Tampereen yliopisto

Taneli Pulkkinen

TIETOKANNAN AUTOMAATTINEN JULKAISUNHALLINTAPROSESSI

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: David Hästbacka
Tarkastaja: Outi Sievi-Korte
Maaliskuu 2021

TIIVISTELMÄ

Taneli Pulkkinen: Tietokannan automaattinen julkaisunhallintaprosessi
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-tutkinto-ohjelma
3/2021

Tietokannoille käytetään selvästi sovelluksia harvemmin automaattista julkaisunhallintaprosessia. Tietokantamuutokset tehdään usein käsin suoraan tietokantoihin. Tämä aiheuttaa lukuisia ongelmia, kuten vaikeuksia ennakoida muutosten vaikutuksia, inhimillisten erehdysten aiheuttamia virhetilanteita, puutteellista dokumentaatiota ja tarvetta suurelle määrälle manuaalista testaustyötä.

Tämän diplomityön tavoitteena oli selvittää, mitä hyötyjä automaattisesta julkaisunhallintaprosessista on tietokannalle, ja miten tällainen prosessi toteutetaan. Tutkimusmuotona käytettiin konstruktiivista tutkimusta. Alan teoriaan ja kohdeyrityksen tarpeisiin tutustumisen jälkeen kehitettiin julkaisunhallintaprosessi, jota testattiin toteuttamalla se yhdelle tietokannalle. Kehitettyyn prosessiin kuuluu tietokannan versionhallinta, automaattitestaus sekä tietokantamuutosten automaattinen toimitus.

Työssä kehitetyn prosessin arvioitiin lisäävän tietokantajulkaisujen luotettavuutta ja tehokkuutta, vähentävän tuotannon virhetilanteita ja parantavan tietokannan dokumentaatiota. Prosessin käytännön toteutus onnistui kuitenkin vain osittain, jolloin saavutettiin hyötyjä julkaisujen luotettavuuteen ja tietokannan dokumentaatioon, mutta ei merkittävästi julkaisujen tehokkuuteen.

Avainsanat: tietokanta, julkaisunhallinta, julkaisu, automaatio, prosessi, konstruktiivinen tutkimus

ABSTRACT

Taneli Pulkkinen: Automatic database release management process
Master's thesis
Tampere University
Degree Programme in Information Technology, Msc (Tech)
3/2021

Automatic release management processes are much less commonly used with databases than with software applications. Changes are often done directly into databases by hand. This causes several issues, such as unpredictable effects from changes, errors caused by human mistakes, poor documentation, and a need for a large amount of manual testing.

The goal of this thesis was to investigate the benefits that an automatic release management process provides to a database, and how such a process is implemented. The research method was a case study. An automatic release management process was developed after investigating relevant studies and the needs of the target company, and this process was then tested by implementing it for one database. The designed process includes database version control, automatic testing, and automatic delivery.

The process developed in this study was estimated to improve the reliability and efficiency of database releases, as well as reducing the amount of errors in production and improving database documentation. The implementation of the process was only partially successful, however. It improved release reliability and database documentation but did not provide notable improvements to release efficiency.

Keywords: database, release, release management, automation, process, case study

ALKUSANAT

Tämä diplomityö tehtiin Tampereen Yliopiston tietotekniikan yksikköön, tosin työtä aloitettaessa yliopiston nimi oli vielä Tampereen Teknillinen Yliopisto. Työn kohdeyrityksenä toimi Enersoft Oy, joka on ollut työpaikkani opintojen ohella jo lähes neljän vuoden ajan. Aloittaessani diplomityötä tiesin tietokannoista lähinnä sen mitä yliopiston kursseilla oli opetettu. Työtä tehdessäni opin aiheesta paljon uutta, samoin kuin pitkän ja itsenäisen työprojektin tekemisestä.

Suuret kiitokset Timo Jokiselle diplomityön aiheesta ja ohjauksesta yrityksen puolella, sekä Enersoftin väelle yleisesti hienosta työympäristöstä. Työn ohjaamisesta yliopiston puolella ja sen tarkastamisesta kiitos apulaisprofessori David Hästbackalle. Kiitos myös työn toiselle tarkastajalle Outi Sievi-Kortelle.

Perheelle, ystäville ja avopuolisolleni Kristiina Koskelle lämpimät kiitokset kaikesta tuesta ja kannustuksesta opintojen aikana ja elämässä muutenkin. Selinalle ja Kaarinalle kiitokset motivoinnista diplomityön teossa. Kiitos myös roolipelikerho Excaliburille, jonka toiminnassa en ollut kovin aktiivinen, mutta josta löysin hyviä ystäviä ja hauskaa illanviettoa.

Tampereella, 04.03.2021

Taneli Pulkkinen

SISÄLLYSLUETTELO

KUVALUETTELO

1. JOHDANTO

Tietokanta on yleinen ja tärkeä ohjelmistokokonaisuuden osa, jonka tehtävä on säilöä tietoa. Oikein rakennettu tietokanta suojaa tiedon saatavuutta, eheyttä ja luottamuksellisuutta. Elinkaarensa aikana tyypillinen tietokanta muuttuu useita kertoja. Varsinkin ketterissä projekteissa tietokannan yksityiskohtainen suunnittelu ja toteutus on tapana tehdä pala kerrallaan projektin edetessä, ja seuraavia tietokannan osia toteutettaessa ilmenee monesti tarve tehdä muutoksia aiemmin rakennettuihin osiin [1]. Myös tietokantamuutoksille tarvetta luovat asiakasvaatimusten muutokset, jatkokehitystarpeet ja testauksessa havaitut virheet ovat ohjelmistoprojekteissa normaaleja. Tietokannasta syntyy siis sen elinkaaren aikana monia versioita.

Redgate Softwaren kyselyn [2] mukaan tietokantojen julkaisunhallinnan suosio on kasvussa, mutta se on selvästi jäljessä ohjelmistojen julkaisunhallinnasta. Tietokannan versionhallinta on käytössä joka toisella kyselyn vastaajalla, automaattitestit tietokantapäivityksille kuudesosalla ja automaattiset julkaisut neljäsosalla.

Puutteellinen tietokantojen julkaisunhallinta aiheuttaa lukuisia riskejä. Täsmällinen muutoshistoria on vaikea selvittää ilman versionhallinnan luomaa lokia. Testaamattomat muutokset lisäävät yllättävien virheiden määrää, ja virhetilanteista palautuminen on erityisen työlästä, jos tietokantaan viimeksi tehdyt muutokset eivät ole tarkasti tiedossa. Tietokantojen määrän kasvaessa manuaalinen päivitys vie yhä enemmän aikaa. Riskit ja vaivallisuus kasvattavat haluttomuutta tehdä hyödyllisiäkään päivityksiä tietokantaan. Tietokanta- ja sovellustiimien välille tarvitaan paljon kommunikaatiota ja yhteistyötä, missä on riski väärinymmärryksille ja hidastuksille. Pahimmillaan tietokannasta muodostuu sovelluskehitykselle pullonkaula.

1.1 Diplomityön tavoitteet ja rajaukset

Tämän diplomityön tarkoitus on kehittää ja evaluoida ratkaisumalli, jolla tietokannat voidaan yhdistää yrityksen olemassa olevaan julkaisunhallintaan. Mallia käyttäen tietokantamuutosten kehittäminen ja julkaisu tehdään mahdollisimman automaattiseksi. Tutkimuksen tavoitetilassa muutokset tietokantoihin ovat hallittuja, tehokkaita, luotettavia ja ennakoitavia. Tietokantapäivitysten julkaisu on nopeampaa ja manuaalinen ihmistyö vähenee. Dokumentaatio kannasta ja sen muutoshistoriasta on paremmin saatavilla. Tietokannan häiriöt ovat harvinaisempia ja niistä palaudutaan nopeammin.

Selvitettävät tutkimuskysymykset ovat:

- Miten tietokannalle toteutetaan automaattinen julkaisunhallintaprosessi?
- Mitä hyötyjä saavutetaan tietokannan automaattisella julkaisunhallinnalla, verrattuna manuaaliseen julkaisunhallintaan?

Tutkimus tehdään Enersoft Oy:lle, joka on tamperelainen vuonna 1992 perustettu ohjelmistoyritys. Ratkaisusta pyritään kehittämään yritykselle yleisesti hyödyllinen, niin että sitä voidaan hyödyntää mahdollisimman moneen yrityksen tietokantaan. Tutkimuksessa tapausesimerkkinä käytetään yrityksessä pitkään käytössä ollut tietokantaa, johon muutokset on toistaiseksi julkaistu manuaalisesti.

Tutkimuksen rajoituksena on, että kehitettävä prosessi suunnitellaan kohdeyritystä vastaaviin olosuhteisiin. Prosessilla tehdään julkaisuja vain SQL-relaatiotietokannoille. Versionhallintana toimii Git, jota käytetään Bitbucket-sivustolla. Julkaisut eri palvelimille tehdään Octopus Deploy-julkaisupalvelun kautta. Nämä järjestelmät ovat yrityksessä jo laajassa käytössä, eikä niiden rinnalle toivota toista toteutusta. Lisäksi kehitettävä ratkaisumalli ei saa heikentää tietokantojen tietoturvaa, aiheuttaa tiedon menetystä, lisätä virhetilanteiden määrää tai muuten tuottaa haitallisia sivuvaikutuksia tietokantojen toimintaan.

1.2 Konstruktiivinen tutkimusote

Tutkimusmenetelmänä käytetään konstruktiivista tutkimusta [3]. Menetelmässä ratkotaan käytännön ongelmia luomalla uusia konstruktiota, jotka voivat olla esimerkiksi malleja, suunnitelmia tai kaupallisia tuotteita. Tavoitteena on, että konstruktio ratkaisee käsiteltävän tosielämän ongelman. Samalla konstruktio joko luo uutta teoriaa tai testaa, jalostaa tai havainnollistaa olemassa olevaa teoriaa. Tässä tutkimuksessa luotava konstruktio on automaattinen tietokantojen julkaisunhallintaprosessi.

Konstruktiivinen tutkimusote on case-tutkimuksen alalaji, jossa pyritään ratkaisemaan tosielämän ongelmia luomalla uusia konstruktioita. Konstruktio tarkoittaa mitä tahansa ihmisen luomaa ideaa tai asiaa, vastakohtana kaikelle valmiiksi olemassa olevalle. Konstruktiot keksitään ja kehitetään, niitä ei löydetä. Konstruktiivisella tutkimuksella siis luodaan jotain uutta, rakennelma, jota ei ennen ollut olemassa. Konstruktioita ovat esimerkiksi uudet kaupalliset tuotteet, suunnitelmat, tietojärjestelmät ja toimintamallit. [3]

Konstruktiivinen tutkimusote on luonteeltaan käytännönläheinen ja kokeellinen. Tutkimuksen aikana tutkija ja käytännön edustajat työskentelevät läheisesti yhdessä ja oppivat kokemuksen kautta. Tutkimuksen aikana suunnitellaan innovatiivinen uusi konstruktio ja toteutetaan se käytännössä, minkä toivotaan ratkaisevan tutkimuksen kohteena oleva tosielämän ongelma. Täten tutkimukseen kuuluu tutkijan voimakas vaikutus tutkit-

tavaan tilanteeseen. Tutkimuksen tuotoksena syntyy luodun konstruktion lisäksi teoreettista tietoa, jota saadaan analysoimalla konstruktiota sekä tutkimusprosessia kokonaisuutena. [3]

Konstruktiivinen tutkimusprosessi alkaa, kun on tunnistettu sopiva ongelma ja määritetty tutkimuskysymykset, joiden ratkaisemiseen käytännön konstruktion luomisen arvioidaan olevan paras keino. Tutkimuskysymysten tulee olla sellaisia, joiden ratkaisemisesta on sekä käytännöllistä että teoreettista hyötyä. Prosessi alkaa sopimalla tutkimusprojektin toteutuksesta ongelmaan ratkaisua kaipaavan kohdeorganisaation kanssa. Tämän jälkeen tutkija kerää tietoa lähtötilanteesta. Tähän kuuluu alan teoriaan, olemassa oleviin ratkaisuihin ja kohdeorganisaatioon tutustumista. Muodostettuaan ymmärryksen lähtötilanteesta tutkija kehittää yhdessä kohdeorganisaation omien asiantuntijoiden kanssa ratkaisumallin, määrittelyn siitä millainen konstruktio ratkaisisi ongelman ja saavuttaisi tutkimuksen tavoitteet. Määrittelyn mukainen ratkaisu toteutetaan ja otetaan käyttöön, jolloin sen toimivuus saadaan testattua. Toteutuksen ja sen testauksen valmistuttua tutkija arvioi kehitetyn ratkaisun soveltamismahdollisuuksia, eli kuinka laajasti ja millaisilla muutoksilla sitä voisi siirtää toisiin tilanteisiin ja organisaatioihin. Lopuksi tutkija tarkastelee tutkimuksen tuloksia suhteessa aiempaan tietoon ja analysoi projektin tuoman lisän alan teoriaan. [3]

Konstruktiivisessa tutkimuksessa on tärkeää huolehtia, että käytännöllisen ongelmanratkaisun lisäksi kiinnitetään huomiota työn tieteellisyyteen ja teoreettiseen kontribuutioon. Tavoitteena on saada uutta tietoa, ei pelkästään ratkaista käytännön ongelmaa valmiiksi tunnetuilla ja laajasti testatuilla menetelmillä. [3]

1.3 Diplomityön rakenne

Diplomityö sisältää yhteensä seitsemän lukua. Luvussa 2 esitellään tietokantojen ja julkaisuhallinnan teoriaa. Luvussa 3 käydään läpi ohjelmistotyökaluja, joilla tietokantojen julkaisunhallinta voidaan toteuttaa. Luku 4 selostaa kehitetyn tietokantojen julkaisunhallintaprosessin. Lisäksi luvussa määritellään tavoitteet, joiden toteutumisen pohjalta kehitettävä ratkaisu arvioidaan. Luku 5 esittelee tapausesimerkkinä käytettävän tietokannan ja kuvaa julkaisunhallintaprosessin toteutuksen siihen. Luvussa 6 tarkastellaan toteutuksen tuloksia ja arvioidaan tutkimuksen tavoitteiden toteutumista. Lopuksi luku 7 sisältää työn yhteenvedon.

2. TIETOKANTA

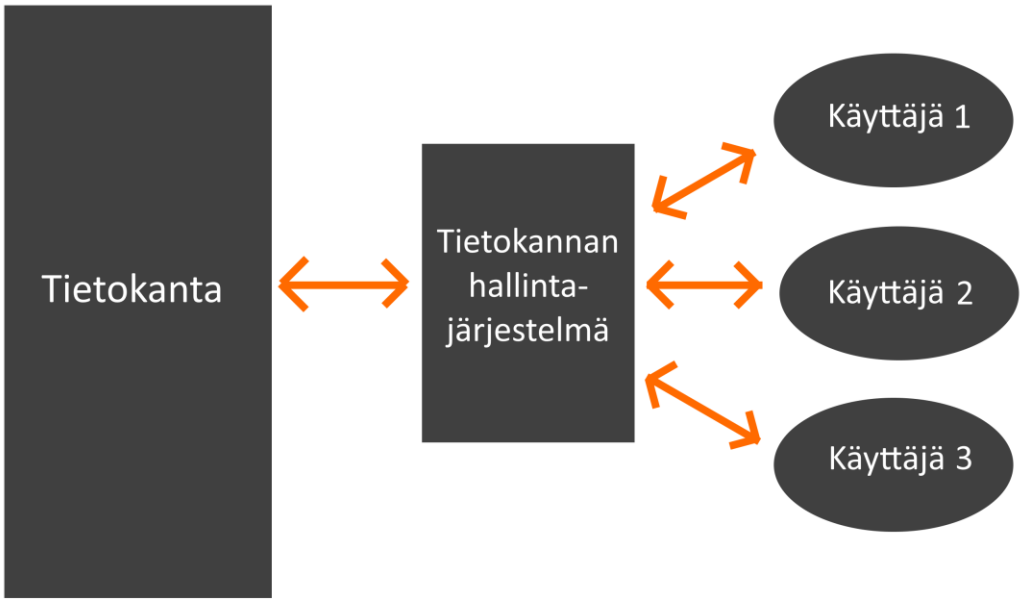
Tässä luvussa tutustutaan tietokantojen teoriaan. Osiossa 2.1 kuvataan yleisesti tietokannan toiminnan perusteet. Osiossa 2.2 selitetään tietokantojen kaksi päätyyppiä. Osiossa 2.3 käydään läpi tietokannan versionhallintaa sekä eri vaihtoehtoja sen toteutukseen. Osio 2.4 kertoo tietokannan jatkuvasta integraatiosta. Osio 2.5 selostaa tietokannan jatkuvan julkaisun vaiheet.

2.1 Tietokannasta yleisesti

Tietokanta on organisoitu elektroninen kokoelma tietoa. Tietokannan rakenteen on tarkoitus tukea tiettyä tarkoitusta varten kootun tiedon tehokasta tallentamista, hakua, muokkausta ja poistoa. Tieto säilötään tietokantaan tavallisimmin tietokokonaisuuksina eli tietueina, jotka sisältävät ryhmän ominaisuuksia [4]. Esimerkiksi kaupan tuotetietokannassa jokainen tuote on oma tietueensa, jonka ominaisuuksiin voi kuulua muun muassa nimi ja hinta.

Tietokannat tarjoavat useita merkittäviä etuja tiedon käsittelyyn. Näitä ovat muun muassa tiedon nopea saatavuus useille eri ohjelmille samanaikaisesti, suurten tietomäärien säilöminen verrattain pieneen määrään muistia, täsmällinen tietojen suojaus ja käyttöoikeuksien kontrolli sekä suurten tietomäärien ja monimutkaisesti toisistaan riippuvien tietojen hallinta [5, pp. 1-3]. Toimiva ja oikein rakennettu tietokanta tukee tietoturvan peruspilarien eli tiedon saatavuuden, luottamuksellisuuden ja eheyden ylläpitoa.

Tietokannat ovat äärimmäisen yleisiä. Tarkka määrä muuttuu jatkuvasti ja sitä on mahdoton arvioida, mutta se on varmasti vähintään miljoonia. Erityisen suosittuja tietokannat ovat yrityksissä, kunnissa ja muissa organisaatioissa, mutta myös yksityishenkilöiden tietokantoja on paljon. Tietokantoja voidaan luoda ja ottaa käyttöön nopeasti, ja niitä voidaan ajaa pienitehoisillakin tietokoneilla. Suuria tietomääriä säilöttäessä tietokantoja pidetään tavallisesti palvelimilla eli tietokoneilla, joissa on erityisen suuri määrä muistia.



***Kuva 1**: Tietokanta ja hallintajärjestelmä.*

Tietokantaa käytetään tyypillisesti tietokannan hallintajärjestelmänä (database management system) tunnetun ohjelmiston kautta. Kuvan 1 mukaisesti hallintajärjestelmä tarjoaa käyttöliittymän ja rajapinnan tietokannan käyttöön, ylläpitoon ja hallinnointiin [6]. Käyttäjät eivät ota suoraa yhteyttä tietokantaan, vaan välittävät käskyt hallintajärjestelmän kautta. Hallintajärjestelmällä voidaan esimerkiksi muuttaa tietokannan rakennetta, tehdä tietokantaan kyselyjä ja säätää tietokannan käyttöoikeuksia [5, pp. 4-5].

Tietokanta ja sen hallintajärjestelmä esiintyvät käytännössä aina yhdessä. Ne ovat niin läheisesti toisiinsa sidottuja, että kokonaisuutta kutsutaan arkikielessä usein yksinkertaisesti tietokannaksi. Esimerkiksi MySQL-hallintajärjestelmän kautta käytettävää tietokantaa kutsutaan MySQL-tietokannaksi.

2.2 Relaatio- ja NoSQL-tietokanta

Tietokannat voidaan jakaa sisäisen rakenteensa pohjalta kahteen luokkaan: relaatiotietokantoihin ja relaatiomallista poikkeaviin tietokantoihin. Relaatiotietokantoja kutsutaan usein SQL-kannoiksi ja muita kantoja NoSQL-kannoiksi (Not only SQL). Näistä relaatiotietokannat ovat pitkään olleet ylivoimaisesti laajemmassa käytössä. Viime vuosina NoSQL-kannat ovat kuitenkin yleistyneet, ja ero tyyppien välisessä suosiossa on pienentynyt [7]. Suosittuja relaatiotietokantoja ovat esimerkiksi Oracle Database ja MySQL, NoSQL-kantoja puolestaan MongoDB ja Redis [8].

Relaatiotietokantojen rakenne pohjautuu matematiikan joukko-oppiin ja relaatiomalliin. Relaatiotietokanta koostuu tietoalkioiden muodostamista kokonaisuuksista, relaatioista. Relaatiot esitetään tyypillisesti tauluina, joissa jokainen tietoalkio on oma sarakkeensa. Taulu määrittelee siis tietyn tietokokonaisuuden, ja jokainen sarake on tämän kokonaisuuden ominaisuus. Taulun rakenne määritellään tarkasti taulua luotaessa, jokaisella sarakkeella on oma tietotyyppinsä sekä mahdollisesti muita rajoitteita kuten minimi- ja maksimiarvot. Tieto tallennetaan tauluihin monikoina, jotka kuvataan taulun riveinä. Eri taulujen monikoiden väliset suhteet määritellään myös täsmällisesti, linkittäen eri taulujen sarakkeita toisiinsa vierasavaimiksi. Toiston välttämiseksi asiakokonaisuudet monesti pilkotaan useaan eri tauluun, jotka yhdistetään toisiinsa vierasavaimilla. [5, pp. 12-18]

NoSQL-kantoja on useita eri tyyppejä, sillä niihin kuuluvat kaikki tietokannat, jotka eivät käytä relaatiomallia. Yleisiä rakenteita ovat esimerkiksi dokumenttitietokannat ja graafitietokannat. [9]

Dokumenttikannat tallentavat tietoa kokonaisuuksina, joiden ominaisuuksia ja suhteita ei tarvitse määritellä yhtä täsmällisesti kuin relaatiokannoissa [9]. Samantyyppisilläkin dokumenteilla voi olla keskenään eri ominaisuuksia. Dokumenttien sisältö ei kuitenkaan ole täysin vapaamuotoista, vaan sillä on sisäinen rakenne, esimerkiksi nimetyt ja toisistaan erotetut ominaisuudet. Dokumenttikannat sopivat hyvin puolirakenteellisen ja rakenteeltaan muuttuvan tiedon säilömiseen [10].

Graafitietokannat muistuttavat dokumenttikantoja, mutta itse dokumenttien lisäksi niissä säilötään myös tieto eri dokumenttien välisistä suhteista ja näiden suhteiden ominaisuuksista [9]. Niiden tarkoitus on käsitellä tehokkaasti tietoa, joka on vahvasti linkitettyä. Esimerkiksi verkkoyhteisöpalvelut ja suosituksia luovat ohjelmistot käyttävät tällaista tietoa [10].

Relaatio- ja NoSQL-kantojen merkittävimpiin eroihin kuuluu joustavuus tiedon rakenteessa [10]. Relaatiotietokannassa tieto säilötään tarkasti etukäteen määriteltyihin tauluihin, joissa on tietyt sarakkeet ja jokaisella sarakkeella omat vaatimuksensa ja rajoitteensa säilöttävälle tiedolle. Jokaisen tietueen pitää sopia näihin tauluihin ja sarakkeisiin. Toisteisuuden välttämiseksi ja tallennustilan säästämiseksi samojen tietoalkioiden säilömistä useaan paikkaan pyritään välttämään, minkä seurauksena asiakokonaisuudet jaetaan usein moneksi tauluksi. NoSQL-kantoihin sen sijaan säilötään usein puolirakenteellista tai rakenteetonta tietoa, ja samaan kokoelmaan voidaan tallentaa keskenään eri ominaisuuksia sisältäviä dokumentteja tai muita kokonaisuuksia [9].

Toinen olennainen ero tietokantatyyppien välillä on skaalautuvuus. Relaatiomallisen tietokannan skaalaaminen tapahtuu tyypillisimmin vertikaalisesti eli kasvattamalla tietokannan kokoa ja tehokkuutta yhdellä palvelinkoneella, esimerkiksi päivittämällä palvelinkoneen laitteistoa. NoSQL-kanta sen sijaan skaalautuu horisontaalisesti eli tietokanta voidaan hajauttaa usealle palvelimelle. Relaatiotietokannan jakaminen usealle palvelinkoneelle on haastavaa, koska relaatiokannassa eri tauluista löytyy toisiinsa vahvasti sidottua tietosisältöä, eikä useimpia relaatiotietokantoja ole suunniteltu käsittelemään hajautetussa järjestelmässä toisinaan syntyvää tietosisällön yhtenäisyyden puutetta. NoSQL-kannoissa sen sijaan tieto säilötään itsenäisempinä kokonaisuuksina, ja tietokantajärjestelmät on suunniteltu tukemaan hajautettua rakennetta uhraamalla yhtenäisyyttä saatavuuden parantamiseksi. Hajautettu rakenne voi lisätä tietokannan tehokkuutta valtavasti, sillä jokainen palvelin voi käsitellä tietokantakutsuja samaan aikaan. Haittapuolena hajautettu tietokanta voi hetkellisesti päätyä epäyhtenäiseen tilaan, jossa kannan eri osissa on keskenään ristiriitaista tietoa. [9]

2.3 Tietokannan versionhallinta

Tietokannan rakentaminen ja ylläpito on prosessi, jonka aikana tietokanta muuttuu jatkuvasti. Erityisesti ketterissä ohjelmistoprojekteissa korostetaan muutokseen reagoimista [11] ja yksityiskohtainen suunnittelu ja toteutus tehdään usein pala kerrallaan projektin edetessä [12]. Riippumatta ohjelmistoprojektissa käytetyistä menetelmistä tarve tehdä muutoksia tietokantaan ilmenee usein jossain kohdin ohjelmiston elinkaarta. Kehitysprosessin aikana sekä sovelluksiin että tietokantoihin lisätään uusia ominaisuuksia ja muokataan jo toteutettuja osia. Julkaisun jälkeen muuttuvat vaatimukset, toteutettujen ohjelmistojen jatkokehitys ja ylläpidossa paljastuvat muutostarpeet ovat yleisiä ohjelmistokehityksen osia.

