Juho Valkonen

ORM-kehyksen vaikutus suorituskykyyn .NET-sovelluksessa

Tietotekniikan pro gradu -tutkielma

14. maaliskuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Juho Valkonen

Yhteystiedot: `juho.a.valkonen@student.jyu.fi`

Ohjaaja: Tommi Mikkonen

Työn nimi: ORM-kehyksen vaikutus suorituskykyyn .NET-sovelluksessa

Title in English: Performance effects of using ORM in .NET Application

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 61+89

Tiivistelmä: ORM-kehyksillä pyritään helpottamaan relaatiotietokantojen sekä olio-ohjelmoinnilla toteutettujen sovellusten yhteen sovittamista. ORM-kehykset helpottavat yleisesti sovelluskehitysprosessia, mutta samalla ne lisäävät sovellukseen kerroksen, joka saattaa tietyissä tilanteissa hidastaa sovelluksen toimintaa.

Tutkielmassa toteutettiin tapaustutkimus, jossa vertailtiin eri tapausten toteutusta ORM-kehystä (Entity Framework Core) hyödyntäen sekä ilman ORM-kehystä toteutettuna. Tapaukset suoritettiin eri tietokanta-alustoja vasten ja niiden suoritusajat mitattiin.

Mittaustulosten perusteella EF Core hidastaa tapausten suorittamista noin 100–200 prosenttia riippuen tapauksen luonteesta, tietokanta-alustasta sekä tietokannan taulujen rivimääristä. Ääritapauksissa suhteellinen hidastuminen oli yli 1000 %. Osassa tapauksista EF Coren hidastavan vaikutuksen suhteellinen osuus vaihteli tietokannan rivimääristä riippuen.

Avainsanat: .NET, ADO.NET, Entity Framework Core, Suorituskykymittaus, Tietokannat, ORM, ORM-kehys

Abstract: ORM-frameworks are tools that help software developers to connect between relational databases and object-oriented programming. In general, ORM-frameworks provide functionalities that simplify the process of developing software. However, they also add an abstraction layer to a software, which in some cases may slow down its performance.

This study was conducted as a case study. Multiple cases were implemented with an ORM-framework (Entity Framework Core), and without one. Those implementations were benchmarked against several different database management systems.

According to the results of the benchmarking process, EF Core slows down the performance of the selected cases by about 100–200 percents on average. In the extreme cases, EF Core slows down the performance for about 1 000 percents. The effect depends on the case, selected database, and the number of rows in the database. In some of the cases, the relative effect of EF Core slowing down the case, depends on the amount of rows in target database.

Keywords: .NET, ADO.NET, Benchmarking, Entity Framework Core, Databases, ORM, ORM-Framework

Termiluettelo

.NET 6	Microsoftin kehittämä alustariippumaton sovelluskehys. Aiemmat versiot on tunnettu nimellä .NET Core (Microsoft 2022g).
ADO.NET	Osana .NET Frameworkia tuleva kokonaisuus, jolla tuetaan tietokantaoperaatioiden toteuttamista .NET koodista (Hamilton ja MacDonald 2003, s. 3–7).
Benchmark.net	Avoimen lähdekoodin kirjasto .NET sovelluksen suorituskyvyn mittaamiseen (. Foundation 2023).
C#	Microsoftin ohjelmointikieli .NET ympäristöihin (Microsoft 2023).
Data Access Layer	Kerrosarkkitehtuurin kerros, jonka kautta muut kerrokset ovat yhteydessä tietokantaan.
Dapper	Avoimen lähdekoodin ORM-kehys .NET kehitykseen (GitHub 2023).
Entity Framework Core	Microsoftin kehittämä avoimen lähdekoodin ORM-kehys, joka on kehitetty toimimaan .NET 6 sovelluskehyksessä (Microsoft 2022d).
Java	Eräs maailman eniten käytetty ohjelmointikieli (Oracle 2022c).
MS SQL Server	Microsoftin relaatiotietokanta (Gorman ym. 2020).
MariaDB	Avoimeen lähdekoodiin perustuva relaatiotietokanta (M. Foundation 2023).
MySQL	Oraclen omistama avoimeen lähdekoodiin perustuva relaatiotietokanta (Oracle 2022b).
Nuget	Julkinen pakettienhallintajärjestelmä .NET kehittäjille (Oracle 2022b).
Oracle Database	Oraclen toteuttama tietokannanhallintajärjestelmä (Oracle 2022a).

ORM-kehys	Lyhenne ORM, tulee englanninkielisistä sanoista Object Relational Mapping. Tätä teknologiaa hyödyntäen sovelluskehittäjä voi toteuttaa tietokantaoperaatioita sovelluskoodin olioiden kautta välittämättä tietokannanhallintaohjelmiston vaatimasta syntaksista (KumarP ja Suaib0TP, n.d.).
PostgreSQL	Eräs avoimeen lähdekoodiin perustuva relaatiotietokanta (The PostgreSQL Global Development Group 2022).

Kuviot

Kuvio 1. Tutkimuksessa vertailtavat kerrosmallit. 3
Kuvio 2. Oliomalli perinnästä. 5
Kuvio 3. Tietokantayhteyksien evoluutio .NET-ympäristöissä. 11
Kuvio 4. Tutkimussovelluksen oleelliset luokat sekä tutkimuksen tietokannat. 22
Kuvio 5. Mittauskertojen suorittaminen. 30
Kuvio 6. SELECT-1, mittaustulokset viiva- ja pylväskaavioissa. 33
Kuvio 7. SELECT-2, Pohjadatan määrä ei hidastanut tapauksen suoritusta merkittävästi. 34
Kuvio 8. SELECT-3, tapausten suoritusaika kasvaa ja EF Coren aiheuttama suhteellinen viive pienenee mittauskertojen edetessä. 35
Kuvio 9. SELECT-4, suhteellinen hidastuminen vaihtelee paljon eri tietokantojen välillä. 36
Kuvio 10. UPDATE-1, EF Coren aiheuttama viive oli keskimäärin noin 50–75 %. 37
Kuvio 11. UPDATE-2, EF Core -toteutuksen suoritusajat kasvavat merkittävästi mittauskertojen edetessä. 38
Kuvio 12. DELETE-1, datamäärien kasvaminen ei vaikuttanut suorituskykyyn. 40
Kuvio 13. DELETE-2, EF Core hidastaa suoritusta noin 130–170 %. 41
Kuvio 14. DELETE-3, tuloksia koottuna. 42
Kuvio 15. INSERT-1, EF Coren aiheuttamassa viiveessä oli hajontaa eri tietokantojen välillä. 43
Kuvio 16. Adventureworks-tietokannan ER-kaavio. 64

Taulukot

Taulukko 1. Tutkimuksessa tutkittavat tapaukset koostettuna. 24
Taulukko 2. Pohjadatan määrät eri mittauskerroilla (tuhatta riviä). 31

Koodiesimerkit

Koodiesimerkki 1. ORM-kehys kuvaa tietokantaoperaation oliomaisesti. 7
Koodiesimerkki 2. Muutosten seuranta EF Core ORM-mallissa. 7
Koodiesimerkki 3. EF Core ja viiteavainten huomiointi INSERT-operaatiossa. 8
Koodiesimerkki 4. EF Core ja tietokantaoperaatioiden määrä. 10
Koodiesimerkki 5. EF Core (Data annotations), ominaisuuden kartoittamatta jättäminen. 16
Koodiesimerkki 6. EF Core (Fluent-API), ominaisuuden kartoittamatta jättäminen. 17

Sisällys

1 JOHDANTO ... 1

2 ORM-KEHYS ... 4
2.1 Miksi ORM-kehyksiä on kehitetty? ... 4
2.1.1 Object relational impendance mismatch ... 4
2.2 ORM-kehysten ominaisuuksia ... 7
2.3 ORM-kehysten hyvät ja huonot puolet ... 9

3 TIETOKANTAYHTEYKSIEN HISTORIAA .NET-YMPÄRISTÖISSÄ ... 11
3.1 ODBC ... 12
3.2 OLE DB ja ADO ... 12
3.3 ADO.NET ... 12
3.3.1 ADO.NET tuottajat ... 13
3.3.2 ADO.NET DataSet-luokka ... 13

4 ENTITY FRAMEWORK CORE ... 14
4.1 Entity Framework -tuottaja ... 14
4.2 Kehittämisen lähestymistavat EF Coressa ... 15
4.2.1 Tietokanta ensin ... 15
4.2.2 Koodi ensin ... 16
4.3 Tietokantatransaktiot ... 17

5 TUTKIMUSASETELMA ... 18
5.1 Tutkimuskysymys ... 18
5.2 Tutkimusmenetelmä ja -strategia ... 18

6 TUTKIMUKSEN TOTEUTTAMINEN ... 20
6.1 Tutkittavat tietokanta-alustat ja EF Core -tuottajat ... 20
6.2 Tutkimusta varten kehitetty sovellus ... 21
6.3 Tutkimusympäristö ... 21
6.3.1 Tutkimusympäristön alkutila ... 22
6.4 Tutkittavat tapaukset ... 23
6.4.1 SELECT ... 25
6.4.2 UPDATE ... 26
6.4.3 DELETE ... 27
6.4.4 INSERT ... 28
6.5 Mittauskertojen suorittaminen ... 29
6.6 Tutkimusdata ... 30

7 TUTKIMUKSEN TULOKSET. ... 32
7.1 SELECT ... 32
7.2 UPDATE ... 37
7.3 DELETE ... 39
7.4 INSERT ... 43

8 POHDINTA ... 44
8.1 Tutkimuskysymykseen vastaaminen ... 44
8.2 Tulosten merkitys ... 45
8.3 Tutkimuksen luotettavuuden arviointi ... 46

9 JOHTOPÄÄTÖKSET ... 47

LÄHTEET ... 49

LIITTEET ... 54

1 Johdanto

Olio-ohjelmointi on yksi tunnetuimmista ohjelmointiparadigmoista. Sitä käytetään paljon etenkin asiakas-palvelin-tyyppiseen arkkitehtuuriin perustuvissa sovelluksissa tai hajautetuissa järjestelmissä. Olioparadigman etuna on muun muassa se, että palvelimelta pyydetty tieto pystytään kapseloimaan tietorakenteeksi eli olioksi, ja toimittamaan asiakkaalle riippumatta siitä, mitä tietoja muut asiakkaat saman aikaisesti pyytävät palvelimelta (Koskimies ja Mikkonen 2005, s. 137).

Tiedon määrän kasvaessa ihmisten sekä organisaatioiden tarpeisiin kehitetään jatkuvasti lisää sovelluksia, joiden kautta käyttäjät pystyvät hyödyntämään tietokannoissa olevaa tietoa tai luomaan uutta tietoa tietokantoihin (Hilbert ja López 2011). Relaatiotietokannat ja olio-ohjelmointi on tarkoitettu eri ongelmien ratkaisemiseen, mutta tietojärjestelmien toteuttamiseksi tarvitaan usein molempia. Näiden kahden paradigman yhdistäminen toimivaksi ja ylläpidettäväksi kokonaisuudeksi vaatii tietojen kartoittamista olioiden sekä tietokannan taulujen välillä. Tehtävä ei ole yksinkertainen, joten sen helpottamiseen on kehitetty erilaisia työkaluja.

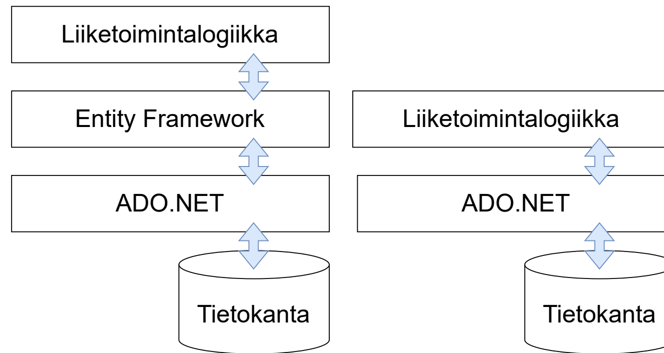
Tässä tutkielmassa syvennytään jo mainittujen haasteiden ratkaisemiseen .NET ympäristössä, ja tutkimuksen kohteena ovat erityisesti Microsoftin Entity Framework Core (Myöhemmin EF Core) sekä ADO.NET. EF Core on ORM-kehys (Object Relational Mapper), eli työkalu olioparadigman ja relaatiotietokantojen yhteensovittamiseen. ADO.NET on teknologia, jonka läpi tietokanta-asiointi pääosin hoidetaan .NET kehityksessä. Myös EF Core käyttää sisäisesti ADO.NET:iä, mutta se lisää sovellukseen uuden abstraktiotason helpottamaan sovelluskehittäjien työtä. EF Coren, tai muidenkaan ORM-kehysten käyttö ei tuo mukanaan mitään loppukäyttäjälle näkyvää toiminnallisuutta, ja yleisesti ottaen ne hidastavat sovellusten toimintaa hieman. Toisaalta niiden mukanaan tuomat hyödyt nopeuttavat ohjelmistokehityksen prosesseja, ja helpottavat tietokanta-asioinnin toteutusta sellaisille ohjelmistokehittäjille, jotka eivät ole tietokanta-asiantuntijoita. Mielipiteitä ORM-kehysten käytöstä riittää puolesta ja vastaan.

EF Core on Microsoftin ylläpitämä avoimen lähdekoodin ORM-kehys, joka on kehitetty toimimaan välittäjänä sovelluskerroksessa käytettyjen luokkien ja tietokantayhteytenä toimivan ADO.NET:in välillä (Halpin 1998; Smith 2021; Hamilton ja MacDonald 2003). Jere Moilanen (Moilanen 2020) on vertaillut Jyväskylän yliopistossa kirjoittamassaan tutkielmassa Entity Frameworkin toteutusta eri tietokanta-alustojen päällä toiminnallisesta näkökulmasta. Moilasen tutkimuksessa havaittiin paljon eroja eri Entity Framework tuottajien toiminnallisuuksissa eri tietokanta-alustoilla.

Tutkimuksessa verrataan EF Coren tuottamien tietokantaoperaatioiden nopeutta käsin koodattujen, ADO.NET kehyksen läpi suoritettavien tietokantaoperaatioiden nopeuteen. Koska EF Core hyödyntää ADO.NET:iä rajapintana tietokantaan, edellä mainittu vertailu auttaa keräämään numeerista tietoa siitä, kuinka paljon EF Core hidastaa sovelluksen toimintaa eri tietokantaoperaatioissa verrattuna suoraan ADO.NET toteutukseen. Käytännössä tämä tarkoittaa kahden eri kerrosmallin vertailua, jotka on kuvattu kuviossa 1. Tutkimuksen tarkoitus ei ole kertoa, onko ORM-kehysten käyttö hyvä vai huono asia, mutta jokaisen ORM-kehyksiä hyödyntävän ohjelmistokehittäjän ja tiimin kannattaisi tehdä tietoinen päätös, milloin ORM-kehyksen käytöstä on enemmän hyötyä kuin haittaa toimivan kokonaisuuden kannalta.

Tutkimuksen tarkoitus on selvittää 13 eri käyttötapauksen perusteella, ovatko EF Coren aiheuttamat suorituskykyvaikutukset niin merkittäviä, että niillä olisi merkitystä järjestelmien suorituskyvyn kannalta. Tuomas Östman (Östman 2020) on vertaillut tutkinut ORM-kehysten suoritusaikoja Lappeenrannan yliopistolle tekemässään tutkielmassa, mutta Östman vertaili Entity Frameworkia ja Dapperia ADO.NET:iin käyttäen tietokantana Microsoftin SQL Serveriä. Tässä tutkielmassa vertaillaan EF Corea ADO.NET:iin useammalla eri tietokannanhallintajärjestelmällä.

Aihe on teollisuuden näkökulmasta kiinnostava, sillä teknologiavalintojen onnistuminen on yksi onnistuneen projektin kulmakivistä, kun taas epäonnistunut teknologiavalinta voi johtaa suorituskykyongelmiin, jos järjestelmän tietokantakyselyt muodostavat pullonkaulan esimerkiksi kuorman kasvaessa ennakoitua suuremmaksi. (Bondi 2000). Myös mikropalveluarkkitehtuurissa yksittäinen hidas mikropalvelu voi hidastaa tiedon kulkua eri palveluiden välillä, mikäli sen tietokantakyselyissä ilmenee suorituskykyongelmia (Dragoni ym. 2017, s. 11).



Kuvio 1: Tutkimuksessa vertailtavat kerrosmallit.

Luvussa 2 kuvataan tarkemmin, mitä ongelmia ORM-kehykset on kehitetty ratkaisemaan, ja esitellään ORM-kehyksiä yleisellä tasolla. Luvussa 3 käydään lyhyesti läpi tietokantaohjelmoinnin käsitteitä ja historiaa .NET-ympäristöissä, koska ne ovat oleellisia pohjatietoja ADO.NET:in ja EF Coren ymmärtämiseksi. Luku 4 kuvaa tarkemmin EF Coren toiminnallisuuksia muun muassa koodiesimerkkien avulla. Luku 5 esittelee tutkimusasetelman, tutkimusmenetelmän sekä tutkimuskysymyksen, ja luvussa 6 kuvaillaan tutkimuksen toteutusta ja sen yhteydessä kehitettyä sovellusta. Luvussa 7 esitellään tutkimuksen tulokset sellaisenaan, ja luvussa 8 käydään tulosten syitä ja niiden merkitystä sanallisesti pohtien läpi. Gradun päättää luku 9, jossa tehdään tiivistetty yhteenveto tutkimuksesta, ja pohditaan mahdollisia jatkotutkimuskohteita.

2 ORM-kehys

Luvussa esitellään ORM-kehyksiä yleisellä tasolla sekä käydään läpi niiden hyviä ja huonoja puolia ohjelmistokehityksessä. ORM on lyhenne sanoista *Object Relational Mapper*, ja ohjelmistoalalla on käytössä paljon eri ohjelmointikielillä kehitettyjä, eri tietokannanhallintajärjestelmien kanssa yhteensopivia ORM-kehyksiä.

2.1 Miksi ORM-kehyksiä on kehitetty?

ORM-kehykset on kehitetty helpottamaan tietokantapohjaisten olio-ohjelmoinnilla toteutettavien sovellusten kehitystä. Olio-ohjelmoinnissa tieto kapseloidaan tietorakenteeseen eli olioon, ja ORM-kehysten avulla tiedon tallentaminen oliosta relaatiotietokantaan sekä tiedon hakeminen tietokannasta olioon on helppoa. (Wiphusitphunpol ja Lertrusdachakul 2017, s. 423)

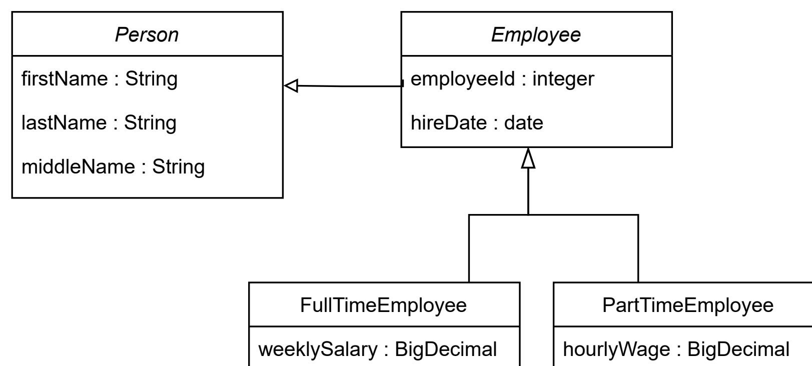
Kuten Ireland ym. (2009) ja Russell (2008) artikkeleissaan kirjoittavat, ohjelmistokehityksessä vallitsevat kulloinkin valittujen paradigmojen lainalaisuudet. Esimerkkeinä paradigmoista mainittakoon tämän gradun kannalta oleellisimmat: olio-ohjelmointi ja relaatiotietokannat. Nämä kaksi paradigmaa esiintyvät usein tietokantapohjaisissa sovelluksissa yhtä aikaa. Oliokielissä kuten Java tai C#, sovelluskerroksen toteutus tehdään tyypillisesti olio-ohjelmoinnilla, ja tietokannoista suosituimpia ovat relaatiotietokannat. Kun käytetään olio-ohjelmointia sovelluskehityksessä, sovelluksessa käytettävät käsitteet eli luokat ja niiden väliset suhteet suunnitellaan kuvaamaan sovellusalueen reaalimaailmaa. Tietokannan suunnittelussa taas siirrytään matalammalle abstraktiotasolle, jolloin yhden olion tieto saattaa hajautua useampaan tietokantatauluun säilytettäväksi.

2.1.1 Object relational impendance mismatch

Ei ole yksinkertainen tehtävä sovittaa yhteen sovelluskerroksessa suosittua oliomallia ja relaatiotietokantoja, sillä molemmissa edellä mainituista paradigmoista pätevät niiden omat lainalaisuutensa. Esimerkiksi C#-kielellä merkkijono-tyyppinen kenttä on tyyppiä `String`, mutta SQL-tietokannoissa merkkijono-tyyppisissä sarakkeissa on mukana mahdollisuus ra-

joittaa merkkijonon pituutta. Toisena esimerkkinä mainittakoon desimaaliluvut, joiden tarkkuuden määritys tapahtuu C#-kielessä oliotasolla, mutta tietokantatasolla kaikki samaan sarakkeeseen tallennetut desimaaliluvut tallennetaan samalla tarkkuudella.

Relaatiotietokannoista puuttuvat myös esimerkiksi olio-ohjelmoinnin perintä, rajapinnat sekä abstraktit luokat, joilla pyritään lisäämään koodin uudelleenkäytettävyyttä ja vähentämään koodin toisteisuutta. Tästä esimerkkinä Russell (2008, s. 19) kuvaa oliorakenteen, jossa abstrakti luokka `Employee` perii `Person`-luokan, ja luokat `FullTimeEmployee` ja `PartTimeEmployee` perivät abstraktin `Employee`-luokan.



Kuvio 2: Oliomalli perinnästä. (Russell 2008, s. 19)

.

Tällaista oliomallia vastaavan tietokantarakenteen voi toteuttaa kolmella eri tavalla: yksi taulu (*Single table*), taulu per luokka (*Table per class*) tai taulu per konkreettinen luokka (*Table per concrete class*). Suoraviivaisin näistä on yksittäisen taulun toteutus, jossa kaikki kuvion 2 luokkien sisältämät sovellusalueen tiedot tallennetaan yhteen tauluun. Sovellusalueen tietojen lisäksi tähän yksittäiseen tauluun tarvittaisiin myös sarake kertomaan, mitä sovellusalueen oliota kukin rivi vastaa. Tuo sarake voisi olla tyyppiä `char(1)`, ja se voisi sisältää arvot `S`, `E`, `F` tai `P`, kuvaten sovellusalueen luokan alkukirjainta. (Russell 2008, s. 20)

Taulu per luokka- ja taulu per konkreettinen luokka -lähestymistavoissa luodaan sovellusalueen luokkia vastaavat taulut ja niiden väliset relaatiot viiteavaimia hyödyntäen. Näissä toteutustavoissa hyvänä puolena on, että sovelluskehityksen kannalta rakenne on selkeämpi, koska se vastaa sovellusalueen luokkia. Huono puoli näissä on se, että tietojen hakeminen

tietokannasta vaatii taulujen liittämistä toisiinsa `JOIN`-lauseilla, jotka saattavat tapauksista riippuen johtaa epäoptimaalisiin tietokantakyselyihin. (Russell 2008, s. 20)

Tässä aliluvussa kuvattuja ongelmia kutsutaan englanniksi termillä *Object-relational Impedance Mismatch*. Ireland ym. (2009, s. 36) luokittelevat näitä ongelmia artikkelissaan neljään eri luokkaan (Ireland ym. 2009):

- **Paradigma (paradigm):** Tämän luokan ongelmia ovat olio-ohjelmoinnin sekä relaatiotietokantojen väliset konseptuaaliset erot. Olio-ohjelmoinnissa tieto ilmenee toisiinsa kytkeytyneiden olioiden verkostona, kun taas relaatiotietokannoissa tietoa käsitellään tietokantataulujen välisten relaatioiden avulla.
- **Kieli (language):** Tämän luokan ongelmia ovat toisistaan poikkeavien ohjelmointikielten (esim. C# ja SQL) syntaksiin sekä niihin sisäänrakennettuihin tietorakenteisiin liittyvät ongelmat.
- **Kaavio (schema):** Tämän luokan ongelmat liittyvät siihen, kuinka sovellusalueen käsitteet esitetään eri tavalla eri abstraktiotasoilla (tietokantakerros vs. sovelluskerros) ja siihen, miten tietoa kartoitetaan (*mapping*) näiden kahden kerroksen välillä.
- **Ilmentymä (instance):** Näitä ongelmia ovat esimerkiksi merkkijonojen pituusrajoituksen puuttuminen sovelluskerroksesta.

Graig Russell (2008) kuvaa artikkelissaan ORM-kehyksiä sillaksi näiden kahden toisistaan poikkeavan paradigman yhteensovittamiseksi. ORM-kehykset eivät ole täydellinen ratkaisu, mutta ne yrittävät ratkaista ongelman parhaansa mukaan.

Hieman negatiivisemmin ORM-kehyksiin suhtautuu Piilaaksossa vaikuttava tietokirjailija ja ohjelmistoarkkitehti Ted Neward, joka on kirjoittanut suositun esseen, "The Vietnam of Computer Science"(Neward 2006). Esseessä todetaan sen aikaisten ORM-kehysten olevan houkutteleva ja usein projektin alkuvaihetta nopeuttava työkalu, mutta niiden kanssa ajaudutaan helposti ongelmiin, jos järjestelmän vaatimukset muuttuvat ennakoitua monimutkaisemmiksi. Esseessään Neward toteaa, ettei täydellistä ORM-kehyksen määritelmää ole. Hän itse määrittelee myös luvussa 3 esiteltyjen ADO.NET tai ODBC-tietohakujen yhteydessä tapahtuvan tietojen kartoittamisen esimerkkinä kevyestä ORM-kehyksestä, koska siinäkin kartoitetaan tietoja oliomallin sekä relaatiotietokannan välillä. Hän myös pitää tällaisen, kevyem-

män lähestymistavan yhdistämistä mallipohjaiseen koodigenerointiin hyödyllisenä. Newardin mukaan monet Javan tai .NET-frameworkin *Data access layer* -toteutukset ovat yhdistelmä ADO.NET:iä, tietojen kartoitusta ja koodigenerointia.

Maailmalla on yrityksiä, joissa perinteisen ORM-mallin sijaan käytetään itse kehitettyä koodigeneraattoria generoimaan SQL-tietokantahakuja. Yksi näistä on suomalainen SC Software (entinen SoulCore), jonka mallinnus- ja generointityökalua on tutkinut Laura Fadjukoff Tampereen yliopistossa kirjoittamassaan diplomityössä (Fadjukoff 2021).

2.2 ORM-kehysten ominaisuuksia

Aliluvussa listataan ORM-kehysten yleisimpiä ominaisuuksia, jotka osaltaan selittävät niiden suosiota. Luvun koodiesimerkit on toteutettu C#-kielellä ja EF Corella, mutta syntaksin sijaan luvussa on tarkoitus tarkastella ORM-kehyksen käyttöä yleisellä tasolla.

Tietokantaoperaation toteutus oliomaisesti: Kun kehittäjä on ottanut ORM-kehyksen käyttöön projektissa, voidaan koodiesimerkki 1:n mukaisesti tietokannasta hakea `Product`-olioita esimerkiksi `ID`-kentän perusteella ilman, että kehittäjällä on tarkempaa tietoa relaatiotietokannan rakenteesta tai hakuun vaadittavasta SQL-syntaksista.

```
using (GraduDBContext ctx = new GraduDBContext())
{
    Product p = ctx.Product.FirstOrDefault(x => x.ID == productId);
}
```

Koodiesimerkki 1: ORM-kehys kuvaa tietokantaoperaation oliomaisesti.

Muutosten seuranta: Koska ORM-kehykset ovat ikään kuin paikallinen tietokanta sovelluksen muistissa, voidaan edellisessä koodiesimerkissä tietokannasta haettuun `Product p`:hen tehdä ohjelmakoodissa muutoksia, kuten koodiesimerkissä 2:

```
p.ProductNumber = "123";
p.Color = "Red";
ctx.SaveChanges();
```

Koodiesimerkki 2: Muutosten seuranta EF Core ORM-mallissa.

Esimerkin 2 tapauksessa tiedot tallentuvat tietokantaan `ctx.SaveChanges()` -kutsulla. Ennen sitä koodissa voitaisiin asettaa esimerkiksi `p.Color` useita kertoja. ORM-kehykseen sisäänrakennettu muutosten seuranta pitää huolen siitä, että tietokantaan lähetetään vain yksi `UPDATE`-lause, jossa tietokantariville päivitetään ohjelmakoodissa asetetut arvot vain muuttuneisiin kenttiin.

Jos `Product`-olioon asetettaisiin koodissa samat arvot, jotka ovat jo tietokannasta, muutosten seuranta pitäisi huolen siitä, että tietokantaan ei ajettaisi turhaan `UPDATE`-lausetta. Muutosten seuranta lisää sovelluksen muistinkulutusta sekä tietokantahakujen vasteaikaa, minkä vaikutus voi olla merkittävä, jos sovelluksessa haetaan suuri tietomäärä kerrallaan. Muistinkulutuksen takia muutosten seuranta voidaan ORM-kehyksissä kytkeä pois päältä, jos käyttötapauksessa ei ole tarvetta tallentaa tietokannasta haettua tietoa takaisin tietokantaan. Tällainen käyttötapaus voi olla esimerkiksi tuotteiden listaus käyttäjälle verkkokaupan käyttöliittymällä. (Microsoft 2022e)

Tietokantaoperaatioiden automaattinen järjestäminen ja viiteavainten huomiointi: Jos tietokantaan lisätään rivejä, joilla on viiteavaimella määritetty relaatio tietokannassa, olisi suoraan tietokantaa vasten tehtävässä toteutuksessa huomioitava, että pääavain–viiteavain suhteen viitattava entiteetti tallennetaan tietokantaan ennen siihen viittaavaa entiteettiä. Esimerkkinä tällaisesta voisi olla uuden tuotekategorian ja uuteen kategoriaan kuuluvan tuotteen lisääminen tietokantaan. Relaatioparadigman periaatteiden mukaan ensin tulisi tallentaa tuotekategoria, jonka jälkeen tuotteen voisi lisätä. ORM-kehyksiä hyödynnettäessä tällaisesta ei tarvitse huolehtia. Koodiesimerkissä 3 nähdään tuotekategorian ja tuotteen lisääminen tietokantaan.

```
using (GraduDBContext ctx = new(GraduDBType.SQLServer))
{
    ProductCategory cat = new ProductCategory(){
        Name = "Shoes"
    };

    Product p = new Product()
    {
        ProductCategory = cat,
        Name = "Worker␣shoes",
```

```
        ProductNumber = "888"
    };

    ctx.Product.Add(p);
    ctx.ProductCategory.Add(cat);

    ctx.SaveChanges();
}
```

Koodiesimerkki 3: EF Core ja viiteavainten huomiointi INSERT-operaatiossa.

Esimerkkitoteutuksen `ctx.SaveChanges();` tallentaa tiedot tietokantaan oikein riippumatta komentojen `ctx.Product.Add(p);` ja `ctx.ProductCategory.Add(cat);` välisestä järjestyksestä. ORM-kehykset osaavat huomioidaan viiteavainten määrittämän järjestyksen myös tietoja poistettaessa. (Microsoft 2022e)

2.3 ORM-kehysten hyvät ja huonot puolet

Koska ORM-kehyksen avulla tietokanta mallinnetaan olioiksi, pystyvät ohjelmistokehittäjät toteuttamaan olioiden tietokantakäsittelyyn (tiedon hakeminen, muokkaus, tallentaminen tai poistaminen tietokannasta) liittyvän ohjelmistokoodin välittämättä sen enempää siitä, minkälaisia tietokantakyselyitä eri operaatioiden toteuttaminen valitulla tietokanta-alustalla vaatisi. Tämä nopeuttaa huomattavasti sovelluskehitystä ja madaltaa kehittäjien vaatimustasoa, sillä ORM-kehystä käytettäessä perusoperaatioissa ei vaadita minkään tietyn tietokannanhallintajärjestelmän erikoisosaamista. (KumarP ja Suaib0TP, n.d., s. 735–736)

Lisäksi tietokantarakenteen muuttuessa tietokannan rakenne ja sovelluksen oliomalli on helppoa synkronoida automaattisesti keskenään ilman erityistä tietokantaosaamista (Bauer ja King 2006, s. 24–35). Yleisesti käytössä olevien ORM-kehysten käyttö lisää myös sovelluksen tietoturvaa. Mikäli sovelluksen kaikki tietokantaoperaatiot on toteutettu ORM-kehyksen avulla, ei kehittäjän tarvitse erikseen huomioida esimerkiksi SQL-injektion mahdollisuutta, sillä laajasti käytössä olevat ORM-kehykset estävät SQL-injektion mahdollisuuden.

Edellä mainituista hyödyistä huolimatta ORM-kehysten käytöstä seuraa myös haasteita. Yksi ORM-kehysten merkittävä haittapuoli on se, että kun aidot tietokantaoperaatiot piilotetaan ohjelmistokehittäjältä, saattaa varomaton ORM-kehyksen käyttö johtaa suorituskyvyn heikkenemiseen. Eri ORM-kehysten välillä on myös laatueroja, joten valitun ORM-kehyksen suorituskykyä ei välttämättä pystytä ennalta todentamaan ilman käytännön kokeilemista. (Moilanen 2020, s. 6)

ORM-kehysten käyttö saattaa myös lisätä tietokantaan suoritettavien kyselyiden määrää. Esimerkiksi kuvitteellisessa tuotteiden hakeminen pääkategorian nimen perusteella toteutettaisiin EF Corella koodiesimerkki 4 mukaisesti.

```
using (GraduDBContext ctx = new GraduDBContext())
{
    var cat = ctx.ProductCategory
                                .Where(x => x.Name == searchKey);
    var subCats = ctx.ProductCategory
                        .Where(x => x.ParentProductCategoryID == cat.Id);
    var products = new List<Product>();
    subCategories.ForEach(delegate (ProductCategory cat)
    {
        products.AddRange(cat.Product);
    });
    return products;
}
\
```

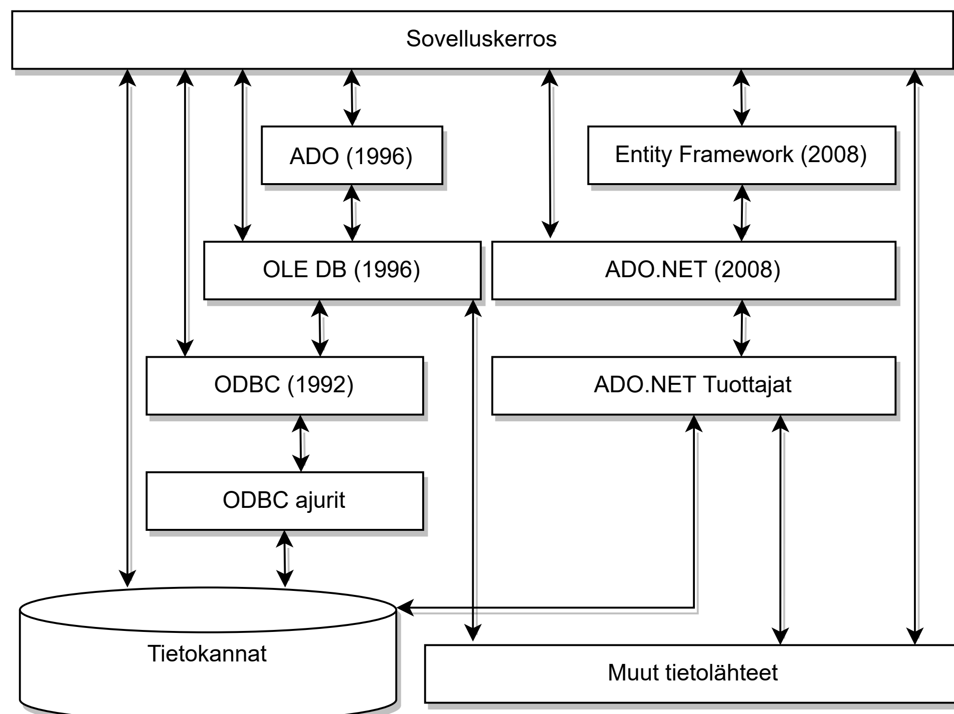
Koodiesimerkki 4: EF Core ja tietokantaoperaatioiden määrä.

Pääkategorian hakemiseksi tehdään ensin yksi tietokantakysely, jonka jälkeen haetaan toisella kyselyllä alikategoriat. Alikategorioiden määrästä riippuen tehdään vielä 0–n tietokantakyselyä tuotteiden hakemiseksi. Tämä voi johtaa merkittävään määrään tietokantakyselyitä, alikategorioiden määrän kasvaessa.

3 Tietokantayhteyksien historiaa .NET-ympäristöissä

Tässä luvussa kuvataan lyhyesti tietokantayhteyksien historiaa sovelluskehityksessä Microsoftin .NET ympäristöissä. Luvussa esitellään lyhyesti ODBC, OLE DB ja ADO.NET sekä niiden historiaa, sillä ne ovat olleet osa tietokantaohjelmoinnin evoluutiota .NET sovelluskehityksessä. Lisäksi ne ovat kaikki edelleen vaihtoehtoisia tapoja yhdistää sovelluksia tietolähteisiin. Luvussa taustoitetaan aihetta, jotta voidaan paremmin ymmärtää ORM-kehysten ja erityisesti EF Coren viitekehystä. Luvussa esitellyt teknologiat ja niiden väliset suhteet kuvataan kuviossa 3.

Kuten Jere Moilanen (2020, s. 3) tutkielmassaan esittää, ennen ORM-kehysten yleistymistä tietokantakyselyt kirjoitettiin osana sovellusten kehittämistä. Ennen ODBC:n (Open Database Connectivity) julkaisua, eli vuoteen 1992 (Moilanen 2020, s. 3) asti jokaiselle tietokannanhallintajärjestelmälle tarvittiin oma ajurinsa, sillä standardisoitua tapaa tietokantaoperaatioiden suorittamiseksi ei ollut.



Kuvio 3: Tietokantayhteyksien evoluutio .NET-ympäristöissä (Microsoft 2022b).

3.1 ODBC

Vuonna 1992 Microsoft julkaisi ODBC:n (Open Database Connectivity Standard), jonka läpi sovelluksista pystyy suorittamaan tietokantaoperaatioita sellaisiin tietokannanhallintajärjestelmiin, joille on kehitetty ODBC-ajuri.1990-luvun puolivälin jälkeen Microsoftin teknologioiden lisäksi myös muiden ohjelmointikielten kehityksessä alettiin tukea standardoitua tapaa yhdistää sovelluksia tietokannanhallintajärjestelmiin. Esimerkiksi vuonna 1997 julkaistiin JDBC, joka on ODBC:n vastine Java-ympäristöihin (Patel ja Moss 1997). Vaikka ODBC ja JDBC ovat tarjoavat kehittäjille standardin tavan yhdistää sovelluksia eri tietokannanhallintajärjestelmiin, niiden läpi suoritetut tietokantaoperaatiot tulee kirjoittaa ohjelmistokoodiin aina valitun tietokannanhallintajärjestelmän mukaisella syntaksilla. (Abdalhakim 2009)

3.2 OLE DB ja ADO

Vuonna 1996 Microsoft julkaisi OLE DB:n. OLE DB on oliorajapinta, joka kehitettiin laajentamaan Microsoftin aiemmin kehittämää OLE Component Object Model (COM) -mallia tietokantaominaisuuksilla. OLE DB -komponentti on mikä tahansa komponentti, joka toteuttaa OLE DB -rajapinnan. (Blakeley 1996)

OLE DB hyödyntää tietokantayhteyksissään ODBC:ta, mutta ODBC:sta poiketen OLE DB:n avulla voidaan yhdistää sovelluksia myös muihin tietolähteisiin, kuten Excel-taulukoihin tai XML-tiedostoihin. Samaan aikaan OLE DB:n kanssa julkaistiin ADO, jonka on tarkoitus toimia sovellusten rajapintana OLE DB:hen. (Blakeley 1996)

3.3 ADO.NET

Vuonna 2008 osana .NET Frameworkille kehitettyä Entity Frameworkia julkaistiin ADO.NET, joka on eri tietolähteisiin integroitumista helpottava kokoelma luokkia ja kirjastoja. ADO.NET voidaan jakaa arkkitehtuuriltaan karkeasti kahteen osaan: kyselyiden suorittamiseen (*execution*) ja välimuistin käyttöön (*caching*). (Bai 2012)

3.3.1 ADO.NET tuottajat

Kyselyiden suorittaminen toteutuu ADO.NET:ssa ADO.NET tuottajien avulla. ADO.NET tuottajat ovat kullekin tietolähteelle toteutettuja komponentteja, joiden tehtävä on luoda tietokantayhteys, suorittaa tietokantakysely ja katkaista tietokantayhteys kyselyn jälkeen (Bai 2012, s. 93–122). Kun ADO.NET julkaistiin, samoihin aikoihin julkaistiin seuraavat ADO.NET-tuottajat, jotka ovat yhä paljon käytössä:

- Open DataBase Connectivity (Odbc) -tuottaja (ODBC.NET)
- Object Linking and Embedding DataBase (OleDb) -tuottaja (OLEDB.NET)
- SQL Server (Sql) -tuottaja (SQL Server.NET)
- Oracle (Oracle) Data Provider (Oracle.NET).

Yllä olevasta listasta huomataan, että ADO.NET:lle on toteutettu suoraan tietokantakohtaisia tuottajia, mutta on kehitetty myös tietokantateknologiasta riippumattomia tuottajia kuten ODBC.NET sekä OLEDB.NET. (Bai 2012, s. 96–97)

3.3.2 ADO.NET DataSet-luokka

Kun ADO.NET on suorittanut tietokantakyselyn, tieto säilötään `DataSet`-luokan instanssiksi eli olioksi tietokoneen muistiin. `DataSet` oliot ovat tietokanta-alustasta riippumattomia olioita, jotka sisältävät muistinvaraista tietokantaa mallintavia olioita, kuten `DataTable` (vastine tietokannan taululle) tai `DataRow` (vastine tietokannan riville). Sovelluksen toteuttaja voi ADO.NET tietokantakyselyn suorituksen jälkeen kirjoittaa ohjelmakoodin, joka hakee tiedon `DataSet`:istä muun ohjelmakoodin käsiteltäväksi. (Bai 2012, s. 93–122). Näitä luokkia myös EF Core käyttää sisäisesti, mutta piilottaa ne sovelluskehittäjältä.

4 Entity Framework Core

Luvussa esitellään Entity Framework Core, joka on Microsoftin alustariippumattomalle .NET 6 (entinen .NET Core) -sovelluskehykselle kehittämä avoimen lähdekoodin ORM. EF Coressa on paljon ominaisuuksia, jotka jäävät tämän gradun ulkopuolelle, mutta tässä luvussa on koostettuna tutkimuksen kannalta oleellisimmat asiat.

