Jani Sourander

Delta Lake tietovarastona

Tradenomi

Tietojenkäsittely

Syksy 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Sourander Jani

Työn nimi: Delta Lake tietovarastona

Tutkintonimike: Tradenomi (AMK), Tietojenkäsittely

Asiasanat: Apache Spark, Delta Lake, tietoallas, tietovarasto

Tässä opinnäytetyössä keskitytään tietovaraston ja tietoaltaan yhdistelmän eli data lakehouse -arkkitehtuuriin ja sen käyttöönottoon. Opinnäytetyön taustalla on Polar Electro Oy:n tietovarastouudistus, jota minä olen ollut toteuttamassa data engineerinä. Tiimi, jossa työskentelen, vertaili palveluita ja arkkitehtuureja: korvasimme edeltävän tietovaraston modernilla ratkaisulla, jonka toteutusympäristönä on Databricks Lakehouse Platform -tietoalusta. Opinnäytetyössä esitellään alan keskeisimmät käsitteet ja tietovarastoinnin historian merkittävimmät vaiheet.

Työn produktiivisessa osiossa esitellään tietoalustassa ajettava koodi sekä luodaan Delta Lake -formaattiin perustuva medaljonkiarkkitehtuurin mukainen tietoaltaan pronssikerros. Esimerkeissä käytetään keinotekoista dataa, joka simuloi muodoltaan AWS Database Migration Service -migraatiopalvelun lataamia relaatiotietokannan tauluja. Tieto siirretään lastauslaiturilta tietoaltaaseen Apache Spark -ohjelmistolla, jota käskytetään ja orkestroidaan Pythonilla. Osion esimerkkikoodit on testattu tavallisella kannettavalla tietokoneella avoimen lähdekoodin toteutuksena. Opinnäytetyön vaiheiden toisintaminen ei vaadi maksullisen tietoalustan käyttöönottoa. Työssä esitellyt käyttökokemukset tuotantodatan kanssa perustuvat kuitenkin Databricks Lakehouse Platform -alustan käyttöön.

Kokonaisuutena tietovarastouudistus täyttää sille asetetut vaatimukset ja on onnistunut. Uusi tietoalusta on sekä nopeampi että edullisempi kuin edeltäjänsä, toimittajaloukun riski on matalampi avoimen lähdekoodin ratkaisuiden takia, uusia datalähteitä on helpompi lisätä kuin edeltäjäänsä ja alusta mahdollistaa koneoppimisen käyttöönoton matalammalla kynnyksellä kuin edeltäjänsä. Heikoiten täyttynyt kriteeri uuden alustan suhteen on sen maturiteetti: useat ominaisuuksista ovat yhä kehitysvaiheessa. Yhä tuoreen data lakehouse -arkkitehtuurin merkittävyyttä on turhan aikaista arvioida, mutta se esitellään yhtenä mahdollisena jatkeena tietovarastoinnin historialle.

KAMK • University of Applied Sciences

Abstract

Author(s): Sourander Jani

Title of the Publication: Delta Lake as a data warehouse

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: Apache Spark, Delta Lake, data lake, data warehouse

This thesis aims to introduce a data management architecture called the data lakehouse and demonstrate an example deployment in production. The architecture combines functionalities between the data warehouse and data lake. The process of studying the tools and trends in the modern data warehousing ecosystem was set in motion when my employer Polar Electro Oy had a need to modernize the company's existing data warehouse stack. The data team in which I work in was assigned with this modernization task. Our team compared and benchmarked various products and architectures: the chosen product is Databricks Lakehouse Platform.

After explaining the key terminology and the brief history of data warehousing and data lakes, the thesis introduces a demo implementation of the lakehouse architecture utilizing an open-source Apache Spark processing engine and open-source data storage layer Delta Lake. The process starts by generating an artificial dataset that simulates staging data written by AWS Database Migration Service, assuming that the original data source is a relational database server. The staging data contains the full load data and the continuously appended change capture data. The full load data are transferred to bronze layer of the medallion architecture. After this, the change capture data are merged into the bronze layer using scheduled incremental tasks written in Python. Replicating these steps does not require having access to the proprietary Databricks platform. However, the in-production experiences shared in the thesis rely in my experiences with the commercial platform.

Overall, the data warehouse modernization project was a success. The new platform is both faster and cheaper than its predecessor. Other met criteria were low risk of vendor-lock, low complexity in adding more data sources, and the low threshold of running machine learning trials. However, the maturity of the platform remained a possible concern. Many of the key functionalities of the platform were still in public preview and thus are officially labeled as not fully mature. It is still too early to evaluate the global significance of the data lakehouse architecture, but the thesis introduces it as a possible step in the future history of data warehousing.

Sisällys

Symboliluettelo

ACID	Neljän tekijän (atomisuus, eheys, eristyvyys ja pysyvyys) kokonaisuus. ACID-periaatteita noudattava kanta on ihanteellisessa maailmassa virheenkestävä jopa vikatilanteissa tai suorittaessa useita kilpailevia kirjoitusoperaatiota yhtä aikaa.
CDC	Muutostiedon kaappaus (englanniksi change data capture). Prosessi, jossa tunnistetaan lähdekannan muutokset joko lähes reaaliaikaisesti tai ajastetusti.
Data Lakehouse	Arkkitehtuuri, joka yhdistää tietovaraston (eng. data warehouse) sekä tietoaltaan (eng. data lake) toiminnallisuuksia.
Delta Lake	Avoimen lähdekoodin tiedontallennuksen kerros, joka lisää ACID-periaatteiden mukaisen toiminnallisuuden tiedostopohjaiseen tietoaltaaseen.
ELT	Alemman akronyymin eli ETL:n mukaelma, jossa tiedon muokkaus ja tiedon lataus kohdekantaan suoritetaan käänteisessä järjestyksessä. Tietomallia muokataan vasta määränpäässä, joka voi olla esimerkiksi tietovarasto tai jokin moderni tietoalusta.
ETL	Prosessi, jossa tieto kaapataan lähdekannasta (extract), muunnetaan kohdekannan tietomallin mukaiseksi (transform) ja ladataan kohdekantaan (load), joka on tyypillisesti tietovarasto.
Tietoallas	Suurten tietomassojen tallennukseen tarkoitettu arkkitehtuuri. Mahdollistaa strukturoimattoman tiedon tallennuksen ja käsittelyn. Tallennuskapasiteetti on usein hajautettua ja tiedostopohjaista, mikä mahdollistaa edullisen horisontaalisen skaalauksen. Englanniksi data lake.
Tietovarasto	Järjestelmä, johon tuodaan ETL tai ELT-prosessin avulla useista eri tietokannoista tietoa, jotta hajallaan olevaa dataa voidaan käsitellä ja analysoida keskitetysti. Englanniksi data warehouse.

1 Johdanto

Kun aloitin kokoaikaisen työni Polar Electro Oy:ssä elokuussa 2020 tittelillä data engineer, yrityksellä oli käytössään tietovarasto, joka ei täyttänyt loppukäyttäjien odotuksia. Tietovarasto vastasi sen oletetusta perustehtävästä eli siihen tuotiin joka yö operatiivisista tuotantokannoista dataa, joka kirjoitettiin analytiikan tarpeita vastaavaan tietomalliin. Loogisella tasolla järjestelmä kykeni vastaamaan kysymyksiin kuten: kuinka monta treeniä tänä vuonna on tehty hiihtäen verrattuna viime vuoteen? Vastauksen palautumisessa kesti kuitenkin kauemmin kuin loppukäyttäjät toivoivat. Hakujen nopeuttaminen laskentatehoa skaalaamalla ei ollut kustannustehokasta.

Oli siis selkeää, että uudelle järjestelmälle olisi tilausta. Järjestelmälle asetettuja tavoitteita olivat muun muassa: kustannustehokkuus verrattuna edelliseen, uusien datalähteiden lisäämisen helppous, tuotteen maturiteetti, toimittajaloukun riskin vähyys, modulaarisuuden aste ja sopivuus mahdollisiin tulevaisuuden tarpeisiin kuten koneoppimisen hyödyntämiseen. Useat näistä ominaisuuksista liittyvät löyhästi määriteltyyn kokonaisuuteen, joka kulkee kirjoissa nimellä *modern data stack*.

Minä ja kollegani tutustuimme kahden hengen tiiminä modernin datankäsittelyn muotisanojen takana oleviin teknologioihin. En avaa vertailuprosessia yksityiskohtaisesti tässä opinnäytetyössä tai paljasta edellisen tietovaraston teknisiä ratkaisuja. Sen sijaan esittelen valituksi tulleen työkalun nimeltään The Databricks Lakehouse Platform ja esitän, kuinka se täyttää vaaditut kriteerit. Tavoiteltu kokonaisuus ei ole tietovarasto vaan pikemminkin tietoallas, joka näyttäytyy loppukäyttäjälle tietovarastona.

2 Tietovarastojen perusteet ja historia

Ralph Kimball [1] jakaa tietokannat kahteen luokkaan käyttötarkoituksen mukaan: operatiivisiin kantoihin ja tietovarastoihin (eng. data warehouse). Operatiiviset transaktiokannat käsittelevät tilauksia, varauksia, ottoja ja muita tilamuutoksia pääasiassa rivi kerrallaan. Nämä tuotannon kannalta kriittiset kannat eivät tyypillisesti tallenna historiadataa vaan edustavat datan nykytilannetta. Tietovarastot toimivat hyvin päinvastaisesti: tietokannan haut käsittelevät useita tuhansia tai miljoonia rivejä kerrallaan, ja haettu tieto edustaa esimerkiksi keskimääräistä varausmäärää per tietty asiakassektori. Näille tiedon käsittelyn malleille on vakiintuneet lyhenteet: tietovarastokäyttöä vastaa lyhenne OLAP (Online Analytics Processing), ja operatiivisten kantojen käyttöä OLTP (Online Transaction Processing). Mallien eroavaisuudet on esitelty alla olevassa taulukossa (ks. Taulukko 1). Taulukon tiedot on yhdistetty Geeks for Geeks -sivuston kirjoituksesta [2], ja Usage-Driven Database Design [3] sekä Designing Data-Intensive Applications [4] -kirjojen vastaavista taulukoista.

	OLTP	*OLAP*
Toiminta-ajatus	Tallentaa ja palauttaa tietueita yrityksen päivittäisen toiminnan takaamiseksi.	Tukea päätöksentekoa ja mahdollistaa datan louhinnan ja analysoinnin.
Tyypillinen haku	Loppukäyttäjä sovelluksen käyttöliittymän kautta. Pieni määrä rivejä, jotka löydetään primääriavaimella.	Data-analyytikko SQL-kyselyllä. Usein aggregoitua dataa, jota on suodatettu tai ryhmitelty sarakearvojen mukaan.
Tyypillinen kirjoitus	Yksittäiseen riviin kohdistuva kirjoitus tai muutos.	Suuri erä rivejä ajastetussa ETL-prosessissa.
Kyselyn laajuus	Yksittäinen tietue eli rivi, joka haetaan sen id:tä tai avainta vasten.	Useita tietueita tai niiden aggregaatti (esimerkiksi keskiarvo).
Viive	Nano- tai millisekunteja	Sekunteja tai tunteja

Taulukko 1. OLTP- ja OLAP-vertailu.

On tärkeää huomioida, että OLTP tai OLAP eivät edusta mitään tiettyä tietokantateknologiaa vaan ovat pikemminkin tiedon hakemisen malleja. Tieto voi olla loogisesti ja fyysisesti tallennettu mallista riippumatta eri formaatteihin. Looginen malli voi olla esimerkiksi relaatiotietokannan taulu, dokumentti tai graafi. Tietokantajärjestelmä voi olla esimerkiksi MySQL, MongoDB tai Neo4j. Tietokantojen alkuaikoina tyypilliset operaatiot olivat transaktioita eli kaupankäynnin tapahtumia; termi on jäänyt käyttöön sellaisenaan, vaikka nykyisin yksittäinen tietokannan taulun rivi voi edustaa ei-kaupankäynnillistä entiteettiä.

2.1 Tietokantojen ja tietovarastojen tietomallit

Operatiivisten tuotantokantojen tietomalli rakennetaan sovelluksen tarpeen mukaan, joten kahden eri tuotantokannan tietomallit voivat olla keskenään hyvinkin poikkeavia. Tyypillisin tietomalli on relaatiotietokanta, jossa data on tallennettu tauluihin ja tietokanta noudattaa vähintään ensimmäisen asteen normaalimuotoa, mutta käytössä on jatkuvasti enenevissä määrin muita tietokantatyyppejä, kuten dokumentti- tai graafitietokantoja. Näiden relaatiotietokannoista poikkeavien kantatyyppien yleisnimenä on NoSQL. Tietovarastojen tietomallien diversiteetti on sen sijaan huomattavasti kapeampi; tietovarastot noudattavat lähes poikkeuksetta tähtiskeemaa ja mukailevat Kimballin dimensiomallia. Tietovarastojen tietomalli rakennetaan yrityksen prosessien ja tietovaraston käyttäjien tarpeiden mukaan. Näin ollen operatiivisten kantojen ja tietovarastojen tietomallit rakennetaan hyvin erilaisista lähtökohdista; kummassakin on sama informaatio, mutta eri tavoin järjesteltynä. [4.]

Tietomallinnus ja erilaiset tietokantatyypit ovat omat, laajat aiheensa, eikä niihin syventyminen ole tämän opinnäytetyön tehtävä. Tämän opinnäytetyön keskeisenä tutkittavana ongelmana on tietoaltaan käyttö tietovarastona. Seuraavat luvut käsittelevät lähdekantoja sillä oletuksella, että ne ovat normalisoituja relaatiotietokantoja. Tietovarastossa nämä kannat ovat tietomallinnettu loppukäyttäjää varten siten, että ne noudattavat denormalisoitua tähtiskeemaa, joka koostuu fakta- ja dimensiotauluista. Normalisoinnin ja denormalisoinnin tai tähtiskeeman syvempi käsittely on tämän opinnäytetyön skoopin ulkopuolella. Selvennän tässä kappaleessa käsitteet tiivistetyssä muodossa siten, kuten koen niiden olevan aiheen kannalta välttämättömiä.

Tietokantojen normalisointiprosessin keksi IBM:n tutkija E.F. ”Ted” Codd 1970-luvun alussa. Relaatiotietokantamallin peruskäsitteitä ovat entiteetit (eng. relation, entity) ja niiden väliset suh-

teet (eng. relationships). Codd havaitsi, että normalisoimattomat taulut johtavat helposti ongelmiin, joita hän kutsui anomalioiksi. Näiden vaikutus on, että tietoa voi olla mahdotonta lisätä, muokata tai poistaa siten, että operaatio rajautuisi vain yhden entiteetin tietoihin. Tästä johtuen tieto normalisoidaan eli hajautetaan useisiin tauluihin ja taulujen välisiä suhteita kuvataan avaimilla ja viiteavaimilla. Yksittäisen entiteetin tai taulun sarakkeet eivät sisällä riippuvuuksia toisiinsa; tiedolla on suora riippuvuus vain ja ainoastaan taulun primääriavaimeen, joka on useimmiten keinotekoinen, juokseva kokonaisluku. Mikäli käyttäjä haluaa tulostaa esimerkiksi ostotapahtumasta kuitin, tietoa pitää hakea ja yhdistää useista eri tauluista, kuten: invoice, invoice_line_item, product, customer. [5.]

Tietovarasto ei hyödy normalisoinnista, koska sen käyttötarkoitus ei ole palvella sovellusta tai käyttäjää nopeilla rivikohtaisilla luku- ja kirjoitusoperaatioilla. Tietovarastoon tuotu data muokataan toisenlaiseen tietomalliin: tähtiskeemaan [6]. Tähtiskeema koostuu fakta- ja dimensiotauluista. Faktataulut mallintavat yrityksen prosesseja. Faktataulu on nimeltään esimerkiksi `fact_sales`, jonka yksittäinen rivi on myyntioperaatio tai kuittirivi sisältäen tuotteen senhetkisen hinnan ja alennushinnan. Tähän liittyvä dimensiotaulu on esimerkiksi `dim_product`, jonka yksittäinen rivi kuvastaa tuotetta, tai `dim_customer`, jonka yksittäinen rivi kuvastaa asiakasta. Yhteen faktatauluun liittyy monta dimensiota. Näin syntyy tähti, jossa keskiössä on faktataulu, ja tähden sarakkeet ovat dimensiotauluja. Tähtiskeema sisältää redundanttia, toistuvaa tietoa, joten taulut ovat fyysisesti (tavuina mitattuna) huomattavasti suurempia kuin mitä niiden alkuperäiset tietolähteet eli normalisoidut relaatiotaulut. Dimensiotaulut ovat usein myös hyvin leveitä: niissä voi kymmeniä tai satoja sarakkeita. Denormalisoitu dimensiotaulu saa ja voi sisältää toisistaan riippuvia sarakkeita, kuten: syntymäpäivä ja ikä; pituus, leveys ja pinta-ala; päivämäärä ja viikonpäivä. Normalisoidussa kannassa nämä tietoparit nähtäisiin redundantteina, koska niillä on riippuvuus; yhden arvon voi laskea toisesta. [4.]

Tähtiskeeman tarjoama hyöty on OLAP-hakujen helppous. Esimerkiksi kaikkien tammikuussa rekisteröityjen tuotteiden laskeminen väreittäin ei vaadi kuin yhden JOIN-lausekkeen. SQL-esimerkki (ks. Koodi 1), jossa kyseinen haku on toteutettu, palauttaa rekisteröityjen laitteiden värit (ks. Taulukko 2). Esimerkissä oletetaan, että kuvitteellisesta faktataulusta löytyy sarake `par_yyyymm`. Partitiointi, mikä selittäisi kyseisen sarakkeen olemassaolon, käsitellään myöhemmin tässä opinnäytetyössä.

```
SELECT
      dim_product.color,
      COUNT(*) AS num_regs
FROM fact_registrations
      JOIN dim_product
        ON fact_registrations.product_key = dim_product.product_key
WHERE
      fact_registrations.par_yyyymm = 202101
GROUP BY
      dim_product.color
```

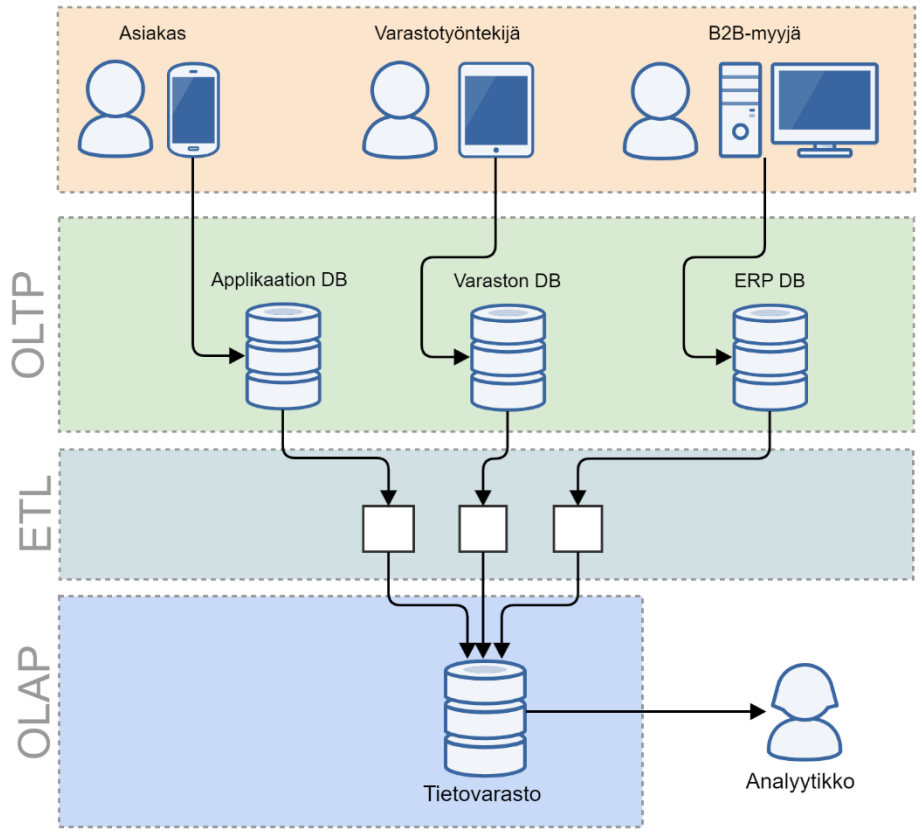
Koodi 1. Kuvitteelliseen tietovarastoon kohdennettu SQL-kysely, jossa lasketaan tammikuussa 2021 rekisteröidyt tuotteet väreittäin.

color	num_regs
Red	127543
Blue	135001
Green	125205

Taulukko 2. SQL-kyselyn kuvitteellinen vastaus

2.2 Tietovarastoteknologioiden lyhyt historia 1960-luvulta 2000-luvulle

Aikoinaan, ennen 1980- ja 1990-lukujen vaihdetta, samoja kantoja käytettiin sekä operatiiviseen käyttöön että datan analysointiin. Mikäli johtoa kiinnosti kokonaismyynti tammikuussa, data haettiin samasta tuotantokannasta, mihin se oli kirjoitettukin. Kyselykieli oli sekä OLTP- että OLAP-kyselyissä yleisimmin SQL. Datan määrän sekä analytiikan tarpeen kasvaessa yritykset alkoivat integroida tuotantojärjestelmien dataa keskitettyyn varastoon, joka on optimoitu OLAP-käyttöä varten. Näistä järjestelmistä käytetään nimeä tietovarasto (eng. data warehouse). Tietovarasto sisältää siis yhden tai useamman tuotantokannan dataa. Eri tuotantokannat voivat olla eri tietokantajärjestelmiä; sisään ladatessa tieto käännetään tietovaraston ymmärtämään formaattiin. Tämän prosessin eri vaiheista käytetään termejä kaappaus (eng. extract), muunnos (eng. transform) ja lataus (eng. load) – näiden yhdistelmästä käytetään lyhennettä ETL. Muunnoksella viitataan nimenomaan tietomallin muunnokseen (lue lisää luvusta Tietokantojen ja tietovarastojen tietomallit). Datan kulku järjestelmän komponentilta toiselle on kuvattuna visuaalisena kaaviona alla, ja se koostuu kuvitteellisen yrityksen kolmesta eri tietokannasta ja niiden ensisijaisista datan tuottajista. [4.]



Kuva 1. Datan kulku tiedon tuottajilta analyytikoille.

