Haaga-Helia
ammattikorkeakoulu Oy

SQL Serverin ja MongoDB:n vertailu ja transaktioiden hallinta

Julia Björklund

Opinnäytetyö
Tietojenkäsittelyn
koulutusohjelma
2019

Tiivistelmä

Haaga-Helia ammattikorkeakoulu Oy

Tekijä(t)
Julia Björklund

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
SQL Serverin ja MongoDB:n vertailu ja transaktioiden hallinta

Sivu- ja liitesivumäärä
50

Tämän tutkimustyön tavoitteena on tutkia relaatiotietokantojen ja NoSQL-tietokantojen eroavaisuuksia. Tutkimuksen tavoitteena on myös selvittää, miten transaktioiden hallintaa sovelletaan kyseisissä tietokantaympäristöissä. Vertailukohteina toimivat SQL Server ja MongoDB.

Työssä käytetään vetoketjumallia tarkoittaen, että tietoperusta ja tulosten käsittely kulkevat käsi kädessä työn alusta loppuun. Tutkimus koostuu tietoperustasta, tietoperustan analysoinnista ja vertailusta sekä käytännön kokeiluista.

Teoriaosuudessa keskitytään tietokantojen perusteisiin, jotta transaktioiden rooli tietokantojen toiminnallisuudessa olisi helpommin ymmärrettävissä. Tutkimuksessa lukijalle käy ilmi: Mikä relaatiotietokanta on? Mikä NoSQL-tietokanta on? Entä minkälaisia tietokantaohjelmistoja näiden hallitsemiseen on kehitelty ja minkälaisia ominaisuuksia ne pitävät sisällään.

Tutkimuksessa selvitetään myös, miten näiden kahden kantamuodon ominaisuudet tulisi huomioida sovelluksen palvelutarpeen näkökulmasta. Tietokantaohjelmistoja on monia erilaisia ja niiden ominaisuudet vaihtelevat ohjelmistokohtaisesti huomattavasti. Tutkimuksessa selvitetään, mitä käyttäjän tulisi ottaa huomioon ennen tietokantaohjelmiston käyttöönottoa.

Transaktioita käsittelevässä osiossa vertaillaan transaktioiden soveltamista relaatiotietokannoissa ja NoSQL-tietokannoissa, myöhemmin vielä tarkemmin SQL Serverin ja MongoDB:n välillä. Transaktioiden tarpeellisuutta havainnoidaan esimerkkikuvien avulla. Käytännön kokeilut SQL Serverin ja MongoDB:n välillä esitetään tässä osiossa. Käytännön kokeiluissa havainnollistetaan transaktiokomentojen toimintaa käytännössä.

Lopuksi työssä vielä vertaillaan merkittävimpiä työssä esille tulleita eroavaisuuksia SQL Serverin ja MongoDB:n transaktioidenhallinnan välillä. Työn päättää pohdinnat.

Asiasanat
Transaktio, Relaatiotietokanta, MongoDB, SQL Server, NoSQL

Sisällys

1 Johdanto

Tietokantoja on käytetty osana sovelluksia ja tietojärjestelmiä 1970-luvulta lähtien. Tietokantaohjelmistojen markkinoita ovat hallinneet 1990-luvulta asti relaatiotietokantamalliin pohjautuvat tietokantaohjelmistot. (Vidhya, Jeyaram & Ishwarya 2016, 38) Kuitenkin viimeisen vuosikymmenen aikana NoSQL-tietokannat ovat alkaneet kasvattaa suosiotaan yhä enemmän varsinkin nuorten ohjelmoijien parissa. (Ari Hovi 2015.)

Monesti tietokantojen käsittelyyn käytettävien tietokantaohjelmistojen tärkeimpiin ominaisuuksiin mielletään tiedon säilymisen eheys ja luotettavuus tietokannassa. Relaatiotietokantaohjelmistot ovat tunnettuja tiedon eheyden painottamisesta. (Tivi 2004.) Relaatiotietokantaohjelmistojen tiedon eheyden ylläpitämiseen on kehitetty monenlaisia keinoja, joista yksi tärkein keino on transaktiot. (Laiho & Wendelius 2015, 1.) Vaikka monet pitävät datan eheyttä yhtenä tietokantojen tärkeimpänä piirteenä, NoSQL-tietokannat ovat nostattaneet suosiotaan eheyden ylläpidon puutteellisuudesta huolimatta. Jos tiedon eheydestä voidaan joustaa, millaisia mahdollisuuksia se meille luo erilaisten tietokantaohjelmistojen käytölle?

Tämän tutkimustyön tavoitteena on selvittää, mitä ovat transaktiot ja miten niitä sovelletaan eri tietokantaohjelmistoissa. Jotta transaktioita ja tietokantaohjelmistojen eroja ymmärrettäisiin, työn tarkoituksena on myös tutustuttaa lukija relaatiotietokantojen ja NoSQL-kantojen toiminnallisuuslogiikkaan. Työn tutkimuskysymyksinä ovat: 1. Mitä ovat transaktiot? 2. Miten niitä sovelletaan tietokantaympäristöissä? 3. Mitä eroja on relaatiotietokantojen ja NoSQL-tietokantojen transaktioiden hallinnassa?

Tutkimuksessa vertaillaan jo pidemmän aikaa tietokantamarkkinoilla asemaansa vakiinnuttanutta relaatiotietokantaohjelmistoa SQL Serveriä ja yhtä tunnetuimmista NoSQL-tietokannoista MongoDB:tä. Tässä opinnäytetyössä esitellyt esimerkit toteutetaan avoimen lähdekoodin SQL Server Express 2012- versiolla. Mielenkiintoisin aspekti aiheeseen tuodaan MongoDB:n kautta, sillä viimeisin markkinoille tullut versio 4.0 mahdollistaa ensimmäistä kertaa laajemman transaktioiden käsittelyn MongoDB:llä. Tässä työssä vertailuun käytetään juuri tätä viimeisintä 4.0 versiota MongoDB:stä. (MongoDB 2019.)

Tutkimus on suoritettu laadullisin menetelmin, hyödyntäen akateemista kirjallisuutta ja ajankohtaisia julkaisuja aiheesta. Tutkimuksen laatija tutustuu aiheeseen opinnäytetyötä laatiessa. Tietääkseni aikaisemmin tehtyjä opinnäytetöitä aiheesta transaktiot, tai niiden vertailu SQL Serverissä ja MongoDB:ssä ei ole juurikaan tehty. Relaatiotietokantoja ja NoSQL-tietokantojen vertailua on tehty jonkin verran. Opinnäytetöissä on keskitytty mo-

nesti joko relaatiotietokantoihin tai NoSQL-tietokantoihin. MongoDB:n transaktioiden hallinnasta en myöskään onnistunut löytämään akateemista kirjallisuutta, sillä toiminnallisuus on kyseisessä ympäristössä niin uusi. Aiheesta löytyi tietoa MongoDB:n nettisivuilta ja muutamasta luotettavasta asiantuntijan tekemästä artikkelista. Tiedustelin aiheesta myös kaikilta ohjelmointia työkseen tekeviltä tuttaviltani, mutta he eivät joko olleet olleet tekemisissä MongoDB:n kanssa, tai he eivät olleet vielä tulleet käyttäneeksi uusinta transaktioita sisältävää versiota ohjelmistosta. Tämä luo osaltaan omat haasteensa tämän tutkimustyön toteutukselle.

Sovelluksen palvelutarpeen näkökulmasta tietokantaohjelmistojen ominaisuuksien ja erovaisuuksien erojen ymmärtäminen on ensisijaisen tärkeää. Tietokantoja, tietokantamalleja ja tietokantaohjelmistoja löytyy nykyään jo melkein jokaiseen tarpeeseen. Tietokantaohjelmistoa valikoitaessa käyttäjän tulisi olla tietoinen varsinkin niistä ominaisuuksista, joista hän on ohjelmistossa valmis joustamaan.

Käsitteet

DBMS (Database management system) = Tietokannan hallintajärjestelmä.

DML (Data manipulation language) = SQL- kielen alakielenä käytetty tietokannan muokkauksen ohjelmointikieli, joka sisältää lisäys-, muokkaus- ja poisto-ominaisuudet.

MongoDB = Dokumenttimalliin perustuva NoSQL-tietokantaohjelmisto.

NoSQL-tietokanta = NoSQL-tietokannat ovat tietokantoja, jotka eivät välttämättä perusta rakennettaan mihinkään tiettyyn tietokantamalliin. Omainsuuksiin kuuluu suurten struktuuroimattomien datamäärien tehokas prosessointi.

Pääavain (Primary key) = Relaatiotietokantojen tauluissa pääavaimella identifioidaan rivit omalla uniikilla avaimella, jota voidaan hyödyntää taulujen välisissä relaatioissa.

Phantom = Phantom riveiksi kutsutaan tietokantarivejä, jotka on lisätty tai joita on muokattu niin, etteivät järjestelmään tehdyt kyselyt ole ehtineet ottaa niitä vielä huomioon.

Proseduuri = Tietokantaohjelmistoille voidaan määritellä proseduureja, toimintamalleja, jotka määrittelevät tietokannan toimintaa esimerkiksi ongelmatilanteissa. Proseduurit voidaan määritellä järjestelmäkohtaisesti.

Relaatiotietokanta = Relaatiomalliin perustuva tietokanta.

Retry Wrapper = Relaatiotietokantojen uudelleenkäynnistyslogiikan ohjelmointimalli.

SQL ohjelmointikieli (Structured Query Language) = Relaatiotietokannoissa sovellettu ohjelmointikieli, jolla pystytään hallitsemaan tietokannan sisältöä.

SQL Server = Microsoftin kehittämä, relaatiomalliin perustuva relaatiotietokantaohjelmisto.

Skaalautuminen= Tietokannan suorituskyvyn tai tallennustilan kasvattamista kuvaava termi.

Transaktio = Transaktioiden avulla pyritään estämään virheellisen tiedon pääsy tietokantaan. Transaktiot koostuvat komennoista ja komentosarjoista.

Viiteavain (Foreign key) = Relaatiotietokannoissa käytetyillä viiteavaimilla voidaan linkittää tauluja keskenään eli luoda taulujen välisiä relaatioita.

Välimuisti = Paikka, johon dataa säilötään vain hetkellisesti, ja josta tietoa voidaan hakea nopeasti.

2 Relaatiotietokannat

2.1 Mikä on tietokanta?

Tietokanta (Database), on jaettu kokoelma loogisesti toisiinsa liittyvästä datasta. (Connolly & Begg 2015, 63.) Tietokannoilla on tärkeä rooli monissa arkipäivän toiminnoissa. Tietokantoja tarvitaan esimerkiksi rahan nostamiseen pankkitililtä tai junan istumapaikan varaamiseen. Tietokantojen merkitystä tietotekniikan infrastruktuurissa ei osata aina edes ajatella. (Connolly & Begg 2015, 63.)

Tietokannat kehiteltiin jo 1970-luvulla, jolloin ne olivat hierarkisia tai verkkopohjaisia. Hierarkkinen tietomalli oli ensimmäinen tietomalli, jota käytettiin tietokantojen toteutuksessa. Hierarkkinen tietomalli perustui puumalliin, jossa tietoa pystyttiin liittämään yhden tietyn aihealueen alle. Tämä teki tiedon tallettamisesta ja hakemisesta tehokasta, mutta monimutkaisempien kyselyiden tekeminen kantaan ei ollut mahdollista, sillä kantaa analysoitiin aina yksisuuntaisesti esimerkiksi ylhäältä alaspäin.

Hierarkkisen mallin ohella otettiin käyttöön verkkopohjaiset tietokantamallit, jotka perustuivat kaaviorakenteisiin. Tällaisilla kaaviorakenteilla oli mahdollista suorittaa myös monimutkaisempien kyselyiden tekeminen tietokantaan, mutta ne myös vaativat käyttäjiltä yksityiskohtaista tietämystä tietokannan rakenteesta, jotta kanta oli mahdollista toteuttaa. Sekä hierarkkisilla, että verkkopohjaisilla tietokantamalleilla, ei ollut mahdollista toteuttaa rakenteellisia riippuvaisuussuhteita, mikä teki niistä nykyajan tietokantamalleja epäkäytännöllisempiä. (Vidhya ym. 2016, 38)

Tiedon määrän lisääntyessä sen hallitsemiseen kehiteltiin tietokannan hallintajärjestelmät. Tietokannan hallintajärjestelmiä eli tietokantaohjelmistoja ovat esimerkiksi relaatiotietokantaohjelmistot ja NoSQL-tietokantaohjelmistot. (McCreary & Kelly 2013.) Tietokantaohjelmistoilla luodaan tietokantaa koskevat määritykset ja sisältö. Tietokantaohjelmistoilla pystytään myös lisäämään, päivittämään ja poistamaan tietokantoihin määriteltyä dataa. (Connolly & Begg 2015, 63.) Data voi olla esimerkiksi tekstiä, numeroita tai kuvia. (McCreary & Kelly 2013.)

2.2 Relaatiomalli

Kaikki relaatiotietokannat perustuvat IBM:n tutkija E.F. Coddin vuonna 1970 kehittämään relaatiomalliin. Relaatiomalli puolestaan perustuu joukko-oppiin, matematiikkaan ja kolmiarvoiseen predikaattilogiikkaan. SQL (Structured Query Language) on standardoitunut

kaikkien relaatiotietokantatoimittajien tietokantakieleksi. SQL:ää käytetään relaatiotietokantaohjelmissa, kuten MySQL. SQL mahdollistaa kyselyiden (query) tekemisen tietokantaan. SQL-kielen avulla pystytään esimerkiksi lisäämään, päivittämään tai poistamaan dataa. Relaatiotietokantojen datan käsittelyssä käytetään aina tietokantaan talletettuja tauluja, jotka perustetaan SQL-kielellä. Taulut sisältävät sarakkeita (eng.column) ja rivejä (eng. row). Jokaisella taululla on tunnisteena pääavain (eng.primary key), jolla identifioidaan taulu. Pääavaimella ei voi ikinä olla tyhjää arvoa eli NULL-arvoa. Pääavaimen avulla taululle muodostetaan oma yksilöllinen identifikaattori. Tauluille voidaan määritellä myös viiteavaimia (eng. foreign key), jotka ovat toisten taulujen perusavaimia. Näiden viiteavaimien avulla tauluja voidaan linkittää keskenään. Linkittämisestä käytetään termiä isä-lapsiyhteys, johon relaatiotietokantojen "relaatio" myöskin viittaa (eng. sanasta relation=suhde). Katso kuva 1. (Hovi 2003, 5.)

Pankkirelaatio

panktunnus (PK)*	panknimi	koodi
1	Nordea	121
2	Osuuspankki	213
3	Danske Bank	1313
4	Suomen Pankki	313131

Henkilörelaatio

htun	etunimi	snimi	kunta	**panktunnus(FK)***
111191-111A	Elsa	Mäkinen	Helsinki	1
110381-1123	Matti	Meikäläinen	Helsinki	2
080895-3212	Maija	Mattila	Helsinki	4
040591-1334	Anneli	Auttava	Helsinki	1

Kuva 1. Relaatiotietokantatauluja (mukaillen Hovi 2013) * PK= Pääavain (Primary key), FK= Viiteavain (Foreign key).

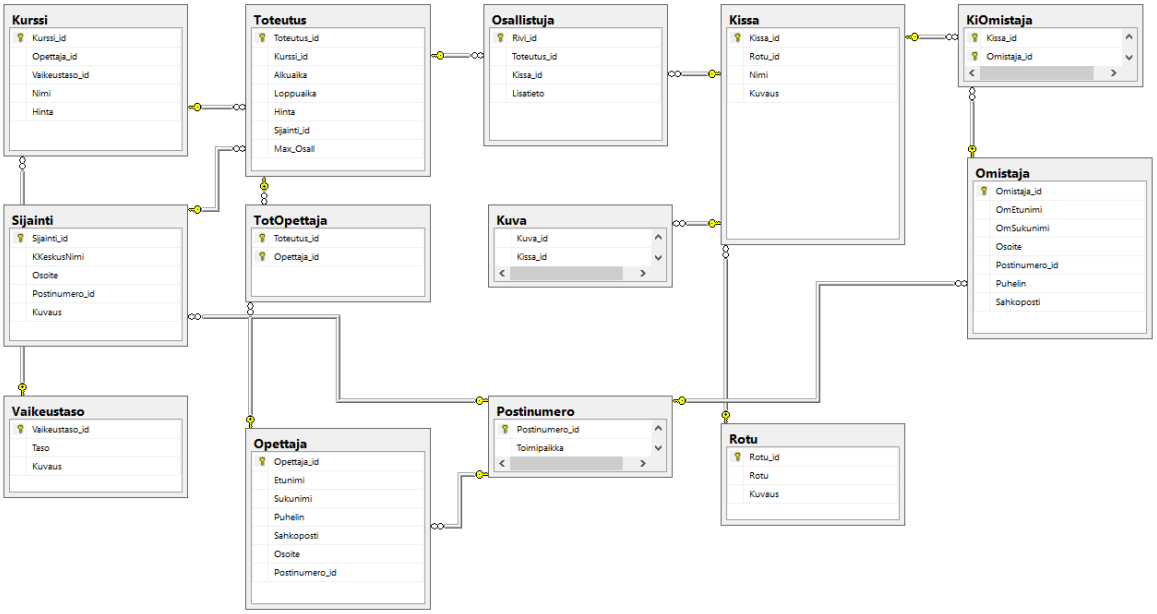
Kuva 1 sisältää kaksi tietokantataulua. Näillä kahdella taululla kuvataan isä-lapsisuhdetta. "Panktunnus" -pääavain Pankkirelaatio taulussa toimii niin sanotusti isänä, joka määrittelee lapsen suhteen isä-kaavioon. Numero "1" on id joka kertoo, että kaikki Henkilörelaatiotaulun henkilöt, joilla on rivi id "1" "panktunnus" sarakkeessa kuuluvat Nordea-pankin asiakkaisiin.