EF Core (versio 6) on valikoitunut suorituskykyvertailussa tarkasteltavaksi ORM-kehykseksi, koska se on yleisesti käytössä ja se on toteutettu avoimen lähdekoodin periaatteella. Lisäksi vanhan Entity Frameworkin kehitys on loppumassa Microsoftin keskittyessä enemmän alustariippumattomien komponenttien kehitykseen. Entity Framework ja Entity Framework Core ovat periaatteiltaan kuitenkin niin toistensa kaltaisia, että tässä työssä käytetään esimerkkejä ja lähdeaineistoa myös Entity Frameworkiin liittyen.

4.1 Entity Framework -tuottaja

Samalla tavalla kuin luvussa 3 kuvailtiin ADO.NET-tuottajia, käytetään myös Entity Frameworkin yhteydessä sanaa tuottaja, joka tarkoittaa tietokannanhallintajärjestelmää varten toteutettua sovituskomponenttia sovelluskerroksen ja tietokantakerroksen välille. Eri tuottajien käyttö mahdollistaa käytännössä eri tietokannanhallintajärjestelmien käyttämisen EF Corella. Koska Entity Framework on alun perin Microsoftin kehittämä, tyypillisesti sitä käytetään MS SQL -tietokannan päällä, ja tällöin tuottaja on esimerkiksi Microsoft.EntityFrameworkCore.SqlServer. Tuottajia on kuitenkin toteutettu paljon eri tietokannanhallintajärjestelmille, kuten Oracle DB, MySQL, MariaDB, Db2. Eri tietokannanhallintajärjestelmille toteutetut tuottajat voivat olla maksullisia, tai tietokanta-alustan kehittäjäyhteisön toteuttamia, ja samalle tietokanta-alustalle voi olla toteutettuna useampia eri tuottajia, joiden ominaisuudet ja suorituskyky voivat poiketa toisistaan. (Microsoft 2022a)

4.2 Kehittämisen lähestymistavat EF Coressa

EF Corella kehitettäessä tulee valita kehitysmalli, jolla EF Corea käytetään. Valittavana on kaksi eri vaihtoehtoa: tietokanta ensin (*Database first*) tai koodi ensin (*Code first*). Vanhassa Entity Frameworkissa on olemassa myös EDM-kartoitustiedostoihin perustuva malli ensin (*Model first*) -lähestymistapa, mutta se ei ole enää mukana EF Coren uusimmissa versioissa, joten se on jätetty tässä tutkimuksessa esittelemättä. (Peres 2016)

4.2.1 Tietokanta ensin

Tietokanta ensin -lähestymistavassa on aina oltava jokin tietokanta, jonka pohjalta generoidaan Entity Framework konteksti ja C#-luokat. Tietokanta ensin -lähestymistapa ei ole ominaisuuksiltaan niin joustava kuin koodi ensin, koska siinä syntyneet luokat vastaavat aina tietokannan rakenteita. Tietokanta ensin on kuitenkin suosittu tapa helppoutensa takia varsinkin, jos aletaan kehittää EF Core -sovellusta jonkin suuren tai jo pitkään käytössä olleen tietokannan päälle, jonka rakenteeseen ei haluta tehdä muutoksia. Tämä lähestymistapa on myös suositeltava valinta, jos yksi tiimi on vastuussa tietokannan rakenteesta ja toinen tiimi vastaa koodin kehittämisestä (Singh ym. 2015).

Tietokanta ensin -lähestymistavassa on myös mahdollista generoida luokkia tietokantanäkymistä (*VIEW*), mikä mahdollistaa yhden tavan kartoittaa tietoja tietokantatauluista sovellusalueen olioihin. Esimerkiksi Kuvion 2 luokkakaavio voitaisiin toteuttaa tietokantaan taulu per luokka -lähestymistavalla, ja luoda tietokantaan näkymä `PartTimeEmployee`, joka piilottaa SQL-liitokset sisäänsä ja sisältää vain ja ainoastaan osa-aika-työntekijöiden tiedot. Tämän jälkeen yksittäinen ohjelmistokehittäjä pystyisi hakemaan osa-aikatyöntekijöiden tietoja `dbContext.PartTimeEmployee.Where()`-komennolla välittämättä siitä, että `PartTimeEmployee` luokka kartoittuu tietokannassa useampaan eri tauluun. Tietokantanäkymien kanssa toimittaessa tietoja ei kuitenkaan voida tallentaa takaisin tietokantaan päin yhtä helposti kuin suoraan taulujen kanssa. (Peres 2016)

4.2.2 Koodi ensin

Koodi ensin -lähestymistavassa tietokanta generoidaan käsin koodattujen `DBContext`, sekä sovellusalueen entiteettejä kuvaavien C#-luokkien perusteella (Peres 2016). Tämä on suosituin tapa varsinkin pienehköissä sovelluksissa tai jos sovelluskehitys aloitetaan ilman olemassa olevaa tietokantaa. Tietojen kartoitusta voidaan konfiguroida koodi ensin -lähestymistavassa kahdella eri menetelmällä, joiden nimet ovat englanniksi *Data Annotations* ja *Fluent API* (Microsoft 2022c, Entity Properties). *Data Annotations* -menetelmässä sovellusalueen entiteettejä kuvaavia C#-luokkia rikastetaan hakasulkeisiin kirjoitetuilla "tägeillä"eli annotaatioilla koodiesimerkin 5 mukaisesti. *Fluent API* -menetelmässä C#-luokkiin ei lisätä "tägejä", vaan samat asiat kuvataan EF Core -konteksissa `OnModelCreating` erilaisina funktiokutsuina, kuten koodiesimerkissä 6 on kuvattu. (Peres 2016)

Koodi ensin -lähestymistavassa tietojen kartoitus on mahdollista tehdä hyvin yksityiskohtaisesti, kun taas tietokanta ensin -lähestymistavassa tietokannan rakenne kartoitetaan lähes sellaisenaan olioiksi. Koodi ensin -lähestymistapa on näistä kahdesta joustavampi. Esimerkiksi luokan ominaisuus (*property*) voidaan jättää kartoittamatta tietokantatauluun. Käytännössä tämä tarkoittaisi, että kyseiselle ominaisuudelle ei generoida tietokantaan saraketta ollenkaan.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

Koodiesimerkki 5: EF Core, ominaisuuden kartoittamatta jättäminen Data annotations -tekniikalla (Microsoft 2022c).