Inmon, Lindstedt ja Levins [7] jakavat nykyistä big data -aikakautta edeltävät vaiheet kuuteen aikakauteen. Ensimmäinen vaihe on 1960-luvua edeltävä aika, jolloin käyttöjärjestelmien ja ohjelmointikielien runsaus ja standardien puute edusti kaaosta. Seuraava aikakausi, 1960–1970, oli IBM 360 -suurtietokoneiden ja saman yrityksen tietokannan hallintajärjestelmän, IMS:n, valta-aikaa. Kolmannen aikakauden eli 1970- ja 1990-lukujen välisenä aikana operatiiviset transaktiokannat mahdollistivat yrityksille nykypäivänä itsestäänselviä prosesseja, kuten käteisnoston pankkiautomaatilla tai lennon varauksen. 1990-luvulla, neljäntenä aikakautena, IBM:n haastaja eli Teradata-yhtiö mahdollisti MPP:n (eng. massively parallel processing) eli massiivisen rinnakkaislaskennan. Viimeiset kaksi aikakautta keskittyvät kumpikin Hadoop-teknologiaan: 2000–2005 oli avoimen lähdekoodin Hadoopin syntyaikaa, kun taas 2005 vuodesta nykypäivään kestäneenä kautena IBM on Inmonin ja kumppaneiden kirjoittaman *”Data Architecture: A Primer for the Data Scientist”*-kirjan [7] mukaisesti mennyt Hadoop-teknologian reppuselässä. Kirja ei mainitse muita yrityksiä kuin IBM:n, mutta Apache Hadoop -pohjaisia ratkaisuja valmistivat myös esimerkiksi Cloudera ja Hortonworks [8]. Fishtown Analyticsin Tristan Handy tiivistää modernin tietoinfraan (eng. modern data stack) nykyisen vaiheen alkaneen 2012 Amazon Redshiftin myötä [9]. Hän listaa kolme vaihetta, jotka ovat leikkisästi nimetyt ensimmäisen kambrikauden räjähdys 2012–

2016, käyttöönottokausi 2016–2020, ja hänen ennustelmiensa mukaan toinen kambrikauden räjähdys 2020–2025. Siinä missä Inmon ja kumppanit tiivistävät 2000-luvun keskittyvän Hadoopiin, Tristan ei mainitse Hadoopia sanallakaan kirjoituksessaan, mutta mainitsee Snowflaken sekä Google BigQueryn sekä useita muita tietoinfraan liittyviä tuotteita kuten Looker, Fivetran, Stitch, Redash ja heidän oma tuotteensa eli dbt.

2.3 Hadoop-ekosysteemi

Internetin käytön lisääntymisen myötä eli datan määrän kasvaessa syntyi tarve entistä suuremmalle laskentakapasiteetille, mikä loi tarvetta sekä arkkitehtuurien että teknologioiden uudistamiselle. Hadoop sai alkunsa, kun Doug Cutting ja projektiin liittynyt Mike Cafarella työskentelivät Apache Nutch -hakukoneen parissa. Projektilla oli tarve laskentaklusterin hallintaohjelmalle, joka on vikasietoinen, osaa tasata kuormaa eikä kadota dataa, vaikka yksittäiset klusterin työkoneet lakkaisivat toimivasta. Juuri tällöin, vuonna 2003, Google julkaisi 15-sivuisen artikkelin heidän GFS:n (Google File System) toiminnastaan. Tähän julkaisuun pohjautuen Cutting ja Cafarella aloittivat NDFS:n (Nutch Distributed File System) parissa työskentelyn, joka valmistui 2004. Google julkaisi samana vuonna uuden artikkelin otsikolla MapReduce: Simplified Data Processing on Large Clusters. Cutting ja Cafarella, jotka työskelivät tähän aikaan Yahoo!:ssa, integroivat Googlen MapReduce-idean heidän Nutch-projektiinsa, ja luovuttivat sen voittoa tavoittelemattomalle Apache Software Foundationille. Näin sai alkunsa Apache Hadoop -kehys, johon kuuluvat komponentit Hadoop Common, MapReduce, HDFS (Hadoop Distributed File System) sekä myöhemmin Apache Hadoop YARN (Yet Another Resource Negotiator). Lopputuloksena oli järjestelmä, joka hajauttaa dataa useille koneille ja suorittaa laskennan lokaalisti siellä, missä data on. [10.]

Hadoop ei ole tietokanta vaan rinnakkaisen laskennan ekosysteemi, johon kuuluvat tiiviisti sekä datan tallennus että prosessointi – HDFS sekä MapReduce. Hadoopia voi kuitenkin käyttää tietovarastona, kuten Facebook ja AOL ovat tehneet [11]. MapReduce aloitti aikakauden, jolloin tietovaraston pystyy rakentamaan tavallisten tiedostojen varaan. Tuotantokantojen ja tietovarastojen tekniset toteutukset ovat erkaantuneet toisistaan 1990-luvulta lähtien, mutta loppukäyttäjän näkökulmasta ero on pysynyt silti maltillisena, sillä SQL-kieli on pitänyt vahvasti asemansa kyselykielenä sekä OLTP- että OLAP-käytössä. HDFS-tiedostojärjestelmässä lojuvia irrallisia tiedostoja on työläs analysoida, mikäli niiden päälle ei rakenna tietomallia, joka mahdollistaa samat SQL-kyselyt kuin edellisen sukupolven tietovarastot.

2.4 Hadoopista Apache Sparkiin

Hadoop MapReducen käyttö on vaikeaa, kuten Learning Spark -kirja vahvistaa [8]. Kirjan mukaan MapReducen monimutkaisuus johti useisiin rinnakkaisprojekteihin, kuten Apache Hive, Apache Giraph, Apache Drill ja Apache Mahout, joista jokainen vaati omat käyttöliittymänsä ja konfiguraationsa. UC Berkeleyn opiskelijat, joiden joukossa oli Databricksin perustajiin kuuluva ja nykyinen toimitusjohtaja Ali Ghodsi, aloittivat projektin, jonka tavoitteena oli MapReducen nopeuttaminen. Projektin nimi oli Spark. Varhaisten julkaisuiden aikaan Sparkin todistettiin suoriutuvan samoista tehtävistä 10–20 kertaa nopeammin kuin Hadoop MapReduce. Tämänhetkinen Apache Sparkin versio, 3.x, mainostaa etusivullaan yli 100-kertaista nopeutta Hadoop MapReduceen verrattuna.

Täysin suora vertailu Hadoopin ja Sparkin välillä on vaikeaa tai jopa epärelevanttia, sillä niiden toimintatavat poikkeavat huomattavasti toisistaan [12]. Spark täydentää olemassa olevaa Hadoop-ekosysteemiä komponentilla, joka toimii kuten MapReduce, mutta jossa laskenta ja tiedon tallennus on eriytetty. Hadoopissa HDFS ja MapReduce, eli tallennus ja laskenta, ovat tiukasti yhteensidotut. Hadoop-laskenta lukee tietoa tiedostojärjestelmästä ja kirjoittaa tulokset takaisin tiedostojärjestelmään. Spark on pelkkä tiedon prosessori, joten pysyvä data voidaan ladata useista eri lähteistä, joita ovat esimerkiksi Hadoop (HDFS), Amazon S3, Apache Hive, MariaDB, Apache Cassandra. Toisin kuin vahvasti levyjärjestelmään tukeutuva Hadoop, Spark nojaa keskusmuistin kapasiteettiin ja nopeuteen (eng. in-memory computing). Spark kirjoittaa dataa levylle vain, mikäli keskusmuisti loppuu kesken: tätä sanotaan tiedon vuotamiseksi levylle (eng. disk spill) ja sitä tulisi välttää optimoimalla sekä kyselyitä että laskentaklusterin kokoa ja tyyppiä [13].

Hadoop-ekosysteemin ratkaisut ovat tehokkaita säilömään ja liikuttamaan tera- tai petatavuittain dataa [14]. Toisin kuin relaatiotietokantojen kanssa, datan kirjoittajan ei tarvitse tuntea kohdetaulun skeemaa eli sarakkeiden nimiä ja tietotyyppejä – jos kohdetta voi edes kutsua tauluksi. HDFS:ään voi kirjoittaa tiedoston missä tahansa formaatissa, joten ymmärrys skeemasta on lukijan vastuulla. Näitä big data -tiedostosäilöjä kutsutaankin nimellä tietoallas (eng. data lake), ja niiden vahvuutena tietovarastoihin verrattuna on niiden kyky tallentaa ei-kuratoitua raakadataa sekä tallennuskapasiteetin matala hinta [15]. Tietoallas voi olla myös pilvipalvelussa, kuten Amazon S3 -objektisäiliössä, joka on yksinkertainen avain-arvo-säilö. Tietoaltaaseen ladattu tiedosto voi olla esimerkiksi skeeman sisältävä Parquet-tiedosto, osittain jäsennelty JSON-tiedosto, JPEG-kuvatiedosto tai täysin vailla struktuuria olevaa sensoridataa.

2.5 Apache Spark tuotteistettuna

Vuonna 2013 Spark-projektin jäsenet luovuttivat Sparkin Apache Software Foundationille ja perustivat Databricks-yrityksen, jonka päätuote on data-analytiikan alusta, joka käyttää Sparkia datan prosessointiin. Databricksin muihin projekteihin kuuluu myös Delta Lake -formaatti. Delta Lake on avointa lähdekoodia, mutta Databricksin Spark-ympäristöön kuuluu kaupallinen elementti Delta Engine [16]. Kun ensimmäisen kerran tutustuin Databricksin työkaluihin loppuvuodesta 2020, heidän alustansa oli yhä nimeltään Databricks Unified Analytics Platform [17]. Työkalu yhdensi data-analytiikan ja data engineering -alueen tehtäviä saman alustan alle, ja Lakehouse on ollut esiteltynä sekä blogissa että heidän YouTube Tech Talk -videoissaan. En tiedä, minä päivänä alusta on saanut uuden nimensä, mutta kirjoitushetkellä Databricksin sivusto kutsuu heidän alustaansa nimellä The Databricks Lakehouse Platform. Kokemukseni on, että kuluneen vuoden aikana Lakehouse-arkkitehtuuri on kypsynyt altavastaajasta yhdeksi merkittävimmistä arkkitehtuureista. Tähän viittaa muun muassa se, että Bill Inmon, jonka nimi esiintyy useissa lukemissani tietovarastoinnin ja esimerkiksi Data Vault 2.0-mallinnuksen teoksissa joko kirjoittaja tai viitteenä, on juuri opinnäytetyön valmistumisen kynnyksellä julkaissut kirjan Data Lakehouse -arkkitehtuurista (18). Monet käyttämistäni alustan työkaluista ja toiminnoista ovat yhä ”Public Preview”-tilassa ja esimerkiksi käyttäjien ”Single Sign-On”-autentikoinnin aktivointi ei onnistunut ilman asiakaspalvelun apua. Databricksin osalta maturiteetti on johdannossa esitellyistä vaatimuksista heikommalla tasolla kuin useilla kilpailijoilla.

Delta Lake -formaattiin nojautuvan Lakehouse -arkkitehtuurin voi ottaa käyttöön maksamatta Databricksin Lakehouse Platformista, mutta Databricksin asiakkaat pääsevät hyötymään Delta Enginen tarjoamista lisäominaisuuksista, kuten z-ordering, auto loader tai auto optimizer, joita ei ole toteutettu avoimen lähdekoodin Spark-moottorissa, sekä Databricksin muista ominaisuuksista kuten käyttäjähallinnan mahdollistavista Table Access Managed -palvelimista. Data itsessään pysyy kaiken aikaa avoimen lähdekoodin formaatissa eikä poistu Polarin omasta tietoaltaasta: tämä mahdollistaa matalan toimittajaloukun (eng. vendor-lock). Toimittajaloukulla tarkoitetaan tässä asiayhteydessä vakavaa riippuvuutta yksittäisen palveluntarjoajan tuotteesta.

Maturiteettihuolista riippumatta The Databricks Lakehouse Platform osoittautui Polarin käyttötarkoituksessa verrokkejaan tehokkaammaksi työkaluksi. Kilpailijoista merkittävimmät olivat Databricks, Snowflake ja Redshift, mutta tarkastelussa oli mukana myös muita toimijoita kuten Starburst ja Dremio. Tämä ei kuitenkaan osoita Databricksin suoraa paremmuutta: jokainen yritys joutuu punnitsemaan omat käyttötarkoituksensa ja tarpeensa. Polarin kriteeristöllä esimerkiksi

uusien datalähteiden lisääminen oli huomattavan paljon helpompaa Databricksin alustalla kuin kilpailijoilla; varsinkin jos lähteenä toimivat Parquet-tiedostot. Yhteistä kaikille alustoille on se, että ne mahdollistavat Hadoop-tyylisen laskennan ja tallennuksen erottamisen toisistaan – jopa Redshift pystyy siihen nykyisin. Eroavaisuuksia on muun muassa siinä, että tallennetaanko data asiakkaan vai palveluntoimittajan tietoaltaaseen, missä tiedostoformaatissa tieto on, voiko tietoa lukea muilla kuin valitulla työkalulla, ja että millaisen käyttöliittymän tai API:n läpi palveluun tehdään kutsuja.

2.6 Data Lakehouse

Lakehouse-arkkitehtuuri, joka on yhdistelmä sanoista data warehouse ja data lake, pyrkii nimensä mukaisesti yhdistämään tietovarastojen ja tietoaltaiden ominaisuudet [19]. Tieto tallennetaan yhä matalan kustannuksen tietoaltaaseen, kuten Amazon S3:een, mutta tieto kirjoitetaan ennalta määrätyssä formaatissa. Databricksin näkemys Lakehouse-arkkitehtuurista tai -paradigmasta hyödyntää Delta Lake -kerrosta, joka on Databricksin perustama avoimen lähdekoodin The Linux Foundation -projekti. Delta Lake ei itsessään ole tiedostoformaatti: se on tiedon tallennukseen ja lukemiseen osallistuva kerros (eng. storage layer), jonka fyysinen toteutus on tietoaltaaseen luotu hakemisto nimeltään `_delta_log/` sisältöineen; käytännössä kansio on WAL-tyylinen transaktioloki (eng. write-ahead log). Tieto itsessään tallennetaan Parquet-tiedostoformaatissa. Delta Engine ja Sparkin natiivimoottori kirjoittavat keskenään yhteensopivia tiedostoja. Databricksin asiakkaat eivät siis kirjoita tiedostoja maksumuurin takana olevaan formaattiin vaan avoimen lähdekoodin tiedostoformaattiin. Kirjoittamista optimoivat ja muut moottorin ominaisuudet eivät silti välttämättä sisälly ilmaisversioon.

Snowflaken työntekijän, Jeremiah Hansenin, kommentit Lakehouse-arkkitehtuurista ovat vähemmän mairittelevia. Hänen mukaansa Lakehouse on paljon melua tyhjästä, ja hän todistaa kuvien kera, että termin Data Lakehouse tiettävästi ensimmäinen käyttökerta on vuodelta 2017, ja nimenomaan Snowflaken asiakkaan suusta [20]. Polarin käyttöön Lakehouse tuntui kuitenkin istuvan. Pidimme auki hyvin myöhäiseen vaiheeseen asti vaihtoehdon, jossa kasaisimme tietoaltaan ja suorittaisimme esimerkiksi koneoppimistehtävät ja datan esikäsittelyn Databricksin palvelun avulla tietoaltaassa, ja jatkaisimme tietoallasta erillisellä tietovarastolla, joka olisi mahdollisesti ollut Snowflake. Toistaiseksi tälle ei kuitenkaan ole ollut tilausta. Databricksin työkalujen sekä niiden dokumentaation hioutuminen ovat vakuuttaneet minut siitä, että valintakriteerimme ovat

olleet oikeat. Sekin on positiivista, että tietovarastojen isänä pidetty Bill Inmon on liittynyt Databricksin ja Data Lakehousen puolestapuhujiin.

Datan päivittäminen ja tuoreimman tiedon hakeminen Amazon S3:n kaltaisissa pilvipalveluissa on haaste, jonka Delta Lake pyrkii korjaamaan. Teknologia mahdollistaa niin sanotut ACID-yhteensopivat operaatiot [21]. ACID itsessään on Theo Härdenin ja Andreas Reuterin vuonna 1983 lanseeraama akronyymi, joka tulee englanninkielisistä sanoista atomicity, consistency, isolation ja durability. Termit ovat avattu ja suomennettu alla (ks. Taulukko 3). ACID-vaatimus ei määritä, kuinka tietokanta käytännössä vastaa vaatimukseen, mikä mahdollistaa sen käytön pelkkänä markkinointina – varsinkin C:n eli konsistenssin suhteen [4]. Databricksin dokumentaation [22] mukaan Delta Laken ACID lupaa, että useat kirjoittavat voivat kirjoittaa samaan tauluun yhtä aikaa, ja taulu pysyy lukukelpoisena kaiken aikaa. Päällekkäiset kirjoitusoperaatiot ratkaistaan optimistisesti (eng. optimistic concurrency control). Databricks ei ole ainut yhtiö, joka pyrki ratkaisemaan tietoaltaiden ACID-ongelman: Delta Lakeen verrattavia teknologioita ovat Apache Hudi (Uber) ja Apache Iceberg (Netflix).

KIRJAIN	TERMI	KUVAUS
A	Atomisuus	Transaktiot suoritetaan atomisesti eli kokonaan tai ei ollenkaan. Vian ilmaantuessa kirjoitusoperaatio peruutetaan.
C	Eheys	Monimerkityksellinen ja siten ongelmallinen termi, joka viittaa muun muassa siihen, ovatko primääri- ja viiteavaimet pätevässä tilassa, ja siihen, että vastataanko kyselyihin aina tuoreimmalla datalla vai onko käytössä eventual consistency -malli.
I	Eristyvyys	Kirjoitusoperaatiot suoritetaan siten, että ne tapahtuvat toisistaan erillään. Tyypillinen esimerkki on, että kaksi eri applikaatiota yrittävät muokata saman sarakkeen arvoa samalla hetkellä.
D	Pysyvyys	Onnistuneiden transaktioiden pysyvyys pyritään säilyttämään vikatilanteissa.

Taulukko 3. ACID-lyhenteen osatekijät

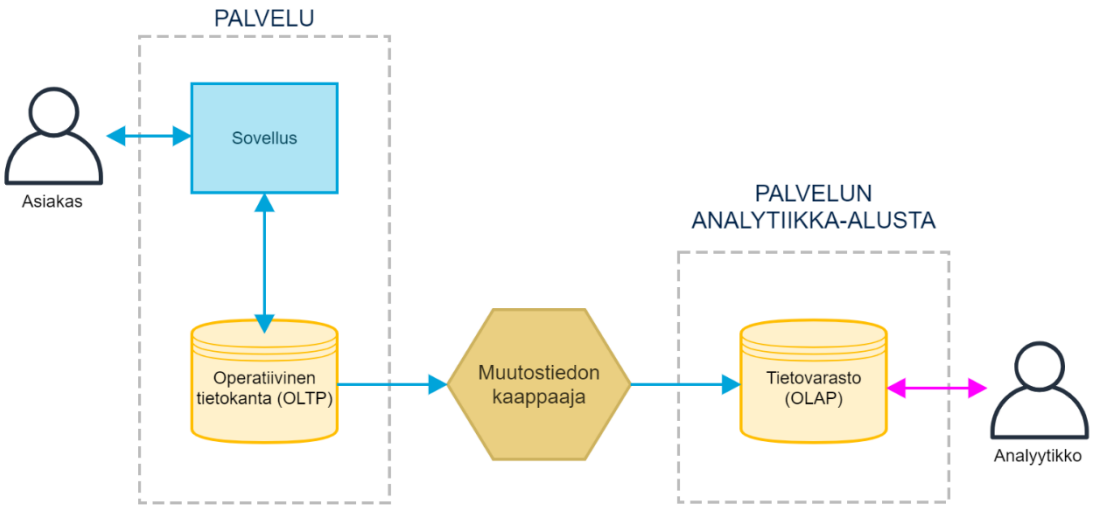
Seuraavissa luvuissa esitelty malli perustuu oletukseen, että yrityksellä on halu siirtää eri palveluiden omista tai yhteisistä tietokannoista data yksittäiseen tietovarastoon tai -järveen. On hyvä

huomioida, että tämä malli, jossa tietovarasto on eräänlainen monoliitti tai valtava keskitetty siilo, ei ole ainut tapa toimia. Piethein Strengholtin esittelemä Scaled Architecture [23] ja Zhamak Dehghanin esittelemä Data Mesh [24] ovat arkkitehtuureja tai paradigmoja, jotka perustuvat datanhallinnan ja datan omistajuuden siirtämiseen datan tuottajille eli esimerkiksi mikropalveluille. Scaled Architecture ja Data Mesh ovat kummatkin erittäin tuoreita ideoita dataekosysteemissä. Strenholt on julkaissut arkkitehtuuristaan vuoden 2020 lopulla ja Dehghanin kirja on yhä ”early access”-tilassa, mutta sitä voi lukea, mikäli on O’Reillyn kirjaston maksava asiakas. Olen tämän uuden arkkitehtuurin puolestapuhuja ja tutustunut varsinkin Strenholdtin ajatuksiin syvällisesti, mutta arkkitehtuurin käyttöönotto vaatii muutoksia myös sovelluksista vastaavien tiimien käytäntöihin, joten koin realistisemmaksi ottaa Polarilla käyttöön vanhan keskitetyn tietovaraston – joskin siten, että muutosvara on otettu huomioon toteutuksessa. Arkkitehtuurisesti tämä toteutus edustaa hyvin pitkälti edeltäjäänsä, johon ei ole tarvetta pureutua sen tarkemmin tässä opinnäytetyössä: suurimpana erona on komponenttien vaihto suljetuista ja maksullisista työkaluista (eng. proprietary) joko avoimen koodin ratkaisuihin tai sellaisiin ratkaisuihin, jotka ovat myöhemmin korvattavissa toisella tuotteella. Tämä mahdollistaa toimittajaloukun (eng. vendor-lock) välttämisen. Olen pyrkinyt kirjoittamaan tämän opinnäytetyön siten, että se on toteutettavissa kotikoneelle tai oppilaitoksen ympäristöön käyttämättä Databricksin maksullisia palveluita.

3 Muutostiedon kaappaaminen relaatiotietokannasta

Muutostiedon kaappaaminen (eng. change data capture) on prosessi, jossa tunnistetaan lähdekantaan tapahtuneet muutokset, jotta ne voidaan replikoida tietovarastoon. Replikoinnin voi suorittaa kopioimalla lähdetietokannan taulut tietovarastoon joka yö kokonaisuudessaan. Suurten taulujen kanssa tämä on kuitenkin huomattavan raskas operaatio. Lähdekantaan kohdistuu matalampi kuorma, mikäli sieltä haetaan vain se osa tiedosta, mikä eroaa viime latauksesta. Tätä muutostietoa voi kaapata eri metodein: tietokantajärjestelmän binäärilokia lukemalla, luomalla lokia SQL-triggereiden avulla tai SQL-palvelimen ulkopuolisella CDC-skriptillä. [25.]