2.3 **Eheyssäännöt**

Eheyssäännöillä varmistetaan relaatiotietokannan eheys. Eheyssääntöjä ylläpidetään eheysrajoitteilla. Eheysrajoitteilla varmistetaan myös se, ettei tieto katoa, sillä kaikilla riveillä on omat yksilölliset tunnisteensa. (Hovi 2003, 10.) Eheysrajoitteilla vältetään esimerkiksi sellaiset tilanteet, joissa tietokantaan talletettaisiin vahingossa kaksi samannimistä henkilöä. Tämän jälkeen ei enää tiedettäisi, kumpi näistä oli se "oikea". (Hovi 2003, 9.) Eheysrajoitteita ovat avaineheys, arvojoukkoeheys ja viite-eheys. Avaineheyssäännöllä varmistetaan, että pääavaimen arvo ei saa olla NULL toisin sanoen pääavain on aina pakollinen. Arvojoukkoeheydellä varmistetaan tietojen oikeaoppinen sijoittaminen omiin lokeroihinsa niin, etteivät esimerkiksi nimiarvot saa puhelinnumeroarvoja. (Virkki 25.4.2019.) Viite-eheydellä varmistetaan tietojen välisten viittausten oikeaoppisuus. Viitatessa käyttäjän tulee viitata johonkin olemassa olevaan riviin, toisin sanoen "lapset eivät saa jäädä orvoiksi". (Hovi 2003, 10.)

2.4 **Relaatiomallin määrittely**

Relaatiomallin, ja sille määriteltyjen sääntöjen tehtävänä on ylläpitää tietokannan datan eheyttä. Jotta relaatiomalliin perustuvat taulukot saataisiin kehiteltyä, täytyy datan rakenne taulukoissa ensin määritellä. Toisin sanoen taulurakenteet täytyy määritellä. Taulurakenteen määrittelyllä data saadaan jaoteltua sinne, minne se kuuluu niin, että tieto säilyy muutoksista huolimatta eheänä. Tietokantamalli määrittelee, minkälaista dataa tietokantaan viedään, ja minkälaiset rajoitukset kyseisellä datalla on. Onko kyseessä esimerkiksi tekstikenttä (varchar) tai päivämääräkenttä (datetime) yms. Tietokantamallista käyttäjä voi nähdä myös, millaiset pituusrajoitukset tekstikenttien merkeille on asetettu. Tietokantamallista pystytään myös näkemään jokaisen taulun pää- ja viiteavaimet. Kuva 2 esittää esimerkin relaatiotietokantamallista SQL Serverillä. (Brade 2019.)



Kuva 2. Esimerkki tietokantamallista.

2.5 **Relaatiotietokantojen skaalautuvuus**

Relaatiotietokannat pystyvät yleisen ajatusmallin mukaisesti skaalautumaan ainoastaan ylöspäin. Skaalautumisella tarkoitetaan tässä yhteydessä tietokannan mahdollisuuksia ja valmiuksia resurssien lisäämiseen. Eli jos relaatiotietokannan suorituskykyä tai tallennustilaa halutaan lisätä, voidaan se tehdä vain ostamalla isompi ja tehokkaampi palvelin. (Eelco, Membrey & Hawkins 2010, 7.) Käyttäjän tulisi kuitenkin muistaa, että resurssien hajauttaminen horisontaalisesti eli useammalle eri palvelimelle, on joissakin tilanteissa mahdollista. Tällaisissa tilanteissa käyttäjän tulee kuitenkin suunnitella hajauttaminen etukäteen, sillä jo valmiiksi toteutetuissa relaatiotietokannoissa hajauttaminen ei välttämättä tule kyseeseen. (Dave Pinal 2016.)

Jos tietokannan hajautusta ei ole suunniteltu etukäteen ja ainoana resurssien lisäämisen keinona on ostaa isompi ja tehokkaampo palvelin, tekee se tietokannan resurssien lisäämisestä hankalaa. Kun isompaa ja tehokkaampaa palvelinta ei enää ole, ainoana ratkaisuna on, tietokannan hajauttaminen kahdelle eri palvelimelle. Tämä on kuitenkin joillekin tietokantaohjelmistoille iso haaste, sillä ne eivät välttämättä tue kirjoitus- ja lukutoiminnallisuutta kahdella eri palvelimella. Tällainen tietokantaohjelmisto on esimerkiksi MySQL. Toimimattomuus perustuu siihen, että relaatiotietokantaohjelmistot odottavat tiedon sijaitsevan vain yhdessä paikassa. Jos data on hajautettuna kahdelle palvelimelle, ei tietokantaohjelmisto osaa ottaa toiselle palvelimelle tehtyjä kirjoitus- eli muokkaustapahtumia huomioon. Tämä tekee relaatiotietokannoista osaltaan vähemmän joustavia. Näin ne kaikki eivät aina sovi ideaaleiksi tallennusratkaisuiksi. (Eelco ym. 2010, 7.)

2.6 **Relaatiotietokantaohjelmistot**

Relaatiotietokantaohjelmistoja on saatavilla sekä avoimen lähdekoodin tietokantaohjelmistoina että maksullisina tietokantaohjelmistoina. Kaikki relaatiotietokantaohjelmistot perustuvat relaatiomalliin. Relaatiotietokantaohjelmistot ovat vuodesta toiseen pysyneet toimintamalliltaan samanlaisina, sillä relaatiomallin toimintalogiikka perustuu edelleen E.F Coddin kehittämään relaatiomalliin. (Connolly & Begg 2015, 72-73.) SQL (Structured Query Language) on standardoitunut kaikkien relaatiotietokantatoimittajien tietokantakieleksi. (Hovi 2003, 5.)

Ensimmäinen julkisille markkinoille tulleen relaatiotietokantaohjelmiston julkaisi vuonna 1979 Oracle. Se on edelleen yksi suosituimmista relaatiotietokantaohjelmistoista. (Connolly & Begg 2015, 73.) DB-Engines sivusto pitää tilastoa tietokantaohjelmistoista ja niiden suosiosta. Sivustolta löytyvän tilaston mukaan tämän hetken suosituimpiin relaatiotietokantaohjelmistoihin lukeutuvat Oraclen ohella MySQL, Microsoftin SQL Server, PostreSQL ja IBMdb2. Relaatiotietokannat pitävät top viiden suosituimman tietokantaohjelmiston paikkaa tässä tilastossa. (DB-Engines 2019.)

2.7 **MS SQL Server tietokantaohjelmisto**

Microsoftin kehittämä SQL Server on vakiinnuttanut asemaansa jo pitkään tietokantaohjelmistojen markkinoilla. Alun perin pienten järjestelmien tietokantaohjelmistoksi suunniteltu SQL Server, on nostattanut suosiotaan myös keskisuurten järjestelmien tietokantaohjelmistona. SQL Server on tullut tunnetuksi esimerkiksi helppokäyttöisyydestään. UNIX-versiota ohjelmistosta ei ollut ennen vuoden 2017 versiota. Ohjelmisto toimi siis ennen vuotta 2017 ainoastaan NT- ja Windows-ympäristöissä. (Hovi 2003, 9.)

SQL Serverin historia ulottuu aina 1980-luvulle asti, mutta ensimmäinen Microsoftin markkinoille tuoma, nykyistä toimintalogiikkaa muistuttava versio julkaistiin vuonna 1998. (McQuillan 2015.) Tällä hetkellä markkinoiden uusin versio on SQL Server 2017, mutta vuoden 2019 aikana saataneen uusin versio, SQL Server 2019, julkaistua. (Microsoft 2018.) Tässä opinnäytetyössä esitellyt esimerkit toteutetaan avoimen lähdekoodin SQL Server Express 2012-versiolla.

SQL Server perustuu relaatiotietokantojen tapaan relaatiomalliin. Tauluihin tallennetaan tarvittava data, ja taulut yhdistetään toisiinsa pää- ja viiteavaimien avulla. SQL Server koostuu pähkinänkuoressa tauluista, taulunäkymistä, proseduureista ja erilaisista toiminnallisuuksista. Järjestelmää hallinnoidaan järjestelmä- ja käyttäjätietokantojen avulla. Järjestelmätietokannoilla SQL Server ylläpitää oikeaoppista toimintaansa. Eli järjestelmätie-

tokannat sisältävät tietokannan tietorakennemallin kannalta tärkeät asetukset, eikä tavallisen käyttäjän tarvitse tai kannata koskea niihin. Käyttäjä lisää käyttäjätietokannat ohjelmistoon itse. Nämä tietokannat voivat esimerkiksi sisältää asiakkaiden tarjoaman tai käyttäjän itse luoman datan, jota käyttäjä voi muokata tarpeidensa mukaisesti. (McQuillan 2015.)

3 NoSQL-tietokannat

Relaatiotietokannat taipuvat monenlaisiin käyttötarkoituksiin, mutta joissakin tilanteissa ne eivät ole ideaaleja tietokantaratkaisuja. Eheysrajoitteet ja taulujen väliset relaatiot tekevät joissakin tilanteissa datan käsittelystä raskasta ja jossain määrin rajoittunutta. NoSQL (not only SQL) -tietokannat luotiin haastamaan relaatiotietokannat niiden hallittua tietokantamarkkinoita jo useiden vuosikymmenten ajan. (Hovi 2015.)

NoSQL-termi esiteltiin ensimmäistä kertaa Carlo Strazzin toimesta vuonna 1998. Strazzi kuvaili luomaansa avoimen lähdekoodin tietokantaa relationaalisena tietokantana, joka ei toiminnallisuudessaan hyödyntänyt SQL:ää. Mutta vasta kymmenen vuotta myöhemmin NoSQL-tietokanta -termi vakiinnutti asemansa suuren yleisön keskuudessa Evan Eriksson ja Johan Oskarssonin käytettyä termiä kuvaaman NoSQL-tietokantoja. (Dataversity 2018.)

3.1 Miksi NoSQL- tietokannat kehitettiin?

NoSQL-tietokantojen tärkeimpiä ominaisuuksia joustavuuden ja skaalautumisen ohella, ovat niiden kyky käsitellä suurta määrää strukturoimatonta eli jäsentelemätöntä dataa. Tämä tekee niistä helppokäyttöisiä ja nopeita. Helppokäyttöisiä siksi, että käyttäjän ei tarvitse ennalta muokata tietokantaan syötettävää dataa, vaan se voidaan tallettaa sellaisenaan esimerkiksi olio-muotoisena. Nopeita siksi, että NoSQL-tietokannat ovat erittäin tehokkaita. Mutta ennen kaikkea ne kehitettiin sen takia, että datan käsittelystä haluttiin joustavampaa. Relaatiotietokannat kun noudattavat datan käsittelyssä aina tietynlaista kaavaa. NoSQL-tietokannat ovat myös useimmiten avoimen lähdekoodin kantoja. Täytyy kuitenkin muistaa, että sekä relaatiotietokantojen että NoSQL-tietokantojen ohjelmistoista on sekä maksullisia että maksuttomia versioita. NoSQL-tietokantojen kehittymiselle tietä ovat luoneet strukturoimattoman datan nopeaa käsittelyä vaativat sovellukset, kuten Facebook ja Twitter. (Rys 2011, 48.)

3.2 Miten NoSQL-tietokannat eroavat relaatiotietokannoista?

Huomattavin ero NoSQL-tietokannoissa ja relaatiotietokannoissa on, ettei NoSQL-tietokantojen relaatiotietokannoista poiketen tarvitse perustua mihinkään yhteen tiettyyn malliin. Relaatiotietokantoja luodessa taulurakenteet ja eheyssäännöt täytyy määritellä, ennen kuin tietoa voidaan viedä kantaan. Toisin kuin relaatiotietokannoissa, ei NoSQL-tietokannoissa tietokantakaaviorakennetta tarvitse määritellä etukäteen, vaan kaaviorakenne voidaan määritellä vasta datan lisäämisvaiheessa. Tästä joustavuudesta on paljon

hyötyä, kun kantaan tehdään usein muutoksia. NoSQL-tietokannat voivat perustua myös useampaan kuin yhteen malliin. (Hovi 2015.)

NoSQL-kannoissa ei siis myöskään ole relaatiotietokantojen viite-eheyksiä, tai taulujen välisiä pakollisia relaatioita, jotka varmistaisivat tiedon, ja ennen kaikkea tietokannan eheyden. Vastaavaa roolia ajavia sääntöjä pystytään kuitenkin määrittelemään ohjelmistokohtaisesti jonkin verran. On kuitenkin huomioitava, että vaikka NoSQL-tietokannat eivät sisältäisikään näitä perinteisiä tiedon oikeaoppisuuden varmistavia määrityksiä, ei se tarkoita, etteivätkö ne silti olisi tietorakenteeltaan luotettavia tai etteikö niihin voisi tallentaa tärkeää dataa. (Hovi 2015.)

3.3 **NoSQL-tietokantojen skaalautuminen**

Jos tiedon eheydestä voidaan tinkiä hieman, tarjoaa se mahdollisuuden tehokkaammalle ja skaalautuvammalle järjestelmälle. Skaalautumisella NoSQL-tietokantojen yhteydessä viitataan NoSQL-tietokantojen kykenemiseen horisontaaliseen skaalautumiseen. Niin kuin jo aikaisemmin mainitsin, skaalautumisella tarkoitetaan tietokannan mahdollisuuksia ja valmiuksia resurssien lisäämiseen. Relaatiotietokantojen kohdalla skaalautuminen on ollut ensisijaisesti ylöspäin skaalautumista. NoSQL-tietokannat taas skaalautuvat horisontaalisesti eli lineaarisesti. Lineaarinen skaalautuminen mahdollistaa tiedon tallentamisen useammalle kuin yhdelle palvelimelle. Esimerkiksi MongoDB:n dokumenttipohjaisen tietokantaohjelmiston kohdalla palvelimia voi olla satoja. Tietoa voidaan pilkkoa useammalle pienemmälle ja tehottomammalle palvelimelle, mutta kokonaisuus säilyy silti yhtenäisenä. Tämä horisontaalinen skaalautuvuus tekee siis NoSQL-tietokannoista relaatiotietokantoja resurssien lisäämisen suhteen joustavampia. (Eelco ym. 2010,7.)

3.4 **NoSQL-tietokantaohjelmistot**

NoSQL-tietokantoja voi olla monenlaisia, mutta yleisesti ne voidaan jakaa neljään eri kaavamalliin: dokumentti-, column family-, key value- ja graph -pohjaisiin tietokantamalleihin. Tiedon tallennusmuoto on tietokantakohtainen. Esimerkiksi MongoDB:ssä tieto talletetaan JSON-formaatissa (Konkka 2016.) Oikeanlaista NoSQL-tietokantaa valikoitaessa täytyy käyttäjän olla tietoinen siitä, mitkä ominaisuudet ovat käyttäjän omasta näkökulmasta tärkeimpiä ominaisuuksia tietokantaohjelmistossa. Onko esimerkiksi tärkeää, että ohjelmisto on tehokas tai nopea? Vai onko datan pitkäaikainen säilömisen mahdollisuus tärkeintä? Näin ollen käyttäjä joutuu myös joustamaan todennäköisesti joistakin toiminnallisuuksista. Tämä ei välttämättä kuitenkaan ole käyttäjää haittaava tekijä, jos tietokantaa tarvitaan esimerkiksi suorittamaan vain tehokkuutta vaativia tehtäviä. (McCreary & Kelly 2013.)

DB-Engines sivuston mukaan suosituin NoSQL-tietokantaohjelmisto on MongoDB. MongoDB on kuudenneksi suosituin kaikista tietokantaohjelmistoista, edellä viisi relaatiotietokantaohjelmistoa. Muita suosittuja NoSQL-tietokantaohjelmistoja ovat key-value pohjainen Redis, Column family -tietomalliin pohjautuva Cassandra sekä dokumenttitietomalliin ja key-value -tietomalliin perustuva Amazon DynamoDB. (DB-Engines 2019.)

3.5 **Column family -tietokantamalli**

Column family -tietokantamallit, kuten esimerkiksi Cassandra, soveltuvat hyvin Big Datan käyttöön. Column family -tietokannat soveltuvat käsittelemään dataa joka on laajaa, hajautunutta ja sisällöltään vaihtelevaa. Column family -tietokannat mielletään usein tietovarastoiksi tietokantojen sijaan. Näiltä ohjelmistoilta puuttuu monesti sellaisia toiminnallisuuksia, jotka mielletään tietokantaohjelmistoille ominaisiksi. Näissä tietokantaohjelmistoissa ei monesti esimerkiksi välttämättä ole nimettyjä sarakkeita, tai kaavioon sulautettuja dokumentaatioita, jotka tarjoaisivat lisää tietoa alkuperäisestä aiheesta. Column family -tietokantamalliin perustuvat tietokantaohjelmistot käyttävät sarake- ja rivikohtaisia identifikaattoreita kyselyiden suorittamiseen. Käytännössä tämä tarkoittaa, että tiedon lokalisointi tapahtuu rivin ja sarakkeen hakuyhdistelmällä. Jos tietomallista (katso kuva 3) haluttaisiin esimerkiksi löytää "Etsi minut!"-tieto, tulisi käyttäjän etsiä dataa 2B yhdistelmällä. (McCreary & Kelly 2013.)

	A	B	C
1			
2		Etsi minut!	
3			

Kuva 3. Esimerkki Column family -tietokantamallista (mukaillen McCreary & Kelly 2013.)

Kuvissa 4 nähdään esimerkki column family -tietokantamallista, kun kantamallia verrataan relaatiomallin esimerkkiin (kuva 5). Tiedolle on olemassa vain yksi rivikohtainen avain, mutta yhdelle riville voidaan määritellä useita eri tietoja. Käyttäjän täytyy kuitenkin tietää avainyhdistelmä näiden tietojen löytämiseksi.

Rivi_id	Nimi	Puhelinnumero	Kunta
1	Elsa	040 453 343	Helsinki
2	Matti	050 347222	Helsinki
3	Maija	050 321 1112	Helsinki

Kuva 4. Esimerkki column family -tietokantamallista, kun kantamallia verrataan relaatiomallin esimerkkiin.