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Ignore(b => b.LoadedFromDatabase);
}
```

Koodiesimerkki 6: EF Core, ominaisuuden kartoittamatta jättäminen Fluent API -tekniikalla (Microsoft 2022c).

4.3 Tietokantatransaktiot

Tietokantatransaktiot ovat mekanismi, jolla hallitaan tiedon eheyttä tietokannassa (Laitila 2005, s. 29–30). Esimerkiksi rahaa siirrettäessä tililtä toiselle järjestelmän on vähennettävä rahaa tililtä A ja lisättävä rahan määrää tilille B. Jos jokin virhe tapahtuu operaatioiden aikana, transaktion käyttö varmistaa, että tietokanta ei jää tilaan, jossa vain toinen näistä kahdesta operaatiosta olisi onnistunut. Transaktion aikana tehdyt tietokantaoperaatiot siis saadaan ajettua tietokantaan kaikki ilman virheitä, tai virheen tapahtuessa kaikki transaktion aikana suoritetut toimenpiteet tietokantaan perutaan.

EF Corea käytettäessä oletuksena käytetään implisiittistä transaktiokäsittelyä. Implisiittinen transaktiokäsittely tarkoittaa, että kun kutsutaan `ctx.SaveChanges();`, kaikki muutosten seurannassa mukana olleet muutokset tallentuvat tietokantaan. Jos tallennuksen aikana tapahtuu virhe, mitään muutoksia ei tallenneta. Implisiittinen transaktiokäsittely riittää useimmissa sovelluksissa, mutta vaativampiin käyttötapauksiin on mahdollista hyödyntää ADO.NET -ohjelmoinnin kaltaista erikseen koodattavaa transaktiokäsittelyä erilaisten, monimutkaisempien skenaarioiden toteuttamiseen. (Microsoft 2022f).

5 Tutkimusasetelma

Luvussa esitellään tutkimuskysymys sekä tutkimusmenetelmä ja -strategia, joilla tutkimuskysymykseen haetaan vastausta. Jatkuvasti kasvavien tietomäärien johdosta eri liiketoiminta-alueille toteutettavien sovellusten suunnittelussa olisi tärkeää pystyä ratkaisemaan olioparadigman ja relaatiotietokantojen yhteensovittaminen niin, että sovellusten suorituskyky on riittävällä tasolla.

5.1 Tutkimuskysymys

Luvuissa 2, 3 ja 4 on perehdytty vaihtoehtoisiin toimintatapoihin (ORM-kehysten käyttö yleisesti sekä EF Core ja ADO.NET vaihtoehtoisina lähestymistapoina). Näiden pohjalta asetettu gradun tutkimuskysymys on:

- Kuinka EF Coren käyttäminen vaikuttaa .NET-sovelluksen suorituskykyyn?

Tutkimuskysymykseen vastaamalla saadaan arvokasta tietoa sovelluskehityksessä mukana olevien osapuolten käyttöön. Lisäksi tutkimuskysymystä ja tutkimuksessa syntyneitä mittaustuloksia analysoimalla saadaan toivottavasti selkeytettyä käsitystä siitä, millaisissa käyttötapauksissa ORM-kehysten käytöstä aiheutuu suhteettoman paljon suorituskykyvaikutuksia saavutettaviin hyötyihin verrattuna. Tutkimus ei vastaa kysymykseen siitä, tulisiko ORM-kehyksiä suosia sovelluskehityksessä nykyistä enemmän tai vähemmän. Tutkimuksessa yritetään kuitenkin tapaustutkimuksen hengessä tutustua EF Coren aiheuttamiin suorituskykyvaikutuksiin, mikä toivottavasti antaa lukijalle ajatuksia tilanteista, joissa ORM-kehyksen käyttämisen sijaan kannattaisi harkita vaihtoehtoisia lähestymistapoja.

5.2 Tutkimusmenetelmä ja -strategia

Tutkielmaan valittu tutkimusstrategia on tapaustutkimus. Tapaustutkimus on syvällinen ja käytännönläheinen tutkimus tai koe, jossa tutkitaan jonkin yleisemmän asian yksittäistä ilmentymää, kuten yksilöä, ryhmää, tapahtumaa tai yhteisöä jossakin tietyssä kontekstissa (Case ja Light 2011).

Tapaukset suunnitellaan aina aiempaan tietoon ja aineistoon perustuen, ja kirjassaan "Making social science matter: why social inquiry fails and how it can succeed again", Flyvbjerg (2001) esittelee strategioita tapausten valintaan riippuen tutkimuskysymyksistä:

- Tutkimukseen kannattaa valita mahdollisimman monipuolisesti erilaisia aiheeseen liittyviä tapauksia.
- Tutkimukseen kannattaa valita myös harvinaisia tai epätavallisia tapauksia mahdollisten ongelmien löytämiseksi.
- Tutkimukseen kannattaa valita sellaisia tapauksia, joista saadaan yleistettyä loogisia johtopäätöksiä: Jos tämä pätee yhteen tapaukseen, se pätee myös muihin tapauksiin.

Tapausten tutkimisen myötä pyritään saamaan laajempaa ymmärrystä kyseisestä ilmiöstä. Tapaustutkimuksesta ei voi yleensä suoraan yleistää tuloksia laajempaan kontekstiin. Tapaustutkimuksen kriitikoiden mukaan yleinen, kontekstista riippumaton teoria on arvokkaampaa kuin kontekstisidonnainen konkreettinen tieto. Flyvbjerkin on vastannut edellä mainittuun kritiikkiin, että kontekstisidonnaisuus ja konkreettiset tulokset ovat tapaustutkimuksen vahvuuksia. (Laine, Bamberg ja Jokinen 2015). Myös Hans Eysenck on todennut: "sometimes we simply have to keep our eyes open and look carefully at individual cases – not in the hope of proving anything, but rather in the hope of learning something."(Eysenck 1976, s. 1- 15).

Tutkielmassa tarkasteltavat tapaukset on valittu mahdollisimman monipuolisesti. Tavoitteena on vastata tutkimuskysymykseen suunnittelemalla 13 erilaista käyttötapausta luvuissa 3, 2 ja 4 esitettyyn teoriaan ja aineistoon pohjautuen. Jokaiselle tapaukselle mitataan suoritusaika sekä EF Corella että ADO.NET:llä toteutettuna. Jokainen tällainen parivertailu suoritetaan useammalla eri tietokanta-alustoja varten kehitetyllä toteutuksella. Lisäksi käyttötapausten suoritusaikoja mitataan eri pohjadatan määrällä tietokannoissa. Näin saadaan lisää tietoa tutkimustulosten luotettavuudesta ja yleistettävyydestä. Tapaustutkimus soveltuu strategiana hyvin tässä tutkielmassa tutkittavien tapausten vertailuun. (Laine, Bamberg ja Jokinen 2015)

6 Tutkimuksen toteuttaminen

Tutkimus toteutettiin mittaamalla tietokantapohjaisissa sovelluksissa tyypillisesti esiintyvien käyttötapausten suoritusaikoja sekä EF Corea hyödyntäen, että suoraan ADO.NET-toteutuksen kautta. Tietokantarakenteena käytettiin Microsoftin Adventureworks-esimerkkitietokannan kevytversiota, joka noudattelee rakenteeltaan yksinkertaisen verkkokauppasovelluksen tietorakennetta. Adventureworks-esimerkkitietokanta MS SQL Serverille ladattiin Microsoftin verkkosivuilta (Microsoft 2020) ja siitä tehtiin vastaavat kopiot tutkimuksen muita tietokanta-alustoja varten.

6.1 Tutkittavat tietokanta-alustat ja EF Core -tuottajat

Tutkimuksessa tutkittiin tapausten suorittamista maailman suosituimpiin relaatiotietokantoihin, jotka ovat seuraavat (Statista 2022; DB-Engines.com 2022):

- Microsoft SQL Server (2019, Developer Edition)
- MySQL (Community Server 8.0.31), avoimen lähdekoodin tietokanta
- PostgreSQL (14.5), avoimen lähdekoodin tietokanta
- Oracle DB(19c), jonka käyttö on ilmaista, jos sitä ei käytetä tuotantoympäristössä.

Tunnettuuden lisäksi toinen peruste kyseisten tietokantojen valinnalle oli se, että ne olivat ilmaiseksi ladattavissa ja käytettävissä tutkimusta varten. Mainituille tietokannanhallintajärjestelmille valittiin vertailtavaksi seuraavat EF Core -tuottajat:

- Microsoftin MS SQL -tuottaja (Microsoft.EntityFrameworkCore.SqlServer v6.0.12)
- MySQL Connector/NET -tuottaja (MySql.EntityFrameworkCore, v6.0.7)
- Npgsql (PostgreSQL -tuottaja) (Npgsql.EntityFrameworkCore.PostgreSQL, v6.0.8)
- Oraclen ODP.NET -tuottaja (Oracle.EntityFrameworkCore, v6.21.61).

Suluissa olevat pisteellä erotellut nimet ovat tuottajan ohjelmapaketin nimiä Nuget-pakettienhallintajärjestelmässä, ja valitut versionumerot ovat ohjelmapakettien viimeisimpiä .NET 6 -versioita mittausten aloitusvaiheessa.

6.2 Tutkimusta varten kehitetty sovellus

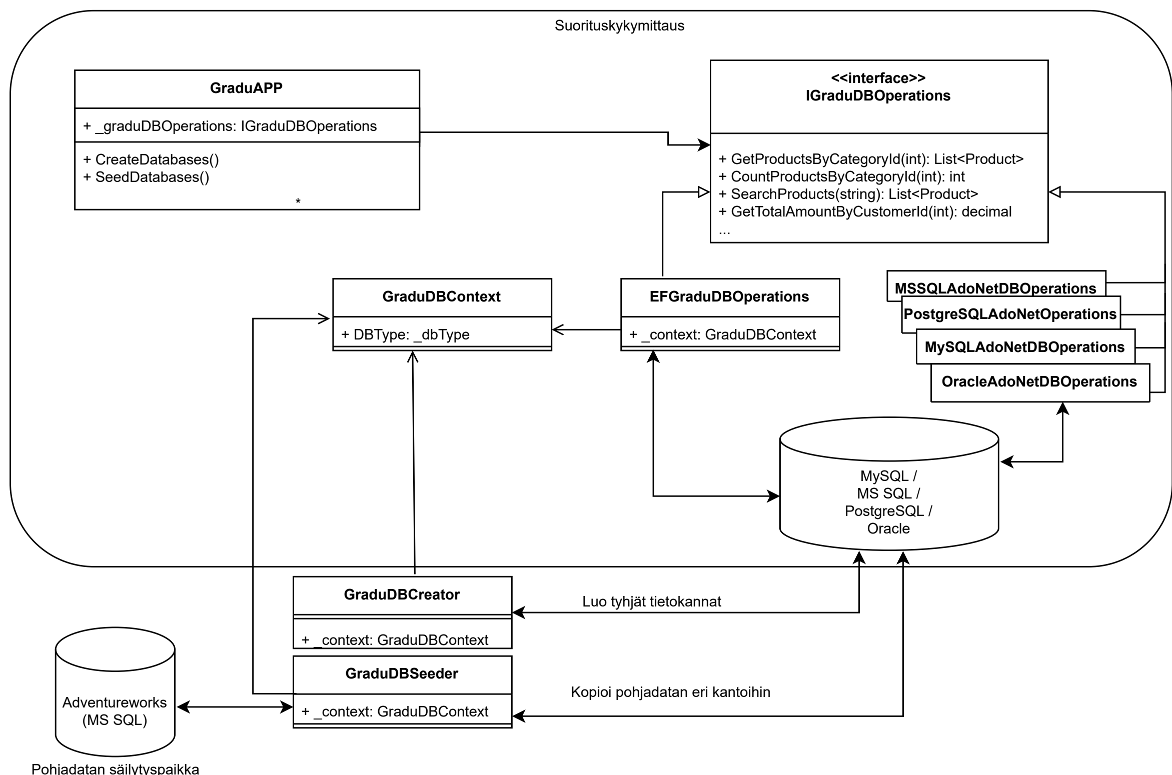
Suorituskykymittausta varten kehitettiin .NET 6 -sovellus, jonka puitteissa kehitettiin ja testattiin tutkimukseen liittyvien tapausten ADO.NET- ja EF Core -toteutuksia. Sovelluksessa ei ole graafista käyttöliittymää. Se on toteutukseltaan kokoelma luokkia, joita käytetään eri tietokantoja vasten toteutettujen integraatiotestien sekä Benchmark.net -kirjaston avulla toteutetun mittaussovelluksen kautta. Tutkimuksessa vertailtavat tapaukset on kuvattu oliorajapinta `IGraduDBOperations`-ohjelmakoodissa metodeina. `IGraduDBOperations`-rajapinnasta tehtiin yksi toteutus, jota kaikki EF Core tuottajat käyttävät. Lisäksi `IGraduDBOperations`-rajapinnasta toteutettiin neljä erillistä ADO.NET:iä hyödyntävää `IGraduDBOperations`-rajapinnan toteutusta, yksi jokaista edellä esiteltyä tietokannanhallintajärjestelmää kohti. `IGraduDBOperations`-oliorajapinnnan ja sen toteuttavien luokkien periaatteena on, että sovelluksen logiikka ei ole riippuvainen siitä, mikä tietokannanhallintajärjestelmä milloinkin on taustalla ja onko .NET-puolen teknologiaksi valittu EF Core vai ADO.NET. Sovelluksen lähdekoodiin voi tutustua GitHub-palvelussa (Valkonen 2023).

6.3 Tutkimusympäristö

Tutkimusympäristönä toimi gradun kirjoittajan tietokone (Dell Latitude 5520), johon oli asennettu edellä listatut tietokannan hallintaohjelmistot ja tutkimussovellus. Tutkimusympäristön käyttöjärjestelmä oli Windows 11, siinä oli Intel Core i71185G7 prosessori ja 32GB muistia. Tutkimusympäristö suunniteltiin niin, että tutkimus voitiin toteuttaa ilman verkkoyhteyksiä. Näin suljettiin pois mahdolliset verkkoliikenteen vaikutukset tutkimustuloksiin. Tutkimuksessa raportoidut suoritusajat ovat tutkimusympäristön sisällä tapahtuneita mittauksia, joten ne eivät ole suoraan vertailukelpoisia muissa ympäristöissä tapahtuviin mittauksiin. Tutkittavien tapausten osalta suhteellinen ero EF Coren ja ADO.NET:in välillä lienee kuitenkin karkeasti yleistettävissä muihin ympäristöihin.

6.3.1 Tutkimusympäristön alkutila

Osana tutkimusta varten kehitettyä sovellusta kehitettiin ominaisuus, jolla alustetaan tutkittaviin tietokantoihin haluttu alkutila. Tällä tavalla varmistetaan, että eri tapausten suoritukset ja mittauskerrat ovat riippumattomia toisistaan, ja että eri tietokanta-alustojen lähtötilanteet mittauksissa olivat rakenteeltaan ja sisältömääriltään keskenään identtiset. Toiminnolla saatiin myös varmistettua, että eri mittausajot olivat vertailukelpoisia keskenään, eivätkä mittaukset hidastuneet tietokannan sisällön muuttuessa yksittäisen mittauskerran aikana.



Kuvio 4: Tutkittavat tietokannat, oleelliset luokat sekä niiden oleellisimmat metodit tutkimuksen näkökulmasta.

Teknisesti tietokantojen alkutilan palautus ja ylläpito tapahtui niin, että tutkimusympäristössä oli koko ajan yksi mittauskerran alkutilaa niin taulurakenteen kuin sisällön osalta vastaava tietokanta. Tätä tietokantaa ei käytetty tutkimuksessa suorituskyvyn mittaamiseen. Varsinaisten tutkimuksessa vertailtavien tietokantojen rakenteet luotiin `GraduDBCreator`-luokan avulla, jonka tehtävä oli poistaa mittauskannat ja sen jälkeen luoda ne uudelleen. `GraduDBCreator` loi tutkimustietokannat käytännössä kopioiden alkutilakannan rakenteen

eri tietokannanhallintajärjestelmiin.

Kun mittauskantojen taulurakenteet oli luotu, lisättiin tietokantoihin pohjadata `GraduDBSeeder`-luokan avulla. Kuten kuviossa 4 kuvataan, `GraduDBSeeder`- luokan tarkoitus on kopioida haluttu pohjasisältö Adventureworks-alkutilakannasta eri tietokanta-alustoille perustettuihin tutkimustietokantoihin. Alkutilakannan rivimääriä lisättiin mittauskertojen välillä taulukon 2 mukaisesti.

6.4 Tutkittavat tapaukset

Aliluvussa esitellään tapaukset, joiden kohdalta tutkimuksessa verrattiin suoritusaikaa EF Core toteutuksen ja ADO.NET-toteutuksen välillä. Lisäksi aliluvussa esitellään tapausten jako neljään eri ryhmään.

Koska tutkimuksessa haluttiin tutkia EF Coren sekä ADO.NET:in operaatioiden suoritusaikoja mahdollisimman kokonaisvaltaisesti, valittiin tutkittavaksi 13 tapausta, jotka jaettiin neljään eri ryhmään:

- Tietojen hakeminen (SELECT).
- Tietojen päivittäminen (UPDATE).
- Tietojen poistaminen (DELETE).
- Uuden tiedon tallentaminen (INSERT).

Koska yksittäisen metodin yksittäinen suorituskerta olisi kestänyt vain muutamia millisekunteja, mittausten luotettavuuden kannalta jokainen tapaus koostuu metodin suorituksesta n kertaa eri parametreilla, ja tämän suorituksen kokonaiskestoa käytetään varsinaisena mitattavana yksikkönä.

Jokaiseen ryhmään suunniteltiin 3–4 tapausta. Tutkimuksen tapauksissa **SELECT-1**, **SELECT-2**, **SELECT-3** ja **SELECT-4** haetaan tietoa tietokannasta, mutta ei päivitetä sitä takaisin tietokantaan. Kuten aliluvussa 2.2 esitellään, tämän kaltaisissa tapauksissa EF Coren suorituskykyä on mahdollista optimoida poistamalla muutosten seuranta käytöstä. Koska tutkimuksessa haluttiin kokeilla erilaisten konfiguraatioiden vaikutusta suorituskykyyn, suoritettiin kyseiset tapaukset sekä muutosten seurannalla että ilman sitä. Jokainen

Taulukko 1: Tutkimuksessa tutkittavat tapaukset koostettuna. Sarake n kertoo, kuinka monta kertaa metodia kutsutaan yhden tapauksen aikana.

ID	**n**	**Kuvaus**
SELECT-1	2 000	Tuotetietojen hakeminen kategoriatunnisteen perusteella.
SELECT-2	2 000	Tuotteiden lukumäärän laskeminen kategoriatunnisteen perusteella.
SELECT-3	200	Tuotteiden hakeminen nimen perusteella.
SELECT-4	2 000	Asiakkaan tekemien ostosten kokonaissumman laskeminen asiakastunnisteen perusteella.
UPDATE-1	2 000	Asiakkaan sukunimen päivittäminen asiakastunnisteen perusteella.
UPDATE-2	2 000	Kategorian tuotteiden hinnan nostaminen 10 prosenttia kategoriatunnisteen perusteella.
UPDATE-3	2 000	Kategorian tuotteiden siirtäminen toiseen kategoriaan kategoriatunnisteen perusteella.
DELETE-1	1 000	Yksittäisen tilausrivin poistaminen tilaukselta tunnisteen perusteella.
DELETE-2	150	Kaikkien asiakkaaseen liittyvien tietojen poistaminen asiakastunnisteen perusteella.
DELETE-3	50	Yksittäisen kategorian ja siihen liittyvien tuotteiden poistaminen kategoriatunnisteen perusteella.
INSERT-1	500	Yksittäisen tuotteen lisääminen.
INSERT-2	500	Tilauksen lisääminen.
INSERT-3	500	Uuden asiakkaan lisääminen.

tapaus vastaa yhtä `IGraduDBOperations`-rajapinnan metodia. Rajapinnan EF Corelle tehty toteutus löytyy liitteestä B

6.4.1 SELECT

SELECT-1, Tuotetietojen hakeminen kategoriatunnisteen perusteella: `IGraduDBOperations` -rajapinnan metodissa `GetProductsByCategoryId(int categoryId)` haetaan tietokannan `Product`-taulusta tuoterivit indeksoidun `categoryID`-kentän perusteella. Kategoriassa olevien tuotteiden määrä kasvaa taulukon 2 mukaisesti, joten metodi palauttaa kasvavan määrän tuloksia eri mittauskertojen välillä. Tapauksessa kutsutaan `GetProductsByCategoryId`-metodia eri parametreilla 2 000 kertaa.

SELECT-2, Tuotteiden lukumäärän laskeminen kategoriatunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodissa `CountProductsByCategoryId(int categoryId)` lasketaan tietokannan `Product`-taulun rivimäärä indeksoidun `CategoryID`-kentän perusteella. Tapauksessa kutsutaan `CountProductsByCategoryId`-metodia eri parametreilla 2 000 kertaa.

Tapaukset **SELECT-1** ja **SELECT-2** ovat hakuehdoiltaan identtisiä, mutta **SELECT-1** palauttaa joukon, olioita, ja **SELECT-2** pelkän tietokannasta löytyvien rivien lukumäärän. On mielekästä tutkia, syntyykö tästä erosta merkittäviä poikkeamia suorituskyvyn suhteen.

SELECT-3, Tuotteiden hakeminen merkkijonon perusteella: `IGraduDBOperations`-rajapinnan metodissa `SearchProducts(string keyword)` haetaan `Product`-taulusta rivit, joiden `Name`-kentässä esiintyy parametrina syötetty merkkijono. `Name`-kenttää ei ole indeksoitu. Rivien määrä `Product`-taulussa kasvaa mittauskertojen edetessä. Tapauksessa kutsutaan `SearchProducts`-metodia eri parametreilla 200 kertaa. Jokainen parametrina syötetty merkkijono on sellainen, jolla löytyy vähintään yksi tuote.

SELECT-4, Asiakkaan tekemien ostosten kokonaissumman laskeminen asiakastunnisteen perusteella: Tapauksessa tutkitaan hieman monimutkaisempaa hakua. `IGraduDBOperations`-rajapinnan metodissa `GetTotalAmountByCustomerId (int customerId)` käytetään `JOIN` operaatiota taulujen `SalesOrderDetail` sekä `SalesOrderHeader` liittämiseksi toisiinsa. Liitoksessa käytetään pääavain–viiteavain relaa-

tiota. Hakuehtona käytetään indeksoitua `SalesOrderHeader`-taulun `CustomerID`-kenttää, jossa säilytetään asiakastunnistetta. Löytyneistä ostosriveistä summataan `LineTotal`-kentän luvut ja palautetaan ne kutsujalle. Tapauksessa kutsutaan `GetTotalAmountByCustomerId`-metodia eri parametreilla 2 000 kertaa.

6.4.2 UPDATE

UPDATE-1, Asiakkaan sukunimen päivittäminen: `IGraduDBOperations`-rajapinnan metodissa `UpdateLastName(int customerId, string newLastName)` päivitetään asiakkaan sukunimi asiakastunnisteen perusteella. EF Core -toteutuksessa tämä tarkoittaa, että ensin haetaan koko asiakasrivi `Customer`-taulusta vastaavaan olioon, minkä jälkeen asetetaan olion `LastName`-kenttään uusi sukunimi ja kutsutaan EF-kontekstin `SaveChanges`-metodia. ADO.NET-toteutus tapahtuu yksinkertaisesti yhdellä `UPDATE`-lauseella. Tapauksessa kutsutaan `UpdateLastName`-metodia eri parametreilla 2 000 kertaa.

UPDATE-2, Kategorian tuotteiden hinnan muuttaminen kategoriatunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodissa `MultiplyPricesByCategoryId (int categoryId, decimal priceChange)` kerrotaan yksittäisen kategorian kaikkien tuotteiden hinta kertoimella, joka syötetään metodin parametrina. Tuloksena kaikkien kategorian tuotteiden hinta muuttuu parametrina syötetyn kertoimen määrän. EF Core-toteutuksessa tämä toteutetaan niin, että haetaan `Product`-taulusta kaikki ne rivit, jossa indeksoidun `CategoryId`-sarakkeen arvo vastaa parametrina syötettyä kategoriatunnistetta. Tämän jälkeen EF Core muodostaa haetuista tietokantariveistä oliot. Oliot käsitellään listana, ja jokaiselle listan oliolle asetetaan `ListPrice`-kenttään uusi hinta, joka lasketaan kaavalla `ListPrice * PriceChange`. Lopuksi kutsutaan Entity Framework -kontekstin SaveChanges-metodia. ADO.NET-toteutus tapahtuu yksinkertaisesti yhdellä `UPDATE`-lauseella. Tapauksessa kutsutaan `MultiplyPricesByCategoryId`-metodia eri parametreilla 2 000 kertaa.

UPDATE-3, Kategorian tuotteiden siirtäminen toiseen kategoriaan kategoriatunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodissa `UpdateProductCategoryByCategoryId(int categoryId , int newCategoryId)` muutetaan yksittäisen kategorian kaikkien tuotteiden kategoria toiseksi kategoriaksi, jonka tunniste syötetään metodin parametrina. EF Core -toteutuksessa tämä on toteutettu niin, että haetaan kaikki rivit `Product`-taulusta, jossa indeksoidun `CategoryId` sarakkeen arvo vastaa parametrina syötettyä kategoriatunnistetta. Tämän jälkeen EF Core muodostaa tietokannasta haetuista riveistä oliot. Oliot käsitellään listana, ja jokaiselle listan oliolle asetetaan `ProductCategoryId`-kenttään uusi kategoriatunniste. Lopuksi kutsutaan Entity Framework -kontekstin `SaveChanges`-metodia. ADO.NET-toteutus tapahtuu yksinkertaisesti yhdellä `UPDATE`-lauseella. Tapauksessa kutsutaan `UpdateProductCategoryByCategoryId`-metodia eri parametreilla 2 000 kertaa.

6.4.3 DELETE

DELETE-1, Yksittäisen tilausrivin poistaminen tilaukselta tunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodissa `DeleteSalesOrderDetailById(int salesOrderDetailID)` poistetaan yksittäinen tilausrivi tilaukselta. Poistaminen tapahtuu `SalesOrderDetail`-taulusta, eikä mittauskertojen edetessä lisääntyvä rivimäärä aiheuta ylimääräisiä SQL-kyselyitä EF Core -toteutukseen. Tapauksessa kutsutaan `DeleteSalesOrderDetailById`-metodia eri parametreilla 1 000 kertaa.

DELETE-2, Kaikkien asiakkaaseen liittyvien tietojen poistaminen asiakastunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodissa `DeleteCustomerDataByCustomerId(int customerId)` poistetaan asiakkaan tiedot neljästä taulusta: `Customer`, `SalesOrderHeader`, `SalesOrder`, sekä `CustomerAddress`. Näiden taulujen välillä vallitsee viiteavainten asettamia riippuvuuksia, ja poistettavien tietojen määrä kasvaa mittauskertojen myötä taulukon 2 mukaisesti. Tapauksessa kutsutaan `DeleteCustomerDataByCustomerId`-metodia eri parametreilla 150 kertaa.

DELETE-3, Yksittäisen kategorian ja siihen liittyvien tuotteiden poistaminen kategoriatunnisteen perusteella: `IGraduDBOperations`-rajapinnan metodin `DeleteProductCategoryByID(int productCategoryID)` suorituksessa tietokannasta poistetaan paljon tietoa, koska kategorian ja tuotteiden lisäksi viiteavaimet pakottavat poistamaan myös tilausrivejä, joista viitataan `Product`-tauluun. Tämän lisäksi tiedon eheyden säilyttämiseksi poistetaan kaikki rivit `ProductModel`-taulusta, joihin viitataan poistettavista `Product`-riveistä.

Lisäksi `ProductModel`-tauluun viitataan viiteavaimella `ProductModelProductDescription`-välitaulusta, josta on myös toinen viiteavain-viittaus `ProductDescription`-tauluun. Näistä tuotemalleihin ja tuotekuvauksiin liittyvistä tauluista poistetaan ne rivit, jotka liittyvät kategorian poistamisen yhteydessä poistettaviin tuotteisiin. Tietokannan rakenne on kuvattuna liitteessä C, ja tässä kuvattu tietokannan rakenne mahdollistaa tuotekuvausten säilyttämisen tietokannassa eri kielivaihtoehdoille. Poistettavien tietojen määrä kasvaa mittauskertojen myötä taulukon 2 mukaisesti. Tapauksessa kutsutaan `DeleteProductCategoryByID`-metodia eri parametreilla 50 kertaa.

6.4.4 INSERT

INSERT-1, Yksittäisen tuotteen lisääminen: `IGraduDBOperations`-rajapinnan metodissa `InsertProduct(Product p)` lisätään uusi tuote olemassa olevaan kategoriaan ja palautetaan sille syntynyt tietokantatunniste. Tapauksessa kutsutaan `InsertProduct`-metodia eri parametreilla 500 kertaa.

INSERT-2, Tilauksen lisääminen: `IGraduDBOperations`-rajapinnan metodissa `InsertSalesOrderHeader(SalesOrderHeader order)` lisätään kolme tilausriviä sisältävä tilaus tietokantaan olemassa olevalle asiakkaalle. Tapauksessa kutsutaan `InsertSalesOrderHeader`-metodia eri parametreilla 500 kertaa.

INSERT-3, Uuden asiakkaan lisääminen: `IGraduDBOperations`-rajapinnan metodissa `int InsertCustomer(Customer customer)` lisätään uusi asiakas sekä asiakkaan osoite tietokantaan. Tapauksessa kutsutaan `InsertCustomer`-metodia eri parametreilla 500 kertaa.

6.5 Mittauskertojen suorittaminen

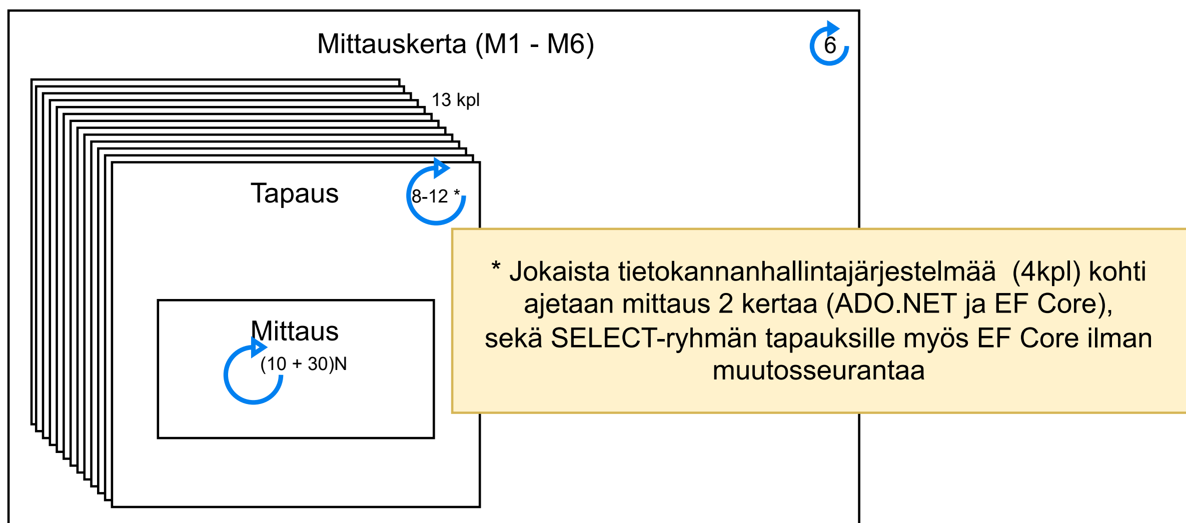
Suorituskykymittaukset suoritettiin aiemmin esitellyllä, Benchmark.net:ia hyödyntävällä mittaussovelluksella. Benchmark.net on .NET-sovelluskehyksessä erilaisten algoritmien suoritusaikojen vertailuun käytetty kirjasto. Liitteessä A on kuvattuna yksinkertainen esimerkkitoteutus kahden järjestämisalgoritmin vertailusta benchmark.net-kirjaston avulla.

Suorituskyvyn vertailussa ei ollut tarkoitus vertailla eri tietokanta-alustojen nopeutta keskenään, vaan verrata saman tietokannan päällä toteutettuja tapauksia EF Core -toteutuksena ja ADO.NET-toteutuksena. Mittausten tuloksena tutkimuksessa saatiin tietoa siitä, kuinka paljon Entity Framework hidastaa tapausten suorittamista. Lisäksi saatiin tapauskohtaisesti tietoa, lisääntyykö EF Coren aiheuttama hidastuminen tietokannan pohjadatan kasvaessa eri mittauskertojen välillä.

Kuvion 5 mukaisesti tutkimuksessa suoritettiin 6 mittauskertaa (M1–M6). Jokaisella mittauskerralla kaikille 13 tapaukselle suoritettiin 8–12 mittausta eri tietokannanhallintajärjestelmän sekä ADO.NET / EF Core yhdistelmällä. Kun yksittäisen tapauksen suorituskykyä mitattiin, toistettiin mittauksessa seuraavat vaiheet jokaiselle tietokanta-alustalle ensin EF Corea hyödyntäen, ja sitten ADO.NET:iä hyödyntäen:

- Lämmittely, jonka aikana toistettiin seuraavat toimenpiteet 10 kertaa:
  - Tietokannan alkutilan palautus.
  - Tapauksen suorittaminen eri parametreilla n kertaa. Tapauskohtainen n on kuvattu taulukossa 1.
- Varsinainen mittaus, jonka aikana toistetaan seuraavat toimenpiteet 30 kertaa:
  - Tietokannan alkutilan palautus.
  - Tapauksen suorittaminen eri parametreilla n kertaa, tallennetaan n:n suorituksen yhteenlaskettu kesto.
- Edellisessä vaiheessa tallennetuista kestoista laskettiin keskiarvo, joka edustaa mittauksen tulosta.

Lämmittelykierroksia sekä tietokannan alkutilan palauutuksia ei laskettu mukaan mittaustuloksiin. Mittausten luotettavuuden kannalta lämmittely oli tarpeen, sillä .NET koodin suori-



Kuvio 5: Mittauskertojen suorittaminen.

tusnopeuksissa voi ilmetä merkittäviäkin eroja riippuen siitä, suoritetaanko käännettyä koodia ensimmäistä kertaa, vai onko se suoritettu jo muutaman kerran. Tämä johtuu siitä, että .NET sovellusta suoritettaessa tapahtuu ensimmäisen ajon aikana niin sanottu *JIT*, eli *just in time* kääntäminen, joka hidastaa suoritusta (Telerik 2013). Koska tutkimuksessa ei haluttu sisällyttää *JIT*-kääntämiseen kuluvaa aikaa mittaustuloksiin, poistettiin sen osuus lisäämällä lämmittelykierrokset osaksi mittauksen valmisteluja.

Edellä kuvattujen toistojen määrän myötä yksittäisen tapauksen mittaaminen ensimmäisellä testikerralla kesti noin 1,5 tuntia tietokannan alkutilojen palautuksineen. Kesto hidastui mittauskertojen edetessä (Taulukon 2 mukaan lisätystä datamäärästä johtuen), mutta korkeahko toistomäärä mahdollisti tutkimuksessa alhaisemman virhemarginaalin ja sitä myötä luotettavampia mittaustuloksia.

6.6 Tutkimusdata

Jokaisella mittauskerralla ennen varsinaisia mittauksia tietokantaan syötettiin edeltävään mittauskertaan nähden kaksinkertainen määrä pohjadataa tauluihin `ProductDescription`, `ProductModelDescription`, `ProductModel`, `Product`, `SalesOrderDetail` ja `SalesOrderHeader`. Datamäärän kasvattamisella mittauskertojen edetessä pyrittiin löytämään tietokannan pohjadatan määrästä aiheutuvia suorituskykyvaikutuksia tutkittaviin

tapauksiin.

Taulukossa 2 kuvataan pohjadatan rivimäärien kasvattaminen eri mittauskerroilla. Mittausten kannalta merkittävin pohjadatan kasvatus tapahtuu tuotteiden määrässä sekä tuotemäärien ja tuotekategorioiden suhteessa. Datamäärien lisäysten johdosta kategoriassa olevien tuotteiden määrä kehittyi mittauskertojen edetessä seuraavasti:

- 1. mittauksessa jokaisessa kategoriassa oli keskimäärin 7,5 tuotetta.
- 2. mittauksessa jokaisessa kategoriassa oli keskimäärin 15 tuotetta.
- 3. mittauksessa jokaisessa kategoriassa oli keskimäärin 30 tuotetta.
- 4. mittauksessa jokaisessa kategoriassa oli keskimäärin 60 tuotetta.
- 5. mittauksessa jokaisessa kategoriassa oli keskimäärin 120 tuotetta.
- 6. mittauksessa jokaisessa kategoriassa oli keskimäärin 240 tuotetta.

Taulukko 2: Pohjadatan määrät eri mittauskerroilla (tuhatta riviä).

Taulu	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**
Address	5	5	5	5	5	5
CustomerAddress	5	5	5	5	5	5
Customer	5	5	5	5	5	5
ProductCategory	0,5	0,5	0,5	0,5	0,5	0,5
ProductDescription	2,5	5	10	20	40	80
ProductModelProductDescription	2,5	5	10	20	40	80
ProductModel	1,25	2,5	5	10	20	40
Product	1,25	2,5	5	10	20	40
SalesOrderDetail	3,75	7,5	15	30	60	120
SalesOrderHeader	1,25	2,5	5	10	20	40

7 Tutkimuksen tulokset.

Mittaukset suoritettiin kuutena eri mittauskertana taulukon 2 mukaisilla pohjadatan määrillä. Mittauskertojen aloitusajankohdat olivat seuraavat:

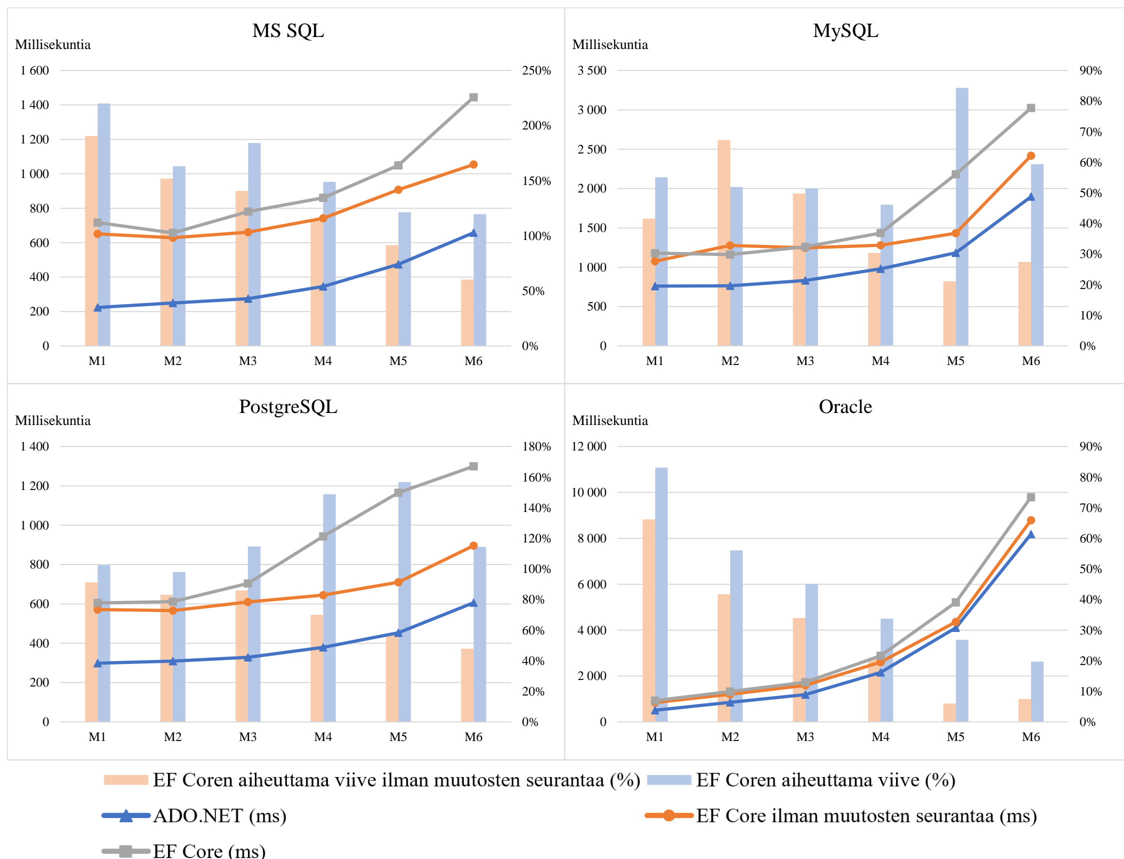
- Mittaus 1: 28.1.2023 klo 20:04.
- Mittaus 2: 29.1.2023 klo 12:57.
- Mittaus 3: 29.1.2023 klo 17:57.
- Mittaus 4: 29.1.2023 klo 21:20.
- Mittaus 5: 30.1.2023 klo 13:45.
- Mittaus 6: 4.2.2023 klo 11:30.

Ennen jokaista mittauskertaa tutkimusympäristöstä sammutettiin tutkimuksen kannalta tarpeettomat sovellukset ja prosessit. Lisäksi käyttöjärjestelmän virransäästöasetuksia säädettiin niin, että tietokone ei mennyt mittausten aikana lepotilaan. Tässä luvussa käydään läpi mittausten tuloksia tapausten ja ryhmien tasolla. Kuvioiden kaavioissa viitataan eri mittauskertoihin tunnisteilla M1–M6. Luvun kaavioissa pylväät kuvaavat EF Coren aiheuttamaa suhteellista viivettä prosentteina, kun taas viivakaaviot kuvaavat absoluuttista aikaa millisekunneissa, joka mittauksissa kului kunkin tapauksen suorittamiseen keskimäärin. Mittauspöytäkirjoissa (Liitteet D–I) on taulukoituna jokaisen tapauksen tulokset jokaiselle eri mittauskerralle millisekunnin kymmenyksen tarkkuudella.

7.1 SELECT

SELECT-1, Tuotetietojen hakeminen kategoriatunnisteen perusteella: Tutkimuksessa havaittiin, että EF Corella toteutettuna tapauksen suorittaminen kestää noin 19–220 % kauemmin verrattuna ADO.NET:llä toteutettuun tapaukseen. Lisäksi havaittiin, että sekä Oracle- että MS SQL -tietokantojen osalta EF Coren aiheuttama suhteellinen viive laskee selkeästi mittauskertojen edetessä ja datamäärien kasvaessa. PostgreSQL- ja MySQL-tietokannoissa suhteellinen viive ei laskenut mittauskertojen edetessä. PostgreSQL:n osalta EF Coren aiheuttama suhteellinen viive kasvoi aina viidenteen mittauskertaan asti, kunnes kuudennella mittauskerralla se oli selkeästi viidettä mittauskertaa alemmalla tasolla.

MySQL tietokannassa EF Coren aiheuttama suhteellinen viive pysyi tasaisesti noin 50 prosentin tuntumassa aina mittauskertaan viisi asti, missä viive oli noin 85 prosenttia. Kuudennella mittauskerralla MySQL-tietokannassa EF Coren aiheuttama viive oli pienempi kuin viidennellä mittauskerralla, enää noin 59 %.

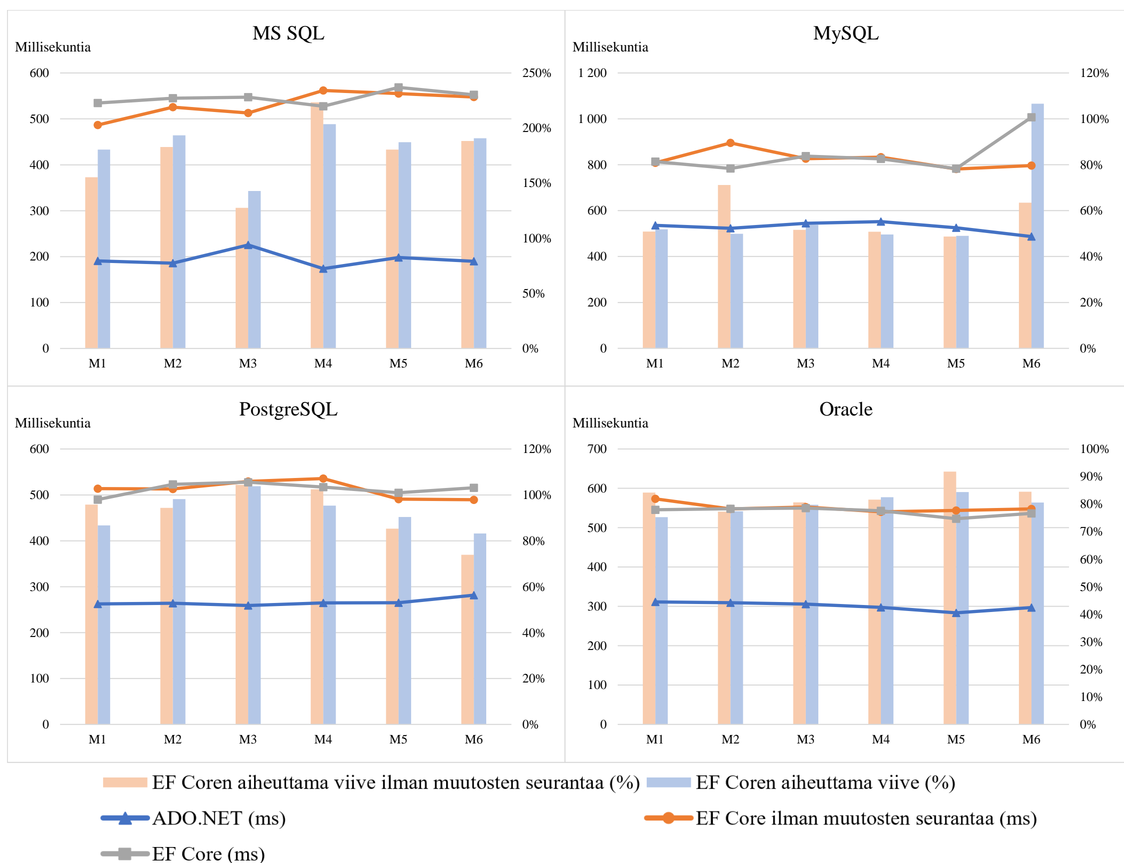


Kuvio 6: SELECT-1, mittaustulokset viiva- ja pylväskaavioissa.