2.3.1 Tietokannan ja sovellusten versionhallinnan erot

Sovellusten kehityksessä versionhallinta on laajasti käytetty tapa hallita muutoksia, mutta tietokannoilla tämä on harvinaisempaa. Redgaten raportin mukaan [2] 83% yrityksistä käyttää versionhallintaa sovellusten kehityksessä, mutta tietokantojen kehityksessä näin tekee vain 55% yrityksistä.

Versionhallinnalla on useita merkittäviä hyötyjä yleisessä ohjelmistonkehityksessä. Versionhallintajärjestelmän avulla kehittäjät voivat käydä läpi ohjelmiston yksityiskohtaisen muutoshistorian, selvittäen kuka, miksi ja milloin on tehnyt minkäkin muutoksen. Ohjelmisto voidaan palauttaa ohjelmiston aiempaan versioon, mikäli uudesta versiosta löytyy virheitä. Usean henkilön yhteistyö ja samanaikainen kehitys helpottuu, sillä kehittäjät voivat samanaikaisesti muokata samaa tiedostoa sekä ratkaista helpommin päällekkäisten

muutosten aiheuttamia ristiriitoja. Versionhallintajärjestelmä voi toimia myös varmuuskopiona vikatilanteissa. Näin versionhallinta parantaa ohjelmiston kehitysprosessin vastuullisuutta, tehostaa kehittäjien yhteistyötä ja nopeuttaa kriisitilanteista toipumista. [13]

Tietokannan versionhallinta on haastavampaa ja rajallisempaa kuin sovelluksen, koska tietokanta koostuu useista osista, joita täytyy hallita eri tavoin. Näitä osia ovat tietokannan rakenne eli skeema, käytön aikana syntyvä liiketoimintatieto, tietokannan eri asetustiedostot sekä sovellusten vaatima alustustietosisältö. Sovellus voidaan koota uudelleen jokaisen muutoksen jälkeen, mutta tietokanta sisältää liiketoimintatietoa, jonka menettäminen päivityksessä ei ole hyväksyttävää. Tätä tietoa on tyypillisesti suuri määrä ja se muuttuu jatkuvasti, jolloin sen kopioiden ja muutosten jatkuva säilöminen versionhallintaan ei ole kannattavaa. Niinpä liiketoimintatieto on parempi suojata vikatilanteilta versionhallinnan sijaan säännöllisillä varmuuskopioilla. Versionhallintaan on suositeltavaa säilöä kaikki muut osat, jotta tietokannasta voidaan luoda suoraan toimivia kopioita eri ympäristöihin. Kuitenkin täytyy huomioida, että tietokannan asetukset ovat ympäristöstä riippuvaisia. Samoja asetuksia ei voida käyttää kaikissa ympäristöissä. [14]

Versionhallinnan sopivuus ja käyttötapa riippuu tietokannan tyypistä. Monissa NoSQL-kannoissa ei ole lainkaan tietokannassa määriteltyä skeemaa, johon sopimattoman tiedon säilöminen ja käsittely olisi estetty. Tällöin skeeman määritelmä sijaitsee tietokantaa käyttävissä sovelluksissa ja niiden tavassa tallentaa ja käsitellä kannan tietosisältöä [15]. Tietokannan skeemalle ei siis tarvita omaa versionhallintaa, vaan se on luonnostaan osa sovelluksia ja niiden versionhallintaa. Skeeman muutokset syntyvät implisiittisesti sovelluksen muutoksista. Toisissa tietokannoissa, esimerkiksi SQL-kannoissa, puolestaan skeema on määritelty erikseen itse kannassa ja tiedon sopivuutta siihen valvotaan. Tällöin tietokannan skeemaa ei voida muuttaa jo olemassa olevaan tietoon sopimattomilla tavoilla, ellei samalla määritellä miten nykyistä tietosisältöä tulee muuttaa, jotta se sopii uuteen skeemaan. Samoin tietokantaan ei voi tallentaa tietoa, jonka rakenne ja arvot eivät vastaa määriteltyä skeemaa. Näissä tapauksissa skeemasta täytyy pitää kirjaa erillään sovelluksista, ja muutokset siihen joudutaan tekemään eksplisiittisesti. Nämä tietokannat vaativat oman, sovelluksista erillisen skeeman hallinnointinsa ja hyötyvät siten omasta versionhallinnastaan [14].

Riippuen sitä käyttävistä sovelluksista, tietokanta voidaan säilöä versionhallinnassa omaan repositorioonsa tai samaan repositorioon sitä käyttävän sovelluksen kanssa. Jos tietokanta ei ole tarkoitettu tietyn yksittäisen sovelluksen käytettäväksi, oma repositorio on selkeä valinta. Näin on selvää, että tietokanta on oma komponenttinsa ja sitä voidaan kehittää joustavasti irrallaan eri sovellusten koodista [14]. Sen sijaan, jos tietokanta on olemassa yhtä sovellusta varten, yhteinen repositorio parantaa tietokannan ja sovelluksen synkronointia [16]. Yhdessä versionhallintaan säilötyllä sovelluksella ja tietokannalla

on aina selvää, mitkä versiot on tarkoitettu yhdessä toimiviksi. Samoin molemmille voidaan määrittää yhteinen julkaisuputki (release pipeline), joka huolehtii, että kumpikin päivitetään samanaikaisesti [17].

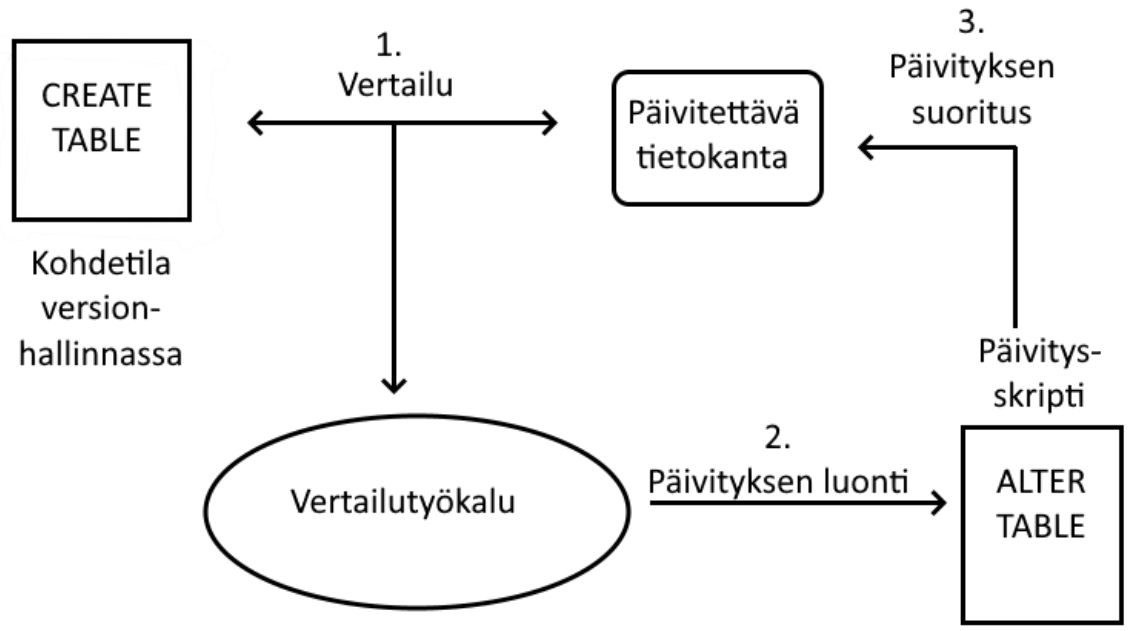
Tarve huolehtia tietokannan säilyvyydestä hankaloittaa versionhallinnan käyttöä. Sovelluskoodin jokainen versio on itsenäinen kokonaisuus, joka määrittelee, kuinka ohjelman nykyisen version kuuluu toimia. Sovelluspäivitykset tehdään tyypillisesti poistamalla edellinen asennettu versio ja asentamalla uusi versio puhtaalta pöydältä. Tuotantoon asennettua tietokantaa päivitettäessä sen sijaan kantaa ei haluta purkaa ja rakentaa uudelleen, vaan päivitykset suoritetaan olemassa olevien rakenteiden muutoksina.

Esimerkiksi SQL-tietokantaan luodaan uusi taulu komennolla CREATE TABLE, mutta olemassa olevaa taulua muokataan komennolla ALTER TABLE. Taulun muokkaukseen käytetään siis eri komentoa kuin taulun luomiseen. Tietokantaan tehtävä päivitys ei siis ole itsenäinen kokonaisuus kuten sovelluksen lähdekoodin uusi versio, vaan sen täytyy ottaa huomioon kannan edellinen tila.

Tietokantaa ei päivitetä suorittamalla kannan lopputilan määritteleviä komentoja, vaan kannan olemassa olevaa rakennetta muokkaavia komentoja. Tästä seuraa haasteita, varsinkin kun tietokanta on asennettu useaan eri ympäristöön ja siihen tekee muutoksia useampi kuin yksi kehittäjä samaan aikaan. Ratkaisuna tähän ongelmaan on olemassa kaksi lähestymistapaa tietokannan versionhallintaan. [18]

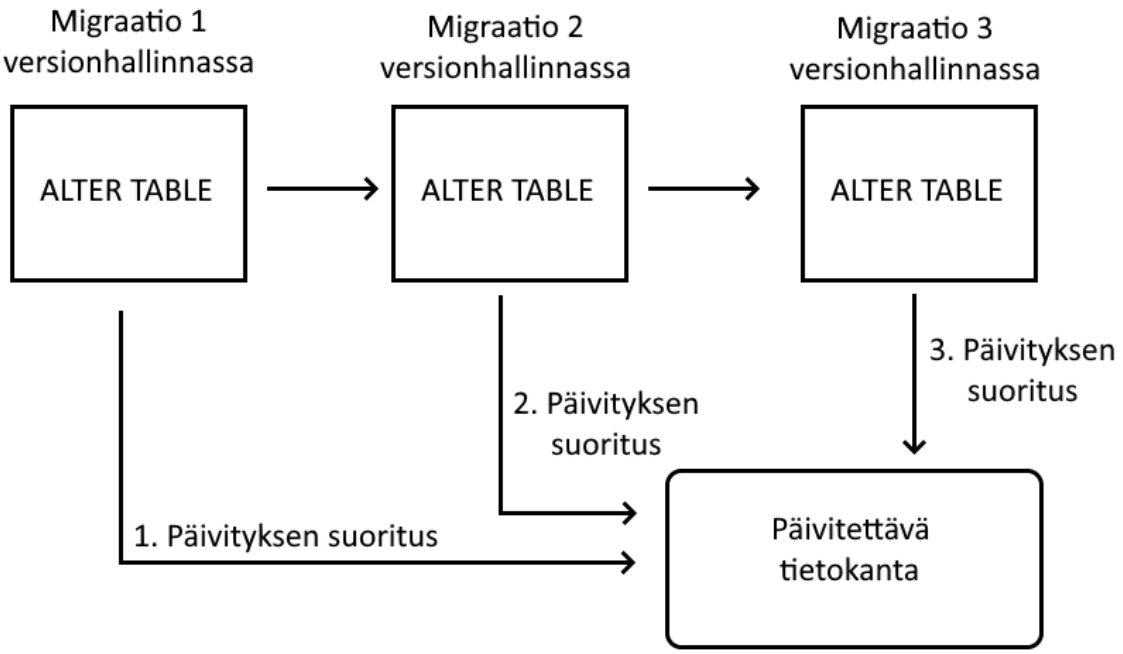
2.3.2 Tila- ja migraatiopohjaiset lähestymistavat

Tietokantapäivitys alkaa määrittelemällä kohdetila, johon tietokanta halutaan saada. Tämän jälkeen luodaan komennot, joilla kanta saadaan muutettua kohdetilaan eli migraatioskripti. Tietokannan päivittäjä voi kirjoittaa migraatioskriptin itse, tai hän voi käyttää automaattista työkalua, joka luo migraation vertaamalla tietokannan kohdetilaa ja nykytilaa. Nämä ovat kaksi päätapaa päivittää tietokanta, ja tästä seuraa myös kaksi päätapaa tietokannan versionhallintaan. [19]



***Kuva 2**: Tilapohjainen tietokannan versionhallinta*

Tilapohjaisessa (state based) tavassa kehittäjä määrittää halutun lopputilan ja säilöö sen versionhallintaan. Esimerkiksi SQL-kannan taulua muokatessa kehittäjä lisää alkuperäiseen CREATE TABLE komentoon ylimääräisen sarakkeen. Päivityksiä suoritettaessa automaattinen vertailutyökalu vertaa kuvan 2 mukaisesti päivitettävän tietokannan todellista tilaa versionhallinnassa määritettyyn kohdetilaan ja luo migraatioskriptin, joilla tietokanta saadaan päivitettyä kohdetilaan. Esimerkkitapauksessa vertailutyökalu luo tarvittavan ALTER TABLE komennon. Tämä komento voidaan sitten ajaa päivitettävään tietokantaan, jolloin siihen saadaan tehtyä haluttu muutos. [20]



***Kuva 3**: Migraatiopohjainen versionhallinta.*

Migraatiopohjaisessa (migration based) tavassa kehittäjä kirjoittaa suoraan halutut muutokset toteuttavan migraatioskriptin ja säilöö sen versionhallintaan. Päivitykset suoritetaan kuvan 3 mukaisesti ajamalla nämä migraatioskriptit suoraan kohdetietokantaan, jolloin se päivittyy haluttuun tilaan. Esimerkiksi taulua muokatessa kehittäjä luo tarvittavan ALTER TABLE komennon itse. Jokainen skripti on tyypillisesti oma tiedostonsa, ja niiden suoritusjärjestys määrätään esimerkiksi tiedostojen nimien perusteella. Tällöin on tyypillistä aloittaa skriptien nimet numeroilla tai päivämäärillä, kuten ”0001_AlustaTietokanta” tai ”2020-02-13_LisääIndeksit”. [18]

Migraatiopohjainen lähestymistapa tarjoaa kehittäjille enemmän manuaalista kontrollia. Migraatioskripti kirjoitetaan itse ja suoritetaan tietokantaa päivitettäessä sellaisenaan, joten päivityksen laatu ja toimivuus on migraation kirjoittajan hallinnassa. Tilapohjaisessa lähestymistavassa sen sijaan käytetään migraation luomiseen automaattista työkalua, joka saattaa luoda virheellisiä tai tehottomia skriptejä. [20]

Tyypillinen esimerkki on tietoa sisältävän taulun uudelleennimeäminen. Migraatiopohjaisessa tavassa ihminen kirjoittaa suoraan skriptin, joka vaihtaa taulun nimen. Tilapohjaisessa tavassa ihminen määrittelee uuden tilan, jossa on eri niminen taulu. Määritellyn tilan perusteella automaattinen työkalu luo varsinaisen tietokantaan ajettavan migraatioskriptin. Tämä työkalu voi tulkita, että sen kuuluu luoda uusi tyhjä taulu ja poistaa vanha taulu sisältöineen. Tällöin tietokannan päivittäjän täytyy manuaalisesti tarkistaa automaation luoma skripti ja korjata se. Ongelmaa pahentaa, että lopulliset skriptit luodaan vasta, kun työkalu yhdistää päivitettävään tietokantaan ja vertaa sitä tavoitetilaan.

Tietokantaan ajettavia skriptejä ei siis tallenneta versionhallintaan, missä ne olisivat koko kehitystiimin saatavilla ja pysyvästi katselmoitavissa. Toisaalta migraatiopohjaisessa tavassa on aina riski, että kehittäjä tekee virheen ja luo sopimattoman skriptin. Lisätty kontrolli tarkoittaa myös lisättyä vastuuta. [19]

Toinen migraatiopohjaisen tavan vahvuus on kyky käsitellä kaikki muutokset, joille vain voidaan kirjoittaa SQL-skripti. Tämä mahdollistaa erityisesti tietosisällön muunnokset, joita ei voi tehdä tilapohjaisesti määrittelemällä tietokannan uutta skeemaa. Rajoitus voidaan kiertää tilapohjaisessa tavassa liittämällä päivityksiin erillisiä käsin kirjoitettuja skriptejä, jotka suoritetaan automaattista päivitystä ennen tai sen jälkeen. Migraatiopohjainen toteutus on kuitenkin yksinkertaisempi, sillä kaikki muutokset voidaan käsitellä samanlaisina migraatioskripteinä ilman erityiskohtelua osalle muutoksista. [20]

Tilapohjainen menetelmä on yksinkertaisempi ja turvallisempi usean kehittäjän tehdessä muutoksia samanaikaisesti. Tilapohjaisia muutoksia on helppo vertailla keskenään versionhallinnassa ja yhdistää muutokset sovelluskoodin tapaan. Migraatiopohjainen tapa puolestaan vaatii tarkkuutta skriptien järjestyksen ja sisällön kanssa. Koska skriptit sisältävät muutoksia edelliseen tilaan eivätkä suoraa halutun lopputilan kuvausta, skriptin kirjoittajan täytyy tietää mikä tietokannan lähtötila on skriptin suorituksen alussa. Usean kehittäjän kirjoittaessa migraatioskriptejä samanaikaisesti voi syntyä tilanteita, joissa kehittäjät eivät tiedä missä järjestyksessä heidän skriptinsä tullaan suorittamaan ja mikä kannan lähtötila on minkäkin skriptin alussa. Ongelma kasvaa erityisen vakavaksi, kun tietokantaan kehitetään useita suuria muutoksia eri haaroissa. Haarojen yhdistämisen aikaan joudutaan käymään manuaalisesti läpi kaikki lisätyt muutosskriptit ja rakentamaan niistä toimiva kokonaisuus. Skriptien läpikäynnissä ei voida käyttää versionhallinnan tavallista muutosten vertailua kuten tilapohjaisilla muutoksilla, koska skriptin sisältö ei ole uusia versioita alkuperäisistä luontikomennoista vaan erillisiä muokkauskomentoja. Täten ristiriidat ja toisiaan haittaavat skriptit eivät paljastu nopealla vilkaisulla, joten kommunikaatio toisiinsa vaikuttavia muutoksia tekevien kehittäjien välillä ja muutosskriptien huolellinen läpikäynti ovat tärkeitä vikatilanteiden välttämiseksi. Tilapohjaiseen lähestymistapaan ei kuulu migraatioskriptien kirjoittaminen käsin ja säilöminen versionhallintaan, joten koko ongelma vältetään. [19]

Tyypillinen esimerkki ristiriitaisista migraatioista on tilanne, jossa kaksi kehittäjää tekee muutoksia samaan proseduuriin kommunikoimatta keskenään. Alussa tietokannassa on proseduuri

```
CREATE PROCEDURE HaeNimet AS
BEGIN
    SELECT Nimi FROM Asiakkaat
END
```

joka hakee tietokannasta kaikkien asiakkaiden nimet. Eräs kehittäjä päättää laajentaa proseduuria niin, että se hakee myös asiakkaiden yksilöivät tunnisteet. Tuloksena on migraatioskripti:

```
ALTER PROCEDURE HaeNimet AS
BEGIN
    SELECT Id, Nimi FROM Asiakkaat
END
```

Samaan aikaan toinen kehittäjä muuttaa alkuperäisen proseduurin hakemaan asiakkaiden tiedot taulun Asiakkaat sijaan taulusta Vieraat. Tätä varten hän kirjoittaa migraatioskriptin:

```
ALTER PROCEDURE HaeNimet AS
BEGIN
   SELECT Nimi FROM Vieraat
END
```

Molemmat kehittäjät tallentavat muutoksensa versionhallintaan. Jos syntynyttä ristiriitaa skriptien välillä ei huomata, jälkimmäisenä suoritettava skripti tulee ylikirjoittamaan aiemmin suoritettavan. Lopputuloksena jompikumpi skripteistä ei käytännössä astu voimaan, vaikka se on tallennettu versionhallintaan. Tilapohjaisella lähestymistavalla ongelma huomattaisiin lähes varmasti, sillä muutokset tehtäisiin samaan taulun tavoitetilan määrittelevään tiedostoon. Versionhallintaohjelmat varoittavat samaan tiedostoon tehdyistä päällekkäisistä muutoksista, ja vaativat kehittäjiä ratkaisemaan konfliktin.

Migraatiopohjaisessa versionhallinnassa migraatioskriptejä kasaantuu ajan mittaan yhä suurempi määrä. Jo tuotantoon asennettuja skriptejä ei tule myöhemmin muokata tai poistaa. Jos näin tehdään, kadotetaan yksiselitteinen tieto siitä, mitä muutoksia tuotantokantaan on tehty. Lisäksi myöhemmät kopiot samasta tietokannan versiosta eivät välttämättä enää vastaa tuotantoympäristön tietokantaa. Asennettujen skriptien muokkaamisen sijaan muutokset kuuluu tehdä aina kirjoittamalla uusia migraatioita. Tästä seuraa, että säilöttävien skriptien määrä kasvaa jokaisella muutoksella. [18]

Tilapohjainen versionhallinta ei kohtaa kasvavan tiedostomäärän ongelmaa, koska uusien skriptitiedostojen lisäämisen sijaan siinä muokataan alkuperäisiä tietokannan luon-

tiskriptejä. Versionhallintaan säilötään suoraan tietokannan haluttu lopputila yhtenä tiedostona tai tiedostokokoelmana, jonka koko ei automaattisesti kasva tietokantaa päivitettäessä.

Kasvava skriptien määrä aiheuttaa migraatiopohjaisille järjestelmille kaksi ongelmaa. Ensinnäkin luettavan tekstin määrän kasvaessa on yhä haastavampaa saada selvää tietokannan lopullisesta tilasta käymällä skriptejä läpi. Toiseksi tietokannan uuden kopion luonti ja päivitys viimeisimpään versioon hidastuu. Uutta kantaa luodessa alustetaan ensin tyhjä tietokanta, sitten suoritetaan migraatioskriptit järjestyksessä ensimmäisestä viimeiseen. Mitä enemmän skriptejä on, sitä enemmän aikaa kuluu niiden suorittamiseen.

Suoraviivainen ratkaisu liian suureksi kasvaneelle skriptien määrälle on valita tietty versio tietokannasta uudeksi alkutilaksi ja luoda uusi alustusskripti, joka luo tyhjästä tätä versiota vastaavan tietokannan [18]. Tämän jälkeen vanhempi alustusskripti ja kaikki uutta alkutilaa vanhemmat päivitysskriptit voidaan poistaa. Uusien tietokantojen luominen nopeutuu, kun luomisskripti asettaa kannan lähemmäs lopullista tilaa ja tarvitaan pienempi määrä päivitysskriptejä. Uudella alustusskriptillä luoduilla tietokannoilla ei kuitenkaan ole täysin sama versiohistoria kuin edellistä alustusskriptiä käyttäneillä, joten ne eivät ole täydellisiä kopioita aiemmista.

