Examensarbete

HÖGSKOLAN I SKÖVDE

LÄSA OCH LAGRA DATA I JSON FORMAT FÖR SMART SENSOR

En jämförelse i svarstid mellan hybriddatabassystemet PostgreSQL och MongoDB

READ AND STORE DATA IN JSON FORMAT FOR SMART SENSOR

A comparison in response time between the hybrid database PostgreSQL and MongoDB

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2018

Fredrik Edman

Handledare: Henrik Gustavsson
Examinator: Gunnar Mathiason

Sammanfattning

Sociala media genererar stora mängder data men det finns fler saker som gör det och lagrar i NoSQL databassystem och smarta sensorer som registrerar elektrisk förbrukning är en av de. MongoDB är ett NoSQL databassystem som lagrar sin data i dataformatet JSONB. PostgreSQL som är ett SQL databassystem har i sina senare distributioner också börjat hantera JSONB. Det gör att PostgreSQL är en typ av hybrid då den hanterar operationer för både SQL och NoSQL. I denna studie gjordes ett experiment för att se hur dessa databassystem hanterar data för att läsa och skriva när det gäller JSON för smarta sensorer. Svarstider registrerades och försökte svara på hypotesen om PostgreSQL kan vara lämplig för att läsa och skriva JSON data som genereras av en smart sensor. Experimentet påvisade att PostgreSQL inte ökar svarstid markant när mängden data ökar för *insert* men för MongoDB gör det. Svaret på hypotesen om PostgreSQL kan vara lämplig för JSON data är att det är det möjligt att den kan vara det men svårt att svara på och ytterligare forskning behövs.

Nyckelord: mongodb, postgresql, json, jsonb, smart sensor, hybrid

Innehållsförteckning

1 Introduktion

Det finns många olika sorts databassystem tillgängliga att använda och de finns både kommersiella samt de av typen ”open source” (öppen källkod) som är gratis. Enligt DB-Engines (2018) så är de populäraste så kallade relationsdatabassystemen (RDBMS). Dessa använder frågespråket SQL som är ett standardiserat sätt att ställa frågor och ändra på data i ett databassystem. I takt med att datamängderna ökar på grund av bland annat fler elektriska enheter som kopplas upp på Internet samt den mängd data sociala media skapar kommer också behovet av att kunna spara stora volymer av ostrukturerade data också att göra det. Vilket till skillnad i SQL databassystem som på ett strukturerat sätt behöver veta i förväg vad det är du ska spara och i vilken ordning enligt ett specifikt schema. Det är möjligt att SQL databassystemen inte kan anpassa sig till det när volymerna utökas som beskrivs av Abramova, V., Bernardino, J (2013). I dessa fall är icke RDBMS av typen NoSQL användbara då de inte följer samma uppbyggnad för att lagra data som det görs i ett databassystem av typen SQL och klarar komplexa data som Chandra (2015) nämner. Strukturerade data är formaterad på ett sätt som gör det läsbart för en människa och maskin. Exempelvis en adresslista eller mötesbokningar som är sparad i en tabellstruktur i en RDBMS. Ostrukturerade data kan vara video, musik, bilder och löpande text i e-post. Det genereras också av företag och elektriska enheter som en exempelvis en mobiltelefons GPS-position och smarta sensorer i hemmamiljö där de kopplas upp via Internet heller i det lokala nätverket.

Denna studie undersökte om det finns ett lämpligare alternativ mellan de två databassystemen MongoDB och PostgreSQL som kan passa lagring av data från smarta sensorer som används i hemmet. PostgreSQL har funnits med sedan omkring 1995 och fram till för några år sedan så hanterar den också formatet JSON precis som MongoDB. men just också för att PostgreSQL verkar vara en sorts hybriddatabas. Det som jämfördes mellan databassystemen var hur de prestandamässig hanterar dataformatet JSON som genererades av en simulerad smart sensor som hanterar elektrisk förbrukning då det enligt Weiss, M., Friedemann, M., Graml, T., Staake, T., Fleisch, E (2009) är allt fler människor som önskar övervaka sina hem med att bland annat kolla av luftfuktigheten, temperatur och ljusstyrka men även för att kunna hushålla med elektriciteten.

Ett experiment utfördes i en lokal miljö där dessa två databassystem installerades på en server. En webbapplikation skapades som simulerade en smart sensor och utförde mätningar. Dessa mätningar användes för statistik som jämfördes med varandra för att se om det var någon skillnad prestandamässigt mellan databassystemen.

2 Bakgrund

I detta kapitel berättas det om vad dataformaten JSON och JSBONB är samt kort om databassystemen. I kapitlet 2.1 Datalagring nämns dataformat som kan vara lämplig för datautbyte i webbapplikationer. BSON nämns också som är en binär form av JSON. Avsnittet om smarta sensorer berättar om vad dessa enheter är och hur de kommunicerar i ett nätverk samt hur det skulle kunna se ut i en hemmamiljö.

Det finns lösningar på marknaden i olika former för detta redan men frågan är att se vilket databassystem av PostgreSQL och MongoDB som kan tänkas prestera bättre när det gäller att hantera dataformatet JSON data som en smart sensor genererar. PostgreSQL är en slags hybrid när det gäller vilken typ den tillhör och lägger sig mellan databasteknikerna NoSQL och SQL. Den kan lagra dataformatet JSON men datatypen för detta måste specificeras först vilket till skillnad mot MongoDB inte behöver göras då den redan lagrar i det dataformatet.

2.1 Dataformat i webbapplikationer

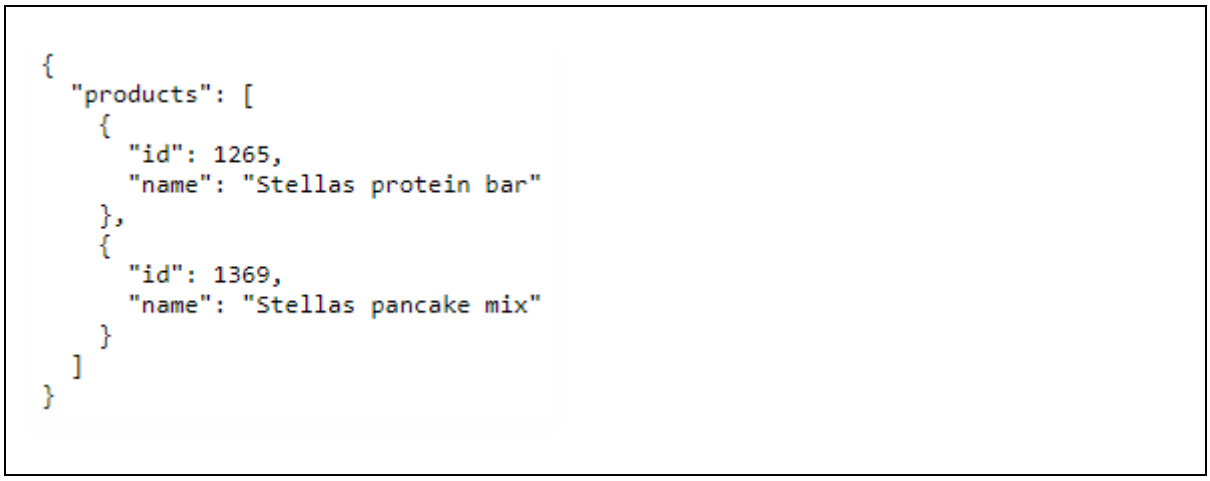
Det finns olika dataformat som används av webbapplikationer för datautbyte men det som denna studie ska kolla mer på är just JSON då det är de som sensorerna som undersökts genererar samt att det är det mest populära formatet för datautbyte enligt Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D (2016). Varför JSON är populärt beror enligt de på att det finns stöd för det i HTTP protokollet. Detta gör att det inte behöver tolkas eller serialiseras av något separat bibliotek som i exempelvis Javascript. Det är också helt textbaserat vilket HTTP protokollet kommunicerar med över Internet. Nedan förklaras det lite kort om två av dessa dataformat som till viss del påminner om varandra. Det visas också lite exempel på hur data för dessa kan se ut.

2.1.1 JSON / JSONB

JSON betyder JavaScript Object Notation och är ett populärt format för datautbyte då det på grund av sin enkelhet och läsbarhet för både människor och datorer används mycket på Internet enligt Bourhis, P., Reutter, J.L., Suárez, F., Vrgoč, D (2107). Många webbtjänster använder sig av JSON för att returnera data mellan server- och klientdator. Wang (2011) menar att data i JSON-format är lätt sätt för datorer att analysera och hantera då formatet är helt textbaserat så det är inte svårt att läsa och förstå vad data består utav. Detta gör att det används flitigt vid så kallade API (Application Programming Interface) som blir som ett gränssnitt för applikationer där de kommunicerar och utbyter data med servern.

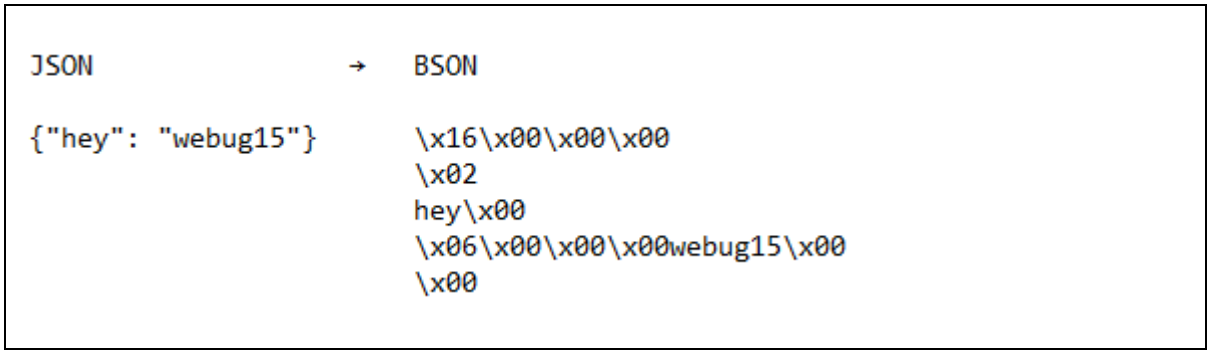
Ett JSON dokument består av ett antal nycklar och dessa nycklar har ett värde så kallade nyckel/värde. Dessa värden kan vara av typen array, nummer, objekt eller sträng (det finns några till). Ett exempel på hur ett JSON dokument ser ur går att se i Figur 1 där nycklar är det som är kommer först med citationstecken och värdet för de kommer efter kolontecken inom citationstecken. Det som hamnar mellan

klammerparentes kallas för objekt och ett sådant kan ha ytterligare underliggande objekt i sig och på så vis byggs dokumentet upp. JSON objekt tolkas som en sträng av data vilket gör att den kan läsas fortare än exempelvis i dataformatet XML som först måste analyseras som ett DOM (Document Object Mode) vilket tar lite längre tid Wang (2011). Detta nämns också av Soliman, M., Abiodun, T., Hamouda, T., Zhou, J., Chung-Horng, L (2013) där JSON har en fördel i hastighet i jämförelse med XML just för att XML måste tolkas först, det vill säga översättas från binär kod till objekt i datorns minne.



Figur 1 Ett JSON dokument med två produkter

Binary JSON heller JSONB (BSON) lagrar data i en binär form av JSON. Detta betyder att strukturen för lagring ser lite annorlunda och tanken är att data ska bli kompaktare och läsas av snabbare i Javascript. Det som sker är bland annat att texten kompakteras och exempelvis mellanslag försvinner. JSON data behåller sin struktur som den är skriven men BSON trycker ihop den. BSON har några mål för detta och det är att den snabbare ska läsa av dokument. Själva manipuleringen av data i dokument som är skapade ska vara enkel och det finns några datatyper till som inte finns med i JSON och det är en datumtyp och en Bindatumtyp (BSON 2018). Ett exempel på hur JSON lagras i binär form går att se i Figur 2.



Figur 2 JSON i binär form BSON

2.2 Databastekniker

2.2.1 NoSQL

Not Only SQL (NoSQL) betyder icke-relationsdatabassystem. Det är en samling av olika databassystem som lagrar sin data på ett annat sätt än i databassystemen med SQL. De har oftast inte några designade scheman att följa för datalagring som det görs i den tabellstruktur som byggs upp. Några av det största fördelarna med NoSQL är enligt Jing, H., Haihong, E., Guan, L., Jian, D (2011) att de är snabba på att läsa, skriva och hantera stora mängder data samt enkla att expandera till en låg kostnad. Det som de däremot inte är så bra på är att hantera SQL som är standardiserat frågespråk men det betyder inte att de inte kan hantera SQL-frågor utan det finns databaser som klarar av grundläggande databaskommandon ändå som exempelvis *select*, *insert*, *update*, och *delete*. Det finns olika sätt att lagra data på och det sker med hjälp av datamodeller som är det sätt som databassystemen arbetar med. Några av de nämns av både Jing m.fl (2011) och Győrödi, C.G., Győrödi, R.G., Pecherle, G., Olah, A (2015). De modeller som de tar upp är av typen ”key-value”, ”document”, ”column” och ”graph-oriented” och nedanför är en kort beskrivning om de.

- *”Key-value”* betyder att data som läggs in i databasen får unika nycklar som består av ett nyckelvärde och ett datavärde. Detta kan jämföras med en primärnyckel som används i SQL. Detta gör att data blir sökbar i databasen. Denna modell använder inget förbestämt schema som ska följas för lagringen.

- *”Document”* påminner mycket om nyckelvärde men här lagras data som dokument. Ett dokument kan lagras i dataformatet JSON och i vissa andra databassystem inom samma modell i dataformatet XML. Denna modell har inget schema att följa men till skillnad mot nyckelvärde lägger den även till ett ytterligare index i den data som finns lagrad. Detta gör att söktiden reduceras istället för att databasen måste söka igenom alla samlingar efter den data som efterfrågats.

- *”Column”* använder tabeller men lagrar data i kolumner istället för i rader som i SQL. Databasen lägger till ID för det som lagras i kolumnen. På detta sätt kan databasen peka ut specifika data.

- *”Graph”* lagrar data med multi attribut som kollar olika relationer som data har mellan varandra och dess entiteter. Ett exempel på detta är sociala nätverk där en person har relationer med andra personer. Exempelvis som på Facebook och Instagram.

2.2.2 SQL

SQL står för ”Structured Query Language” och är ett frågespråk som är standardiserat där det går att hämta och ändra på data i relationsdatabaser. Dessa typer av databaser lagrar data i specificerade tabeller med rader och kolumner. Tabellerna är organiserade efter vad de har för koppling (relation) med varandra. Exempelvis vad är det tabellen ska vara och innehålla för data. Är det i stil med en person kan det vara att lagra personnummer, för- och efternamn. Är det en anställd kanske anställningsnummer, avdelning och lön är viktigt för just den tabellen. Genom att på detta sätt bygga upp databasen med ett förutbestämt schema för tabellerna och dess relationer så går det att ställa avancerade frågor mot databasen för att plocka ut data som efterfrågas.