Tapausta tutkittaessa mitattiin myös suorituksen kesto sellaisella toteutuksella, jossa on käytetty EF Corea ilman muutosten seurantaa. Kuten kuviosta 6 näkee, ilman muutosten seurantaa toteutetun EF Core -toteutuksen ja ADO.NET:in välillä mittaustulokset ovat johdonmukaisempia kaikilla tietokanta-alustoilla. EF Coren aiheuttama prosentuaalinen suoritusajan hidastuminen pienenee tasaisesti mittauskertojen edetessä ja datamäärien kasvaessa. Ainoa poikkeus edellä mainittuun on MySQL-tietokanta, jonka tulokset poikkeavat hieman yleisestä linjasta (laskeva trendi) mittauskerroilla M1 ja M6.

Vaikka ilmiö oli selkeimmillään vasta kun EF Core -toteutuksesta poistettiin muutosten seuranta, tässä tapauksessa voidaan kuitenkin todeta prosentuaalisten erojen keskimäärin pienentyvän EF Coren ja ADO.NET:in välillä sitä mukaa kuin datamäärät kasvavat. Tapauksen suorittamisen kesto millisekunneissa kasvoi tasaisesti mittauskerrasta toiseen sekä ADO.NET- että EF Core -toteutuksissa.

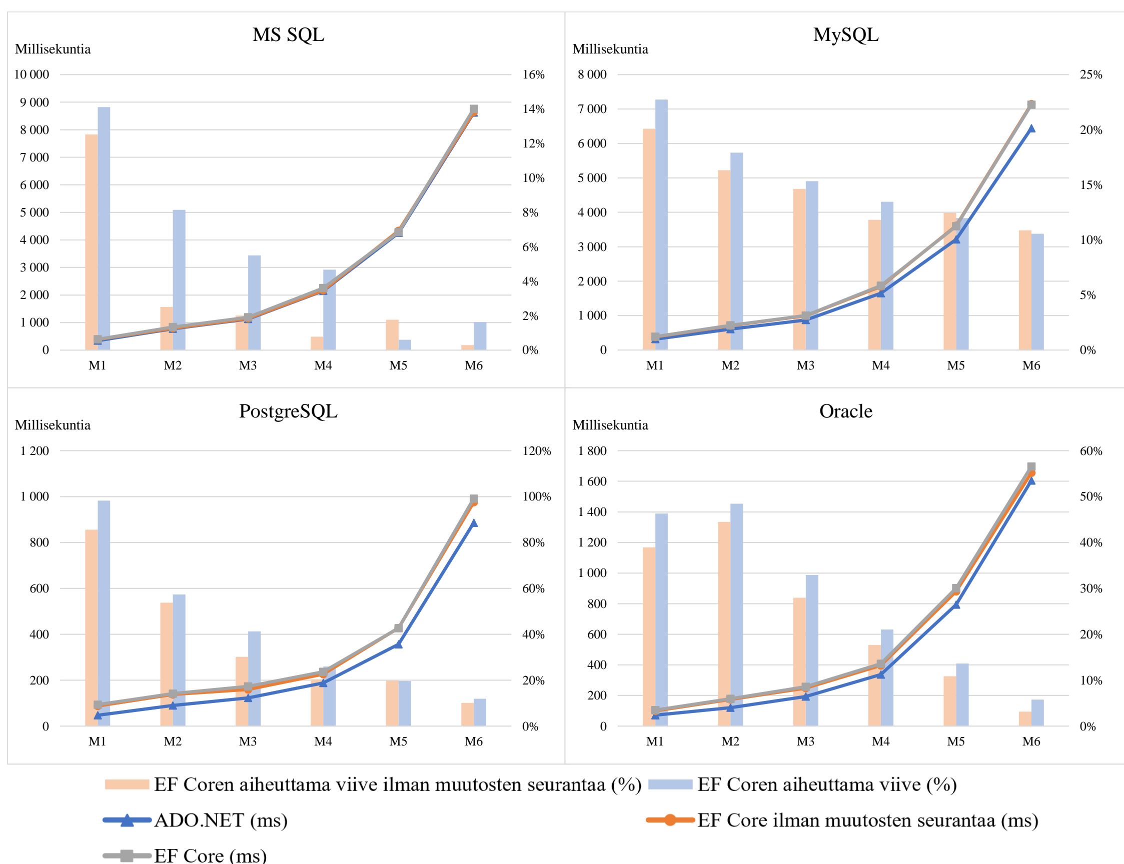
SELECT-2, Tuotteiden lukumäärän laskeminen tuotetietojen perusteella: Tutkimuksessa havaittiin, että tapauksen suorittaminen ei hidastunut millään tietokanta-alustalla mittauskertojen edetessä ja datamäärien kasvaessa. Myös EF Coren aiheuttama suorituskykyvaikutus pysyi tietokantakohtaisesti tasaisena eri mittauskertojen välillä, poikkeuksena tähän on MS SQL-tietokanta, jonka EF Core -toteutuksessa prosentuaalinen hidastuminen (noin 143 %) oli mittauskerralla kolme selkeästi muiden mittauskertojen hidastumisprosentteja (noin 180–203 %) alhaisemmalla tasolla.



Kuvio 7: SELECT-2, Pohjadatan määrä ei hidastanut tapauksen suoritusta merkittävästi.

Pieni poikkeama tuloksissa huomataan myös MySQL-tietokannan osalta, jossa mittauskerralla 6 EF Coren aiheuttama suhteellinen viive saavutti 106 % muiden mittauskertojen viiveen ollessa tiiviisti välillä 49–54 %. Kun tarkastellaan kaikkien tietokanta-alustojen keskiarvoa, EF Coren aiheuttama suhteellinen viive tapauksen **SELECT-2** osalta on jokaisella mittauskerralla noin 100 %, ja karkeasti voidaan todeta, että lisääntyvä datamäärä ei vaikuta keskiarvoon. Tasainen suorituskyky eri datamäärillä näkyy hyvin lähes vaakasuorina viivadiagrammeina kuviossa 7.

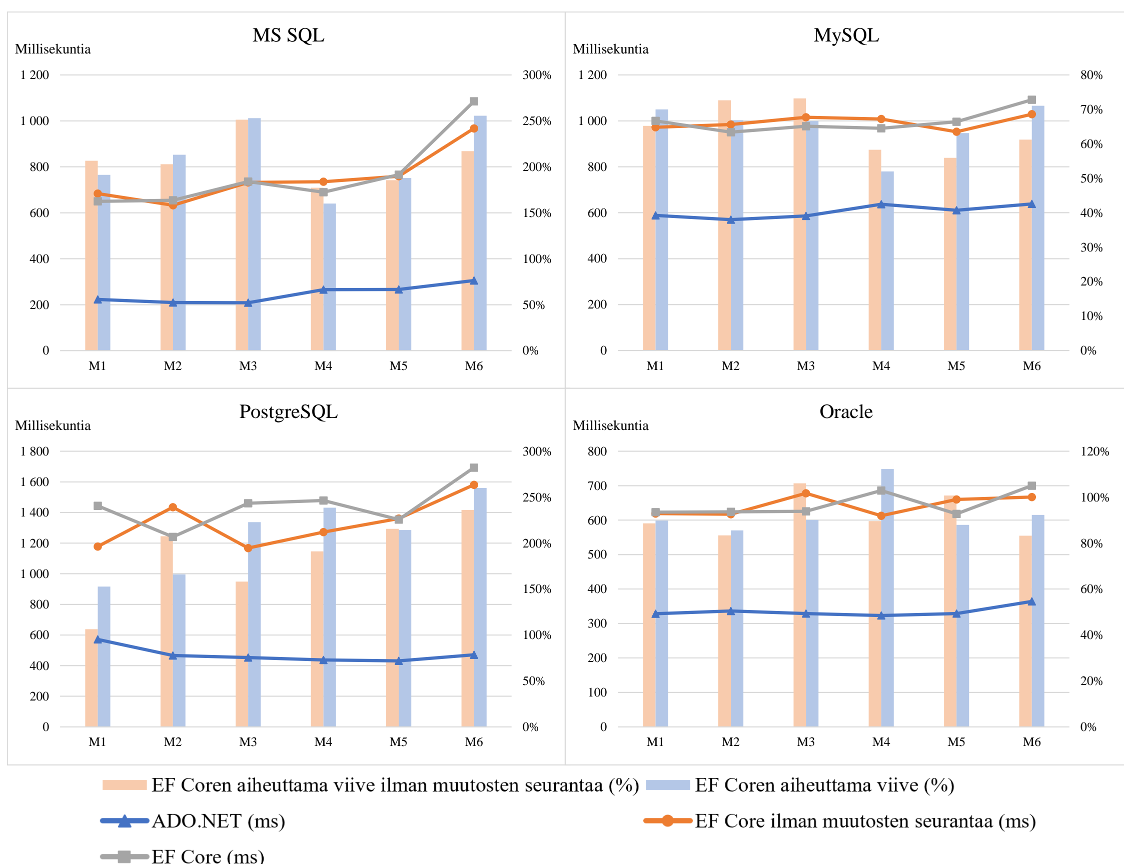
SELECT-3, Tuotteiden hakeminen merkkijonon perusteella: EF Coren aiheuttama suhteellinen viive vaihteli tässä tapauksessa 98,26 prosentista (PostgreSQL, ensimmäinen mittauskerta) 0,6 prosenttiin (MS SQL) pienentyen aina mittauskertojen edetessä. Mittaustulosten perusteella voidaan melko luotettavasti todeta, että suurempi datamäärä johtaa tapauksessa **SELECT-3** EF Coren aiheuttaman viiveen pienenemiseen.



Kuvio 8: SELECT-3, tapausten suoritusaika kasvaa ja EF Coren aiheuttama suhteellinen viive pienenee mittauskertojen edetessä.

Edellä mainittu mittauskertojen välinen laskeva trendi toistuu myös, kun vertailuun otetaan EF Core ilman muutosten seurantaa. Muutosten seurannan poistaminen nopeuttaa tapauksen suorittamista hieman, joten viiveprosentti on normaaliin EF Core -toteutukseen verrattuna pienempi. Kuvion 8 palkkikaavioista voi havaita, kuinka EF Coren aiheuttama viiveprosentti lähestyy nollaa mittauskertojen edetessä ja datamäärien kasvaessa.

SELECT-4, Asiakkaan tekemien ostosten kokonaissumman laskeminen asiakastunnisteen perusteella: Tutkimuksessa havaittiin, että EF Coren hidastava vaikutus on prosentuaalisesti noin 50 % ja 250 % välillä. Toisin kuin esimerkiksi tapauksessa **SELECT-3**, **SELECT-4**:n kohdalla eri mittauskertojen välinen trendi vaihtelee voimakkaasti tietokannasta riippuen, ja erot eri tietokantojen välillä vaikuttavat epäjohdonmukaiselta, kuten kuviossa 9 esitetään.

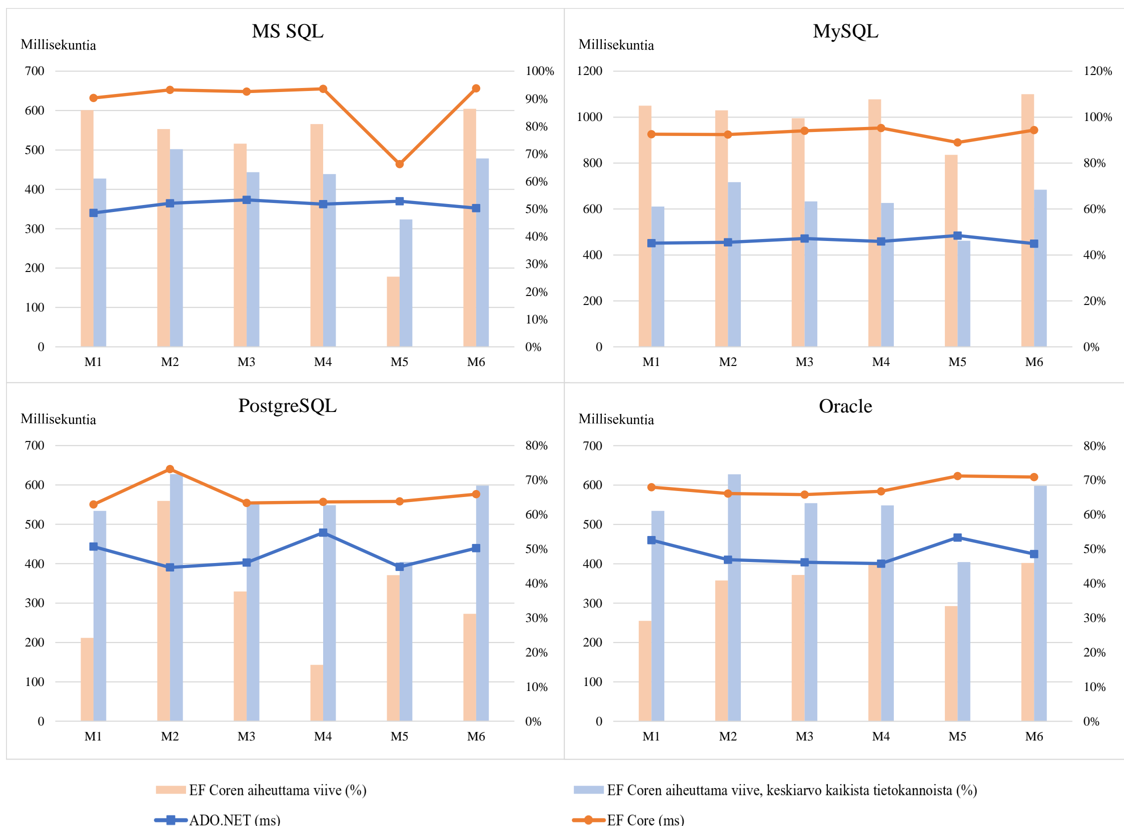


Kuvio 9: SELECT-4, suhteellinen hidastuminen vaihtelee paljon eri tietokantojen välillä.

Kun hidastumista tarkastellaan prosenttien sijaan millisekunneissa, voidaan todeta että EF Coren aiheuttama viive pysyy PostgreSQL-tietokantaa lukuun ottamatta lähes tasaisena (noin 300–500 millisekuntia) aina kuudenteen mittauskertaan saakka, kunnes muidenkin tietokanta-alustojen välilä syntyy enemmän hajontaa.

7.2 UPDATE

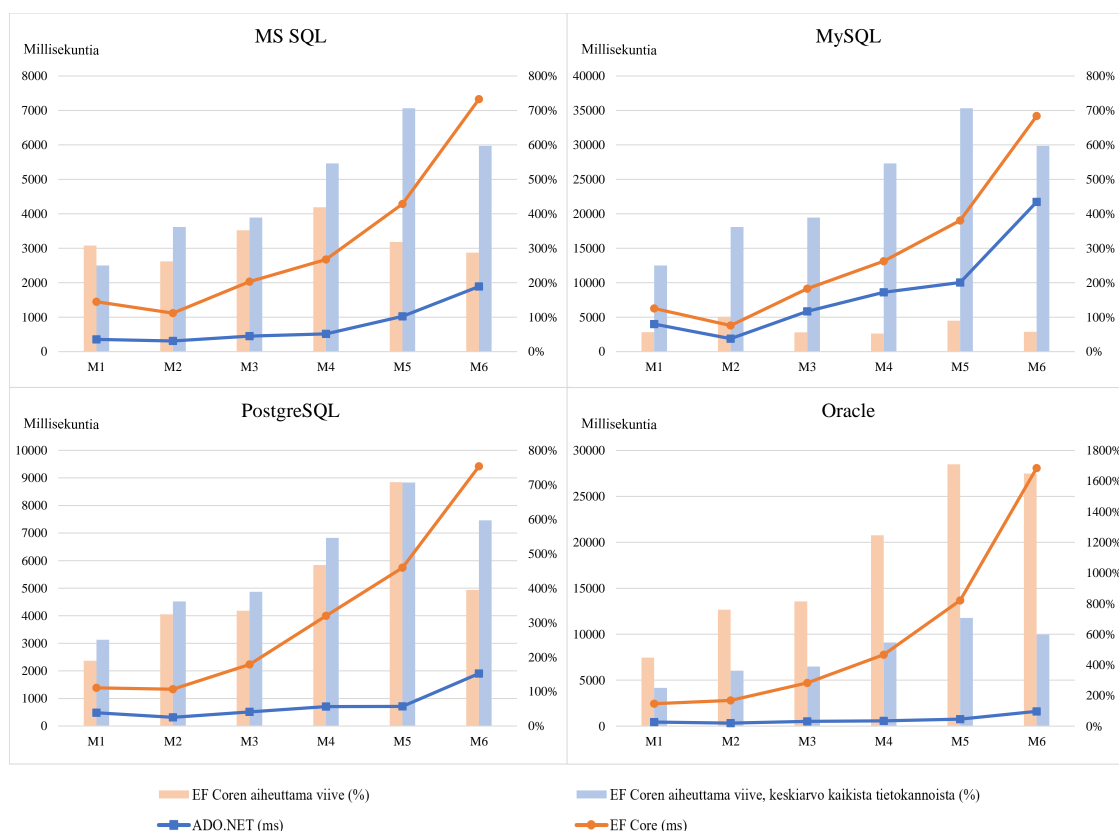
UPDATE-1, Asiakkaan sukunimen päivittäminen: Tapauksen suorittaminen kesti EF Core -toteutuksessa noin 25–100 % kauemmin verrattuna ADO.NET-toteutukseen. Eri tietokantojen välillä hajontaa oli melko paljon. Hidastuminen oli prosentuaalisesti vähäisintä Oraclen tietokannassa (noin 29–46 %), ja suurinta MySQL tietokannassa (83–109 %). MS SQL ja PostgreSQL-tietokantojen osalta hajontaa prosentuaalisessa hidastumisessa oli enemmän, eikä se myöskään kasvanut tai vähentynyt mittauskertojen edetessä ja datamäärien kasvaessa.



Kuvio 10: UPDATE-1, EF Coren aiheuttama viive oli keskimäärin noin 50–75 %.

MS SQL:n osalta prosentuaalinen hidastuminen on viidennellä mittauskerralla vain 25,44 %, kun se muilla mittauskerroilla oli välillä 73–86,4 %. Kuvion 10 viivadiagrammissa tämä näkyy selkeänä poikkeamana yleisestä linjasta. PostgreSQL:n osalta hajontaa on myös melko paljon hidastumisprosentin kasvaessa ensin ensimmäisen mittauskerran lukemasta 24,17 % toisen mittauskerran 63,96 %, minkä jälkeen seuraavissa mittauksissa hidastumisprosentti oli 37,67, 16,33, 42,38 ja 31,15. Tutkimuksessa ei havaittu merkittävää hidastumista tapauksen suorituksen keston suhteen datamäärien kasvaessa.

UPDATE-2, Asiakkaan sukunimen päivittäminen ja UPDATE-3, Kategorian tuotteiden siirtäminen toiseen kategoriaan kategoriatunnisteen perusteella: Mittaustulosten perusteella nämä kaksi tapausta ovat niin samankaltaiset, että niiden tulokset esitellään yhdessä.



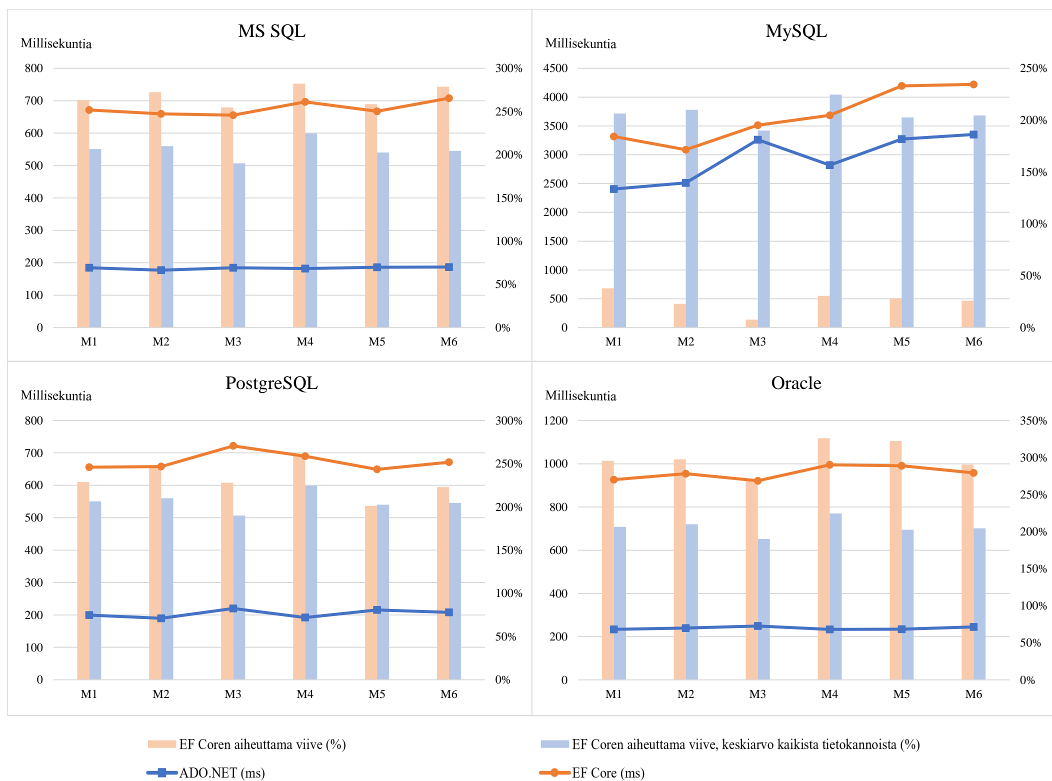
Kuvio 11: UPDATE-2, EF Core -toteutuksen suoritusajat kasvavat merkittävästi mittauskertojen edetessä.

Molemmissa tapauksissa suorituksen kesto kasvaa mittauskertojen edetessä ja datamäärien kasvaessa. Molemmissa tapauksissa Oraclen tietokantaa vasten suoritetut mittaukset osoittavat, että EF Coren aiheuttama prosentuaalinen hidastuminen kasvaa erittäin voimakkaasti eri mittauskertojen välillä verrattuna ADO.NET:iin ollen ensimmäisellä mittauskerralla noin 450 %, ja viidennellä mittauskerralla jopa noin 1710 %. Muiden tietokantojen osalta prosentuaalinen hidastuminen on suhteessa maltillisempaa: vähäisintä MySQL-tietokannassa (n. 56–101 %). MS SQL:n ja PostgreSQL:n osalta EF Coren aiheuttama hidastumisprosentti oli noin 300 % ja 700 % välillä, kasvaen tasaisesti mittauskertojen edetessä ja datamäärien kasvaessa.

7.3 DELETE

DELETE-1, Yksittäisen tilausrivin poistaminen tilaukselta tunnisteen perusteella: Mittaustulosten mukaan tämän tapauksen suorittaminen EF Corella tai ADO.NET:llä ei hidastunut mittauskertojen edetessä millään tietokanta-alustalla. Prosentuaalisessa hidastumisessa oli eroja tietokanta-alustojen välillä, mutta kuten kuviossa 12 esitetään, yksittäisen tietokanta-alustan osalta tulokset olivat melko tasaisia mittauskertojen edetessä ja datamäärien lisääntyessä. Prosentuaalisesti vähiten (mutta millisekunteina eniten) EF Coren aiheuttamaa hidastumista havaittiin MySQL-tietokannassa, hidastumisprosentin vaihdellessa 7,7 % ja 37,96 % välillä.

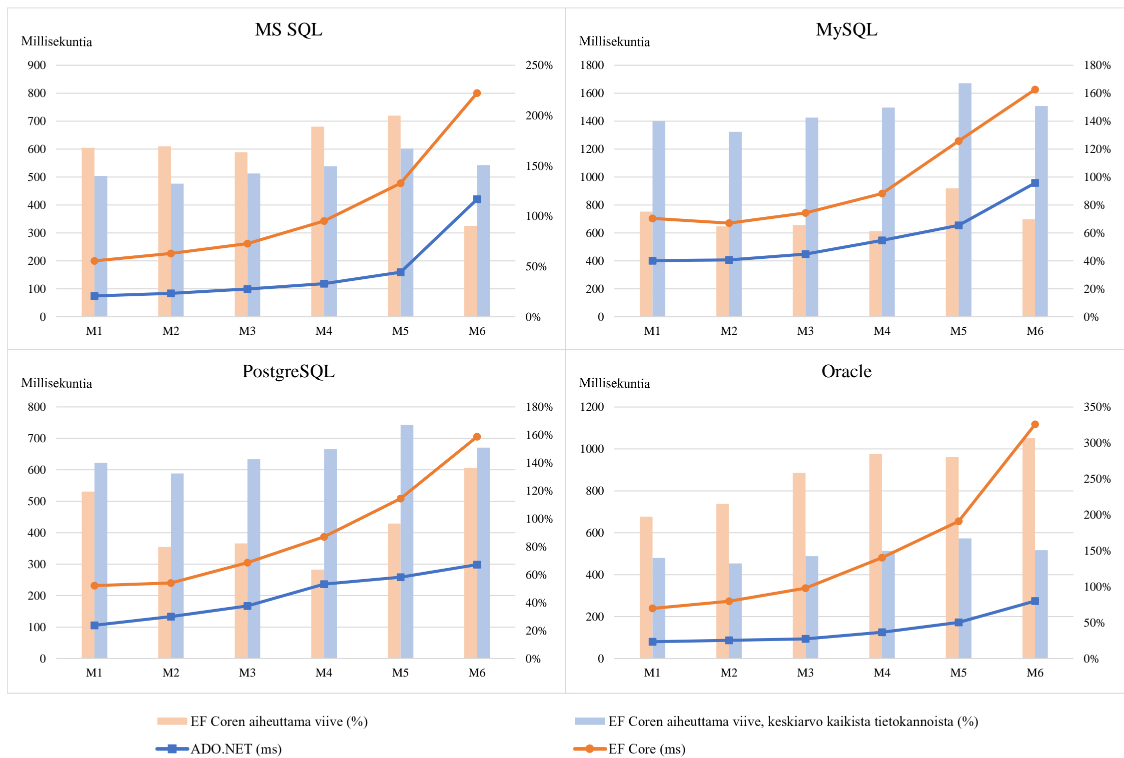
Suurin hidastumisprosentti oli Oraclen tietokannassa, jossa hidastumista tapahtui eri mittauskerroilla 269,82–326,03 %. Myös EF Coren aiheuttama viive millisekunteina oli melko tasaista. Millisekunteina ilmaistuna viive oli MS SQL- sekä PostgreSQL -tietokannoissa hieman alle 500 millisekuntia, Oraclen tietokannassa noin 672–761 millisekuntia. MySQL:n kanssa havaittu EF Coren hidastava vaikutus oli millisekunneissa mitattuna vaihtelevampaa, sen vaihdellessa välillä 253,3–923,5 ms. Keskimäärin voidaan todeta, että EF Core hidastaa tapausta DELETE-1 noin 200 prosenttia verrattuna ADO.NET toteutukseen.



Kuvio 12: DELETE-1, datamäärien kasvaminen ei vaikuttanut suorituskykyyn.

DELETE-2, Kaikkien asiakkaaseen liittyvien tietojen poistaminen asiakastunnisteen perusteella: Jos katsotaan eri tietokantojen keskiarvoa, on EF Coren aiheuttama suhteellinen hidastuminen mittausten mukaan n. 150 prosenttia. Suhteellinen hidastuminen oli vähäisintä MySQL-tietokannassa, jossa EF Corella tehty toteutus toimi 64,76–91,95 % hitaammin kuin saman tapauksen ADO.NET-toteutus. Oracle-tietokannassa EF Coren aiheuttama hidastuminen oli prosentuaalisesti suurin, noin 200–306 %. Lisäksi muista tietokannoista poiketen hidastumisprosentti kasvoi tasaisesti mittauskertojen edetessä ja datamäärän kasvaessa.

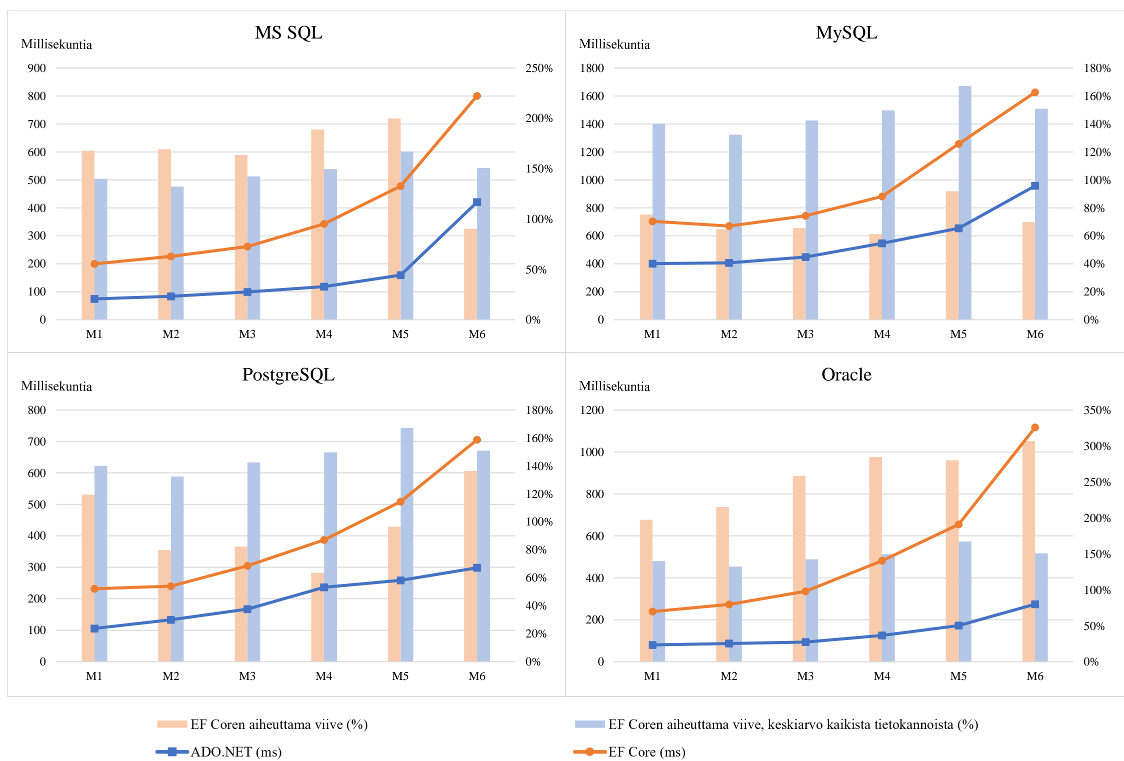
Suhteellinen hidastuminen jakautuu MS SQL -tietokannan osalta muutoin melko tasaisesti noin 50 prosenttiyksikön sisään vaihteluvälille 163,60–199,87 %, mutta kuudennella mittauskerralla prosentuaalinen hidastuminen oli poikkeuksellisesti vain 90,35 %. Tähän prosentuaalisesti alhaisempaan hidastumiseen syynä on ADO.NET-toteutuksen suhteellisen voimakas hidastuminen viidenteen mittauskertaan nähden (159,4 vs. 420,7 millisekuntia) tuntemattomasta syystä.



Kuvio 13: DELETE-2, EF Core hidastaa suoritusta noin 130–170 %.

Kaikissa tämän tapauksen mittauksissa suoritusajat hidastuivat mittauskertojen edetessä, joka näkyy kuvion 13 viivadiagrammeista. Keskimäärin tämän tapauksen osalta voidaan sanoa, että EF Core hidastaa tapauksen suorittamista noin 130–170 %, ja karkeasti tarkasteltuna EF Coren aiheuttama suhteellinen hidastuminen ei vaikuta kasvavan eikä pienenevän datamäärän kasvaessa.

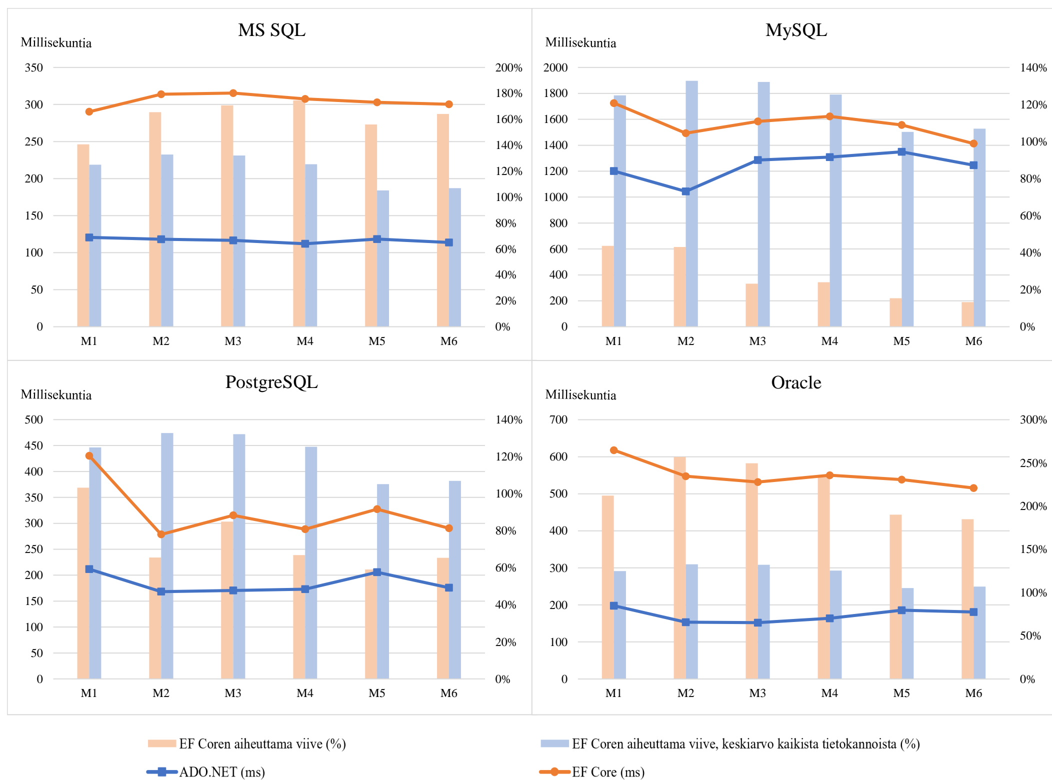
DELETE-3, Yksittäisen kategorian, ja siihen liittyvien tuotteiden poistaminen kategoriatunnisteen perusteella; Mittaustulosten mukaan suoritusaika pitenee kaikilla tietokanta-alustoilla mittauskertojen edetessä riippumatta siitä onko toteutustapana ADO.NET vai EF Core. PostgreSQL- ja Oracle-tietokannoissa EF Core -toteutus hidastuu huomattavasti enemmän verrattuna ADO.NET-toteutukseen, kun taas MS SQL- ja MySQL-tietokannoissa hidastumista kuvaavat kuvaajat seuraavat toisiaan melko tarkasti, kuten kuviossa 14 esitetään. Tämä johtaa suhteellista hidastumista tarkastellessa siihen, että MS SQL- sekä MySQL-tietokannoissa suhteellinen hidastuminen ei selkeästi kasva mittauskerrasta toiseen edetessä ja datamäärien kasvaessa.



Kuvio 14: DELETE-3, tuloksia koottuna.

7.4 INSERT

INSERT-1, Tuotteen lisääminen, INSERT-2, Tilauksen lisääminen ja INSERT-3, Asiakkaan lisääminen: Kaikissa INSERT-ryhmään kuuluvissa tapauksissa tulokset olivat niin saman kaltaisia, että ne käsitellään yhdessä. Mittausten perusteella EF Coren voidaan sanoa hidastavan INSERT-ryhmän tapauksia keskimäärin noin 80–120 %, riippuen hieman tapauksesta. Jokaisessa tapauksessa MySQL- ja PostgreSQL-tietokantoja vasten toteutetut EF Core -toteutukset hidastivat tapauksen suorittamista reilusti keskimääräisiä EF Core -toteutuksia vähemmän. Oraclen tietokantaa vasten toteutettu EF Core -toteutus oli hitaimmillaan jopa 261 % hitaampi verrattuna Oracle-tietokannan ADO.NET-toteutukseen.



Kuvio 15: INSERT-1, EF Coren aiheuttamassa viiveessä oli hajontaa eri tietokantojen välillä.

INSERT-ryhmän tapausten mittaustulosten perusteella voidaan sanoa, että datamäärän lisääminen ei hidasta tapauksen suoritusta ADO.NET- tai EF Core -toteutuksella. Lisäksi myös EF Coren aiheuttama hidastusprosentti pysyy tietokantakohtaisesti melko samalla tasolla mittauksesta toiseen, mutta eri tietokantojen välillä hajontaa on melko paljon, joka näkyy kuviossa 15.

8 Pohdinta

Edellisessä luvussa esiteltiin mittaustulokset sellaisenaan. Tässä luvussa syvennytään mittaustuloksiin tarkemmin ja vastataan niiden perusteella tutkimuskysymykseen sekä pohditaan edellisessä luvussa esiteltyjä tuloksia laajemmin. Lopuksi arvioidaan tutkimuksen luotettavuutta.

8.1 Tutkimuskysymykseen vastaaminen

Yleisesti ottaen voidaan todeta, että EF Coren aiheuttamat suorituskykyvaikutukset ovat ainakin prosentuaalisesti merkittäviä kaikilla tietokanta-alustoilla. Hajontaa oli paljon eri tapausten ja tietokanta-alustojen välillä, mutta kaikissa tapaukissa EF Core aiheutti merkittävää suhteellista hidastumista. Keskimäärin EF Coren aiheuttama suorituskykyvaikutus oli tapauksesta riippuen noin 100–200 %, mutta äärimmäisimmissä tapauksissa suhteellinen hidastuminen oli reilusti yli 1000 %. EF Coren aiheuttama hidastuminen oli kaikissa tapauksissa ja jokaisella mittauskerralla suhteellisesti pienintä MySQL-tietokannassa, mutta pieni prosentuaalinen luku selittyy sillä, että MySQL-tietokantaa vasten tapausten suorittaminen oli aina hitainta.

Tästä voi johtaa laajemman johtopäätöksen: Mitä hitaampi on *tietokannassa* tapahtuva suoritus, sitä pienempi on EF Coren aiheuttama *suhteellinen* suorituskykyvaikutus ADO.NET-toteutukseen verrattuna. Tästä esimerkkinä tapaus **SELECT-3**, jossa haetaan ne tuotteet, joiden nimessä esiintyy hakusanana syötetty merkkijono. Koska `Product`-taulun `Name` kenttää ei ole indeksoitu merkkijonohakua varten, on tietokannassa tapahtuva haku jo sellaisenaan melko hidas, ja se hidastuu selkeästi `Product`-taulun rivimäärien kasvaessa. Tapauksen suorituksen muut osat eivät hidastu läheskään yhtä paljon, minkä seurauksena kokonaisuuksien suoritusaika kasvaa sekä EF Corella että ADO.NET:llä toteutettuna, ja EF Coren suhteellinen vaikutus jää melko pieneksi. Tämä ei kuitenkaan tarkoita, että EF Coren käyttö tällaisessa tapauksessa olisi ongelmatonta, vaan pikemminkin sitä, että sujuvan kokonaisuuden toteuttamiseksi tietokantaan tulisi lisätä merkkijonohakua tukevaa indeksointia.

Vaikka tietokantoja ei ollut tarkoitus vertailla keskenään, mainittakoon myös, että tietokannoista nopeiten suoriutui PostgreSQL. Tämä on tulosten tulkinnassa oleellista sen takia, että kääntäen aiemmin mainittuun MySQL-tietokannan hitauteen, nopea PostgreSQL-tietokanta aiheutti mittauksissa sen, että sitä vasten suoritetuissa tapauksissa EF Coren aiheuttama suhteellinen viive oli aina keskiarvoa suurempi.

8.2 Tulosten merkitys

Kun datamäärä kasvoi, osassa tapauksista suoritusajat hidastuvat millisekunneissa mitattuna paljon nopeammin EF Core -toteutuksissa verrattuna ADO.NET-toteutukseen, mikä näkyy esimerkiksi kuviossa 11 toisistaan erkanevina kuvaajina viivadiagrammeissa. Ilmiö johtuu siitä, että näissä tapauksissa mittausten tietokannan rivimäärän kasvaminen aiheuttaa suhteessa enemmän työtä EF Corelle verrattuna ADO.NET-toteutukseen. Selkeintä tämä on tapauksissa **UPDATE-2** ja **UPDATE-3**, mutta sama ilmenee myös tapauksissa **DELETE-2** ja **DELETE-3**. Tämä johtuu siitä, että ADO.NET-toteutuksiin verrattuna EF Core suorittaa yksittäisiä tietokantakyselyitä paljon enemmän. Ilmiö on kuvattuna aliluvussa 2.3 sekä koodiesimerkissä 4. Käytännössä hidastumisen kasvu johtuu näissä tapauksissa siitä, että sekä tuotteiden lukumäärä suhteessa tuotekategorioihin että tilausrivien määrä suhteessa tilauksiin kasvaa mittauskertojen edetessä. Tästä nousee esiin huomio, että joskus myös sisällön tuotannolla (kuinka monta tuotetta per kategoria), voi olla merkitystä tietokantapohjaisen sovelluksen suorituskykyyn.

Noin puolet käsiteltävistä tapauksista (kaikki INSERT-ryhmän tapaukset, **SELECT-2**, **SELECT-4** ja **DELETE-1**) olivat sellaisia, joissa suoritusajat eivät kasvaneet merkittävästi ADO.NET- tai EF Core -toteutuksissa tietokannan rivimäärien kasvaessa. Nämä kaikki olivat tapauksia, joissa tietokantaan toteutettiin sama määrä kyselyitä ja sieltä palautettiin sama tietomäärä pohjadatan rivimäärästä riippumatta. Vaikka EF Core/ADO.NET-toteutukset eivät näissä merkittävästi hidastuneet mittausten edetessä, oli EF Coren aiheuttama hidastusvaikutus merkittävä (noin 50–250 %, tietokannasta ja tapauksesta riippuen).

Tulokset ovat linjassa Östmanin (Östman 2020) sekä Wiphusitphunpol ja Lertrusdachakul (2017) mittaustulosten kanssa, mutta edellä mainittuihin verrattuna tässä tutkimuksessa tutkittiin muitakin kuin SELECT-ryhmän tapauksia.

8.3 Tutkimuksen luotettavuuden arviointi

Tutkielmassa on vertailtu Entity Framework Coren toimintaa neljään eri tietokantaan eri tuottajien läpi, joten tulokset ovat melko hyvin yleistettäviä eri tietokantojen välillä. Jokainen ORM-kehys on kuitenkin hieman erilainen, joten tutkimuksen tuloksia ei sellaisenaan pitäisi yleistää koskemaan kaikkia ORM-kehyksiä. Lisäksi ADO.NET-toteutusten osalta laatu ja suorituskyky riippuvat aina myös sovelluskehittäjän osaamistasosta, koska ADO.NET-toteutuksissa ohjelmistokehittäjän tulee itse toteuttaa tietokantakyselyt, toisin kuin EF Core -toteutuksissa.

Pienimpänä mitattavana yksikkönä käytettiin metodin suorittamista n kertaa eri parametreilla (tapauskohtainen n on kuvattu taulukossa 1), koska pienemmillä datamäärillä yksittäinen metodin suoritus olisi kestänyt vain joitakin millisekunteja eikä tämä olisi ollut kovin luotettava mittausten tarkkuuden kannalta. Lisäksi, kun metodi suoritettiin n kertaa *eri* parametreilla, pyrittiin samalla varmistamaan, etteivät EF Coren tai eri tietokannanhallintajärjestelmien sisäiset välimuistit vaikuttaneet ratkaisevasti mittaustuloksiin.

Tutkimuksessa käytetyt ohjelmistoversiot on listattu tekstissä, joten tutkimus voidaan tarvittaessa toistaa tulosten varmistamiseksi. Lisäksi luvussa 6 esitelty tutkimussovellus löytyy GitHub palvelusta (Valkonen 2023), joten sen toistaminen eri tutkimusympäristössä pitäisi olla melko vaivatonta.

9 Johtopäätökset

Tutkimuksessa esiteltiin ORM-malleja käymällä läpi niiden ominaisuuksia, heikkouksia ja vahvuuksia yleisellä tasolla. ORM-kehysten yleisesittelyn jälkeen esiteltiin lukijalle tietokantaohjelmoinnin kehitystä .NET-ympäristöissä 1990-luvulta nykyhetkeen. Teknisen pohjustuksen lisäksi työssä tutustuttiin tarkemmin EF Coreen, yhteen ORM-kehyksistä muun muassa koodiesimerkkien kautta. Teoriaosuuden jälkeen käytiin läpi tutkimuksen lähtökohtia, esiteltiin tutkimuskysymys, ja kuvattiin tutkimusstrategiaksi valikoitunut tapaustutkimus yleisellä tasolla.

Luvussa 6 kuvailtiin tutkimuksen toteutus: Mitä tapauksia tutkittiin EF Coren aiheuttaman suorituskykyvaikutuksen selvittämiseksi eri tietokanta-alustoja vasten? Millainen sovellus tutkimusta varten kehitettiin, millaisessa ympäristössä ja miten tulokset kerättiin? Luvussa 7 esiteltiin, kuinka EF Core vaikuttaa suoritusaikoihin tutkimusta varten valituissa tapauksissa: vaikka EF Coren suhteellinen vaikutus vaihteli eri tietokannanhallintajärjestelmien ja tapausten välillä, voidaan johtopäätöksenä todeta, että EF Core aiheuttaa pääosin merkittävä suhteellista hidastumista kaikissa tutkimuksessa käsitellyissä tapauksissa. Lisäksi tutkimuksessa tunnistettiin tapauksia, joissa tietokannan rivimäärien kasvaminen lisää EF Coren aiheuttamia suorituskykyvaikutuksia merkittävästi.

Tulosten valossa voidaan yleisesti todeta, että ORM-kehyksen käyttö hidastaa aina tietojärjestelmän ajonaikaista toimintaa. Joskus prosentuaalisesti suuri hidastuminen, (esim. yksi vs. kaksi millisekuntia) on järjestelmän vaatimusten tai käyttökokemuksen kannalta merkityksetön (esimerkiksi pienet järjestelmät tai demot). Suurempien tietomassojen käsittelyssä ORM-kehysten käytössä tulee kuitenkin todennäköisesti eteen tilanteita, joissa ORM-kehyksen aiheuttamasta viiveestä on konkreettista haittaa sovelluksen käyttäjälle. Pilvipalveluissa ORM-kehysten aiheuttamaa hidastusta pystytään kompensoimaan skaalaamalla tietokantaa tehokkaammaksi, mutta se lisää sovelluksen käyttökustannuksia. Kiinnostava tutkimusaihe voisi olla laskea ORM-kehyksen käytön kustannusvaikutuksia verrattuna ADO.NET:iin eri pilvipalveluntarjoajien tietokannoissa.

Tietojärjestelmän kehittäminen on usein tasapainoilua käytettävissä olevien resurssien sekä riittävän laadun välillä, joten tutkimuksen tuloksista huolimatta ORM-kehyksen käyttö voi olla perusteltua ohjelmistoprojekteissa. Niiden hyvät puolet nopeuttavat ohjelmistokehitysprosessia ja oletettavasti laskevat sovelluskehitykseen käytettyjen työtuntien määrää sovelluksen elinkaaren eri vaiheissa. Eräs jatkotutkimuksen aihe voisi olla, kuinka paljon ORM-kehysten käyttö lopulta nopeuttaa sovelluskehitysprosessia. Kaikkien ohjelmistokehitykseen osallistuvien tulisi kuitenkin tiedostaa ORM-kehyksen hidastava vaikutus, ja järjestelmien suunnittelussa mahdollistaa keinoja joko tapauskohtaisesti ohittaa ORM-kehyksen lisäämä hidastava kerros kokonaan tai hyödyntää ORM-kehyksen tarjoamia ominaisuuksia käsin toteutettujen SQL-lauseiden tai proseduurikutsujen kirjoittamiseen. Näillä keinoilla pahimmat ORM-kehysten aiheuttamat hidastumiset pystyttäneen ratkaisemaan ilman, että ORM-kehysten hyvistä puolista tarvitsee luopua. On myös mahdollista kehittää kustannustehokkaasti ja ketterästi tietomalliltaan suuriakin sovelluksia ilman ORM-kehyksiä, Esimerkiksi sovellusgeneraattoria hyödyntäen, kuten Fadjukoff 2021 diplomityössään esitti.

Lähteet

Abdalhakim, Hawaf. 2009. "Addressing Burdens of Open Database Connectivity Standards on the Users". Teoksessa *2009 Third International Symposium on Intelligent Information Technology Application Workshops,* 305–308. https://doi.org/10.1109/IITAW.2009.40.

Bai, Ying. 2012. "Introduction to ADO.NET". Teoksessa *Practical Database Programming with Visual Basic.NET,* 91–148. https://doi.org/10.1002/9781118249833.ch3.

Bauer, Christian, ja Gavin King. 2006. *Java Persistence with Hibernate.* Revised. Manning. ISBN: 1932394885.

Blakeley, J.A. 1996. "OLE DB: a component DBMS architecture". Teoksessa *Proceedings of the Twelfth International Conference on Data Engineering,* 203–204. https://doi.org/10.1109/ICDE.1996.492108.

Bondi, André B. 2000. "Characteristics of scalability and their impact on performance". Teoksessa *Proceedings of the 2nd international workshop on Software and performance,* 195–203.

Case, Jennifer M., ja Gregory Light. 2011. "Emerging Research Methodologies in Engineering Education Research". *Journal of Engineering Education* 100 (1): 186–210. https://doi.org/https://doi.org/10.1002/j.2168-9830.2011.tb00008.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2168-9830.2011.tb00008.x. https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2168-9830.2011.tb00008.x.

Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin ja Larisa Safina. 2017. "Microservices: yesterday, today, and tomorrow". *Present and ulterior software engineering,* 195–216.

DB-Engines.com. 2022. "DB-Engines Ranking of Relational DBMS". Viitattu 27. joulukuuta 2022. https://db-engines.com/en/ranking/relational+dbms.

Eysenck, Hans Jürgen. 1976. *Case studies in behaviour therapy* [kielellä eng]. xii–355.

Fadjukoff, Laura. 2021. "Toimialakohtainen mallinnus sovelluskehityksessä". Tutkielma.

Flyvbjerg, Bent. 2001. *Making social science matter : why social inquiry fails and how it can succeed again* [kielellä eng]. Cambridge: Cambridge University Press. ISBN: 0-521-77268-0.

Foundation, .NET. 2023. “Home | BenchmarkDotNet”. Viitattu 26. helmikuuta 2023. https://benchmarkdotnet.org/.

Foundation, MariaDB. 2023. “MariaDB Server: The open source relational database”. Viitattu 26. helmikuuta 2023. https://mariadb.org/.

GitHub. 2023. “Dapper - a simple object mapper for .Net”. Viitattu 26. helmikuuta 2023. https://github.com/DapperLib/Dapper.

Gorman, Kellyn, Allan Hirt, Dave Noderer, Mitchell Pearson, James Rowland-Jones, Dustin Ryan, Arun Sirpal ja Buck Woody. 2020. *Introducing Microsoft SQL Server 2019: Reliability, scalability, and security both on premises and in the cloud.* Packt Publishing Ltd.

Halpin, Terry. 1998. “Object-role modeling (ORM/NIAM)”. Teoksessa *Handbook on architectures of information systems,* 81–103. Springer.

Hamilton, Bill, ja Matthew MacDonald. 2003. *ADO. NET in a Nutshell.* "O’Reilly Media, Inc.".

Hilbert, Martin, ja Priscila López. 2011. “The world’s technological capacity to store, communicate, and compute information”. *science* 332 (6025): 60–65.

Ireland, Christopher, David Bowers, Michael Newton ja Kevin Waugh. 2009. “A Classification of Object-Relational Impedance Mismatch”. Teoksessa *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications,* 36–43. https://doi.org/10.1109/DBKDA.2009.11.

Koskimies, Kai, ja Tommi Mikkonen. 2005. *Ohjelmistoarkkitehtuurit.*

KumarP, Ravindra, ja Mohd Suaib0TP. n.d. “A Review on Object Oriented Database using Object Relational Modelling and MVC”.

Laine, toimittaja, Markus, toimittaja Bamberg Jarkko ja toimittaja Jokinen Pekka, toimittaneet. 2015. *Tapaustutkimuksen taito.* 3. painos. Helsinki: Gaudeamus Helsinki University Press. https://www.ellibslibrary.com/jyu/9789524956970.

Laitila, Ville. 2005. *Understanding and analyzing SQL/CLI database usage of Java software: empirical study.*

Microsoft. 2020. "SQL samples". Viitattu 10. lokakuuta 2022. https://learn.microsoft.com/en-us/sql/samples/sql-samples-where-are?view=sql-server-ver16.

———. 2022a. "Database Providers". https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli.

———. 2022b. "Driver history for Microsoft SQL Server". Viitattu 26. helmikuuta 2023. https://learn.microsoft.com/en-us/sql/connect/connect-history?view=sql-server-ver16&redirectedfrom=MSDN&viewFallbackFrom=sqlallproducts-allversions.

———. 2022c. "Entity Properties". Viitattu 5. joulukuuta 2022. https://learn.microsoft.com/en-us/ef/core/modeling/entity-properties?tabs=data-annotations.

———. 2022d. "GitHub - dotnet/efcore". Viitattu 27. joulukuuta 2022. https://github.com/dotnet/efcore.

———. 2022e. "Tracking vs. No-Tracking Queries". Viitattu 27. marraskuuta 2022. https://learn.microsoft.com/en-us/ef/core/querying/tracking.

———. 2022f. "Using transactions". Viitattu 6. joulukuuta 2022. https://learn.microsoft.com/en-us/ef/core/saving/transactions.

———. 2022g. "What is .NET?" Viitattu 27. joulukuuta 2022. https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet.

———. 2023. "A tour of the C language". Viitattu 26. helmikuuta 2023. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

Moilanen, Jere. 2020. "Entity Framework 6:n käyttäminen eri tietokantojen päällä", marraskuu. Viitattu 7. lokakuuta 2022. https://jyx.jyu.fi/handle/123456789/72957.

Neward, Ted. 2006. "The Vietnam of Computer Science".

Oracle. 2022a. “Database 19c and 21c | Oracle”. Viitattu 27. joulukuuta 2022. https://www.oracle.com/database/technologies/.

———. 2022b. “Using transactions”. Viitattu 27. joulukuuta 2022. https://www.mysql.com/.

———. 2022c. “What is Java technology and why do I need it?” Viitattu 26. helmikuuta 2023. https://www.java.com/en/download/help/whatis_java.html.

Patel, Pratik, ja Karl Moss. 1997. *Java database programming with JDBC.* Coriolis Group Books.

Peres, Ricardo. 2016. *Entity Framework Core cookbook : leverage the full potential of Entity Framework with this collection of powerful and easy-to-follow recipes.* Toimittanut Ricardo Peres. Packt Publishing. https://sfx.finna.fi/nelli09?url_ver=Z39.88-2004&ctx_ver=Z39.88-2004&ctx_enc=info:ofi/enc:UTF-8&rfr_id=info:sid/sfxit.com:opac_856&url_ctx_fmt=info:ofi/fmt:kev:mtx:ctx&sfx.ignore_date_threshold=1&rft.object_id=3710000000942790&svc_val_fmt=info:ofi/fmt:kev:mtx:sch_svc&.

Russell, Craig. 2008. “Bridging the Object-Relational Divide: ORM Technologies Can Simplify Data Access, but Be Aware of the Challenges That Come with Introducing This New Layer of Abstraction.” *Queue* (New York, NY, USA) 6, numero 3 (toukokuu): 18–28. ISSN: 1542-7730. https://doi.org/10.1145/1394127.1394139. https://doi.org/10.1145/1394127.1394139.

Singh, Rahul Rajat, ym. 2015. *Mastering Entity Framework.* Packt Publishing.

Smith, Jon. 2021. *Entity Framework core in action.* Simon / Schuster.

Statista. 2022. “Ranking of the most popular relational database management systems worldwide, as of January 2022”. Viitattu 27. joulukuuta 2022. https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/.

Telerik, Tsvetomir Y. Todorov. 2013. “Understanding .NET Just-In-Time Compilation”. Viitattu 18. helmikuuta 2023. https://www.telerik.com/blogs/understanding-net-just-in-time-compilation.

The PostgreSQL Global Development Group. 2022. “PostgreSQL”. Viitattu 27. joulukuuta 2022. https://www.postgresql.org/.

Valkonen, Juho. 2023. “GraduApp”. Viitattu 26. helmikuuta 2023. https://github.com/juho Valkonen/GraduApp.

Wiphusitphunpol, Witoon, ja Thitiporn Lertrusdachakul. 2017. “Fetch performance comparison of object relational mapper in .NET platform”. Teoksessa *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON),* 423–426. https://doi.org/10.1109/ECTICon.2017.8096264.

Östman, Tuomas. 2020. “ORM-mallien aikatehokkuusvertailu. NET-alustoilla”. Tutkielma.

Liitteet

A Esimerkki: Järjestämisalgoritmien vertailu benchmark.net kirjastolla

