4.4.4 Tilallinen tiedon prosessointi

Reaaliaikaisessa datan käsittelyssä sessiointi on kätevä tapa määrittää, milloin yksi datasetti on loppunut. Sen ongelmana voidaan pitää sitä, että se vaatii tietyn aikavälin erottamaan, milloin koko session data on tiedossa. Tämä aikaväli

esiintyy lopullisessa datassa viiveenä - jos session minimiväliksi on määritetty 10 minuuttia, voidaan saada tieto koko session datasta 10 minuuttia viimeisen datasetin alkion saapumisesta. Apache Beam -ohjelmointikirjaston tilallisen tiedon prosessoinnin avulla voidaan luoda funktioita, jotka pitävät yllä tilaa läpikäytävästä datasta, ja kun käytetty ehto täyttyy voidaan niin sanotusti "laukaista" uusi tapahtuma, joka on riippuvainen mahdollisesti monesta aiemmasta tietoalkiosta. Uuden tapahtuman luonti ei siis tässä tapauksessa aiheuta lähes ollenkaan viivettä.

Lokijärjestelmässä tilallista tiedon prosessointia käytettiin selvittämään, kuinka kauan ohjelmakoodin ajoprosessi kesti ja että onnistuiko se virheettömästi vai ei. Toteutetun ohjelmakoodin ideana oli, että tilalliseen funktioon tallennetaan loki, joka saapuu järjestelmään statuksella start, joka merkitsee ohjelmakoodin ajon aloitusta. Kun funktio havaitsee molemmat, ajon aloitus- sekä lopetus -statukset, luodaan uusi tapahtuma, joka sisältää tiedon ohjelman ajon kestosta. Uusi tapahtuma voidaan siis tallentaa tietovarastoon heti, kun ohjelman ajo havaitaan lokien perusteella päättyneeksi.

```
class TrackTaskState(DoFn):
    ...
    def process(
         self,
         element: typing.Tuple[str, typing.Dict],
         timestamp=DoFn.TimestampParam,
         start_status_logs=DoFn.StateParam(STARTING_STATE),
         end_status_logs=DoFn.StateParam(ENDING_STATE),
         max_timestamp=DoFn.StateParam(MAX_TIMETAMP),
         timer=DoFn.TimerParam(CHECK_FOR_TIMEOUT)
    ):
        _, log = element
        task_name = log["task_name"]
        start_dict = start_status_logs.read() or {}
        end_dict = end_status_logs.read() or {}
        if log["task_status"] == "START"
            start_dict[task_name] = log
        else:
            end_dict[task_name] = log
        starting_log_event = start_dict.get(task_name)
        ending_log_event = end_dict.get(task_name)
        if starting_log_event and ending_log_event:
            start_dict.pop(task_name)
            end_dict.pop(task_name)
            start_status_logs.write(start_dict)
            end_status_logs.write(end_dict)
```

```
            yield (starting_log_event, ending_log_event)
        else:
            start_status_logs.write(start_dict)
            end_status_logs.write(end_dict)
            max_timestamp.write(timestamp)
            expiration_time = max_timestamp.read() + Duration(
                seconds=3 * 60 * 60
            )
            timer.set(expiration_time)
```

Esimerkkikoodi 7. Apache Beam -ohjelmakirjaston säännöin toteutettu tilaa ylläpitävä funktio-luokka.