2.3 Databassystem

2.3.1 MongoDB

MongoDB är ett NoSQL databassystem som är dokument-orienterad. Den lagrar dessa dokument i BSON format. MongoDB är av typen *open source* vilket betyder att den är gratis samt att källkoden är tillgänglig för allmänheten att ladda ner. Genom detta kan programutvecklare som utvecklar egna projekt anpassa den till sina egna behov. Att den är dokument-orienterad betyder att data lagras i fält med rader som tillhör en specifik samling det vill säga ett dokument mycket likt dataformatet JSON. Győrödi m.fl. (2015) beskriver hur detta används genom att ge exempel på ett diskussionsforum med tusentals användare där varje dokument tillhör en användare med eget ID. På detta sätt får varje dokument ett unikt nyckelvärde som också är dess primärnyckel ”_id” se Figur 3. Detta gör att strukturen blir specifik för just den användaren som då kan organisera upp sina forum i egna underkategorier som i sin tur kan ha ytterligare underkategorier kopplade till användarens ID.

```
{
"_id": "d4acee3a76e4675b853aa15fde226752",
"username": "yonki",
"email": "yonki@gmail.com",
}
```

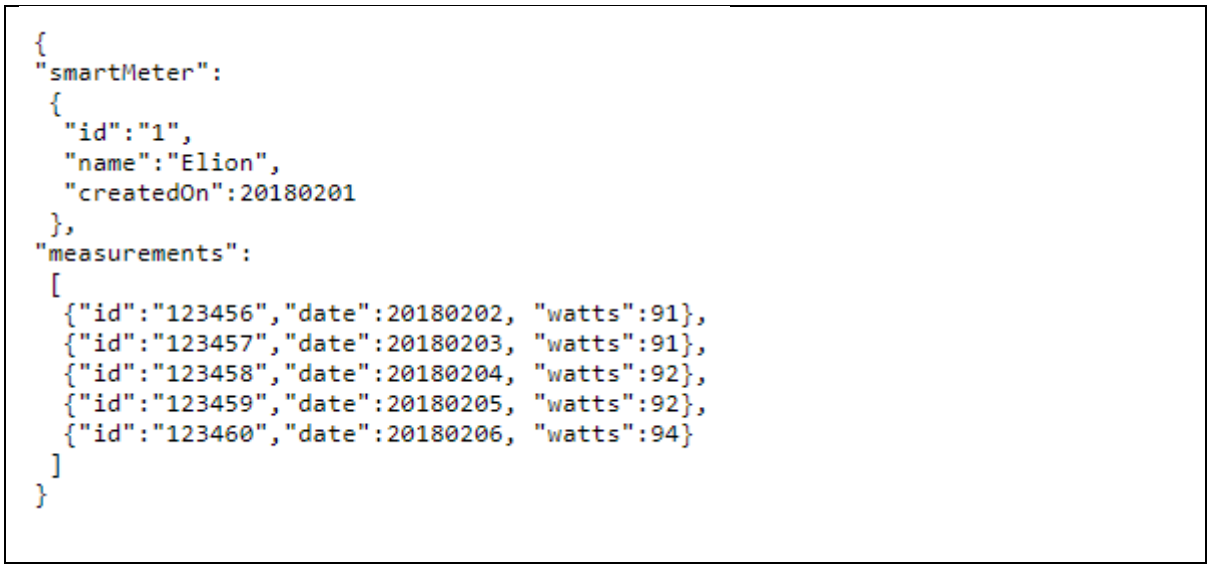
Figur 3 Ett MongoDB dokument

2.3.2 PostgreSQL

PostgreSQL är ett SQL databassystem som är objektorienterad och är också baserad på öppen källkod vilket är gratis att ladda ner och installera. Den fungerar med olika programmeringsspråk och har stöd för SQL frågor som den började hantera kring 1994 (PostgreSQL 2018). Den lagrar data i objekt och använder även klasser och arv precis som exempelvis i Java. I sina senare distributioner har den börjat hantera datatypen BSON precis som MongoDB (PostgreSQL 2018).

2.4 Smarta sensorer

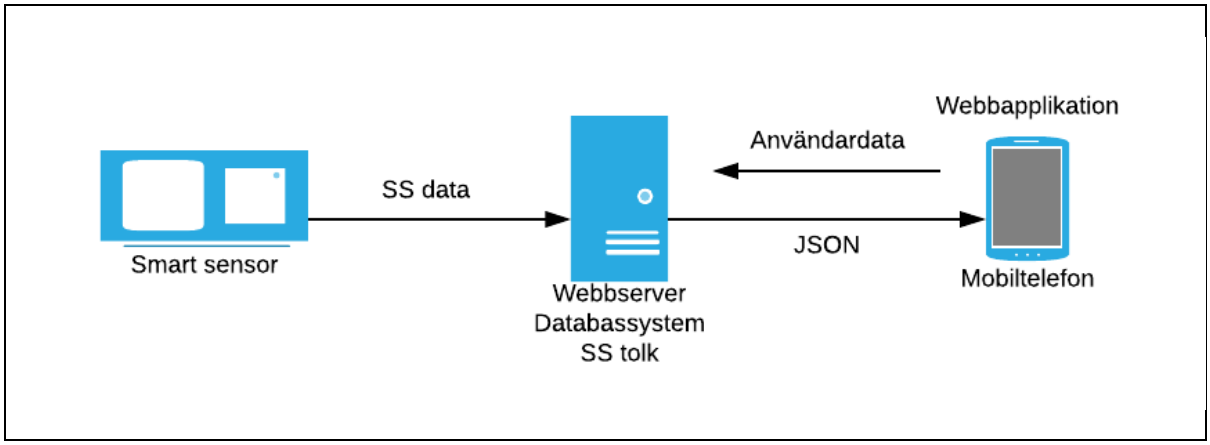
Många elektriska enheter med sensorer som kan kopplas upp på Internet blir allt fler i vad som kallas ”Internet of Things”. Det finns redan en hel del av dessa på marknaden för att bland annat kunna övervaka olika saker i sitt hem. Det som kopplas upp är alltifrån webbkameror till badrumsvågar, brandvarnare och kylskåp. Ett exempel på en sådan enhet är en strömbrytare där det går att koppla på och stänga av elektriska enheter från din mobiltelefon oavsett var du befinner dig bara det finns internetuppkoppling. Dessa brytare kan även ha inbyggda sensorer som läser av elektriciteten som förbrukas precis som nämns i artikeln av Weiss, M. m.fl. (2009). Där visar de hur en sådan sensor skulle kunna se ut specifikt för ett elskåp. Ett exempel på hur JSON data skulle kunna se ut för en smart sensor går att se i Figur 4. Här skickar sensorn förbrukningen den läst av en gång om dagen för exempelvis en lampa som är kopplad till den.



```
{
"smartMeter":
 {
  "id":"1",
  "name":"Elion",
  "createdOn":20180201
 },
"measurements":
 [
  {"id":"123456","date":20180202, "watts":91},
  {"id":"123457","date":20180203, "watts":91},
  {"id":"123458","date":20180204, "watts":92},
  {"id":"123459","date":20180205, "watts":92},
  {"id":"123460","date":20180206, "watts":94}
 ]
}
```

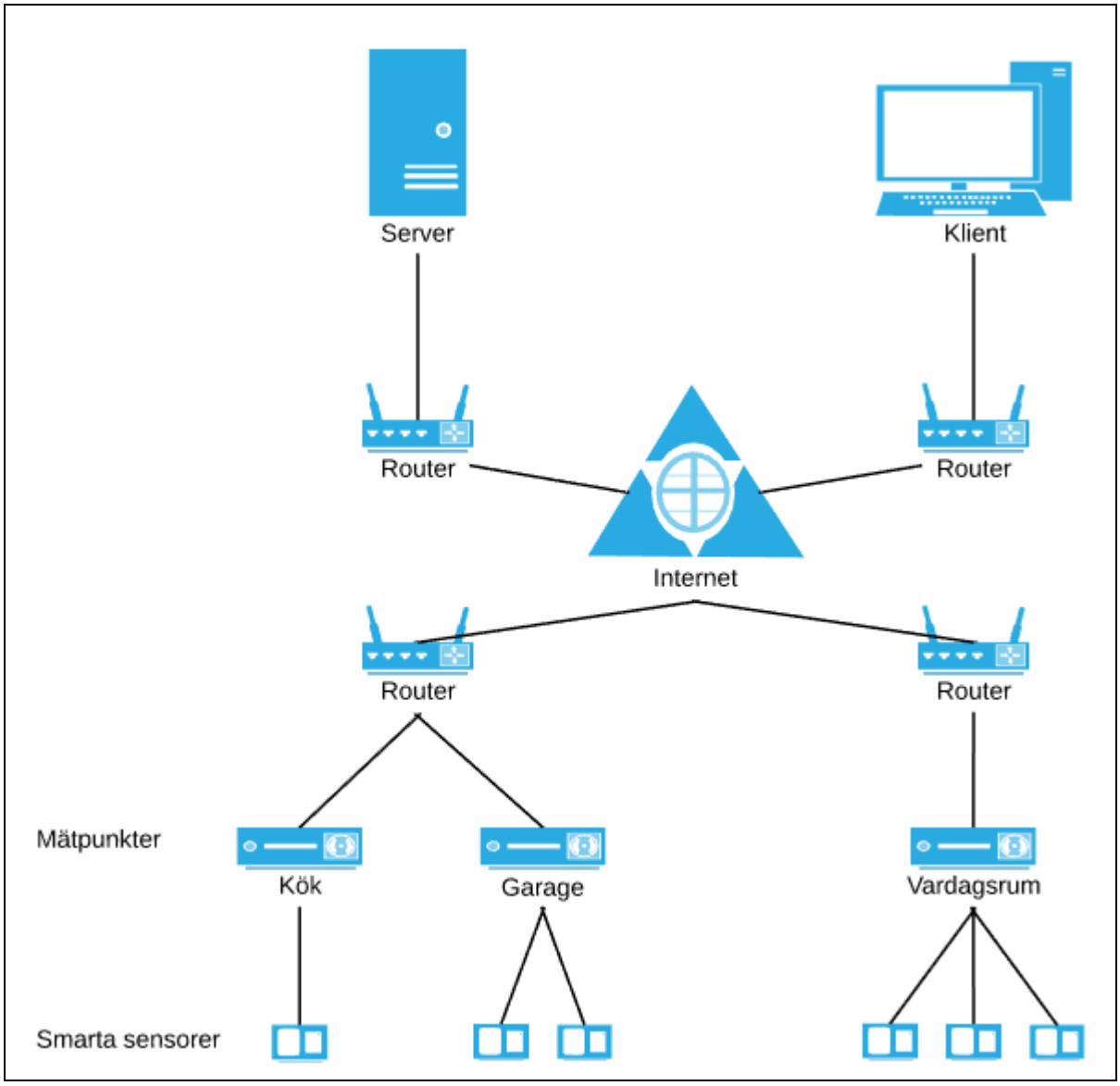
Figur 4 Exempel på hur JSON data kan se ut för en smart sensor

Det är möjligt att ha många olika sensorer i hemmamiljö som finns på fler än ett ställe och det går att se exempel på det i Figur 5. Här skickas SS data (Smart Sensor Data) kontinuerligt till SS tolken som sänder det vidare till databassystemet antingen genom att göra det en gång i månaden heller flera gånger i minuten helt beroende på inställningar för den smarta sensorn. Webbapplikation läser av detta med hjälp av förfrågningar mot databassystemet som skickar tillbaka svar i dataformatet JSON. Denna data visualiseras sedan i form av exempelvis cirkeldiagram och staplar helt utöver hur inställningar är gjorda i webbapplikationen samt hur den är utformad.



Figur 5 Smart sensor kommunicerar med webbapplikationen i mobiltelefon (Lucidcharts (2018)

Hur detta skulle kunna visualiseras i ett större sammanhang beskrivs av Facchinetti, T. m.fl. (2016) där flera sensorer kopplas till speciella punkter. Dessa punkter kan kallas för mätpunkter och är egentligen en del av hushållets olika rum som exempelvis köket. Sensorerna skickar data regelbundet precis som tidigare exempel till en central server antingen en gång i månaden, om dagen eller flera gånger i minuten, allt beroende på inställningar. För att ge ett exempel på hur det skulle se ut lokalt i användarens hemmanätverk med Internetuppkoppling så går det se detta i Figur 6.



Figur 6 Nätverk med sensorer i olika mätpunkter (Lucidcharts (2018)

3 Problemformulering

De så kallade relationsdatabassystemen (RDBMS) har varit väldigt framgångsrika då de använt välbyggda scheman men att de inte alltid det första valet då det gäller data som är flexibel i både form och variationer enligt Liu, Z.H., Hammerschmidt, B., McMahon, D (2017) och speciellt när många av dagens webbsidor och applikationer kräver alltmer snabbhet, prestanda och skalbarhet med databassystemen som Chandra (2015) beskriver. Det är möjligt att RDBMS inte räcker till och det är där som NoSQL databassystemen tillkommit för att kunna hantera den mängd av data som genereras enligt Győrödi m.fl. (2015). När allt fler enheter kopplas upp på Internet genereras stora mängder data som också lagras för att kunna analyseras. Några av dessa enheter som skapar dessa data är exempelvis smarta sensorer kopplade till elektriska enheter i ett hem som nämndes i kapitel 2.3. Dessa sensorer kan registrera alla möjliga data från sin omgivning. Precis som Weiss m.fl. (2009) nämner så vill människor vara mer medvetna om den energi de förbrukar i hemmet för att bland annat kunna påverka hur dessa enheter uppför sig, spara energi och pengar. Problemet är att det inte finns någon bra information tillgänglig på ett smidigt sätt som kan visualisera hur mycket en enhet förbrukar i elektricitet då en månadsfaktura om elförbrukning inte säger så mycket mer än vad som förbrukats totalt på en månad. Det finns en mängd olika databassystem som kan lagra denna data och de som valdes ut för denna studie blev:

- MongoDB
- PostgreSQL

Båda är några av de populäraste databassystemen enligt DB-Engines (2018). MongoDB är ett renodlad NoSQL men PostgreSQL valdes ut för att den är lite av en hybrid då den kan ta emot frågor som hanterar operationer för både SQL och NoSQL. PostgreSQL är egentligen ett databassystem av typen SQL men har i sina senare distributioner börjat hantera dataformatet JSON/JSONB vilket gör att den lägger sig mellan NoSQL och RDBMS och att den kan lagra ostrukturerade data precis som MongoDB gör.

3.1 Frågeställning

Både MongoDB och PostgreSQL hanterar dataformatet JSON och denna studies frågeställningar är att besvara följande frågor:

1. Kan PostgreSQL som hybriddatabasystem vara lämplig för sensordatalagring?
2. Har mängden data betydelse när det gäller svarstid för lagring?