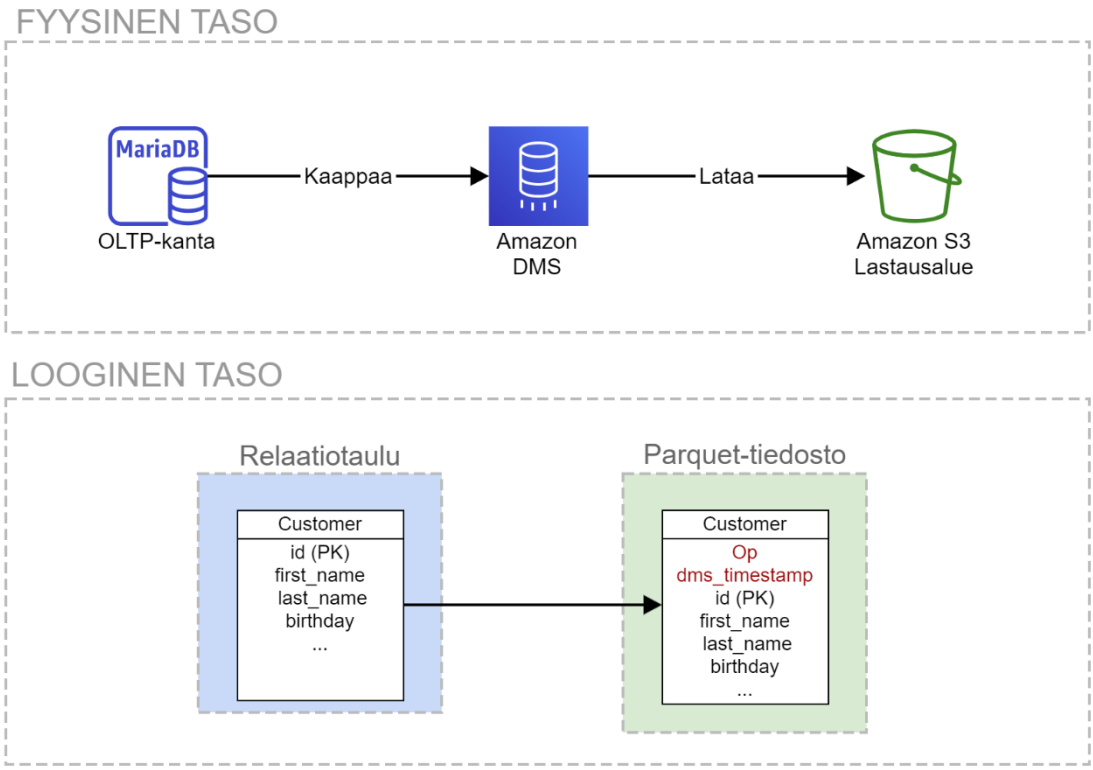
Muutostiedon kaappaamiseen soveltuvien palveluiden ja työkalujen vertailu on liian laaja aihe tässä opinnäytetyössä käsiteltäväksi, mutta muutostiedon kaappaajana (eng. CDC tool) voi käyttää esimerkiksi SaaS-palveluita, kuten Fivetran, Amazon Database Migration Service tai Confluent Cloud. Avoimen lähdekoodin ohjelmaa, joka suoriutuisi samasta, ei työn alkaessa ollut olemassa. Opinnäytetyön kirjoitushetkellä lupaavin vaihtoehto on Airbyte, reilun vuoden vanha avoimen lähdekoodin integraatiotyökalu. Mikäli yrityksellä on käytössä useiden eri valmistajien lähdekantoja, on täysin mahdollista, että yksi työkalu ei hallitse kaikki lähteitä, vaan eri kantojen data pitää integroida tietovarastoon eri työkaluilla. Muutostiedon voi kaapata esimerkiksi ajastetusti joka yö, muutaman tunnin välein tai lähes reaaliaikaisena muutostiedon virtana. Mikäli muutostiedon kaappaaja hoitaa myös tietomallinnuksen mukaisen muunnoksen, kokonaisuus noudattaa ETL-prosessia (extract, transform, load). Mikäli tietomallinnus suoritetaan vasta tietovarastossa, eli muunnos ja tallennus käännetään toiseen järjestykseen, kokonaisuus noudattaa ELT-prosessia (extract, load, transform) [26].



Kuva 2. Muutostiedon kaappaaja sijoittuu OLTP-kantojen ja tietovaraston tai muun kohdemääränpään väliin.

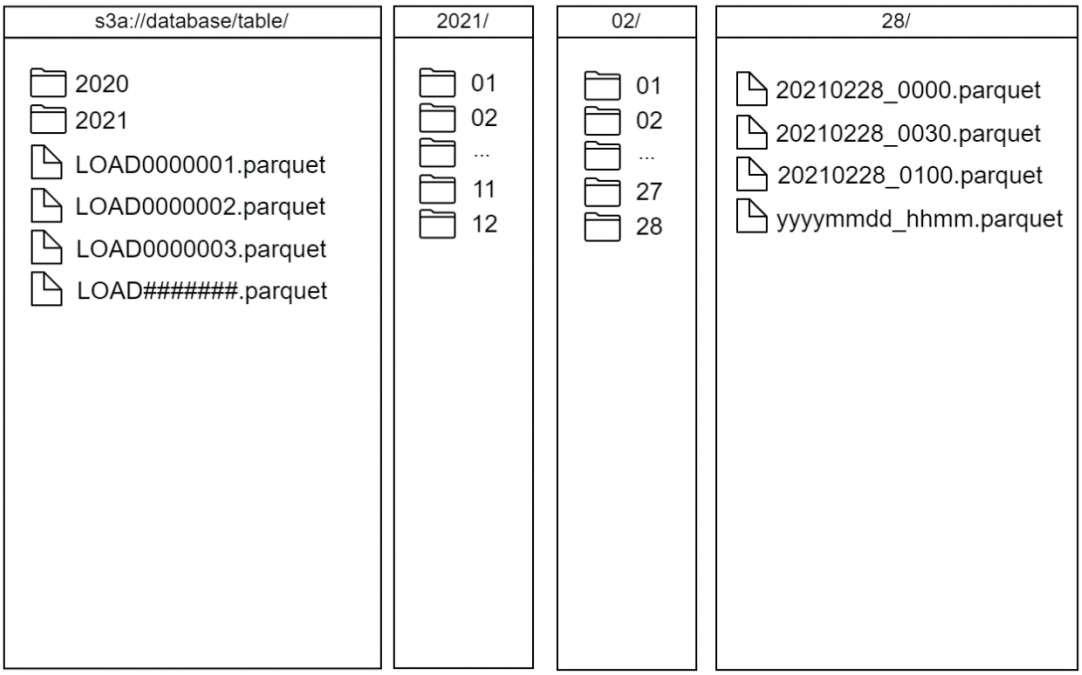
3.1 Muutostieto MariaDB:stä S3:een

Seuraavissa luvuissa oletetaan, että lähdetietokantajärjestelmä on MariaDB ja muutostiedon kaappaaja on Amazon DMS (Database Migration Service), joka lukee MariaDB:n sisäistä binäärilokia ja kirjoittaa muutokset Parquet-tiedostomuodossa lastausalueelle. Tämä prosessi on kuvattuna alla (Kuva 3). Kun prosessi ajetaan ensimmäisen kerran, DMS-palvelu lataa täyden kopion taulusta (eng. full load). Jatkossa DMS seuraa MariaDB:n binäärilokia ja kirjoittaa muutokset kohteeseen. Jokainen kirjoituskerta luo uuden tiedoston ja tiedostot ovat aikaleiman mukaan nimetyt. Lastausalueen muutostieto jakaa lähdetaulun kanssa muutoin saman skeeman, mutta Amazon DMS on lisännyt siihen kaksi uutta kenttää: `Op`, joka kuvastaa operaatiota (insert, update, delete) ja `dms_timestamp`, joka on binäärilokin aikaleima. Lastausalue ei ole osa tietovarastoa; se on välivaihe, josta se pitää ladata tietovarastoon. Nämä vaiheet käydään yksityiskohtaisesti läpi seuraavassa luvussa (ks. Luku 4).



Kuva 3. DMS kaappaa lähdetaulun ja kirjoittaa sen Parquet-formaatissa lastausalueelle.

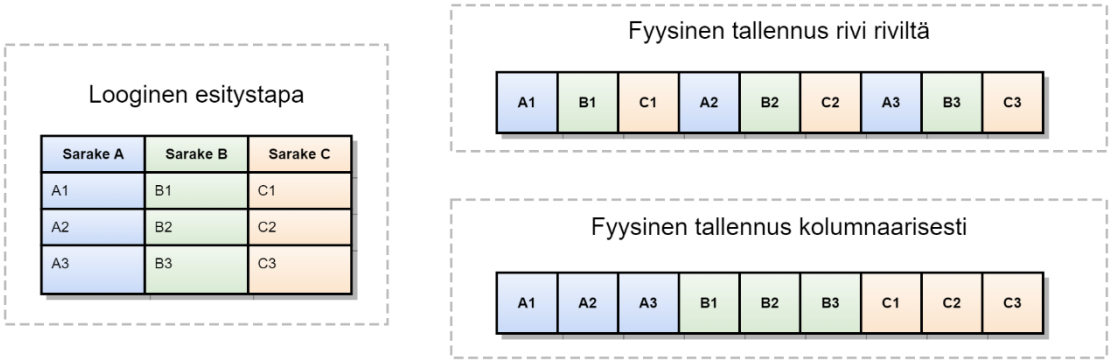
Lastausalue ei pakota tiedostojen skeemaa, joten skeeman tulkitseminen on lukijan vastuulla. Mikäli lähdekannan skeema muuttuu, esimerkiksi uuden sarakkeen myötä, voi lastausalue sisältää tiedostoja, joissa on keskenään eri skeema. Mikäli DMS:n määränpään asetuksissa ei erikseen määritä BucketFolder-arvoa, ja päivämäärän mukainen partitiointi on aktivoitu, tiedostot kirjoitetaan S3-koriin alla olevan kuvan mukaisesti (Kuva 4). Seuraavissa luvuissa lastausalueen oletetaan noudattavan tätä nimeämisstandardia; oli lastauslaituri S3-korissa tai jossakin muussa ympäristössä, kuten testikoneen levyjärjestelmässä (esim. `/mnt/staging/` tai `<projektikansio>/S3/`). Huomaa, että taulun hakemiston juuressa olevat `LOAD`-alkuiset tiedostot eivät ole minkään partition sisällä: nämä tiedostot sisältävät kertalatausdatan (eng. full load data) eli DMS:n käynnistyshetkellä kopioidun, senhetkisen kaiken tiedon lähdekannasta. Tiedostojen määrä riippuu lähdetaulun koosta: DMS jatkaa kirjoittamista toiseen tiedostoon, mikäli asetuksissa määritelty tiedoston maksimikoko ylittyisi. Muutosdataa aletaan kirjoittaa tuosta hetkestä alkaen päivämääräkohtaisiin alihakemistoihin.



Kuva 4. DMS:n nimeämisstandardi hakemistoittain.

On mainittavan arvoista, että Amazon DMS:n voi korvata millä tahansa työkalulla, joka toteuttaa saman toiminnallisuuden, eli kaappaa datan MariaDB-relaatiotietokannasta ja lataa sen sovittua tiedostomuotoa ja nimeämisstandardia käyttäen sellaiseen lokaatioon, jota Spark osaa lukea. Vaihtoehtoisia työkaluja on useita, kuten Microsoftin kilpaileva tuote Azure Data Factory tai Kafka ja sen ekosysteemiin kuuluva Debezium-liitännäinen.

Yksi merkittävä suorituskykyyn liittyvä piirre moderneissa OLAP-käyttöön suunnitelluissa järjestelmissä on tiedon tallentaminen kolumnimaisesti tiedostoon. Parquet-tiedosto, jota Delta Lake käyttää tallennukseen, on Twitterin ja Clouderan yhteisestä ponnistuksesta syntynyt tiedostoformaatti, joka on nykyisin Apache-lisenssin alainen ja siten avointa lähdekoodia, kuten yritysten edustajat kertovat Hadoop Summit 2014 -tallenteessa. Parquet-tiedosto on kolumnaarinen tiedostoformaatti. Tyypillinen CSV-tiedosto ja Parquet-tiedosto näyttävät esitysmuodossa samalta, mikäli niiden sisältöä tulostaa näytölle esimerkiksi Sparkin show-metodilla. Tiedoston fyysinen tallennus poikkeaa kuitenkin huomattavasti riveittäin tallennetuista tiedostoista, joita ovat monien muiden joukossa CSV ja JSON sekä useiden relaatiokantojen sisäiset tiedostot. [27.] Kolumnaarisen ja rivisuuntaisen tiedoston eroavaisuudet on visualisoitu alla olevassa kuvassa (Kuva 5). Kolumnaarinen tiedosto on data-analyytikon käsittelyssä eli OLAP-kyselyitä tehdessä paremmin käyttöön optimoitavissa kuin rivi riviltä tallennettu tiedosto.



Kuva 5. Kolumnaariset ja perinteiset tabulaariset tiedostot.

Parquet-tiedosto sisältää myös skeeman. Tiedoston loppuun kirjoitetaan FileMetaData, johon sisältyy SchemaElement. Parquet-tiedostoa tukevat lukijat, kuten Spark, eivät siis joudu arvaamaan tietyn sarakkeen datatyyppiä. Tämä on huomattava parannus esimerkiksi CSV-tiedostoon, jota lukiessa taulukon skeemasta tiedetään vain sarakkeiden nimet. Ellei CSV-tiedoston skeemaa ole erikseen tallennettu johonkin katalogiin, tiedon muoto pitää voida päätellä arvoista (eng. infer schema), mikä on sekä epätehokasta että epäluotettavaa. Mikäli valittu työkalu on DMS ja tiedostot halutaan tallentaa aiemmin määritellyllä tavalla, täytyy DMS S3-kohteen (eng. target endpoint) parametrit asettaa alla olevan taulukon (ks. Taulukko 4) mukaisesti [28].

Asetus	**Arvo**
DatePartitionEnabled	True
DatePartitionSequence	YYYYMMDD
DatePartitionDelimiter	SLASH
TimestampColumnName	dms_timestamp
DataFormat	Parquet
cdcMaxBatchInterval	1800

Taulukko 4. DMS Endpoint -parametrit, joilla saavutetaan tämän opinnäytetyön esimerkkejä vastaavaa dataa.

On tärkeää muistaa, että Parquet ja Delta Lake eivät ole synonyymejä. Amazon DMS:llä luotu lastausalue sisältää tiedostoja Parquet-tiedostomuodossa, mutta ei Delta Lake -kerrosta. Mikäli DMS kirjoittaisi Delta Lake -muotoa, Parquet-tiedostojen rinnalla olisi aina `_delta_log/`-hakemisto.

3.2 GDPR ja Lifecycle Policy

DMS:n kirjoittama lastauslaituri on GDPR:n kannalta hankala alue; delete-operaatiot ovat vain ja ainoastaan rivejä muiden joukossa ja pysyvät tallessa, ellei niitä erikseen tuhota. Asiakkailla on oikeus tulla unohdetuksi, mikä vaikuttaa väistämättä data-alustan arkkitehtuuriin [29]. Yksi ratkaisu on, että lastauslaiturin S3-ämpäriin asetetaan ”Lifecycle Policy”, joka tuhoaa tiedostot, kun tiedoston luomisesta on kulunut 30 päivää. Kun data siirretään lastauslaiturilta Delta-formaattiin, suoritetaan MERGE-toiminto, jossa delete-operaatio poistaa rivin kohdekannasta ja update-operaatio ylikirjoittaa rivin. Tästä toimintatavasta on se välitön hyöty, että tuotantokannoista poistetut rivit eivät jää data-altaaseen ikuiseen säilöön vaan poistuvat kaikista mahdollisista data-alustan säilöistä noin 30 päivässä. Haittapuoli tällä metodilla on, että historiallista dataa voi tarkistella vain sillä tarkkuudella, millä se on tallennettu tuotantokantoihin. Voisi olla mielenkiintoista tutkia, vaikuttaako esimerkiksi paikkakunnalta toiselle muuttaminen ihmisten käyttäytymiseen – Polarin tapauksessa ihmisten treeniaktiivisuuteen. Tämä ei kuitenkaan ole mahdollista, ellei data-alustaan tallennu sekä muuttoa edeltävä että seuraava osoite päivämäärineen. Yrityksen tarpeista riippuen voi siis olla, että joidenkin yksittäisten taulujen kohdalla on kannattavaa suorittaa MERGE-operaatio toisenlaista logiikkaa käyttäen, josta toimii esimerkkinä Slowly Changing Dimension Type 2, kunhan yrityksen tietosuojalausunto ja -käytännöt mahdollistavat tämän.

3.3 Ongelmalliset tietotyypit

DMS tukee kattavasti MySQL-yhteensopivien kantojen tietotyyppejä, mutta etumerkittömät kokonaisluvut (eng. unsigned integer) voivat aiheuttaa päänvaivaa. Spark herjaa Parquet-tiedostoista, jotka sisältävät etumerkittömiä kenttiä: myös dokumentaatio listaa vain ja ainoastaan etumerkilliset kokonaisluvut eli kokonaisluvut, jotka voivat olla arvoltaan negatiivisia eli pienempiä kuin nolla [30]. Ongelman voi kiertää suorittamalla tietotyypin vaihdon DMS:llä. Tämä onnistuu `TableMappings`-toiminnolla `change-data-type` [31]. Alla (ks. Koodi 2) otos AWS DMS:n tarvitsemasta `TableMappings` JSON-tiedostosta, jossa 1-tavuinen eli 8-bittinen etumerkitön kokonaisluku, joka kattaa arvoalueen 0–255, muutetaan 2-tavuiseksi kokonaisluvuksi, joka kattaa arvoalueen -32768–32767. Mikäli tuotantokannassa on 8-tavuisia etumerkittömiä kokonaislukuja, on riski, että numeroavaruus loppuu kesken, sillä DMS tai Parquet-tiedostomuoto eivät tue 16-tavuisia eli 128-bittisiä kokonaislukuja. Käytännössä tuo on onneksi huomattavan epätodennä-

köistä, sillä 8-tavuiset numerot ovat huomattavan suuria. Etumerkittömän 8-tavuisen kokonaisluvun suurin mahdollinen positiivinen arvo on 2 potenssiin 64 miinus yksi. Mikäli tuotantokantaan kirjoitettaisiin yksi rivi joka millisekunti eli tuhat riviä sekunnissa, niin juokseva rivinumerointi saavuttaisi aiemmin mainitun suurimman positiivisen arvon noin 550 miljoonassa vuodessa.

```
"object-locator": {
    "schema-name": "%",
    "table-name": "%",
    "column-name": "%",
    "data-type": "uint1"
},
"data-type": {
    "type": "int2" }
```

Koodi 2. AWS DMS -parametrit, jotka muuttavat etumerkittömän 1-tavuisen kokonaisluvun 2-tavuiseksi etumerkilliseksi kokonaisluvuksi.

4 Muutoshistorian lataaminen Delta Lake -muotoon

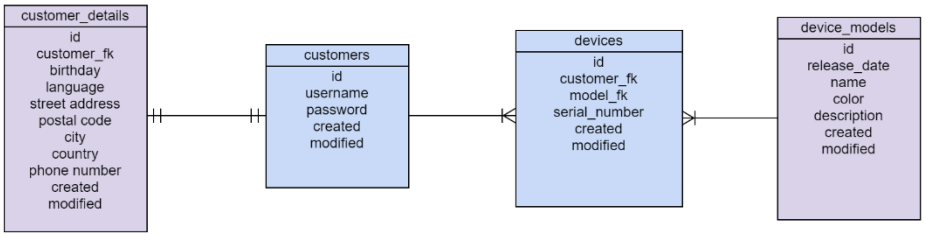
Tämän luvun esimerkit perustuvat simuloituun tietoaineistoon, joka on kuvitteellisen yrityksen kuluttajien tuottamaa tietoa. Muodoltaan tiedostot ovat samanlaisia kuin Amazon DMS:n kirjoittamat Parquet-tiedostot. Tieto on kuvitteellista muun muassa tietosuojaan ja yrityssalaisuuksiin liittyvistä syistä, mutta esimerkkikoodi on kuitenkin hyvinkin samanlaista kuin millä Polar Electro Oy:n vastaava tieto ladataan tietoaltaaseen.

Python-koodi, jolla tämän luvun tietueet on tuotettu, löytyy minun henkilökohtaisesta Github-repositoriosta kokonaisuudessaan [32]. Repositorion tärkeimmät tiedostot ovat Jupyter Notebook -formaatissa, ja niiden nimeäminen noudattaa kaavaa, jossa kaksi ensimmäistä numeroa kasvavat (esim. `01_generate_dms_output.ipynb` ja `02_load_to_bronze.ipynb`).

Tämä luku nojaa luvun kaksi lopussa tehtyyn pohjaoletukseen, että haluttu arkkitehtuuri on tyyliltään keskitetty monoliitti eikä esimerkiksi Scaled Architecture -paradigman mukainen kokonaisuus. Valittu arkkitehtuuri ei kuitenkaan estä Scaled Architeturen käyttöönottoa myöhemmin. Tulevaisuudessa datanhallinnan ja omistajuuden voi siirtää palvelua luovan tiimin käsiin. Tässä tapauksessa palvelua tuottava tiimi toisi kuratoidun, kolumnaarisen tietokannan esimerkiksi Delta Lake -formaatissa BI-raportteja luovan järjestelmän luettavaksi. Data-analytiikkatiimin vastuut alkaisivat vasta siitä, kun dataa yhdistellään hopea- tai kultatasolla.

4.1 Lähdekannan tietomalli

Esimerkeissä käytetty tietomalli (ks. Kuva 6) on pelkistetty eikä edusta tyypillisen tuotantokannan monimutkaisuutta. Kuvan kaksi keskimmäistä eli sinistä taulua edustavat reaalimaailman entiteettejä: asiakkaita (`customers`) sekä laitteita (`devices`). Tässä esimerkissä yhdellä asiakkaalla voi olla monta hänen nimiinsä aktivoitua laitetta, mutta yksi laite voi kuulua vain yhdelle käyttäjälle. Laitimmaiset, violetit taulut edustavat ei-fyysisiä entiteettejä tai kokoelmia: asiakkaan yhteystietoja (`customer_details`) sekä laitemallia (`device_models`). Yhtä laitemallia voidaan valmistaa useita kappaleita.



Kuva 6. Lastauslaiturin tiedostojen tietomalli.

DMS:n kertalatauksella (eng. full load) luotua dataa simuloivat tiedostot on generoitu Python-skriptillä ja kirjoitettu projektipolussa sijaitsevaan S3-nimiseen hakemistoon ja sen alihakemistoihin (ks. Kuva 7). Esimerkkikoodi on ajettu Windows-ympäristössä, mutta käytän siitä huolimatta opinnäytetyössä POSIX-hakemistopolkujen erottimia (eli /-merkkiä). Skripti löytyy kokonaisuudessaan edellä mainitusta GitHub-repositoriosta. S3-kansio edustaa tässä minimalistisessa koeympäristössä data-allasta; tuotannossa data tallennattaisiin esimerkiksi Amazon S3 -objektisäilöön tai hajautettua levyjärjestelmää, kuten HDFS:ää, hyödyntäen. Hakemistopolun ensimmäinen elementti, `staging`, edustaa yksittäistä S3 buckettia. Hakemistopolun `abc`-alihakemisto edustaa RDS-kannan nimeä, josta tiedostot on ladattu. Tässä yksinkertaisessa esimerkissä oletetaan, että kaikki neljä taulua ovat tulleet samasta lähdekannasta. Tuotantokäytössä lähdekantoja voi olla useita, kuten esimerkiksi yksi per mikropalvelu.

```
.\S3\staging\dms\abc\customers\customers\LOAD00000001.parquet
.\S3\staging\dms\abc\customers\customer_details\LOAD00000001.parquet
.\S3\staging\dms\abc\devices\device_models\LOAD00000001.parquet
.\S3\staging\dms\abc\devices\devices\LOAD00000001.parquet
```

Kuva 7. Lastauslaiturin tiedostot, jotka simuloivat DMS:n kertalatauksen kirjoittamia tiedostoja.