Tila- ja migraatiopohjaiset lähestymistavat reagoivat eri tavoin tietokantoihin, jotka poikkeavat versionhallinnassa määritellystä tilasta. Tyypillinen esimerkkitapaus on pikainen korjaus suoraan tietokantaan kiireellisen virhetilanteen ratkaisemiseksi. Molemmilla lähestymistavoilla normaalin prosessin ulkopuoliset muutokset aiheuttavat ongelmia tai sivuvaikutuksia. Tietokannan versionhallintaa käytettäessä on tärkeää, että kaikki muutokset tehdään versionhallintaprosessin kautta. [19]

Tilapohjainen tapa luo migraatiot automaattisesti vertaamalla kannan nykytilaa haluttuun tilaan. Vaikka tietokanta olisi eri tilassa kuin sen kuuluisi olla, sille luodaan tilanteeseen sopivat migraatiot. Poikkeamat poistetaan ja kanta päätyy kohdetilaan, ellei päivitysprosessissa ilmene muita ongelmia. Näin tilapohjainen menetelmä välttää virhetilanteiden syntymisen. Kuitenkin on tärkeää huomata, että versionhallinnan ulkopuolella tehdyt muutokset perutaan ja tietokanta palaa versionhallinnassa määriteltyyn tilaan. Tästä voi seurata sivuvaikutuksia, jos muutokset oli tehty hyvästä syystä. Ratkaisu on tehdä kaikki muutokset versionhallinnan kautta, jolloin päivitykset eivät automaattisesti ylikirjoita niitä. Samalla saadaan kaikki versionhallinnan yleiset hyödyt. [19]

Migraatiopohjainen tapa vain suorittaa kehittäjien kirjoittamat migraatiot, huomioimatta tietokannan todellista tilaa. Migraatioiden turvallinen käyttö perustuu oletukseen, että skriptin kirjoittaja tietää aina mikä kannan tila skriptin alussa on. Jos tietokantaan on tehty muutoksia normaalin prosessin ulkopuolella, tähän ei automaattisesti reagoida millään

tavalla. Päivityksen suorituksesta saattaa syntyä virheilmoitus, mutta on myös täysin mahdollista, että päivitys näennäisesti onnistuu normaalisti. Muutokset jäävät voimaan ja päivityksen jälkeen tietokanta päätyy tuntemattomaan tilaan. Tällöin tilanne on hyvin vaarallinen, sillä tietokanta on eri tilassa kuin kehittäjät luulevat sen olevan. Tieto tietokannan epänormaalista tilasta saattaa nousta esiin vasta, kun jokin tuleva migraatio epäonnistuu tai muita ongelmia syntyy. [19]

Versionhallinnan ulkopuolisten muutosten automaattinen valvonta on suositeltavaa. Jos havaitaan, että tietokantaan on tehty versionhallinnan ulkopuolisia muutoksia, on kaksi tapaa korjata tilanne. Molempien tavoitteena on saattaa kaikki tietokannat samaan tilaan ja pitää versionhallinta tietokantojen tilan totuuden lähteenä. Ensimmäinen on tehtyjen muutosten kumoaminen. Tällöin ajetaan muokattuun kantaan migraatio, joka palauttaa sen normaaliin tilaan. Toinen tapa on virallistaa tehdyt muutokset. Tämä tehdään käytännössä luomalla uusi migraatioskripti, joka sisältää kantaan tehdyt muutokset, ja tallentamalla se versionhallintaan. Versionhallinnasta migraatio voidaan suorittaa kaikille tietokannoille, jotta ne saadaan päivitettyä samaan tilaan. Jälkimmäinen menetelmä on suositeltava, kun tietokantaan tehty muutos on tarpeellinen ja halutaan säilyttää. Muulloin on parempi kumota haitallinen muutos. [18]

2.4 Tietokannan jatkuva integraatio

Jatkuva integraatio (continuous integration) on automaattinen prosessi, jonka tavoitteena on varmistaa, että versionhallinnassa oleva versio tietokannasta on aina toimiva ja valmiina asennettavaksi. Jatkuvassa integraatiossa eri kehittäjien tekemät muutokset tietokantaan yhdistetään säännöllisesti, tietokannasta kootaan ja asennetaan uusi versio, ja tämän uuden version toimivuus tarkistetaan suorittamalla automaattitestejä [21]. Integraatio on tarkoitus suorittaa usein (jatkuvasti) ja siinä paljastuvat ongelmat korjata nopeasti. Näin jatkuva integraatio tekee tietokannan asennuksista luotettavia ja tehokkaita, sekä tuottaa kehittäjille nopeaa palautetta heidän tekemistään muutoksista. Tietokannan jatkuva integraatio vaatii kaksi asiaa: tietokanta pitää voida koota ja asentaa automaattisesti, ja sille täytyy olla automaattitestejä.

Käytännössä jatkuva integraatio toteutetaan usein kolmannen osapuolen alustalla. Näitä on markkinoilla saatavilla kymmeniä. Suosittuja ovat esimerkiksi TeamCity, Jenkins, Travis, GitLab CI ja Bitbucket Pipelines [22].

2.4.1 Päivitysten asennus

Jotta tietokantapäivitykset voidaan testata, tietokannan uusin versio täytyy asentaa johonkin ympäristöön. Testaukseen on erittäin suositeltavaa käyttää omaa, tuotannosta ja

kehityksestä erillistä ympäristöään. Näin testien suoritus ei aiheuta sivuvaikutuksia muuhun tietokannan käyttöön. Samoin erillisen testausympäristön käyttö mahdollistaa tietokannan pitämisen aina tunnetussa tilassa testejä varten. Erinomainen vaihtoehto käytännön toteutukselle on käyttää testiympäristönä kontteja tai muita virtuaaliympäristöjä, jolloin tietokannan ja ympäristön tila testauksen alussa on aina tarkasti tiedossa ja kontrolloitu. Virtuaaliympäristöillä testien suoritus voidaan kevyesti siirtää palvelimelta toiselle minimaalisilla muutoksilla asennuksen toteutukseen. Palvelinresurssien säästämiseksi kontit voidaan myös poistaa testauskertojen välissä.

Tietokannan automaattinen koonti ja asennus pohjautuu versionhallintaan. Versionhallintaa tulee kohdella ylimpänä auktoriteettina tietokannan tilan ja asetusten suhteen, niin että se määrittää millainen kannan täsmälleen kuuluu olla. Mikäli versionhallinnan ja jonkin toisen tietolähteen välillä on ristiriitaa, toista tietolähdettä tulee korjata niin että se vastaa versionhallintaa. On tärkeää huolehtia, että versionhallintaan säilötään kaikki tietokannan asennukseen vaadittavat tiedot. Näitä tietoja ovat tietokannan rakenne eli skeema, tietokannan asetukset sekä tietokannan käyttöön tarvittava staattinen alustustieto. [14]

Automaattisen asennuksen tulee käynnistyä heti, kun uusi versio tietokannasta tallennetaan versionhallintaan. Näin huolehditaan jatkuvan integraation tavoitteesta tuottaa nopeaa palautetta tietokannan kehittäjille. Asennusprosessin täytyy myös kyetä päivittämään tietokanta mistä tahansa versiosta mihin tahansa myöhempään versioon. Muutoin asennukset eivät ole luotettavia.

Uuden tietokannan asennus tyhjälle pohjalle on yksinkertaista. Versionhallintaan säilötyt skriptit tarvitsee vain suorittaa kohdeympäristössä, ja uusi tietokanta luodaan [23]. Jos käytössä on tilapohjainen julkaisunhallinta, tietokannan uusimman tavoitetilan suoraan määrittelevät luontiskriptit suoritetaan sellaisinaan. Migraatiopohjaisessa lähestymistavassa puolestaan suoritetaan kaikki migraatioskriptit ensimmäisistä alkaen järjestyksessä. Skriptien automaattiseen suorittamiseen voidaan käyttää erikseen tähän suunniteltua työkalua, tai esimerkiksi yksinkertaista komentoriviskriptiä.

Vanhan, jo asennetun tietokannan päivittäminen uuteen versioon on vain hieman haastavampaa kuin uuden luonti. Tilapohjaista julkaisunhallintaa käytettäessä tarvitaan vertailutyökalu, joka tunnistaa erot tietokannan nykyisen ja uuden version välillä. Näiden erojen pohjalta vertailutyökalu luo migraatioskriptin, joka päivittää tietokannan uuteen versioon. On suositeltavaa, että kehittäjä tarkistaa tämän migraation manuaalisesti virheiden varalta viimeistään ennen päivityksen asennusta tuotantoon. Migraatiopohjaisessa julkaisunhallinnassa puolestaan täytyy tietää, mitkä migraatiot tietokantaan tulee suorittaa ja mitkä on jo asennettu. Yksinkertainen ratkaisu on lisätä tietokantaan taulu, johon tallennetaan tietokannan nykyinen versio ja migraatioiden suoritushistoria. [18]

2.4.2 Tietokannan testaus

Tietokannan testaukseen käytetään yhtä tai useampaa testiympäristöä. Testiympäristöt asettuvat tietokannan julkaisuputkessa henkilökohtaisten kehitysympäristöjen ja lopullisen tuotantoympäristön väliin. Testiympäristöjä käytetään automaattisten yksikkötestien, integraatiotestien ja tarvittaessa manuaalisten testien suorittamiseen. Niiden käytön tarkoituksena on paljastaa päivitysten mahdolliset viat ennen asennusta tuotantoon, jotta vältetään kalliit ongelmatilanteet. Tulosten luotettavuuden vuoksi testitietokannat tulee palauttaa tunnettuun alkutilaan aina testausten välissä. Varma tapa asettaa testitietokanta aina samaan tilaan on poistaa ja asentaa se uudelleen ennen jokaista testausta. Testitietokantojen tulee siis olla riittävän pieniä ja kevyitä, jotta uudelleenasennus on nopeaa. [24]

Automaattisen integraatiotestauksen tavoitteena on varmistaa tietokannan toimivuus sekä omana kokonaisuutenaan, että yhdessä muiden komponenttien kanssa. Testauksella lisätään asennusten ja päivitysten luotettavuutta. Testeillä pyritään löytämään puutteita ja virheitä erityisesti tietokannan päivitetystä versiosta, mutta mahdollisesti myös sen asennusprosessista [23]. Testejä on useita tyyppejä, ja niillä on eri tavoitteet [25]. Toiminnallisuustestit tarkistavat, että tietokannan yksittäiset osat ja laajemmat kokonaisuudet toimivat määritelmien mukaisesti. Suorituskykytestit selvittävät tietokannan tehokkuutta ja turvallisuustestit kannan tietoturvaa. Testien tulosten tulee olla helposti kehittäjien näkyvillä, ja hälyttävistä tuloksista lähetetään välittömästi viestiä vastuussa oleville henkilöille esimerkiksi sähköpostilla.

Tietokannan yksikkötestauksen tavoitteena on paljastaa tietokannan sisäisiä toiminnallisuusongelmia. Tällaisia ongelmia ovat esimerkiksi funktiot, jotka viittaavat puuttuviin taulun sarakkeisiin. Ongelmatilanne syntyy, kun sarakkeita poistetaan tai uudelleennimetään päivittämättä kaikkia niihin kohdistettuja viittauksia. Tietokannan rakennetta muokatessa on aina tärkeää varmistaa, että kaikki viittaukset ja riippuvaisuudet ovat yhä kunnossa. [25]

Yksikkötestit kannattaa kohdistaa niihin tietokannan osiin, jotka sisältävät logiikkaa tai viittaavat toisiin tietokannan osiin. Tällaisia objekteja ovat funktiot, proseduurit, laukaisimet ja näkymät. Yksinkertaiset regressiotestit auttavat varmistamaan jokaisen tietokantamuutoksen jälkeen, että ne toimivat yhä.

Jotta testaus varmistaa tietokannan uuden version toimivan yhdessä muiden ohjelmistokokonaisuuden komponenttien kanssa, tietokantapäivityksen yhteydessä kannattaa suorittaa tietokantaan yhteydessä olevien sovellusten ja muiden ohjelmistojen integraatiotestit [24]. Näin paljastuvat tilanteet, joissa itse tietokanta toimii, mutta siihen tehdyt muutokset aiheuttavat ongelmia muissa ohjelmissa. Mikäli tietokanta on sidottu yhteen sovellukseen ja ne on säilötty versionhallinnassa samaan repositorioon, tämä tapahtuu

oletuksena. Jos taas tietokantaa käyttää useampi eri sovellus, niiden integraatiotestien suoritus pitää määritellä erikseen osaksi tietokannan testausta.

Lyhyt vasteaika ja kyky käsitellä suuria tietomääriä ovat tärkeitä tietokannan ominaisuuksia. Niitä on hyvä mitata automaattitestauksessa. Luotettava ja todellista tuotantoympäristöä vastaava testaus kuitenkin vaatii tuotantoa vastaavan tiedon syöttöä testitietokantaan. Tietosisällön tulisi ihannetilanteessa vastata tuotantoa sekä määrältään että sisällöltään. Testaukseen käytettävä tietosisältö voidaan luoda manuaalisesti tai generoida tarkoitukseen kehitetyn työkalun avulla, tai se voidaan kopioida tuotantotietokannasta ja muokata anonyymiksi. Viimeisin vaihtoehto antaa parhaiten todellista tuotantokäyttöä vastaavaa tietoa, jolla saadaan siten luotettavimpia testituloksia. Tietoturvan vuoksi on kuitenkin erittäin tärkeää, että henkilötietoja sisältävää tuotantotietoa ei kopioida suoraan eri ympäristöihin testattavaksi. Täten suositeltava tapa luoda testaukseen käytettävä tieto on joko tuotantokannan sisällön kopiointi ja henkilötietojen häivytys, tai laadukkaan tietosisältögeneraattorin käyttö. [25]

Vasteaikojen mittauksilla testiympäristössä on haastavaa saada täsmällisiä tietoja tietokannan toiminnasta tuotantoympäristöstä. Ensimmäinen rajoite tarkalle mittaukselle on, että testi- ja tuotantoympäristöillä pitäisi olla käytössään samat laitteistoresurssit [25]. Käytännössä nämä ympäristöt asennetaan usein eri palvelimille, joilla on erilainen laitteisto ja muut resurssit. Testeillä voidaan kuitenkin seurata, miten päivitykset muuttavat tietokannan toimintaa verrattuna edellisiin versioihin. Toiseksi tarkka testaus vaatisi testiympäristöön tuotantoa vastaavan määrän tietosisältöä [25], mutta nopeita uudelleenasennuksia varten testiympäristöstä halutaan kevyt. Tämä ongelma voidaan kiertää ajamalla jatkuvasti nopeat vakiotestit, ja niistä erillään säännöllisin väliajoin laajemmat ja pidempikestoiset suorituskykytestit suurella tietomäärällä [23].

2.5 Tietokannan jatkuva julkaisu ja toimitus

Jatkuva julkaisu (continuous deployment) muistuttaa osittain jatkuvaa integraatiota. Molemmissa tavoitteisiin kuuluu asennusten automaatio. Jatkuva integraatio kuitenkin keskittyy vain testiympäristöihin tehtäviin asennuksiin ja päivitysten automaattiseen testaukseen. Jatkuvassa julkaisussa sen sijaan automatisoidaan tietokantapäivitysten asennus kaikkiin käytössä oleviin ympäristöihin, tärkeimpänä tuotantoon. Jatkuva julkaisu tarkoittaa siis tilannetta, jossa kehittäjien tekemät muutokset julkaistaan automaattisesti tuotantoon käyttäjille, ellei niistä löydy puutteita julkaisuputken automaattisissa tarkistuksissa. Missään kohdin julkaisuputkea ei vaadita manuaalista vahvistusta ihmiseltä. [16]

Jatkuva toimitus (continuous delivery) on lähes sama asia kuin jatkuva julkaisu. Ainoa ero on, että päivityksiä ei asenneta tuotantoympäristöön ilman ihmisen manuaalista hyväksyntää. Versionhallinnan pohjalta tietokantapäivitys siis automaattisesti asennetaan

ja testataan järjestyksessä kaikkiin muihin ympäristöihin. Päivitys myös kyetään asentamaan nopeasti ja automaattisesti tuotantoon, kun tälle saadaan hyväksyntä. Jatkuvan toimituksen etuna on lisätty kontrolli. Tuotantopäivitykset voidaan manuaalisesti tarkastaa ja katselmoida vielä viimeisen kerran juuri ennen julkaisua. Haittapuoli taas on julkaisujen hidastuminen, kun päivitysten täytyy odottaa manuaalista hyväksyntää. [26]

Jatkuvan julkaisun hyödyt nousevat esiin järjestelmissä, joissa tietokanta asennetaan useampaan kuin yhteen ympäristöön. Normaalisti julkaisuympäristöjä on vähintään kolme: kehitys, testaus ja tuotanto. Useamman kuin kolmen julkaisuympäristön käyttö on myös yleistä. Esimerkiksi erityyppisiä testejä varten voidaan luoda omia julkaisuympäristöjään, kuten omat ympäristönsä integraatiotesteille ja hyväksymistesteille. Toinen suosittu lisäys käytettäviin julkaisuympäristöihin on tuotantokopiokanta (staging), joka on mahdollisimman suora kopio tuotantokannasta. Tuotantokopiokannassa tehdään viimeiset tarkistukset päivityksen toimivuudesta ja vaikutuksista. [16]

Julkaisuputkessa asennukset ja testaukset tehdään vaiheittain, yksi ympäristö kerrallaan. Ympäristöille määritellään järjestys, jossa päivitys niihin asennetaan ja testataan. Jos tietokantapäivityksen asennus onnistuu ja päivitys läpäisee kaikki testit, se siirretään seuraavaan ympäristöön asennukseen ja testaukseen. Jos taas asennuksessa tai testeissä ilmenee ongelmia, asiasta ilmoitetaan kehittäjille ja päivitystä ei enää siirretä eteenpäin. Asennukset aloitetaan testiympäristöstä, jossa päivityksen perustoimivuus voidaan testata nopeasti, ja jossa vaihtelu eri versioiden välillä sekä mahdolliset viat päivityksessä aiheuttavat vähiten ongelmia. Tästä siirrytään muunlaisiin testeihin kuten suorituskyky- ja tietoturvatesteihin, mahdollisesti omissa ympäristöissään. Jokaisen onnistuneen asennuksen ja testauksen tuloksena päivityksen luotettavuus kasvaa. Viimeisenä askeleena päivitys asennetaan tuotantotietokantaan, joka on todellisten asiakkaiden käytössä.

Asennusten toteutus vaihtelee ympäristöjen välillä. Ensinnäkin eri ympäristöjen vaihtelevat palvelintason ominaisuudet tulee huomioida automaattisissa asennuksissa [14]. Toiseksi ennen päivityksen asennusta ympäristö tulee valmistella asennusta varten, mikä voidaan tehdä eri tavoin riippuen ympäristön käyttötarkoituksesta. Kolmanneksi asennuksen jälkeen valmistaudutaan testaukseen, minkä toteutus riippuu suoritettavista testeistä.

Huomioitavia palvelintason ominaisuuksia ovat muun muassa palvelimen asetukset kuten käytettävän muistin maksimimäärä, ajastetut SQL Agent -taustatehtävät sekä palvelimen laajuiset tietoturva-asetukset. Tietokannan toimintaan vaikuttavat muutokset palvelintason ominaisuuksiin tulee tehdä versionhallinnan kautta, kuten kaikki muutkin muu-

tokset. Jos eri ympäristöissä tarvitaan eri asetuksia, ne kuuluu erotella selkeästi toisistaan ja säätää julkaisut käyttämään jokaisessa ympäristöissä sille kuuluvia asetuksia. [24]

Ympäristön valmistelu sekä päivityksen asennusta että testausta varten riippuu siitä, mitä tietokannan ominaisuuksia ympäristössä aiotaan tarkastaa. Esimerkiksi integraatiotestausympäristössä tavoitteena on varmistaa, että versionhallinnassa määritelty versio tietokannasta on toimiva. Asennuksessa tietokannan tulee siis päätyä täsmälleen versionhallinnassa määriteltyyn tilaan, ja testejä varten kantaan tarvitaan pieni määrä täsmälleen tunnettua testitietoa [16]. Tällaisessa ympäristössä tietokanta aluksi poistetaan ja sitten asennetaan tyhjästä uudelleen, minkä jälkeen siihen ladataan testausta varten pieni paketti etukäteen luotua testitietoa. Tällainen paketti voi olla esimerkiksi sensuroitua vanhaa tuotannon tietoa. Toisena esimerkkinä tuotantokopiotietokannan tarkoitus on selvittää, miten tuotantotietokanta reagoi päivitykseen [24]. Tätä varten on luontevaa alustaa päivityksen asennus poistamalla tuotantokopiotietokanta ja korvaamalla se ajantasaisella kopiolla tuotantokannasta. Tietosisällön suhteen suositaan kaiken tuotantotietokannan sisällön kopioimista sisään alustettavaan tuotantokopiokantaan ennen päivityksen asennusta ja testausta. Itse tuotantoympäristöön asennuksessa tietokanta taas sisältää valmiiksi asiakkaiden aitoa liiketoimintatietoa, joka halutaan ehdottomasti säilyttää, eikä sen sekaan syötetä mitään testitietoa. Päivitys asennetaan suoraan, ilman ympäristön tilaa muuttavia valmisteluja.

Julkaisuprosessin sisältämästä testauksesta huolimatta on tärkeää tehdä suunnitelma, miten toivutaan, jos julkaistusta päivityksestä löytyy vika vasta kun se on jo asennettu tuotantoon. Tavoitteena on varmistaa mahdollisimman hyvin, ettei päivityksestä aiheudu häiriötä asiakkaille tai varsinkaan arvokkaan tiedon menetystä. Tätä varten on useita keinoja, joilla julkaisujen ongelmista toivutaan mahdollisimman nopeasti ja turvallisesti. Suoraviivainen tapa on ottaa tuotantotietokannasta ennen päivitystä varmuuskopio tai tilannevedos (snapshot) ja palauttaa tietokanta tähän aiempaan tilaan heti, jos ongelmia ilmenee. Toinen tapa on käyttää kahta tuotantotietokantaa, jolloin kaikki asiakasliikenne voidaan ohjata yhteen samalla kun toinen päivitetään. Jos päivitys onnistuu, ohjataan liikenne päivitettyyn tietokantaan. Kolmas keino on luoda jokaiselle päivitykselle palautusskripti, joka kumoaa päivityksen muutokset ja palauttaa tietokannan edelliseen versioon. Neljäntenä vaihtoehtona voidaan tehdä ja nopeasti julkaista uusi päivitys, joka korjaa äskeisen päivityksen tuomat ongelmat. Jokaisella tavalla on omat vahvuutensa ja heikkoutensa, joten valinta riippuu käsillä olevasta tilanteesta ja kehitystiimin prioriteeteista. [24]

Vikatilanteiden määrä kasvaa epäsuorasti, jos julkaisuprosessi vie paljon aikaa. Automaattista julkaisuputkea käytettäessä kaikki muutokset täytyy tehdä saman prosessin

kautta. Jos näin ei tehdä, syntyy merkittäviä ongelmia kuten aiempien muutosten katoamisia, tietokannan päätymistä tuntemattomaan tilaan sekä mahdollisesti liiketoimintatiedon katoamista tai korruptoitumista. Hidas julkaisuprosessi muuttuu vakavaksi ongelmaksi, jos tuotantotietokantaan täytyy tehdä kiireellinen muutos. Ainoat vaihtoehdot ovat joko tehdä muutos nopeasti normaalin prosessin ulkopuolelta, jolloin syntyy vakava sivuvaikutusten riski, tai tehdä muutos prosessin kautta, missä kestää aikaa. [27]

Jatkuva julkaisu vaatii kattavaa päivitysten testausta, sekä luottoa automaattisen julkaisuputken kykyyn toimia ongelmitta ja havaita viat päivityksissä. Vastineeksi se tarjoaa hyvin nopeat ja tiheät päivitysten julkaisut asiakkaiden käyttöön sekä työajan säästöä. Kaikkien päivitysten automaattinen asennus ja testaus useissa ympäristöissä lisää päivitysten luotettavuutta huomattavasti verrattuna manuaalisiin päivityksiin. Jatkuva toimitus tarjoaa pitkälti samat edut, mutta lopun manuaalisen tarkistuksen vuoksi vaatii vähemmän luottoa julkaisuputkeen ja tuottaa hitaampia päivitysten julkaisuja.

3. TIETOKANNAN JULKAISUNHALLINNAN TYÖKALUT

Tietokannan automaattisessa julkaisunhallinnassa on useita vaiheita, joiden toteutus hyötyy suuresti ohjelmistotyökalujen käytöstä. Nämä työkalut voivat olla itse kehitettyjä tai kolmansien osapuolten tuottamia. Valmiita työkaluja on kaikkiin vaiheisiin saatavilla kymmeniä tai satoja, joten niitä kaikkia ei voida tässä työssä käydä läpi. Jokaiseen osioon on poimittu muutamia huomionarvoisia työkaluja esiteltäväksi.

Tässä luvussa tutustutaan julkaisunhallinnan eri vaiheiden työkalutarpeisiin ja vaatimuksiin, sekä kunkin vaiheen kohdalla muutamaan suosittuun ja julkisesti saatavilla olevaan työkaluun. Osiossa 3.1 käsitellään tietokantamuutosten määrittelyä ja tietokantaskriptien tallentamista versionhallintaan, osiossa 3.2 tietokannan toiminnan yksikkötestausta, ja osiossa 3.3 tietokannan muutosten julkaisemista eri ympäristöihin.

3.1 Versionhallinnan työkalut

Tietokannan versionhallinnassa voidaan hyödyntää työkaluja, jotka avustavat tietokantamuutosten suunnittelussa ja versionhallintaan säilöttävien skriptien luonnissa. Näiden lisäksi tarvitaan versionhallintajärjestelmä sekä -alusta. Tietokannan versionhallinta ei aseta kummallekaan merkittäviä erityistarpeita, joten niihin voidaan käyttää samoja ohjelmistotyökaluja kuin muille sovelluksille. Tämän vuoksi niiden ominaisuuksiin ei paneuduta tässä työssä yksityiskohtaisesti.

Kuten aiemmin luvussa 2.3 kuvattiin, versionhallintaan säilötään tietokannan osalta skeeman muutokset sekä tietokannan vaatimat asetusten muutokset ja alustustieto. Skeeman muutokset voidaan tallentaa joko suoraan määrittelynä tietokannan nykyisestä kohdetilasta, tai migraatioskripteinä, joilla tietokannan aiemmat versiot voidaan päivittää kohdetilaan. Myös asetusmuutoksille ja alustustiedolle luonteva tallennusmuoto on skriptit, jolloin ne on helppo suorittaa. Tietokannan osalta versionhallintaan säilötään siis pelkästään tekstimuotoisia skriptitiedostoja. Niiden säilöminen ei itsessään vaadi muiden sovellusten käyttämästä versionhallinnasta poikkeavia erityisominaisuuksia.