Om PostgreSQL är lämplig menas med att hantera den datamängd i svarstid som genereras för *insert* samt svarstid för *select*.

3.2 Hypotes

Hypotesen är att se om PostgreSQL är lämpligare än MongoDB i att hantera den JSON data som skapas av den simulerade sensorn som webbapplikationen genererar.

3.3 Metodbeskrivning

Metoden var i formen av ett experiment då enligt Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. & Wesslén, A. (2012) är ett bra sätt för att prova en hypotes samt att det sker i en sluten miljö som gör att experimentet sker under kontrollerad form. Detta experiment utfördes genom att databassystemen installerades på en server i ett lokalt nätverk med klientdator. Data som genererades och som fyllde databassystemen utfördes av webbapplikationen genom att använda ett skript som kördes i ett antal iterationer.

När den första databasen var installerad fylldes den med sensordata från webbapplikationen som då också utförde mätning på det. När mätningen var slutförd installerades den andra databasen för att också fyllas med sensordata och mätningar gjordes även på den. Ytterligare en mätning gjordes med förfrågningar *select* mot databassystemen. Mätningarna jämfördes sedan med varandra för att få ut en skillnad i de svarstider som noterades i aktiviteterna som utfördes. Dessa skillnader användes till statistik som presenterades i form av diagram. Webbapplikation använde ett antal variabler för att kunna förändra den data som lagrades och hur många gånger det skulle göras.

MongoDB lagrade sin data direkt i det binära dataformatet JSONB när data kom in i dataformatet JSON. För PostgreSQL del så ställdes det in först innan det gick att lagra något och detta gjordes genom ett schema som tillfördes. Detta experiment utfördes genom att simulera en typ av sensor via en webbapplikation som genererade JSON data till databassystemen. För en framtida forskning kan ytterligare sensorer med olika utseende på sitt data set läggas till för att få en mer realistisk data från olika kunder och sensorers mätpunkter samt att utöka mängden data i databassystemen. Det är inget som säger att resultaten som blev var rätt och det finns många saker i ett experiment som kan bete sig felaktigt och att resultaten såg fel ut. Hypotesen kan därför inte sägas vara bevisad utan istället sker argumentation om att resultaten pekar på att hypotesen är rätt.

Det här experimentet grundade sig på tidigare tester i svarstid mot databassystem gjorda av Jung, M.G., Youn, S.A., Bae, J., Choi, Y.L (2015) där de ställer frågor mot databassystemen genom att använda förfrågningarna *insert*, *select*, *update*, och *delete*. Testerna som de gjorde var fördelat på fem mätningar med 30 000, 90 000, 150 000, 210 000 och 300 000 förfrågningar mot databasen. Detta experiment hade en liknande uppbyggnad och kördes på servern enbart, det vill säga att servern hanterade Hypertext Preprocessor (PHP) filerna och inga processer skedde på klientdatorn. I detta experiment var det endast test med förfrågan *insert* och *select* som användes och det är för att data normalt inte uppdateras heller tas bort när det

lagras på detta sätt för en sensor utan det ska lagras för att kunna analyseras på vad som registrerats och inte göra ändringar på det i efterhand.

3.3.1 Alternativa metoder

Eftersom detta experiment utfördes i sluten miljö och bara simulerade en sensor och inte någon i fysisk form så kan en alternativ metod med detta experiment vara en fallstudie. Där data samlas in från en mängd olika sensorer installerade i ett riktigt hus. Detta för att som Wohlin m.fl. (2012) nämner är att fallstudier som sker i en miljö som är riktig blir också mer lik verkligheten. Genom att utföra detta så skulle en större bredd och mängd av data kunna samlas in för analys till databasen. Dessa sensorer kan mäta en hel del av olika saker i hemmet bland annat temperaturer och luftfuktighet. Det som kan vara ett problem är att det kan vara svårt att kontrollera hur sensorernas system fungerar när de är igång om det inte finns tillgång till huset. Möjligen skulle huset kunna övervakas med fjärranslutning men då kan det bli problem med de etiska delarna då människor blir inblandade i det. Alternativt skulle huset kunna vara en laborationsmiljö utan människor heller kanske en tom lägenhet som då skulle kunna fungera bättre för experimentet som utförs.

3.3.2 Etik

Gällande de etiska aspekterna så utfördes experimentet lokalt i en datamiljö bestående av en server och en klient som ingick i ett mindre nätverk. Detta var ett teknikorienterat experiment som enligt Wohlin m.fl. (2012) är en av två metoder för experiment. Den andra är människoorienterad och användes inte. Genom detta fanns inte några människor med i experimentet annat än den som utförde testet vilket gjorde att den mänskliga faktorn föll bort som kan påverka resultatet.

Den data som lagrades i databassystemen genererades av webbapplikationen för att simulera en smart sensor. Data som genererades var uppbyggd av *loop* och *array* i PHP filer men den hade en struktur för att efterlikna det som kom från sensorns data set 2.4 Smarta sensorer. Innehåll som kan uppfattas som känslig och peka ut någon människa personligen som exempelvis personnummer genererades inte av webbapplikationen.

All programkod som skapades finns tillgänglig på webbsidan Github samt i Appendix. Genom att göra detta är det möjligt att återupprepa experimentet för framtida forskning. Det finns också information om hårdvara samt versioner på mjukvara som användes. Den data som genererades och lagrades i databassystemen lades inte till i studien utan bara information om vad det var för något som användes samt hur den genererades av webbapplikationen.

Andra saker som påverkade resultatet var valet av hårdvara och den nätverksmiljön som används. Även program som var igång på servern. Detta gjorde att denna studie inte är absolut och påvisade inte att den ena databassystemet skulle vara bättre än det andra utan detta är endast ett experiment. Den hård- och mjukvara som användes i denna studie går att se i 5.1 Förstudie.

4 Relaterad forskning

I artikeln av Facchinetti, T. m.fl. (2016) nämner de hur ett system för sensorer skulle kunna se ut och de visar också exempel på den data som sensorerna genererar. De visar ett diagram över hur sensorer är anslutna till olika mätpunkter och en av dessa skulle exempelvis kunna vara köket där det finns ett antal sensorer anslutna. För att relatera till experimentet skulle sensorn exempelvis kunna vara kopplad till ett eluttag med en elektrisk enhet som exempelvis ett kylskåp. Denna artikel samt den skriven av Weiss, M. m.fl. (2009) och Jung, M.G. m.fl. (2015) användes som grund för experimentet som beskrivs under metodbeskrivningen.

En annan artikel som också relaterar till detta arbete är skriven av Van der Veen, J.S., Van der Waaij, B (2098). Där de jämför tre olika databaser i förhållandet att hantera data för sensorer. Det de tar upp är bland annat att SQL databassystemen borde bli en aning långsammare när det kommer till att hantera stora volymer av data samt att NoSQL möjligen kan lösa detta då de försöker göra det genom att vara mindre konsistenta. När det gäller databassystemens mätningar i prestanda så gjordes liknande mätningar som beskrivs av Jung, M.G. m.fl. (2015).

Győrödi m.fl. (2015) beskriver också en jämförelse i prestanda mellan databassystem och tar även upp några exempel på hur kod för detta skulle kunna se ut. De mätningarna som de utför i denna artikel är på funktionerna *insert*, *select*, *update* och *delete*.

5 Genomförande

I detta kapitel tas delar upp som behövdes för att genomföra experimentet och kunna utföra mätningar och svara på hypotesen som ställts i studien. Beskrivning av den hård- och mjukvara som användes samt några testfall. Resultatet och analysen tas upp under 6 Utvärdering. För att experimentet ska kunna utföras är ett fungerande system nödvändigt med en webbserver som har PHP stöd så det går att koppla ihop databassystemen och köra kommandon mot de. Det skapades en webbapplikation i PHP som genererade data som lagrades i databassystemen. Hur detta genomfördes beskrivs i 5.2 Progressionen. På webbplatsen Github där all programkod sparades sker länkning till kod i form av ett kondensat #cb5a2bf så kallad *hash* genom detta går det att få fram en historik på hur programkoden förändrade sig under progressionen.

5.1 Förstudie

För att kunna utveckla webbapplikationen som simulerade den smarta sensorn så är kunskaper i PHP nödvändigt då det är ett programmeringsskript som körs på servern. PHP används ofta på internetsajter för att skapa dynamiska hemsidor som förändrar sig beroende på vad besökaren vill se för information samt att det går att koppla ihop en hemsida med ett databassystem och utbyta data. PHP är också plattformsoberoende och har stöd för de flesta webbservrar som finns på marknaden. Det har också många sorters så kallade moduler för att koppla ihop olika typer av databassystem med webbservern. Just gällande PHP kommer mycket inspiration från hemsidan stackoverflow.com (2018) där det finns mycket kodexempel och förslag på lösningar som många utvecklare för hemsidor och applikationer kan tänkas råka på under sin utvecklingsprocess för ett projekt. Ett bra sätt för att lära sig PHP är att börja med ett besök på exempelvis hemsidan W3schools.com där det finns guider för grundliga kunskaper inom detta och många andra områden. En bok som kan vara bra för att komma igång är Webbutveckling med PHP och MySQL skriven av Montathar, F (2016) som även tar upp delar för att koppla ihop en MySQL databas. Behövs det djupare kunskaper på en mer detaljerad nivå kan ett besök hos php.net (2018) göras då det där finns dokumentation på den syntax och kommandon som finns att använda.

Inspiration för webbapplikationen kom bland annat ifrån ett gränssnitt som beskrivs i artikeln av Soliman, M. (2013). Där är den uppbyggd i tabellform för att visa innehåll i databassystemen efter att en *select* utförts. Webbapplikationen får genom detta ett visuellt bättre utseende med den data som efterfrågades. Ett exempel för hur detta gränssnitt kan se ut för *select* går att se i Figur 7. I artikeln skriven av Weiss, M. m.fl. (2009) finns fler exempel på gränssnitt.

.	ID	date	destination	sensor	token	value
☐	15018	2018-03-03	5	11	642679871	L
☐	15019	2018-03-04	5	9	642679871	L
☐	15020	2018-03-05	5	11	642679871	L
☐	15022	2018-03-06	5	11	642679871	L
☐	15023	2018-03-07	5	9	642679871	H
☐	15024	2018-03-08	5	9	642679871	H
☐	15025	2018-03-09	5	11	642679871	L
☐	15018	2018-03-10	5	11	642679871	L
☐	15018	2018-03-11	5	9	642679871	H

Figur 7 Sensordata i tabellform i webbapplikationen

Inspiration för visualisering av data finns i artikeln skriven av Facchinetti, T. m.fl. (2016) där ett exempel på diagram som skapas med hjälp av Javascript och ramverket Data Driven Documents (D3.js). Detta ramverk kan skapa avancerade 2D- och 3D diagram baserad på data som den tar emot. Detta gör att det kan effektivisera analys av data direkt på skärmen.

Gällande kunskaper om databassystemen kommer de från deras respektive hemsidor postgresql.org (2018) och mongodb.com (2018) men också hur de kopplas ihop med PHP. På webbplatsen w3schools.com finns grundliga kodexempel som kan modifieras och användas och det finns guider för exempelvis att koppla en MySQL databas. Vid mer avancerade saker användes forumet stackoverflow.com (2018) nackdelen på den hemsidan är att det visserligen finns mycket kodexempel men för att få saker och ting att fungera för just detta experiment så modifierades koder som kom från kodexempel. För att lära sig grundliga kunskaper om databassystem och hur dessa fungerar är boken Databasteknik skriven av McCarthy-Padron, T & Risch, T. (2005) en bra start.

Den data som den simulerade sensorn genererade var baserad på datasetet för JSON data i Figur 5 under 2.4 Smarta sensorer. Detta exempel kommer från artikeln skriven av Weiss, M. m.fl. (2009) och detta experiment efterliknade detta data set. Det finns fler data set att välja på och de kan se ut olika för de sensorer som finns. I Figur 8 visas den hård- och mjukvara som användes samt vilka versioner av distributionen som de hade. Klientdatorn specificeras inte eftersom mätningarna sker på servern där PHP-filerna körs.

Hårdvara	Specifikation
RAM	4x2 DDR2 (8 GB)
CPU	AMD Phenom 9650 Quad-Core Processor
Moderkort	Acer RS780HVF
Hårddisk	Western Digital 1000MB SATA
Nätverkskort	Level ONE WNC-0305USB
Router	Technicolor MediaAccess TG389ac
Server	Acer Aspire M5201

Mjukvara	Specifikation
Operativsystem	Debian v9.4
Webbserver	Apache v2.4.25
Databas 1	PostgreSQL v9.6.7
Databas 2	MongoDB v3.2.11
Skriptspråk	PHP v5.6

Figur 8 Förteckning över den hård- och mjukvara som användes

5.2 Progression

I detta kapitel finns hänvisning till versionshanteringswebbplatsen Github i form av *hash* som pekar mot den aktuella kodens version. Det fanns en server på högskolan som var tillgänglig för framställandet av detta experiment men det togs ett beslut att undvika den på grund av att de fanns risk att andra använder den samtidigt vilket kan påverka resultatet av de mätningar som gjordes. Istället användes en egen dator med fria programvaror som experimentet utfördes på.