Yllä listatut Parquet-tiedostot ovat Pandas DataFrameina Faker-kirjaston avulla generoituja, joten kaikki yhteystiedot ovat täysin tekaistuja – poikkeuksena customers-taulun ensimmäisen henkilön nimi ja sähköposti, jotka ovat minun tietojani. Taulujen sisältö on helposti silmäiltävissä, kun tarkastelee taulujen ensimmäisiä rivejä (ks. Kuva 8).

Nämä lastauslaiturin eli staging-alueen taulut edustavat kahta poikkeusta lukuunottamatta samaa skeemaa kuin yllä esitellyt kuviteelliset lähdekannat. Ensimmäinen eroavaisuus on, että salasana (`customers.customers.password`) on pudotettu pois. Tuotannossa tämän voi tehdä DMS:n ”remove column”-säännöllä. Salasanat, session tokenit ja muu tietoturvan kannalta arka

tieto kannattaa jättää tuomatta tietovarastoon, mikäli sillä ei ole selkeää analyyttista arvoa. Toinen eroavaisuus on, että tauluihin on lisätty sarake `dms_timestamp`. Kaikki ajat on tiivistetty sekuntitarkkuuteen, jotta data veisi vähemmän näyttötilaa. Muut aikaleimat (`created`, `modified`) ovat Pythonin `datetime`-objekteja, mutta `dms_timestamp` on tekstimuodossa.

dms_timestamp	id	username	created	modified
2021-08-05 15:19:54	1	janisourander@kamk.fi	1970-01-15 10:00:00	1970-02-20 12:34:56

dms_timestamp	id	customer_fk	birthday	language	street_address	postal_code	city	country	phone_number	created	modified
2021-08-05 15:19:54	1	1	1963-12-23	ff	817 Robbins Parkway Suite 056	46333	North Calvin	GW	001-250-991-3804x31652	1970-01-15 10:00:00	1970-01-15 10:00:00

dms_timestamp	id	release_date	name	color	description	created	modified
2021-08-05 15:19:54	1	2010-05-15	Super Gadget 100	Red	lorem ipsum	2010-03-21 12:00:01	2010-03-21 12:00:01

dms_timestamp	id	customer_fk	model_fk	serial_number	created	modified
2021-08-05 15:19:54	1	1	1	862-86-8047	1970-01-15 10:18:54	1970-01-15 10:52:11

Kuva 8. Kaikkien lastauslaiturille kirjoitettavien Pandas-taulujen ensimmäiset rivit. Taulut järjestyksessä: customers, customer_details, device_models, devices.

Yllä olevat taulut esittävät DMS:n kertalatauksen luomia tiedostoja, joten niissä ei ole muutostietoa laisinkaan. DMS ei sisällytä ”Op”-saraketta kertalataukseen laisinkaan; jos se sisällyttäisi, sen arvo olisi oletettavasti joka rivillä ”I” eli INSERT.

4.2 Data lakehousen tietomalli: pronssi, hopea ja kulta

Aiemmissa luvuissa on määritelty, että lastauslaituri sisältää sekä kertaluontoisen kopion että jatkuvalla tahdilla sisään virtaavan muutostiedon. Kummankin kirjoittamisesta vastaavat DMS – tai simuloitu DMS – esimerkkidatan kanssa. Lastauslaiturilta tieto poistetaan 30 päivässä GDPR-syistä S3:n Lifecycle Policy:n avulla. Ennen kuin tieto ehtii poistua lastauslaiturilta, se tulee ladata data lakehouseen ja yhdistää aiemmin ladatun tiedon kanssa. Jos DMS kirjoittaa muutoshistorian tänään lastauslaiturille kello 12:30, ja tieto ladataan ja yhdistetään data lakehouseen tämän jälkeen, niin data lakehouse sisältää ikään kuin tilannekuvan (eng. snapshot) lähdekannasta tuolla kellonlyömällä. Tätä kerrosta, joka sisältää kopion lähdekannasta, kutsutaan pronssikerrokseksi, ja se on data lakehousen osalta totuuden lähde (eng. single source of truth). Tähän Multi-Hop- tai medaljonkiarkkitehtuuriin kuuluvat myös kerrokset hopea ja kulta (ks. lyhyet kuvaukset Taulukko 5:stä), jotka sisältävät eri taulujen yhdistelmiä tai muunnoksia [33].

Kerros	Ominaisuudet
Pronssi	▪ Useita normalisoituja tauluja. Data ja tietomalli kaapattu lähdekannoista. ▪ Lähtökohtaisesti kielletty pääsy kaikilta ihmiskäyttäjiltä. Automaattiset järjestelmät lukevat kerrosta.
Hopea	▪ Fakta- ja dimensiotauluja sekä jalostettua ja rikastettuja tauluja. ▪ Pääsy sallittu tehokäyttäjille. Käyttö vaatii dimensiomallin ymmärrystä sekä SQL-osaamista. ▪ Mikäli taulut sisältävät henkilötietoja, tieto näytetään vain mikäli katsojalla on siihen pääsy. Tyypillinen käyttäjä näkee asiakkaan sähköpostin sijasta tekstin ”REDUCTED”, ellei hänellä ole merkittävää tarvetta sekä asiakkaan suostumusta käsitellä sähköpostiosoitteita. ▪ Tauluista luodaan BI-käyttöön visuaalisia raportteja ja visualisointeja.
Kulta	▪ Taulut ovat leveitä ja monet tauluista summattu aikaikkunan tai asiakasryhmän yli siten, että taulussa lukeva tieto on anonyymiä. ▪ Pääsy sallitaan laajemmin kuin hopeatasolla. ▪ Tauluista luodaan BI-käyttöön raportteja ja visualisointeja

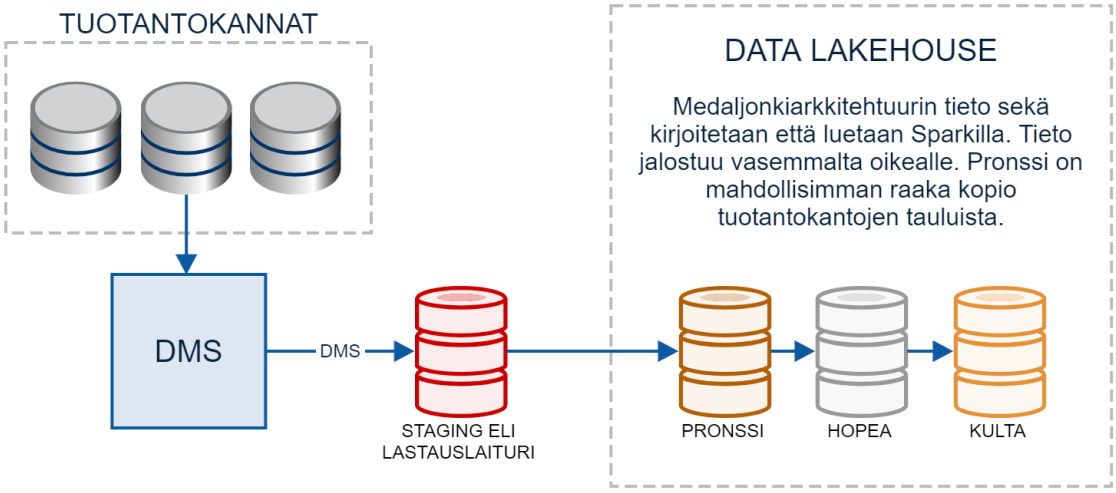
Taulukko 5. Medaljonkitasojen ominaisuudet.

Fyysisesti tieto voi sijaita joko saman S3-ämpärin eri poluissa tai kokonaan eri S3-ämpäreissä. Fyysisesti irralliset S3-ämpärit ovat pääsyoikeuksien hallinnan kannalta mielestäni helpompi ratkaisu. Kuva arkkitehtuurista kolmen eri S3-ämpärin kanssa on nähtävissä alempana (ks. Kuva 9). MariaDB:n ja vastaavien relaatiotietokantojen kanssa yksi ehdotelma on seuraava:

- s3://staging-bucket/<tool>/<source_system>/<database>/<table>/
- s3://bronze-bucket/<source_system>/<database>/<table>/

Yllä olevan ehdotelman hakasulkeiden sisällä olevat arvot ovat: `tool` eli CDC-työkalu, `source_system` eli lähdejärjestelmä, `database` eli tietokanta ja `table` eli taulu. Tämän luvun esimerkeissä CDC-työkalu on DMS ja lähdejärjestelmä on nimeltään abc. Tosielämän tilanteessa yrityksellä voi olla useita järjestelmiä, joiden data tuodaan samalla työkalulla lastauslaiturille.

Eri työkalut, kuten DMS tai Fivetran, sekä eri lähdejärjestelmät, kuten relaatiokannat tai API:t, voivat tuottaa ja sisältää hyvin erimuotoista tietoa, joten lastauslaituri ja pronssi noudattavat väkisinkin muutamaa hieman erilaista nimeämisstandardia. Tässä opinnäytetyössä keskitytään kuitenkin DMS:n luomiin tiedostoihin, joten tämän luvun esimerkeissä ei muita nimeämisstandardeja tule vastaan.



Kuva 9. Medaljonkiarkkitehtuuri alkaa pisteestä, jossa Spark lukee datan lastauslaiturilta pronssitasolle. Data yhdistellään ja rikastetaan sekä se muovataan valittuun tietomalliin hopea- ja kultatasoille.

4.3 Alustan asennus tai käyttöönotto

Mikäli koodi ajetaan Databricksin ympäristössä, Sparkin, Deltan, Hiven ja muiden komponenttien asennus hoituu heidän toimestaan. Databricks nimittää heidän uutta, yrityksille suunnattua arkkitehtuuriaan, E2-arkkitehtuuriksi. Kyseisessä arkkitehtuurissa on kaksi eri komponenttia: control plane ja data plane, joista ensimmäinen on Databricksin omalla AWS-tilillä ja jälkimmäinen asiakkaan (kuten Polar) AWS-tilillä. Databricksille annetaan IAM cross-account -roolin avulla lupa pystyttää EC2-instansseja eli Spark-laskentaklustereita asiakkaan hallitsemaan VPC:hen eli privaattiin

virtuaalipilveen. Data ei siis poistu asiakkaan käsistä, mutta käyttäjän autentikointi ja palvelun käyttö hoituu Databricksin verkko-osoitteiden läpi. [34.]

Databricksin sijasta on myös muita vaihtoehtoja, kuten asentaa Spark-ekosysteemi lokaalisti. Tämän opinnäytetyön esimerkkikoodit on ajettu Dell XPS 15 -kannettavalla Windows 10 -ympäristössä. Ohjelmointiympäristönä on JetBrain PyCharm opiskelijalisenssillä sekä Jupyter Notebook. Mikäli haluat toisintaa opinnäytetyön vaiheet, lyhyet ohjeet asentamiseen löytyvät liitteistä (ks. Liite 1). Sparkin hyödyt tulevat esille suurten tietomäärien käsittelyssä hajautetusti, joten yhdelle koneelle luotu single node -asennuksen suorituskyvystä ei kannata tehdä suuria päätelmiä. Käyttämäni standalone-asennus käyttää vakioasetuksia, jotka sinänsä toimivat, mutta eivät mahdollista muun muassa katalogin pysyvyyttä ainakaan Windows-ympäristössä. Repositoriostani löytyvät koodiesimerkit käyttävät tämän tähden väliaikaisia näkymiä (eng. temporary view) varsinaisten taulujen sijasta.

4.4 Tiedon latauksen vaiheet

Tiedon lataaminen lastauslaiturilta sisältää eri vaiheita, jotka esitellään tarkemmin seuraavien aliotsikoiden yhteydessä. Lyhyesti avattuna vaiheita ovat:

- Orkestrointi. Alla olevassa esimerkissä tehdään orkestrointia kahdessa tasossa:
    - Koko tehtäväkokonaisuuden orkestrointi. Lataa monta taulua ja sisältää useita eri vaiheita. Ajastettava tehtävä.
    - Yksittäisten taulujen latauksen orkestrointi yllä olevan tehtävän sisällä.
- Kertalataus. Tämä suoritetaan vain jos taulua ei vielä ole Hive-kannassa tai vaihtoehtoisesti orkestrointiskriptille syötetyn parametrin mukaan (esim. `force_full_load: TRUE`)
- Jatkuva lataus. Tämä suoritetaan aina kun skripti ajetaan. Myös ensimmäisellä kerralla.
- Muut toimenpiteet. Muita mahdollisia toimenpiteita ovat skeeman muutosten tarkistaminen, datan laadun tarkistaminen laatuodotuksia vasten, lokin kirjoittaminen la-

tausoperaatiosta sekä ylläpidolliset tehtävät kuten datan uudelleenpartitiointi ja -järjestäminen siten että lataus on tehokkaampaa, tai vanhojen turhaksi käyneiden tiedostojen poisto.

Useat yllä listattujen kohtien yksityiskohdat riippuvat valituista työkaluista (Airflow, Prefect vai natiivi Databricksin Jobs) sekä ympäristöstä (Databricks Platform vai open source). Tästä syystä alla esiteltyjä tapoja kannattaa kohdella suuntaa antavina ohjeina. Yksi johdannossa määritellyistä tavoitteista tai valintakriteereistä on modulaarisuus: se selvästikin toteutuu Databricksin alustassa. Useimmat alla olevat vaiheet voi tehdä joko Databricksin omilla työkaluilla tai ulkopuolisilla ohjelmistoilla. Kaikkien vaihtoehtojen syvällinen läpikäynti veisi huomattavasti aikaa. Koinkin parhaaksi tehdä useimmat vaiheet siten, että toteutin sen ensimmäisenä helpoimmalla ja nopeimmalla mahdollisella työkalulla – usein Databricksin itsensä tarjoamalla työkalulla. Tätä työnkulkua suosittelee myös David Suarez Medium-kirjoituksessaan [35]. Myöhemmin esimerkiksi öisten ajojen orkestroinnin voi vaihtaa Apache Airflow:iin tai Prefectiin ja suoritettavan koodin voi siirtää Notebookeista Python wheel -kirjastoihin. Myös käyttäjähallinnan voi ensin toteuttaa yksinkertaisella Notebookilla ja myöhemmin siirtää toisiin työkaluihin.

4.4.1 Orkestrointi

Tiedon lataaminen lastauslaiturilta pronssi-tasolle on siinä mielessä hyvin suoraviivainen tehtävä, että jokainen taulu kokee saman käsittelyn. On siis selvää, että prosessi kannattaa automatisoida eikä kirjoittaa jokaisen taulun logiikkaa käsin. Tähän tarvitsee ainakin kolme komponenttia:

- Konfiguraatio
    - Tietokanta tai tiedosto, joka sisältää listauksen kaikista tauluista, jotka tulee ladata (esim. `dms_tables`).
- Orkestroija (eng. orchestrator)
    - Python-skripti, Databricks Notebook tai esimerkiksi Airflow DAG, joka lukee yllä mainittua konfiguraatiota ja syöttää tiedot parametreinä työläisille.
- Työläinen (eng. worker)

- Jupyter Notebook tai Python-skripti, joka siirrretään Spark-klusterin ajettavaksi esimerkiksi Sparkin pyspark-submit -komennolla tai Databricks Notebook API:n avulla. Sama työläisskripti voi suorittaa sekä kertalatauksen (full load) että seuraavina ajokertoina tapahtuvan jatkuvan latauksen tai nämä voi jakaa eri skripteihin.

Konfiguraatiotaulun voi luoda SQL-kyselyiden ja Python-skriptin avulla joko johonkin relaatiotietokantaan (Amazon RDS) tai taulu voi itsessään olla osa Delta Lakehousea. MariaDB:n taulujen primääriavaimet voi listata yhdistelemällä tietoja `information_schema` databasen tauluista `tables`, `table_constraints` ja `key_column_usage` [36]. Tarkka skeema riippuu orkestrointiskriptin tarkemmasta toiminnasta, mutta taulun tulee sisältää vähintään sarakkeet: `source_system`, `database`, `table`, `primary_keys`. Kolmesta ensimmäisestä voi päätellä taulun sijainnin lastauslaiturilla (esim. `s3://staging/dms/source_system/db/table/`). Viimeinen sarake `primary_keys` on tarpeen, kun pronssitason taulun arvoja päivitetään. Sarakkeen nimi on monikossa, koska joissakin lähdekannan tauluissa voi olla useasta sarakkeesta koostuva komposiittiavain.

Erittäin pelkistetty Databricksin Notebook API:a hyödyntävä orkestroija on esiteltynä alla olevassa koodiesimerkissä (Koodi 3) ja logiikkaa on havainnollistettu ohessa kuvamuodossa (ks. Kuva 10). Mikäli tauluja on kymmeniä tai satoja, työtä voi hajauttaa ajamalla useita orkestroijia rinnakkain. Tämän voi toteuttaa usein eri tavoin, kuten sisällyttämällä konfiguraatiotauluun uuden sarakkeen `run_on_orchestrator`, joka sisältää orkestroijan `id`:n kuten luvun 1, 2 tai 3. Hajauttaminen voi olla tarpeen myös silloin, jos taulut ovat keskenään huomattavan eri suuruisia. Tällöin hakuehtoon voi lisätä esimerkiksi filtterin, joka on pienen taulun kohdalla `WHERE size_category="0-32 GB"`, ja suuren taulun kohdalla vaikkapa `"512-1024 GB"`. Tämä luonnollisesti vaatii, että `dms_tables` taulussa on vastaava kenttä. Tästä on se suora hyöty, että eri kokoluokan taulut voidaan ladata eri kokoisilla klustereilla. Pienen taulun latausta ei voi loputtomiin ajaa rinnakkain, joten valtavat klusterit ovat epäkustannustehokkaita pienten taulujen kanssa. Pienet klusterit eivät sen sijaan selviä suurista tauluista ilman levylle kirjoittamista, mikä pidentää prosessin kestoa huomattavasti. Tuotannossa run-funktio voidaan myös suorittaa `try-except`-logiikan sisällä, jolloin mahdollisiin virhekoodeihin voidaan reagoida valitulla tavalla.

Mikäli saman logiikan haluaa rakentaa ilman Databricksin Notebook API:ia, sen voi hoitaa yllä mainituilla tavoilla, kuten Apache Airflow:lla. Tällöin konfiguraation voi lukea tiedostosta tai tietokannasta ja `notebook.run`-funktion tilalle korvautuu Airflow:n oma operaattori, kuten `SparkSubmitOperator`-luokka. Notebook API:n hyödyntäminen mahdollistaa alustan nopean

käyttöönoton, mutta git-tyylinen versionhallinta on huomattavan vaikeaa interaktiivisten Notebookien kanssa. Versionhallinnan ja CI/CD-putken hallitseminen Databricksissä on liian laaja aihe käsiteltäväksi tässä opinnäytetyössä, joten esimerkit sisältävät vain ja ainoastaan Databricksissä ajettavaa koodia.

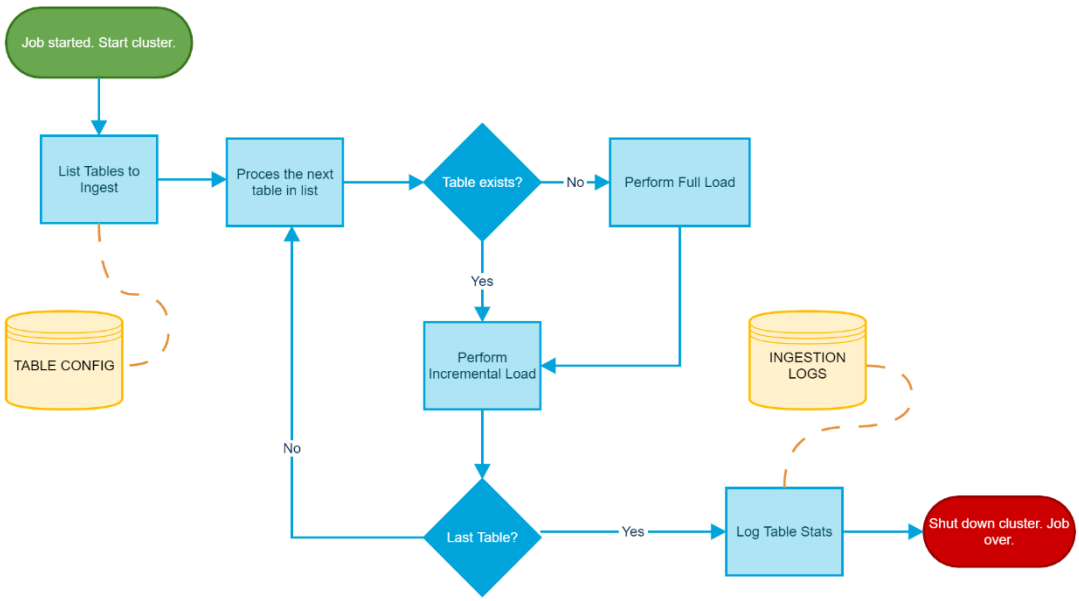
```
# Fetch settings from configuration table
x_tables = spark.sql(f"SELECT * FROM config.dms_tables").collect()


for x in x_tables:

    # Parametes of the tables_to_ingest
    params = {
        "FULL_LOAD": x.full_load_needed,
        "source_system": x.source_system,
        "layer": layer,
        "database": x.database,
        "table": x.table,
        "primary_keys": x.primary_keys
    }

    # Perform full load and incremental CDC load.
    response = dbutils.notebook.run("dms_worker_script", timeout, params)
```

Koodi 3. Orkestrointiskriptin typistetty esimerkkitoteutus.



Kuva 10. Loop-perusteisen orkestrointiskriptin toiminta flowchart-kaaviona.

Databricksin uusi ominaisuus ”multitask jobs”, eli useita tehtäviä sisältävät ajastettavat tehtävät, on tämän opinnäytetyön kirjoitushetkellä Public Preview -tilassa. Se on mahdollinen vaihtoehto

Apache Airflow:n kaltaiselle DAG-riippuvuuksia (suunnattu syklitön verkko, eng. directed acyclic graph) sisältävästä latauksesta. Tehtävien orkestroinnin voi järjestää niin monella tavalla, joten suosittelen jokaista yritystä punnitsemaan vaihtoehdot ja yrityksessä valmiiksi olevan osaamisen soveltuvuuden eri vaihtoehtoihin.