3.1.1 Versionhallintatyökalujen hyödyt

Mahdollisuus ohjelmistotyökalun käyttöön avautuu päivitysten määrittelyssä graafisesti ja itse skriptien generoinnissa. Muuttuneen tilan määrittelyt tai migraatioskriptit voidaan kirjoittaa käsin, mutta niiden luontiin on olemassa myös työkaluja. Niillä muutokset voidaan määritellä graafisesti tietokannan rakennekaaviota muuttamalla, tai valikoiden kautta vaihtoehtoja poimimalla. Kun halutut rakennemuutokset on määritelty, työkalut

osaavat generoida tietokantaskriptit, jotka määrittelevät tietokannan uuden tilan (jos käytetään tilapohjaista versionhallintaa) tai tehtävät muutokset (jos käytetään migraatiopohjaista versionhallintaa).

Graafisen mallinnustyökalun käyttö yhdistää tietokannan suunnittelua, dokumentointia ja muutosten toteutusta. Tietokantamallia käsittelemällä tietokannan tavoitetila on selkeästi nähtävissä, jolloin väärinkäsitysten riski laskee. Graafinen malli on myös arvokas lisä tietokannan dokumentaatioon, ja se voidaan jakaa kehittäjältä toiselle tehostaen kommunikaatiota. SQL-skriptejä graafisen mallin pohjalta luova ohjelmistotyökalu mahdollistaa tietokannan kehityksen pienemmällä määrällä skriptien kirjoitusta, jolloin mallin muokkaus on itsessään osa tietokannan muokkausta.

Maailmanlaajuisesti suosituin versionhallintajärjestelmä on ylivoimaisesti Git [28], jota käyttää yli 80% ohjelmistonkehittäjistä [29]. Muita vaihtoehtoja on useita, muun muassa Apache Subversion [30] ja Mercurial [31], mutta ne ovat selvästi pienemmässä käytössä. Suuren suosion, kattavien ominaisuuksien ja laajasti saatavilla olevan tuen vuoksi Git on ensisijaisesti suositeltava valinta tietokannan versionhallintajärjestelmäksi.

Julkisia versionhallinta-alustoja on useita. Suosittuja ovat muun muassa Atlassian Bitbucket [13], GitHub [32], GitLab [33] ja moni muu. Myös oman versionhallinta-alustan kehittäminen on mahdollista. Tietokannan julkaisunhallintaa toteutettaessa on suositeltavaa käyttää alustaa, joka tukee automaattisen julkaisuputken luontia. Näin voidaan välttää tarve erilliselle julkaisunhallinta-alustalle ja yksinkertaistaa julkaisuprosessia.

3.1.2 Tietokantaskriptien luonnin työkalujen esittelyjä

Versionhallinnan kannalta työkalujen käytön tavoitteena on löytää menetelmä, jolla tietokantaan tehtävät muutokset saadaan tallennettua versionhallintaan mahdollisimman luotettavasti, tehokkaasti ja helposti. Käytännössä tämä tarkoittaa migraatioiden tai tietokannan tavoitetilaa määrittelevien skriptien luomista suunnitelluista muutoksista. Monet tietokannan versionhallinnan tueksi sopivista työkaluista tarjoavat myös paljon ominaisuuksia yleiseen tietokantojen hallintaan, ylläpitoon ja kehitykseen, sekä mahdollisesti esimerkiksi testaukseen, raportointiin ja varmuuskopiointiin. Mielekkäintä on, jos samalla ohjelmalla voidaan tehdä sekä tietokantamuutosten suunnittelu ja tietokantaskriptien että tietokannan yleinen kehitys, hallinta ja ylläpito. Tällöin käytettävien ohjelmien pienempi määrä ja tuttuus kehittäjille tekevät muutosten teosta nopeampaa ja helpompaa. Muutamia suosittuja esimerkkejä hyvin versionhallinnan tueksi sopivista ohjelmistoista ovat DBeaver [34], dbForge Studio [35], Entity Framework Core [36], Navicat Data Modeler [37], Oracle SQL Developer [38], SQL Server Data Tools [39], SQL Server Management Studio [40] ja Toad Data Modeler [41].

DBeaver [34] on DBeaver Corporationin kehittämä maksullinen ohjelma tietokantojen yleiseen hallintaan, josta on tarjolla myös ilmainen versio rajoitetuilla ominaisuuksilla. Se tukee useita kymmeniä eri tietokantoja, joihin kuuluu sekä SQL- että NoSQL-kantoja. Tuettuja tietokantoja ovat muun muassa MySQL, PostgreSQL, Microsoft SQL Server, MongoDB, AWS Athena ja Redis. Sillä voi tehdä muutoksia tietokantaan graafisen käyttöliittymän kautta tauluja muokkaamalla, ja generoida näistä muutoksista joko tila- tai migraatiopohjaiset skriptit.

dbForge Studio [35] on Devartin kehittämä maksullinen ohjelma tietokantojen yleiseen hallintaan, josta on tarjolla myös ilmainen versio rajoitetuilla ominaisuuksilla. Ohjelma tukee useita eri tietokantoja, kuten Microsoft SQL Server, MySQL, Oracle ja PostgreSQL, mutta se täytyy ostaa erikseen jokaiselle tietokannalle, ja ohjelman ominaisuudet myös vaihtelevat eri tietokantojen välillä. Microsoft SQL Server tarjoaa laajimman valikoiman ominaisuuksia ja PostgreSQL pienimmän. Parhaimmillaan eli maksullisena versiona Microsoft SQL Server-tietokannassa dbForge Studio tarjoaa monipuolisen tuen versionhallinnalle. Tähän kuuluu skriptien luominen, niiden tallentaminen versionhallintaan ja eri kehittäjien tekemien muutosten vertailu ja yhdistäminen. Lisäksi ohjelma tukee tietokannan testausta, raportointia, eri kantojen vertailua toisiinsa ja muita kehityksen ja ylläpidon ominaisuuksia. Heikoimmillaan eli ilmaisversiona PostgreSQL-kannassa se ei tue versionhallintaa lähes lainkaan. Ohjelman hinta vaihtelee version mukaan välillä 200-700€.

Entity Framework Core [36] on Microsoftin kehittämä ilmainen ohjelma olio-relaatiomallinnukseen. Ohjelma tukee oletuksena tietokantoja SQL Server, Azure SQL Database, SQLite ja Azure Cosmos DB, minkä lisäksi siihen voidaan liittää kirjastoja, jotka tarjoavat tuen muille tietokannoille. Sillä voi tehdä muutoksia tietokantaan C#-olioita muokkaamalla ja generoida näistä muutoksista joko migraatiopohjaiset skriptit. Ohjelma tukee rakennekaavioiden vapaata muokkausta irrallaan itse tietokannasta.

Navicat Data Modeler [37] on PremiumSoftin kehittämä maksullinen ohjelma tietokantojen mallinnukseen, josta on tarjolla myös ilmainen versio rajoitetuilla ominaisuuksilla. Ohjelma tukee useita eri tietokantoja, kuten MySQL, MariaDB, Oracle, SQL Server, PostgreSQL ja SQLite. Sillä voi luoda ja muokata graafisia tietokannan rakennekaavioita, ja generoida näistä muutoksista joko tila- tai migraatiopohjaiset skriptit. Navicat tukee rakennekaavioiden vapaata muokkausta ilman muutosten tekoa tietokantaan. Ohjelman hinta on $459.

Oracle SQL Developer [38] on Oraclen kehittämä ilmainen ohjelma tietokantojen yleiseen hallintaan. Se tukee ainoastaan Oracle Database-tietokantoja. Sillä voi tehdä muutoksia tietokantaan graafisen käyttöliittymän kautta tauluja muokkaamalla, ja generoida

näistä muutoksista joko tila- tai migraatiopohjaiset skriptit. Ohjelma tukee rakennekaavioiden vapaata muokkausta irrallaan itse tietokannasta.

SQL Server Management Studio [40] on Microsoftin kehittämä ilmainen ohjelma tietokantojen yleiseen hallintaan. Se tukee ainoastaan Microsoftin SQL Server- ja Azure SQL-tietokantoja. Sillä voi tehdä muutoksia tietokantaan graafisen käyttöliittymän kautta tauluja muokkaamalla, ja generoida näistä muutoksista joko tila- tai migraatiopohjaiset skriptit.

Toad Data Modeler [41] on Quest Softwaren kehittämä maksullinen ohjelma tietokantojen mallinnukseen ja raportointiin. Se tukee useita eri tietokantoja, kuten Oracle, SAP, MySQL, SQL Server ja PostgreSQL. Sillä voi tehdä muutoksia tietokantaan graafisen käyttöliittymän kautta tauluja muokkaamalla, ja generoida näistä muutoksista joko tila- tai migraatiopohjaiset skriptit. Ohjelma tukee rakennekaavioiden vapaata muokkausta irrallaan itse tietokannasta. Ohjelman hinta on reilu 600€ per lisenssi.

SQL Server Data Tools [39] on Microsoftin kehittämä ilmainen ohjelma tietokantojen päivitykseen. Se tukee tietokantoja SQL Server ja Azure SQL. Ohjelma on lisäosa Visual Studio -ohjelmistonkehitysympäristöön. Sillä voi tehdä muutoksia tietokantaan graafisen käyttöliittymän kautta tauluja muokkaamalla, ja generoida näistä muutoksista joko tila- tai migraatiopohjaiset skriptit. Ohjelman ominaisuudet ovat rajalliset, eikä edistyneitä muokkauksia voi tehdä kirjoittamatta SQL-koodia käsin.

3.2 Testauksen työkalut

Kattavaan tietokannan testaukseen kuuluu yksikkö-, integraatio- ja suorituskykytestausta. Näistä integraatiotestaus toteutetaan tietokantaa käyttävien sovellusten testeinä. Integraatiotesteissä sovellukset muodostavat yhteyden tietokantaan, tekevät erilaisia operaatioita ja varmistavat että sovellus ja tietokanta toimivat yhdessä oikein. Integraatiotestaus ei siis vaadi tietokannan osalta erillisten työkalujen käyttöä, riittää että tietokanta asennetaan ja alustetaan testiympäristöön ennen sovellusten testien ajoa. Tietokannan yksikkötesteille ja suorituskykytesteille sen sijaan voidaan hyödyntää testausta helpottavia ohjelmistotyökaluja.

3.2.1 Testaustyökalujen hyödyt

Tietokannan yksikkötestaus ei pohjimmiltaan vaadi työkaluilta paljoa. Minimivaatimus on, että työkalulla voidaan ajaa testaajan kirjoittamia SQL-komentoja testitietokantaan. Saatavilla on kuitenkin edistyneempiä työkaluja, jotka tekevät testien kirjoittamisesta ja suorittamisesta sekä testitulosten läpikäynnistä helpompaa. Työkalut voivat myös mitata tietoja testien suorituksesta.

Suorituskykytestausta varten testattavaan kantaan tarvitaan suuret määrät testitietoa. Testitiedon tuottamisessa voidaan käyttää työkaluja, jotka toimivat jommallakummalla kahdesta päätavasta. Ensimmäinen tapa tuottaa testitietoa on generoida taulujen rakenteeseen ja odotettuun sisältöön sopivaa satunnaista tietosisältöä. Toinen tapa on kopioida sopivaa tietosisältöä toisesta tietokannasta ja tarvittaessa muokata sitä niin, ettei testiympäristöön tallenneta arkaluontoista tietoa.

3.2.2 Tietokannan yksikkötestauksen työkalujen esittelyjä

Tietokannan yksikkötestejä voidaan ajaa täysin samoilla työkaluilla kuin sovellusten yksikkötestejä, tai niihin voidaan käyttää omaa työkaluaan. Seuraavaksi esitellään tietokannan yksikkötestaustyökalut DbFit [42], DbUnit [43] ja tSQLt [44], sekä tapa yksikkötestata tietokantaa samoin kuin sovellusta.

DbFit [42] on DbFit-tiimin kehittämä ilmainen avoimen lähdekoodin FitNesse-testausohjelman laajennus tietokantojen yksikkö- ja integraatiotestaamiseen. Siitä on Java- ja C#-kieliset versiot. Java-versio tukee tietokantoja Oracle, SQL Server, MySQL, DB2, PostgreSQL, HSQLDB ja Derby. C#-kielinen versio tukee tietokantoja Oracle, SQL Server, MySQL ja Sybase. Testit kirjoitetaan DbFitin omalla formaatilla, joka muistuttaa taulukoksi muotoiltua tekstiä, ja tulokset näytetään verkkosivuna. Testeistä muodostuu wikityyppinen verkkosivusto, jota selaamalla testejä ajetaan ja tulokset näytetään.

DbUnit [43] on DbUnit-tiimin kehittämä ilmainen avoimen lähdekoodin JUnit-testauskirjaston laajennus tietokantojen yksikkö- ja integraatiotestaukseen. Se on toteutettu Java-kielellä ja tukee pääasiassa Oracle ja HypersonicSQL tietokantoja. Ohjelmalla on myös puolivirallinen tuki useille muille tietokantatyypeille, mutta niitä testataan vähemmän eikä niiden toimivuudelle anneta takuita. Testit kirjoitetaan Java-kielellä, ja tietokantaan syötettävä testitieto määritetään XML-tiedostona. Testitulokset tulee käsitellä testien koodissa.

tSQLt [44] on tohtori Sebastian Meinen ja Dennis Lloyd Juniorin kehittämä ilmainen avoimen lähdekoodin ohjelma tietokantojen yksikkötestaamiseen. Se on kirjoitettu pääasiassa SQL-kielellä ja tukee vain Microsoft SQL Server-tietokantoja. Testit kirjoitetaan SQL-kielellä, ja niiden tulokset näytetään tekstinä tai XML-muodossa. tSQLt yksikkötestit tallennetaan testattavan tietokannan sisään proseduureina. Ne ajetaan automaattisesti aina transaktioissa, joiden lopuksi testien tekemät tietokantamuutokset kumotaan. tSQLt tukee testikohteiden eristämistä korvaamalla aitoja tauluja ja näkymiä testin suorituksen ajaksi jäljitelmillä.

Erillisen testausohjelman käytön sijaan tietokantaa voidaan myös yksikkötestata hyvin samankaltaisesti kuin sovelluksia. Halutulla ohjelmointikielellä kirjoitetaan vain automaattitestejä, joissa yhdistetään tietokantaan, ajetaan SQL-komentoja ja tarkastetaan

tulos. Tällaiset testit voidaan luontevasti kirjoittaa samoilla työkaluilla kuin tietokantaa käyttävän sovelluksen omat automaattitestit. Sovelluksen ja tietokannan ollessa versionhallinnassa samassa repositoriossa, niiden testit voidaan säilöä yhdessä ja ajaa julkaisuputkessa samaan aikaan. Riippuen käytettävistä testi- ja yhteyskirjastoista menetelmä voi tukea hyvin laajaa valikoimaa eri tietokantoja. Käytettävät kirjastot määräävät pitkälti, kuinka helppoa ja kevyttä testien luonti on.

3.2.3 Testitietosisällön luonnin työkalujen esittelyjä

Varsinkin suorituskykytestaus hyötyy testitietosisällön suuresta määrästä. Pelkän tietokannan perustoimivuuden tarkastamiseen riittää usein, että yksikkö- ja integraatiotestien alussa kantaan syötetään muutamia rivejä vakiotietosisältöä. Suorituskykytesteissä tarvitaan kuitenkin vähintään tuhansia rivejä tietosisältöä, jotta voidaan kunnolla vertailla tietokantamuutosten vaikutusta tietokantakäskyjen suoritusnopeuteen. Testitiedon tuottamiseen sopivia työkaluja ovat DTM Data Generator [45], EMS Data Generator [46] ja Jailer [47].

DTM Data Generator [45] on DTM soft-yrityksen kehittämä maksullinen suljetun lähdekoodin ohjelma, jolla tietokantaan generoidaan satunnaista tietosisältöä. Se tukee tietokantoja Microsoft SQL Server, Oracle, IBM DB2, Sybase, Informix, MySQL, PostgreSQL ja Interbase. Ohjelmalla voi luoda tietosisältöä useilla eri tavoilla, vaihdellen täysin satunnaisesta olemassa olevan tiedon matkimiseen. Ohjelma huomioi tiedon generoinnissa automaattisesti vierasavaimet, missä sarakkeissa NULL-arvo on sallittu, ja sarakkeiden arvoille tietokannassa määritetyt muut rajoitteet. Ohjelman hinta on valitusta lisenssin tyypistä riippuen $149 - $439 per lisenssi.

EMS Data Generator on EMS Software Development-yrityksen kehittämä maksullinen suljetun lähdekoodin ohjelma, jolla tietokantaan generoidaan satunnaista tietosisältöä. Siitä löytyy versio tietokannoille Oracle, DB2, MySQL, SQL Server, PostgreSQL ja Interbase. Ohjelmasta täytyy ostaa kullekin tietokannalle oma versionsa. Ohjelmalla voi luoda tietosisältöä useilla eri tavoilla, vaihdellen täysin satunnaisesta olemassa olevan tiedon matkimiseen. Ohjelma huomioi tietosisällön generoinnissa automaattisesti vierasavaimet ja tukee tiedon generointia kaikille tietokantojen käyttämille tietotyypeille. Ohjelman hinta on valitusta lisenssityypistä riippuen $110 - $187 per lisenssi.

Jailer [47] on Wisser-nimimerkkisen kehittäjän luoma ilmainen avoimen lähdekoodin ohjelma, jolla tietokannasta voi poimia tietokokoelman, jonka kaikki viitteet ovat ehjiä. Se on kehitetty Java-kielellä ja käyttää JDBC-rajapintaa, joka tukee laajaa valikoimaa eri tietokantoja. Ohjelman käyttäjä valitsee tietokantataulut, joista tietosisältöä halutaan kerätä, sekä ehdot, joilla taulujen sisältö suodatetaan. Jailer poimii tietokannasta ensin valitun sisällön, sitten muista tauluista kaikki rivit, joihin valittu sisältö viittaa, sitten kaikki

rivit, joihin nämä viitatut rivit viittaavat ja niin edelleen. Lopputuloksena kerätään yhtenäinen tietokokoelma, joka sisältää kaikki rivit joihin kokoelman mikään osa viittaa. Jailer tukee myös kerätyn tiedon muokkausta SQL-komennoilla, joten käyttäjän on mahdollista piilottaa tietokokoelman mahdollisesti sisältämä arkaluontoinen sisältö. Suorituskykytestauksessa Jailerilla voidaan kopioida osa tuotantokannan sisällöstä testikantaan.

3.3 Julkaisun työkalut

Automaattiset päivitystyökalut voidaan jakaa tila- ja migraatiopohjaisiin, samoin kuin versionhallinnan käyttö. Tilapohjaiset työkalut vertaavat päivitettävän tietokannan nykytilaa päivityksen haluttuun lopputilaan ja generoivat erojen pohjalta suoritettavan migraatioskriptin. Migraatiopohjaiset työkalut puolestaan eivät luo migraatioita itse vaan suorittavat annetut migraatioskriptit.

Saatavilla on myös työkaluja, jotka tukevat molempia lähestymistapoja. Lisäksi on mahdollista jakaa päivitys kahden työkalun välille, joista toinen vertailee tietokantojen tiloja ja luo migraatioskriptin, ja toinen suorittaa migraation tietokantoihin. Tämä on kuitenkin lähtökohtaisesti tarpeetonta monimutkaisuutta, sillä sama lopputulos voidaan saavuttaa yhdelläkin työkalulla.

Päivitystyökalun tulee vähintäänkin tukea ominaisuuksia, jotka lukujen 2.4 ja 2.5 mukaisesti ovat edellytyksiä toimivalle jatkuvan integraation ja julkaisun prosessille. Automaattinen päivitystyökalu tulee voida käynnistää heti, kun uusi versio tietokannasta tallennetaan versionhallintaan. Sen täytyy kyetä päivittämään tietokantoja mistä tahansa versiosta mihin tahansa myöhempään versioon. Asennus pitää voida toteuttaa eri tavoin riippuen päivitettävästä ympäristöstä. Jos käytössä on tilapohjainen versionhallinta, päivitystyökalun luomien migraatioskriptien on oltava niin luotettavia kuin mahdollista. Migraatiopohjaisen työkalun puolestaan tulee kyetä tunnistamaan päivitettävän tietokannan versio, ja valitsemaan sen pohjalta mitkä migraatiot suoritetaan. Käytännössä kaikki migraatiopohjaiset työkalut tekevät tämän lisäämällä tietokantoihin lokitaulun, jossa pidetään kirjaa suoritetuista päivityksistä ja tietokannan nykyisestä versiosta.

3.3.1 Julkaisutyökalujen hyödyt

Tietokannan muutosten julkaisu on selkeä kohde ohjelmistotyökalujen käytölle. Muutosten manuaaliset asennukset useaan ympäristöön yksi kerrallaan vievät aikaa, ja niissä on inhimillisten erehdysten riski. Jokin ympäristöistä saattaa jäädä päivittämättä tai sen päivitys voidaan toteuttaa virheellisesti. Manuaalisella asennuksella päivityksen automaattitestit pitää myös käynnistää manuaalisesti. Prosessin automaatio tekee tietokannan asennuksista luotettavampia ja nopeampia, sekä tuottaa kehittäjille nopeampaa palautetta heidän tekemistään muutoksista. [24]

Virhetilanteita estävät ja niistä toipumista nopeuttavat lisäominaisuuksia ovat päivitystyökaluille arvokkaita. Virheiden riski pienenee, jos työkalu kykenee varmistamaan, ettei päivitettävään tietokantaan ole tehty versionhallinnan ulkopuolisia muutoksia. Päivitysten teko transaktioina tukee myös virheiden välttämistä. Tämä tarkoittaa päivitysten tekoa atomisina kokonaisuuksina, jotka perutaan kokonaan, jos jokin osa epäonnistuu. Virhetilanteista toipumiseen puolestaan on monia eri keinoja, joilla on omat vahvuutensa ja heikkoutensa. Näitä keinoja ovat esimerkiksi varmuuskopion otto ennen päivitystä sekä mahdollisuus määritellä palautusskriptejä, joilla tietokanta voidaan siirtää uudemmasta versiosta takaisin vanhempaan menettämättä tietosisältöä. Useimmat turvallisuutta lisäävät varmistusmenetelmät kuitenkin tekevät päivityksistä työläämpiä luoda tai hitaampia suorittaa. Toipumisen valmistelu voidaan myös toteuttaa erillisenä osana asennusprosessia, käyttäen toista ohjelmistotyökalua tai skriptiä. [24]

3.3.2 Tietokannan julkaisun työkalujen esittelyjä

Migraatiopohjaiset työkalut ovat perustoiminnaltaan yksinkertaisempia kuin tilapohjaiset, sillä niiden tarvitsee vain suorittaa määritellyt migraatioskriptit. Niihin kuuluu paljon kevyitä, pienten kehittäjäryhmien luomia avoimen lähdekoodin ohjelmistoja. Saatavilla olevia migraatiopohjaisia päivitystyökaluja ovat esimerkiksi Datical [48], DbDeploy [49], DbUp [50], Evolve [51], Flyway [52], Knex [53], Liquibase [54], MyBatis Migrations [55], Redgate SQL Toolbelt [56] ja Sqitch [57].

Tilapohjaisilta päivitystyökaluilta vaaditaan migraatiopohjaisia edistyneempää toiminnallisuutta, sillä työkalujen täytyy osata luoda migraatioskriptejä vertaamalla kahta tietokannan tilaa. Monet tilapohjaiset työkalut tarjoavat laajan valikoiman tietokannan hallintaominaisuuksia ja ovat yritysten kehittämiä kaupallisia tuotteita. Myös useat olio-relaatiomappauksen ohjelmat tarjoavat mahdollisuuden päivittää tietokanta automaattisesti, kun olioluokkien määrittelyjä muutetaan. Tilapohjaisia päivitystyökaluja ovat esimerkiksi DB Ghost Change Manager Professional [58], DBmaestro Release Automation [59], Entity Framework Core [36], Redgate SQL Toolbelt [56] ja SQL Server Data Tools [60].

Runsaan määrän takia kaikkia työkaluja ei esitellä. Nämä työkalut ovat kuitenkin erittäin tärkeä osa julkaisuprosessia, joten tässä työssä kuvataan edellä mainituista työkaluista 9. Niistä 4 ovat migraatiopohjaisia, 4 tilapohjaisia ja yksi tukee molempia menetelmiä. Työkalut on valittu suosion, ominaisuuksien ja saatavilla olevan dokumentaation määrän perusteella. Migraatiopohjaisista työkaluista esitellään DbUp, Flyway, Evolve ja Liquibase. Tilapohjaisista työkaluista kuvataan SQL Server Data Tools, Entity Framework Core, Hibernate. Lisäksi esitellään molempia menetelmiä tukeva työkalupaketti Redgate SQL Toolbelt.

DbUp [50] on pääasiassa DbUp-yhteisön [61] kehittämä ilmainen avoimen lähdekoodin .NET-ohjelma tietokantojen migraatiopohjaiseen päivittämiseen. Sitä ohjataan C#-kielellä tai PowerShell-komennoilla. Migraatiot määritellään SQL-skripteinä tai C#-koodina. DbUp on ladattu hieman yli 2 miljoonaa kertaa. Se tukee seuraavia tietokantoja: SQL Server, SQL Azure, SQLite, SQL CE, MySQL, PostgreSQL ja Firebird. Turvallisuuden osalta DbUp tukee transaktioita.