Pankkirelaatio

panktunnus (PK)*	panknimi	koodi
1	Nordea	121
2	Osuuspankki	213
3	Danske Bank	1313
4	Suomen Pankki	313131

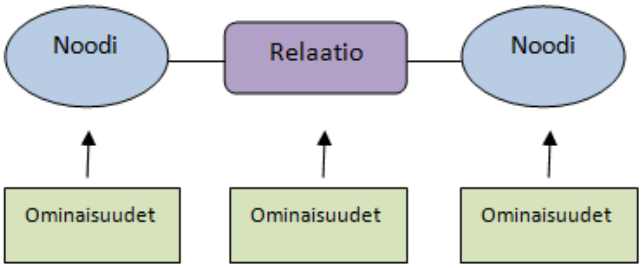
Henkilörelaatio

htun	etunimi	snimi	kunta	**panktunnus(FK)***
111191-111A	Elsa	Mäkinen	Helsinki	1
110381-1123	Matti	Meikäläinen	Helsinki	2
080895-3212	Maija	Mattila	Helsinki	4
040591-1334	Anneli	Auttava	Helsinki	1

Kuva 5. Relaatiotietokantatauluja vertailua varten (mukaillen Hovi 2013) * PK= Pääavain (Primary key), FK= Viiteavain (Foreign key).

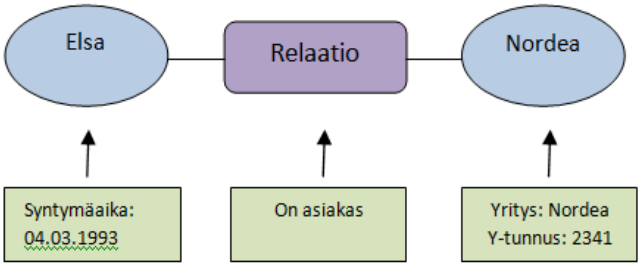
3.6 **Graph-tietokantamalli**

Graph-tietokantamallit taas soveltuvat datan välisten riippuvuuksien esittämiseen. Graph-malliin perustuvat tietokantaohjelmistot voivat sisältää tietoa esimerkiksi sosiaalisen median kanaviin tehtävistä kyselyistä. Grap-tietokantamallilla selitetään muunmuuassa objektien välisiä suhteita. Tietokantamallille ominaisia ominaisuuksia ovat kyselyt ja tiedon varastointi. Nimensä mukaisesti data esitetään tietokantaohjelmistoissa kaavioina (katso kuva 6), joista ilmenee tietojen väliset suhteet, relaatiot. Graph- tietokantamallissa data määritellään niin sanottujen noodien avulla. Noodit voivat olla esimerkiksi tietoa yrityksistä, henkilöistä tai puhelinnumeroita. Näille noodeille määritellään relaatiot ja noodeille ja relaatioille voidaan määritellä yksilökohtaisia tietoja. Relaatioiden avulla pystytään esimerkiksi selvittämään, ovatko nämä kaksi noodia yhteydessä toisiinsa vai eivät. Useita tietoryppäitä linkittäviin monimutkaisiin hakuihin Graph-tietokantamalli ei sovellu. (McCreary & Kelly 2013.)



Kuva 6. Esimerkki Graph-tietokantamallista (mukaillen McCreary & Kelly 2013.)

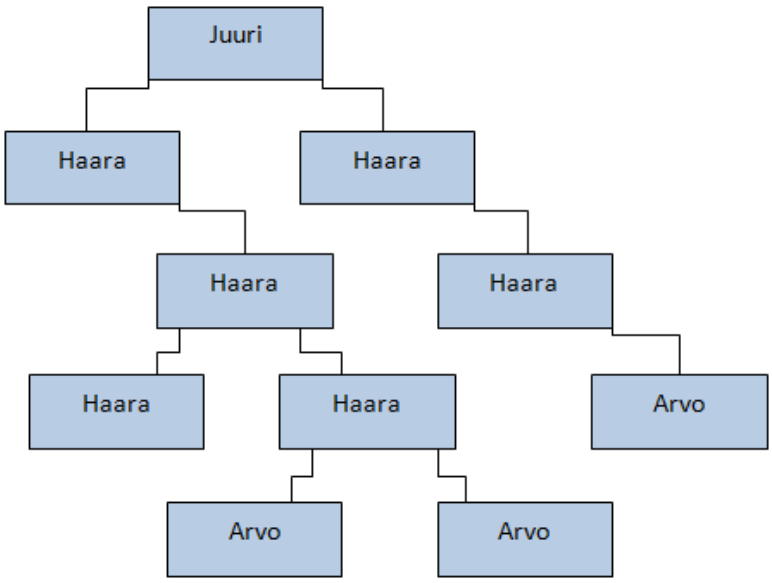
Kuvasta 7 voidaan nähdä miltä Graph-tietokantamallin tietosisältö näyttää, kun siihen sisällytetään relaatiomallin tietoja (katso kuva 5).



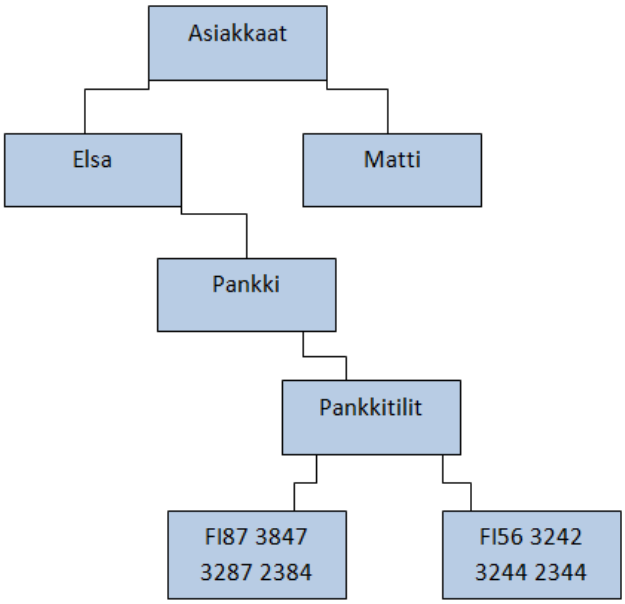
Kuva 7. Esimerkki Graph-tietokantamallista, kun kantamallia verrataan relaatiomallin esimerkkiin. Elsalle voidaan määritellä yhteys Nordea-pankkiin relaation avulla. Näille yksittäisille tiedoille voidaan vielä määritellä omat tietonsa. Esimerkiksi Elsan lisätiedoista voidaan selvittää, että hän on syntynyt 04.03.1993.

3.7 **Dokumenttipohjainen tietokantamalli**

Dokumenttipohjaiset tietokantamallit ovat NoSQL-tietokantamalleista suositumpia. Dokumenttipohjaisiin tietokantamalleihin perustuvat tietokantaohjelmistot ovat tulleet tunnetuiksi joustavuudestaan ja tehokkuudestaan. Dokumenttipohjaiset tietokantaohjelmistot soveltuvat parhaiten esimerkiksi ison datamäärän pitkäaikaiseen tallettamiseen ja nopeisiin hakuihin. Dokumenttipohjaisten tietokantaohjelmistojen toimintaperiaate perustuu datan indeksointiin. Indeksoinnin avulla järjestelmä tunnistaa tiettyyn aihe-alueeseen liittyvän datan helposti. Näitä tietoja pystytään useimmissa tapauksissa hakemaan tietokantaohjelmistokohtaisilla ohjelmointikyselykielillä. Tämä tekee dokumenttipohjaisten tietokantajärjestelmien tietojen hausta käyttäjäystävällistä ja tehokasta. Kuten kuvasta 8 nähdään, arvo voidaan hakea useammankin haaran kautta, kunhan haarat on linkitetty toisiinsa dokumenttien indeksien avulla. Dokumenttipohjaisia tietokantaohjelmistoja ovat esimerkiksi BaseX ja MongoDB. (McCreary & Kelly 2013.)



Kuva 8. Esimerkki dokumenttipohjaisesta tietokantamallista (mukaillen McCreary & Kelly 2013.)



Kuva 9. Esimerkki dokumenttipohjaisesta tietokantamallista, kun kantamallia verrataan relaatiomallin esimerkkiin (katso kuva 5). Kuvasta voidaan nähdä miten relaatiomallin pankkitietoja on ilmennetty kyseisessä tietokantamallissa. Asiakkaiden tiedoista pystytään etsimään kaikki tarvittava asiakkaan pankkitietoihin liittyvä data dokumenttien liitosten takia.

3.8 **Key value -tietokantamalli**

Dokumenttipohjaisten NoSQL-tietokantojen ohella, key value -tietokannat ovat myös käyttäjien suuressa suosiossa. Key value -tietokannat saattavat olla erittäin tehokkaita, mutta myös äärimmäisen rajoittuneita. Rajoittuneisuutta aiheuttaa se, että datalle pystytään antamaan ainoastaan yksi avain-arvo (key). Rajoittuneisuus johtuu myös siitä, että niissä on monesti saatettu joustaa jostakin perustoiminnallisuudesta, jotta järjestelmässä on saatu nostettua esimerkiksi tehokkuutta. Tästä hyvä esimerkki on Memcache key value -tietokantaohjelmisto, joka on ollut käytössä ainakin sellaisilla isoilla yrityksillä kuten Facebook ja LiveJournal. (McCreary & Kelly 2013.)

Key value -tietokantamallin tehokkuus perustuu sen nopeuteen. Nopeutta saadaan, kun jätetään kaksi käyttäjien perspektiivistä mahdollisesti tärkeää toiminnallisuutta pois. Ensinnäkin järjestelmään ei pystytä tekemään tavallisia hakuja. Tiedoista voi etsiä hakemaansa vain key-avaimella. Se ei myöskään tallenna tietoja levylle, vaan välimuistiin, joka mahdollistaa tiedon katoamisen esimerkiksi sähkökatkoksen aikana. Tämä ei välttämättä kuitenkaan ole käyttäjille este, jos tietokannan tarkoituksena ei olekaan säilöä dataa loputtomiin. (McCreary & Kelly 2013.) Kuten kuvasta 10 nähdään key-value tietokantamalli sisältää vain avaimen ja sille osoitetun arvon. Vaikka dataa voidaan tallettaa monessa eri muodossa, ei sitä voida hakea kuin kyseiselle tietueelle asetetulla key-arvolla.

Key (avain)	Value (arvo)
Kuva-131324.jpg	Kuva
http://www.esimerkki.com/esimerkki.html	Sivuston HTML
N:/folder/data/dataa.pdf	PDF dokumentti
Kuva-134553424.jpg	Kuva

Kuva 10. Esimerkki key-value tietokantamallista (mukaillen McCreary & Kelly 2013.)

Key (avain)	Value (arvo)
Etunimi	Elsa
Sukunimi	Mäkinen
Kunta	Helsinki
Pankki	Nordea

Kuva 11. Esimerkki key-value tietokantamallista, kun kantamallia verrataan relaatiomallin esimerkkiin (katso kuva 5). Kuvasta voidaan nähdä miten esimerkiksi nimiarvo voidaan asettaa vain yhdellä key arvolle.

3.9 **MongoDB**

Ensimmäinen julkisille markkinoille tuotu versio MongoDB:stä julkaistiin vuonna 2009. Se nostatti nopeasti suosiotaan varsinkin nuorten ohjelmoijien parissa. MongoDB on niin sanottu dokumenttivarasto, johon tieto talletetaan JSON formaatissa. MongoDB soveltuu erinomaisesti monimutkaisten tietojen tallettamiseen ja käsittelemiseen. Ohjelmistoa voidaan soveltaa esimerkiksi palvelimilla, virtuaalikoneissa tai pilvipalveluiden applikaatioissa. MongoDB:n tiedetään olevan käytössä esimerkiksi sellaisilla suuryrityksillä kuin eBay ja New York Times. (Eelco ym. 2010, 1.)

MongoDB kehiteltiin suurten tietovarastojen skaalautuvuuden helpottamiseksi. Tavoitteena oli myös kehitellä tuote, jossa tietokannan tietynlaisten tietojen käsittely olisi mahdollisimman helppoa. Ohjelmoijien keskuudessa se on nostattanut suosiotaan sen oliotyyppisen datan sovellettavuuden vuoksi. Olio-muotoisen datan tallettaminen kantaan on helppoa, sillä toisin kuin relaatiotietokannoissa dataa ei tarvitse muokata ennen kantaan tallettamista, vaan se hyväksyy olio-muotoisen datan sellaisenaan. Toisin sanoen data on siis strukturoimatonta. (Ari Hovi 2015.)

Vaikka MongoDB:n toiminnallisuuksiin mielletäänkin strukturoimattoman datan käyttö, on kuitenkin jonkinasteinen kannan sisällön suunnittelu suositeltavaa. Dokumenttien muokkaaminen on helppoa, sillä dokumentit eivät ole riippuvaisia toisistaan. Dokumentteihin voidaan kuitenkin liittää lisätietoa, esimerkiksi henkilöitä koskevaan dokumenttiin voidaan sulauttaa indeksien avulla osoitetietoja koskeva dokumentti. Tämä tulisi aina huomioida tietokantaa toteutettaessa, vaikkei tietorakenteen rakennetta tarvitsekaan suunnitella etukäteen. (Eelco ym. 2010, 7.)

MongoDB:lle kehitettiin oma ohjelmointikieli BSON, joka muistuttaa paljon rakenteeltaan JSON ohjelmointikieltä. JSON -kielestä poiketen BSON -kielelle on kehitelty joitakin tietokantaohjelmiston käyttöä helpottavia ominaisuuksia, joita ei sellaisenaan JSON -kielessä ole. BSON tekee tietokantaohjelmiston toiminnasta nopeampaa, sillä se helpottaa tietokoneen tekemiä hakua ja prosesseja kantaan**.** Ajatuksena on, että jos käyttäjä osaa kirjoittaa JSON- ohjelmointikieltä, riittää se kattamaan tarvittavan osaamisen MongoDBn käytölle. (Eelco ym. 2010, 5-6.)

Jotta ymmärrettäisiin miten MongoDB toimii, esimerkiksi SQL Serveriin verrattuna, on nostettava esille muutama seikka. Relaatiotietokannoissa käytetyt rivit vastaavat MongoDB:ssä dokumentteja. Koska dokumenttien sisältö voi olla melkein mitä tahansa, ovat ne tietorakenteeltaan paljon joustavampia kuin relaatiotietokantojen rivit. Varchar, dateti-

me yms. arvojen sijaan käytetään labeleita, joilla kuvaillaan kannan sisältöä. Pääavain (PrimaryKey), on korvattu uniikilla identifioijalla (_id). Kaikilla kentillä ei myöskään tarvitse olla arvoa, toisin kuin relaatiotietokannoissa, joissa käytetään NULL-arvoa, jos kentän arvoa ei tiedetä. Dokumenteissa ei ole tauluja vaan niiden virkaa ajavat niin sanotut kokoelmat (collection). Käyttäjä saa itse määritellä kokoelmien sisällön, ja kuinka monta hän niitä tekee. Kokoelmien ei tarvitse sisältää esimerkiksi pelkästään autoja koskevia tietoja, vaan kokoelman aiheena voi olla esimerkiksi kulkuneuvot tarkoittaen, että sinne voi olla koottuna tiedot kaikista autoista, veneistä yms. (Eelco ym. 2010, 10-11.)

MongoDB:n perustoiminnallisuuteen kuuluu ajatus siitä, että tietokannasta tehdään aina yksi tai useampi kopio muille palvelimille. Tämä eliminoi ongelmatilanteita, joissa tietokantaan kuuluva data tuhoutuisi yhden palvelimen tuhoutuessa. Jos yhdelle laitteistolle sattuu jotain, on eheä versio kannasta löydettävissä toiselta palvelimelta. MongoDB:n nopea toiminta saattaa olla joskus myös haitaksi tietokannalle. Virhetilanteissa ohjelmiston käyttämät oikotiet sen tehokkuuden takaamiseksi saattavat jopa sellaisenaan pahentaa virhetilanteita. (Eelco ym. 2010, 5.)

Miten MongoDB sitten eroaa esimerkiksi key value -tietokannoista kuten aikaisemmin mainitsemastani Memchacedista? MongoDB:n toiminnallisuus perustuu sen kykyyn tallettaa dataa pitkäaikaisesti useammalle eri palvelimelle ja levylle. Jos ohjelmiston tietokantayhteys esimerkiksi katkeaa, on tietokannan tietojen haku mahdollista silloin levyltä. Tämä toiminnallisuus on mahdollinen ilman, että ohjelmisto joutuu joustamaan huomattavasti tehokkuudestaan. Mutta silti se on hitaampi kuin Memcache, joka on tehokkuudessaan omaa luokkaansa. Memcachen ei tarvitse joustaa tehokkuudestaan datan tallettamisen suhteen, koska data talletetaan vain välimuistiin, josta se häviää aikanaan. (Eelco ym. 2010, 8.)

Entä miten MongoDB eroaa esimerkiksi CouchDB:stä, toisesta dokumenttivarastosta, joka on myös MongoDB pahimmaksi kilpailijaksi nimetty tietokantaohjelmisto? Relaatiotietokantojen tapaan MongoDB tukee dynaamisten hakujen (query) tekemistä. Eli hakujen tekemistä kantaan spesifioiduin hakuehdoin. Usein tätä toiminnallisuutta voidaan pitää tietokantaohjelmistojen peruspilarina, mutta CouchDB on luonut tähän uudenlaisen, korvaavan ratkaisun. Käyttäjän laatimien hakujen sijaan se on tallettanut datan valmiiksi oletettujen hakujen mukaisesti. Koska CouchDB:n data on binääristä, sitä voidaan käsitellä eri tavoin kuin esimerkiksi MongoDB:n dataa. Data on tällöin määritelty valmiiksi indeksein ja silloin se on myös saatavissa erittäin nopeasti. Tietokantaohjelmiston ei siis tarvitse tehdä joka kerta erillistä hakua, joka veisi aikaa ja resursseja. Hakutulokset ovat ja pysyvät samoina valmiina omilla listoillaan. Jos kantaan tulee uutta dataa, lisätään se vain sille

listalle, jolle se indeksin mukaisesti kuuluu. Jotta tietokantaohjelmiston käyttö onnistuisi, tarvitaan komentojen luomiseen kokeneempaa ohjelmoijaa. Siksi MongoDB:tä dynaamisine komentoineen, voidaan pitää helppokäyttöisempänä kuin CouchDB:tä. (Eelco ym. 2010, 12.