```
using BenchmarkDotNet.Running;
using GraduApp.Benchmark.Benchmarks;

namespace GraduApp.BenchMark
{
    public class Program
    {
        public static void Main(string[] args)
        {
            _ = BenchmarkRunner.Run<SortingBenchMark>();
        }
    }
}
```

```
    public class SortingBenchMark
    {
        private readonly int[] numbers =
            new int[] { 1, 45, 45, 2, 3, 5, 67};

        public static void InsertionSort(int[] input)
        {
             // Implement insertion sort
             // (implementation not relevent for this attachment)
        }

        public static void SelectionSort(int[] input)
        {
           // Implement selection sort
           // (implementation not relevent for this attachment)
        }

        [Benchmark]
        public void SortWithSelection()
        {
            SelectionSort(numbers);
        }

        [Benchmark]
        public void SortWithInsertion()
        {
            InsertionSort(numbers);
        }
    }
}
```

B EFGraduOperations-luokan toteutus

```
using GraduApp.DataAccess.Enums;
using GraduApp.DataAccess.GraduModels;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System.Data;



namespace GraduApp.DataAccess.GraduDBOperations
{
 public class EFGraduDBOperations : IGraduDBOperations
 {
    private GraduDBType _dbType;
    private bool _enableLogging;
    private IConfiguration configuration;
    private bool useChangeTrackingOnSelects = true;
    public EFGraduDBOperations(GraduDBType dbType,
                                bool enableLogging,
                                IConfiguration _configuration,
                                bool _useChangeTrackingOnSelects)
    {
        _dbType = dbType;
        _enableLogging = enableLogging;
        configuration = _configuration;
        useChangeTrackingOnSelects = _useChangeTrackingOnSelects;
        }
    public List<Product> GetProductsByCategoryId(int categoryId)
    {
        using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging, useChangeTrackingOnSelects))
        {
            return ctx.Product
                .Where(x => x.ProductCategoryID == categoryId)
                .ToList();
        }
    }
```

```
public int CountProductsByCategoryId(int categoryId)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
        false, _enableLogging, useChangeTrackingOnSelects))
    {
        return ctx.Product
            .Count(w => w.ProductCategoryID == categoryId);
    }
}

public List<Product> SearchProducts(string keyword)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
        false, _enableLogging, useChangeTrackingOnSelects))
    {
        return ctx.Product
            .Where(x => x.Name.Contains(keyword))
            .ToList();
    }
}

public decimal GetTotalAmountByCustomerId(int customerId)
{
    using (GraduDBContext ctx = new(_dbType, configuration
        , false, _enableLogging, useChangeTrackingOnSelects))
    {
        var sum = (from d in ctx.SalesOrderDetail
                   join h in ctx.SalesOrderHeader
                   on d.SalesOrder.ID equals h.ID
                   select new
                   {
                       h.CustomerID,
                       d.LineTotal
                   })
                 .Where(x => x.CustomerID == customerId)
                 .Sum(x => x.LineTotal);
```

```
            return sum;
        }
    }

    public void UpdateLastName(int customerId, string newLastName)
    {
        using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging))
        {
            Customer c = ctx.Customer.First(x => x.ID == customerId);
            c.LastName = newLastName;
            ctx.SaveChanges();
        }
    }

    public void MultiplyPricesByCategoryId(int categoryId,
                                          decimal priceChange)
    {
        using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging))
        {
            List<Product> products = ctx.Product
                .Where(x => x.ProductCategoryID == categoryId)
                .ToList();

            foreach (Product product in products)
            {
                product.ListPrice = product.ListPrice * priceChange;
            }
            ctx.SaveChanges();
        }
    }

    public void UpdateProductCategoryByCategoryId(int categoryId,
                                                  int newCategoryId)
    {
        using (GraduDBContext ctx = new(_dbType, configuration,
```

```
            false, _enableLogging))
        {
            List<Product> products = ctx.Product
                .Where(x => x.ProductCategoryID == categoryId).ToList();
            foreach (Product product in products)
            {
                product.ProductCategoryID = newCategoryId;
            }
            ctx.SaveChanges();
        }
    }
    public void DeleteCustomerDataByCustomerId(int customerId)
    {
        using (GraduDBContext ctx = new(_dbType, configuration,
                                        false, _enableLogging))
        {

                var customer = ctx.Customer
                                    .Include(x => x.CustomerAddress)
                                    .Include(x => x.SalesOrderHeader)
                                    .ThenInclude(x => x.SalesOrderDetail)
                                    .Where(x => x.ID == customerId)
                                    .First();


                customer.SalesOrderHeader.ToList().ForEach(x =>
                {
                    x.SalesOrderDetail.ToList()
                                .ForEach(x => ctx.Remove(x));

                    ctx.Remove(x);
                });

                customer.CustomerAddress.ToList()
                            .ForEach(x => ctx.Remove(x));
                ctx.Remove(customer);
```

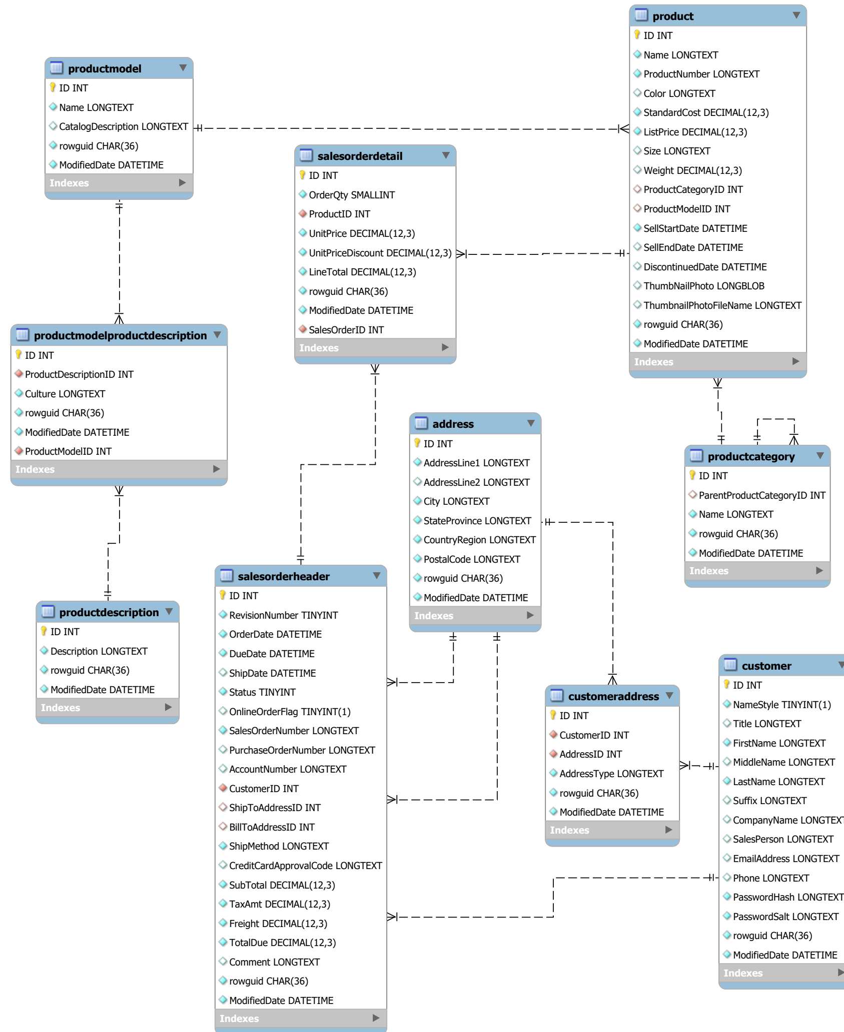
```
        ctx.SaveChanges();

    }
}

public void DeleteSalesOrderDetailById(int salesOrderDetailID)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
                                    false, _enableLogging))
    {
        var customer = ctx.SalesOrderDetail
                        .Where(x => x.ID == salesOrderDetailID)
                        .First();

        ctx.Remove(customer);
        ctx.SaveChanges();
    }
}

public void DeleteProductCategoryByID(int productCategoryID)
{

    using (GraduDBContext ctx = new(_dbType, configuration,
                                    false, _enableLogging))
    {
    var products = ctx.Product
                    .Where(x =>
                        x.ProductCategoryID == productCategoryID
                        ).ToList();
    var productIdList = products.Select(x => x.ID).ToList();


    var salesOrderDetails = ctx.SalesOrderDetail
                    .Where(x => productIdList
                                .Contains(x.ProductID))
                    .ToList();
```

```
ctx.RemoveRange(salesOrderDetails);

var category = ctx.ProductCategory
                .Where(x => x.ID == productCategoryID)
                .ToList();

ctx.RemoveRange(category);
ctx.RemoveRange(products);

var productModelIdList = products
                        .Select(x => x.ProductModelID)
                        .ToList();
var pModelpDescriptions = ctx.ProductModelProductDescription
                                .Where(
                                    x => productModelIdList
                                    .Contains(x.ProductModelID)
                                    ).ToList();

ctx.RemoveRange(pModelpDescriptions);

var productDescriptionIdList = pModelpDescriptions
                                .Select(
                                    x => x.ProductDescriptionID
                                )
                                    .ToList();
var productModels = ctx.ProductModel
                        .Where(x => productModelIdList
                            .Contains(x.ID))
                            .ToList();

ctx.RemoveRange(productModels);

var productDescriptions = ctx.ProductDescription.
                                Where(x =>
                                    productDescriptionIdList
                                    .Contains(x.ID)
```

```
                                ).ToList();
    ctx.RemoveRange(productDescriptions);

    ctx.SaveChanges();

}

}

public int InsertProduct(Product p)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging))
    {
        ctx.Product.Add(p);
        ctx.SaveChanges();
    }
    return p.ID;
}



public int InsertSalesOrderHeader(SalesOrderHeader order)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging))
    {
        ctx.SalesOrderHeader.Add(order);
        ctx.SaveChanges();
    }
    return order.ID;
}

public int InsertCustomer(Customer customer)
{
    using (GraduDBContext ctx = new(_dbType, configuration,
            false, _enableLogging))
```