Evolve [51] on pääasiassa ohjelmistokehittäjä Philippe Lécaillonin kehittämä ilmainen avoimen lähdekoodin .NET-ohjelma tietokantojen migraatiopohjaiseen päivittämiseen. Sitä ohjataan C#-kielellä tai komentorivikäskyillä. Ohjelma on Flyway-työkalun inspiroima, ja vastaa tätä läheisesti käyttötavaltaan. Migraatiot määritellään SQL-skripteinä. Evolve on ladattu noin 200 000 kertaa. Se tukee seuraavia tietokantoja: PostgreSQL, SQLite, SQL Server, MySQL, MariaDB, Cassandra ja CockroachDB. Turvallisuuden osalta Evolve tukee transaktioita.

Flyway [52] on Red Gate Software Ltd:n tytäryhtiön Boxfuse GmbH:n kehittämä ilmainen avoimen lähdekoodin Java-ohjelma tietokantojen migraatiopohjaiseen päivittämiseen. Siitä on myynnissä myös maksullinen versio laajennetuilla ominaisuuksilla. Sitä ohjataan Java-kielellä tai komentorivikäskyillä. Migraatiot määritellään SQL-skripteinä tai Java-koodina. Flyway on ladattu yli 10 miljoonaa kertaa. Se tukee 20 eri tietokantaa. Turvallisuuden osalta Flyway tukee transaktioita, ja sen maksullinen versio tukee palautusskriptejä.

Liquibase [54] on Daticalin kehittämä ilmainen avoimen lähdekoodin Java-ohjelma tietokantojen migraatiopohjaiseen päivittämiseen. Siitä on myynnissä myös lisäominaisuuksilla varustettu maksullinen versio. Ohjelmaa ohjataan komentorivikäskyillä. Migraatioskriptien formaatti on SQL, XML, YAML tai JSON. Liquibase on ladattu yli 30 miljoonaa kertaa. Se tukee 14 eri tietokantaa. Turvallisuuden osalta Liquibase tukee varmuuskopioiden ottoa, transaktioita ja palautusskriptejä.

Redgate SQL Toolbelt [56] on Red Gate Software Ltd:n kehittämä maksullinen suljetun lähdekoodin ohjelmistopaketti, johon kuuluu 14 työkalua SQL Server-tietokantojen hallintaan. Tietokantojen automaattiseen päivittämiseen erikoistuu pakettiin kuuluva Redgate SQL Change Automation-työkalu. Se on suljetun lähdekoodin ohjelma tietokantojen päivittämiseen sekä tila- että migraatiopohjaisesti. Ohjelmaa ohjataan PowerShell-komennoilla tai lisäosana julkaisualustoille kuten TeamCity, Octopus Deploy tai Team Foundation Server. Migraatiot tai tietokannan tavoitetila voidaan määritellä SQL-skripteillä tai graafisen käyttöliittymän kautta. Redgaten SQL-työkaluja käyttää 800 000 asiakasta, tarkempaa erittelyä eri työkalujen latausmääristä yritys ei ilmoita. SQL Toolbelt tukee ainoastaan SQL Server-tietokantaa. Turvallisuusominaisuuksina SQL Toolbelt tukee varmuuskopioiden ottoa, transaktioita ja palautusskriptejä.

DB Ghost Change Manager Professional [58] on Innovartis Ltd:n kehittämä maksullinen suljetun lähdekoodin ohjelma tietokantojen tilapohjaiseen versionhallintaan, luomiseen ja päivittämiseen. Sitä ohjataan graafisen käyttöliittymän tai komentorivikäskyjen kautta. Tietokannan tavoitetila määritellään SQL-skripteinä. Latausmääristä ei ole selkeää tietoa. DB Ghost tukee SQL Server- ja Azure SQL-tietokantoja. Ohjelma ei sisällä virhetilanteita estäviä ja niistä toipumista nopeuttavia lisäominaisuuksia.

DBmaestro Release Automation [59] on DBmaestron kehittämä maksullinen suljetun lähdekoodin ohjelma tietokantojen tilapohjaiseen versionhallintaan ja päivittämiseen. Ohjelmistosta on tarjolla hyvin vähän dokumentaatiota ja yksityiskohtaista toimintaperiaatteiden kuvausta. Sitä ohjataan ainakin graafisen käyttöliittymän kautta. Tietokannan tavoitetila määritellään SQL-skripteinä. Ohjelman käyttäjämäärästä ei ole luotettavaa julkista tietoa. Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL ja MariaDB ovat tuettuja tietokantoja. DBmaestro Release Automation kykenee tarkistamaan päivitettävän tietokannan tilan sekä ennen että jälkeen päivityksen. Se tunnistaa poikkeamat versionhallinnassa määritellystä tilasta, ja osaa poikkeamien yksityiskohtien mukaan mukauttaa päivityksen automaattisesti tai varoittaa kehittäjiä epäselvästä tilanteesta. Tämä lisää merkittävästi päivitysten turvallisuutta.

Entity Framework Core [36] on Microsoftin kehittämä ilmainen avoimen lähdekoodin .NET-ohjelma olio-relaatiomappaukseen (object-relational mapping). Sitä voidaan käyttää myös tilapohjaiseen tietokannan päivitykseen. Ohjelmaa ohjataan joko komentorivikäskyillä tai C#-kielellä. Tietokannan tavoitetila määritellään C#-olioina. Latauskertoja Entity Framework Corella on 83 miljoonaa. Ohjelma tukee oletuksena tietokantoja SQL Server, Azure SQL Database, SQLite ja Azure Cosmos DB, minkä lisäksi siihen voidaan liittää kirjastoja, jotka tarjoavat tuen muille tietokannoille. Transaktiot ja päivitysten automaattinen palautus ovat tuettuja päivitysten turvallisuutta parantavia ominaisuuksia.

SQL Server Data Tools [39] on Microsoftin kehittämä ilmainen suljetun lähdekoodin ohjelma tietokantojen tilapohjaiseen suunnitteluun ja päivitykseen. Se on lisäosa Visual Studio-ohjelmistonkehitysympäristöön. Tietokannan tavoitetila määritellään graafisen käyttöliittymän kautta tai SQL-skripteinä. Ohjelman latausmääriä ei ole ilmoitettu. SQL Server Data Tools tukee tietokantoja SQL Server ja Azure SQL. Turvallisuusominaisuuksina se tarjoaa varmuuskopiointia.

4. KEHITETTY JULKAISUNHALLINTAPROSESSI

Tässä luvussa esitellään tässä tutkimuksessa kehitetty prosessi, jolla tietokantojen julkaisunhallintaa voidaan automatisoida. Osiossa 4.1 kuvataan prosessille asetetut tavoitteet. Osiossa 4.2 käydään läpi ehdot ja rajoitteet, joiden puitteissa prosessin tulee toimia. Osiossa 4.3 selostetaan itse prosessi tyypilliselle tietokannalle, mukaan lukien ensisijaisesti käytettävät työkalut ja perustelut niiden valinnalle. Osio 4.4 kuvaa prosessin eri variaatioita ja vaihtoehtoisia työkaluja, sekä millaisissa tilanteissa niiden käyttöä suositellaan.

4.1 Prosessin tavoitteet

Prosessin keskeinen tavoite on tietokantamuutosten luotettavuuden ja ennakoitavuuden lisääminen. Uuden prosessin toivotaan vähentävän virhetilanteita ja lisäävän päivitysten turvallisuutta. Muita tavoitteita ovat työtuntien säästö päivitysten tehokkuutta kasvattamalla, dokumentaation parantaminen sekä päivitysten julkaisunopeuden kasvu. Nämä ovat toivottavia lopputuloksia, mutta vähemmän tärkeiksi kuin luotettavuuden nousu. Tavoitteet asetettiin yhdessä kohdeyrityksen johdon kanssa, yrityksen tarpeisiin ja havaittuihin puutteisiin perustuen.

Edellisten tavoitteiden lisäksi kehitetyn julkaisunhallintaprosessin tulee olla mahdollisimman laajasti hyödynnettävissä yrityksen eri tietokannoille. Tämä lisää kehitetyn prosessin hyödyllisyyttä. Se myös helpottaa tulevaisuudessa ylläpitoa, kun tietokantakehittäjien ei tarvitsisi opetella montaa merkittävästi erilaista prosessia siirtyessään projektista toiseen. Niinpä prosessista kehitetään sellainen, että sillä on yleiskäyttöinen runko ja perusrakenne, sekä eri projektien vaatimusten mukaisesti vaihdettavia toteutuksen yksityiskohtia.

4.2 Prosessille asetetut reunaehdot

Prosessille asetettuja ehtoja ja rajoituksia ovat, että sitä käytetään SQL-relaatiotietokannoilla, versionhallintana tulee olla Git Bitbucket-sivustolla, ja julkaisut tehdään Octopus Deploy-palvelusta. Lisäksi noudatetaan tietokannoille samankaltaista versionhallinnan haarojen käyttöä kuin kohdeyrityksen sovelluksille, ja aiemmin käytössä olleet julkaisuympäristöt säilytetään.

***Kuva 4**: Tyypillinen versionhallinnan haarojen käyttö. Ylhäällä feature-haaroja, keskellä develop-haara ja alhaalla master-haara.*

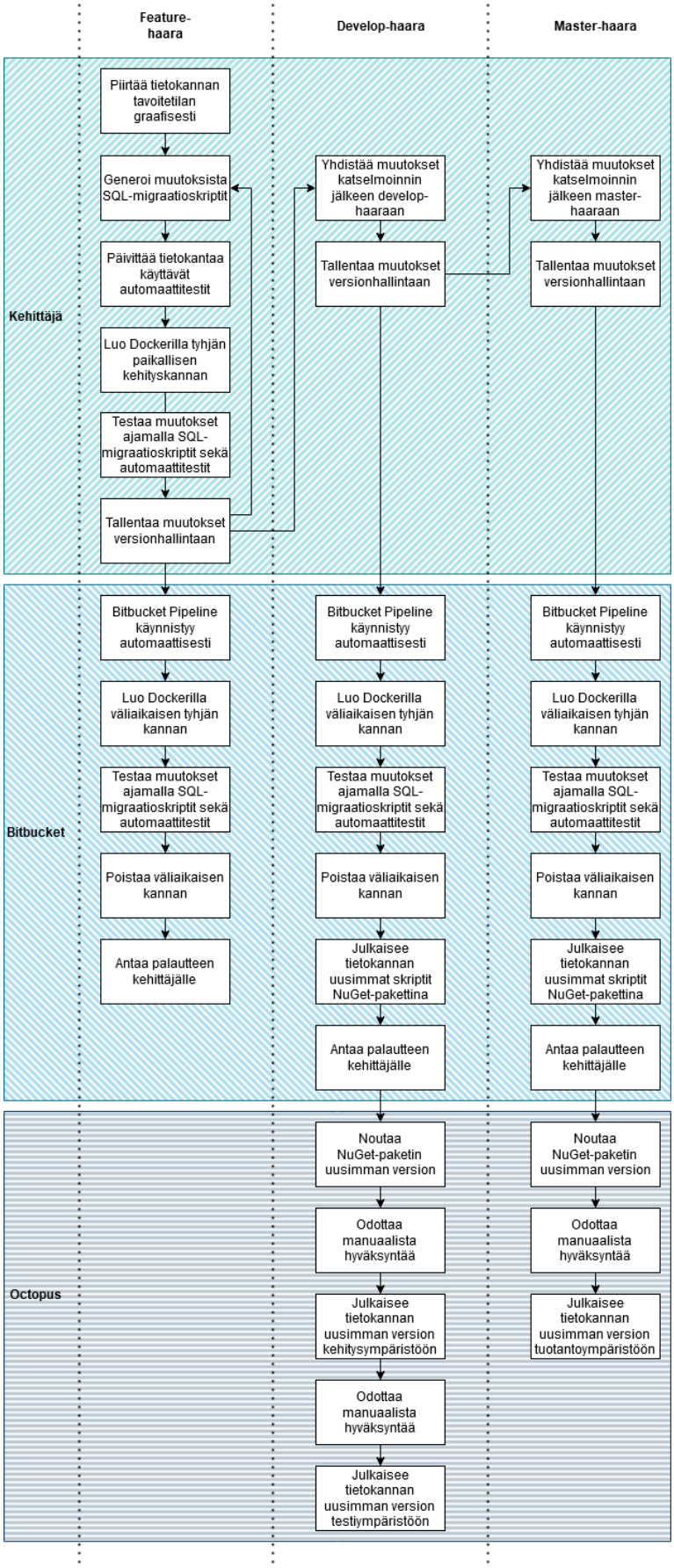
Olemassa oleva versionhallinnan haarojen käyttötapa perustuu siihen, että valtaosa kehitystyöstä tehdään lyhytikäisissä feature-haaroissa, joissa toteutetaan yksittäinen ominaisuus tai pieni muutoskokonaisuus. Ominaisuuden valmistuttua haaran sisältö yhdistetään develop-haaraan, joka sisältää kullakin hetkellä kaikki valmiiksi kehitetyt ominaisuudet. Erityisen pienet ja yksinkertaiset ominaisuudet voidaan toteuttaa myös suoraan develop-haaraan. Asiakkaille julkaistavaksi tarkoitetut versiot erotetaan yhdistämällä kyseinen versio develop-haarasta erilliseen master-haaraan, jonka on tarkoitus sisältää ainoastaan julkaisuvalmiita ohjelmaversioita. Muutosten tekoa suoraan master-haaraan pyritään aina välttämään. Tarkoitus on, että jokainen master-haaran ohjelmistoversio on käynyt läpi monivaiheisen testaus- ja hyväksyntäprosessin. Toimivuuden testaamiseksi kaikissa haaroissa ajetaan jokaiselle tallennetulle muutokselle ohjelmiston automaattitestit, minkä lisäksi yhdistettäessä muutoksia haarasta toiseen sisältö katselmoidaan.

Julkaisuympäristöjen puolesta niin sovelluksille kuin tietokannoille pyritään käyttämään vähintään kolmea ympäristöä eli kehitys-, testi- ja tuotantoympäristöä. Osassa projekteja on käytössä myös neljäs ympäristö staging eli tuotantokopio. Kehitysympäristö on tyypillisesti yrityksen sisäisessä käytössä, ja siellä varmistetaan alustavasti ohjelmistomuutosten toimivuus ennen niiden tarjoamista asiakkaiden nähtäväksi. Julkaisut kehitysympäristöön tehdään usein develop-haarasta. Mikäli kehityksessä ei ilmene ongelmia, ohjelmistoversio julkaistaan testiympäristöön. Asiakkaiden testaajilla on normaalisti pääsy testiympäristöön, ja siellä suoritetaan monipuolista testausta. Tavoitteena on varmistaa entistä kattavammin, että uusi versio täyttää kaikki asiakkaiden tarpeet. Eri projektien käytännöistä ja asiakkaiden mieltymyksistä riippuen julkaisut testiympäristöön tehdään joko develop- tai master-haarasta. Develop-haara tarjoaa uudet ominaisuudet nopeammin asiakkaiden koitettaviksi, kun taas master-haara sisältää viimeistellympiä kokonaisuuksia. Läpäistyään sekä sisäisen että asiakkaiden tekemän testauksen ohjelmistoversio on valmis julkaistavaksi tuotantoon. Julkaisut tuotantoon tehdään aina master-haarasta, ja vain ohjelmistoversioille, joiden toimivuuteen sekä asiakkaalla että kehittäjillä on vahva luottamus.

4.3 Kehitetty prosessi

Tässä tutkimuksessa kehitetyn tietokantojen julkaisunhallintaprosessin päävaiheet ovat tehtävien tietokantamuutosten määrittely, tietokantaskriptien luonti, muutosten testaus

sekä julkaisu eri ympäristöihin. Näistä vaiheista testaus ja julkaisu automatisoidaan pisimmälle. Muutosten määrittelyä ja tietokantaskriptien luontia helpotetaan tarkoitukseen sopivilla työkaluilla, mutta ne vaativat yhä myös manuaalista työtä. Prosessin toiminta on esitetty alla kuvassa 5.



***Kuva 5**: Tietokantamuutosten julkaisunhallintaprosessi tyypilliselle tietokannalle.*

4.3.1 Versionhallinta

Prosessin alussa tietokantamuutosten määrittely tehdään graafisesti tietokantamallia muokkaamalla. Näistä muutoksista generoidaan SQL-migraatioskriptit, joita tarvittaessa muokataan käsin ja jotka sitten säilötään versionhallintaan. Tietokantamallin muokkaukseen ja skriptien generointiin käytetään Navicat Data Modeler-sovellusta. Migraatioskriptien suoritus prosessin loppupäässä tehdään Flyway-työkalulla, mikä asettaa tiettyjä vaatimuksia versionhallintaan säilöttäville migraatioille ja asetustiedostoille. Flywayn käytöstä ja perustelut sen valinnalle kerrotaan tarkemmin luvussa 4.3.3, muut valinnat on perusteltu alla.

Prosessi käyttää migraatiopohjaista versionhallintaa, koska se tukee prosessin päätavoitetta kohdeyrityksen olosuhteissa paremmin kuin tilapohjainen versionhallinta. Migraatioita käytettäessä jokainen tietokantaan tehtävä muutos on helposti katselmoitavissa, ja täsmälleen samat SQL-komennot suoritetaan kaikissa julkaisuympäristöissä. Tämä tekee muutosten testaamisesta helpompaa ja lisää niiden luotettavuutta. Migraatioilla on myös suoraviivaista tehdä monimutkaisia operaatioita kuten tietosisällön muunnoksia ilman erityisjärjestelyjä. Näin tietokantamuutosten luotettavuus ja ennakoitavuus paranee.

Useimmat migraatioiden heikkoudet koskevat tilanteita, joita kohdeyrityksessä kohdataan harvoin. Vakavin huomioitava heikkous on prosessin ulkopuolella tehtävät muutokset tietokantaan. Migraatioita käytettäessä prosessin ulkopuoliset muutokset jäävät helposti huomaamatta, ja ne saattavat aiheuttaa vakavia ja vaikeasti selvitettäviä ongelmia myös pitkän ajan päästä. Niinpä prosessin ulkopuoliset tietokantamuutokset täytyy kieltää täysin, ja huolehtia että tätä kieltoa noudatetaan. Toinen huomioitava migraatiopohjaisen versionhallinnan heikkous on tilanteet, joissa useampi kehittäjä tekee samanaikaisesti samaan tietokannan osaan vaikuttavia muutoksia. Tällöin vaaditaan huolellisuutta, etteivät migraatiot sisällä keskenään ristiriitaisia muutoksia ja että migraatiot suoritetaan oikeassa järjestyksessä. Kolmas heikkous on migraatioskriptien kasaantuminen ajan mittaan, mikä hidastaa uusien tietokantakopioiden luontia. Käytännössä kohdeyrityksessä tietokantamuutoksia tehdään harvakseltaan ja niitä tekee yksi kehittäjä kerrallaan. Niinpä useiden samanaikaisten muutosten aiheuttamia ristiriitoja ei synny, ja migraatioiden määrä kasaantuu hyvin harvoin niin suureksi, että asiaan on tarpeellista puuttua.

Navicat Data Modeler tarjoaa kaikki tietokantamuutosten määrittelyssä ja migraatioiden luonnissa tarvittavat ominaisuudet kilpailijoitaan halvemmalla. Työkalulla luodaan rakennekaavio tietokannan nykytilasta joko lukemalla olemassa olevat migraatioskriptit tai yhdistämällä nykyiseen tuotanto(kopio)kantaan ja skannaamalla sen rakenne. Tämän jälkeen kaaviota muokataan graafisesti haluttuun tilaan. Muokatusta kaaviosta tallenne-

taan sekä kuva tietokannan dokumentaatiota varten, että työkalun generoimat SQL-migraatioskriptit, joilla tietokanta saadaan päivitettyä. Navicat Data Modeler tukee kaikkia kohdeyrityksessä käytössä olevia tietokantatyyppejä. Työkalulla on lisäksi mahdollista verrata migraatioskriptien määrittelemää tietokannan rakennetta todellisen kannan rakenteeseen ja varmistaa niiden olevan samat. Näin voidaan havaita julkaisuprosessin ulkopuolelta tehdyt muutokset ja korjata tilanne. Kilpailevat ohjelmat joko eivät tue kaikkia kohdeyrityksessä käytössä olevia tietokantatyyppejä, eivät tue tietokannan rakennekaavion vapaata muokkausta erillään tietokannasta, tai ovat merkittävästi kalliimpia kuin Navicat Data Modeler.

Versionhallinnassa migraatiot säilötään samaan repositorioon sovelluskoodin kanssa, mikäli tietokanta on tarkoitettu pääasiassa yhden sovelluksen käytettäväksi. Tällöin on selkeää, mitkä sovelluksen ja tietokannan versiot on suunniteltu yhdessä toimiviksi. Molempiin vaikuttavat päivitykset on helppo julkaista yhdessä, käyttäen samaa julkaisuputkea. Sovelluksen ja tietokannan muutosten testaaminen yhdessä on myös suoraviivaista, ja testauksen automaatio helpottuu.

```
master/
├── sql/
│   ├── V1_1__createLogins.sql
│   ├── V1_2__addLinkedServers.sql
├── sql_development/
│   └── V1_0__createDatabase.sql
├── sql_testing/
│   └── V1_0__createDatabase.sql
├── sql_production/
│   └── V1_0__createDatabase.sql
├── flyway.DEVELOPMENT.conf
├── flyway.TESTING.conf
└── flyway.PRODUCTION.conf
adventureWorks/
├── sql/
│   ├── V1_1__createInitialTables.sql
│   ├── V1_2__createInitialViews.sql
├── sql_development/
│   └── V1_3__insertTestData.sql
├── flyway.DEVELOPMENT.conf
├── flyway.TESTING.conf
└── flyway.PRODUCTION.conf
```

***Kuva 6**: Versionhallinnan tiedostorakenne.*

Versionhallintaan säilöttäville tiedostoille käytetään yksinkertaista rakennetta, joka ottaa huomioon Flyway-työkalun asettamat muutamat vaatimukset. Versionhallintaan säilötään kullekin tietokannalle omaan kansioonsa tietokannan migraatioskriptit sekä jokaista julkaisuympäristöä varten oma Flyway asetustiedostonsa. Kuvassa 6, yllä, kansio ”adventureWorks” sisältää tietokannan AdventureWorks migraatioskriptit ja asetustiedostot.

Palvelintason muutoksia tekevät migraatiot säilötään omaan kansioonsa ”master”, ja ne julkaistaan yhdistämällä palvelimilta oletuksena aina löytyvään järjestelmätietokantaan ”master”. Palvelintason migraatioihin lasketaan muun muassa tietokannan luontiskripti CREATE DATABASE, uusien käyttäjätunnusten lisääminen palvelimelle sekä SQL Server Agent-tehtävien tai Linked Server-yhteyksien muokkaus. Tällaisten migraatioiden säilöminen omaan kansioonsa ja erillisten yhteysasetuksien käyttö niiden julkaisuun on perusteltua kahdesta syystä. Ensinnäkin varsinaisen kohdetietokannan luontiskripti ja muut tarpeelliset valmistelevat komennot palvelimella täytyy suorittaa, ennen kuin kyseiseen tietokantaan voidaan muodostaa yhteys. Mikäli tietokannan luontiskripti asetettaisiin versionhallinnassa muiden saman tietokannan migraatioskriptien joukkoon, tarvittaisiin kuitenkin järjestelmä, jossa tietokannan luonti tehdään yhdistämällä johonkin jo valmiiksi olemassa olevaan kantaan ja tämän jälkeen muut migraatioskriptit suoritettaisiin ottamalla yhteys juuri luotuun tietokantaan. Toiseksi järjestelmätietokanta master on luonteva paikka palvelintason migraatioiden versiohistoriataululle. Flyway pitää kirjaa suoritetuista migraatioista luomalla tarkoitusta varten tietokantataulun aina siihen tietokantaan, johon yhdistettynä migraatiot ajetaan. Mikäli useamman samalle palvelimelle asennetun tietokannan Flyway-migraatiot tekisivät palvelintason muutoksia ja näistä muutoksista kirjattaisiin historiatieto kunkin tietokannan sisäiseen versiohistoriatauluun, olisi palvelimella lopputuloksena vaikeaa saada kokonaiskuva kaikista tehdyistä muutoksista. Sen sijaan tekemällä palvelintason muutokset aina niin, että versiohistoria päivitetään master-tietokantaan, koko palvelimen yhtenäinen muutoshistoria on helposti nähtävillä.

