Santeri Nurminen

TIETOKANNANHALLINTAJÄRJESTELMIEN SOVELTUVUUDEN ARVIOINTI VAIHTOEHTOISENA RATKAISUNA TIEDOSTOJÄRJESTELMÄPOHJAISELLE TIETOKANNALLE

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2024

TIIVISTELMÄ

Nurminen, Santeri
Tietokannanhallintajärjestelmien soveltuvuuden arviointi vaihtoehtoisena ratkaisuna tiedostojärjestelmäpohjaiselle tietokannalle
Jyväskylä: Jyväskylän yliopisto, 2024, 94 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaaja(t): Taipalus, Toni & Zhidkikh, Denis

Järjestelmän tiedon varastointiin valittavan ratkaisun soveltuvuus käytössä olevalle tietomallille ja tietotarpeille on erittäin tärkeä tekijä sekä järjestelmän suorituskyvyn että kehittämisen kannalta. Tämän pro gradu -tutkielman tarkoituksena on tunnistaa tutkielmassa tarkasteltavan järjestelmän nykyiselle tiedon varastoinnin ratkaisulle soveltuvin mahdollinen vaihtoehtoinen toteutus tietokannanhallintajärjestelmistä. Tutkielman motivaationa toimii reaalimaailmasta tunnistettu ongelma, jossa tarkasteltavan järjestelmän toiminta on hidastunut käytössä olevan tiedon varastoinnin ratkaisun soveltumattomuuden takia. Tutkielma suoritetaan suunnittelutieteellisenä konstruktiivisena tutkimuksena hyödyntäen DSRM-metodologiaa tutkielman rakenteen ohjaajana. Tutkielmassa esitellään aluksi tarkasteltava järjestelmä, minkä jälkeen kirjallisuuskatsauksella muodostettavan teoriapohjan avulla rajataan tarkempaan arviointiin valittavat tietokannanhallintajärjestelmät. Rajauksessa hyödynnetään soveltuvuuden arviointia tarkasteltavan järjestelmän käyttötapausten sekä tietomallin vaatimusten näkökulmista. Rajauksen jälkeen tarkasteltavan järjestelmän tietomalli mallinnetaan kolmelle valitulle tietokannanhallintajärjestelmälle, joille suoritetaan vertaileva suorituskykytestaus käyttäen tarkasteltavan järjestelmän dataa sekä valittua käyttötapausta. Tutkielman tuloksissa pystytään tunnistamaan alustavasti tehokkain ratkaisu tarkasteltavasta järjestelmästä havaitulle ongelmalle tutkielmalle asetettujen rajoitteiden kontekstissa. Tuloksista voidaan myös havaita, että tietokannanhallintajärjestelmien soveltuvuusarviointi on hyvin työläs sekä monivaiheinen prosessi, jonka oikeellinen suorittaminen vaatii kriittistä tarkastelua niin teorian kuin käytännön näkökulmista. Tutkielma tuottaa kontribuutioita erityisesti esitetyn käytännön ongelman ratkaisuun luotujen artefaktien kautta. Tämän lisäksi tutkielmassa onnistuttiin luomaan tieteellisen näkökulman kontribuutioita muun muassa mittavan jatkotutkimusaiheiden määrän muodossa.

Asiasanat: tietokannanhallintajärjestelmä, suorituskykytestaus, tietokantaparadigmat, CAP-teoreema

ABSTRACT

Nurminen, Santeri
Evaluating database management systems as an alternative solution for a filesystem-based database
Jyväskylä: University of Jyväskylä, 2024, 94 pp.
Information Systems Science, Master's Thesis
Supervisor(s): Taipalus, Toni & Zhidkikh, Denis

A system's data storage solution's suitability for its data model is an essential factor for the system's performance and development. The aim of this thesis is to identify the most suitable alternative database management system for the reviewed system's current data storage solution. The motivation for the thesis arises from a real-life problem, where the reviewed system has been observed to slow down due to the current storage solution's unsuitability. This thesis uses a constructive design science approach, applying the DSRM-methodology for guiding the thesis' structure. First, the reviewed system is introduced, after which using a theory base constructed from a conducted literature review, database management systems are evaluated and delimited for further assessment from the viewpoints of use cases and the data model's needs. The reviewed system's data model is then modelled into three different database management systems, after which a comparative performance benchmark is conducted. The results of the study identify the most efficient solution in the confines of the presented problem. In addition to this, the results show that database management system suitability evaluation is an extensive multiphase process that requires critical analysis from both theoretical and practical standpoints. The contributions of the study are mainly related to solving a practical problem, but theoretical contributions were also identified, among other things, in the form of many suggestions for further study.

Keywords: database management system, performance benchmark, database paradigms, CAP-theorem

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Web-pohjaisten palveluiden latausaikojen vaikutus käyttäjäkokemukseen on laajasti tutkittu ilmiö, jonka vaikutukset käyttäjäkokemukseen ovat selvät: nopeammat latausajat lisäävät havaittua palvelun laatua (Egger ym., 2012). Järjestelmän kehityksen näkökulmasta on tällöin oleellista pyrkiä mahdollisimman tehokkaaseen toteutukseen. Tämän perusteella myös järjestelmälle mahdollisimman soveltuvan sekä tehokkaan tiedon varastointiratkaisun löytäminen on erittäin tärkeää, koska varastointiratkaisun tehokkuus vaikuttaa suoraan itse palvelun nopeuteen.

Tässä tutkielmassa tarkastellaan TIM (The Interactive material) -oppimisalustaa, jonka sivujen latausnopeudessa on kasvavan datamäärän takia havaittu selkeää hidastumista. TIMin kehitystiimin mukaan yhdeksi merkittävimmistä hidastumisen syistä on havaittu nykyisen tiedon varastointiratkaisun sopimattomuus suurien datamäärien käsittelyyn. Tutkielman tarkoituksena on täten löytää soveltuvin ja tehokkain vaihtoehtoinen ratkaisu järjestelmän tiedon varastointiin, joka on tällä hetkellä toteutettu Linux-tiedostojärjestelmällä. TIM valikoitui tutkielmassa tarkasteltavaksi järjestelmäksi TIMin kehitystiimin kanssa tapahtuneen yhteydenpidon kautta, kun esitetyn ongelman tarkemman tutkimisen tarve havaittiin.

Vaihtoehtoisten ratkaisujen etsiminen tarkentui heti tutkielmaa hahmoteltaessa luonnollisesti tietokannanhallintajärjestelmiin, joiden tarkoituksena on ratkaista suurien datamäärien tehokkaan käsittelyn tuottamat ongelmat. Täten tutkielmassa esitetään seuraavat tutkimuskysymykset:

- *Miten pystytään rajaamaan sekä valitsemaan tarkasteltavalle järjestelmän osalle parhaiten soveltuvat tietokannanhallintajärjestelmät?*
- *Mikä valikoiduista tietokannanhallintajärjestelmistä on tehokkain vaihtoehto järjestelmästä havaitun ongelman ratkaisemiseen?*

Esitettyihin tutkimuskysymyksiin lähdettiin etsimään vastauksia konstruktiivisen tutkimuksen menetelmällä, jonka valintaperusteet sekä hyödyntäminen tutkielmassa on kuvailtu luvussa kolme.

Ensimmäiseen tutkimuskysymykseen pyritään löytämään vastaus kirjallisuuskatsauksen kautta. Tietokannanhallintajärjestelmätuotteiden valtavan määrän sekä näiden välisen vaihtelevuuden takia tiettyyn käyttötapaukseen sopivan järjestelmän löytäminen ei ole yksiselitteistä tai yksinkertaista. Koska käyttötapauksen tietotarpeisiin soveltuvan tietokannanhallintajärjestelmän löytäminen on kuitenkin erittäin tärkeää järjestelmän toiminnan kannalta, on myös sopivuuden selvittäminen tärkeää. Aiempaan tutkimustietoon perustuva lähestymistapa tarjoaa perustellun lähtökohdan alustavalle tietokannanhallintajärjestelmien rajaukselle, jolloin tiettyjä tietokannanhallintajärjestelmien ryhmiä pystytään rajaamaan pois heti soveltuvuusarvioinnin alkuvaiheessa käyttötapauksiin peilaten aiemman tutkimustiedon perusteella. Tällä pyritään saavuttamaan ”reilu”, aiempaan tietoon perustuva rajaus, jonka pohjalta tuotekohtaisen tarkemman arvioinnin työmäärä vähenee huomattavasti. Kirjallisuuskatsauksessa pyritään hyödyntämään sekä teoreettisempaa, että käytännönläheisempää näkökulmaa, kuitenkin olemassa olevaan tieteelliseen taustatutkimukseen perustuen. Kirjallisuuskatsauksen aineistoa haettiin muun muassa Google Scholar-, ACM- ja IEEE-tietokannoista esimerkiksi seuraavilla hakusanoilla: *database suitability, dastabase evaluation, database paradigms, CAP-theorem, RDBMS, NoSQL* sekä *NewSQL*.

Toiseen tutkimuskysymykseen pyritään löytämään vastaus konkreettisen suorituskykytestauksen avulla. Järjestelmässä käytössä oleva tietomalli mallinnetaan edellä esitetyn kirjallisuuskatsauksen avulla valituille tietokannanhallintajärjestelmille, jonka jälkeen niihin ajetaan nykyisen tuotantotietokannan data. Tämän jälkeen suoritetaan mahdollisimman reiluksi suunniteltu suorituskykytestaus, jonka tuloksia vertailemalla pyritään löytämään vastaus tutkimuskysymykseen. Suorituskykytestauksen toteutus on perusteltua sekä merkittävää, koska vain tilannetta oikeasti testaamalla pystytään saamaan luotettavia, vertailukykyisiä tuloksia arvioinnin tueksi.

Tutkielman alustavana hypoteesina voidaan olettaa, että tuloksina pystytään tunnistamaan tarkasteltavalle järjestelmälle soveltuvimmilta vaikuttavat tietokannanhallintajärjestelmät sekä määrittämään näistä tehokkain asetettujen rajoitteiden valossa. Tutkielman näkökulmasta rajataan pois muun muassa sen määrittäminen, onko löydetty vaihtoehtoinen ratkaisu alkuperäistä toteutusta tehokkaampi. Tämän näkökulman arviointi jätetään jatkotutkimuksen aiheeksi.

Tutkielman rakenne on seuraava. Seuraavassa luvussa (luku 2) esitellään tarkasteltava järjestelmä, TIM. Kolmannessa luvussa esitellään tutkielman tutkimusmenetelmä, sen tueksi valittu metodologia sekä näiden yhteys tutkielman osioihin. Neljännessä ja viidennessä luvussa suoritetaan edellä mainittu kirjallisuuskatsaus, jonka tarkoituksena on rajata soveltuvia tietokannanhallintajärjestelmiä tutkielmassa suoritettavaan testaukseen. Kuudennessa luvussa valittuja tuotteita esitellään lyhyesti, minkä jälkeen TIMin käytössä oleva tietomalli mallinnetaan tietokantamalleille sopivaksi. Seitsemännessä luvussa suoritetaan edellä mainittu suorituskykytestaus, tarkasti sekä läpinäkyvästi dokumentoiden koko prosessi testialustasta tuloksiin tutkimuksen toistettavuuden edistämiseksi. Kahdeksannessa luvussa suoritetaan pohdintaa tutkimuskysymyksien

vastauksista, kontribuutioista, rajoitteista sekä jatkotutkimuksen aiheista. Lopuksi yhdeksännessä kappaleessa esitetään lyhyt yhteenveto tutkielmasta.

2 TIM

TIM (The Interactive Material) on avoimen lähdekoodin MIT-lisensoitu oppimisalusta, joka toimii alustana kymmenille sen avulla rakennetuille kursseille ja jota on käyttänyt yli 20 000 käyttäjää vuodesta 2014 lähtien (Jyväskylän yliopiston informaatioteknologian tiedekunta, ei pvm. a). Suuren käyttöasteen takia alustan tiedon varastoinnin tehokkuus ja toimivuus on oleellinen osa niin edellä mainittujen kurssien toimintaa, kuin opiskelijoiden opintojen suorittamista. TIMin dokumenttien, toisin sanoen ”sivujen”, varastointi on alun perin toteutettu Linuxin tiedostonhallintajärjestelmällä (filesystem), joka valikoitui senhetkisessä kehitysvaiheessa suorituskykytestauksen sekä arvioinnin perusteella parhaaksi ratkaisuksi.

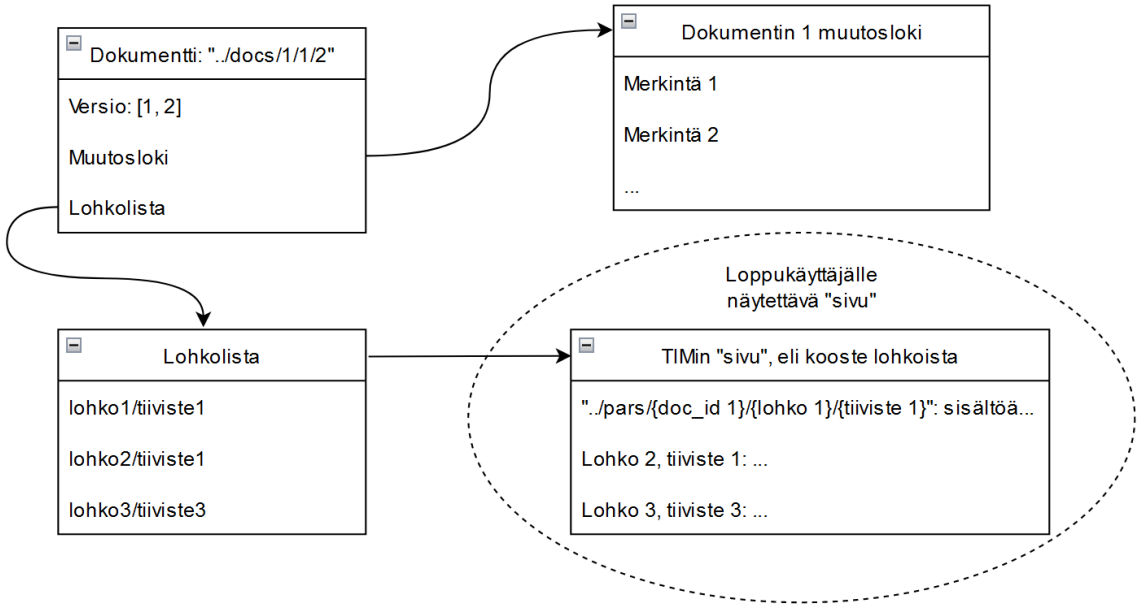
Tässä tutkielman luvussa esitellään TIMin dokumenttien varastointiin käytössä olevan tiedostojärjestelmän rakenne sekä tiedostojärjestelmän yleisimpiä käyttötapauksia. Luvun tarkoituksena on pohjustaa suoritettavaa tutkimusta sekä luoda yleistä tilannekuvaa. Erityisen tärkeää lukijan kannalta on tuntea luvussa esiteltävä terminologia, jota hyödynnetään seuraavissa tutkielman luvuissa jatkuvasti.

2.1 TIMin dokumenttien tietorakenne

Jotta vaihtoehtoisten ratkaisujen soveltuvuutta pystytään arvioimaan TIMin tietotarpeisiin, on oleellista tuntea TIMin dokumenttien varastoinnin tietorakenne. Seuraavaksi esitettävä kuvaus perustuu TIMin tarjoamaan, julkisesti saatavilla olevaan dokumentaatioon (Jyväskylän yliopiston informaatioteknologian tiedekunta, ei pvm. b).

TIMin dokumenttien tietorakenne on rakennettu Linux-tiedostojärjestelmää hyödyntäen. Tämän takia tietorakenteen perusta rakentuu paljolti tiedostojen sekä näiden polkujen ympärille, polussa olevien hakemistojen kuvatessa mm. käsiteltävän kohteen ID:tä tai versiota. Seuraavaksi annettavissa esimerkeissä polkujen aloituspisteenä, eli käytännössä ”tietokannan” alkupisteenä toimii

hakemisto ”tim_files”. Seuraavassa kuviossa (kuvio 1) visualisoidaan tietorakennetta sekä sen sisältämien kohteiden välistä suhdetta, jonka jälkeen tietorakenteessa esiintyvät entiteetit kuvaillaan tarkemmin.



KUVIO 1 TIMin tietorakenne yksinkertaistettuna visualisointina

2.1.1 Dokumentti

Dokumentti on määritelty TIMin kontekstissa versioiduksi listaksi dokumentin osista, eli lohkoista. Dokumentti voidaan käsittää myös, etenkin loppukäyttäjän näkökulmasta, TIMin yksittäisenä ”sivuna”, jonka käyttäjä avaa. TIMin dokumenttien hallinnan kontekstissa dokumentti toimii sisällön ”yhteen kokoajana”. Dokumenttiin sisältyvät seuraavat tietueet: 1) ID, uniikki dokumentin tunniste, 2) versio, 3) lohkolista sekä 4) muutosloki.

TIMin dokumentit on versioitu. Dokumentin versio koostuu kahdesta eri osasta: pää- ja alaversiosta, esimerkiksi ”2.1” (pää : 2, ala: 1). Dokumentin versiointia hallitaan seuraavalla logiikalla: Pääversiota nostetaan, jos dokumenttiin 1) lisätään uusi lohko, tai 2) dokumentista poistetaan lohko. Alaversiota nostetaan, kun dokumentin lohkoa muokataan. Dokumentin versio muodostaa myös dokumentin polun tiedostojärjestelmässä, seuraavalla logiikalla: ”/tim_files/docs/{dokumentin ID}/{pääversio}/{alaversio}”, esimerkiksi ”/tim_files_docs/1/1/0”.

Jokaiselle dokumentin versiolle (ylä- ja alaversion yhdistelmälle) tallennetaan oma lohkolistansa tekstidokumentin muodossa, jonka ”nimenä” toimii dokumentin alaversio, kuten edellä annetussa dokumentin polun selitteessä. Lohkolista itsessään koostuu yksittäisistä, rivinvaihdolla erotetuista riveistä, joiden järjestys määrää lohkojen järjestyksen, sekä jonka sisällöllä tunnistetaan lohko. Yksittäisen lohkolistan rivin sisältö on seuraavassa muodossa: ”{lohkon ID}/{lohkon tiiviste}”, viitaten tiettyyn lohkoon ja sen tiettyyn versioon.

Dokumentaatiossa mainitaan myös, että vanhemmissa dokumenteissa rivi voi olla muodossa ”{lohkon ID}”, jolla ilmaistaan lohkon mahdollisimman uutta versiota, mutta tässä muodossa olevia rivejä ei enää kirjoiteta.

Muutosloki, joka sijaitsee tiedostona polussa ”/tim_files/docs/{dokumentin ID}/changelog, sisältää rivinvaihdolla erotettuja JSON-objekteja, jotka kuvaavat dokumenttiin kohdistuneita muutoksia. Muutoslokin käyttötarkoitukseksi mainitaan muun muassa dokumentin versioiden välisten eroavaisuuksien muodostaminen. JSON-objektit ovat seuraavassa muodossa:

- Group_id: TIM-ryhmä, joka teki muutoksen.
- Par_id: lohko, jota muutos koski.
- Op: muutosoperaatio, joka on tehty.
- Op_params: muutosoperaation mahdolliset lisäparametrit.
- Ver: dokumentin versio
- Time: muutoksen aikaleima

2.1.2 Lohko

Lohko on määritelty TIMin kontekstissa yksittäisen JSON-objektin sisältäväksi tiedostoksi, joka sisältää lohkon metatiedot, sisällön, sekä mahdollisesti asetuksia ja sisällön HTML-käännöksen. Käytännössä lohko on siis TIMin ”sivun osa”, johon sisältyy sisältönsä lisäksi erilaisia kenttiä, jotka ohjaavat lohkon toimintaa. Lohkotiedostot sijaitsevat tiedostojärjestelmässä polussa ”tim_files/pars/{lohkon doc_ID}/{lohkon ID}/{lohkon tiiviste}”. Lohkotiedoston JSON-objekti sisältää seuraavat kentät:

- Attrs: lohkon attribuutit avain-arvo -pareina
- ID (merkkijono): lohkon tunniste. Huomioitavaa tässä on, että lohkon ID ei ole taatusti globaalisti uniikki, vaan ainoastaan saman dokumentin doc_id:n alla olevat lohkojen ID:t ovat uniikkeja.
- Md: lohkon sisältö tekstinä, oletuksena markdown-muodossa.
- T: lohkon versiotiiviste, joka on laskettu lohkon sisällön ja attribuuttien perusteella. Lohkon versiotiivistettä käytetään lohkojen versiointiin.
- H: lohkon ”html-välimuisti” avain-arvo -parina, eli lohkon sisältö muutettuna valmiiksi HTML-muotoon turhien käännösten välttämiseksi. Ei välttämättä ole olemassa, ennen kuin uusi dokumentti avataan ensimmäisen kerran.

2.2 Dokumenttien yleisimmät käyttötapaukset

Samoin kuin TIMin dokumenttien tietorakenteen tapauksessa, myös dokumenttien kohdistuvat yleisimmät käyttötapaukset ovat oleellisia tunnistaa, jotta

tietokannanhallintajärjestelmien soveltuvuutta pystytään arvioimaan oikeellisesti. Myös seuraava käyttötapausten listaus perustuu TIMin tarjoamaan dokumentaatioon (Jyväskylän yliopiston informaatioteknologian tiedekunta, ei pvm. b). Dokumentaatiossa esitettyjä yleisimpiä käyttötapauksia on neljä kappaletta.

Ensimmäinen yleisimmistä käyttötapauksista on dokumentin luominen, jossa käyttäjä luo uuden TIM-dokumentin ”uusi dokumentti” -toiminnolla. Käyttötapauksen sisäisessä toteutuksessa luodaan PostgreSQL-tietokannassa dokumentille uusi tietue sekä ID, jonka jälkeen dokumenttitietokannassa luodaan dokumentista versio ”1.0”. Koska kyseisen PostgreSQL-tietokannan tarkoitusta tai toimintaa ei kuvata tarkemmin, käyttötapaus näyttäytyy tässä tutkielmassa pelkästään uuden dokumentin luomisena versiolla ”1.0”.

Toinen yleisimmistä käyttötapauksista, dokumentin lohkojen hakeminen, näyttäytyy loppukäyttäjälle käytännössä TIMin sivun latauksena. Sisäisessä toteutuksessa, dokumenttien kontekstissa, dokumenttien ”tietokannasta” haetaan kyseisen dokumentin halutun version lohkolistan esittämät lohkot kokonaisuudessaan. TIMin kehitystiimi arvioi (tutkielman suorittamisen aikana tapahtuneessa viestinnässä) tämän käyttötapauksen olevan eniten tapahtuva kaikista dokumentteja koskevista käyttötapauksista sekä toimivan suurimpana ”pullonkaulana” dokumenttien varastoinnin toiminnalle.

Kolmas yleisimmistä käyttötapauksista on dokumentin versiohistorian hakeminen. Käyttötapauksessa loppukäyttäjä avaa dokumentin ”manage”-näkymän ja tarkastelee muokkaushistoriaa. Sisäisessä toteutuksessa dokumenttien ”tietokannasta” haetaan koko dokumentin versiohistoria eli muutosloki. Käyttötapaus on tällä hetkellä käytössä myös dokumentin latauksessa, jossa versiohistoriaa hyödynnetään uusimman dokumentin version selvittämisessä.

Neljäs, viimeinen esitetyistä yleisimmistä käyttötapauksista, on dokumentin muokkaaminen. Käyttötapauksessa loppukäyttäjä tekee dokumenttiin muutoksia lisäämällä, poistamalla tai muokkaamalla dokumentin lohkoja. Sisäisessä toteutuksessa, riippuen muutoksen tyypistä, dokumentista luodaan uusi pää- tai alaversio, jonka jälkeen uusi lohko lisätään tai poistetaan. Muokkauksen käyttötapaus ei itsessään muokkaa olemassa olevaa tietoa, vaan lisää tai poistaa sitä: dokumentista luodaan aina uusi versio riippumatta muutoksen tyypistä. Lohkon poistamisen tapauksessa lohkon viite poistetaan lohkolistasta, mutta itse lohko säilytetään tietokannassa. Muokkauksessa puolestaan lohkosta luodaan uusi versio, eikä olemassa olevaa suoranaisesti muokata.

3 TUTKIMUSMENETELMÄ

Tässä tutkielman luvussa esitellään tutkielmassa suoritettavan tutkimuksen tavoitteet, tutkimuskysymykset, valittu tutkimusmenetelmä ja perusteet sen valinnalle. Tämän lisäksi esitetään katsaus tutkielman rakenteeseen, erityisesti tarkastellen yksittäisten lukujen merkitystä valittuun tutkimusmenetelmään peilaten.

3.1 Tutkimuksen tavoitteet ja tutkimuskysymykset

Suoritettavan tutkimuksen tavoitteena on löytää tutkittavasta järjestelmän osasta havaittuun ongelmaan, toisin sanoen TIMin kehitystiimin havaitsemaan dokumenttien varastoinnin käytössä olevan ratkaisun hidastumiseen, soveltuvin vaihtoehtoinen ratkaisu. Tutkimuksessa etsittävän ratkaisun näkökulma on rajautunut luonnollisesti tietokannanhallintajärjestelmiin, joiden alkuperäinen sekä nykyinen päätarkoitus on tiedon tehokas varastointi. Kuitenkin, koska tietokannanhallintajärjestelmätuotteiden määrä markkinoilla on valtava, tulee ennen soveltuvuuden arviointia suorittaa teoriapohjaista rajausta arviointiin valittavista tuotteista. Tämän perusteella tutkimuksen tutkimuskysymykset voidaan esittää seuraavassa muodossa:

- *Miten pystytään rajaamaan sekä valitsemaan tarkasteltavalle järjestelmän osalle parhaiten soveltuvat tietokannanhallintajärjestelmät?*
- *Mikä valikoiduista tietokannanhallintajärjestelmistä on tehokkain vaihtoehto järjestelmästä havaitun ongelman ratkaisemiseen?*

Koko järjestelmälle yleistettävällä tasolla tehtävän kattavan tietokannanhallintajärjestelmän soveltuvuusarvioinnin tekeminen on äärimmäisen työläs prosessi, joka vaatisi huomattavaa iteratiivista työskentelyä sekä järjestelmän toteutuksen yksityiskohtaista tuntemista. Tässä tutkielmassa suoritettavan tutkimuksen tarkoituksena ei ole löytää absoluuttisesti parasta vaihtoehtoa esitettyyn ongelmaan, vaan toimia pohjana jatkotutkimukselle tai -kehitykselle, antaen

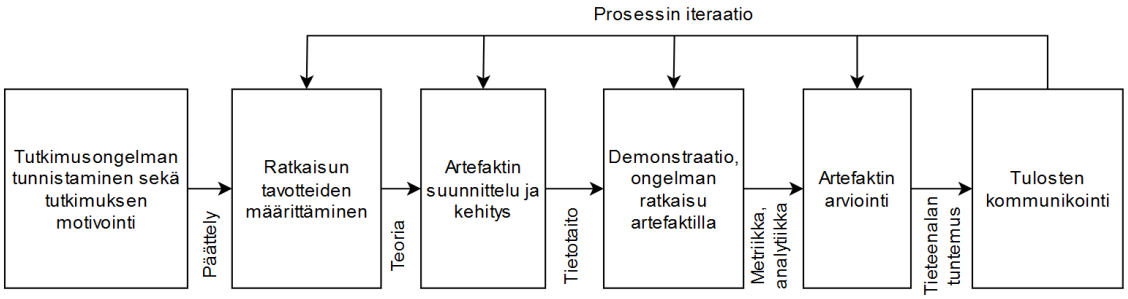
alustavan arvion valikoitujen tietokannanhallintajärjestelmien soveltuvuudesta. Tarkastelu painottuu tietokannanhallintajärjestelmien tehokkuuden arviointiin havaitussa järjestelmän toiminnan ongelmakohdassa. Tulee huomioida, että tutkielmassa ei oteta kantaa siihen, ovatko toteutetut artefaktit, eli valittujen tietokannanhallintajärjestelmien toteutukset tehokkaampia kuin alkuperäinen toteutus. Tämän reilu testaaminen todettiin tutkimusta tehtäessä hyvin työlääksi prosessiksi ja näin ollen rajattiin jatkotutkimuksen aiheeksi tutkielman pituus huomioon ottaen.

3.2 Suunnittelutiede ja DSRM

Tietokannanhallintajärjestelmien suorituskykymittauksia on suoritettu niin akateemisessa kuin teollisessa kontekstissa huomattavia määriä (Taipalus, 2024). Tässä tutkielmassa esitetty tutkimuskehys sekä ongelma ovat kuitenkin uniikkeja: edes jokseenkin samankaltaista tilannetta tutkivaa aiempaa tutkimusta ei kyetty tunnistamaan. Täten, tilanteen uniikkiuden sekä aiemman datan puutteen takia, jotta varsinkin toiseen tutkimuskysymykseen pystytään vastaamaan, tulee tutkimuksessa toteuttaa konstruktiot vertailtavista vaihtoehtoisista ratkaisuista.

Suunnittelutieteellinen tutkimus, josta syntyy arvioitava konstruktio, on erittäin yleistä tietojärjestelmätieteen tieteenalalla (Hevner ym., 2004). Suunnittelutiede on ongelmanratkaisuun painottunut, iteratiivinen sekä inkrementaalinen menetelmä, jossa ongelmaa lähestytään kehittämällä vaihtoehtoinen malli, testaamalla mallia tutkimukselle asetettujen rajoitteiden puitteissa sekä jatkokehittämällä mallia tulosten pohjalta (Hevner ym., 2004). Tästä lähestymistavasta muodostuu ”tutkimussilmukka”, jossa jatkuvasti kehitetään sekä testataan ongelman ratkaisua, päättyen tilanteeseen, jossa ratkaisu on valmis täyttäessään ongelmassa esitetyt vaatimukset sekä rajoitteet (Hevner ym., 2004). Tämä käytännön kehitystyötä mukaileva menetelmä soveltuu tässä tutkielmassa suoritettavaan tutkimukseen erityisesti toisen tutkimuskysymyksen vastauksen muodostamiseksi.

Koska suunnittelutiede ei itsessään ota kantaa tutkimuksen konkreettiseen suorittamiseen, on tässä tutkielmassa hyödynnetty tutkimuksen pohjaksi tietojärjestelmätutkimukselle kehitettyä suunnittelutieteeseen pohjautuvaa metodologiaa, DSRM:ää (Design Science Research Methodology) (Peffers ym., 2007). DSRM:ssä kuvaillaan tietojärjestelmätieteen tieteenalalla tehtävälle suunnittelutieteelliselle tutkimukselle kuusivaiheinen iteratiivinen prosessi, joka on havainnollistettu seuraavassa kuviossa (kuvio 2).



KUVIO 2 DSRM yksinkertaistettuna ja suomennettuna, mukaillen Peffers ym. (2007).

Tässä tutkielmassa sovelletaan DSRM:ää niin tutkielman rakenteen kuin itse tutkimuksen suorittamisen ja jäsentämisen tukena. DSRM:n prosessit jakavat tutkielman selviin lukuihin, jotka ohjaavat tutkimusta kokonaisuutena. Tutkielman rakenne sekä prosessien ilmeneminen rakenteessa kuvataan seuraavassa alaluvussa.