Esimerkkikoodissa 7 on osa ohjelmakoodien ajojen tilasta kirjaa pitävästä funktiosta. Tässä tapauksessa funktiolle täytyi tehdä oma luokkansa, TrackTaskState-luokka, koska se sisältää tilaa pitäviä muuttujia. Luokan process-metodissa element-parametri vastaa funktioon tulevaa lokitiedon alkiota, jolla on sen aiemmin palvelimen ja ohjelmakoodin nimen avulla luotu avain. Avaimen perusteella elementit erotellaan omiin datasetteihinsä ennen funktioon tuloa. Tällä tavalla varmistetaan, että funktiossa vertaillaan vain samalta palvelimelta ja samasta ohjelmakoodista tulleita alkioita. Muut process-metodin parametrit kuuluvat sisäisen tilan ylläpitämiseen. Start_status_logs- ja end_status_logs-parametrit viittaavat funktion tilaan tallennettavia hajautustauluja, joilla pidetään yllä lokeja, eritellen ne statuksen mukaan. Kun status-pari havaitaan, ne poistetaan funktion tilasta ja palautetaan yield-avainsanan avulla funktiosta.

```
@on_timer(CHECK_FOR_TIMEOUT)
def expiry_callback(
    self,
    start_status_logs=DoFn.StateParam(STARTING_STATE),
    end_status_logs=DoFn.StateParam(ENDING_STATE),
    timer_tag=DoFn.StateParam(MAX_TIMETAMP),
):
    start_status_logs.clear()
    end_status_logs.clear()
    timer_tag.clear()
```

Esimerkkikoodi 8. Callback-funktio funktion tilan vapauttamiseen.

Funktion tilalle ei ohjelmakirjastossa ole valmiiksi asetettu toimintoa, joka automaattisesti hallitsisi tai poistaisi turhan, vanhaksi menneen tilan. Jos tilan eli muistin vapauttamista ei hallita, saattaa dataputken suorituksessa tapahtua muistivuoto, joka käyttää ajoalustan resursseja turhaan. Esimerkkikoodi 7:n timer-, timestamp- ja max_timestamp-parametreja käytetään tämän vuoksi funk-

tion tilan roskienkeräykseen. Max_timestamp-muuttujassa pidetään yllä tietoa ajasta, jolloin viimeksi funktioon saapui loki, jolle ei löydy status-paria. Muuttujassa olevaan aikaleimaan lisätään kolmen tuntia, mikä luo expiration_time-muuttujan, jossa on aikaleima tilan erääntymisajankohdalle. Aikaleima syötetään timer-muuttujaan, jonka ajan kohdalla kutsutaan esimerkkikoodi 8:ssa määritettyä expiry_callback-metodia. Jos kolmen tunnin aikana lokille ei ole saapunut vastaavaa status-paria, voidaan olettaa, että dataputkessa tai lokia pidettävästä järjestelmässä on tapahtunut jokin virhe, ja tila poistetaan roskienkeräykseen käytettävien muuttujien clear-metodilla.

4.4.5 Tiedon varastointi

Lokeista kerättyä hyödyllistä tietoa tuli kerryttää tietokantaan, josta sitä voitiin tarkastella. Apache Beam -ohjelmakirjasto tekee BigQuery-tietovaraston käytöstä helppoa, mikä tarjoaa transformaation, joka tallentaa syötetyn datasetin konfiguroituun tietovaraston tauluun.

```
from apache_beam import io

table_schema_completed_tasks = (
    "customer_id:STRING, "
    "start_timestamp_utc:TIMESTAMP, "
    "end_timestamp_utc:TIMESTAMP, "
    "task_name:STRING, "
    "task_run_status:STRING, "
    "runtime_total_seconds:INTEGER"
)

_ = (
    completed_tasks
    | "Tallenna valmis ajotapahtuma tietovarastoon"
    >> io.WriteToBigQuery(
        table=f"{project}:{dataset}.daily_runtime_per_customer",
        schema=table_schema_completed_tasks,
        create_disposition=io.BigQueryDisposition.CREATE_IF_NEEDED,
        write_disposition=io.BigQueryDisposition.WRITE_APPEND,
        method="STREAMING_INSERTS"
    )
)
```