För att komma igång så installerades servern med passande operativsystem, webbserver och databassystemen. Det skapades en databas i PostgreSQL och vid installationen fanns redan en databas att använda med namnet ”test1” och en fil skapades för att lagra inloggningsuppgifterna till den se Appendix D. Databassystemen är inte förändrade i sin funktion på något sätt utan installerades med sina standardinställningar. PHP har en möjlighet att köra en cachefunktion som fungerar som en proxyserver där den sparar PHP-sidor för att snabba på laddningen av dessa när de filerna körs igen. Denna cachefunktion kommer inte användas i denna studie eftersom alla programvaror är installerade med sina standardinställningar. För att börja lagra data så skapades ett schema som talar om hur tabellerna ska se ut och vad de innehåller för data typ. I detta fall skapades kolumnerna ID och data se. Kolumnen data fick datatypen jsonb som talar om att den ska lagra data som kommer in till databasen i det formatet. En guide för att utföra detta följdes på adressen (http://www.postgresqltutorial.com/postgresql-json/). I Appendix F går det se hur schemat för detta såg ut och som sparades i en separat SQL-fil. Denna fil kan sedan användas i PHP för att nollställa och återskapa databasen smidigt efter varje mätning

se Appendix E. I början av SQL-filen användes funktionen *drop table if exists* som tar bort tabellen *json_table* om den redan finns och sedan skapar den igen med funktionen *Create Table.*

För att börja sätta in data skapades en PHP fil som genererar en del av den JSON data som baseras på Figur 5 under 2.4 Smarta sensorer. Denna fil skapar en PHP *array* med objekt som sedan kodas om till en sträng med text innan den skickas in i databasen. Hur denna *array* till en början såg ut och byggdes upp går att se i Figur 9 och på Github #5d9f35. Den data som denna *array* hade var påhittad och bestod av fast data med information om sensorn och i detta fall en sensor med ett ID-nummer, namnet Eliond, typen elektrisk och datum när den aktiverades.

```
$jsonArray = array(

'smartMeter' => array(

      'id' => '1',
      'device' => 'Eliond',
      'sensorType' => 'Electric',
      'createdOn' => '20180205',
        ),

'measurements' => array(),
);
```

Figur 9 PHP array

Nu är det endast en *insert* som gjordes men för att börja mäta gjordes denna *insert* flera gånger genom att upprepa insättningen för att få ut ett bättre medelvärde. Funktionen *FOR* användes för att kunna göra flera iterationer med samma uppgift så i Figur 10 är det variabeln *measures* som talar om hur många gånger detta ska göras. Variabelns värde ändras i början av PHP filen. För att kunna börja göra insättningar av data i databasen så byggdes arrayen upp med kodexempel från webbplatsen StackOverflow. Här visar de hur det går att köra en *loop* med en *array* med flera nivåer med *FOR*. I detta fall bestående av en *array* med underliggande nivåerna smartMeter och measurements.

```
for ($i=0; $i <$measures ; $i++)
```

Figur 10 FOR loop som körs 10 000 gånger satt genom variabeln measuers

För att mäta svarstiderna så användes en typ av klocka som startar i början av den kod som ska mätas och en som stannar den i slutet. En differens räknades ut mellan dessa

variabler och lagrades tillfälligt i en *array* timeArray. Denna *array* användes sedan för att spara svarstiderna i en textfil på servern. Denna textfil analyserades och användas för analys. Funktionen som användes för mätning heter *Microtime()* och ställdes in med hjälp av guider på PHP.net (2018) och Stack Overflow (2018). Tiden som räknas ut är i formatet *Unix timestamp* och är i mikrosekunder. För att få några mindre decimaler i differensen så användes funktionen *number format* som ger 4 decimaler i svaret se Figur 11.

```
$timeStart = microtime(true);
 // code to run
$timeEnd = microtime(true);

//Measure response time and push to array
$timeDiff = $timeEnd - $timeStart;
$timeDiff = number_format(($timeDiff), 4);
array_push($timeArray, $timeDiff);
```

Figur 11 Räknar ut differensen mellan starttid och sluttid för körning av *insert*

Figur 12 visar de variabler som användes i arrayen. *Iterations* talar om hur många gånger loopen ska utföras. Att den görs 10 gånger var för att få bra resultat på mätningen och det är bättre att göra flera stycken för att få ut ett bättre medelvärde. *Insert*s betyder att det är en kund med en sensor i hemmet. *Measures* betyder att sensorn skickar den avläsning den gjort varje minut under ett dygn (60*24=1440). *Idm* är ett slumpad *id-nummer* för avläsningen. *Devicewatt* är vad den elektriska enheten förbrukar och i detta fall 60 watt. *Startwatt* är startvärde för avläsningen för att simulera att sensorn har använts ett tag och inte är ny. *Time* är för att få med en tidsstämpling när avläsningen gjordes och den ökar med 1 minut varje gång.

```
// variables
$iterations = 10;               // how many runs
$inserts = 10000;               // inserts to do (customers)
$measures = 1440;               // measures per inserts
$fileNr = 0;                    // save file counter
$idM = 123456;                  // random measurements ID
$deviceWatt = 60;               // the device watts
$startWatt = 25;                // starting watts value for device
$time = date('Y-m-d H:i:s');    // get time
$timeArray= [];                 // array to push response time
```

Figur 12 Variablerna för arrayen

Ett exempel på hur arrayen ser ut går att se i Figur 15. Här är första delen av arrayen *jsonArray* uppbyggd av två stycken *While* loopar. Denna array innehåller *smartMeter* med statiska data för *id*, *device*, *sensorType* och *createdOn*. *Measurments* är en till array på samma nivå där variablerna i Figur 13 används för att den ska kunna simulera avläsningarna från sensorn.

```
// start iteration
$index = 1;
while($index <= $iterations){
        $index++;
        $fileNr++;

        // generate json data
        $index2 = 1;
        while($index2 <= $inserts){
                $timeStart = microtime(true);
                $index2++;
                $jsonArray = array(

                'smartMeter' => array(

                      'id' => '1',
                      'device' => 'Eliond',
                      'sensorType' => 'Electric',
                      'createdOn' => '20180205',
                        ),

                'measurements' => array(),
                );
```

Figur 13 Övre delen av arrayen

Den nedre delen av arrayen visas här i Figur 14 där loopen for genererar data och trycker in det i *JsonArray* under arrayen measurements med funktionen *array_push*. Detta för att arrayen measurements i Figur 13 ska få in all mätdata som simulerats av webbapplikationen. Värdet för kWh skapas genom att räkna ut hur mycket den elektriska enheten förbrukar per timme. Detta görs genom att först slumpa fram ett nummer hur länge enheten är påslagen i variabeln *wattsRand* som betyder att enheten är på mellan 0 och 8 timmar. Detta multipliceras sedan med *deviceWatt* som i detta fall är 60 watt och delas med 1000. Värdet lagras genom att addera detta till *startWatt* som sedan används i arrayen Data. *Time* är en tidsstämpel och ökar med 1 minut för varje iteration.

```
for ($i=0; $i <$measures ; $i++) {
    $idM++;

    // to simulate the consumption in watt for the device
    $wattsRand = 0;
    $wattsRand = $wattsRand + rand(0,8); // device is on between 0-8 hours/day
    $startWatt = $startWatt + ($deviceWatt * $wattsRand)/1000;
    $startWatt = number_format(($startWatt), 2);
    $time = date('Y-m-d H:i:s', strtotime($time.'+1 minute'));

        $Data = array(
                'id' => $idM,
                'date' => $time,
                'kWh' => $startWatt,
        );
                array_push($jsonArray['measurements'], $Data);
        }
```

Figur 14 Nedre delen av arrayen

Själva uppbyggnaden av arrayen är av en typ hur den skulle kunna se ut men det finns många fler utseenden på detta och det är beroende på hur sensorerna generar sin data. Några fler exempel på detta går att se i artikeln skriven av Facchinetti, T. m.fl. (2016).

När arrayen fungerade och data som sparades såg okej ut så kunde mätningar påbörjas. För att kunna mäta av tider för körning av PHP skript så användes återigen funktionen ”*microtime*(*true*)” och sedan räknades mellanskillnaden ut. För att spara mätningarna så lades funktion till #b8b511 för att lagra detta i textfiler som kan importeras till kalkylprogrammet när diagram gjordes.

När arrayen *jsonArray* genererats så kan insättningen ske i PostgreSQL detta görs genom nedanstående rader kod i Figur 15.

```
// encode php array to string
$jsonArrayEncoded = json_encode($jsonArray);

$sqlQuery = "INSERT INTO json_table (data) VALUES ('$jsonArrayEncoded')";
$runQuery = pg_query($dbconn, $sqlQuery);
```

Figur 15 Kod för *insert* i PostgreSQL

Fram hit i progressionen är filerna klar för PostgreSQL och motsvarande kod för insättningen i MongoDB ser ut som i Figur 16 och är sparad i en egen PHP fil för MongoDB.

```
$client = new MongoDB\Client("mongodb://localhost:27017");
$collection = $client->exjobb->json_data;

// insert array to database
$result = $collection->insertOne( $jsonArray );
```

Figur 16 Kod för *insert* i MongoDB

För att kunna spara de tider det tar för skriptet att köra så gjordes det på två sätt. Det första är att spara tider för varje insättning i databasen och den andra är hur lång tid det tog totalt. Sparandet av varje iteration sker genom att lagra det i en array som heter timeArray. Den används sedan för att skapa de filerna med mätdata enligt Figur 17 och varje fil får ett nummer som representerar iterationen från variabeln fileNr.

```
// write values from timeArray to file
$file = 'measurements_plot_'.$fileNr.'.txt';
foreach ($timeArray as $key=>$value) {
        file_put_contents($file, $value.PHP_EOL, FILE_APPEND | LOCK_EX);
}
```

Figur 17 Spara fil med iterationsnummer och mätdata

PHP filerna för *insert* sparades som en kopia att användas till MongoDB där ändringar gjordes för att passa *select* frågorna se Figur 18. PostgreSQL söker efter ett slumpat nummer för *ID* genom att använda funktionen *rand()* och *maxRows* talar om hur många *insert*s det är i databasen.

```
// choose random ID
$randomNmb = rand(1,$maxRows);

// query the database
$result = pg_query($dbconn,"SELECT ID FROM json_table WHERE ID=$randomNmb");
```

Figur 18 Kod för *select* i PostgreSQL

För MongoDB letar den efter ett *ID* under *measurements* se Figur 19. För att ställa in detta följdes en guide hos MongoDB (2018).

```
// query the database
$q = array('measurements.id'=> 123457);
$cursor = $collection->find($q);
```

Figur 19 Kod för *select* i MongoDB

5.3 Pilotstudie

För att testa denna array och mätningar på databassystemen så gjordes ett mindre pilottest på PostgreSQL för att prova att verktygen som skapas fungerar och kan användas för de större mätningarna som skedde senare i studien. Detta verktyg som också användes till MongoDB var en aning annorlunda i uppbyggnad det beror på att kommandon som skickas till databassystemen skiljer sig åt lite grand i PHP genom modulerna som kommunicerar med databassystemen. Ett försök gjordes ändå att få PHP filerna att likna varandra så gott det går. När filerna var klara så utfördes tester med olika testfall och första testet gjordes med *insert* och det andra en *select*. Svarstiden sparades och användes för att göra diagram till analysen.

De tester som gjordes var av typen *insert* och *select* med en iteration på 10 gånger, antalet *insert* på 10 000 och med en mängd på 24. Detta betyder att det är 10 000 kunder som har en sensor var som avlästes varje timme under ett dygn. I Tabell 1 nedanför visas testfallen som utfördes. Efter detta så kördes en *select* med 10 000 rader och mätning även där. Frågorna att hämta ett ID gjordes i detta fall 10 000 gånger. Ett medelvärde räknades sedan ut som användas till de diagram som ska skapades.

Data som lades in gjordes genom använda funktionen ”WHILE” kombinerat med ”FOR”. I testet simulerades en sensor vars data läses av varje timme under ett dygn. Detta ger 24 rader i arrayen ”measurements” som ligger i sin del av datasetet och i Tabell 1 är det under kolumnen ”Mängd”.

Tabell 1 Testfall för pilotstudien

	Databassystem	**Typ**	**Antal**	**Mängd**	**Iterationer**
TF1	PostgreSQL	*Insert*	10 000	24	10
TF2	MongoDB	*Insert*	10 000	24	10
TF3	PostgreSQL	*Select*	10 000	24	10
TF4	MongoDB	*Select*	10 000	24	10

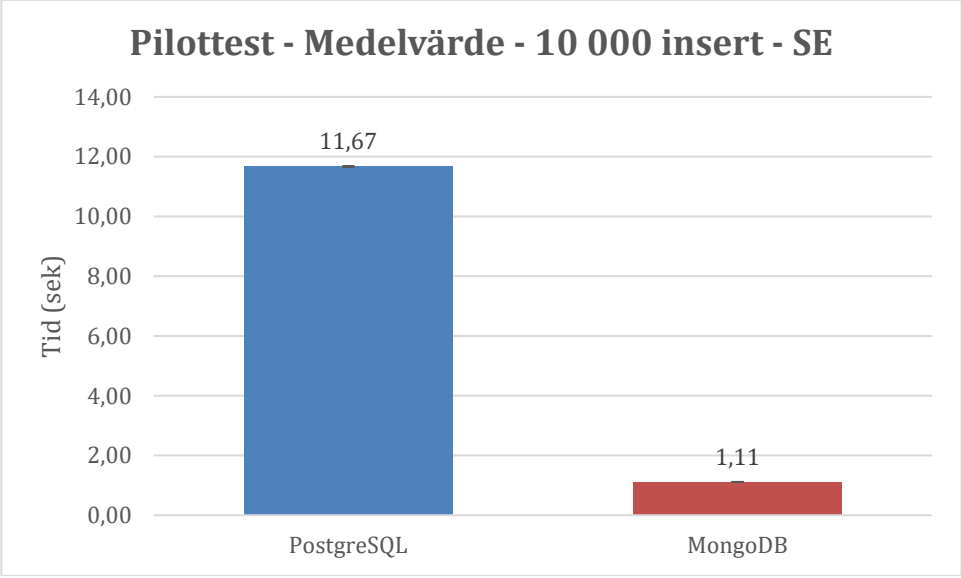
Arrayen som skapades i PHP baseras också på Figur 5. Denna användes för att simulera den data som lagrades i databassystemen. Den data som arrayen genererar är i betydelsen av en kund som har en sensor vilken avläses varje timme under ett dygn. För att få någorlunda realistiska genererade data så lagrades den data som från sensorn med tanken att den var kopplad till en elektrisk enhet på 60 watt samt att denna är påslagen mellan 0 och 8 timmar per dag. Ett antal testfall utfördes i form av ”*insert*” och ”*select*” med olika parametrar för att få olika resultat. Detta presenteras sedan under 5.3.1 Resultat.

Det visade sig att arrayen fungerade bra och hade en enkel uppbyggnad #4e62e4 men en del ändringar måste göras för att få in mer data i databassystemen. Efter pilottestet var gjort så behöll arrayen sin struktur men det lades till så att den data som byggdes upp blev mer trovärdig för den simulerade sensorn. Eftersom det i pilottestet bara var statiskt data som sattes in så lades det in variabler som förändras över tiden exempelvis förbrukning av watt och tidstämplar se Figur 13. Arrayen fick denna förändrade data från variabler enligt #ac55cd. Genom att tillföra dessa olika variabler så gick det att få ut mer data som fyllde arrayen enligt #bfc96e.