Tutkielmassa ei kuitenkaan suoriteta yleiskattavaa iteraatiota, kuten metodologia ehdottaa, jotta tutkielman pituus sekä työmäärä säilyvät toteutuksen mahdollistavissa rajoissa. Tutkielman tarkoituksena on toimia ”pohjatyönä”: Iteraatioon pyritään kannustamaan esitettävissä jatkotutkimuksen aiheissa, joissa luotua konstruktiota pystyttäisiin parantamaan kaikkien käytyjen prosessien osalta. Tutkielman sisällä olevia prosesseja on kuitenkin työstetty iteratiivisesti: Etenkin artefaktin suunnittelussa sekä kehityksessä työskentely on ollut iteratiivista ongelmatilanteissa sekä ratkaisun optimoinnissa.

3.3 Tutkielman rakenne

Tutkielman rakenne pohjautuu DSRM:ään (Peffers ym., 2007). Johdantoluvussa annettiin lukijalle yleinen katsaus tutkielmasta niin aiheen, ongelman, menetelmän kuin tulosten osalta. Luvussa kaksi annettiin yleinen katsaus TIMistä, erityisesti esitetyn tutkimusongelman taustoituksen näkökulmasta. Kolmannessa luvussa kuvaillaan tutkielman toteutus yleisellä tasolla, mukaan lukien tutkimuskysymykset sekä tutkimusmenetelmä. Nämä kolme lukua muodostavat DSRM:n ensimmäisen sekä toisen prosessin kokonaisuuden, jossa tunnistetaan ja motivoidaan tutkimusongelma sekä määritetään tavoiteltava ratkaisu.

Neljäs ja viides luku muodostavat tutkielman teoriapohjan, joka on toteutettu kirjallisuuskatsauksena. Taustakirjallisuutta tarkastellaan kahdesta näkökulmasta: luvussa neljä teoreettisemman CAP-teoreeman näkökulmasta sekä luvussa viisi käytännönläheisemmällä, mutta silti taustatutkimukseen pohjautuvalla yleisellä tietokantaparadigmojen näkökulmalla. Nämä luvut muodostavat DSRM:n kolmannen prosessin, artefaktin suunnittelun ja kehityksen teoriapohjan.

Luvussa kuusi kehitetään tutkimuksessa arvioitavat artefaktit iteratiivisesti hyödyntäen aiempaa tutkimustietoa sekä muun muassa tuotteiden kehittäjiltä saatavilla olevaa ohjeistusta. Artefakteihin valittujen tuotteiden rajaus sekä

tarkemmat valintaperusteet pohjautuvat em. teoriataustoitukseen, itse kehityksen perustuessa aiemmissa luvuissa esiteltyyn tutkimusongelmaan sekä kohdealueesta tehtyyn yleiskatsaukseen. Luku toteuttaa DSRM:n kolmannen prosessin ”konkreettisen” kehittämisen osa-alueen.

Seitsemännessä luvussa toteutetaan tutkielman konkreettinen tutkimus, eli suorituskykytestaus, jolla pyritään vastaamaan toiseen tutkimuskysymykseen yksiselitteisesti. Luku täydentää DSRM:n kolmatta prosessia konkretisoimalla kehitettyjä artefakteja entisestään tutkimuksen näkökulmaan, sekä toteuttaa DSRM:n neljännen prosessin, jossa ongelma pyritään ratkaisemaan artefaktilla. Samalla luvun lopussa aloitetaan DSRM:n viidettä prosessia, artefaktin arviointia.

Kahdeksannessa luvussa esitetään tutkielman pohdinta, mukaan lukien tutkimuskysymyksiin löydetyt vastaukset, näiden arviointi, jatkotutkimuksen aiheet sekä tutkimuksen rajoitteet. Tämän jälkeen yhdeksännessä luvussa esitetään lyhyt yhteenveto koko tutkielmasta. Kahdeksas luku käsittelee loppuun DSRM:n viidennen prosessin, jossa annetaan arvio ongelman ratkaisun onnistumisesta. Molemmat luvut toimivat myös vahvasti DSRM:n kuudennen prosessin eli tulosten kommunikoinnin välineenä, käsitellen yhteen vetävästi sekä pohtivasti tutkimuksen tuloksia. Koska kaikki edellä mainitut luvut kommunikoivat ja dokumentoivat tutkimuksen etenemistä vaiheittain, voidaan DSRM:n kuudenneksi prosessiksi käsittää kuitenkin itse tutkielman kirjoittaminen sekä julkaisu.

4 CAP-TEOREEMA

CAP-teoreema on Brewerin (2000) esittämä teoreema, jossa tunnistetaan kolme jaettua dataa hyödyntävien verkkopohjaisten palveluiden haluttua ominaisuutta: eheys, saatavuus ja verkon osituksen sietokyky. Teoreemassa todetaan, että näistä kolmesta ominaisuudesta voidaan saavuttaa vain kaksi jaetun datan järjestelmissä (Brewer, 2000). Gilbert ja Lynch (2002) todistivat teoreeman pitävyyden osoittaen, että kahden edellä mainitun ominaisuuden saavuttaminen on mahdollista, mutta ei kaikkien. Koska CAP-teoreema on hyvin yleisessä käytössä oleva sekä nykyään erityisesti tietokannanhallintajärjestelmien kontekstissa hyödynnettävä teoreema, voidaan sen olettaa soveltuvan tutkielman kontekstissa tehtävän rajauksen pohjaksi.

Teoreeman avulla voidaan todeta, että tietokannanhallintajärjestelmät (DBMS, database management system) jakautuvat joihinkin seuraavista kolmesta kategoriasta: CA (eheä ja saatava, consistent & available), CP (eheä ja verkon ositusta sietävä, consistent & partition tolerant) sekä AP (saatava ja verkon ositusta sietävä, available & partition tolerant). Tämän perusteella on mahdollista suorittaa DBMS:ien vertailua teoreemasta saatua kategorioita käyttäen ja peilaten niitä DBMS:ää käyttävän järjestelmän käyttötapausten vaatimuksiin.

Tulee kuitenkin huomata, että käytännön tilanteissa CAP-teoreemaa ei voida pitää absoluuttisena ohjeistuksena järjestelmän valinnassa. Stonebraker (2010) toteaa, että teoreemaa on käytetty liian kevyenä syynä eheyden hylkäämiselle tietyissä skenaarioissa (NoSQL-tietokannanhallintajärjestelmien kontekstissa). Esimerkiksi äärimmäisessä tilanteessa, jossa kaikki fyysiset palvelimet tuhoutuvat luonnonkatastrofin seuraamuksena, saatavuutta on mahdotonta saavuttaa valitusta järjestelmän ”kategoriasta” huolimatta (Stonebraker, 2010). Tämän tutkielman kontekstissa teoreema tarjoaa kuitenkin soveltuvan viitekehyksen DBMS:ien vertailuun, koska vertailua on tarkoitus suorittaa ”toimivan” järjestelmän käyttötapausten näkökulmasta, eikä ole tarkoituksenmukaista ottaa vertailun piiriin esimerkiksi katastrofaalisten tilanteiden näkökulmaa.

4.1 Eheys

Eheydellä CAP-teoreemassa tarkoitetaan käytännössä, että (jokainen) palvelin palauttaa jokaiselle pyynnölle oikeellisen vastauksen (Gilbert & Lynch, 2012). Gilbert ja Lynch (2002) muotoilivat CAP-teoreeman eheän palvelun käsitteen ”atomisten dataobjektien” kautta. Atomisuuden eheysrajoitteen rajoissa jokainen järjestelmälle toteutettu operaatio, kuten luku tai kirjoitus, tulee toteuttaa siten, että ne näyttävät toteutuvan yhdessä hetkessä (Gilbert & Lynch, 2002). Tämä vastaa tilannetta, jossa hajautetulle järjestelmälle toteutetut operaatiot vaikuttavat toteutuvan yhdessä paikassa, vastaten operaatioihin yksi kerrallaan järjestyksessä (Gilbert & Lynch, 2002). Hajautettujen tietokantojen kontekstissa tämä tarkoittaa, että tietokantaklusterin jokainen solmu sisältää saman datan.

Yksittäisiä tietokannanhallintajärjestelmiä tarkastellessa eheyden tarkka määritelmä vaihtelee paljon esimerkiksi transaktiomallien mukaan. Kahdessa yleisimmässä mallissa, ACIDissa ja BASEssa, tietokannan eheys määritellään hyvin toisistaan poikkeavasti. ACIDissa tietokanta on eheä, jos ja vain jos se sisältää onnistuneiden transaktioiden tuloksia (Haerder & Reuter, 1983). Tällöin jokainen tietokantaan suoritettu transaktio ylläpitää sen eheyttä, johtaen ”transaktioeheään”, toisin sanottuna ”loogisesti eheään” tietokantaan (Haerder & Reuter, 1983). Koska vain onnistuneiden transaktioiden tulokset hyväksytään, tietokannan data pysyy aina eheänä ja ajantasaisena. Perinteisesti erityisesti relaatioparadigmaan pohjautuvat tietokannanhallintajärjestelmät (RDBMS, relational database management system) perustuvat ACID-malliin (Lehner & Sattler, 2010).

BASEssa, jota sovelletaan perinteisesti NoSQL -tietokantoihin, eheys käsitetään ”lopulta eheäksi” (eventually consistent) tietokannan tilaksi (Ganesh Chandra, 2015). BASE takaa, että vähintään 80 % ajasta tietokannan data on oikeellista hajautetussa järjestelmässä, mutta täyttä taetta jatkuvasta eheydestä ei ole johtuen tallennussolmujen määrän kasvusta (Ganesh Chandra, 2015).

Edellä mainittujen lisäksi tietokantojen kontekstissa on olemassa muita eheysmalleja, kuten ”vahvasti lopulta eheä” (strong eventual consistency), jossa yhdistellään kahta edellä mainittua eheysmallia (Gomes ym., 2017). Voidaan huomata, että yksittäisiä tietokannanhallintajärjestelmiä tarkastellessa eheys ei ole käsitteenä yksiselitteinen. Se muodostuu pikemminkin tietylle järjestelmälle asetetuista rajoitteista, jotka takaavat tietyn eheyden tason. Koska kuitenkin jokaiseen tietokannanhallintajärjestelmään kohdistetaan operaatioita, jotta niissä sijaitsevaa dataa pystytään hyödyntämään, tulee DBMS:ien tarjota jonkin tasoista eheyttä. Tällöin, jotta tietokantojen eheyttä pystytään luokittelemaan ja vertailemaan CAP-teoreemassa Gilbertin ja Lynchin (2012) määritelmän mukaan, tässä tutkielmassa eheyden ominaisuudella tarkoitetaan tällöin ”vahvaa eheyttä”.

Erityisesti relaatioparadigmaan pohjautuvat tietokannanhallintajärjestelmät (RDBMS) perustuvat pääsääntöisesti ACID-malliin, täten toteuttaen vahvaa eheyttä (Lehner & Sattler, 2010). NoSQL-tietokannanhallintajärjestelmien puolella tilanne ei ole näin yksiselitteinen, vaan eheyden taso voi riippua niin tuote-, asetus-, kuin tapahtumakohtaisesti (Ganesh Chandra, 2015). Täten, tässä

tutkielmassa suoritettavan luokittelun ja vertailun piirissä CAP-teoreeman C-”kategorioihin” (CA ja CP) sijoittuu RDBMS:ien lisäksi tuotekohtaisesti NoSQL-tietokannanhallintajärjestelmiä.

4.2 Saatavuus

CAP-teoreemassa saatavuudella tarkoitetaan, että palvelu lopulta vastaa jokaiseen sille tehtyyn pyyntöön (Gilbert & Lynch, 2012). Käytännössä tämä merkitsee sitä, että jokainen palvelun ylhäällä oleva solmu vastaa sille tuleviin pyyntöihin. Tällöin, jos järjestelmässä on vain yksi solmu, täyttää se saavutettavuuden määritelmän automaattisesti solmun ollessa ylhäällä. Tulee kuitenkin huomata, että saatavuus ei yksinään takaa 1) pyynnön ja vastauksen välisen viiveen kohtuullisuutta, tai 2) palvelun lähettämän vastauksen eheyttä (datan oikeellisuutta). Kleppmannin (2015) mukaan teoreeman alkuperäinen saatavuuden määritelmä on suhteellisen tulkinnanvarainen, eikä siinä oteta kantaa siihen, tarkoitetaanko saatavuudella algoritmien kautta mahdollistettavaa saavutettavuutta, vai havaittavaa järjestelmän ominaisuutta. Esimerkkinä hän antaa tilanteen, jossa eheä ja saatava (CA) järjestelmä kohtaa verkon osituksen: Jos kaikki järjestelmän solmut hajoavat osituksen seurauksena, CAPin saatavuuden määritelmä ei päde solmuihin, jolloin järjestelmä täyttää saatavuuden määritelmän (Kleppmann, 2015).

Tietokantojen kontekstissa saatavuuden käsite on vahvasti sitoutunut tietokannan virheensietokykyyn, eroten CAP-teoreeman määritelmästä. Erilaisten häiriöiden, kuten laite- ja verkkovikojen, tapahtuessa korkean saatavuuden tietokannanhallintajärjestelmät takaavat, että järjestelmä pysyy saatavana mahdollisimman pienellä häiriöajalla (Zamanian ym., 2019). Saavutettavuus tyypillisesti toteutetaan replikoimalla data monelle eri palvelimelle (Zamanian ym., 2019) erilaisilla replikointiprotokollilla (Dhamane ym., 2014), jotka voidaan luokitella 1) innokkaiksi tai laiskoiksi sekä 2) ”primäärikopioiksi” (primary copy) tai ”kaikkialla päivittäviksi” (update anywhere) (Gray ym., 1996). Replikaatiomenetelmät vaihtelevat huomattavasti tuotekohtaisesti, eroten toisistaan mm. käytettyjen edellä mainittujen protokollien sekä replikointistrategioiden kautta.

Eri tietokannanhallintajärjestelmien kuuluminen CAP-saatavuuden piiriin on näin ollen paljolti tuotekohtaista, monien DBMS:ien mahdollistaessa saatavuuden järjestelmän konfiguroinnin kautta. Yleisesti ottaen RDBMS:t (Jowan ym., 2016) sekä monet NoSQL-tuotteet, kuten Amazon Dynamo, Cassandra ja Riak (Ganesh Chandra, 2015) luokitellaan kuitenkin ilman erityistä konfigurointia CAP-saataviksi tietokannanhallintajärjestelmiksi.

4.3 Verkon osituksen sietokyky

Verkon osituksen sietokyvyllä viitataan CAP-teoreemassa järjestelmän hajautuksesta johtuvaan piirteeseen. Kun järjestelmä hajautetaan moneen solmuun,

käytännössä eri palvelimille, solmujen välisestä kommunikaatiosta tulee epäluotettavaa ja järjestelmä voi ”osittua”, jolloin tietyt solmut eivät pysty kommunikoimaan toisten solmujen kanssa (Gilbert & Lynch, 2012). Verkon osituksen sietokyvyllä viitataan tällöin järjestelmän kykyyn toimia järjestelmän solmujen osituksen tapahtuessa. Verkon osituksen sietokyky saattaa sisältää myös muita hajautetuissa järjestelmissä ilmenevien yleisten ongelmien kuten solmujen kaatumisen tai väliaikaisen pysähtymisen sietokykyä, mutta tämä jää teoreemassa tulkinnanvaraiseksi (Kleppmann, 2015).

Tietokannanhallintajärjestelmien yhteydessä on olemassa monia hyvin erilaisia tapoja sietää verkon osittumista. Lähtökohtana kuitenkin CAP-teoreeman perusteella on, että verkon osittumisen sietäminen johtaa joko eheyden tai saatavuuden heikentymiseen järjestelmässä. Lähestymistapoja tähän on teoreemasta johdettuna kaksi: CP- ja AP-”kategorioiden” mukaan. CP-järjestelmässä osituksen tapahtuessa kaikkien järjestelmän solmujen data pystytään pitämään eheänä, mutta kaikki solmut eivät ole täysin saatavilla esimerkiksi solmujen välisen datan synkronisoinnin aikana. AP-järjestelmissä puolestaan solmut pysyvät saatavilla, mutta ei ole taetta, että jokaisen solmun data on jokaisella hetkellä eheää, vaan data on esimerkiksi ”lopulta eheää” synkronisoiduttuaan kaikille solmuille.

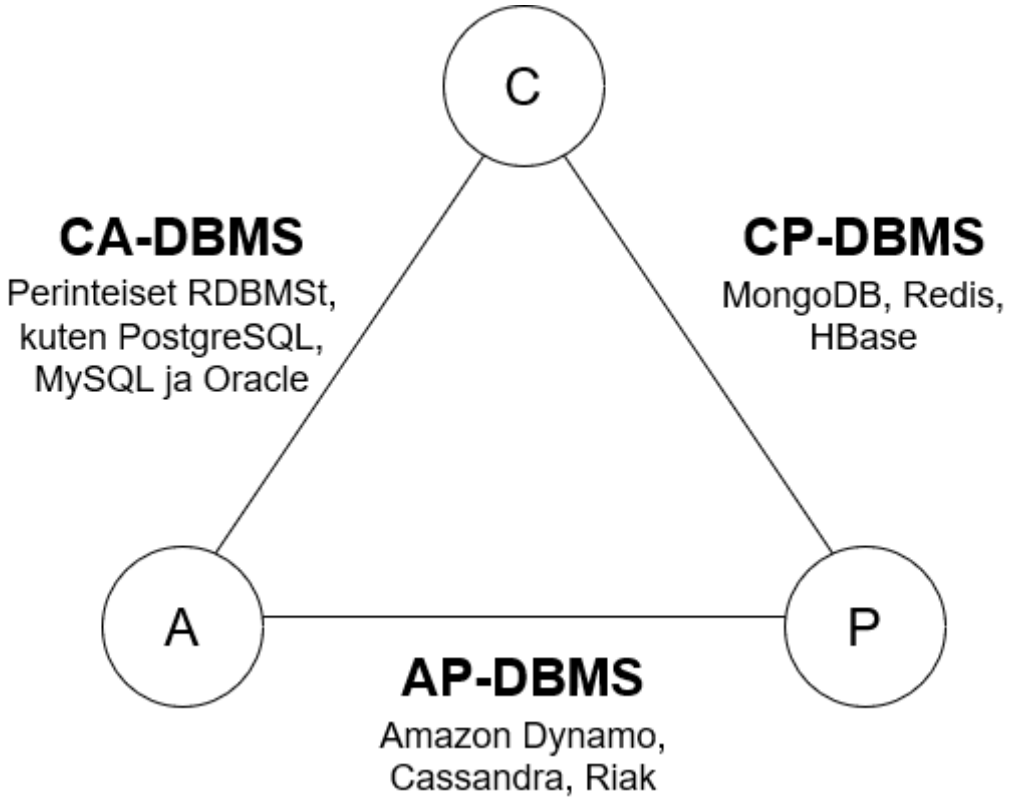
Myös tietokannanhallintajärjestelmien verkon osituksen sietokyky on hyvin tuote- ja konfiguraatiokohtaista. Hyvin monet NoSQL-tietokannanhallintajärjestelmät tarjoavat verkon osituksen sietokyvyn valmiina ominaisuutena, kun taas perinteiset RDBMS:t eivät tue sitä käytännössä ollenkaan (Han ym., 2011). DBMS:ään suunniteltu verkon osituksen sietokyky viittaa siihen, että DBMS on alun perin suunniteltu toimimaan hajautettuna. RDBMS:ien tapauksessa tulee kuitenkin huomata, että vaikka perinteisesti tämän kategorian tietokannanhallintajärjestelmät kuten PostgreSQL ajetaan yhdellä solmulla, voidaan instansseja sijoittaa monelle eri solmulle replikoinnin kautta (IBM, ei pvm.).

4.4 Tietokannanhallintajärjestelmien kategorisointi teoreeman mukaan

Voidaan huomata, että CAP-teoreeman määrittelemät palveluiden halutut ominaisuudet eivät ole niin yksinkertaisia, kuin aluksi vaikuttaa. Ominaisuuksien määritelmät vaihtelevat niin kontekstin, kuin tulkinnan mukaan, jonka lisäksi määritelmät ovat nykyaikaisten DBMS:ien kannalta liukuvia konfiguraation kautta. Tämän lisäksi itse teoreemaa kohtaan on löydetty perusteltua kritiikkiä. Tästä huolimatta teoreema on vieläkin yleisessä käytössä muun muassa tietokannanhallintajärjestelmien suunniteltaessa, mikä puoltaa teoreeman soveltamisen hyödyllisyyttä sekä oikeellisuutta tässäkin tutkielmassa.

Teoreemasta tunnistettavia kategorioita voidaan käyttää lajittelemaan tietokannanhallintajärjestelmiä. Tätä voidaan puolestaan hyödyntää tietyn tietokannanhallintajärjestelmän soveltuvuuden arviointiin tietokantaa käyttävän ohjelmiston käyttötapauksiin verraten. Esimerkiksi jos tietokannalta vaaditaan

ehdotonta datan eheyttä kaikissa käyttötapauksissa, viestii tämä, että joko CA- tai CP-kategorian DBMS:t täyttävät kyseisen ohjelmiston tarpeet. Kuviossa 3 esitellään joitakin yleisimpiä tietokannanhallintajärjestelmiä luokiteltuna näihin kategorioihin, mukaillen Han ym. (2011) sekä monissa muissa julkisissa lähteissä esiintyviä luokittelukuviota.



KUVIO 3 CAP-teoreeman kategorioihin luokiteltuja tietokannanhallintajärjestelmiä

On tärkeää huomioida, että kuviossa (kuvio 3) on esitettynä vain hyvin suppeasti tietokannanhallintajärjestelmiä johtuen markkinoilta löytyvien tuotteiden valtavasta määrästä. Tulee myös painottaa, että tietokannanhallintajärjestelmien sijoittuminen kategorioihin voi muuttua DBMS:n konfiguraation perusteella jopa tapahtumatasolla. Esimerkiksi Cassandran halutut tapahtumat voidaan konfiguroida AP-kategoriasta CP-kategoriaan lisäämällä niille eheysrajoitteita (The Apache Software Foundation, ei pvm.).

Luokittelun pohjalta on kuitenkin mahdollista tehdä alustavaa tietokannanhallintajärjestelmien soveltuvuuden rajausta peilaamalla haluttuja ominaisuuksia sekä kategorioita järjestelmään, jolle soveltuvuutta arvioidaan. Markkinoilta löytyvien tuotteiden valtavan määrän takia tämä on hyödyllistä sekä perusteltua, jotta 1) löydetään teoreettinen lähtökohta DBMS:ien rajaukselle sekä 2) vältetään jokaisen mahdollisen DBMS:n tuotekohtainen tarkastelu, joka olisi äärimmäisen aikaa vievä prosessi. Toisaalta kategoriapohjaista luokittelua ei voida pitää tiettyjä DBMS:iä täysin pois rajaavana tekijänä, johtuen ominaisuuksien tulkinnanvaraisuudesta sekä järjestelmien liikkumavarasta niiden välillä.

4.5 TIMin käyttötapausten vaatimusten arviointi teoreeman näkökulmasta

TIMin yleisimmät käyttötapaukset tietokannan näkökulmasta koostuvat lähinnä tavanomaisista operaatioista, kuten luku ja kirjoitus. Oletettavasti järjestelmä pystyisi mukautumaan mihin tahansa CAPin kategorioista, mutta jokaisen ominaisuuden tarpeelle voidaan tunnistaa sekä puoltavia että hylkääviä tekijöitä. Täten teoreeman esittämien kategorioiden vertaaminen käyttötapauksiin on mielekästä, koska se tarjoaa suuntaa antavan näkökulman tietokannanhallintajärjestelmän valinnan pohjaksi.

Ensimmäinen esitetty käyttötapaus, dokumentin luominen, näyttäytyy dokumenttivaraston näkökulmasta kirjoitusoperaatioina. TIMin kehittäjät arvioivat käyttötapaukselle seuraavat rajoitteet:

- Dokumentin ensimmäisen version tulee syntyä oikeassa muodossa (jossain solmussa).
- Dokumentin ensimmäisen version tulee syntyä ja olla heti saatavilla (jossain solmussa).
- Jos tietokanta on hajautettu, dokumentin tulee syntyä ainakin yhdessä solmussa, mutta välitöntä kaikille solmuille tapahtuvaa päivitystä ei vaadita.

Käyttötapauksen vaatimukset viittaavat TIMin kehityksen arvion mukaan seuraavaan teoreeman esittämien ominaisuuksien tärkeysjärjestykseen: 1. eheys, 2. saatavuus ja 3. osituksen sietokyky. Arvio vaikuttaa paikkansapitävältä. Erityisesti eheyden ja saatavuuden tärkeys korostuu: järjestelmän käytettävyyden kannalta on oleellista, että dokumentti syntyy ja on saatavilla oikeassa muodossa heti luonnin jälkeen, mikä tukee CA-kategorian DBMS:n valintaa. Jos tietokanta hajautetaan, voidaan arvioida, että dokumentin luonnin käyttökokemuksen kannalta on tärkeämpää, että DBMS tukee eheyttä. Tilanne, jossa dokumentin luontioperaatio on näennäisesti saatavilla, mutta luotu dokumentti ei ole eheä (esim. dokumentista puuttuu osioita tai pahimmassa tapauksessa koko dokumentti), vaatii käyttäjältä korjaustoimenpiteitä. Tämä voi puolestaan johtaa tilanteeseen, jossa korjaukset eivät ole eheitä, jolloin dokumentin oikeellisuus kärsii entisestään, mikä puolestaan puoltaa heti alkuperäisen operaation hylkäämistä, jos sen eheyttä ei voida taata. Näin ollen voidaan olettaa, että hetkellinen katkos saatavuuteen on siedettävämpää kuin eheyden menettäminen dokumentin luonnissa. Täten hajautetun tietokannan tilanteessa käyttötapaus vaikuttaa vaativan enemmän CP-, kuin AP-kategorian DBMS:ää.

Toinen käyttötapauksista eli dokumentin lohkojen hakeminen näyttäytyy dokumenttivarastolle lukuoperaatioina. TIMin kehittäjien käyttötapaukselle arvioimat rajoitteet ovat:

- Dokumentin lohkojen uusimmat versiot tulee löytyä (jostain solmusta). Siedetään kuitenkin, että haku voi hetkellisesti palauttaa vanhempia versioita.
- Dokumentin halutun version kaikki lohkot tulee lopulta palauttaa. Siedetään (/suositaan) kuitenkin, että lohkoja voidaan palauttaa useamman pyynnön joukoissa.
- Lisäksi siedetään, jos dokumentin versio on luettavissa vain yksittäisessä solmussa.

Kehitystiimi arvioi tämän pohjalta teoreeman ominaisuuksien tärkeysjärjestykseksi seuraavan: 1. saatavuus, 2. eheys ja 3. osituksen sietokyky. Arvio vaikuttaa aluksi oikealta, mutta siitä nousee muutamia näkökulmallisia kysymyksiä erityisesti hajautetun tietokannan tilanteessa. Jälleen käyttötapauksen vaatimukset vaikuttavat vahvasti tukevan CA-kategorian DBMS:n valintaa, perustuen eheyden ja saatavuuden oleellisuuteen järjestelmän käytön kannalta, kategorian täyttäessä kaikki esitettyjen rajoitteiden vaatimukset normaalitoiminnassa.

Hajautetun tietokannan tilanteessa käyttötapaus on kuitenkin monimutkaisempi. Mitä kauemmin siedetään vanhempien dokumenttiversioiden palauttamista, sitä enemmän eheyden tärkeys vähenee, tukien saatavuuden priorisointia. Koska tälle siedettävää ajanjaksoa on vaikea määrittää, ilmeten kehityksen kuvailemasta ”hetkellisyydestä”, eheyden tärkeyttä tässä käyttötapauksessa on vaikea yleistää. Riippuen haetusta dokumentista, sen ajantasaisuuden kriittisyys voi olla hyvin käyttötapaus- ja käyttäjäkohtaista. Oletettavasti TIMin kaltaisessa opintojärjestelmässä ei kuitenkaan käsitellä sellaista tietoa, että sen hetkellinen ajantasaisuuden puute vaikuttaisi kriittisesti järjestelmän toimintaan ja/tai käyttökokemukseen. Saatavuuden priorisointia tukee myös lohkojen hakutapa. Jos lohkojen hakemisen hajauttamista useille kyselyille hyödynnetään, toisin sanoen kyselyjen määrä kasvaa, nousee saatavuuden tärkeys entisestään, jotta pystytään hakemaan edes jonkin version dokumentti. Jos eheyttä suosittaisiin tässä tilanteessa, jotkut lohkojen hakupyynnöt saattaisivat jäädä toteuttamatta, tällöin kuitenkin rikkoen dokumentin eheyden kokonaisuuden näkökulmasta. Täten hajautetun tietokannan tilanteessa lohkojen haun käyttötapaus näyttää pääsääntöisesti suosivan enemmän AP-, kuin CP-kategorian DBMS:ää, riippuen kuitenkin mahdollisista dokumenttikohtaisista tekijöistä.

Kolmas käyttötapaus, versiohistorian hakeminen, mukailee dokumentin lohkojen hakua dokumenttivaraston näkökulmasta. Käyttötapaukselle on arvioitu seuraavat rajoitteet:

- Siedetään, että käyttäjä ei saa ajantasaista versiohistoriaa.
- Käyttäjän tulee saada jokin vastaus, mutta siedetään, jos versiohistoriaa ei ole hetkellisesti ladattavissa näkyviin.
- Versiohistoriaa ei tarvitse ladata, jos se ei ole saatavilla.

Kehitystiimi arvioi näiden pohjalta teoreeman ominaisuuksien tärkeysjärjestykseksi seuraavan: 1. saatavuus, 2. eheys, 3. osituksen sietokyky.

Ominaisuuksien tärkeysjärjestystä on hankalaa lähteä arvioimaan, koska käyttötapaus ei selkeästi esitettyihin rajoitteisiin perustuen ole kriittinen. Sivuhuomiona on kuitenkin esitetty, että tämänhetkisessä toteutuksessa versiohistoriaa käytetään dokumenttien lohkojen haussa uusimman version selvittämiseen. Tällöin käyttötapaus nousee hyvin kriittiseksi, koska ilman tietoa uusimmasta versiosta myös lohkojen haun käyttötapaus epäonnistuu.

Jos oletetaan, että vaihtoehtoisessa toteutuksessa tieto uusimmasta versiosta haetaan samalla tavalla kuin tällä hetkellä, nousee käyttötapauksen merkitys hyvin suureksi. Koska lohkojen haun käyttötapauksessa siedetään, että haku palauttaa hetkellisesti vanhempia versioita, voidaan olettaa, että myös versiohistorian haun tapauksessa siedetään hetkellistä vanhan ”uusimman” version palauttamista. Tällöin käyttötapauksessa saatavuuden tärkeys nousisi eheyttä korkeammaksi, kuten alun perin on arvioitu. Kuitenkin myös eheyden merkitys on hyvin suuri: Ilman oikeellista tietoa haettavan dokumentin versiosta dokumentin lohkojen haulla ei ole edes mahdollisuutta hakea haluttuja lohkoja. Täten vaikuttaisi, että käyttötapaus suosii enemmän AP-, kuin CP-kategorian DBMS:iä hajautetussa tilanteessa, mutta molemmilla kategorioilla on painavat hyöty- ja haittapuolensa. Tulee myös huomata, että DBMS:tä riippuen uusimman dokumenttiversion haku voidaan luultavimmin sisällyttää lohkojen hakemiseen, jolloin erillistä versiohistorian kyselyä ei tarvita.