Esimerkkikoodi 9. Valmiiden ajotapahtumien tallennus tietovarastoon

Esimerkkikoodissa 9 jatketaan esimerkkikoodi 7:n tilallisesta prosessoinnista, jossa yhdistettiin ohjelmakoodin aloitus- ja lopetus-lokiviestit. completed_tasks-muuttuja kuvaa dataputken vaihetta, jossa aloitus- ja lopetus-lokiviestistä on eristetty tarpeelliset tiedot, kuten ohjelmakoodin aloitus- ja lopetusaika, sekä näistä laskettu ajon kesto sekunteina. Tieto on myös muokattu table_schema_task_runtime-muuttujan merkkijonon mukaiseksi. Tämä muuttuja määrittää siis tallennettavan taulun tietomallin. Apache Beam -ohjelmakirjaston WriteToBigQuery-transformaatio ottaa argumenttina tallennettavan tiedon, tässä tapauksessa completed_tasks-muuttujan. Transformaation table-argumentti määrittää BigQuery-tietovarastokohtaisen polun, mihin ennalta luotuun GCP-projektiin ja datasettiin sekä datasetin tauluun valmis data tallennetaan. Tässä tapauksessa project- ja dataset-muuttujat on määritelty erikseen vastaamaan käytettyä konfiguraatiota. Create_disposition-argumentin avulla ohjeistetaan, että tallennettava taulu voidaan luoda ellei sitä jo ole olemassa. Write_disposition-argumentti määrittää tiedon tallennustavan. Tässä tapauksessa WRITE_APPEND-merkkijono ohjeistaa tallennuksen tapahtuvan lisäämällä rivin uudesta tiedosta tietovaraston taulun loppuun. Method-argumentilla määritetään tapa, jolla BigQuery-tietovarasto ja ajoalusta vaihtavat tietoa. STREAMING_INSERTS-merkkijonolla asetetaan yhteydelle asetukset, jotka parhaiten sopivat tiedon suoratoistoon, mikä pienentää latenssia tiedon valmistumisen ja tietovarastoon tallentamisen välillä.

Esimerkkikoodi 9:ssä esitettyä ohjelmakoodia voitiin toistaa jokaisen tallennettavan taulun kohdalla erikseen.

4.4.6 Dataputken suoritus Dataflow-palvelussa

Apache Beam -ohjelmointikirjastolla tehtyä ohjelmakoodia oli toistaiseksi ajettu lokaalisti, mikä tekee kehitysprosessista nopeampaa. Dataputken toteuttavan ohjelmakoodin ollessa käyttövalmis se tuli määrittää ajettavaksi Dataflow’n suoritusympäristössä.

```
pipeline_options = PipelineOptions(
```

```
  pipeline_args,
  streaming=True,
  project='elt-logging',
  job_name='elt-logging',
  runner='DataflowRunner',
  temp_location='gs://elt-logging/tmp',
  region='europe-north1'
)
with Pipeline(options=pipeline_options) as pipeline:
   …
```

Esimerkkikoodi 10. Dataputken käyttöönotto Dataflow-palvelussa.

Esimerkkikoodi 10:ssä määritellään asetukset, joiden avulla ohjelmakoodin suoritus saadaan siirrettyä Dataflow-palveluun. Pipeline_options-muuttujalle määritellään streaming-parametri, joka ohjeistaa dataputken toimivan datasetin kanssa, jonka kokoa ei tiedetä ennalta. Project- ja job_name-parametreilla asetetaan käytetty GCP-pilvialustan projekti ja ohjelmakoodille uniikki, Dataflow'ssa näkyvä tunniste. Dataflow-palvelussa ohjelmakoodin ajaminen vaatii ohjelmakoodin ajon ajaksi tallenuspaikan Python-ohjelmakoodille ja siihen liittyville kirjastoille, joka asetetaan temp_location-parametrin avulla. Tässä tapauksessa gs://-alkuinen polku tarkoittaa Cloud Storage -pilvipalvelua, joka on Googlen ylläpitämä verkkotallennustila. Region-parametrilla määritetään, missä Googlen pilvipalveluiden fyysisessä ympäristössä ohjelmakoodi tullaan ajamaan. Esimerkkikoodissa oleva europe-north1-alue viittaa Suomessa oleviin palvelimiin. Lopuksi luotu asetusobjekti, pipeline_options, syötetään argumenttina dataputken luontiin.