4.4.2 Kertalataus

Kertalatauksen voi suorittaa joko samalla skriptillä tai eri skriptillä kuin inkrementaalisen, jatkuvan latauksen. Mikäli kertalataus suoritetaan samalla skriptillä, sille tarvitsee syöttää parametri, joka määrittää, tarvitaanko kertalataus. Tavallisesti kertalataus tarvitsee tehdä vain kerran: siitä lähtien tauluun tuodaan muutostieto inkrementaalisilla latauksilla. Mikäli lastauslaiturin data poistetaan automaattisesti 30 päivän jälkeen, niin jälkikäteen suoritettu uusi kertalataus vaatii, että tuotantokannan taulu ladataan lastauslaiturille uudestaan DMS:llä. Alla oleva koodiesimerkki (ks. Koodi 4) esittelee tyypillisen lataus- ja kirjoitusoperaation lastauslaiturilta pronssille. Koodissa esiintyvä `DMSPathMerge` on luokka, joka tarjoilee standardoituja S3- ja Hive-polkuja DMS-datalähteille. Luokan tarkempi toteutus riippuu yrityksen valitsemista käytännöistä, joten olen jättänyt sen toteutuksen tämän opinnäytetyön skoopin ulkopuolelle. Esimerkissä oletetaan, että muuttuja `spark` on SparkSession-luokan olio. Databricks Notebookissa tämä muuttuja on automaattisesti alustettu. Mikäli käytössä on jokin muu ympäristö, luokka `pyspark.sql.SparkSession` tulee instantioida muuttujaan `SparkSession.builder.getOrCreate()`-komennolla. Tarkemmat asetukset riippuvat ympäristöstä.

```
import pyspark.sql.functions as F
from pathmerger import DMSPathMerger

# Generate
ss, db, table = 'abc', 'customers', 'customer_details'
pm = DMSPathMerger(ss, db, table)

# Load from STAGING
# (e.g. /mnt/staging/dms/ss/db/table/LOAD*.parquet)
df = (
    spark
    .read
    .option("pathGlobFilter", "LOAD*.parquet")
    .format("parquet")
    .load(pm.staging_path)
    # .withColumn("par", F.col("id") % n_pars)
    .withColumn("src_file", F.input_file_name())
    .withColumn("src_batch_id", F.lit(None).cast("integer"))
)

# Write to BRONZE lake
# (e.g. /mnt/bronze/ss/db/table)
# And to Hive(e.g. bronze.ss_db_table)
(
    df
    .write
    .format("delta")
    .mode("overwrite")
    .option("overwriteSchema", "true")
    .option("path", pm.bronze_path)
    # .partitionBy("par")
    .saveAsTable(pm.hive)
)
```

Koodi 4. Kertalatauksen suorittavan koodin esimerkkitoteutus.

Yllä olevassa koodiesimerkissä (Koodi 4) on kommentoituna risuaitamerkillä rivi, jossa funktio `withColumn` lisää `DataFrameen` uuden sarakkeen `par`. Myös `DataFrameWriter`-objektiin kohdistuva partitionBy on kommentoituna. Sparkissa tehdään kahta toisiaan muistuttavaa partitiointia, jotka menevät helposti sekaisin: muistinsisäistä (eng. in memory) ja levylle tallentuvaa (eng. on disk) [37]. Muistissa tapahtuvan parallelismin optimointi on tärkeää koodin suorittamisee tehokkuuden kannalta, mutta en käsittele sitä tässä opinnäytetyössä. `DataFrameWriter`:n `partitionBy` on nimenomaan levylle tallentuvaa partitiointia ja perii formaattinsa Hive-katalogista. Alle yhden gigatavun tauluja ei tarvitse partitioida: sitä suuremmat voi partitioida siten, että jokaiseen partitioon jää vähintään yksi gigatavu dataa [38]. Tämän opinnäytetyön esimerkkidata on suppea, joten partitiointi on jätetty välistä, mutta tuotantokäytössä tehokas partitiointi voi lisätä lukemisen, kirjoittamisen tai parhaassa tapauksessa kummankin operaation tehokkuutta. Yllä esitelty id:n jakojäännöksen mukaan partitiointi edistää kirjoittamisen tehokkuutta ja vaatii

toimiakseen sen, että sarake sisältää kokonaislukuja. Jakojäännös takaa, että peräkkäiset id-numerot päätyvät juoksevasti `n`-määrään laareja. Koodiesimerkissä tämän `n`-luvun määrää muuttuja `n_pars`, joka tulisi päätellä taulun fyysisestä koosta taulukohtaisesti. Kun taulu tallennetaan levylle fyysisiksi tiedostoiksi, partitiot näyttäytyvät kansioina, joiden tiedostonimeäminen noudattaa kaavaa `/key=value/`. Mikäli valittu partitioiden määrä olisi 4, niin eri juoksevat id-numerot päätyisivät neljään eri partitioon (ks. Taulukko 6).

id	Jakojäännös 4:llä	Partitiokansion nimi
1	1	/par=1/
2	2	/par=2/
3	3	/par=3/
4	0	/par=0/
5	1	/par=1/

Taulukko 6. ID:n muunnos partitionumeroksi, kun valittu partitioiden määrä on neljä, ja partitiointistrategia perustuu jakojäännökseen.

Kun taulua myöhemmin luetaan levyltä esimerkiksi MERGE-operaatiota varten tai tiettyä id:tä etsiessä SELECT-lausekkeella, voi tätä tietoa hyödyntämällä kohdentaa haku vain osaan datasta. Jos 300 gigatavun taulu on partitioitu kolmeensataan yhden gigatavun partitioon, niin Sparkin tarvitsee skannata vain yksi gigatavu tiedostoja 300 gigatavun sijasta, mikäli SELECT-lausekkeessa mukana on tarvittava filtteri, kuten `”WHERE par = 2”`. Taulun voi partitioida myös muita sarakkeita kuin taulun primääriavainta käyttäen, jolloin yksi vahva kandidaatti olisi `created`-päivämäärästä johdetut sarakkeet `year` ja `month`. Mikäli `partitionBy(”year”, ”month”)` olisi käytössä, niin tammikuun 2021 data päätyisi lokaatioon `year=2021/month=1/`. Kokemukseni mukaan tämä on tehoton tapa kirjoittaessa, sillä operaatiot kohdistuvat usein myös historiadataan, mutta valtaosa CDC-datan operaatioista osuu vain tuoreimpiin kuukausiin, jolloin `MERGE`-operaatio kohdentuu eri partitioihin epätasaisesti. Tämä kuorman epätasaisuus (eng. skew) aiheuttaa sen, että enemmistö laskentatehtävistä keskittyy parille työläisille, mikä on epätehokasta. Tauluja lukiessa tämän sortin skeemat ovat kuitenkin hyviä, sillä tyypilliset haut sisältävät useissa tapauksissa aikarajauksen, joten hopea- ja kultatason taulut kannattaa partitioida näiden usein käytettyjen SQL-filttereiden mukaisesti. Pronssitasoon kohdistuu merkittävä määrä

eri operaatioita (`INSERT`, `UPDATE`, `DELETE`), ja varsinkin poistot voivat osua hyvinkin vanhaan dataan, eikä tauluja ole lähtökohtaisesti tarkoitettu loppukäyttäjille, joten taulujen optimointi kirjoittamista suosien on perusteltua.

On mainitsemisen arvoista, että joissakin tuotantokannoissa voi olla päivämääriä, jotka ovat ajassa ennen 15. lokakuuta 1582 tai aikaleimoja, jotka ovat ennen 1. tammikuuta 1900, mikäli käyttäjä saa vapaavalintaisesti asettaa aikoja tai mikäli ohjelmiston kehittäjät ovat käyttäneet jotakin historiallista päivää Null-arvon korvikkeena. Mikäli näin on, Spark antaa varoituksen, sillä Spark 2.x käytti sekaisin sekä juliaanista että gregoriaanista kalenteria. Spark 3.0+ käyttää taaksepäin jatkettua eli proleptista gregoriaanista kalenteria. Mikäli olet varma, että jatkossa tietotaulua lukevat vain Spark 3.0+ ja muut kyseistä kalenteria ymmärtävät järjestelmät – tai mikäli historiallisilla päivämäärillä ei ole tarkempaa analyyttistä arvoa – voit turvallisesti asettaa arvon `spark.sql.legacy.parquet.datetimeRebaseModeInWrite` tilaan ”`CORRECTED`”. Mikäli haluat varmistaa taaksepäin yhteensopivuuden, aseta se ”`LEGACY`”-tilaan. [39.] `Timestamp`-tyyppisten sarakkeiden kanssa tulee myös ottaa huomioon, että Spark SQL olettaa kirjoittaessa, että kaikki ajat ovat UTC. Aikoja tulostaessa ajetaan aikaleiman konversio käyttäjän aikavyöhykkeeseen. Tähän voi vaikuttaa asetuksella `spark.sql.session.timeZone`.

Mikäli kertalataus suoritetaan demoympäristössä, pronssille kirjoitetaan Delta Lake -formaatissa kukin neljästä taulusta. Taulun polku määräytyy sovitun nimeämisstandardin mukaan. Repositorion esimerkkikoodi kirjoittaa taulut `DMSPathMergen`-luokan logiikan mukaisesti eri kansioihin (ks. Kuva 11). Kukin kansio sisältää Sparkin kirjoittamat tiedostot, jotka Parquet-tiedosto sekä Delta Lake -tallennuskerros tarvitsevat. Kukin kansio sisältää Parquet, CRC sekä JSON tiedostoja (ks. Kuva 12).

```
.\S3\bronze\abc\customers\customers\
.\S3\bronze\abc\customers\customer_details\
.\S3\bronze\abc\devices\device_models\
.\S3\bronze\abc\devices\devices\
```

Kuva 11. Github-repositorion esimerkkikoodieni käyttämä relatiivinen hakemistosijainti per lähdetaulu.

```
\_delta_log\00000000000000000000.json
\.part-00000-ded40f7e-cbc3-4a74-b015-54323bcd6570-c000.parquet.crc
\part-00000-ded40f7e-cbc3-4a74-b015-54323bcd6570-c000.parquet
```

Kuva 12. Github-repositorion esimerkkikoodini suorittamisesta syntyneet taulut customer_details taulun osalta.

Parquet-tiedosto sisältää taulun datan. Tiedoston nimi noudattaa kaavaa, jossa on muutama osatekijä. Näitä ovat partition sisäisen lohkon numero eli Spark Task Id (part-00000), 128-bittiseen satunnaislukuun perustuva 32-merkkinen UUID (8+4+4+4+12 heksanumeron ryhmissä), kirjoitetun tiedoston numero (c000) sekä tiedostopääte [40]. Esimerkissä käytetty taulu on hyvin pienikokoinen, joten koko taulu on kirjoitettu yhteen Parquet-tiedostoon. Tuotantokäytössä on tyypillistä, että yhtä UUID:tä eli yhtä Spark Jobia vasten on useita eri `part-####`-alkuisia tiedostoja.

Deltaloki (eli `_delta-log/)` on hyvin verrattavissa Git-versionhallinnassa ”.git”-kansion sisältöön. Tällä hetkellä kansiossa on vain yksi JSON-tiedosto, sillä olemme kommitoineet (eng. commit) vain yhden kirjoitusoperaation, joka kirjoitti kaikki taulun `customer_details` kymmenen riviä Parquet-tiedostoon. JSON-tiedoston sisältö on ihmisen luettavissa (ks. Liite 2). Huomaathan, että liitteen `schemaString`-arvo on korvattu kuvaavalla tekstillä, sillä varsinainen tekstiarvo olisi epäkäytännöllisen pitkä tulostettavaksi, ja lisäksi tiedostoon on lisätty lukemista helpottavat rivivaihdot – muutoin tiedosto on koskematon kopio alkuperäisestä JSON:sta. Databricksin ympäristössä luotu transaktioloki sisältää hieman enemmän informaatiota kuin avoimen lähdekoodin versio. Näitä tietoja ovat esimerkiksi operaation suorittaneen laskenklusterin, notebookin sekä käyttäjän nimet tai id:t.

CRC-tiedostoja syntyy kahdesta eri syystä ja eri lokaatioihin, mikä hämmensi ainakin minua aluksi. Databricksin software engineer Shixiong (Ryan) Zhu vastasi kysymykseeni Delta Slack -kanavalla, että Delta-taulusta löytyy kahdenlaisia CRC-päätteisiä tiedostoja. Ensimmäinen näistä on Hadoopiin kuuluvan ChecksumFileSystem:n kirjoittama checksum-tiedosto, jota käytetään datan validointiin. Tiedosto sisältää binääridataa eikä ole tarkoitettu ihmisen ymmärrettäväksi. Tämän luvun esimerkki on ajettu Windows-ympäristössä käyttäen virtuaalista Hadoop-tiedostojärjestelmää: siksi tuo CRC-tiedosto on luotu yllä. Toinen mainittu CRC-tiedosto syntyy, kun Databricksin ympäristössä suoritetaan kirjoitusoperaatio esimerkiksi S3-objektisäilöön. Tällöin CRC-tiedosto sijaitsee polussa `_delta_log/<delta_version>.crc`, jossa deltan versionumero on sama kuin vierellä olevan JSON-tiedoston. Tämän luvun esimerkkikoodi on ajettu paikallisella asennuksella eikä Databrickissä, joten deltalokissa sijaitsevaa CRC-tiedostoa ei ole olemassa. Zhun mukaan tiedosto on jäänne vanhasta bugista, joka aiheutti datan korruptoitumista, ja todennäköisesti tiedosto poistetaan tulevissa Databricks Delta -versioissa. [41.] Toisin kuin Hadoopin luoma CRC-tiedosto, tämä tiedosto on ihmiselle selkokielinen. Esimerkkidata on nähtävissä alla.

```
{
    "tableSizeBytes": 1048576,
    "numFiles": 1,
    "numMetadata": 1,
    "numProtocol": 1,
    "numTransactions": 1
}
```


Kuva 13. Databricks Spark-ympäristön luoma CRC-tiedosto Delta Laken lokikansiossa.

4.4.3 Jatkuva lataus

Mikäli aiemman luvun koodi on ajettu onnistuneesti, pronssitasolta löytyy tällä hetkellä DMS:n kertalatauksen taulut Delta Lake -formaatissa, joten pronssitaso on valmiina vastaanottamaan muutostietoa ajastettuina ajoina. Tieto haetaan latauslaiturin tiedostoista, jotka DMS on kirjoittanut sisäkkäisten `vuosi/kuukausi/päivä` kansioiden sisälle. Tiedostot sijaitsevat alikansioissa, joten meidän tulee aktivoida rekursiivinen tiedostojen haku ja määrittää tiedostonimelle glob-filtteri, joka estää LOAD-sanalla alkavien Parquet-tiedostojen hakemisen; muuten lataisimme saman tiedon kahdesti. Huomaa, että DMS:n luoma päivämääräkohtainen partitiointi ei noudata yllä esiteltyä kaavaa, jossa partitio on muotoa `key=value/`. Tämän takia Spark `DataStreamReader` ei oleta kansioiden olevan Parquet-tiedostolähteen partitioita. Kertalataus suoritettiin `spark.read`-komennolla, joka alustaa `DataFrameReader`-luokan, kun taas jatkuva lataus suoritetaan Structured Streaming API:n avulla, joka käyttää DataStreamReader-luokkaa. Merkittävä ero näiden välillä on se, että kertalataus lukee skeeman lähdekansion Parquet-tiedostoista, kun taas striimatun datan kanssa datan nuuskiminen tiedostoista on vastoin Apache Sparkin suosituksia [42]. Structured Streaming API:n nimi viittaa striimaukseen, mutta operaation voi suorittaa myös ajastettuina mikroerinä (eng. micro-batch), kun striimauskyselyn laukaisin (eng. trigger) asetetaan johonkin ei-jatkuvaan tilaan, kuten tilaan ”once”. Tällöin komponentin käyttämän logiikan voi esittää seuraavasti:

- Skannaa kaikki tiedostot lokaatiossa ”pm.staging”.
- Pidä tiedostot, jotka ovat uudempia kuin ”pm.checkpoint_path”-sijainnin kirjanmerkki.
- Aja funktio “merge_to_delta”. Parametrejä ovat DataFrame, joka sisältää valittujen tiedostojen datan, sekä id, joka on juokseva numero, joka on yhtä suurempi kuin nykyisen kirjanmerkin id.

- Päivitä kirjanmerkki.

Yllä listattua logiikkaa ohjaava Python-koodi on nähtävissä alla (ks. Koodi 5). Huomaathan, että funktio `merge_to_delta` on toistaiseksi jätetty tyhjäksi, jotta koodiesimerkki pysyisi lyhyenä. Nykyisellään dataa ei siis kirjoitettaisi mihinkään – vain kirjanmerkki päivittyisi, olettaen ettei operaatio kohtaa virheitä.

```
def merge_to_delta(batch_df, batch_id):
    # Apply event compaction and MERGE logic here.
    pass


# Schema is forced to match the target table.
schema = spark.read.format("delta").load(pm.bronze).schema
schema = schema.add("Op", "string")

# Init PathGenerator
pm = PathMerger(ss, db, table)

# Prepare Spark Auto Loader
df = ( spark.readStream
        .format("parquet")
        .option("recursiveFileLookup", "true")
        .option("pathGlobFilter", "[!L][!O][!A][!D]*.parquet")
        .schema(readers_schema)
        .load(pm.staging)
  )

# Get the files that have been added since the last run
streamingquery = (
    df
    .writeStream
    .trigger(once=True)
    .foreachBatch(merge_to_delta)
    .option("checkpointLocation", pm.checkpoint_path)
    .start()
)
```

Koodi 5. Inkrementaalisen latauksen suorittavan koodin esimerkkitoteutus. Merge to delta -funktion logiikka puuttuu, jotta koodi mahtuisi esille kerralla.

Nykyisellään yllä oleva koodi päivittäisi kirjanmerkkiä, mutta ei kirjoittaisi uutta dataa pronssille. Tämä prosessi tapahtuu `merge_to_delta` funktion sisällä. Funktio sisältää useita rivejä koodia, joten esittelen sen osio kerrallaan. Ensimmäinen osio (Koodi 6) lisää SQL-operaatiota kuvaavan numeron, jossa INSERT saa tärkeysarvon 1, UPDATE arvon 2 ja DELETE arvon 3. Logiikka tämän järjestysnumeron pohjalla on se, että operaatioiden on pakko tapahtua tässä järjestyksessä; riviä ei voi muokata ennen kuin se on lisätty, eikä rivejä voida muokata tai lisätä poistamisen jälkeen. Toinen lisätty kenttä on `dms_temp`. Alkuperäinen `dms_timestamp` on string-tyyppiä eli tavallista

tekstiä. Uusi kenttä sisältää `Timestamp`-tyyppisen päivämäärän, joka on parsittu AWS DMS:n kirjoittamasta tekstistä. Mikäli pronssitason taulu olisi partitioitu, tähän tauluun tulisi lisätä sama sarake. Partitiointiin käytetty sarake `par` on kommentoitu ulos risuaitamerkillä, koska taulu on niin pieni, ettei sitä tarvitse partitioida. Kaksi muuta saraketta `src_file` ja `src_batch_id` ovat hyödyllisiä mahdollisia vikatilanteita selvitellessä ja ovat mukana esimerkin vuoksi. Huomaa, että koodissa käytetty `F` on `spark.sql.functions` ja se tulee olla importoituna. Vaihtoehtoisesti ne voi importata yksitellen komennolla `from spark.sql.functions import when`. En suosittele lataamaan kyseisen moduulin funktioita jokerimerkkiä (`*`) käyttäen, sillä moduuli sisältää useita funktioita, joilla on päällekkäinen nimi Pythonin sisäänrakennettujen moduulien kanssa. Näitä ovat esimerkiksi `max` ja `abs`.

```
# Step 1 in merge_to_delta function.
df_batch = (
    df_batch
     .withColumn("op_numeral", F.when(F.col("Op") == "I", 1)
                               .when(F.col("Op") == "U", 2)
                               .when(F.col("Op") == "D", 3)
                               .cast("int")
                               )
    .withColumn('dms_temp', F.to_timestamp(F.col("dms_timestamp")))
    # .withColumn("par", F.col(all_pks[0]) % n_pars)
    .withColumn("src_file", F.input_file_name())
    .withColumn("src_batch_id", F.lit(batch_id))
)
```

Koodi 6. Otos `merge_to_delta` funktion logiikasta, joka lisää inkrementtiin kuuluvaan datasettiin sarakkeita.

Numeraalisen, operaatiota vastaavan arvon ja DMS:n aikaleiman lisäämisen jälkeen voimme koontaa tai tiivistää rivit siten, että niitä on vain yksi per id. Tavoitteena on pitää tuorein operaatio, mutta varmistaen, että DELETE voittaa UPDATE-operaation ja UPDATE voittaa INSERT-operaation, mikäli aikaleimat ovat samat. Alla oleva koodiesimerkki (ks. Koodi 7) suorittaa tämän operaation ja myötäilee Delta Laken dokumentaation esimerkkiä [43]. Koodi palauttaa aina vain yhden rivin per id, vaikka tietueessa olisi rivejä, jotka ovat keskenään täysin identtisiä. Koodiesimerkissäni käytetty `selectExpr`-lause palauttaa erikseen valittujen sarakkeiden osalta kaksi saman nimistä saraketta, koska `dms_temp` on sekä erikseen valittuna että sisältyy jokerivalitsimeen eli tähtisymboliin. Duplikaattisarakkeista ei kuitenkaan aiheudu ongelmaa alla olevissa koodiesimer-

keissä, koska näitä sarakkeita ei oteta mukaan MERGE-lausekkeeseen. Mikäli duplikaateilta haluaa kuitenkin välttyä, mikä voi olla tarpeen, jos DataFramen sisältöä halutaan tarkastella, voi sarakkeen nimetä valintalausekkeessa uudestaan erimerkiksi ”dms_temp as aaa”, kuten Github-repositorioni esimerkeissä on tehty.

Koodiesimerkki funktio `selectExpr` suorittaa annetut SQL-komennot DataFramea vasten ja palauttaa tuloksen [44]. Jälkimmäinen syötetty SQL-lauseke sisältää kuitenkin hyvin ei-tyypillisen SQL-funktion nimeltään `struct`, joka palauttaa Sparkin complex-datatyypin `StructType`. Tämä struct on ikään kuin lista, joka koostuu eri tyyppiä olevista kentistä. Tähän structiin, joka on koodissa nimetty `others`, kohdennetaan aggregaattifunktio `max`, joka palauttaa rivin per primääri- tai komposiittiavain. Mikäli `dms_temp` ei ratkaise voittajaa, vertaillaan seuraavaa arvoa, ja tätä jatketaan, kunnes ollaan viimeisessä sarakkeessa. Mikäli viimeisen sarakkeen kohdalla rivejä on yhä useita, duplikaatit pudotetaan pois. Tapa on tiiviin ja helppolukuisen koodin lisäksi huomattavan tehokas verrattuna siihen, että taulusta tehtäisiin ensin näkymä, joka yhdistettäisiin alkuperäiseen datasettiin JOIN-lausekkeella [45]. On huomattava, että tämä ratkaisu on hyväksyttävissä vain, mikäli tietokantaan ei tule useita UPDATE-muutoksia yhden sekunnin sisällä. MariaDB-tietokannan muutoshistoria ja siten myös DMS:n kirjoittama timestamp ovat sekunnin tarkkuudella, joten aikakoodin osalta samankalaisten rivien suhteen voittaja ratkaistaisiin siten, kummasta rivistä löytyy ensimmäiseksi suurempi arvo. Usein päivittyvistä tauluista tulee löytyä jokin kenttä, jota päivitetään aina päivitysoperaation yhteydessä, ja tämän kentän tietotyypin pitää mahdollistaa riittävä tarkkuus, kuten esimerkiksi yhden milli- tai mikrosekunnin tarkkuus.

```
# Step 2 in merge_to_delta function
latest_uniques = (
    df_batch
        .selectExpr(*all_pks, "struct(dms_temp, op_numeral, *) as others")
        .groupBy(*all_pks)
        .agg(F.max("others").alias("latest"))
        .select("latest.*")
)
```

Koodi 7. Otos `merge_to_delta`-funktion logiikasta, joka tiivistää rivit siten, että yhtä id:tä kohden on vain tuorein rivi läsnä.