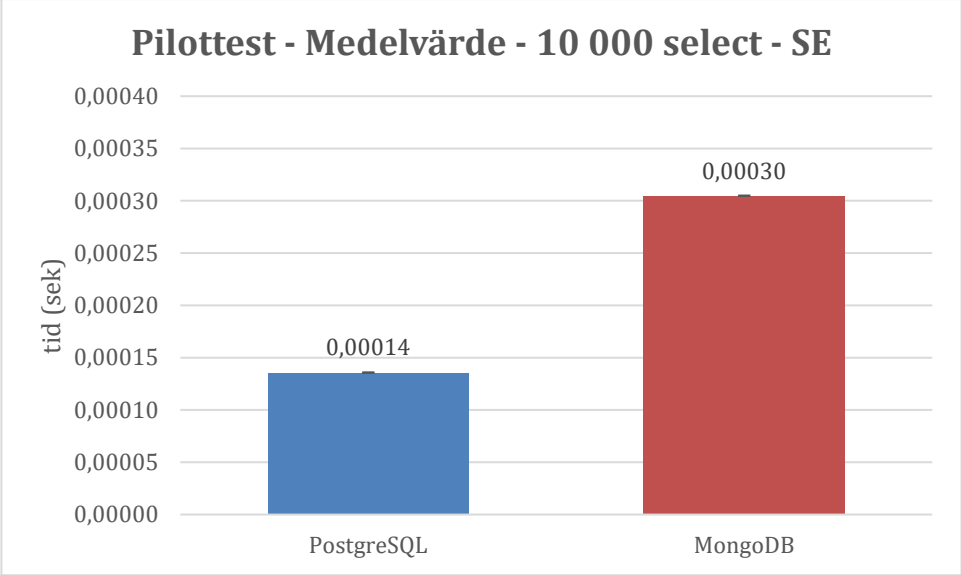
5.3.1 Resultat

Resultatet av TF1 och TF2 visas här i Figur 20. Testfallen som gjordes med 10 000 *insert* och mängden 24. Det som kan ses i diagrammet är tiden i sekunder det tog att slutföra varje iteration. PostgreSQL ligger på 11,67 sekunder och MongoDB kring 1,11 sekunder.

Resultatet av TF3 och TF4 som gjordes med 10 000 *select* visas här i Figur 21. Iterationen är även här på 10 gånger. PostgreSQL ligger kring 0,00014 sekunder och MongoDB ungefär kring 0,00030 sekunder.



Figur 20 Medelvärde för *insert* mellan databassystemen



Figur 21 Medelvärde för *select* mellan databassystemen

5.3.2 Analys

Mätningarna gjordes med 10 iterationer och den datamängd som valdes för detta var den på 24 som är den mellanliggande för testfallen som gjordes i de kommande mätningarna när progressionen var klar. Pilotstudien gjordes för att kunna prova så att arrayen fungerar samt att mätningar går att göra. Det som går att se på diagrammen för totala tiden på *insert* är att PostgreSQL ligger på kring 11,67 ms mot MongoDB på kring 1,11 ms vilket är ca 1/10 del. För medelvärde med *select* ligger PostgreSQL kring 0,00014 sekunder och MongoDB kring 0,00030 sekunder. En slutsats enligt pilotstudien uppfattas vara att MongoDB är snabbare än PostgreSQL när det gäller *insert* och när det kommer till *select* är MongoDB lite långsammare. En teori varför PostgreSQL tar längre tid på sig när det gäller *insert* är att den måste få PHP objekten som skapas i jsonArray omvandlad till en textsträng först genom att använda funktionen json_encode innan den kan överföras till databasen. MongoDB verkar kunna överföra jsonArray direkt se Appendix B. Gällande *select* så är värdena väldigt låg och knappt mätbara.

6 Utvärdering

I detta kapitel är utvärderingen av experimentet samt resultaten som blev. Detta experiment bestod av att mäta svarstider mellan databassystemen PostgreSQL och MongoDB vid läsa och skriva data i JSON format. PHP filerna som gjordes skapades i två stycken uppsättningar för att passa databaserna PostgreSQL och MongoDB eftersom kommandon mot de skiljer sig lite grand men i övrigt så är PHP filerna för webbapplikationen liknande för båda databassystemen.

Mätningarna gjordes genom att fylla databassystemen med data enligt testfallen och sedan göra *select* mot de strax efter för att sedan gå vidare till nästa testfall. Varje mätning för *insert* och *select* gjordes om 5 gånger för att samla in mer data och få tydligare resultat med medelvärde inför analysen.

6.1 Resultat

De testfall som gjordes för PostgreSQL och MongoDB i experimentet visas här i Tabell 2. Dessa består av 3 testfall fördelade på hur många gånger sensorn avlästes per dag samt hur många antal *insert* som gjordes. Detta kördes i iterationer av 5 gånger var för att få ut tydligare medelvärde.

Tabell 2 Testfall med *insert* för PostgreSQL och MongoDB

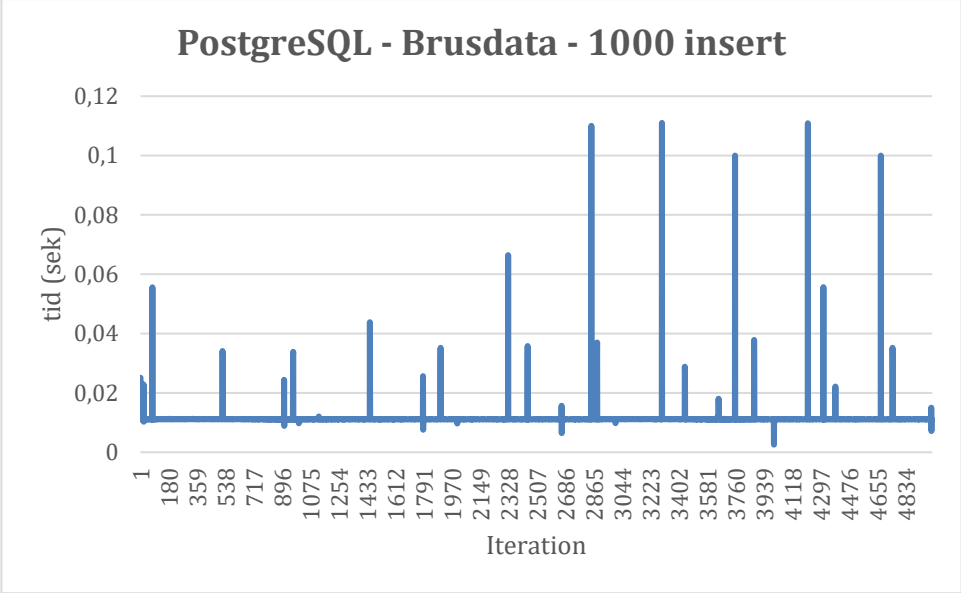
	Typ	**Antal**	**Mängd**	**Iterationer**
TF1 **1 gång / dag**	*Insert*	1 000	1	5
		2 000	1	5
		5 000	1	5
		10 000	1	5
TF2 **Varje timme**	*Insert*	1 000	24	5
		2 000	24	5
		5 000	24	5
		10 000	24	5
TF3 **Varje minut**	*Insert*	1 000	1 440	5
		2 000	1 440	5
		5 000	1 440	5
		10 000	1 440	5

Här i Tabell 3 är det 3 stycken testfall som är fördelade på *select* frågor mot databassystemen. Tabellen är uppbyggd på samma sätt som den för *insert* så när insättning var klar så kördes *select* mot databassystemet direkt efter.

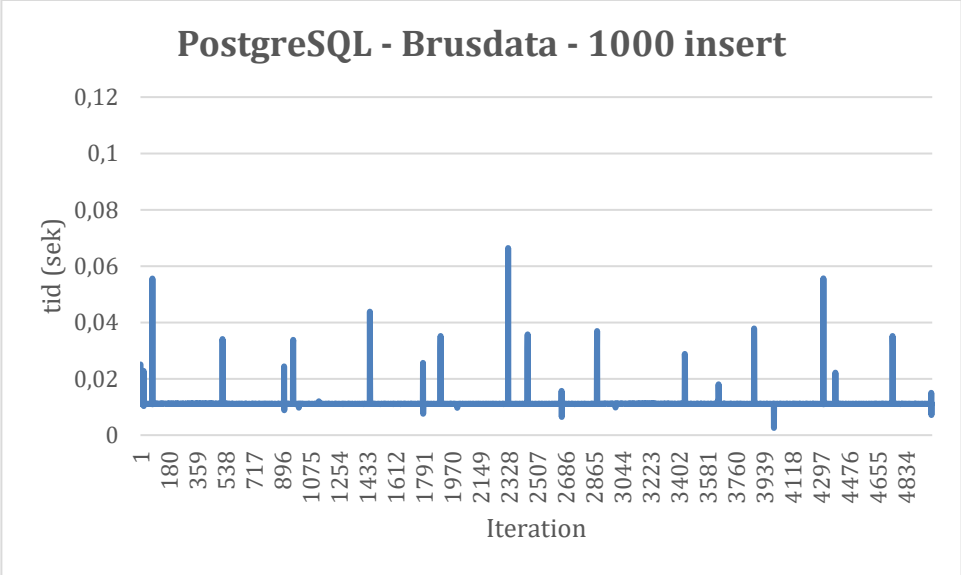
Tabell 3 Testfall med *select* för PostgreSQL och MongoDB

	Typ	**Antal**	**Mängd**	**Iterationer**
TF4 **1 gång / dag**	*Select*	1 000	1	5
		2 000	1	5
		5 000	1	5
		10 000	1	5
TF5 **Varje timme**	*Select*	1 000	24	5
		2 000	24	5
		5 000	24	5
		10 000	24	5
TF6 **Varje minut**	*Select*	1 000	1 440	5
		2 000	1 440	5
		5 000	1 440	5
		10 000	1 440	5

När mätningarna var klar gjordes diagram av resultaten men för att få bättre medelvärde så lästes alla mätvärden först in för respektive antal *insert* och *select* för att få diagram över brusdata och detta för att leta efter så kallade spikar. Är dessa spikar väldig höga kan det påverka medelvärdet så de allra högsta togs bort. För PostgreSQL nedanför i Figur 22 går det se ur spikarna såg ut innan rensning. I Figur 23 syns det hur det såg ut efteråt när de 5 största spikarna togs bort.

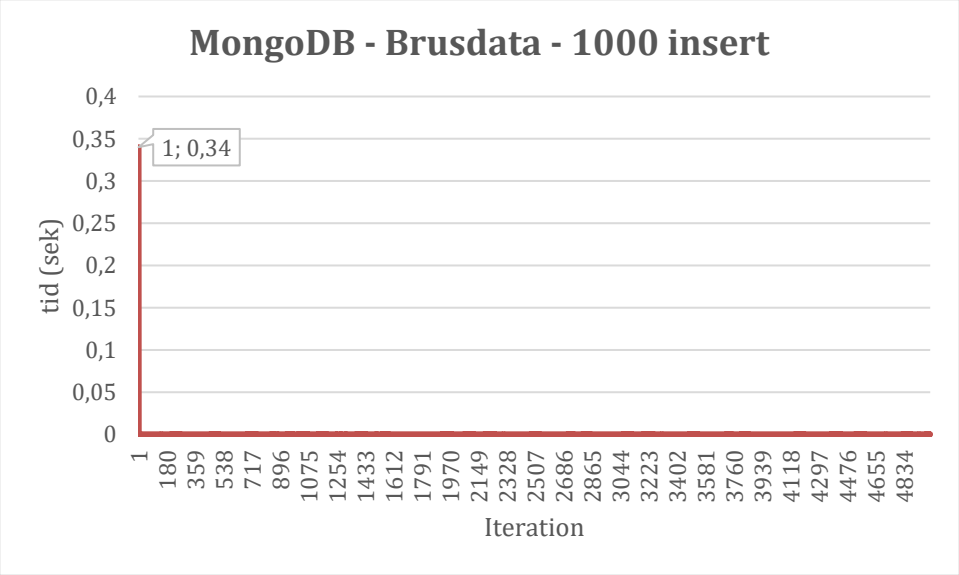


Figur 22 Brusdata för PostgreSQL innan rensning

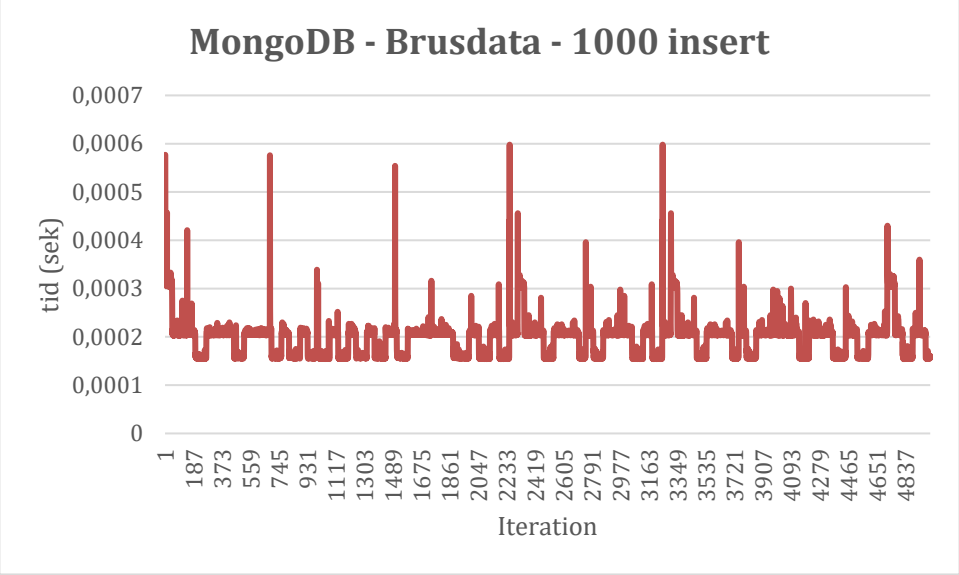


Figur 23 Brusdata för PostgreSQL efter rensning

MongoDB hade en hög spik i början som togs bort i Figur 24. Denna typ av rensning på spikar gjordes också i de andra testfallen men tas inte upp mer detaljerat under analysen utan bara ett exempel från testfall 1 med brusdata för 1000 *insert* i PostgreSQL och MongoDB visas. Det som kom varje gång var att MongoDB hade en hög spik i början av varje mätning men i övrigt ingenting annat och mätresultaten är väldigt låg på knappt 1 millisekund upp till ungefär 6 millisekunder för MongoDB. I Figur 25 är y-axeln neddragen från 0,4 till 0,0007 för att visa brusdata tydligare.