4 Sovelluksen palvelutarve

Tietokantaohjelmistoja on siis monia erilaisia. Tietokantaohjelmistoilla on myös omat ominaisuutensa, jotka tulisi ottaa huomioon sovelluskehityksessä. Käyttäjäkohtaisia tarpeita voi olla monenlaisia, esimerkiksi jollekulle saattaa olla tärkeää monimutkaisten kyselyiden tekemisen mahdollisuus, kun taas jollekulle toiselle tärkeää on tehokas välimuistina toimiva tietokanta, johon on mahdollista tehdä vain yksinkertaisia kyselyitä. Tietokantaohjelmistojen suorituskykyyn vaikuttavat toiminnallisuusominaisuuksien ohella, datan määrä ja rakenne, ja tämä tulisi myös huomioida sovelluskehityksessä.

Kuvassa 12 on vertailtu relaatiotietokantaohjelmistojen ja NoSQL-tietokantaohjelmistojen merkittävimpiä eroavaisuuksia. Positiiviseksi mielletyt ominaisuudet on esitetty vihreällä värillä, ja negatiiviseksi mielletyt ominaisuudet punaisella värillä. Kuten taulukosta voidaan havaita, kummallakin osapuolella on sekä positiiviset, että negatiiviset puolensa. Näitä tietoja voidaan hyödyntää, kun vertaillaan sovelluskehityksen kannalta tärkeimpiä ominaisuuksia.

Relaatiotietokantaohjelmistot	NoSQL-tietokantaohjelmistot
Strukturoidun datan tallettamisen pakollisuus.	Strukturoimattoman datan tallettamisen mahdollisuus.
Pääasiassa ylöspäin skaalautuva.	Horisontaalisesti eli lineaarisesti skaalautuva.
Eheysrajoitteet	Datan ei tarvitse olla liitettynä mihinkään tiettyyn avaimeen.
Valmiiksi rakennettu tietokannan toimintamalli proseduureineen.	Toimintamalli sääntöineen pitää järjestelmäkohtaisesti itse määritellä ohjelmistolle.
Normalisoitu data, joka hidastaa järjestelmää joissakin tilanteissa.	Normalisoimaton data tekee datan päivittämisestä työlästä.
Vähemmän avoimen lähdekoodin tietokantaohjelmistoja.	Enemmän avoimen lähdekoodin tietokantaohjelmistoja.
Skeema	Skeemattomuus
SQL	Vakiintumaton kyselykieli

Kuva 12. Tärkeimmät eroavaisuudet (mukaillen Keith Foote 2016.)

Relaatiotietokantoja suositaan eheyden ylläpitoa painottavissa järjestelmissä. Vaikka relaatiotietokannat mielletäänkin usein rajoittuneisuuden takia monimutkaisiksi ja hanka-

lemmiksi käyttää kuin NoSQL-tietokannat, ovat ne kuitenkin omalla osa-alueellaan edelleen lyömättömiä. (Virkki 13.5.2019.) Toisin sanoen monimutkaisuudelle on syynsä, sillä relaatiotietokannoissa myös tietosisältö on monimutkainen. Tiedon eheyttä ylläpitävät toiminnallisuudet vaativat käyttäjältä hieman kärsivällisyyttä järjestelmää toteutettaessa. Kuitenkin, jokaiselle tietueelle on oma paikkansa järjestelmässä, joka osaltaan myös helpottaa käyttäjän työtä. Hukatun tiedon uhka on pienempi, kuin strukturoimattomissa NoSQL-tietokantajärjestelmissä.

NoSQL-tietokantojen teho perustuu strukturoimattoman datan nopeaan tallettamiseen ja käyttöön. (Rys 2011, 48.) Tietokantamallit taipuvat moniin erilaisiin käyttötarkoituksiin, ja niitä voidaan myös soveltaa yhteiskäytössä. (Hovi 2015.) Tämä tekee NoSQL-tietokannoista monikäyttöisiä. Käyttäjä pystyy itse melko hyvin hallitsemaan ja vastaamaan järjestelmänsä tarpeisiin, jo siinä vaiheessa, kun tietokantajärjestelmä on käyttäjän käytössä. Vaikkei datan eheyttä ylläpidetä niin tehokkaasti kuin relaatiotietokannoissa, ei se silti tarkoita, etteivätkö NoSQL-tietokantajärjestelmät sovi tärkeänkin tiedon tallettamiseen. Järjestelmät ovat joustavia, sillä tietorakennetta voidaan määritellä milloin tahansa. Koska tietoa ei asetella välttämättä omiin lokeroihinsa, altistaa se järjestelmän relaatiotietokantaohjelmistoja todennäköisemmin virhetilanteisiin.

4.1 **Relaatiotietokantojen ja NoSQL-tietokantojen yhteiskäyttö**

Relaatiotietokantoja ja NoSQL-kantoja on sovellettu paljon myös yhteiskäytössä. Yhteiskäyttöä on sovellettu tilanteissa, joissa tarvitaan yhtä aikaa tehokas, hyvin skaalautuva, mutta myös tiedon talletuksen eheyden ylläpitävä tietokanta. Viimeisen vuosikymmenen aikana se onkin ollut ainoa mahdollinen ratkaisu NoSQL-tietokantojen ja transaktioiden yhdistämiselle. Tällaisissa tilanteissa voidaan esimerkiksi lisätä relaatiokantaan tilitiedot ja NoSQL-tietokantaan dokumentit. Käytännössä tällaisiin ratkaisuihin ovat päätyneet sellaiset yritykset, kuten Sourceforge. (Eelco ym. 2010, 6.)

5 Transaktiot

Transaktioilla varmistetaan tietokannan eheys. Transaktion tarkoituksena on siirtää tietokanta yhdestä eheästi, toiseen eheään tilaan. Käytännössä tämä tarkoittaa sitä, että transaktio pyrkii estämään virheellisen tiedon pääsyn tietokantaan. (Laiho & Wendelius 2015,1.) Transaktiot koostuvat komennoista ja komentosarjoista. Jos yksikin virheellinen komentorivi komentosarjassa havaitaan, koko transaktio peruutetaan. Transaktioita käytetään tiedon hakuun ja tallettamiseen. (McQuillan 2015.) Vaikka tietokantaohjelmistojen luotettavuus on nykypäivänä huipussaan, tarvitaan tietokantatransaktioita tietokannan tietojen eheyden ja oikeaoppisuuden ylläpitämiseen. (Laiho & Wendelius 2015,1.)

Jos esimerkiksi halutaan varmistaa, että käyttäjän pankkitilin saldo päivittyy tietojärjestelmässä oikein, tarvitaan tähän transaktioita. Ilman transaktioita käyttäjä saattaa esimerkiksi saada väärää tietoa pankkitilinsä saldosta. Jos transaktioita ei olisi, saattaisi hälytysjärjestelmän tekemä hälytys jäädä kirjaamatta. Silloin ei esimerkiksi hälytyksen tarkastelija tietäisi, mikä aiheutti esimerkiksi koko ostoskeskuksen evakuoimisen. (Laiho & Wendelius 2015,21.) Transaktioita käytetään siis hyvin laajalti, ja monesti niiden merkitystä ei ymmärretäkään, ennen kuin jokin transaktioiden hallinnassa menee vikaan.

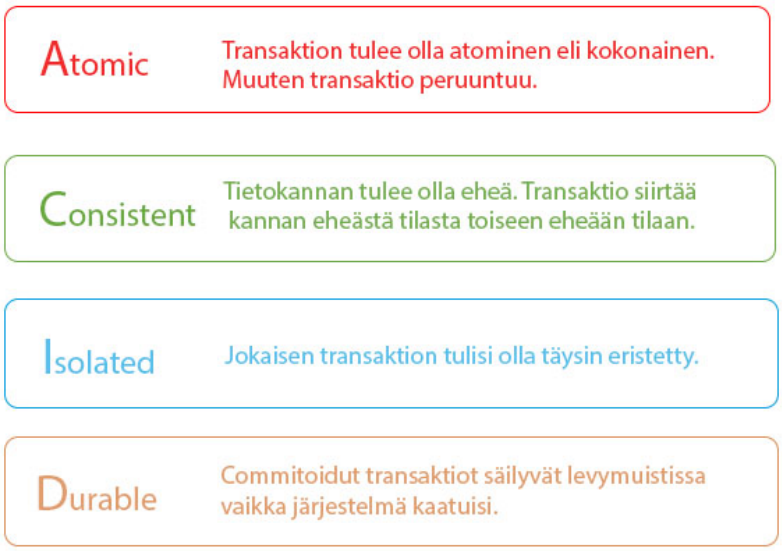
Transaktioiden hallinta on jo pitkään ollut relaatiotietokantaohjelmistoille ominainen ominaisuus. Viime vuosina transaktioiden hallinta on kuitenkin kasvattanut suosiotaan myös NoSQL-tietokantaohjelmistojen parissa. Sovellusohjelmoinnissa transaktioiden käsittely on hieman erilainen, jolloin transaktioprotokollaa käytetään API-liittymillä. (Laiho & Wendelius 2015,1.)

5.1 **ACID-periaate (Atomic, Consistent, Isolated, Durable)**

Relaatiotietokantoihin ja NoSQL-tietokantoihin voidaan soveltaa ACID-periaatetta. Tämä Theo Häderin ja Andreas Reuretin kehittämän periaatemallin pääkäyttötarkoitus, on mahdollistaa useamman käyttäjän samanaikainen transaktioiden käyttö. Ideana on, että jos komentosarja ei täytä kaikkia näitä periaatteen sisältämiä vaatimuksia, tulee se kokonaisuudessaan peruuttaa. ACID-periaate koostuu sanoista atomisuus, johdonmukaisuus, eristys ja kestävyys. ACID-periaatteen täydellistä noudattamista on kuitenkin hieman sovellettu relaatiotietokantaohjelmistojen kohdalla. Nykypäivän tietokantaohjelmistot eivät kaikki pysty noudattamaan tätä totaalisen eristäytyneisyyden sääntöä täydellisesti. Totaalinen eristäytyminen johtaisi järjestelmän huomattavaan hidastumiseen. (Laiho & Wendelius 2015,26.) Voitaneen kuitenkin sanoa, että jos tietokantaohjelmisto noudattaa ACID -periaatetta, täyttää se toiminnallisuudeltaan ne vaatimukset, joita siltä odotetaan. Näin

voidaan esimerkiksi välttyä likaisen lukemisen, eli virheellisten tai keskeneräisten komentojen lukemisen kaltaisilta ongelmatilanteilta. (Laiho & Wendelius 2015,25.)

Kuten kuvasta 13 nähdään, ACID-periaate on neljäosainen, sisällyttäen ehdot atomisuus, johdonmukaisuus, eristys ja kestävyys. Atomisuudella viitataan "kaikki tai ei mitään" periaatteeseen. Transaktion sisältämän komentosarjan tulee olla eheä alusta loppuun. Jos jokin osa virheellisyyden vuoksi peruuntuu, tulee koko transaktion peruuntua. Toinen periaate on eheys, jolla viitataan tietokannan viemistä yhdestä eheästä toiseen eheään tilaan. Eli transaktiota ennen vallinnut tietokannan eheys, säilyy transaktion jälkeenkin. Kolmas periaate on eristyneisyys. Eli jokaisen transaktion tulisi olla täysin eristetty muista transaktioista tarkoittaen, että mitkään vaiheet keskeneräisestä transaktiosta eivät ole näkyvissä muille käyttäjille. Kestävyydellä tarkoitetaan järjestelmän kestävyyttä ja kykyä palautua esimerkiksi järjestelmän kaatuessa. Ajatuksena on, että tieto talletetaan välimuistin sijaan levymuistille, josta se ei voi järjestelmän kaatuessa kadota. (Laiho & Wendelius 2015,25.)



Kuva 13. ACID-periaate.

6 Relaatiotietokantojen transaktiot

Relaatiotietokannat tunnetaan ennen kaikkea transaktioiden hallinnastaan. Siksi ne soveltuvat erinomaisesti esimerkiksi pankkitilejä hallinnoivien järjestelmien tietokannoiksi. Relaatiotietokantojen transaktioiden hallinta perustuu relaatiotietokannoissa pitkään hallinneeseen toiminnallisuuteen, jota esitellään seuraavien kappaleiden aikana.

Ennen tietokannan ja transaktioiden toteutusta, tulee käyttäjän huomioida, että tietokannat rakennetaan liiketoimintasuunnitelmien mukaisesti. Liiketoimintasuunnitelma määrittelee myös transaktioiden rakennetta. Transaktiot koostuvat tietokanta- ja käyttäjätransaktioista. Transaktiot perustuvat liiketoimintatransaktioon, joka selittää käyttötapausten (use case) avulla sovellussuunnitelmaa. Käyttäjätransaktiot eli dialogit käydään käyttäjän ja tietokannan välillä, tietokantatransaktiot eli tietokantadialogit kuitenkin vasta kirjaavat itse transaktion tietokantaan. Tietokantadialogien tarkoituksena on myös kerätä aluksi tietoa käyttäjädialogeja varten. Osa toimintalogiikasta määritellään tietokantaan valmiiksi proseduureina. Proseduurit määrittävät esimerkiksi sen, mitä tapahtuu komennolle, jos komennon vahvistamisen aikana tapahtuu sähkökatkos. (Laiho & Wendelius 2015,2.)

6.1 **Relaatiotietokantojen transaktio- komennot**

Transaktiot toimivat järjestelmäkohtaisesti. Transaktioita voidaan hallita istuntokohtaisten käynnistys-, peruutus- ja vahvistus-komentojen avulla. Transaktioiden manuaalista käyttöönoton tarvetta määrittelee esimerkiksi se, onko järjestelmä automaattisessa komentojen hyväksyttämistilassa (AUTOCOMMIT). (Laiho & Wendelius 2015,4.)

Jotkin järjestelmät toimivat oletuksena AUTOCOMMIT-tilassa, jolloin kaikki käyttäjän tekemät muutokset tallentuvat automaattisesti. Tällaisia AUTOCOMMIT- järjestelmiä ovat esimerkiksi SQL Server ja MySQL. Tämä mahdollistaa virhetilanteet, joissa käyttäjä tekee muutoksia, joita käyttäjä ei voi enää halutessaan perua. AUTOCOMMIT -toiminnallisuus, saattaa siis aiheuttaa tietojärjestelmän tiedon oikeaoppisuutta uhkaavia tilanteita. (Laiho & Wendelius 2015,4.)

Istuntokohtaisia transaktioita käytetään, kun järjestelmä on AUTOCOMMIT tilassa. Istuntokohtaisissa transaktioissa transaktiota hallitaan manuaalisesti. Istuntokohtaisia transaktioita hallitaan esimerkiksi BEGIN-, START-, ROLLBACK -, COMMIT TRANSACTION JA START WORK -komennoilla. BEGIN -komento aloittaa transaktion, ROLLBACK -komento peruuttaa tehdyt muutokset ja COMMIT -komento vahvistaa tehdyt muutokset. COMMIT TRANSACTION -komennon jälkeen transaktiomuutoksia ei pystytä enää peruuttamaan.

Jos käyttäjä ei tee BEGIN TRANSACTION -komentoa istuntokohtaisesti, ei transaktiota aloiteta ollenkaan. Tällöin ei käyttäjän tekemiä muutoksia pystytä peruuttamaan edes ROLLBACK TRANSACTION -komennolla. (Laiho & Wendelius 2015,4.)

Joissakin järjestelmissä, AUTOCOMMIT-tila ei kuitenkaan toimi oletuksena, jolloin transaktiota ei tarvitse erikseen määritellä aloitetuksi, tai järjestelmää poistaa AUTO-COMMIT-tilasta. AUTOCOMMIT-tila on SQL-Standardin vastainen tila, niin kuin Laiho ja Wendelius tutkimuksessaan toteavat: " SQL-standardin mukaan SQL-istunto toimii transaktionaalisessa moodissa siten, että jos transaktio ei ole käynnissä, niin ensimmäinen transaktiossa mahdollinen SQL-komento aloittaa uuden transaction implisiittisesti. " (Laiho & Wendelius 2015,4) Tällaisia SQL- komentostandardin mukaisia tietokantajärjestelmiä, ovat esimerkiksi DB2 ja Oracle. Niiden Client Server ratkaisut voivat kuitenkin käyttää toisinaan rajapinnoissaan AUTOCOMMIT-komentoa. Tällaiset AUTOCOMMIT-tilaa käyttävät tietokantajärjestelmät voivat aina käyttää tarvittaessa ROLLBACK-peruutuskomentoa. Näiden järjestelmien ei siis tarvitse myöskään erikseen määritellä, missä vaiheessa ne aloittavat transaktion. (Laiho & Wendelius 2015,4.)

Transaktioprotokollaan kuuluvat myös niin sanotut SAVEPOINT vaiheet. Savepoint vaiheiden avulla käyttäjä pystyy peruuttamaan tekemiään muutoksia johonkin tiettyyn pisteeseen asti, jonka käyttäjä on ennalta määritellyt. Näitä savepoint pisteitä voi olla useampia, ja niiden toiminta tai toimimattomuus riippuu käyttäjän järjestelmästä. (Laiho & Wendelius 2015,6.)