```
        {
            ctx.Add(customer);
            ctx.SaveChanges();
        }
        return customer.ID;
    }
 }
}
```

C Adventureworks-tietokannan ER-kaavio



Kuvio 16: Adventureworks-tietokannan ER-kaavio.

D Mittauspöytäkirja 1, 28.1.2023

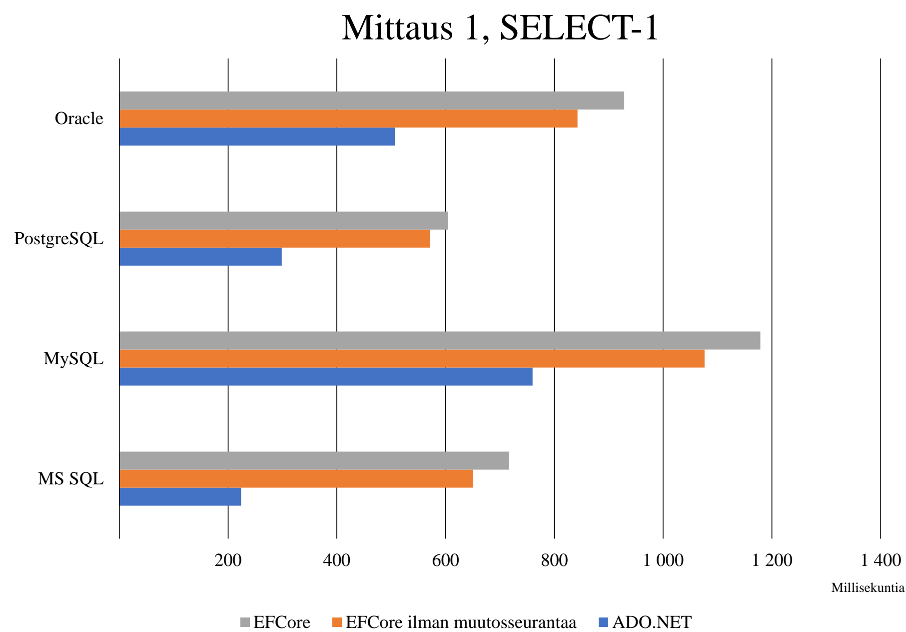
Mittauksen aloitus: 28.1.2023 klo 20:04

Mittauksen päättyminen: 28.1.2023 klo 22:08

Tulokset:

Taulukko 3: Mittaus 1, SELECT-1 tulokset.

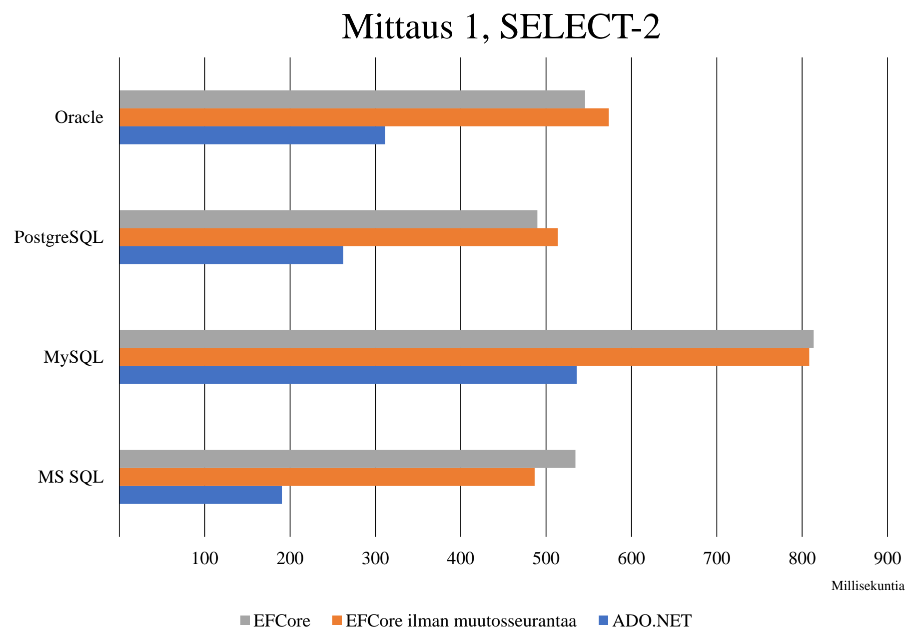
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	223,90	759,90	298,60	506,90	
EFCore	716,60	1 178,70	604,90	928,30	
EFCore, viive (%)	220,05	55,11	102,58	83,13	115,22
EFCore, viive (ms.)	492,70	418,80	306,30	421,40	
EFCore ilman muutosseurantaa	650,70	1 076,10	571,00	842,40	
EFCore ilman muutosseurantaa, viive (%)	190,62	41,61	91,23	66,19	97,41
EFCore ilman muutosseurantaa, viive (ms.)	426,80	316,20	272,40	335,50	



Kuvio 17: Mittaus 1, SELECT-1 tulokset palkkikaaviona.

Taulukko 4: Mittaus 1, SELECT-2 tulokset.

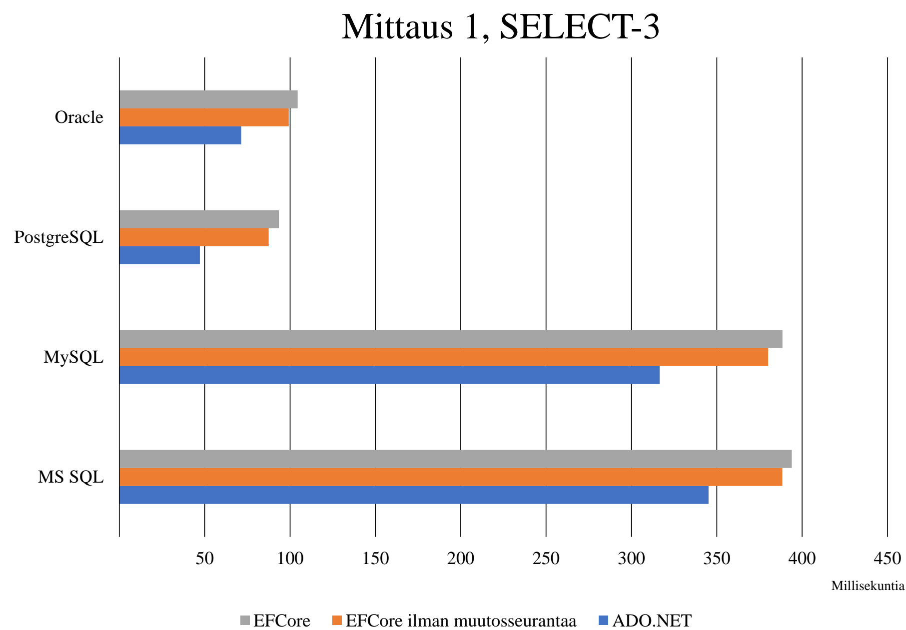
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	190,50	535,80	262,30	311,30	
EFCore	534,40	813,50	489,80	545,60	
EFCore, viive (%)	180,52	51,83	86,73	75,27	98,59
EFCore, viive (ms.)	343,90	277,70	227,50	234,30	
EFCore ilman muutosseurantaa	486,60	808,30	513,60	573,30	
EFCore ilman muutosseurantaa, viive (%)	155,43	50,86	95,81	84,16	96,57
EFCore ilman muutosseurantaa, viive (ms.)	296,10	272,50	251,30	262,00	



Kuvio 18: Mittaus 1, SELECT-2 tulokset palkkikaaviona.

Taulukko 5: Mittaus 1, SELECT-3 tulokset.

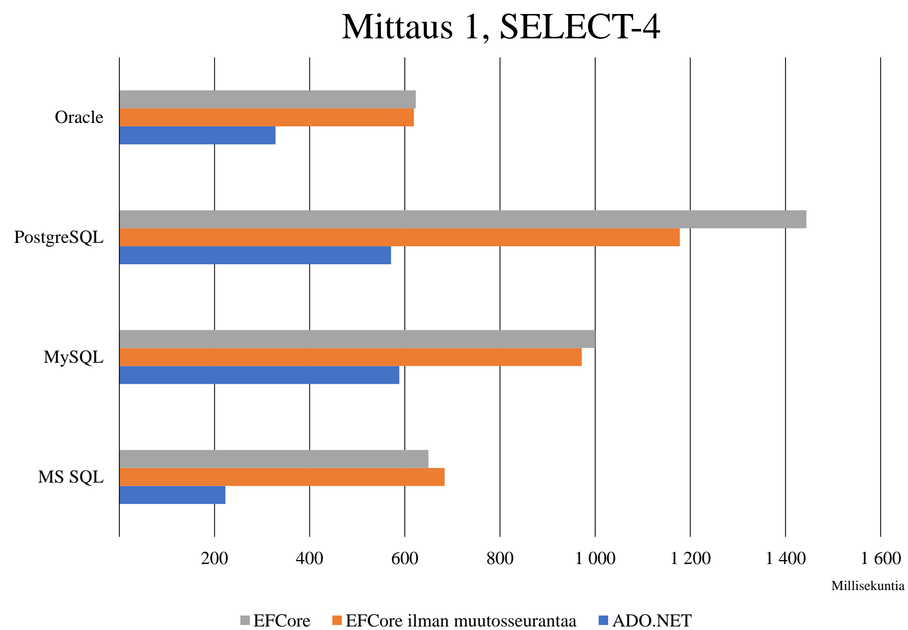
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	345,18	316,50	47,15	71,40	
EFCore	393,91	388,46	93,48	104,46	
EFCore, viive (%)	14,12	22,74	98,26	46,30	45,35
EFCore, viive (ms.)	48,73	71,96	46,33	33,06	
EFCore ilman muutosseurantaa	388,41	380,06	87,50	99,21	
EFCore ilman muutosseurantaa, viive (%)	12,52	20,08	85,58	38,95	39,28
EFCore ilman muutosseurantaa, viive (ms.)	43,23	63,56	40,35	27,81	



Kuvio 19: Mittaus 1, SELECT-3 tulokset palkkikaaviona.

Taulukko 6: Mittaus 1, SELECT-4 tulokset.

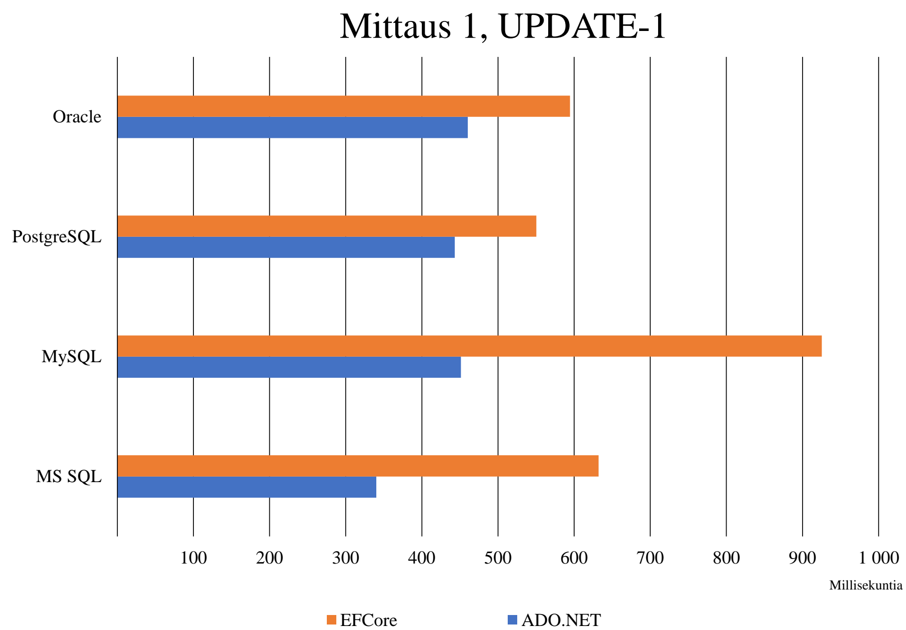
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	223,00	588,30	571,20	328,20	
EFCore	649,70	1 000,00	1 443,80	623,00	
EFCore, viive (%)	191,35	69,98	152,77	89,82	125,98
EFCore, viive (ms.)	426,70	411,70	872,60	294,80	
EFCore ilman muutosseurantaa	683,80	971,90	1 178,00	619,00	
EFCore ilman muutosseurantaa, viive (%)	206,64	65,20	106,23	88,60	116,67
EFCore ilman muutosseurantaa, viive (ms.)	460,80	383,60	606,80	290,80	



Kuvio 20: Mittaus 1, SELECT-4 tulokset palkkikaaviona.

Taulukko 7: Mittaus 1, UPDATE-1 tulokset.

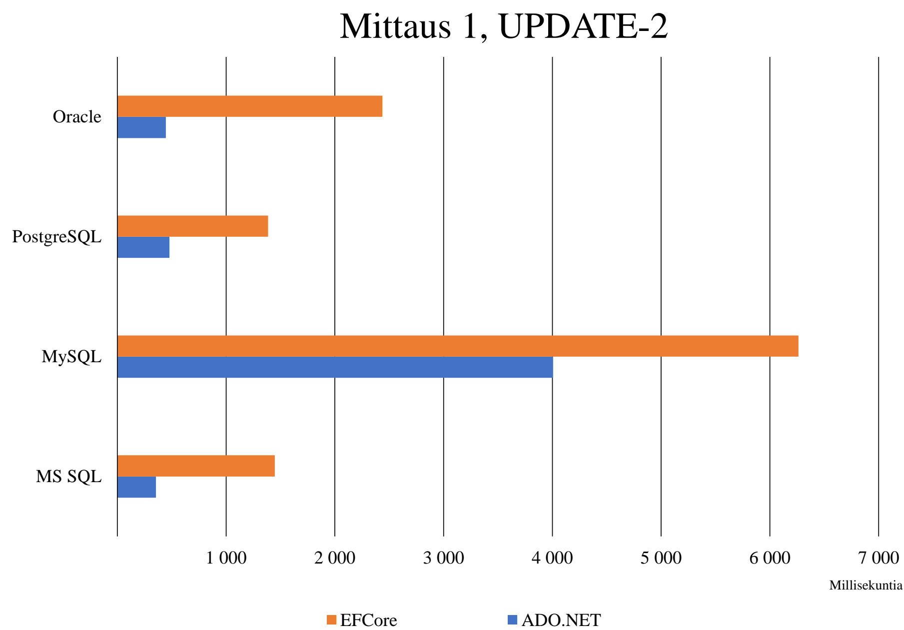
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	340,10	451,40	443,20	460,40	
EECore	632,20	925,40	550,30	594,60	
EECore, viive (%)	85,89	105,01	24,17	29,15	61,05
EECore, viive (ms.)	292,10	474,00	107,10	134,20	



Kuvio 21: Mittaus 1, UPDATE-1 tulokset palkkikaaviona.

Taulukko 8: Mittaus 1, UPDATE-2 tulokset.

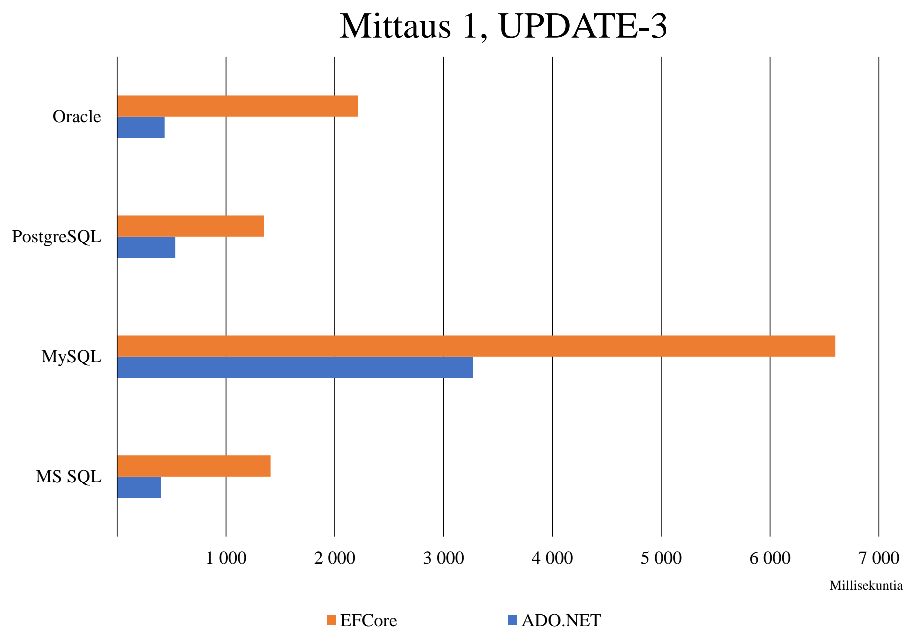
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	355,20	4 005,10	478,80	445,70	
EECore	1 448,30	6 262,60	1 386,40	2 437,20	
EECore, viive (%)	307,74	56,37	189,56	446,83	250,12
EECore, viive (ms.)	1 093,10	2 257,50	907,60	1 991,50	



Kuvio 22: Mittaus 1, UPDATE-2 tulokset palkkikaaviona.

Taulukko 9: Mittaus 1, UPDATE-3 tulokset.

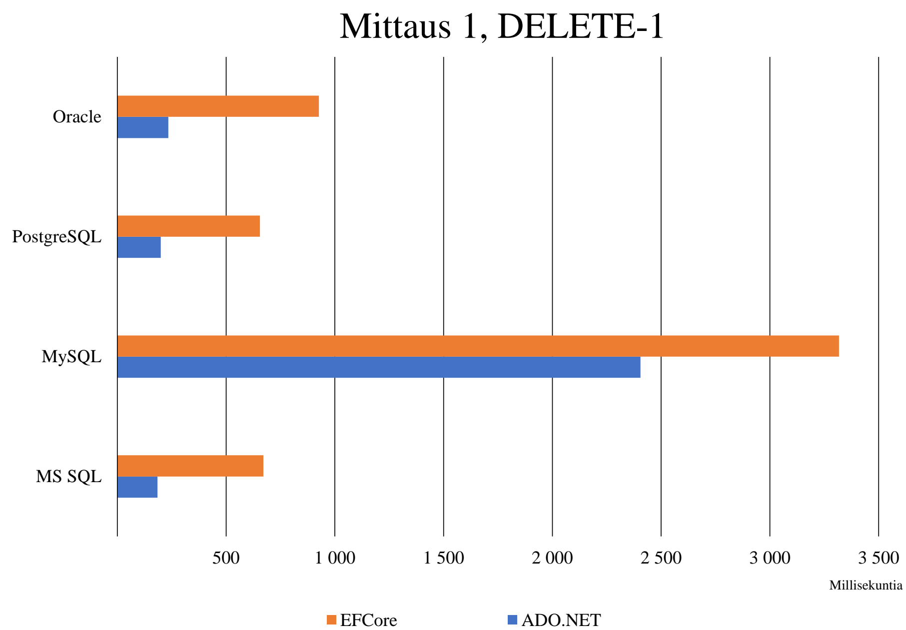
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	401,80	3 269,00	534,10	435,90	
EECore	1 409,30	6 599,50	1 350,00	2 214,20	
EECore, viive (%)	250,75	101,88	152,76	407,96	228,34
EECore, viive (ms.)	1 007,50	3 330,50	815,90	1 778,30	



Kuvio 23: Mittaus 1, UPDATE-3 tulokset palkkikaaviona.

Taulukko 10: Mittaus 1, DELETE-1 tulokset.

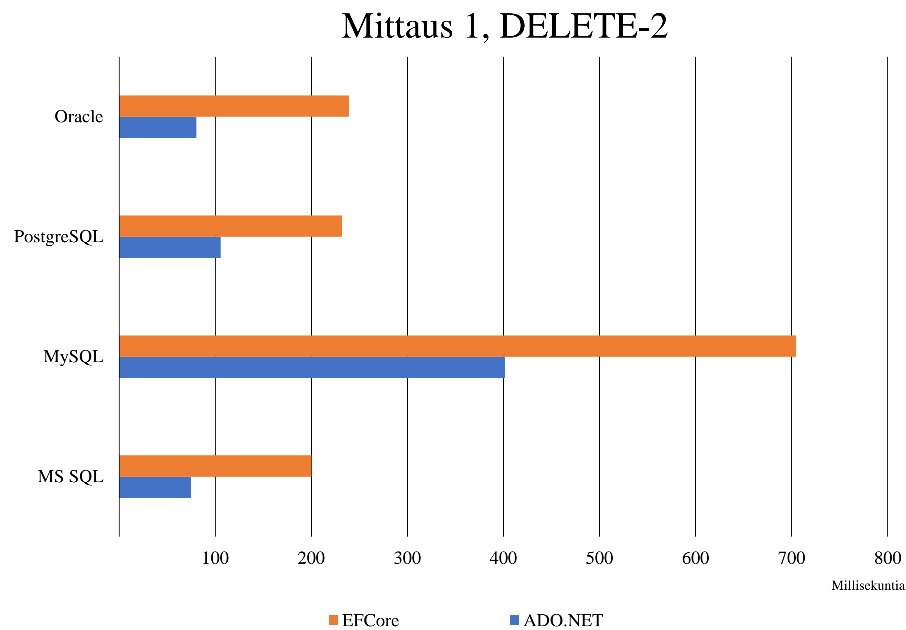
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	184,80	2 405,00	199,50	234,10	
EECore	671,50	3 317,90	655,90	926,50	
EECore, viive (%)	263,37	37,96	228,77	295,77	206,47
EECore, viive (ms.)	486,70	912,90	456,40	692,40	



Kuvio 24: Mittaus 1, DELETE-1 tulokset palkkikaaviona.

Taulukko 11: Mittaus 1, DELETE-2 tulokset.

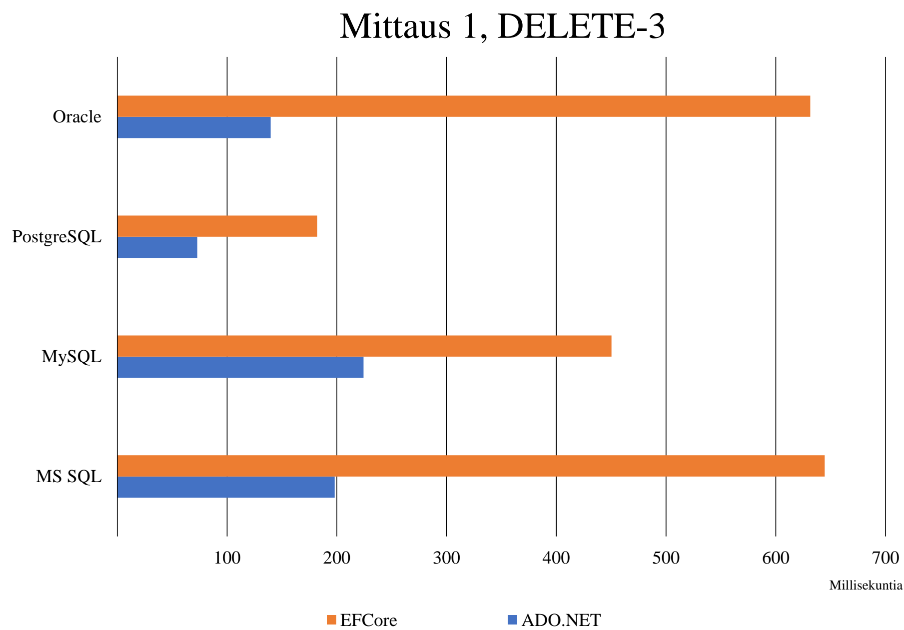
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	74,69	401,80	105,63	80,44	
EECore	200,16	704,44	231,85	239,26	
EECore, viive (%)	167,99	75,32	119,49	197,44	140,06
EECore, viive (ms.)	125,47	302,64	126,22	158,82	



Kuvio 25: Mittaus 1, DELETE-2 tulokset palkkikaaviona.

Taulukko 12: Mittaus 1, DELETE-3 tulokset.

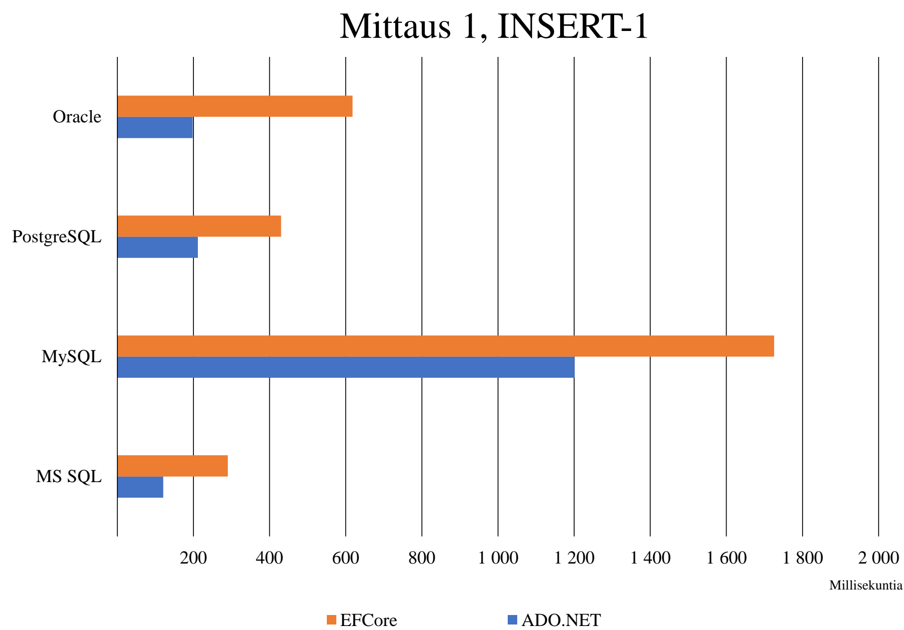
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	198,11	224,29	72,84	139,66	
EECore	644,60	450,33	182,15	631,39	
EECore, viive (%)	225,37	100,78	150,07	352,09	207,08
EECore, viive (ms.)	446,49	226,04	109,31	491,73	



Kuvio 26: Mittaus 1, DELETE-3 tulokset palkkikaaviona.

Taulukko 13: Mittaus 1, INSERT-1 tulokset.

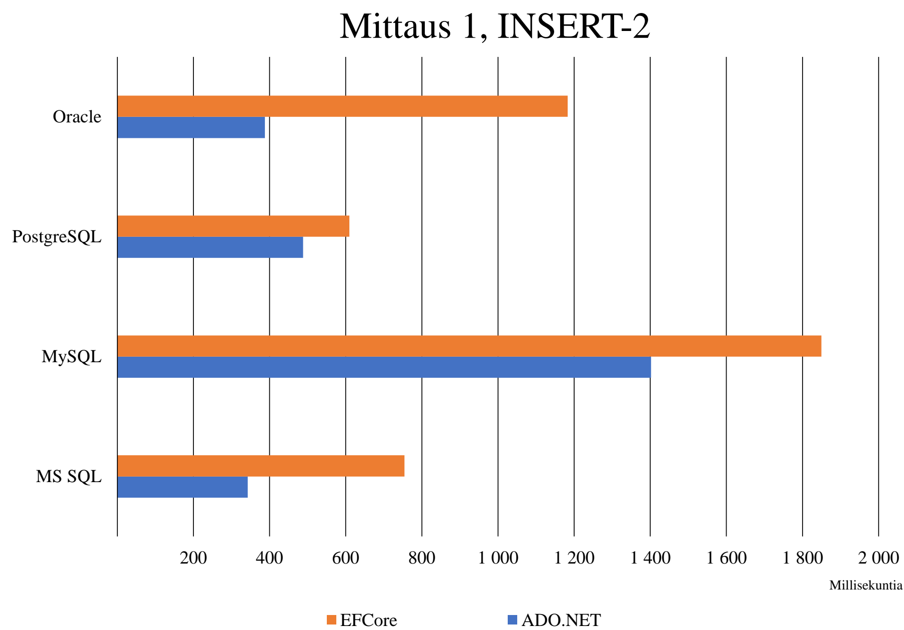
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	120,60	1 201,20	211,70	197,90	
EECore	290,20	1 725,30	430,30	617,90	
EECore, viive (%)	140,63	43,63	103,26	212,23	124,94
EECore, viive (ms.)	169,60	524,10	218,60	420,00	



Kuvio 27: Mittaus 1, INSERT-1 tulokset palkkikaaviona.

Taulukko 14: Mittaus 1, INSERT-2 tulokset.

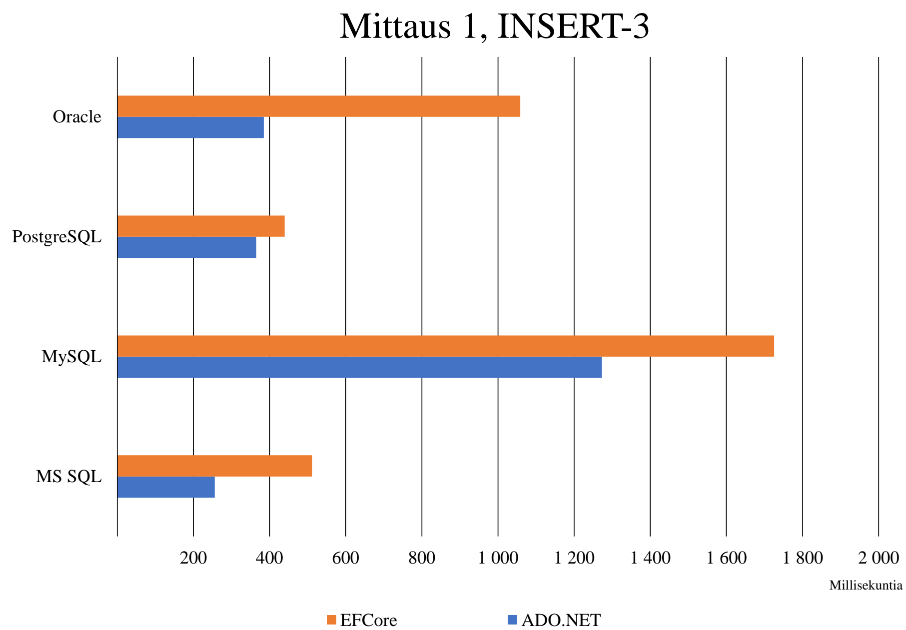
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	342,50	1 402,10	487,80	387,50	
EECore	754,40	1 849,70	609,70	1 182,90	
EECore, viive (%)	120,26	31,92	24,99	205,26	95,61
EECore, viive (ms.)	411,90	447,60	121,90	795,40	



Kuvio 28: Mittaus 1, INSERT-2 tulokset palkkikaaviona.

Taulukko 15: Mittaus 1, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	256,00	1 272,80	365,00	384,70	
EECore	511,50	1 725,40	439,50	1 058,50	
EECore, viive (%)	99,80	35,56	20,41	175,15	82,73
EECore, viive (ms.)	255,50	452,60	74,50	673,80	



Kuvio 29: Mittaus 1, INSERT-3 tulokset palkkikaaviona.

E Mittauspöytäkirja 2, 29.1.2023

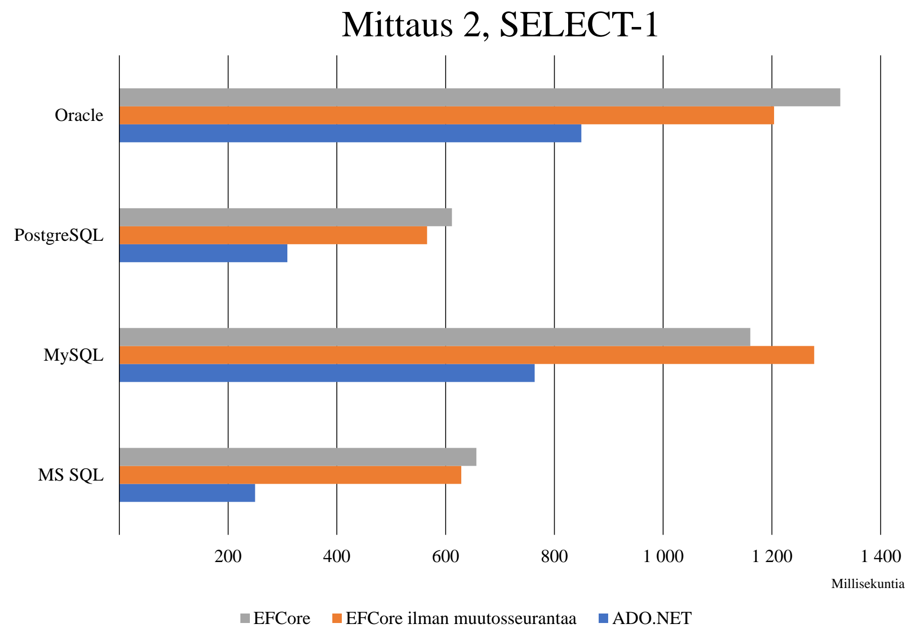
Mittauksen aloitus: 29.1.2023 klo 12:57

Mittauksen päättyminen: 29.1.2023 klo 15:23

Tulokset:

Taulukko 16: Mittaus 2, SELECT-1 tulokset.

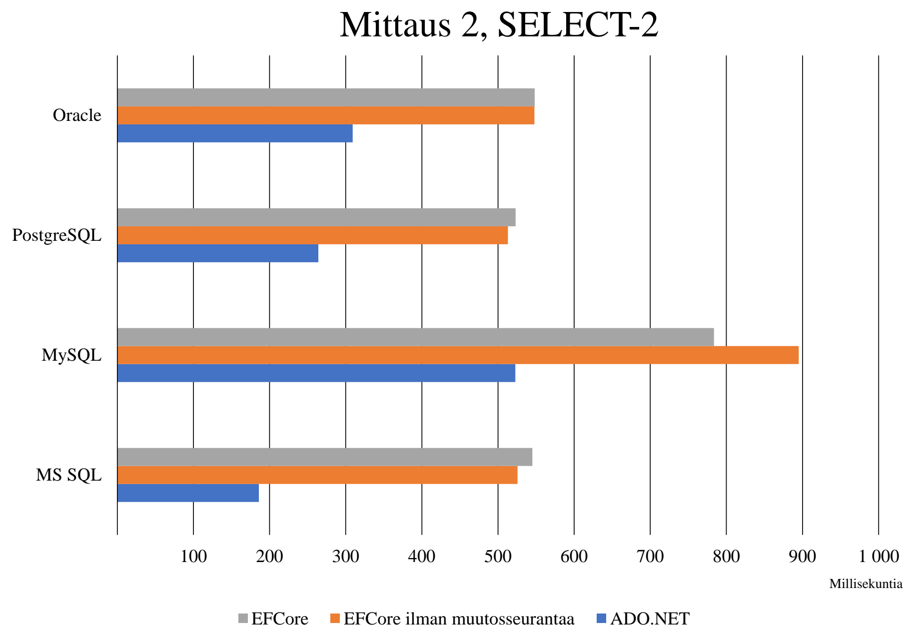
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	249,60	763,80	309,10	849,60	
EFCore	656,70	1 160,40	611,70	1 325,80	
EFCore, viive (%)	163,10	51,92	97,90	56,05	92,24
EFCore, viive (ms.)	407,10	396,60	302,60	476,20	
EFCore ilman muutosseurantaa	628,80	1 277,70	566,00	1 204,10	
EFCore ilman muutosseurantaa, viive (%)	151,92	67,28	83,11	41,73	86,01
EFCore ilman muutosseurantaa, viive (ms.)	379,20	513,90	256,90	354,50	



Kuvio 30: Mittaus 2, SELECT-1 tulokset palkkikaaviona.

Taulukko 17: Mittaus 2, SELECT-2 tulokset.

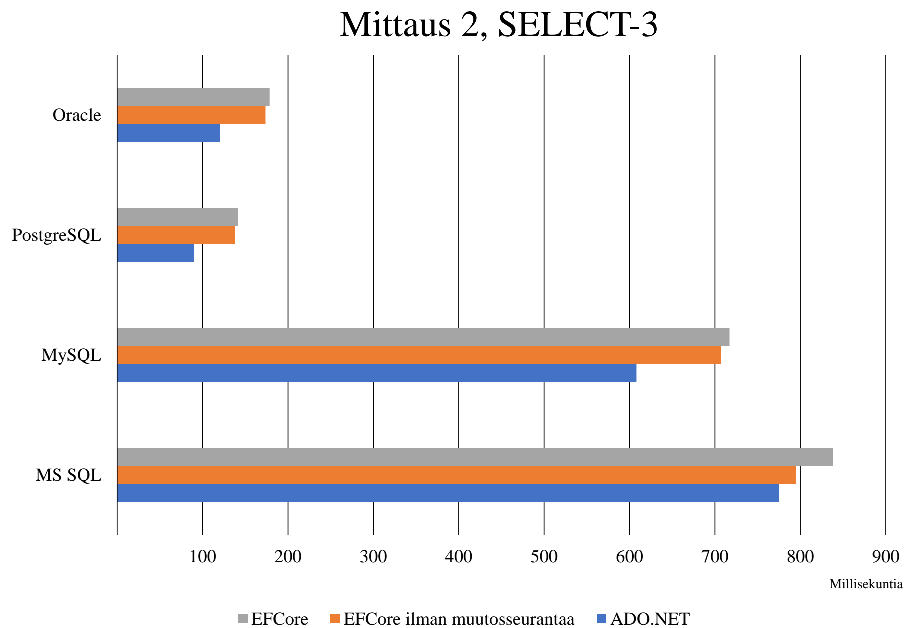
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	185,80	522,80	264,00	309,10	
EFCore	545,20	783,70	523,20	548,30	
EFCore, viive (%)	193,43	49,90	98,18	77,39	104,73
EFCore, viive (ms.)	359,40	260,90	259,20	239,20	
EFCore ilman muutosseurantaa	525,60	895,10	513,00	547,90	
EFCore ilman muutosseurantaa, viive (%)	182,88	71,21	94,32	77,26	106,42
EFCore ilman muutosseurantaa, viive (ms.)	339,80	372,30	249,00	238,80	



Kuvio 31: Mittaus 2, SELECT-2 tulokset palkkikaaviona.

Taulukko 18: Mittaus 2, SELECT-3 tulokset.

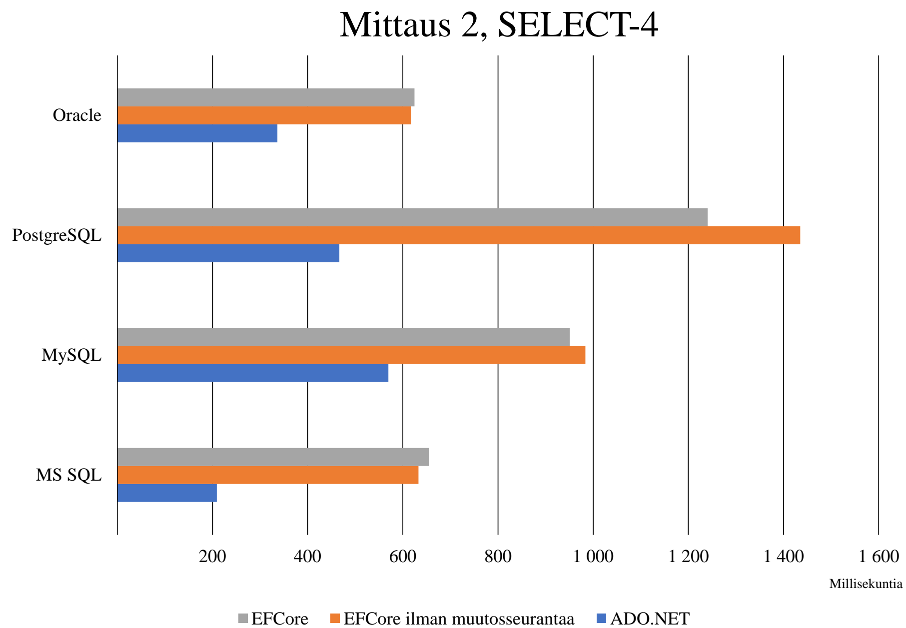
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	775,19	608,14	89,83	120,23	
EFCore	838,38	717,07	141,35	178,48	
EFCore, viive (%)	8,15	17,91	57,35	48,45	32,97
EFCore, viive (ms.)	63,19	108,93	51,52	58,25	
EFCore ilman muutosseurantaa	794,67	707,41	138,10	173,69	
EFCore ilman muutosseurantaa, viive (%)	2,51	16,32	53,73	44,46	29,26
EFCore ilman muutosseurantaa, viive (ms.)	19,48	99,27	48,27	53,46	



Kuvio 32: Mittaus 2, SELECT-3 tulokset palkkikaaviona.

Taulukko 19: Mittaus 2, SELECT-4 tulokset.

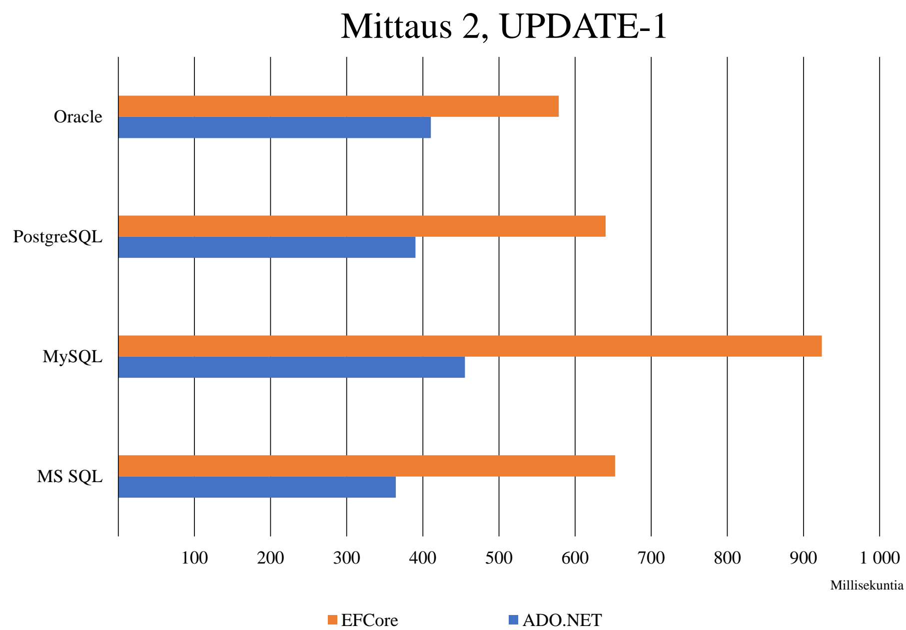
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	209,00	569,80	466,60	336,50	
EFCore	654,60	950,70	1 240,70	624,40	
EFCore, viive (%)	213,21	66,85	165,90	85,56	132,88
EFCore, viive (ms.)	445,60	380,90	774,10	287,90	
EFCore ilman muutosseurantaa	632,80	983,80	1 435,10	617,00	
EFCore ilman muutosseurantaa, viive (%)	202,78	72,66	207,57	83,36	141,59
EFCore ilman muutosseurantaa, viive (ms.)	423,80	414,00	968,50	280,50	



Kuvio 33: Mittaus 2, SELECT-4 tulokset palkkikaaviona.

Taulukko 20: Mittaus 2, UPDATE-1 tulokset.

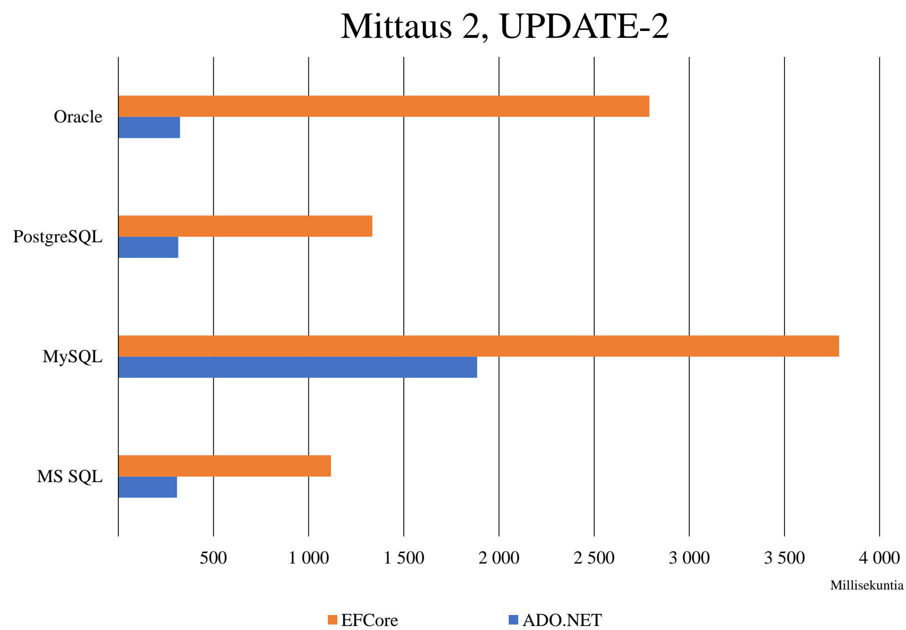
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	364,50	455,20	390,40	410,60	
EECore	652,50	924,00	640,10	578,50	
EECore, viive (%)	79,01	102,99	63,96	40,89	71,71
EECore, viive (ms.)	288,00	468,80	249,70	167,90	



Kuvio 34: Mittaus 2, UPDATE-1 tulokset palkkikaaviona.

Taulukko 21: Mittaus 2, UPDATE-2 tulokset.

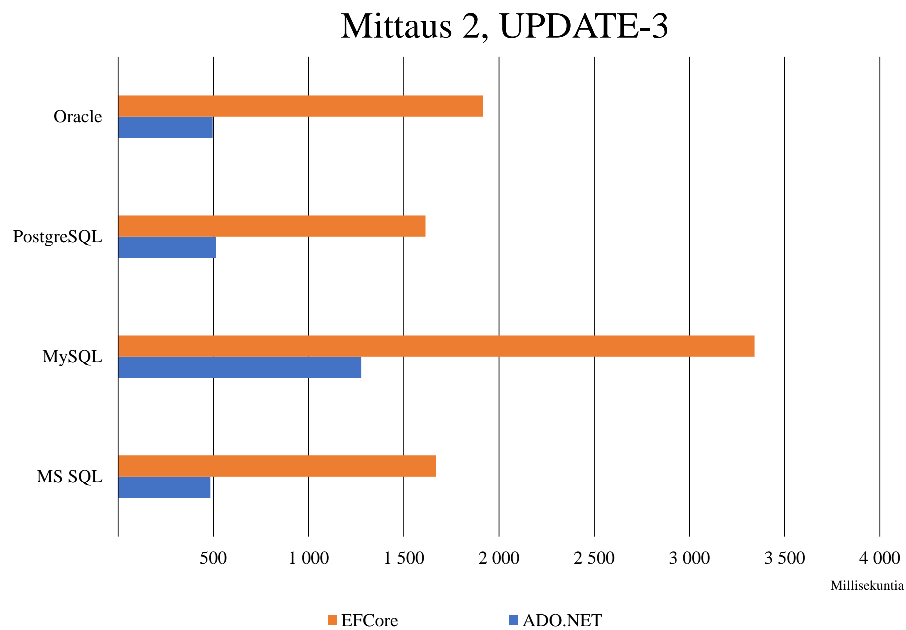
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	308,60	1 885,40	314,50	324,50	
EECore	1 117,40	3 787,20	1 334,40	2 790,20	
EECore, viive (%)	262,09	100,87	324,29	759,85	361,77
EECore, viive (ms.)	808,80	1 901,80	1 019,90	2 465,70	



Kuvio 35: Mittaus 2, UPDATE-2 tulokset palkkikaaviona.

Taulukko 22: Mittaus 2, UPDATE-3 tulokset.

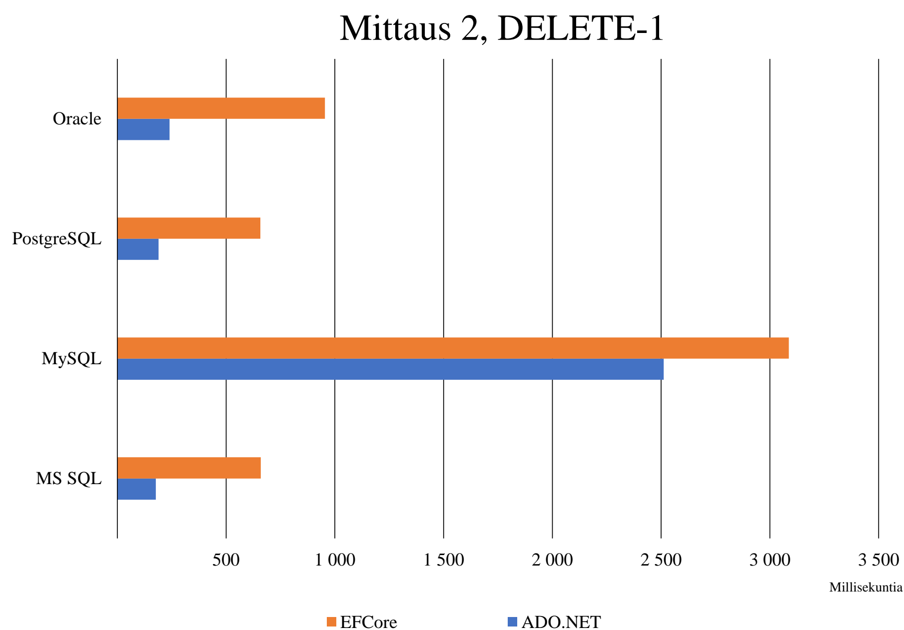
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	484,20	1 277,00	513,00	496,20	
EECore	1 670,00	3 342,25	1 613,75	1 915,25	
EECore, viive (%)	244,90	161,73	214,57	285,98	226,80
EECore, viive (ms.)	1 185,80	2 065,25	1 100,75	1 419,05	



Kuvio 36: Mittaus 2, UPDATE-3 tulokset palkkikaaviona.

Taulukko 23: Mittaus 2, DELETE-1 tulokset.

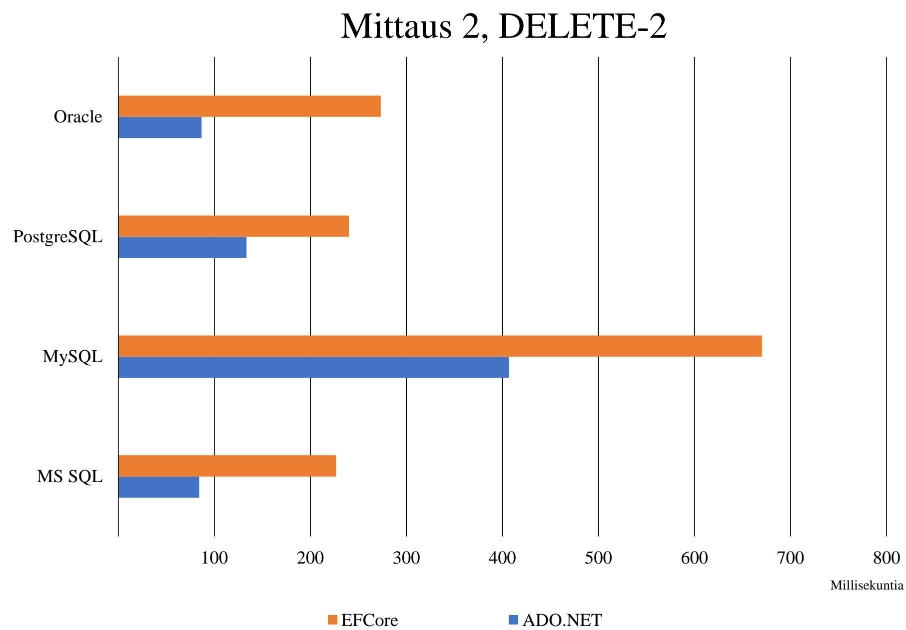
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	177,10	2 511,50	189,70	240,00	
EECore	659,60	3 087,20	657,60	954,50	
EECore, viive (%)	272,44	22,92	246,65	297,71	209,93
EECore, viive (ms.)	482,50	575,70	467,90	714,50	



Kuvio 37: Mittaus 2, DELETE-1 tulokset palkkikaaviona.

Taulukko 24: Mittaus 2, DELETE-2 tulokset.

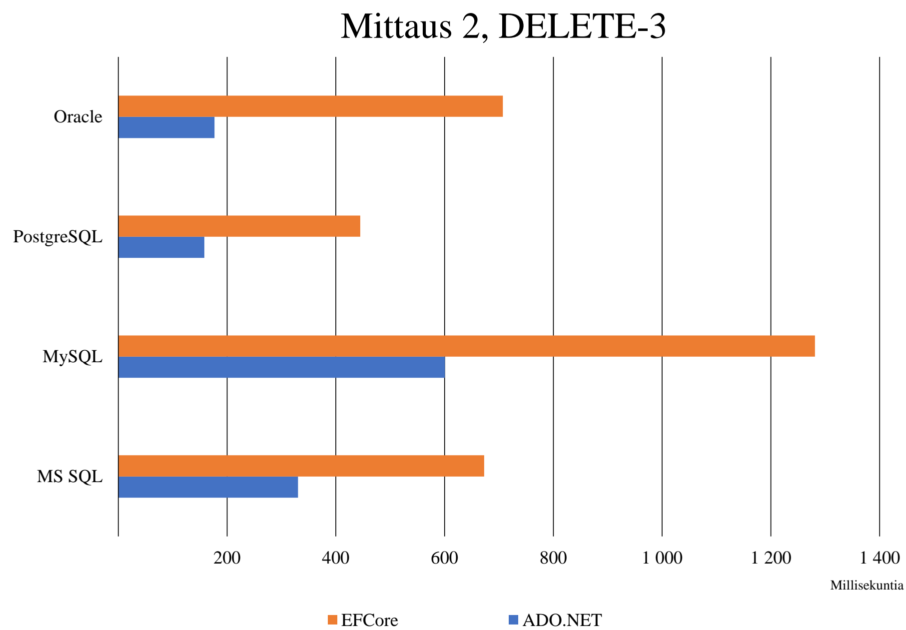
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	84,09	406,80	133,48	86,67	
EECore	226,59	670,25	240,05	273,28	
EECore, viive (%)	169,46	64,76	79,84	215,31	132,34
EECore, viive (ms.)	142,50	263,45	106,57	186,61	



Kuvio 38: Mittaus 2, DELETE-2 tulokset palkkikaaviona.

Taulukko 25: Mittaus 2, DELETE-3 tulokset.

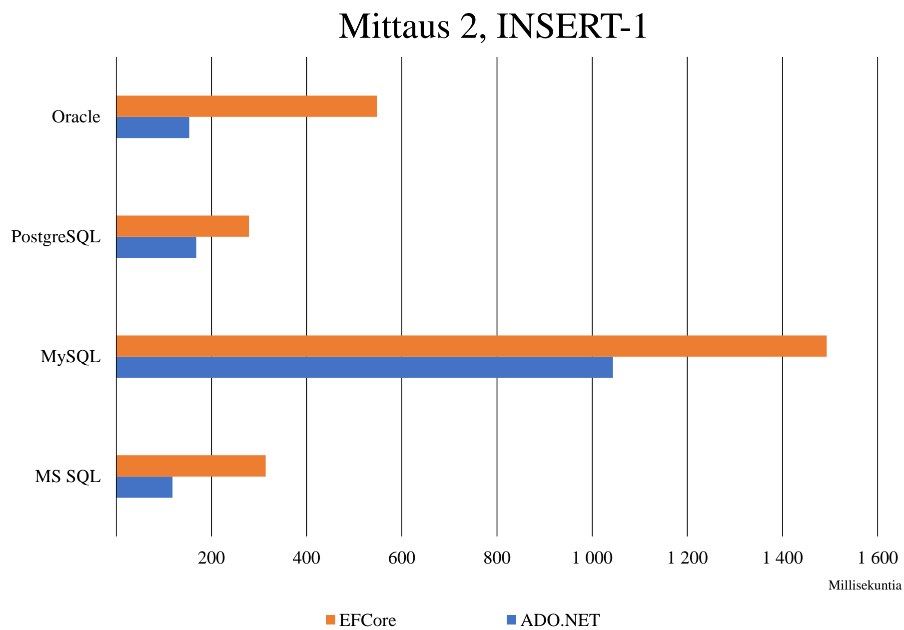
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	330,55	600,50	158,30	176,90	
EECore	672,70	1 281,15	444,85	707,10	
EECore, viive (%)	103,51	113,35	181,02	299,72	174,40
EECore, viive (ms.)	342,15	680,65	286,55	530,20	



Kuvio 39: Mittaus 2, DELETE-3 tulokset palkkikaaviona.

Taulukko 26: Mittaus 2, INSERT-1 tulokset.

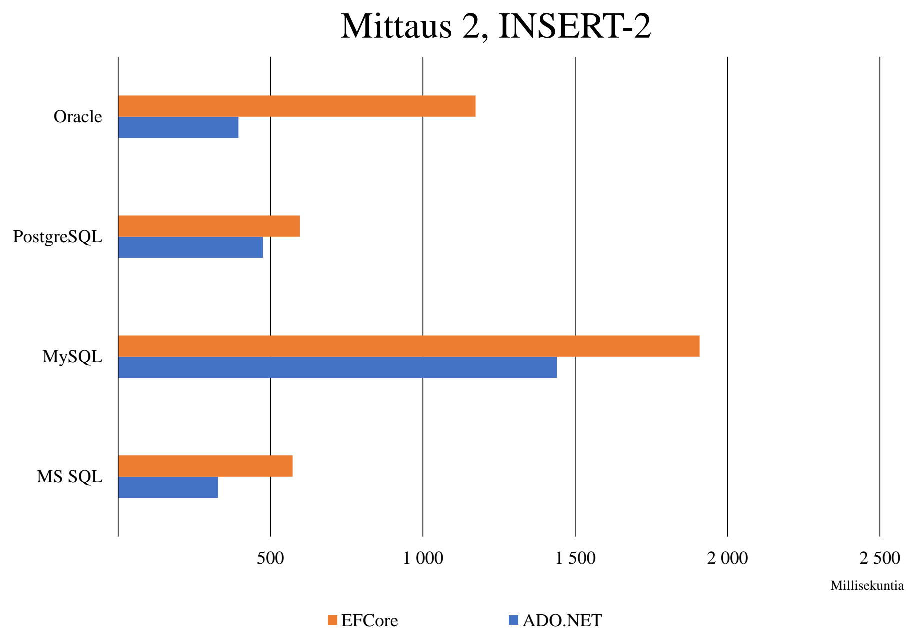
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	118,20	1 043,70	168,20	153,40	
EECore	313,80	1 493,00	278,50	547,60	
EECore, viive (%)	165,48	43,05	65,58	256,98	132,77
EECore, viive (ms.)	195,60	449,30	110,30	394,20	



Kuvio 40: Mittaus 2, INSERT-1 tulokset palkkikaaviona.

Taulukko 27: Mittaus 2, INSERT-2 tulokset.

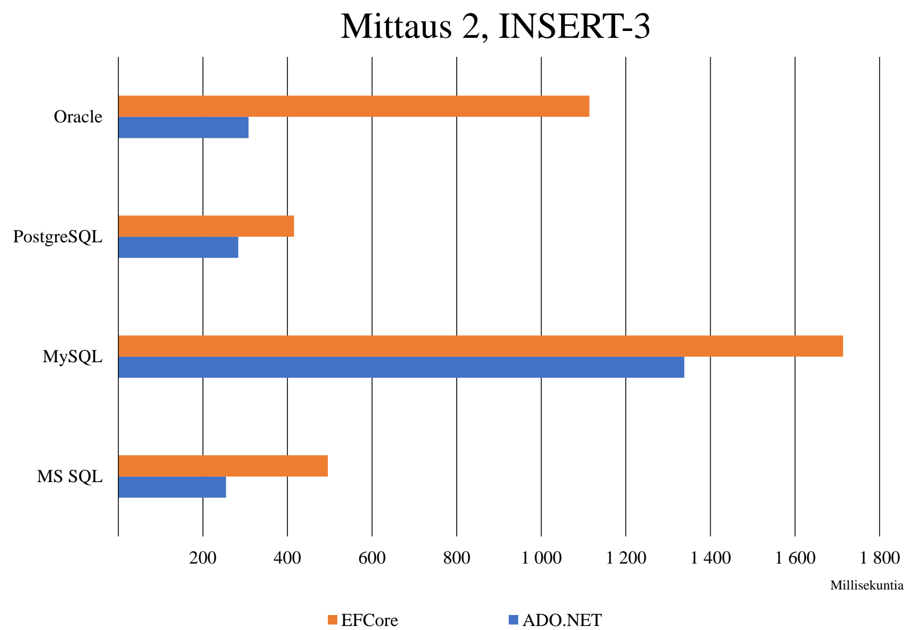
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	328,10	1 440,20	475,00	394,60	
EECore	572,50	1 908,20	595,90	1 172,80	
EECore, viive (%)	74,49	32,50	25,45	197,21	82,41
EECore, viive (ms.)	244,40	468,00	120,90	778,20	



Kuvio 41: Mittaus 2, INSERT-2 tulokset palkkikaaviona.

Taulukko 28: Mittaus 2, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	254,60	1 338,30	283,70	307,90	
EECore	495,30	1 713,70	415,60	1 113,90	
EECore, viive (%)	94,54	28,05	46,49	261,77	107,71
EECore, viive (ms.)	240,70	375,40	131,90	806,00	