Funktion kolmas ja viimeinen vaihe (ks. Koodi 8) listaa valitut sarakkeet, luo primääriavaimesta tai useasta avaimesta JOIN-operaation ehdon (eng. join condition) ja suorittaa luokan `DeltaTable` komennon `merge`, joka palauttaa `DeltaMergeBuilder` luokan olion. Olio asettaa ehdot sille, kuinka `when`-lausekkeita tulee käyttää, ja ne löytyvät luokan dokumentaatiosta [46]. Luokka `DeltaTable` tulee importoida polusta `delta.DeltaTable`. Koodi noudattaa logiikkaa,

joka on suoraan luettavissa näkyvistä ehdoista: mikäli primääriavainta vastaava rivi löytyy kohdetaulusta, ja operaatio on DELETE, rivi tuhotaan; mikäli primääriavainta vastaava rivi löytyy kohdetaulusta, ja operaatio on UPDATE, rivin arvot päivitetään; mikäli primääriavainta vastaavaa riviä ei löydy kohdetaulusta, rivi kirjoitetaan kohdetauluun, jos rivin operaatio on INSERT tai UPDATE, mutta ei DELETE. Myös UPDATE-operaatiolla korvamerkityt operaatiot tulee kirjoittaa kohdetauluun, koska on täysin mahdollista, että `df_batch` sisältää sekä INSERT että UPDATE-operaation samaa primääriavainta vasten. Tällöin viimeisintä uniikkia operaatiota (`latest_uniques`) tulisi kohdella INSERT-operaationa.

```
# Step 3 in merge_to_delta function

# Init the target table
target = DeltaTable.forPath(spark, pm.bronze)

# Using target schema, create map like: { "id": "s.id" }
col_map = {x.name: f"s.{x.name}" for x in target.toDF().schema}

# Using PKS, create map like "t.id = s.id AND t.foo = s.foo"
join_cond = " AND ".join([f"t.{pk} = s.{pk}" for pk in all_pks])

# Delta Merge
(
  target.alias("t")
  .merge(
          source = latest_uniques.alias("s"),
          condition = f"{join_cond}"
      )
      .whenMatchedDelete(condition = "s.Op = 'D')
      .whenMatchedUpdate(condition = "s.Op = 'U'", set = col_map)
      .whenNotMatchedInsert(condition = "s.Op != 'D'", values = col_map)
      .execute()
)
```

Koodi 8. Otos `merge_to_delta`-funktion logiikasta, joka valmistelee ja suorittaa MERGE-operaation.

Otetaan tarkasteluun aiemmin luotu `devide_models` taulu, jonka sisältö on nähtävissä alla (ks. Taulukko 7), ja kohdistetaan siihen muutoksia. Koodi, jolla taulu on luotu, löytyy Github-repositoriostani [32]. Taulu on kertalatauksella luotu, joten `src_batch_id` sarake sisältää ainoastaan `null` arvoja. Taulun fyysinen polku on `S3\bronze\abc\devices\device_models\` ja sen osoite Hive-katalogissa on `bronze.abc_device_devicemodels`. Yksittäinen taulun rivi edustaa yhtä tuotetta. Kaikki tuotteet ovat kuviteltuja tuotteita. Rivi, jonka `id` on 1, on punainen Super Gadget 100 -tuote, joka on julkaistu 15. toukokuuta 2010. Tuote on lisätty tietokantaan muutama kuukausi ennen julkaisua eli 21. maaliskuuta 2010. DMS on ladannut nämä historialliset rivit

vuonna 2021. Muutokset, joita tähän tauluun on lähdekannassa kohdistunut kertalatauksen jälkeen, on nähtävissä alla (ks. Taulukko 8). Tämä lastauslaiturilta löytyvä tieto on generoitu Pythonilla, mutta muistuttaa muodoltaan DMS:n luomaa tiedostoa, olettaen että asetukset ovat samat kuin aiemmissa luvuissa on määritelty. Yksittäinen rivi edustaa yhtä operaatiota. Rivin kaikilla sarakkeilla on arvo, vaikka varsinainen UPDATE-operaatio olisi muuttanut vain yhtä saraketta. Muutoshistoriadataa tarkkailemalla voi huomata, että osa muutoksista kohdistuu samoihin kohdetaulun riveihin. Tämän voi päätellä id-kentän arvoista. Muutokset voi purkaa aikajärjestyksessä lauseiksi. Kuvitteelliset aamupäivän tapahtumat ovat:

- Kello 11:37:17
    - Tauluun on lisätty saman sekunnin sisällä kaksi uutta tuotetta: musta ja vaaleanpunainen Super Gadget 300.
- Kello 11:37:47 eli puoli minuuttia äskeistä myöhemmin
    - Kaksi vanhaa tuotetta on päivitetty. Tuotekuvauksen sisältävän sarakkeen vanha arvo ”lorem ipsum” on päivitetty arvoihin ”update id 1” ja ” upddddddate id 2” [sic].
- Kello 11:39:29
    - Mustan Super Gadget 100:n tuotekuvauksessa on havaittu kirjoitusvirhe eli kuudesti toistuva d-kirjain. Se on korjattu.
- Kello 11:43:31
    - Tuotehallinnossa on huomattu, että musta Super Gadget 300 -tuote pitää poistaa taulusta. Kenties tuotetta valmistetaan vain vaaleanpunaisena, ja musta oli vanhasta skriptistä jäänyt jäänne.

dms_timestamp	id	release_date	name	color	description	created	modified	src_batch_id
2021-09-11 11:30:04	1	2010-05-15	Super Gadget 100	Red	lorem ipsum	2010-03-21 12:00:01	2010-03-21 12:00:01	NaN
2021-09-11 11:30:04	2	2010-05-15	Super Gadget 100	Black	lorem ipsum	2010-03-21 12:00:02	2010-03-21 12:00:02	NaN
2021-09-11 11:30:04	3	2010-11-01	Super Gadget 100	Pink	lorem ipsum	2010-08-05 07:00:00	2010-08-05 07:00:00	NaN
2021-09-11 11:30:04	4	2018-05-13	Super Gadget 200	White	lorem ipsum	2018-03-20 12:01:01	2018-03-20 12:01:01	NaN

Taulukko 7. Taulun device_models sisältö kertalatauksen jälkeen.

Op	dms_timestamp	id	release_date	name	color	description	created	modified
I	2021-09-11 11:37:17	5	2021-12-31	Super Gadget 300	Black	new device	2021-09-11 11:37:17	2021-09-11 11:37:17
I	2021-09-11 11:37:17	6	2021-12-31	Super Gadget 300	Pink	new device	2021-09-11 11:37:17	2021-09-11 11:37:17
U	2021-09-11 11:37:47	1	2010-05-15	Super Gadget 100	Red	update id 1	2010-03-21 12:00:01	2021-09-11 11:37:47
U	2021-09-11 11:37:47	2	2010-05-15	Super Gadget 100	Black	upddddddate id 2	2010-03-21 12:00:02	2021-09-11 11:37:47
U	2021-09-11 11:39:29	2	2010-05-15	Super Gadget 100	Black	update id 2	2010-03-21 12:00:02	2021-09-11 11:39:29
D	2021-09-11 11:43:31	5	2021-12-31	Super Gadget 300	Black	new device	2021-09-11 11:37:17	2021-09-11 11:43:31

Taulukko 8. Tauluun kohdistuva muutosdata lastauslaiturilla. Taulun järjestykseksi on valittu dms-aikaleima nousevasti.

Huomaa, että `dms_timestamp` ja `modified` kenttien arvot ovat samat tai hyvin lähelle samat; tämä on luonnollista, koska DMS lukee rivit MariaDB:n binäärilokista yleensä hyvin pian operaation suorittamisen jälkeen. Mikäli DMS:n ja tietokantapalvelimen välillä olisi tietokatkos, näiden arvojen välille syntyisi viivettä.

Aiemmin selitetty `latest_uniques`-käsittely eli `merge_to_delta` funktion keskimmäinen osio tiivistää tai koontaa taulun tapahtumat siten, että tapahtumia on vain yksi per id. Tuo välivaihe, joka annetaan myöhemmin `merge`-operaatiolle syötteeksi, on nähtävissä alla (ks. Taulukko 9), ja näiden rivien op-kentän arvot määräävät, kuinka `merge`-operaation `when`-lausekkeet niitä käsittelevät. Aiemmin esitelty ACID-vaatimus atomisuudesta täyttyy, sillä operaatio tapahtuu joko kokonaisuudessaan tai ei laisinkaan. Vaatimus eheydestä täyttyy, sillä kirjoitusoperaation päätteeksi deltaloki päivitetään siten, että jatkossa taulu palauttaa tuoreimman version. Mikäli toinen Spark-instanssi lukee tätä taulua kaiken aikaa, se jatkaa lukemista taulun versiosta nolla. Mikäli kirjoitusoperaation jälkeen aloitetaan uusi lukuoperaatio, se kohdistuu versioon yksi. Tämä merge-operaation jälkeinen versio on nähtävissä alla (ks. Taulukko 10). On tärkeää muistaa, että `DataStremaWriter`-oliota alustaessa sille annettiin määre `checkpointLocation`. Tämän polun määräämästä lokaatiosta löytyvät kansiot `commits`, `offsets` ja `sources`. Tiedostosta `offsets/0` löytyy JSON-dataa, ja yksi avaimista on `batchTimestampMs`, jonka arvo on esimerkiksi `1631428148193`. Tämä `merge`-operaation aikaleima kertoo, kuinka monta millisekuntia on kulunut UNIX-epookin alusta eli tammikuun ensimmäisestä päivästä vuonna 1970. Mikäli striimausoperaatio ajetaan uusiksi, samoja tiedostoja ei käsitellä uusiksi; vain tätä aikaleimaa tuoreemmat tiedostot otetaan huomioon.

Op	dms timestamp	id	release date	name	color	description	created	modified	op numeral	src batch id
U	2021-09-11 11:37:47	1	2010-05-15	Super Gadget 100	Red	update id 1	2010-03-21 12:00:01	2021-09-11 11:37:47	2	0
U	2021-09-11 11:39:29	2	2010-05-15	Super Gadget 100	Black	update id 2	2010-03-21 12:00:02	2021-09-11 11:39:29	2	0
D	2021-09-11 11:43:31	5	2021-12-31	Super Gadget 300	Black	new device	2021-09-11 11:37:17	2021-09-11 11:43:31	3	0
I	2021-09-11 11:37:17	6	2021-12-31	Super Gadget 300	Pink	new device	2021-09-11 11:37:17	2021-09-11 11:37:17	1	0

Taulukko 9. Muutoshistoria `latest_uniques`-käsittelyn jälkeen. Järjestetty id:n mukaan.

dms_timestamp	id	release_date	name	color	description	created	modified	src_batch_id
2021-09-11 11:37:47	1	2010-05-15	Super Gadget 100	Red	update id 1	2010-03-21 12:00:01	2021-09-11 11:37:47	0.0
2021-09-11 11:39:29	2	2010-05-15	Super Gadget 100	Black	update id 2	2010-03-21 12:00:02	2021-09-11 11:39:29	0.0
2021-09-11 11:30:04	3	2010-11-01	Super Gadget 100	Pink	lorem ipsum	2010-08-05 07:00:00	2010-08-05 07:00:00	NaN
2021-09-11 11:30:04	4	2018-05-13	Super Gadget 200	White	lorem ipsum	2018-03-20 12:01:01	2018-03-20 12:01:01	NaN
2021-09-11 11:37:17	6	2021-12-31	Super Gadget 300	Pink	new device	2021-09-11 11:37:17	2021-09-11 11:37:17	0.0

Taulukko 10. Kohdetaulu MERGE-operaation jälkeen. Järjestetty id:n mukaan.

4.4.4 Vacuum

Jatkuvan latauksen suorittamisen jälkeen taulu `bronze.abc_devices_devicemodels` vastaa tuotantokannan alkuperäistaulua sillä ajanhetkellä, kun DMS kirjoitti Parquet-tiedoston lastauslaiturille. Tarkemmin sanottuna: taulun versio yksi vastaa tuotantokannan taulua tuolla hetkellä, ja versio nolla vastaa kertalatauksen aikaista versiota (ks. Taulukko 11). Edellinen taulun versio on yhä olemassa levyllä, mutta deltalokin tuorein versio viittaa tähän tauluun, joten taulua lukevat Spark-instanssit lukevat vakiona tuoreinta versiota. Mikäli taulussa olisi asiakkaiden henkilötietoja, jotka asiakas halusi poistettavaksi, ne olisivat yhä järjestelmässä tallessa edellisessä versiossa. Tieto ei ole tallessa pelkästään kylmänä varmuuskopiona vaan se on haettavissa. Aiempaan taulun versioon pääsee käsiksi kutsumalla sitä versionumerolla tai aikakoodilla. Tätä deltan aikamatkustukseksi (eng. time travel) kutsumaa ominaisuutta voi hyödyntää myös SQL-kielellä, mutta tätä toiminnallisuutta en ole saanut toimimaan nykyisellä avoimen lähdekoodin versiolla (pyspark 3.1.2, delta 1.0.0). En ole kohdannut samoja ongelmia Databricksin ympäristössä.

version	timestamp	operation	operationParameters	operationMetrics
0	2021-09-12 06:26:03.746	CREATE OR REPLACE TABLE AS SELECT	{'description': None, 'partitionBy': '[]', 'properties': '{}', 'isManaged': 'false'}	{'numOutputRows': '4', 'numOutputBytes': '2569', 'numFiles': '1'}
1	2021-09-12 06:29:13.334	MERGE	{'matchedPredicates': '[{"predicate":"(s.`Op` = 'D')","actionType":"delete"},{"predicate":"(s.`Op` = 'U')","actionType":"update"}]', 'predicate': '(t.`id` = s.`id`)', 'notMatchedPredicates': '[{"predicate":"(NOT (s.`Op` = 'D'))","actionType":"insert"}]'}	{'numOutputRows': '5', 'numTargetRowsInserted': '1', 'numTargetRowsUpdated': '2', 'numTargetFilesAdded': '6', 'numTargetFilesRemoved': '1', 'numTargetRowsDeleted': '0', 'scanTimeMs': '2358', 'numSourceRows': '4', 'executionTimeMs': '4477', 'numTargetRowsCopied': '2', 'rewriteTimeMs': '2114'}

Taulukko 11. Otos sarakkeista, jotka DESCRIBE HISTORY -komento palauttaa.

```
# Timestamp
t = '2021-09-12T06:27:00'

# Version
v = 0

# Queries
df1 = spark.read.option("timestampAsOf", t).table(pm.hive)
df2 = spark.read.option("versionAsOf", v).table(pm.hive)
```

Koodi 9. Kaksi eri tapaa kutsua taulun aiempaa versiota: aikakoodi sekä versionumero.

Taulun vanhoista versioita pääsee pysyvästi eroon ajamalla VACUUM-komennon. Yllä näkyvää historiaa (Taulukko 11) tutkimalla selviää, että deltataulun versio nolla sisälsi kuusi riviä mutta vain yhden tiedoston. Versio yksi sisältää viisi riviä, mutta kuusi tiedostoa. Kaiken kaikkiaan kansiossa on seitsemän Parquet-tiedostoa. Mikäli haluamme poistaa taulun version numero nolla pysyvästi, meidän tulisi tuhota tiedosto, joka on merkattu deltalokissa poistetuksi. Jos JSON-tiedostosta löytyvät `add` ja `remove` avaimet ja niiden sisälle tallennetut `path` arvot tulostaa, saa helppolukuisen listan deltataulun tuoreimpaan versioon kuuluvista tiedostoista (ks. Kuva 14). Tiedostoja ei kuitenkaan tarvitse hallita käsin, vaan vanhat tiedostot voi poistaa SQL-komennolla `VACUUM bronze.abc_devices_devicemodels`. Oletusasetuksilla komento poistaa yli 168 tuntia vanhemmat tiedostot, jotka eivät ole käytössä taulun tuoreimmassa versiossa. Mikäli tiedostoja halutaan säilöä pidempään, mikä voi olla tarpeen jos ETL-tehtävä ajastetaan kerran viikossa, niin komentoon lisätään perään `RETAIN num HOURS`, jossa `num` tulee korvata haluamallaan kokonaisluvulla. Alle 168 tuntia tuoreempien tiedostojen poistaminen aiheuttaa virheilmoituksen, jonka voi kiertää asettamalla Sparkin konfiguraatioon arvon `"false"` asetukselle `spark.databricks.delta.retentionDurationCheck.enabled`. VACUUM-komennon

ajaminen ei ainakaan kirjoitushetkellä ole Databricksissä automatisoitavissa siten, että komento suoritettaisiin esimerkiksi aina, kun tauluun kohdistuu muutoksia.

```
[INFO] Mark files that were removed from delta table during commit:
[x] part-00000-5a0ae9da-f510-4400-8377-9a1ed88cb259-c000.parquet          2.51 KB
[ ] part-00000-d5cf549d-bdb9-4d07-9eb6-a84aeb5a4aaa-c000.snappy.parquet   0.98 KB
[ ] part-00045-2193ab89-242b-413d-811c-9bd3f6d19bdc-c000.snappy.parquet   2.66 KB
[ ] part-00069-78a281b9-e5c6-4d90-abf1-3a9fe31afb59-c000.snappy.parquet   2.66 KB
[ ] part-00107-513e811f-e296-4c9b-89f8-c204a3198269-c000.snappy.parquet   2.62 KB
[ ] part-00128-2d0c9196-a633-4efb-9645-758b76739c26-c000.snappy.parquet   2.68 KB
[ ] part-00140-8a607eac-4b5f-4b41-bab3-98e085012dac-c000.snappy.parquet   2.63 KB
```

Kuva 14. Deltalokin JSON-tiedostoja parsimalla selviää, että yhteen tiedostoon on kohdistunut poisto-operaatio ja loput kuusi tiedostoa on lisätty.

4.4.5 Optimize

Aiemmassa esimerkissä on huomioimisen arvoista, että deltataulun versio nolla eli kertalatausversio sisälsi vain yhden tiedoston. `Merge`-operaation jälkeinen versio sisältää kuusi tiedostoa, mikä tässä tapauksessa tarkoittaa sitä, että taulun jokainen rivi on eri tiedostossa. Ongelma johtuu siitä, että `merge`-operaatio aiheuttaa datan siirtelyä (eng. shuffle) Spark-suorittajien (eng. Spark executors) välillä. Tämän `shuffle`-operaation sivuvaikutuksena alkuperäinen data pilkotaan RDD-partitioihin eli muistinsisäisiin partitioihin, joiden vakiomäärä on 200. Vakiomäärää voi säätää asetuksella `spark.sql.shuffle.partitions`. Minun demoympäristössäni on vain yksi tietokone, mutta tietokoneessa on 12-säikeinen prosessori. Eri prosessorin säikeet ottavat työn alle eri RDD-partitiot ja päätyvät kirjoittamaan ne eri tiedostoihin. Databricks Spark sisältää komennon `OPTIMIZE`, joka kirjoittaa taulun uudestaan kompaktissa muodossa, pyrkien noin 1 gigatavun kokoisiin tiedostoihin. Lisäksi taululle tai Spark-sessiolle voi kytkeä päälle ”Auto Optimize”- ja ”Auto Compact”-ominaisuuden, jolloin tauluun kohdistuvat kirjoitusoperaatiot optimoidaan siten, että tiedostot pyritään pitämään noin 128 megatavun kokoisina. Pronssitason tauluihin kohdistuu suuri määrä MERGE-operaatioita, jotka aiheuttavat pienten tiedostojen ongelmaa. Pronssitauluihin tai niille kirjoittaville Spark-sessioille on siis kannattavaa kytkeä tuo ominaisuus päälle, mikäli käytössä on Databricksin Spark-ympäristö. `OPTIMIZE` on maksullisen Databricks-ympäristön ominaisuus, joten avoimen lähdekoodin delta lakessa pienten tiedostojen ongelman hoitaminen on täysin käyttäjän vastuulla, ja keinoina siinä ovat `repartition` ja `coalesce` funktiot [47].

4.4.6 Skeeman evoluutio

Demoympäristöni jatkuvan latauksen esimerkistä puuttuu OPTIMIZE-komennon korvaajan lisäksi toiminallisuus, joka mahdollistaisi automaattisen skeeman evoluution. Aiemmin määritelty logiikka kaappaa skeeman pronssitasolta komennolla `spark.read.load(pm.bronze).schema`. Tätä skeemaa käytetään CDC-datan lataamiseen lastauslaiturilta. Mikäli lastauslaiturin Parquet-tiedostoihin on tullut uusia kenttiä, nämä eivät päädy pronssitasolle laisinkaan. Toisin kuin `OPTIMIZE`, tätä ongelmaa ei ole ratkaistu maksullisessa Databricksin ympäristössä ainakaan kirjoitushetkellä, jolloin tuorein Databricks Runtime versio on 9.0. Skeeman evoluutio JSON-tiedostoille lisättiin DBR 8.2:ssa ja CSV-tiedostoille 8.3:ssa, mutta Parquet-tiedostoja ladattaessa logiikka tulee kirjoittaa itse. Ongelman voi ratkaista esimerkiksi siten, että `schema` muuttujan alustamisen jälkeen etsitään ja luetaan taulun tuorein CDC-tiedosto lastauslaiturilta ja verrataan skeemoja. Mikäli eroavaisuuksia löytyy, skeemat voi joko yhdistää automaattisesti tai skriptin suorittamisen voi pysäyttää aiheuttamalla virheilmoituksen (eng. raise exception). Automaattista skeeman yhdistämistä en suosittele tuotannossa.