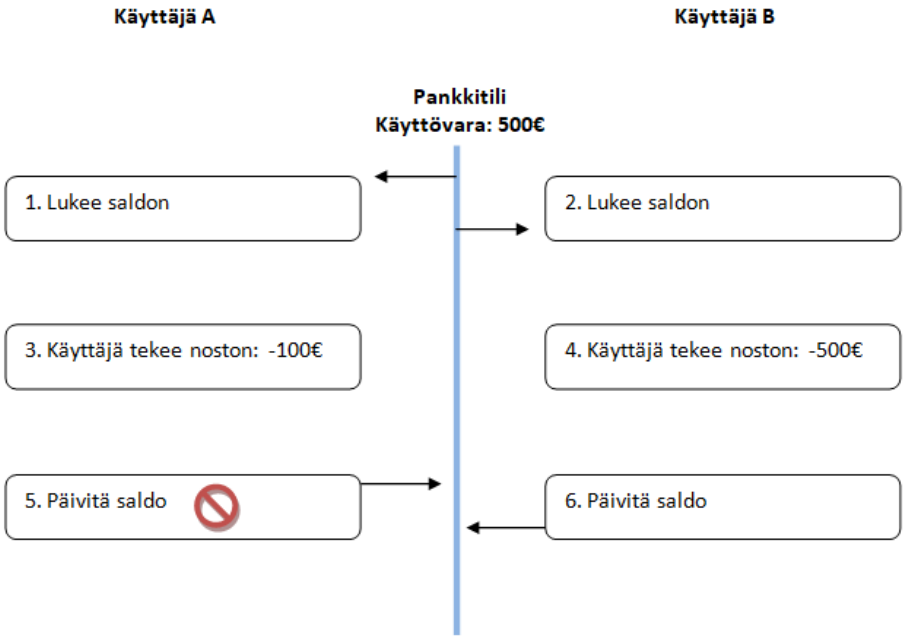
6.2 **Tunnistetut ongelmatilanteet**

Jos tietokannan tietosisältöön tulee epäjohdonmukaisuuksia, saattaa se aiheuttaa vakaviakin seurauksia. Tällaisia ovat esimerkiksi tilauksen, maksun tai toimituksen tekemättä jääminen tai aikaisemmin mainitsemieni hälytysjärjestelmien tekemien hälytysten häviäminen. Jotta tällaisilta tilanteilta vältyttäisiin, on samanaikaisten transaktioiden käsittelemiseen kehitelty erilaisia menetelmiä. (Laiho & Wendelius 2015,21.)

Ensinnäkin relaatiotietokannoissa kirjoitus suojataan aina kirjoituslukolla. Kirjoituslukot suojaavat esimerkiksi tilanteilta, joissa kaksi eri käyttäjää nostaa samanaikaisesti liikaa rahaa samalta pankkitililtä. Kirjoituslukoilla ehkäistään esimerkiksi seuraavanalaisia sokean päällekirjoituksen ongelmaksi kutsuttuja tilanteita.

Käyttäjä A ja B kirjautuvat samanaikaisesti tililleen tehdäkseen noston pankkitililtään. Kuten kuvasta 14 nähdään, käyttäjä A tekee ensin noston, josta ei tallennu tietoa kantaan.

Sen jälkeen käyttäjä B tekee noston, josta ei myöskään tule tietoa kantaan. Tällaisenaan ohjelmisto mahdollistaa käyttäjä B:n saldon päivityksen ilman, että transaktio huomaisi käyttäjän A:n tekemää päivitystä. Eli käyttäjän A tekemä päivitys kantaan estyy. Tällaista tarkastamatta jääneen kirjoitustapahtuman tekemää muutosta kutsutaan myös hukatun päivityksen ongelmaksi. (Laiho & Wendelius 2015,21.)



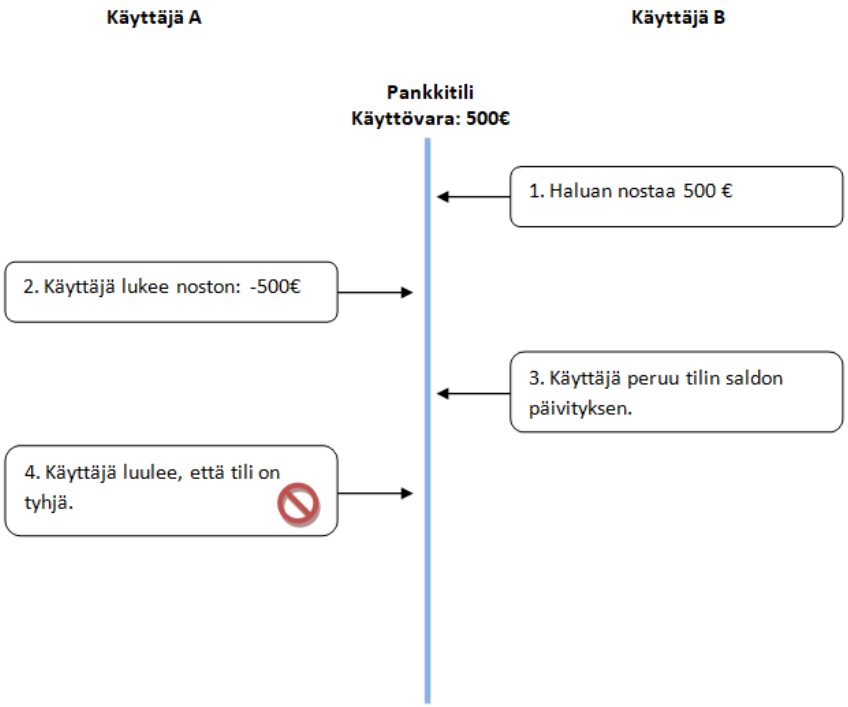
Kuva 14. Sokean päällekirjoituksen ongelma havainnollistettuna (mukaillen Laiho & Wendelius 2015)

Sokean päällekirjoituksen ongelmaa, jota myös hukatun päivityksen ongelmaksi kutsutaan, pystytään hallinnoimaan kahdella eri tavalla. Ensimmäinen keino on käyttää UPDATE -komentoa, jolloin uusi arvo lasketaan vanhan perusteella. Toinen tapa on hyödyntää tietokantaohjelmistokohtaista lukitusta, esimerkiksi taulu- tai rivikohtaista lukitusta. Taululukinnassa koko taulu lukittuu transaktiolle. Taululukintaa ei kuitenkaan suositella, sen eliminoidessa muiden käyttäjien mahdollisuudet kyseisen taulun muokkaamiseen. Näin ollen, se saattaa myös hidastaa järjestelmän toiminnallisuutta. Rivikohtaiset lukinnat taas ovat rivikohtaisia tarkoittaen, että vain tietty rivi sisältöineen lukitaan transaktiota varten. Nämä rivikohtaiset lukinnat ovat taulukohtaisia lukituksia suositellumpia, sillä ne eivät estä toisia käyttäjiä tekemästä omia muutoksiaan aivan niin radikaalisti, kuin taulukohtaiset lukinnat. (Laiho & Wendelius 2015,21-22.)

Sokean päällekirjoituksen ongelmaa ehkäistään myös tietokantaohjelmistokohtaisella deadlock-toiminnallisuudella. Deadlock toiminnallisuus perustuu deadlock detector -

toiminnallisuuteen, joka kaksi samanaikaista transaktiota havaitessaan ohjelmisto käyttää automaattista ROLLBACK-komentoa. Näin toinen komennoista peruuntuu automaattisesti. Tämäkin on osaltaan toimintaa hidastava tekijä. Peruuttaessaan toisen transaktioista ohjelmisto pakottaa käyttäjän lähettämään transaktionsa uudelleen. Jos siis käyttäjä peruuntumisen jälkeen, haluaa sen vielä tehdä. (Laiho & Wendelius 2015,21-22.)

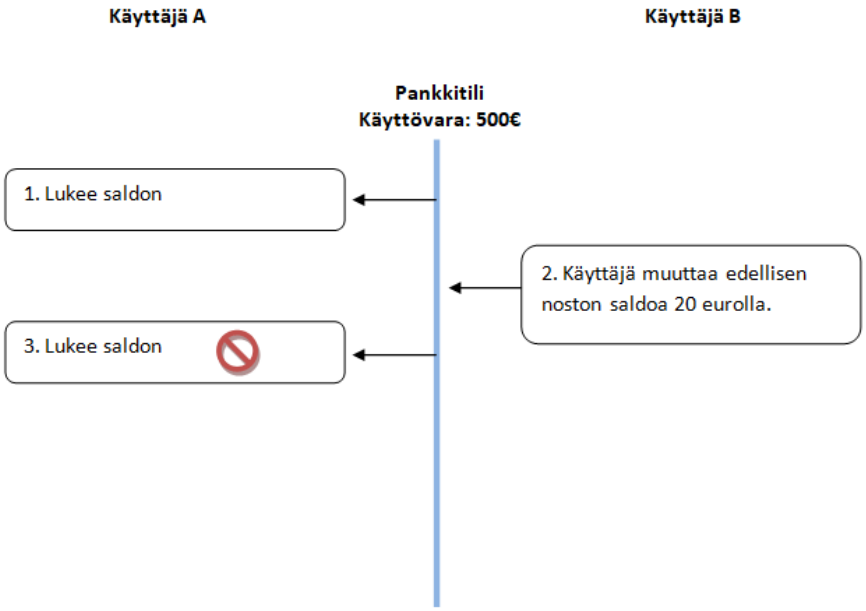
Sokean päällekirjoituksen ongelman lisäksi, on olemassa myös likaisen lukemisen ongelmaksi kutsuttu virhetilanne. Jos tietokantaohjelmiston transaktiot eivät ole eristäytyneitä, eli niitä ei suojata lukulukoin toisistaan, saattaa ilmaantua tilanteita, joissa käyttäjä A:n transaktio poimii jotain keskeneräistä tietoa käyttäjä B:n transaktiosta (katso kuva 15). Käyttäjä B saattaa esimerkiksi perua oman transaktionsa, eikä käyttäjän A etsimä tieto enää siinä tilanteessa ole validia. Siksi on haitallista jos käyttäjä A poimii tätä niin sanottua epävalidia dataa omaan transaktioonsa. (Laiho & Wendelius 2015, 22-23.)



Kuva 15. Likaisen lukemisen ongelma (mukaillen Laiho & Wendelius 2015)

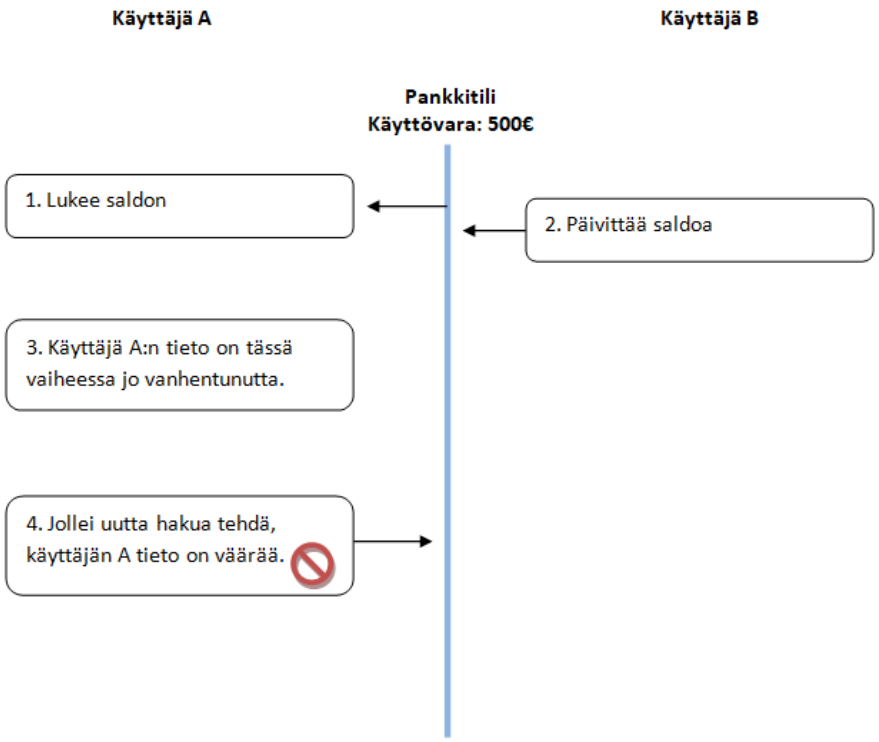
Palatkaamme aikaisempaan esimerkkiin. Jos käyttäjä B olisi vahvistanut transaktionsa, saattaa sekin johtaa omanlaiseensa, toistumattoman lukujoukon ongelmaan (katso kuva 16). Tällaisessa tilanteessa käyttäjä A poimii tietoa jostain tietystä taulusta. Käyttäjä B tekee tähän samaan kyseiseen tauluun muutoksen, esimerkiksi päivittämällä tai poistamalla dataa. Kun käyttäjä A suorittaa uuden kyselyn samoin hakuehdoin tähän samaan tauluun, saa hän vastaukseksi uuden arvon, joka poikkeaa vastauksellaan ensimmäisestä

kyselystä. Tämä johtaa tilanteisiin, joissa datan eheydestä ei voida olla varmoja, eivätkä arvot välttämättä täsmää niitä arvoja, joita käyttäjä A transaktiossaan olisi halunnut käyttää. (Laiho & Wendelius 2015,23.)



Kuva 16. Esimerkki toistumattoman lukujoukon ongelmasta (mukaillen Laiho & Wendelius 2015)

Aikaisemmasta kyselystä saatetaan saada väärä arvo myös tilanteessa, jossa käyttäjä A kohtaa kasvavan lukujoukon ongelman. Tällaiset mahdolliset virhetilanteet syntyvät (katso kuva 17) kun käyttäjä A tekee haun tietyin hakukriteerein tiettyyn tauluun. Sitten käyttäjä B lisää tähän tauluun jotain INSERT -komennolla. Tässä vaiheessa käyttäjän A tieto on jo vanhentunutta. Jos käyttäjä A ei tee uutta hakua, joka sisältäisi päivitykset, ei hänen ole mitenkään mahdollista saada oikeaa dataa. Näitä lisättyjä rivejä kutsutaan myös nimellä Phantom riveiksi. (Laiho & Wendelius 2015,24.)



Kuva 17. Kasvavan lukujoukon ongelma (mukaillen Laiho & Wendelius 2015)

6.3 **Samanaikaisuuden hallinta**

Samanaikaisuuden hallintaan on kehitelty useita eri menetelmiä, joilla eliminoidaan virhetilanteet. Samanaikaisuuden hallinnalla estetään tietokantaa käyttävien, tai muokkaavien henkilöiden, samanaikaisesta käsittelystä aiheutuvat ongelmatilanteet. Yksi tärkeimmistä, ja käyttäjän näkökulmasta selkeimmistä hallintamenetelmistä, on eristystaso. Eristystaso on tietokantaohjelmakohtainen, ja se voidaan erikseen määritellä transaktiolle tai koko istunnolle. Eristystasoa määrittelevät komennot ovat monesti tietokantakohtaisia, ja niissä nähdään jonkin verran vaihtelua. Eristystasoa ei transaktion alettua saa enää muuttaa, mutta jotkin tietokantaohjelmistot saattavat sisältää poikkeuksia tähän liittyen. Esimerkiksi joskus eristyneisyystaso voidaan määritellä lausekohtaisesti Hint-toiminnallisuuden avulla. Eristystasolla määritellään siis transaktion tai istunnon eristäytyneisyyttä muista järjestelmän käyttäjistä. Eristystason haittapuoliksi koetaan järjestelmän hidastuminen, kun esimerkiksi tietyt taulut lukitaan, vain yhden käyttäjän käytettäviksi. Eristystasosta huolimatta transaktiot suojataan aina lukoilla. (Laiho & Wendelius 2015,26.)

Eristystason ylläpitämiseen on kehitelty joitakin menetelmiä, joita nykyaikaiset relaatiotietokantaohjelmistot hyödyntävät. Luku- ja kirjoitustilanteista riippumattoman optimistisen samanaikaisuudenhallinnan tehtävänä on varmistaa, että kirjoitustilanteiden data tulee

muille käyttäjille näkyväksi vasta siinä vaiheessa, kun transaktio on COMMIT -komennolla vahvistettu. (Laiho & Wendelius 2015,28.)

Yksi tietokannan virhetilanteita eliminoiva toiminnallisuus, on jo aikaisemminkin mainitsemani, Deadlock detector -toiminallisuus. Tällä järjestelmän sisään rakennetulla toiminnallisuudella, on tärkeä rooli järjestelmän samanaikaisuuden hallinnassa. Deadlock detectorille pystytään määrittelemään järjestelmäkohtaisesti oma odotusaika. Eli kuinka kauan se odottaa käyttäjä A:n tekemän lukituksen avautumista, ennen kuin se automaattisesti peruuttaa käyttäjä B:n transaktion. Tämä odotusaika määritellään niin sanotulle deadlock detector säikeelle, jonka tehtävä on eliminoida toinen transaktioista. Nämä deadlock detector säikeet ovat järjestelmäkohtaisia, mutta ne sisältyvät suurimpaan osaan moderneja relaatiotietokantaohjelmistoja. (Laiho & Wendelius 2015,31.)

Vaikka samanaikaisuuden ongelmaa pystytään hallitsemaan rivi- ja taulukkokohtaisilla lukituksilla, voivat nämä rivi- ja taulukkokohtaiset lukitukset myös luoda omat ongelmansa. Mitä tapahtuu jos yhdelle taululle määritellään liian monta rivilukkoa, ja jos taululle itselleenkin määritellään oma lukko? Tällaisia tilanteita varten on kehitelty toiminnallisuus nimeltä aielukitus. Aielukituksen tehtävä on sulauttaa lukot yhdeksi lukoksi, jos samalle kohteelle määritellään liian monta lukkoa. Aielukitus käynnistyy kun lukitusten määrä ylittää lukituksille asetetun rajan. Näin saadaan sekä taululle, että riveille, määriteltyä sama lukko. (Laiho & Wendelius 2015,31.)