Jokaisen migraatioskriptin nimi sisältää kyseisen skriptin versionumeron sekä kuvauksen muutoksesta, esimerkiksi ”V1_3__insertTestData.sql”. Kukin skripti sisältää yhden tietokantamuutoksen, lukuun ottamatta ensimmäisiä skriptejä, jotka alustavat tietokannan. Varsinkin isoille olemassa oleville tietokannoille versionhallintaa käyttöön otettaessa luodaan jokaiselle tietokannan osiolle kuten tauluille tai näkymille yksi skripti, sen sijaan että luotaisiin jokaiselle taululle tai näkymälle oma skriptinsä mistä seuraisi tarpeettoman suuri määrä migraatioskriptejä. Kaikki samaa tietokantaa koskevat migraatioskriptit säilötään lähtökohtaisesti samaan alikansioon ”sql”. Rakenteen tarkoitus on olla mahdollisimman yksinkertainen ja suoraviivainen, jolloin sen käyttö ja muutosten seuranta pysyy helppona. Erityistapauksissa jonkin skriptin sisältöä on pakko vaihdella ympäristön mukaan, esimerkiksi kun luodaan käyttäjätili ja asetetaan sille eri salasana eri ympäristöissä. Tämä tehdään mieluiten käyttäen Flyway Placeholder-muuttujia. Tämän ollessa mahdotonta, esimerkiksi monimutkaisen tietokannan luontiskriptin tapauksessa, luodaan eri julkaisuympäristöille omat alikansiot ja tallennetaan kuhunkin kansi-

oon sitä vastaavaan ympäristöön sopiva version skriptistä. Ympäristökohtaisissa asetustiedostoissa määritetään tietokantojen yhteysasetukset, Placeholder-muuttujien arvot, sekä kansiot, joista migraatioskriptit kuhunkin ympäristöön luetaan.

4.3.2 Muutosten testaus

Tietokantamuutosten testaukseen kuuluu prosessissa useita eri testaustyyppejä, joita tehdään prosessin eri vaiheissa. Muutosten testaus koostuu migraatioiden ajamisesta, tietokantaa käyttävien automaattitestien ajosta, sekä manuaalisesta testauksesta. Näitä testausvaiheita tehdään paikallisesti ennen muutosten tallennusta versionhallintaan, automaattisesti versionhallintaan tallennuksen jälkeen, sekä manuaalisesti kehitys- ja testiympäristöissä ennen julkaisua tuotantoon.

Ensimmäisen kerran tietokantamuutokset testaa niiden kehittäjä omassa paikallisessa kehitysympäristössään ennen muutosten tallennusta versionhallintaan. Luotuaan migraatioskriptit kehittäjä perustaa tyhjän tietokannan, joko manuaalisesti tai käyttäen automatisoitua Docker-kontin sisäisen virtuaaliympäristön pystytystä. Kehittäjä ajaa migraatiot tähän tyhjään kantaan ja tarkistaa, että kaikki migraatiot onnistuivat ja lopputuloksena kanta päätyi tilaan, johon sen oli tarkoitus päätyä. Lisäksi hän ajaa tietokannan toimintaa testaavat automaattiset yksikkötestit, sekä tietokantaa käyttävien sovellusten integraatiotestit. Varmistuttuaan näin migraatioiden toimivuudesta hän tallentaa ne versionhallintaan. Noudattamalla tätä järjestelyä voidaan odottaa, että versionhallintaan tallennetuissa migraatioissa ei ole räikeitä virheitä ja että tietokantaan yhteydessä olevat sovellukset toimivat päivitetyn tietokannan kanssa ainakin automaattitestien kattamien kokonaisuuksien osalta.

Versionhallintapalveluun tallennuksen jälkeen muutosten toimivuus varmistetaan uudelleen automaattisella testausprosessilla. Bitbucket-sivusto tarjoaa mahdollisuuden määrittää Pipelinen, automaattisen prosessin, joka suoritetaan aina kun versionhallintaan tallennetaan muutoksia. Tietokannan julkaisunhallintaa varten luodaan Pipeline-prosessi, joka perustaa Docker-kontin sisälle tyhjän tietokannan, päivittää sen ajamalla kaikki versionhallintaan säilötyt migraatiot, ja sen jälkeen suorittaa kaikki tietokantaa käyttävät automaattitestit tätä Docker-kontin sisäistä kantaa vasten. Kyseessä on käytännössä sama prosessi, jonka kehittäjät tekevät ennen muutosten tallennusta versionhallintaan, mutta ilman manuaalista testausta. Testaamalla migraatioiden ja automaattitestien toimivuuden joka muutokselle automaattisesti ajettavassa Bitbucket Pipelines-prosessissa varmistetaan, ettei kehittäjän unohdus tai huolimattomuus päästä testaamattomia muutoksia julkaistavaksi. Mikäli jokin testi epäonnistuu Pipelinessä, kehittäjälle lähetetään virheilmoitus ja kyseisiä muutoksia ei päästetä eteenpäin julkaisunhallintaan ennen kuin vika on korjattu.

Lopuksi muutokset testataan huolellisesti julkaisemalla ne ensin kehitys- ja sitten testiympäristöön. Molemmissa pidetään asennettuina sekä tietokantoja että niitä käyttäviä sovelluksia. Asentamalla tietokantamuutokset näihin ympäristöihin voidaan testata manuaalisesti sekä kannan että sovellusten toimivuus. Kehitysympäristössä muutoksia testataan yrityksen sisäisesti, kunnes niiden toimivuus on varmistettu riittävästi, että niitä voidaan tarjota asiakkaan testattavaksi. Testiympäristössä muutokset ovat myös asiakkaan testattavina, eikä julkaisua tuotantoon tehdä ennen kuin asiakas on vakuuttunut päivityksen laadusta ja antanut luvan sen asennukselle tuotantoon. Ennen tietokantamuutosten julkaisua tuotantoon testataan ainakin, että muutokset eivät riko mitään osaa tietokantaa käyttävistä sovelluksista, muutokset eivät merkittävästi haittaa tietokannan suorituskykyä tai kykyä toimia kovassa rasituksessa, ja että muutokset eivät aiheuta tiedon menetystä tai muuta vauriota tietokannalle.

Tietokannan yksikkötestaukseen käytetään samankaltaisia testejä kuin tietokantaa käyttävälle sovellukselle. Näin sovelluksen ja tietokannan testit voidaan helposti määrittää ajettaviksi yhdessä, eikä kehittäjien tarvitse vaihdella kahden erilaisen testaustavan välillä. Mikäli tietokantaa käyttää esimerkiksi C#-sovellus, jolla on jo omia NUnit-yksikkötestejä, lisätään niiden ohelle omaan testiprojektiinsa samankaltaisia NUnit-testejä, joissa yhdistetään tietokantaan ja testataan sen sisäistä toimintaa.

Testiympäristön suorituskykytestausta varten testitieto tuotetaan kopioimalla sitä tuotannosta Jailer-työkalulla. Näin saadaan paremmin todellisuutta vastaavaa testitietoa kuin puolisatunnaisilla generointityökaluilla. Tuotannon sisältöä kopioitaessa on kuitenkin oltava tarkkana, että kaikki arkaluontoinen sisältö peitetään. Esimerkiksi oikeiden asiakkaiden henkilötietoja ei tule paljastaa testiympäristössä.

4.3.3 Julkaisut

Tietokantojen julkaisunhallintaprosessin viimeisenä osuutena tietokantamuutokset julkaistaan automatisoidusti eri ympäristöihin. Julkaisut tehdään Octopus Deploy-palvelusta Flyway-työkalua käyttävillä komentorivikäskyillä. Muutokset julkaistaan ensin kehitysympäristöön, sitten testiympäristöön ja lopuksi tuotantoympäristöön. Julkaisu jokaiseen ympäristöön pitää käynnistää manuaalisesti, mutta sen suoritus tapahtuu täysin automaattisesti.

Jotta julkaisut voidaan tehdä, tietokannan migraatio- ja asetustiedostot siirretään versionhallinnasta NuGet-tiedostopakettina yrityksen sisäiseen NuGet-varastoon, missä ne ovat Octopus Deploy-palvelun käytettävissä. Tämä tehdään versionhallinnan Bitbucket Pipeline-automaattiprosessissa sen jälkeen, kun automaattitestit on suoritettu onnistuneesti. NuGet-tiedostopaketteja julkaistaan ainoastaan versionhallinnan develop- ja master-haaroista, sillä feature-haarat sisältävät keskeneräisiä ominaisuuksia, jotka eivät

ole valmiita otettavaksi käyttöön. Tiedostot välitetään NuGet-paketteina, koska Octopus tukee niiden käyttöä hyvin ja tutkimuksen kohdeyrityksellä on jo käytössään sisäinen NuGet-varasto. NuGet-paketit myös muistuttavat rakenteeltaan hyvin läheisesti ZIP-paketteja, eli ne ovat kevyitä ja yksinkertaisia.

Tietokantamuutokset julkaistaan ajamalla migraatioskriptit kohdetietokantaan Flyway-työkalulla käyttäen yksinkertaista komentorivikäskyä. Flyway on suoraviivainen työkalu, jonka toiminta noudattaa migraatiotyökaluille tyypillisiä periaatteita. Tietokannan päivittäminen vaatii vain yhden komentorivikäskyn, jossa Flywaylle annetaan parametreinä tietokantaan yhdistämiseen vaadittavat tiedot sekä tiedostokansiot, joista migraatioskriptit löytyvät. Nämä tiedot, kuten muutkin Flywayn asetukset, voidaan säilöä yksinkertaisiin asetustiedostoihin. Kun migraatioita ajetaan johonkin tietokantaan ensimmäisen kerran, Flyway luo kantaan päivityshistoriataulun. Tähän tauluun päivitetään automaattisesti jokaisen migraation suorituksen yhteydessä tiedot, mitkä migraatiot tietokantaan on ajettu ja milloin. Ennen migraatioiden suorittamista Flyway lukee taulun sisällön ja päättelee sen perusteella, mitkä saatavilla olevista migraatioskripteistä on jo ajettu kantaan. Näitä migraatioita ei ajeta uudelleen, vaan ainoastaan aiemmin suorittamattomat migraatioskriptit ajetaan. On kuitenkin mahdollista luoda myös toistettavia skriptejä, jotka suoritetaan jokaisen suorituskerran alussa tai lopussa.

Flyway voidaan määrittää automaattisesti poistamaan kohdetietokannan ennen uuden version asennusta. Tämä on suositeltavaa kehitys- ja mahdollisesti testiympäristössä, jotta vältetään mahdollisten julkaisuprosessin ulkopuolella tehtyjen muutosten aiheuttamat ongelmat. Tietokannan jatkuvaa poistoa ja uudelleenluontia ei kuitenkaan suositella sellaisissa testiympäristöissä, joissa on esimerkiksi suorituskykytestausta varten säilöttynä suuret määrät tietoa. Tämän tietosisällön toistuva poistaminen ja uudelleen tuottaminen hidastaa julkaisuprosessia. Tuotantokannan poistamista päivityksen yhteydessä ei myöskään suositella missään olosuhteissa.

Flyway valittiin prosessiin koska se on erityisen helppokäyttöinen ja tarjoaa samalla laajan valikoiman asetuksia ja ominaisuuksia, joilla tietokantojen tilaa voidaan seurata ja migraatioiden suorittamisen yksityiskohtia hallita. Tässä luvussa listataan joukko huomionarvoisia ominaisuuksia. Migraatioita voidaan manuaalisesti merkitä tietokantaan jo suoritetuiksi, jos niiden kuvaamat muutokset on tehty ilman Flywayta. Migraatioskriptejä voidaan tarvittaessa ajaa normaalista poikkeavassa järjestyksessä. Päivitettävän tietokannan tyypistä ja migraatioskriptin sisällöstä Flyway päättelee automaattisesti, suoritetaanko migraatio transaktion sisällä vai ei. Transaktiot voi myös manuaalisesti pakottaa päälle tai pois, mukaan lukien yksittäisille skripteille. Migraatioskriptien SQL-komentoihin voidaan sisällyttää muuttujia, joiden arvo vaihtelee ympäristökohtaisesti. Migraatioita suoritettaessa voidaan määritellä tavoiteversio, jota pidemmälle migraatioita ei suoriteta.

Päivityshistoriataulua lukiessaan Flyway varmistaa samalla tarkistussummia käyttäen, että aiempia migraatioskriptejä ei ole muokattu tietokantaan ajamisen jälkeen. Jos skriptejä on muokattu, on olemassa vaara, että tietokantaan ajetut skriptit eivät ole samat kuin versionhallintaan tallennetut. Tällöin Flyway varoittaa käyttäjää riskialttiista tilanteesta eikä päivitä kantaa ennen kuin tilanne on tarkistettu ja kuitattu ratkaistuksi.

Octopus Deploy-palvelun rajoitteiden vuoksi jokainen julkaisu täytyy käynnistää manuaalisesti. Käytännössä tämä tarkoittaa muutamaa napin painallusta, joilla valitaan julkaistava NuGet-paketin versio sekä ympäristö johon julkaisu tehdään. Täysin automaattisia julkaisuja Octopuksesta ei voida toteuttaa, koska niitä tuetaan vain, jos NuGet-paketit noudetaan Octopus-palvelun sisäisestä pakettivarastosta. Kohdeyrityksessä ei haluta käyttää tätä varastoa, joten julkaisujen täysautomaatio ei ole vaihtoehto. Julkaisuprosessilla ei siis saavuteta täysin jatkuvan julkaisun tai edes jatkuvan toimituksen määritelmää, sillä julkaistavat tietokantapäivitykset eivät kulje automaattisesti eri julkaisuympäristöjen läpi tuotantoympäristölle saakka. Toisaalta vaatimalla manuaalisen hyväksynnän jokaiselle asennukselle säilytetään kehittäjillä suurempi kontrolli julkaisujen aikataulusta. Näin vältetään esimerkiksi tilanteet, joissa esimerkiksi julkaistaisiin testiympäristöön epähuomiossa tietokantapäivitys, kun edellisen version manuaalinen testaus on yhä kesken.

4.4 Prosessivariantit

Edellisessä osiossa on kuvattu automaattinen tietokannan julkaisuprosessi tyypilliselle tietokantaprojektille. Prosessin on kuitenkin tarkoitus olla mahdollisimman yleiskäyttöinen, joten sen toteutuksessa on otettava huomioon erilaisten tietokantaprojektien väliset erot. Tässä osiossa käydään läpi vaihtoehtoisia toteutuksia prosessin eri osille. Kustakin vaihtoehdosta kuvataan, mitä seurauksia sillä on tyypilliseen toteutukseen verrattuna, ja millaisissa tilanteissa sen käyttö on suositeltavaa.

Versionhallinnassa käytetään tietokannalle omaa repositoriota, mikäli tietokantaa käyttää useampi eri sovellus. Tällöin tietokannan ja sovellusten versiopäivitysten ja yhteensopivuuden koordinointi muuttuu haastavammaksi, samoin kuin niiden automaattitestaaminen yhdessä. Kuitenkin erilliset repositoriot ovat ainoa järkevä vaihtoehto tilanteessa, jossa tietokanta on oma itsenäisesti päivitettävä kokonaisuutensa, jonka ei voida katsoa olevan osa mitään tiettyä sovellusta. Kun tietokanta ja sovellukset ovat eri repositorioissa, on erityisen haastavaa saada Bitbucket Pipeline-prosessissa kaikkien tietokantaa käyttävien sovellusten automaattitestit ajettua virtuaaliympäristöön luotua tietokantaa vasten. Ongelma muuttuu vielä vaikeammaksi, kun sovellusten automaattitestit tarvitsevat muokkausta tietokantamuutoksen seurauksena. Tällöin Pipelinen pitäisi automaatti-

sesti päätellä, mistä kehityshaarasta löytyy kunkin sovelluksen oikea versio, joka on tarkoitettu yhteensopivaksi nykyisen tietokantaversion kanssa. Erilliseen repositorioon säilötty tietokanta pakottaa siis joko kehittämään huomattavasti monimutkaisemman järjestelmän sovellusten automaattitestien ajoon, tai panostamaan enemmän tietokannan omiin yksikkötesteihin sekä julkaisuprosessin muihin testauksen osiin.

Versionhallinnassa säilöttävien ympäristökohtaisten Flyway-asetustiedostojen sijaan asetuksia voidaan hallita Octopus-palvelussa ympäristökohtaisina muuttujina. Kyseessä on pitkälti makuasia. Mikäli asetukset on säilötty versionhallintaan, ne ovat helposti nähtävillä kaikille tietokantamuutoksien kehittäjille. Valmiilla asetustiedostoilla on myös helppoa yhdistää tietokantoihin ja ajaa Flyway komentorivikäskyjä esimerkiksi poikkeuksellisten ongelmatilanteiden ratkaisemiseksi. Octopus-palveluun säilöttyjä asetuksia puolestaan voi muokata kevyemmin, tallentamatta muutoksia versionhallintaan ja laukaisemalla täten koko automaattista julkaisuprosessia. Mikäli osana kehitysprosessia pidetään paikallisten kehityskantojen perustaminen ja niissä muutosten testaus, tarvitaan versionhallintaan joka tapauksessa valmiit asetukset, joilla migraatiot saadaan ajettua näihin paikallisiin tietokantoihin.

Migraatioskriptien suoritukseen käytetään Evolvea, jos tilanne ei salli Flywayn käyttöä. Tämä tarve voi syntyä esimerkiksi, jos Flyway ei tue julkaisunhallinnan kohteena olevaa tietokantaa. Toinen mahdollinen tapaus on, jos tietokannan päivitys vaatii muutoksia joita SQL-komennoilla ei voida suorittaa, eikä Flywayn toisena vaihtoehtona tarjoaman Java-koodin suoritus ole mahdollista. Evolve on valittu julkaisuprosessin varatyökaluksi, koska sen käyttö- ja toimintatavat vastaavat läheisesti Flywayta. Siitä löytyy myös valtaosa kaikista Flywayn tukemista ominaisuuksista, vaikkakin joitain mukavuuksia ja pieniä ominaisuuksia puuttuu. Projektien muuntaminen yhden työkalun käytöstä toiseen on varsin helppoa, ja kehittäjät, jotka ymmärtävät yhden työkalun käytön pystyvät nopeasti sisäistämään myös toisen.

Julkaisuprosessiin voidaan lisätä tai poistaa julkaisuympäristöjä. Suositeltavin muutos on ottaa käyttöön neljäntenä julkaisuympäristönä tuotantokopio- eli staging-ympäristö. Sen tarkoitus on vastata mahdollisimman läheisesti tuotantoa, ja toimia julkaisuprosessin viimeisenä testausvaiheena, jossa huomaamattomimmatkin tietokantapäivityksen virheet paljastuvat. Tuotantokopiokanta on myös hyvä kohde suorituskykytestaukselle. Koska se muistuttaa mahdollisimman läheisesti tuotantoa, siellä saadaan muita ympäristöjä luotettavampaa tietoa tietokantajulkaisujen vaikutuksesta kannan suorituskykyyn. Tuotantokopiokannassa on myös normaalia pitää suurin piirtein samaa tietosisältöä kuin tuotannossa. Julkaisut tuotantokopioympäristöön tehdään master-haarasta, prosessin viimeisenä vaiheena ennen tuotantojulkaisua. Julkaisuympäristöjen määrän muuttaminen on teknisesti hyvin yksinkertainen muutos prosessiin, mutta on tärkeää määritellä

selkeästi jokaisen ympäristön tarkoitus. Samalla tulee huolehtia, että tietokantamuutokset testataan yhä riittävän kattavasti ja johdonmukaisella tavalla.

5. TAPAUSESIMERKKI: JULKAISUNHALLINTA-PROSESSIN TOTEUTUS

Tutkimuksen tapausesimerkkinä edellisessä luvussa kuvattu tietokannan automaattinen julkaisunhallintaprosessi otettiin käyttöön yhdelle tietokannalle. Tapausesimerkkinä käytetty tietokanta on jo pitkään käytössä ollut kanta, johon oli suunniteltu rakennemuutoksia. Uusi julkaisunhallintaprosessi haluttiin ottaa käyttöön tälle tietokannalle ennen muutosten tekoa. Näin suunniteltujen muutoksien tekoon saadaan prosessin hyödyt, ja samalla itse prosessia testataan tositilanteessa.

Tutkimuksen kohteena olleeseen tietokantaan toteutettiin versionhallinta sekä jatkuva toimitus juuri perustettuun tuotantokopioympäristöön. Lisäksi toteutettiin puoliautomaattiset julkaisut tuotantokantaan, virtuaaliympäristöjen automatisoitu luonti Docker-konttien avulla sekä rakennettiin pohja, jonka päälle automaattitestausta on myöhemmin helppo toteuttaa. Jatkuva integraatio Bitbucket Pipeline-automaattiprosessissa osoittautui mahdottomaksi esimerkkitietokantaa käyttävien sovellusten suuren määrän vuoksi. Jatkuvaa toimitusta testi- ja tuotantoympäristöihin ei myöskään voitu toteuttaa, johtuen palvelinten yhteys- ja tietoturva-asetuksista. Julkaisut tuotantoon voidaan kuitenkin tehdä puoliautomaattisesti komentorivikäskyllä. Julkaisunhallintaprosessin toteutuksessa päätettiin muutamassa kohtaa poiketa prosessin perusmuodosta ja käyttää luvussa 4.4 kuvattuja vaihtoehtoisia toteutustapoja.

Tässä luvussa käydään vaiheittain läpi edellisen luvun julkaisunhallintaprosessin käytännön toteutus tapausesimerkkinä olleeseen tietokantaan. Osiossa 5.1 esitellään kohdetietokanta lähtötilanteessa, osiossa 5.2 kuvataan versionhallinnan toteutus kohdekannalle, osiossa 5.3 selostetaan kannalle käyttöönotetut testausvaiheet ja osiossa 5.4 selitetään, miten julkaisut eri ympäristöihin toteutettiin.

5.1 Kohdekanta

Automaattinen julkaisunhallinta toteutettiin vuonna 2015 perustettuun Microsoft SQL Server 2012 tietokantaan. Tämä esimerkkitietokanta on varsin suuri, vieden lähes 200 gigatavua muistia ja sisältäen noin 400 taulua, noin 650 säilöttyä proseduuria (stored procedure) ja lähes tuhat käyttäjätiliä. Näiden lisäksi kanta sisältää näkymiä (view), indeksejä (index) ja laukaisimia (trigger). Vaikka kyseessä on SQL Server 2012 tietokanta, sille on asetettu Compatibility Level eli yhteensopivuustaso 90. Se on siis yhteensopiva vanhempien tietokantaversioiden kanssa versioon SQL Server 2005 asti, mutta ei SQL

Server 2014 tai sitä uudempien versioiden kanssa. Tietokanta on riippuvainen palvelimelle määritetystä Linked Server-yhteydestä toiseen tietokantaan, jota ilman osa kantaan säilötyistä proseduureista ei toimi.

Tapausesimerkin kohdekantaa käyttää muutama kymmenen eri sovellusta, joista valtaosa oli C#-kielellä toteutettuja. Sovellusten dokumentaation taso vaihtelee. Yleisesti ottaen niillä on vähän tai ei lainkaan automaattisia integraatiotestejä, joilla varmistettaisiin sovellusten toiminta yhdessä tietokannan kanssa.

Tapausesimerkkitietokannassa on Linked Server-yhteys toiseen tietokantaan, jota ilman kanta ei toimi oikein. Alun perin tapausesimerkkikanta perustettiin eriyttämällä osa vanhempaa, liian laajaksi kasvanutta tietokantaa omaksi kannakseen. Tämä toteutettiin käytännössä kopioimalla alkuperäinen tietokanta ja poistamalla kopiosta uuteen tarkoitukseen selvästi tarpeettomat osat. Tämän jälkeen kopiota muokattiin ja siihen lisättiin osia, kunnes lopputulos sisälsi kaiken uudelle tietokannalle tarpeellisen. Kantoja ei eriytetty täysin toisistaan, sillä kummastakin kannasta siirretään säännöllisesti sisältöä toisiinsa. Näitä siirtoja varten kannoissa on määritetty Linked Server-yhteydet toisiinsa, jota säilötyt proseduurit kummassakin tietokannassa käyttävät.

Johtuen esimerkkitietokannan luontitavasta ja myöhemmistä muutoksista huomattava osa kannan rakenteesta on nykyään tarpeetonta. Tarpeettomia tauluja on tietokannassa vanhempien kehittäjien arvioiden mukaan noin 100-200. Tätä ylimääräistä sisältöä ei kuitenkaan ole siivottu pois. Tietokannan osien karsiminen olisi riskialtista, sillä dokumentaatio on puutteellista ja tietokannan testikattavuus erittäin matala. On osittain epävarmaa, mitkä kannan osat ovat yhä tarpeellisia. Joitain tietokantatauluja käytetään ainoastaan kuukausittain tai vuosittain ajettavissa ajastetuissa prosesseissa, joten näennäisesti käyttämättömän taulun poistaminen voisi aiheuttaa vakavia ongelmia pitkän ajan kuluttua.

Tuotantokannan lisäksi käytössä on testikanta. Tämä kanta sijaitsee samalla palvelimella kuin tuotantokanta, ja muutaman kerran vuodessa se korvataan suoralla kopiolla tuotantokannasta, jotta erot näiden kahden kannan välillä saadaan poistettua. Kopiointien välissä testikantaan tehdään kokeellisia ja väliaikaisia muutoksia, joiden vuoksi tuotanto- ja testitietokantojen rakenteissa on suurimman osan ajasta eroja.

Tämän tutkimuksen alkaessa tapausesimerkkinä käytetyllä tietokannalla ei ollut käytössä standardoitua julkaisunhallintaprosessia. Kehittäjät tekivät tietokantamuutokset manuaalisesti, yhdistämällä tietokantoihin ja tekemällä muutokset joko kirjoittamalla käsin SQL-komentoja tai muokkaamalla tietokannan rakennetta SQL Server Management Studio-sovelluksella. Tietokantamuutokset tehtiin ensin testiympäristöön. Siellä niitä tes-

tattiin manuaalisesti, minkä jälkeen muutokset tehtiin tuotantokantaan. Testi- ja tuotantokannan välisten erojen aiheuttamia riskejä tietokantapäivitysten onnistumiselle ei huomioitu järjestelmällisesti.