Viimeinen käyttötapaus eli dokumentin muokkaaminen mukailee uuden dokumentin luomista, koska muokkausten tapahtuessa dokumentista luodaan uusi versio. Käyttötapauksen rajoitteet ovat kuitenkin jokseenkin erilaisia johtuen esimerkiksi mahdollisesta monen käyttäjän samanaikaisesti suorittamasta operaatiosta. TIMin kehittäjien käyttötapaukselle arvioimat rajoitteet ovat seuraavat:

- Muutosten tulee kohdistua dokumentin uusimpaan versioon. Ristiriitatilanteessa, jossa moni käyttäjä tekee samanaikaisia muutoksia, siedetään, että muutokset kohdistuvat samaan ”vanhaan” versioon, vaikka uudempi muokattu versio olisi luotu muutosten teon aikana.
- Muutoksia ei saa kadottaa. Samanaikaisten muutosten tilanteessa kaikkien muutosten tulee tallentua.
- Siedetään, että muutokset tallentuvat hetkellisesti vain yhteen solmuun, mutta näiden tulee lopulta synkronisoitua kaikkialle.

Tämän pohjalta teoreeman ominaisuuksien tärkeysjärjestykseksi käyttötapauksessa on arvioitu kehittäjien puolesta seuraava: 1. saatavuus, 2. eheys, 3. osituksen sietokyky. Tässä tapauksessa saatavuuden priorisointia eheyden edelle tukee rajoitteista ilmenevä siedettävyys. Koska ristiriitatilanteissa siedetään, että muokkaukset voivat kohdistua dokumentin ”vanhaan” versioon, mutta samalla muokkauksia ei saa kadottaa, voidaan yleistää, että saatavuuden priorisointi ja ”lopulta eheä” eheystaso täyttäisivät esitetyt rajoitteet. Jos eheyttä priorisoitaisiin esimerkiksi lukitusten kautta (sekä verkon osittuessa, olettaen että kanta on

hajautettu), voisivat muutokset kadota. Täten hajautetun kannan tilanteessa käyttötapaus vaatii AP-kategorian DBMS:än.

Yleisesti ottaen voidaan todeta, että käyttötapausten vaatimusten kategorisointi CAP-teoreeman kategorioihin ei ole suoraviivaista. Vaikka esitetyt alustavat arviot vaikuttavat pääsääntöisesti paikkaansa pitäviltä, tietyt poikkeustilanteet voivat vaatia käyttötapauksen näkökulmasta täysin vastakkaista sietokykyä eheys-saatavuus-akselilla. Tulee myös painottaa, että jos tietokanta hajautetaan, osituksen sietokyky on DBMS:lle pakollinen ominaisuus, jos tietokannan toimintakyky halutaan varmistaa. Täten osituksen sietokyvyn tarvetta ei voida vertailla saatavuuden ja eheyden kanssa suoraan. Arviointi tulisi näin ollen teoreeman kaksi kolmesta ominaisuudesta -implikaation pohjalta tehdä saatavuutta ja eheyttä vertaillen, kuten esitetyistä rajoitteista on pyritty tässä tutkielmassa päättelemään.

Käyttötapauksia tarkastellessa voidaan huomata, että sekä eheys ja saatavuus ovat tapauskohtaisesti molemmat tärkeitä, mikä tukee CA-kategorian DBMS:än valintaa vahvasti. Jos tietokanta puolestaan päätetään hajauttaa, vaikuttaisi että saatavuutta tulisi priorisoida eheyden edelle. Arvio perustuu varsinkin järjestelmässä käytettävään dataan: Koska järjestelmässä ei käsitellä erityisen kriittistä dataa, kuten esimerkiksi pankkitapahtumia, sekä sen käytön saatavuuden taso halutaan pitää mahdollisimman korkeana, suositaan AP-kategorian DBMS:iä datan loogisen eheyden varmistamisen kustannuksella.

Koska moderneja DBMS:iä ei voida tiukasti sijoittaa kategorioihin, vaan ne voivat liikkua kategorioiden välillä tuotekohtaisesti konfiguroituna jopa transaktiotasolla, tulee arviointia suorittaa myös toisesta näkökulmasta vaihtoehtojen rajaamiseksi. Tästä syystä tutkielmassa tarkastellaan seuraavaksi erilaisia tietokantaparadigmoja, joita pystytään peilaamaan järjestelmän tietotarpeisiin, rajaten valintaa lisää.

5 TIETOKANTAPARADIGMAT

Tässä luvussa käsitellään erilaisia tietokantaparadigmoja, arvioiden niiden soveltuvuutta TIMin tietomallin tarpeisiin. Tietokantaparadigmat määrittävät vahvasti minkälainen data soveltuu tietokannan käyttöön, miten data mallinnetaan tietokantaan sekä minkälaisen järjestelmän käyttöön itse DBMS on suunniteltu. Täten tietokantaparadigmojen kriittisen tarkastelun perusteella voidaan alustavasti arvioida sekä rajata soveltuvia yksittäisiä DBMS-tuotteita laajemmasta näkökulmasta.

Arvioinnin alla on kolme eri tietokantamallia: relaatiotietokannat, NoSQL-tietokannat sekä NewSQL-tietokannat. Arvioinnin piiristä on jätetty tietoisesti pois hierarkia- ja verkkotietokannat sekä oliotietokannat. Hierarkia- ja verkkotietokantojen rajaaminen tutkielman ulkopuolelle johtuu mallien korvaantumisesta toisilla malleilla sekä niiden käytön vaikeudesta. Davoudian ym. (2018) toteavat, että hierarkia- ja verkkotietokantojen tietomalli johtaa näitä tietovarastona käyttävän sovelluksen huomattavaan monimutkaisuuteen niin luonnissa kuin muokkauksessa. Tämä itsessään viestii mallien soveltumattomuutta TIMin käyttötapaukseen, koska mallien tietokantojen käyttöönotto vaatisi huomattavaa ja monimutkaista kehitystyötä, mikä ei ole jo valmiiksi toiminnassa olevan järjestelmän näkökulmasta toivottavaa. Tämän lisäksi relaatiotietokannat ovat pääsääntöisesti korvanneet hierarkia- ja verkkotietokannat yleisessä käytössä (Davoudian ym., 2018), mikä puolestaan viittaa mallien soveltumattomuuteen moderneissa käyttöympäristöissä. Oliotietokannat puolestaan vaativat järjestelmältä vahvaa suuntautuneisuutta olio-ohjelmointiin (Davoudian ym., 2018), joten koska tässä tutkielmassa ei pyritä muuttamaan olemassa olevaa TIMin toteutusta, eivät oliotietokannat sovi tutkielman arvioinnin piiriin.

5.1 Relaatiotietokannat

Relaatiotietokannat pohjautuvat Coddin (1970) kehittämään relaatiomalliin. Relaatiomallissa data mallinnetaan monikkoihin, joiden välisiä suhteita

muodostetaan matematiikkaan pohjautuvilla relaatioilla (Codd, 1970). Tällöin datan looginen ja fyysinen rakenne erottuu, mikä edistää datan riippumattomuutta itse ohjelmistojen toteutuksesta (Codd, 1970). Riippumattomuus edistyy entisestään Chamberlin ja Boycen (1974) kehittämällä SQL-kyselykielellä (alun perin SEQUEL), jolla pystytään luonnonkielisesti kohdistamaan operaatioita relaatiotietokantoihin. SQL:stä on myöhemmin muodostunut standardikieli datan operointiin relaatiotietokantoja käytettäessä (Davoudian ym., 2018). Relaatiotietokannat ovat vielä tänäkin päivänä erittäin suosittuja, seitsemän kymmenestä suosituimmasta DBMS:tä ollessa relaatiomalliin perustuvia (solid IT gmbh, 2023a).

Relaatiotietokantojen ydinominaisuuksiin kuuluvat mm. aiemmin mainitut ACID-transaktiot sekä standardisoitu kyselykieli SQL, monimutkaisten kyselyiden tuki, vahvasti mallinnettu data sekä vertikaalinen skaalautuvuus (Mihai, 2020). Näistä ydinominaisuuksista voidaan johtaa yksi relaatiotietokantojen suurimmista hyödyistä: Relaatiotietokannat ovat erityisen hyviä käsittelemään ja ylläpitämään järjestettyä sekä tarkasti määriteltyä dataa. Ennen datan syöttämistä relaatiotietokantaan tulee sille määritellä skeema, joka puolestaan määrittää tarkasti minkälaista dataa tietokannan taulut hyväksyvät sekä esimerkiksi mahdollisia eheysehtoja taulujen välisten relaatioiden välille. Relaatiotietokannoille on kehitetty monia eri metodeja muun muassa takaamaan niissä säilytettävälle datalle yhtenäisyyttä ja tietosuojaa (Kamel & Kamel, 2011 sekä Xu ym., 2010) sekä ohjeistuksia erilaisten ongelmallisten datamuotojen mallintamiseen (Tansel, 2004). Laajasta näkökulmasta voidaan todeta, että relaatiotietokantojen pitkää historiaa sekä mittavan tutkimus- ja (käytännön) käyttötaustaa hyödyntämällä relaatiotietokantoja pystytään soveltamaan erittäin suureen määrään hyvin erilaisia tietotarpeita.

Relaatiotietokantojen käytössä ilmenee mittavista hyödyistä huolimatta myös perustavanlaatuisia haasteita. Relaatiomallin sekä monimutkaisten ACID-transaktioiden takia tietokannan hajauttaminen on äärimmäisen vaikeaa (Băzăr & Iosif, 2014), vaikkakin käytännössä mahdollista esimerkiksi lisäosien avulla (Cubukcu ym., 2021). Tämän takia relaatiotietokannat skaalautuvat pääsääntöisesti vertikaalisesti, mikä tarkoittaa, että relaatiotietokannan ainoalle solmulle pitää lisätä enemmän ja enemmän resursseja, jotta sille saadaan enemmän suorituskykyä (Agrawal ym., 2011). Tällöin datamäärän sekä tietokannan käytön kasvaessa saattaa ilmentyä tilanne, jossa solmulle ei ole enää mahdollista lisätä resursseja kohtuullisten kustannusten rajoissa. Tiukasti määritelty skeema puolestaan saattaa koitua ongelmaksi, jos käytettävän datan muoto muuttuu, tai jos tietomalli on hyvin monimutkainen. Muutokset dataan ovat ongelmallisia, koska muutosten takia itse skeemaa pitää muuttaa, mikä taas puolestaan voi johtaa tarpeeseen mallintaa isompi osa dataa uudestaan. Jo käytössä olevalla suurella tietokannalla tämä voi vaatia huomattavia resursseja, johtaen ylimääräisiin kustannuksiin. Monimutkaisen datan mallintaminen relaatiomalliin voi puolestaan itsessään olla hyvin vaativa tehtävä ja se voi vaikeuttaa relaatiotietokannan käyttöönottoa huomattavasti.

5.2 NoSQL

Edelleen kehittyvän web-ekosysteemin räjähdysmäinen kasvu vuosituhannen alussa on luonut tietokannanhallintajärjestelmille tarpeen pystyä käsittelemään erittäin suuria määriä eri lähteistä saatavaa rakenteellista, puolirakenteellista ja rakenteetonta dataa (Davoudian ym., 2018). On huomattu, että perinteiset relaatiotietokannat eivät ominaisuuksiensa vuoksi sovellu tähän tarpeeseen optimaalisesti, koska ne vaativat erittäin rakenteellista dataa sekä skaalautuvat vain vertikaalisesti.

NoSQL (Not Only SQL tai Not Relational) (Cattell, 2011) -tietokannat kehitettiin vastaamaan näihin uusiin tietokannoille ilmestyneisiin vaatimuksiin. Vaikka NoSQL-tietokannoilla ei ole yhteistä tietomallia, on niillä joitain yhdistäviä ominaisuuksia perustuen edellä mainittuihin vaatimuksiin. Näihin sisältyy Davoudianin ym. (2018) mukaan tiivistettynä seuraavat:

- Joustavat tietorakenteet, mahdollinen skeemattomuus.
- Heikon eheyden transaktiomallit, jotka mahdollistavat horisontaalisen skaalautuvuuden.
- Optimoitu hajautukselle niin datan solmuille jakamisen, indeksöinnin, tiivisteiden, välimuistiin tallentamisen, kuin replikoinnin kannalta.
- Web-pohjaisen käytön helpottaminen yksinkertaisilla asiakassovelluksilla. (Davoudian ym., 2018)

NoSQL-tietomallit, toisin sanoen NoSQL-DBMS:ien paradigmat, voidaan jakaa yleisesti ottaen neljään eri kategoriaan: avain-arvo-, wide-column-, graafi-, ja dokumenttitietokantoihin (Davoudian ym., 2018). Seuraavaksi tarkastellaan näiden kategorioiden erityispiirteitä.

5.2.1 Avain-arvo-tietokannat

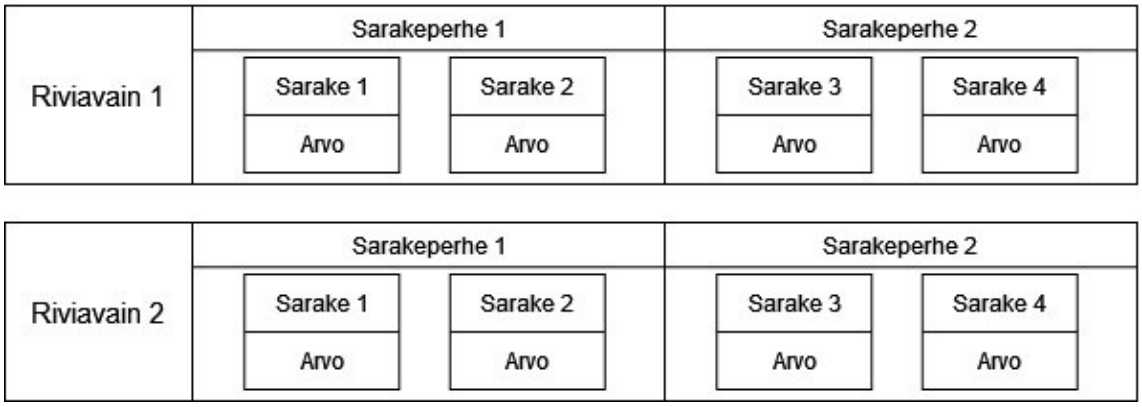
Avain-arvo-tietokannoissa data mallinnetaan avain-arvo-pareihin (Zhou ym., 2014), joissa avaimena yleensä toimii merkkijono ja arvona joku käytettävän ohjelmointikielen primitiivi datatyyppi, kuten merkkijono, taulukko, luku ja niin edelleen (Seeger, 2009). Yleisesti hyväksytyn määritelmän puutteesta huolimatta avain-arvo-tietokannoilla on yhteisiä piirteitä, joita ovat Zhoun ym. (2014) mukaan muun muassa seuraavat:

- Esimerkiksi transaktioeheyden sekä reaaliaikaisten luku- ja kirjoitusoperaatioiden hylkääminen skaalautuvuuden, reliabiliteetin sekä rakenteettoman datan prosessoinnin puolesta.
- Korkea kapasiteetti sekä massavarastoinnin mahdollistaminen, tuki erittäin laajalle hajauttamiselle sekä massiiviselle rinnakkaiskäytölle.
- Vahva laajennettavuus konfiguraation kautta. (Zhou ym., 2014)

Skeemattomina (Davoudian ym., 2018) tietokantoina avain-arvo-tietokannat soveltuvat edellä mainittujen piirteiden perusteella erittäin suurten, rakenteettomien datamäärien tehokkaaseen varastointiin sekä käsittelyyn. Rakenteettomuuden takia datasta kuitenkin häviävät esimerkiksi erilaisten tietokenttien väliset suhteet, jotka pitää tarvittaessa mallintaa sovelluskerroksessa. Täten avain-arvo-tietokannat soveltuvat käyttötapauksiin, joissa käytetään ainoastaan yhtä avainta hakemaan dataa, kuten esimerkiksi ostoskorit sekä sessiotietojen säilytys (Davoudian ym., 2018). Suosituimpia avain-arvo-DBMS:iä ovat Redis, Amazon DynamoDB, Microsoft Azure Cosmos DB sekä Memcached (solid IT gmbh, 2023b), joista ainoastaan Redis yltää kymmenen suosituimman DBMS:n listalle (solid IT gmbh, 2023a).

5.2.2 Wide-column-tietokannat

Wide-column-tietokannoissa dataa jaotellaan sarakeperheissä sijaitseviin riveihin, jotka koostuvat 1…n määrästä sarakkeita, jotka ovat loogisesti toisiinsa liittyviä ja yleensä yhdessä haettuja (Davoudian ym., 2018). Seuraava kuvio (kuvio 4) havainnollistaa wide-column-tietomallia yksinkertaisena versiona mukaillen Davoudian ym. (2018).



KUVIO 4 Wide-column-tietorakenne yksinkertaistettuna, mukaillen Davoudian ym. (2018)

Wide-column-tietokannat tarjoavat tämän yksinkertaisen esimerkin lisäksi lisää tietorakenteita tuotekohtaisesti, kuten esimerkiksi sisäkkäisiä sarakeperheitä eli ”supersarakeperheitä” (Lakshman & Malik, 2010). Tämän perusteella wide-column-tietokannat tarjoavat enemmän mahdollisuuksia mallintaa datalle rakennetta, kuin avain-arvo-mallissa, kuitenkin säilyttäen mahdollisuuden horisontaaliselle skaalautuvuudelle. Tämän lisäksi yleisinä ominaisuuksina toistuvat muun muassa datan versiointi / aikaleimaaminen (Davoudian ym., 2018) sekä suurten yhteen koottujen datapakettien rinnakkainen käsittely (Dean & Ghemawat, 2008).

Wide-column-tietokannat soveltuvat erityisesti sovelluksille, jotka analysoivat sekä käsittelevät suuria määriä dataa sarjoina (Davoudian ym., 2018). Esimerkiksi datavarastot, BI- ja OLAP-sovellukset sekä reaaliaikaiset

analytiikkasovellukset, kuten IoT-sensoreita hyödyntävät monitorointiohjelmistot, kuuluvat wide-column-tietokannoille hyvin soveltuvien käyttötapausten piiriin. Suosituimpia wide-column-DBMS:iä ovat Cassandra, HBase, Microsoft Azure Cosmos DB, Datastax Enterprise sekä ScyllaDB (solid IT gmbh, 2023c), joista Cassandra on 12. suosituin DBMS yleisessä kontekstissa (solid IT dmbh, 2023a).

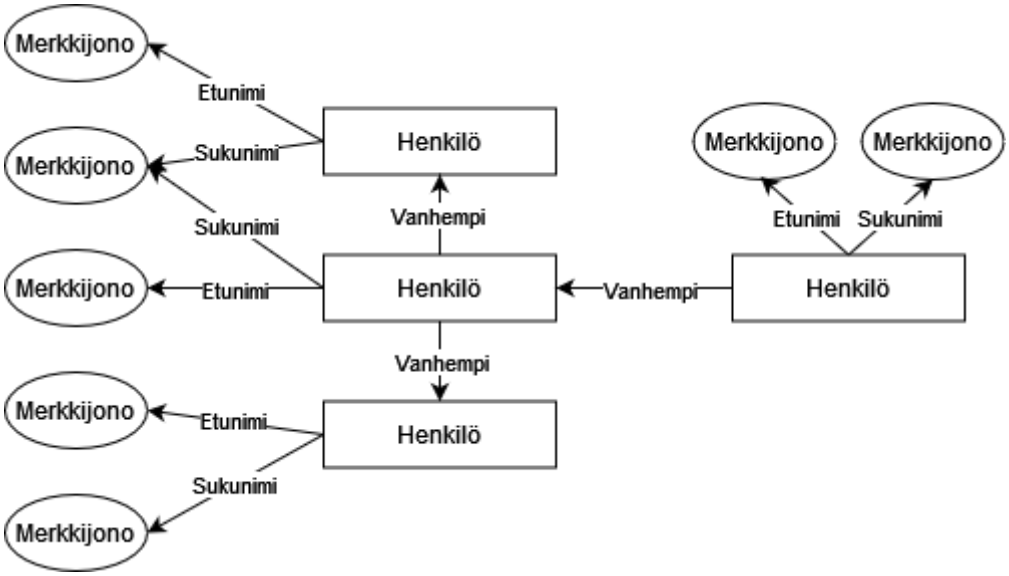
5.2.3 Dokumenttitietokannat

Dokumenttitietokannat ovat ”laajennettuja” avain-arvo-tietokantoja, joissa raakojen arvojen sijaan dataa tallennetaan dokumentteihin puolirakenteellisessa muodossa, esimerkiksi XML:nä tai JSONina (Davoudian ym., 2018). Dokumenttien muoto on joustava: Niiden sisäisiä attribuutteja pystytään muokkaamaan ajon aikana (Davoudian ym., 2018), mahdollistaen tuen alati muuttuville tietotarpeille. Dokumentit voivat itsessään sisältää niin rakenteellista (esimerkiksi avainsanoja, viitteitä muihin dokumentteihin), kuin rakenteetonta (esimerkiksi tekstitiedostoja) dataa (Clifton & Garcie-Molina, 2000). Wide-column-tietokantojen tapaan jotkut dokumenttitietokannat tarjoavat tuotekohtaisesti tapoja lisätä datalle (dokumenteille) rakennetta koosteiden muodossa. Esimerkiksi MongoDB tarjoaa dokumenteille ”collection”eja, jotka vastaavat relaatiotietokantojen tauluja (MongoDB, Inc., 2023a), mahdollistaen dokumenttien ryhmittelyn loogisesti.

Dokumenttitietokannat soveltuvat mallinsa takia käyttötapauksiin, joissa tallennettava tieto voidaan sujuvasti käsittää dokumenttina. Tämän kaltaisia käyttötapauksia on erittäin suuri määrä, mutta esimerkiksi sisällönhallintasovellukset ovat erittäin soveltuva käyttötapaus johtuen hyvin vaihtelevasta (sekä mahdollisesti jatkuvasti muuttuvasta) datastaan. Dokumenttitietokannat voivat yleisellä tasolla myös mm. vähentää kehittäjien työmäärää erittäin joustavan tietomallinsa takia. Suosituimpia dokumenttitietokannanhallintajärjestelmiä ovat MongoDB, Amazon DynamoDB, Databricks, Microsoft Azure Cosmos DB sekä Couchbase (solid IT gmbh, 2023d) joista MongoDB on viidenneksi suosituin kaikki DBMSt huomioiden (solid IT gmbh, 2023a).

5.2.4 Graafitietokannat

Graafitietokantojen tietomalli perustuu nimensä mukaan datan graafeihin mallintamiseen. Graafimallissa skeeman ja/tai tietokantainstanssien tietorakenteet mallinnetaan verkkoihin tai näiden yleistyksiin (Angles & Gutierrez, 2008). Graafitietokannoissa hyödynnetään monia erilaisia graafimalleja, joiden rakenne, toiminta ja sovelluskohteet vaihtelevat toistensa välillä (Davoudian ym., 2018) järjestelmän tietotarpeiden mukaan. Graafitietokantojen tietomallissa keskitytään erityisesti datapisteiden välisten suhteiden hyödyntämiseen, mahdollisesti jopa enemmän kuin itse dataan (Angles & Gutierrez, 2008). Seuraavassa kuviossa (kuvio 5) havainnollistetaan yksinkertaistettuna yhtä graafitietokantojen mahdollista tietomallia, GMOD:ia, mukaillen Anglesia ja Gutierrezia (2008).



KUVIO 5 GMOD-graafimalli, mukaillen Angles ja Gutierrezia (2008)

Graafitietokantojen hyötyihin sisältyy erityisesti datan muodon sekä datapisteiden välisten suhteiden mallintaminen (Angles & Gutierrezia, 2008). Näin ollen myös graafitietokantojen optimaaliset sovelluskohteet hyödyntävät näitä ominaisuuksia. Esimerkkinä sopivista graafitietokantojen sovelluskohteista ovat mm. sosiaaliset verkostot (Angles & Gutierrezia, 2008) sekä proteiinien välisten vaikutusten mallintaminen biologisissa järjestelmissä (Davoudian ym., 2018). Suosituimpia graafitietokannanhallintajärjestelmiä ovat Neo4j, Microsoft Azure Cosmos DB, Virtuoso, ArangoDB sekä OrientDB (solid IT gmbh, 2023e), joista Neo4j on suosituimpana sijalla 22. yleisessä vertailussa (solid IT gmbh, 2023a).

5.3 NewSQL

NewSQL on vuonna 2011 muodostunut termi, johon sisältyvät modernit relaatiopohjaiset tietokannanhallintajärjestelmät, jotka yhdistelevät perinteisten RDBMS:ien ACID-transaktioperiaatetta NoSQL-DBMS:ien skaalautuvuuteen (Pavlo & Aslett, 2016). CAP-teoreeman takia NewSQL-järjestelmät eivät voi kuitenkaan täysin tarjota kaikkia haluttuja ominaisuuksia, vaan niiden on tehtävä kompromisseja em. haluttujen ominaisuuksien välillä. NewSQL-järjestelmät voidaan jaotella kolmeen kategoriaan, joita ovat: uuden arkkitehtuurin järjestelmät, hajautusta tukevat väliohjelmistot (middleware) ja pilvipohjaiset NewSQL DBaaS-tuotteet (Pavlo & Aslett, 2016). NewSQL-järjestelmien sisäiset arkkitehtuurit sekä mekanismit vaihtelevat erittäin paljon tuotteiden välillä (Chaudhry & Yousaf, 2020), joten tämän ”luokan” alaisuuteen sijoittuvien tietokannanhallintajärjestelmien arviointia on vaikeaa suorittaa yleispätevästi menemättä tuotekohtaisen tarkastelun tasolle.

Yleisellä tasolla voidaan kuitenkin todeta, että NewSQL-tietokannanhallintajärjestelmien käyttökohteena ovat modernit, vertikaalista hajautusta vaativat

sovellukset, joiden data soveltuu relaatiomalliin tai on jopa jo mallinnettu perinteiseen RDBMS:ään. Tarkemmin ottaen Pavlon ja Aslettin (2016) mukaan NewSQL-DBMS:t soveltuvat sovelluksille, joiden tietokantaan kohdistuvat transaktiot ovat lyhytikäisiä (eivät odota käyttäjän toimenpiteitä), toisteisia (vaihtelevilla syötteillä) sekä käsittelevät vain pieniä datan osia indeksejä hyödyntäen (eivät käytä laajoja hajautettuja yhdistelmälauseita tai kokonaisten taulujen skannauksia). Tämän pohjalta voidaan johtaa, että NewSQL-DBMS:t eivät sovellu monimutkaisia, vaihtelevia tietotarpeita omaaville järjestelmille, vaan yksinkertaisia, toisteisia kyselyitä käyttäville sovelluksille, jotka vaativat suuren datamäärän puolesta mahdollisuuden hajautukselle kuitenkin säilyttäen ACID-transaktioiden tarjoamat ominaisuudet. Pavlon ja Aslettin (2016) esittelemistä ”uuden arkkitehtuurin järjestelmistä” suosituimpia NewSQL-DBMS:iä ovat SAP HANA, CockroachDB, SingleStore (entinen MemSQL), VoltDB sekä NuoDB (solid IT gmbh, 2023a).

5.4 Yhteenveto tietokantaparadigmoista

Voidaan huomata, että tietokantaparadigmasta (sekä niiden mahdollisista sisäisistä luokitteluista) riippuen, tietokannanhallintajärjestelmille sopivat sovelluskohteet vaihtelevat erittäin paljon. Tämän lisäksi tulee huomata, että tuotekohtaiset eroavaisuudet voivat vaikuttaa hyvin paljon soveltuvuuden arvioinnissa. Kuitenkin yleisellä tasolla tietokantaparadigman kautta arviointi mahdollistaa kaikkien DBMS:ien joukosta potentiaalisten valintojen rajaamiseen sovelluskohteelle.

Seuraavaksi esitellään relaatiotietokantojen, NoSQL-tietokantojen sekä NewSQL-tietokantojen yleiset ydinominaisuudet toisistaan eroteltuna (Taulukko 1), minkä jälkeen koostetaan esiteltyjen tietokantamallien etuja, rajoitteita sekä sovelluskohteita (Taulukko 2). Tulee huomioida, että listaukset ovat erittäin yleistettyjä sekä kattavat vain osittain tietokantojen haluttuja ominaisuuksia. Tämän lisäksi osa listatuista ominaisuuksista on johdettu hyvin yleisen näkökulman arvioinnista, jolloin ominaisuudet voivat vaihdella huomattavasti mallien sisäisten toteutusten mukaan. Tästä huolimatta ominaisuuksien esittelyllä saavutetaan tässä tutkielmassa vaadittava tarkkuus tietokannanhallintajärjestelmien alustavaan rajaukseen tietokantaparadigmojen perusteella.

TAULUKKO 1 Relaatio-, NoSQL- ja NewSQL-tietokantojen ydinominaisuudet

Kategoria	Transaktiomalli	Kyselykieli	Datan mallinnus	Skaalautuvuus	Kompleksisuus
Relaatio	ACID	SQL	Tiukka	Vertikaalinen	Korkea
NoSQL	Pääsääntöisesti BASE	Ei standardia, tuotekohtainen	Pääsääntöisesti hyvin joustava	Horisontaalinen (hajautettu)	Yleisesti ottaen matala
NewSQL	ACID	SQL	Tiukka	Horisontaalinen	Hyvin korkea (relaatiokantojen ominaisuudet yhdistettynä hajautettuun ympäristöön)

TAULUKKO 2 Esitettyjen tietokantamallien edut, rajoitteet sekä soveltuvuus

Malli	Edut	Rajoitteet	Soveltuvuus
Relaatio	Datan rakenteen ylläpitäminen, laaja tutkimus- ja käyttöhistoria, ACID	Hajauttamisen haastavuus (skaalautuu vertikaalisesti), skeeman muuttaminen ongelmallista, datan mallintaminen mahdollisesti haastavaa	Relaatiomalliin soveltuvan datan sovellukset, erittäin laajat sovellusmahdollisuudet
Avain-arvo	Korkea kapasiteetti, massiivinen hajautus helppoa, yksinkertaisuus	Datan rakenteen ja osien välisten suhteiden mallintaminen tietokantatasolla	Yksinkertaiset kyselyt, kuten ostoskorit ja sessiotietojen säilytys
Wide-column	Suurten datamäärien varastointi helppoa, datan versiointi, rakenteen lisäämismahdollisuudet	Skeeman joustamattomuus, yksittäisten ad-hoc -kyselyiden tekeminen (Davoudian ym., 2018)	Analyyttiset sarjoittaista dataa käsittelevät sovellukset, kuten datavarastot ja BI-sovellukset
Dokumentti	Joustava tietomalli, ajonaikainen skeeman muokkaus, mahdollisuudet rakenteeseen tuotekohtaisesti	Dataa tulee pystyä mallintamaan dokumenttimuotoon	Dokumenttina mallinnettavan datan sovellukset, erittäin laajat sovellusmahdollisuudet
Graafi	Tietorakenteiden välisten suhteiden mallinnus	Jos järjestelmä ei painotu suhteiden tarkasteluun, ei välttämättä sovellu hyvin	Tiedon suhteita vaativat sovellukset, kuten sosiaaliset verkostot
NewSQL	Relaatiokantojen hyödyt + hajautus	Erittäin monimutkaisia järjestelmiä, ei sovellu monimutkaisiin tietotarpeisiin	Relaatiomallia käyttävät yksinkertaiset tietotarpeet, jotka vaativat hajautusmahdollisuuden

5.5 TIMin tietotarpeen vaatimusten arviointi tietokantaparadigmojen näkökulmasta

Voidaan huomata, että järjestelmän tietomallin sekä käyttötarkoituksen soveltuvuuden arviointi eri tietokantaparadigmoille tukee soveltuvimman mahdollisen DBMS:n valintaa. Seuraavaksi suoritetaan arviointia TIMin näkökulmasta, jotta aiemmin esitettyä CAP-teoreeman pohjalta tehtyä valinnan rajausta voidaan tarkentaa.