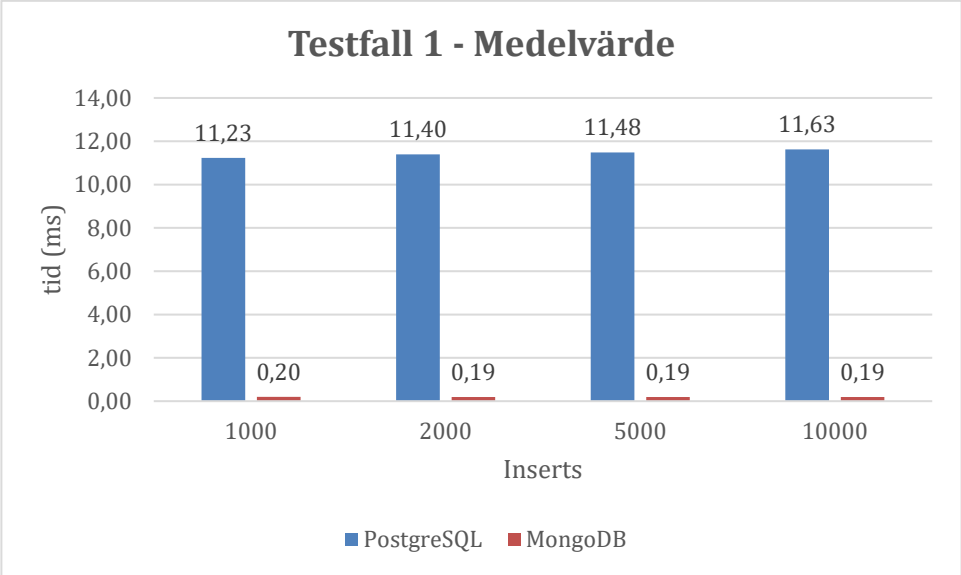


Figur 24 Brusdata för MongoDB innan rensning



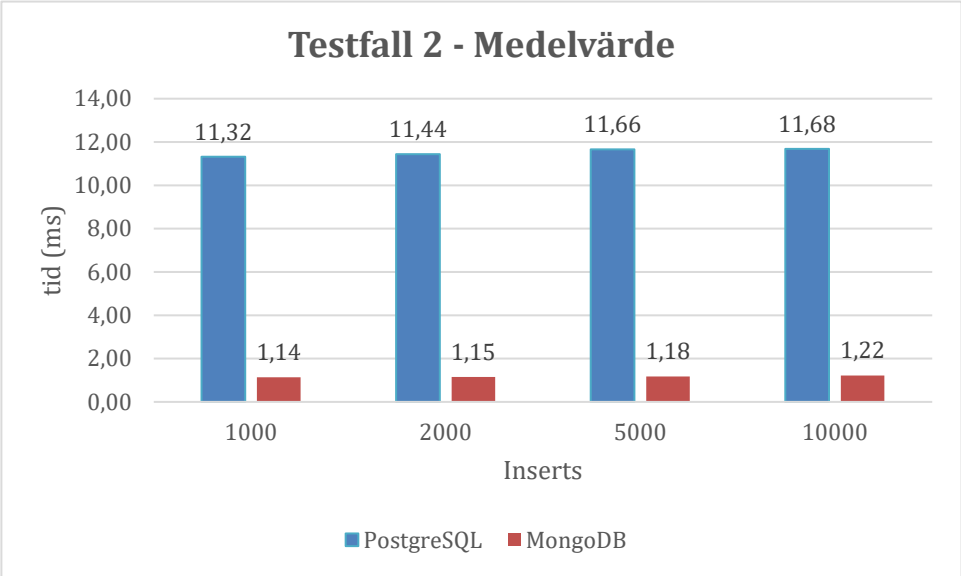
Figur 25 Brusdata för MongoDB efter rensning

Testfall 1 i Figur 26 med mängden 1 visar att MongoDB är snabb i *insert* på knappt 0,20 millisekunder och PostgreSQL ligger på ca 11 millisekunder.



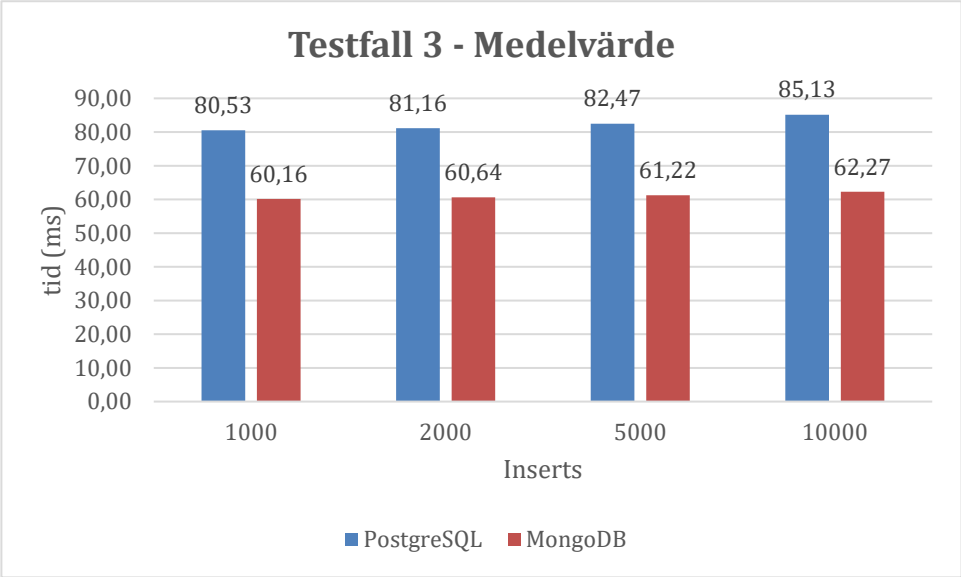
Figur 26 Svarstider för PostgreSQL och MongoDB med mängden 1

Testfall 2 i Figur 27 med mängden 24 visar på att MongoDB är snabb i *insert* på knappt 1,22 millisekunder och PostgreSQL ligger på ungefär 12 millisekunder.



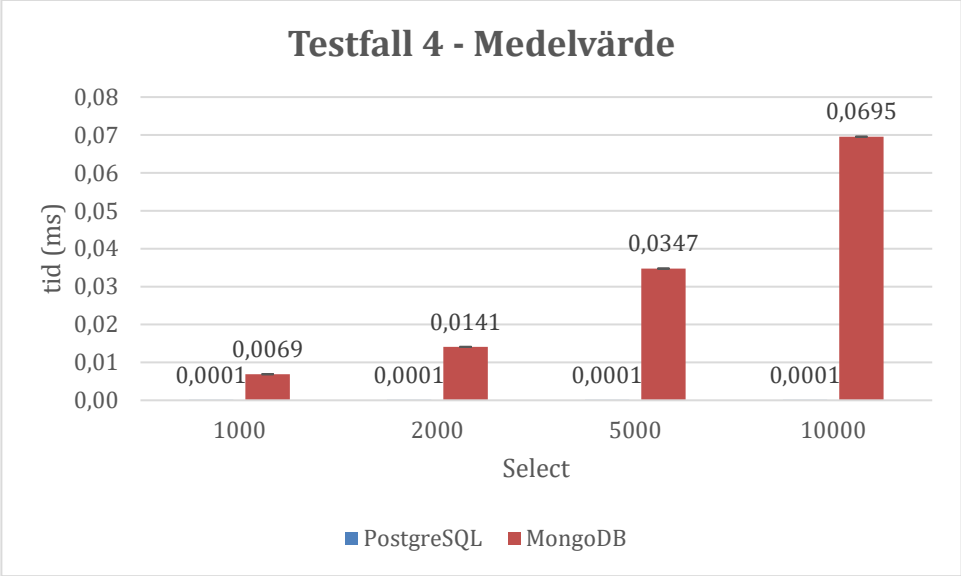
Figur 27 Medelvärde för PostgreSQL och MongoDB med mängden 24

Testfall 3 i Figur 28 med den största mängden 1440 visar på att MongoDB är snabb i *insert* på knappt 60 millisekunder och PostgreSQL ligger på ungefär 82 millisekunder.



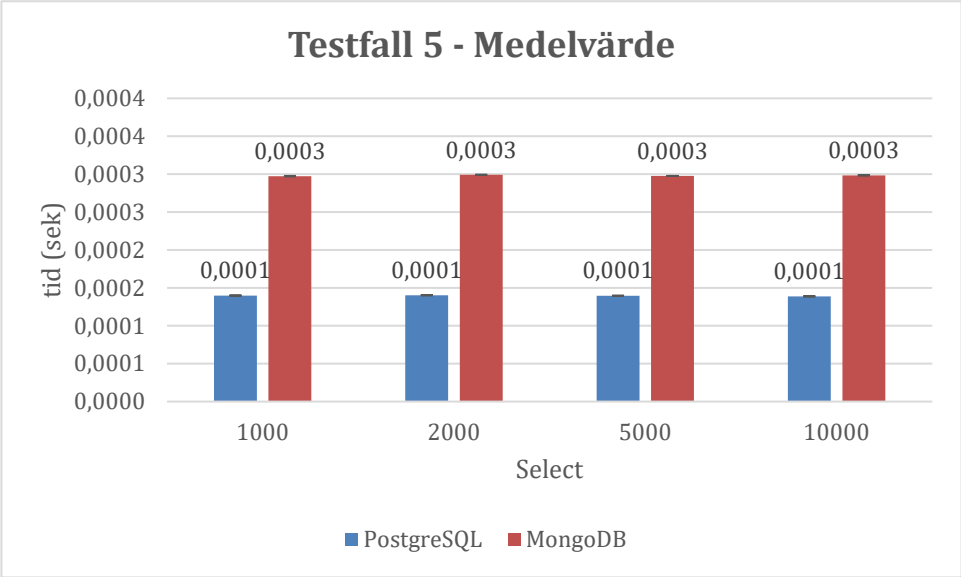
Figur 28 Medelvärde för PostgreSQL och MongoDB med mängden 1440

Testfall 4 i Figur 29 visar *select* som gjordes med mängden 1. PostgreSQL knappt mätbara på grund av att värdena är låga.



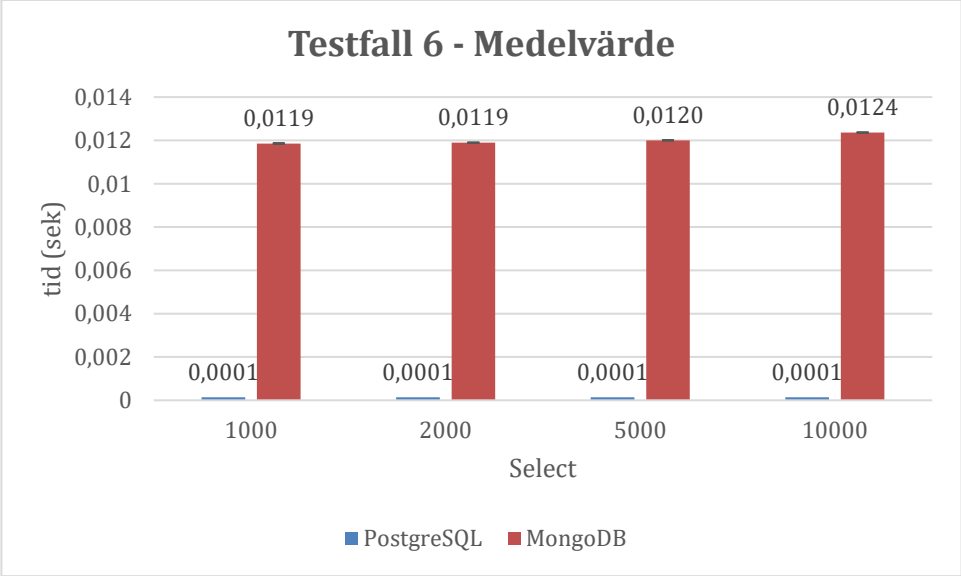
Figur 29 Medelvärde *select* för PostgreSQL och MongoDB

Testfall 5 i Figur 30 visar *select* för mängden 24 och här är båda databassystemen snabba och värdena väldigt låga.



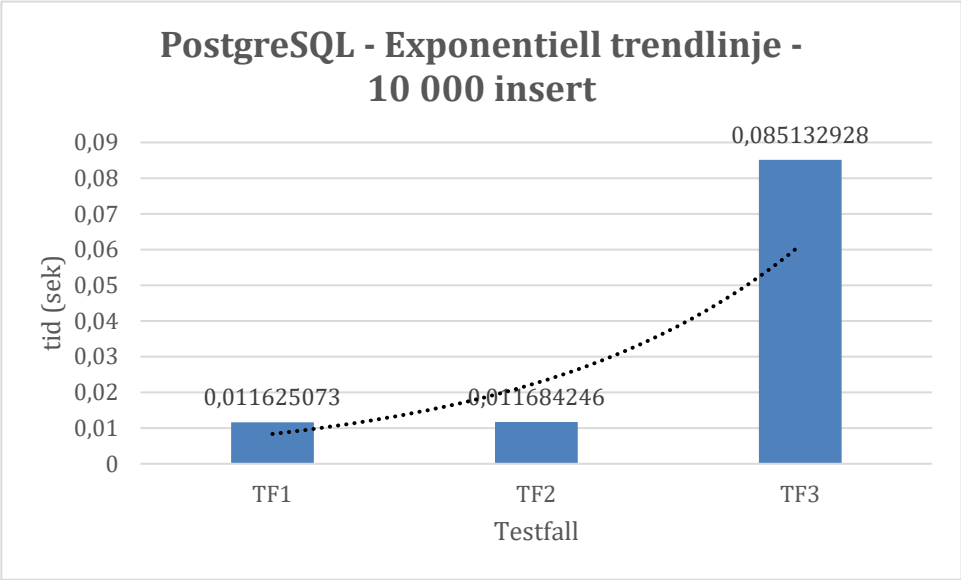
Figur 30 Medelvärde *select* för PostgreSQL och MongoDB

Testfall 6 i Figur 31 visar *select* för mängden 1440 och här är PostgreSQL knappt mätbar med låga värden.

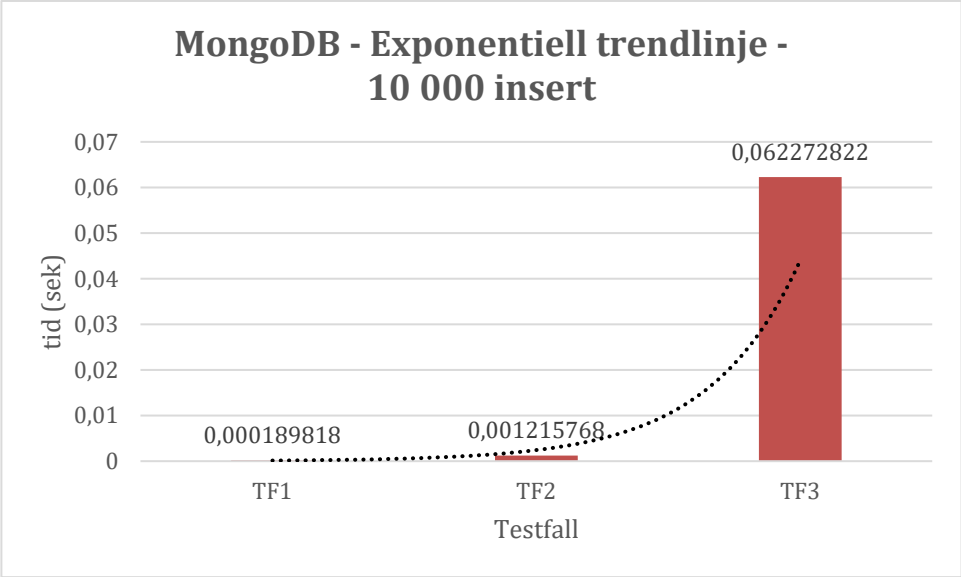


Figur 31 Medelvärde *select* för PostgreSQL och MongoDB

Trendlinjer gjordes för PostgreSQL Figur 32 och MongoDB Figur 33 där den största mängden med data 10 000 kunder och 1440 avläsningar per sensor genererades i *insert*.