Kuvio 42: Mittaus 2, INSERT-3 tulokset palkkikaaviona.

F Mittauspöytäkirja 3, 29.1.2023

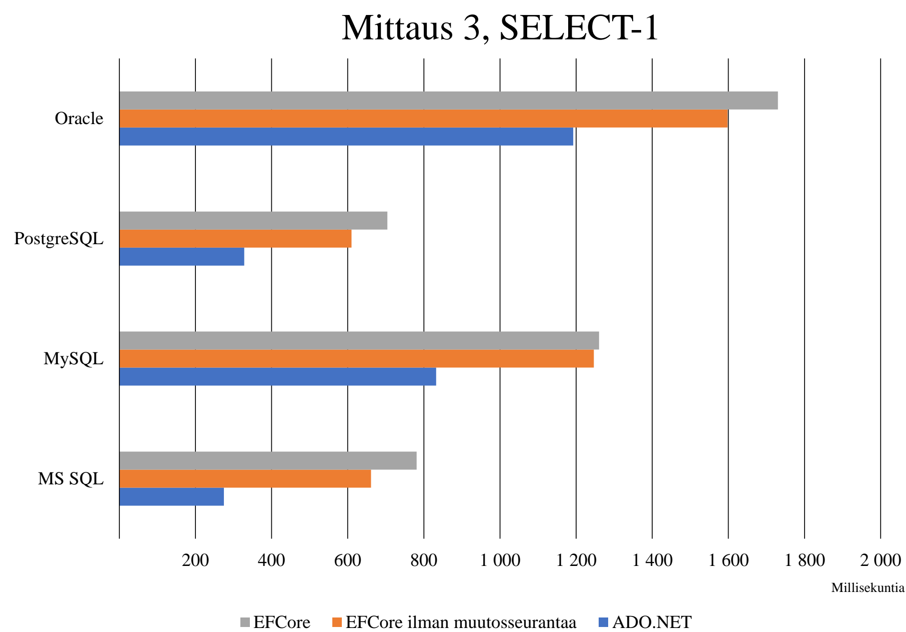
Mittauksen aloitus: 29.1.2023 klo 17:57

Mittauksen päättyminen: 29.1.2023 klo 20:48

Tulokset:

Taulukko 29: Mittaus 3, SELECT-1 tulokset.

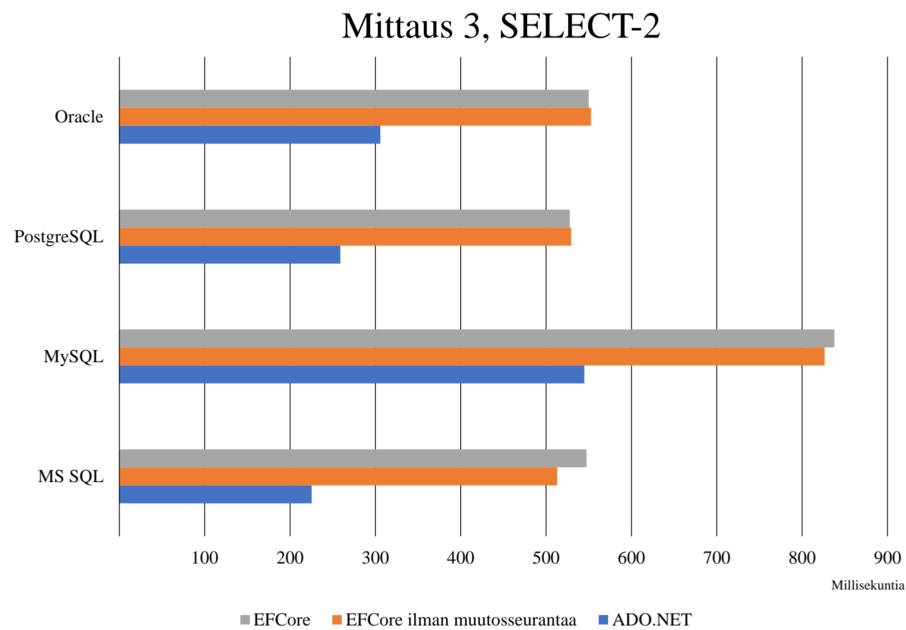
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	274,70	832,30	328,10	1 192,40	
EFCore	780,90	1 260,20	704,20	1 730,20	
EFCore, viive (%)	184,27	51,41	114,63	45,10	98,85
EFCore, viive (ms.)	515,10	427,90	376,10	537,80	
EFCore ilman muutosseurantaa	661,30	1 246,70	610,10	1 597,70	
EFCore ilman muutosseurantaa, viive (%)	140,74	49,79	85,95	33,99	77,62
EFCore ilman muutosseurantaa, viive (ms.)	386,60	414,40	282,00	405,30	



Kuvio 43: Mittaus 3, SELECT-1 tulokset palkkikaaviona.

Taulukko 30: Mittaus 3, SELECT-2 tulokset.

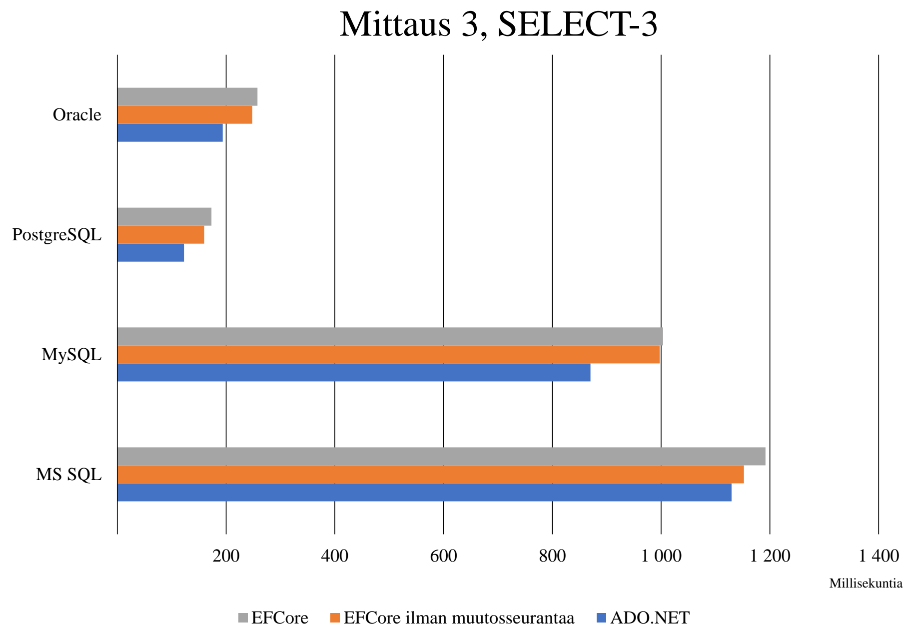
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	225,40	544,80	259,00	305,90	
EFCore	547,40	837,80	527,70	549,90	
EFCore, viive (%)	142,86	53,78	103,75	79,76	95,04
EFCore, viive (ms.)	322,00	293,00	268,70	244,00	
EFCore ilman muutosseurantaa	513,00	826,30	529,30	552,60	
EFCore ilman muutosseurantaa, viive (%)	127,60	51,67	104,36	80,65	91,07
EFCore ilman muutosseurantaa, viive (ms.)	287,60	281,50	270,30	246,70	



Kuvio 44: Mittaus 3, SELECT-2 tulokset palkkikaaviona.

Taulukko 31: Mittaus 3, SELECT-3 tulokset.

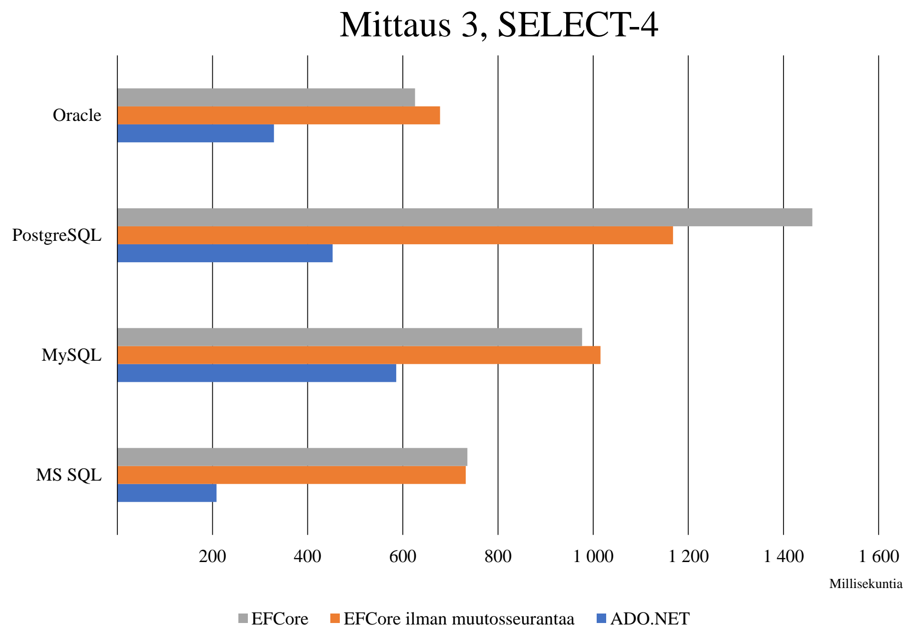
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	1 129,60	870,10	122,60	193,80	
EFCore	1 191,80	1 003,50	173,20	257,60	
EFCore, viive (%)	5,51	15,33	41,27	32,92	23,76
EFCore, viive (ms.)	62,20	133,40	50,60	63,80	
EFCore ilman muutosseurantaa	1 152,30	997,40	159,60	248,00	
EFCore ilman muutosseurantaa, viive (%)	2,01	14,63	30,18	27,97	18,70
EFCore ilman muutosseurantaa, viive (ms.)	22,70	127,30	37,00	54,20	



Kuvio 45: Mittaus 3, SELECT-3 tulokset palkkikaaviona.

Taulukko 32: Mittaus 3, SELECT-4 tulokset.

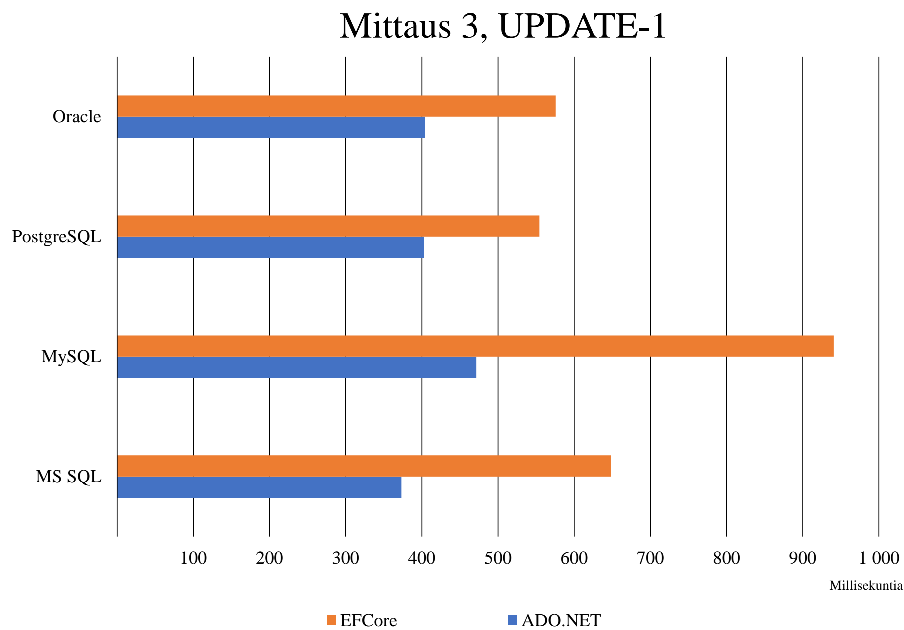
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	208,40	586,20	452,50	329,10	
EFCore	735,70	976,60	1 460,50	625,70	
EFCore, viive (%)	253,02	66,60	222,76	90,12	158,13
EFCore, viive (ms.)	527,30	390,40	1 008,00	296,60	
EFCore ilman muutosseurantaa	732,20	1 015,30	1 168,00	678,10	
EFCore ilman muutosseurantaa, viive (%)	251,34	73,20	158,12	106,05	147,18
EFCore ilman muutosseurantaa, viive (ms.)	523,80	429,10	715,50	349,00	



Kuvio 46: Mittaus 3, SELECT-4 tulokset palkkikaaviona.

Taulukko 33: Mittaus 3, UPDATE-1 tulokset.

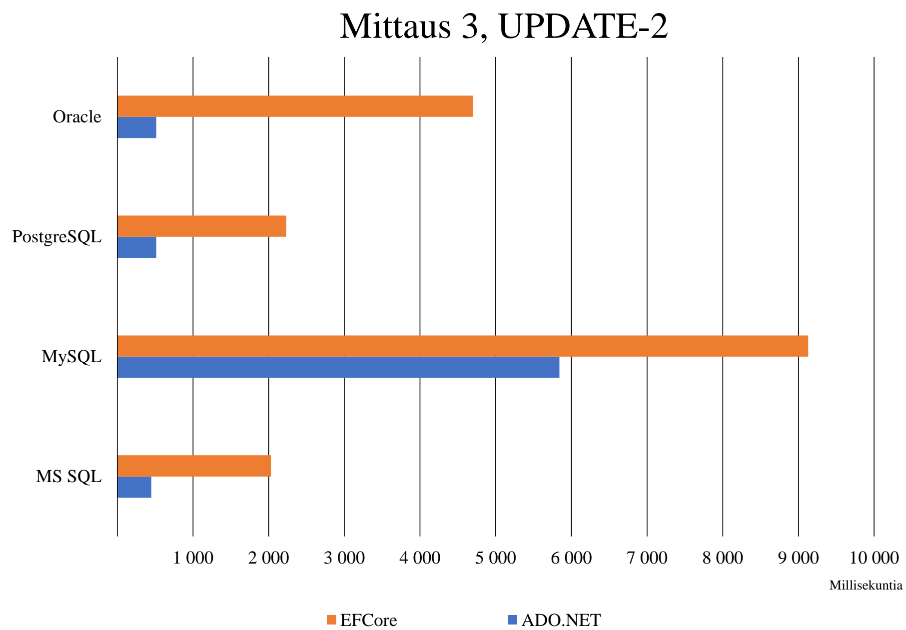
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	373,20	471,50	402,70	404,10	
EECore	648,30	940,60	554,40	575,70	
EECore, viive (%)	73,71	99,49	37,67	42,46	63,34
EECore, viive (ms.)	275,10	469,10	151,70	171,60	



Kuvio 47: Mittaus 3, UPDATE-1 tulokset palkkikaaviona.

Taulukko 34: Mittaus 3, UPDATE-2 tulokset.

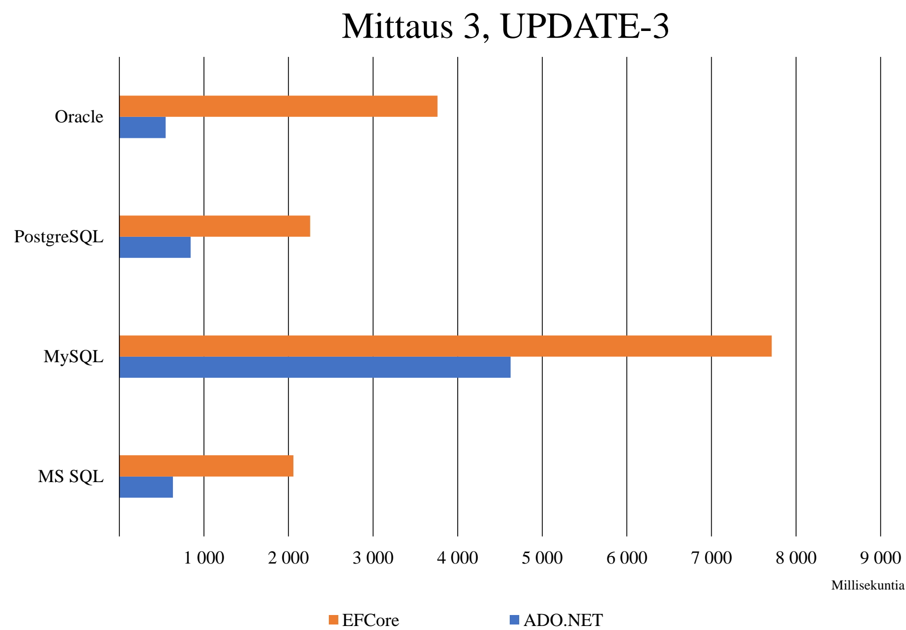
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	448,90	5 842,80	513,30	513,60	
EECore	2 029,50	9 130,30	2 230,80	4 695,70	
EECore, viive (%)	352,11	56,27	334,60	814,27	389,31
EECore, viive (ms.)	1 580,60	3 287,50	1 717,50	4 182,10	



Kuvio 48: Mittaus 3, UPDATE-2 tulokset palkkikaaviona.

Taulukko 35: Mittaus 3, UPDATE-3 tulokset.

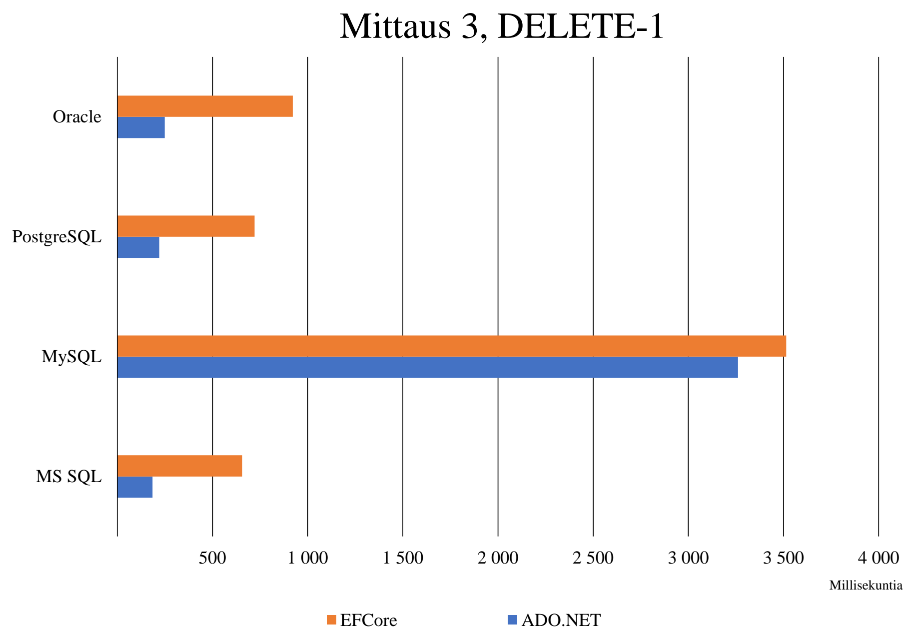
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	633,60	4 625,40	843,20	546,70	
EECore	2 057,20	7 713,20	2 255,50	3 761,10	
EECore, viive (%)	224,68	66,76	167,49	587,96	261,72
EECore, viive (ms.)	1 423,60	3 087,80	1 412,30	3 214,40	



Kuvio 49: Mittaus 3, UPDATE-3 tulokset palkkikaaviona.

Taulukko 36: Mittaus 3, DELETE-1 tulokset.

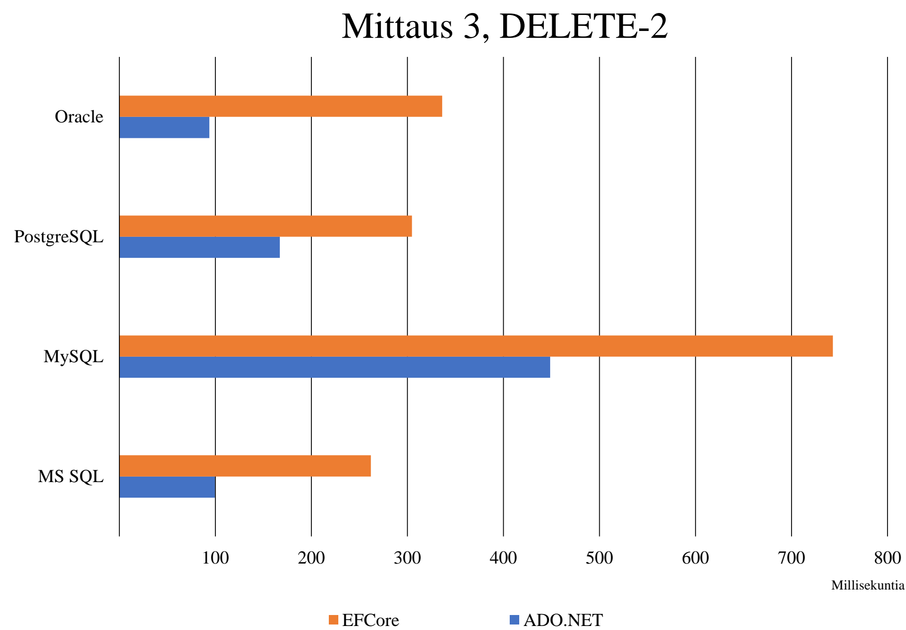
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	184,80	3 261,30	220,00	249,20	
EECore	655,50	3 514,60	721,50	921,60	
EECore, viive (%)	254,71	7,77	227,95	269,82	190,06
EECore, viive (ms.)	470,70	253,30	501,50	672,40	



Kuvio 50: Mittaus 3, DELETE-1 tulokset palkkikaaviona.

Taulukko 37: Mittaus 3, DELETE-2 tulokset.

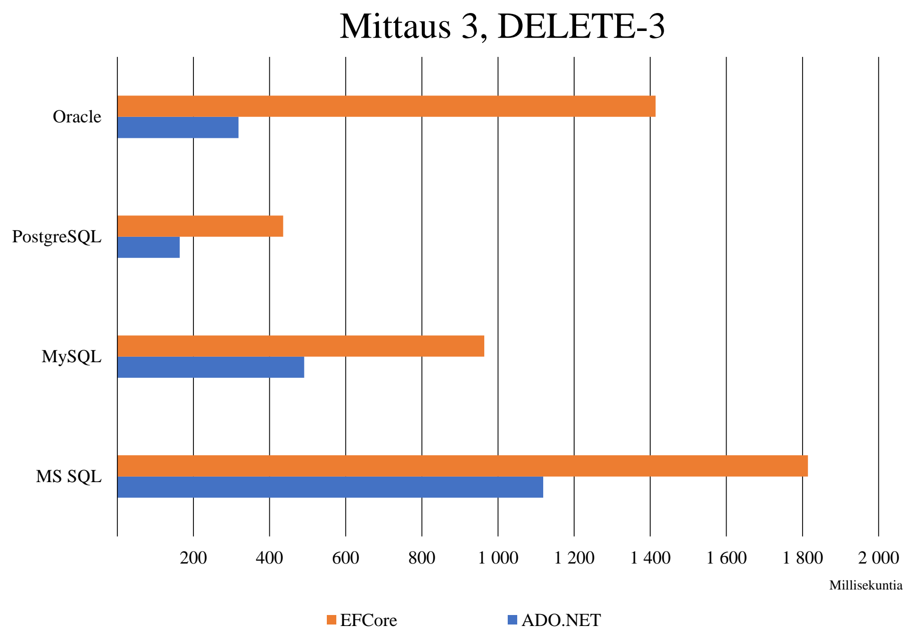
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	99,38	448,60	167,13	93,78	
EECore	261,97	743,09	304,73	336,15	
EECore, viive (%)	163,60	65,65	82,33	258,45	142,51
EECore, viive (ms.)	162,59	294,49	137,60	242,37	



Kuvio 51: Mittaus 3, DELETE-2 tulokset palkkikaaviona.

Taulukko 38: Mittaus 3, DELETE-3 tulokset.

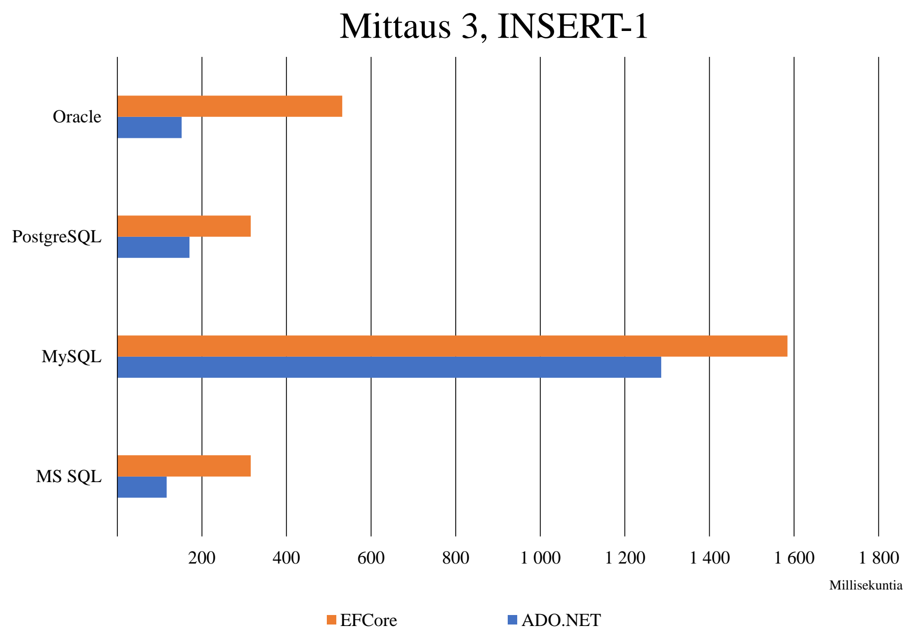
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	1 118,70	490,80	164,00	318,40	
EECore	1 814,20	963,90	435,50	1 414,00	
EECore, viive (%)	62,17	96,39	165,55	344,10	167,05
EECore, viive (ms.)	695,50	473,10	271,50	1 095,60	



Kuvio 52: Mittaus 3, DELETE-3 tulokset palkkikaaviona.

Taulukko 39: Mittaus 3, INSERT-1 tulokset.

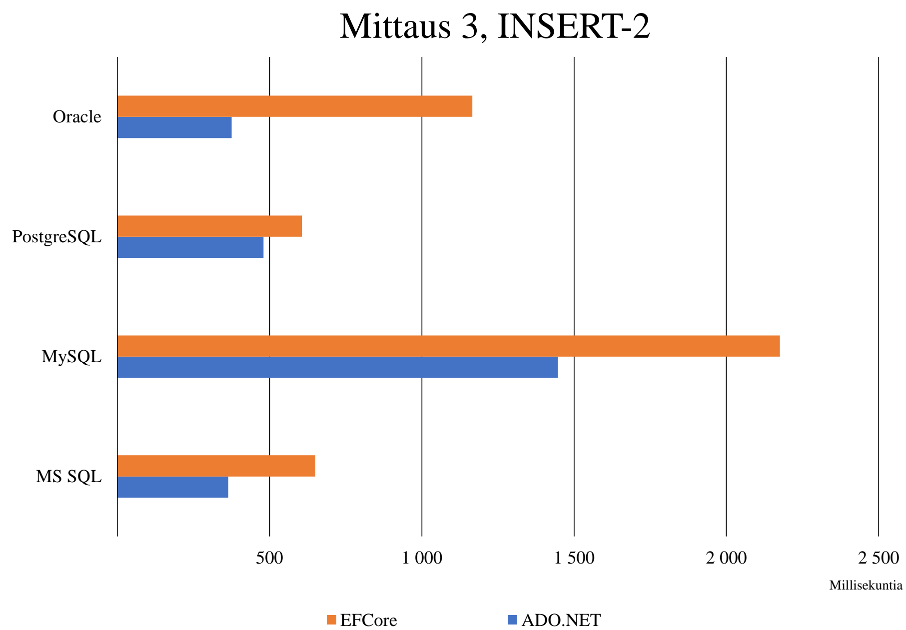
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	116,50	1 285,80	170,60	152,10	
EECore	315,40	1 584,30	315,50	531,80	
EECore, viive (%)	170,73	23,22	84,94	249,64	132,13
EECore, viive (ms.)	198,90	298,50	144,90	379,70	



Kuvio 53: Mittaus 3, INSERT-1 tulokset palkkikaaviona.

Taulukko 40: Mittaus 3, INSERT-2 tulokset.

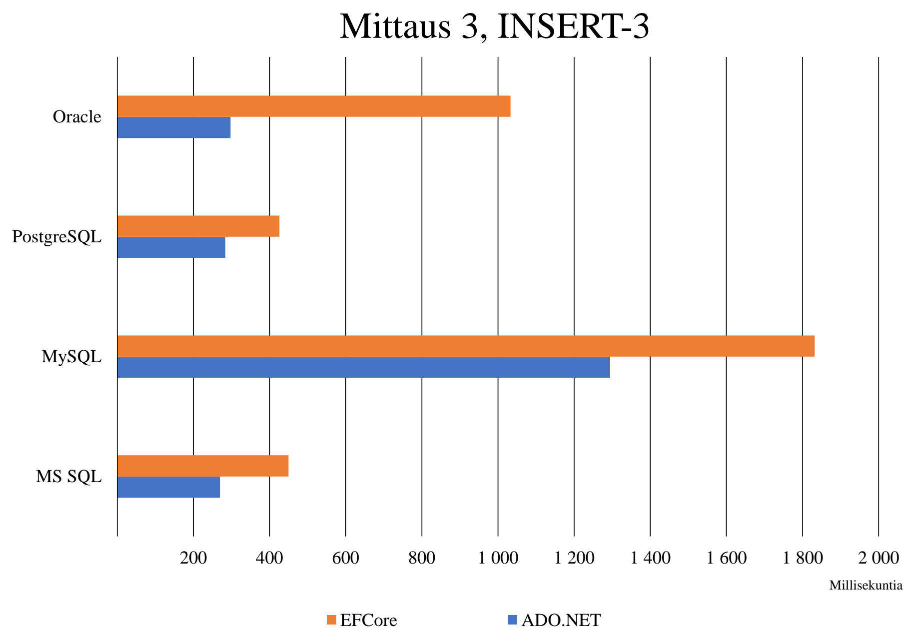
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	364,30	1 446,70	480,00	375,30	
EECore	650,10	2 175,70	605,90	1 165,90	
EECore, viive (%)	78,45	50,39	26,23	210,66	91,43
EECore, viive (ms.)	285,80	729,00	125,90	790,60	



Kuvio 54: Mittaus 3, INSERT-2 tulokset palkkikaaviona.

Taulukko 41: Mittaus 3, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	269,80	1 294,40	283,80	297,50	
EECore	449,70	1 832,00	425,90	1 032,80	
EECore, viive (%)	66,68	41,53	50,07	247,16	101,36
EECore, viive (ms.)	179,90	537,60	142,10	735,30	



Kuvio 55: Mittaus 3, INSERT-3 tulokset palkkikaaviona.

G Mittauspöytäkirja 4, 29.1.2023

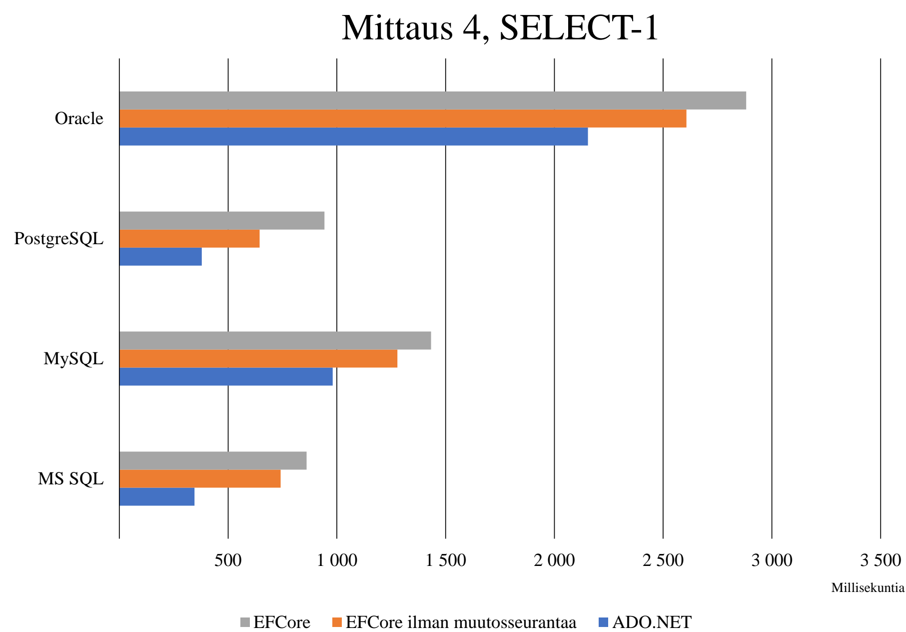
Mittauksen aloitus: 29.1.2023 klo 21:20

Mittauksen päättyminen: 30.1.2023 klo 01:25

Tulokset:

Taulukko 42: Mittaus 4, SELECT-1 tulokset.

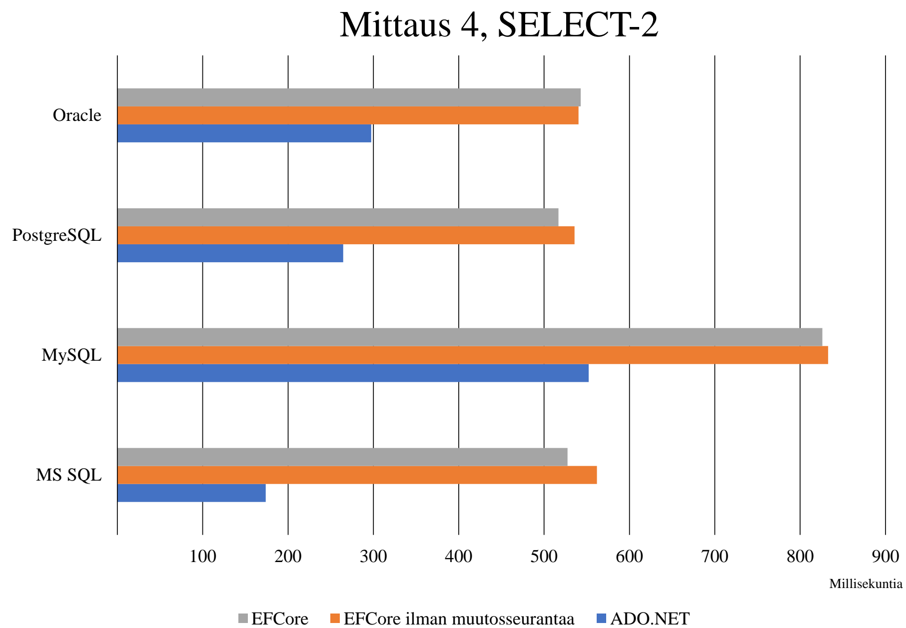
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	345,70	980,60	379,10	2 154,30	
EFCore	860,80	1 433,20	943,30	2 881,60	
EFCore, viive (%)	149,00	46,16	148,83	33,76	94,44
EFCore, viive (ms.)	515,10	452,60	564,20	727,30	
EFCore ilman muutosseurantaa	741,60	1 278,20	644,70	2 607,20	
EFCore ilman muutosseurantaa, viive (%)	114,52	30,35	70,06	21,02	58,99
EFCore ilman muutosseurantaa, viive (ms.)	395,90	297,60	265,60	452,90	



Kuvio 56: Mittaus 4, SELECT-1 tulokset palkkikaaviona.

Taulukko 43: Mittaus 4, SELECT-2 tulokset.

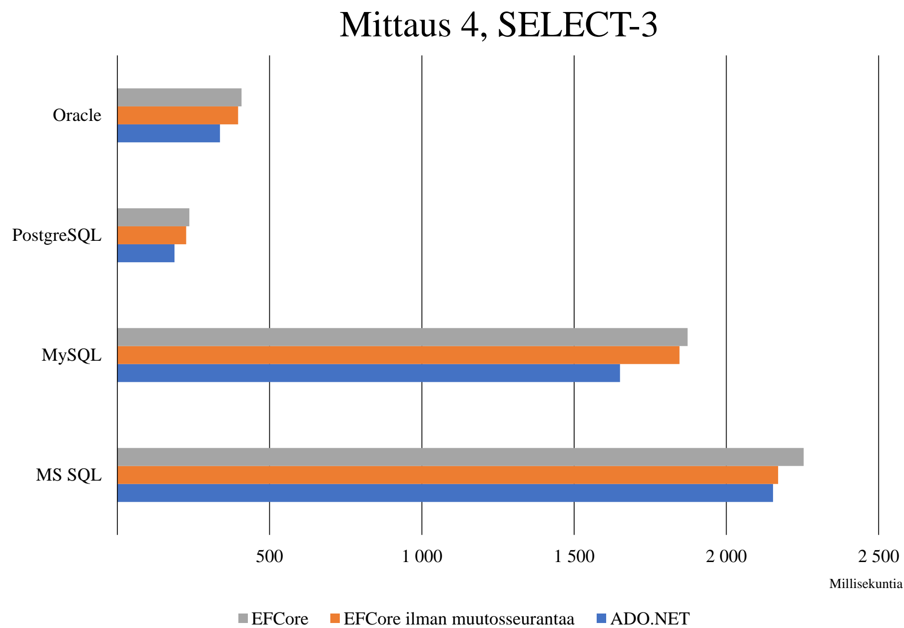
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	173,80	552,20	264,60	297,50	
EFCore	527,50	826,00	516,90	542,90	
EFCore, viive (%)	203,51	49,58	95,35	82,49	107,73
EFCore, viive (ms.)	353,70	273,80	252,30	245,40	
EFCore ilman muutosseurantaa	561,90	832,80	535,60	540,30	
EFCore ilman muutosseurantaa, viive (%)	223,30	50,81	102,42	81,61	114,54
EFCore ilman muutosseurantaa, viive (ms.)	388,10	280,60	271,00	242,80	



Kuvio 57: Mittaus 4, SELECT-2 tulokset palkkikaaviona.

Taulukko 44: Mittaus 4, SELECT-3 tulokset.

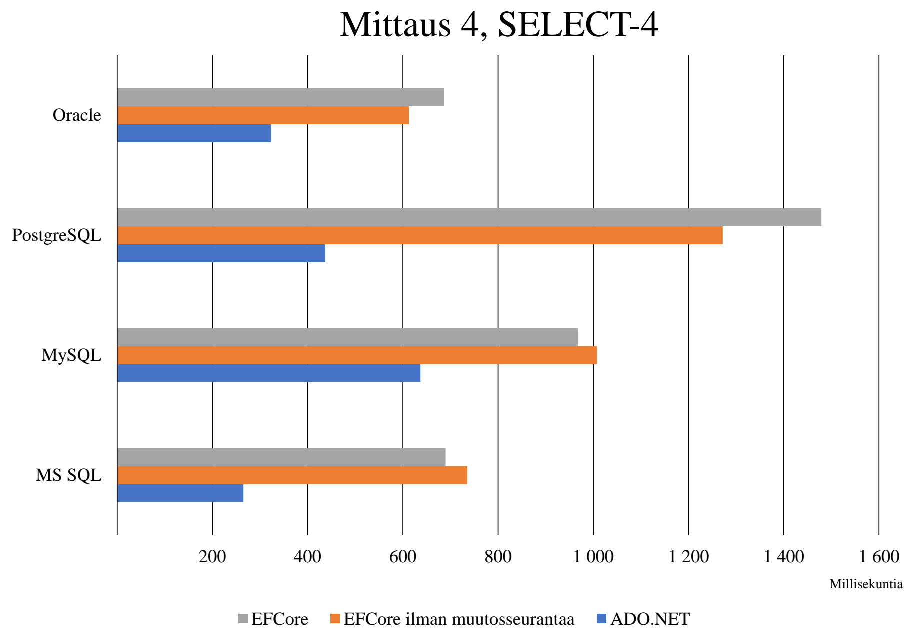
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	2 153,20	1 650,60	187,80	337,10	
EFCore	2 253,90	1 872,70	236,60	408,00	
EFCore, viive (%)	4,68	13,46	25,99	21,03	16,29
EFCore, viive (ms.)	100,70	222,10	48,80	70,90	
EFCore ilman muutosseurantaa	2 170,10	1 845,80	226,10	396,70	
EFCore ilman muutosseurantaa, viive (%)	0,78	11,83	20,39	17,68	12,67
EFCore ilman muutosseurantaa, viive (ms.)	16,90	195,20	38,30	59,60	



Kuvio 58: Mittaus 4, SELECT-3 tulokset palkkikaaviona.

Taulukko 45: Mittaus 4, SELECT-4 tulokset.

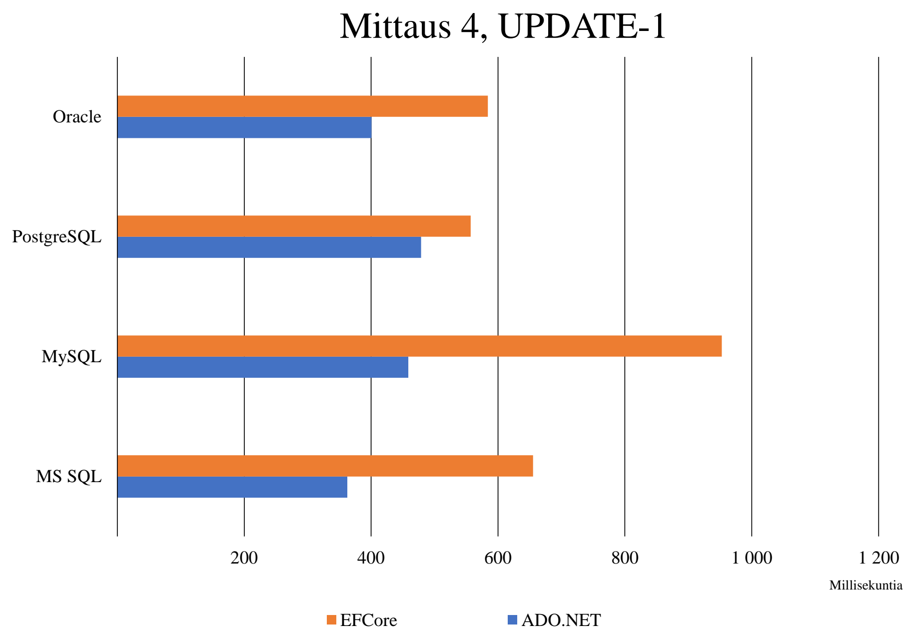
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	265,20	636,80	436,90	323,20	
EFCore	689,70	967,80	1 479,00	686,00	
EFCore, viive (%)	160,07	51,98	238,52	112,25	140,71
EFCore, viive (ms.)	424,50	331,00	1 042,10	362,80	
EFCore ilman muutosseurantaa	735,30	1 007,70	1 271,90	612,80	
EFCore ilman muutosseurantaa, viive (%)	177,26	58,24	191,12	89,60	129,06
EFCore ilman muutosseurantaa, viive (ms.)	470,10	370,90	835,00	289,60	



Kuvio 59: Mittaus 4, SELECT-4 tulokset palkkikaaviona.

Taulukko 46: Mittaus 4, UPDATE-1 tulokset.

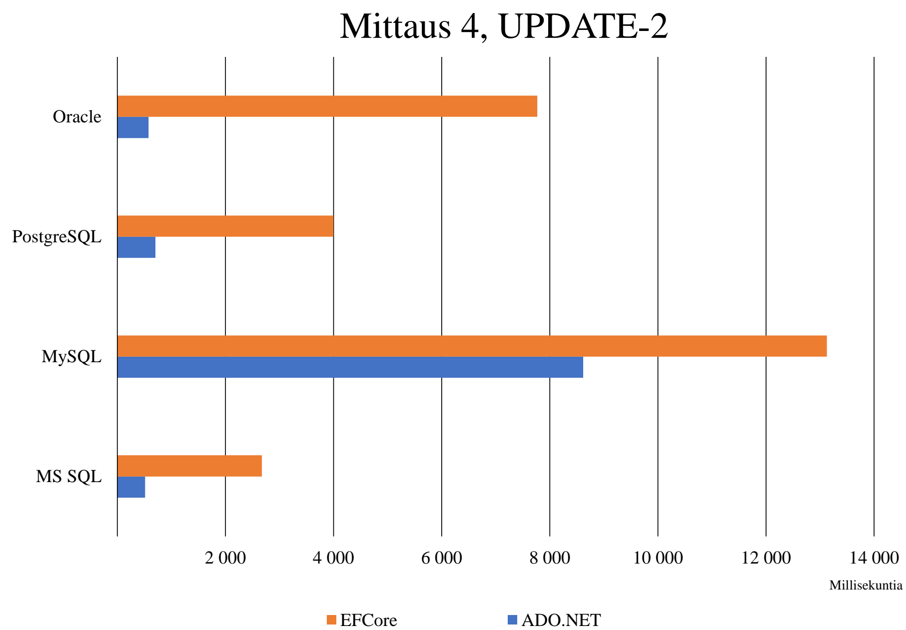
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	362,40	458,60	478,80	400,60	
EECore	655,30	952,70	557,00	584,00	
EECore, viive (%)	80,82	107,74	16,33	45,78	62,67
EECore, viive (ms.)	292,90	494,10	78,20	183,40	



Kuvio 60: Mittaus 4, UPDATE-1 tulokset palkkikaaviona.

Taulukko 47: Mittaus 4, UPDATE-2 tulokset.

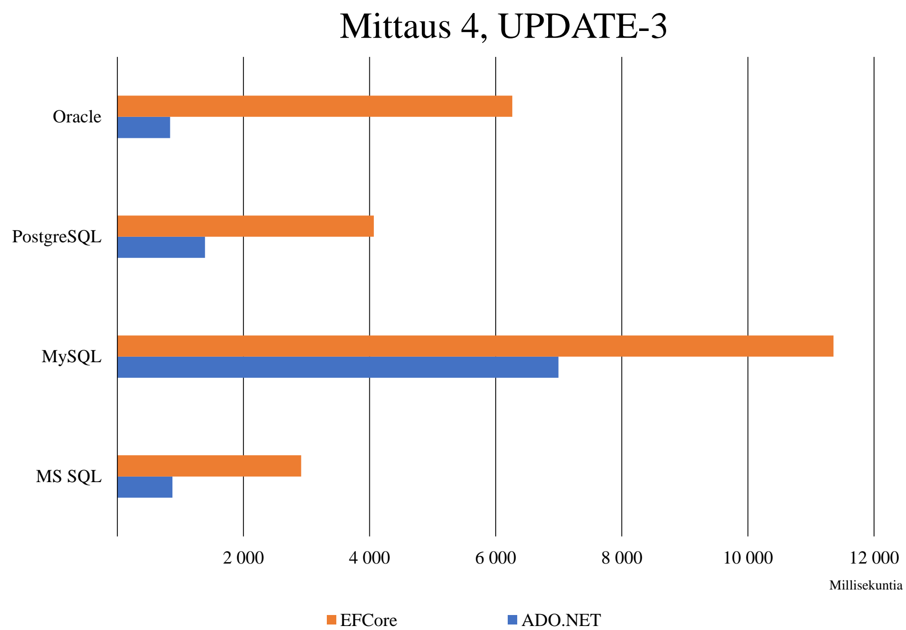
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	514,90	8 618,60	704,80	577,10	
EECore	2 673,70	13 126,70	3 998,90	7 769,20	
EECore, viive (%)	419,27	52,31	467,38	1 246,25	546,30
EECore, viive (ms.)	2 158,80	4 508,10	3 294,10	7 192,10	



Kuvio 61: Mittaus 4, UPDATE-2 tulokset palkkikaaviona.

Taulukko 48: Mittaus 4, UPDATE-3 tulokset.

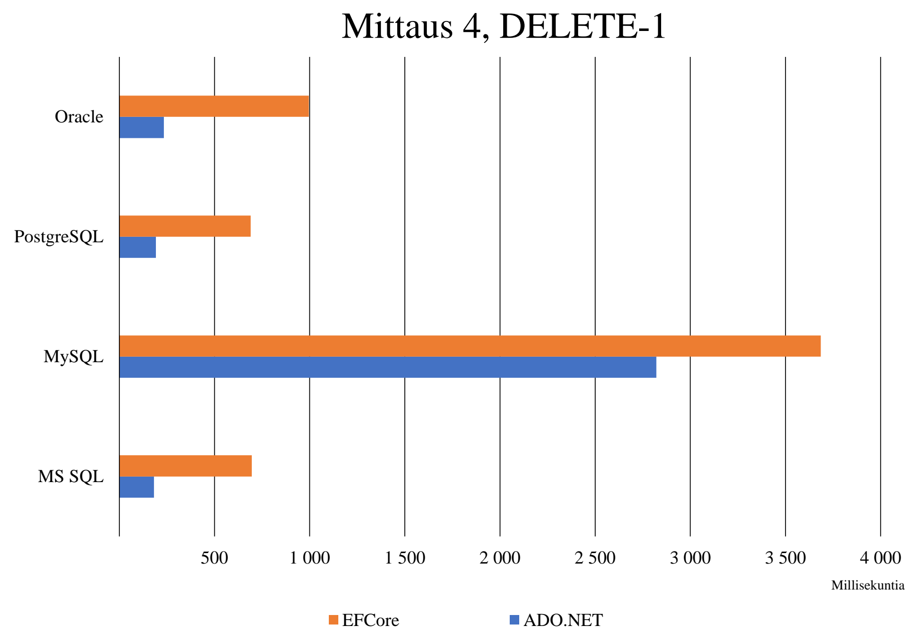
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	873,60	6 995,50	1 390,60	837,40	
EECore	2 915,80	11 355,70	4 066,50	6 262,40	
EECore, viive (%)	233,77	62,33	192,43	647,84	284,09
EECore, viive (ms.)	2 042,20	4 360,20	2 675,90	5 425,00	



Kuvio 62: Mittaus 4, UPDATE-3 tulokset palkkikaaviona.

Taulukko 49: Mittaus 4, DELETE-1 tulokset.

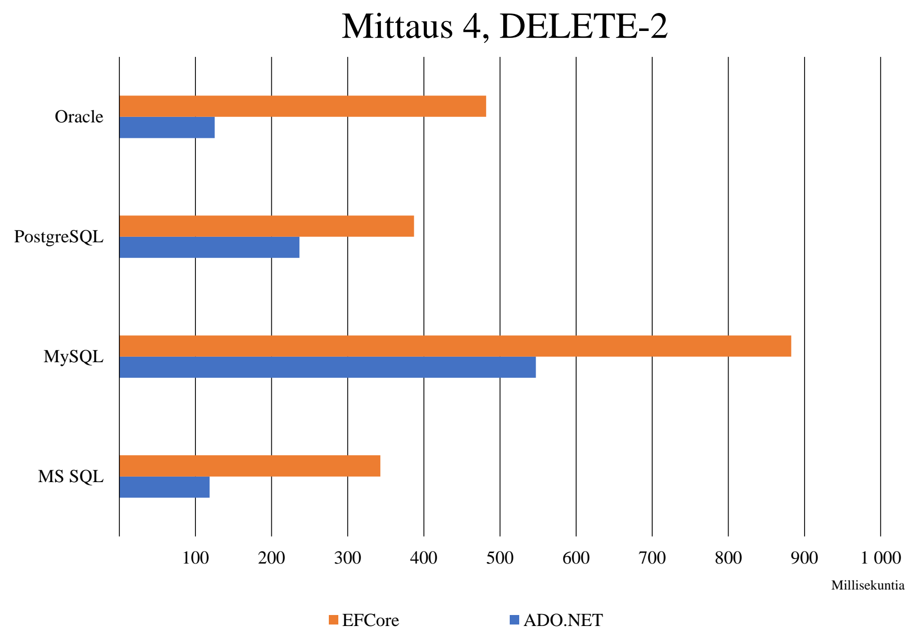
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	182,10	2 821,60	191,90	233,60	
EECore	696,20	3 685,40	689,90	995,20	
EECore, viive (%)	282,32	30,61	259,51	326,03	224,62
EECore, viive (ms.)	514,10	863,80	498,00	761,60	



Kuvio 63: Mittaus 4, DELETE-1 tulokset palkkikaaviona.

Taulukko 50: Mittaus 4, DELETE-2 tulokset.

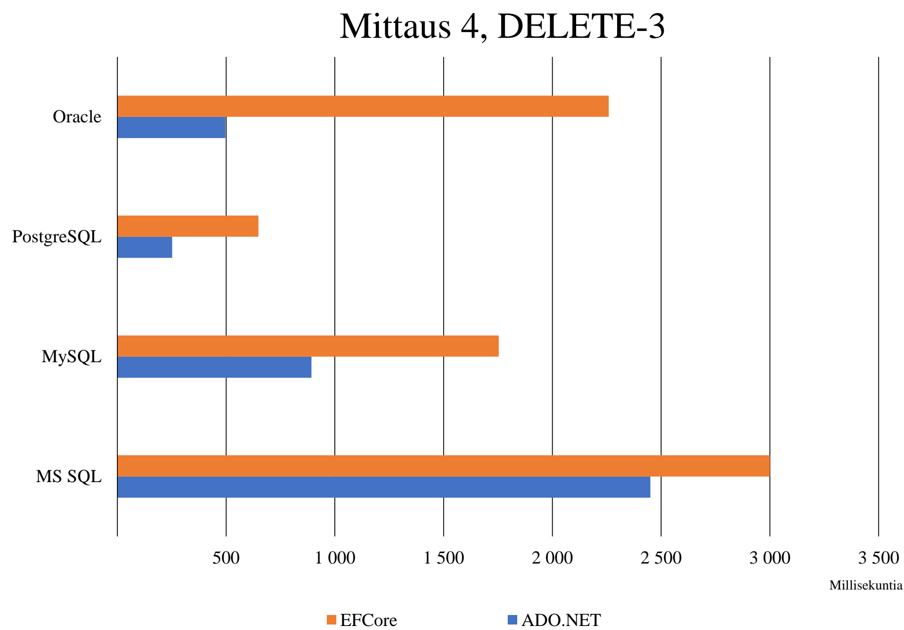
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	118,60	547,20	236,60	125,20	
EECore	342,80	882,60	387,00	481,80	
EECore, viive (%)	189,04	61,29	63,57	284,82	149,68
EECore, viive (ms.)	224,20	335,40	150,40	356,60	



Kuvio 64: Mittaus 4, DELETE-2 tulokset palkkikaaviona.

Taulukko 51: Mittaus 4, DELETE-3 tulokset.

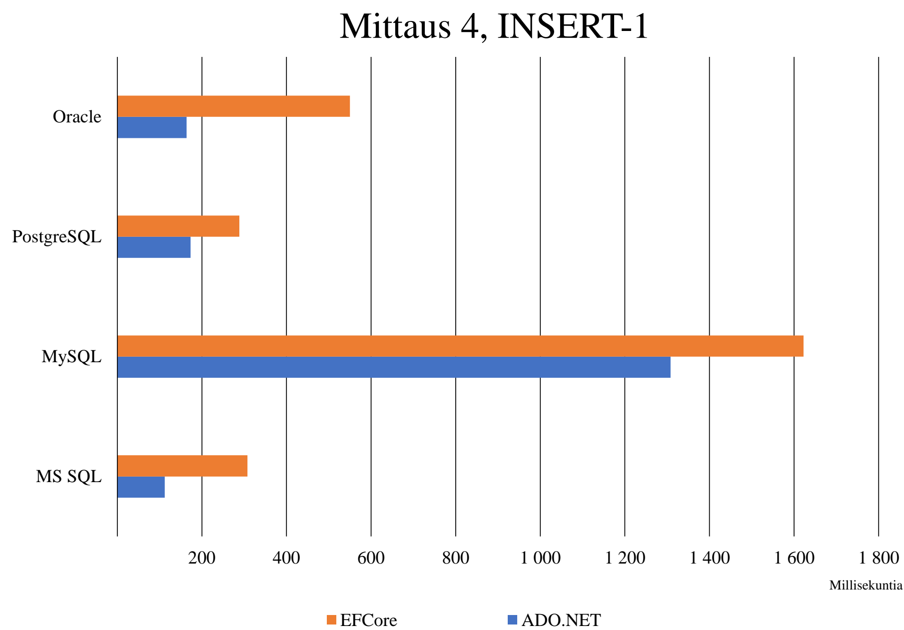
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	2 450,20	892,40	252,10	496,20	
EECore	2 999,80	1 753,20	648,60	2 259,30	
EECore, viive (%)	22,43	96,46	157,28	355,32	157,87
EECore, viive (ms.)	549,60	860,80	396,50	1 763,10	



Kuvio 65: Mittaus 4, DELETE-3 tulokset palkkikaaviona.

Taulukko 52: Mittaus 4, INSERT-1 tulokset.

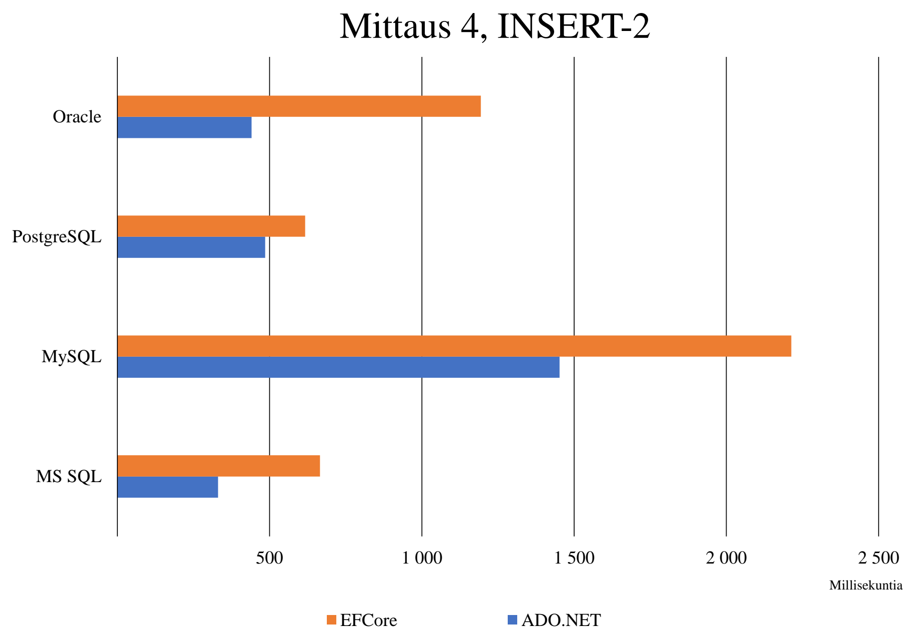
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	112,00	1 308,00	173,00	163,70	
EECore	307,60	1 622,50	288,60	550,00	
EECore, viive (%)	174,64	24,04	66,82	235,98	125,37
EECore, viive (ms.)	195,60	314,50	115,60	386,30	



Kuvio 66: Mittaus 4, INSERT-1 tulokset palkkikaaviona.

Taulukko 53: Mittaus 4, INSERT-2 tulokset.

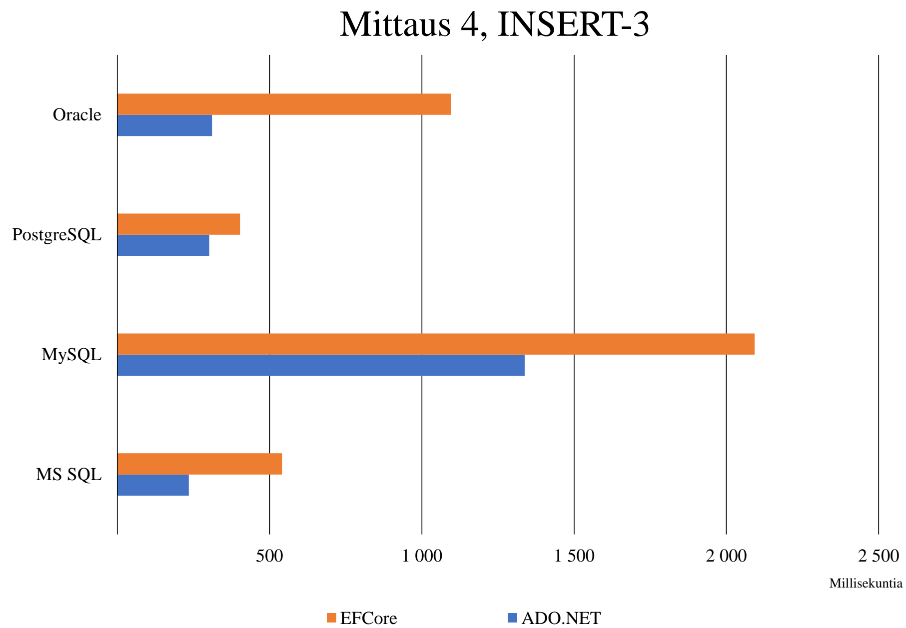
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	330,70	1 452,10	485,60	440,90	
EECore	665,50	2 213,00	616,70	1 193,90	
EECore, viive (%)	101,24	52,40	27,00	170,79	87,86
EECore, viive (ms.)	334,80	760,90	131,10	753,00	



Kuvio 67: Mittaus 4, INSERT-2 tulokset palkkikaaviona.

Taulukko 54: Mittaus 4, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	234,70	1 337,40	301,80	310,80	
EECore	540,80	2 092,90	403,10	1 095,70	
EECore, viive (%)	130,42	56,49	33,57	252,54	118,25
EECore, viive (ms.)	306,10	755,50	101,30	784,90	



Kuvio 68: Mittaus 4, INSERT-3 tulokset palkkikaaviona.