TIMin dokumenttien tietorakenne on suhteellisen yksinkertainen. Itse dokumentit yksilöidään ID:llä sekä niihin liitetään versionumero, lohkolista sekä muutosloki. Lohkot, jotka tällä hetkellä ovat rakenteeltaan yksittäisiä JSON-objekteja, sisältävät erilaisia yksittäisiä kenttiä, jotka sisältävät joko lukuja, tekstiä tai avain-arvo-pareja. Muutosloki on hyvin samanlainen, sisältäen JSON-objekteja dokumenttiin tehdyistä muutoksista omilla kentillään.

Relaatiomallin tietokannanhallintajärjestelmän valintaa TIMin käyttöön tukee järjestelmässä käytetyn datan rakenne. Tietomallia tarkastellessa voidaan huomata, että dokumenteissa käytetyt erilaiset datarakenteet ovat vahvasti toisiinsa liitoksissa ja että ne ovat tarkasti määriteltyjä. Relaatiomallin vahvuuksista erityisesti ACID-kyselyiden tarjoama eheys ja saatavuus, kuten luvussa 5.1 on mainittu, ovat tärkeitä. Tämän lisäksi datan loogisen rakenteen ylläpitäminen on järjestelmälle hyvin hyödyllistä, jotta myös tietokannan puolella voidaan varmistaa, että esimerkiksi lohkoja ei lisätä dokumentille, jota ei ole olemassa. Relaatiomallin yleisistä rajoitteista oletettavasti ainoastaan hajauttamisen vaikeus vaikuttaa negatiivisesti mallin valintaan. Dokumenttien datan skeema on vakiintunut järjestelmän pitkäjänteisen kehitysajan myötä, joten siihen ei odoteta samanlaisia äkillisiä muutoksia, kuten esimerkiksi uudessa projektissa voitaisiin olettaa olevan. Täten skeeman muuttamisen hankaluuden ei oleteta vaikuttavan soveltuvuuteen. Myöskään datan mallintaminen relaatiomalliin ei vaikuta haastavalta, vaan esimerkiksi alustavat taulurakenteet on helppo hahmottaa nykyisestä kuvauksesta. Hajauttamisen haastavuus, eli horisontaalisen skaalautuvuuden puute, tulee näyttäytymään kasvavan datamäärän takia suorituskyvyssä ja/tai järjestelmän ylläpidon kustannuksissa. Koska relaatiomalli vaikuttaa kuitenkin sopivan järjestelmän tarpeisiin erinomaisesti, toimii se varteenotettavana vaihtoehtona valittavaksi malliksi.

Avain-arvo-malli vaikuttaa ensisilmäyksellä hyvältä vaihtoehdolta suorituskykynsä sekä helpon hajautettavuutensa perusteella, mutta mallista nousevat rajoitteet johtavat sen sopimattomuuteen TIMille. Koska TIMin dokumenttien hallinnan sovellusrakenne perustuu tällä hetkellä vahvasti datan rakenteisuuteen sekä sen sisäisiin suhteisiin, vaatisi mallin käyttöönotto huomattavia muutoksia järjestelmän sovellus- ja tietorakenteisiin. Tämän lisäksi järjestelmän tietotarpeet eivät ole erityisen yksinkertaisia. Täten avain-arvo-malli ei sovellu TIMin käyttöön.

Wide-column-malli puolestaan ei vaikuta soveltuvan TIMin käyttöön sovelluksen luonteen takia. Koska wide-column-mallin tietokannat on suunniteltu

erityisesti hyödynnettäväksi sarjoittaisen, ”analyyttisen” datan sovelluksissa, voidaan heti huomata, ettei mallia ole suunniteltu TIMin käyttötarkoitukseen. Vaikka esimerkiksi datan versioinnin mahdollisuudet voisivat olla hyödyllisiä, tietomalli ei täten sovellu TIMin käyttötarkoitukseen.

Dokumenttimalli, nimensä mukaisesti, vaikuttaa soveltuvalta vaihtoehdolta TIMin dokumenttien hallinnan tietotarpeeseen. Vaikka monet dokumenttimallista nousevat hyödyt, kuten skeeman joustavuus, eivät luultavimmin tule tarpeeseen TIMin käyttötapauksessa, tarjoaa dokumenttimalli NoSQL-pohjaisen alustan TIMin dokumenttien tietovaraston toteutukselle. Suurimpana hyötynä dokumenttimallin käyttämisessä tässä tapauksessa on DBMS:ien tarjoama horisontaalinen skaalautuvuus, joka on mahdollisesti hyvin tärkeä ominaisuus järjestelmän tulevaisuutta ajatellen. Dokumenttimallin tietokannanhallintajärjestelmän valintaa tukee myös sen mahdollistama datan rakenteen mallintaminen, jonka tarve voidaan huomata TIMin nykyistä tietomallia tarkastellessa. Dokumenttimallin esittämä rajoite tarpeesta pystyä mallintamaan data dokumenttimuotoon ei luultavimmin vaikuta TIMin käyttötapaukseen, koska suuri osa dokumenttien tietomallista on jo nyt JSON-muodossa tai se pystytään helposti muuttamaan tarvittavaan muotoon lisäämällä kentille avaimia. Esitettyjen seikkojen perusteella dokumenttimalli toimii hyvänä vaihtoehtona valittavaksi malliksi TIMin dokumenttien hallintaan.

Graafimallia tarkastellessa voidaan huomata, että mallia ei olla suunniteltu TIMin käyttötarkoitukseen. Koska graafimallissa keskitytään etenkin tietorakenteiden välisten suhteiden mallinnukseen, eikä tämänkaltaista toimintaa tarvita TIMin dokumenttien hallinnassa, voidaan huomata, ettei malli sovellu TIMin tarpeisiin.

NewSQL-malli vaikuttaa alustavasti kaikista käsitellyistä malleista lupaavimmalta. Koska TIMin data sopii hyvin relaatiomalliin, mutta sen määrän takia hajautuksen tarve on olemassa, NewSQL:n mahdollistamat hyödyt vaikuttavat vastaavan TIMin tietotarpeisiin lähes täydellisesti. Koska TIMin dokumenttien hallinnan tietotarpeet eivät ole hyvin monimutkaisia, esimerkiksi monien taulujen yhdistelmälauseita, vaan tarkoin määriteltyjä sekä toisteisia vaihtelevilla syötteillä tehtäviä operaatioita, voidaan olettaa, että NewSQL-järjestelmät voisivat soveltua TIMin käyttöön. Toisaalta valinnassa voidaan huomata myös olennainen haaste, järjestelmän monimutkaisuus. NewSQL-järjestelmien monimutkaisuuden takia voidaan joutua tilanteeseen, jossa dokumenttien tietovarasto on liian monimutkainen ylläpitää sen tarpeisiin verrattuna. Esimerkiksi kuten aiemmin on huomattu, TIM ei välttämättä vaadi täydellisiä ACID-ominaisuuksia tietokannaltaan, jolloin voidaan ajautua tilanteeseen, jossa monimutkaisen NewSQL-järjestelmän konfigurointiin sekä ylläpitoon käytetty resurssimäärä ei ole järkevää. Myös jos TIMiin lisättäisiin jatkokehityksessä esimerkiksi monimutkaisia tietotarpeita vaativia toimintoja, voisi NewSQL-järjestelmän valinta aiheuttaa ongelmia. Kuitenkin NewSQL:n esittämien hyötyjen valossa malli vaikuttaa lupaavalta vaihtoehdolta TIMin tämänhetkisille vaatimuksille.

Näiden huomioiden perusteella voidaan arvioida, että esitetyistä tietokantamalleista TIMin käyttötapauksiin soveltuvimmat ovat relaatiomalli,

dokumenttimalli sekä NewSQL-malli. Jokainen malleista tarjoaa omat hyötynsä ja haasteensa, mikä voidaan nähdä erityisesti yhdistettäessä arvio CAP-teoreeman ominaisuuksien tärkeysarviointiin. Relaatiomalli tarjoaa tarvittavaa datan loogista rakennetta dokumenteille sekä eheyttä ja saatavuutta, mutta hajautus on ongelmallista. Dokumenttimalli toimii potentiaalisena NoSQL-vaihtoehtona datan nykyisen muodon sekä hajautuksen tarpeen vuoksi, mutta kaikkia mallin keskeisiä hyötyjä ei välttämättä pystytä hyödyntämään. NewSQL vaikuttaa lupaavimmalta vaihtoehdolta yhdistäen kahden edellisen hyödyt käyttötapauksen näkökulmasta, mutta monimutkaisuutensa takia saattaa johtaa mittaviin teknisiin haasteisiin. Tämän takia on olennaista, että soveltuvuusarviossa tarkastellaan jokaista näistä malleista.

6 TIETOKANTATUOTTEET

Tässä luvussa esitellään teorian kautta tutkielman testaukseen valitut tietokantatuotteet sekä niille suunnitellut rakenteet. Vaikka kirjallisuuskatsauksesta muodostettavan teoriapohjan avulla pystyttiin tekemään tietyn tasoista rajausta valittavista tuotteista, voidaan huomata, että itse tuotteiden valinta rajattujen paradigmojen sisällä perustuu lähinnä olemassa olevaan järjestelmään sekä siinä valmiiksi hyödynnettävään teknologiaan.

Valittuja tietokantatuotteita on kolme: PostgreSQL, MongoDB sekä CockroachDB, jotka jokainen kuuluvat aiemmassa luvussa järjestelmälle soveltuvaksi havaittuihin tietokantaparadigmoihin. Jokaisen tietokantatuotteen kohdalla annetaan lyhyt yleiskatsaus tuotteesta, jonka jälkeen kuvataan tuotteen rakenteen suunnitteluprosessi sekä suunnittelutyön lopputulos.

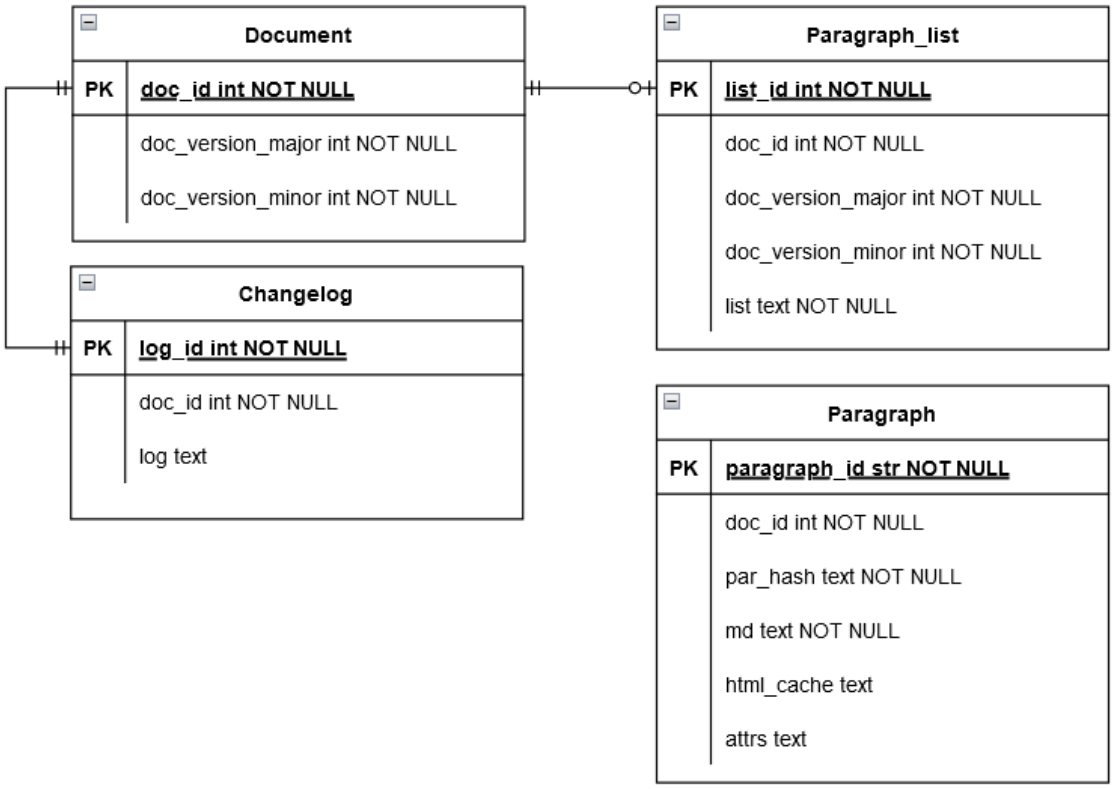
6.1 PostgreSQL

Relaatiotietokannanhallintajärjestelmä (RDBMS) -tuotteita on markkinoilla suuri määrä, sekä näiden välillä on tunnistettu tutkimuksissa vaihtelevasti eroja suorituskyvyissä (esim. Almeida ym., 2008). Koska tietokantojen suorituskykytestaustutkimuksissa on voitu huomata perustavanlaatuisia vaikeuksia (Taipalus, 2024), ei aiempiin tutkimustuloksiin nojaaminen tuotteen valinnassa ole välttämättä paras mahdollinen ratkaisu. TIMin laajempaa rakennetta tarkastellessa voitiin huomata, että järjestelmän osissa käytetään jo nyt RDBMS:ää, PostgreSQL:ää. Täten PostgreSQL osoittautuu luonnolliseksi valinnaksi myös dokumenttien varastoinnin relaatiotietokantaratkaisun testituotteeksi. Valintaa tukee muun muassa kehittäjien aiempi kokemus tuotteesta: Koska järjestelmässä on jo tietty tuote käytössä muualla, ei ole tarkoituksenmukaista pyrkiä integroimaan hyvin samankaltaisen toiminnallisuuden, mutta kuitenkin tuotetasolla eroavaa eri tuotetta järjestelmään. Tuotevalintaa tukee myös esimerkiksi avoin lähdekoodi (The PostgreSQL Global Development Group, 2024a), sekä erittäin avokätinen lisenssi (The PostgreSQL Global Development Group, 2024b).

6.1.1 PostgreSQL:n rakenne

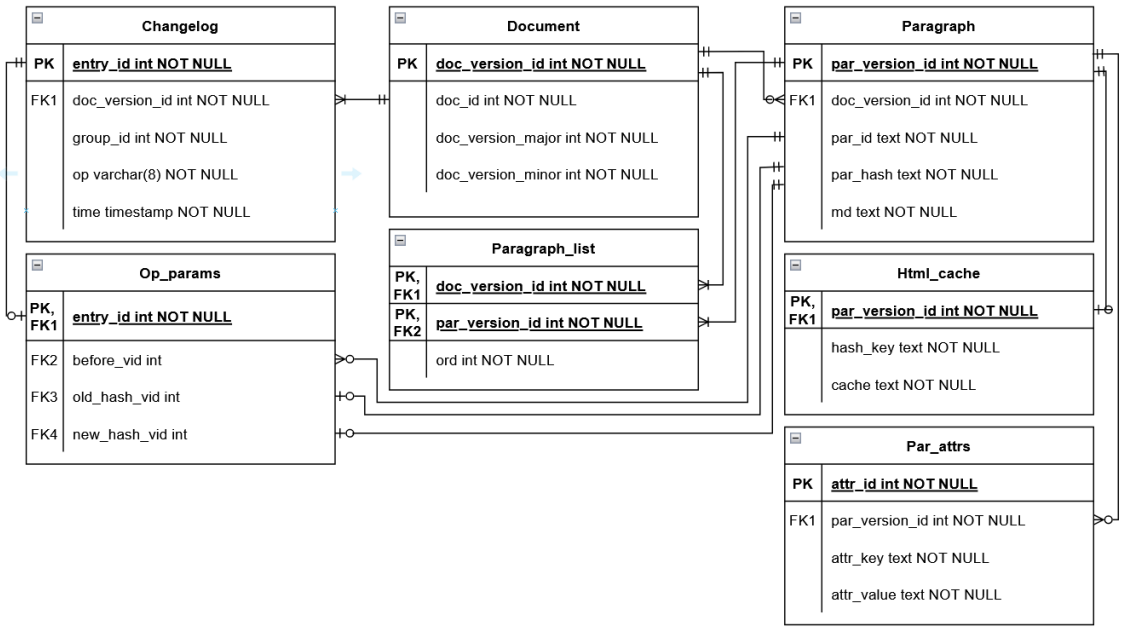
PostgreSQL:ää voidaan kuvailla perinteisenä RDBMS:nä, joten sen rakenteen suunnittelussa voidaan soveltaa relaatiotietokantoja koskevan laajan tutkimuspohjan kautta löydettyjä periaatteita. Relaatiotietokannan rakenteen suunnittelun lähtökohdaksi otetaan myös olemassa oleva tiedostopohjainen tietokanta, jotta nykyiset sovelluksen käyttötapauksien tietotarpeet pystyttäisiin täyttämään ilman massiivisia muutoksia sovellusrakenteeseen. Tällä lähestymistavalla vältetään samalla mahdolliset suurista sovelluksen ja tietorakenteen muutoksista johtuvat suorituskykyyn vaikuttavat tekijät, jotka saattaisivat vaikuttaa merkittävästi itse suorituskykytestaukseen.

TIMin tietorakennetta tarkastellessa voidaan tunnistaa neljä entiteettiä, jotka tietokannan tulee mallintaa sekä säilyttää. Nämä ovat 1) dokumentti, 2) dokumentin lohkolista, 3) dokumentin muutosloki sekä 4) lohko. Tunnistetuista entiteeteistä saadaan muodostettua tietokannan alustavat taulut. Tietokannan rakenteen kuvaamisen helpottamiseksi käytetään Chenin (1976) kehittämää ER (Entity-Relationship) -diagrammia. Mukaillen yleisiä notaatioita (Song ym., 1995), kuvattavia kardinaalisuuksia mallinnetaan ”variksenjalka”-notaatiolla sekä entiteettien ominaisuudet (sekä mm. alustavat pää- ja vierasavaimet) entiteettien sisällä. Kuvioissa 7 ja 8 visualisoidaan tämän lisäksi viiteavainten viittaamat kentät notaation avulla. Kuviossa 6 kuvataan hahmotellun relaatiotietokannan alustava rakenne pohjautuen TIMin dokumenttivaraston nykyiseen rakenteeseen.



KUVIO 6 Relaatiotietokannan rakenteen ensimmäinen hahmotelma

Tietokannan rakenteen alustava hahmotelma sisältää TIMin nykyisen dokumenttien tietorakennemallin sisältämät tiedot. Dokumentti sisältää tunnisteen sekä pää- että alaversion. Dokumentin muutosloki sisältää viitteen dokumenttiin sekä itse lokin tekstinä. Dokumentin lokilista sisältää viitteen dokumenttiin ja sen versioon sekä itse listan tekstinä. Lohkot sisältävät viitteen dokumenttiin, jossa ne on luotu sekä tarvittavat attribuutit, mutta eivät ole suoraan liitännäisiä dokumentteihin, koska lohkojen kuuluminen dokumenteille ratkaistaan tällä hetkellä sovelluskerroksessa lohkolistan avulla. Alustava hahmotelma, mikä pohjautuu tiukasti TIMin nykyiseen toteutukseen, ei ole järkevä relaatiotietokannan näkökulmasta: Vaikka se näennäisesti sisältää kaiken tarvittavan tiedon, on siinä monta suunnittelupohjaista ongelmaa. Esimerkiksi kaikki taulut eivät sisällä relaatioita toisiinsa, vaikka niiden välinen tieto on loogisesti yhteydessä toisiinsa. Lisäksi tauluissa on kenttiä, jotka tekstimuodossa sisältävät tietoa, jota voitaisiin tallentaa erillisille riveille omiin tauluihinsa sekä tieto on toistuvaa. Näitä ongelmia on pyritty korjaamaan hahmotelman toisessa versiossa, jonka visualisointi on nähtävissä kuviossa 7.



KUVIO 7 Relaatiotietokannan rakenteen toinen hahmotelma

Tietokannan toisen iteraation rakenne on muuttunut huomattavasti edellä mainittujen ongelmien takia. Tietokannassa säilytettäviä tietoja on hajautettu omiksi tauluikseen, jotta päästään relaatiomaiseen tiedon tallennukseen – käytännössä rivien arvoille on omat sarakkeensa, eikä yhdessä sarakkeessa säilytetä näennäisesti monia arvoja sisältäviä merkkijonoja. Eriytettyihin tauluihin kuuluvat seuraavat: Op_params (muutosloki-merkinnän operaation mahdolliset parametrit), Html_cache (lohkon html-”välimuisti”) sekä Par_attrs (mahdolliset lohkokohtaiset asetukset).

Myös olemassa olevien taulujen sisäiset rakenteet ovat muuttuneet huomattavasti. Dokumenttitauluun on lisätty sarake version id:lle, jotta uniikkien

versioiden toisistaan erottaminen ja näihin viittaaminen mahdollistuu, muodostaen taulun pääavaimen. Sekä muutosloki-, että lohkolista -taulut muutettiin rivimäiseen muotoon, jossa jokainen kirjaus tallennetaan omalle rivilleen, eikä koottuna merkkijonona. Muutoslokin merkintöjen mahdolliset parametrit erotettiin omaan tauluunsa, jotta näiden eheys pystytään varmentamaan lohkotauluun kohdistuvilla lohkon version id:n viiteavaimilla, joiden avulla halutut tiedot saadaan noudettua. Muutosloki-tauluun lisättiin myös surrogaattiavain (entry_id), jotta uniikit lokimerkinnät pystytään (helposti) erottamaan toisistaan. Lohkolista-tauluun lisättiin järjestyssarake (”ord”), jotta lohkojen järjestys listan sisällä pystytään säilyttämään. Tähän olisi käytännössä voitu käyttää esimerkiksi surrogaattiavainta kyselyiden lajittelussa, mutta koska automaattisesti kasvavat avaimet eivät välttämättä tietokantatuotteesta riippuen takaa, että lisäämisjärjestys pysyy vakiona, tulee lohkojen järjestystä ylläpitää ”listan sisällä” erillisellä sarakkeella. Lohkotauluun on lisätty pääavaimeksi surrogaattiavain ”par_version_id” lohkon versiolle, jotta lohkoille saadaan globaalisti uniikki tunniste, jota ”par_id” ei alkuperäisessä toteutuksessa ollut. Lohkon HTML-välimuisti on erotettu omaksi taulukseen, jotta välimuistin tietorakenne saadaan tallennettua omille riveilleen.

6.1.2 Normalisointi ja denormalisointi

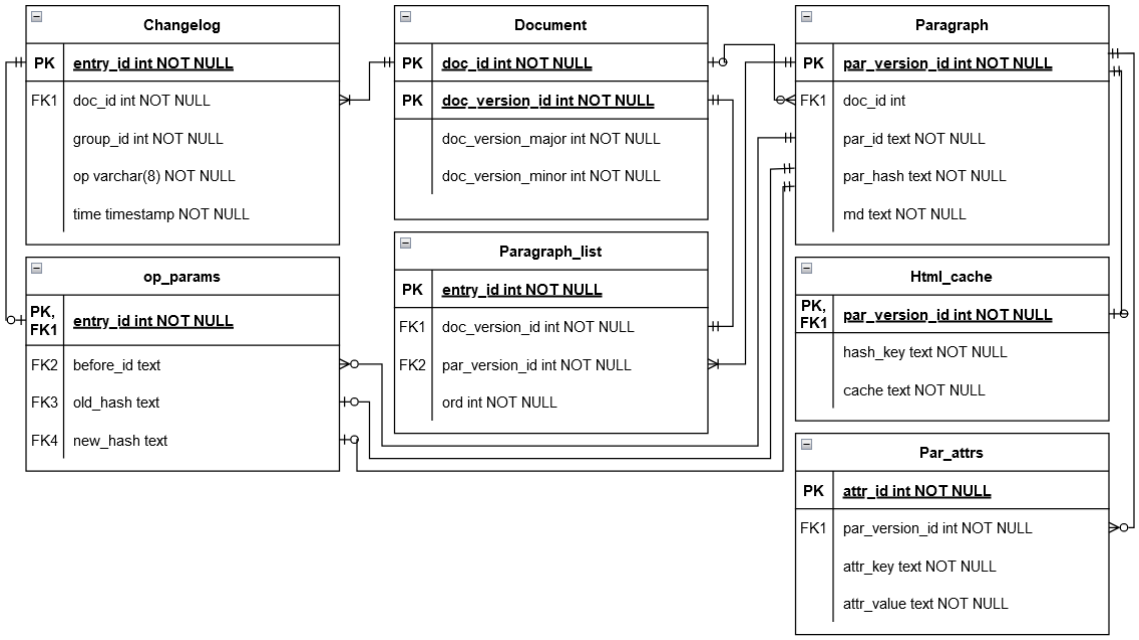
Relaatiotietokanta tulee myös normalisoida (Codd, 1970), jotta datan päällekkäisyys minimoidaan, mikä lisää tietokannassa olevan datan eheyttä ja tietokannan ylläpidettävyyttä (Albarak ym., 2020). Codd (1970) esitteli artikkelissaan kolme ensimmäistä normaalimuotoa sekä jatkoi kolmannen normaalimuodon tarkentamista Boyce-Codd-normaalimuotoon (BCNF) (Codd, 1974). Kent (1983) muodosti tämän pohjalta yhteen vetävän ohjeistuksen relaatiotietokantojen normalisoinnista, jota seuraavaksi käytetään normalisoinnin ohjeistuksena. Tietokannalle optimaalisen normalisaation tason määrittäminen ei ole yksinkertaista, mutta jos tietokantaa ei normalisoida vähintään kolmanteen tasoon (3NF), tietokannassa ilmenee vakavia ongelmia niin lisäys-, poisto- kuin päivitysoperaatioiden kanssa (Wang ym., 2010).

Hahmoteltu tietokanta vaikuttaisi täyttävän esitetyt vaatimukset kolmanteen normaalimuotoon asti. Alkuperäisten taulujen kompleksien arvojen hajautuksella omiksi tauluikseen saavutettiin rivien arvojen atomisuus, eli jokainen rivi sisältää saman määrän kenttiä (1NF) (Kent, 1983). Tietokannassa on vain yksi yhdistelmäavain taulussa ”Paragraph_list”, jonka ainoa ei-avain-kenttä riippuu selvästi molemmista avaimen osista. Tällöin jokaisen tietokannan yhdistelmäavaimia sisältävien taulujen ei-avain-kentät riippuvat koko yhdistelmäavaimesta, jolloin 2NF täyttyy (Kent, 1983). Kolmas normaalimuoto on kahta ensimmäistä monimutkaisempi: Jokaisen tietokannan ei-avain-kentän tulee olla riippuvainen vain avaimista, toisin sanoen ei-avain-kentät eivät voi riippua toisistaan (Kent, 1983). Hahmotellussa tietokannassa suurin osa kentistä täyttää tämän ehdon, kuten esimerkiksi muutosloki-taulussa merkinnän aikaleima ei anna informaatiota muusta kuin itse lokimerkinnästä (entry_id). Tulkinnanvaraisia kenttiä on kuitenkin muutamia: Lohkotaulun lohkotiiviste (par_hash) -kenttä käytännössä

riippuu laskennallisesti muista taulun kentistä. Koko kentän olemassaolo on suhteellisen kyseenalainen – käytännössä kentän sisältämän tiedon käyttötarkoitus täytetään jo muilla taulun kentillä, eikä kenttään viitata, kuin muutoslokissa, jonka toteutusta muuttamalla viittaus voitaisiin korvata lohkon version id:hen viittaavaksi. Lohkon tiivisteen laskeminen voitaisiin myös jättää kokonaan suorituksenaikaisen laskennan tehtäväksi, jolloin arvoa ei tallennettaisi kantaan ollenkaan. Koska nykyisen sovelluskerroksen toteutuksen muutokset ovat tämän tutkielman ulkopuolella, kenttä säilytetään tietokannassa tällä hetkellä. Vaikka tiivisteen laskeminen riippuu taulun muista kentistä, on se arvoltaan käytännössä uniikki, pois lukien täysin identtiset lohkot. Tällöin voidaan tulkita, että itse tiivisteen arvo onkin riippuvainen vain lohkon version id:stä, jolloin se täyttäisi kolmannen normaalimuodon vaatimukset. Sama ongelmallisuus toistuu lohkon asetusten taulussa: periaatteessa asetuksen avain ja arvo voivat olla toisistaan riippuvaisia. Koska mahdollisista asetuksista ei kuitenkaan ole olemassa dokumentaatiota on tätä vaikeaa näyttää todeksi. Yleisesti ottaen voidaan tulkita, että tietokanta on kokonaisuudessaan kolmannessa normaalimuodossa. Koska taulujen ei-avain-kentät riippuvat vain avainehdokkaista, on tietokanta myös BCNF:ssä (Vincent & Srinivasan, 1993).

Koska 3NF:ään asti normalisointi poistaa merkittävän osan vakavista lisäys-, poisto-, ja päivitysoperaatioiden anomalioista, tyydytään tässä tutkielmassa esitettyyn normalisointitasoon. Esitetty relaatiotietokannan rakenteen toinen hahmotelma toimisi näin ollen teoreettisesti tietokannan rakenteena. Alustavan käytännön testauksen kautta huomataan rakenteessa kuitenkin ongelma: lohkolista-taulun rivimäärät kasvavat äärimmäisen suuriksi. Viisi dokumenttia sisältävä testidata generoi lohkolista-tauluun yli 19 miljoonaa riviä, jolloin voidaan huomata, että tietokannan rakenne tuotantokäytössä, jossa on yli sata tuhatta dokumenttia, ei tule realistisesti toimimaan. Täten taulu pitää denormalisoida muotoon, jossa vältytään rivimäärien äärimmäiseltä kasvulta. Pienin mahdollinen rivimäärä taululle saadaan palaamalla alkuperäistä hahmotelmaa mukailevaan toteutukseen, jossa dokumenttien versioiden lohkolistat tallennetaan kokonaisuudessaan merkkijonoina omille riveilleen. Tällöin dokumentin lohkolista pitää parsia sovelluskerroksessa, jonka jälkeen dokumentille kuuluvat lohkot haetaan tietokannasta erikseen. Lähestymistavalla menetetään huomattavasti relaatiomallin hyötyjä, kuten viitteiden kautta taattu lohkolistojen eheys (esimerkiksi ei voida enää taata, että lohkot varmasti ovat olemassa pelkästään sillä, että niistä löytyy merkintä lohkolistasta). Koska ilman suuria dokumenttien tietomallin muutoksia, jotka ovat tämän tutkielman näkökulman ulkopuolella, ja myös koska muita varteenotettavia toteutustapoja toiminnallisuudelle ei olla onnistuttu tunnistamaan nykyisen tietomallin sisällä, on perusteltua denormalisoida taulu esitettyyn muotoon. Tämän lisäksi tuotaessa TIMin tuotannossa olevaa dataa tietokantaan havaittiin, että lohkotauluun suunniteltu viite dokumenttiin, jossa lohko on luotu, epäonnistuu, johtuen itse datasta. Iso osa lohkoista, joita tuotantojärjestelmässä säilytetään, on luotu dokumenteissa, joita järjestelmässä ei ole enää olemassa. Tämän perusteella Paragraph-taulun doc_version_id-kenttä muutettiin sallimaan NULL-arvoja, jotta tuotantodataa pystytään

silti käyttämään testauksessa. Tietokannan lopullinen rakenne on nähtävissä kuviossa 8. Tietokannan luontiin käytetyt SQL-lauseet on dokumentoitu liitteessä 1.