Julkaisunhallintaprosessin käyttöönottoa suunniteltaessa päätettiin tapausesimerkkinä käytettävälle tietokannalle ottaa käyttöön uusia julkaisuympäristöjä. Yhden uuden ympäristön tarkoitus on toimia tuotantokopiona ja viimeisenä muutosten testauspaikkana ennen tuotannon päivitystä. Tämä on myös ainoa pysyvä tietokanta, jota automaattisella julkaisunhallintaprosessilla voidaan päivittää, sillä palomuuri estää yhteyden muodostaminen Octopus Deploy-palvelun ja tuotanto/testitietokannat sisältävän palvelimen välillä. Toisena lisäyksenä käyttöön otettiin Docker-kontissa ajettava virtuaalinen kehitys- ja testiympäristö. Koska kyseessä on virtuaalinen kontti, sen kopioita voidaan helposti ja nopeasti luoda ja poistaa tarpeen mukaan. Esimerkiksi jokaista automaattitestien ajoa varten voidaan luoda oma konttinsa, jossa testit ajetaan puhtaalta pöydältä ja joka poistetaan testauksen lopuksi. Tämä ympäristö on välttämätön migraatioskriptien testaukselle Bitbucket Pipeline-automaattiprosessissa, ja sitä voidaan käyttää myös paikalliseen kehitys- ja testaustyöhön esimerkiksi ennen migraatioiden tallentamista versionhallintaan.

5.2 Versionhallinnan toteutus

Valmiiksi käytössä olevalle tietokannalle versionhallinnan käyttöönoton ensimmäinen askel on tuottaa ja tallentaa versionhallintaan migraatioskriptit, joilla saadaan luotua tyhjästä tietokanta, jonka rakenne vastaa täydellisesti nykyhetkistä tuotantokantaa. Ilman tätä vaihetta versionhallinta ei olisi luotettava totuuden lähde ja kattava dokumentaatio tietokannan rakenteelle. Tällöin olisi mahdotonta luoda pelkän versionhallinnan pohjalta kopioita tuotantokannasta. Täten puutteellinen versionhallinta tekisi sekä tietokantamuutosten testauksesta että ensimmäisistä julkaisuista uusiin ympäristöihin huomattavasti vaikeampaa. Samoin kuin sovelluksen versionhallinnasta täytyy löytyä kaikki sovelluskoodi, tietokannan versionhallinnasta täytyy löytyä skriptit tietokannan jokaiselle osalle.

Tapausesimerkkinä käytetylle tietokannalle luotiin versionhallinnassa oma repositorio. Tämä ei ole kehitetyssä julkaisunhallintaprosessissa ensisijaisesti suositeltu ratkaisu, mutta sitä voidaan käyttää, kun tietokanta on useiden eri sovellusten käytössä. Esimerkkitietokantaa käyttävät kymmenet eri sovellukset, joten valinta oli selvä. Tietokantaa ei voitu tulkita osaksi minkään tietyn sovelluksen kokonaisuutta, vaan se on oma itsenäinen komponenttinsa, josta monet sovellukset ovat riippuvaisia.

Ensimmäisenä työvaiheena tuotantokannasta generoitiin koko tuotantokannan rakenteen luova skripti SQL Server Management Studiolla. Tämä koko rakenteen luontiskripti siivottiin ja pilkottiin osiin, muodostaen joukon erillisiä skriptejä, joista kukin loi tietyn osan tietokannasta. Yksi skripti taulujen luonnille, yksi näkymille, yksi tietokannan asetusten

säätämiselle ja niin edelleen. Lisäksi itse tietokannan luontikomennolle sekä tietokannan toiminnalle tarpeellisten palvelintason olioiden ja asetusten muokkauksille muodostettiin omat skriptinsä samankaltaisella prosessilla. Kehitetylle julkaisunhallintaprosessille tyypilliseen tapaan itse tietokannan skriptit säilöttiin yhteen kansioon ja ennen tietokantaan yhdistämistä ajettavat palvelintason skriptit säilöttiin toiseen.

Valtaosa luoduista migraatioskripteistä ovat ympäristöstä riippumattomia. Itse tietokannan luontikomennosta ja Linked Server-tietokantayhteyden luonnista tarvittiin kuitenkin ympäristökohtaiset skriptit. Näitä komentoja ei siis voida ajaa samanlaisina kaikissa julkaisuympäristöissä. Jotta alkuperäinen tuotantokanta olisi mahdollisimman tarkasti säilöttynä versionhallintaan, säilytettiin tietokannan luontikomennosta tuotantoympäristöön kohdistettu versio. Skripti määrittää muun muassa tietokannan sisältö- ja lokitiedostojen sijainnin ja koon. Tätä skriptiä ei kuitenkaan voi käyttää luotettavasti kaikissa ympäristöissä, joten se asetettiin muista skripteistä erilleen tuotantoympäristökohtaisten skriptien kansioon, ja muille ympäristöille luotiin kullekin oma kansionsa ja niihin korvaava tietokannan luontikomento. Samoin tuotannossa on käytössä Linked Server-yhteys toiseen tuotantotietokantaan. Muista julkaisuympäristöistä ei tule muodostaa yhteyttä tuotantokantoihin, joten niille luotiin kullekin omat ympäristökohtaiset versionsa Linked Server-yhteyden määrittelevästä skriptistä.

Migraatioskriptien lisäksi versionhallintaan tallennettiin ympäristökohtaiset Evolve asetustiedostot, sekä Dockerin ja Bitbucket Pipelinen vaatimat määrittelytiedostot. Evolven asetuksissa määritellään kullekin julkaisuympäristölle yhteysasetukset, mistä kansioista migraatioskriptit etsitään, sekä sekalaisia yksityiskohtia kuten skripteissä käytetty tekstin enkoodaus ja Evolve versiohistoriataulun nimi. Dockerin ja Pipelinen tiedostot puolestaan sisältävät käskyt, jotka virtuaaliympäristöä pystyttäessä tai Bitbucket Pipeline-automaattiprosessia suoritettaessa ajetaan.

Flywayn sijaan migraatioiden suorittamiseen käytettiin Evolvea, koska Flywayn ilmaisversio ei tue SQL Server 2012-tietokantoja. Työkalut ovat hyvin samankaltaisia, joten vaihdolla ei ollut suurta vaikutusta julkaisunhallintaprosessin toteutukseen. Evolve tarjoaa tapausesimerkin julkaisujen näkökulmasta melkein kaikki samat ominaisuudet ja asetukset kuin Flyway, vain eri nimillä. Yksi harmillinen puute on, että Evolve ei tarjoa automaattista tietokantayhteyden uudelleenyritystä yhteysongelmia kohdatessaan.

5.3 Testauksen toteutus

Tietokantamuutosten paikallista testausta varten toteutettiin aluksi julkaisunhallintaprosessille tyypillinen Docker-kontin sisäisen virtuaaliympäristön luonti ja siellä migraatioskriptien ajaminen tyhjään tietokantaan. Dockerfile-tiedostossa luodaan virtuaaliym-

päristö käyttäen pohjana Microsoftin virallista SQL Server 2017 konttia. Tähän ympäristöön asennetaan Evolve sekä kopioidaan migraatioskriptit ja Evolven asetustiedosto. Tietokantamuutoksia testatessa kontti pystytetään yhdellä Docker-komentorivikäskyllä. Migraatiot ajetaan yksinkertaisella komentoriviskriptillä, joka suorittaa Docker-kontin sisällä Evolvella migraatioskriptit.

Tapausesimerkkitietokannan vaatima Linked Server-tietokantayhteyden luonti esti yksinkertaisen yhden kontin Docker-virtuaaliympäristön käytön. Tuotanto- ja tuotantokopioympäristöjä varten versionhallintaan lisättiin yksinkertaisesti ympäristökohtaiset skriptit, jotka muodostavat yhteyden sopivaan valmiiksi olemassa olevaan tietokantaan. Docker-kontin sisäiselle virtuaaliympäristölle ei kuitenkaan voitu määrittää yhteyttä mihinkään tiettyyn olemassa olevaan tietokantaan. Virtuaalisen kehitys- ja testiympäristön kuuluu olla itsenäinen kokonaisuus, joka ei ole riippuvainen yhteyksistä ulkopuolisiin tietokantoihin. Virtuaaliympäristön tulee olla käytettävissä eri konteksteissa ja verkoissa, ja sitä täytyy voida hyödyntää eri tarkoituksiin. Linked Server-yhteyttä ei voitu myöskään jättää pois virtuaaliympäristön skripteistä, sillä tietokannassa on säilöttyjä proseduureja, jotka ovat riippuvaisia toisesta tietokannasta. Näitä proseduureja ei voi luoda, mikäli tietokantayhteys ei toimi tai yhteyden päässä olevasta tietokannasta ei löydy proseduurien vaatimia tauluja.

Ratkaisuna otettiin käyttöön Docker Compose, jolla voidaan luoda ja hallita samanaikaisesti useita Docker-kontteja. Versionhallintaan lisättiin itse esimerkkitietokannan lisäksi myös migraatioskriptit ja asetustiedostot, joilla saadaan automaattisesti luotua toisen Docker-kontin sisään merkittävästi karsittu versio esimerkkitietokannan tarvitsemasta toisesta tietokannasta. Tämä karsittu versio sisältää vain taulut, joihin esimerkkitietokannan proseduureissa viitataan suoraan. Docker Compose käynnistysprosessi määritettiin toimimaan siten, että se luo kaksi konttia tietokantoja varten. Tarvittu toinen tietokanta luodaan ja alustetaan ensin yhdessä kontissa, minkä jälkeen itse esimerkkitietokanta luodaan ja siihen ajetaan migraatiot toisessa kontissa. Lopputuloksena yhdellä komentorivikäskyllä Docker Compose luo tarvittavat kaksi tietokantaa konttien sisällä ja ajaa migraatioskriptit niihin, jolloin kehittäjä saa käyttöönsä tietokannan, joka vastaa versionhallintaan tallennettua tapausesimerkkitietokannan tilaa. Koska Evolve ei tue automaattista tietokantaan yhdistämisen uudelleenyritystä, joudutaan Docker Compose käynnistysprosessissa käyttämään vakiopituista odotusaikaa ennen esimerkkitietokannan migraatioiden ajoa. Odotuksella varmistetaan, että tarvittava toinen kanta on alustettu omassa kontissaan ennen esimerkkitietokannan migraatioiden suoritusta.

Tietokannan luonti kontissa testaa migraatioskriptien toimivuuden ja antaa samalla tietokantakehittäjälle käyttöönsä tietokannan, jota vasten testata tietokantamuutosten vaikutuksia. Kehittäjä voi testata tietokannan ja siitä riippuvaisten sovellusten toimivuutta

muutosten jälkeen ajamalla sovellusten integraatiotestit ja mahdollisesti myöhemmin lisättävät tietokannan yksikkötestit kontin sisäistä tietokantaa vasten. Kehittäjä voi myös manuaalisesti tarkistaa muutoksen vaikutuksia käyttämällä näitä sovelluksia, sekä yhdistää kontin sisäiseen tietokantaan ja tutkia tietokannan uutta rakennetta sekä ajaa kantaa vasten SQL-komentoja.

Bitbucket Pipelinessa testataan migraatioskriptien toimivuus aina, kun versionhallintaan tallennetaan muutoksia. Aiemmin kuvatulla tavalla Pipelinessa luodaan kaksi Docker-konttia ja niihin kaksi tietokantaa. Tietokantaa käyttävien sovellusten integraatiotestejä ei kuitenkaan ajeta, sillä tietokanta on säilötty versionhallinnassa omaan repositorioonsa ja sitä käyttäviä sovelluksia on useita kymmeniä. Kuten luvussa 4.4 kuvataan, tilanne vaatisi erittäin monimutkaisen Bitbucket Pipeline-toteutuksen, joka kärsisi kuitenkin hyvin helposti ongelmista yhteensopivien sovellus- ja tietokantaversioiden tunnistamisessa. Tietokantamuutosten testaus Pipelinessa joudutaan siis jättämään vajavaiseksi.

Yksikkötestejä tapausesimerkkikannan sisäisen toiminnan testaamiseksi ei toteutettu. Lähtötilanteessa tietokannalla ei ollut lainkaan omia testejä. Säilöttyjen proseduurien määrä tietokannassa on erittäin suuri, joten kanta olisi vaatinut hyvin suuren määrän testejä, jotta ne testattaisiin kattavasti. Tämä ei ollut tutkimuksen resurssien puitteissa mahdollista, joten tietokannan yksikkötestit jätettiin toistaiseksi pois prosessista. Toteutettu Docker-virtuaaliympäristö ja Bitbucket Pipeline-prosessi antavat kuitenkin selkeän pohjan, jolle automaattisia yksikkötestejä voidaan myöhemmin lisätä.

Lopullista testausta varten tietokantamuutokset julkaistaan tuotantokopiokantaan. Näin migraatioiden toimivuus testataan tuotantoa läheisesti vastaavassa ympäristössä. Tuotantokopiokantaa vasten voidaan myös ajaa automaattitestejä ja tehdä manuaalisia testejä, jotta migraatioiden toiminnasta varmistutaan ennen niiden julkaisua tuotantoon. Käytössä on vähemmän julkaisuympäristöjä kuin kehitetyssä julkaisunhallintaprosessissa suositellaan. Syynä tähän ovat alla luvussa 5.4 tarkemmin kuvatut palomuuriongelma ja tapa käyttää testitietokantaa versionhallinnan kanssa huonosti yhteensopivalla tavalla.

Tuotantotietokannan vanha SQL-yhteensopivuustaso, joka on käytettävissä ainoastaan SQL Server 2005-2012 tietokannoilla, estää aivan identtisten tuotannon kopioiden luonnin. Tuotantokopiotietokanta perustettiin resurssirajoitteiden vuoksi SQL Server 2014 palvelimelle. Tuotantokopiokannassa joudutaan siis käyttämään eri yhteensopivuustasoa kuin tuotantokannassa. Docker-konttien pohjaksi puolestaan Microsoft tarjoaa ainoastaan SQL Server 2017, eikä luotettavia kolmannen osapuolen SQL Server 2012 kontteja ole saatavilla. Tämä heikentää hieman esimerkkitietokannan muutoksille julkaisunhallintaprosessissa tehtävän testauksen luotettavuutta. Sekä Docker-konteissa että tuo-

tantokopioympäristössä tietokannat käyttävät yhteensopivuustasoa 100, kun taas tuotantokannassa on edellinen yhteensopivuustaso 90. Käytännössä ero tietokantojen toiminnassa näillä eri yhteensopivuustasoilla on pieni. On kuitenkin mahdollista, että tulevaisuudessa syntyy tilanne, jossa tietokantamigraatio toimii eroavien yhteensopivuustasojen vuoksi muissa ympäristöissä oikein, mutta aiheuttaakin tuotantokannassa odottamattomia vaikutuksia.

Suorituskykytestauksen tietosisällön tuottamiseen ei tarvittu mitään työkalua. Tuotantotietokannan sisältö voitiin kopioida sellaisenaan tuotantokopiokantaan, jota voidaan jatkossa käyttää suorituskykytestaukseen. Samoin kuin integraatiotestaus, myös suorituskykytestit täytyy ajaa manuaalisesti. Puutteellinen automaatio heikentää suorituskykytestauksen arvoa ja lisää riskiä, että testit jätetään tekemättä ennen päivitysten julkaisua. Uusi tuotantokopiokanta tarjoaa kuitenkin aiempaa paremmat mahdollisuudet havaita tietokantamuutosten aiheuttamia suorituskykyongelmia ennen niiden julkaisua tuotantoon.

5.4 Julkaisujen toteutus

Automaattiset julkaisut saatiin toteutettua tapausesimerkin kohdetietokannalle vain osittain. Tapausesimerkkikannalla on käytössä kolme julkaisuympäristöä eli testi-, tuotantokopio- ja tuotantotietokanta. Näistä testi- ja tuotantokanta ovat asiakasyrityksen palvelimella, jonka palomuuri estää yhteyden muodostuksen Octopus-julkaisupalvelusta. Asiakasyritys ei tutkimuksen aikana ollut halukas tekemään muutoksia palomuuriin, tai ylipäätään sallimaan tietokantamuutosten julkaisua automaattisella prosessilla. Siispä automaattiset julkaisut voitiin toteuttaa vain tuotantokopiokantaan, joka sijaitsee eri palvelimella.

Tuotantotietokantaan otettiin käyttöön puoliautomaattiset julkaisut. Palomuuriongelman takia julkaisuja tuotantoon ei voida tehdä suoraan Octopus-palvelusta. Tästä huolimatta tapausesimerkkitietokannalle haluttiin mahdollisuuksien mukaan saada julkaisunhallintaprosessin hyödyt. Näin päädyttiin ratkaisuun, jossa muutokset julkaistaan tuotantokantaan prosessin lopuksi käyttämällä Evolve-työkalua Octopuksen sijaan komentoriviltä. Suoritettava Evolve migraatiokomento on sama kuin Octopuksessa käytettävä, ennen tuotantoon julkaisua käydään läpi koko aiemmin tässä luvussa kuvattu julkaisunhallintaprosessi, ja tuotantokantaan ajetaan täsmälleen versionhallinnasta löytyvät migraatioskriptit. Tuotantojulkaisua ei kuitenkaan tehdä Octopuksen kautta, vaan tietokantakehittäjän omalta työpisteeltä komentorivikäskyllä. Mikäli palomuuriasetukset tulevaisuudessa muuttuvat ja asiakasyritykseltä saadaan lupa, voidaan Octopus-julkaisut tuotantokantaan toteuttaa helposti käyttäen pohjana jo toteutettuja julkaisuja tuotantokopiokantaan.

Mikäli julkaisunhallintaprosessia noudatetaan oikein, julkaisujen teosta komentoriviltä ei seuraa merkittäviä haittoja. Viemällä kaikki tietokantamuutokset prosessin aiempien vaiheiden läpi ja julkaisemalla kaikki muutokset tuotantokantaan Evolve-työkalulla saadaan tuotantokannalle yhä versiohistorian seuranta ja muut prosessin hyödyt. Tämä normaalista poikkeava julkaisutapa kuitenkin lisää inhimillisen erehdyksen riskiä. Ennen muutosten julkaisua tuotantoon kehittäjän on huolehdittava, että kaikki muutokset on julkaistu tuotantokopiokantaan ja testattu huolellisesti. Normaalissa prosessissa julkaisu tuotantoon ei ole mahdollista, jos migraatioskriptien Bitbucket Pipeline-prosessi tai Octopuksesta tuotantokopion päivitys epäonnistuu. Tehtäessä tuotantokannan päivitykset komentoriviltä on taas mahdollista, että varomaton tietokantakehittäjä ajaa migraatioskriptejä tuotantoon ennen kuin julkaisunhallintaprosessin aiemmat vaiheet on käyty läpi.

Julkaisut voitaisiin tehdä komentoriviltä myös testikantaan, mutta tämä torjuttiin testikantaan usein tehtävien väliaikaisten muutosten takia. Migraatiopohjaisessa tietokannan versionhallinnassa on kriittisen tärkeää, että versionhallinnan käyttöönoton jälkeen tietokantaan ei enää saa tehdä mitään rakennemuutoksia prosessin ulkopuolella. Prosessin ulkopuoliset muutokset asettavat tietokannan versionhallinnan kannalta tuntemattomaan tilaan, jolloin tulevilla migraatioilla voi olla odottamattomia ja vakavia sivuvaikutuksia. Toteutettavaan julkaisunhallintaprosessiin kuuluu myös tavoite, että kaikkiin julkaisuympäristöihin ajetaan niin samanlaiset migraatioskriptit kuin mahdollista. Näin tuotantokantaan tarkoitetut muutokset saadaan testattua mahdollisimman monessa eri ympäristössä. Tämä tarkoittaa, että tietokannan rakenteen tulisi pysyä eri julkaisuympäristöissä lähes identtisenä. Tapausesimerkkitietokannan testikantaan luodaan kuitenkin jatkuvasti väliaikaisia tauluja ja tehdään erilaisia pieniä rakennemuutoksia, joita käytetään vaihteleviin testaustarkoituksiin ja joita ei ole tarkoitus koskaan tehdä tuotantokantaan. Näiden väliaikaisten rakennemuutosten määrä on niin suuri, että esimerkkitietokannan kehittäjät kokivat versionhallinnan käyttöönoton testikannalle haitalliseksi. Heidän arvionsa mukaan näiden muutosten teko versionhallinnan ja muiden julkaisunhallintaprosessin vaiheiden kautta lisäisi muutosten työmäärää tarpeettomasti. Samalla syntyisi myös ajan mittaan epätoivotun suuri määrä testikantakohtaisia migraatioskriptejä, joissa tehdään rakennemuutoksia ja myöhemmin kumotaan niitä. Vielä ongelmallisemmaksi tilanne kävisi aina, kun testikanta korvataan kopiolla tuotannosta. Kopioinnin tuloksena testikannan rakenne ei vastaisikaan enää sitä, joka syntyy ajamalla kaikki versionhallinnan testiympäristökohtaiset migraatioskriptit. Näistä syistä testikanta jätettiin kokonaan julkaisunhallintaprosessin ulkopuolelle.

Julkaisut tuotantokopiokantaan toimivat pitkälti julkaisunhallintaprosessille suunniteltuun normaaliin tapaan. Kun migraatioskriptien toimivuus on testattu Bitbucket Pipeline-prosessissa, Pipeline paketoi migraatioskriptit ja asetustiedostot NuGet-tiedostopaketiksi ja julkaisee tämän paketin yrityksen sisäiseen NuGet-varastoon. Octopus-julkaisupalvelu hakee tiedostopaketin varastosta, purkaa sen ja ajaa paketin sisältökansiossa komentoriviltä Evolve migraatiokäskyn käyttäen kohdeympäristöä vastaavaa asetustiedostoa. Asetustiedosto sisältää valmiiksi kaikki muut tarvittavat tiedot paitsi salasanan, jolla päivitettävään tietokantaan kirjaudutaan sisään. Tietoturvasyistä tuotantokopiokannan salasana tallennetaan Octopus-palveluun muuttujana, koska palvelu tukee salattuja muuttujia, joiden arvoa ei näytetä.

Migraatioiden ajo komentoriviltä tuotantokantaan toimii samankaltaisesti. Tietoturvasyistä tuotantoympäristökohtaiseen asetustiedostoon tarvitsee lisätä tuotantokannan salasana, jota ei säilytetä versionhallinnassa kaikkien nähtävillä. Kun salasana on lisätty, ajetaan tapausesimerkkitietokannan migraatioskriptit ja asetustiedostot sisältävässä kansiossa komentoriviltä Evolve migraatiokäsky käyttäen tuotantoympäristön asetustiedostoa.

6. TULOSTEN TARKASTELU

Tässä luvussa arvioidaan, mitä hyötyjä ja haittoja kehitetty tietokannan julkaisunhallintaprosessi ja sen käytännön toteutus tutkimuksen tapausesimerkissä tarjoaa verrattuna manuaaliseen julkaisunhallintaan. Yhteenveto tuloksista on esitetty alla, taulukossa 1.

	Prosessi ihannetapauksessa	**Tapausesimerkki**
Julkaisujen luotettavuus ja häiriöiden välttäminen	Merkittävä parannus	Kohtalainen parannus
Julkaisujen tehokkuus	Kohtalainen parannus	Ei muutosta
Tietokannan dokumentaatio	Merkittävä parannus	Merkittävä parannus
Häiriöistä toipuminen	Sekä hyötyä että haittaa	Sekä hyötyä että haittaa

***Taulukko 1**: Kehitetyn prosessin ja sen käytännön toteutuksen tarjoamat hyödyt.*

Osiossa 6.1 kuvataan prosessin tuomaa tietokantamuutosten luotettavuuden parantumista ja virhetilanteiden vähentymistä. Osiossa 6.2 analysoidaan, tekeekö prosessi tietokantamuutoksista nopeampia ja tehokkaampia. Osiossa 6.3 pohditaan prosessin yleiskäyttöisyyttä kohdeyrityksessä.

6.1 Luotettavuuden parantuminen

Kehitetty tietokantojen julkaisunhallintaprosessi lisää tietokantamuutosten luotettavuutta useilla tavoilla. Näihin parannuksiin kuuluvat näkyvän tiedon määrän kasvu, tulevien tietokantamuutosten lisätty katselmointi, lisätty muutosten testaus, täsmälleen samojen muutosten julkaisu jokaiseen ympäristöön ja inhimillisten virheiden riskin pienentäminen. Tapausesimerkissä näistä saatiin toteutettua näkyvän tiedon lisäys, lisätty katselmointi, osittainen muutosten testaus sekä pienennetty inhimillisten virheiden riski. Virhetilanteista toipumiseen kehitetty prosessi tarjoaa sekä etuja että haittoja.