5 Yhteenveto

Insinöörityössä käytiin läpi, kuinka käyttää moderneja tiedon käsittelyn työkaluja ja pilvipalveluita luomaan virheensietokykyinen ja tiedon määrän mukaan skaalautuva lokijärjestelmä, joka kerää tiedon lataamiseen ja prosessointiin käytettävien ohjelmakoodien ajoista, data-analytiikan maailmassa ELT-ajoiksi kutsutuista prosesseista hyödyllistä tietoa. Insinöörityön toimeksiantavan yrityksen toiminnan kannalta hyödyllisimmiksi tiedoiksi osoittautui ohjelmakoodien ajojen kestojen ja statusten seuranta, jotka saatiin tilallisen tiedon prosessoinnin avulla reaaliaikaisiksi. Muita hyödyllisiä seurattavia tietoja olivat ohjelmakoodin tallentaman tiedon määrä, josta lokijärjestelmä luo merkinnän tietovarastoon ses-

sioinnista aiheutuvan viiveen takia noin 10 minuuttia ohjelmakoodin ajon jälkeen. Näiden tietojen avulla voitiin alkaa seuraamaan, kuinka ohjelmakoodien ajojen kestot sekä käsiteltävän datan määrät muuttuvat ajan kuluessa ja ennaltaehkäisemään tilanteita, joissa ohjelmakoodien ajot kasvavat liian pitkiksi.

Projektissa päädyttiin käyttämään Googlen pilvipalveluympäristön, GCP:n, toiminnallisuuksia mahdollisimman hyvän yhteensopivuuden takaamiseksi. Pub/Sub-viestijono, Dataflow-palvelu ja BigQuery-tietovarasto loivat vakaan alustan, jonka päälle lokijärjestelmä voitiin luoda. Dataflow-palvelussa ajettava Apache Beam -ohjelmakirjastolla toteutettu lokijärjestelmän ohjelmakoodi ja sen ohjelmointimalli osoittautuivat projektin suurimmiksi haasteiksi. Jotkin lokijärjestelmän vaatimat toiminnot, kuten kahden toisiinsa liittyvän tapahtuman, tässä tapauksessa ohjelmakoodin alku- ja lopputapahtumien seuraaminen tilallisen tiedon prosessoinnilla, ei ollut täysin suoraviivaista. Toiminnon kehitys vaati pidempää perehtymistä ohjelmakirjaston toiminnallisuuksiin ja kehittämään hieman logiikaltaan erikoista ohjelmakoodia, sekä manuaalista muistin roskienkeruun kehittämistä, joka johti pohdintaan siitä, olisiko ohjelmakirjasto soveltuva tätä tehtävää monimutkaisempaan tiedon prosessointiin. Toisaalta kehityksen aikana tehdyn arvion mukaan Apache Beam -ohjelmointikirjaston aikaan liittyvät toiminnallisuudet rajata tietoa aikaleimojen avulla osoittautui käytännölliseksi ja hyvin toimivaksi tavaksi käsitellä suoratoistona saapuvaa, viiveitä sisältävää tietoa.

Lähteet

1. Sherif Sakr, Albert Y. Zomaya. 2019. Encyclopedia of Big Data Technologies. Springer.

2. Google. 2022. Dataproc. Verkkoaineisto. <https://cloud.google.com/dataproc>. Luettu 13.7.2022.

3. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing, Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing.

4. Google. 2022. Dataflow. Verkkoaineisto. <https://cloud.google.com/dataflow>. Luettu 14.7.2022.

5. The Apache Software Foundation. 5.2.2021. Beam Capability Matrix. Verkkoaineisto. <https://beam.apache.org/documentation/runners/capability-matrix>. Luettu 20.7.2022.

6. Google. 2022. What is Pub/Sub? Verkkoaineisto. <https://cloud.google.com/pubsub/docs/overview>. Luettu 18.7.2022.

7. Free Software Foundation. 21.12.2020. Pipelines. Verkkoaineisto. <https://www.gnu.org/software/bash/manual/html_node/Pipelines.html>. Luettu 10.8.2022.

8. The Apache Software Foundation. 2022. Apache Beam Programming Guide. Verkkoaineisto. <https://beam.apache.org/documentation/programming-guide>. Luettu 11.7.2022.