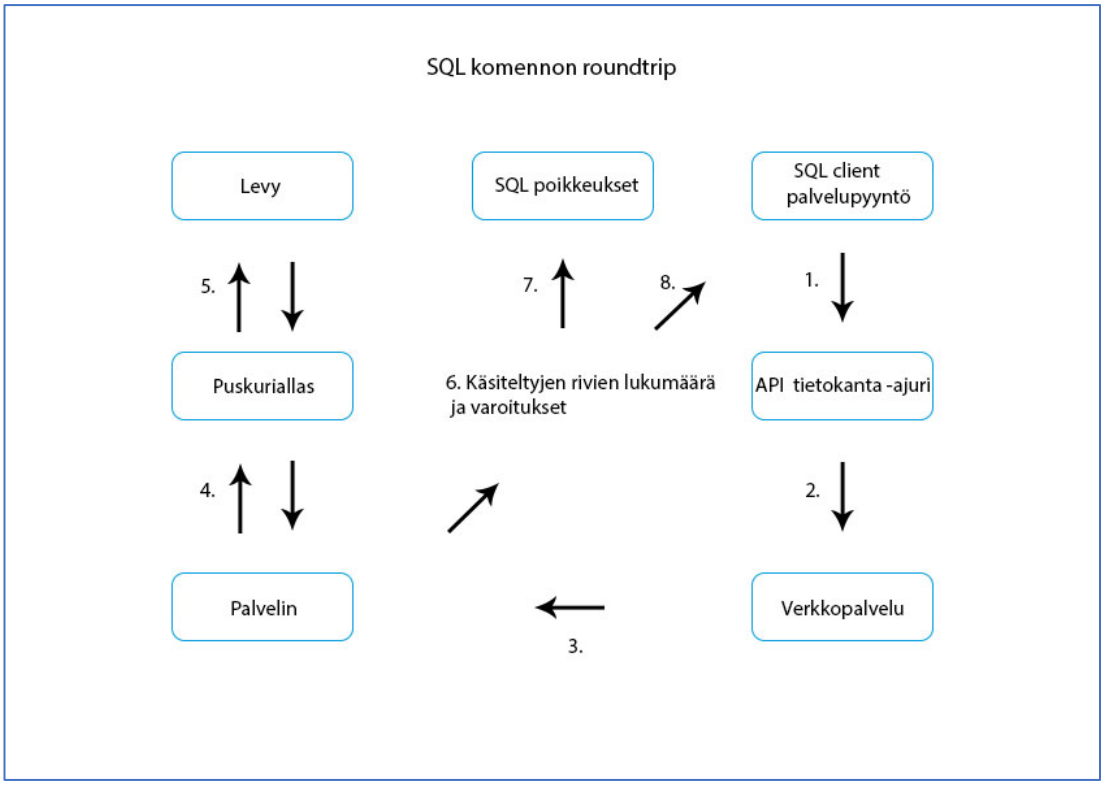
Relaatiotietokannoissa samanaikaisuutta voidaan hallita ja määritellä järjestelmäkohtaisesti. Järjestelmän haltija voi itse määritellä kullekin toiminnolle sen tarvitseman suojauksen. Suojauksia määriteltäessä tulisi kuitenkin aina huomioida, niiden vaikutus järjestelmän tehokkuuteen. Suojaukset vievät tilaa myös muistista. Suojauksia tulisi siis käyttää harkiten.(Laiho & Wendelius 2015,31.)

6.4 **Virhetilanteista toipuminen**

Virhetilanteista toipuminen mahdollistaa tietokannan eheyden säilymisen tilanteissa, joissa tietokannat eheyden ylläpitoa uhkaa jokin virhetilanne. Tällaisia virhetilanteita ovat esimerkiksi kirjoituslukkojen lukkojen purkaminen, ohjelmointivirhe tai järjestelmävirhe. Virhetilanteista toipuminen on osa relaatiotietokantojen järjestelmälogiikkaa. Varsinkin sovelluskehityksessä virhetilanteista toipuminen on tärkeää, koska muuten järjestelmään saattaisi tulla järjestelmän käytön estäviä virhetilanteita. (Laiho & Wendelius 2015,7-8.)

Järjestelmän virhetilanteissa, kuten sähkön katketessa, statistiikkatiedoista koostettu virhediagnostiikka on tietokantaohjelmiston tiedon eheyden ylläpitämisen kannalta ensiarvoisen tärkeää. Palvelimen kautta saatua diagnostiikkaa voidaan hyödyntää, kun halutaan esimerkiksi selvittää, mikä aiheutti SQL -komennon keskeytymisen. Jokainen komento ja transaktio tulisi aina käsitellä erikseen ohjelmistodiagnostiikassa. Sovelluksen ohjelmalogiikka vastaa komentojen peruuttamisen tarpeellisuudesta. Jos diagnostiikasta ei ole löydettävissä kaikkia komennon tekemiä tietoja, tulisi komento peruuttaa. Sovelluksen ohjelmalogiikka suorittaa silloin ROLLBACK eli peruutuskomennon. (Laiho & Wendelius 2015,7-8.)

Kuva 18 havainnollistaa SQL- komennon tekemän kierroksen tietokantajärjestelmässä. Vaiheesta 6 nähdään palvelimen mahdollisesti löytämien varoitusten palauttaminen SQL clientille. SQL clientillä viitataan tässä tietokantaohjelmistoon komennon tehneeseen käyttäjään. Jos palvelimelta saatu vastaus on virheetön, SQL transaktio on saatu onnistuneesti vietyä läpi.



Kuva 18. SQL komennon roudtrip

1. SQL client (käyttäjä) tekee palvelupyynnön.
2. Palvelupyyntö ajetaan API tietokanta-ajurin läpi.
3. Palvelupyyntö ajetaan verkkopalvelun läpi palvelimelle.

4. Palvelin jäsentää SQL lauseet ja analysoi ne systeemitaulujen tietoja vasten. Palvelin tarkistaa myös käsittelyoikeudet ja laatii komennolle suoritussuunnitelman tai etsii jo valmiin suoritussuunnitelman. Sen jälkeen palvelin käsittelee pyynnön. Keskusmuistissa säilytetään puskurialtaassa olevia jo ennestään haettuja rivejä.
5. Jos puskurialtaasta ei löytynyt haettavaa tietoa haetaan tieto levyltä. Levyltä hakeminen on hidasta, joten jo aikaisemmin haetut tiedot pyritään säilyttämään altaassa.
6. Tieto palautetaan palvelimelle, josta se lähetetään SQL clientille. Vastaus sisältää käsiteltyjen rivien lukumäärän ja mahdolliset varoitukset.
7. Varoituksia kutsutaan myös SQL-poikkeuksiksi, jotka SQL clientin ajuri vie sovellukselle käsiteltäviksi.
8. Jos tieto on atominen(eheä), se viedään SQL clientille analysoitavaksi.

6.5 **SQL Server ja transaktioiden hallinta**

SQL Server käyttää transaktioidensa hallinnassa, monia relaatiotietokantojen transaktioiden hallintaan luotuja työkaluja. Toiminnallisuudet ovat kuitenkin aina järjestelmäkohtaisia, esimerkiksi komentojen kirjoitusasu voi poiketa tavanomaisesta, mutta komennot muistuttavat kuitenkin pitkälti toisiaan. Tietokantaohjelmistoa valittaessa, täytyy käyttäjän olla tietoinen ohjelmistokohtaisista eroavaisuuksista transaktioiden hallinnassa.

SQL komentoihin lisätty, esimerkiksi MySQL:ssä toimiva GET DIAGNOSTICS -komento palauttaa käyttäjälle arvokasta tietoa ohjelmiston diagnostiikasta. Komennon avulla saadaan yksityiskohtaista tietoa komennon sisällön sekö komentoon sidonnaisten tekijöiden diagnostiikasta. SQL Serverillä ja Oraclessa tätä toiminnallisuutta ei kuitenkaan ole toteutettu. (Laiho & Wendelius 2015,7.) Diagnostiikan keräämisen helpottamiseksi SQL Serverille on kehitelty Transact-SQL -kieli. T-SQL kieli on erityisesti SQL Serverillä sovellettu, mutta myös muiden ohjelmistojen käytettävissä oleva tapahtumienhallintaa helpottava SQL lisäproseduuri. T-SQL kielen avulla SQL Serverillä pystytään käsittelemään diagnostiikkaa mm. seuraavin komennoin:
- @ @ERROR
- @ @ROWCOUNT

Error-komento ilmaisee havaittiinko komennon aikana virheitä ja rowcount-komento ilmaisee DML -komentojen käsittelemien rivien lukumäärän. (Laiho & Wendelius 2015,7-8.)

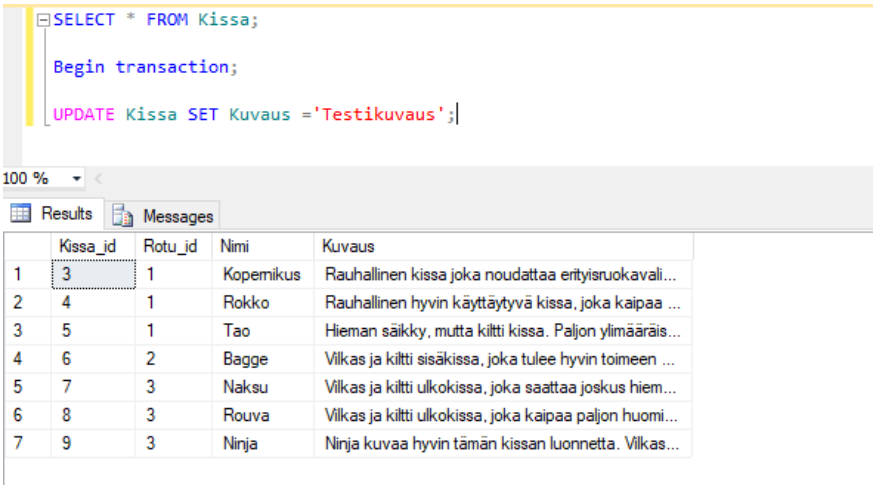
Proseduureja voidaan hyödyntää myös SQL Serverissä. Proseduuri sp_.sp_who kertoo käyttäjälle kuinka monta lukitusta kirjoitus- ja lukutapahtumille on määritelty kyseisessä järjestelmässä. Tällä komennolla voidaan myös selvittää, kuka lukitsee jotain tiettyä taulua tai riviä, joka lukitsemisellaan estää omaa transaktiota toimimasta. (McQuillan 2015.)

SQL Serverissä eristystason voi määritellä SET TRANSACTION ISOLATION LEVEL-asetuksella toiminnallisuuskohtaisesti. (Laiho & Wendelius 2015, 27.) SET TRANSACTION ISOLATION LEVEL -asetus sisältää READ COMMITTED ja READ UNCOMMITTED -komennot. READ COMMITTED-komennolla varmistetaan, etteivät komennot ota huomioon muita keskeneräisiä, ei vielä COMMIT -komennon kautta vahvistettuja transaktiotapahtumia. Tämä auttaa järjestelmää välttämään likaisen lukemisen ongelman. READ UNCOMMITTED -komennolla taas puolestaan voidaan määritellä järjestelmän ottavan huomioon muiden käyttäjien tekemät muutokset, joita ei ole vielä edes järjestelmään COMMIT-komennolla vahvistettu. (Microsoft 2018.)

6.5.1 SQL Server ja transaktiokomennot

SQL Server toimii oletuksena AUTOCOMMIT-tilassa, jolloin kaikki tehdyt muutokset tallentuvat järjestelmään automaattisesti. SQL Serverissä käyttäjä voi poistua AUTOCOMMIT-tilasta SET IMPLICIT_TRANSACTIONS ON -komennolla. Tällöin AUTOCOMMIT toiminto on pois päältä koko järjestelmästä. Jos käyttäjä haluaa palata takaisin AUTOCOMMIT-tilaan, voi hän palata siihen SET IMPLICIT_TRANSACTIONS OFF -komennolla. Yksittäisten istuntojen tai komentojen transaktiota pystytään hallitsemaan myös BEGIN-, ROLLBACK JA COMMIT TRANSACTION -komennoilla. (Laiho & Wendelius 2015,4.) SELECT @@TRANCOUNT -komento kertoo kuinka monta avointa transaktiota käyttäjällä on yhdessä ikkunassa auki. (McQuillan 2015.)

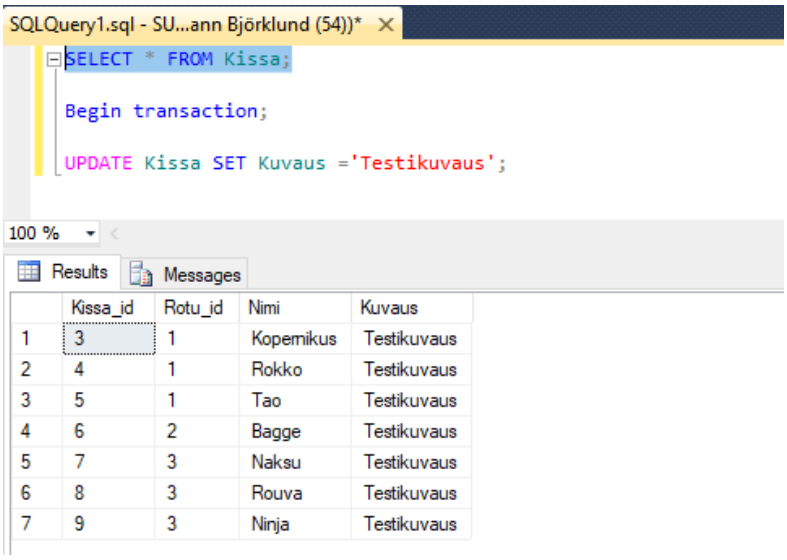
Virhetilanne ja sen korjaaminen havainnollistettuna SQL Serverin BEGIN-, ROLLBACK- ja COMMIT TRANSACTION -komennoilla.



	Kissa_id	Rotu_id	Nimi	Kuvaus
1	3	1	Kopernikus	Rauhallinen kissa joka noudattaa erityisruokavali...
2	4	1	Rokko	Rauhallinen hyvin käyttäytyvä kissa, joka kaipaa ...
3	5	1	Tao	Hieman säikky, mutta kiltti kissa. Paljon ylimääräis...
4	6	2	Bagge	Vilkas ja kiltti sisäkissa, joka tulee hyvin toimeen ...
5	7	3	Naksu	Vilkas ja kiltti ulkokissa, joka saattaa joskus hiem...
6	8	3	Rouva	Vilkas ja kiltti ulkokissa, joka kaipaa paljon huomi...
7	9	3	Ninja	Ninja kuvaa hyvin tämän kissan luonnetta. Vilkas...

Kuva 19. Transaktion aloitus.

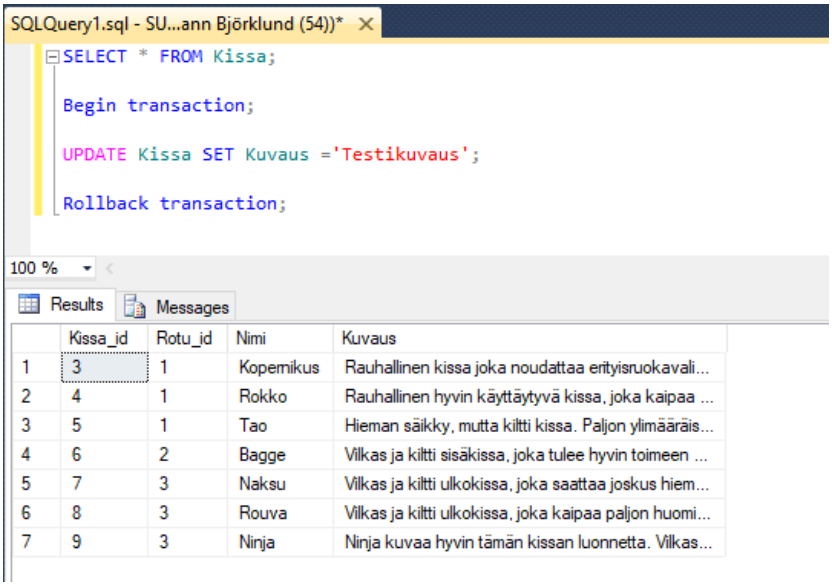
Kuvassa 19 tietokannasta on haettu kaikki Kissataulun tiedot. Kuvauksessa on selitetty yksityiskohtaisesti kunkin kissan luonteenpiirteistä. Kyseisessä tilanteessa tapahtuman-hallinta on käynnistetty BEGIN TRANSACTION -komennolla. UPDATE-komennon tarkoituksena on korvata vain ensimmäisen rivin (Kissa_id =3) kuvaus, uudella kuvauksella 'Testikuvaus'.



	Kissa_id	Rotu_id	Nimi	Kuvaus
1	3	1	Kopemikus	Testikuvaus
2	4	1	Rokko	Testikuvaus
3	5	1	Tao	Testikuvaus
4	6	2	Bagge	Testikuvaus
5	7	3	Naksu	Testikuvaus
6	8	3	Rouva	Testikuvaus
7	9	3	Ninja	Testikuvaus

Kuva 20. Virheellinen komento suoritettu.

Kuten kuvasta 20 nähdään, epämääräisen UPDATE-komennon takia kaikkien kissojen kuvaus päivittyykin sanalla "Testikuvaus". Tämä on tietokannan tiedon oikeaoppisuuden kannalta vaarallinen tilanne, kun kaikki aikaisemmin määritelty data häviääkin kyseisen taulun kaikilta kuvausriveiltä. Esimerkiksi asiakasprojektissa vastaavanlainen tilanne voisi olla hyvinkin ikävä, jos poistettaisiin vaikka kaikkien asiakkaiden osoitetiedot korvaamalla ne väärällä tiedolla.



	Kissa_id	Rotu_id	Nimi	Kuvaus
1	3	1	Kopemikus	Rauhallinen kissa joka noudattaa erityisruokavali...
2	4	1	Rokko	Rauhallinen hyvin käyttäytyvä kissa, joka kaipaa ...
3	5	1	Tao	Hieman säikky, mutta kiltti kissa. Paljon ylimääräis...
4	6	2	Bagge	Vilkas ja kiltti sisäkissa, joka tulee hyvin toimeen ...
5	7	3	Naksu	Vilkas ja kiltti ulkokissa, joka saattaa joskus hiem...
6	8	3	Rouva	Vilkas ja kiltti ulkokissa, joka kaipaa paljon huomi...
7	9	3	Ninja	Ninja kuvaa hyvin tämän kissan luonnetta. Vilkas...

Kuva 21. Virheellisen komennon peruuttaminen.

Kuten kuvasta 21 nähdään, ROLLBACK-komennolla pystytään peruuttamaan kaikki virheelliset muutokset, joita tehtiin kyseisen transaktion aikana. Jos käyttäjä olisi ennen ROLLBACK-komentoa käyttänyt COMMIT-komentoa, olisivat kyseiset muutokset tallentuneet tietokantaan.