KUVIO 8 Relaatiotietokannan lopullinen rakenne

6.2 MongoDB

NoSQL-dokumenttitietokantatuotteen valinnan perusteluiksi on haastavampaa löytää konkreettisia perusteluita TIMiä tarkastellessa, sillä vastaavaa DBMS:ää ei ole käytössä muualla järjestelmässä. Tarkasteltaessa DBMS:ien suosiota voidaan huomata, että dokumenttitietokannoista ylivoimaisesti suosituin on MongoDB (solid IT gmbh, 2023d). Tämän suosion perusteella voidaan olettaa, että DBMS:lle löytyy myös laajin tuki niin käytännön kuin akatemian konteksteista. Tuotteen dokumentaatiota (MongoDB Inc., 2024b) tarkastellessa voidaan myös huomata, että tuote pystytään konfiguroimaan aiemmin tutkielmassa tunnistettujen, järjestelmän käyttötapausten vaatimien CAP-teoreeman ominaisuuksien mukaiseksi. MongoDB:n lisensointi on myös suhteellisen vapaata (MongoDB Inc., 2024c), jolloin se sopisi TIMin tuotantokäyttöön. Näiden tietojen perusteella MongoDB valitaan tutkielman dokumenttitietokannanhallintajärjestelmäksi.

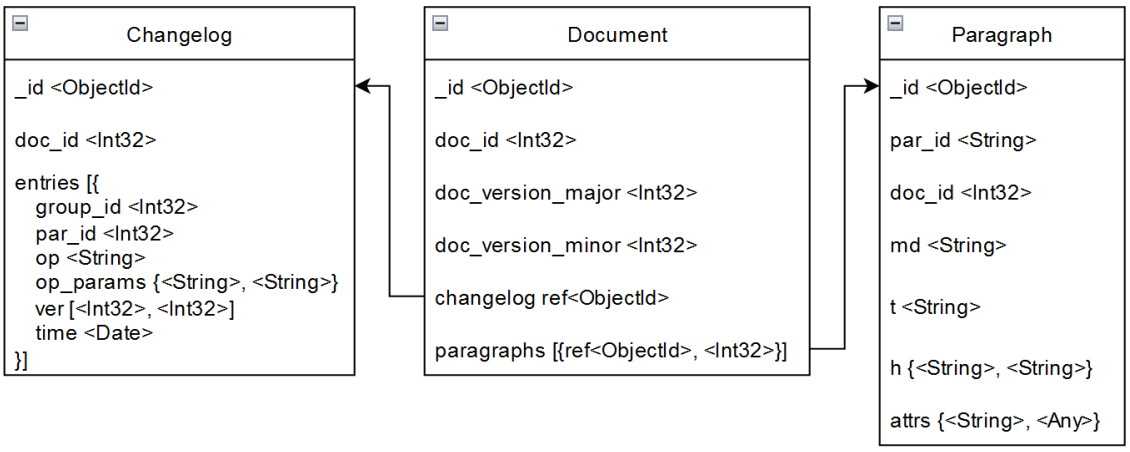
6.2.1 NoSQL-tietokantojen rakenteen suunnittelu

NoSQL-tietokantojen rakenteen suunnittelu, eteenkin taustatiedon keräämisen näkökulmasta, eroaa huomattavasti relaatiotietokantojen suunnittelusta muutamasta eri syystä. Ensimmäisenä NoSQL on käsitteenä suhteellisen uusi (verrattuna relaatiomalliin), jolloin tieteellistä taustatutkimusta ja täten tietopohjaa ei ole läheskään yhtä paljon akateemisessa kontekstissa. Toiseksi, koska NoSQL-

tietokantojen tietomallit eroavat hyvin paljon toisistaan on hyvin vaikeaa, ellei mahdotonta, muodostaa yleismaailmallista teoriapohjaa tietokantojen suunnittelujen tueksi, toisin kuin relaatiomallissa. Tämä pätee myös dokumenttitietokantoihin, joiden toteutukset voivat vaihdella suuresti tuotekohtaisesti. Täten on perusteltua, että suunnittelussa hyödynnetään tuotteen kehittäjien tarjoamaa dokumentaatiota sekä ohjeistusta tietokannan rakenteen mallintamisessa pelkästään tieteellisten lähteiden sijaan.

6.2.2 MongoDB:n rakenne

Myös dokumenttitietokannan rakenteen lähtökohdaksi otetaan olemassa oleva TIMin tietomalli. Koska TIMin dokumenttien varastoinnin nykyinen toteutus pohjautuu suurelta osin myös MongoDB:n käyttämään JSON-rakenteeseen, voidaan valmiita entiteettejä laajalti hyödyntää. Skeeman suunnitteluperiaatteena toimii dokumenttitietokannoille yleinen lähtökohta: Dataa pyritään säilyttämään mahdollisimman lähekkäin (MongoDB Inc., 2023d), eli käytännössä pyritään mahdollisimman pieneen taulumäärään. Seuraten MongoDB:n (2023d) antamia datan mallinnusohjeita on muodostettu dokumenttitietokannan rakenne, joka on visualisoitu kuviossa 9.



KUVIO 9 Dokumenttitietokannan rakenne

Tietokannan rakenteen visualisoinnissa on hyödynnetty yksinkertaista notaatiota: collectionit (tietokannan ”taulut”) visualisoidaan laatikoina, joiden sisällä on kenttiä, joiden (oletettu) tietotyyppi on merkitty ”< >”-sulkeiden sisälle. Collectionien välisillä nuolilla visualisoidaan, mihin kenttään <ObjectId>-viittaukset viittaavat.

Tietokannan rakenne mukailee paljolti TIMin nykyisen tietomallin rakennetta muun muassa kenttien nimeämisessä. Tietomalliin on tehty seuraavat muutokset: Collectioneihin on lisätty _id -kentät, joihin tallennetaan DBMS:n itse generoima kirjaus-id. Document-collectioniin on lohkolistan sijaan lisätty taulukko, joka sisältää kenttiä, joissa on viittaus dokumentin lohkoon sekä lohkon sijainti dokumentissa numeerisena arvona. Tämän lisäksi dokumentissa on viittaus muutosloki-collectioniin dokumenttia vastaavan muutosloki-tiedoston

hakemisen mahdollistamiseksi. Changelog-collection sisältää muutosloki-tiedostot, joissa lokimerkinnät on mallinnettu omiksi objekteikseen entries-taulukon sisälle. Lokimerkinnän doc_id -kenttä on varsinaisesti käytössä vain tiedon tuomisessa kantaan, jotta yhden dokumentin (doc_id:n) monista eri versioista (omat dokumenttinsa collectionissa) saadaan muodostettua oikeelliset viitteet muutoslokeihin. Kuitenkin on mahdollista, että muutoslokeja halutaan hakea pelkästään dokumentin id:n perusteella ilman, että dokumentin muita tietoja kuten versioita haetaan. Tällöin kentän säilyttäminen collectionissa on perusteltua.

Koko tietomalli pystyttäisiin käytännössä sisällyttämään yhteen collectioniin tavalla, jossa Document-collectioniin upotettaisiin sekä kokonaiset lohkot, että muutoslokit. Tässä lähestymistavassa esiintyisi kuitenkin muutama ongelma. Erityisesti Document-dokumenttien koko kasvaisi erittäin suureksi, mahdollisesti ylittäen MongoDB:n yksittäisten dokumenttien 16 MB:n kokorajoituksen. Lisäksi käyttötapauksia tarkastellessa voidaan huomata, että läheskään jokaisessa käyttötapauksessa ei ole tarkoituksenmukaista hakea kaikkea dokumentin tietoa kerralla sekä esimerkiksi muutoslokeja voidaan joutua hakemaan erikseen. Tämän perusteella rakenne on hajautettu kolmeen collectioniin: lohkot erillään kokorajoituksen ja mittavan datan toisteisuuden välttämisen vuoksi sekä muutoslokit käyttötapausten takia.

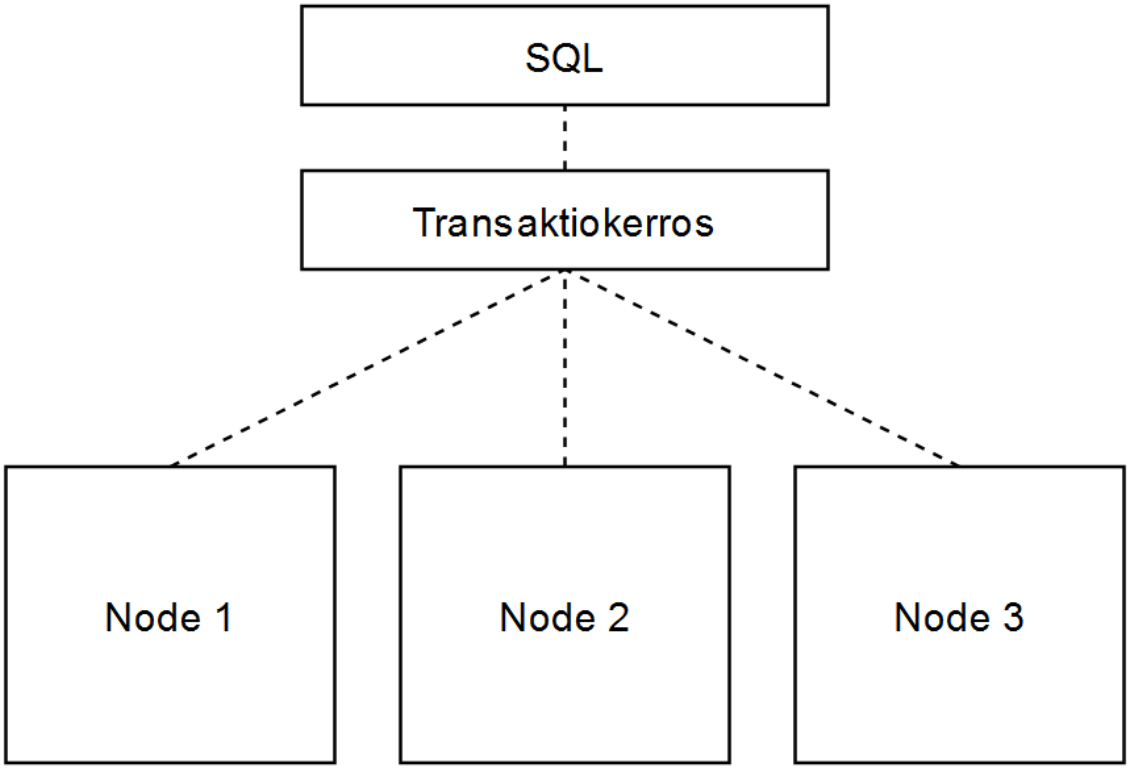
MongoDB ei tarvitse joustavan tietomallinsa takia collectionien luontilauseita kuten PostgreSQL:n tapauksessa, vaan tietomalli mukautuu siihen tuotavan datan perusteella. Dokumentoinnin vuoksi tietokannassa olevien objektien rakenne sekä collectioneille luodut indeksit on kuitenkin kirjattu vapaassa muodossa liitteessä 2.

6.3 CockroachDB

Tutkielmaan valitun NewSQL-tuotteen valinnan päällimmäisenä perusteena toimi aiemmin tehty PostgreSQL:n valinta RDBMS:ksi sekä tahtotila sisällyttää testaukseen ”uuden arkkitehtuurin” NewSQL-tuote. Koska testattava tietomalli on jo mallinnettu PostgreSQL:ään sekä tuote on käytössä muualla järjestelmässä, olisi optimaalista, että NewSQL-tuote kykenisi hyödyntämään näitä esimerkiksi tietokanta-ajurien yhteensopivuuden kautta. Tämän lisäksi tutkielman kannalta on mielekästä testata täysin uuden arkkitehtuurin omaavaa tuotetta eikä esimerkiksi jo olemassa olevia RDBMS-instansseja yhdistävää ”yhdyskäytävää”. Tämän perusteella tutkielman NewSQL-tuotteeksi valitaan CockroachDB, joka täyttää molemmat kriteerit vähintään oleellisimpien ominaisuuksien tasolla (Taft ym., 2020).

Lyhyesti ilmaistuna CockroachDB (CRDB) on hajautettu SQL-pohjainen tietokannanhallintajärjestelmä, joka lupaa 1) virheen sietokykyä sekä korkeaa saatavuutta, 2) maantieteellisesti hajautettua partitiointia sekä replikointia sekä 3) korkean suorituskyvyn transaktioita (Taft ym., 2020). CRDB:n arkkitehtuuri koostuu klusterista, joka muodostuu taas yksittäisistä nodeista (CRDB:n erillisistä instansseista). Arkkitehtuurin toiminnan lähtökohtana on viisikerroksinen

arkkitehtuuri, mikä koostuu seuraavista: SQL-. transaktio-, hajautus-, monistus- sekä varastointikerroksista (Taft ym., 2020). Arkkitehtuurin äärimmäisen yksinkertaistettu visualisointi on esitetty kuviossa 10. Arkkitehtuurin tätä yksityiskohtaisempi tarkastelu ei ole tälle tutkielmalle tarkoituksenmukaista: Tarkempaa itse tuotteen arviointia on tehty akateemisessa kontekstissa muun muassa Pina ym. (2023) toimesta.



KUVIO 10 CockroachDB:n arkkitehtuuri yksinkertaistettuna, mukaillen Pina ym., (2023)

Koska CockroachDB tukee suurinta osaa PostgreSQL:n ominaisuuksista (Taft ym., 2020), voidaan tietokannan rakenteena käyttää tässä tutkielmassa aiemmin mallinnettua relaatiotietokannan rakennetta suoraan. Saman rakenteen käyttämistä tukee muun muassa tuotteiden suorituskyvyn vertailun reiluus: Koska tietokantojen rakenteet ovat samat, näiden väliset suorituskykyjen erot pystytään tarkemmin rajaamaan tuotekohtaisiin ominaisuuksiin. CockroachDB:n taulut on luotu samoilla SQL-lauseilla kuin PostgreSQL:n taulut, jotka on dokumentoitu liitteessä 1.

7 TIETOKANNANHALLINTAJÄRJESTELMIEN SUORITUSKYKYTESTAUS

Tässä luvussa kuvataan valittujen tietokannanhallintajärjestelmien suorituskyvyn testaus. Aluksi määritellään suorituskykytestauksen tavoitteet sekä mittarit, jonka jälkeen kuvataan testiympäristöt niin palvelimen kuin tietokannanhallintajärjestelmien konfiguraation osalta. Tämän jälkeen kuvataan tietokantoihin viety data sekä viennin prosessi, minkä jälkeen esitellään ajettavat testit sekä testisuunnitelma. Testauksesta saadut tulokset esitellään ja tuloksia vertaillaan toisiinsa sekä niiden luotettavuutta arvioidaan, minkä jälkeen lopuksi esitetään lyhyt yhteenveto suorituskykytestauksesta kokonaisuudessaan.

7.1 Suorituskykytestauksen tavoitteet

Tässä tutkielmassa suoritettavan suorituskykytestauksen tavoitteena on löytää aiemmin valikoiduista tietokannanhallintajärjestelmistä tehokkain vaihtoehto TIMin dokumenttien varastointiin. Jotta saatavat tulokset ovat järjestelmän näkökulmasta mielekkäitä, suorituskykytestaus keskitetään järjestelmästä tunnistettuun pullonkaulaan eli dokumentin lohkojen hakemiseen. Jotta testien tulokset heijastavat järjestelmän nykyistä tilannetta mahdollisimman realistisesti, viedään tietokantoihin TIMin tuotantokäytössä olevan tiedostojärjestelmän data, johon on saatu pääsy TIMin kehitystiimiltä.

Tulee huomioida, että tässä tutkielmassa suoritettavan suorituskykytestauksen tavoitteena ei ole verrata valittujen tietokannanhallintajärjestelmien tehokkuutta yleistettävällä tasolla, vaan suorituskykytestaus suoritetaan hyvin spesifin tilanteen arvioimiseksi. Suorituskykytestaus on havaittu ongelmalliseksi monessa koostavassa tutkimuksessa (mm. Raasveldt et al., 2018 sekä Taipalus, 2024) etenkin, jos tavoitteena on ollut selvittää yleistettävällä tasolla ”tehokkain” järjestelmä. Vaikka tämä tutkielma keskittyy erittäin kapeaan näkökulmaan tietokannanhallintajärjestelmien suorituskykytestauksessa, nousee silti asetelmasta mahdollisia ongelmallisuuksia, joita käydään läpi myöhemmässä

pohdintaluvussa. Suoritettavan suorituskykytestauksen päällimmäisenä suunnitteluperiaatteena sekä täten yhtenä tavoitteista on testauksen reiluus, jota arvioidaan pohdintaluvussa.

7.2 Mittarit

Suorituskykytestauksen tavoitteiden pohjalta suorituskykytestaus on rajattu dokumentin lohkojen haun käyttötapaukseen, joka on havaittu erityiseksi pullonkaulaksi TIMin toiminnassa. Tämä heijastuu myös testaukseen valituissa mittareissa.

Pääasiallisena suorituskykytestauksen mittarina käytetään laskettua arvoa siitä, kuinka monta transaktiota eli tietyn dokumentin lohkojen hakua tuote kykenee käsittelemään sekunnissa (transaktiota/s, T/s). Mittarin tarkoituksena on havainnollistaa kuinka monta ”käyttäjäpyyntöä” tuote kykenee käsittelemään samanaikaisesti vaihtelevan kuorman alla. T/s:ää seurataan kahdella eri tavalla: kokonaiskeskiarvolla sekä ”T/s /min”-mittarilla, eli laskemalla T/s:n vaihtelua testin aikana minuutin välein. T/s /min-mittarin ensisijainen tarkoitus on minimoida kylmän kannan aiheuttamat vääristymät testauksen tuloksissa: Jos T/s /min pysyy suunnilleen tasaisena koko testauksen ajan, voidaan todeta, että tulokset eivät ole vääristyneet kannan ”lämpenemisen” johdosta. Tämän lisäksi arvoa käytetään datan visualisoinnissa kaaviomuotoon.

Toisena suorituskykytestauksen metriikkana käytetään dokumentin lohkojen haun vasteaikaa eli suoritetun transaktion latenssia. Hakutransaktioiden vasteajoista on vaikea yleistää johtopäätöksiä DBMS:n suorituskyvystä arvojen suuren keskihajonnan takia. Koska vasteajan mittaaminen on kuitenkin hyvin helppoa sekä se tarjoaa toisen jokseenkin vertailukelpoisen mittarin datasta, sisällytetään se tutkielman mittareihin. Tulee kuitenkin painottaa, että pääasiallisena vertailun kohteena oleva mittari on DBMS:n T/s.

7.3 Testiympäristöt

Kaikki suorituskykytestit ajetaan samalla KVM-virtuaalipalvelimella, jonka tiedot on kuvattu mahdollisimman tarkasti alla. Testit suoritetaan Docker-konteissa, joissa myös itse tietokannanhallintajärjestelmät sijaitsevat, jotta palvelimen Linux-ympäristön aiheuttamat mahdolliset sivuvaikutukset minimoidaan. Kontteihin asennetaan DBMS:n lisäksi testeissä tarvittavat Python-kirjastot, jotta datan tuonti sekä testien suorittaminen onnistuu. Konttien käyttämille resursseille ei ole asetettu rajoitteita, joten käytännössä palvelimen ja yksittäisten ajettavien konttien käyttämien resurssien kapasiteetit ovat samat.

Palvelin itsessään ei ole kuitenkaan dedikoitu pelkästään testikäyttöön johtuen taustalla suoritettavista mahdollisista muista testeistä riippumattomia prosesseja, jotka käyttävät palvelimen resursseja. Näiden prosessien määrä pyritään

minimoimaan testien suorittamisen ajaksi. Tämän lisäksi testien aikana tarkkaillaan, että prosessien määrä sekä resurssien käyttö pysyy vakiona koko testausprosessin ajan, jolloin prosessien käyttämät resurssit näkyvät jokaisen testin tuloksissa vaikutuksiltaan samana. Seuraavassa taulukossa (taulukko 3) on kuvattu palvelin, jolla testit suoritetaan.

TAULUKKO 3 Testiympäristö (KVM-virtuaalipalvelin)

Resurssi	Kapasiteetti	Kuvaus / versio
CPU	4 x64 corea, 2 GHz kellotaajuus / core	-
RAM	32 GB ECC, 2x 16 GB DIMM DDR4	-
Levy	500 GB SSD	-
Käyttöjärjestelmä	-	CentOS Linux 7 (Core)
Docker	-	20.10.3

7.3.1 PostgreSQL:n konfiguraatio

PostgreSQL:n konfigurointi on tehty yleisten standardiksi muodostuneiden nyrkkisääntöjen mukaan iteratiivisesti testaten arvojen vaikutusta optimointiin. Erityisesti PostgreSQL:n tapauksessa tämä konfigurointi on tärkeää reilujen tuloksien saavuttamiseksi: Yleisellä tasolla on huomattu, että PostgreSQL:n vakiokonfiguraatio ei hyödynnä nykyaikaisen laitteiston tarjoamia resursseja hyvin. Täten, jotta testauksessa saavutetaan edes reilu ”perustaso” tuotteiden välille, on PostgreSQL:lle allokoitava enemmän resursseja konfiguraation kautta. Koska yhden tuotteen liiallista optimointia muihin verrattuna on pyrittävä välttämään, keskityttiin konfiguraatiossa vain olennaisimpiin asetuksiin eikä DBMS:ää pyritty optimoimaan mahdollisimman loppuun asti. Seuraavassa taulukossa (taulukko 4) kuvataan ajoympäristö, tehdyt konfiguraatiot sekä niiden mahdolliset perustelut. Kattava kooste PostgreSQL:n ajoympäristön konfiguraatiosta (Dockerfile, ajokomennot, postgresql.conf) on dokumentoitu liitteessä 3. Tärkeimpänä huomioitavana on asetettu max_connections -arvo: Jotta tuotteiden välinen testaus on asetelmaltaan reilu, käytetään jokaisessa testissä samaa ”asiakas”määrää, joka on määritetty kaavalla ydinten määrä * 2 + tuotteen tarvitsemat ylläpitoyhteydet.

TAULUKKO 4 Kooste PostgreSQL:n ajoympäristöstä ja konfiguraatiosta

Resurssi	Kuvaus	Kommentit
DBMS:n versio	16.2	-
Python-ajuri	Psycopg2	-
max_connections	17	4 corea x2 + 5
shared_buffers	8 GB	~25 % kokonais-RAMista
effective_cache_size	23 GB	~70 % kokonais-RAMista
work_mem	128 MB	Iteratiivisesti testattu

7.3.2 MongoDB:n konfiguraatio

Koska MongoDB on moderni DBMS-tuote, oletettavasti se pystyy hyödyntämään modernin järjestelmän tarjoamia resursseja hyvällä tasolla jo vakiokonfiguraatiolla. Alustavan testauksen sekä dokumentaation perusteella tämä havaittiin todeksi: allokoitavien resurssien lisäystä ei suositeltu dokumentaatiossa eikä muuhun konfiguraation kautta tehtävään optimointiin löydetty ohjeistusta. Tämän lisäksi konfiguraatiosta ei löydetty asetuksia vastaamaan PostgreSQL:lle tehtyä ”perustason” konfiguraatiota, joten MongoDB:n konfiguraatiossa tyydyttiin vakioarvoihin. Ainoana muutettavana asetuksena oli samanaikaisia käyttäjiä rajoittava maxIncomingConnections, jonka pienimmäksi mahdolliseksi arvoksi ilman virheitä kahdellatoista clientillä havaittiin 31. Seuraavassa taulukossa (taulukko 5) annetaan kooste MongoDB:n ajoympäristöstä, joka on kokonaisuudessaan dokumentoitu liitteessä 4.

TAULUKKO 5 Kooste MongoDB:n ajoympäristöstä ja konfiguraatiosta

Resurssi	Kuvaus	Kommentit
DBMS:n versio	7.0.0-ubuntu2204	MongoDB community server
Python-ajuri	pymongo	-
maxIncomingConnections	31	Testattu iteratiivisesti

7.3.3 CockroachDB:n konfiguraatio

Hajautettuna NewSQL-järjestelmänä CockroachDB:n konfiguraatio eteenkin klusterina on hyvin monimutkainen. Koska tässä tutkielmassa ei kuitenkaan testata DBMS:iä hajautettuna, vaan yksittäisinä järjestelminä, CockroachDB:n konfiguraatio yksinkertaistuu huomattavasti.

Tässä tutkimuksessa suoritettavaan testaukseen käytetään CockroachDB:n ”single-node” -tilaa, eli tietokantaklusteri koostuu pelkästään yhdestä paikallisesta solmusta. Lähestymistapa ei suoranaisesti heijasta CockroachDB:n vahvuuksia sekä suunnitteluperiaatteita, jotka keskittyvät erityisesti tietokannan hajauttamiseen sekä tästä johtuvaan viansiedon vaatimukseen: yhden solmun klusterissa näitä vahvuuksia ei ole mahdollista hyödyntää. Koska tässä tutkielmassa ei käsitellä hajautuksen vaikutuksia, yhden solmun tilassa ajettava klusteri mahdollistaa testauksen suoritettavan tutkimuksen kontekstissa. Tämän lisäksi yhden solmun klusterilla tuetaan suorituskykytestauksen reiluutta: Koska muiden tuotteiden testaus suoritetaan yksittäisillä solmuilla, myös tässä tapauksessa on reilua suorittaa testaus yhdessä solmussa.

Konfiguraation määrittämisessä on seurattu CockroachDB:n kehittäjien tarjoamaa ”tuotanto-tarkastuslistaa” (Cockroach Labs, ei pvm.) sillä tarkkuudella, joka on arvioitu reiluksi muiden DBMS:ien konfiguraation taso huomioon ottaen. Konfiguraatio on näin ollen tehty resurssien allokoinnin näkökulmasta, tarkemmin ottaen DBMS:n välimuistin sekä SQL-muistin määrää lisäämällä. Seuraavassa taulukossa on koostettu CockroachDB:n ajoympäristö, joka on

dokumentoitu kokonaisuudessaan liitteessä 5. Tulee huomata, että CockroachDB ei tue solmukohtaisesti erillisiä konfiguraatiotiedostoja, vaan asetukset annetaan solmun käynnistyskomennon vivuissa.

TAULUKKO 6 Kooste CockroachDB:n ajoympäristöstä ja konfiguraatiosta

Resurssi	Kuvaus	Kommentit
DBMS:n versio	23.2.4	-
Python-ajuri	Psycopg2	-
--cache	.35	Välimuistin koko, 35 % RAMista = 11.2GB
--max-sql-memory	.35	SQL-kyselyille allokoitu välimuisti, 35 % RAM:ista = 11.2GB

7.4 Data

Testattaviin tietokannanhallintajärjestelmiin ajettiin sisään TIMin tuotantokäytössä olevan tiedostopohjaisen tietokannan tilannekuva (snapshot), jotta testin tulokset heijastavat realistisesti nykyistä tilannetta ja että testien tulokset eivät vääristy esimerkiksi liian pienen testidatan määrä takia. Seuraavassa taulukossa (taulukko 7) kuvataan tilastoja käytetystä tilannekuvasta, jonka jälkeen alaluvuissa kuvataan DBMS-kohtaisesti datan tuominen sekä sen jakautuminen tietokantoihin. Kappalemääristä on poistettu tiedostot sekä kansiot, joita ei olla hyödynnetty testauksessa, mm. ”current”-linkkitiedostot, joita ei olla käsitelty tässä tutkielmassa.

TAULUKKO 7 Testauksessa käytetty tuotantotietokannan tilannekuva

Kuvaus	Koko	Kappalemäärä
Koko tiedostojärjestelmä (./tim_files)	55 GB	
Dokumentit (./tim_files/docs)	24 GB	
Dokumenttien pääversiot		103355
Dokumenttien alaversiot		3505111
Muutosloki-tiedostot		91608
Lohkot (./tim_files/pars)	31 GB	
Yksittäiset lohkot		1930073
Lohkojen versiot		3176338

7.4.1 PostgreSQL:n data

Data ajettiin suunniteltuun PostgreSQL-tietokantaan hyödyntämällä Python-skriptiä, joka on dokumentoitu liitteessä 6. Skriptin luonnin tukena käytettiin ChatGPT:tä sekä internetistä löytyviä resursseja, minkä lisäksi siihen etsittiin iteratiivisesti parannuksia erityisesti optimoinnin näkökulmasta. Skriptin tuoman datan oikeellisuus tietokannassa on tarkastettu käsin.

Mahdollista jatkotutkimusta tekevien on huomioitava, että skripti on erittäin epätehokas (tuotantokannan tuonti tietokantaan kesti ~30 tuntia), joten skriptin suoraa käyttöä ei suositella vaan skriptin optimoinnin tutkimista suositellaan vahvasti. Tämän lisäksi voidaan suositella perehtymistä mm. indeksien vaikutukseen datan tuomisessa: osa indekseistä perustettiin tuomisen aikana sekä osa vasta tuonnin loputtua. Seuraavassa taulukossa (taulukko 8) kuvataan tuonnin jälkeinen data PostgreSQL:ssä.

TAULUKKO 8 Datan jakautuminen PostgreSQL:ssä lajiteltuna fyysisen koon mukaan laskevasti

Taulu	Kokonaiskoko	Indeksien osuus	Rivimäärä
Paragraph_list	15 GB	75 MB	3505111
Html_cache	11 GB	44 MB	2052660
Paragraph	2043 MB	173 MB	3176337
Changelog	276 MB	75 MB	3494369
Par_attrs	250 MB	108 MB	2734954
Document	248 MB	100 MB	3505111
Op_params	105 MB	36 MB	1690052

7.4.2 MongoDB:n data

Data ajettiin suunniteltuun MongoDB-tietokantaan hyödyntämällä Python-skriptiä, joka on dokumentoitu liitteessä 7. Kuten aiemmin, skriptin luonnin tukena käytettiin ChatGPT:tä sekä internetistä löytyviä resursseja iteratiivisesti parannellen skriptiä, kunnes tietokantaan saatiin tuotua haluttu data oikeellisesti. Skriptin tuoman datan oikeellisuus on myös tässä tapauksessa tarkastettu käsin.

Myös tässä tapauksessa voidaan todeta, että skripti havaittiin erittäin epätehokkaaksi (tuonnin kesto ~30 h, josta suurin osa painottui dokumenttien tuontiin), joten suoraa käyttöä ei suositella, vaan optimoinnin tutkimista kehotetaan. Erityisen haastavaksi sekä suorituskykyä vaativaksi havaittiin Document-collectionin dokumenttien ”paragraphs”-taulukkoon sijoitettavien objektien ”pos”-arvon eli lohkoviitteen järjestysnumeron laskeminen.