4.4.7 Lokin luominen

Muutostiedon ja sen sisältämien operaatioiden (`INSERT`, `UPDATE`, `DELETE`) määrä on hyödyllistä tietoa itsessään. Mikäli tauluihin kohdistuvista operaatioista haluaa piirtää kuvaajia, on hyödyllistä kerätä tämä tieto talteen jokaisen ajon yhteydessä. Alla on typistetty esimerkki, jossa entuudestaan tuttuun `merge_to_delta` funktioon lisätään rivejä (ks. Koodi 10). Aiemmin tunnettu logiikka, millä muodostetaan `latest_uniques` ja suoritetaan `merge`, on jätetty pois, jotta esimerkki pysyisi lyhyenä. Tieto kirjoitetaan globaaliin muuttujaan nimeltään `response`, jotta se saadaan käyttöön funktion ulkopuolella. Myöhemmin arvot voi kirjoittaa esimerkiksi erilliseen deltatauluun, kuten `spark_logs.dms_batch` tai ulkopuoliseen tietokantaan. Tauluun lisätään vain uusia rivejä ilman poistoja, joten sitä voi kirjoittaa ilman monimutkaista MERGE-logiikkaa komennolla `spark.write.mode(”append”).write(path)`. Tauluun kannattaa lisätä myös muuta hyödyllistä tietoa, kuten ajetun skriptin nimi ja versio, aloitus- ja lopetusajankohdat sekä deltataulun koko levyllä. Deltataulun koon saa selville komennolla `DESCRIBE TABLE`. Tavujen määrä eli taulun nykyisen version koko levyllä lukee sarakkeessa `sizeInBytes`.

```
response = {}

def merge_to_delta(batch_df, batch_id):

    # Init operation counts
    op_counts = {"I": 0, "U": 0, "D": 0}

    # Get operation counts using Cube
    for r in latest_uniques.cube("Op").count().collect():
        op_counts[r["Op"]] = r["count"]

    # Set the response
    global response
    response["stats_cdc_inserts"] = op_counts["I"]
    response["stats_cdc_updates"] = op_counts["U"]
    response["stats_cdc_deletes"] = op_counts["D"]
    response["batch_id"] = batch_id
```

Koodi 10. Muutostiedon lokitus jatkuvan tiedon latauksessa.

5 Data-altaan käyttö tietoalustana

Aiemman luvun listaamien vaiheiden tuloksena on pronssitaso, johon yhdistetään muutostieto ajastetusti. Pronssin taulut edustavat mahdollisimman raakaa kopiota lähdekantojen tauluista. Mikäli lähdekanta on kolmannessa normaalimuodossa tai muutoin pitkälle normalisoitu, toisiinsa liittyvä tieto löytyy useista eri tauluista. Tämä on tuotantokantojen toiminnallisuuden kannalta tärkeää, mutta hankalaa loppukäyttäjän kannalta.

Polarilla minä ja kollegani päädyimme ratkaisuun, jossa hopeakerros edustaa Ralph Kimballin [1] ohjeita myötäilevää mallia: tämä saavutetaan pääasiassa suorittamalla JOIN-operaatioita tauluille, jotka liittyvät primääri- ja viiteavainten avulla toisiinsa. Toisin kuin edellinen luku, tämä luku on kirjoitettu täysin Databricks-käyttökokemuksen pohjalta ja pohjaa teoriaa enemmän käytännön kautta opittuihin parhaisiin käytäntöihin.

5.1 Taulujen määrittely hopeatasolle

Aiemman kappaleen esimerkissä olleet taulut `customers_details`, `customers`, `devices` ja `devices_models` liittyvät kaikki toisiinsa primääri- ja viiteavainten kautta. Taulujen normalisoinnin aste ei edusta tyypillisen tuotantokannan taulurakennetta, joten niistä saa helposti rakennettua esimerkiksi dimensiotaulun `silver.dim_devices` (ks. Koodi 11 ja Koodi 12). Dimensiomallin luomiseen liittyy sekä taulujen (dimensioiden ja faktojen) että sarakkeiden nimien päättäminen. Tämä prosessi on yrityksen sisäistä työtä, joten en käsittele sitä tässä sen tarkemmin. On huomattavaa, että alla olevissa koodiesimerkeissä taulu partitioidaan sarakkeen `device_name` mukaan. Toisin kuin pronssitaso, joka on partitioitu kirjoittamisoperaatiota ajatellen, hopeataso kannattaa partitioida loppukäyttäjien tyypillisten kyselyiden mukaan. Ehdotettu partitiointi on kannattava, mikäli haut sisältävät useimmiten `WHERE device_name = ”Super Gadget 100”` -tyylisen filtterin.

Luodussa `dim_devices` taulussa on yksi rivi per yksittäinen rekisteröity laite. Esimerkissä esitelty partitiointi on kannattavaa, mikäli yksittäistä laitemallia edustavaa tietoa on vähintään yksi gigatavu. Kokemukseni mukaan se, kuinka tauluja kannattaa partitioida ja käsitellä, riippuu paljolti taulujen koosta. Esimerkissä taulu kirjoitetaan kokonaisuudessaan uudestaan. Mikäli dataa on useita teratavuja, kannattaa harkita esimerkiksi vuoden lisäämistä ensimmäiseksi partitiointiavaimeksi ja `replaceWhere`-asetuksen käyttämistä kirjoitusoperaatiossa.

```sql
SELECT
    A.id,
    A.serial_number,
    A.created as first_registration_date,
    A.modified as previous_registration_date,
    CASE
        WHEN C.id IS NULL THEN 'Not currently registered'
        ELSE 'Registered to a user'
    END AS current_registration_status,
    B.release_date,
    B.name as device_name,
    B.color as device_color,
    B.description as device_description
FROM bronze.abc_devices_devices A
LEFT JOIN bronze.abc_devices_devicemodels B ON A.model_fk = B.id
LEFT JOIN bronze.abc_customers_customers C ON A.customer_fk = C.id
```

Koodi 11. SQL-lauseke, jonka palauttama taulu olisi hyvä ehdokas hopeatason "devices"-taululle.

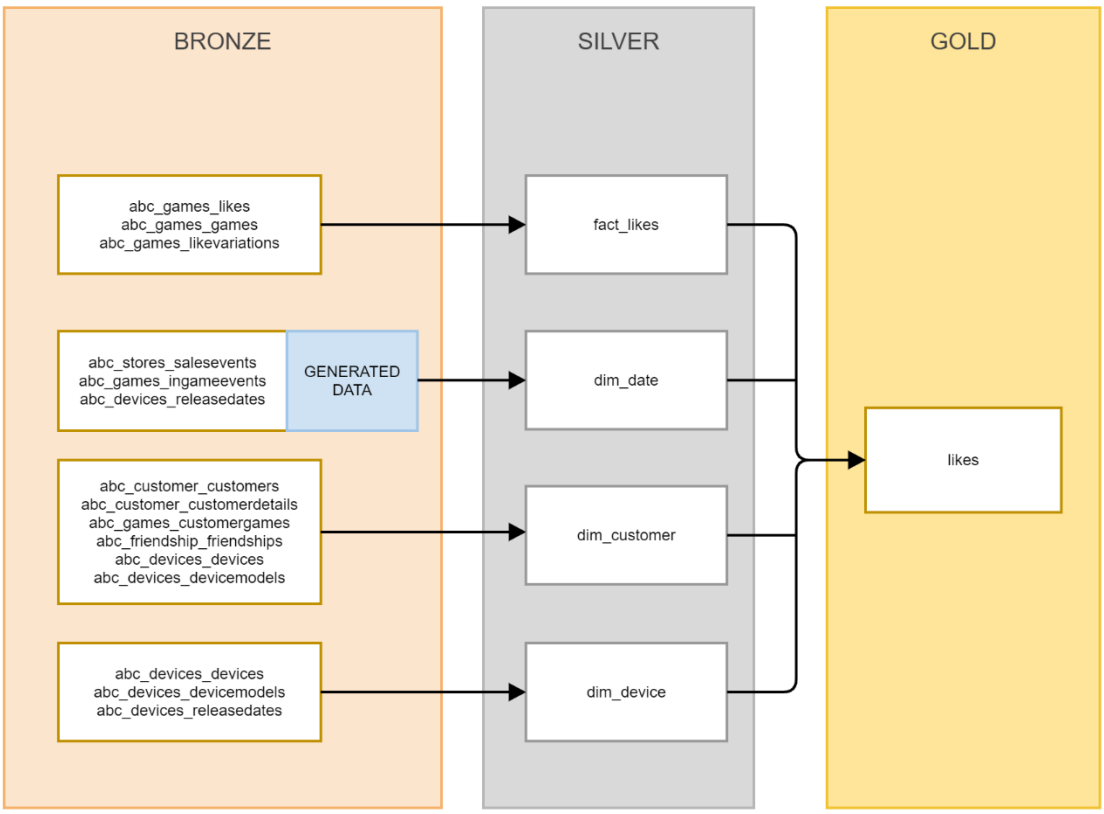
```python
# Load
df = spark.sql(query)

# Write
(
    df
    .write
    .format('delta')
    .mode('overwrite')
    .option('overwriteSchema', 'true')
    .option('path', '/mnt/silver/dim_devices')
    .partitionBy('device_name')
    .saveAsTable('silver.dim_devices')
)
```

Koodi 12. Python-rivit, jotka lukevat ja tallentavat halutun SELECT-lausekkeen hopeatason tauluksi.

Aiemman esimerkin suhteen on hyvä muistaa, että edellisen luvun esimerkkikannat ovat poikkeuksellisen yksinkertaisia rakenteeltaan. Todellisessa tuotantokäytössä JOIN-operaatio voi olla huomattavasti monimutkaisempi sekä vaatia monimutkaista logiikkaa. Databricksissä tämän logiikan ajaminen on mielestäni kohtalaisen helppoa: alustassa voi käyttää Pythonia ja SQL:ää rinnakkain tarpeen mukaan. Todellisempi esimerkki taulujen sukupuusta (eng. data lineage) sisältää useita tauluja pronssitasolta sekä generoitua tai muualta tuotua tietoa (ks. Kuva 15). Kuvan esimerkissä oletetaan, että kultatasolla on tarve asiakkaiden tykkäyksiä kuvastavalle erittäin leveälle taululle, johon on JOIN-operaation avulla yhdistetty tietoa eri dimensiosta. Tällainen taulu on loppukäyttäjälle helppo, sillä useimmat tyypilliset kyselyt hoituvat ilman JOIN-lausekkeita. Taulut materialisoidaan S3-säilöön Delta Lake -formaatissa, joten kokonaisuus on alusta loppuun asti tietoallas. Dataa ei koskaan siirretä data mart -tyyliseen tietovarastoon tai CUBE-muotoon. Mikäli

yksikään laskentaklusteri ei ole päällä, dataa voi silti lukea Delta Lake -lukijalla suoraan S3:lta eli tietoaltaasta.



Kuva 15. Kuvitteellinen esimerkki medaljonkiarkkitehtuurin taulujen sukupuusta, jossa faktataulu koostuu asiakkaiden tykkäyksistä ja dimensiotaulut tykkäyksiä kuvaavista tekijöistä, kuten asiakkaista, laitteista tai päivämääristä.

5.2 Pääsynhallinta Databricksissä

Databricks mahdollistaa SQL-kantajärjestelmistä tutun tavan hallita käyttäjiä ja käyttäjäryhmiä, jossa esimerkiksi `GRANT USAGE, SELECT ON TABLE silver.dim_device TO support_personnel` -lauseke antaisi asiakaspalvelun henkilökunnalle USAGE- ja SELECT-oikeudet aiemmin määriteltyyn dimensiotauluun, mahdollistaen heidän tarkastella taulua. Ominaisuus vaatii, että kaikki käyttö tapahtuu laskentaklusterin läpi, jossa on Table Access Control aktivoituna. Kyseinen ominaisuus rajaa haut siten, että loppukäyttäjillä ei ole suoraa pääsyä tiedostotasolla tauluihin, vaan operaatiot tulee suorittaa Python DataFrame API:n ja Spark SQL API:n läpi. Taulutasolla on mahdollista tehdä myös sarakkeisiin tai rivitason tietoon kohdistuvia rajauksia,

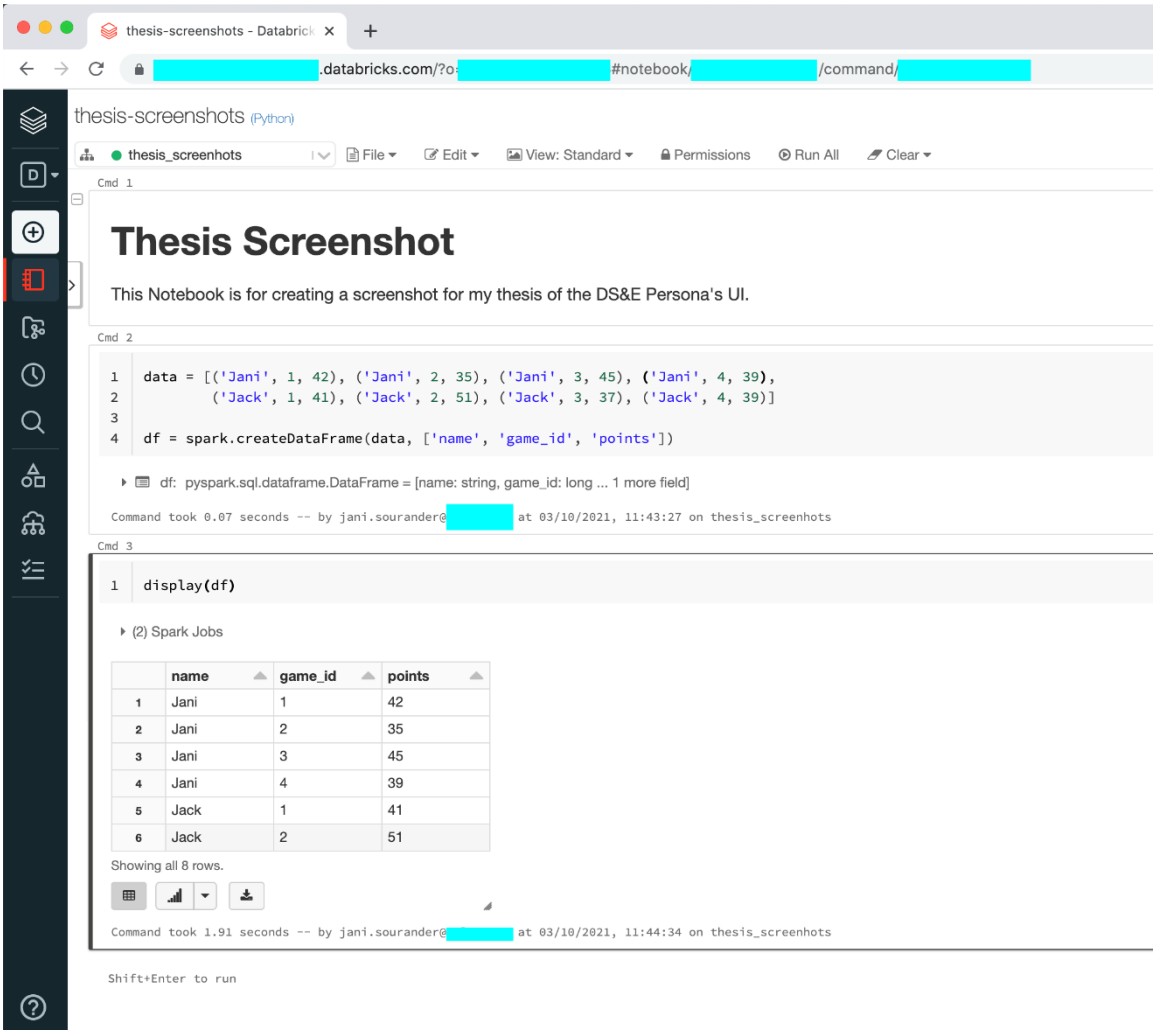
jolloin esimerkiksi laitteen sarjanumero näkyy vain tietyille ryhmille – muut pääsevät käsiksi vain sotkettuun (eng. hashed) sarjanumeroon, josta ei voi päätellä laitteen alkuperäistä sarjanumeroa. [48.]

5.3 Käyttöliittymä

The Databricks Lakehouse Platform jakautuu kolmeen osioon, joita alusta nimittää termillä ”Persona”. Nämä ovat: Data Science & Engineering (jatkossa DS&E), Machine Learning, SQL. Keskityn tässä luvussa DS&E:n ja SQL:n käyttöliittymiin, jättäen Machine Learning personan käsittelyn seuraavien opinnäytetöiden aiheeksi. Tyypillinen loppukäyttäjä käyttää kumpaakin osiota eli ”personaa” web-selaimen kuten Google Chromen läpi. Kirjautuminen ja käyttöliittymä sijaitsevat verkko-osoitteessa, joka on muotoa `<workspace>.cloud.databricks.com`, jossa `<workspace>` on luodun työympäristön nimi, kuten esimerkiksi `abc-marketing`. Yhdellä yrityksellä voi olla useita eri työympäristöjä esimerkiksi organisaatiorakenteen tai mikropalveluiden mukaisesti. Käyttäjäryhmille voidaan erikseen sallia pääsy DS&E- ja SQL-osioihin. Lisäksi käyttäjäryhmille voidaan erikseen sallia käyttöoikeus vain tiettyihin laskentaklustereihin, ja eri klustereilla voi olla pääsy vain tiettyihin S3-bucketteihin.

Tehokäyttäjät voivat lisäksi ottaa yhteyden Databricks Connectin avulla laskentaklustereihin. Tällöin kehitysympäristönä toimii tyypillinen IDE kuten PyCharm, IntelliJ tai VSCode: oma tietokone toimii ikään kuin driverinä, mutta Spark-jobit suoritetaan etänä. Databricks Connectiin liittyy useita rajoituksia, kuten se, että Table Access Control -aktivoidut laskentaklusterit eivät ole tuettuja. Databricks Connectin käsittely kattavasti on liian laaja aihe tähän opinnäytetyöhön.

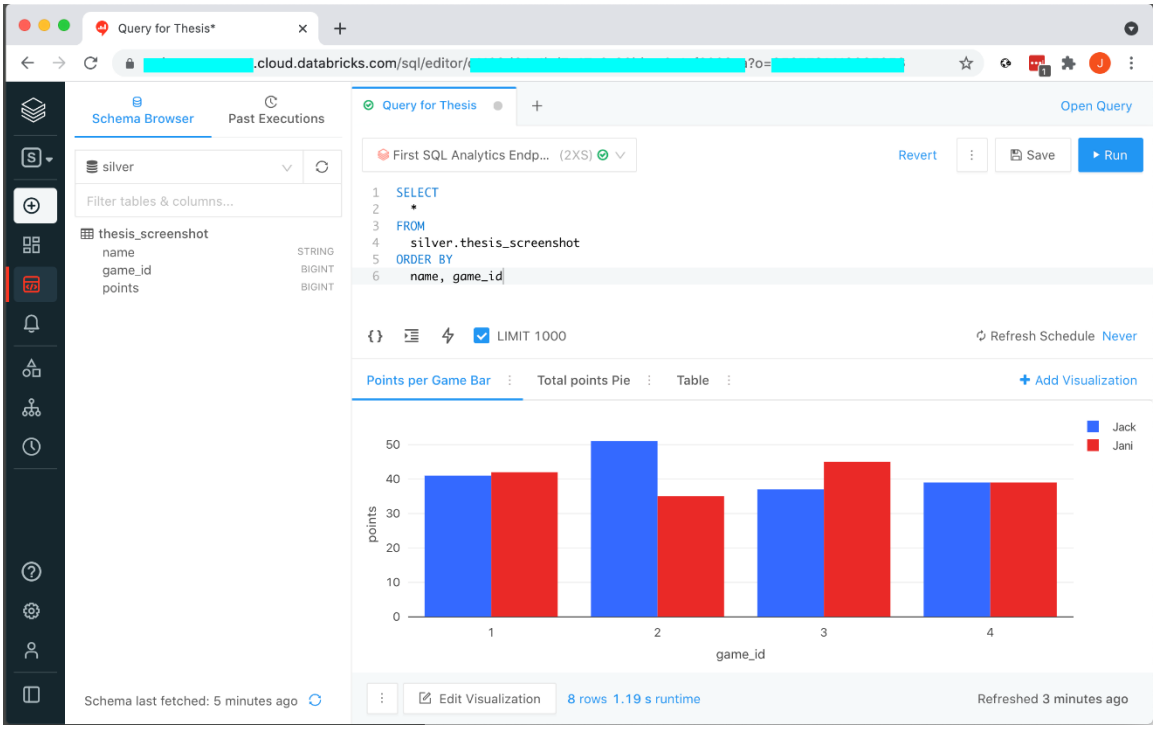
DS&E-osio käyttää laskentaan Compute-klustereita. Näiden laskentaklustereiden kokoonpano on pitkälti käyttäjän määriteltävissä. Koodi kirjoitetaan Jupyter Notebook -tiedostoihin (ks. Kuva 16). Mikäli klustereissa on Table Access Control aktivoituna, käyttäjät voivat tehdä hakuja vain sallittujen etuoikeuksien (eng. privileges) mukaisesti. Laskentaklusteriin voidaan sallia ODBC-yhteydet, jolloin queryjä voi muodostaa myös esimerkiksi PyCharmissa lokaalisti ajettavasta koodista. Kaksi aiemmin mainittua ominaisuutta on kytkettävissä päälle vain High-Concurrency-tyylisissä klustereissa: ei Standard tai Single-Node. Sparkin DataFrame API muistuttaa huomattavasti suositun Pandas-kirjaston käyttöliittymää; siitä huolimatta Sparkin käyttö vaatii opettelua heiltä, jotka ovat ennen tottuneet JDBC-yhteyksien ja Pandas-kirjaston varaan rakennettuihin ratkaisuihin.



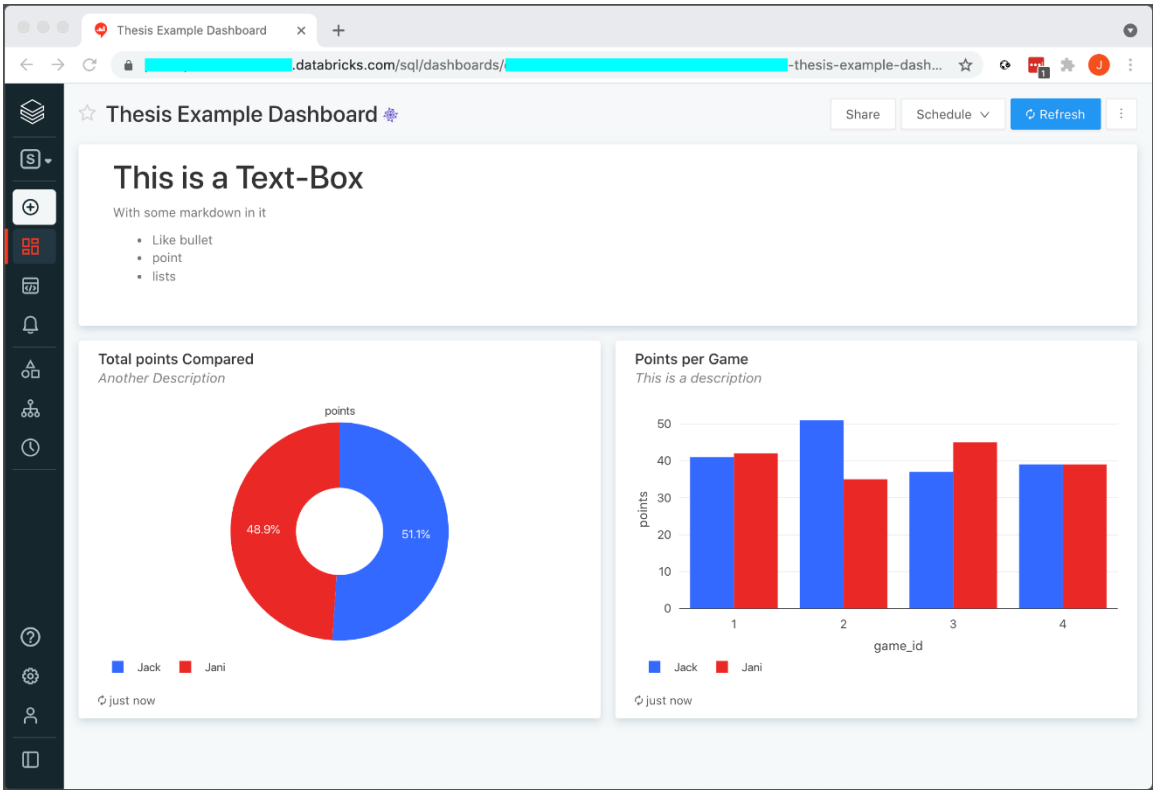
Kuva 16. Data Science & Engineerin -osion Notebook-näkymä. Koodi kirjoitetaan soluihin ja ajetaan solu-solulta.