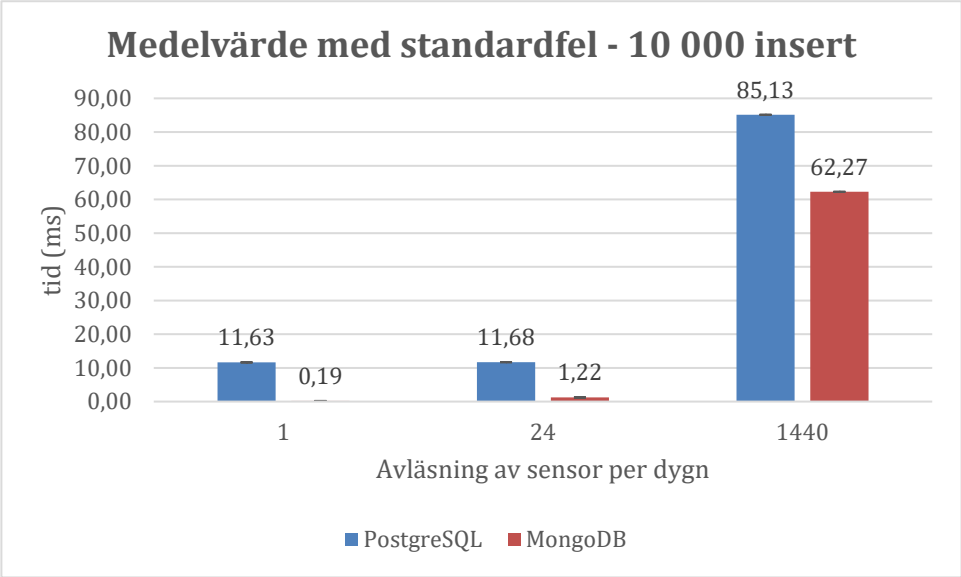
Figur 32 Medelvärde och exponentiell trendlinje



Figur 33 Medelvärde och exponentiell trendlinje

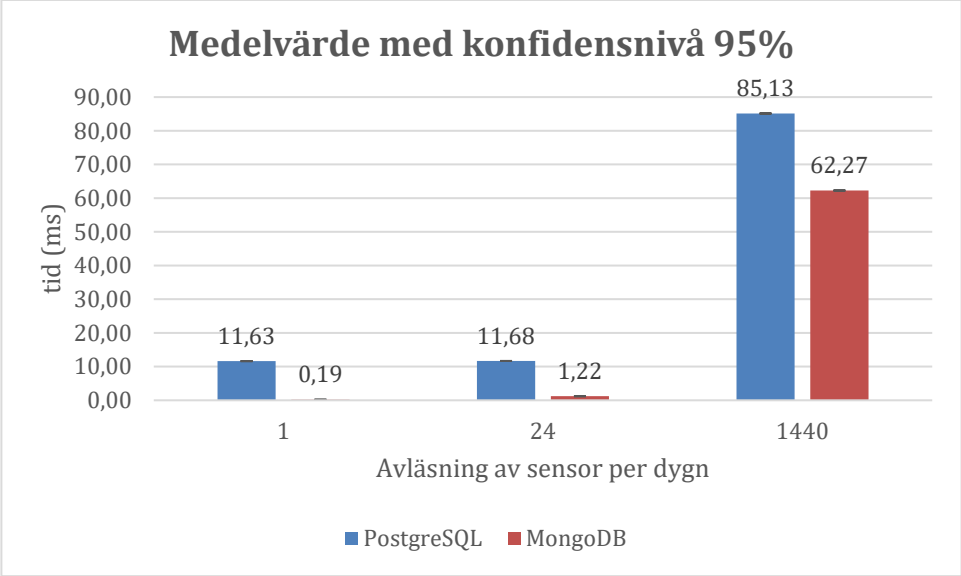
Trendlinjerna visar en svag stigning för PostgreSQL och för MongoDB är det en skarpare stigning när mängden data ökar. Det är ett relativt stort steg att gå från avläsningsmängden på 24 (varje timme) till 1440 (varje minut).

Medelvärde med standardfel där *insert* är 10 000 och staplarnas gruppering är avläsning av sensor per dygn Figur 34.



Figur 34 Medelvärde med standardfel (SE)

Medelvärde med konfidensnivå där *insert* är 10 000 och staplarnas gruppering är avläsning av sensor per dygn Figur 35



Figur 35 Medelvärde med konfidensnivå 95%

7 Avslutande diskussion

7.1 Sammanfattning

Jämförelse mellan två databassystem i hanteringen av JSON data genererad av en webbapplikation. Denna applikation simulerade en smart sensor som används i hemmet för att läsa av elektrisk förbrukning av en enhet som är kopplad till den. Databassystemen som användes var PostgreSQL och MongoDB. PostgreSQL är en typ av hybrid då den kan hantera SQL frågor men även frågor mot datatypen JSONB som är det som MongoDB använder. Den datamängd som skickades till databassystemen bestod av en *array* med flera nivåer byggd i PHP som simulerade ett data set som en sensor skickade över under ett dygn. Denna data genererades direkt i webbapplikationen och inte under ett riktigt dygn. Tre olika testfall för insättning av denna data utfördes och dessa var att sensorn lästes av 1 gång per dygn, varje timme och varje minut.

De frågor som togs upp i frågeställningen i 3.1 Frågeställning var:

> *”Kan PostgreSQL som hybriddatabasystem vara lämplig för sensordatalagring?”*
>
> *”Har mängden data betydelse när det gäller svarstid för lagring?”*

Hypotesen som ställdes i 3.2 Hypotes var:

> *”Hypotesen är att se om PostgreSQL är lämpligare än MongoDB i att hantera den JSON data som skapas av den simulerade sensorn som webbapplikationen genererar.”*

Ett pilottest gjordes innan experimentet för att få fram mätdata och hur det såg ut så att testfallen för huvudexperimentet kunde utföras på *insert* och *select*. Mätningarna som gjordes därefter delades in i sex stycken testfall där de första tre var för *insert* och de andra tre för *select*. De *insert* som gjordes var fördelade på mängden data som genererades av den simulerade sensorn som webbapplikationen bestod av. För *insert* så ponerade det vara antalet kunder i form av 1000, 2000, 5000 och 10 000 stycken som hade 1 sensor var. Testfallen för *insert* var också indelad i hur ofta sensorn lästes av på ett dygn. Testfall 1 har endast en avläsning per dygn, testfall 2 varje timme och testfall 3 varje minut. Dessa mätningar gjordes sedan om i 5 iterationer för att få bättre medelvärde.

Enligt resultatet från testfallen ser det ut som att MongoDB håller sig snabb när det gäller mängden data på 1 dygn och varje timme. I testfall 3 ökar det mycket för MongoDB som börjar närma sig PostgreSQL. Så svaret på hypotesen om PostgreSQL kan vara lämplig för JSON data är svaret att det är det möjligt att den kan vara det men svårt att svara på. En teori är att mängden data kan ha betydelse för *insert* och svarstid. För MongoDB så ser det ut att bli tyngre när den ökar. Dock är det svårt att

se på de resultat som visas i mätningarna utan ytterligare mängd data behövs. Genom detta är också hypotesen svår att svara på utan mer forskning behövs.

7.2 Diskussion

Vid analys av diagrammen för trendlinjer Figur 32–33 visar att PostgreSQL inte stiger allt för mycket även om datamängden ökar så teoretiskt borde linjen följa en liknande rörelse när mängden data över ytterligare. För MongoDB så är det knappt någon skillnad mellan TF1 och TF2. Däremot när datamängden ökar så stiger kurvan rätt drastiskt uppåt och teoretiskt bör den stiga ytterligare när datamängden ökar från sensorerna. Troligtvis kan det vara möjligt att det blir trögare för MongoDB när mängden data ökar men PostgreSQL ser ut att inte öka tiden för *insert* mycket trots ökande datamängd. Gällande *select* så ligger PostgreSQL väldigt låg i svarstid Figur 29–31 och en teori om det är att *select* frågan mot databassystemet bör ses över. Den visuella delen för webbapplikationen som nämndes under 5.1 Förstudie användes inte då intresset mer låg i svarstiderna för *select*.

7.2.1 Samhällsnytta och risker

Nyttan med denna studie är att den ska kunna vara till hjälp inom intresseområden för smarta sensorer där lagring av ostrukturerade data sker i NoSQL databassystem. Det kan också vara bra för de som funderar på vilka databaser som kan passa JSON data men också om intresse finns för hybriddatabassystem som PostgreSQL. Risker kan vara NoSQL databassystem som MongoDB som släpper lite på säkerhet och konsistens för att kunna vara mer skalbara och hålla bättre prestanda. Ett exempel för säkerhet i denna studie var att det gick att skapa och ta bort dokumentsamlingar direkt i databassystemet utan att använda användarnamn och lösenord.

7.2.2 Etik

Det finns möjlighet till att återupprepa detta experiment då all programkod finns publicerad på webbplatsen Github men också under appendix i slutet av denna studie. Detta gör att det går arbeta vidare och utveckla arbetet. Storlek på datamängden blev som mest 10 000 kunder i form av *insert*s med 1440 rader som blir 14 400 000 avläsningar från den simulerade sensorn. Frågan är hur experimentets resultat ser ut med många fler kunder och där de har mer än en sensor som avläses?

7.3 Framtida arbeta

För framtida arbeten kan denna studie vara till användning om det finns intresse för smarta sensorer och hur data för de lagras i NoSQL databassystem. Det skulle också vara intressant att se om schemat för PostgreSQL kan ändras för att mer likna en NoSQL då det den nu följer ett schema för ID som primärnyckel och Data som JSONB. Databassystemen i denna studie var PostgreSQL och MongoDB men det skulle kunna ske undersökning på andra NoSQL och hybriddatabassystem för att se hur de presterar i liknande experiment. Själva dataseten skulle kunna ses över då det finns många olika för de mängder av sensorer som finns. Då denna studie endast använde

en typ av data set går det kanske köra detta experiment med flera olika typer av dessa och öka mängden data på insättningar för att se vad det ger för resultat när mängden sensorer och data ökar. Frågan mot databassystemen som ställs i *select* kan också ses över för framtida forskning för som den ser ut nu så skiljer den sig för mycket från MongoDB. Det beror på att *select* i PostgreSQL söker på ett slumpat nummer i kolumnen ID vilket är felaktigt då MongoDB söker i efter ID i JSON data. Detta kan ändras genom att skriva om *select* för PostgreSQL och lägga till operatorer för NoSQL så den kan söka i kolumnen där JSON data finns. Det är möjligt att schemat kan behöva ändras för detta som nämndes tidigare i detta kapitel.

Referenser

Abramova, V., Bernardino, J (2013). NoSQL databases: MongoDB vs cassandra. C3S2E '13 International C* Conference on Computer Science and Software Engineering. Porto, Portugal 10 - 12 July 2013.

Bourhis, P., Reutter, J.L., Suárez, F., Vrgoč, D (2107). JSON: Data model, Query languages and Schema specification. 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. Chicago, Illinois, USA 14 – 19 May 2017.

BSON (2018). BSON (Binary JSON) Serialization. http://bsonspec.org/ [Hämtad 2018-02-11].

Chandra, D.G (2015). BASE analysis of NoSQL database. Future Generation Computer Systems Volume 52, Pages 13-21. November 2015.

DB-Engines (2018). DB-Engines Ranking - popularity ranking of database management systems. https://db-engines.com/en/ranking [Hämtad 2018-02-09].

Facchinetti, T., Benetti, G., Koledoye, M.A., Roveda, G (2016). Design and implementation of a web-centric remote data acquisition system. IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). Berlin, Germany 6-9 Sept 2016.

Győrödi, C.G., Győrödi, R.G., Pecherle, G., Olah, A (2015). A comparative study: MongoDB vs. MySQL. Engineering of Modern Electric Systems (EMES), 13th International Conference. Oradea, Romania 11–12 June 2015.

Jing, H., Haihong, E., Guan, L., Jian, D (2011). Survey on NoSQL database. 2011 6th International Conference on Pervasive Computing and Applications (ICPCA). Port Elizabeth, South Africa 26 - 28 October 2011.

Jung, M.G., Youn, S.A., Bae, J., Choi, Y.L (2015). A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment. 2015 8th International Conference on Database Theory and Application (DTA). Jeju, South Korea 25-28 November 2015.

Liu, Z.H., Hammerschmidt, B., McMahon, D (2017). JSON Data Management – Supporting Schema-less Development in RDBMS. Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference. Klagenfurt, Austria 20-24 Feb 2017.

Lu, H., Xu Yu, J., Wang, G., Zheng, S., Jiang, H., Yu, G., Zhou, A (2003). What makes the differences: benchmarking XML database implementations. Data Engineering, 2003. 19th International Conference. Bangalore, India 5-8 March 2003.

Lucidcharts (2018). Online Diagram Software & Visual Solution. https://www.lucidchart.com/ [Hämtad 2018-02-17].

MongoDB (2018). MongoDB Documentation. https://docs.mongodb.com/ [Hämtad 2018-02-16].

MongoDB (2018). MongoDB\Collection::findOne() — PHP Library Manual 1.2. https://docs.mongodb.com/php-library/v1.2/reference/method/MongoDBCollection-findOne/ [Hämtad 2018-04-14].

Montathar, F (2016). Webbutveckling med PHP och MySQL. Studentlitteratur. ISBN: 9 789 144 105 567.

McCarthy-Padron, T & Risch, T. (2005). Databasteknik. Lund: Studentlitteratur. ISBN 978-91-44-04449-1.

Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D (2016). Foundations of JSON Schema. 25th International Conference on World Wide Web. Montréal, Québec, Canada 11 – 15 April 2016.

PHP.net (2018). PHP: Hypertext Preprocessor (2018). http://www.php.net/ [Hämtad 2018-04-04].

PHP.net (2018). PHP: microtime – Manual. http://php.net/manual/en/function.microtime.php. [Hämtad 2018-04-04].

PostgreSQL (2018). PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/ [Hämtad 2018-02-17].

SoftQE (2018). Read JSON file using Python – Read and parse JSON file http://www.softqe.com/read-json-file-using-python/ [Hämtad 2018-02-09].

Soliman, M., Abiodun, T., Hamouda, T., Zhou, J., Chung-Horng, L (2013). Smart Home: Integrating Internet of Things with Web Services and Cloud Computing. Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference. Bristol, UK 2-5 Dec 2013.

Stack Overflow (2018). Accurate way to measure execution times of php scripts. https://stackoverflow.com/questions/6245971/accurate-way-to-measure-execution-times-of-php-scripts [Hämtad 2018-04-14].