Seuraavassa taulukossa (taulukko 9) on dokumentoitu skriptin ajosta luodut MongoDB:n collectionit, näiden fyysiset kompressoidut koot, collectionien indeksien koko sekä dokumenttien määrä collectioneissa. Tietokannan collectionien dokumenttien määrät vastaavat relaatiotietokantojen vastaavien taulujen (Document ja Paragraph) rivimääriä. Changelog-collectionin huomattavasti dokumenttien pienempi määrä johtuu siitä, että muutoslokit talletettiin kokonaisina dokumentteina tietokantaan, eikä yksittäisinä riveinä.

TAULUKKO 9 Datan jakautuminen MongoDB:ssä lajiteltuna kompressoidun koon mukaan laskevasti

Collection	Kompressoitu koko	Indeksien osuus	Dokumenttien määrä
Paragraph	10.04 GB	308.48 MB	3176337
Document	4.35 GB	149.17 MB	3505111
Changelog	130.29 MB	3.14 MB	91608

7.4.3 CockroachDB:n data

Kuten tietokannan rakenne, myös datan ajaminen CockroachDB:hen on tehty samalla tavalla kuin PostgreSQL:ään. Täten myös käytetty skripti on sama (liite 6), pois lukien tietokantaan yhdistämiseen käytetyt tunnistetiedot. Skriptin tuoman datan oikeellisuus tietokannassa on tarkastettu myös tässä tapauksessa varmuuden vuoksi käsin.

Tutkimusta tehtäessä oletettiin, että sama skripti ajaisi täysin saman datamäärän tietokantaan. Datan oikeellisuutta tarkastellessa huomattiin kuitenkin ongelmallisuus: Par_attrs -taulun rivimäärästä puuttuu noin 230000 riviä verrattuna PostgreSQL:n vastaavan taulun rivimäärään. Skriptin toiminnan eroavaisuutta tuotteiden välillä ei pystytty tunnistamaan ajon jälkeen, mutta syyksi arvioitiin Pythonin PostgreSQL-ajurin (psycopg2) ja CockroachDB:n STRING-tietotyypin (CockroachDB:n alias PostgreSQL:n TEXT-tietotyypille) väliset dokumentoimattomat eroavaisuudet. Koska taulujen rivimäärien ero on kuitenkin alle kymmenen prosenttia (~8.5 %), fyysisen kompressoimattoman datamäärän fyysinen kokoero pienehkö (21,1 MB) ja tietokantojen rivimäärien ero kokonaisuudessaan ~1 %, voidaan olettaa, että erolla ei ole suurta vaikutusta tietokantojen suorituskykyjen välisessä tarkastelussa. Johtuen tutkimuksen suorittamisen rajallisista resursseista sekä skriptin muokkauksen ja ajon (+50 h) vaatimasta työmäärästä, kuvailtu ero hyväksytään. Rivimäärien eroavaisuus on otettu huomioon sekä tuloksia, että tutkimuksen luotettavuutta arvioitaessa.

CockroachDB ei tarjoa suoraa helposti noudettavaa statistiikkaa taulujen datan koosta levyllä johtuen sen sisäisestä toteutuksesta sekä kompressoinnista. Seuraavaan taulukkoon (taulukko 10) on dokumentoitu taulujen rivimäärien lisäksi Cockroach Consolen tarjoama arvio datan koosta kompressoituna sekä kompressoimattomana ladattuna ”live datana”, jonka määrä jokaisessa taulussa on 100 % testitilanteessa.

TAULUKKO 10 Datan jakautuminen CockroachDB:ssä lajiteltuna arvioidun kompressoidun koon mukaan laskevasti

Taulu	**Kompressoitu koko**	**”Live data” (100 %) koko**	**Rivimäärä**
Html_cache	8.3 GB	18.2 GB	2052660
Paragraph_list	7.2 GB	14.0 GB	3505111
Paragraph	958 MB	2.8 GB	3176337
Document	185.6 MB	364.2 MB	3505111
Par_attrs	129 MB	228.9 MB	2502742
Changelog	102.7 MB	199.1 MB	3494369
Op_params	54.3 MB	78.9 MB	1690052

7.5 Testit

Tietokantatuotteiden suorituskykyä testataan kahdella testillä (T1 ja T2), jotka pohjautuvat dokumentin lohkojen haun käyttötapaukseen. Testitapaukset on kuvattu lyhyessä muodossa seuraavassa taulukossa (taulukko 11), jonka jälkeen valittuja testejä tarkastellaan tarkemmin.

TAULUKKO 11 Suoritettavat suorituskykytestit

Testin tunniste	Kuvaus	Mallinnettava käyttötapaus
T1	Haetaan satunnaisen dokumentin tuoreimman version lohkot.	TIM:in yleinen käyttö – käyttäjät avaavat satunnaisia dokumentteja.
T2	Haetaan tietyn dokumentin tuoreimman version lohkot.	Luentotilanne, jossa suuri määrä käyttäjiä avaa saman dokumentin kerralla.

T1 mittaa esitetyillä mittareilla satunnaisen dokumentin tuoreimman version kaikkien lohkojen noutoa simuloiden reaalimaailman yleistä käyttöä, jossa tietokannasta haetaan vaihtelevia dokumentteja. Satunnainen dokumentti valitaan etukäteen muodostetusta listasta kaikista tietokannassa olevista dokumenteista ennen jokaista transaktiota. Järjestelmästä ei ole saatavilla esimerkiksi lokitietoja, jolla tiettyjä dokumentteja pystyttäisiin painottamaan realistisemman yleisen käytön mallintamiseksi, joten testissä tyydytään Pythonin tarjoamaan satunnaisgeneraattoriin. Testin kestoa määrittäessä on otettu huomioon epätodennäköinen mutta mahdollinen tilanne, jossa satunnaisgeneraattori valitsisi huomattavan osan ajasta tiettyjä joko erittäin suuria tai erittäin pieniä dokumentteja. Koska dokumentin koon on havaittu vaikuttavan lohkojen haun vaativuuteen, testissä ei voida taata, että jokaiselle tuotteelle suoritettu testi vastaa täysin toistaan. Voidaan kuitenkin olettaa, että tarpeeksi pitkällä ajanjaksolla sekä tästä johtuvalla suurella transaktioiden määrällä suoritetulla testillä mittareiden keskiarvot tasoittuvat realistisiksi.

T2 eroaa T1:sestä siten, että testissä haetaan pelkästään tietyn valitun ID:n omaavan dokumentin lohkoja. Tällä lähestymistavalla havainnollisestaan esimerkiksi luentotilannetta, jossa suuri määrä käyttäjiä pyrkii avaamaan saman dokumentin samanaikaisesti. Valitun dokumentin doc_id on 1, joka on havaittu tietokannan suurimmaksi yksittäiseksi dokumentiksi. Täten T2:sella simuloidaan tietokantaan kohdistuvaa mahdollisimman raskasta hakujen ”piikkiä”. Samalla testi tarjoaa täysin tasavertaisen alustan kaikkien testattavien tuotteiden vertailuun, jossa esimerkiksi satunnaisuudella ei ole vaikutusta. Koska testissä haetaan täysin samaa tietoa useasti, voidaan olettaa, että tuotteet hyödyntävät välimuistiin tallennettua (cachetettua) tietoa. Täten on tärkeää, että ennen testiä tietokantaa on lämmitetty tarpeeksi, jotta välimuistit täyttyvät tasavertaisesti, mikä on huomioitu testisuunnitelmassa.

Kaikki testit on suoritettu kaikille tietokantatuotteille samalla Python-skriptillä (tuotekohtaiset ajurit sekä kyselylauseet pois lukien), jotta tulosten reiluutta

pystytään ylläpitämään. Itse tuotettuun testiskriptiin päädyttiin sen perusteella, että tutkielmaa kirjoittaessa ei onnistuttu löytämään yleismaailmallista testausohjelmistoa, joka tukisi vahvistetusti reilulla vertailulla kaikkia testattavia tuotteita. Tulee huomata, että testausskripti ei ole läheskään yhtä kehittynyt kuin kolmansien osapuolten ”oikeat” benchmark-sovellukset, vaan skripti on rakennettu hyvin tietyn tilanteen testaukseen, eikä se sisällä yleisesti suorituskykytestauksessa tehtäviä operaatioita kuten kirjoituksia. Testausta tehdessä myös tunnistetaan mahdolliset testiskriptin puutteet sekä ongelmallisuudet, jotka johtuvat mm. kehittäjän tietotaidon tasosta käytetyssä ohjelmointikielessä. Koska jokaisessa testissä käytetään kuitenkin samaa skriptiä jokaiselle tuotteelle (pl. aiemmin mainitut tuotekohtaiset muutokset), voidaan olettaa, että testit ovat lähtökohtaisesti reilut: Esimerkiksi itse testausskriptin puutteellisesta optimoinnista johtuvat vääristymät tuloksissa toistuvat kaikkien tuotteiden testien tuloksissa.

Testauksessa käytetty skripti on kokonaisuudessaan dokumentoitu liitteessä 8. Liitteessä esitettävän skriptin koodissa on kommentoitu tuotekohtaiset asetukset sekä kysely, jotta tutkielmassa vältytään kolmelta hyvin samanlaiselta liitetiedostolta. Täten skriptiä lukevalta tai uudelleenkäyttävältä taholta oletetaan kyvykkyyttä ymmärtää käytettyä ohjelmointikieltä.

7.6 Testisuunnitelma

Testisuunnitelmaa kehittäessä on pyritty välttämään monimutkaisuutta, joka saattaisi johtaa tulosten vääristymiseen, kuitenkin ylläpitäen testauksen oikeellisuutta sekä tuotteiden välisen vertailun reiluutta. Seuraavaksi kuvataan yksittäiselle tuotteelle suoritettavan testauksen testisuunnitelma, joka on toistettu kaikille testaukseen valituille tuotteille samalla tavalla.

Kun testauksessa käytetty data on tuotu tietokantaan sekä tarkastettu oikeellisuudeltaan, käytetty kontti sammutetaan ja poistetaan sekä luodaan uudestaan ilman tiedostojärjestelmän Docker-volumea. Testiskripti konfiguroidaan tuotteen vaatimusten mukaisesti: asetetaan mm. testin transaktiossa käytetty kysely, DBMS:n ajuri sekä satunnainen (T1) tai kiinteä (T2) doc_id. Yksittäisen testiskriptin ajon kestoksi on asetettu 15 min, jotta satunnaisuuden aiheuttamat häiriöt tuloksissa minimoidaan. Tämän lisäksi 15 min havaittiin kestoksi, jossa testin T/s-arvon hajonta tasaantuu, eli tietokanta lämpenee tarpeeksi testiä varten. Testiskripti ajetaan kaksi kertaa peräkkäin testissä: Ensimmäisen ajon tarkoituksena on lämmittää tietokanta, joten ajon tulokset hylätään. Toisen ajon eli varsinaisen testin tulokset tulostetaan tekstitiedostoon, jonka pohjalta muodostetaan seuraavassa alaluvussa esitettävä raportointi. Testin jälkeen kontti sammutetaan, poistetaan sekä luodaan uudestaan, jonka jälkeen suoritetaan toinen tuotteen testeistä samalla tavalla testin vaatimilla konfiguraatioilla.

7.7 Tulokset

Seuraavaksi esitetään tehtyjen suorituskykytestien tulokset testikohtaisesti yksityiskohtaisesti taulukkomuodossa sekä visualisoidussa muodossa. Tämän jälkeen esitetään yhteen vetävä tuloksien vertailu sekä arviointi.

Mitatusta datasta esitetään seuraavat arvot: T/s keskiarvo, T/s keskihajonta, latenssin keskiarvo millisekunteina ja latenssin keskihajonta millisekunteina, joista jokainen on tarpeen mukaan pyöristetty kolmen desimaalin tarkkuudelle. Tämän lisäksi testeistä mitatut ja lasketut T/s /min -arvot esitetään kuvaajissa minuutin intervallein (T/s, aika) -akseleilla datan visualisoimiseksi.

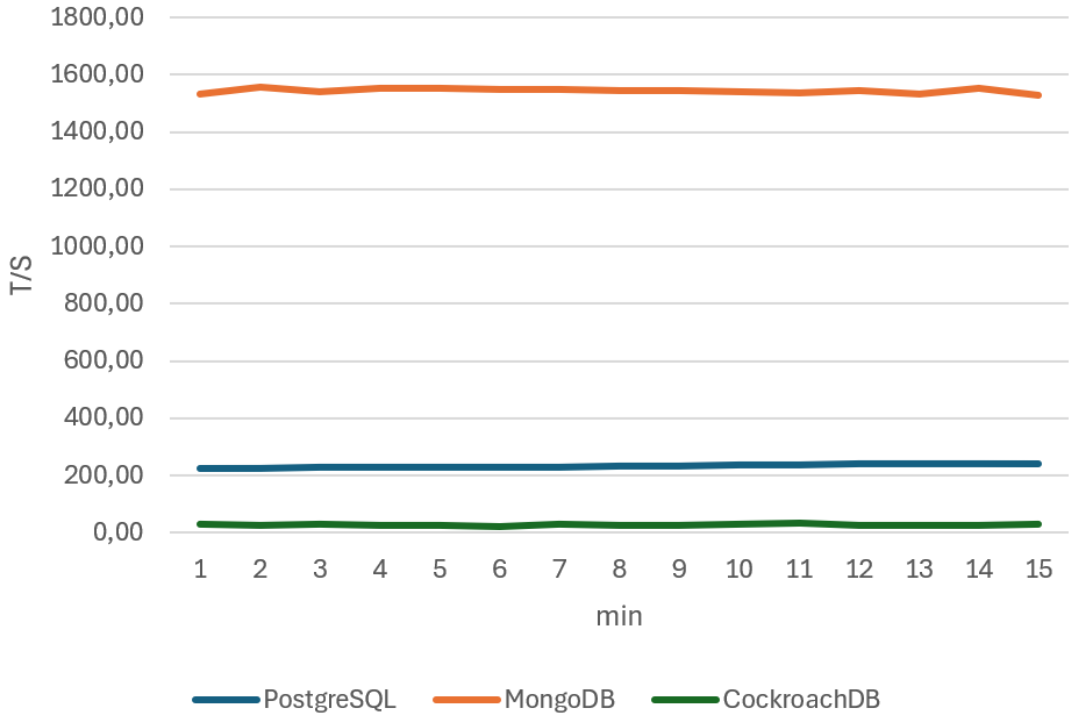
Seuraavissa taulukoissa (taulukko 12 & taulukko 13) esitetään ensimmäisen (T1) ja toisen (T2) suorituskykytestin tulokset. Näiden jälkeisissä kuvioissa (kuvio 11 & kuvio 12) esitetään tulokset visuaalisessa muodossa, (T/s, kuluneet minuutit) -kuvaajassa.

TAULUKKO 12 T1 tulokset

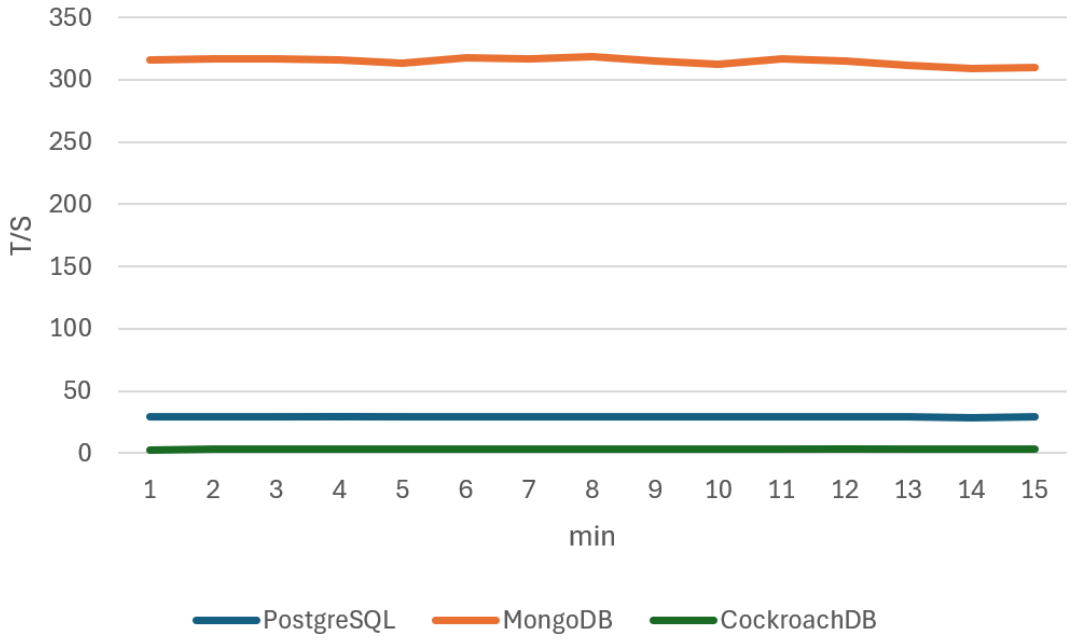
DBMS	**T/s keskiarvo**	**T/s keskihajonta**	**Latenssi keskiarvo**	**Latenssi keskihajonta**
PostgreSQL	232,938	6,116	50,81 ms	170,12 ms
MongoDB	1544,477	8,663	6,25 ms	6,79 ms
CockroachDB	27,15	2,701	440,93 ms	1165,66 ms

TAULUKKO 13 T2 tulokset

DBMS	**T/s keskiarvo**	**T/s keskihajonta**	**Latenssi keskiarvo**	**Latenssi keskihajonta**
PostgreSQL	29,375	0,144	406,49 ms	109,16 ms
MongoDB	315,044	2,895	34,02 ms	16,96 ms
CockroachDB	3,235	0,097	3703,07 ms	395,85 ms



KUVIO 11 T1 tulokset visualisoituna



KUVIO 12 T2 tulokset visualisoituna

7.8 Tulosten vertailu

Testien tulokset ovat yksiselitteiset: Molemmissa testeissä MongoDB on ylivoimaisesti tehokkain esitetyssä asetelmassa testattavista tietokannanhallintajärjestelmistä. Toiseksi tehokkain on myös selvästi PostgreSQL, viimeisen ollessa CockroachDB.

MongoDB:n tehokkuus näkyy molemmissa mittareissa molemmissa testeissä. Ensimmäisessä testissä MongoDB:n tarjoama maksimaalinen T/s on noin seitsemän kertaa isompi kuin PostgreSQL:n sekä noin kuusikymmentä kertaa isompi kuin CockroachDB:n. Toisessa testissä ero kasvaa: Verrattuna PostgreSQL:ään, T/s on noin kymmenkertainen, CockroachDB:hen noin satakertainen. Arvojen keskihajonnat ovat hyvin pieniä, noin prosentin, lukuun ottamatta T1:sen CockroachDB:n hajontaa, joka on noin kymmenen prosenttia. Tämän yksittäisen tapauksen suurehko keskihajonta muihin verrattuna on voinut johtua esimerkiksi siitä, että satunnaisgeneraattori on arponut DBMS:lle haettavaksi dataltaan isoja dokumentteja merkittävän määrän peräkkäin. Vaikka isompi keskihajonta otettaisiin huomioon, tulokset ovat silti selkeät tehokkuusjärjestyksen näkökulmasta. Tulosten keskihajonnan pieni koko kertoo myös siitä, että testisuunnitelman mukaan suoritettu tietokantojen lämmittäminen on onnistunut. Jos tietokannat eivät olisi lämmenneet riittävästi, T/s olisi testien aikana kasvanut selvästi nousujohteisesti lisäten keskihajontaa. Täten sekä keskihajontaa että tulosten visualisointia seuraten voidaan todeta, että testisuunnitelma on onnistunut. Tämän perusteella voidaan todeta, että tulokset ovat lähtökohtaisesti luotettavat. Testauksessa tunnistettuja mahdollisesti tuloksia vääristäviä rajoitteita tarkastellaan tarkemmin pohdintaluvussa.

Latenssin keskiarvoa tarkastellessa MongoDB on molemmissa testeissä noin 90 % ja 99 % nopeampi verrattuna PostgreSQL:ään ja CockroachDB:hen. Koska latenssien keskihajonnat ovat kuitenkin hyvin suuret tuotekohtaisesti, ei tätä arvoa voida yksinään käyttää suorituskyvyn tarkasteluun. Yleisesti ottaen voidaan kuitenkin todeta, että MongoDB:n latenssit ovat merkittävästi pienemmät molemmissa testeissä muihin DBMS:iin verrattuna.

7.9 Suorituskykytestauksen yhteenveto

Tässä luvussa esiteltiin sekä dokumentoitiin tutkielmassa tehty suorituskykytestaus. Suorituskykytestaukselle asetettiin tavoitteet sekä mittarit, minkä jälkeen kuvattiin testien ajoympäristö, konfiguraatiot ja DBMS:iin ajettu data kattavasti. Tämän jälkeen esitettiin testisuunnitelma, jonka toteutuksen tulokset esitettiin niin numeerisessa, kuin visuaalisessa muodossa.

Havaitut tulokset vastaavat yksiselitteisesti tutkielmassa esitettyyn toiseen tutkimuskysymykseen: ”Mikä valikoiduista tietokannanhallintajärjestelmistä on tehokkain vaihtoehto järjestelmästä havaitun ongelman ratkaisemiseen?”, johon

vastaus on MongoDB. Täten myös suorituskykytestaukselle asetettu tavoite eli esitettyyn tilanteeseen tehokkaimman vaihtoehdon määrittäminen on täytetty.

Seuraavassa luvussa esitetään kattavaa pohdintaa tutkielmasta kokonaisuutena, mukaan lukien saatujen tulosten arviointia sekä pohdintaa siitä, mitkä tekijät ovat mahdollisesti vaikuttaneet niihin.

8 POHDINTA

Tässä luvussa esitetään kattavaa pohdintaa ja arviointia tutkielmassa suoritetusta tutkimuksesta sekä sen tuloksista. Luvussa käsitellään aluksi esitetyt tutkimuskysymykset ja joko pyritään muodostamaan niihin vastaus tai arvioidaan löydettyä vastausta. Seuraavaksi esitellään sekä arvioidaan tutkielman niin tieteellisiä kuin käytännön kontribuutioita. Tämän jälkeen käydään läpi tutkielmasta tunnistettuja rajoitteita tutkielman reliabiliteetille ja validiteetille sekä arvioidaan rajoitteiden mahdollisia vaikutuksia tuloksiin. Lopuksi esitetään tutkielmasta nousseita mahdollisia jatkotutkimuksen aiheita.

8.1 Ensimmäinen tutkimuskysymys

Tutkielmassa esitetty ensimmäinen tutkimuskysymys oli seuraava: ”Miten pystytään rajaamaan sekä valitsemaan tarkasteltavalle järjestelmän osalle parhaiten soveltuvat tietokannanhallintajärjestelmät?”.

Esitettyyn tutkimuskysymykseen etsittiin vastausta teorian kautta kahdesta eri näkökulmasta: yleisesti hyväksytyn sekä käytetyn CAP-teoreeman ja käytännönläheisemmän, mutta silti vahvasti akateemiseen tutkimukseen perustuvan paradigmapohjaisen vertailun näkökulmista. Tutkielmassa onnistuttiin näiden näkökulmien avulla suorittamaan jokseenkin onnistunut DBMS-tuotteiden rajaus jatkotestaukseen, mutta suoraa ja kaikenkattavaa vastausta tutkimuskysymykseen ei onnistuttu muodostamaan.

Esitetyistä näkökulmista erityisesti paradigmapohjainen vertailu havaittiin hyödylliseksi tuotteiden rajaukseen. Vertailun perusteella pystyttiin selvästi tunnistamaan kolme paradigmaa, joiden soveltuvuus tietomallille vaikutti teorian pohjalta parhaalta. DBMS-tuotteiden suuren määrän takia tämän kaltainen, eri paradigmojen vahvuuksia sekä heikkouksia ja tietomallia (ja mallin tarpeita) peilaava arviointi on oleellista, jotta tuotteiden määrää pystytään perustellusti rajaamaan käytettävän tuotteen valinnan helpottamiseksi.

CAP-teoreemaan perustuva rajaus havaittiin puolestaan haastavaksi. Koska tutkittavasta järjestelmästä ei tunnistettu erityisen tiukkoja vaatimuksia tai preferenssejä esimerkiksi saatavuuden tai eheyden välillä, vaan vaatimukset olivat lähinnä muotoa ”kaikki olisi hyvä saada mutta mitään ei ehdottomasti vaadita”, teoreemasta tunnistettuja ”kategorioita” ei pystytty hyödyntämään yksiselitteisesti rajauksen tukena. Tämän lisäksi teoreeman ikä oli selvästi havaittavissa yhdistettäessä se nykyaikaisiin DBMS:iin: Huomattiin, että moderneja järjestelmiä pystytään konfiguroimaan kategorioihin vaatimusten mukaan jopa transaktiotasolla. Moderneja DBMS:iä ei pystytä näin ollen tiukasti asettamaan teoreemasta tunnistettuihin kategorioihin, joten myöskään teoreeman pohjalta tehtävä rajaus ei tässä tilanteessa tuottanut yksiselitteisiä tuloksia. Tämän lisäksi CAP-teoreeman näkökulma keskittyy erityisesti hajautettuihin järjestelmiin, joita tässä tutkielmassa ei tarkasteltu syvällisemmin. Vaikuttaisi siltä, että CAP-teoreemasta johdetulle näkökulmalle parempi sovelluskohde olisi esimerkiksi valmiiksi hajautetun järjestelmän käyttötapausten vaatimusten tunnistaminen, jonka pohjalta pystyttäisiin hahmottelemaan valitun DBMS:n konfiguraatiota tuotantokäyttöön.

Voitiin myös huomata, että ainakaan tässä tutkielmassa tehdyn pelkästään teoriaan perustuvan rajauksen pohjalta ei pystytä tekemään suoraa DBMS-tuotteen valintaa. Vaikka soveltuvimmat tietokantaparadigmat pystyttiin tunnistamaan, paradigmojen sisäinen tuotevalikoima on silti erittäin suuri. Tutkielmassa tehdyt konkreettiset tuotevalinnat perustuivat täten joko järjestelmässä valmiiksi hyödynnettyihin teknologioihin tai tuotteen suosioon. Tämän pohjalta voidaan spekuloida, että jos valintaa pitäisi tehdä täysin uudelle järjestelmälle ilman olemassa olevaa pohjaa, tietomallin arvioinnin jälkeistä konkreettista tuotteen valintaa on vaikea perustella pelkällä soveltuvuusarvioinnilla. Tällöin luultavasti esiin nousee muita tekijöitä, kuten järjestelmän kehittäjien tai omistajien kokemustaso, preferenssi tuotteiden välillä, tuotteen lisensointi sekä löydettävän dokumentaation taso ja määrä.

Lopuksi suorituskykytestauksen avulla pystyttiin myös huomaamaan, että vaikka teorian näkökulmasta arvioitiin tuotteiden soveltuvan tietomallille, tuotteiden välisissä tehokkuuksissa ja näin ollen myös soveltuvuuden tasossa oli monien kertaluokkien selkeitä eroja. Tämä havainnollistaa, että vaikka teorian tasolla tietty tietokantatuote soveltuisi käytettävälle tietomallille, todellinen tilanne pystytään näkemään vasta konkreettisen testauksen kautta. Tämä osoittaa, että tietokannanhallintajärjestelmien soveltuvuuden arviointi järjestelmille on selkeästi hyvin monihaarainen ja moninäkökulmainen prosessi, jossa vaaditaan sekä teorian, että käytännön näkökulmien tutkimusta ja arviointia oikeellisiin tuloksiin pääsemiseksi.

Edellä mainittujen huomioiden perusteella ensimmäiseen tutkimuskysymykseen voidaan osittain vastata seuraavasti. Tarkasteltavalle järjestelmälle, tai sen osalle, parhaiten soveltuvien tietokannanhallintajärjestelmien rajaaminen sekä valitseminen tulisi tehdä vaiheittain, peilaten ensin arviointiin valittavia tietokantaparadigmoja järjestelmän tietotarpeeseen. Tämän jälkeen tuotteiden valintaan tulisi pyrkiä tunnistaa puoltavia tekijöitä kuten olemassa olevia

järjestelmässä hyödynnettäviä teknologioita tai esimerkiksi tarkastella järjestelmän kehittäjien kokemustasoa eri tuotteista. Alustavien valintojen valmistuttua on ehdottoman tärkeää, että tietokannanhallintajärjestelmää tai -järjestelmiä konkreettisesti testataan tietomallilla, mielellään vertaillen vaihtoehtoisia tuotteita toisiinsa. Esimerkiksi suorituskykytestaus voi toimia tämän testauksen pohjana, kuten tässä tutkielmassa on toimittu, mutta myös muita testauksen tapoja voitaisiin suosia: järjestelmän ylläpidon helppous, infrastruktuuriin sopiminen (tarvitaanko esimerkiksi hajautusta) tai kehittämisen nopeus ja helppous voisivat olla muutamia mahdollisia mitattavia asioita.

8.2 Suorituskykytestaus, tulokset ja toinen tutkimuskysymys

Tutkielmassa esitettyyn toiseen tutkimuskysymykseen, ”Mikä valikoiduista tietokannanhallintajärjestelmistä on tehokkain vaihtoehto järjestelmästä havaitun ongelman ratkaisemiseen?”, löydettiin tutkimukselle tehtyjen rajausten valossa yksiselitteinen vastaus.

Vaikka tulokset vaikuttavat hyvin selkeiltä, on tutkimuksen asetelmassa kuitenkin huomioitavia tekijöitä, jotka vaikuttavat tulosten tulkintaan sekä tutkimuskysymyksen vastauksen täydellisyyteen. Tulee painottaa, että suorituskykytestauksen tulokset kuvaavat vain erittäin pieneen järjestelmän osan tarkastelua. Testattavan osion valinnan perusteena oli aiemmin havaittu selkeä ongelmakohta järjestelmän toiminnassa, johon haluttiin erityisesti löytää ratkaisu. Suorituskykytestauksen tulokset vastaavat juuri tähän hyvin spesifiin kysymykseen / ongelmaan, mutta niiden yleistettävyys lopun järjestelmän toimintaan jää täysin spekulaation varaan. Esimerkiksi DBMS:n kirjoitusoperaatioita ei sisällytetty suorituskykytestaukseen, vaikka myös niiden toiminta on hyvin oleellista yleisen tehokkuuden arvioinnin kannalta. Vaikka rajaus on tämän tutkielman näkökulmasta jo työmääränsä takia perusteltua, yleistettävästi järjestelmän tietomallille tehokkaimman DBMS:n löytämiseksi olisi oleellista suorittaa huomattavasti laajempaa suorituskykytestausta ottaen huomioon käyttötapauksia laajemmalla näkökulmalla.