6.5.2 Uusintayritysten toimintalogiikka

SQL Serverin ja monien muiden relaatiotietokantojen uudelleenkäynnistyslogiikan ohjelmointimalli (Retry Wrapper), määrittelee komentojen mahdolliset uudelleenkäynnistymiset. Joskus käsityksenä saattaa olla, että komennot käynnistyisivät uudestaan automaattisesti, mutta tämä toiminnallisuus pitää kuitenkin erikseen määritellä. Uudelleenkäynnistäminen tulee kyseeseen, kun komento tai komentosarjan läpivienti jostain syystä keskeytyy. Tällaisissa tilanteissa uudelleenyritys ei kuitenkaan aina ole mahdollista. Tämän toiminnallisuuden määrittely, on aina sovelluksen tai sovelluspalvelimen vastuulla. Uusintayrityksiä on suositeltavaa kuitenkin luoda vain rajallinen määrä ohjelmistokohtaisesti, sillä ne kuormittavat järjestelmää kuluttaen resursseja ja hidastaen vastausaikaa. (Laiho & Wendelius 2015,3.)

7 NoSQL-tietokantojen transaktiot

Viimeisen vuosikymmenen aikana NoSQL-tietokantaohjelmistot, ovat nostattaneet suosiotaan tehokkuutensa ja datankäsittelyhelppoutensa vuoksi. Transaktiot ovat kuitenkin aina olleet tärkeä osa tietokantaohjelmistojen hallintaa. Tämä pakotti ohjelmoijat yhdistämään relaatiotietokantaohjelmistot ja NoSQL-tietokantaohjelmistot keskenään. Näin varmistettiin tiedon talletuksen eheys, mutta myös tehokas järjestelmä. Datan käsittely ei silti ollut helppoa, sillä kaikki uutena järjestelmään tullut data piti käydä läpi ja muuntaa, jotta se voitiin tallettaa kantaan. Herkkä data jonka oli pysyttävä eheänä, talletettiin relaatiotietokantaan, ja vähemmän tärkeä data NoSQL-kantaan. Tätä herkkää dataa oli aina analysoitava manuaalisesti, joka teki datan käsittelystä hidasta ja virhealtista. Siksi tähän tarpeeseen vastattiin kehittämällä NoSQL transaktiot. (Choudbury 2018.)

NoSQL transaktioista ei ole löydettävissä vielä kovinkaan paljon akateemisia tutkimuksia. Amazonin DynamoDB oli ensimmäinen NoSQL-tietokantaohjelmisto, joka yhdisti ACID-periaatteen omaan toiminnallisuuteensa. Transaktiot sisälsivät kuitenkin joitakin rajoitteita, esimerkiksi datan kokoon ja sijaintiin liittyen. Viimeisen vuoden aikana transaktioiden hallinta on levinnyt yhtä useamman NoSQL-tietokantaohjelmiston käyttöön. Rajoitteet ovat ohjelmistokohtaisia, ja monet soveltavat ainoastaan ACID-periaatetta omien transaktioidensa hallinnassa. Jonkinasteista samanaikaisuuden hallintaa on myös sovellettu. Kun transaktioiden hallinta NoSQL-tietokantaohjelmistoissa kehittyy, päästään kustannustehokkaisiin ja käytännöllisempiin tietokantaratkaisuihin. (Choudbury 2018.)

7.1 BASE-periaate (Basic Available, Soft state, Eventual consistency)

NoSQL-tietokantaohjelmistoille on määritelty oma ACID- periaatetta (Kuva 13) korvaava, toimintalogiikkaa kuvaava periaate, jota kutsutaan nimellä BASE. Periaate ei ole kaikille tuttu ja alkoi myös menettää merkitystään ACID-periaatteen yleistymisen myötä. Käytännössä se tarkoittaa käytettävyyden, virheiden sallittavuuden ja eheyden painottamista NoSQL-järjestelmissä. Käytettävyydellä tarkoitetaan, että vaikka esimerkiksi käyttäjäkohtaisia muutoksia tehtäisiin järjestelmään, on järjestelmä silti vapaasti käytettävissä muille käyttäjille. Virheitä sallitaan jossain määrin niin, ettei se estä muita käyttäjiä suorittamasta omia toimintojaan järjestelmässä. Virhetilanteet pyritään kuitenkin aina ratkaisemaan. Eheydellä tarkoitetaan tässä kontekstissa sitä, että kun kaikki palvelulogiikkaa koskevat säädökset on viety päätökseensä, on järjestelmä silloin eheä. (McCreary & Kelly 2013.)

ACID-periaatteesta eroten, BASE-periaatteella halutaan painotuttaa eheyden sijaan järjestelmän käytettävyyttä säädöksistä huolimatta. BASE-periaate ei estä käyttäjää käyttä-

mästä järjestelmää, vaikka sen osa olisikin samanaikaisesti jollakulla toisella muokattavissa. Tämä pohjautuu siihen, että järjestelmän ei tarvitse hallita kirjoitus- ja lukulukkoja, jotka veisivät muistitilaa, tai jotka hidastaisivat järjestelmän toimintaa. BASE-periaatteen on kritisoitu kuitenkin olevan liiketoimintamielessä liian suppea käytettäväksi järjestelmissä. BASE-periaate on vain mukava lisä, mutta se ei varmista tiedon eheyden säilymistä sillä tasolla, jota siltä vaadittaisiin esimerkiksi asiakasprojektijärjestelmissä. (McCreary & Kelly 2013.)

7.2 MongoDB ja transaktiot 1.x, 2.x ja 3.x versioissa

Eelcon, Membreyn ja Hawkinsin teoksessa painotetaan sitä, miten transaktioiden puute tekee MongoDB:stä hyvin epäideaalin ympäristön, esimerkiksi pankkitilejä käsitteleväksi tietokantaohjelmistoksi. (Eelco ym. 2010, 4.) Aikaisemmat versiot MongoDB:stä eivät tukeneet minkäänlaista transaktioiden hallintaa, sillä tärkeintä oli nopeus, yksinkertaisuus ja skaalautuvuus. MongoDB ei myöskään tukenut tapahtumalokia, jonka avulla olisi voitu tarkkailla käyttäjien tekemiä muutoksia tietokantaan. Tietokannan sisältöä ja muokkauksia hallinnoitiin käyttäjille määritellyin käyttöoikeuksin. (Mäenpää 2013) Toiminnallisuus perustuu siihen, että tietylle dokumentaatiolle annettaisiin vain yksi muokkausoikeus. Eli samalle dokumentaatiolle ei voi olla muokkausoikeuksia kahdella eri palvelimella. Käytännössä tämä on kuitenkin mahdollista, mutta ei suositeltavaa. MongoDB eroaakin tässä monista muista NoSQL-tietokantaohjelmistoista. Yhden tai kahden pää (master)-tietokannan sijaan, sisällön muokkausoikeudet jaetaan osa-alueisiin, jotka voidaan jakaa jopa satojen palvelimien kesken. (Eelco ym. 2010,8.)

7.3 Samanaikaisuuden hallinta 3.0 versiossa

Samanaikaisuuden hallintaa sovellettiin ensimmäisen kerran MongoDB:n 3.0 versiossa. Samanaikaisuuden hallinta perustuu luku-ja kirjoituslukkoihin. Luku- ja kirjoituslukituksia voidaan määritellä globaalilla-, tietokanta- ja kokoelmatasolla (collection). Näillä luku-ja kirjoituslukoilla pystytään siis hallitsemaan usemman käyttäjän samanaikaista luku-ja kirjoitustoiminnallisuutta järjestelmässä. Lukulukoista käytetään myös merkintää (S) ja kirjoituslukoista merkintää (X). Näiden lukitusten lisäksi on olemassa Intent-lukituksia, joilla voidaan myös määritellä lukituksia globaalilla-, tietokanta- ja kokoelmatasolla. Intent-lukituksen asettaja hallinnoi esimerkiksi sitä, miten tietokannan kirjoitusoikeuksia voidaan käyttää. Näistä Intent-lukituksista käytetään lisämerkintää (I), joka voidaan lisätä luku (IS)- ja kirjoitus (IX) -määritelmille. Jos kokoelmalle esimerkiksi asetetaan X-lukko, on sekä tietokannalle, että globaalin tason vastaavalle lukolle asetettava IX-lukko. Yksi tietokanta voi olla sekä IS, että IX -tilassa samanaikaisesti. Jos kokoelma on X-lukko tilassa, ei tieto-

kanta voi olla muuta kuin IX-lukko tilassa. Jos kokoelma on S-lukko tilassa, ei tietokanta voi olla muuta kuin IS-lukko tilassa. (MongoDB 2019.)

Samanaikaiset luku- ja kirjoituspyynnöt käsitellään MongoDB:n järjestelmässä aina jono-järjestyksessä. Jotta toiminta olisi mahdollisimman tehokasta, voi järjestelmä esimerkiksi antaa useammalle S-lukkopyynnölle samanaikaisesti oikeudet. Kun S-lukk pyynnöt on käsitelty järjestelmä avaa mahdolliset X-lukkopyynnöt, vaikka jonoon olisikin S-lukkopyyntöjen käsittelyn aikana tullut uusia S-lukkopyyntöjä. Näin varmistetaan lukko-pyyntöjen eteneminen niin, ettei mikään pyyntö jää liian pitkäksi ajaksi odottamaan omaa vapautumistaan. (MongoDB 2019.)

7.4 **MongoDB 4.x ja transaktiot**

Nyt markkinoille on kuitenkin tullut uusi MongoDB 4.0 versio, jonka toiminnallisuuteen on lisätty samanaikaisuuden hallinnan lisäksi ACID-periaate (Kuva 13). Transaktioiden toiminnallisuus perustuu siis ACID- periaatteeseen, jota relaatiotietokantojen transaktioissa jo ennestään sovellettiin. Käytännössä tämä toiminnallisuuden muutos selitetään näin: Dokumenttipohjaisen MongoDB:n yksittäisiin dokumentteihin tehtävät komennot ovat aina atoomisia. Vaikka komennot liittyisivät dokumentteihin joihin on liitetty muita dokumentteja, pystytään haku suorittamaan silti niin, että se on atominen. Uusimmassa MongoDB:n versiossa transaktiota pystytään kuitenkin käyttämään niin, että transaktiot koskevat useampaa eri dokumenttia. (MongoDB 2019.)

Useampaa dokumenttia koskevaa transaktioita käytettäessä tulisi aluksi ottaa huomioon, että yksittäisten tietokantakaavioiden oikeaoppinen määrittely dokumenttikohtaisesti, on aina transaktiota kannattavampaa. Tämä johtuu siitä, että transaktiot vaikuttavat ohjelmiston tehokkuuteen. Siksi transaktioita tulisi käyttää harkitummin, kuin esimerkiksi relaatiotietokannoissa. (MongoDB 2019.)

Vaikka toiminnallisuus perustuu pitkälti samaan ajattelumalliin kuin relaatiotietokantojen transaktioissa, on MongoDB:n transaktiokäsittelyssä myös eroja. Useampaa kuin yhtä dokumenttia koskevat transaktiot ovat mahdollisia vain, jos transaktiolla ei lisätä niin sanottua uutta kokoelmaa (collection). Transaktiot ovat myös aina istuntokohtaisia. Tarkoittaen, että yhdellä istunnolla voi olla vain yksi avoin transaktio, ja vain yksi istunto voi kirjoittaa transaktiota. Jos istunto keskeytyy, transaktio peruuntuu. Transaktioita voidaan tehdä vain, jos käyttäjällä on luku- ja kirjoitusoikeudet järjestelmään. Jokaiselle transaktiokomennolle pitää myös määritellä kyseisen istunnon tunniste. (MongoDB 2019.)

7.4.1 MongoDB ja transaktiokomennot

Kuten relaatiotietokannoissa, myös MongoDB:ssä ACID-periaate toimii "kaikki tai ei mitään" ajatusmallin mukaisesti. Jos jokin vaihe transaktiossa päättyy virhetilanteeseen, koko transaktio peruutetaan. Transaktiot eivät myöskään tallennu kantaan ennen COMMIT-komentoa. Ja jos transaktio peruuntuu, kaikki transaktiossa olleet vaiheet hylätään. Transaktiokomennot ovat MongoDB:ssä seuraavanlaiset:

```
2
3
4  Session.startTransaction()
5
6  Session.commitTransaction()
7
8  Session.abortTransaction()
9
```

Kuva 22. MongoDB:n transaktio komennot.

Niinkuin kuvasta 22 nähdään, komennot muistuttavat siis jonkin verran relaatiotietokannoissa käytettyjä komentoja. Session.startTransaction() aloittaa transaktion. Session.abortTransaction() peruuttaa transaktion. Session.commitTransaction() vahvistaa transaktion. (MongoDB 2019.)

Transaktioiden käyttö havainnollistettuna MongoDB 4.0 versiossa.

```
// v4.0.0-rc0


// Vaihe 1
// 'Use test' -komento ottaa 'julian_tietokanta' nimisen tietokannan käyttöön,
//Insert lisää kokoelmille(collection) arvoja.
use julian_tietokanta;
db.auto.insert({"_id": 1, "merkki": "Honda", "malli": "Civic"});
db.auto.insert({"_id": 2, "merkki": "Audi", "malli": "A3"});


// Vaihe 2
// Aloitetaan sessio ja transaktio
var session1 = db.getMongo().startSession();
var session1AutoCollection = session1.getDatabase('julian_tietokanta').getCollection('auto');
session1.startTransaction({readConcern: {level: 'snapshot'}, writeConcern: {w: 'majority'}});


// Vaihe 3
//Lisätään yksi rivi transaktion sisällä
session1AutoCollection.insert({"_id": 3, "merkki": "Volvo", "malli": "Z3"});
(tässä kohtaa ilmoitus muutoksista)


//Vaihe 4
// Tarkistetaan, että äsken tehdyt muutokset eivät ole vielä tulleet voimaan dokumentin sisällä
db.auto.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}


// Vaihe 5
// Transaktiota kutsuttaessa käyttäjä näkee transaktion sisältämät muutokset
session1AutoCollection.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}
// {"_id": 3, "merkki": "Volvo", "malli": "Z3"}


// Vaihe 6
// Hyväksytetään muutos ja lopetetaan sessio
// commit and end the session
session1.commitTransaction()
session1.endSession()


// Vaihe 7
// Tutustutaan vielä transaktion tekemiin muutoksiin
db.auto.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}
// {"_id": 3, "merkki": "Volvo", "malli": "Z3"}

```

Kuva 23. Transaktion käynnistys ja vahvistaminen.

Kuvasta 23 nähdään, kuinka käyttäjä on ensimmäisessä vaiheessa luonut tietokannan, joka sisältää autoja koskevan dokumentin. Vaiheessa kaksi käyttäjä käynnistää transaktion, ja luo kyseisellä transaktiolle session. Sen jälkeen käyttäjä päivittää dokumenttia lisäämällä yhden uuden kokoelman (collectionin) vaiheessa 3. Sen jälkeen käyttäjä tarkistaa, etteivät transaktioon sisältyvät muutokset ole tulleet vielä voimaan vaiheessa 4. Kun transaktiota kutsutaan vaiheessa 5, pystytään näkemään mitä muutoksia transaktio pitää sisällään. Vaiheessa 6 käyttäjä on valmis vahvistamaan tekemänsä muutokset commit transaction -komennolla. Sen jälkeen käyttäjä lopettaa aloittamansa session. Vaiheessa 7 nähdään vielä, kuinka transaktion sisältämä muutos on astunut voimaan käyttäjän vahvistettua muutokset.

```
// v4.0.0-rc0

// Vaihe 1
// 'Use test' -komento ottaa 'julian_tietokanta' nimisen tietokannan käyttöön,
//Insert lisää kokoelmille(collection) arvoja.
use julian_tietokanta;
db.auto.insert({"_id": 1, "merkki": "Honda", "malli": "Civic"});
db.auto.insert({"_id": 2, "merkki": "Audi", "malli": "A3"});


// Vaihe 2
// Aloitetaan sessio ja transaktio
var session1 = db.getMongo().startSession();
var session1AutoCollection = session1.getDatabase('julian_tietokanta').getCollection('auto');
session1.startTransaction({readConcern: {level: 'snapshot'}, writeConcern: {w: 'majority'}});


// Vaihe 3
//Lisätään yksi rivi transaktion sisällä
session1AutoCollection.insert({"_id": 3, "merkki": "Volvo", "malli": "Z3"});
(tässä kohtaa ilmoitus muutoksista)


//Vaihe 4
// Tarkistetaan, että äsken tehdyt muutokset eivät ole vielä tulleet voimaan dokumentin sisällä
db.auto.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}


// Vaihe 5
// Transaktiota kutsuttaessa käyttäjä näkee transaktion sisältämät muutokset
session1AutoCollection.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}
// {"_id": 3, "merkki": "Volvo", "malli": "Z3"}


// Vaihe 6
// Peruutetaan muutos ja lopetetaan sessio
// commit and end the session
session1.abortTransaction()
session1.endSession()


// Vaihe 7
// Varmistetaan vielä että transaktio ei tehnyt mitään muutoksia dokumenttiin.
db.auto.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}


```

Kuva 24. Transaktion peruuttaminen.