Julkaisunhallintaprosessi lisää näkyvää tietoa tietokannasta kolmella tavalla. Ensinnäkin tietokantamuutosten määrittely versionhallinnassa säilöttyä tietokantakaaviota muokkaamalla aiheuttaa sen, että kehittäjien saatavilla on kullekin tietokantaversiolle ajantasainen kuva tietokannan rakenteesta. Toiseksi kaikkien tietokannan migraatioskriptien

säilöminen versionhallintaan tekee tietokannan eri versioiden välisistä muutoksista näkyviä. Versionhallinnassa säilötään jokaisen migraation luontiaika, tekijä ja tekijän kommentit. Kolmanneksi migraatioiden suorittamiseen käytetyt migraatiotyökalut luovat tietokantojen sisälle versiohistoriataulut, joista nähdään missä versiossa mikäkin tietokanta on ja miten niitä on aiemmin päivitetty. Näin tiedetään selkeästi ennen muutosten julkaisua missä versiossa kohdetietokanta entuudestaan on, mitä muutoksia siihen ollaan tekemässä ja mihin tilaan se päätyy. Tapausesimerkissä kaikki kolme tapaa otettiin onnistuneesti käyttöön.

Lisätty dokumentaatio ja versionhallintaan avoimesti säilöttävät migraatioskriptit parantavat ymmärrystä tietokantojen tilasta ja tarjoavat mahdollisuuksia katselmoida ja suunnitella tulevia muutoksia entistä varmemmin. Kun tietokantaan seuraavassa julkaisussa tehtävät muutokset ovat helposti nähtävissä sekä tietokannan kaaviokuvan muutoksina että migraatioskripteinä, eri kehittäjien on helppo tutustua toistensa suunnittelemiin muutoksiin ja arvioida niitä. Versionhallinnan haarojen käyttötapa, jossa muutokset tehdään ominaisuuskohtaisissa haaroissa ja julkaisut tehdään eri haarasta, myös pakottaa kehittäjiä tekemään katselmointeja yhdistettäessä muutoksia haarasta toiseen. Migraatioskriptien säilöminen versionhallintaan sekä suunniteltu haarojen käyttötapa ja siihen kuuluva muutosten katselmointi otettiin tapausesimerkissä käyttöön hyvin tuloksin.

Julkaisunhallintaprosessi sisältää useita varmuutta lisääviä testauskierroksia ennen migraatioiden julkaisua tuotantoon. Prosessissa testataan migraatioiden oikea toimivuus monipuolisesti automaattisella ja manuaalisella testauksella useassa ympäristössä, kuten luvussa 4.3.2 kuvataan. Tämä vähentää merkittävästi virhetilanteiden syntymisen riskiä, koska jokainen testauskierros tarjoaa uusia mahdollisuuksia havaita ongelmia ennen tietokantamuutosten julkaisua tuotantoon. Testaus toteutettiin tapausesimerkissä hyvin puutteellisesti, mutta sillä saavutettiin kuitenkin rajallisia hyötyjä verrattuna vanhaan julkaisunhallintaan, jossa tietokantamuutokset testattiin ennen tuotantoon julkaisua ainoastaan tekemällä muutokset testikantaan. Uusi toteutus tarjoaa myös pohjan, jonka avulla tapausesimerkkitietokannan automaattitestausta voidaan tulevaisuudessa laajentaa.

Käyttämällä automaattista julkaisuprosessia pienennetään julkaisujen teossa tapahtuvan inhimillisen virheen riskiä. Useita ympäristöjä yksi kerrallaan käsin päivitettäessä on olemassa riski, että jokin ympäristö jää epähuomiossa päivittämättä tai että johonkin niistä tehdään eri muutokset kuin muihin. Versionhallintaan säilöttäviin migraatioskripteihin sidottu julkaisuprosessi välttää nämä ongelmat. Migraatiot ajamalla kaikkiin ympäristöihin julkaistaan varmasti samat muutokset, ja käytetyn julkaisupalvelun kautta nähdään mitkä migraatiot on ajettu mihinkin ympäristöön. Tapausesimerkissä julkaisujen aikaisten

inhimillisten virheiden riski poistettiin vain osittain, koska muutoksia ei julkaista kaikkiin ympäristöihin automaattisesti julkaisupalvelun kautta.

Virhetilanteista palautumiseen julkaisunhallintaprosessi tarjoaa sekä hyötyä että haittaa. Kehitetyn prosessin etuna on, että tietokantaan tehtyjen muutosten historia on selkeästi näkyvissä. Näin virhetilanteen aiheuttanut tietokantamuutos voidaan tunnistaa entistä nopeammin. Haittapuoli on, että virheen korjaavaa muutosta ei saa tehdä suoraan tietokantaan, vaan siitä täytyy luoda migraatioskripti, joka tallennetaan versionhallintaan ja viedään julkaisunhallintaprosessin vaiheiden läpi. Tämä voi hidastaa korjauksen julkaisua, varsinkin jos kyseessä on vain yksi tuotantokanta, johon muutosten teko käsin olisi erittäin nopeaa. Toisaalta tämä hallitumpi tapa julkaista korjauspäivitys pienentää riskiä, että korjaus itsessään sisältää uuden virheen. Julkaisunhallintaprosessi voi myös kasvattaa virhetilanteiden riskiä, mikäli sen käyttöönoton jälkeen tuotantotietokantaan tehdään muutoksia prosessin ulkopuolella eikä sen jälkeen huolehdita, että versionhallinnassa määritetty tila ja tuotantokannan todellinen tila saatetaan vastaamaan toisiaan.

6.2 Tehokkuuden parantuminen

Kehitetty julkaisunhallintaprosessi saattaa nopeuttaa tietokantamuutosten tekoa. Pelkkä tietokantamigraation luominen käsin tai sopivalla ohjelmalla ja sen ajaminen tuotantoon on erittäin nopeaa, jos kaikki menee kuten pitää. Prosessi, johon kuuluu dokumentaation päivitys, muutosten katselmointi ja runsaasti testausta on lähtökohtaisesti selvästi hitaampi. Lisätty dokumentaatio kuitenkin auttaa uusia kehittäjiä ymmärtämään tietokantaa nopeammin, ja katselmointi ja testaus vähentävät aikaa vievien virhetilanteiden syntymistä. Itse julkaisujen teko pitkälti automaattisesti julkaisupalvelun kautta on suunnilleen yhtä nopeaa kuin migraatioiden ajaminen käsin. Julkaisupalvelu on selvästi tehokkaampi vaihtoehto vain, jos kerralla ajetaan paljon migraatioita ja useaan eri ympäristöön.

Luontevampi vertailukohta kehitetylle prosessille onkin tilanne, jossa dokumentointi, katselmointi ja testaus tehdään manuaalisesti. Tällöin on selvää, että kehitetyn julkaisunhallintaprosessin tuoma automaattitestaus sekä kaikkien migraatioiden julkaisut muutamalla napinpainalluksella vähentävät tietokantapäivitysten työmäärää ja parantavat tehokkuutta. Samoin prosessin tavat yhdistää dokumentaatiota ja migraatioskriptien luontia, lisätä saatavilla olevan tiedon määrää ja tuoda katselmoinnit helpoksi versionhallinnan osaksi helpottavat kehittäjien työtä ja säästävät työaikaa. Nämä ominaisuudet auttavat kehittäjiä ymmärtämään tietokannan toimintaa ja varmistumaan migraatioiden luotettavuudesta nopeammin kuin jos samat asiat tehtäisiin joka kerta käsin yksi vaihe kerrallaan.

Tapausesimerkin kohdetietokannalla aiemmin käytössä ollut prosessi oli hyvin yksinkertainen ja nopea, mutta riippuvainen tietokannan kehittäjien vahvasta osaamisesta ja huolellisuudesta. Uuden julkaisunhallintaprosessin käyttöönotto lisäsi työvaiheita tietokantapäivitysten tekoon, mikä kasvatti tyypillisen päivityksen vaatimaa työmäärää. Työmäärää lisäsi myös julkaisunhallintaprosessin osittainen toteutus, sillä kaikkia prosessiin normaalisti kuuluvia automatisointeja ei saatu käyttöön. Kuitenkin tapausesimerkkikannan dokumentaatio parani, ja julkaisuja ja testausta saatiin automatisoitua jonkin verran. Lopputuloksena itse julkaisujen teko muuttui raskaammaksi, mutta samalla erehdysten riski pieneni ja tietokantaa koskevan tiedon saatavuus parani. Lopputuloksena tehokkuus pysyi kokonaisuutena suunnilleen samana.

6.3 Prosessin yleiskäyttöisyys

Kehitetyn julkaisunhallintaprosessin perusmuoto sekä huomioidut erityistilanteet ja niihin kehitetyt prosessin variaatiot kattavat valtaosan kohdeyrityksen tietokantaprojekteista. Tapausesimerkki osoitti, että projektien lähtökohtien poiketessa ideaaleista voidaan prosessista joutua karsimaan osia pois. Kuitenkin tapausesimerkin kohdekanta oli kohdeyrityksen mittapuulla haastava tapaus, ja siihen saatiin toteutettua suuri osa prosessista. Puutteellisellakin toteutuksella prosessi paransi tietokantamuutosten luotettavuutta ja tietokannan dokumentaatiota. Tästä voidaan päätellä, että prosessin käyttöönotolla saataisiin hyötyjä useisiin kohdeyrityksen tietokantaprojekteihin.

Julkaisunhallintaprosessi sopii hyvin uudemmille tietokannoille, joilla on useita julkaisuympäristöjä ja joita käyttää pääasiassa yksi sovellus. Tietokannan riippumattomuus muista kannoista sekä valmiiksi saatavilla olevat kantaa käyttävän sovelluksen kattavat integraatiotestit tekevät tietokannasta erityisen hyvän kohteen prosessille. Tällaisia tietokantoja on kohdeyrityksessä monia. Niihin voitaisiin mitä luultavimmin toteuttaa prosessi kokonaisuudessaan, jolloin saataisiin sen täydet hyödyt.

7. YHTEENVETO

Tässä työssä kehitettiin automaattinen julkaisunhallintaprosessi, jolla tietokantapäivitysten luotettavuutta ja tehokkuutta voidaan parantaa. Tapausesimerkkinä prosessi otettiin käyttöön vanhalle tietokannalle, jonka julkaisuja oli aiemmin hallittu manuaalisesti. Kehitetyn prosessin päätavoite oli tehdä tietokantapäivityksistä mahdollisimman luotettavia ja ennakoitavia sekä vähentää tietokannan häiriöitä. Lisäksi pyrittiin parantamaan päivitysten tehokkuutta sekä lisäämään dokumentaatiota tietokannasta ja sen muutoshistoriasta. Prosessista toivottiin mahdollisimman monelle tutkimuksen kohdeyrityksen tietokannalle sopivaa. Selvitettävät tutkimuskysymykset olivat:

- Miten tietokannalle toteutetaan automaattinen julkaisunhallintaprosessi?
- Mitä hyötyjä saavutetaan tietokannan automaattisella julkaisunhallinnalla, verrattuna manuaaliseen julkaisunhallintaan?

Tutkimus aloitettiin selvittämällä tietokantojen ja niiden julkaisunhallinnan teoriaa. Tämän jälkeen tutustuttiin joukkoon ohjelmistotyökaluja, joiden avulla julkaisunhallinnan eri osia voidaan toteuttaa. Seuraavaksi kerättiin tietoa tapausesimerkkinä käytettävästä tietokannasta sekä kohdeyrityksen tietokannoista yleisesti. Kootun tiedon perusteella kehitettiin automaattinen julkaisunhallintaprosessi, jonka arvioidaan saavuttavan sille määritetyt tavoitteet. Tätä prosessia testattiin käytännössä ottamalla se käyttöön yhdelle tietokannalle. Kyseisen tietokannan erityispiirteet estivät osittain prosessin toteutuksen, ja toisia osia toteutettiin muunnellusti. Osittainen prosessi saatiin kuitenkin käyttöön toimivana.

Vastauksena ensimmäiseen tutkimuskysymykseen kehitettiin julkaisunhallintaprosessi, joka arvioitiin käyttökelpoiseksi suurimmalle osalle kohdeyrityksen tietokannoista. Prosessin yleisrakenne muistuttaa yrityksessä jo käytössä ollutta sovellusten julkaisunhallintaprosessia. Tietokannoille määriteltiin tapa säilöä muutoshistoria versionhallintaan migraatioskriptien ja asetustiedostojen muodossa. Sovellusten koontia ja testausta vastaavana vaiheena tietokannoille suunniteltiin Bitbucket Pipelinessa automaattinen tietokantamigraatioiden ja tietokannan testien suoritus. Tietokantapäivitysten julkaisut eri ympäristöihin tehdään Octopus Deploy-julkaisupalvelun kautta lähes automaattisesti muutamalla napinpainalluksella. Jos tietokantaa käyttää vain yksi sovellus, kehitetty prosessi mahdollistaa tietokannan liittämisen osaksi sovelluksen julkaisunhallintaprosessia. Sovellus ja tietokanta voidaan käsitellä yhteisenä pakettina, jossa molemmat säilötään samaan versionhallintaan, testataan yhdessä ja julkaistaan samalla kertaa.

Toisen tutkimuskysymyksen vastauksena todettiin, että automaattinen julkaisunhallinta parantaa erityisesti tietokantapäivitysten ennakoitavuutta ja vähentää virheiden riskiä.

Tietokannan dokumentaatioon ja muutoshistorian seurantaan saatiin myös huomattavia parannuksia. Päivitysten julkaisun tehokkuus ei suoraan parantunut merkittävästi. Kuitenkin lisääntynyt päivitysten luotettavuus vähentää aikaa, jota tarvitaan tietokantapäivitysten manuaaliseen testaamiseen ja häiriötilanteiden käsittelyyn. Parantunut dokumentaatio myös auttaa uusia tietokantakehittäjiä sisäistämään tietokannan rakenteen nopeammin, ja auttaa virheiden paikallistamisessa, mikäli häiriötilanne syntyy.

8. LÄHDELUETTELO

[1] S. Ambler, Agile Database Techniques: Effective Strategies for the Agile Software Developer, 1st toim., Wiley Publishing, 2003.

[2] Red Gate Software Ltd, ”State of Database DevOps,” 2019.

[3] K. Lukka, ”Konstruktiivinen tutkimusote,” METODIX, 19 Toukokuu 2014. [Online]. Available: https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/. [Haettu 5 Helmikuu 2020].

[4] Encyclopaedia Britannica, ”Database,” 15 Maaliskuu 2017. [Online]. Available: https://www.britannica.com/technology/database. [Haettu 16 Helmikuu 2020].

[5] H. Laine, ”Tietokantojen perusteet, osa 1,” Helsinki, Helsingin yliopisto, tietojenkäsittelytieteen laitos, 1999, pp. 1-3.

[6] M. Raza, ”What is a DBMS? Database Management Systems Explained,” 29 Elokuu 2018. [Online]. Available: https://www.bmc.com/blogs/dbms-database-management-systems.

[7] ScaleGrid, ”2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use,” 4 Maaliskuu 2019. [Online]. Available: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/.

[8] solid IT gmbh, ”DB-Engines Ranking,” Helmikuu 2020. [Online]. Available: https://db-engines.com/en/ranking.

[9] M. Drake, ”A Comparison of NoSQL Database Management Systems and Models,” 9 Elokuu 2019. [Online]. Available: https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models.

[10] IBM Cloud Education, ”NoSQL Databases,” 6 Elokuu 2019. [Online]. Available: https://www.ibm.com/cloud/learn/nosql-databases.

[11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S.

Mellor, K. Schwaber, J. Sutherland ja D. Thomas, ”The Agile Manifesto,” 2001. [Online]. Available: http://agilemanifesto.org/.

[12] Agile Alliance, ”Incremental Development,” [Online]. Available: https://www.agilealliance.org/glossary/incremental-development. [Haettu 16 Helmikuu 2020].

[13] Atlassian, ”Versionhallintaohjelmisto ammattilaistiimeille,” [Online]. Available: https://bitbucket.org/product/fi/version-control-software. [Haettu 16 Helmikuu 2020].

[14] M. Skelton, ”Database Version Control,” 24 Maaliskuu 2016. [Online]. Available: https://www.red-gate.com/simple-talk/sql/database-delivery/database-version-control/.

[15] MongoDB, Inc, ”Data Modeling Introduction,” 13 Elokuu 2019. [Online]. Available: https://docs.mongodb.com/manual/core/data-modeling-introduction/.

[16] A. Bhowmick, ”Continuous Integration and Continuous Delivery for Database Changes,” 15 Elokuu 2018. [Online]. Available: https://dzone.com/articles/continuous-integration-and-continuous-delivery-for-2.

[17] E. Piairo, ”Why and How Database Changes Should Be Included in the Deployment Pipeline,” 30 Tammikuu 2021. [Online]. Available: https://www.infoq.com/articles/deployment-pipeline-database-changes/. [Haettu 1 Helmikuu 2021].

[18] M. Skelton, ”Database Migrations: Modifying Existing Databases,” 9 Maaliskuu 2016. [Online]. Available: https://www.red-gate.com/simple-talk/sql/database-delivery/database-migrations-modifying-existing-databases/. [Haettu 1 Helmikuu 2021].

[19] A. Yates, ”Critiquing two different approaches to delivering databases: Migrations vs state,” 18 Kesäkuu 2015. [Online]. Available: http://workingwithdevs.com/delivering-databases-migrations-vs-state/. [Haettu 1 Helmikuu 2021].

[20] V. Khorikov, ”State vs migration-driven database delivery,” 18 Elokuu 2015. [Online]. Available: https://enterprisecraftsmanship.com/posts/state-vs-migration-driven-database-delivery/. [Haettu 1 Helmikuu 2021].

[21] CodeShip, ”Continuous Integration Essentials,” [Online]. Available: https://codeship.com/continuous-integration-essentials. [Haettu 16 Helmikuu 2020].

[22] V. Pecanac, ”Top 8 Continuous Integration Tools,” 20 Helmikuu 2016. [Online]. Available: https://code-maze.com/top-8-continuous-integration-tools/.

[23] G. Fitchey, ”Database Continuous Integration,” Red Gate Software Ltd., 10 Helmikuu 2016. [Online]. Available: https://www.red-gate.com/simple-talk/sql/database-delivery/database-continuous-integration/. [Haettu 15 Helmikuu 2020].

[24] G. Fitchey, ”Database Lifecycle Management: Deployment and Release,” Red Gate Software Ltd., 20 Kesäkuu 2016. [Online]. Available: https://www.red-gate.com/simple-talk/sql/database-delivery/database-lifecycle-management-deployment-and-release/. [Haettu 15 Helmikuu 2020].

[25] E. Elliott, ”A DLM Approach to Database Testing,” Red Gate Software Ltd., 13 Huhtikuu 2017. [Online]. Available: https://www.red-gate.com/simple-talk/sql/database-delivery/dlm-approach-database-testing/. [Haettu 22 Helmikuu 2020].

[26] S. Pittet, ”Continuous integration vs. continuous delivery vs. continuous deployment,” [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

[27] T. Austin, ”Moving from application automation to true DevOps by including the database,” Red Gate Software Ltd., 20 Kesäkuu 2018. [Online]. Available: https://www.red-gate.com/hub/product-learning/sql-change-automation/moving-from-application-automation-to-true-devops-by-including-the-database. [Haettu 25 Helmikuu 2020].

[28] Git community, ”Git,” [Online]. Available: https://git-scm.com/. [Haettu 26 Helmikuu 2020].

[29] Stack Overflow, ” Developer Survey Results 2018,” Stack Overflow, Tammikuu 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018/. [Haettu 26 Helmikuu 2020].

[30] Apache Software Foundation, ”Apache® Subversion®,” Apache Software Foundation, 2018. [Online]. Available: https://subversion.apache.org/. [Haettu 26 Helmikuu 2020].

[31] A. Babenhauserheide, ”Mercurial SCM,” Mercurial community, 26 Heinäkuu 2019. [Online]. Available: https://www.mercurial-scm.org/. [Haettu 26 Helmikuu 2020].

[32] GitHub, Inc., ”GitHub,” GitHub, Inc., 2020. [Online]. Available: https://github.com/. [Haettu 4 Maaliskuu 2020].

[33] GitLab, ”About GitLab,” GitLab, 2020. [Online]. Available: https://about.gitlab.com/. [Haettu 4 Maaliskuu 2020].

[34] DBeaver Corp, ”DBeaver,” DBeaver Corp, [Online]. Available: https://dbeaver.com/. [Haettu 23 Huhtikuu 2020].

[35] Devart, ”Database Development and Management Software,” Devart, 2020. [Online]. Available: https://www.devart.com/dbforge/. [Haettu 5 Maaliskuu 2020].

[36] Microsoft, ”Overview of Entity Framework Core,” Microsoft, 27 Lokakuu 2016. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Haettu 20 Maaliskuu 2020].

[37] PremiumSoft™ CyberTech Ltd., ”Navicat Data Modeler,” PremiumSoft™ CyberTech Ltd., 2021. [Online]. Available: https://www.navicat.com/en/products/navicat-data-modeler. [Haettu 28 Tammikuu 2021].

[38] Oracle, ”Oracle SQL Developer,” Oracle, 2020. [Online]. Available: https://www.oracle.com/database/technologies/appdev/sql-developer.html. [Haettu 4 Maaliskuu 2020].

[39] Microsoft, ”SQL Server Data Tools for Visual Studio,” Microsoft, 2021. [Online]. Available: https://visualstudio.microsoft.com/vs/features/ssdt/. [Haettu 27 Tammikuu 2021].

[40] Microsoft, ”What is SQL Server Management Studio (SSMS)?,” Microsoft, 9 Marraskuu 2019. [Online]. Available: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms. [Haettu 4 Maaliskuu 2020].

[41] Quest Software Inc., ”Toad Database Developer and Administration Software Tools,” Quest Software Inc., 2020. [Online]. Available: https://www.quest.com/toad/. [Haettu 12 Huhtikuu 2020].

[42] J. Benilov, ”DbFit: Test-driven database development,” 2015. [Online]. Available: http://dbfit.github.io/dbfit/. [Haettu 27 Tammikuu 2021].

[43] DbUnit Team, ”About DbUnit,” DbUnit Team, 2020. [Online]. Available: http://dbunit.sourceforge.net/. [Haettu 27 Tammikuu 2021].

[44] S. Meine ja D. J. Lloyd, ”tSQLt - Database Unit Testing for SQL Server,” sqlity.net llc, 2010. [Online]. Available: https://tsqlt.org/. [Haettu 27 Tammikuu 2021].

[45] DTM soft., ”DTM Database Tools,” DTM soft., 2021. [Online]. Available: http://www.sqledit.com/index.html. [Haettu 27 Tammikuu 2021].

[46] EMS Software Development, LLC, ”Data Generator Products Family,” EMS Software Development, LLC, 2021. [Online]. Available: https://www.sqlmanager.net/en/products/datagenerator. [Haettu 27 Tammikuu 2021].

[47] Wisser, ”Jailer,” Sourceforge, 2020. [Online]. Available: https://sourceforge.net/projects/jailer/. [Haettu 27 Tammikuu 2021].

[48] Datical, ”Agile Database Release Automation,” Datical, 2020. [Online]. Available: https://www.datical.com/. [Haettu 20 Maaliskuu 2020].

[49] J. B. R. K. Graham Tackley, ”Welcome to the dbdeploy project,” dbdeploy, 2020. [Online]. Available: http://dbdeploy.com/. [Haettu 20 Maaliskuu 2020].

[50] J. Ginnivan, ”DbUp: Easy SQL Server change script runner,” 3 Lokakuu 2019. [Online]. Available: https://dbup.readthedocs.io/en/latest/. [Haettu 20 Maaliskuu 2020].

[51] P. Lécaillon, ”What is Evolve ?,” 2 Maaliskuu 2020. [Online]. Available: https://evolve-db.netlify.com/. [Haettu 20 Maaliskuu 2020].

[52] Boxfuse GmbH, ”Flyway by Redgate,” Boxfuse GmbH, 2020. [Online]. Available: https://flywaydb.org/. [Haettu 20 Maaliskuu 2020].

[53] T. Griesser, ”Knex.js,” 2020. [Online]. Available: http://knexjs.org/. [Haettu 20 Maaliskuu 2020].

[54] Datical, ”Liquibase,” Datical, 2020. [Online]. Available: https://www.liquibase.org/. [Haettu 20 Maaliskuu 2020].

[55] MyBatis.org, ”MyBatis Migrations,” MyBatis, 28 Elokuu 2019. [Online]. Available: https://mybatis.org/migrations/. [Haettu 20 Maaliskuu 2020].

[56] Red Gate Software Ltd, ”SQL Toolbelt,” Red Gate Software Ltd, 2020. [Online]. Available: https://www.red-gate.com/products/sql-development/sql-toolbelt/. [Haettu 20 Maaliskuu 2020].

[57] Sqitch yhteisö, ”Sqitch,” Sqitch yhteisö, 2 Lokakuu 2019. [Online]. Available: https://sqitch.org/. [Haettu 20 Maaliskuu 2020].

[58] Innovartis Ltd, ”DB Ghost Change Manager,” Innovartis Ltd, 2018. [Online]. Available: http://www.dbghost.com/products/ChangeManager.aspx. [Haettu 20 Maaliskuu 2020].

[59] DBmaestro, ”DevOps For Databases,” DBmaestro, 2020. [Online]. Available: https://www.dbmaestro.com/. [Haettu 20 Maaliskuu 2020].

[60] Microsoft, ”SQL Server Data Tools,” Microsoft, 02 Syyskuu 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/ssdt/sql-server-data-tools. [Haettu 4 Maaliskuu 2020].

[61] Github, Inc., ”DbUp,” Github, Inc., 2020. [Online]. Available: https://github.com/DbUp. [Haettu 20 Maaliskuu 2020].