Stack Overflow (2018). Adding minutes to date time in PHP. https://stackoverflow.com/questions/8169139/adding-minutes-to-date-time-in-php. [Hämtad 2018-04-14].

Stack Overflow (2018). I'm using PHP and need to *Insert* into sql using a while loop. https://stackoverflow.com/questions/18765899/im-using-php-and-need-to-*insert*-into-sql-using-a-while-loop [Hämtad 2018-04-14].

Stack Overflow (2018). PHP create JSON with foreach. https://stackoverflow.com/questions/43834471/php-create-json-with-foreach. [Hämtad 2018-04-05].

Stack Overflow (2018). Where Developers Learn, Share, & Build Careers (2018). https://stackoverflow.com/ [Hämtad 2018-04-04].

W3schools (2018). PHP: MySQL Database. https://www.w3schools.com/php/php_mysql_intro.asp. [Hämtad 2018-04-12].

W3schools (2018). PHP 5 Tutorial. https://www.w3schools.com/php/default.asp [Hämtad 2018-04-12].

W3schools (2018). W3Schools Online Web Tutorials. http://www.w3schools.com/ [Hämtad 2018-04-12].

Van der Veen, J.S., Van der Waaij, B., Robert J. Meijer, R.J (2012). Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. Cloud Computing (CLOUD), 2012 IEEE 5th International Conference. Honolulu, HI, USA 24-29 June 2012.

Wang, G (2011). Improving Data Transmission in Web Applications via the Translation between XML and JSON. Communications and Mobile Computing (CMC), Third International Conference. Qingdao, China 18-20 April 2011.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. & Wesslén, A (2012). Experimentation in Software Engineering. Berlin Heidelberg: Springer-Verlag. ISBN:978-3-642-29043-5.

Weiss, M., Friedemann, M., Graml, T., Staake, T., Fleisch, E (2009). Handy feedback: connecting smart meters with mobile phones. MUM '09 Proceedings of the 8th International Conference on Mobile and Ubiquitous Multimedia. Cambridge, United Kingdom Nov 22 - 25, 2009.

Appendix A - mongodb/dropdb.php

```
<?php
// clear database
require_once "vendor/autoload.php";
$db = (new MongoDB\Client)->exjobb;

$result = $db->drop();

var_dump($result);
?>
```

Appendix B - mongodb/insert.php

```
<?php
require "vendor/autoload.php";
$client = new MongoDB\Client("mongodb://localhost:27017");
$collection = $client->exjobb->json_data;

set_time_limit(0);

// variables
$iterations = 5;                // how many runs
$inserts = 2000;                // inserts to do (customers)
$measures = 1440;               // measures per inserts
$fileNr = 0;                    // save file counter
$idM = 123456;                  // random measurements ID
$deviceWatt = 60;               // the device watts
$startWatt = 25;                // starting watts value for device
$time = date('Y-m-d H:i:s');    // get time
$timeArray= [];                 // array to push response time

// start iteration
$index = 1;
while($index <= $iterations){
      $timeStart2 = microtime(true);
      $index++;
      $fileNr++;

      // generate json data
      $index2 = 1;
      while($index2 <= $inserts){
            $timeStart = microtime(true);
            $index2++;
            $jsonArray = array(

            'smartMeter' => array(

                  'id' => '1',
                  'device' => 'Eliond',
                  'sensorType' => 'Electric',
                  'createdOn' => '20180205',
                 ),

            'measurements' => array(),
            );

            for ($i=0; $i <$measures ; $i++) {
           $idM++;

          // to simulate the consumption in watt for the device
```

```
          $wattsRand = 0;
          $wattsRand = $wattsRand + rand(0,8); // device is on
between 0-8 hours/day
          $startWatt = $startWatt + ($deviceWatt * $wattsRand)/1000;
          $startWatt = number_format(($startWatt), 2);
          $time = date('Y-m-d H:i:s', strtotime($time.'+1 min'));

                $Data = array(
                   'id' => $idM,
                   'date' => $time,
                   'kWh' => $startWatt,
                );
                array_push($jsonArray['measurements'], $Data);
           }
           // insert array to database
           $result = $collection->insertOne( $jsonArray );

           $timeEnd = microtime(true);

           //Measure response time and push to array
           $timeDiff = $timeEnd - $timeStart;
           $timeDiff = number_format(($timeDiff), 6);
           array_push($timeArray, $timeDiff);
     }
     $timeEnd2 = microtime(true);

     // write values from timeArray to file
     $file = 'measurements_plot_'.$fileNr.'.txt';
     foreach ($timeArray as $key=>$value) {
          file_put_contents($file, $value.PHP_EOL, FILE_APPEND |
LOCK_EX);
          }

     //clear timeArray
     $timeArray= [];

     // calc time
  $timeDiff2 = $timeEnd2 - $timeStart2;
  $timeDiff2 = number_format(($timeDiff2), 3);

  // write values from timeArray to file
  $file = 'measurements.txt';
  file_put_contents($file, $timeDiff2.PHP_EOL, FILE_APPEND |
LOCK_EX);

     // clear DB after each iteration except after last one
     if ($index < $iterations) {
          include('initdb.php');
     }
```

```
}
?>
```

Appendix C - mongodb/select_query.php

```
<?php
require 'vendor/autoload.php'; // include Composer's autoloader

$client = new MongoDB\Client("mongodb://localhost:27017");
$collection = $client->exjobb->json_data;

set_time_limit(0);

// variables
$iterations = 5;           // how many runs
$queries = 10000;          // measures per inserts
$fileNr = 0;               // save file counter
$timeArray= [];            // array to push response time

// start iteration
$index = 1;
while($index <= $iterations){
      $timeStart2 = microtime(true);
      $index++;
      $fileNr++;

      $index2 = 1;
      while($index2 <= $queries){
            $timeStart = microtime(true);
            $index2++;

    // query the database
    $q = array('measurements.id'=> 123457);
    $cursor = $collection->findOne($q);

            $timeEnd = microtime(true);

            //Measure response time and push to array
            $timeDiff = $timeEnd - $timeStart;
            $timeDiff = number_format(($timeDiff), 6);
            array_push($timeArray, $timeDiff);
            }

            $timeEnd2 = microtime(true);

            // calc time
            $timeDiff2 = $timeEnd2 - $timeStart2;
            $timeDiff2 = number_format(($timeDiff2), 3);

            // write values from timeArray to file
            $file = 'measurements_query.txt';
```

```
            file_put_contents($file, $timeDiff2.PHP_EOL, FILE_APPEND
| LOCK_EX);

            // clear DB after each iteration except after last one
            if ($index < $iterations) {
                  include('initdb.php');
            }

            // write values from timeArray to file
            $file = 'measurements_plot_'.$fileNr.'.txt';
            foreach ($timeArray as $key=>$value) {
                  file_put_contents($file, $value.PHP_EOL,
FILE_APPEND | LOCK_EX);
                  }

            //clear timeArray
            $timeArray= [];
      }
?>
```

Appendix D - postgresql/dbconnect.php

```
<?php
   $host        = "host = localhost";
   $port        = "port = 5432";
   $dbname      = "dbname = test1";
   $credentials = "user = userName password=pass123";

   $dbconn = pg_connect( "$host $port $dbname $credentials"  );
?>
```

Appendix E - postgresql/type1/initdb.php

```
<?php

      // this will include the file dbconnect.php which contains
credentials
      include "../dbconnect.php";

      if(!$dbconn) {
            echo "<span style='background-color: #f44336'>Error:
Unable to open the database</span>";
      } else {
            echo "<span style='background-color: #4CAF50'>The
database was opened successfully</span><br><br>";
      }

  // init the database and create table
      $initQuery = file_get_contents("initdb.sql");

      echo "<span>Creating database..</span><br>";
      try {
            // check if error occured
            $ret = pg_query($dbconn, $initQuery);
            if(!$ret) {
              echo pg_last_error($dbconn);
            } else {
              echo "<span style='background-color: #4CAF50'>The table
was created successfully!</span><br><br>";
            }

      } catch (PDOException $e) {
            echo "<span style='background-color: #f44336'>An error
occured</span>";
      }
?>
```

Appendix F - postgresql/type1/initdb.sql

```
drop table if exists json_table;
     CREATE TABLE json_table(
          ID serial  PRIMARY KEY,
          data jsonb      NOT NULL
     );
```

Appendix G - postgresql/type1/insert.php

```
<?php
// this will include the file dbconnect.php which contains
credentials
include "../dbconnect.php";

set_time_limit(0);

// variables
$iterations =5;                 // how many runs
$inserts = 5000;                // inserts to do (customers)
$measures = 1440;               // measures per inserts
$fileNr = 0;                    // save file counter
$idM = 123456;                  // random measurements ID
$deviceWatt = 60;               // the device watts
$startWatt = 25;                // starting watts value for device
$time = date('Y-m-d H:i:s');    // get time
$timeArray= [];                 // array to push response time

// start iteration
$index = 1;
while($index <= $iterations){
      $timeStart2 = microtime(true);
      $index++;
      $fileNr++;

      // generate json data
      $index2 = 1;
      while($index2 <= $inserts){
            $timeStart = microtime(true);
            $index2++;
            $jsonArray = array(

            'smartMeter' => array(

                  'id' => '1',
                  'device' => 'Eliond',
                  'sensorType' => 'Electric',
                  'createdOn' => '20180205',
                 ),

            'measurements' => array(),
            );

            for ($i=0; $i <$measures ; $i++) {
           $idM++;

          // to simulate the consumption in watt for the device
```

```
          $wattsRand = 0;
          $wattsRand = $wattsRand + rand(0,8); // device is on
between 0-8 hours/day
          $startWatt = $startWatt + ($deviceWatt * $wattsRand)/1000;
          $startWatt = number_format(($startWatt), 2);
          $time = date('Y-m-d H:i:s', strtotime($time.'+1 minute'));

                $Data = array(
                   'id' => $idM,
                   'date' => $time,
                   'kWh' => $startWatt,
                );
                array_push($jsonArray['measurements'], $Data);
           }
           // encode php array to string
    $jsonArrayEncoded = json_encode($jsonArray);

    $sqlQuery = "INSERT INTO json_table (data) VALUES
('$jsonArrayEncoded')";
    $runQuery = pg_query($dbconn, $sqlQuery);

           $timeEnd = microtime(true);

           //Measure response time and push to array
           $timeDiff = $timeEnd - $timeStart;
           $timeDiff = number_format(($timeDiff), 6);
           array_push($timeArray, $timeDiff);
     }
     $timeEnd2 = microtime(true);

     // write values from timeArray to file
     $file = 'measurements_plot_'.$fileNr.'.txt';
     foreach ($timeArray as $key=>$value) {
          file_put_contents($file, $value.PHP_EOL, FILE_APPEND |
LOCK_EX);
          }

     //clear timeArray
     $timeArray= [];

     // calc time
  $timeDiff2 = $timeEnd2 - $timeStart2;
  $timeDiff2 = number_format(($timeDiff2), 3);

  // write values from timeArray to file
  $file = 'measurements.txt';
  file_put_contents($file, $timeDiff2.PHP_EOL, FILE_APPEND |
LOCK_EX);
```

```
        // clear DB after each iteration except after last one
        if ($index < $iterations) {
              include('initdb.php');
        }
}
?>
```

Appendix H - postgresql/type1/*select*_query.php

```
<?php
// this will include the file dbconnect.php which contains
credentials
include "../dbconnect.php";

set_time_limit(0);

// variables
$iterations = 5;           // how many runs
$maxRows = 10000;          // rows in database
$queries = 10000;          // measures per inserts
$fileNr = 0;               // save file counter
$timeArray= [];            // array to push response time

// start iteration
$index = 1;
while($index <= $iterations){
      $timeStart2 = microtime(true);
      $index++;
      $fileNr++;


      $index2 = 1;
      while($index2 <= $queries){
            $timeStart = microtime(true);
            $index2++;

            // choose random ID
            $randomNmb = rand(1,$maxRows);

            // query the database
            $result = pg_query($dbconn,"SELECT ID FROM json_table
WHERE ID=$randomNmb");

            $timeEnd = microtime(true);

            //Measure response time and push to array
            $timeDiff = $timeEnd - $timeStart;
            $timeDiff = number_format(($timeDiff), 6);
            array_push($timeArray, $timeDiff);
            }

            $timeEnd2 = microtime(true);

            // calc time
            $timeDiff2 = $timeEnd2 - $timeStart2;
            $timeDiff2 = number_format(($timeDiff2), 3);
```

```
            // write values from timeArray to file
            $file = 'measurements_query.txt';
            file_put_contents($file, $timeDiff2.PHP_EOL, FILE_APPEND
| LOCK_EX);

            // clear DB after each iteration except after last one
            if ($index < $iterations) {
                  include('initdb.php');
            }


            // write values from timeArray to file
            $file = 'measurements_plot_'.$fileNr.'.txt';
            foreach ($timeArray as $key=>$value) {
                  file_put_contents($file, $value.PHP_EOL,
FILE_APPEND | LOCK_EX);
                  }

            //clear timeArray
            $timeArray= [];
      }
?>
```

Appendix I - Anova Testfall 1

Anova: En faktor						
alpha 0.05						
SAMMANFATTNING						
Grupper	*Antal*	*Summa*	*Medelvärde*	*Varians*		
PostgreSQL	49998	581,230403	0,0116251	3,76E-05		
MongoDB	49999	9,490718	0,0001898	1,04E-09		
ANOVA						
Variationsursprung	*KvS*	*fg*	*MKv*	*F*	*p-värde*	*F-krit*
Mellan grupper	3,2690283	1	3,2690283	173979	0	3,841552
Inom grupper	1,87888466	99995	1,879E-05			
Totalt	5,14791295	99996				

Appendix J - Anova Testfall 2

Anova: En faktor						
alpha 0.05						
SAMMANFATTNING						
Grupper	*Antal*	*Summa*	*Medelvärde*	*Varians*		
PostgreSQL	50000	584,212312	0,011684246	4,7E-05		
MongoDB	49999	60,787181	0,001215768	5,7E-09		
ANOVA						
Variationsursprung	*KvS*	*fg*	*MKv*	*F*	*p-värde*	*F-krit*
Mellan grupper	2,739698553	1	2,739698553	116567,8	0	3,841552
Inom grupper	2,350233856	99997	2,3503E-05			
Totalt	5,089932409	99998				

Appendix K - Anova Testfall 3

Anova: En faktor

Alpha 0,05

SAMMANFATTNING

Grupper	*Antal*	*Summa*	*Medelvärde*	*Varians*
PostgreSQL	50000	4256,646407	0,085132928	0,000121
MongoDB	49999	3113,578823	0,062272822	7,67E-05

ANOVA

Variationsursprung	*KvS*	*fg*	*MKv*	*F*	*p-värde*	*F-krit*
Mellan grupper	13,0644808	1	13,06448077	132116,1	0	3,841552
Inom grupper	9,88833681	99997	9,88863E-05			
Totalt	22,9528176	99998				