SQL-osio käyttää laskentaan SQL Endpoint -nimisiä klustereita, jotka valitaan t-paitakokoja noudattavasta listasta. SQL-kieli on ainut tunnettu kyselykieli ja kyselyt kirjoitetaan Query Editor -työkalulle (ks. Kuva 17), joka on Databricksin ostamasta Redash-palvelusta omittu näkymä. Tallennettujen queryjen visualisoinnit voi kerätä Dashboardeiksi (ks. Kuva 18). Dashboardin käyttö ei vaadi loppukäyttäjältä ohjelmointitaitoja tai ymmärrystä Sparkin toiminnasta. Dashboardit vastaavat tyylillisesti alan tyypillisiä BI-raporttinäkymiä, joita voi luoda esimerkiksi Tableau- tai PowerBI-ohjelmistoilla, mutta ominaisuuksia on vähemmän kuin näissä BI-raportointiin erikoistuneissa työkaluissa. Nämä helposti lähestyttävät graafit lienevät useimmissa yrityksissä se, mitä yrityksen johto, myynti tai markkinointi eniten tuijottavat. Oheisissa kuvissa olevat esimerkit ovat äärimmäisen pelkistettyjä esimerkkejä. Queryihin ja Dashboardeihin voi liittää parametrejä, jotka perustuvat toisiin SQL-kyselyihin. Näin esimerkiksi Dashboardin ylälaidassa voisi sijaita vetova-

likko ”Product Names” ja kaksi kalenterivalitsinta: ”Starting Date” ja ”End Date”, joilla loppukäyttäjä voi muokata visualisointien alla olevien SQL-kyselyiden filttereitä tai funktioiden parametrejä. Työkalujen määrä on kuitenkin suppeampi kuin BI-työkaluissa. Niiden käyttö Databricksin kanssa on liian laaja aihe käsiteltäväksi tässä.



Kuva 17. SQL-osion Query Editor mahdollistaa SQL-kyselyiden ja niihin liittyvien visualisointien tallentamisen.



Kuva 18. SQL-osion Dashboard-näkymään voi koota useiden eri kyselyiden tuloksia ja visualisointeja.

5.4 Hakujen suorituskyky

Databricks Lakehouse ei ole tietovarasto vaan tietoaltaan päälle rakenneltu tietovarastoa muistuttava hybridi. Tästä huolimatta järjestelmä on hyvin suorituskykyinen. En voi antaa tarkkoja lukemia Polarin datamääristä, mutta esimerkiksi parin sadan gigan kultatason tauluun kohdistuvat haut palautuvat sekunnissa tai kahdessa, ja kyselyiden päälle rakennettu kymmenen infograafin BI-raportti latautuu muutamissa sekunneissa. Hive-pohjaisia tietovarastoja on kritisoitu siitä, että yksittäisen taulun partitioinnin voi optimoida vain yhtä käyttötarkoitusta (eng. access pattern) ajatellen, ja toissijainen indeksi puuttuu täysin [49]. Delta Lake -kerros ja Databricksin OPTIMIZE sekä Z-ORDER paikkaavat kokemukseni mukaan tätä natiivin Hadoop-ekosysteemin Hive-komponentin puutetta riittävästi. En ole vielä toistaiseksi törmännyt sellaiseen tauluun, jonka käyttö vaatisi kohtuuttoman suuren laskentaklusterin käyttöä. Tämä toki edellyttää, että kun tauluja materialisoidaan hopeatasolla ajastetuilla prosesseilla, niin näissä kertalaskettavissa tehtävissä lasketaan etukäteen sellaiset arvot, jotka vaativat leveitä lukuoperaatioita – kuten `window` tai `groupBy` funktioita. Delta Lake ja Spark mahdollistavat myös hyvin poikkeuksellisen tietomallin,

jossa yksittäinen sarake on muotoa array eli lista. Kokemukseni mukaan tämä on suoritustehokas tapa upottaa one-to-many -suhteella olevaa tietoa dimensiotauluun.

Oikean klusterin valinta eri työtehtäviin vaati alkuun opettelua, kuten myös se, että opin käyttämään JOIN-operaation vihjeitä. Databricksin YouTube-tilin Tech Talk -videoita seuraamalla tämän on kuitenkin oppinut. Alkuun sekava Spark UI, josta klusterin suorittamia tehtäviä voi seurata, alkoi tuntua jo parin kuukauden käyttökokemuksen jälkeen selkeältä ja hyödylliseltä työkalulta. Sparkin vakiona suosima JOIN eli ”sort-merge join” toimii kaikissa tilanteissa, mutta varsinkin dimensiomallissa tulee usein tilanteista, jossa ”shuffle hash join” tai ”broadcast join” ovat parempia valintoja, ja ainakin nykyisen Databricks Runtimen (DBR 9.1 LTS) kanssa nuo pitää osata määritellä itse. Pahimmillaan ei-optimaalisen JOIN-metodin valinta voi aiheuttaa sen, että työ kestää 50 minuuttia, kun se voisi olla parissa minuutissa suoritettuna. Sparkin suorituskyky on siis suurilta osin käyttäjän taidoista kiinni: järkevä tietomalli, jossa vaikeimmat laskusuoritukset ovat materialisoituina levylle, on loppukäyttäjän käsissä helppoa käytettävää.

Lakehouse -arkkitehtuurissa laskenta ja tallennus ovat eriytetyt, ja klusterin koon voi valita juuri kyseiseen tehtävään sopivaksi. Lopputuloksena palvelu on edelliseen Polarin tietovarastoratkaisuun verrattuna sekä nopeampi että edullisempi. Osa nopeudesta selittyy tietomalliin tehdyillä muutoksilla, mutta tämä ei sulje pois sitä, että pystyttämämme The Databricks Lakehouse Platform -toteutus näyttäytyy loppukäyttäjille nopeampana ja tehokkaampana kuin edeltäjänsä. Nykyisellään toteutus maksaa noin 20 prosenttia eli yhden viidenneksen edellisen järjestelmän hinnasta kuukausittain. Vuositasolla säästö on merkittävä.

6 Päätelmät

Tietovaraston korvaavan uuden järjestelmän kriteerejä olivat kustannustehokkuus, datalähteiden lisäämisen helppous, tuotteen maturiteetti, toimittajaloukun riskin vähyys, modulaarisuus sekä sopivuus koneoppimiseen ja muihin lähitulevaisuuden haasteisiin. Kokonaisuutena päivitys onnistui kaikkien kriteereiden osalta. Ilmiselvästi onnistunut osio on kustannustehokkuus: samat BI-rapotit aukeavat nopeammin kuin ennen, vaikka kustannukset ovat vain viidennes edellisestä. Myös toimittajaloukun riski on jäänyt toivotulla tavalla pieneksi. Tiedostot ovat avoimen lähdekoodin formaatissa Polarin omassa tietovarastossa ja niitä voi käsitellä muillakin työkaluilla kuin Databricksin alustan pystyttämillä klustereilla. Myös alustan soveltuminen koneoppimiskäyttöön tuntuu ilmiselvältä, vaikka sen käyttö on tätä kirjoittaessa osittain yhä opetteluvaiheessa.

Keskinkertaisesti onnistuneita tavoitteita ovat alustan maturiteetti ja datalähteiden lisäämisen helppous. Alustan maturiteetin huolet syntyvät siitä, että suuri osa Databricksin työkaluista on ”Public Preview”-tilassa. Datalähteiden suhteen halusimme ratkaisun, jossa lastauslaituri sisältää mahdollisimman vähän prosessoitua muutostietoa, mieluiten Parquet-tiedostomuodossa. Amazon Database Migration Service (DMS) onnistuu tehtävässään, mutta toivoisin helpommin konfiguroitavaa ja hallittavaa työkalua. Databricks ei toistaiseksi tarjoa datan sisäänvientiin muita työkaluja kuin Sparkin natiivit lukumetodit, joten CDC-datalähteet on pakko lisätä ulkopuolisilla työkaluilla.

Se, onko lopulta Databricks Lakehouse Platform merkittävästi parempi kuin muut vertaillussa mukana olleet palvelut, ei ole yhtä yksinkertainen kysymys kuin että onko uusi järjestelmä parempi kuin vanha. Työkalut ovat keskenään sen verran erilaisia, että niiden suora vertailu on täysin mahdotonta. Toisin kuin useat kilpailijansa, Lakehouse ei ole varsinaisesti tietovarasto vaan tietovaraston tavoin käyttäytyvä tietoallas. Datatiimillämme oli entuudestaan Python-osaamista, mikä vei SQL-pohjaiselta kilpailijalta pisteitä. Jokaisen yrityksen tulee arvioida Lakehouse-arkkitehtuurin ja Databricksin alustan sopivuus heidän käyttöönsä.

Nostin tekstissä esille useita jatkotutkimuksen aiheita, joista tärkeimmät ovat mielestäni muutostiedon kaappaamiseen soveltuvien työkalujen ja palveluiden vertailu, ohjelmistokehityksen hyvien käytäntöjen kuten CI/CD:n toteutus Databricks-ympäristössä, tietomallinnus Lakehouse-arkkitehtuurissa, Data Mesh -arkkitehtuuri sekä pääsyoikeuksien hallinta keskitetysti.

Lähteet

1 Kimball R, Ross M. The data warehouse toolkit: the definitive guide to dimensional modeling. Third ed. Indianapolis, IN: John Wiley & Sons, Inc.; 2013.

2 Jindal R. (16.9.2020). Difference between OLAP and OLTP in DBMS. Haettu 9.4.2021 sivustolta Geeks for Geeks. Internetosoite: https://www.geeksforgeeks.org/difference-between-olap-and-oltp-in-dbms/

3 Tillmann G. Usage-driven database design: from logical data modeling through physical schema definition. New York: Apress; 2017.

4 Kleppmann M. Designing Data-Intensive Applications. Sebastopol, California: O'Reilly Media; 2017.

5 Oppel AJ. Data modeling: a beginner's guide. New York: McGraw-Hill; 2010.

6 Ung D. (9.4.2019). Data Warehousing: Basics of Relational Vs Star Schema Data Modeling. Haettu 11.4.221 sivustolta Medium. Internetosoite: https://medium.com/@daryl.ung/data-warehousing-basics-of-relational-vs-star-schema-data-modeling-75a68eeaf0e3

7 Inmon W, Lindstedt D, Levins M. Data Architecture: A Primer for the Data Scientist. 2nd ed. London: Academic Press; 2019.

8 Damji J, Wenig B, Tathagata D, Lee D. Learning Spark. 2nd ed. Sebastopol: O'Reilly Media; 2020.

9 Handy T. (1.12.2020). The Modern Data Stack: Past, Present, and Future. Haettu 13.8.2021 sivustolta dbt blog. Internetosoite: https://blog.getdbt.com/future-of-the-modern-data-stack/

10 Singh C, Kumar M. Mastering Hadoop 3. Birmingham: Packt Publishing.

11 Emmanuel OT. (2.12.2019). What is Hadoop? Haettu 10.4.2021 sivustolta Medium: Better Programming. Internetosoite: https://betterprogramming.pub/what-is-hadoop-b90591ffae89

12 Birchard T. (25.4.2021). Learning Apache Spark with PySpark & Databricks. Haettu 15.5.2021 sivustolta Hackers and Slackers. Internetosoite: https://hackersandslackers.com/learning-to-use-apache-spark-pyspark/

13 Databricks. (11.11.2020). Data Collab Lab: Notes from the perf lab with fish and joe. Haettu 15.5.2021 sivustolta Youtube. Internetosoite: https://youtu.be/JsY9UP3SI5c

14 Intricity101. (12.9.2014). What is Hadoop?: SQL Comparison. Haettu 15.5.2021 sivustolta Youtube. Internetosoite: https://youtu.be/MfF750YVDxM

15 Amazon AWS. (17.4.2021). What is a data lake? Haettu 17.4.2021 sivustolta Amazon AWS. Internetosoite: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/

16 Databricks. (12.8.2021). Delta Engine. Haettu 12.8.2021 sivustolta Databricks Docs. Internetosoite: https://docs.databricks.com/delta/optimizations/index.html

17 Databricks. (16.10.2020). Databricks. Haettu 16.10.2021 sivustolta Internet Archive Wayback Machine. Internetosoite: https://web.archive.org/web/20201016194505/https://databricks.com/

18 Technics Publifications. (3.10.2021). Building the Data Lakehouse. Haettu 3.10.2021 sivustolta Technics Publifications. Internetosoite: https://technicspub.com/data-lakehouse/

19 Inmon B, Levins M. (19.5.2021). Evolution to the Data Lakehouse. Haettu 3.10.2021 sivustolta Databricks Blogs. Internetosoite: https://databricks.com/blog/2021/05/19/evolution-to-the-data-lakehouse.html

20 Hansen J. (1.4.2021). Selling the Data Lakehouse for a Data Cloud. Haettu 13.4.2021 sivustolta Medium. Internetosoite: https://medium.com/snowflake/selling-the-data-lakehouse-a9f25f67c906

21 Armbrust M, Das T, Sun L, et al. (1.8.2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Haettu 13.4.2021 sivustolta Databricks. Internetosoite: https://databricks.com/wp-content/uploads/2020/08/p975-armbrust.pdf

22 Databricks. (20.8.2021). Concurrency Control. Haettu 16.10.2021 sivustolta Databricks Docs. Internetosoite: https://docs.databricks.com/delta/concurrency-control.html

23 Strengholt P. Data Management at Scale. 2nd Release ed. Sebastopol, California: O'Reilly Media; 2020.

24 Dehghani Z. Data Mesh. Early Access 3rd Release ed. Sebastopol, California: O'Reilly Media; 2021.

25 Stitch. (18.4.2021). How to use change data capture to optimize the ETL process. Haettu 18.4.2021 sivustolta Stich. Internetosoite: https://www.stitchdata.com/resources/change-data-capture/

26 Wang C. (4.10.2021). ETL vs ELT: Choose the Right Approach for Data Integration. Haettu 16.10.2021 sivustolta Fivetran Blog. Internetosoite: https://fivetran.com/blog/etl-vs-elt

27 DataWorks Summit. (23.6.2014). Hadoop Summit 2014: Efficient Data Storage for Analytics with Parquet 2.0. Haettu 23.4.2021 sivustolta Youtube. Internetosoite: https://youtu.be/MZNjmfx4LMc

28 Amazon. (23.4.2021). Using Amazon S3 as a target for AWS Database Migration Service. Haettu 23.4.2021 sivustolta Amazon Docs. Internetosoite: https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Target.S3.html#CHAP_Target.S3.Configuring

29 Srivastava M. (19.1.2021). Principles of GDPR and its impact your Data Analytics Platform. Haettu 15.4.2021 sivustolta Towards Data Science. Internetosoite: https://towardsdatascience.com/principles-of-gdpr-and-its-impact-your-data-analytics-platform-2c64463b2ae5

30 Apache Spark. (6.7.2021). Data Types. Haettu 6.7.2021 sivustolta Apache Spark Docs. Internetosoite: https://spark.apache.org/docs/latest/sql-ref-datatypes.html

31 Amazon. (6.7.2021). Transformation rules and actions. Haettu 6.7.2021 sivustolta Amazon DMS Docs. Internetosoite: https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Tasks.CustomizingTasks.TableMapping.SelectionTransformation.Transformations.html

32 Sourander J. (18.9.2021). Thesis - Playground. Haettu 18.9.2021 sivustolta Github. Internetosoite: https://github.com/sourander/kamk-thesis

33 Heintz B, Lee D. (14.8.2019). Productionizing Machine Learning with Delta Lake. Haettu 5.8.2021 sivustolta Databricks Blog. Internetosoite: https://databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html

34 Databricks. (30.8.2021). Databricks architecture overview. Haettu 5.8.2021 sivustolta Databricks Docs. Internetosoite: https://docs.databricks.com/getting-started/overview.html

35 Suarez D. (22.9.2021). A love-hate relationship with Databricks Notebooks. Haettu 10.10.2021 sivustolta Towards Data Science. Internetosoite: https://towardsdatascience.com/databricks-notebooks-a-love-hate-relationship-8f73e5b291fb

36 Gawruch B. (15.1.2019). List tables with their primary keys (PKs) in MySQL database - MySQL Data Dictionary Queries. Haettu 1.7.2021 sivustolta Dataedo. Internetosoite: https://dataedo.com/kb/query/mysql/list-tables-with-their-primary-keys

37 NNK. (28.8.2021). PySpark partitionBy() – Write to Disk Example. Haettu 28.8.2021 sivustolta Spark by Examples. Internetosoite: https://sparkbyexamples.com/pyspark/pyspark-partitionby-example/

38 Databricks. (6.10.2021). Best practices: Delta Lake | Databricks on AWS. Haettu 16.10.2021 sivustolta Databricks Docs. Internetosoite: https://docs.databricks.com/delta/best-practices.html

39 Wenchen F. (17.5.2020). SPARK-31404: file source backward compatibility after calendar switch. Haettu 1.7.2021 sivustolta Apache Jira Issues. Internetosoite: https://issues.apache.org/jira/browse/SPARK-31404

40 Konieczny B. (2.5.2020). Idempotent file generation in Apache Spark SQL. Haettu 22.8.2021 sivustolta Waiting for Code. Internetosoite: https://www.waitingforcode.com/apache-spark-sql/idempotent-file-generation-apache-spark-sql/read

41 Zhu R. (25.8.2021). Keskustelu Delta Slack -kanavalla #deltalake-oss. Internetosoite: https://delta-users.slack.com/archives/CJ70UCSHM/p1629823775160500?thread_ts=1629619981.138900&cid=CJ70UCSHM

42 Apache Community. (21.8.2021). Structured Streaming Programming Guide. Haettu 21.8.2021 sivustolta Apache Spark Docs. Internetosoite: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

43 Databricks. (5.9.2021). Table deletes, updates, and merges. Haettu 5.9.2021 sivustolta Delta Lake Docs. Internetosoite: https://docs.delta.io/latest/delta-update.html#-write-change-data-into-a-delta-table

44 Apache Spark. (8.9.2021). selectExpr. Haettu 8.9.2021 sivustolta Apache Spark Docs. Internetosoite: https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.selectExpr.html

45 Bakker K. (9.11.2019). Easy Spark optimization for max record: aggregate instead of join? Haettu 8.9.2021 sivustolta KeesTelksTech. Internetosoite: https://keestalkstech.com/2019/11/easy-spark-optimization-for-max-record-aggregate-instead-of-join/

46 Databricks. (11.9.2021). Welcome to Delta Lake's Python documentation page. Haettu 11.9.2021 sivustolta Delta Lake Docs. Internetosoite: https://docs.delta.io/latest/api/python/index.html

47 Ennenga Z. (3.3.2020). On Spark, Hive, and Small Files: An In-Depth Look at Spark Partitioning Strategies. Haettu 18.9.2021 sivustolta Meidum. Internetosoite: https://medium.com/airbnb-engineering/on-spark-hive-and-small-files-an-in-depth-look-at-spark-partitioning-strategies-a9a364f908

48 Databricks. (10.8.2021). Data Object Privileges. Haettu 29.8.2021 sivustolta Databricks Docs. Internetosoite: https://docs.databricks.com/security/access-control/table-acls/object-privileges.html

49 Singman P. (21.9.2021). Hive Metastore — It Didn't Age Well. Haettu 29.9.2021 sivustolta Medium. Internetosoite: https://medium.com/whispering-data/hive-metastore-it-didnt-age-well-b2e648d5aecc

Liitteet

Sparkin asennus Windows-koneelle

1) Asenna Oracle Java 8. (https://java.com/en/download/).
    a. Vaatii Oracle-tunnuksen.
2) Asenna PyCharm (https://www.jetbrains.com/pycharm/)
    a. Vaatii lisenssin
3) Lataa Spark (https://spark.apache.org/downloads.html)
    a. Lataa versio: Spark 3.1.2 Pre-Built for Hadoop 2.7 and later
    b. Pura ladattu tiedosto esimerkiksi kansioon D:\SPARK\
4) Lataa WinUtils (https://github.com/cdarlint/winutils/tree/master/hadoop-2.7.7/bin)
    a. Luo kansio D:\SPARK\spark-3.1.2-bin-hadoop2.7\hadoop\bin
    b. Lataa winutils.exe-tiedosto tuohon kansioon
    c. Lataa hadoop.dll Windowsin System32-kansioon
5) Lisää seuraavat ympäristömuuttujat Windowsin asetuksista:

SPARK_HOME	D:\SPARK\spark-3.1.2-bin-hadoop2.7
HADOOP_HOME	D:\SPARK\spark-3.1.2-bin-hadoop2.7\hadoop
JAVA_HOME	C:\Program Files\Java\jdk1.8.0_291
PATH (Append)	%SPARK_HOME%\bin %HADOOP_HOME%\bin %JAVA_HOME%\bin

6) Käynnistä PyCharm ja luo projekti

a. Etsi JetBrains Help -sivuston ohje ”Install, Uninstall, and Upgrade Intepreter Paths”

b. Seuraten ohjetta, lisää Python-tulkin hakemistopolkuijen listaan seuraava hakemisto: D:\SPARK\spark-3.1.2-bin-hadoop2.7\python

Polut ovat todennäköisesti kunnossa, mikäli voit PyCharm-projektin skriptissä ladata moduulin pyspark komennolla ”import pyspark”. Kokeile kuitenkin luoda yksinkertainen DataFrame ja kirjottaa se tiedostoon.

Delta Log -kansion JSON-tiedoston sisältö


```
{
    "commitInfo": {
        "timestamp": 1628166178623,
        "operation": "WRITE",
        "operationParameters": {
            "mode": "Overwrite",
            "partitionBy": "[]"
        },
        "isBlindAppend": false,
        "operationMetrics": {
            "numFiles": "1",
            "numOutputBytes": "4370",
            "numOutputRows": "10"
        }
    }
}
{
    "protocol": {
        "minReaderVersion": 1,
        "minWriterVersion": 2
    }
}
{
    "metaData": {
        "id": "4998b445-3b8d-47c5-9b7f-f58e25f378f3",
        "format": {
            "provider": "parquet",
            "options": {}
        },
        "schemaString": "Here would be schema struct as JSON-string"
        "partitionColumns": [],
        "configuration": {},
        "createdTime": 1628166178373
    }
}
{
    "add": {
        "path": "part-00000-ded40f7e-cbc3-4a74-b015-54323bcd6570-
c000.parquet",
        "partitionValues": {},
        "size": 4370,
        "modificationTime": 1628166178482,
        "dataChange": true
    }
}
```