Kuvasta 24 nähdään, kuinka vaiheessa 6 käyttäjä vahvistamisen sijaan, peruuttikin tekemänsä muutokset abort transaction -komennolla. Näin ollen käyttäjän tekemiä muutoksia ei näy myöskään vaiheessa 7, jolla tarkistetaan, etteivät transaktion aikana tehnyt muutokset tallentuneet kantaan.

```
// Vaihe 1
// 'Use test' -komento ottaa 'julian_tietokanta' nimisen tietokannan käyttöön,
//Insert lisää kokoelmille(collection) arvoja.
use julian_tietokanta;
db.auto.insert({"_id": 1, "merkki": "Honda", "malli": "Civic"});
db.auto.insert({"_id": 2, "merkki": "Audi", "malli": "A3"});

// Vaihe 2
// Aloitetaan session1 ja transaktio joka käyttää 'julian_tietokanta' -tietokantaa
var session1 = db.getMongo().startSession();
var session1AutoCollection = session1.getDatabase('julian_tietokanta').getCollection('auto');
session1.startTransaction({readConcern: {level: 'snapshot'}, writeConcern: {w: 'majority'}});

// Vaihe 3
// Luodaan sessio2 ja transaktio joka käyttää 'julian_tietokanta' -tietokantaa
var session2 = db.getMongo().startSession();
var session2AutoCollection = session2.getDatabase('julian_tietokanta').getCollection('auto');
session2.startTransaction({readConcern: {level: 'snapshot'}, writeConcern: {w: 'majority'}});

//Vaihe 4
// Poistetaan session1:ltä rivi, jolla id = 2
session1AutoCollection.deleteOne({"_id": 2});
// { "acknowledged" : true, "deletedCount" : 1 }

// Vaihe 5
// Dokumentaation tietoja kutsuttaessa poistettu rivi ei ole vielä näkyvissä
db.auto.find()
// {"_id": 1, "merkki": "Honda", "malli": "Civic"}
// {"_id": 2, "merkki": "Audi", "malli": "A3"}

// Vaihe 6
// Transaktio tapahtumaa tarkasteltaessa käyttäjä näkee transaktion tekemät muutokset
// Ainoastaan rivi id=1 on näkyvissä
session1AutoCollection.find()
// {"_id": 1, "malli": "Honda", "merkki": "Civic"}

// Vaihe 7
// Poistetaan sessio2:lta rivi, jolla id = 2
session1AutoCollection.deleteOne({"_id": 2});
// { "acknowledged" : true, "deletedCount" : 1 }

// Vaihe 8
// Error viesti virheellisestä transaktiosta ja sen peruutuksesta(sessio 2) on nähtävissä

// Vaihe 9
// Peruutetaan muutokset ja lopetetaan sessiot 1 ja 2
// commit and end the session
session1.abortTransaction()
session1.endSession()
session2.abortTransaction()
session2.endSession()

// Vaihe 10
// Varmistetaan vielä että transaktio ei tehnyt mitään muutoksia dokumenttiin
db.auto.find()
// {"_id": 1, "malli": "Honda", "merkki": "Civic"}
// {"_id": 2, "malli": "Audi", "merkki": "A3"}
```

Kuva 25. Virhetilanne havainnollistettuna.

Kuvassa 25 havainnollistetaan virhetilanne, joka johtaa transaktion peruuntumiseen. Koska transaktiot ovat istuntokohtaisia, on kyseisessä esimerkissä käynnistetty kaksi istuntoa. Kuvasta nähdään, kuinka käyttäjä on luonut kaksi istuntoa vaiheissa 2 ja 3. Vaiheessa 4 käyttäjä poistaa ensimmäiseltä istunnolta(session1) rivin, jonka id on 2. Vaiheesta 6 nähdään, kuinka käyttäjä kutsuu dokumenttia ja poistotapahtuma ei ole vielä rekisteröitynyt kantaan. Vaiheessa 7 käyttäjä tarkastelee ensimmäisen transaktion istunnon tekemiä muutoksia. Seuraavaksi käyttäjä yrittää poistaa samaa riviä (id=2) toisella transaktio istunnolla (session2), siinä kuitenkaan onnistumatta. Järjestelmä peruuttaa tässä vaiheessa toisen istunnon tekemät muutokset, sillä istunto1 teki jo muutoksia kyseiselle id:lle en-

simmäisen istunnon aikana. Lopuksi käyttäjä vielä peruuttaa kaiken ja lopettaa istunnot. Tämän jälkeen hän varmistaa vielä, ettei virhetilanne aiheuttanut häiriöitä kantaan tarkistamalla kokoelmien(taulujen) tilan vaiheessa 11.

7.4.2 Uusintayrityksen toimintalogiikka

Jos transaktio keskeytyy ennen COMMIT- komennon lähetystä, on transaktion uudelleen lähettäminen joissakin tapauksissa mahdollista. RetryWrites on MondoDB -ohjelmistoon kehitelty toiminnallisuus, jonka tarkoituksena on käynnistää ajurien uudelleenlähetysyritys. Komento suoritetaan yhden kerran, sen jälkeen, kun se on ensimmäisellä kerralla epäonnistunut. Toiminto voidaan kytkeä päälle, jos komentosarjalle on määritelty retryWrites arvolla true. Transaktioiden kohdalla, sille on kuitenkin joitain rajoitteita. Yksittäisille transaktiokomennoille ei voida määritellä tätä toiminnallisuutta. Toiminnallisuus voidaan kuitenkin määritellä COMMIT-komennolle, jolloin järjestelmä automaattisesti uudelleen lähettää komennon, sen epäonnistuessa. (MongoDB 2019.)

8 SQL Serverin ja MongoDB:n transaktioiden hallinnan tärkeimmät eroavaisuudet

Alla olevassa taulukossa (katso kuva 26) esitetään SQL Serverin ja MongoDB:n transaktioiden hallintakeinojen tärkeimmät eroavaisuudet. SQL Serverillä transaktioiden hallinta on kehittynyt muiden relaatiotietokantaohjelmistojen tapaan hyvin pitkälle. Ohjelmistossa sovelletaan useita eri samanaikaisuudenhallintakeinoja. SQL Serverin transaktioiden hallinta eroaa muista relaatiotietokantaohjelmistoista ohjelmistokohtaisin proseduurein ja komennoin. SQL Serverissä diagnostiikan saamiseksi on kehitelty Transact-SQL kieli.

MongoDB:ssä toiminnallisuus on tunnetusti uusi ja huomattavasti rajoittuneempi. MongoDB:ssä ei luku-ja kirjoituslukkojen sekä ACID-periaatteen lisäksi ole sovellettu muita transaktioidenhallintakeinoja. Tähän saattaa tietysti tulla muutoksia tulevaisuudessa, mutta tällä hetkellä transaktioiden hallintakeinoja on hyvin suppeasti MongoDB:ssä. Täytyy kuitenkin ottaa huomioon, ettei monissa NoSQL-tietokantaohjelmistoissa ole ollenkaan sovellettu transaktioiden hallintaa. Siksi MongoDB:n transaktioiden hallinta on muihin NoSQL-tietokantaohjelmistoihin verrattuna hyvinkin edistynyttä.

SQL Server	MongoDB
ACID	ACID
Luku- ja kirjoituslukot	Luku- ja kirjoituslukot
Deadlock detector	X
Aielukitus	X
Transact-SQL ja diagnostiikka	X -Ei transaktiokohtaista diagnostiikkaa
Proseduurit	X
Eristystason määrittely SET TRANSACTIONS ISOLATION LEVEL-asetuksella.	X

Kuva 26. SQL Serverin ja MongoDB:n transaktioiden hallinnan vertailu.

Jotta eroja transaktioiden hallinnassa SQL Serverin ja MongoDB:n välillä ymmärrettäisiin vielä paremmin, kuvasta 27 voidaan havaita miten virhetilanteita ja niiden hallintakeinoja on otettu huomioon näiden järjestelmien välillä.

SQL Server	MongoDB
Sokean päällekirjoituksen ongelma	Ei tiedossa
Likaisen lukemisen ongelma	Ei tiedossa
Havaitsemattomien rivien ongelma	Ei tiedossa
Toistumattoman lukujoukon ongelma	Ei tiedossa
Hukatun päivityksen ongelma	Ei tiedossa

Kuva 27. Tunnistettujen ongelmatilanteiden käsittely.

Kuvista 26 ja 27 voitaneen siis todeta, ettei MongoDB:ssä riskitilanteiden hallintaan ole varauduttu esimerkiksi eristystason kaltaisin apuvälinein. MongoDB:ssä riskitilanteita, ei ole analysoitu tai huomioitu. Itse en ainakaan onnistunut löytämään tästä materiaalia. Siksi voidaan siis jälleen todeta, että SQL Serverin transaktioiden hallinta on huomattavasti kehittyneempää ja luotettavampaa kuin MongoDB:ssä.

9 Pohdinnat

Tietokantoja, tietokantamalleja ja tietokantaohjelmistoja on monia erilaisia. Näillä kaikilla on omat tarpeensa ja omat ominaisuuteensa. Tietokantaohjelmistoa valikoitaessa käyttäjän tulee olla tietoinen oman sovelluksensa vaatimista palveluominaisuuksista. Sovelluksen suunnittelussa, käyttäjän tulee olla tietoinen mahdollisista tietokantaohjelmiston puutteista. Sovelluskehityksen näkökulmasta, sekä relaatiotietokantaohjelmistoilla, että NoSQL-tietokantaohjelmistoilla on omat hyvät ja huonot puolensa. Niitä vertailemalla käyttäjä tietää, kumpaa hänen tulisi omassa sovelluksentoteutuksessaan käyttää. Vai tulisiko hänen kenties soveltaa niitä yhteiskäytössä.

Relaatiotietokantojen transaktioiden hallinta on huipussaan. Relaatiotietokantaohjelmistot ovat pitkään pitäneet pintansa tietokantaohjelmistojen suosituimpina tietokantaohjelmistoina. Niiden toiminnallisuus on hyvin pitkään pysynyt, pieniä muutoksia lukuun ottamatta samanlaisena. Koska relaatiotietokantaohjelmistojen joustamattomuus mielletään relaatiotietokantojen suurimmaksi puutteeksi, asettaa se myös tietynlaisia rajoitteita ohjelmistojen kehittymiselle. Vaikka tietokanta pystyttäisiin hajauttamaan kahdelle eri palvelimelle, saattaa se kuitenkin olla esteenä tiettyjen ohjelmistojen resurssien lisäämiselle. Hajautettuinakaan relaatiotietokantaohjelmistot, eivät välttämättä pysty saavuttamaan samaa tehokkuutta kuin skeemattomat (tietokantakaaviottomat) NoSQL-tietokantaohjelmistot. NoSQL-ohjelmistojen kyky skaalautua horisontaalisesti, luo niille mahdollisuuden laajentua relaatiotietokantaohjelmistoja joustavammin.

NoSQL-tietokantaohjelmistot ovat alkaneet soveltamaan transaktioiden hallintaa omissa järjestelmissään yhä enemmän. Vaikka transaktioiden hallinta ei vielä todennäköisesti yllä pitkään aikaan samalle tasolle, kuin relaatiotietokantojen transaktioiden hallinta, on NoSQL-tietokantaohjelmistoilla paljon potentiaalia edetä sinne. Järjestelmäkohtaisia innovaatioita voidaan tulevaisuudessa todennäköisesti yhä enemmissä määrin soveltaa NoSQL-tietokantaohjelmistojen transaktioidenhallinnan kehittymiseen.

Transaktiot heikentävät ainakin tänä päivänä NoSQL-tietokantajärjestelmän tehokkuutta vielä niin paljon, ettei niitä sellaisenaan voida soveltaa kaikkiin ohjelmistoihin tai niiden soveltaminen ei ole järjestelmän kannalta kannattavaa. NoSQL-tietokantaohjelmistot kehiteltiin alun perin juuri tehokkuuden vuoksi ja jos niistä tehdään liian samanlaisia relaatiotietokantaohjelmistojen kanssa, hävittävät ne merkityksensä. Tällä hetkellä toiminnallisuudeltaan paras ratkaisu transaktioiden hallintaan ja tehokkaaseen tietokantaohjelmistotoimintaan on relaatiotietokantojen ja NoSQL-tietokantojen yhteiskäyttö.

9.1 Tutkimuksen hyödyntäminen

Tämän tutkimuksen luettuaan lukijalla tulisi olla ymmärrys relaatiotietokannoista, NoSQL-tietokannoista sekä transaktioiden hallinnasta SQL Serverin ja MongoDB:n välillä. Jotta transaktioita ymmärrettäisiin, on lukijan ymmärrettävä relaatiotietokantoja ja NoSQL-tietokantoja myös yleistasolla. Sovelluksen palvelutarpeen näkökulmasta näiden kahden eri tietokantamuodon ymmärtäminen ja erottaminen on myös tärkeää.

Tutkimuksen luettuaan lukija ymmärtää, että relaatiotietokantojen pitkä historia transaktioiden hallinnassa tekee niistä ensisijaisen välineen transaktioiden hallintaan. Lukija ymmärtää myös eron relaatiotietokantojen ja NoSQL-tietokantojen niin sanottujen hyvien ja huonojen puolien välillä. Tämän tiedon avulla lukijan on mahdollista valita käyttöönsä se tietokantaohjelmisto, joka vastaa käytttäjän tarvetta.

9.2 Opinnäytetyöstä oppiminen

Työn suunnitteluun olisi voitu käyttää hieman enemmän aikaa. Isoa kokonaisuutta oli hankala käsitellä ilman omaa kosketuspintaa aiheeseen. Tutkimuksen tuloksia voidaan mielestäni hyödyntää tämänhetkisessä transaktioista käsittelevissä materiaaleissa. Jos aihe on vieras saa tutkimuksesta hyvän käsityksen relaatiotietokannoista, NoSQL-tietokannoista, tietokantaohjelmistoista sekä transaktioista.

Transaktioiden ymmärtäminen on varsinkin tietokantojen kanssa työskentelevien ihmisten kannalta erityisen tärkeää. Jos transaktioita ja niiden hallintaa ei ymmärretä, voi tietokantaa käyttävät henkilöt aiheuttaa vakaviakin vahinkoja esimerkiksi hävittäessään tietokannasta tärkeää tietoa.

Tutkimuksen laatiminen oli sekä haastavaa, opettavaista että ajoittain helppoakin. Tämän aiheen myötä minulle on kehittynyt tarkka kuva siitä mitä ovat transaktiot ja mikä niiden tulevaisuudenkuva on. Tutkimuksessa pääsin tutustumaan myös NoSQL-tietokantojen maailmaan, josta minulla ei ollut aikaisemmin minkäänlaista käsitystä.

Jälkeenpäin ajateltuna tutkimus olisi pitänyt aloittaa aikaisemmin. Tiesin jo alusta alkaen, että halusin vertailla SQL- ja NoSQL-tietokantoja toisiinsa. Transaktio-näkökulman löytämiseen meni kuitenkin aikaa. Transaktioista kuullessani tiesin heti, että tämä on se näkökulma jonka haluan työhöni sisällyttää.

Tieteellisen tutkimuksen laatiminen oli opettavaista ja tehosti ajattelutapaani, jota tulen tarvitsemaan tulevissa työtehtävissäni kipeästi. Huomasin myös kuinka haastavaa oli laatia pitkä raportti uudesta aiheesta asiantuntevaa kirjoitusasua käyttäen.

Tutkimuksen toteutuksen kannalta haastavimpia tekijöitä oli NoSQL-tietokantaohjelmistoihin liittyvien transaktiomateriaalien puuttuminen. Aihe on niin uusi tietotekniikan maailmassa, että materiaalia ei ole ehtinyt kerääntyä esimerkiksi akateemisen kirjallisuuden osalta. Suurin osa NoSQL-tietokantoja koskevasta kirjallisuudesta oli siis transaktioiden osalta vanhentunutta.

Lähteet

Brade, C. 2019. Tietokantakurssi. Tietokannat ja tiedonhallinta SWD4TA003-3001. Haaga-Helia. Helsinki.

Choudhury, S. 2018. Why are NoSQL Databases Becoming Transactional? Luettavissa: https://blog.yugabyte.com/nosql-databases-becoming-transactional-mongodb-dynamodb-faunadb-cosmosdb/. Luettu: 13.4.2019.

Connolly, T. & Begg, C. 2015. Database Systems. A Practical Approach to Design, Implementation, and Management. Pearson. Edinburg.

Dataversity. 2018. A Brief History of Non-Relational Databases. Luettavissa: https://www.dataversity.net/a-brief-history-of-non-relational-databases/. Luettu: 24.9.2019.

Foote, K. 2016. A Review of Different Database Types: Relational versus Non-Relational. Luettavissa: https://www.dataversity.net/review-pros-cons-different-databases-relational-versus-non-relational/. Luettu: 24.9.2019.

DB-Engines. 2019. DB-Engines ranking. Luettavissa: https://db-engines.com/en/ranking. Luettu: 29.4.2019.

Eelco, P., Membrey, P. & Hawkins, T. 2010. The Definitive Guide to MongoDB. The NoSQL Database for Cloud and Desktop Computing. Apress. New York.

Hovi, A. 2013. SQL- Opas. Hansaprint. Vantaa.

Hovi, A. 2015. MongoDB haastaa relaatiotietokantoja. Luettavissa: https://www.arihovi.com/mongodb-haastaa-relaatiokantoja/#. Luettu: 9.4.2019

Laiho, M. & Wendelius, M. 2014. SQL-transaktioiden käytännön teoriaa ja harjoituksia. DBTech VET.

Mäenpää, M. 2013. NoSQL-tietokantojen tietoturva. Tampereen teknillinen yliopisto. Tampere. Luettavissa: https://wiki.tut.fi/Tietoturva/Tutkielmat/NoSQLSecurity. Luettu: 11.4.2019.

McCreary, D. & Kelly, A. 2013. Making Sense of NoSQL. A guide for managers and rest of us. Manning Publications Co. New York.

McQuillan, M. 2015. Introducing SQL Server. Apress. New York.

Microsoft. 2018. Set transaction isolation level (Transact -SQL). Luettavissa: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-2017. Luettu: 13.4.2019

MongoDB. 2019. MongoDB Manual. Transactions. Luettavissa: https://docs.mongodb.com/master/core/transactions/?_ga=2.252793097.167158179.1554897757-1597815606.1544545522&_gac=1.188266970.1554732202.Cj0KCQjw4qvlBRDiARIsAHme6oscsQoEliBHWx0Rtgp5D63B6wKDMqYw_AVPy3RyXGLLIF7TkVE7QEEaAmwqEALw_wcB. Luettu: 10.4.2019.

MongoDB. 2019. FAQ Concurrency. Luettavissa: https://docs.mongodb.com/manual/faq/concurrency/. Luettu:12.5.2019.

Konkka, P. 2016. SQL vai NoSQL, siinä pulma? Luettavissa: https://petrikonkka.com/fi/pilvipalvelujen-tietokannat-nosql-mongodb/. Luettu: 14.4.2019

Pinal, D. 2016. Understanding Database Scalability – Vertical and Horizontal. SQL Authority.com. Luettavissa: https://blog.sqlauthority.com/2016/04/29/understanding-database-scalability-vertical-horizontal/. Luettu: 5.5.2019.

Rys, M. 2011. Scalable SQL. Communications of the ACM. 2011.

Vidhya, V., Jeyaram, G. & Ishwarya, K. R. 2016. Database Management Systems. Alpha Science International LTD. U.K.

Virkki, O. 25.4.2019. Opettaja. Haaga-Helia ammattikorkeakoulu. Haastattelu. Helsinki.

Virkki, O. 13.5.2019. Opettaja. Haaga-Helia ammattikorkeakoulu. Tiedonanto. Helsinki.