Tämän lisäksi tutkimuksen tuloksia ei vertailtu itse tutkittavan järjestelmän toimintaan tai sen tehokkuuteen. Tiedostojärjestelmän suorituskyvyn mittaaminen arvioitiin tutkielmaa tehtäessä haastavaksi, käytännössä täysin omaksi suorituskykytestikseen, joten se rajattiin tutkimuksen ulkopuolelle. Tämän lisäksi esitetty tutkimuskysymys koskee vain valikoitujen tietokannanhallintajärjestelmien vertailua, joten tulosten vertaaminen alkuperäiseen toteutukseen rajautuu perustellusti pois. Jotta tehdyn tutkimuksen tuloksia pystyttäisiin hyödyntämään konkreettisen ongelman ratkaisussa, tulisi tämä vertailu kuitenkin suorittaa. Tällä pystyttäisiin selvittämään, onko tehokkaimmaksi havaittu DBMS tehokkaampi, kuin alkuperäinen toteutus. Voidaan kuitenkin esittää varovainen olettamus, että DBMS:t ovat yleisesti ottaen tehokkaampia tiedon käsittelyyn ja varastointiin verrattuna tiedostojärjestelmiin.

Tutkielmaa tehtäessä rajattiin pois myös tietomallin muokkaaminen. Tämä on suoraan huomattavissa tuloksista, joissa relaatiopohjaiset tietokannat suoriutuivat monta kertaa huonommin kuin dokumenttitietokanta. Tämän voidaan arvioida muiden mahdollisten syiden lisäksi johtuvan itse käytetystä tietomallista, joka ei vaikuttaisi alkuunsakaan soveltuvan relaatiomalliin. Tämä perustuu teorian pohjalta tehdystä arvioinnista huolimatta havaittuun normalisoinnin mahdottomuuteen käytännön toteutuksessa. Alkuperäisen tietomallin yleinen järkevyys tai soveltuvuus mihinkään tietokannanhallintajärjestelmään havaittiin kyseenalaiseksi. Koska tutkielmassa haluttiin kuitenkin tutkia alkuperäisen tietomallin vaihtoehtoisia toteutusalustoja, itse tietomallia ei arvioitu tarkemmin.

Edellä olevat huomiot huomioon ottaen voidaan silti todeta, että toiseen tutkimuskysymykseen pystyttiin vastaamaan suoritetulla suorituskykytestauksella kysymykselle asetettujen rajoitteiden valossa. Tuloksia tarkastellessa tulee kuitenkin muistaa, että laajemmalla suorituskyvyn tai muun metriikan testauksella, mahdollisesti parannellulla tietomallilla, tulokset luultavasti olisivat erilaiset.

8.3 Kontribuutiot

Tutkielman kontribuutiot painottuvat sen suunnittelutieteellisen luonteen seurauksena erityisesti käytännön ongelmanratkaisuun. Tärkeimpänä käytännön kontribuutiona järjestelmän kehittäjille oli tutkielman toiseen tutkimuskysymyksen vastaus, toisin sanoen esitetyn käytännön ongelman tehokkaimman vaihtoehtoisen ratkaisun tunnistaminen. Tämän lisäksi käytännön kontribuutiona voidaan pitää tutkimuksen toteuttamiseen käytettyjä skriptejä, joita voidaan mahdollisesti uusiokäyttää, jos käytännön ratkaisussa päätytään siirtymään tutkielmassa esitettyyn vaihtoehtoon. Esimerkiksi tietokannan datan tuontiin käytetty skripti olisi lähes tuotantokelpoinen, jos dokumentoimattomat erityistilanteet pystyttäisiin huomioimaan. Tämän lisäksi tutkielma tarjoaa järjestelmän kehittäjille valmiiksi mallinnetun tietokannan loogisen rakenteen, jota on mahdollista joko käyttää suoraan tai jatkokehittää tarpeen mukaan. Tutkielmassa on tehty myös käytännönläheisempää käyttötapausten ja DBMS:ien soveltuvuuden arviointia järjestelmään peilaten, jonka hyötykäyttö esimerkiksi päätöksenteossa on mahdollista.

Yleistettävämpänä käytännön kontribuutiona voidaan pitää tutkielmassa suoritettua suorituskykytestausta sekä sen toteuttavaa skriptirakennetta. Koska täysin erilaisten tietokantatuotteiden välinen kattava suorituskykytestaus on hyvin haastavaa, voidaan tutkielmassa tehtyä suppeaa suorituskykytestausta käyttää yleistettävänä pohjana alustavaan, nopeaan testaukseen. Skriptin laajentaminen kattavammaksi testausalustaksi esimerkiksi sekoittamalla kirjoitusoperaatioita testaukseen ei myöskään ole välttämättä erityisen haastavaa, kunhan testaaja osaa suunnitella testinsä oikeellisesti.

Vaikka tutkielman tieteelliset/teoreettiset kontribuutiot jäävät vähäisemmäksi, myös niitä on tunnistettavissa. Ensimmäisenä tieteellisenä kontribuutiona

voidaan pitää DSRM:n vaivatonta soveltamista jokseenkin teknisempään tutkimukseen, mikä validoi metodin soveltuvuutta tietojärjestelmätieteen moninaiseen tutkimukseen. Toisena tieteellisenä kontribuutiona voidaan pitää tutkielmassa tehtyä erilaisten teoreettisten näkökulmien pohjalta suoritettua tietokannanhallintajärjestelmien kategorisointia sekä näiden yhdistämistä käytännön rajaamiseen. Lopuksi suorituskykytestauksessa suoritettua tarkkaa dokumentointia, reiluuteen pyrkimistä ja testausprosessin käytännössä suoraa toistettavuutta voidaan pitää kontribuutiona, jonka toivotaan edistävän suorituskykytestaukseen keskittyvän tutkimuksen läpinäkyvyyttä ja oikeellisuutta.

8.4 Rajoitteet

Tutkielmaa tehtäessä tunnistettiin monia rajoitteita, joita on oleellista huomioida tutkimuksen reliabiliteetin ja validiteetin arvioinnissa, erityisesti käytännön testauksen näkökulmasta.

Tutkielman teoriaosan pääasiallisena rajoitteena toimii kirjallisuuskatsauksen suppeus. Tutkielmassa suoritettua kirjallisuuskatsausta ei suoritettu systemaattisella menetelmällä, joten teoriapohjan täydellinen pitävyys ja oikeellisuus voidaan kyseenalaistaa. Tämä on tärkeää huomioida, koska kirjallisuuskatsauksessa muodostetun teoriapohjan tuella tehdyllä tietokannanhallintajärjestelmien rajauksella on suora vaikutus käytännön tutkimukseen sekä mahdollisesti sen tuloksiin. Tutkielmasta rajattiin pois muun muassa monta tietokantamallia teorian tasolla perustellusti, mutta koska käytännön testauksen oleellisuus osoitettiin, voisi olla perusteltua sisällyttää myös pois rajatut tietokantamallit edes arvioinnin piiriin. Tämän lisäksi tutkielmaan valikoituneen CAP-teoreeman sopivuutta tutkielman näkökulmaan voidaan kyseenalaistaa: Teoreeman näkökulma koskee erityisesti hajautettuja järjestelmiä, mitä tutkielmassa ei käsitelty. Teoreeman soveltaminen käytännön tasolle jäi kuitenkin suoritetun tutkimuksen kontekstissa suhteellisen vähäiseksi, joten mahdollisen epäsoveltuvuuden vaikutusten oletetaan olevan pienet.

Tutkielman käytännön tason rajoitteet jakautuvat ”taustoituksen” ja suorituskykytestauksen rajoitteisiin. Taustoitukseen, eli järjestelmästä tarjottuun dokumentaatioon sekä tietomallin suunnitteluun ja sen toteutukseen, liittyi tiettyjä ongelmia. Tarjotusta dokumentaatiosta puuttui tiettyjä mahdollisesti järjestelmän toiminnalle oleellisia tekijöitä, jotka piti hylätä käytetystä testidatasta niiden käyttötarkoituksen ymmärryksen puutteen takia. Täten tutkimuksessa käytetty testidata ei itse asiassa täysin vastaa järjestelmän tuotantotietokantaa, vaan pelkästään sen dokumentoituja osia. Toinen ongelmallisuus havaittiin käytetystä tietomallista, jonka järkevyyttä lyhyesti arvioitiin aiemmassa alaluvussa. Tietomallin havaittu käytännön soveltumattomuus tietyille tietokantamalleille luo mahdollisen epäkohdan tutkimuksen reiluuteen: Jos tietomallia olisi muutettu tai osattu soveltaa sopivammaksi relaatiomalliin, tutkimuksen tulokset saattaisivat olla erilaiset. Voidaan spekuloida, että kokeneempi tietokantojen suunnittelija olisi mahdollisesti pystynyt suunnittelemaan tietokantojen mallit

järkevämmäksi ja/tai tehokkaammaksi. Kuitenkin, koska kaikkien tietokantojen mallintajana on toiminut tutkielmassa sama henkilö, jonka kokemustaso tuotteista on suunnilleen samaa tasoa, voidaan olettaa, että mahdolliset virheet ja/tai epäkohdat mallintamisessa ovat toistuneet suunnilleen samalla tasolla tuotteiden välillä.

Vaikka suorituskykytestauksen suunnittelun lähtökohtana oli testauksen mahdollisimman vahva reiluus, myös siitä voidaan tunnistaa rajoitteita. Edellä mainittu tietomallin soveltamisen hankaluus heijastuu myös suorituskykytestien reiluudessa. Koska tietomallia ei käytännön tasolla pystytty normalisoimaan, piti relaatiomallisten tuotteiden kyselylauseessa käyttää regex-funktiota, jotta halutut tiedot saatiin haettua tietokannoista. Tämän funktion toiminta sekä siitä seuraava kyselyn monimutkaisuus on ongelmallista testauksen reiluuden kannalta, koska dokumenttikannassa kysely pystyttiin muodostamaan suhteellisesti yksinkertaisemmin. Tämä kyselyn monimutkaisuus on luultavasti vaikuttanut etenkin CockroachDB:n suorituskykyyn heikentävästi, koska NewSQL-tuotteet soveltuvat lähinnä yksinkertaisille kyselyille. Niin kuin mallintamisen tapauksessa, myös tässä tulee huomata, että kokeneempi kehittäjä olisi mahdollisesti voinut muodostaa kyselyt yksinkertaisemmin tai tehokkaammin, mikä voisi vaikuttaa tutkimuksen tuloksiin. Tämän lisäksi esimerkiksi tuotteiden optimointia ei viety äärimmäisyyksiin, vaan se pyrittiin pysymään ”perustasolla”. Koska optimoinnin tasoa on hyvin vaikea mitata tai arvioida, voidaan olettaa, että tuotteita ei optimoitu täysin tasavertaisesti, mikä saattoi vääristää suorituskykytestauksen tuloksia.

Yleisesti ottaen voidaan todeta, että tutkimuksen suorittajan tietotaidon taso valituista tietokantatuotteista on mitä luultavimmin vaikuttanut tutkimuksen tuloksiin. Vaikka tutkimusta suorittaessa on pyritty hyödyntämään muun muassa taustatutkimusta, järjestelmän dokumentaatiota, valittujen tuotteiden dokumentaatiota sekä yleistiedon mukaisia tietokantasuunnittelun ja testauksen de-facto -standardeja, voidaan olettaa, että suoritettu tutkimus ei ole täydellinen. Kuitenkin voidaan myös olettaa, että mahdolliset puutteet tutkimuksessa yleistyivät kaikille tuotteille sekä koko testausprosessiin, jolloin käytännössä testauksen reiluus säilyy. Testaus on myös pyritty dokumentoimaan mahdollisimman läpinäkyvästi, jotta nämä mahdolliset epäkohdat pystytään huomioimaan joko tutkielman tulosten käytäntöön soveltamisessa tai jatkotutkimuksen teossa.

8.5 Jatkotutkimus

Tutkielman teon aikana esiin nousi monia varteenotettavia jatkotutkimuksen aiheita, joiden tarkastelu voisi tuottaa mielekkäitä kontribuutioita niin teorian kuin käytännön näkökulmista.

Teoreettisella tasolla mahdolliseksi jatkotutkimuksen aiheeksi nousi tutkielmassa tehdyn taustatutkimukseen perustuvan tietokannanhallintajärjestelmien soveltuvuuden arvioinnin jatkokehittäminen yleistettävään muotoon. Vaikka yleistettävän soveltuvuusarvioinnin ohjeistuksen muodostaminen on

oletettavasti hyvin haastavaa, taustatutkimuksen sekä erityisesti tapaustutkimusten tarkastelun avulla voisi olla mahdollista kehittää yleisiä ohjeistuksia siitä, miten vähintään alustavaa tietokannanhallintajärjestelmien valinnan rajausta voitaisiin toteuttaa. Tästä voitaisiin mahdollisesti kehittää esimerkiksi viitekehys, joka yhdistelee mm. järjestelmän käyttötapausten tietotarpeiden peilaamista teorian kautta muodostettuihin tietokannanhallintajärjestelmien kategorioihin.

Käytännön tasolla tutkielmasta nousee suuri määrä jatkotutkimuksen aiheita jo tutkielman näkökulman rajauksen takia. Tässä tutkielmassa ei suoritettu laajaa soveltuvuuden arviointia tai vertailua olemassa olevaan järjestelmään. Tämän takia olisi tärkeää ottaa tämän tutkielman tuloksena tunnistettu ”tehokkain” tietokannanhallintajärjestelmä (tai soveltuvimmiksi arvioidut järjestelmät) ja jatkaa niiden soveltuvuus- ja suorituskykytestausta laajemmasta näkökulmasta. Esimerkiksi DBMS:iä voitaisiin integroida tarkasteltavaan järjestelmään laajemmin tai tarkasteltavien käyttötapauksien määrää voitaisiin lisätä huomattavasti. Tämän kaltaisesta tutkimuksesta saataisiin vielä konkreettisempia tuloksia, joita pystyttäisiin aidosti vertaamaan reaalimaailman tilanteeseen. Erityisesti järjestelmän nykyisen toteutuksen suorituskyvyn reilu mittaaminen toisi huomattavaa lisäarvoa myös tämän tutkielman tulosten tarkasteluun.

Lisäksi jatkotutkimusta voitaisiin tehdä erilaisista näkökulmista. Tietokantojen hajautuksen arviointi rajattiin tästä tutkielmasta pois, vaikka hajautusmahdollisuudet ovat oleellinen osa moderneja tietokannanhallintajärjestelmiä. Täten voitaisiin tutkia esimerkiksi hajautuksen vaikutusta suorituskykyyn, joko tuotekohtaisesti, tai laajemmalla, monia tuotteita yhdistelevällä tutkimuksella. Myös järjestelmän tietomallin arviointi sekä muokkaaminen olisi hyvin mielekäs jatkotutkimuksen kohde, minkä vaikutuksena suorituskyvyn testauksen tulokset saattaisivat huomattavasti erota tässä tutkielmassa saatuihin tuloksiin.

9 YHTEENVETO

Tämän tutkielman tarkoituksena oli löytää tarkasteltavalle järjestelmälle soveltuvin vaihtoehtoinen tiedon varastoinnin toteutus seuraavien tutkimuskysymysten kautta:

- *Miten pystytään rajaamaan sekä valitsemaan tarkasteltavalle järjestelmän osalle parhaiten soveltuvat tietokannanhallintajärjestelmät?*
- *Mikä valikoiduista tietokannanhallintajärjestelmistä on tehokkain vaihtoehto järjestelmästä havaitun ongelman ratkaisemiseen?*

Tutkielmassa pystyttiin vastaamaan ensimmäiseen tutkimuskysymykseen osittain hyvin tapauskohtaisesta näkökulmasta. Tämä osittainen vastaus on kokonaisuudessaan esitetty edellisessä luvussa. Toiseen tutkimuskysymykseen löydettiin näennäisesti yksiselitteinen vastaus suorituskykytestauksen tuloksista, johon kuitenkin liittyi edellisessä luvussa käsiteltyjä olennaisia rajoitteita. Suoritettu tutkimus voidaan vain osittaisesta kysymyksiin vastaamisesta huolimatta arvioida onnistuneeksi, sillä käytännön ongelman ratkaisun pohjaksi pystyttiin muodostamaan tuloksiltaan merkittävä tietopohja, jonka avulla pystytään suorittamaan jatkotutkimusta ja -kehitystä.

Tutkielma suoritettiin konstruktiivisella tutkimusmenetelmällä soveltaen DSRM-metodologiaa tutkielman rakenteen sekä suorittamisen ohjenuorana. Tutkielmassa on hyödynnetty kaikkia metodologian esittämiä prosesseja tilanteen taustoittamisesta tulosten raportointiin. Tutkimuksessa muodostettiin kolme artefaktia, joiden välinen suorituskyvyn vertailu toimi tutkielman konkreettisena datan keräysmenetelmänä.

Tutkielman rakenne oli tiivistetysti seuraava. Taustoituksen jälkeen suoritettiin kirjallisuuskatsaus suorituskykytestaukseen valittavien tietokannanhallintajärjestelmien rajauksen tueksi kahdesta erilaisesta näkökulmasta. Tämän jälkeen suoritettiin suorituskykytestaus, jossa pyrittiin eteenkin reiluuteen erilaisten järjestelmien toisiinsa vertaamisen takia, dokumentoiden prosessi mahdollisimman kattavasti. Tämän jälkeen pohdintakappaleessa esitettiin kattava, yhteen

vetävä arvio tutkielman tuloksista, kontribuutioista, rajoitteista sekä mahdollisista jatkotutkimuksen aiheista.

LÄHTEET

Abadi, D. J. (2012). Consistency Tradeoffs in Modern Distributed Database System Design. *Computer (Long Beach, Calif.), 45*(2), 37-42. https://doi.org/10.1109/MC.2012.33

Agrawal, D., El Abbadi, A., Das, S., & Elmore, A. J. (2011). Database scalability, elasticity, and autonomy in the cloud. *International conference on database systems for advanced applications*. 2-15. https://doi.org/10.1007/978-3-642-20149-3_2

Albarak, M., Bahsoon, R., Ozkaya, I. & Nord, R. (2020). Managing Technical Debt in Database Normalization. *IEEE Transactions on Software Engineering, vol. 48, no. 3, pp. 755-772*. http://dx.doi.org/10.1109/TSE.2020.3001339

Almeida, R., Furtado, P. & Bernardino, J. (2015). Performance Evaluation MySQL InnoDB and Microsoft SQL Server 2012 for Decision Support Environments. *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering (C3S2E '15)*. Association for *Computing Machinery, New York, NY, USA,* 56–62. https://doi.org/10.1145/2790798.2790808

Angles, R., & Gutierrez, C. (2008). Survey of graph database models. *ACM computing surveys, 40*(1), 1-39. https://doi.org/10.1145/1322432.1322433

Brewer, E. (2000). Towards robust distributed systems. *PODC. 7.* https://dl.acm.org/doi/10.1145/343477.343502

Brewer, E. (2012). Pushing the CAP: Strategies for consistency and availability. *Computer (Long Beach, Calif.), 45*(2), 23-29. https://doi.org/10.1109/MC.2012.37

Băzăr, C., & Iosif, C.S. (2014). The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase. *Database Systems Journal*, 5, 49-59.

Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. *SIGMOD record, 39*(4), 12-27. https://doi.org/10.1145/1978915.1978919

Chamberlin, D. D., & Boyce, R. F. (1974). SEQUEL: A structured English query language. *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 249-264. https://doi.org/10.1145/800296.811515

Chaudhry, N., & Yousaf, M. M. (2020). Architectural assessment of NoSQL and NewSQL systems. *Distributed and parallel databases : an international journal, 38*(4), 881-926. https://doi.org/10.1007/s10619-020-07310-1

Chen, P. (1976). The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst. 1*, 1, 9–36. https://doi.org/10.1145/320434.320440

Clifton, C., & Garcie-Molina, H. (2000). The design of a document database. *Proceedings of the ACM conference on Document processing systems (DOCPROCS '88). Association for Computing Machinery, New York, NY, USA*, 125–134. https://doi.org/10.1145/62506.62528

Cockroach Labs. (ei pvm.). Production Checklist. Haettu 25.4.2024 osoitteesta https://www.cockroachlabs.com/docs/v23.2/recommended-production-settings.html

Codd, E. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387. https://doi.org/10.1145/362384.362685

Codd, E. (1974). Recent Investigations in Relational Data Base Systems. *ACM Pacific.*

Cubukcu, U., Erdogan, O., Pathak, S., Sannakkayala, S., & Slot, M. (2021). Citus: Distributed postgresql for data-intensive applications. *Proceedings of the 2021 International Conference on Management of Data*, 2490-2502. https://doi.org/10.1145/3448016.3457551

Davoudian, A., Chen, L., & Liu, M. (2018). A Survey on NoSQL Stores. *ACM computing surveys, 51*(2), 1-43. https://doi.org/10.1145/3158661

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107-113. https://doi.org/10.1145/1327452.1327492

Dhamane, R., Martínez, M., Vianello, V., & Peris, R. (2014). *Performance evaluation of database replication systems*. https://doi.org/10.1145/2628194.2628214

Diogo, M., Cabral, B., & Bernardino, J. (2019). Consistency Models of NoSQL Databases. *Future internet, 11*(2), 43. https://doi.org/10.3390/fi11020043

Egger, S., Hossfeld, T., Schatz, R. & Fiedler M. (2012). Waiting times in quality of experience for web based services. *2012 Fourth International Workshop on Quality of Multimedia Experience, Melbourne, VIC, Australia, 2012*, 86-96. https://doi.org/10.1109/QoMEX.2012.6263888

Eswaran, K., Gray, J., Lorie, R., & Traiger, I. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM, 19*(11), 624-633. https://doi.org/10.1145/360363.360369

Ganesh Chandra, D. (2015). BASE analysis of NoSQL database. *Future generation computer systems, 52*, 13-21. https://doi.org/10.1016/j.future.2015.05.003

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT news, 33*(2), 51-59. https://doi.org/10.1145/564585.564601

Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP Theorem. *Computer (Long Beach, Calif.),* 45(2), 30-36. https://doi.org/10.1109/MC.2011.389

Gomes, V. B. F., Kleppmann, M., Mulligan, D. P., & Beresford, A. R. (2017). Verifying strong eventual consistency in distributed systems. *Proceedings of ACM on programming languages, 1*(OOPSLA), 1-28. https://doi.org/10.1145/3133933

Gray, J., Helland, P., O'Neil, P., & Shasha, D. (1996). The dangers of replication and a solution. *SIGMOD record, 25*(2), 173-182. https://doi.org/10.1145/235968.233330

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM computing surveys, 15*(4), 287-317. https://doi.org/10.1145/289.291

Han, J., Haihong, E., Le, G. & Du, J. (2011). Survey on NoSQL database. *2011 6th International Conference on Pervasive Computing and Applications, Port Elizabeth, 2011,* 363-366. https://doi.org/10.1109/ICPCA.2011.6106531

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

IBM. (ei pvm.) What is the CAP theorem?. Haettu 20.11.2023 osoitteesta https://www.ibm.com/topics/cap-theorem.

Jowan, S. A., Swese, R. F., Aldabrzi, A. Y., & Shertil, M. S. (2016). Traditional RDBMS to NoSQL database: new era of databases for big data. *J. Humanit. Appl. Sci*, *29*(29), 83-102. https://doi.org/10.48550/arXiv.1307.0191

Jyväskylän yliopiston informaatioteknologian tiedekunta. (ei pvm. a). The Interactive Material. Haettu 18.10.2023 osoitteesta https://tim.jyu.fi/view/about/fi#lPH7pfE6LGvn.

Jyväskylän yliopiston informaatioteknologian tiedekunta. (ei pvm. b). Dokumenttien tietorakennemalli. Haettu 12.04.2024 osoitteesta https://tim.jyu.fi/view/tim/TIMin-kehitys/dokumenttien-tietorakennemalli

Kamel, I., & Kamel, K. (2011). *Toward protecting the integrity of relational databases*. https://doi.org/10.1109/WorldCIS17046.2011.5749863

Kent, W. (1983). A simple guide to five normal forms in relational database theory. *Commun. ACM 26, 2 (Feb. 1983)*, 120–125. https://doi.org/10.1145/358024.358054

Kleppmann, M. (2015). A Critique of the CAP Theorem. *arXiv.org*. https://doi.org/10.48550/arxiv.1509.05393

Lakshman, A., & Malik, P. (2010). Cassandra - A Decentralized Structured Storage System. *Operating systems review, 44*(2), 35-40. https://doi.org/10.1145/1773912.1773922

Lehner, W., & Sattler, K.-U. (2010). Database as a service (DBaaS). *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. https://doi.org/10.1109/icde.2010.5447723

Mihai, G. (2020). Comparison between Relational and NoSQL Databases. *Analele Universităţii "Dunărea de Jos" Galaţi. Fascicula I, Economie şi informatica aplicata,* 26(3), 38-42. https://doi.org/10.35219/eai15840409134

MongoDB Inc. (2024a). Collections. Haettu 25.2.2024 osoitteesta https://www.mongodb.com/docs/compass/current/collections/

MongoDB Inc. (2024b). MongoDB Documentation. Haettu 15.3.2024 osoitteesta https://www.mongodb.com/docs/

MongoDB Inc. (2024c). MongoDB Licensing. Haettu 15.3.2024 osoitteesta https://www.mongodb.com/legal/licensing/community-edition

MongoDB Inc. (2024d). Data Modeling. Haettu 15.3.2024 osoitteesta https://www.mongodb.com/docs/manual/data-modeling/

Pavlo, A., & Aslett, M. (2016). What's Really New with NewSQL? *SIGMOD record, 45*(2), 45-55. https://doi.org/10.1145/3003665.3003674

Peffers, K., Tuunanen, T., Rothenberger, M. & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24, 45-77. https://doi.org/10.2753/MIS0742-1222240302

Pina, E., Sá, F., & Bernardino, J. (2023). NewSQL Databases Assessment: CockroachDB, MariaDB Xpand, and VoltDB. *Future internet*, 15(1), 10. https://doi.org/10.3390/fi15010010

Raasveldt, M., Holanda, P., Gubner, T., & Mühleisen, H. (2018). Fair benchmarking considered difficult: Common Pitfalls In Database Performance Testing. *Proceedings of the Workshop on Testing Database Systems*. https://doi.org/10.1145/3209950.3209955

Seeger, M., & Ultra-Large-Sites, S. (2009). Key-value stores: a practical overview. *Computer Science and Media, Stuttgart*.

solid IT gmbh (2023a). DB-Engines Ranking. Haettu 26.11.2023 osoitteesta https://db-engines.com/en/ranking.

solid IT gmbh (2023b). DB-Engines Ranking of Key-value stores. Haettu 26.11.2023 osoitteesta https://db-engines.com/en/ranking/key-value+store.´

solid IT gmbh (2023c). DB-Engines Ranking of Wide Column Stores. Haettu 26.11.2023 osoitteesta https://db-engines.com/en/ranking/wide+column+store.

solid IT gmbh (2023d). DB-Engines Ranking of Document Stores. Haettu 26.11.2023 osoitteesta https://db-engines.com/en/ranking/document+store.

solid IT gmbh (2023e). DB-Engines Ranking of Graph DBMS. Haettu 26.11.2023 osoitteesta https://db-engines.com/en/ranking/graph+dbms.

Stonebraker, M. (2010). In search of database consistency. *Communications of the ACM, 53*(10), 8-9. https://doi.org/10.1145/1831407.1831411

Song, I., Evans, M. & Park, E.K. (1995). A comparative analysis of Entity-Relationship Diagrams. *Journal of Computer and Software Engineering, Vol. 3, No.4 (1995), pp. 427-459*.

Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., Bardea, P., Ranade, A., Darnell, B., Gruneir, B., Jaffray, J., Zhang, L., & Mattis, P. (2020). CockroachDB: The Resilient Geo-distributed SQL Database. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* https://doi.org/10.1145/3318464.3386134

Taipalus, T. (2024). Database management system performance comparisons: A systematic literature review. *Journal of Systems and Software 208*. https://doi.org/10.48550/arXiv.2301.01095.

Tansel, A. U. (2004). Temporal data modeling and integrity constraints in relational databases,. *Lecture Notes in Computer Science*, 459–469. https://doi.org/10.1007/978-3-540-30182-0_47

The Apache Software Foundation. (ei pvm.) Cassandra Basics. Haettu 22.11.2023 osoitteesta https://cassandra.apache.org/_/cassandra-basics.html.

The PostgreSQL Global Development Group (2024a). PostgreSQL: The World's Most Advanced Open Source Relational Database. Haettu 12.3.2024 osoitteesta https://www.postgresql.org/

The PostgreSQL Global Development Group (2024b). License. Haettu 12.3.2024 osoitteesta https://www.postgresql.org/about/licence/

Vincent, M. W., & Srinivasan, B. (1993). A note on relation schemes which are in 3NF but not in BCNF. *Information Processing Letters, 48(6),* 281–283. https://doi.org/10.1016/0020-0190(93)90169-a

Wada, H., Fekete, A. D., Zhao, L., Lee, K., & Liu, A. (2011). Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. *CIDR*, *11*, 134-143.