H Mittauspöytäkirja 5, 28.1.2023

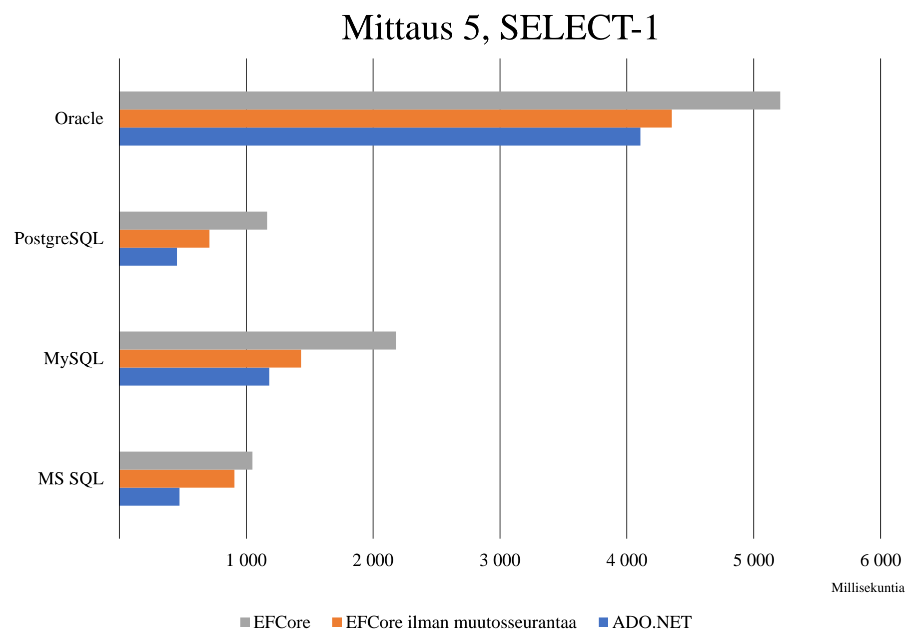
Mittauksen aloitus: 30.1.2023 klo 13:45

Mittauksen päättyminen: 30.1.2023 klo 21:18

Tulokset:

Taulukko 55: Mittaus 5, SELECT-1 tulokset.

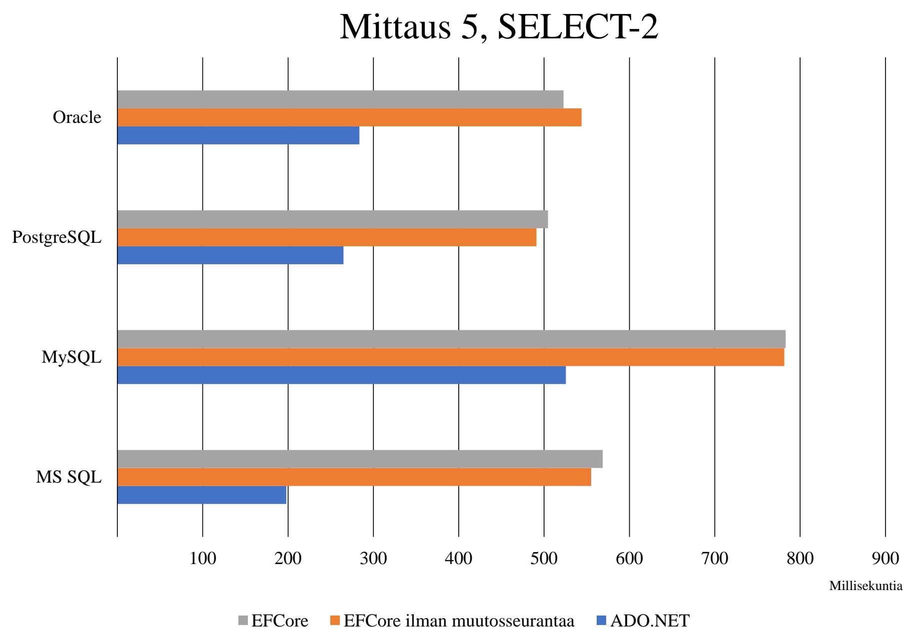
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	474,10	1 182,70	453,60	4 106,50	
EFCore	1 049,50	2 180,00	1 164,90	5 208,40	
EFCore, viive (%)	121,37	84,32	156,81	26,83	97,33
EFCore, viive (ms.)	575,40	997,30	711,30	1 101,90	
EFCore ilman muutosseurantaa	907,90	1 432,00	710,10	4 353,70	
EFCore ilman muutosseurantaa, viive (%)	91,50	21,08	56,55	6,02	43,79
EFCore ilman muutosseurantaa, viive (ms.)	433,80	249,30	256,50	247,20	



Kuvio 69: Mittaus 5, SELECT-1 tulokset palkkikaaviona.

Taulukko 56: Mittaus 5, SELECT-2 tulokset.

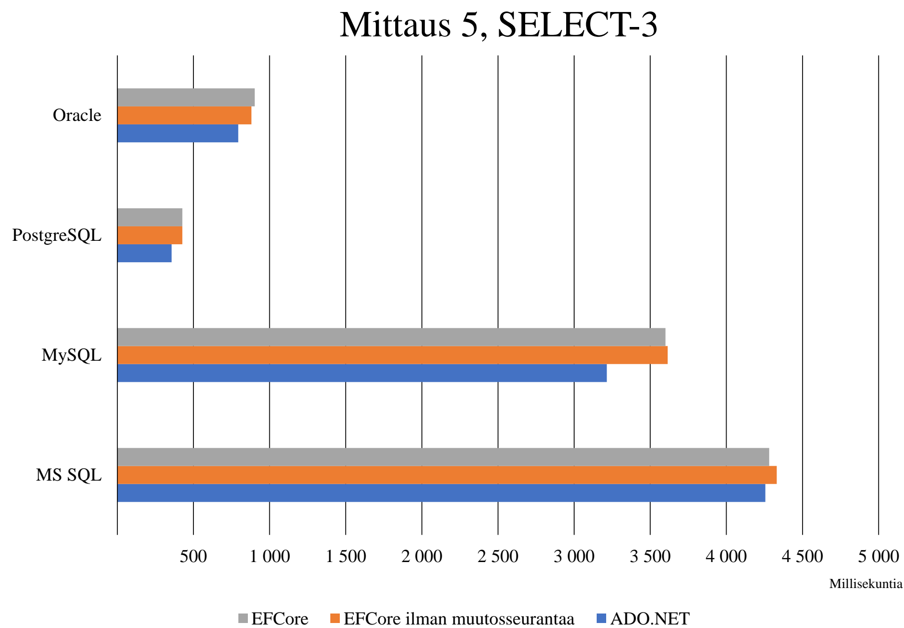
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	198,00	525,50	265,00	283,60	
EFCore	568,60	783,00	504,60	522,80	
EFCore, viive (%)	187,17	49,00	90,42	84,34	102,73
EFCore, viive (ms.)	370,60	257,50	239,60	239,20	
EFCore ilman muutosseurantaa	555,20	781,40	491,00	543,90	
EFCore ilman muutosseurantaa, viive (%)	180,40	48,70	85,28	91,78	101,54
EFCore ilman muutosseurantaa, viive (ms.)	357,20	255,90	226,00	260,30	



Kuvio 70: Mittaus 5, SELECT-2 tulokset palkkikaaviona.

Taulukko 57: Mittaus 5, SELECT-3 tulokset.

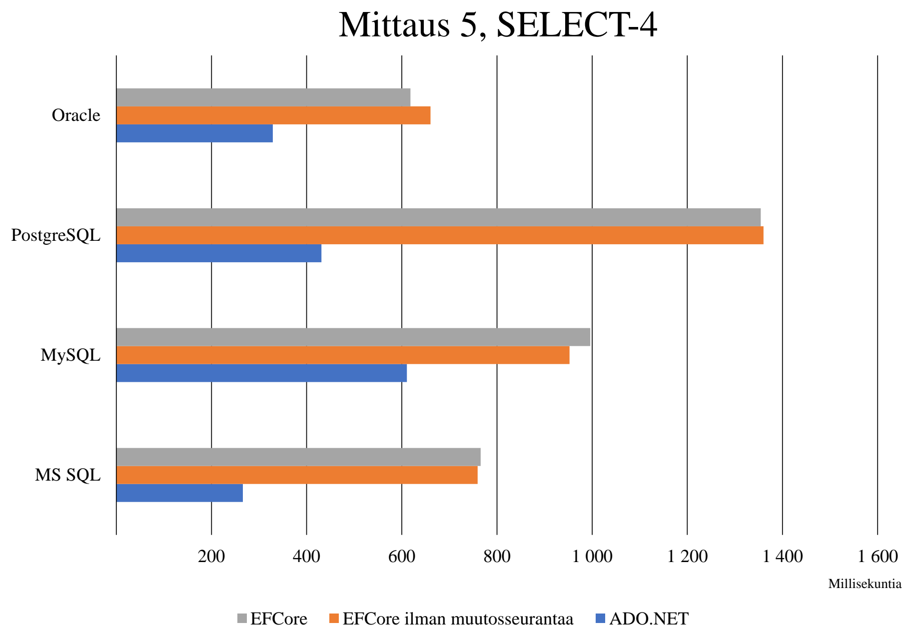
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	4 255,80	3 214,70	356,80	794,20	
EFCore	4 281,30	3 599,60	426,70	902,50	
EFCore, viive (%)	0,60	11,97	19,59	13,64	11,45
EFCore, viive (ms.)	25,50	384,90	69,90	108,30	
EFCore ilman muutosseurantaa	4 331,00	3 614,50	427,30	880,60	
EFCore ilman muutosseurantaa, viive (%)	1,77	12,44	19,76	10,88	11,21
EFCore ilman muutosseurantaa, viive (ms.)	75,20	399,80	70,50	86,40	



Kuvio 71: Mittaus 5, SELECT-3 tulokset palkkikaaviona.

Taulukko 58: Mittaus 5, SELECT-4 tulokset.

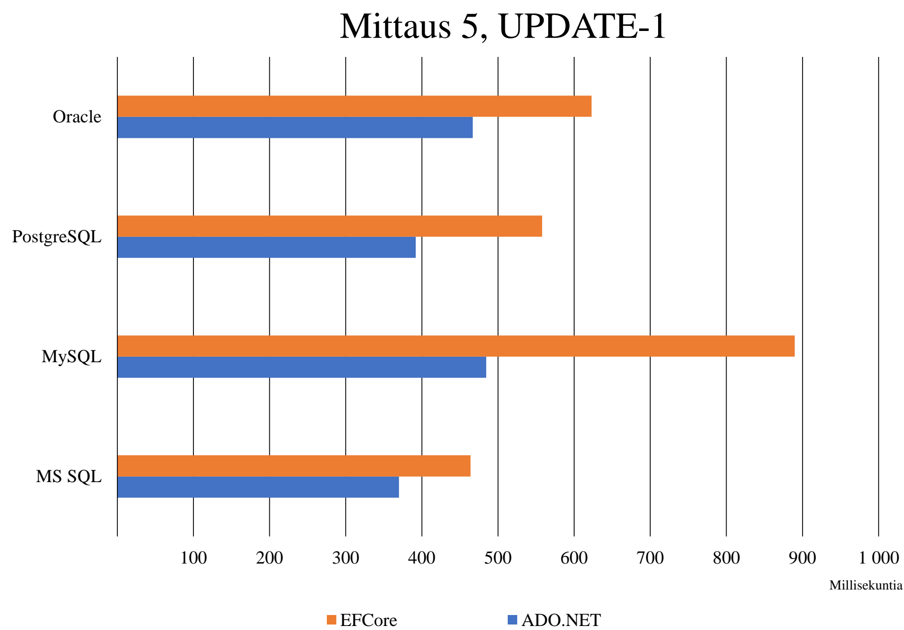
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	266,00	610,80	431,00	329,00	
EFCore	765,80	996,20	1 354,30	618,20	
EFCore, viive (%)	187,89	63,10	214,22	87,90	138,28
EFCore, viive (ms.)	499,80	385,40	923,30	289,20	
EFCore ilman muutosseurantaa	759,20	952,40	1 360,20	660,30	
EFCore ilman muutosseurantaa, viive (%)	185,41	55,93	215,59	100,70	139,41
EFCore ilman muutosseurantaa, viive (ms.)	493,20	341,60	929,20	331,30	



Kuvio 72: Mittaus 5, SELECT-4 tulokset palkkikaaviona.

Taulukko 59: Mittaus 5, UPDATE-1 tulokset.

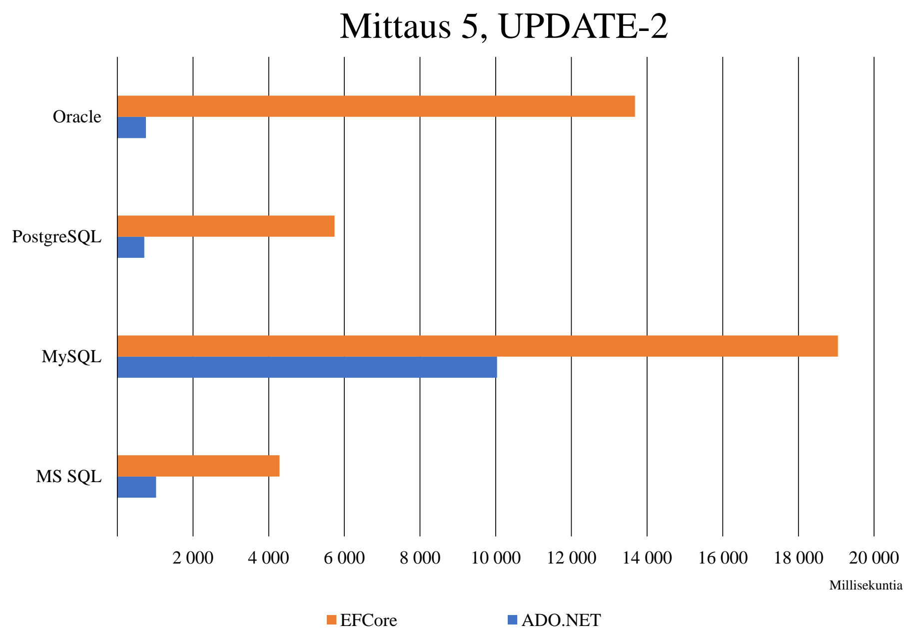
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	369,90	484,60	391,90	466,90	
EECore	464,00	889,70	558,00	623,00	
EECore, viive (%)	25,44	83,59	42,38	33,43	46,21
EECore, viive (ms.)	94,10	405,10	166,10	156,10	



Kuvio 73: Mittaus 5, UPDATE-1 tulokset palkkikaaviona.

Taulukko 60: Mittaus 5, UPDATE-2 tulokset.

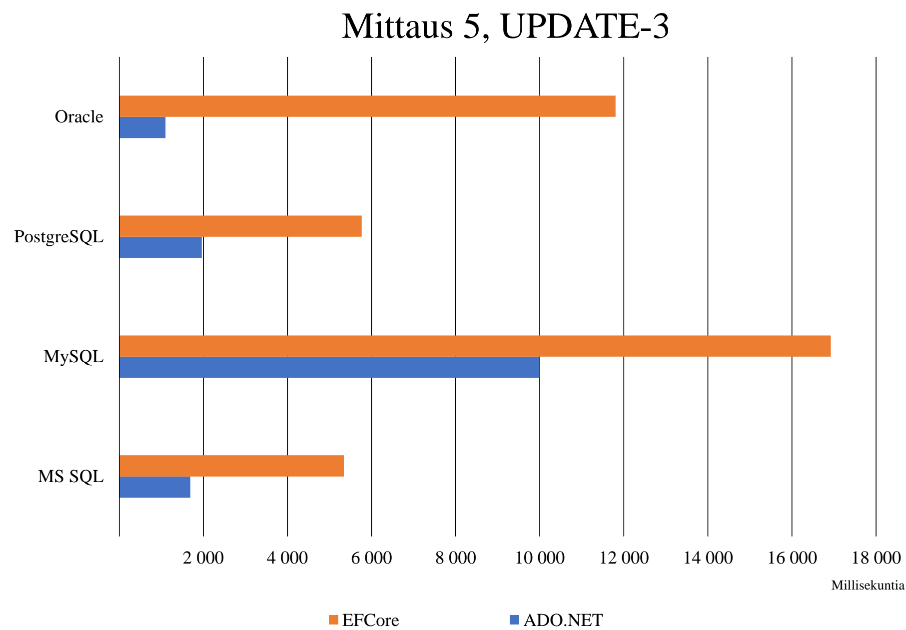
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	1 024,10	10 037,00	710,50	755,80	
EECore	4 284,80	19 042,20	5 739,40	13 680,90	
EECore, viive (%)	318,40	89,72	707,80	1 710,12	706,51
EECore, viive (ms.)	3 260,70	9 005,20	5 028,90	12 925,10	



Kuvio 74: Mittaus 5, UPDATE-2 tulokset palkkikaaviona.

Taulukko 61: Mittaus 5, UPDATE-3 tulokset.

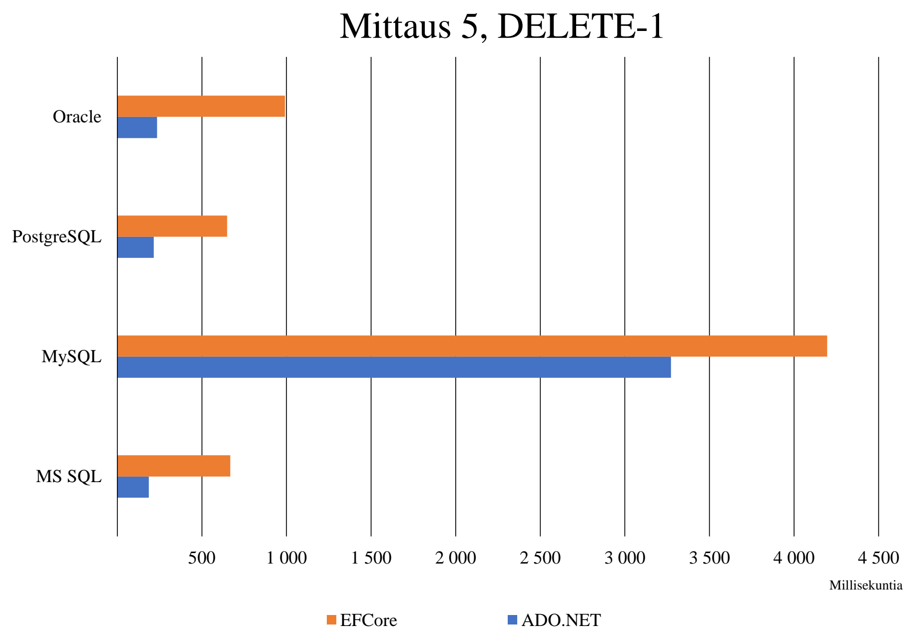
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	1 691,00	9 993,00	1 960,00	1 099,00	
EECore	5 339,00	16 923,00	5 763,00	11 802,00	
EECore, viive (%)	215,73	69,35	194,03	973,89	363,25
EECore, viive (ms.)	3 648,00	6 930,00	3 803,00	10 703,00	



Kuvio 75: Mittaus 5, UPDATE-3 tulokset palkkikaaviona.

Taulukko 62: Mittaus 5, DELETE-1 tulokset.

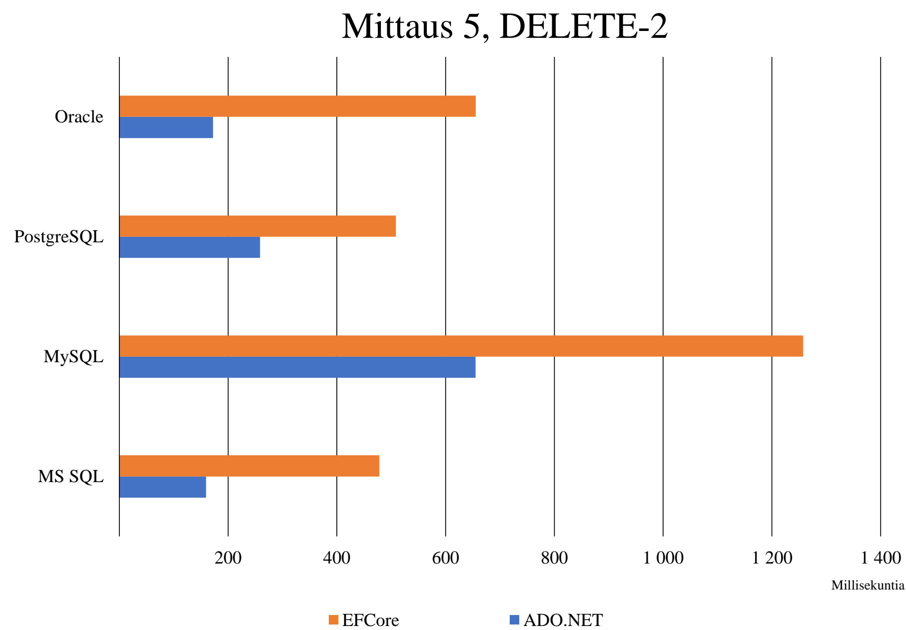
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	186,20	3 272,10	215,50	234,60	
EECore	667,60	4 195,60	649,10	990,90	
EECore, viive (%)	258,54	28,22	201,21	322,38	202,59
EECore, viive (ms.)	481,40	923,50	433,60	756,30	



Kuvio 76: Mittaus 5, DELETE-1 tulokset palkkikaaviona.

Taulukko 63: Mittaus 5, DELETE-2 tulokset.

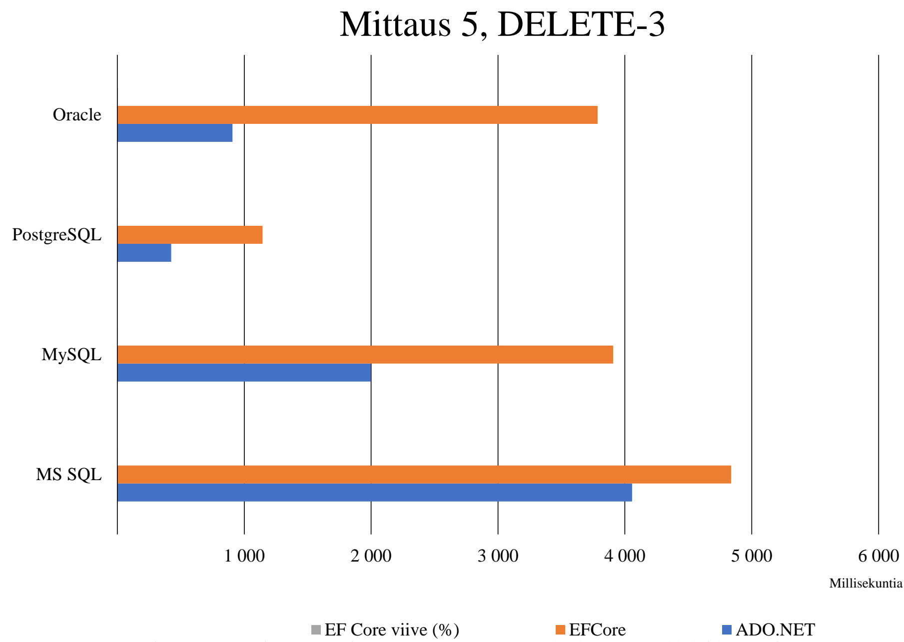
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	159,40	655,00	258,70	172,30	
EECore	478,00	1 257,30	508,60	655,30	
EECore, viive (%)	199,87	91,95	96,60	280,33	167,19
EECore, viive (ms.)	318,60	602,30	249,90	483,00	



Kuvio 77: Mittaus 5, DELETE-2 tulokset palkkikaaviona.

Taulukko 64: Mittaus 5, DELETE-3 tulokset.

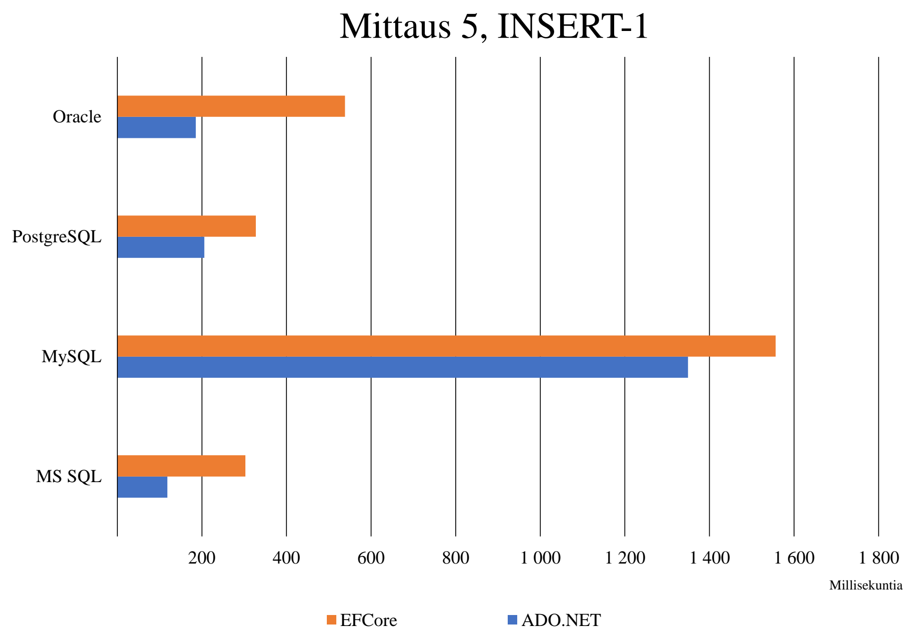
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	4 056,80	1 996,40	424,60	907,30	
EECore	4 837,60	3 907,30	1 144,60	3 785,40	
EECore, viive (%)	19,25	95,72	169,57	317,22	150,44
EECore, viive (ms.)	780,80	1 910,90	720,00	2 878,10	



Kuvio 78: Mittaus 5, DELETE-3 tulokset palkkikaaviona.

Taulukko 65: Mittaus 5, INSERT-1 tulokset.

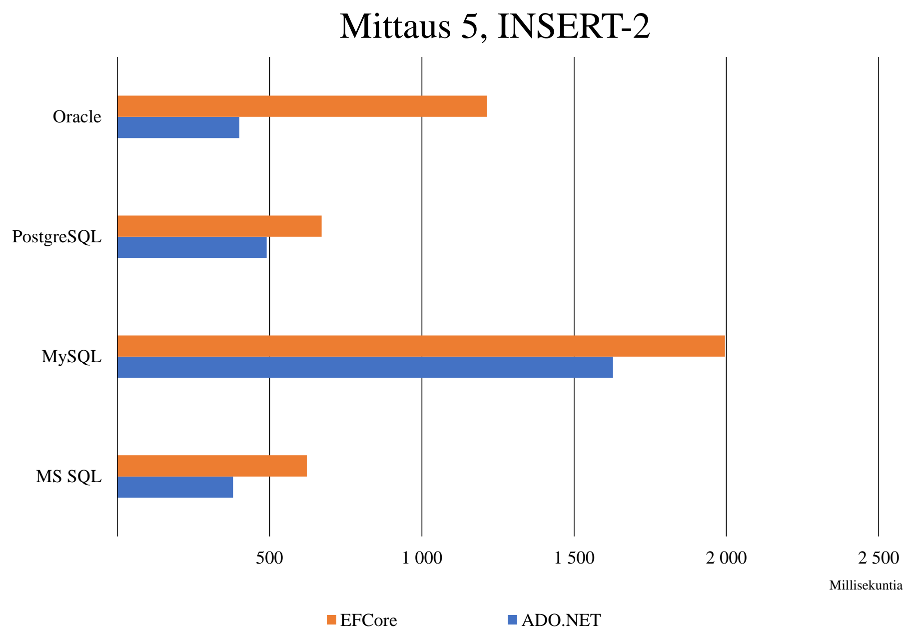
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	118,30	1 349,10	205,80	185,50	
EECore	302,90	1 556,60	327,40	538,30	
EECore, viive (%)	156,04	15,38	59,09	190,19	105,17
EECore, viive (ms.)	184,60	207,50	121,60	352,80	



Kuvio 79: Mittaus 5, INSERT-1 tulokset palkkikaaviona.

Taulukko 66: Mittaus 5, INSERT-2 tulokset.

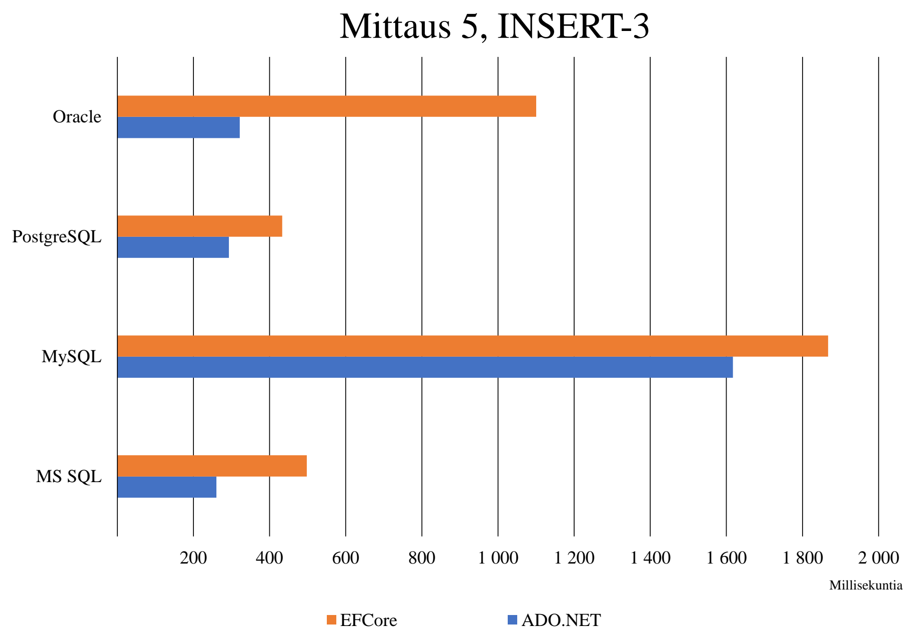
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	380,00	1 627,80	490,20	400,60	
EECore	622,30	1 995,50	670,90	1 214,20	
EECore, viive (%)	63,76	22,59	36,86	203,10	81,58
EECore, viive (ms.)	242,30	367,70	180,70	813,60	



Kuvio 80: Mittaus 5, INSERT-2 tulokset palkkikaaviona.

Taulukko 67: Mittaus 5, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	260,30	1 617,00	293,10	321,60	
EECore	497,70	1 867,30	433,00	1 100,40	
EECore, viive (%)	91,20	15,48	47,73	242,16	99,14
EECore, viive (ms.)	237,40	250,30	139,90	778,80	



Kuvio 81: Mittaus 5, INSERT-3 tulokset palkkikaaviona.

I Mittauspöytäkirja 6, 4.2.2023

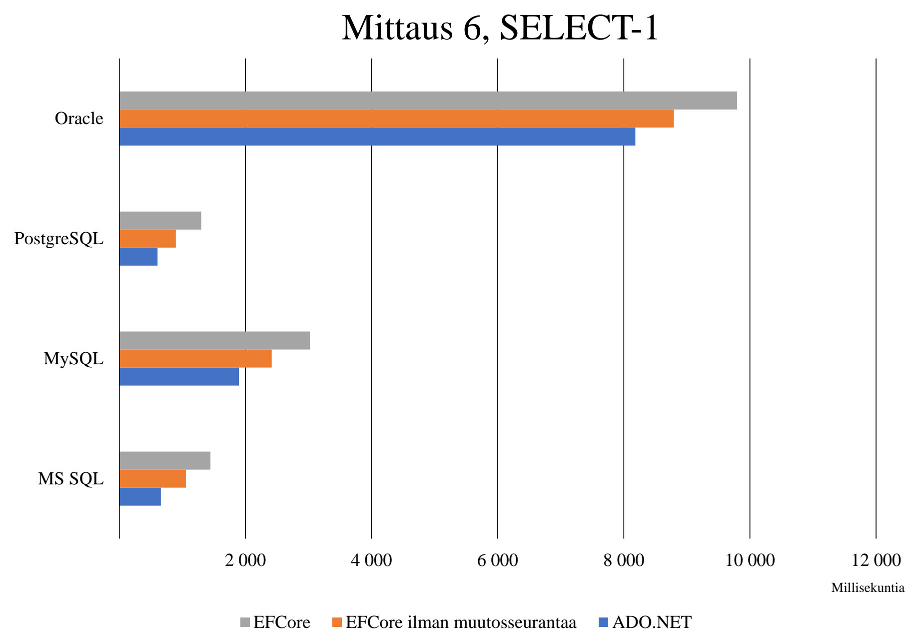
Mittauksen aloitus: 4.2.2023 klo 11:30

Mittauksen päättyminen: 5.2.2023 klo 03:18

Tulokset:

Taulukko 68: Mittaus 6, SELECT-1 tulokset.

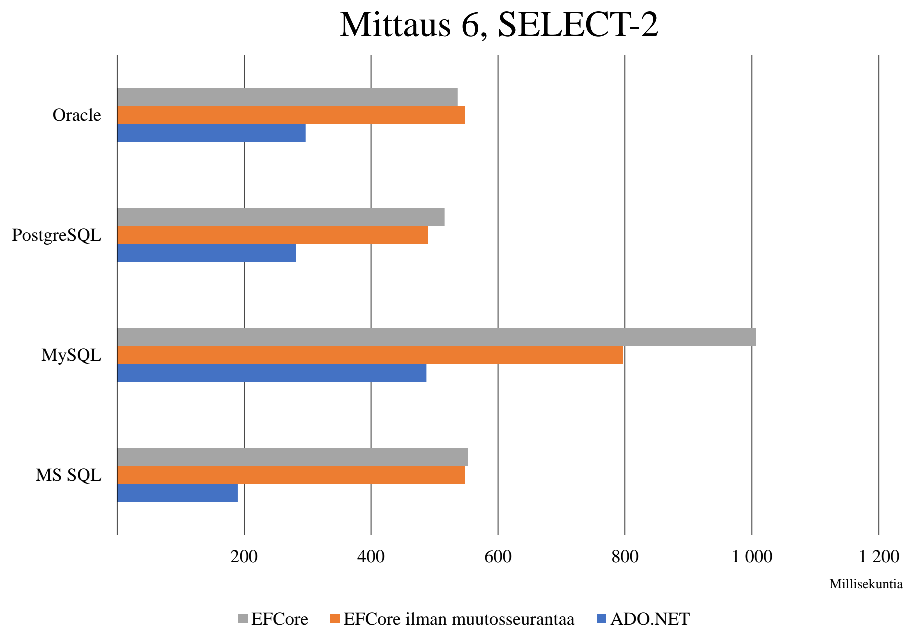
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	657,90	1 896,30	606,00	8 183,00	
EFCore	1 444,70	3 022,80	1 299,50	9 796,90	
EFCore, viive (%)	119,59	59,41	114,44	19,72	78,29
EFCore, viive (ms.)	786,80	1 126,50	693,50	1 613,90	
EFCore ilman muutosseurantaa	1 054,10	2 416,90	896,00	8 793,60	
EFCore ilman muutosseurantaa, viive (%)	60,22	27,45	47,85	7,46	35,75
EFCore ilman muutosseurantaa, viive (ms.)	396,20	520,60	290,00	610,60	



Kuvio 82: Mittaus 6, SELECT-1 tulokset palkkikaaviona.

Taulukko 69: Mittaus 6, SELECT-2 tulokset.

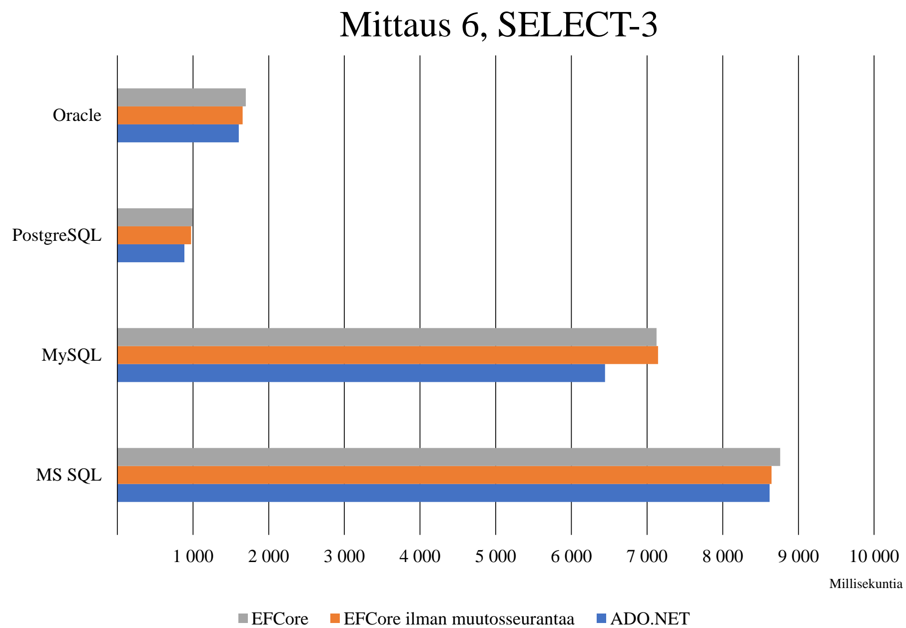
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	190,00	487,20	281,50	297,00	
EFCore	552,50	1 006,60	515,70	536,30	
EFCore, viive (%)	190,79	106,61	83,20	80,57	115,29
EFCore, viive (ms.)	362,50	519,40	234,20	239,30	
EFCore ilman muutosseurantaa	547,60	796,60	489,50	547,90	
EFCore ilman muutosseurantaa, viive (%)	188,21	63,51	73,89	84,48	102,52
EFCore ilman muutosseurantaa, viive (ms.)	357,60	309,40	208,00	250,90	



Kuvio 83: Mittaus 6, SELECT-2 tulokset palkkikaaviona.

Taulukko 70: Mittaus 6, SELECT-3 tulokset.

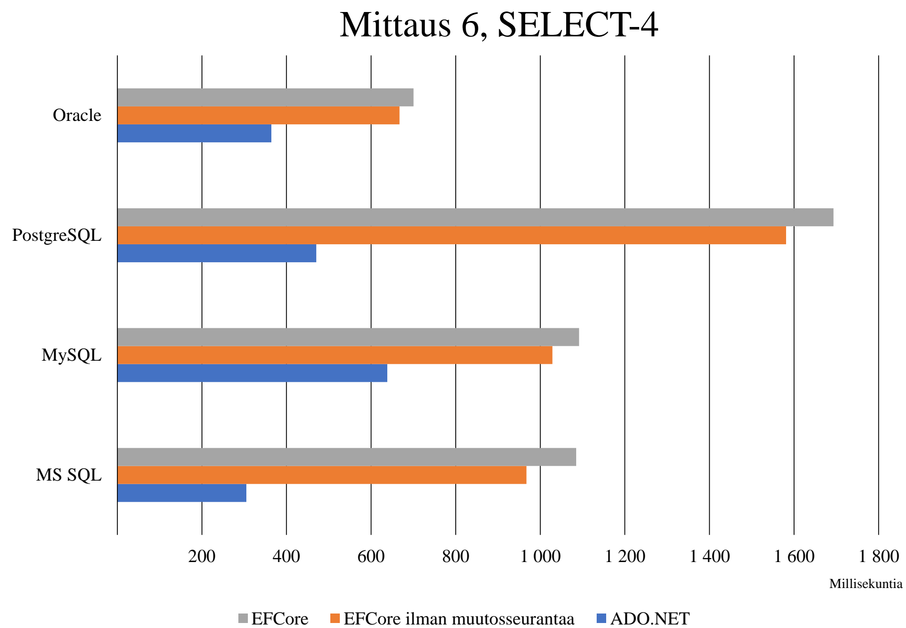
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	8 619,30	6 444,60	885,90	1 604,90	
EFCore	8 759,40	7 124,70	991,70	1 697,60	
EFCore, viive (%)	1,63	10,55	11,94	5,78	7,47
EFCore, viive (ms.)	140,10	680,10	105,80	92,70	
EFCore ilman muutosseurantaa	8 644,30	7 145,40	975,30	1 655,60	
EFCore ilman muutosseurantaa, viive (%)	0,29	10,87	10,09	3,16	6,10
EFCore ilman muutosseurantaa, viive (ms.)	25,00	700,80	89,40	50,70	



Kuvio 84: Mittaus 6, SELECT-3 tulokset palkkikaaviona.

Taulukko 71: Mittaus 6, SELECT-4 tulokset.

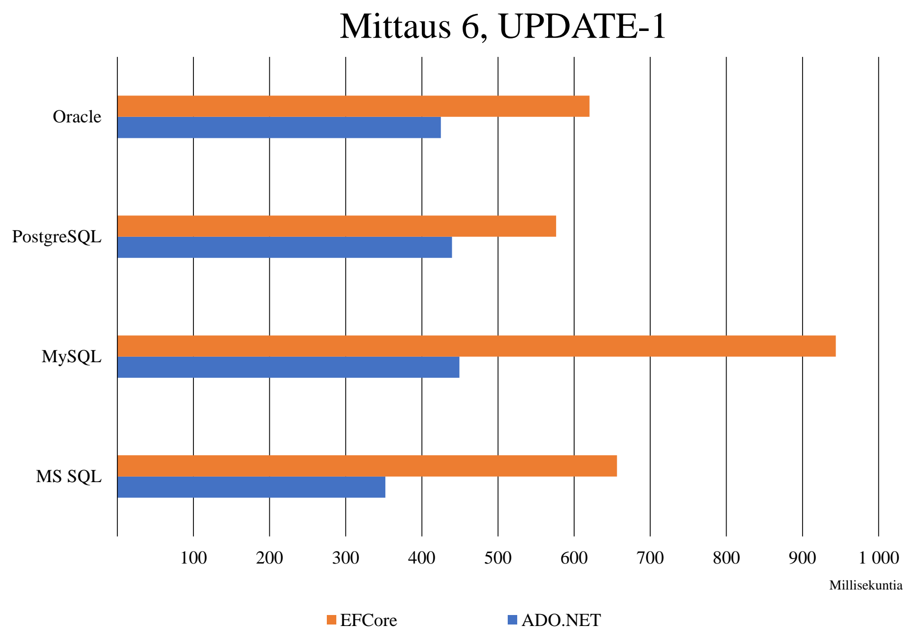
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	305,10	638,20	470,40	364,10	
EFCore	1 084,90	1 091,60	1 693,30	700,20	
EFCore, viive (%)	255,59	71,04	259,97	92,31	169,73
EFCore, viive (ms.)	779,80	453,40	1 222,90	336,10	
EFCore ilman muutosseurantaa	967,30	1 028,70	1 581,10	667,10	
EFCore ilman muutosseurantaa, viive (%)	217,04	61,19	236,12	83,22	149,39
EFCore ilman muutosseurantaa, viive (ms.)	662,20	390,50	1 110,70	303,00	



Kuvio 85: Mittaus 6, SELECT-4 tulokset palkkikaaviona.

Taulukko 72: Mittaus 6, UPDATE-1 tulokset.

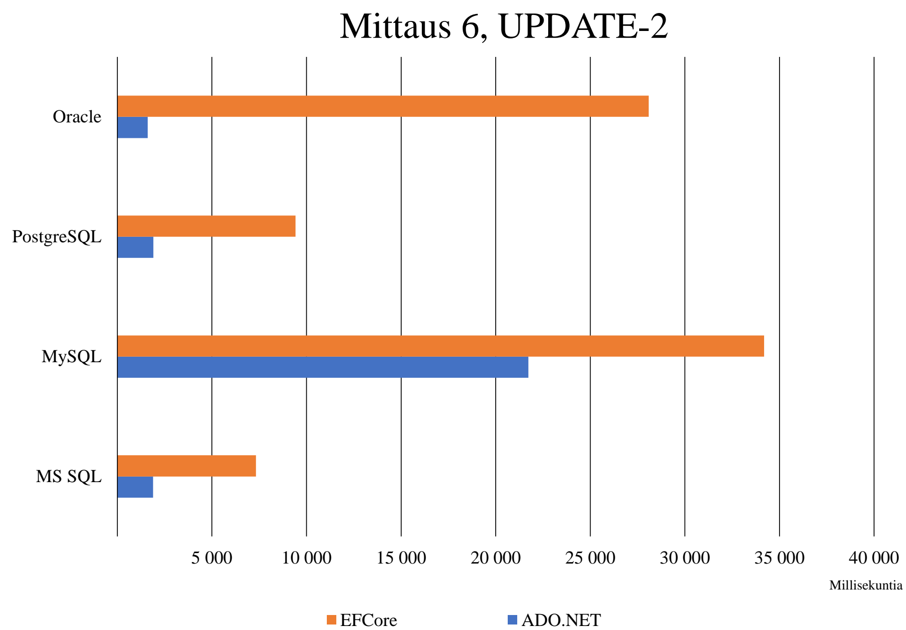
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	352,10	449,40	439,50	424,90	
EECore	656,30	943,70	576,40	620,20	
EECore, viive (%)	86,40	109,99	31,15	45,96	68,37
EECore, viive (ms.)	304,20	494,30	136,90	195,30	



Kuvio 86: Mittaus 6, UPDATE-1 tulokset palkkikaaviona.

Taulukko 73: Mittaus 6, UPDATE-2 tulokset.

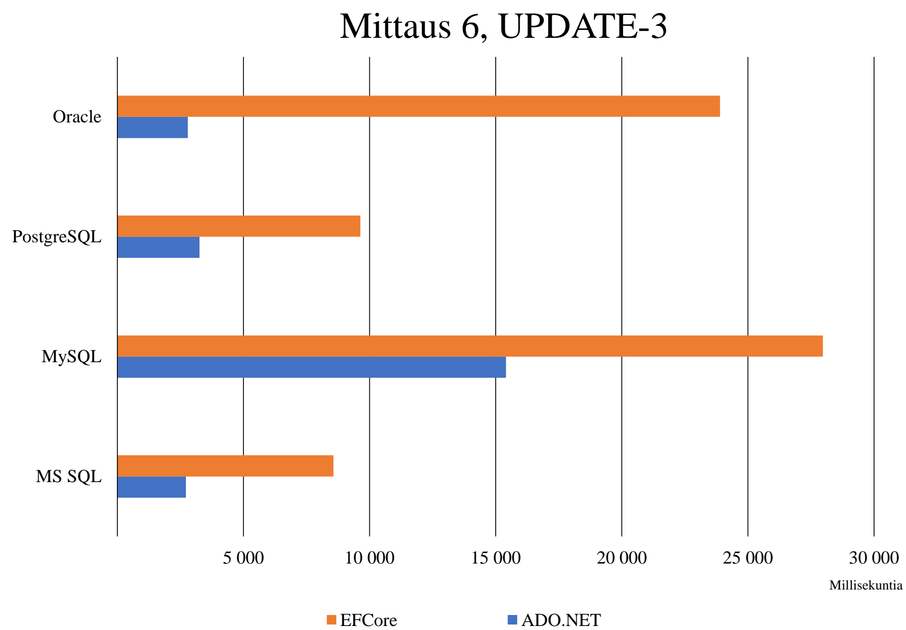
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	1 890,00	21 725,00	1 903,00	1 606,00	
EECore	7 326,00	34 186,00	9 417,00	28 086,00	
EECore, viive (%)	287,62	57,36	394,85	1 648,82	597,16
EECore, viive (ms.)	5 436,00	12 461,00	7 514,00	26 480,00	



Kuvio 87: Mittaus 6, UPDATE-2 tulokset palkkikaaviona.

Taulukko 74: Mittaus 6, UPDATE-3 tulokset.

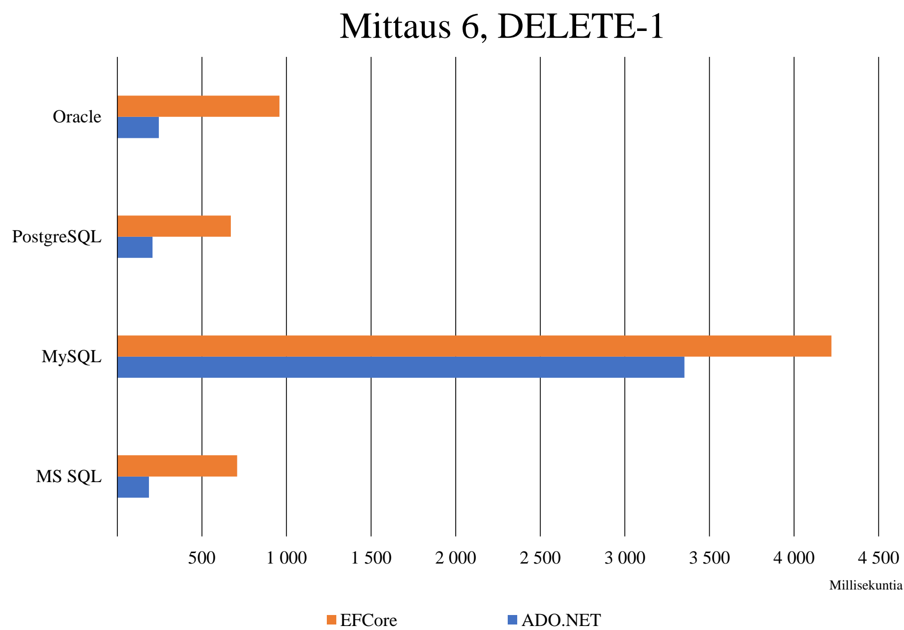
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	2 720,00	15 406,00	3 259,00	2 796,00	
EECore	8 570,00	27 974,00	9 635,00	23 890,00	
EECore, viive (%)	215,07	81,58	195,64	754,43	311,68
EECore, viive (ms.)	5 850,00	12 568,00	6 376,00	21 094,00	



Kuvio 88: Mittaus 6, UPDATE-3 tulokset palkkikaaviona.

Taulukko 75: Mittaus 6, DELETE-1 tulokset.

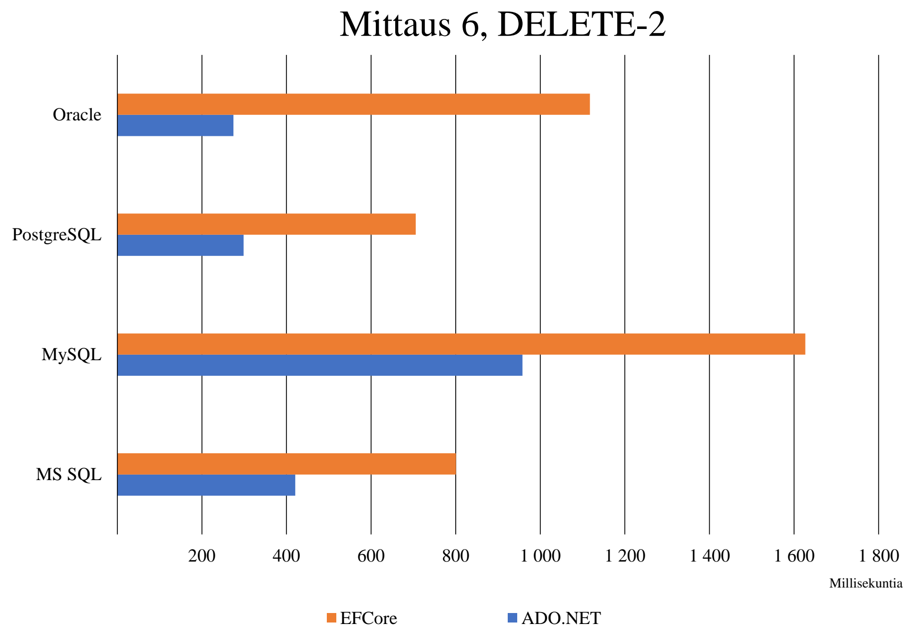
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	186,90	3 351,90	207,80	245,30	
EECore	707,80	4 221,20	671,20	957,90	
EECore, viive (%)	278,71	25,93	223,00	290,50	204,54
EECore, viive (ms.)	520,90	869,30	463,40	712,60	



Kuvio 89: Mittaus 6, DELETE-1 tulokset palkkikaaviona.

Taulukko 76: Mittaus 6, DELETE-2 tulokset.

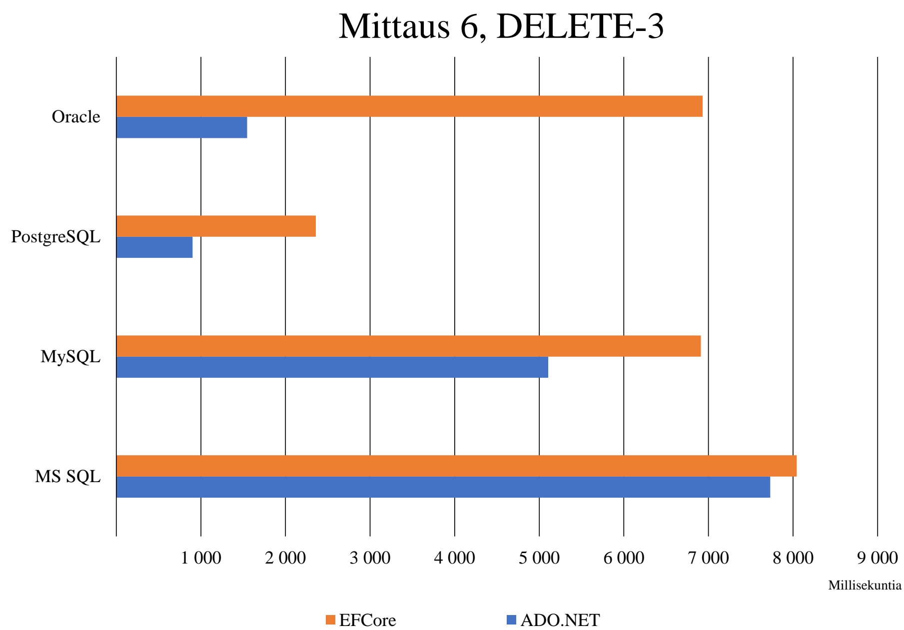
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	420,70	957,90	298,50	274,60	
EECore	800,80	1 626,70	705,40	1 117,30	
EECore, viive (%)	90,35	69,82	136,31	306,88	150,84
EECore, viive (ms.)	380,10	668,80	406,90	842,70	



Kuvio 90: Mittaus 6, DELETE-2 tulokset palkkikaaviona.

Taulukko 77: Mittaus 6, DELETE-3 tulokset.

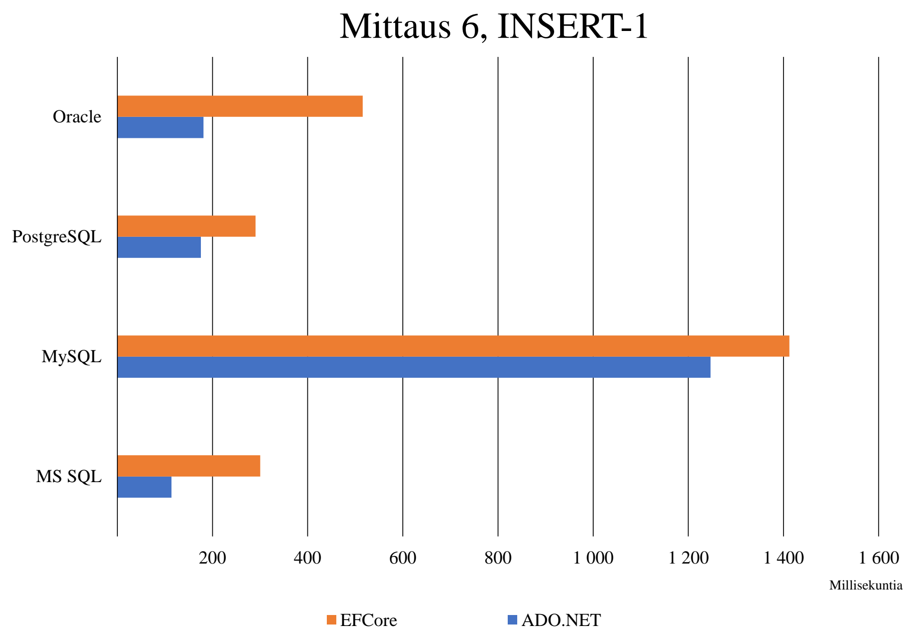
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	7 731,00	5 106,30	901,50	1 546,80	
EECore	8 043,10	6 910,80	2 358,40	6 932,10	
EECore, viive (%)	4,04	35,34	161,61	348,16	137,29
EECore, viive (ms.)	312,10	1 804,50	1 456,90	5 385,30	



Kuvio 91: Mittaus 6, DELETE-3 tulokset palkkikaaviona.

Taulukko 78: Mittaus 6, INSERT-1 tulokset.

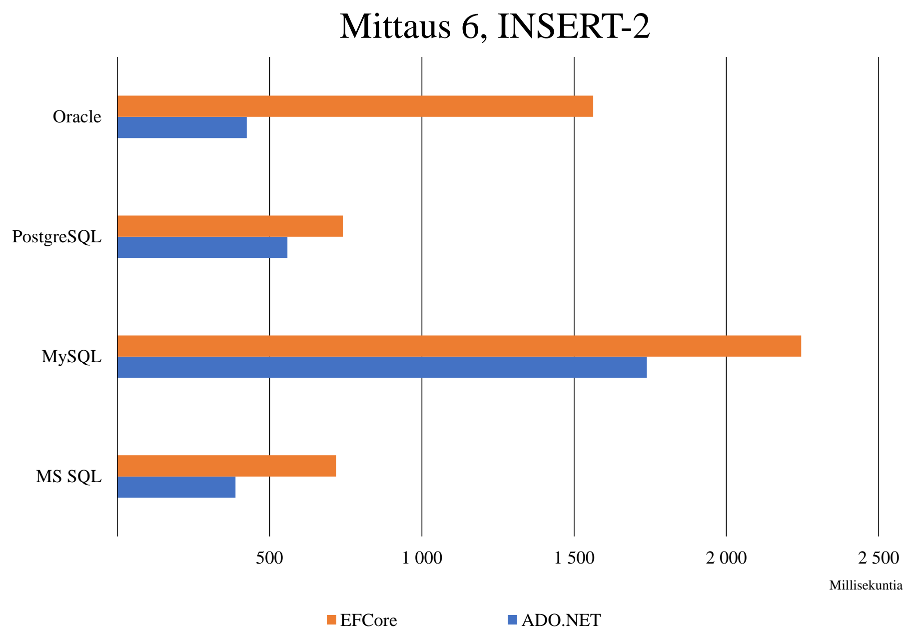
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	113,70	1 246,60	175,70	181,00	
EECore	300,30	1 412,50	290,60	515,60	
EECore, viive (%)	164,12	13,31	65,40	184,86	106,92
EECore, viive (ms.)	186,60	165,90	114,90	334,60	



Kuvio 92: Mittaus 6, INSERT-1 tulokset palkkikaaviona.

Taulukko 79: Mittaus 6, INSERT-2 tulokset.

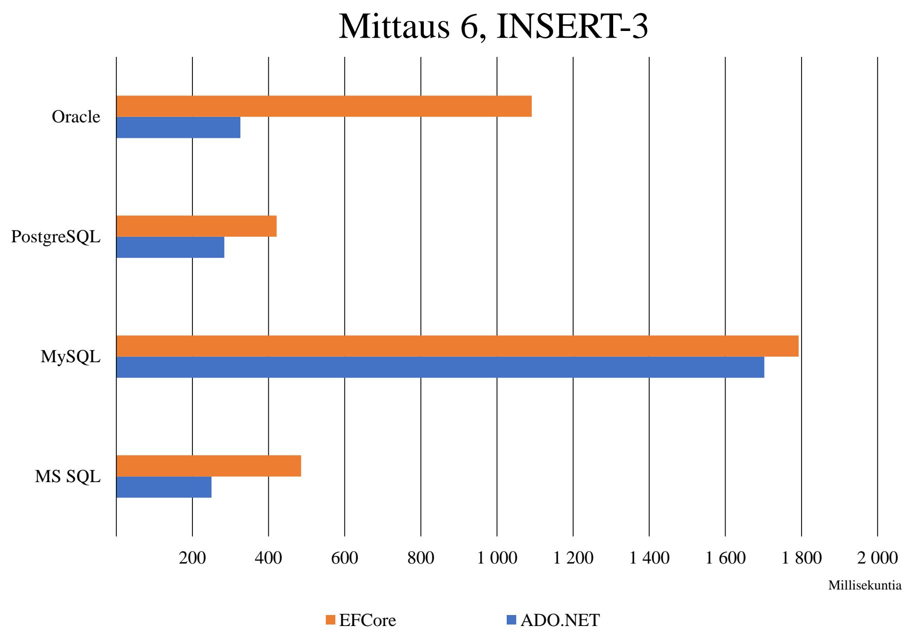
	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	387,90	1 738,80	558,60	425,10	
EECore	718,40	2 245,70	740,30	1 562,80	
EECore, viive (%)	85,20	29,15	32,53	267,63	103,63
EECore, viive (ms.)	330,50	506,90	181,70	1 137,70	



Kuvio 93: Mittaus 6, INSERT-2 tulokset palkkikaaviona.

Taulukko 80: Mittaus 6, INSERT-3 tulokset.

	MS SQL	MySQL	PostgreSQL	Oracle	Keskiarvo
ADO.NET	250,00	1 702,50	283,70	326,10	
EECore	485,30	1 792,40	421,10	1 091,50	
EECore, viive (%)	94,12	5,28	48,43	234,71	95,64
EECore, viive (ms.)	235,30	89,90	137,40	765,40	



Kuvio 94: Mittaus 6, INSERT-3 tulokset palkkikaaviona.