Wang, T. J., Du, H., & Lehmann, C. M. (2010). Accounting For The Benefits Of Database Normalization. *American journal of business education, 3*(1), 41-52. https://doi.org/10.19030/ajbe.v3i1.371

Xu, L., Sun, D., & Liu, D. (2010). Study on methods for data confidentiality and data integrity in relational database. *2010 3rd International Conference on Computer Science and Information Technology*, 292-295. https://doi.org/10.1109/iccsit.2010.5565009

Zamanian, E., Yu, X., Stonebraker, M., & Kraska, T. (2019). Rethinking database high availability with RDMA networks. *Proceedings of the VLDB Endowment, 12*(11), 1637-1650. https://doi.org/10.14778/3342263.3342639

Zhou, P., Li, M., Huang, J., & Fang, H. (2014). Research on Database Schema Comparison of Relational Databases and Key-Value Stores. *Advanced Materials Research, 1049-1050, 1860–1863*. https://doi.org/10.4028/www.scientific.net/amr.1049-1050.1860

LIITE 1 RELAATIOTIETOKANTOJEN TAULUJEN SEKÄ INDEKSIEN LUONTILAUSEET

```
-- Tables
CREATE TABLE document (
        doc_version_id SERIAL NOT NULL PRIMARY KEY,
        doc_id INT NOT NULL,
        doc_version_major INT NOT NULL,
        doc_version_minor INT NOT NULL
);

CREATE TABLE paragraph (
        par_version_id SERIAL NOT NULL PRIMARY KEY,
        doc_version_id INT,
        par_id TEXT NOT NULL,
        par_hash TEXT NOT NULL,
        md TEXT NOT NULL,
        CONSTRAINT fk_doc_version_id
                FOREIGN KEY (doc_version_id)
                        REFERENCES document(doc_version_id)
                        ON DELETE NO ACTION
);

CREATE TABLE html_cache (
        par_version_id INT NOT NULL PRIMARY KEY,
        hash_key TEXT NOT NULL,
        cache TEXT NOT NULL,
        CONSTRAINT fk_par_version_id
                FOREIGN KEY (par_version_id)
                        REFERENCES paragraph(par_version_id)
                        ON DELETE CASCADE
);

CREATE TABLE par_attrs (
        attr_id SERIAL NOT NULL PRIMARY KEY,
        par_version_id INT NOT NULL,
        attr_key TEXT NOT NULL,
        attr_value TEXT NOT NULL,
        CONSTRAINT fk_par_version_id
                FOREIGN KEY (par_version_id)
                        REFERENCES paragraph(par_version_id)
                        ON DELETE CASCADE
);

CREATE TABLE paragraph_list (
        doc_version_id INT NOT NULL PRIMARY KEY,
        list text NOT NULL,
```

```
        CONSTRAINT fk_doc_version_id
                FOREIGN KEY (doc_version_id)
                        REFERENCES document(doc_version_id)
                        ON DELETE CASCADE
);

CREATE TABLE changelog (
        entry_id SERIAL PRIMARY KEY,
        doc_version_id INT NOT NULL,
        group_id INT NOT NULL,
        op varchar(8) NOT NULL,
        time TIMESTAMP NOT NULL,
        CONSTRAINT fk_doc_id
                FOREIGN KEY (doc_version_id)
                        REFERENCES document(doc_version_id)
                        ON DELETE CASCADE
);

CREATE TABLE op_params (
        entry_id INT NOT NULL PRIMARY KEY,
        before_vid INT,
        old_hash_vid INT,
        new_hash_vid INT,
        CONSTRAINT fk_entry_id
                FOREIGN KEY (entry_id)
                        REFERENCES changelog(entry_id)
                        ON DELETE CASCADE,
        CONSTRAINT fk_before_vid
                FOREIGN KEY (before_vid)
                        REFERENCES paragraph(par_version_id)
                        ON DELETE NO ACTION,
        CONSTRAINT fk_old_hash_vid
                FOREIGN KEY (old_hash_vid)
                        REFERENCES paragraph(par_version_id)
                        ON DELETE NO ACTION,
        CONSTRAINT fk_new_hash_vid
                FOREIGN KEY (new_hash_vid)
                        REFERENCES paragraph(par_version_id)
                        ON DELETE NO ACTION
);

-- Indexes
CREATE INDEX par_id_hash_idx ON paragraph(par_id, par_hash);
CREATE INDEX par_attr_pvid_idx ON par_attrs(par_version_id);
CREATE INDEX doc_id_idx ON document(doc_id);
```

LIITE 2 DOKUMENTTITIETOKANNAN COLLECTIONIEN RAKENTEET SEKÄ INDEKSIT

```
-- Document
{
    _id <ObjectId>
    doc_id <Int32>
    doc_version_major <Int32>
    doc_version_minor <Int32>
    changelog ref<ObjectId>
    paragraphs [
        {
            _id ref<ObjectId>
            pos <Int32>
        }
    ]
}

-- Manually created Document indexes

par_id_1_t_1: compound (par_id asc, t asc)
doc_id_1: (doc_id asc)
doc_version_minor_-1_doc_version_major_-1: compound (doc_version_minor
desc, doc_version_major desc)

-- Paragraph

{
    _id <ObjectId>
    par_id <String>
    doc_id <Int32>
    md <String>
    t <String>
    h {<String>, <String>}
    attrs {<String>, <Any>}
}

-- Manually created Paragraph indexes

par_id_1_t_1: compound (par_id asc, t asc)
par_id_1: (par_id asc)

-- Changelog

{
    _id <ObjectId>
    doc_id <Int32>
```

```
    entries [
        {
            group_id <Int32>
            par_id <Int32>
            op <String>
            op_params {<String>, <Any>}
            ver [<Int32>, <Int32>]
            time <Date>
        }
    ]
}

-- Manually created changelog indexes

doc_id_1: (doc_id asc)
```

LIITE 3 POSTGRESQL :N AJOYMPÄRISTÖN KONFIGURAATIO

```
-- postgresql.Dockerfile

FROM postgres:16.2

# Deps
RUN apt-get update && apt-get install -y \
        python3 \
        python3-pip

# Python packages
RUN pip3 install psycopg2-binary tqdm --break-system-packages

-- postgresql.conf -file

max_connections = 17
shared_buffers = 8GB
effective_cache_size = 23GB
work_mem = 128MB

-- docker command used to start test container

docker run --name gradu_postgres -d -v /home/sajuminu/gradu/postgres-
data:/var/lib/postgresql/data -v /home/saju-
minu/gradu/scripts:/home/sajuminu/gradu/scripts -v /home/saju-
minu/gradu/conf/postgresql.conf:/etc/postgresql/postgresql.conf -e
POSTGRES_PASSWORD=root -p 5432:5432 sajuminu/gradu_postgres:latest -c
'config_file=/etc/postgresql/postgresql.conf'
```

LIITE 4 MONGODB:N AJOYMPÄRISTÖN KONFIGURAATIO

```
-- mongodb.Dockerfile

FROM mongodb/mongodb-community-server:7.0.0-ubuntu2204
USER root

# Deps
RUN apt-get update && apt-get install -y \
        python3 \
        python3-pip

# Python packages
RUN pip3 install pymongo tqdm

-- mongod.conf -file

net:
  maxIncomingConnections: 33

-- docker command used to start test container

docker run --name gradu_mongodb -d -v /home/sajuminu/gradu/mongodb-
data:/data/db -v /home/sajuminu/gradu/scripts:/home/saju-
minu/gradu/scripts -v /home/sajuminu/gradu/conf/mongod.conf:/etc/mon-
god.conf -p 27017:27017 gradu_mongodb:latest --config /etc/mongod.conf
```

LIITE 5 COCKROACHDB:N AJOYMPÄRISTÖN KONFIGURAATIO

```
-- cockroach.Dockerfile

FROM cockroachdb/cockroach:v23.2.4

# Deps
RUN microdnf install -y \
        python3 \
        python3-pip

# Python packages
RUN pip3 install psycopg2-binary tqdm

-- docker command used to start container, includes config flags

docker run -d --env COCKROACH_DATABASE=documentdb --env COCK-
ROACH_USER=root --env COCKROACH_PASSWORD=root --name=roach-single --
hostname=roach-single -p 26257:26257 -p 8080:8080 -v "roach-
single:/cockroach/cockroach-data" -v /home/saju-
minu/gradu/scripts:/home/sajuminu/gradu/scripts gradu_roach start-
single-node --insecure --cache=.35 --max-sql-memory=.35 --http-
addr=roach-single:8080
```

LIITE 6 RELAATIOTIETOKANTOJEN IMPORT-SKRIPTI

```
import os
import psycopg2
import psycopg2.extras
import json
from psycopg2 import OperationalError
from tqdm import tqdm

def connect_to_db(dbname, user, pwd, host, port):
 try:
  connection = psycopg2.connect(
  dbname=dbname,
  user=user,
  password=pwd,
  host=host,
  port=port
  )

  print("Connected to database")
  return connection
 except OperationalError as e:
  print(f"Failed to connect to database: {e}")
 return None

def insert_document(connection, doc_id, doc_version_major, doc_version_minor):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO document (doc_id, doc_version_major, doc_version_minor)
   VALUES (%s, %s, %s)
   RETURNING doc_version_id
  """, (doc_id, doc_version_major, doc_version_minor))
  doc_version_id = cursor.fetchone()[0]
  connection.commit()
  return doc_version_id
 except Exception as e:
  connection.rollback()
  print(f"Error inserting document with id {doc_id}, maj {doc_version_major}, min {doc_version_minor}:
{e}")
 finally:
  if connection:
   cursor.close()

def insert_paragraph_list(connection, doc_version_id, par_list):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO paragraph_list (doc_version_id, list)
   VALUES (%s, %s)
  """, (doc_version_id, par_list))
  connection.commit()
 except Exception as e:
  connection.rollback()
  print(f"Error inserting paragraph list for doc version id {doc_version_id}: {e}")
 finally:
  if connection:
   cursor.close()

def insert_paragraph(connection, doc_version_id, par_id, md, par_hash):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO paragraph (doc_version_id, par_id, md, par_hash)
   VALUES (%s, %s, %s, %s)
   RETURNING par_version_id
  """, (doc_version_id, par_id, md, par_hash))
  par_version_id = cursor.fetchone()[0]
  connection.commit()
  return par_version_id
 except Exception as e:
  connection.rollback()
  print(f"Error inserting paragraph in doc {doc_version_id}, par_id {par_id}, par hash {par_hash}:
{e}")
```

```
 finally:
  if connection:
   cursor.close()

def insert_html_cache(connection, par_version_id, hash_key, cache):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO html_cache (par_version_id, hash_key, cache)
   VALUES (%s, %s, %s)
  """, (par_version_id, hash_key, cache))
  connection.commit()
 except Exception as e:
  connection.rollback()
  print(f"Error inserting HTML cache for paragraph version ID {par_version_id}: {e}")
 finally:
  if connection:
   cursor.close()

def insert_par_attrs(connection, attrs_list):
 try:
  cursor = connection.cursor()
  insert_query = 'INSERT INTO par_attrs (par_version_id, attr_key, attr_value) VALUES %s'
  psycopg2.extras.execute_values(
   cursor, insert_query, attrs_list, template=None, page_size=100
  )
 except Exception as e:
  connection.rollback()
  print(f"Error inserting par_attrs: {e}")
 finally:
  if connection:
   cursor.close()

def insert_changelog(connection, doc_version_id, group_id, op, timestamp):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO changelog (doc_version_id, group_id, op, time)
   VALUES (%s, %s, %s, %s)
   RETURNING entry_id
  """, (doc_version_id, group_id, op, timestamp))
  entry_id = cursor.fetchone()[0]
  connection.commit()
  return entry_id
 except Exception as e:
  connection.rollback()
  print(f"Error inserting changelog for document ID {doc_version_id}: {e}")
 finally:
  if connection:
   cursor.close()

def insert_op_params(connection, entry_id, before_id, old_hash, new_hash):
 try:
  cursor = connection.cursor()
  cursor.execute("""
   INSERT INTO op_params (entry_id, before_vid, old_hash_vid, new_hash_vid)
   VALUES (%s, %s, %s, %s)
  """, (entry_id, before_id, old_hash, new_hash))
  connection.commit()
 except Exception as e:
  connection.rollback()
  print(f"Error inserting op_params for changelog entry id {entry_id}: {e}")
 finally:
  if connection:
   cursor.close()

def import_documents(directory, connection, amount_of_docs):
 print("Importing documents and their paragraph lists:")
 pbar = tqdm(total=amount_of_docs)
 for root, dirs, files in os.walk(directory):
  for doc_folder in dirs:
   doc_id = None
   if doc_folder.isdigit():
    doc_id = int(doc_folder)
   else:
    continue
   doc_path = os.path.join(root, doc_folder)
   for version_folder in os.listdir(doc_path):
```

```
    if version_folder == "changelog":
     continue
    doc_version_major = None
    if version_folder.isdigit():
     doc_version_major = int(version_folder)
    else:
     continue
    version_path = os.path.join(doc_path, version_folder)
    for minor_version_file in os.listdir(version_path):
     doc_version_minor_path = minor_version_file.split('.')[0]
     doc_version_minor = None
     if doc_version_minor_path.isdigit():
      doc_version_minor = int(doc_version_minor_path)
     else:
      continue
     doc_version_id = insert_document(connection, doc_id, doc_version_major, doc_version_minor)
     doc_list_path = os.path.join(version_path, doc_version_minor_path)
     with open(doc_list_path, 'r') as file:
      lines = file.readlines()
      txt = '\n'.join(lines)
      insert_paragraph_list(connection, doc_version_id, txt)
      pbar.update(1)
  break
 pbar.close()

def import_changelogs(directory, connection, amount_of_docs):
 print("Importing changelogs:")
 pbar = tqdm(total=amount_of_docs)
 for root, dirs, files in os.walk(directory):
  for doc_folder in dirs:
   doc_id = None
   if doc_folder.isdigit():
    doc_id = int(doc_folder)
   else:
    continue
   doc_path = os.path.join(root, doc_folder)
   for version_folder in os.listdir(doc_path):
    if version_folder != "changelog":
     continue
    changelog_path = os.path.join(doc_path, "changelog")
    with open(changelog_path, 'r') as file:
     for line in file:
      if line is not None:
       try:
        json_data = json.loads(line)
        group_id = json_data['group_id']
        par_id = json_data['par_id']
        op = json_data['op']
        timestamp = json_data['time']
        version = json_data['ver']

        cursor = connection.cursor()
        cursor.execute("""
         SELECT doc_version_id
         FROM document
         WHERE doc_id = %s
         AND doc_version_major = %s
         AND doc_version_minor = %s
        """, (doc_id, version[0], version[1]))
        result = cursor.fetchone()
        doc_version_id = result[0]
        cursor.close()
        if doc_version_id is not None:
         entry_id = insert_changelog(connection, doc_version_id, group_id, op, timestamp)

         if 'op_params' in json_data and json_data['op_params'] is not None:
          op_params = json_data['op_params']
          old_hash_vid = None
          new_hash_vid = None
          before_vid = None

          if "old_hash" in op_params:
           old_hash = op_params['old_hash']
           old_hash_vid = find_par_vid_hash(connection, par_id, old_hash)
          if "new_hash" in op_params:
           new_hash = op_params['new_hash']
           new_hash_vid = find_par_vid_hash(connection, par_id, new_hash)
          if "before_id" in op_params:
```

```
            before_id = op_params['before_id']
            # Just find some version id, can get doc_id from there...
            before_vid = find_some_par(connection, before_id)

           insert_op_params(connection, entry_id, before_vid, old_hash_vid, new_hash_vid)

         pbar.update(1)
        except Exception as e:
         print(f"Error processing changelog in document folder {changelog_path}: {e}")
   break
 pbar.close()

def find_par_vid_hash(connection, par_id, par_hash):
 cursor = connection.cursor()
 cursor.execute("""
  SELECT par_version_id
  FROM paragraph
  WHERE par_id = %s
  AND par_hash = %s
 """, (par_id, par_hash))
 result = cursor.fetchone()
 # Some pars are missing in test data...
 return result[0] if result is not None else None

def find_some_par(connection, par_id):
 cursor = connection.cursor()
 cursor.execute("""
  SELECT par_version_id
  FROM paragraph
  WHERE par_id = %s
  LIMIT 1
 """, (par_id,))
 result = cursor.fetchone()
 # Some pars are missing in test data...
 return result[0] if result is not None else None

def import_paragraphs(directory, connection, amount_of_pars):
 print("Importing paragraphs:")
 pbar = tqdm(total=amount_of_pars)
 for root, dirs, files in os.walk(directory, topdown=True):
  for doc_folder in dirs:
   doc_path = os.path.join(root, doc_folder)
   doc_id = None
   if doc_folder.isdigit():
    doc_id = int(doc_folder)
   else:
    continue
   for par_folder in os.listdir(doc_path):
    par_id = par_folder
    par_path = os.path.join(doc_path, par_folder)
    for json_file in os.listdir(par_path):
     if json_file == "current":
      continue
     json_path = os.path.join(par_path, json_file)
     with open(json_path, 'r') as file:
      try:
       json_data = json.load(file)
       par_hash = os.path.splitext(json_file)[0]
       id = json_data.get('id')
       md = json_data['md']
       doc_version_id = None

       cursor = connection.cursor()
       cursor.execute("""
        SELECT MIN(doc_version_id)
        FROM document
        WHERE doc_id = %s
       """, (doc_id,))
       result = cursor.fetchone()
       doc_version_id = result[0]

       par_version_id = insert_paragraph(connection, doc_version_id, par_id, md, par_hash)
       if par_version_id:
        if 'h' in json_data and json_data['h']:
         cache_data = json_data['h']
         if isinstance(cache_data, str):
          cache_data = {'hashkey': cache_data}
         elif isinstance(cache_data, dict):
```

```
            cache_data = {next(iter(cache_data)): cache_data[next(iter(cache_data))]}
           for cache_key, cache_value in cache_data.items():
            insert_html_cache(connection, par_version_id, cache_key, cache_value)
          if 'attrs' in json_data and json_data['attrs']:
           attrs_data = json_data['attrs']
           attrs_list = [(par_version_id, k, v) for k, v in attrs_data.items()]
           insert_par_attrs(connection, attrs_list)

         pbar.update(1)
        except Exception as e:
         print(f"Error processing JSON file {json_file} in paragraph folder {par_folder}: {e}")
   break
 pbar.close()

if __name__ == "__main__":
 dbname = "documentdb"
 user = "postgres"
 pwd = "root"
 host = "localhost"
 port = "5432"
 pg_dir = "/opt/timgradu/tim_files"

 print("Initializing import...")
 amount_of_docs = sum([len(files) for _, _, files in os.walk(pg_dir + '/docs')])
 # amount_of_pars = file_count = sum(1 for root, dirs, files in os.walk(pg_dir + '/pars') for file in
files if not file.startswith('current'))

 print("Starting import:")
 connection = connect_to_db(dbname, user, pwd, host, port)
 if connection is not None:
  # import_documents(pg_dir + "/docs", connection, amount_of_docs)
  # import_paragraphs(pg_dir + "/pars", connection, amount_of_pars)
  import_changelogs(pg_dir + "/docs", connection, amount_of_docs)
  print("Done importing document database.")
  connection.close()
  print("Database connection closed.")
```

LIITE 7 MONGODB :N IMPORT-SKRIPTI

```
import os
import pymongo
import json
from tqdm import tqdm
from threading import Thread
import concurrent.futures

def connect_to_db(url, db):
  client = pymongo.MongoClient(url)
  db = client[db]
  print("Connected to database")
  return db

def import_paragraphs(dbcon, directory, par_amount):
  try:
    par_collection = dbcon["Paragraph"]

    # Create indexes
    try:
      par_collection.drop_index([("par_id", pymongo.ASCENDING), ("t", pymongo.ASCENDING)])
    except Exception as e:
      print("exception on drop_index, probably didn't exist")
    par_collection.create_index([("par_id", pymongo.ASCENDING), ("t", pymongo.ASCENDING)])

    pbar_par = tqdm(total=par_amount, desc="Inserting paragraphs")

    for root, dirs, files in os.walk(directory):
      par_dir_path = os.path.join(root, "pars")
      for doc_folder in os.listdir(par_dir_path):
        doc_id = None
        if doc_folder.isdigit():
          doc_id = int(doc_folder)
        else:
          continue
        pars_folder_path = os.path.join(par_dir_path, doc_folder)
        for par_folder in os.listdir(pars_folder_path):
          par_folder_path = os.path.join(pars_folder_path, par_folder)
          par_to_insert = []
          if os.path.isdir(par_folder_path):
            for file in os.listdir(par_folder_path):
              if file == "current":
                continue
              file_path = os.path.join(par_folder_path, file)
              with open(file_path, "r") as f:
                json_data = json.load(f)
                data = {
                  "par_id": par_folder,
                  "doc_id": doc_id,
                  "md": json_data.get("md", None),
                  "t": json_data.get("t", None),
                  "attrs": json_data.get("attrs", {}),
                  "h": json_data.get("h", {})
                }

                par_to_insert.append({
                  **data
                })
          if par_to_insert:
            try:
              par_collection.insert_many(par_to_insert)
              pbar_par.update(len(par_to_insert))
            except Exception as e:
              print(f"Exception importing paragraphs (mongo insert): {e}")
      break
    pbar_par.close()
  except Exception as e:
    print(f"Exception in importing paragraphs: {e}")

def import_documents(dbcon, directory, doc_amount):
  try:
    doc_pbar = tqdm(total=doc_amount, desc="Inserting documents")
    par_collection = dbcon["Paragraph"]
```

```
    log_collection = dbcon["Changelog"]

    doc_collection = dbcon["Document"]

    par_collection.create_index([("par_id", 1)])
    par_collection.create_index([("par_id", 1), ("t", 1)])
    log_collection.create_index([("doc_id", 1)])

    for root, dirs, files in os.walk(directory):
      doc_dir_path = os.path.join(root, "docs")
      for doc_folder in os.listdir(doc_dir_path):
        doc_folder_path = os.path.join(doc_dir_path, doc_folder)
        doc_id = None
        if doc_folder.isdigit():
          doc_id = int(doc_folder)
        else:
          continue
        for doc_maj_folder in os.listdir(doc_folder_path):
          if (doc_maj_folder == "changelog"):
            continue
          doc_version_major = None
          if doc_maj_folder.isdigit():
            doc_version_major = int(doc_maj_folder)
          else:
            continue
          doc_maj_folder_path = os.path.join(doc_folder_path, doc_maj_folder)
          docs_to_insert = []
          for doc_min_file in os.listdir(doc_maj_folder_path):
            if doc_min_file.isdigit():
              doc_version_minor = int(doc_min_file)
            else:
              continue
            doc_min_file_path = os.path.join(doc_maj_folder_path, doc_min_file)
            doc_to_insert = process_version_file(dbcon, doc_min_file_path, doc_id, doc_version_major,
doc_version_minor, doc_pbar)
            if doc_to_insert is not None and isinstance(doc_to_insert, dict):
              docs_to_insert.append(doc_to_insert)

          if docs_to_insert:
            try:
              doc_collection.insert_many(docs_to_insert, ordered=False)
            except Exception as e:
              print(f"Error inserting docs_to_insert {docs_to_insert}: {e}")
            doc_pbar.update(len(docs_to_insert))
      break
  except Exception as e:
    print(f"Error importing documents: {e}")

def process_version_file(dbcon, doc_min_file_path, doc_id, doc_version_major, doc_version_minor,
doc_pbar):
  try:
    par_collection = dbcon["Paragraph"]
    log_collection = dbcon["Changelog"]
    data = None

    with open(doc_min_file_path, 'r') as f:
      lines_without_t = False
      identifiers = []
      for line in f:
        line = line.rstrip("\n")
        if ("/" not in line):
          lines_without_t = True
          par_id = line
          identifiers.append(par_id)
        else:
          par_id, t = line.split("/")
          identifiers.append((par_id, t))

      if lines_without_t:
        identifier_map = {(par_id): line_number for line_number, par_id in enumerate(identifiers)}
        ordered_ids = []
        cursor = par_collection.aggregate([
          {"$match": {"par_id": {"$in": [par_id for par_id in identifiers]}}},
          {"$group": {"_id": "$par_id", "doc": {"$first": "$$ROOT"}}},
          {"$project": {"_id": "$doc._id", "par_id": "$doc.par_id", "t": "$doc.t"}}
        ])
        for par in cursor:
          par_id = par["par_id"]
```

```
          line_number = identifier_map.get(par_id)
          if line_number is not None:
            ordered_ids.append({"_id": par["_id"], "pos": line_number})
      else:
        identifier_map = {(par_id, t): line_number for line_number, (par_id, t) in enumerate(identi-
fiers)}
        ordered_ids = []
        for par in par_collection.find({"doc_id": doc_id, "par_id": {"$in": [par_id for par_id, _ in
identifiers]}, "t": {"$in": [t for _, t in identifiers]}}, {"_id": 1, "par_id": 1, "t": 1}):
          par_id = par["par_id"]
          t = par["t"]
          line_number = identifier_map.get((par_id, t))
          if line_number is not None:
            ordered_ids.append({"_id": par["_id"], "pos": line_number})

      data = {
        "doc_id": doc_id,
        "doc_version_major": doc_version_major,
        "doc_version_minor": doc_version_minor,
      }

      if ordered_ids:
        data["paragraphs"] = ordered_ids

      changelog = log_collection.find_one({"doc_id": doc_id}, {"_id": 1})
      if changelog:
        data["changelog"] = changelog["_id"]

    return data
  except Exception as e:
    print(f"Exception in process_version_file: {e}")

def import_changelog(dbcon, directory):
  try:
    log_collection = dbcon["Changelog"]
    # Create indexes
    try:
      log_collection.drop_index("doc_id")
    except Exception as e:
      print("exception on drop_index, probably didn't exist")
    log_collection.create_index("doc_id")

    log_pbar = tqdm(desc="Importing changelogs")
    for root, dirs, files in os.walk(directory):
      doc_dir_path = os.path.join(root, "docs")
      for doc_folder in os.listdir(doc_dir_path):
        doc_folder_path = os.path.join(doc_dir_path, doc_folder)
        doc_id = None
        if doc_folder.isdigit():
          doc_id = int(doc_folder)
        else:
          continue
        for changelog_file in os.listdir(doc_folder_path):
          if (changelog_file != "changelog"):
            continue
          changelog_file_path = os.path.join(doc_folder_path, changelog_file)
          with open(changelog_file_path) as f:
            entries = []
            for line in f:
              if line is not None:
                try:
                  json_data = json.loads(line)
                  entries.append(json_data)
                except Exception as e:
                  print(f"Exception opening changelog file in doc_id: {doc_id}: {e}")

            if entries:
              data = {
                "doc_id": doc_id,
                "entries": entries
              }
              log_collection.insert_one(data)
              log_pbar.update(1)
      break
  except Exception as e:
    print(f"Exception importing changelogs: {e}")
    raise e
```

```
if __name__ == "__main__":
  url = "mongodb://localhost:27017/"
  db = "Documentdb"
  pg_dir = "/opt/timgradu/tim_files"

  print("Initializing import")
  #doc_amount = sum([len(files) for _, _, files in os.walk(os.path.join(pg_dir, 'docs'))])
  doc_amount = 3505111
  #par_amount = sum(1 for root, dirs, files in os.walk(os.path.join(pg_dir, 'pars')) for file in files
if not file.startswith('current'))
  dbcon = connect_to_db(url, db)

  print("Starting import:")
  #import_paragraphs(dbcon, pg_dir, par_amount)
  #import_changelog(dbcon, pg_dir)
  import_documents(dbcon, pg_dir, doc_amount)
  print("Import complete")
```

LIITE 8 SUORITUSKYKYTESTAUKSEN BENCHMARK-SKRIPTI

```
import psycopg2
import multiprocessing
import time
import statistics
import random

# NOTE: Test can only be run on one product at a time. Comment out the ones that aren't being tested.

def execute_transaction(stop_event, latency_list, doc_ids):
  try:
    # ------ POSTGRESQL CONNECTION -----
    db_connection = psycopg2.connect(
      dbname="documentdb",
      user="postgres",
      password="root",
      host="localhost",
      port="5432"
    )

    # ----- COCKROACHDB CONNECTION ------
    db_connection = psycopg2.connect("postgresql://root@localhost:26257/documentdb?sslmode=disable")

    # ----- MONGODB CONNECTION -----
    client = pymongo.MongoClient("mongodb://localhost:27017/", maxPoolSize=1)
    db = client["Documentdb"]
    collection = db["Document"]

    # ----- RDBMS TEST -----
    while not stop_event.is_set():
      with db_connection.cursor() as cursor:
        # T1
        doc_id = random.choice(doc_ids)

        # T2
        # doc_id = 1

        transaction_start = time.time()
        cursor.execute("""
          WITH par_list_rows AS (
            SELECT
            split_part(row_data, '/', 1) AS par_id,
            split_part(row_data, '/', 2) AS par_hash
            FROM (
            SELECT regexp_split_to_table(pl.list, E'\n') AS row_data
            FROM paragraph_list pl
            WHERE pl.doc_version_id IN (
              SELECT doc_version_id
              FROM document
              WHERE doc_id = %s
              ORDER BY doc_version_major DESC, doc_version_minor DESC
              LIMIT 1
            )
            ) AS rows
          )
          SELECT
            p.*,
            hc.cache,
            pa.attr_key,
            pa.attr_value
          FROM
            paragraph p
          LEFT OUTER JOIN
            html_cache hc ON p.par_version_id = hc.par_version_id
          LEFT OUTER JOIN
            par_attrs pa ON p.par_version_id = pa.par_version_id
          JOIN
            par_list_rows pr ON p.par_id = pr.par_id AND p.par_hash = pr.par_hash;
        """, (doc_id,))

        transaction_stop = time.time()
```

```
        # Converted to ms with * 1000
        latency_list.append((((transaction_stop - transaction_start) * 1000), transaction_stop))

    # ----- MONGODB TEST -----
    while not stop_event.is_set():
      # T1
      doc_id = random.choice(doc_ids)

      # T2
      doc_id = 1
      transaction_start = time.time()

      pipeline = [
        {"$match": {"doc_id": int(doc_id)}},
        {"$sort": {"doc_version_major": -1, "doc_version_minor": -1}},
        {"$limit": 1},
        {"$unwind": "$paragraphs"},
        {"$lookup": {
          "from": "Paragraph",
          "localField": "paragraphs._id",
          "foreignField": "_id",
          "as": "referenced_paragraph"
        }},
      ]

      res = collection.aggregate(pipeline)
      transaction_stop = time.time()

      # Convert to ms with * 1000
      latency_list.append((((transaction_stop - transaction_start) * 1000), transaction_stop))

  except Exception as e:
    print(f"Error executing transaction: {e}")
  finally:
    db_connection.close()

def calculate_throughput(latency_list, interval):
  throughput_intervals = []
  timestamps = [entry[1] for entry in latency_list]
  timestamps.sort()
  end_time = timestamps[-1]

  current_interval_start = timestamps[0]
  current_interval_end = start_time + interval

  interval_count = 0
  interval_index = 0

  for timestamp in timestamps:
    if timestamp < current_interval_end:
      interval_count += 1
    else:
      throughput_intervals.append((interval_index, interval_count, interval_count / interval))
      interval_count = 1
      current_interval_start = current_interval_end
      current_interval_end += interval
      interval_index += 1

  throughput_intervals.append((interval_index * interval, interval_count / interval))
  return throughput_intervals

if __name__ == "__main__":
  print("Initializing benchmark")

  # ---- TEST CONFIGURATION ----
  # AMOUNT OF PROCESSES I.E. "CLIENTS"
  num_processes = 12

  # TEST DURATION IN SECONDS
  test_duration = 900

  # INTERVAL OF CALCULATING TPS/x (seconds)
  tps_interval = 60

  # FILE NAME TO OUTPUT CALCULATED RESULTS TO
  res_filename = f"./postgres_t2.txt"

  # ---- INIT REQUIRED VARIABLES ----
```

```
  # Load all possible doc_id:s from file into memory
  with open("ids.txt", "r") as  file:
    lines = file.readlines()
  doc_ids = [line.strip() for line in lines]

  # Init stop signal to end testing
  stop_event = multiprocessing.Event()

  # Init multiprocessing list to store transaction latencies
  latency_list = multiprocessing.Manager().list()

  print("Starting benchmark")

  try:
    processes = []

    # Start testing
    start_time = time.time()

    # Start transaction processes
    for _ in range(num_processes):
      process = multiprocessing.Process(target=execute_transaction, args=(stop_event, latency_list,
doc_ids))
      process.start()
      processes.append(process)

    while time.time() - start_time < test_duration:
      time.sleep(1)

    # Stop testing
    stop_event.set()
    stop_time = time.time()

    # Join processes
    for process in processes:
      process.join()

    # Calculations
    if latency_list:
      # Because process.join waits for processes to finish instead of exiting ASAP, discard transacti-
ons that went over the stop_time
      latency_list = [(lat, timestamp) for lat, timestamp in latency_list if timestamp <= stop_time]

      total_transactions = len(latency_list)
      latencies = [value[0] for value in latency_list]
      avg_latency = sum(latencies) / total_transactions
      sdv_latency = statistics.stdev(latencies)
      avg_tps = total_transactions / test_duration
      throughputs = calculate_throughput(latency_list, tps_interval)

      with open(res_filename, "w") as f:
        print(f"Length of stress test: {test_duration}", file=f)
        print(f"Amount of clients: {num_processes}", file=f)
        print(f"Total amount of transactions: {total_transactions}", file=f)
        print(f"Average tps: {avg_tps} T/s", file=f)
        print(f"Average latency: {avg_latency:.2f} ms", file=f)
        print(f"Standard deviation of average latencies: {sdv_latency:.2f} ms", file=f)
        print(f"Throughputs data (timestamp, transactions in interval, t/s): {throughputs}", file=f)

      print("Done!")

  # In case test is interrupted via keyboard, do not calculate anything
  except KeyboardInterrupt:
    stop_event.set()
    for process in processes:
      process.join()
```